Machine Learning: From Scratch to Mastery

Yasser Hifny

Machine Learning: From Scratch to Mastery

Yasser Hifny

وَمَنْ أَحْسَنُ قَوْلًا مِمَّنْ دَعَا إِلَى اللَّهِ وَعَمِلَ صَالِحًا
وَقَالَ إِنَّنِي مِنَ الْمُسْلِمِينَ

سورة فصلت الآية ٣٣

And who is better in speech than one who invites to Allah and does righteousness and says, "Indeed, I am of the Muslims."

Dedication.
To my father and mother

Preface

Data science is a rapidly growing field that combines statistical analysis, machine learning, and computational thinking to extract meaningful insights from data. As more and more data becomes available in our increasingly digital world, effectively harnessing and analyzing this data is becoming an essential skill for students and professionals alike.

This book, "Machine Learning: From Scratch to Mastery," is designed to provide readers with a solid foundation in the core concepts and techniques of machine learning. This book assumes readers have a basic familiarity with advanced mathematics concepts like calculus and linear algebra, as well as some experience with Python programming. By building on these foundations, the book is able to dive deeper into the core machine learning techniques without having to cover the absolute basics from the ground up.

One of the unique aspects of this book is the way it was developed. It was developed using AI-powered tools, including OpenAI's ChatGPT, Microsoft's Copilot through Skype, and DeepSeek's advanced language model. These powerful tools were used to generate Python scripts for the numerous examples and figures throughout the book, as well as to assist in the writing of mathematical equations and conceptual explanations.

This innovative approach dramatically reduced the time required to write the book, allowing the author to focus on curating the content and ensuring a cohesive and accessible learning experience for the reader. By leveraging the capabilities of these AI-powered tools, the author was able to create a comprehensive and up-to-date resource that covers the essential topics in machine learning, from linear regression and optimization to advanced algorithms such as transformers.

Whether you are an undergraduate student taking your first machine learning course or a postgraduate student looking to expand your skills, "Machine Learning: From Scratch to Mastery" is an invaluable resource that will guide you through the fundamental concepts and practical applications of this exciting field. With the help of ChatGPT, Copilot, and DeepSeek, the author has created a truly unique and accessible learning experience that will empower you to become a confident and capable data scientist.

Contents

1. Introduction

The future of computing is not just about faster machines, but smarter ones.

— Claude Shannon

We begin by establishing links between related fields, information theory, and probabilistic modeling. These connections are crucial for understanding various machine learning techniques. A strong foundation in these concepts will help in explaining advanced models. One such example is diffusion models, which rely on probabilistic principles. Understanding these relationships is essential for grasping their mathematical formulation.

1.1 Fundamental Mathematical Elements

In this section, we present and define fundamental mathematical elements, including scalars, vectors, matrices, and tensors. Each of these mathematical elements is an abstraction that helps describe physical quantities and relationships.

1. Scalar: A scalar is a single number, often representing a quantity that has magnitude but no direction. Scalars are used to measure quantities such as temperature, mass, or time.

   - Example: Temperature at a point, $T = 25\ °C$.
   - Mathematical Notation: Scalars are typically denoted by lowercase or uppercase letters, e.g., a, b, c.

2. Vector: A vector is an ordered list of numbers that represents a quantity with both magnitude and direction. Vectors are often visualized as arrows in space, with the length representing the magnitude and the direction representing its orientation.

- Example: Velocity in 3D space, $\mathbf{v} = (v_x, v_y, v_z)$.
- Mathematical Notation: A vector is usually represented as a bold letter or with an arrow above, e.g., $\mathbf{v}$.

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

3. Matrix: A matrix is a rectangular array of numbers arranged in rows and columns. Matrices are used to represent linear transformations, systems of linear equations, and more.

   - Example: A 2x2 matrix representing a linear transformation in 2D space.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

   - Mathematical Notation: Matrices are typically represented by uppercase bold letters, e.g., $\mathbf{A}$, $\mathbf{B}$.

4. Tensor: A tensor is a more general mathematical object that can be thought of as a multi-dimensional array of numbers. Tensors extend scalars, vectors, and matrices to higher dimensions and are used extensively in physics, engineering, and machine learning.

   - Example: A rank-3 tensor in 3D space.

$$\mathcal{T} = \left[t_{ijk}\right], \quad i, j, k = 1, 2, 3$$

   - Mathematical Notation: Tensors are usually denoted with calligraphic letters or bold, uppercase letters, e.g., $\mathcal{T}$, $\mathbf{T}$.
   - A scalar can be seen as a tensor of rank 0.
   - A vector is a tensor of rank 1.
   - A matrix is a tensor of rank 2.
   - Higher-rank tensors (rank 3 and above) represent more complex relationships. For example, a rank-3 tensor can be used to represent the stress or strain in a material in physics.

1.2 Information Theory for Machine Learning

Certain Probablistic modeling techniques incorporate concepts from information theory. Hence, basic elements of information theory are introduced (for details see

[1, 2, 3]). In his landmark paper [4], Shannon introduced a quantitative measure of information known as *entropy*. For a discrete random variable $\mathbf{s}$ takes values $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n$ with the corresponding probabilities $P(\mathbf{s}_1), P(\mathbf{s}_2), \ldots, P(\mathbf{s}_n)$, the entropy is defined as

$$S(\mathbf{s}) \triangleq -\sum_{i=1}^{n} P(\mathbf{s}_i) \log P(\mathbf{s}_i) \tag{1.1}$$

The entropy is a measure of the average uncertainty of a random variable and $-\log P(\mathbf{s}_i)$ is the amount of information gained by observing the event $\mathbf{s}_i$. Events $\mathbf{s}_i$ with low probabilities produce more information than the events with large probabilities. Hence, rare events produce more information or surprise than frequent events and this is the basic idea behind compression algorithms. The entropy has large values when all $\mathbf{s}_i$ have same probability. The entropy is measured in *nats* when the natural logarithm is used in Equation 1.1 to measure the information conveyed by a random variable or in *bits* when base 2 logarithm is used.
Similarly, the *conditional entropy* is the uncertainty in a random variable $\mathbf{s}$ given another random variable $\mathbf{o}$ and it is given by

$$S(\mathbf{s}|\mathbf{o}) \triangleq -\sum_{\mathbf{s},\mathbf{o}} P(\mathbf{s}, \mathbf{o}) \log P(\mathbf{s}|\mathbf{o}) \tag{1.2}$$

The average *mutual information* is defined as the difference between $S(\mathbf{s})$ and $S(\mathbf{s}|\mathbf{o})$ as shown in Equation 1.3. It is a measure of average reduction in uncertainly about $\mathbf{s}$ after observing $\mathbf{o}$. The mutual information is symmetric $I(\mathbf{s};\mathbf{o}) = I(\mathbf{o};\mathbf{s})$ and is always nonnegative.

$$\begin{aligned} I(\mathbf{s};\mathbf{o}) &\triangleq S(\mathbf{s}) - S(\mathbf{s}|\mathbf{o}) \\ &= -\sum_{\mathbf{s},\mathbf{o}} P(\mathbf{s}, \mathbf{o}) \log \frac{P(\mathbf{s}, \mathbf{o})}{P(\mathbf{s})P(\mathbf{o})} \end{aligned} \tag{1.3}$$

Relative entropy or the *Kullback-Leibler* (KL) divergence between two probability distributions P and Q of a discrete random variable is given by

$$\mathrm{D}_{\mathrm{KL}}(P||Q) \triangleq \sum_{\mathbf{s}} P(\mathbf{s}) \log \frac{P(\mathbf{s})}{Q(\mathbf{s})} \tag{1.4}$$

$\mathrm{D}_{\mathrm{KL}}(P||Q)$ is the information loss when Q is used to approximate P. KL divergence is nonsymmetric (i.e. $KL(P||Q) \neq KL(Q||P)$) and $\mathrm{D}_{\mathrm{KL}}(P||Q) \geq 0$. When the distributions P and Q are identical, $\mathrm{D}_{\mathrm{KL}}(P||Q)$ is exactly zero. The mutual information is the KL divergence between the joint distribution $P(\mathbf{s}, \mathbf{o})$ and the product of $P(\mathbf{o})$ and $P(\mathbf{s})$ distributions.
In probablistic machine learning, $Q = \tilde{P}_\Lambda$ is an hypothesized model that has free parameters Λ. The goal of the training process is to minimize the information loss

in terms of KL divergence between the training data distribution[1] $\tilde{P}$ and its hypothesized model $\tilde{P}_\Lambda$ (i.e. $D_{KL}(\tilde{P}||\tilde{P}_\Lambda)$ is minimum). Hence, Λ^* is given by

$$\begin{aligned}\Lambda^* &= \arg\min_\Lambda\{D_{KL}(\tilde{P}||\tilde{P}_\Lambda)\} \\ &= \arg\min_\Lambda\{\sum_{\mathbf{s}} \tilde{P}(\mathbf{s})\log\tilde{P}(\mathbf{s}) - \sum_{\mathbf{s}} \tilde{P}(\mathbf{s})\log\tilde{P}_\Lambda(\mathbf{s})\} \\ &= \arg\min_\Lambda\{S(\tilde{P}) - \sum_{\mathbf{s}} \tilde{P}(\mathbf{s})\log\tilde{P}_\Lambda(\mathbf{s})\} \\ &\propto \arg\min_\Lambda\{-\sum_{\mathbf{s}} \tilde{P}(\mathbf{s})\log\tilde{P}_\Lambda(\mathbf{s})\} = \arg\min_\Lambda\{S(\tilde{P},\tilde{P}_\Lambda)\}\end{aligned} \tag{1.5}$$

where $S(\tilde{P},\tilde{P}_\Lambda)$ is defined as the *cross entropy* between two distribution $\tilde{P}$ and $\tilde{P}_\Lambda$ and the term $S(\tilde{P})$ is ignored because it is independent of Λ. As a result, the minimization of $S(\tilde{P},\tilde{P}_\Lambda)$ and $D_{KL}(\tilde{P}||\tilde{P}_\Lambda)$ are equivalent. The minimization of the cross entropy between a data model and an hypothesized model is equivalent to the maximization of the log likelihood objective function, which is given by

$$\begin{aligned}\Lambda^* &= \arg\max_\Lambda \mathcal{L}(\tilde{P},\tilde{P}_\Lambda) = \arg\max_\Lambda \sum_{\mathbf{s}} \tilde{P}(\mathbf{s})\log\tilde{P}_\Lambda(\mathbf{s}) \\ &= \arg\max_\Lambda -S(\tilde{P},\tilde{P}_\Lambda)\end{aligned} \tag{1.6}$$

Hence, minimization of KL objective function between a data model and an hypothesized model is related the maximum likelihood estimation (MLE) between the two models. Similarly, maximizing the mutual information can be re-cast as minimizing the cross entropy between a data model and an hypothesized model [5]. We will examine the cross-entropy loss function, a key training objective used in binary classification (Chapter 5), multiclass classification (Chapter 6), and multilabel classification (Chapter 7) scenarios.

1.2.1 Variational Lower Bound (ELBO)

The ELBO (Evidence Lower Bound)[2] is another objective function we introduce. It is widely employed in statistical learning to approximate intractable probabilities, particularly in variational autoencoders (VAEs) [6] and diffusion models [7, 8, 9].
Given a latent variable model with observed data x and latent variable Z, the marginal likelihood is:

$$p(x) = \int p(x,Z)dZ \tag{1.7}$$

[1] A true distribution P that generates a data set is usually not known and is replaced with an empirical distribution $\tilde{P}$ observed from the stochastic process.

[2] https://xyang35.github.io/2017/04/14/variational-lower-bound/

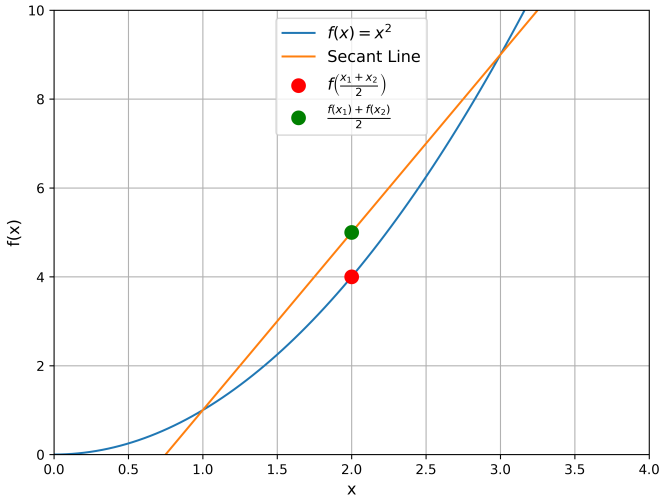


Figure 1.1: Jensen's inequality for a convex function. The plot shows how the secant line of a convex function lies above the curve between any two points, with annotations highlighting the inequality at the midpoint $f\left(\frac{x_1+x_2}{2}\right) < \frac{f(x_1)+f(x_2)}{2}$.

Since direct computation of $p(x)$ is often intractable, we introduce an approximate posterior distribution $q(Z)$. Applying Jensen's inequality $f(\mathbb{E}[x]) \leq \mathbb{E}[f(x)]$ for the concave log function (see Figure 1.1), we start with:

$$\begin{aligned}
\log p(x) &= \log \int_Z p(x, Z) \\
&= \log \int_Z p(x, Z) \frac{q(Z)}{q(Z)} \\
&= \log \left(\mathbb{E}_q \left[\frac{p(x, Z)}{q(Z)} \right] \right) \\
&\geq \mathbb{E}_q \left[\log \frac{p(x, Z)}{q(Z)} \right] \\
&= \mathbb{E}_q [\log p(x, Z)] + H[Z],
\end{aligned} \tag{1.8}$$

where $H[Z] = -\mathbb{E}_q[\log q(Z)]$ is the Shannon entropy. Hence,

$$\mathcal{L}(q) = \mathbb{E}_q [\log p(x, Z)] + H[Z] \tag{1.9}$$

It is evident that $\mathcal{L}(q)$ serves as a lower bound on the log-likelihood of the observed data. Consequently, when aiming to maximize the marginal likelihood, we can equivalently focus on optimizing this variational lower bound $\mathcal{L}(q)$.

The Kullback-Leibler divergence can alternatively used to drive the ELBO. The KL divergence between $q(Z)$ and $p(Z|x)$ is:

$$\begin{aligned} KL[q(Z)||p(Z|x)] &= \int_Z q(Z) \log \frac{q(Z)}{p(Z|x)} \\ &= -\int_Z q(Z) \log \frac{p(Z|x)}{q(Z)} \\ &= -\left(\int_Z q(Z) \log \frac{p(x,Z)}{q(Z)} - \int_Z q(Z) \log p(x) \right) \\ &= -\int_Z q(Z) \log \frac{p(x,Z)}{q(Z)} + \log p(x) \int_Z q(Z) \\ &= -\mathcal{L}(q) + \log p(x) \end{aligned} \tag{1.10}$$

Rearrange the terms:

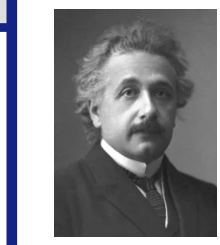
$$\mathcal{L}(q) = \log p(x) - KL[q(Z)||p(Z|x)] \tag{1.11}$$

Since KL divergence is always non-negative, we conclude:

$$\log p(Z) \geq \mathcal{L}(q) \tag{1.12}$$

which confirms that ELBO provides a lower bound on the log-marginal likelihood. Hence, maximizing ELBO indirectly maximizes the data likelihood $p(x)$. Moreover, the KL divergence ensures that the approximate $q(Z)$ is close to $p(Z|x)$.

2. Optimization



I never failed in mathematics.
Before I was fifteen I had mastered differential and integral calculus.

— Albert Einstein

Gradient-based optimization is an essential tool for the field of machine learning. This chapter addresses the basic elements of calculus used to understand gradient-based methods such as gradient descent.

2.1 Convex Functions

A function $f(x)$ is convex if the line segment between any two points on the graph of the function lies above or on the graph. Mathematically, for $x_1, x_2 \in \mathbb{R}$ and $\lambda \in [0, 1]$:

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$$

To illustrate the formal definition of a convex function, we can create a plot showing a convex function $f(x) = x^2$ and a line segment connecting two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ as shown in Figure 2.1. The convexity condition states that the function value at any convex combination of x_1 and x_2 is less than or equal to the corresponding convex combination of $f(x_1)$ and $f(x_2)$. Examples of convex functions are the quadratic function $f(x) = x^2$ and the exponential function $f(x) = e^x$.
A function $f(x)$ is non-convex if there exists at least one line segment between two points on the graph of the function that lies below the graph. It violates the convexity condition as shown in Figure 2.2. Examples of non-convex functions are cosine function $f(x) = \cos(x)$ and the quartic function with multiple minima $f(x) = x^4 - 4x^2 + 3$.

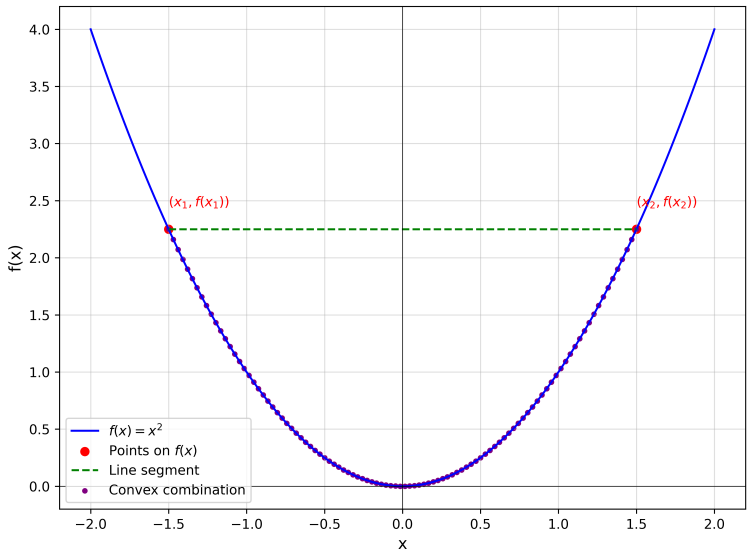


Figure 2.1: Illustration of the convex function definition. The blue curve represents $f(x) = x^2$, which is convex. The green dashed line connects the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. The purple points represent $f(\lambda x_1 + (1 - \lambda)x_2)$, which lie below or on the green line segment, satisfying the convexity condition.

2.2 Derivative

The derivative in calculus is a way of measuring the rate of change of a function at a certain point. It can also be interpreted as the slope of the line that is tangent to the function's curve at that point. The derivative of a function $f(x)$ can be denoted by $f'(x)$ or $\frac{df(x)}{dx}$, where x is the input variable[1]. The derivative mathematically can be defined as follows:

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

if the limit exists. A function is not differentiable at a point if it is not continuous at that point. For example, the function $f(x) = |x|$ is not continuous at $x = 0$, so it has no derivative there.

[1]A partial derivative is a derivative of a function of several variables with respect to one of those variables, while keeping the others constant. For example, if $f(x_1, x_2)$is a function of x_1 and x_2, then the partial derivative of f with respect to x_1 is denoted by $\frac{\partial f}{\partial x_1}$ and it is obtained by differentiating f with respect to x_1 and treating x_2 as a constant. Similarly, the partial derivative of f with respect to x_2 is denoted by $\frac{\partial f}{\partial x_2}$ and it is obtained by differentiating f with respect to x_2 and treating x_1 as a constant. Partial derivatives are used to measure the rate of change of a function along a specific direction or axis

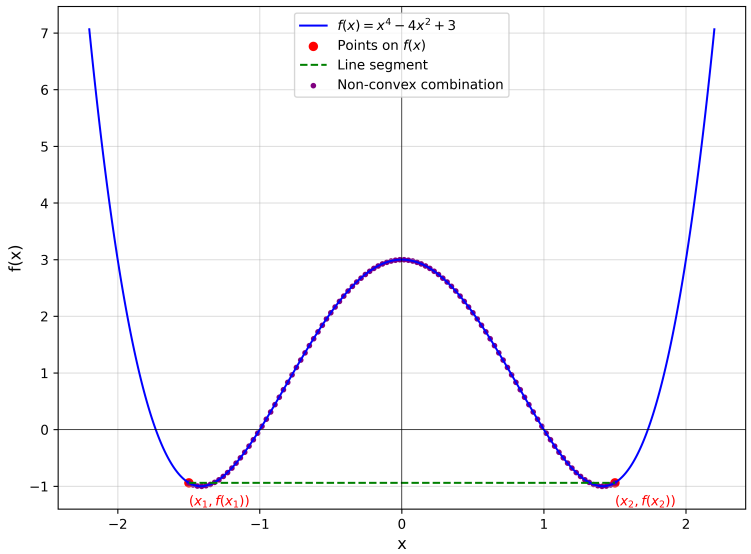


Figure 2.2: The plot shows curve with multiple local minima and maxima for the function ($f(x) = x^4 - 4x^2 + 3$), illustrating non-convexity.

Most machine learning algorithms are formulated as a minimization of a loss or objective function with respect to certain variables. To find a minimum of a function, there are different methods depending on the type and complexity of the function. Some of the common methods are:

- Sketching the function: This method involves plotting the graph of the function and visually identifying the lowest point on the graph. For example, the one-variable quadratic function has one global minimum[2] at $x = 1$:

$$f(x) = (x - 1)^2 \tag{2.2}$$

  where it is easy to find the minimum by inspecting the plot. However, this method is useful for simple functions that can be easily graphed, but it may not be accurate or feasible for more complicated functions.

- Finding analytical solution: This method involves using calculus to find the derivative of the function and setting it equal to zero. This gives the critical points of the function, where the slope is zero or undefined. Figure 2.4 shows

[2]The difference between local and global minimum of a function is that a local minimum is the point where the function value is smaller than (or equal to) the function values at nearby points, while a global minimum is the point where the function value is the smallest among all points in the domain. A function can have multiple local minima, but only one global minimum.

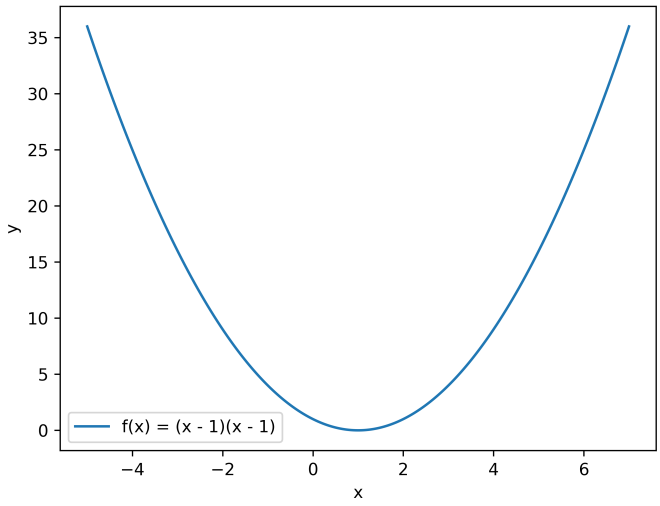


Figure 2.3: A plot of a simple quadratic function has a global minimum.

graphically why we set the first derivative to zero where the slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

Then, using the second derivative test or the first derivative test, we can determine which critical points are local minima, local maxima, or neither. To find the minimum of the quadratic function in Equation (2.2), we differentiate it with respect to x and set the derivative to zero as follows (i.e. using the power rule):

$$f'(x) = 2(x - 1) = 0 \tag{2.3}$$

Hence, the minimum happens at $x = 1$. In practice, this method for finding the minimum of a function does not scale well with the amount of training data commonly seen in machine learning problems.

- Using gradient descent method: The gradient descent method is an iterative optimization algorithm that is used to find the minimum of a function by moving in the opposite direction of the gradient (or the slope) of the function at each point. This method will be detailed in the next section.

2.3 Gradient Descent

The gradient descent method involves starting from an initial guess and iteratively updating it by moving in the opposite direction of the gradient (the vector of partial

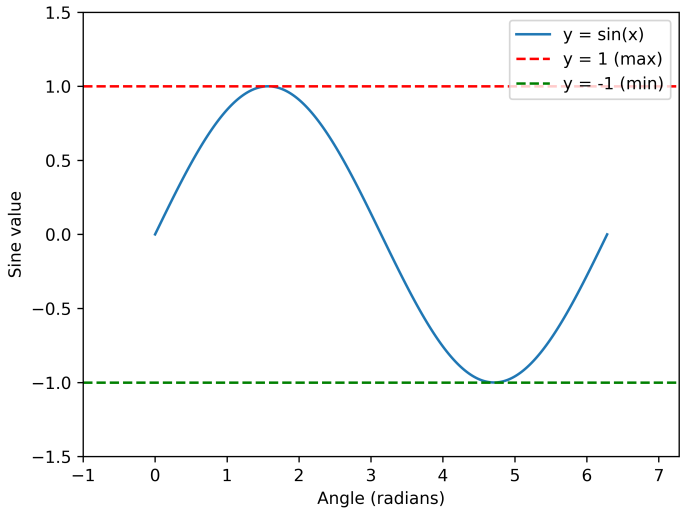


Figure 2.4: The slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

derivatives) of the function. The gradient gives the direction of steepest ascent, so moving against it will lead to a descent. The step size is determined by a learning rate parameter that controls how fast or slow the algorithm converges. This method is useful for finding a local minimum of a function that may not have an analytical solution or may be too complex to solve by calculus. However, this method does not guarantee finding the global minimum of the function, and it may depend on the choice of initial guess and learning rate.
The gradient descent method works as follows (assuming the function has one variable only):

1. Start with an initial guess $x = x^{(0)}$ for the parameters of the function that need to be optimized.

2. Calculate the gradient of the function with respect to the parameters at the current point $g(x) = \frac{df(x)}{dx}|_{x^{(0)}}$.

3. Update the parameters by subtracting a fraction of the gradient from the current values. The fraction is called the learning rate and it controls how big or small the steps are.

$$x^{(1)} = x^{(0)} - \eta g(x) \tag{2.4}$$

where $\eta > 0$ is the learning rate. When the gradient is positive (ascending), it means that the function is increasing in that direction. Therefore, we move

against the gradient direction to find a lower point on the function, aiming to eventually reach a local minimum. Similarly, when the gradient is negative (descending), it means the function is already decreasing in that direction, but we still move against the gradient to continue finding a lower point and approach the local minimum. This way, we hope to eventually reach a local minimum of the function as well. This behavior is shown in Figure 2.5. In the next section, we show mathematically why we need to subtract a fraction of the gradient from the current values.

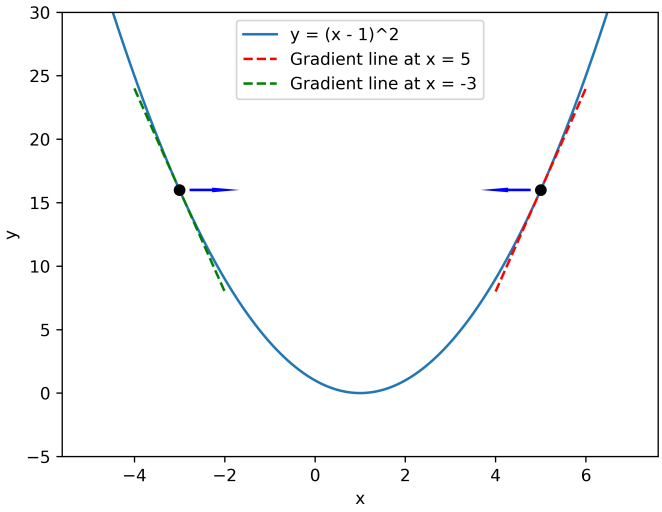


Figure 2.5: Function plot with gradient lines (ascending at $x = 5$ and descending at $x = -3$) and arrows pointing towards the minimum.

4. Repeat steps 2 and 3 until the gradient is close to zero or a maximum number of iterations is reached.

The gradient descent method is widely used in machine learning to train models by minimizing a loss function that measures the difference between the predicted and actual outputs. The gradient descent method can be applied to different types of functions, such as linear, quadratic, or non-linear functions. There are also different variants of the gradient descent method, such as batch gradient descent, stochastic gradient descent, mini-batch gradient descent, and momentum gradient descent, that differ in how they calculate and update the gradients.

2.3.1 Examples

In this subsection, I will use Python code to illustrate how the gradient descent algorithm works and how it can be applied to different functions of one variable or two variables.
The gradient descent algorithm is a method to find the minimum of a function by taking small steps in the direction of the steepest decrease. To apply this algorithm to the function $f(x) = (x-1)^2$, we need to first find its derivative, which is $g(x) = 2(x-1)$. The algorithm starts with an initial guess for x =3, and computes the value of $g(x)$ at $x = 3$. Then it updates x by subtracting an $\eta g(x)$ from it. This gives a new value for x that is closer to the minimum of the function. The algorithm repeats this process until it converges to a value of x that makes $g(x)$ very close to zero or a maximum number of epochs is reached. This value of $x = 1$ is the minimum of the quadratic function $f(x) = (x-1)^2$. The described algorithm can be implemented as a Python code as follows:

```
def grad(x):
    return 2.0 * (x-1.0)

x = 3.0
eta = 0.001
epochs = 50000
for i in range(epochs):
    x -= eta * grad(x)

print(x)
```

Listing 2.1: Python example for finding the minimum of a quadratic function in one variable.

On the other hand, we can use two different learning rates η_{x1}, η_{x2} for the function $f(x1, x2) = (x1-2)^2 + 10 * (x2+3)^2$ because the function has different scales and curvatures along the $x1$ and $x2$ directions. If we use a single learning rate for both variables (e.g. $\eta_{x1} = \eta_{x2} = 0.1$), we might encounter a convergence problem. By using different learning rates for each variable $\eta_{x1} = 0.1$ and $\eta_{x2} = 0.05$, we can adjust the step size according to the shape of the function and find the minimum more efficiently and accurately[3]. A Python implementation to find the minimum of this function is:

```
#Define the function of two variables
def f(x1, x2):
    return (x1 - 2)**2 + 10.0 * ((x2 + 3)**2)

#Define the partial derivatives of the function
def df_dx1(x1, x2):
```

[3]You can play with the learning rates to study the convergence properties.

```
    return 2 * (x1 - 2)

def df_dx2(x1, x2):
    return 20.0 * (x2 + 3)

#Define the learning rates for each variable
eta_x1 = 0.1 # Learning rate for x1
eta_x2 = 0.05 # Learning rate for x2

#Define the initial values for x1 and x2
x1 = 0.0
x2 = 0.0

#Define the tolerance for convergence
epsilon = 0.000001

#Define a variable to store the previous value of the function
prev_f = f(x1, x2)

#Start the gradient descent loop
steps = 0
while True:
    steps += 1
    #Update x1 and x2 using the gradient and the learning rates
    x1 = x1 - eta_x1 * df_dx1(x1, x2)
    x2 = x2 - eta_x2 * df_dx2(x1, x2)

    #Compute the current value of the function
    curr_f = f(x1, x2)

    #Check if the function value has decreased sufficiently
    if abs(curr_f - prev_f) < epsilon:
        break # Exit the loop

    #Update the previous value of the function
    prev_f = curr_f

#Print the final values of x1 and x2 and the minimum value of the
    function
print("x1 =", x1)
print("x2 =", x2)
print("f(x1, x2) =", curr_f)
print("steps for convergence =", steps)
```

Listing 2.2: Python example for finding the minimum of a quadratic function in two variables.

The adaptive learning rate is a technique that adjusts the learning rate dynamically based on the progress of the gradient descent algorithm. The idea is to use a larger learning rate when the function is far from the minimum and a smaller learning

rate when the function is close to the minimum. This way, we can speed up the convergence and avoid overshooting or oscillating. One method to implement the adaptive learning rate is to use the Hessian matrix, which is the matrix of second-order partial derivatives of the function. The Hessian matrix captures the curvature of the function and can be used to scale the gradient vector according to the shape of the function. By using the inverse of the Hessian matrix as a multiplier for the gradient vector, we can obtain a more accurate direction and step size for each iteration of the gradient descent algorithm. The next section will detail these techniques.

2.4 Gradient Descent using Taylor's Series

In mathematics, Taylor's series can be used to make a first-order approximation to a scalar loss function $f(\mathbf{x})$ around the current point vector $\mathbf{x}^{(t)} \in \mathbb{R}^d$ given the first derivative of the function at that point:

$$f(\mathbf{x}^{(t+1)}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)})^T \mathbf{g}, \tag{2.5}$$

where $\mathbf{g}$ is the gradient vector at the point $\mathbf{x}^{(t)}$. It can be written as well as follows:

$$f(\mathbf{x}^{(t)} + \Delta\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + \Delta\mathbf{x}^T \mathbf{g}, \tag{2.6}$$

where $\Delta\mathbf{x} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$. In order to decrease the loss function $f(\mathbf{x}^{(t)} + \Delta\mathbf{x})$, the term $\Delta\mathbf{x}^T\mathbf{g}$ has to be a negative value. Hence,

$$\Delta\mathbf{x}^T\mathbf{g} < 0 \tag{2.7}$$

Let's consider the cosine of angle between the two vectors $\Delta\mathbf{x}^T$ and $\mathbf{g}$:

$$\cos\theta = \frac{\Delta\mathbf{x}^T\mathbf{g}}{|\Delta\mathbf{x}^T||\mathbf{g}|} \tag{2.8}$$

$\cos\theta$ lies between -1 and 1 i.e. $-1 \leq \cos\theta \leq +1$. Hence,

$$-|\Delta\mathbf{x}^T||\mathbf{g}| \leq \Delta\mathbf{x}^T\mathbf{g} \leq |\Delta\mathbf{x}^T||\mathbf{g}| \tag{2.9}$$

Now we want the dot product to be as negative as possible (so that loss can be as low as possible). We can set the dot product to be $-|\Delta\mathbf{x}^T||\mathbf{g}|$ where $cos\theta$ has to be equal to -1 corresponds to $\theta = 180^o$. Therefore,

$$\Delta\mathbf{x} = -\mathbf{g} \tag{2.10}$$

This result explains why we move in the opposite direction of the gradient as we described in Section 2.3.

Intuitively, since the first-order approximation is good only for small $\Delta\mathbf{x}$, we want to choose a small $\eta > 0$ to make $\Delta\mathbf{x}$ small in magnitude. η is called the learning rate. Hence,

$$\Delta\mathbf{x} = -\eta\mathbf{g} \tag{2.11}$$

2.5 Gradient Descent Limitations

The gradient of a function at a specific point represents the direction of the steepest descent of the function at that point. In other words, it points in the direction in which the function decreases most rapidly. On the other hand, the contours[4] of a function are curves along which the function has the same value, so there is no change in the function value as you move along the contour.
Since the gradient points in the direction of maximum decrease, it is orthogonal (perpendicular) to the direction along which there is no change in the function value, which is the contour. This is true for any scalar function.
One limitation of the gradient descent is the zigzag effect. The zigzag effect occurs because the gradient at each point points in the direction of the steepest descent when moving downhill and may not necessarily point directly toward the minimum. As the algorithm moves along the steepest slope, it overshoots the direction of the minimum, and in the next step, it must correct its course. This leads to a back-and-forth zigzagging pattern as the algorithm iteratively converges to the minimum as shown in Figure 2.6. This zigzag effect slows down the convergence of the algorithm. Zigzag effect of gradient descent does not happen for the loss functions that have circular contours or equal curvature in all dimensions or directions. For example, a function like $f(x, y) = x^2 + y^2$ has circular contours and does not have zigzag effect. A straight line to the minimum for these functions is followed by gradient descent as shown in Figure 2.7.
Hessian-based optimization methods, like Newton's method or Quasi-Newton methods (such as BFGS and L-BFGS), make use of second-order information to help guide the optimization process more effectively [10]. They use the Hessian matrix or its approximation to adjust the step size and direction, which can reduce the zigzagging effect and potentially lead to faster convergence. Using Taylor's expansion, the second-order approximation is given by

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x} - \mathbf{x}^{(t)})^T\mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(t)})^T\mathbf{H}(\mathbf{x} - \mathbf{x}^{(t)}), \tag{2.12}$$

where $\mathbf{H}$ is the Hessian matrix at the point $\mathbf{x}^{(t)}$. The local Hessian matrix is a

[4] A contour of a function (e.g. Rosenbrock function) is a curve connecting points with the same function value. A contour is defined mathematically as the set of points (x, y) such that $f(x, y) = c$, where c is a constant.

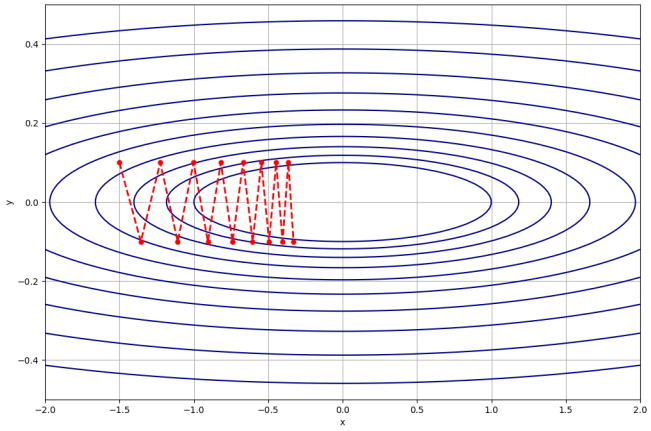


Figure 2.6: The zigzag effect of the gradient descent algorithm.

symmetric matrix and is defined by

$$\mathbf{H} \equiv \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j}|_{\mathbf{x}^{(t)}} \tag{2.13}$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_1 \partial \mathbf{x}_2} & \cdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_2 \partial \mathbf{x}_1} & \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{2.14}$$

The basic idea of Newton's method is to minimize the quadratic approximation of the cost function $f(\mathbf{x})$ around the current point $\mathbf{x}^{(t)}$. Equation (2.12) can be rewritten as follows:

$$\Delta f(\mathbf{x}^{(t)}) = f(\mathbf{x}) - f(\mathbf{x}^{(t)}) \approx \Delta \mathbf{x}^T \mathbf{g} + \frac{1}{2} \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} \tag{2.15}$$

Differentiating the above equation with respect $\Delta \mathbf{x}$ and setting the output to zero to get the minimum:

$$\mathbf{g} + \mathbf{H} \Delta \mathbf{x} = 0, \tag{2.16}$$

Hence, the Newton's update rule is given by

$$\Delta \mathbf{x} = -\eta \mathbf{H}^{-1} \mathbf{g} \tag{2.17}$$

When the matrix $\mathbf{H}$ equals to the identity matrix[5] (i.e. taking the same step in each direction), we reach the gradient descent update rule described in Equation (2.11).

[5] An identity matrix is a square matrix in which all the elements of the principal diagonal are ones and all other elements are zero.

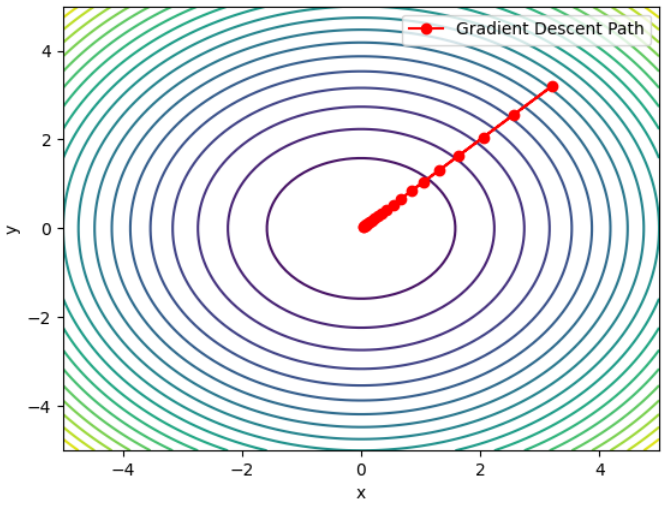


Figure 2.7: The zigzag effect of the gradient descent algorithm does not happen for the functions that have circular contours.

It's important to note that while Hessian-based methods can help overcome the zigzagging effect, they often come with their own challenges, such as increased computational complexity and the need to compute or approximate the Hessian matrix. In practice, these trade-offs need to be considered when selecting an optimization algorithm for a particular problem [11].
Adaptive learning rate methods overcome the zigzag effect in the gradient descent algorithm as well. They are addressed in the next subsection.

2.5.1 Adaptive learning Rate

Adaptive learning rate methods overcome the zigzag effect in gradient descent by adjusting the learning rate for each parameter during the optimization process. These methods take into account the history of gradients, the magnitude of the gradients, or both, to determine an appropriate learning rate for each parameter. As a result, adaptive learning rate methods can effectively navigate the loss surface and reduce oscillations and zigzagging.
Some popular adaptive learning rate methods include:

- **Momentum**: Momentum can reduce the zigzag problem by accumulating a vector that smooths out the gradient updates and aligns them with a consistent direction. Hence, it converges faster and more reliably than the gradient

descent [12]. The momentum with gradient descent algorithm updates the parameters $\mathbf{x}$ as follows:

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta) g^{(t)} \tag{2.18}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta m^{(t)} \tag{2.19}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ is the vector that accumulates the past gradients, β is the momentum coefficient, and η is the learning rate.

- **AdaGrad**: AdaGrad accumulates the squared gradients for each parameter in a diagonal matrix and uses this information to adapt the learning rate for each parameter. Parameters with larger accumulated squared gradients have their learning rate reduced, while those with smaller accumulated squared gradients have their learning rate increased. This makes AdaGrad well-suited for problems with sparse gradients or features that occur with varying frequency [13]. The AdaGrad algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = v^{(t-1)} + g^{(t)} . g^{(t)} \tag{2.20}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{v^{(t)}} + \epsilon} g^{(t)} \tag{2.21}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the sum of the squares of the gradients up to time step t, η is the learning rate, and ϵ is a small constant to prevent division by zero

- **RMSprop**: RMSprop is an improvement over Adagrad that uses an exponentially decaying average of the squared gradients instead of the cumulative sum. This makes RMSprop more robust to situations where the accumulated squared gradients can grow indefinitely, causing the learning rate to shrink too much [14]. The RMSprop algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = \beta v^{(t-1)} + (1-\beta) g^{(t)} . g^{(t)} \tag{2.22}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{g^{(t)}}{\sqrt{v^{(t)}} + \epsilon} \tag{2.23}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the estimate of the second moment of the gradients, β is the decay rate for the moment, η is the learning rate, and ϵ is a small constant to prevent division by zero.

- **Adam**: Adam combines the ideas of RMSprop and momentum-based methods. It computes the first moment (mean) and the second moment (uncentered variance) of the gradients, and uses these moments to adapt the learning rate for each parameter. This helps Adam to navigate the loss surface more effectively, reducing zigzagging and achieving faster convergence [15]. The Adam algorithm updates the variables $\mathbf{x}$ as follows:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)} \tag{2.24}$$

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2) g^{(t)}.g^{(t)} \tag{2.25}$$

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t} \tag{2.26}$$

$$\hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t} \tag{2.27}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} \tag{2.28}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ and $v^{(t)}$ are the estimates of the first and second moments of the gradients, $\hat{m}^{(t)}$ and $\hat{v}^{(t)}$ are the bias-corrected estimates[6], β_1 and β_2 are the decay rates for the moments, η is the learning rate, and ϵ is a small constant to prevent division by zero. Although the Adam algorithm is the state-of-the-art learning algorithm, the algorithms suffer from worse generalization[7] performance than stochastic gradient descent despite their faster training speed [16]. Hence, a practical recipe for training is to start with Adam for a few epochs and then switch to the gradient descent algorithm. In addition, the Adam algorithm accumulates two statistics for each variable or parameter during the optimization process.

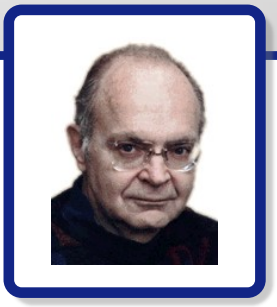
2.6 Assignment

Using Adam algorithm, find the minimum of the function $f(x1, x2) = (x1-2)^2 + 10 * (x2+3)^2$.

[6]The Adam algorithm suffers from a bias problem due to the initialization of the first and second moment estimates at zero. This means that the algorithm tends to underestimate the true values of these moments at the beginning of the training, leading to inaccurate gradient updates. To overcome this problem, the Adam algorithm uses a bias correction term that divides each moment estimate by a factor that accounts for the decay rates of the moments. This way, the algorithm can adjust for the bias and converge faster and more reliably.

[7]The generalization is measured using the test set performance.

3. Input Representation



Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values, they involve important structural relationship between the data elements.

— Donald Ervin Knuth

Most types of data, including tabular data, text, speech, and images, can be formatted for computer processing. In this chapter, we discuss various methods for representing these different types of data.

3.1 Tabular Data Representation

Tabular data refers to data that is organized into rows and columns, making it easy to read and analyze. Each row in a table represents a record or an instance, while each column represents a specific attribute or feature of the data. This structure is common in databases, spreadsheets, and many machine learning applications.
In the context of machine learning, particularly with "feedforward neural networks", tabular data is used to train models by feeding structured data into the network. The table is often converted into a format suitable for processing by the neural networks, typically by normalizing or encoding categorical variables. The key elements of tabular data are:

1. Rows: Each row represents a single observation or data point. For example, in a dataset of customer information, each row might represent a different customer.

2. Columns: Each column represents a feature or attribute of the data. For instance, columns might include customer age, income, and purchase history.

Table 3.1: Raw customer data.

Customer ID	Age	Income ($)	Gender	Purchased (Yes/No)
001	25	50000	Male	Yes
002	32	60000	Female	No
003	45	75000	Female	Yes
004	29	55000	Male	No
005	39	80000	Female	Yes

3. Headers: The first row often contains the column names, which describe the attributes of the data.

4. Data Types: Columns can have different data types, such as numerical, categorical, or boolean.

5. Missing Values: Sometimes, data might be missing or incomplete, and handling these missing values is a crucial step in preprocessing.

Table 3.1 is an example of a simple table that might be used as input for a feedforward neural network. This table represents customer data where each row is a customer and each column is a feature of the customer.
The input table typically requires preprocessing to transform the data into a numerical format suitable for machine learning algorithms. The common preprocessing steps are:

1. Normalization: Numerical values such as Age and Income are often normalized to a range (e.g., 0 to 1) to ensure that all features contribute equally to the model's training.

2. Encoding Categorical Data: Categorical variables like Gender and Purchased are typically converted into numerical format. For instance, "Male" might be encoded as 0 and "Female" as 1. Similarly, the Purchased column could be converted to 1 for "Yes" and 0 for "No".

3. Handling Missing Values: If any values are missing, they need to be filled in or removed. For example, missing Age values could be filled with the average age or a specific value.

Table 3.2 is the output table after preprocessing. In this preprocessed table, all features are numeric and scaled, making them suitable for different machine learning algorithms. Each feature is now on a comparable scale, and categorical variables have been encoded into numerical values.

Table 3.2: Preprocessed customer data.

Customer ID	Age (Normalized)	Income (Normalized)	Gender (Male=0, Female=1)	Purchased (Yes=1, No=0)
001	0.25	0.50	0	1
002	0.32	0.60	1	0
003	0.45	0.75	1	1
004	0.29	0.55	0	0
005	0.39	0.80	1	1

3.2 Text Representation

Text representation involves converting raw text into numerical data that a machine learning model can understand. Tokenization is a crucial step in this process, as it breaks down the text into smaller units called tokens. These tokens are then converted into numerical representations (embeddings) through an embedding layer.

3.2.1 Word Tokenization

Word tokenization splits the text into individual words based on spaces and punctuation. Consider the sentence:

"Machine learning is fun!"

The word tokenization result is:

["Machine", "learning", "is", "fun", "!"]

Each word (including punctuation) is treated as a separate token.
After tokenization, each word is mapped to a unique numerical vector using an embedding layer. An embedding layer is a matrix[1] $W \in \mathbb{R}^{V \times d}$ where V is the size of the vocabulary (number of unique tokens) and d is the embedding dimension (number of features in each token's vector). The embedding layer transforms each token t_i into its corresponding vector e_i using the embedding matrix. If t_i corresponds to the index j in the vocabulary, the embedding for t_i is the j-th row of the embedding matrix. For example, if the embedding dimension is 4, a sample embedding matrix is shown in Table 3.3. Hence, the numerical representation (i.e. embedded text) of the given sentence is:

[1] The matrix can be learned using training data.

Table 3.3: Word Tokens and their corresponding embedding vectors.

Word Token	Embedding Vector
"Machine"	[0.1, 0.3, 0.5, 0.2]
"learning"	[0.4, 0.6, 0.7, 0.1]
"is"	[0.9, 0.8, 0.4, 0.3]
"fun"	[0.2, 0.1, 0.3, 0.7]
"!"	[0.6, 0.4, 0.9, 0.5]
...	...

$$\text{Embedded Text} = \begin{bmatrix} [0.1, 0.3, 0.5, 0.2] \\ [0.4, 0.6, 0.7, 0.1] \\ [0.9, 0.8, 0.4, 0.3] \\ [0.2, 0.1, 0.3, 0.7] \\ [0.6, 0.4, 0.9, 0.5] \end{bmatrix}$$

3.2.2 Character Tokenization

Character tokenization breaks the text down into individual characters, including spaces and punctuation. For the same example:

"Machine learning is fun!"

The character tokenization result is:

["M", "a", "c", "h", "i", "n", "e", " ", "l", "e", "a", "r", "n", "i", "n", "g", " ", "i", "s", " ", "f", "u", "n", "!"]

Hence, each character, including spaces and punctuation, is treated as a separate token. Each character is then mapped to a numerical vector using an embedding layer. The vocabulary size is quite limited, being equal to the number of unique characters in the given language. For example, if the embedding dimension is 4, a sample embedding matrix is shown in Table 3.4. Hence, the numerical representation (i.e. embedded text) of the given sentence is:

$$\text{Embedded Text} = \begin{bmatrix} [0.2, 0.4, 0.1, 0.8] \\ [0.7, 0.2, 0.6, 0.3] \\ [0.5, 0.9, 0.2, 0.4] \\ \cdots \end{bmatrix}$$

3.2.3 WordPiece Tokenization

WordPiece tokenization is a subword-based tokenization method used by models like BERT [17]. It breaks words into smaller units (subwords) that frequently occur

Table 3.4: The mapping of embeddings for character tokens.

Character Token	Embedding Vector
M	[0.2, 0.4, 0.1, 0.8]
a	[0.7, 0.2, 0.6, 0.3]
c	[0.5, 0.9, 0.2, 0.4]
h	[0.1, 0.7, 0.3, 0.9]
i	[0.8, 0.1, 0.4, 0.5]
...	...

Table 3.5: Subword Tokens and their Embeddings.

Subword Token	Embedding Vector
Un	[0.1, 0.5, 0.3, 0.2]
##bel	[0.4, 0.7, 0.2, 0.6]
##iev	[0.3, 0.2, 0.8, 0.9]
##able	[0.9, 0.1, 0.4, 0.5]
!	[0.6, 0.4, 0.7, 0.1]
...	...

in the text. This is especially useful for handling out-of-vocabulary (OOV) words. For example, the sentence

"Unbelievable!"

The wordPiece tokenization result is:

["Un", "##bel", "##iev", "##able", "!"]

Here, "Unbelievable!" is broken down into subwords: "Un", "bel", "iev", "able", with "##" indicating that the subword is part of a larger word. Each subword token is mapped to an embedding vector. For example, if the embedding dimension is 4, a sample embedding matrix is shown in Table 3.5. Hence, the numerical representation (i.e. embedded text) of the given sentence is:

$$\text{Embedded Text} = \begin{bmatrix} [0.1, 0.5, 0.3, 0.2] \\ [0.4, 0.7, 0.2, 0.6] \\ [0.3, 0.2, 0.8, 0.9] \\ [0.9, 0.1, 0.4, 0.5] \\ [0.6, 0.4, 0.7, 0.1] \end{bmatrix}$$

In short, word tokenization divides text into individual words, with the embedding layer transforming each word into a vector. In addition, character tokenization breaks text into individual characters, with the embedding layer converting each character

into a vector. Moreover, wordpiece tokenization segments words into smaller subwords, with the embedding layer encoding each subword as a vector. Hence, these tokenization methods and the embedding layer transform text into numerical values that can be fed into machine learning models.

3.3 Speech Representation

Speech signal processing assumes the speech signal is a *piecewise* stationary signal. As a result, a pre-processor converts the speech signal into a sequence of speech frames or acoustic observations using short time signal analysis. Typically, these frames are calculated every 10-20ms from 20-30ms windows of speech. Speech frames of these lengths are short enough that the estimated parameters can be assumed constant within each frame.
The perceptually motivated front end processing based on Mel filter bank (MFBANK) features and Mel-Frequency Cepstral Coefficients (MFCCs) [18] are the most widely used for speech recognition[2] and synthesis tasks. They are widely used in speech and audio processing to represent the short-term power spectrum of a sound signal. The output representation is a matrix of dimensions $T \times d$, where T denotes the number of frames and d represents the number of features per frame. The steps to calculate MFBANK/MFCCs from an audio signal are as follows:

1. Analog to digital conversion (sampling):

   Sampling is the process of measuring the amplitude of the analog signal at regular intervals in time. These intervals are determined by the sampling rate, f_s , which is the number of samples per second (measured in Hertz, Hz). The sampling time intervals are:

   $$t_n = n \cdot T_s, \tag{3.1}$$

   where $T_s = \frac{1}{f_s}$ is the sampling period, f_s is the sampling rate, and n is the index of the sample. The sampled signal is given by:

   $$x[n] = x(t_n), \tag{3.2}$$

   where $x(t_n)$ represents the value of the analog signal $x(t)$ at the time t_n.

[2] Feature extraction for speech recognition problems aims to find the intrinsic information related to vocal tract shape, which may be considered invariant among all speakers (i.e. invariant acoustic space). A feature vector extracted from a frame contains a set of *independent* features representing the *envelope* of the speech spectrum. The basic assumption behind this idea is that the envelope of the spectrum is a *course* representation of the spectrum that has all relevant information related to the speech recognition problem. In general, the *fine* spectral structure contains information about the excitation (i.e. details related to speakers or voicing) in a source-filter speech production model, which may be a source of noise for speech recognizers [19, 20].

According to the Nyquist-Shannon Sampling theorem, the sampling rate must be at least twice the highest frequency present in the analog signal to accurately capture it without aliasing. Example: For a speech signal with a maximum frequency of 4 kHz, a typical sampling rate would be 8 kHz or higher.

2. Pre-Emphasis: The first step is to apply a pre-emphasis filter to the signal to amplify the high frequencies, which often have lower amplitude compared to lower frequencies. The filter equation is:

$$y[n] = x[n] - \alpha \cdot x[n-1], \tag{3.3}$$

where: $x[n]$ is the original signal, $y[n]$ is the pre-emphasized signal, and α is a pre-emphasis coefficient, typically around 0.97. For example, If the original signal is $x = [1, 2, 3, 4]$, and $\alpha = 0.97$, then:

$$y = [1, 2 - 0.97 \cdot 1, 3 - 0.97 \cdot 2, 4 - 0.97 \cdot 3] = [1, 1.03, 1.06, 1.09]$$

3. Framing:

The continuous signal is divided into overlapping frames to capture temporal characteristics. Each frame typically spans 20-40 ms with an overlap of 50%. There is no specific equation, but conceptually, you can represent it as:

$$\text{Frame}_i = x[n + i \cdot \text{hop\_size}] \text{ for } n = 0 \text{ to frame\_size} - 1, \tag{3.4}$$

where i is the frame index, hop_size is the step between successive frames. For example, a signal $x = [1, 2, 3, 4, 5, 6, 7, 8]$, frame size = 4, and hop size = 2, the frames would be:

$$\text{Frame}_1 = [1, 2, 3, 4], \quad \text{Frame}_2 = [3, 4, 5, 6], \quad \text{Frame}_3 = [5, 6, 7, 8]$$

4. Windowing:

Each frame is multiplied by a window function, typically a Hamming window, to minimize the signal discontinuities at the beginning and end of each frame. The equation for windowing is :

$$y[n] = x[n] \cdot w[n], \tag{3.5}$$

where $w[n]$ is the Hamming window function:

$$w[n] = 0.54 - 0.46 \cdot \cos\left(\frac{2\pi n}{N-1}\right) \tag{3.6}$$

For a frame $x = [1, 2, 3, 4]$ and a Hamming window:

$$w = [0.08, 0.54, 0.54, 0.08],$$

$$y = [1 \cdot 0.08, 2 \cdot 0.54, 3 \cdot 0.54, 4 \cdot 0.08] = [0.08, 1.08, 1.62, 0.32]$$

5. Fast Fourier Transform (FFT):

   FFT is applied to each windowed frame to convert the time-domain signal into the frequency domain. FFT is an efficient algorithm to compute the Discrete Fourier Transform (DFT) of a sequence. The DFT converts the time-domain signal into its frequency components. For a sequence $x[n]$ of length N, the DFT is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}, \tag{3.7}$$

   where N is the number of points in the FFT, $X[k]$ is the frequency component at index k, k ranges from 0 to $N-1$, and j is the imaginary unit ($j = \sqrt{-1}$).

   Let's break down the application of the Fast Fourier Transform (FFT) to the given frame $x = [1, 1.08, 1.62, 0.32]$. This is a sequence of four values representing a discrete signal:

$$X = \text{FFT}([1, 1.08, 1.62, 0.32]) = [4.02, -0.62 - 0.76j, 1.22, -0.62 + 0.76j]$$

   Since the frame has 4 elements, the FFT will produce 4 complex values corresponding to the frequency components. For each k (ranging from 0 to 3), the FFT computes $X[k]$ as follows:

   - $X[0]$:

$$X[0] = 1 \cdot e^{-j \cdot 0} + 1.08 \cdot e^{-j \cdot 0} + 1.62 \cdot e^{-j \cdot 0} + 0.32 \cdot e^{-j \cdot 0}$$

$$X[0] = 1 + 1.08 + 1.62 + 0.32 = 4.02$$

   - $X[1]$:

$$X[1] = 1 \cdot e^{-j \cdot 0} + 1.08 \cdot e^{-j\frac{\pi}{2}} + 1.62 \cdot e^{-j\pi} + 0.32 \cdot e^{-j\frac{3\pi}{2}}$$

   Let's substitute the values of the exponentials:

$$X[1] = 1 + 1.08(-j) + 1.62(-1) + 0.32(j)$$

   Simplifying:

$$X[1] = 1 - 1.62 + (-1.08j + 0.32j)$$

$$X[1] = -0.62 - 0.76j$$

- $X[2]$:

$$X[2] = 1 \cdot e^{-j \cdot 0} + 1.08 \cdot e^{-j \cdot \pi} + 1.62 \cdot e^{-j \cdot 2\pi} + 0.32 \cdot e^{-j \cdot 3\pi}$$

Let's substitute the values of the exponentials:

$$X[2] = 1 - 1.08 + 1.62 - 0.32$$

Simplifying:

$$X[2] = (1 - 1.08) + (1.62 - 0.32)$$

$$X[2] = -0.08 + 1.30 = 1.22$$

- $X[3]$:

$$X[3] = 1 \cdot e^{-j \cdot 0} + 1.08 \cdot e^{-j\frac{3\pi}{2}} + 1.62 \cdot e^{-j\pi} + 0.32 \cdot e^{-j\frac{\pi}{2}}$$

Let's substitute the values of the exponentials:

$$X[3] = 1 + 1.08(j) + 1.62(-1) + 0.32(-j)$$

Simplifying:

$$X[3] = 1 - 1.62 + (1.08j - 0.32j)$$

$$X[3] = -0.62 + 0.76j$$

Hence, the resulting frequency components are:

$$X = [4.02, -0.62 - 0.76j, 1.22, -0.62 + 0.76j]$$

These are the complex numbers that represent the magnitude and phase of the different frequency components in the original signal. $X[0] = 4.02$ represents the DC component (frequency 0), representing the average value of the input signal. The $X[1]$ and $X[3]$ are complex conjugates and represent the positive and negative frequency components, respectively. This component $X[2] = 1.18$ represents another frequency in the signal. Each of these complex numbers can be used to understand the signal's frequency domain characteristics, including both magnitude and phase.

6. Mel Filter Bank (with Triangular Filters):

   The Mel filter bank consists of a series of triangular filters applied to the power spectrum of the signal to mimic the human ear's response to different frequencies. These filters are linearly spaced in the Mel scale but non-linearly spaced in the frequency domain. The steps to compute the triangular filters:

- Determine the Frequency Range: Let f_{min} and f_{max} be the minimum and maximum frequencies in the signal. Convert these frequencies to the Mel scale using the Mel scale formula:

$$m(f) = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right)$$

Convert f_{min} and f_{max} to Mel scale:

$$m_{\text{min}} = m(f_{\text{min}}), \quad m_{\text{max}} = m(f_{\text{max}})$$

- Determine the Number of Filters: - Decide the number of Mel filters M you want to use (typically 40-128).
- Equally Spaced Points in Mel Scale: - Calculate $M + 2$ equally spaced points between m_{min} and m_{max}:

$$m_i = m_{\text{min}} + i \cdot \frac{m_{\text{max}} - m_{\text{min}}}{M + 1}, \quad \text{for } i = 0, 1, \ldots, M + 1$$

- Convert Mel Frequencies Back to Hertz: - Convert the Mel scale points m_i back to frequencies f_i using the inverse Mel scale formula:

$$f_i = 700 \cdot \left(10^{\frac{m_i}{2595}} - 1\right)$$

- Determine the FFT Bins: - Map the frequencies f_i to the nearest FFT bin indices k_i. The FFT bin k corresponding to frequency f is given by:

$$k_i = \left\lfloor \frac{(N + 1) \cdot f_i}{f_s} \right\rfloor$$

where N is the FFT size and f_s is the sampling frequency.

- Construct the Triangular Filters: Each triangular filter $H_m(k)$ is defined over three points k_{m-1}, k_m, and k_{m+1}, corresponding to the lower, center, and upper frequencies of the filter. The triangular filter is defined as:

$$H_m(k) = \begin{cases} 0 & \text{if } k < k_{m-1} \\ \frac{k - k_{m-1}}{k_m - k_{m-1}} & \text{if } k_{m-1} \leq k \leq k_m \\ \frac{k_{m+1} - k}{k_{m+1} - k_m} & \text{if } k_m \leq k \leq k_{m+1} \\ 0 & \text{if } k > k_{m+1} \end{cases}$$

Essentially, the filter is 0 before k_{m-1} and after k_{m+1}, and it linearly rises from 0 to 1 between k_{m-1} and k_m, then linearly falls from 1 to 0 between k_m and k_{m+1}.

- Compute the Power Spectrum: Let $X[k]$ be the DFT of $x[n]$, then the power spectrum is given by:

$$P[k] = |X[k]|^2 \tag{3.8}$$

  Here, $|X[k]|^2$ represents the magnitude squared of the DFT coefficients.

- Apply Mel filters to the power spectrum: To compute the Mel filter bank energies, you convolve the power spectrum $P[k]$ with each Mel filter $H_k[m]$. In discrete time, this operation can be represented as:

$$E_m = \sum_k P[k] \cdot H_m[k], \tag{3.9}$$

  where: E_m is the Mel filter bank energy for the m-th filter.

For example, assume $f_{\text{min}} = 0$ Hz, $f_{\text{max}} = 8000$ Hz, $M = 3$ filters, $N = 512$ (FFT size), and $f_s = 16000$ Hz (sampling frequency).

To compute the Mel filter bank energies E_k exactly in the given example, follow these steps:

- Mel Points: $m_0 = 0$, $m_1 = 710$, $m_2 = 1420$, $m_3 = 2130$, $m_4 = 2840$.

- Frequencies: $f_0 = 0$ Hz, $f_1 \approx 614$ Hz, $f_2 \approx 1767$ Hz, $f_3 \approx 3933$ Hz, $f_4 \approx 8000$ Hz.

- FFT Bin Indices: $k_0 = 0$, $k_1 \approx 19$, $k_2 \approx 56$, $k_3 \approx 126$, $k_4 \approx 256$.

- Construct Triangular Filters: Filter $H_1(k)$ between $k_0 = 0$, $k_1 = 19$, and $k_2 = 56$:

$$H_1(k) = \begin{cases} 0 & \text{if } k < 0 \\ \frac{k - k_0}{k_1 - k_0} & \text{if } 0 \leq k \leq k_1 \\ \frac{k_2 - k}{k_2 - k_1} & \text{if } k_1 \leq k \leq k_2 \\ 0 & \text{if } k > k_2 \end{cases}$$

Filter $H_2(k)$ between $k_1 = 19$, $k_2 = 56$, and $k_3 = 126$:

$$H_2(k) = \begin{cases} 0 & \text{if } k < k_1 \\ \frac{k - k_1}{k_2 - k_1} & \text{if } k_1 \leq k \leq k_2 \\ \frac{k_3 - k}{k_3 - k_2} & \text{if } k_2 \leq k \leq k_3 \\ 0 & \text{if } k > k_3 \end{cases}$$

Filter $H_3(k)$ between $k_2 = 56$, $k_3 = 126$, and $k_4 = 256$:

$$H_3(k) = \begin{cases} 0 & \text{if } k < k_2 \\ \frac{k-k_2}{k_3-k_2} & \text{if } k_2 \leq k \leq k_3 \\ \frac{k_4-k}{k_4-k_3} & \text{if } k_3 \leq k \leq k_4 \\ 0 & \text{if } k > k_4 \end{cases}$$

- Compute Mel Filter Bank Energies: The power spectrum example (arbitrary values for illustration):

$$P = [1.2, 2.5, 3.0, 1.8, 2.0, \ldots]$$

- Applying Filters to Power Spectrum:

- For filter $H_1(k)$:

$$E_1 = \sum_{i=k_0}^{k_2} P[i] \cdot H_1[i]$$

where:

$$H_1[0] = 0, \quad H_1[1] = \frac{1-0}{19-0} = 0.05, \quad \text{up to } H_1[19] = 1$$

- For filter $H_2(k)$:

$$E_2 = \sum_{i=k_1}^{k_3} P[i] \cdot H_2[i]$$

where:

$$H_2[19] = 0, \quad H_2[20] = \frac{20-19}{56-19} \approx 0.03, \quad \text{up to } H_2[56] = 1$$

- For filter $H_3(k)$:

$$E_3 = \sum_{i=k_2}^{k_4} P[i] \cdot H_3[i]$$

where:

$$H_3[56] = 0, \quad H_3[57] = \frac{57-56}{126-56} \approx 0.01, \quad \text{up to } H_3[126] = 1$$

The constructed filters are shown in Figure 3.1. Finally, sum the product of each $P[i]$ with the corresponding $H_k[i]$ values to get the exact Mel filter bank energies E_1, E_2, E_3 for the filters. This process results in the Mel-scaled features used for further audio analysis.

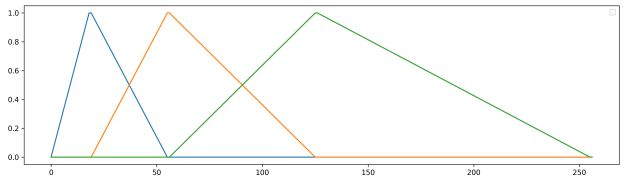


Figure 3.1: Three Mel-spaced triangle filters.

7. Logarithm of filter bank energies:

   Take the logarithm of the filter bank energies to compress the dynamic range.

$$S_m = \log(E_m)$$

   where E_m is the energy of the signal after passing through the mth filter. For example, if $Es_m = [2, 4, 6]$, then:

$$E_m = \log([2, 4, 6]) = [0.69, 1.39, 1.79]$$

   The processing is done to generate the MFBANK features at this step where the output matrix is $T \times d$ where T is the number of frames and d is the number of triangle filters.

8. Discrete Cosine Transform (DCT):

   Apply DCT to decorrelate the filter bank energies and obtain the MFCCs. The transform equation is given by:

$$c_n = \sum_{m=0}^{M-1} E_m \cdot \cos\left[\frac{\pi n(2m+1)}{2M}\right] \tag{3.10}$$

   where: M is the number of Mel filters, n is the index of the coefficient. For example,

   If $E_m = [0.69, 1.39, 1.79]$, then:

$$c_n = \text{DCT}([0.69, 1.39, 1.79]) = [3.87, -0.95262794, -0.15]$$

9. Final MFCC Coefficients:

   Typically, the first 12-13 coefficients are taken as the MFCC features, excluding the 0th coefficient which represents the average log energy of the signal. By following these steps, you can compute the MFCCs for any given audio signal.

10. Mean Removal and Delta Calculations in MFCC:

    In the context of MFCC (Mel-Frequency Cepstral Coefficients), mean removal and delta calculations are crucial steps for improving the robustness of the features. The MFCC features can be normalized by subtracting the mean of each coefficient across all frames, ensuring that the features are centered around zero. This helps in reducing the effects of noise and variations in the amplitude of the signal.

    Let X be a matrix of MFCC features where $X \in \mathbb{R}^{T \times d}$, T is the number of frames, and d is the number of MFCC coefficients per frame. - $X[t, n]$ denotes the n-th MFCC coefficient of the t-th frame. The mean μ_n of the n-th MFCC coefficient across all frames is calculated as:

$$\mu_n = \frac{1}{T} \sum_{t=1}^{T} X[t, n]$$

    The mean-removed MFCC $X'[t, n]$ is then obtained by subtracting the mean from each coefficient:

$$X'[t, n] = X[t, n] - \mu_n$$

    Since speech is a time varying signal, the basic acoustic features extracted from short time signal analysis do not capture speech dynamics. In order to consider the temporal correlation between the adjacent speech frames, the basic acoustic vector is augmented with its first (Δ) and second order derivatives ($\Delta\Delta$) as dynamic features [21]. These features are usually computed from a window of frames centered around the current frame using a simple regression method. Augmenting the feature vector with the dynamic features leads to significant improvements in recognition performance within the HMM framework.[3] The delta coefficient $\Delta X[t, n]$ is calculated using the difference between the MFCC coefficients in neighboring frames. A common approach is to use a symmetric window of size N around each frame:

$$\Delta X[t, n] = \frac{\sum_{k=1}^{K} k \cdot (X[t + k, n] - X[t - k, n])}{2 \sum_{k=1}^{K} k^2}$$

    Here, K is the window size, typically set to 2. The delta-delta coefficient $\Delta^2 X[t, n]$ is calculated similarly to the delta, but applied to the delta coefficients:

$$\Delta^2 X[t, n] = \frac{\sum_{k=1}^{K} k \cdot (\Delta X[t + k, n] - \Delta X[t - k, n])}{2 \sum_{k=1}^{K} k^2}$$

[3] The HMM framework will be described Chapter 13.

The combination of Mel filter bank energies and MFCCs[4] allows speech recognition systems to effectively model and interpret human speech. By leveraging human auditory perception, these techniques enhance the robustness and accuracy of recognition algorithms in various applications.

3.4 Image Representation

An image can be represented as a grid of individual pixels, where each pixel represents a small, uniform block of color. The pixel grid is typically organized in a 2D matrix form where each cell corresponds to a pixel with a color value. For a color image, each pixel usually has three color channels (Red, Green, Blue - RGB), represented as a 3D array (height, width, and color channels). Consider a grayscale image of size 5x5 pixels. It can be represented as a matrix:

$$\text{Image} = \begin{bmatrix} 255 & 200 & 180 & 100 & 50 \\ 230 & 210 & 190 & 120 & 80 \\ 200 & 200 & 200 & 200 & 200 \\ 150 & 140 & 130 & 120 & 110 \\ 100 & 80 & 60 & 40 & 20 \end{bmatrix}$$

Here, each number represents the intensity of a pixel in a grayscale image (0 = black, 255 = white).
Alternatively, an image can be represented as a grid of patches, where each patch is a small block or sub-region of the image. This approach is used in models like the Vision Transformer (ViT), where the image is divided into non-overlapping patches, and each patch is treated as a single unit or token. Consider a 4x4 image divided into four 2x2 patches:

$$\text{Image} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Patches (2x2):
Patch 1: [[1, 2], [5, 6]]
Patch 2: [[3, 4], [7, 8]]
Patch 3: [[9, 10], [13, 14]]
Patch 4: [[11, 12], [15, 16]]

[4]The Mel filter bank energies are usually used in neural networks based speech recognition/synthesis systems and the MFCCs are usually used in diagonal Gaussian based speech recognition systems.

The images below show an example image (i.e. Figure 3.2) along with its pixel representation (i.e. Figure 3.3). They also include a visualization of the image broken down into patches (i.e. Figure 3.4).

Figure 3.2: Example image illustrating the original visual content.

Figure 3.3: Pixel representation of the example image, showcasing individual pixel values.

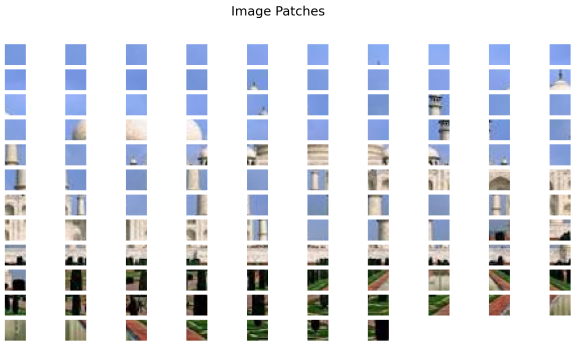


Figure 3.4: The example image divided into patches, highlighting smaller regions of the image.

3.4.1 Video Representation

A video can be thought of as a time series of images. Each frame in the video is an image, and when these frames are displayed sequentially at a certain frame rate (e.g., 24 or 30 frames per second), they create the illusion of motion.
A video is therefore represented as a four-dimensional array:

- Height: Number of pixels in each frame along the vertical axis.
- Width: Number of pixels in each frame along the horizontal axis.
- Channels: Number of color channels (e.g., RGB).
- Time (Frames): The number of frames in the video.

Assume a video of 10 frames, each of size 128x128 pixels with 3 color channels (RGB), can be represented as a tensor of shape:

$$\text{Video Shape: } (10, 128, 128, 3)$$

where: '10' is the number of frames, '128'x'128' is the size of each frame, and '3' is the number of color channels.

4. Linear Regression Networks

All models are wrong, but some are useful.

— George Box

In this chapter, we will introduce one of the most fundamental and widely used methods in statistics, machine learning, and data science: linear regression. Linear regression is a technique that allows us to model the relationship between input variables (also called predictors or features) and output variables (also called responses or targets) using a linear function.

4.1 The model

Linear regression is a method of finding the best linear relationship between input variables and output variables as shown in Figure 4.1. The input variables may be float, binary, or integer values and the output variables must be **real** or **float** values[1]. For example, if the input has 3 dimensions or variables and the output has 2 dimensions, then the relation between the inputs and outputs in the model will look like this:

$$\begin{aligned} y_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ y_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \end{aligned} \tag{4.1}$$

where each output is connected to all inputs as shown in Figure4.2. Hence, Equation (4.1) represents a fully connected or dense network. Equation (4.1) can be written in matrix form as well:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4.2}$$

[1] House price prediction is an example for predicting a float variable (i.e. house price) given the input features. The input features could be the number of rooms, area, number of bathrooms,... etc.

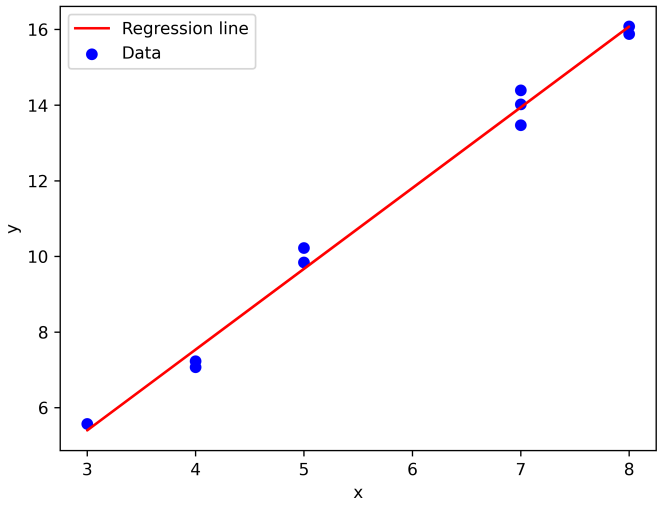


Figure 4.1: Linear function between one input variable and one output variable fitted using linear regression model.

Generally, linear regression single-layer network can be written as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{4.3}$$

where $\mathbf{y} \in \mathbb{R}^K$ is a vector of output variables, $\mathbf{x} \in \mathbb{R}^d$ is a vector of input variables (each variable is a feature), $\mathbf{W} \in \mathbb{R}^{K \times d}$ and $\mathbf{b} \in \mathbb{R}^K$ is a bias vector. The $\mathbf{W}$ and $\mathbf{b}$ are called parameters and they are estimated during the training phase using the training data.
Finding the optimal values for $\mathbf{W}$ and $\mathbf{b}$ using the training data is the subject of the next section.

4.2 Learning problem

Given a training data $(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_N, \mathbf{t}_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{t} \in \mathbb{R}^K$. In supervised learning settings, we define an objective function to measure how close the $\mathbf{t}$ to its predicted value $\mathbf{y}$ over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{W}, \mathbf{b}) = \frac{1}{2}||\mathbf{y}^n - \mathbf{t}^n|| = \frac{1}{2}\sum_{k=1}^{K}(y_k^n - t_k^n)^2 \tag{4.4}$$

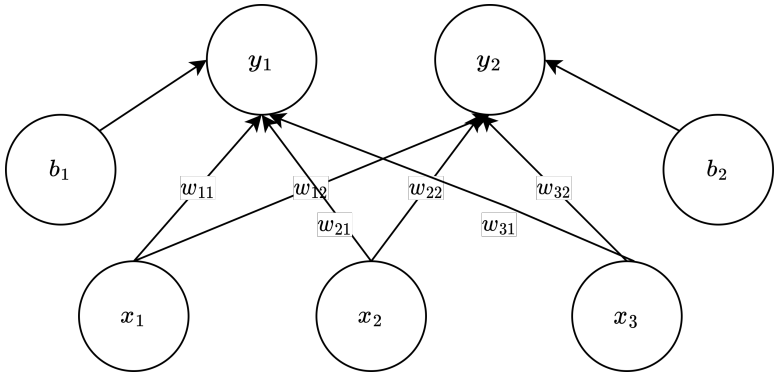


Figure 4.2: Linear regression network that has an input with 3 nodes or variables and the output has 2 nodes.

and

$$E(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{4.5}$$

where E is known as Mean Square Error (MSE) loss function and n is an index for the training sample $(\mathbf{x}_n, \mathbf{t}_n)$. Since $\mathbf{y}$ is a float vector, MSE loss function is a suitable objective function for linear regression. In order to estimate the values of $\mathbf{W}$ and $\mathbf{b}$ parameters, we minimize the loss function with respect to the parameters. The loss function measures how well the linear model fits the data, and the parameters are the coefficients and biases of the linear model. By minimizing the loss function, we can find the optimal values of the parameters that make the best predictions for the output variable.

As described in Chapter 2, linear models with MSE loss function (i.e quadratic loss function) has a unique solution. Moreover, it can be found analytically. Let's assume a multiple regression problem where we have multiple input variables and one output variable. In matrix form, the loss function is given by

$$E(\theta) = \frac{1}{2N}(X\theta - t)^T(X\theta - t) \tag{4.6}$$

where t is the output matrix of size $N \times 1$, X is the input matrix of size $N \times d + 1$ where we add the bias as an extra feature that has a value $= 1$, $\theta = \{\mathbf{W}, \mathbf{b}\}$ is the parameters matrix of size $d + 1 \times 1$. Ignoring the constant term $\frac{1}{2N}$ since it does not affect the optimization results and simplifying the cost function:

$$E(\theta) = \theta^T X^T X\theta - \theta^T X^T t - t^T X\theta + t^T t \tag{4.7}$$

Since $\theta^T X^T t = t^T X\theta =$ scalar,

$$E(\theta) = \theta^T X^T X\theta - 2t^T X\theta + t^T t \tag{4.8}$$

To find the optimal value of θ, we compute the first derivative of the cost function with respect to the parameters and set it to zero.

$$\frac{\partial E(\theta)}{\partial \theta} = 2X^T X\theta - 2X^T t = 0 \tag{4.9}$$

It can be simplified

$$X^T X\theta = X^T t \tag{4.10}$$

Hence, the analytical solution is given by

$$\theta = (X^T X)^{-1} X^T t \tag{4.11}$$

However, finding the values of the parameters $\mathbf{W}$ and $\mathbf{b}$ using the analytical solution does not scale well with the amount of training data. Hence, we will focus on the gradient descent solution in this chapter. The gradient of MSE loss function with respect to the parameters $\mathbf{W}$ and $\mathbf{b}$ can be computed as follows:

$$\frac{\partial E^n(\mathbf{W}, \mathbf{b})}{\partial w_{rs}} = \frac{\partial E^n(\mathbf{W}, \mathbf{b})}{\partial y_r} \frac{\partial y_r}{\partial w_{rs}} = (y_r^n - t_r^n) x_s^n \tag{4.12}$$

where w_{rs} is the element (r, s) of the matrix $\mathbf{W}$. Hence,

$$\frac{\partial E(\mathbf{W}, \mathbf{b})}{\partial w_{rs}} = \frac{1}{N} \sum_{n=1}^{N} (y_r^n - t_r^n) x_s^n \tag{4.13}$$

and the gradient of the loss function with respect to the bias variable b_r is given by

$$\frac{\partial E(\mathbf{W}, \mathbf{b})}{\partial b_r} = \frac{1}{N} \sum_{n=1}^{N} (y_r^n - t_r^n) \tag{4.14}$$

Using the gradient descent, the matrix of the weights and the bias vector can be updated as follows:

$$\begin{aligned} w_{rs}^{t+1} &= w_{rs}^t - \eta \frac{\partial E(\mathbf{W}, \mathbf{b})}{\partial w_{rs}} \\ b_r^{t+1} &= b_r^t - \eta \frac{\partial E(\mathbf{W}, \mathbf{b})}{\partial b_r} \end{aligned} \tag{4.15}$$

where η is the learning rate. The algorithm can update the parameters after the gradient over the whole training set is accumulated. To scale the problem to a large amount of training data, we use a variant called mini-batch stochastic gradient to estimate the parameters of the linear regression model. This algorithm will be described in the next subsection.

4.2.1 Numerical solution

Mini-batch stochastic gradient descent is a variation of gradient descent that updates the parameters of the linear regression model using a subset of the data (called a mini-batch) at each iteration. It is a numerical solution to find the global minimum of the quadratic loss function with respect to the parameters.
The idea is to reduce the computational cost and memory usage of gradient descent, while still achieving a good convergence rate.
The algorithm works as follows:

- Initialize the parameters $\mathbf{W}$ and $\mathbf{b}$ randomly or with zeros.
- Divide the data into small batches of equal size (for example, 32 or 64).
- Repeat until convergence or a maximum number of epochs:
  - For each batch:
    - Compute the predictions of the linear model for the batch.
    - Use Equation (6.7) to compute the loss function for the batch (i.e. mean squared error) .
    - Compute the gradients of the loss function with respect to the parameters for the batch using Equation (4.13) and Equation (4.14).
    - Update the parameters using Equation (5.13).
  - Return the final parameters.

The advantages of mini-batch gradient descent are:

- It can handle large datasets that do not fit in memory.
- It can exploit parallelism and vectorization to speed up computations

but it requires tuning the batch size and the learning rate hyperparameters.

4.3 Example

To illustrate the linear regression algorithm, Table 4.3 with 10 rows of random training data was created. The first three columns represent input features (x_1, x_2, x_3) and the last two columns represent output targets (t_1, t_2):

```
1
2 import numpy as np
3 from sklearn.metrics import mean_squared_error
4
5 # Training data
6 X = np.array([
```

Table 4.1: Synthesized data for linear regression modeling. The input variables are 3 and the output variables are 2 and they are float variables.

x_1	x_2	x_3	t_1	t_2
0.23	0.87	0.95	2.74	1.82
0.34	0.56	0.29	1.42	2.61
0.98	0.68	0.05	3.12	2.49
0.51	0.24	0.64	1.63	1.95
0.42	0.75	0.49	2.84	2.37
0.89	0.11	0.93	2.17	1.68
0.14	0.91	0.25	1.96	2.55
0.76	0.41	0.53	2.29	2.24
0.66	0.04	0.08	1.27	1.35
0.45	0.82	0.76	2.86	2.74

```
    [0.23, 0.87, 0.95],
    [0.34, 0.56, 0.29],
    [0.98, 0.68, 0.05],
    [0.51, 0.24, 0.64],
    [0.42, 0.75, 0.49],
    [0.89, 0.11, 0.93],
    [0.14, 0.91, 0.25],
    [0.76, 0.41, 0.53],
    [0.66, 0.04, 0.08],
    [0.45, 0.82, 0.76],
])
T = np.array([
    [2.74, 1.82],
    [1.42, 2.61],
    [3.12, 2.49],
    [1.63, 1.95],
    [2.84, 2.37],
    [2.17, 1.68],
    [1.96, 2.55],
    [2.29, 2.24],
    [1.27, 1.35],
    [2.86, 2.74],
])

# Analytical solution
def analytical_solution(X, T):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(T)
    return theta

theta_analytical = analytical_solution(X, T)
```

```
print("Analytical solution coefficients:", theta_analytical)

# Gradient descent
def gradient_descent(X, T, eta=0.1, iterations=100000):
    m, n = X.shape
    theta = np.random.randn(n + 1, T.shape[1])
    X_b = np.c_[np.ones((m, 1)), X]
    for i in range(iterations):
        gradients = 2 / m * X_b.T.dot(X_b.dot(theta) - T)
        theta -= eta * gradients
    return theta

theta_gradient_descent = gradient_descent(X, T)
print("Gradient descent coefficients:", theta_gradient_descent)

# Comparing results
Y_analytical = np.c_[np.ones((X.shape[0], 1)), X].dot(
    theta_analytical)
Y_gradient_descent = np.c_[np.ones((X.shape[0], 1)), X].dot(
    theta_gradient_descent)

mse_analytical = mean_squared_error(T, Y_analytical)
mse_gradient_descent = mean_squared_error(T, Y_gradient_descent)

print("Mean Squared Error - Analytical Solution:", mse_analytical)
print("Mean Squared Error - Gradient Descent:", mse_gradient_descent
    )
```

Listing 4.1: Python example for finding the linear regression parameters using the gradient descent algorithm. The analytical solution is provided as well.

Since linear regression has a unique solution, both analytical and gradient descent methods have the same result.

4.4 Assignment

Predict house price using the Keras deep learning library and Google colab. To load the dataset, please use the method: tf.keras.datasets.boston_housing.load_data.

5. Binary Classification Networks

Logistic regression is a powerful tool for modeling the relationship between a categorical response variable and some explanatory variables. It is especially useful when the response variable has only two possible outcomes, such as success/failure, yes/no, or healthy/sick.

— Alan Agresti

In this chapter, we will introduce a probabilistic binary classification model[1] that estimates the probability of an outcome that can only be one of two values, such as yes or no, based on one or more predictor variables. Hence, it can be used to make a binary classifier by choosing a threshold value and classifying inputs with probability greater than the threshold as one class, and below the threshold as the other class

5.1 The model

Logistic regression is a classification method that is used to predict the relationship between a binary dependent variable and one or more independent variables. As shown in Figure 5.1, the network has one output variable and many input variables. The input variables may have float, binary, or integer values, and the output **probability variable** is a float bounded between 0.0 and 1.0. For example, if the input has 3 dimensions or variables and only one output variable, then the relation between the inputs and the output in the model will look like this:

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \tag{5.1}$$

[1] Also known as logistic regression in the literature.

where the intermediate z is connected to all inputs and has a float value. Hence, Equation (5.1) represents a fully connected or dense network. It can be written in matrix form as well:

$$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b \tag{5.2}$$

Generally, logistic regression single-layer network can be written as:

$$z = \mathbf{w}^T \mathbf{x} + b \tag{5.3}$$

where $z \in R$ is a variable, $\mathbf{x} \in \mathbb{R}^d$ is a vector of input variables (each variable is a feature), $\mathbf{w} \in \mathbb{R}^d$ and b is a bias. The $\mathbf{w}$ and b are called parameters and they are estimated during the training phase using the training data.
In addition, the output y is computed in Equation (5.4) using a sigmoid transformation or activation function. It is used to map z to a float bounded between 0.0 and 1.0 as shown in Figure 5.2[2]. Naturally, the binary activation function can transform the input to output that has a value 0 or 1 as shown in Figure 5.3. However, this function is not continuous and not differentiable at 0. This explains why sigmoid was selected as a transformation method for binary classification. The sigmoid activation function is a continuous and differentiable function. Hence, we can compute the gradient of a loss function based on that activation function.

$$y = \frac{1}{1 + e^{-z}} \tag{5.4}$$

Finding the optimal values for $\mathbf{w}$ and b using the training data is the subject of the next section.

5.2 Learning Problem

Given a training data $(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), \ldots, (\mathbf{x}_N, t_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{w}$ and b where $\mathbf{x} \in \mathbb{R}^d$ and $t \in \{1, 0\}$. We define an objective function to measure how close the t to its predicted value y over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{w}, b) = -(t^n \log p^n + (1 - t^n) \log(1 - p^n)) \tag{5.5}$$

and

$$E(\mathbf{w}, b) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{5.6}$$

[2]It is so called because its graph is "S-shaped".

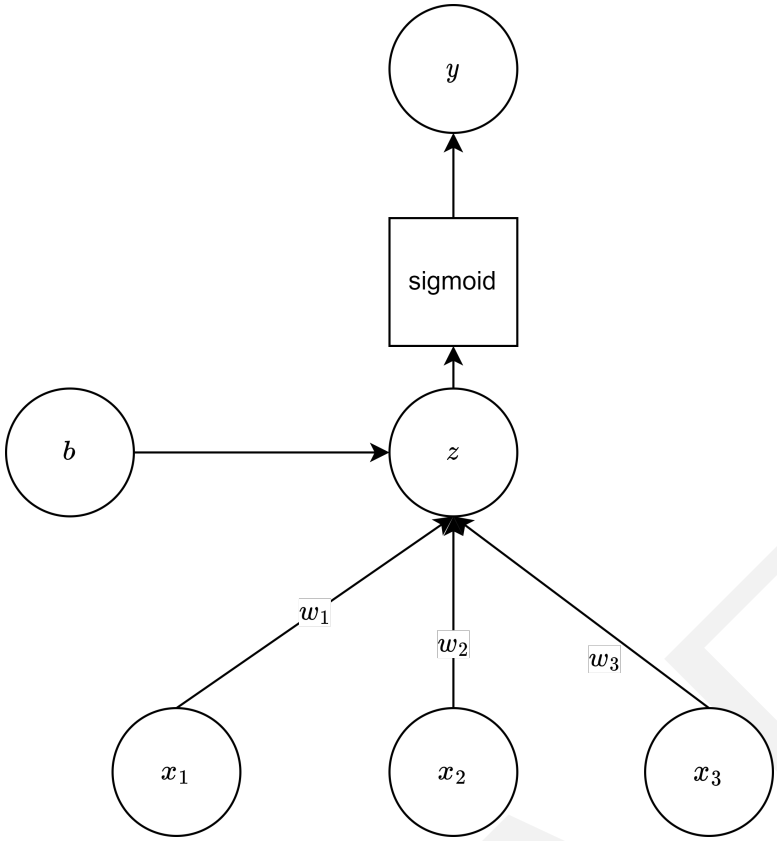


Figure 5.1: A logistic regression model (i.e. binary classification model) must have only one output variable and optional count of input variables.

where E is known as the binary cross-entropy (BCE) loss function and n is an index for the training sample $(\mathbf{x}_n, t_n)$. The loss function is plotted in Figure 5.4. The loss increases exponentially as the predicted probability of the true class gets closer to zero. Since y is a float value between 0 and 1, the BCE loss function is a matching objective function for logistic regression. In order to estimate the values of $\mathbf{w}$ and b parameters, we minimize the loss function with respect to the parameters.
The logistic regression model with BCE loss function has a unique solution[3]. The gradient of BCE loss function with respect to the parameters $\mathbf{w}$ and b can be computed as follows:

$$\frac{\partial E^n(\mathbf{w}, b)}{\partial w_i} = \frac{\partial E^n(\mathbf{w}, b)}{\partial y^n} \frac{\partial y^n}{\partial z^n} \frac{\partial z^n}{\partial w_i} \tag{5.7}$$

where w_i is the element i of the vector $\mathbf{w}$. The three terms can be computed as

[3]It can not be found analytically.

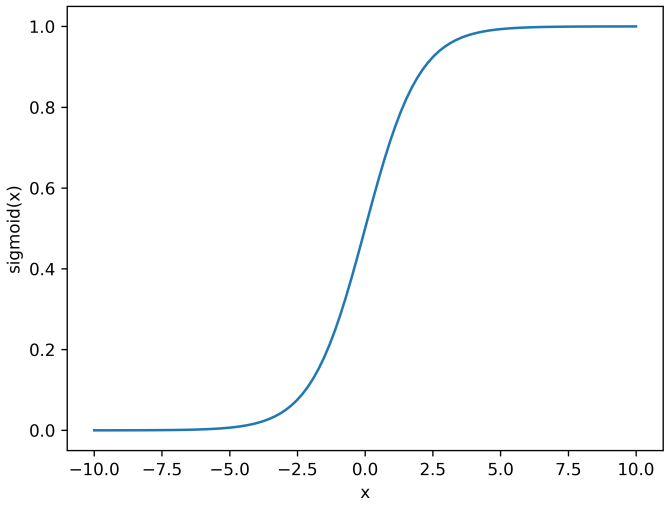


Figure 5.2: Sigmoid activation function.

follows:

$$\begin{aligned}\frac{\partial E^n(\mathbf{w}, b)}{\partial y^n} &= -\frac{t^n}{y^n} + \frac{1-t^n}{1-y^n} \\ &= \frac{-(1-y^n)t + y^n(1-t^n)}{y^n(1-y^n)} \\ &= \frac{(y^n - t^n)}{y^n(1-y^n)}\end{aligned} \tag{5.8}$$

Using Equation 5.4, the gradient of the output y^n with respect to z^n:

$$\begin{aligned}\frac{\partial y^n}{\partial z^n} &= \frac{0 - e^{-z^n}(-1)}{(1+e^{-z^n})^2} \\ &= \frac{1}{(1+e^{-z^n})} \cdot \frac{e^{-z^n}}{(1+e^{-z^n})} \\ &= y^n \cdot \frac{e^{-z^n} + 1 - 1}{(1+e^{-z^n})} \\ &= y^n(1 - \frac{1}{(1+e^{-z^n})}) \\ &= y^n(1-y^n)\end{aligned} \tag{5.9}$$

And $\frac{\partial z^n}{\partial w_i}$ is given by

$$\frac{\partial z^n}{\partial w_i} = x_i^n \tag{5.10}$$

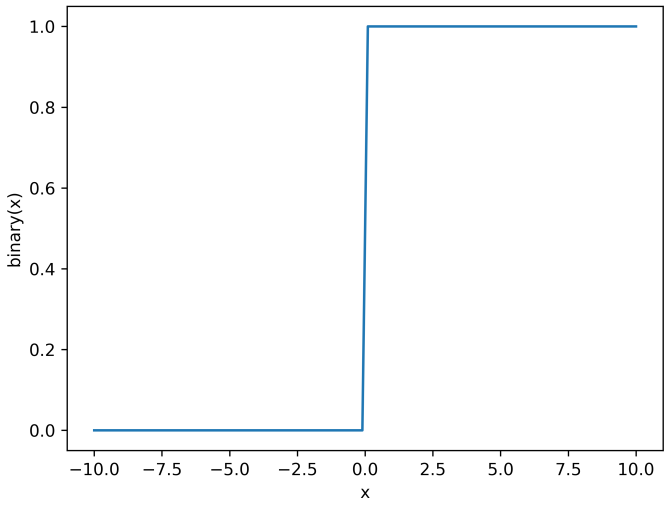


Figure 5.3: Binary activation function.

Hence, Equation (5.7) can be written using the three terms computed above as follows:

$$\begin{aligned}\frac{\partial E^n(\mathbf{w}, b)}{\partial w_i} &= \frac{(y^n - t^n)}{y^n(1 - y^n)} y^n(1 - y^n) x_i^n \\ &= (y^n - t^n) x_i\end{aligned} \tag{5.11}$$

and the gradient of the loss function with respect to the bias variable b is given by

$$\frac{\partial E^n(\mathbf{w}, b)}{\partial b} = (y^n - t^n) \tag{5.12}$$

Using the gradient descent, the vector of the weights and the bias term can be updated as follows:

$$\begin{aligned}w_i^{t+1} &= w_i^t - \eta \frac{1}{N} \sum_{n=1}^{N} (y^n - t^n) x_i^n \\ b^{t+1} &= b^t - \eta \frac{1}{N} \sum_{n=1}^{N} (y^n - t^n)\end{aligned} \tag{5.13}$$

where η is the learning rate. Using mini-batch stochastic gradient descent, it is possible to learn the parameters efficiently for large datasets. The algorithm is very similar to the one described in section 4.2.1.

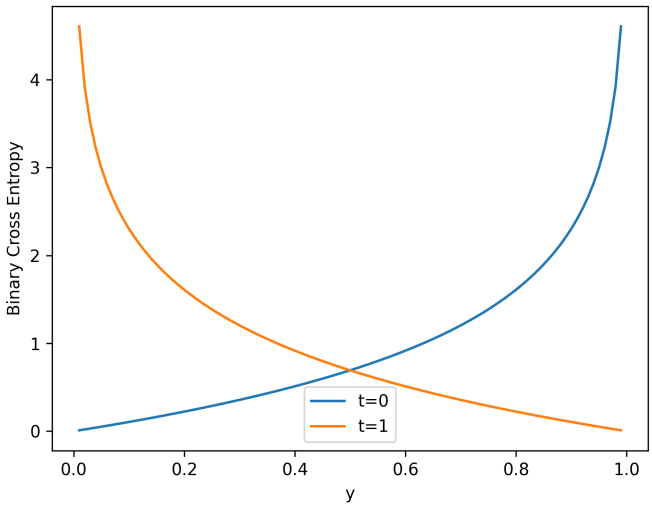


Figure 5.4: Binary cross-entropy objective function.

5.3 Classification Decision

The output of the learning algorithm is to estimate the values of the $\mathbf{w}$ and b parameters. Hence, they can be used at the test time for prediction. Since y is a float value between 0 and 1, it can not be used directly to classify the input samples to 0 or 1. To overcome this problem, we define a threshold for decision

$$\hat{y} = \begin{cases} 1 & \text{if } y \geq 0.5 \\ 0 & \text{if } y < 0.5 \end{cases}$$

where $\hat{y}$ is the final decision either 0 or 1. Logistic regression separates the two classes with a linear decision boundary. The linear decision boundary of logistic regression is the set of all points $\mathbf{x}$ that satisfy:

$$P(y = 1|\mathbf{x}) = P(y = 0|\mathbf{x}) = \frac{1}{1 + e^{-z}} = \frac{1}{2} \tag{5.14}$$

then

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b = 0 \tag{5.15}$$

For two-dimensional data

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{5.16}$$

Hence,

$$x_2 = -\frac{b}{w_2} - \frac{w_1}{w_2} x_1 \tag{5.17}$$

For example, the two-dimensional AND gate[4] is shown in Table 5.5. The first two columns represent input features (x_1, x_2) and the last column columns represent output targets t. The decision boundary is shown in Figure 5.5.

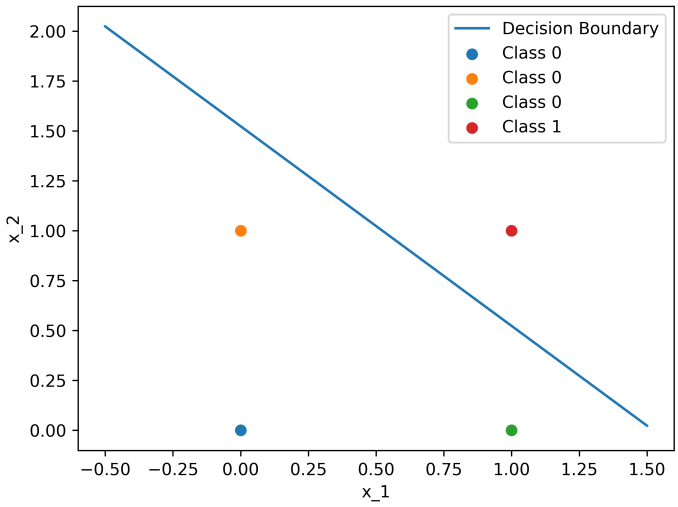


Figure 5.5: The decision boundary for the two-dimensional "AND" gate.

5.4 Evaluation

Binary classification evaluation metrics help assess the performance of a model that categorizes instances into one of two classes: typically a "positive" class (e.g., a disease is present) and a "negative" class (e.g., a disease is absent). The confusion matrix is the starting point for binary classification metrics as shown in Table 5.1. It is a 2×2 table that categorizes predictions:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Table 5.1: The binary classification confusion matrix.

Below is a detailed breakdown of common evaluation metrics, including precision, recall, F1-score, and AUC (Area Under the Curve):

[4]The AND gate is known in logic design literature.

1. Precision: Precision measures the accuracy of positive predictions and is defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5.18}$$

   Precision is important when false positives are costly, for instance in medical diagnoses, where a false positive could lead to unnecessary treatments.

2. Recall (Sensitivity or True Positive Rate): Recall indicates the model's ability to correctly identify actual positives and is defined as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.19}$$

   High recall is crucial when false negatives are costly, such as in spam detection, where failing to flag a spam email is more problematic than accidentally flagging a valid one.

3. F1-Score: The F1-score is the harmonic mean of precision and recall, balancing the two metrics and penalizing extreme values in either. It's particularly useful when the class distribution is imbalanced:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.20}$$

   A high F1-score implies a good balance between precision and recall, meaning the model performs well on both detecting positives and minimizing false positives.

4. Area Under the Curve (AUC): The Area Under the Curve (AUC), specifically for the Receiver Operating Characteristic (ROC) curve, evaluates a model's ability to distinguish between classes across various threshold settings. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR), with TPR as:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.21}$$

   and FPR as:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{5.22}$$

- Initialize variables for TPR and FPR, typically starting at (0, 0) on the ROC curve.
- For each unique threshold t, predict positive for all instances with scores greater than t.
- Calculate the TPR and FPR for the current threshold.
- Calculate AUC $= \int_0^1 TPR(FPR)d(FPR)$ using the Trapezoidal Rule: The AUC can be computed by summing the areas of trapezoids formed by each pair of consecutive points $(\text{FPR}_i, \text{TPR}_i)$ and $(\text{FPR}_{i+1}, \text{TPR}_{i+1})$ on the ROC curve:

$$\text{AUC} = \sum_{i=1}^{n-1} (\text{FPR}_{i+1} - \text{FPR}_i) \cdot \frac{\text{TPR}_{i+1} + \text{TPR}_i}{2} \tag{5.23}$$

  This formula computes the area for each trapezoid under the curve and sums them up to get the total AUC (see the Python implementation for the AUC calculations in Listing 5.1). The AUC is the area under this ROC curve, ranging from 0 to 1 (see Figure 5.6). A model with an AUC of 0.5 performs no better than random guessing, while an AUC close to 1 indicates a strong ability to differentiate between classes. Since AUC-ROC evaluates model performance at multiple thresholds, it provides a threshold-independent metric that generalizes well across different contexts. When the data is extremely imbalanced, meaning the positive class is rare compared to the negative class, AUC-ROC may provide an overly optimistic view. In such cases, even a model that performs poorly in identifying the minority class can yield a high AUC score, as the metric does not give adequate emphasis to the minority class. Metrics like Precision-Recall AUC, which focuses on the positive class, might be more insightful.

Each of these metrics provides unique insights into model performance, with precision and recall balancing error types, F1-score balancing precision and recall, and AUC reflecting general model discrimination ability across thresholds.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.metrics import roc_auc_score

# True labels and predicted probabilities
true_labels = np.array([0, 1, 1, 0, 1, 0, 1, 0])
predicted_probs = np.array([0.1, 0.9, 0.8, 0.3, 0.6, 0.2, 0.95,
    0.4])

```

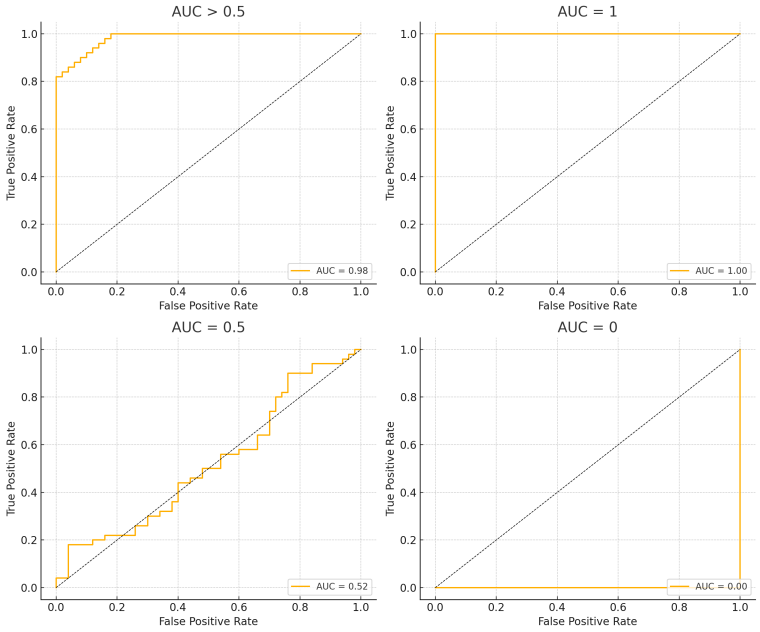


Figure 5.6: The plot shows the ROC curves for different AUC values: AUC>0.5 (the classifier is likely to effectively separate positive class values from negative ones, as it correctly identifies a greater number of True Positives and True Negatives compared to False Positives and False Negatives.), AUC=1 (perfect classifier), AUC=0.5 (random guess), and AUC=0 (an inverse predictor is a classifier that incorrectly labels all negative instances as positive and all positive instances as negative.).

```
# Define thresholds
thresholds = np.linspace(0, 1, 11)  # 0.0 to 1.0 in steps of 0.1

# Calculate TPR and FPR for each threshold
tpr = []
fpr = []
for thresh in thresholds:
    predictions = (predicted_probs >= thresh).astype(int)
    tp = np.sum((predictions == 1) & (true_labels == 1))
    fn = np.sum((predictions == 0) & (true_labels == 1))
    fp = np.sum((predictions == 1) & (true_labels == 0))
    tn = np.sum((predictions == 0) & (true_labels == 0))

    tpr.append(tp / (tp + fn) if (tp + fn) > 0 else 0)
    fpr.append(fp / (fp + tn) if (fp + tn) > 0 else 0)

# Calculate AUC using the trapezoidal rule
auc = np.trapz(tpr, fpr)
```

```
print(f"AUC: {auc:.3f}")

# Verify with sklearn's roc_auc_score
print(f"AUC (sklearn): {roc_auc_score(true_labels, predicted_probs)
    :.3f}")


plt.plot(fpr, tpr, marker='o', label=f"AUC = {auc:.3f}")
plt.plot([0, 1], [0, 1], 'r--', label="Random Guess")
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

Listing 5.1: The script computes the AUC using manually calculated TPR and FPR values.

5.4.1 F1 Curve and threshold tuning

In binary or multilabel classification (see Chapter 7), threshold tuning involves finding an optimal decision threshold for converting predicted probabilities into binary class labels (0 or 1). By adjusting this threshold, we can influence metrics like precision, recall, and ultimately, the F1 score. Plotting the F1 score as a function of the threshold helps visualize how performance changes with the threshold and find an optimal balance between precision and recall for the specific task.
Typically, classifiers produce a probability score, $p \in [0, 1]$, indicating the likelihood of an instance belonging to a particular class. The default threshold is usually set to 0.5; if $p \geq 0.5$, the instance is classified as the positive class; otherwise, it's classified as the negative class (see Section 5.3). Increasing the threshold (e.g., 0.7 or 0.8) usually increases precision since only higher confidence predictions are considered positive. However, this can lower recall, as fewer positive instances are captured. Lowering the threshold (e.g., 0.3 or 0.2) generally increases recall but might reduce precision since more instances, including those with low probability scores, are classified as positive. The F1 score is the harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Since F1 balances precision and recall, tuning the threshold for the highest F1 can help achieve an effective tradeoff between the two. To identify the optimal threshold, we compute the F1 score at different thresholds, typically ranging from 0 to 1, and plot the results. The steps to Generate the F1 Curve:

- Split the dataset into training, development, and test sets.

- Train the model on the training set.
- On the development set, use the model to output probability scores for each instance.
- Define a range of thresholds, e.g., threshold $\in \{0.0, 0.1, 0.2, \ldots, 1.0\}$.
- For each threshold, calculate predicted labels and then compute precision, recall, and F1 score.
- Plot the F1 score on the y-axis and the threshold on the x-axis.
- The peak of the F1 curve indicates the threshold with the highest F1 score.
- The threshold at the peak of the F1 curve maximizes the F1 score and represents a balance between precision and recall.
- Depending on the application, you might also consider thresholds where F1 score is near-maximal but skewed toward higher precision or recall if one is more critical.

As an example, let's say we have a range of threshold values $\{0.1, 0.2, \ldots, 0.9\}$ and the computed F1 for each threshold are:

Threshold	Precision	Recall	F1
0.1	0.65	0.95	0.77
0.2	0.70	0.90	0.79
0.3	0.75	0.85	0.80
0.4	0.80	0.82	0.81
0.5	0.82	0.78	0.80
0.6	0.85	0.75	0.80
0.7	0.88	0.70	0.78
0.8	0.90	0.65	0.76
0.9	0.92	0.60	0.73

Table 5.2: An example of F1 calculations using different thresholds.

From this table, we observe that the F1 score peaks at a threshold of 0.4 with an F1 score of 0.81. This is the optimal threshold for balancing precision and recall in this example. A Python code example for plotting F1 Curve are listed below:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_score, recall_score, f1_score

# Sample probability predictions and true labels
```

Table 5.3: Two-dimensional AND gate truth table.

x_1	x_2	t
0	0	0
0	1	0
1	0	0
1	1	1

```
probabilities = np.random.rand(100)  # simulated probabilities
true_labels = np.random.randint(0, 2, size=100)  # simulated binary
    labels

thresholds = np.linspace(0, 1, 50)
f1_scores = []

for threshold in thresholds:
    predictions = (probabilities >= threshold).astype(int)
    f1 = f1_score(true_labels, predictions)
    f1_scores.append(f1)

# Plotting the F1 score vs. threshold
plt.figure(figsize=(10, 6))
plt.plot(thresholds, f1_scores, marker='o', color='b', label='F1
    Score')
plt.title('F1 Score vs. Decision Threshold')
plt.xlabel('Threshold')
plt.ylabel('F1 Score')
plt.legend()
plt.grid()
plt.show()
```

Listing 5.2: A sample Python code to plot the F1 score against thresholds using a binary classifier's probability predictions.

Maximizing the F1 approach is particularly effective in maximizing the classifier's performance in scenarios where both precision and recall are essential and provides insight into how adjusting the decision threshold impacts classifier behavior.

5.5 An Example

A Python code is provided to illustrate the logistic regression learning algorithm and plot the decision boundary:

```

import numpy as np
import matplotlib.pyplot as plt

```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def predict(X, w):
    return sigmoid(np.dot(X, w))

def cost_function(X, t, w):
    N = len(t)
    y = predict(X, w)
    E = -1/N * (np.dot(t.T, np.log(y)) + np.dot((1-t).T, np.log(1-y)
    ))
    return E

def gradient_descent(X, t, w, alpha, iterations):
    m = len(t)
    E_history = []
    for i in range(iterations):
        y = predict(X, w)
        w = w - alpha * (1/m) * np.dot(X.T, (y-t))
        E_history.append(cost_function(X, t, w))
    return w, E_history

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
t = np.array([[0], [0], [0], [1]])

m = len(t)
X = np.hstack((np.ones((m, 1)), X))

n = X.shape[1]
w = np.zeros((n, 1))

alpha = 0.01
iterations = 100000

w_final, E_history = gradient_descent(X, t, w, alpha, iterations)

print (predict(X, w_final))

x_1 = np.linspace(-0.5, 1.5)
x_2 = -(w_final[0] + w_final[1]*x_1) / w_final[2]

plt.plot(x_1,x_2,label='Decision Boundary')
plt.scatter(0,0,label='Class 0')
plt.scatter(0,1,label='Class 0')
plt.scatter(1,0,label='Class 0')
plt.scatter(1,1,label='Class 1')
plt.xlabel('x_1')
plt.ylabel('x_2')
plt.legend()
```

```
53 plt.savefig('decision_boundary_and.png')
```

Listing 5.3: Python example for plotting the logistic regression decision boundary for AND Problem.

Since logistic regression has a unique solution, running the script several times will lead to the same result.

5.6 Assignment

Spam email detection (binary classification task) using the Keras deep learning library and Google colab. To load the dataset, please visit https://archive.ics.uci.edu/ml/datasets/spambase.

6. Multiclass Classification Networks

When we make inferences based on incomplete information, we should draw them from that probability distribution that has the maximum entropy permitted by the information we do have.

— E. T. Jaynes

In this chapter, we will introduce the probabilistic multiclass or multinomial classification algorithm[1]. It aims to classify the input samples into one of three or more classes (for two classes only see Chapter 5 of binary classification).

6.1 The model

A multiclass softmax classifier is a supervised learning algorithm that can handle multiple classes. It assigns a probability to each class based on the input features and the learned parameters. The input variables may be float, binary, or integer values and the output variables must be categorical variables or class labels. Categorical variables or class labels in multiclass problem setting are usually encoded using one-hot vectors. A one-hot vector is a vector that has only one element with a value of 1 and the rest are 0. For example, if there are three classes A, B, C and D, we can use the following one-hot vectors to represent them:

A: [1, 0, 0]
B: [0, 1, 0]
C: [0, 0, 1]

The advantage of using one-hot vectors is that they can be easily used with models that output probabilities for each class.

[1] It is known as the softmax [22] or maximum entropy classifier [23]

If the input has 4 dimensions or variables and the output has 3 classes, then the relation between the inputs and outputs in the model will look like this:

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 + b_2 \\ z_3 &= w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 + b_3 \end{aligned} \tag{6.1}$$

where each output is connected to all inputs as shown in Figure 6.1. Hence, Equation (6.1) represents a fully connected or dense network. Equation (6.1) can be written in matrix form as well:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{6.2}$$

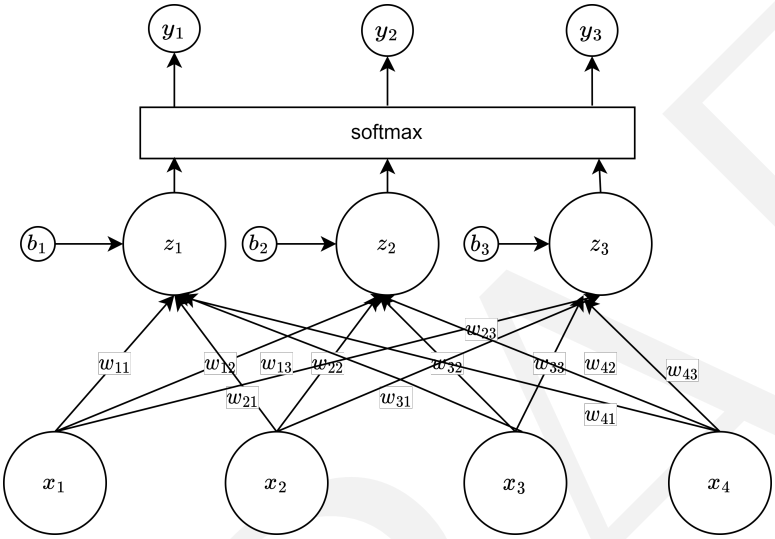


Figure 6.1: Multiclass classification network that has an input with 4 nodes or variables and the output has 3 nodes or classes.

Generally, softmax regression single-layer network can be written as:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{6.3}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z}) \tag{6.4}$$

and the elements of the softmax y_i are given by

$$y_i = \frac{e^{z_i}}{\sum_{c=1}^{K} e^{z_c}} \text{ for } i = 1, \ldots, K \tag{6.5}$$

where $\mathbf{y} \in \mathbb{R}^K$ is a vector of output classes or variables (the softmax transform is a normalized function, meaning that all elements of the output vector $\mathbf{y}$ are in the range (0, 1) and sum up to 1), $\mathbf{x} \in \mathbb{R}^d$ is a vector of input variables (each variable is a feature), $\mathbf{W} \in \mathbb{R}^{K \times d}$ and $\mathbf{b} \in \mathbb{R}^K$ is a bias vector. The $\mathbf{W}$ and $\mathbf{b}$ are called parameters and they are estimated during the training phase using the training data.
Finding the optimal values for $\mathbf{W}$ and $\mathbf{b}$ using the training data is the subject of the next section.

6.2 Learning problem

Given a training data $(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_N, \mathbf{t}_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{t}$ is a one-hot vector of length K can be mathematically denoted as an element of the set $0, 1^K$ that has exactly one element equal to 1 and the rest equal to 0. In supervised learning settings, we define an objective function to measure how close the $\mathbf{t}$ to its predicted value $\mathbf{y}$ over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{W}, \mathbf{b}) = -\sum_{k=1}^{K} t_k^n \log y_k^n \tag{6.6}$$

and

$$E(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{6.7}$$

where E is known as categorical cross-entropy loss function and n is an index for the training sample $(\mathbf{x}_n, \mathbf{t}_n)$. Since $\mathbf{y}$ is a float vector (its elements float between 0 and 1), the categorical cross-entropy loss function is a suitable objective function for softmax regression. In order to estimate the values of $\mathbf{W}$ and $\mathbf{b}$ parameters, we minimize the loss function with respect to the parameters. The loss function measures how well the softmax model fits the data, and the parameters are the coefficients and biases of the model. By minimizing the loss function, we can find the optimal values of the parameters that make the best predictions for the output classed. The softmax regression model with the categorical cross-entropy loss function has a unique solution[2].
In order to derive the gradient of the objective function with respect to the parameters of the softmax classifier, we will need the derivative of the softmax function with respect to its inputs. The gradient of softmax with respect to its inputs is a matrix

[2]It can not be found analytically.

known as the Jacobian matrix[3]. It is KxK matrix and it is given by:

$$\begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \frac{\partial y_1}{\partial z_3} & \cdots & \frac{\partial y_1}{\partial z_K} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \frac{\partial y_2}{\partial z_3} & \cdots & \frac{\partial y_2}{\partial z_K} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial z_1} & \frac{\partial y_K}{\partial z_2} & \frac{\partial y_K}{\partial z_3} & \cdots & \frac{\partial y_K}{\partial z_K} \end{pmatrix} \tag{6.8}$$

When $k = i$ (i.e. diagonal elements of the matrix):

$$\begin{aligned} \frac{\partial y_k}{\partial z_i} &= \frac{e^{z_i} \sum_j e^{z_j} - e^{z_k} e^{z_i}}{(\sum_j e^{z_j})^2} \\ &= \frac{e^{z_i}}{\sum_j e^{z_j}} \frac{\sum_j e^{z_j} - e^{z_k}}{\sum_j e^{z_j}} \\ &= y_i(1 - y_k) \\ &= y_i - y_i y_k \end{aligned} \tag{6.9}$$

and when $k \neq i$:

$$\frac{\partial y_k}{\partial z_i} = \frac{-e^{z_i} e^{z_k}}{(\sum_j e^{z_j})^2} = -\frac{e^{z_i}}{\sum_j e^{z_j}} \frac{e^{z_k}}{\sum_j e^{z_j}} = -y_i y_k \tag{6.10}$$

Hence, the two equations can be combined into one equation:

$$\frac{\partial y_k}{\partial z_i} = y_i(\delta_{ik} - y_k) \tag{6.11}$$

where Kronecker delta δ_{ik} is defined as follows:

$$\delta_{ik} = \begin{cases} 0 \text{ when } i \neq k \\ 1 \text{ when } i = k \end{cases} \tag{6.12}$$

It can be written using two separate matrices:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} y_1 & 0 & 0 & \cdots & 0 \\ 0 & y_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & y_K \end{pmatrix} - \begin{pmatrix} y_1 y_1 & y_1 y_2 & y_1 y_3 & \cdots & y_1 y_K \\ y_2 y_1 & y_2 y_2 & y_2 y_3 & \cdots & y_2 y_K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_K y_1 & y_K y_2 & y_K y_3 & \cdots & y_K y_K \end{pmatrix} \tag{6.13}$$

Using Equation (6.11), the

$$E^n(\mathbf{W}, \mathbf{b}) = -\sum_{k=1}^{K} t_k^n \log y_k^n \tag{6.14}$$

[3]The softmax function is a vector and differentiating a vector with respect to a vector of parameters generates a matrix.

$$
\begin{aligned}
\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial w_{ij}} &= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{\partial E^n(\mathbf{W},\mathbf{b})}{\partial y_k^n}\frac{\partial y_k^n}{\partial z_i^n}\frac{\partial z_i^n}{\partial w_{ij}} \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left(\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}y_i^n(\delta_{ik}-y_k^n)\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}y_i^n(y_k^n-\delta_{ik})\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n\sum_{k=1}^{K}t_k^n - y_i^n\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}\delta_{ik}\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n - t_i^n\right)x_j^n,
\end{aligned}
\tag{6.15}
$$

where $\sum_{k=1}^{K} t_k^n = 1.0$. Similarly, the gradient of the cross-entropy loss function with respect to the bias b_i:

$$
\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n - t_i^n\right) \tag{6.16}
$$

Using the gradient descent, the vector of the weights and the bias term can be updated as follows:

$$
\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^{t} - \eta\frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)x_j^n \\
b_i^{t+1} &= b_i^{t} - \eta\frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)
\end{aligned}
\tag{6.17}
$$

where η is the learning rate. Using mini-batch stochastic gradient descent, it is possible to learn the parameters efficiently for large datasets. The algorithm is very similar to the one described in section 4.2.1.

6.3 Classification decision

The output of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ parameters. Hence, they can be used at the test time for prediction. The class with the highest probability is the predicted class. To find this winner class $\hat{y}$ for a given

sample:

$$\begin{aligned}\hat{y} &= \arg \max_{1 \le k \le K} (y_k) \\ &= \arg \max_{1 \le k \le K} (z_k)\end{aligned} \tag{6.18}$$

The softmax classifier separates the classes with linear decision boundaries. To find the decision boundary for two-dimensional data with three classes, let:

$$\begin{aligned}z_1 &= w_{11}x_1 + w_{12}x_2 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + b_2 \\ z_3 &= w_{31}x_1 + w_{32}x_2 + b_3\end{aligned} \tag{6.19}$$

We have three decision boundaries between classes 1 and 2, 1 and 3, and 2 and 3. The linear decision boundary between classes 1 and 2 must satisfy:

$$w_{11}x_1 + w_{12}x_2 + b_1 = w_{21}x_1 + w_{22}x_2 + b_2 \tag{6.20}$$

then

$$(w_{11} - w_{21})x_1 + (w_{12} - w_{22})x_2 = b_2 - b_1 \tag{6.21}$$

For example, the decision boundaries are shown in Figure 6.2 (details in the next section).

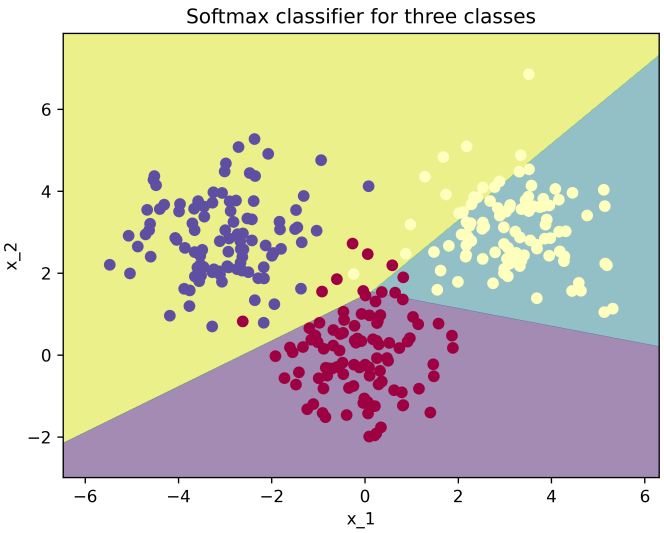


Figure 6.2: The decision boundaries between three classes for the two-dimensional data.

6.4 Example

A Python code is provided to illustrate the learning algorithm of the softmax classifier and plot the decision boundaries between three classes:

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def softmax(x):
6      x = x - np.max(x, axis=1, keepdims=True)
7      exp_x = np.exp(x)
8      return exp_x / np.sum(exp_x, axis=1, keepdims=True)
9
10 def cross_entropy(y_true, y_pred):
11     y_pred = np.clip(y_pred, 1e-12, 1 - 1e-12)
12     return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
13
14 def gradient_descent(X, y_true, y_pred, w, b, learning_rate):
15     m = X.shape[0]
16     dw = (1/m) * np.dot(X.T, (y_pred - y_true))
17     db = (1/m) * np.sum(y_pred - y_true, axis=0)
18     w = w - learning_rate * dw
19     b = b - learning_rate * db
20     return w, b
21
22 np.random.seed(42)
23 X0 = np.random.multivariate_normal(mean=[0, 0], cov=[[1, 0], [0,
       1]], size=100)
24 y0 = np.array([[1, 0, 0]] * 100)
25 X1 = np.random.multivariate_normal(mean=[3, 3], cov=[[1, 0], [0,
       1]], size=100)
26 y1 = np.array([[0, 1, 0]] * 100)
27 X2 = np.random.multivariate_normal(mean=[-3, 3], cov=[[1, 0], [0,
       1]], size=100)
28 y2 = np.array([[0, 0, 1]] * 100)
29
30 X = np.concatenate((X0, X1, X2), axis=0)
31 y_true = np.concatenate((y0, y1, y2), axis=0)
32
33 w = np.random.randn(2, 3)
34 b = np.random.randn(3)
35
36 epochs = 100
37 learning_rate = 0.01
38
39 for epoch in range(epochs):
40     y_pred = softmax(np.dot(X, w) + b)
41     loss = cross_entropy(y_true, y_pred)
42     if epoch % 10 == 0:
```

```
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
    w, b = gradient_descent(X, y_true, y_pred, w, b, learning_rate)

# Plotting the decision boundaries
x_min = X[:, 0].min() - 1
x_max = X[:, 0].max() + 1
y_min = X[:, 1].min() - 1
y_max = X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min,x_max,.01),np.arange(y_min,
    y_max,.01))
Z=np.argmax(softmax(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b),
    axis=1).reshape(xx.shape)

plt.contourf(xx ,yy ,Z ,alpha=.5)

plt.scatter(X[:,0],X[:,1],c=np.argmax(y_true,axis=1), cmap=plt.cm.
    Spectral)
plt.xlabel("x_1")
plt.ylabel("x_2")

plt.show()
plt.savefig("softmax_classifier.png", dpi =600)
```

Listing 6.1: Python example for plotting the softmax decision boundaries for three classes problem.

Since softmax regression has a unique solution, running the script several times will lead to the same result.

6.5 Assignment

Implement an optical recognition of handwritten digits (multiclass classification task) using the Keras deep learning library and Google colab. To load the dataset, please visit https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits.

7. Multilabel Classification Networks

Multilabel classification is a classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the "cat" label and the "dog" label.

— François Chollet

In this chapter, we introduce the multilabel classification algorithm where each instance can belong to more than one class at the same time. For example, a document can have multiple topics, such as politics, religion, and education. The multilabel classification is different from the multiclass classification where each data point or instance must belong to one class only.

7.1 Model

Multilabel classification is a variant of the classification problem where multiple nonexclusive labels may be assigned to each instance. For example, suppose you want to classify a movie based on its genres. A movie can belong to more than one genre, such as comedy, romance, and action. In this case, the input is the movie and the output is a set of genres that describe the movie (three genre in total). For example, a movie that is only comedy would have an output of:

$$t = [1\ 0\ 0]$$

A movie that is both romance and action would have an output of:

$$t = [0\ 1\ 1]$$

A movie that has no genres would have an output of:

$$t = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

This output is called a multi-hot vector, where each element corresponds to a label and has a value of 0 or 1 depending on whether the label is present or not. A movie can have any combination of genres, so there are $2^3 = 8$ possible outputs for this problem.

One way to approach this problem is to transform it into binary classification problems, where each label is predicted independently by a binary logistic regression classifier. The target of such classifiers can be represented as a multi-hot vector, where each element corresponds to a label and has a value of 0 or 1 depending on whether the label is predicted or not. The model can be written as:

$$\begin{aligned} y &= \begin{bmatrix} y_1 & y_2 \vdots y_L \end{bmatrix} \\ &= \begin{bmatrix} \sigma(\mathbf{w}_1^T \mathbf{x} + b_1) & \sigma(\mathbf{w}_2^T \mathbf{x} + b_2) \vdots \sigma(\mathbf{w}_L^T \mathbf{x} + b_L) \end{bmatrix} \end{aligned} \tag{7.1}$$

where σ is the sigmoid function, $\mathbf{x}$ is the input vector, $\mathbf{w}_i$ and b_i are the weight vector and bias term for the i-th label, L is the number of labels. An example of multilabel network is shown in Figure 7.1.

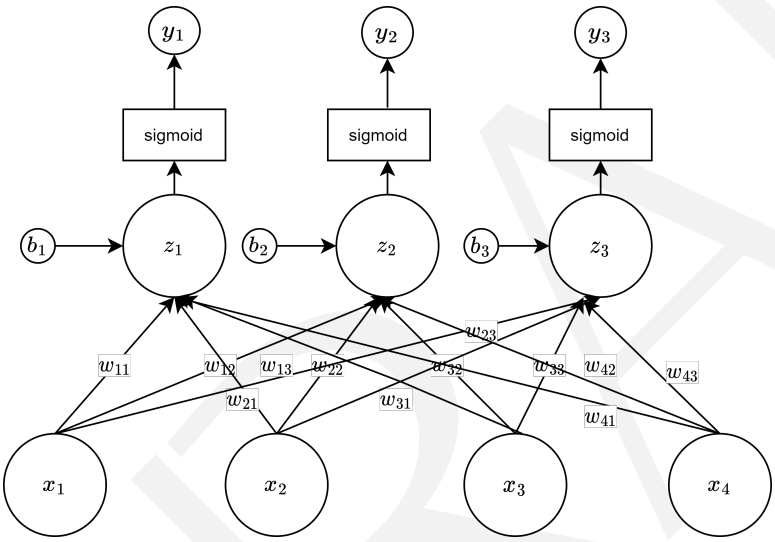


Figure 7.1: Multilabel classification network that has an input with 4 nodes or variables and the output has 3 nodes or classes.

7.2 Learning problem

The objective function for multilabel classification should be the summation of the binary cross-entropy losses for each label. Binary cross-entropy is a special case of categorical cross-entropy where the target is 0 or 1. It measures how well a model predicts a binary outcome. The binary cross-entropy loss for a single label can be written as:

$$E(t, y) = -t \log(y) - (1 - t) \log(1 - y) \tag{7.2}$$

where t is the true label (0 or 1) and y is the predicted probability (between 0 and 1).
The objective function for multilabel classification can be obtained by summing the binary cross-entropy losses for each label:

$$E(\mathbf{t}, \mathbf{y}) = -\sum_{i=1}^{L} [t_i \log(y_i) + (1 - t_i) \log(1 - y_i)] \tag{7.3}$$

where L is the number of labels, $\mathbf{t}$ is the true multi-hot vector, and $\mathbf{y}$ is the predicted probability vector.
This objective function can be used to train a multilabel classifier by minimizing it with respect to the model parameters (see Chapter 5 for details about gradient computation).
A summary of the learning problems discussed so far is shown in the following table:

Model	linear regression	binary classification	multiclass classification	multilabel classification
activation	linear	sigmoid	softmax	sigmoid
# of output nodes	$K \geq 1$	K=1	$K \geq 3$	$K \geq 2$
objective function	mean square error	binary cross entropy	categorical cross entropy	K binary cross entropy

Table 7.1: A summary for the learning algorithms.

7.3 Evaluation

Micro F1-score is an evaluation metric for multilabel classification that provides a harmonic mean of precision and recall across all labels, focusing on the aggregate performance rather than individual label performance. In multilabel settings, we compute this by treating each true positive, false positive, and false negative across

all classes as one combined group, which helps to account for class imbalance and gives an overall sense of model performance.
Let us define the following:

- True Positives (TP): The number of times a label is correctly predicted as present in the instance.
- False Positives (FP): The number of times a label is incorrectly predicted as present.
- False Negatives (FN): The number of times a label is incorrectly predicted as absent.

For a set of labels $\{1, 2, \ldots, L\}$ across N samples, we calculate micro precision, recall, and F1 as follows:
The micro precision P_{micro} is defined as:

$$P_{\text{micro}} = \frac{\sum_{l=1}^{L} \text{TP}_l}{\sum_{l=1}^{L} (\text{TP}_l + \text{FP}_l)}, \tag{7.4}$$

where $\sum_{l=1}^{L} \text{TP}_l$ represents the total true positives across all labels, and $\sum_{l=1}^{L} (\text{TP}_l + \text{FP}_l)$ represents the total predicted positives.
Similarly, the micro recall R_{micro} is:

$$R_{\text{micro}} = \frac{\sum_{l=1}^{L} \text{TP}_l}{\sum_{l=1}^{L} (\text{TP}_l + \text{FN}_l)}, \tag{7.5}$$

where $\sum_{l=1}^{L} (\text{TP}_l + \text{FN}_l)$ is the total actual positives.
The micro F1 score is the harmonic mean of P_{micro} and R_{micro}:

$$F1_{\text{micro}} = \frac{2 \cdot P_{\text{micro}} \cdot R_{\text{micro}}}{P_{\text{micro}} + R_{\text{micro}}} \tag{7.6}$$

Since micro precision and micro recall are computed over all instances and labels together, this score emphasizes the classifier's global effectiveness rather than how it performs on individual labels.
Consider a scenario with three labels $L = 3$ across four instances $N = 4$. We summarize the true positives, false positives, and false negatives in a table:

Label	TP	FP	FN
1	10	5	3
2	20	4	6
3	15	3	7

1. Calculate Micro Precision:

$$P_{\text{micro}} = \frac{10 + 20 + 15}{(10 + 20 + 15) + (5 + 4 + 3)} = \frac{45}{45 + 12} = \frac{45}{57} \approx 0.789$$

2. Calculate Micro Recall:

$$R_{\text{micro}} = \frac{10 + 20 + 15}{(10 + 20 + 15) + (3 + 6 + 7)} = \frac{45}{45 + 16} = \frac{45}{61} \approx 0.738$$

3. Calculate Micro F1 Score:

$$F1_{\text{micro}} = \frac{2 \cdot 0.789 \cdot 0.738}{0.789 + 0.738} \approx \frac{2 \cdot 0.582}{1.527} \approx 0.762$$

Micro F1 gives each true/false positive and negative an equal contribution, which is particularly useful when labels have different frequencies. Hence, Micro F1 is especially helpful when the goal is to evaluate how well a model performs across all predictions, rather than for each individual label, in a multilabel setting.

7.3.1 Decision Boundary Threshold Tuning

For binary classifiers in a multilabel setup, the default decision boundary (typically 0.5) might not optimize Micro F1 performance for each label. Threshold tuning allows adjusting this decision boundary for each label, maximizing specific performance metrics like Micro F1 score. By tuning thresholds, we adapt the sensitivity of the model for each label based on the data distribution and the specific needs of the application.

For a binary classifier outputting scores s_l for label l, the predicted label is defined as:

$$\hat{y}_l = \begin{cases} 1 & \text{if } s_l \geq \tau_l, \\ 0 & \text{if } s_l < \tau_l, \end{cases} \tag{7.7}$$

where τ_l is the decision threshold for label l, which can be tuned based on the goal of maximizing F1, precision, or recall for that specific label. Tuning these thresholds allows balancing between false positives and false negatives across labels, significantly impacting overall F1 performance.

For a set of thresholds $\{\tau_1, \tau_2, \ldots, \tau_L\}$ applied across all L labels, tuning can be performed using methods like grid search, where multiple threshold values are evaluated to maximize the desired metric (often F1). Instead of assigning a separate threshold for each label, using a single global threshold could serve as an alternative.

7.3.2 Macro F1-Score

An alternative metric to consider is Macro F1-Score which offers a complementary approach by averaging the F1 scores computed for each label individually. In a multilabel setup, this approach treats each label equally, irrespective of its frequency, which can provide a balanced view of model performance across all labels.
Given individual precision P_l and recall R_l for each label l, the F1-score for each label is:

$$F1_l = \frac{2 \cdot P_l \cdot R_l}{P_l + R_l} \tag{7.8}$$

The Macro F1-Score, $F1_{\text{macro}}$, is the average of these per-label F1 scores:

$$F1_{\text{macro}} = \frac{1}{L} \sum_{l=1}^{L} F1_l, \tag{7.9}$$

where the Macro Precision P_{macro}:

$$P_{\text{macro}} = \frac{1}{L} \sum_{l=1}^{L} P_l = \frac{1}{L} \sum_{l=1}^{L} \frac{\text{TP}_l}{\text{TP}_l + \text{FP}_l}, \tag{7.10}$$

and Macro Recall R_{macro}:

$$R_{\text{macro}} = \frac{1}{L} \sum_{l=1}^{L} R_l = \frac{1}{L} \sum_{l=1}^{L} \frac{\text{TP}_l}{\text{TP}_l + \text{FN}_l} \tag{7.11}$$

Finally, the Micro F1 is heavily influenced by the performance on frequently occurring labels, as it treats all TP, FP, and FN equally across labels. On the other hand, the Macro F1 treats each label independently, providing an average score across labels, beneficial when classes are imbalanced or certain labels are sparse.
Together, these metrics and threshold tuning provide a comprehensive evaluation strategy for multilabel classification, enhancing model generalization and adaptability to specific goals, whether precision-oriented or recall-oriented, within the multilabel context.

7.4 Example

A Python code is provided to illustrate the learning algorithm of the multilabel classifier for three classes:

```
1
2 import numpy as np
3
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class LinearNN:
    def __init__(self, n_inputs, n_outputs):
        self.weights = np.random.rand(n_inputs, n_outputs)
        self.bias = np.random.rand(n_outputs)

    def train(self, X, T, epochs, lr):
        for epoch in range(epochs):
            # forward propagation
            y_pred = self.predict(X)

            # compute gradient
            d_weights = np.dot(X.T, (y_pred - T) *
    sigmoid_derivative(y_pred))
            d_bias = np.sum((y_pred - T) * sigmoid_derivative(y_pred
    ), axis=0)

            # update weights and bias
            self.weights -= lr * d_weights
            self.bias -= lr * d_bias

            if epoch % 100 == 0:
                loss = np.mean(-T*np.log(y_pred) - (1-T)*np.log(1-
    y_pred))
                print(f'Loss at epoch {epoch}: {loss}')

    def predict(self, X):
        return sigmoid(np.dot(X, self.weights) + self.bias)

if __name__ == "__main__":
    X = np.array([[0, 0, 1],
                  [0, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]])
    T = np.array([[0, 0, 1],
                  [1, 0, 0],
                  [0, 1, 0],
                  [1, 1, 1]])

    model = LinearNN(X.shape[1], T.shape[1])
    model.train(X, T, epochs=1000000, lr=0.001)

    print("Model weights:")
    print(model.weights)
    print("Model biases:")
```

```
51      print(model.bias)
52      print("Predictions on training data:")
53      print(np.round(model.predict(X)))
```

Listing 7.1: Python example for mutlilabel classifier. The classifier has three output classes.

7.5 Assignment

Implement a multilabel classification model using the Keras deep learning library and Google colab. To load the dataset, please visit https://www.kaggle.com/datasets/shivanandmn/multilabel-classification-dataset.

8. Deep Neural Networks

I have never claimed that I invented backpropagation. David Rumelhart invented it independently long after people in other fields had invented it. It is true that when we first published we did not know the history so there were previous inventors that we failed to cite. What I have claimed is that I was the person to clearly demonstrate that backpropagation could learn interesting internal representations and that this is what made it popular.

— Geoffrey E. Hinton

8.1 Motivation

Deep neural networks are composed of multiple layers of neurons that can learn complex patterns and features from the input data. However, if each layer of neurons uses a linear activation function, such as g(z) = x, then the output of the network will be just a linear combination of the inputs, regardless of how many layers there are. Assume a network with L layers and the output layer is a regression problem, then the output of the network is given by

$$\begin{aligned} y &= W^L W^{L-1} \dots W^1 x \\ &= Wx \end{aligned} \tag{8.1}$$

This means that the network will not be able to capture any non-linear relationships or interactions among the input variables, and will fail to model complex phenomena that do not follow linearity.

Table 8.1: Two-dimensional XOR gate truth table.

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

To overcome this limitation, we use non-linear activation functions, such as sigmoid, tanh, relu, etc. (see Figure 8.1), that can introduce non-linearity into the network.

$$y = W^L g(W^{L-1} \dots g(W^2 g(W^1 x + b^1) + b^2) + b^{L-1}) + b^L \tag{8.2}$$

Non-linear activation functions can map the input to a different range or domain, such as (0, 1) for sigmoid or (-1, 1) for tanh, and can create non-linear decision boundaries via non-linear combinations of the weights and inputs. Non-linear activation functions can also help the network to avoid saturation or vanishing gradients, which are problems that occur when the derivative of the activation function becomes very small or zero, and prevent the network from learning effectively.
By using non-linear activation functions in deep neural networks, we can enable the network to learn more expressive and powerful representations of the input data, and to approximate any continuous function

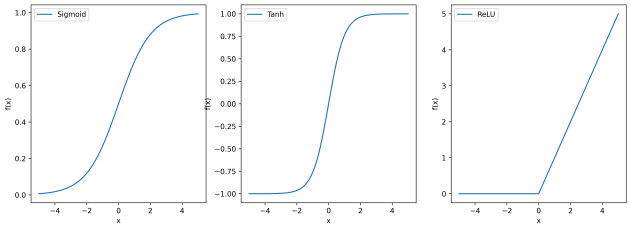


Figure 8.1: Three different activation functions commonly used in deep networks.

One of the functions that can not be solved using linear decision boundaries is the XOR gate that is used in design logic (see Table 8.1). If there is a linear decision boundary can solve the XOR problem, then its equation should be given by

$$x_1 w_1 + x_2 w_2 \lesseqgtr_0^1 \theta \tag{8.3}$$

where θ is a threshold. Since the points $(1, 0)$ and $(0, 1)$ lie on the same side of decision boundary, then

$$\begin{aligned} w_1 &> \theta \\ w_2 &> \theta \end{aligned} \tag{8.4}$$

on the other hand, the point (1, 1) lies on the other side of decision boundary:

$$w_1 + w_2 < \theta \tag{8.5}$$

which is a contradiction of Equation (8.4). Hence, such a linear decision boundary does not exist. This means that there is no single line that can separate the four points that represent the XOR table into two classes. Hence, the XOR problem can not be solved using linear networks. To solve the XOR problem, a non-linear neural network with at least one hidden layer is needed.
The output layer is the final layer in the network where the desired predictions are obtained. Depending on the type and goal of the task, the output layer can have different activation functions and loss functions.
For example, if the task is a regression problem, where the output is a continuous value, such as predicting the price of a house, then the output layer can have a linear activation function, such as g(x) = x, and a mean squared error loss function, which measures the difference between the predicted and actual values.
If the task is a binary classification problem, where the output is either 0 or 1, such as predicting whether an email is spam or not, then the output layer can have a sigmoid activation function, such as $g(x) = 1/(1 + exp(-x))$, and a binary cross-entropy loss function, which measures the probability of the correct class.
If the task is a multi-class classification problem, where the output is one of several possible classes, such as predicting the type of animal in an image, then the output layer can have a softmax activation function, such as $g(x) = \exp(x)/\sum(\exp(x))$, and a categorical cross-entropy loss function, which measures the probability of the correct class. An example of a deep neural network (DNN) is shown in Figure 8.2.
By having a flexible output layer, deep neural networks can adapt to different types of tasks and produce accurate and meaningful predictions.

8.2 Model

Consider a neural network model with two hidden layers and an output layer. The activation function for all layers, including the output layer, is the sigmoid function, ideal for a multi-class classification problem with K classes.
Let's denote:

- $\boldsymbol{x}$: Input data vector.
- $W^{[1]}, W^{[2]}, W^{[3]}$: Weight matrices for Hidden Layer 1, Hidden Layer 2, and Output Layer, respectively.
- $b^{[1]}, b^{[2]}, b^{[3]}$: Bias vectors for the corresponding layers.
- $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$: Linear transformation for each layer l.

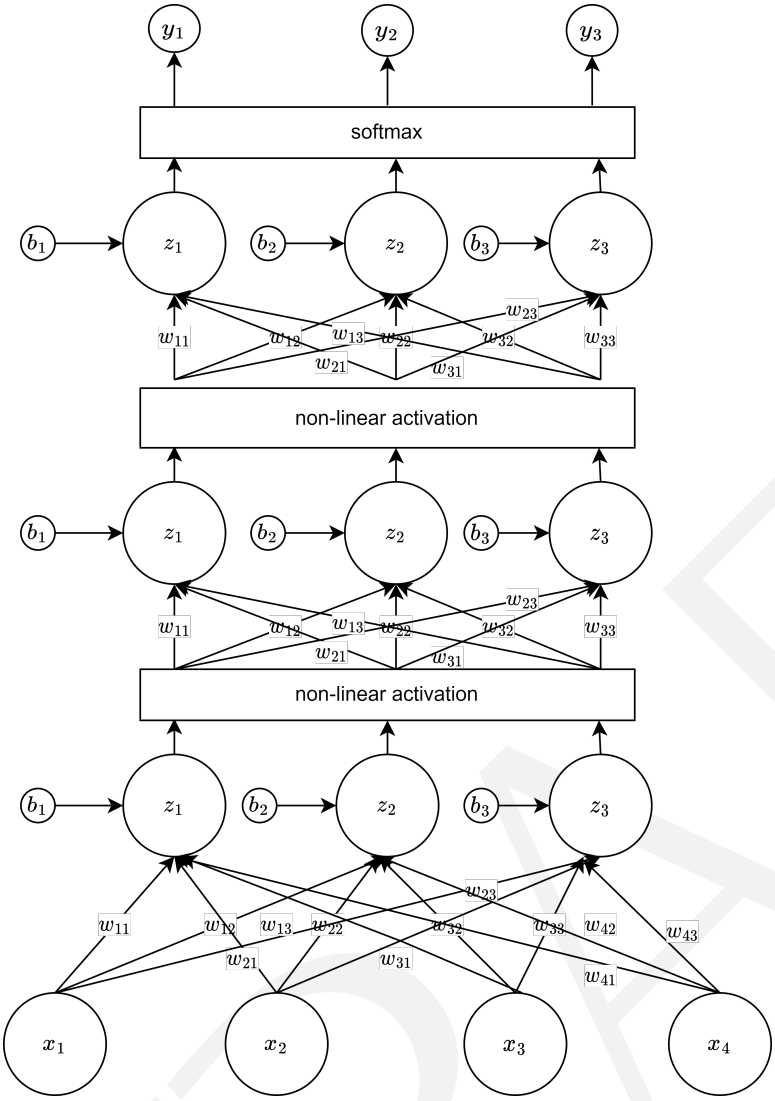


Figure 8.2: A deep neural network consists of 2 hidden layers (each layer has 3 nodes) with non-linear activation such as sigmoid function. The output layer has 3 nodes and a softmax activation to support a multi-class classification problem. The input layer has 4 nodes. The parameters – Please note the index of weight matrices and biases were removed in the figure for simplicity.– are the weights $W^{[3]}$, $W^{[2]}$, $W^{[1]}$ and the biases $b^{[3]}$, $b^{[2]}$, $b^{[1]}$.

- $a^{[l]} = \text{sigmoid}(z^{[l]})$: Sigmoid activation for each hidden layer.
- $a^{[L]} = \text{softmax}(z^{[l]})$: Softmax activation for the output layer.

8.3 Learning via Backpropagation

The backpropagation algorithm is fundamental for training artificial neural networks, and particularly deep learning networks. It is responsible for optimizing the weights in the network by computing gradients, allowing the network to learn from the error made during prediction.
The algorithm is based on a simple principle: it feeds input data forward through the network, computes the error by comparing the predicted and actual outputs, and then propagates this error backward through the network, adjusting the weights along the way. This iterative process is performed over multiple epochs and effectively trains the network.

8.3.1 Forward Propagation

First, we perform forward propagation to compute the activations for each layer:
1. Hidden Layer 1:

$$z^{[1]} = W^{[1]}\boldsymbol{x} + b^{[1]}$$
$$a^{[1]} = \text{sigmoid}(z^{[1]})$$

2. Hidden Layer 2:

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \text{sigmoid}(z^{[2]})$$

3. Output Layer:

$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$$
$$a^{[3]} = \text{softmax}(z^{[3]})$$

8.3.2 Backward Propagation

Next, we compute the gradients and propagate them back through the network:
The output layer uses the softmax activation function. We can write the softmax activation function as:

$$a_i^{[3]} = \frac{e^{z^{[3]}i}}{\sum_{j=1}^{K} e^{z_j^{[3]}}} \tag{8.6}$$

Where the denominator sums over all K classes.
The cost function is cross-entropy loss, which we can write as:

$$E = -\sum_{i=1}^{K} t_i \log a_i^{[3]} \tag{8.7}$$

Where the sum goes over all K classes.
Based on equation (6.15), the gradients are give by:

$$\frac{\partial E}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial E}{\partial a} \tag{8.8}$$

.

$$\frac{\partial E}{\partial a} = \begin{pmatrix} -t_1/a_1 \\ \vdots \\ -t_n/a_n \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ -1/a_j \\ \vdots \\ 0 \end{pmatrix} \tag{8.9}$$

where j is the index of the correct class. In addition, $\frac{\partial \mathbf{a}}{\partial \mathbf{z}}$ is given by

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \begin{pmatrix} a_1 & 0 & 0 & \cdots & 0 \\ 0 & a_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_K \end{pmatrix} - \begin{pmatrix} a_1a_1 & a_1a_2 & a_1a_3 & \cdots & a_1a_K \\ a_2a_1 & a_2a_2 & a_2a_3 & \cdots & a_2a_K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_Ka_1 & a_Ka_2 & a_Ka_3 & \cdots & a_Ka_K \end{pmatrix} \tag{8.10}$$

Hence,

$$\frac{\partial E}{\partial z} = \frac{1}{a_j}[\begin{pmatrix} a_1a_j \\ \vdots \\ a_ja_j \\ \vdots \\ a_Ka_j \end{pmatrix} - \begin{pmatrix} 0 \\ \vdots \\ a_j \\ \vdots \\ 0 \end{pmatrix}] = a - t \tag{8.11}$$

1. Output Layer:

$$\delta^{[3]} = a^{[3]} - t \tag{8.12}$$

$$\frac{\partial E}{\partial W^{[3]}} = \delta^{[3]}(a^{[2]})^T \tag{8.13}$$

$$\frac{\partial E}{\partial b^{[3]}} = \delta^{[3]} \tag{8.14}$$

2. Hidden Layer 2:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ a^{[2]} \circ (1 - a^{[2]}) \tag{8.15}$$

$$\frac{\partial E}{\partial W^{[2]}} = \delta^{[2]} (a^{[1]})^T \tag{8.16}$$

$$\frac{\partial E}{\partial b^{[2]}} = \delta^{[2]} \tag{8.17}$$

3. Hidden Layer 1:

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \circ a^{[1]} \circ (1 - a^{[1]}) \tag{8.18}$$

$$\frac{\partial E}{\partial W^{[1]}} = \delta^{[1]} \boldsymbol{x}^T \tag{8.19}$$

$$\frac{\partial E}{\partial b^{[1]}} = \delta^{[1]} \tag{8.20}$$

Here, $\circ$ denotes element-wise multiplication, which is also known as the Hadamard product. Note that these updates are performed after each iteration or epoch.
Finally, the weights and biases are updated using gradient descent:

$$W^{[l]} = W^{[l]} - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E^n}{\partial W^{[l]}}, \tag{8.21}$$

$$b^{[l]} = b^{[l]} - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E^n}{\partial b^{[l]}}, \tag{8.22}$$

where η is the learning rate.
This process is repeated for multiple epochs until the network is adequately trained. The delta term $\delta^{[l]}$ represents the error or gradient of the loss function with respect to the pre-activation value $z^{[l]}$ of layer l, i.e.,

$$\delta^{[l]} = \frac{\partial E}{\partial z^{[l]}} \tag{8.23}$$

The exact computation of δ differs based on the layer. For the output layer (layer 3):
We use the chain rule of calculus,

$$\delta^{[3]} = \frac{\partial E}{\partial z^{[3]}} = \frac{\partial E}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \tag{8.24}$$

$$\delta^{[3]} = a^{[3]} - t$$

For the hidden layer 2, the delta term is computed using the delta term of the next layer (output layer) and the weight matrix that connects them. This is because the loss function depends on $z^{[2]}$ indirectly through $z^{[3]}$. The derivative of the loss function with respect to $z^{[2]}$ is:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \tag{8.25}$$

Using the fact that $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$, where $W^{[3]}$ and $b^{[3]}$ are the weight matrix and bias vector of layer 3, respectively, and applying the chain rule, we get:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} \frac{\partial (W^{[3]}a^{[2]} + b^{[3]})}{\partial z^{[2]}} \tag{8.26}$$

The derivative of $(W^{[3]}a^{[2]} + b^{[3]})$ with respect to $z^{[2]}$ is simply $W^{[3]}$, since $a^{[2]}$ depends on $z^{[2]}$ and $b^{[3]}$ does not. Therefore, we can simplify the expression as:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} W^{[3]} \frac{\partial a^{[2]}}{\partial z^{[2]}} \tag{8.27}$$

Using the fact that $\frac{\partial E}{\partial z^{[3]}} = \delta^{[3]}$, we can write this in vector form as:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \frac{\partial a^{[2]}}{\partial z^{[2]}} \tag{8.28}$$

Recall that $a^{[2]} = g(z^{[2]})$, where g is the activation function of layer 2. Hence, we need to multiply $\delta^{[2]}$ by the derivative of g with respect to $z^{[2]}$, which is denoted by $g'(z^{[2]})$. This gives us:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ g'(z^{[2]}) \tag{8.29}$$

where $\circ$ denotes element-wise multiplication, also known as the Hadamard product. If we assume that g is a sigmoid function, such as $g(z) = \frac{1}{1+e^{-z}}$, then $g'(z) = g(z)(1 - g(z))$, and we can simplify the expression as:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ a^{[2]} \circ (1 - a^{[2]}) \tag{8.30}$$

For the hidden layer 1, the delta term is computed in a similar way as for hidden layer 2, using the delta term of the next layer (hidden layer 2) and the weight matrix that connects them.
The back-propagation algorithm is illustrated in Figure 8.3.

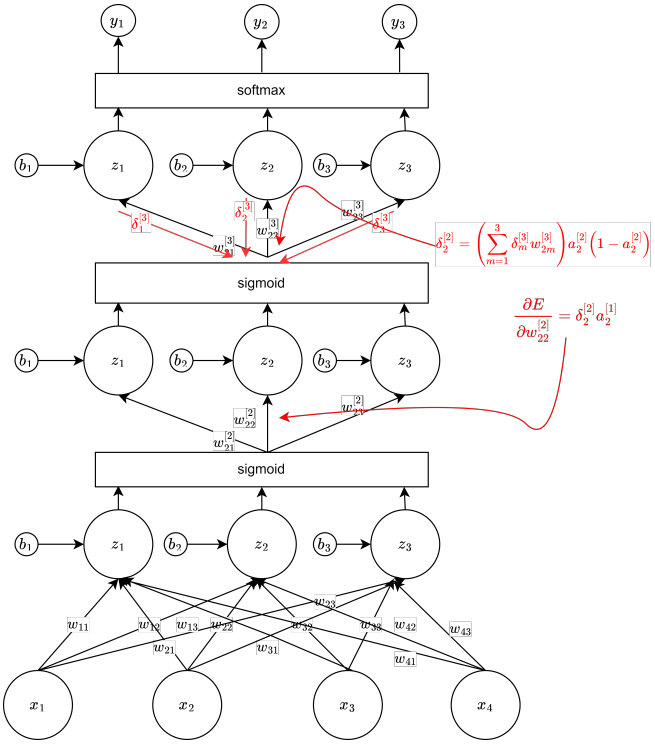


Figure 8.3: The back-propagation algorithm.

8.4 Example

A Python code is provided to illustrate the learning algorithm of the XOR problem and plot the decision boundary between the two classes:

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def sigmoid(x):
6      return 1 / (1 + np.exp(-x))
7
8  def sigmoid_derivative(x):
9      return x * (1 - x)
10
11 # XOR input
12 X = np.array([[0,0], [0,1], [1,0], [1,1]])
13 # XOR output
14 Y = np.array([[0],[1],[1],[0]])
15
16 np.random.seed(0)
```

```
17
18 # initialize weights and biases randomly with mean 0
19 w0 = 2*np.random.random((2,4)) - 1
20 b0 = 2*np.random.random((1,4)) - 1
21 w1 = 2*np.random.random((4,1)) - 1
22 b1 = 2*np.random.random((1,1)) - 1
23
24 # Learning rate
25 lr = 0.1
26 # Number of iterations
27 epochs = 10000
28
29 for epoch in range(epochs):
30     # Forward propagation
31     l0 = X
32     l1 = sigmoid(np.dot(l0, w0) + b0)
33     l2 = sigmoid(np.dot(l1, w1) + b1)
34
35     # Backward propagation
36     l2_error = Y - l2
37     l2_delta = l2_error * sigmoid_derivative(l2)
38
39     l1_error = l2_delta.dot(w1.T)
40     l1_delta = l1_error * sigmoid_derivative(l1)
41
42     # Update weights and biases
43     w1 += l1.T.dot(l2_delta) * lr
44     b1 += np.sum(l2_delta, axis=0, keepdims=True) * lr
45     w0 += l0.T.dot(l1_delta) * lr
46     b0 += np.sum(l1_delta, axis=0, keepdims=True) * lr
47
48 # Plot decision boundary
49 h = 0.01
50 x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
51 y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
52 xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
       y_max, h))
53
54 l0 = np.c_[xx.ravel(), yy.ravel()]
55 l1 = sigmoid(np.dot(l0, w0) + b0)
56 l2 = sigmoid(np.dot(l1, w1) + b1)
57
58 # threshold the output
59 Z = (l2 > 0.5).astype(int)
60
61 Z = Z.reshape(xx.shape)
62 plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
63 plt.scatter(X[:, 0], X[:, 1], c=Y[:, 0], cmap=plt.cm.Spectral)
64
```

```
65 plt.savefig('xor_decision_boundary.png', dpi=600)
```

Listing 8.1: Python example for plotting the XOR decision boundary.

In Figure 8.4, the 2D plot for the XOR problem results in a decision boundary that separates the space into four regions. Depending on how the network learns, the shape of the boundary may vary, but it will always be able to separate [0, 0] and [1, 1] from [0, 1] and [1, 0].

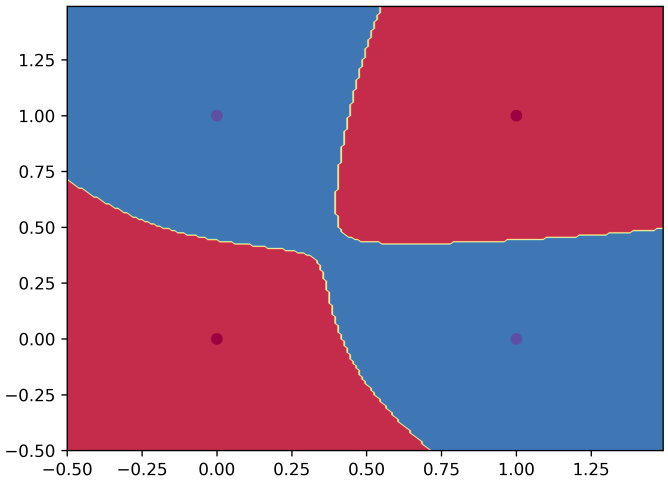


Figure 8.4: The decision boundaries between two classes for the two-dimensional data XOR problem.

8.5 Assignment

Implement an optical recognition of handwritten digits (multiclass classification task) using deep neural networks. The Keras deep learning library and Google colab can be used for this task. To load the dataset, please visit https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits.

9. Convolutional Networks

Deep Learning (DL) is far, far more than old-style neural nets with more than a couple of layers. DL is an "architectural language" with enormous flexibility. The bestiary of DL architectural elements has diversified tremendously over the last decade.

— Yann LeCun

Convolutional Networks or Convolutional Neural Networks (CNNs) are a class of deep learning models designed for processing and analyzing structured grid data, such as images and videos. CNNs have proven to be highly effective in various computer vision tasks due to their ability to automatically learn hierarchical representations of features directly from the data. In this chapter, we will introduce CNNs and address their usage to extract salient features automatically from images without using feature engineering approaches.

9.1 Motivation

The shift from Feed-Forward Networks (FFNs) to Convolutional Neural Networks (CNNs) in the context of image processing is driven by several key advantages that CNNs offer over FFNs. Here's an overview of the motivations and benefits:

1. Handling high dimensionality: FFNs treat each pixel in an image as a separate input feature, resulting in a very high-dimensional input space. For instance, a small image of 64x64 pixels with three color channels (RGB) has 12,288 input features. This high dimensionality leads to a large number of parameters, making the network computationally expensive and prone to overfitting. CNNs utilize local receptive fields (filters) that process small regions of the image at a time. This significantly reduces the number of parameters as the same

filter is applied across different regions of the image. By focusing on local patterns and reusing parameters, CNNs handle high-dimensional data more efficiently.

2. Parameter sharing: FFNs use separate parameters for each input feature, leading to a large number of unique parameters. This lack of parameter sharing results in high computational and memory requirements. CNNs employ the concept of parameter sharing, where the same filter is used across different parts of the image. This reduces the number of unique parameters and enhances computational efficiency. Parameter sharing also allows CNNs to detect features regardless of their position in the image, contributing to translation invariance.

3. Translation invariance: FFNs do not have an inherent mechanism to recognize patterns irrespective of their spatial location in the image. This means that the same object in different positions would be treated as different inputs, requiring more training data to learn positional variations. The shared filters in CNNs enable the network to recognize features regardless of their location in the image. This translation invariance allows CNNs to generalize better and recognize objects in varying positions.

4. Hierarchical feature learning: FFNs treat all input features equally and do not inherently capture hierarchical patterns in the data. This makes it challenging to learn complex structures and relationships within the image. CNNs learn hierarchical features, where early layers capture low-level features (e.g., edges, textures) and deeper layers capture high-level features (e.g., shapes, objects). This hierarchical learning mimics the human visual system and enables CNNs to build more abstract and meaningful representations of the input data.

5. Dimensionality reduction through pooling: FFNs do not have a built-in mechanism for reducing the dimensionality of the input data, often leading to an overwhelming number of parameters. CNNs use pooling layers (e.g., max pooling, average pooling) to downsample the feature maps, reducing their dimensionality while retaining important information. - Pooling layers help in reducing computational complexity and making the network more robust to spatial variations and noise.

CNNs have achieved state-of-the-art performance in various image recognition benchmarks and competitions (e.g., ImageNet). They require less manual feature engineering as they can learn features directly from raw image data, leading to more accurate and efficient models. In the upcoming sections, we will present an

overview of the convolution operation, the pooling operation, and the learning problem associated with convolutional neural networks (CNNs).

9.2 Convolutions

9.2.1 Definition

In one dimension, convolution is a mathematical operation that combines two functions to produce a third function. The convolution of two functions $x(t)$ and $w(t)$ is denoted as $(x * w)(t)$ and is defined by the integral:

$$(x * w)(t) = \int_{-\infty}^{\infty} x(\tau) \cdot w(t - \tau)\, d\tau \tag{9.1}$$

Alternatively, for discrete signals, the convolution of two sequences $x[n]$ and $w[n]$ is defined as:

$$(x * w)[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot w[n - k] \tag{9.2}$$

In simpler terms, for each point t or n, you multiply the values of the two functions at different points and sum up these products over all possible points. This process is repeated for each point in the range of t or n to obtain the resulting convolution function or sequence.
Convolution is a fundamental operation in signal processing and plays a crucial role in various fields, including image processing, audio processing, and, more recently, in the context of convolutional neural networks (CNNs) in deep learning. In the context of CNNs, the convolution operation is applied to learn features from input data, and the kernels (filters) used in the convolution are adaptively adjusted during the training process. The convolution operation for two-dimensional data (e.g. images) is defined as follows:

$$s_{ij} = (x * w)[i, j] = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} x_{k,l} \cdot w_{i-k,j-l} \tag{9.3}$$

In this equation, x represents the input, s represents the output, w represents the kernel, and m and n are the height and width of the kernel, respectively. Please note that the convolution operation is commutative:

$$s_{ij} = (w * x)[i, j] = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} x_{i-k,j-l} \cdot w_{k,l} \tag{9.4}$$

So, in practice, when working with convolutional layers in deep learning frameworks, you would use the term "convolution," but the operation performed is a cross-correlation without flipping the kernel. The equation for this operation would be:

$$s_{ij} = (x \otimes w)[i, j] = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} x_{i+k,j+l} \cdot w_{k,l} \tag{9.5}$$

The cross-correlation operation is computed by summing the element-wise product of the input and the kernel as it slides over the input. The cross-correlation operation simplifies the implementation and doesn't affect the network's ability to learn and generalize from the data. The choice between convolution and cross-correlation does not affect a neural network's ability to learn from data because the convolutional filters (kernels) themselves are learnable parameters, and the network adjusts them during the training process.

9.2.2 Examples of the 1D cross-correlation operations

Cross-correlation is a measure of similarity between two sequences as a function of the displacement of one relative to the other. In the context of 1D input data and a filter, cross-correlation can be used to find how much one sequence (the filter) matches with a portion of another sequence (the input data) as the filter slides over the input.

Let's walk through a simple example of 1D cross-correlation with a vector input of 5 elements and a kernel (filter) of 3 elements in Figure 9.1.[1] We perform the cross-correlation operation step by step. In each position, we calculate the sum of element-wise multiplication between the input and the kernel at that position. The resulting vector is the cross-correlation output.

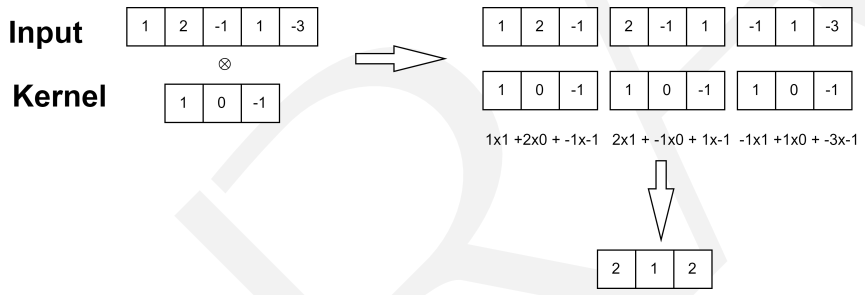


Figure 9.1: An example of 1D cross-correlation output [24].

Padding refers to the process of adding additional values around the borders of the input feature map or image. Padding is an important concept in CNNs as it helps to

[1] https://cs231n.github.io/convolutional-networks/

preserve the spatial dimensions of the feature maps during the convolution operation. If you have a specific padding strategy, the results may vary. For example, to ensure that the output vector size equals the input vector size, we need to zero-padding the input vector from the left and right (i.e. for positions where the kernel extends beyond the input boundaries) In Figure 9.2, we add "0" to the left and right of the input. Hence, the cross-correlation output will maintain the same size as the input.

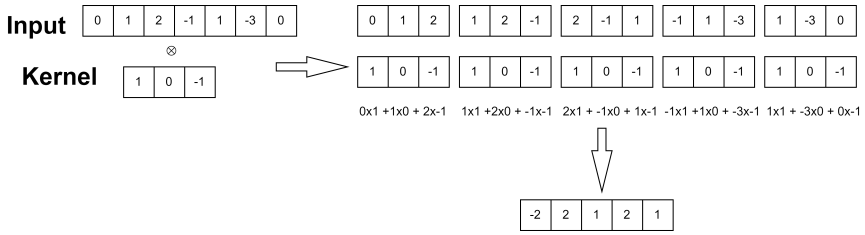


Figure 9.2: An example of 1D cross-correlation output where the zero-padding is implemented.

The stride is a hyperparameter that determines the number of elements by which the convolution kernel slides over the input during the cross-correlation operation. Specifically, the stride defines the step size of the kernel as it moves across the input. A stride of "1" means the kernel moves one element at a time, a stride of "2" means the kernel moves two elements at a time, and so on. The stride has a significant impact on the size of the output. We show an example where the amount of padding added to the input is "1" element and the stride which is the number of elements the kernel moves at each step is "2" in Figure 9.3.

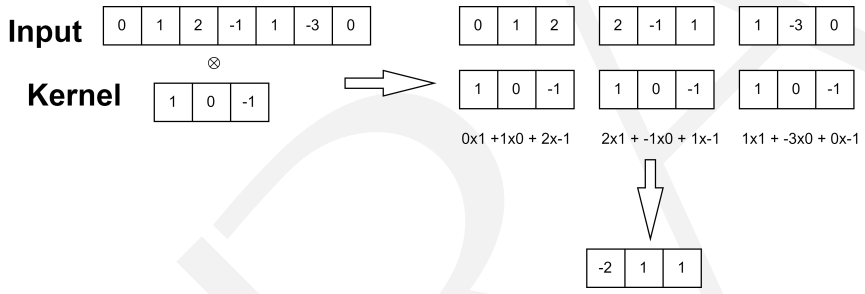


Figure 9.3: An example of 1D cross-correlation output where the zero-padding = 1 and the stride = 2.

In general, we can compute the size of the output vector, o, as a function of the input vector size n, the receptive field size of the kernel f, the stride with which they are

applied s, and the amount of zero padding used p on the border as follows:

$$o = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1, \tag{9.6}$$

where $\lfloor \cdot \rfloor$ denotes the floor function. The floor function is used to ensure that the output size is an integer, representing the discrete grid positions where the filter is applied.

9.2.3 Examples of the 2D cross-correlation operations

In the context of image processing and computer vision, 2D cross-correlation is a fundamental operation used to measure the similarity between two 2D signals, such as images. The 2D cross-correlation is defined in Equation (9.5). The key aspects of 2D cross-correlation are:

- Sliding window: The kernel is slid over the entire input image, with the center of the kernel positioned at each element in the image.
- Element-wise multiplication: At each position, the values of the kernel are multiplied element-wise with the corresponding values in the input image.
- Summation: The results of the element-wise multiplications are summed up to produce the final cross-correlation value at the position.

A simple example of 2D cross-correlation with an input matrix of 6×6 elements and a kernel (filter) matrix of 3×3 elements in Figure 9.4. When the stride parameter is 2, the output is given in Figure 9.5. In general, we can compute the size of the output matrix as a function of the input matrix size $n \times n$, the receptive field size of the kernel $f \times f$, the stride with which they are applied s, and the amount of zero padding used p on the border as follows:

$$o_1 x o_2 = [\left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1] \times [\left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1] \tag{9.7}$$

.

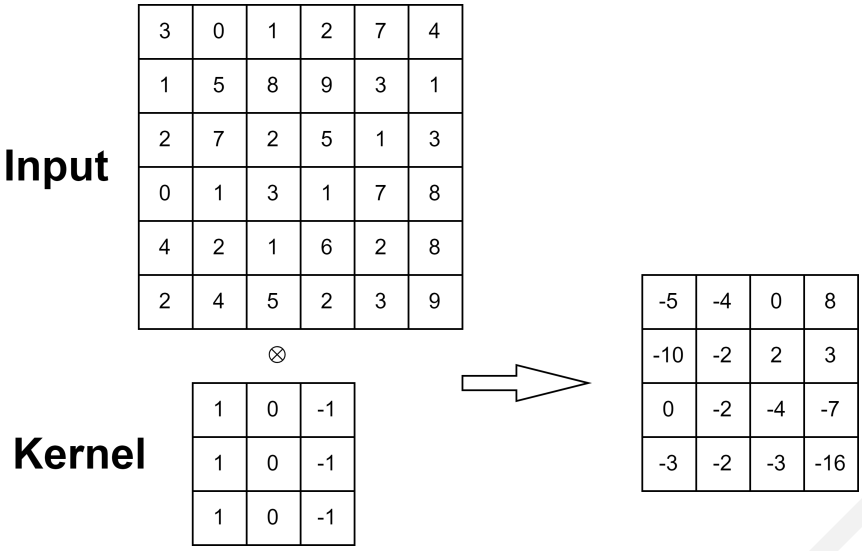


Figure 9.4: An example of 2D cross-correlation where the input is a matrix of 6x6 elements and the kernel is a matrix of 3x3 elements [25].

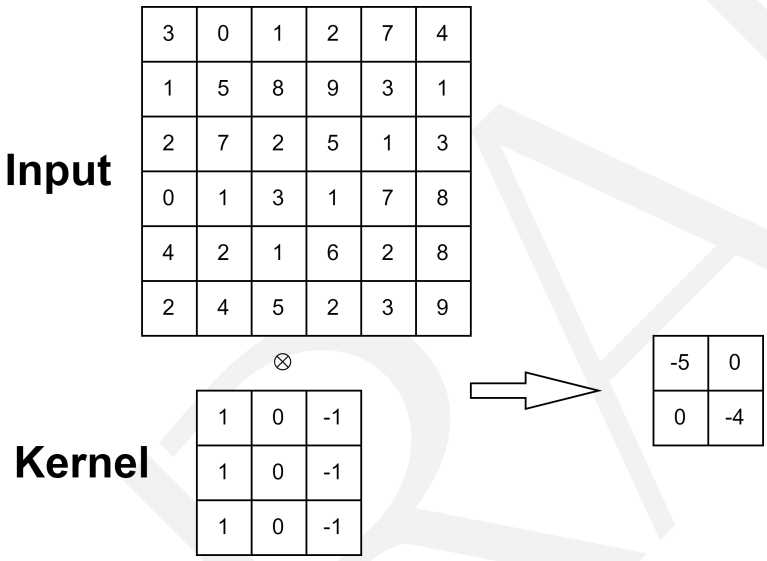


Figure 9.5: An example of 2D cross-correlation where the input isFor example, the height of the input equals to the height of the kernel a matrix of 6×6 elements, the kernel is a matrix of 3×3 elements, and the stride of 2.

In the case where the height of the 2D kernel is equal to the height of the 2D

input data, in this special case, the 2D cross-correlation operation can be reduced to a 1D cross-correlation along the width dimension. For example, the height of the 2D input matrix equals the height of the kernel (both are 3) in Figure 9.6. Hence, the 2D cross-correlation operation reduces to 1D cross-correlation along the width dimension.

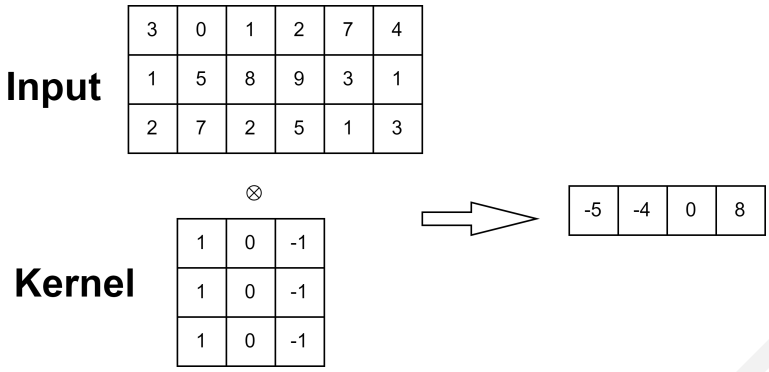


Figure 9.6: The 2D cross-correlation operation can be reduced to a 1D cross-correlation along the width dimension, where the height of the 2D kernel is equal to the height of the 2D input data.

9.2.4 Examples of the 3D cross-correlation operations

Imagine we have a 3D image with dimensions $6 \times 6 \times 3$ instead of a 2D image. How would we apply convolution to this image? We would use a $3 \times 3 \times 3$ filter instead of a 3×3 filter. Let's see an example:
Input: $6 \times 6 \times 3$
Filter: $3 \times 3 \times 3$
The dimensions represent the height, width, and channels of the input and filter. Note that the number of channels in the input and filter must be the same. This will result in an output of 4×4. Let's visualize this example in Figure 9.7:
Since the input has three channels, the filter will also have three channels. After convolution, the output will be a 4 X 4 matrix. The first element of the output is the sum of the element-wise product of the first 27 values from the input (9 values from each channel) and the 27 values from the filter.

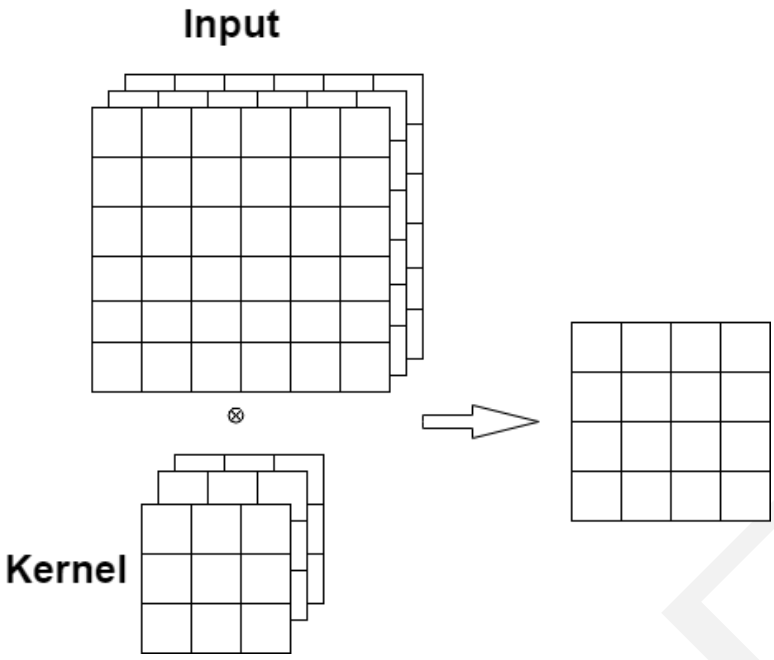


Figure 9.7: An example of cross-correlation over a volume [25].

Instead of using only one filter, we can apply multiple filters simultaneously. For instance, the first filter might detect vertical edges, while the second filter identifies horizontal edges in the image. Using multiple filters alters the output dimensions. Therefore, instead of a 4×4 output as in the previous example, we would obtain a $4 \times 4 \times 2$ output if two filters were used as shown in Figure 9.8.

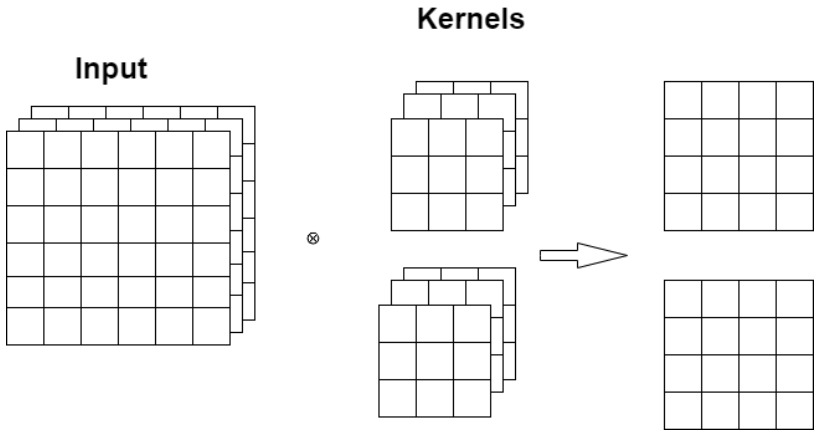


Figure 9.8: An example of cross-correlations over volume using multiple filters [25].

9.3 Resolution control (subsampling)

Max pooling and average pooling are two common down-sampling techniques used in Convolutional Neural Networks (CNNs) to reduce the spatial dimensions of the input data. They both help in reducing the number of parameters and computational cost, and they also provide some degree of translation invariance.
Max pooling selects the maximum value from a defined region (pooling window) of the input data. This operation highlights the most prominent features in that region. A fixed-size window (e.g., 2x2 or 3x3) slides over the input feature map. For each window position, the maximum value within that window is selected and assigned to the corresponding position in the output feature map. The window slides according to a specified stride, typically 2, meaning the window moves by 2 elements at a time. An example of the max pooling operation is shown in Figure 9.9. The output size of a max pooling operation for 2D input data can be determined using the following equation:

$$o = \left\lfloor \frac{n - f + 2p}{s} \right\rfloor + 1 \tag{9.8}$$

where o is the size of the output, n is the size of the input, f is the size of the filter (pooling window), p is the amount of padding applied, s is the stride of the pooling operation, and $\lfloor \cdot \rfloor$ denotes the floor function, which rounds down to the nearest integer. If the input size n is 4, the filter size f is 2, the padding p is 0, and the stride s is 2, the output size o is calculated as follows:

$$o = \left\lfloor \frac{4 - 2 + 2 \cdot 0}{2} \right\rfloor + 1 = \left\lfloor \frac{2}{2} \right\rfloor + 1 = 1 + 1 = 2 \tag{9.9}$$

Thus, the output size is 2 in each dimension.
The advantages of max pooling are:

- Reduces the dimensionality of the feature map while retaining the most important features.
- Helps to reduce overfitting by providing an abstracted form of the input representation.
- Provides translational invariance to small shifts and distortions in the input.

Similar to max pooling, average pooling calculates the average value from a defined region (pooling window) of the input data. This operation smooths the feature map by averaging the features in each region.

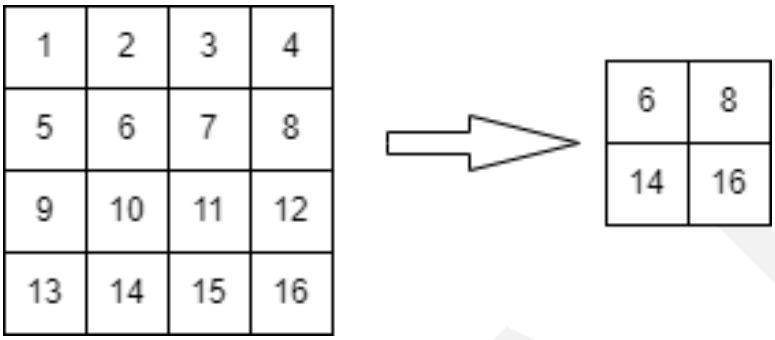


Figure 9.9: For a 4x4 input matrix and a 2x2 max pooling window, the output is a 2x2 matrix.

9.4 Model and learning problem

9.4.1 Convolutional layers

To compute the gradients during backpropagation in the convolutional layer [26], let us assume that the convolutional layer has only one channel. Hence, the equations for computing the gradients during backpropagation simplify slightly.
Let's consider the convolutional layer with a single-channel input a^{l-1} and output z^l. The forward pass is computed as follows:

$$\underbrace{\begin{bmatrix} z^l_{00} & z^l_{01} \\ z^l_{10} & z^l_{11} \end{bmatrix}}_{z^{[l]}} = \underbrace{\begin{bmatrix} a^{[l-1]}_{00} & a^{[l-1]}_{01} & a^{[l-1]}_{02} \\ a^{[l-1]}_{10} & a^{[l-1]}_{11} & a^{[l-1]}_{12} \\ a^{[l-1]}_{20} & a^{[l-1]}_{21} & a^{[l-1]}_{22} \end{bmatrix}}_{a^{[l-1]}} \otimes \underbrace{\begin{bmatrix} w^{[l]}_{00} & w^{[l]}_{01} \\ w^{[l]}_{10} & w^{[l]}_{11} \end{bmatrix}}_{w^{[l]}} + b^{[l]} \tag{9.10}$$

$$
\begin{aligned}
z^l_{00} &= w^{[l]}_{00}a^{[l-1]}_{00} + w^{[l]}_{01}a^{[l-1]}_{01} + w^{[l]}_{10}a^{[l-1]}_{10} + w^{[l]}_{11}a^{[l-1]}_{11} + b^{[l]} \\
z^l_{01} &= w^{[l]}_{00}a^{[l-1]}_{01} + w^{[l]}_{01}a^{[l-1]}_{02} + w^{[l]}_{10}a^{[l-1]}_{11} + w^{[l]}_{11}a^{[l-1]}_{12} + b^{[l]} \\
z^l_{10} &= w^{[l]}_{00}a^{[l-1]}_{10} + w^{[l]}_{01}a^{[l-1]}_{11} + w^{[l]}_{10}a^{[l-1]}_{20} + w^{[l]}_{11}a^{[l-1]}_{21} + b^{[l]} \\
z^l_{11} &= w^{[l]}_{00}a^{[l-1]}_{11} + w^{[l]}_{01}a^{[l-1]}_{12} + w^{[l]}_{10}a^{[l-1]}_{21} + w^{[l]}_{11}a^{[l-1]}_{22} + b^{[l]}
\end{aligned}
\tag{9.11}
$$

The gradient of the loss E with respect to the weight matrix $\mathbf{w}^l$ can be computed as follows:

$$
\begin{aligned}
\frac{\partial E}{\partial w^{[l]}_{00}} &= \frac{\partial E}{\partial z^{[l]}_{00}} \cdot a^{[l-1]}_{00} + \frac{\partial E}{\partial z^{[l]}_{01}} \cdot a^{[l-1]}_{01} + \frac{\partial E}{\partial z^{[l]}_{10}} \cdot a^{[l-1]}_{10} + \frac{\partial E}{\partial z^{[l]}_{11}} \cdot a^{[l-1]}_{11} \\
\frac{\partial E}{\partial w^{[l]}_{01}} &= \frac{\partial E}{\partial z^{[l]}_{00}} \cdot a^{[l-1]}_{01} + \frac{\partial E}{\partial z^{[l]}_{01}} \cdot a^{[l-1]}_{02} + \frac{\partial E}{\partial z^{[l]}_{10}} \cdot a^{[l-1]}_{11} + \frac{\partial E}{\partial z^{[l]}_{11}} \cdot a^{[l-1]}_{12} \\
\frac{\partial E}{\partial w^{[l]}_{10}} &= \frac{\partial E}{\partial z^{[l]}_{00}} \cdot a^{[l-1]}_{10} + \frac{\partial E}{\partial z^{[l]}_{01}} \cdot a^{[l-1]}_{11} + \frac{\partial E}{\partial z^{[l]}_{10}} \cdot a^{[l-1]}_{20} + \frac{\partial E}{\partial z^{[l]}_{11}} \cdot a^{[l-1]}_{21} \\
\frac{\partial E}{\partial w^{[l]}_{11}} &= \frac{\partial E}{\partial z^{[l]}_{00}} \cdot a^{[l-1]}_{11} + \frac{\partial E}{\partial z^{[l]}_{01}} \cdot a^{[l-1]}_{12} + \frac{\partial E}{\partial z^{[l]}_{10}} \cdot a^{[l-1]}_{21} + \frac{\partial E}{\partial z^{[l]}_{11}} \cdot a^{[l-1]}_{22}
\end{aligned}
\tag{9.12}
$$

Assume $z^{[l]}$ is m x n dimensional

$$
\begin{aligned}
\frac{\partial E}{\partial w^{[l]}_{p,q}} &= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} \frac{\partial E}{\partial z^{[l]}_{i,j}} \cdot a^{[l-1]}_{p+i,q+j} \\
&= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a^{[l-1]}_{p+i,q+j} \cdot \frac{\partial E}{\partial z^{[l]}_{i,j}} \\
&= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a^{[l-1]}_{p+i,q+j} \cdot \delta^{[l]}_{i,j}
\end{aligned}
\tag{9.13}
$$

where $\delta^{[l]}_{i,j} = \frac{\partial E}{\partial z^{[l]}_{i,j}}$. It can written as follows:

$$
\underbrace{\begin{bmatrix} \frac{\partial E}{\partial w^{[l]}_{00}} & \frac{\partial E}{\partial w^{[l]}_{01}} \\ \frac{\partial E}{\partial w^{[l]}_{10}} & \frac{\partial E}{\partial w^{[l]}_{11}} \end{bmatrix}}_{\frac{\partial E}{\partial w^{[l]}}} = \underbrace{\begin{bmatrix} a^{[l-1]}_{00} & a^{[l-1]}_{01} & a^{[l-1]}_{02} \\ a^{[l-1]}_{10} & a^{[l-1]}_{11} & a^{[l-1]}_{12} \\ a^{[l-1]}_{20} & a^{[l-1]}_{21} & a^{[l-1]}_{22} \end{bmatrix}}_{a^{[l-1]}} \otimes \underbrace{\begin{bmatrix} \frac{\partial E}{\partial z^{[l]}_{00}} & \frac{\partial E}{\partial z^{[l]}_{01}} \\ \frac{\partial E}{\partial z^{[l]}_{10}} & \frac{\partial E}{\partial z^{[l]}_{11}} \end{bmatrix}}_{\frac{\partial E}{\partial z^{[l]}}}
\tag{9.14}
$$

We observe that the gradient with respect to the weights is a cross-correlation

operation as well. For the bias term:

$$\frac{\partial E}{\partial b^{[l]}} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \delta_{i,j}^{[l]} \tag{9.15}$$

The computation of $\delta_{ij}^{[l-1]}$ is given by:

$$\begin{aligned}
\frac{\partial E}{\partial a_{00}^{[l-1]}} &= \frac{\partial E}{\partial z_{00}^{l}} \cdot w_{00}^{[l]} \\
\frac{\partial E}{\partial a_{01}^{[l-1]}} &= \frac{\partial E}{\partial z_{00}^{l}} \cdot w_{01}^{[l]} + \frac{\partial E}{\partial z_{01}^{l}} \cdot w_{00}^{[l]} \\
\frac{\partial E}{\partial a_{02}^{[l-1]}} &= \frac{\partial E}{\partial z_{01}^{l}} \cdot w_{01}^{[l]} \\
&\vdots \\
\frac{\partial E}{\partial a_{11}^{[l-1]}} &= \frac{\partial E}{\partial z_{00}^{l}} \cdot w_{11}^{[l]} + \frac{\partial E}{\partial z_{01}^{l}} \cdot w_{10}^{[l]} + \frac{\partial E}{\partial z_{10}^{l}} \cdot w_{01}^{[l]} + \frac{\partial E}{\partial z_{11}^{l}} \cdot w_{00}^{[l]} \\
&\vdots \\
\frac{\partial E}{\partial a_{22}^{[l-1]}} &= \frac{\partial E}{\partial z_{11}^{l}} \cdot w_{11}^{[l]}
\end{aligned} \tag{9.16}$$

$$\underbrace{\begin{bmatrix} \frac{\partial E}{\partial a_{00}^{[l-1]}} & \frac{\partial E}{\partial a_{01}^{[l-1]}} & \frac{\partial E}{\partial a_{02}^{[l-1]}} \\ \frac{\partial E}{\partial a_{10}^{[l-1]}} & \frac{\partial E}{\partial a_{11}^{[l-1]}} & \frac{\partial E}{\partial a_{12}^{[l-1]}} \\ \frac{\partial E}{\partial a_{20}^{[l-1]}} & \frac{\partial E}{\partial a_{21}^{[l-1]}} & \frac{\partial E}{\partial a_{22}^{[l-1]}} \end{bmatrix}}_{\frac{\partial E}{\partial a^{[l-1]}}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial E}{\partial z_{00}^{l}} & \frac{\partial E}{\partial z_{01}^{l}} & 0 \\ 0 & \frac{\partial E}{\partial z_{10}^{l}} & \frac{\partial E}{\partial z_{11}^{l}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Padded } \frac{\partial E}{\partial z^{l}}} \otimes \underbrace{\begin{bmatrix} w_{11}^{[l]} & w_{10}^{[l]} \\ w_{01}^{[l]} & w_{00}^{[l]} \end{bmatrix}}_{\text{Rotated } w^{[l]}} \tag{9.17}$$

hence,

$$\delta_{ij}^{[l-1]} = g'(z_{ij}^{[l]}) \odot \left(\text{pad}(\delta^{[l]}) \otimes \text{rot180}(w^{[l]}) \right)[i, j] \tag{9.18}$$

In this equation: $g'(z_{ij}^{[l]})$ represents the derivative of the activation function applied to the output of the convolutional layer at position (i, j) and $w^{[l]}$ represents the weight matrix of the convolutional layer. The $\otimes$ denotes the cross-correlation operation and $\text{pad}(\delta^{[l]})$ represents the padded error term from the next layer.

9.4.2 Pooling layers

In 2D max pooling, the output at position (i, j) is the maximum value within the filter window applied to the input:

$$O_{i,j} = \max_{(m,n)\in\text{filter window}} X_{i\cdot s+m, j\cdot s+n} \tag{9.19}$$

where: - X is the input matrix. - O is the output matrix. - s is the stride. - The filter window is the region of the input matrix over which the filter is applied, typically of size $f \times f$.
In the backward pass, the gradient is propagated to the position in the input matrix that contributed the maximum value to the output:

$$\frac{\partial L}{\partial X_{i\cdot s+m, j\cdot s+n}} = \begin{cases} \frac{\partial L}{\partial O_{i,j}} & \text{if } X_{i\cdot s+m, j\cdot s+n} = O_{i,j} \\ 0 & \text{otherwise} \end{cases} \tag{9.20}$$

where $\frac{\partial L}{\partial O_{i,j}}$ is the gradient of the loss with respect to the output at position (i, j).
In 2D average pooling, the output at position (i, j) is the average value within the filter window applied to the input:

$$O_{i,j} = \frac{1}{f^2} \sum_{(m,n)\in\text{filter window}} X_{i\cdot s+m, j\cdot s+n} \tag{9.21}$$

where: - f is the filter size.
In the backward pass, the gradient is equally distributed among all positions in the filter window:

$$\frac{\partial L}{\partial X_{i\cdot s+m, j\cdot s+n}} = \frac{1}{f^2} \frac{\partial L}{\partial O_{i,j}} \tag{9.22}$$

where $\frac{\partial L}{\partial O_{i,j}}$ is the gradient of the loss with respect to the output at position (i, j).
These equations describe how the gradients are calculated and propagated back through the pooling layers during backpropagation in a convolutional neural network.

9.5 An Example

LeNet-5 is a classic convolutional neural network architecture that was proposed by Yann LeCun et al. in 1998 [27]. It was designed primarily for handwritten digit recognition tasks, such as the MNIST dataset. LeNet-5 played a significant role in popularizing convolutional neural networks and was a pioneering architecture for deep learning.
Here are the key details of the LeNet-5 architecture (i.e. Figure 9.10):

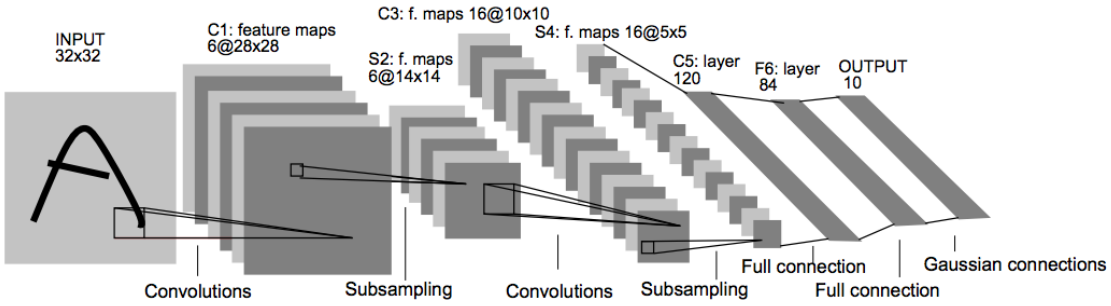


Figure 9.10: LeNet-5 is a simple convolutional neural network for handwritten character recognition [27].

1. The input to LeNet-5 is a grayscale image of size 32x32 pixels.

2. The first layer is a convolutional layer with 6 filters. Each filter has a size of 5x5 and the convolution is followed by a sub-sampling (pooling) operation. The subsampling operation is typically done using average pooling with a window size of 2x2 and a stride of 2. The idea of using subsampling layers is to reduce the spatial dimensions while retaining the important features.

3. The second layer is another convolutional layer with 16 filters. Each filter has a size of 5x5. Similar to the first layer, this layer is also followed by a subsampling operation.

4. After the convolutional layers, the output is flattened into a vector. This flattened vector is then fed into a fully connected layer. LeNet-5 has three fully connected layers with 120, 84, and 10 neurons respectively. The final fully connected layer has 10 neurons, corresponding to the 10 possible classes in the MNIST dataset.

5. LeNet-5 uses the sigmoid activation function in all layers except for the output layer. The output layer typically uses the softmax activation function to produce a probability distribution over the classes.

The LeNet-5 architecture is trained using the backpropagation algorithm with gradient descent optimization. The loss function used is typically the cross-entropy loss. During training, the weights of the network are updated iteratively to minimize the loss and improve the network's performance.
LeNet-5 achieved remarkable performance on handwritten digit recognition tasks and demonstrated the effectiveness of convolutional neural networks for image clas-

sification. It laid the foundation for more advanced CNN architectures that followed [28, 29].

9.6 Assignments

Please implement LeNet-5 network using Keras and Google colab. To load the dataset, please use the method: tf.keras.datasets.mnist.load_data().

10. Recurrent Neural Networks

Recurrent nets are in principle capable to store past inputs to produce the currently desired output. Because of this property recurrent nets are used in time series prediction and process control. Practical applications involve temporal dependencies spanning many time steps, e.g. between relevant inputs and desired outputs. In this case, however, gradient based learning methods take too much time. The extremely increased learning time arises because the error vanishes as it gets propagated back.

— Sepp Hochreiter

Recurrent neural networks, and in particular, Long Short-Term Memory (LSTM) networks, have revolutionized the field of sequence modeling. LSTMs provide a powerful framework for capturing long-range dependencies in sequential data, enabling machines to understand and generate complex sequences with remarkable fluency and precision. In this chapter, we will introduce RNNs and LSTMs and address the difficulty to train these networks.

10.1 Model

A simple RNN is a type of artificial neural network that is designed to process sequential data. It has a recurrent structure that allows it to maintain an internal state (hidden state) that captures information from previous steps in the sequence. Unlike linear dynamical systems, RNNs are nonlinear models and can learn complex

patterns and dependencies in sequential data. The hidden state of an RNN is updated at each time step based on the current input and the previous hidden state, using nonlinear activation functions. RNNs possess a recurrent connection enabling them to store information about past inputs within their hidden state.
Unlike feedforward neural networks, which process data in a single forward pass and have no memory of past inputs, RNNs have a recurrent connection that allows them to maintain information about previous inputs in their hidden state. A dynamical system may be defined by:

$$h_t = f_h(X_t, h_{t-1}) \tag{10.1}$$
$$y_t = f_o(h_t) \tag{10.2}$$

A simple RNN [30] comprises three layers: an input layer, a hidden layer, and an output layer as shown in Figure 10.1. The input layer receives sequential data during each time step, the hidden layer retains memory and processes the sequential information, and the output layer generates the final output or prediction.

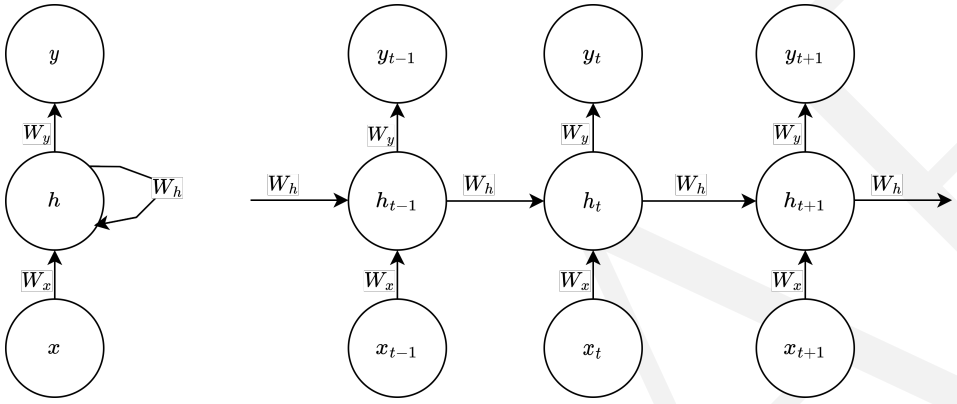


Figure 10.1: An unfolded simple recurrent neural network. The matrices W_x, W_h, W_y are shared between time steps. Unfolding a simple RNN helps to illustrate how the hidden state is passed along and updated as the network processes sequential data. It provides a visual representation of the temporal dependencies and the flow of information within the RNN.

Let's define the forward pass equations of the simple RNN model:

- Hidden state activation:

$$z_t^h = W_h h_{t-1} + W_x x_t + b_h \tag{10.3}$$
$$h_t = \tanh(z_t^h) \tag{10.4}$$

- Output activation:

$$z_t^y = W_y h_t + b_y \tag{10.5}$$
$$y_t = \text{softmax}(z_t^y) \tag{10.6}$$

The hidden activation function is tanh and its derivative is $1 - \tanh^2(x)$. The output activation function is softmax, which for a specific class i is defined as:

$$\text{softmax}(z_i^y) = \frac{e^{z_i^y}}{\sum_j e^{z_j^y}} \tag{10.7}$$

Where y represents the output and l represents the target. The activations of the hidden state and output are z^h and z^y respectively. The weight matrix connecting the hidden layer to the output is W_y.
Let's denote d_{neurons} as the number of neurons in the hidden layer and d_{inputs} as the number of input features. Then:

- x_t is a vector of size $d_{\text{inputs}}x1$ containing the inputs at time t.
- h_{t-1} is a $d_{\text{neurons}}x1$ vector containing the hidden state of the previous time-step. At the first time step, $t = 0$, there are no previous hidden state, so $h_{t-1} = 0$.
- W_x is a $d_{\text{neurons}} \times d_{\text{inputs}}$ matrix containing the connection weights between input and the hidden layer.
- W_h is a $d_{\text{neurons}} \times d_{\text{neurons}}$ matrix containing the connection weights between two hidden layers.
- W_y is a $d_{output} \times d_{\text{neurons}}$ matrix containing the connection weights between the hidden layer and the output.
- b_h is a vector of size $d_{\text{neurons}}x1$ containing each neuron's bias term.
- b_y is a vector of size $d_{\text{output}}x1$ containing the bias term of the output layer.
- y_t is a vector of size $d_{\text{output}}x1$ containing the layer's outputs at time step t.

RNNs are designed to handle sequential data with varying lengths, dividing the input data into multiple time steps, where each step represents an element of the sequence.
A distinguishing feature of RNNs is the recurrent connection, which allows the network to retain information across time steps. At each time step, the hidden layer takes input from the current step as well as the hidden state from the previous step, enabling the hidden state to carry information from past steps. Moreover, the same set of weights and biases is used at each time step, enabling effective processing of sequences with different lengths.

10.2 Learning Problem

RNNs are trained using backpropagation through time (BPTT), an extension of the standard backpropagation algorithm that accounts for the recurrent nature of the network. BPTT calculates gradients and updates the weights and biases to minimize the discrepancy between the predicted output and the target value. For the backward pass, we will compute the gradients of the loss with respect to the activations and parameters. Let's define the loss at each time step as:

$$E_t = -\sum_i l_{ti} \log y_{ti} \tag{10.8}$$

This is the categorical cross-entropy loss for softmax outputs at time t. The overall loss is given by

$$E = \sum_{t=1}^{T} E_t(l_t, y_t) \tag{10.9}$$

$$= -\sum_{t}^{T} l_t^T \log y_t \tag{10.10}$$

$$= -\sum_{t=1}^{T} l_t^T \log \left[\text{softmax}(z_t^y)\right] \tag{10.11}$$

Firstly, let's define the quantities δ_t^y and δ_t^h:

$$\delta_t^y = \frac{\partial E_t}{\partial z_t^y} \tag{10.12}$$

To get the gradient with respect to the pre-activation z_t^y, we need to apply the chain rule to differentiate L_t:

$$\frac{\partial E_t}{\partial z_t^y} = \frac{\partial E_t}{\partial y_t} \times \frac{\partial y_t}{\partial z_t^y} \tag{10.13}$$

This derivative is calculated as the derivative of the loss function with respect to the output, multiplied by the derivative of the softmax activation function. Using the properties of the softmax function and the cross-entropy loss (i.e. see Chapter 6), the above simplifies to:

$$\delta_t^y = y_t - l_t \tag{10.14}$$

Now let's define δ_t^h:

$$\delta_t^h = \frac{\partial E_t}{\partial z_t^h} \tag{10.15}$$

This derivative includes all paths through the computational graph where z_t^h affects the loss function E. This is calculated using the chain rule as:

$$\delta_t^h = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial z_t^h} = \left(\frac{\partial E_t}{\partial z_t^y}\frac{\partial z_t^y}{\partial h_t} + \frac{\partial E_t}{\partial z_{t+1}^h}\frac{\partial z_{t+1}^h}{\partial h_t}\right)\frac{\partial h_t}{\partial z_t^h} \tag{10.16}$$

Substituting δ_t^y and δ_{t+1}^h and the derivative of the tanh function:

$$\delta_t^h = \left({W_y}^T \delta_t^y + {W_h}^T \delta_{t+1}^h\right) \odot (1 - \tanh^2(z_t^h)) \tag{10.17}$$

This equation is the form of the backpropagation through time (BPTT) for hidden states in RNN. The $\odot$ denotes the Hadamard product (element-wise multiplication). The term $1 - \tanh^2(z_t^h)$ is the derivative of the tanh activation function. The terms $\delta_t^y {W_y}^T$ and $\delta_{t+1}^h {W_h}^T$ represent the influence of the error at the output at time t and the influence of the error at the hidden state at time $t+1$ respectively. They are both backpropagated to the hidden state at time t through the respective weight matrices W_y and W_h.
For an input sequence of length T, the gradient accumulations are given by
For the weight matrix W_y:

$$\frac{\partial E}{\partial W_y} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^y}\frac{\partial z_t^y}{\partial W_y} = \sum_{t=1}^{T}\delta_t^y \cdot h_t^T \tag{10.18}$$

For the bias b_y:

$$\frac{\partial E}{\partial b_y} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^y} = \sum_{t=1}^{T}\delta_t^y \tag{10.19}$$

For the recurrent weight matrix W_h:

$$\frac{\partial E}{\partial W_h} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^h}\frac{\partial z_t^h}{\partial W_h} = \sum_{t=1}^{T}\delta_t^h \cdot h_{t-1}^T \tag{10.20}$$

For the input weight matrix W_x:

$$\frac{\partial E}{\partial W_x} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^h}\frac{\partial z_t^h}{\partial W_x} = \sum_{t=1}^{T}\delta_t^h \cdot x_t^T \tag{10.21}$$

For the bias b_h:

$$\frac{\partial E}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial z_t^h} = \sum_{t=1}^{T} \delta_t^h \tag{10.22}$$

After accumulating the gradients over the entire sequence, you can use an optimization algorithm (like SGD, Adam, etc.) to update the weights and biases:

$$W = W - \eta \frac{\partial E}{\partial W} \tag{10.23}$$

$$b = b - \eta \frac{\partial E}{\partial b} \tag{10.24}$$

where η is the learning rate.
Simple RNNs face a challenge known as the vanishing or exploding gradients problem [31]. This arises when gradients in the network become excessively small (vanishing gradients) or large (exploding gradients) while processing lengthy sequences. This issue hampers the training process and the model's ability to retain long-term dependencies.

10.3 The Difficulty of Training Simple RNN

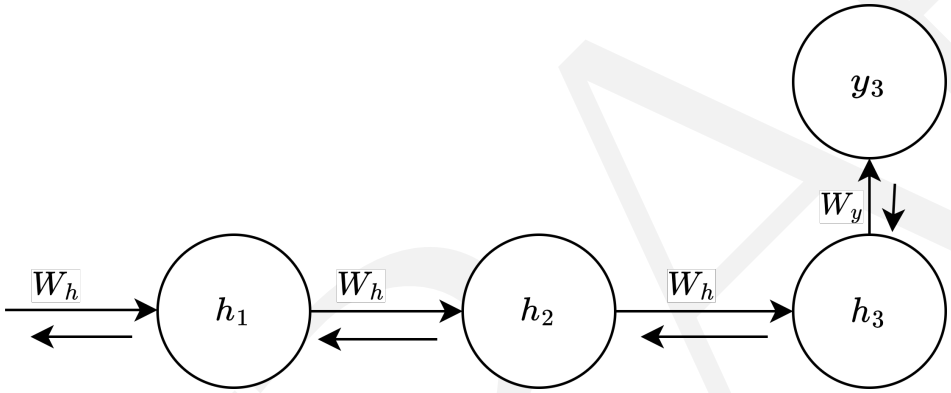


Figure 10.2: Propagating gradients through the unfolded RNN. The memory unit, h_t, is a function of its previous memory unit h_{t-1}. Hence, we differentiate h_3 with h_2 and h_2 with h_1.

In the following Figure 10.2, we have three-time steps. Then

$$\frac{\partial E_3}{\partial W_h} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial W_h} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_h} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_h}, \tag{10.25}$$

where the first term is a direct application of the chain rule. However, we have to take into consideration the previous time steps. So, we differentiate the cost function with respect to memory units h_2 as well as h_1 taking into consideration the weight matrix W_h. Please note that a memory unit h_t is a function of its previous memory unit h_{t-1} according to the recursive formulation ($h_t = f(W_h h_{t-1} + W_x x_t + b_h)$). Hence, we differentiate h_3 with h_2 and h_2 with h_1.
Hence, at the time-step t, we can compute the gradient and further use backpropagation through time from t to 1 to compute the overall gradient with respect to W_h:

$$\frac{\partial E_t}{\partial W_h} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_h}, \tag{10.26}$$

and $\frac{\partial h_t}{\partial h_k}$ can be computed using a chain rule. It can be written as follows

$$\frac{\partial E_t}{\partial W_h} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_h} \tag{10.27}$$

where

$$\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W_h^\top \operatorname{diag}[f'(h_{j-1})] = \frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \ldots \frac{\partial h_{k+1}}{\partial h_k}, \tag{10.28}$$

Since we differentiate a vector function with respect to a vector, the result is a matrix (called the Jacobian matrix) whose elements are all point-wise derivatives. Aggregate the gradients with respect to W_h over the whole time-steps with back-propagation, we can finally yield the following gradient with respect to W_h:

$$\frac{\partial E}{\partial W_h} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_h} \tag{10.29}$$

Hence:

$$\frac{\partial E}{\partial \mathbf{W}_h} = \sum_{t=1}^{T} \left(\sum_{k=1}^{t} \frac{\partial E_t}{\partial h_k} \prod_{j=k+1}^{t} W_h^\top \cdot \operatorname{diag}[f'(W_x x_j + W_h h_{j-1} + b_h)] \cdot f'(W_x x_k + W_h h_{k-1} + b_h) h_{k-1}^\top \right) \tag{10.30}$$

Let's take the norms[1] of these Jacobians $\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\|$:

[1] The 2-norm may be interpreted as an absolute value, of the Jacobian matrix.

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \left\| W_h^\top \right\| \left\| \text{diag}[f'(h_{j-1})] \right\| \tag{10.31}$$

In this equation, we set γ_W, the largest eigenvalue associated with $\left\| W_h^\top \right\|$ as its upper bound, while γ_h largest eigenvalue associated with $\left\| \text{diag}[f'(h_{j-1})] \right\|$ as its corresponding upper-bound. Thus, the chosen upper bounds γ_W and γ_h end up being a constant term resulting from their product:

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \gamma_W \gamma_h \tag{10.32}$$

Depending on the chosen activation function f, the derivative f' will be upper bounded by different values. For hyperbolic tangent function as shown in Figure 10.3, we have $\gamma_h = 1$ while for sigmoid function, we have $\gamma_h = 0.25$.
The gradient $\frac{\partial h_t}{\partial h_k}$ is a product of Jacobian matrices that are multiplied many times, $t - k$ times in our case:

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\gamma_W \gamma_h)^{t-k} \tag{10.33}$$

The equation can be paraphrased as follows: The magnitude of the partial derivative of the hidden state at time step t with respect to the hidden state at time step k is equal to the magnitude of the product of the partial derivatives of consecutive hidden states from time step $k + 1$ to time step t. This magnitude is bounded by the value of the product of two factors, γ_W and γ_h, raised to the power of $t - k$. When the sequence becomes longer, meaning there is a larger distance between time steps t and k, the equation shows that the value can either become very small or very large quickly. This violates the assumption of locality in gradient descent[2]. The outcome depends on the value of gamma: if it is large, the gradient can explode (become very large), and if it is small, the gradient can vanish (become very small). These problems highlight that when the gradient vanishes, it implies that the earlier hidden states do not significantly influence the later hidden states. In other words, the network fails to learn long-term dependencies, as the information from earlier time steps becomes negligible or irrelevant in the later ones.
Assume $\gamma_h = 1$ then if the norm of the weight matrix $W_h^\top$ is less than 1, each time we multiply the gradient by $||W_h^\top||$, the magnitude of the gradient decreases. Imagine multiplying a number by 0.5 repeatedly - the result gets smaller and smaller. This is

[2]Both vanishing and exploding gradients violate the assumption of locality in gradient descent because they disrupt the smoothness and stable progression of the optimization process.

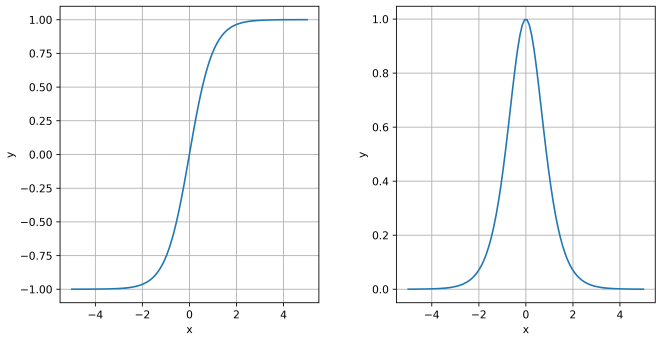


Figure 10.3: The hyperbolic tangent activation function and its derivative.

why $||W_h^\top|| < 1$ is a sufficient[3] condition for the gradients to vanish. It guarantees vanishing gradients because multiplying by a number less than one repeatedly will always cause the result to tend towards zero. However, it's not a necessary condition because there are other reasons gradients might vanish, such as saturating activation functions.

On the other hand, if the norm of the weight matrix $W_h^\top$ is greater than 1, each multiplication by $||W_h^\top||$ increases the magnitude of the gradient. Think about multiplying a number by 2 over and over - the result gets larger and larger. So $||W_h^\top|| > 1$ is a necessary condition for gradient explosion. But just because $||W_h^\top|| > 1$ doesn't guarantee that the gradients will explode. Other factors might prevent this, such as gradient clipping or careful initialization of the weights. This is why $||W_h^\top|| > 1$ is necessary, but not *sufficient*, for gradient explosion.

Remember that these are simplifications - in reality, the behavior of the gradients in an RNN will depend on a combination of many factors, including the specific sequences of inputs, the activation functions, and the structure of the network, in addition to the weight matrices. But considering the norms of the weight matrices provides some insight into the challenges faced when training RNNs, namely the problems of vanishing and exploding gradients.

The problem of vanishing gradients is not exclusive to recurrent neural networks (RNNs) but also occurs in deep feedforward neural networks. However, RNNs are typically deeper, which makes this issue more common in RNNs.

The vanishing or exploding gradients problem can hinder the effective capture of long-range dependencies by simple RNNs. To mitigate this issue, more advanced

[3]The difference between necessary and sufficient conditions can be quite subtle. In mathematics, a necessary condition must be true for the given statement to be true, but it is not enough on its own to guarantee the statement is true. A sufficient condition, on the other hand, if true, guarantees the statement is true.

RNN architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were introduced. These architectures incorporate mechanisms that enable better control of information flow within the network, addressing the challenges associated with vanishing or exploding gradients.

10.4 Long Short-Term Memory Networks

In practice, RNNs cannot capture long term dependencies due to the gradient exploding and vanishing problems [31]. LSTMs were introduced [32] to overcome the gradient vanishing problems and they are an essential component for many applications.

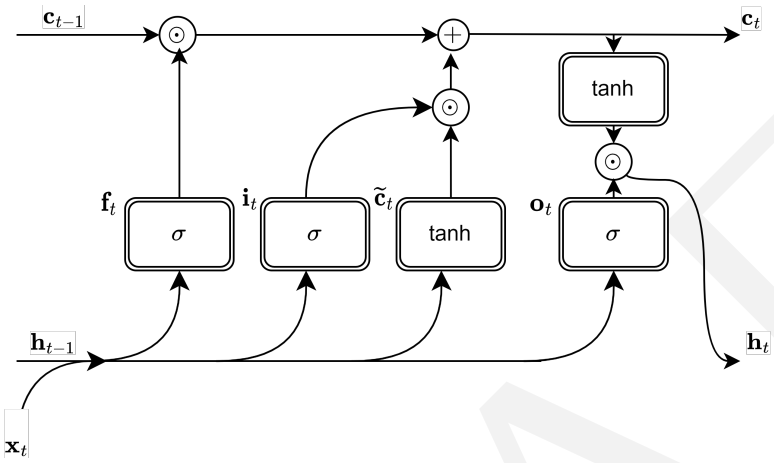


Figure 10.4: The hidden state computation in an LSTM network.

An LSTM network (see Figure 10.5) has a memory cell and three gating units: the input gate is used to control the amount of information to *add* to the current memory, the forget gate is used to control the amount of information to *remove* from the previous memory, and the output gate is used to control the amount of information to *output* from the current memory. These gates take as input the previous hidden state and the current input, and outputs a number between 0 and 1 (i.e. logistic function). The flow of information into or out of the memory is controlled by the multiplication of the output of these gates. The updates at each time step t are

given by:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{h}_{t-1} + \mathbf{U}_i\mathbf{x}_t) \tag{10.34}$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{h}_{t-1} + \mathbf{U}_f\mathbf{x}_t) \tag{10.35}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o\mathbf{h}_{t-1} + \mathbf{U}_o\mathbf{x}_t) \tag{10.36}$$
$$\widetilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t) \tag{10.37}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \widetilde{\mathbf{c}}_t \tag{10.38}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{10.39}$$

where $\mathbf{i}_t$ is the input gate, $\mathbf{f}_t$ is the forget gate, $\mathbf{o}_t$ is the output gate, $\mathbf{c}_t$ is the memory cell, and $\mathbf{h}_t$ is the hidden state. $\odot$ denotes element-wise multiplication. $\mathbf{W}_i$,$\mathbf{U}_i$, $\mathbf{W}_f$,$\mathbf{U}_f$, $\mathbf{W}_o$,$\mathbf{U}_o$,$\mathbf{W}_c$,$\mathbf{U}_c$, are weight matrices (parameters) of the LSTM network. A variant of LSTM known as bidirectional BiLSTM [33] allows the integration of both past and future information. It is a combination of two LSTMs in two directions: one operates in the forward direction and the other operates in the backward direction. Hence, each input word at time t is aware about the past and future contexts which may improve the results.
Similar to LSTMs, Gated Recurrent Units (GRUs) were developed to handle long term dependencies [34]. The gated recurrent units (GRUs) [35] which have the following forward updates:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z\mathbf{h}_{t-1} + \mathbf{U}_z\mathbf{x}_t) \tag{10.40}$$
$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{U}_r\mathbf{x}_t) \tag{10.41}$$
$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h(\mathbf{h}_t \odot \mathbf{r}_t) + \mathbf{U}_h\mathbf{x}_t) \tag{10.42}$$
$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t \tag{10.43}$$

where $\mathbf{z}_t$ is the update gate, $\mathbf{r}_t$ is the reset gates. $\mathbf{W}_z$,$\mathbf{U}_z$, $\mathbf{W}_r$,$\mathbf{U}_r$, $\mathbf{W}_h$,$\mathbf{U}_h$, are the parameters of the GRU networks.

10.4.1 Vanishing/Exploding Gradients with LSTMs

The cell state in the LSTMs is given by

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \widetilde{\mathbf{c}}_t \tag{10.44}$$

To find the derivative $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$, we notice that $\mathbf{c}_t$ is a function of $\mathbf{f}_t$ (the forget gate), $\mathbf{i}_t$ (input gate) and $\widetilde{\mathbf{c}}_t$ (candidate input), and each of these being a function of $\mathbf{c}_{t-1}$

(since they are all functions of $\mathbf{h}_{t-1}$). Using the multivariate chain rule we get:

$$\begin{aligned}\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} &= \frac{\partial \mathbf{f}_t}{\partial \mathbf{c}_{t-1}} \cdot \mathbf{c}_{t-1} + \mathbf{f}_t + \frac{\partial \mathbf{i}_t}{\partial c_{t-1}} \cdot \widetilde{\mathbf{c}}_t + \frac{\partial \widetilde{\mathbf{c}}_t}{\partial c_{t-1}} \cdot \mathbf{i}_t \\ &= \sigma'(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t) \cdot \mathbf{W}_f \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \mathbf{c}_{t-1} \\ &+ \mathbf{f}_t \\ &+ \sigma'(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t) \cdot \mathbf{W}_i \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \widetilde{\mathbf{c}}_t \\ &+ \sigma'(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t) \cdot \mathbf{W}_c \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \mathbf{i}_t\end{aligned} \tag{10.45}$$

Hence, the cell state gradient is an additive function of the four terms computed in the above equation. During the backpropagation, it is possible for these additive terms to have a value of $\vec{1}$. Therefore, using LSTMs, the neural network is trained to determine when the gradient should vanish and when it should be retained by adjusting the values of the four terms.
The LSTM design is not always sufficient to prevent the issue of exploding gradients. Therefore, in successful applications of LSTM, an additional technique called gradient clipping is often employed. Gradient clipping helps mitigate the problem of exploding gradients by constraining the magnitude of the gradients during the training process.

10.5 Independently Recurrent Neural Network

The Independently Recurrent Neural Network (IndRNN) [36] is a type of RNN designed to address the issues of gradient vanishing and exploding that often occur in traditional RNNs. The key idea behind IndRNN is to have separate recurrent connections for each neuron, making the neurons operate independently across different time steps. This independence allows for better control over the gradient flow during backpropagation, thereby mitigating the problems of gradient vanishing and exploding.
For IndRNN, the hidden state $h_{i,t}$ for each neuron i at time step t is computed as:

$$h_{i,t} = f(W_x x_t + u_i h_{i,t-1} + b_i), \tag{10.46}$$

where $h_{i,t}$ is the hidden state of the i-th neuron at time step t, x_t is the input at time step t, W_x is the input weight shared across all neurons, u_i is the recurrent weight for the i-th neuron, b_i is the bias for the i-th neuron, and f is an activation function, typically ReLU.
To understand how IndRNN addresses gradient vanishing and exploding, let's derive the gradient of the loss $\mathcal{L}$ with respect to the hidden state $h_{i,t}$. Using the chain rule, the gradient of the loss $\mathcal{L}$ with respect to the hidden state $h_{i,t}$ can be decomposed as follows:

$$\frac{\partial \mathcal{L}}{\partial h_{i,t}} = \sum_{k=t+1}^{T} \frac{\partial \mathcal{L}}{\partial h_{i,k}} \cdot \frac{\partial h_{i,k}}{\partial h_{i,t}}, \tag{10.47}$$

and the partial derivative $\frac{\partial h_{i,k}}{\partial h_{i,t}}$ is:

$$\frac{\partial h_{i,k}}{\partial h_{i,t}} = \prod_{j=t+1}^{k} \frac{\partial h_{i,j}}{\partial h_{i,j-1}} \tag{10.48}$$

Given:

$$\frac{\partial h_{i,j}}{\partial h_{i,j-1}} = u_i f'(W_x x_j + u_i h_{i,j-1} + b_i) \tag{10.49}$$

Therefore, the overall gradient becomes:

$$\frac{\partial \mathcal{L}}{\partial h_{i,t}} = \sum_{k=t+1}^{T} \frac{\partial \mathcal{L}}{\partial h_{i,k}} \cdot \prod_{j=t+1}^{k} u_i f'(W_x x_j + u_i h_{i,j-1} + b_i) \tag{10.50}$$

The key here is that the recurrent weight u_i is a single scalar, making it easier to control the gradient flow. The activation function f, typically chosen as ReLU, helps mitigate vanishing gradients because ReLU does not saturate like tanh or sigmoid where $f(x) = \max(0, x)$, $f'(x) = 1$ for $x > 0$, preventing the gradient from vanishing. Additionally, by having a recurrent weight u_i that is scalar, it can be initialized and constrained to values that prevent the gradients from diminishing too quickly. By carefully constraining the recurrent weights u_i, IndRNN can prevent the gradients from growing exponentially. For example, setting u_i to be within a specific range (e.g., $|u_i| < 1$ for stability) can maintain control over the gradient magnitude.

10.6 Bidirectional Recurrent Neural Networks

Bidirectional Recurrent Neural Networks (Bidirectional RNNs) are an extension of the traditional RNNs that can capture information from both past and future contexts by processing the sequence in both forward and backward directions. This is particularly useful for tasks where the entire context of the input sequence is important, such as in natural language processing and speech processing.

A Bidirectional RNN consists of two separate RNNs. A forward RNN that processes the sequence from the beginning to the end and backward RNN that processes the sequence from the end to the beginning.

Given an input sequence $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$, where T is the length of the sequence, the Bidirectional RNN computes two sets of hidden states. The forward hidden states $\overrightarrow{\mathbf{h}}_t$ and the backward hidden states $\overleftarrow{\mathbf{h}}_t$:

- The forward hidden states are computed as:

$$\overrightarrow{\mathbf{h}}_t = f(\mathbf{x}_t, \overrightarrow{\mathbf{h}}_{t-1}), \tag{10.51}$$

  where f is the RNN cell (e.g., LSTM or GRU) function applied to the input at time step t and the previous forward hidden state $\overrightarrow{\mathbf{h}}_{t-1}$.

- The backward hidden states are computed as:

$$\overleftarrow{\mathbf{h}}_t = f(\mathbf{x}_t, \overleftarrow{\mathbf{h}}_{t+1}), \tag{10.52}$$

  where f is the RNN cell function applied to the input at time step t and the next backward hidden state $\overleftarrow{\mathbf{h}}_{t+1}$.

- The final hidden state $\mathbf{h}_t$ for each time step t can be a combination of $\overrightarrow{\mathbf{h}}_t$ and $\overleftarrow{\mathbf{h}}_t$.

The combination of the forward and backward hidden states can be done in various ways, commonly referred to as merge modes. Two typical merge modes are "sum" and "concat":

- In the "sum" merge mode, the forward and backward hidden states are added element-wise:

$$\mathbf{h}_t = \overrightarrow{\mathbf{h}}_t + \overleftarrow{\mathbf{h}}_t, \tag{10.53}$$

  where this merge mode results in a hidden state with the same dimensionality as each of the forward and backward hidden states.

- In the "concat" merge mode, the forward and backward hidden states are concatenated along the feature dimension:

$$\mathbf{h}_t = \text{concat}(\overrightarrow{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t), \tag{10.54}$$

  where this merge mode results in a hidden state with double the dimensionality of each of the forward and backward hidden states.

Bidirectional RNNs are particularly powerful for tasks that require understanding context from both directions in a sequence. By using bidirectional processing, these models can achieve a more comprehensive understanding of the input sequence, leading to improved performance on various tasks.

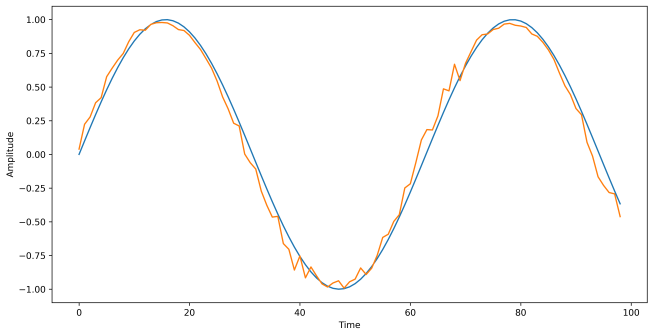


Figure 10.5: Sine wave prediction using RNN model. The curve with the orange color is the predicted one.

10.7 An Example

A Python code is provided to illustrate the learning algorithm of a regression problem based on the simple RNN model (Elman's network).

```
import numpy as np
import matplotlib.pyplot as plt

# Define the sine wave sequence
def generate_sequence(length):
    freq = 0.1  # Frequency of the sine wave
    x = np.arange(0, length)
    y = np.sin(freq * x)
    return y

# Define the ElmanRNN class
class ElmanRNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize the weights
        self.W_xh = np.random.randn(hidden_size, input_size)
        self.W_hh = np.random.randn(hidden_size, hidden_size)
        self.W_hy = np.random.randn(output_size, hidden_size)

        # Initialize the biases
        self.b_h = np.zeros((hidden_size, ))
        self.b_y = np.zeros((output_size, ))

```

```
    def forward(self, x):
        T = x.shape[0]
        self.h = np.zeros((T + 1, self.hidden_size))
        self.y = np.zeros((T, self.output_size))

        for t in range(T):
            self.h[t + 1] = np.tanh(np.dot(self.W_xh, x[t]) + np.dot
(self.W_hh, self.h[t]) + self.b_h)
            self.y[t] = np.dot(self.W_hy, self.h[t + 1]) + self.b_y

        return self.y

    def backward(self, x, y, learning_rate):
        T = x.shape[0]
        dL_dW_xh = np.zeros_like(self.W_xh)
        dL_dW_hh = np.zeros_like(self.W_hh)
        dL_dW_hy = np.zeros_like(self.W_hy)
        dL_db_h = np.zeros_like(self.b_h)
        dL_db_y = np.zeros_like(self.b_y)
        dh_next = np.zeros_like(self.h[0])


        for t in reversed(range(T)):
            dL_dy = 2 * (self.y[t] - y[t])
            dL_dW_hy += np.dot(dL_dy.T, self.h[t+1].reshape(-1, 1).T
)
            dL_db_y += dL_dy

            dh = np.dot(self.W_hy.T, dL_dy.T) + dh_next.reshape(-1,
1)
            dh_raw = (1 - self.h[t+1] ** 2) * dh[0]
            dL_db_h += dh_raw

            dL_dW_hh += np.dot(dh_raw.reshape(-1, 1), self.h[t].
reshape(1, -1))
            dL_dW_xh += np.dot(dh_raw.reshape(-1, 1), x[t].reshape
(1, -1))

            dh_next = np.dot(self.W_hh.T, dh_raw.reshape(-1, 1))



        # Update weights and biases
        self.W_xh -= learning_rate * dL_dW_xh
        self.W_hh -= learning_rate * dL_dW_hh
        self.W_hy -= learning_rate * dL_dW_hy
        self.b_h -= learning_rate * dL_db_h
        self.b_y -= learning_rate * dL_db_y

    def train(self, x, y, learning_rate, num_epochs):
```

```
        for epoch in range(num_epochs):
            y_pred = self.forward(x)
            self.backward(x, y, learning_rate)

            if (epoch + 1) % 100 == 0:
                loss = np.mean((y_pred - y) ** 2)
                print(f'Epoch: {epoch+1}/{num_epochs}, Loss: {loss}')

# Define the sequence length and generate the sine wave sequence
sequence_length = 100
sequence = generate_sequence(sequence_length)

# Prepare the training data
train_data = sequence[:-1]
train_target = sequence[1:]

# Reshape the training data for input to the Elman RNN
train_data = train_data.reshape(sequence_length-1, 1)
train_target = train_target.reshape(sequence_length-1, 1)

# Define hyperparameters
input_size = 1
hidden_size = 16
output_size = 1
learning_rate = 0.001
num_epochs = 10000

# Initialize the Elman RNN model
model = ElmanRNN(input_size, hidden_size, output_size)

# Train the model
model.train(train_data, train_target, learning_rate, num_epochs)

# Generate predictions for the sequence
predictions = model.forward(train_data)

# Plot the original sine wave and the predicted sine wave
plt.figure(figsize=(12, 6))
plt.plot(sequence[:-1], label='Original Sine Wave')  # Original sine wave
plt.plot(predictions, label='Predicted Sine Wave')  # Predicted sine wave
plt.xlabel('Time')  # X-axis label
plt.ylabel('Amplitude')  # Y-axis label
#plt.title('Original vs Predicted Sine Wave')  # Title of the plot
#plt.legend()  # Display legend
#plt.show()  # Display the plot
```

```
118 plt.savefig('rnn_sine.png', dpi=600)
```

Listing 10.1: Python example for sine wave prediction problem.

10.8 Assignments

Implement a neural langauge modeling algorithm using the RNN model using the Keras deep learning library and Google colab. To load the dataset, please visit https://github.com/karpathy/char-rnn/blob/master/data/tinyshakespeare/input.txt.

11. Attention Networks

Attention is all you need.

— Ashish Vaswani and colleagues

Recurrent Neural Networks (RNNs) operate sequentially, which makes them slow for both training and inference. In this chapter, we introduce self-attention networks as a replacement for RNN layers. One application of self-attention networks in the sequence-to-sequence Transformer model is in machine translation. In this context, the self-attention mechanism allows the model to learn the relationships between words in a sentence, regardless of their positions. This means that the model can capture long-range dependencies between words, which is crucial for accurately translating sentences from one language to another. For example, in translating an English sentence to French, the self-attention network helps the Transformer model to identify which words in the English sentence are most relevant to the words being generated in the French translation, resulting in more accurate and contextually appropriate translations. The Transformer model is built on a complex architecture. We will start by introducing the fundamental components (i.e. the scaled dot-product, self-attention networks, and masked self-attention networks) required to construct the Transformer model, and then proceed to provide a detailed explanation of the model itself.

11.1 Scaled Dot-Product Similarity Measure

One way to measure the similarity between two vectors is to compute the dot-product between them. Consider the dot-product $\mathbf{q} \cdot \mathbf{k}$ of two vectors $\mathbf{q}$ and $\mathbf{k}$ each of dimension d_k:

$$\mathbf{q} \cdot \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \tag{11.1}$$

Assume the following query and input vectors in a 3-dimensional space:

$$\mathbf{q} = [0.8, 0.3, 0.5]$$

$$\mathbf{k}_1 = [0.9, 0.1, 0.4], \quad \mathbf{k}_2 = [0.4, 0.7, 0.5], \quad \mathbf{k}_3 = [0.7, 0.2, 0.6]$$

Let us compute the dot-product similarity between the $\mathbf{q}$ and the three vectors $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3$:

$$\mathbf{q} \cdot \mathbf{k}_1 = (0.8)(0.9) + (0.3)(0.1) + (0.5)(0.4) = 0.72 + 0.03 + 0.2 = 0.95$$

$$\mathbf{q} \cdot \mathbf{k}_2 = (0.8)(0.4) + (0.3)(0.7) + (0.5)(0.5) = 0.32 + 0.21 + 0.25 = 0.78$$

$$\mathbf{q} \cdot \mathbf{k}_3 = (0.8)(0.7) + (0.3)(0.2) + (0.5)(0.6) = 0.56 + 0.06 + 0.3 = 0.92$$

We can rank the input vectors based on their similarity scores with $\mathbf{q}$: $\mathbf{k}_1 > \mathbf{k}_3 > \mathbf{k}_2$ Thus, vector 1 is the most relevant, followed by vector 3 and vector 2.
As the scores tend to increase with the dimensionality of the query vector, the scaled dot-product is often used to normalize this effect:

$$\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}} = \frac{\sum_{i=1}^{d_k} q_i k_i}{\sqrt{d_k}}, \tag{11.2}$$

If q_i and k_i are independently drawn from a distribution with zero mean and variance σ^2, then: Since the mean of each q_i and k_i is zero:

$$\mathbb{E}[q_i] = 0, \quad \mathbb{E}[k_i] = 0$$

The expectation of the dot-product is:

$$\mathbb{E}[\mathbf{q} \cdot \mathbf{k}] = \mathbb{E}\left[\sum_{i=1}^{d_k} q_i k_i\right] = \sum_{i=1}^{d_k} \mathbb{E}[q_i k_i]$$

Since q_i and k_i are independent:

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = 0$$

Therefore:

$$\mathbb{E}[\mathbf{q} \cdot \mathbf{k}] = 0$$

To compute the variance of the dot-product, we consider the variance of each term $q_i k_i$:

$$\text{Var}(q_i k_i) = \mathbb{E}[(q_i k_i)^2] - (\mathbb{E}[q_i k_i])^2$$

Since $\mathbb{E}[q_i k_i] = 0$, we have:

$$\text{Var}(q_i k_i) = \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = \sigma^2 \sigma^2 = \sigma^4$$

For the sum of d_k independent products $q_i k_i$, the variance is the sum of the variances of each product:

$$\text{Var}(\mathbf{q} \cdot \mathbf{k}) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = d_k \sigma^4$$

The standard deviation of the dot-product grows with $\sqrt{d_k}$. Without scaling, the variance increases linearly with d_k, leading to instability. Scaling by $\sqrt{d_k}$ normalizes the variance to:

$$\text{Var}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right) = \frac{d_k \sigma^4}{d_k} = \sigma^4$$

Hence, this normalization keeps the variance of the dot-product scores consistent, independent of the dimension d_k.

11.2 Multi-head Self-Attention Networks

Self-Attention Networks (SANs) form the foundation of Transformer models [37]. These networks are designed to learn contextual relationships between input vectors and can effectively capture long-term dependencies, replacing the recurrent connections used in Recurrent Neural Networks (RNNs). Additionally, SANs are significantly faster than RNNs because they operate in parallel.

Consider an input matrix and a query vector: SANs calculate a similarity score between the query vector and parts of the input matrix, giving more attention to similar parts. This score is then used to transform the input matrix into an output vector. The output vector is a weighted sum (or average) of the input matrix, resulting in a richer representation than the original input.

Mathematically, the Self-Attention process involves the following steps in matrix form:

1. Calculate Query, Key, and Value vectors: For an input sequence $X \in \mathbb{R}^{n \times d_{\text{model}}}$ (where n is the sequence length and d_{model} is the dimension of each input vector), we linearly transform the input using weight matrices $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$:

$$Q = XW^Q \quad (n \times d_k), \quad K = XW^K \quad (n \times d_k), \quad V = XW^V \quad (n \times d_v)$$

2. Compute the attention scores: The attention score between two tokens is the dot-product of their Query and Key vectors. We compute a score matrix $A \in \mathbb{R}^{n \times n}$:

$$A = QK^T \quad (n \times d_k) \times (d_k \times n) = (n \times n)$$

3. Scale the attention scores: To ensure stable gradients, the scores are scaled by the square root of the dimension of the key vectors, d_k as discussed in the previous section:

$$A = \frac{QK^T}{\sqrt{d_k}} \quad (n \times n)$$

4. Apply the softmax function: To get the attention weights, we apply the softmax function to the scaled scores. Then, we compute the attention output:

$$Z = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \quad (n \times n) \times (n \times d_v) = (n \times d_v) \tag{11.3}$$

Figure 11.1 illustrates the process of computing the attention mechanism for the input query $\mathbf{x}_2$. It demonstrates how $\mathbf{x}_2$ interacts with other inputs in the sequence to generate attention scores. These scores are used to weigh the value vectors, resulting in the output for $\mathbf{z}_2$. Ultimately, the attention mechanism captures relevant information from other inputs to produce a context-aware representation of $\mathbf{x}_2$. The attention score α_i for each time step is computed as:

$$\alpha_{2i} = \frac{\exp(\mathbf{q}_2^\top \mathbf{k}_i)}{\sum_j \exp(\mathbf{q}_2^\top \mathbf{k}_j)},$$

where $\mathbf{q}_2$: Query vector for the input $\mathbf{x}_2$ and $\mathbf{k}_i$: Key vector at time step i.
The output is a weighted combination of value vectors $\mathbf{v}_i$:

$$\mathbf{z}_2 = \sum_i \alpha_{2i} \mathbf{v}_i,$$

where $\mathbf{v}_i$ is the value vector associated with the key $\mathbf{k}_i$.

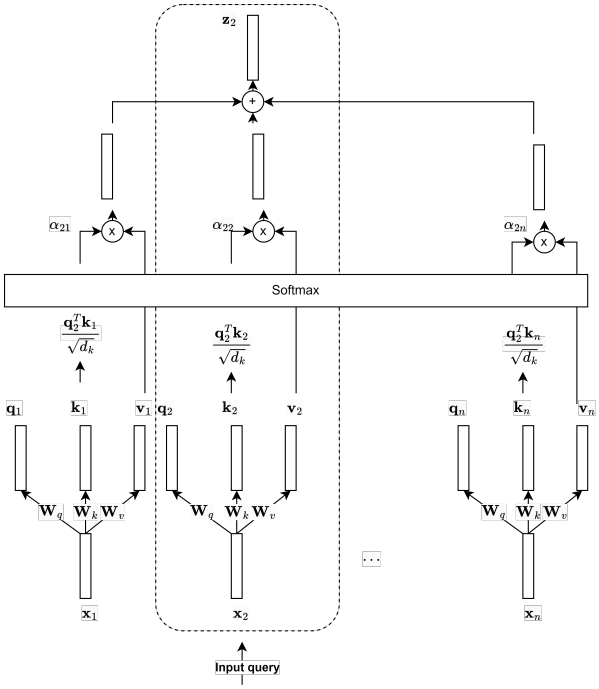


Figure 11.1: For the input query $\mathbf{x}_2$, self-attention generates an output vector $\mathbf{z}_2$. This vector $\mathbf{z}_2$ has a dimensionality of d_v, which corresponds to the size of the value vectors in the self-attention mechanism. The attention process computes a weighted sum of the value vectors, using attention scores based on the similarity between $\mathbf{x}_2$ and other inputs. Ultimately, $\mathbf{z}_2$ represents the contextualized embedding for $\mathbf{x}_2$ after the attention mechanism.

It is possible to improve the self-attention performance by running i self-attention blocks (i.e. multi-head attention) in parallel. This means that the key, query, and value matrices are split into a number of heads and projected. The individual splits are then passed into a self-attention block as described above. The Multi-Head Self-Attention process involves the following steps in matrix form:

1. Split the input: For each head, we linearly transform the input X into Q_i, K_i, and V_i using different weight matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$:

$$Q_i = XW_i^Q \quad (n \times d_k), \quad K_i = XW_i^K \quad (n \times d_k), \quad V_i = XW_i^V \quad (n \times d_v)$$

2. Compute the attention: For each head, compute the attention output:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \quad (n \times d_v)$$

3. Concatenate the head: Concatenate the attention outputs from all heads. If there are h heads and each head has an output dimension of d_v, the concatenated output has dimension $n \times (h \cdot d_v)$:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) \quad (n \times (h \cdot d_v))$$

4. Final linear transformation: Apply a final linear transformation to the concatenated output using weight matrix $W^O \in \mathbb{R}^{(h \cdot d_v) \times d_{\text{model}}}$:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) W^O, \tag{11.4}$$

where the final matrix shape is $(n \times (h \cdot d_v)) \times ((h \cdot d_v) \times d_{\text{model}}) = (n \times d_{\text{model}})$

Figure 11.2 demonstrates the process of implementing multihead self-attention. It shows multiple self-attention mechanisms operating in parallel. Each attention head processes the input independently and the outputs are later combined. This approach helps the model capture different aspects of the input sequence.

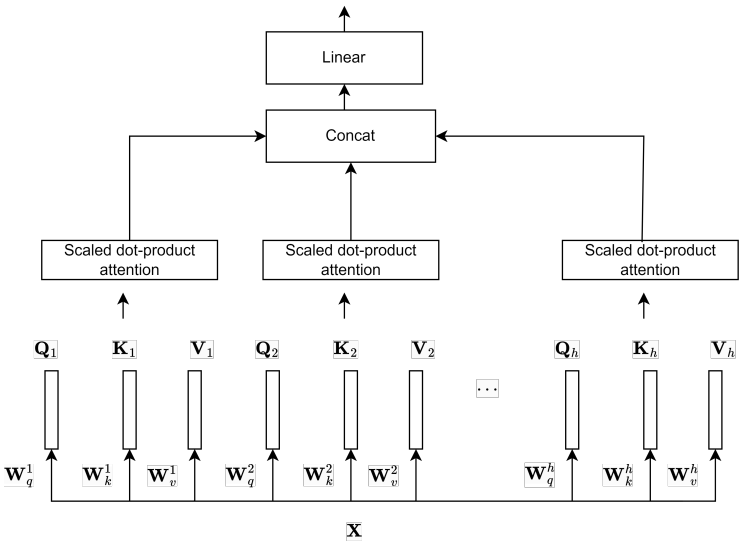


Figure 11.2: Multihead self-attention operates by utilizing multiple self-attention mechanisms, with h attention heads working concurrently. Each attention head independently processes the input, capturing different aspects of the sequence relationships. These results are then combined and transformed through a linear layer to create the final output. This parallel processing allows the model to focus on diverse features of the input simultaneously, enhancing its ability to capture complex dependencies.

11.2.1 Numerical Example for Self-Attention

Consider a simple example with a sequence of 2 inputs where the dimension of each input is 2. Here, $d_{\text{model}} = d_x = d_k = d_v = 2$.

1. Input sequence X:

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

2. Weight matrices W^Q, W^K, and W^V:

$$W^Q = W^K = W^V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

3. Calculate Q, K, and V:

$$Q = XW^Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

$$K = XW^K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

$$V = XW^V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

4. Compute the attention scores A:

$$A = QK^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2 \times 2)$$

5. Scale the attention scores:

$$A = \frac{QK^T}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.707 & 0 \\ 0 & 0.707 \end{bmatrix} \quad (2 \times 2)$$

6. Apply the softmax function:

$$\text{softmax}(A) = \begin{bmatrix} \frac{e^{0.707}}{e^{0.707}+e^0} & \frac{e^0}{e^{0.707}+e^0} \\ \frac{e^0}{e^{0.707}+e^0} & \frac{e^{0.707}}{e^{0.707}+e^0} \end{bmatrix} = \begin{bmatrix} 0.669 & 0.33 \\ 0.33 & 0.669 \end{bmatrix} \quad (2 \times 2)$$

7. Calculate the final attention output:

$$\text{Attention}(Q, K, V) = \text{softmax}(A)V = \begin{bmatrix} 0.669 & 0.33 \\ 0.33 & 0.669 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.669 & 0.33 \\ 0.33 & 0.669 \end{bmatrix} \quad (2 \times 2)$$

This example illustrates how self-attention works with simple values. In practice, the dimensions and values would be larger and more complex.

11.2.2 Masked Self-Attention

Masked self-attention is a variant of the self-attention mechanism used primarily in the decoder part of the Transformer model [37]. It ensures that the prediction for a particular position in the sequence does not depend on future positions. This is crucial for autoregressive tasks like language modeling, where the model generates tokens one by one. Masked self-attention is also referred to as causal attention because it ensures that the attention mechanism respects the causal structure of

the sequence (i.e., the prediction at a given time step depends only on the previous time steps).
In masked self-attention, future tokens are masked out. This means that when computing the attention for a token at position t, the model only attends to tokens at positions $\leq t$. The masked Self-Attention process involves the following steps in matrix form:

1. Compute Queries, Keys, and Values:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

2. Compute Raw Attention Scores:

$$A = \frac{QK^T}{\sqrt{d_k}}$$

3. Apply the Mask: A mask matrix M is created, where $M_{ij} = -\infty$ if $j > i$ and $M_{ij} = 0$ otherwise. The raw attention scores are then adjusted using this mask:

$$A_{\text{masked}} = A + M$$

This operation effectively nullifies the influence of future tokens by setting their corresponding scores to $-\infty$.

4. Apply the Softmax Function:

$$\text{Attention}(Q, K, V) = \text{softmax}(A_{\text{masked}})V \tag{11.5}$$

Consider an input sequence $X = [x_1, x_2, x_3]$ with corresponding query, key, and value matrices Q, K, and V. We can compute the raw attention scores A as follows:

$$A = \frac{QK^T}{\sqrt{2}} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Then we build the mask matrix M:

$$M = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}$$

and compute the masked attention scores:

$$A_{\text{masked}} = A + M = \begin{bmatrix} a_{11} & -\infty & -\infty \\ a_{21} & a_{22} & -\infty \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

then the softmax is applied:

$$\text{Attention}(Q, K, V) = \text{softmax}(A_{\text{masked}})V$$

This ensures that when computing the attention for the first token, only the first token is considered; for the second token, only the first and second tokens are considered; and so on.
Masked self-attention is crucial for tasks where the model needs to generate sequences in an autoregressive manner, ensuring that the prediction for each token depends only on the previous tokens and not on future tokens. This mechanism is fundamental in the decoder of the Transformer model, enabling it to handle sequence generation effectively.

11.3 Stacking Self-Attention Layers

Stacking self-attention layers without non-linear transformations or feed-forward networks between them results in a single linear transformation. This equivalence follows from the associative property of matrix multiplication and the linearity of the self-attention mechanism. Let me provide a clearer mathematical derivation to show how multiple stacked self-attention layers without feed-forward networks (FNN) can be equivalent to a single self-attention layer.
A self-attention layer computes the output Z from an input matrix $X \in \mathbb{R}^{n \times d_k}$ using the
Consider two self-attention layers stacked on top of each other. The output of the first layer is:

$$Z_1 = \text{Attention}(Q_1, K_1, V_1) = \text{softmax}\left(\frac{Q_1 K_1^\top}{\sqrt{d_k}}\right) V_1$$

where:

$$Q_1 = XW_1^Q, \quad K_1 = XW_1^K, \quad V_1 = XW_1^V$$

Substituting these:

$$\begin{aligned} Z_1 &= \text{softmax}\left(\frac{(XW_1^Q)(XW_1^K)^\top}{\sqrt{d_k}}\right)(XW_1^V) \\ &= AXW_1^V \end{aligned}$$

The input to the second layer is Z_1, and its output is:

$$Z_2 = \text{Attention}(Q_2, K_2, V_2) = \text{softmax}\left(\frac{Q_2 K_2^\top}{\sqrt{d}}\right) V_2$$

where:

$$Q_2 = Z_1 W_2^Q, \quad K_2 = Z_1 W_2^K, \quad V_2 = Z_1 W_2^V$$

Substituting Z_1 from the first layer:

$$Q_2 = AX W_1^V W_2^Q$$

Similarly:

$$K_2 = AX W_1^V W_2^K$$

$$V_2 = AX W_1^V W_2^V$$

This is equivalent to a single self-attention layer with combined weights:

$$W^Q = W_1^V W_2^Q, \quad W^K = W_1^V W_2^K, \quad W^V = W_1^V W_2^V$$

Now, let's show how this can be reduced to a single self-attention layer. By combining the two layers into one, we observe that:

$$Z_2 = \text{Softmax}\left(\frac{(AX W_1^V W_2^Q)(AX W_1^V W_2^K)^\top}{\sqrt{d_k}}\right) AX W_1^V W_2^V \tag{11.6}$$

$$= \text{Softmax}\left(\frac{(AX W^Q)(AX W^K)^\top}{\sqrt{d_k}}\right) AX W^V \tag{11.7}$$

Therefore, the stacked self-attention layers can be seen as a single self-attention layer with weights that are the product of the individual layers' weights. Adding non-linearity through the use of feed-forward networks (FFN) enables stacking multiple self-attention layers.

Self-Attention Networks Complexity

The complexity of self-attention networks per layer is $O(n^2 d)$ where n is the input sequence length and d is the embedding dimension. The self-attention networks compute the attention weights for each token with respect to every other token. Hence, it is $O(n)$ operations for each token and therefore $O(n^2)$ for all the tokens. Moreover, the complexity of the number of sequential steps is $O(1)$ where all n operations run in a single step (i.e. all the n tokens are processed in parallel).

On the other hand, the complexity per layer is $O(nd^2)$ for RNNs where the previous step's hidden states with the weight matrix multiplication run in d^2 operations (i.e. $O(nd^2)$ for n steps). In addition, the complexity of the number of sequential steps is $O(n)$ where all n operations run in n steps.
The self-attention mechanism has a complexity of $O(n^2)$ for all input tokens, which creates challenges in modeling long-term dependencies in Transformer models. As a result, there is a limitation on the context window size in models like BERT [17] and GPT [38].

11.3.1 Position-wise Feed-Forward Network (FFN)

Stacking self-attention layers without non-linear transformations or feed-forward networks between them results in a single linear transformation. This equivalence follows from the associative property of matrix multiplication and the linearity of the self-attention mechanism.
The Feed-Forward Network (FFN) in Transformers is an essential component within each self-attention block of the model. Each self-attention block of the Transformer architecture consists of a multi-head self-attention mechanism followed by a feed-forward network applied to each position independently. This feed-forward network is responsible for further transforming the input features.
The FFN consists of two linear transformations with a ReLU activation function in between. Given an input vector $\mathbf{x}$, the FFN operation can be summarized as:

$$\mathbf{y} = \mathbf{W}_2 \,\mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2, \tag{11.8}$$

where $\mathbf{x}$ is the input vector of shape (d_{model}), $\mathbf{W}_1$ is the first weight matrix of shape $(d_{\text{model}}, d_{\text{ff}})$, $\mathbf{b}_1$ is the first bias vector of shape (d_{ff}), $\mathbf{W}_2$ is the second weight matrix of shape $(d_{\text{ff}}, d_{\text{model}})$, $\mathbf{b}_2$ is the second bias vector of shape (d_{model}), and $\mathbf{y}$ is the output vector of shape (d_{model}). The $\mathrm{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$ activation function is used for the non-linear transformation. The FFN acts on the output of the attention mechanism, which is in d_{model}. By expanding to d_{ff}, the network can process and refine information more thoroughly before reducing it back to d_{model}[1]. Figure 11.3 illustrates the position-wise FFN implementation.
The term "position-wise Feed-Forward Network (FFN)" is used because the FFN is applied independently to each position in the input sequence. This means that the same FFN is used for every position in the sequence, processing each position's features without considering information from other positions. Equation (11.8) allows the model to learn complex transformations of the input features, enhancing its capability to capture intricate patterns in the data.

[1] Typically, $d_{\text{ff}} = 4d_{\text{model}}$

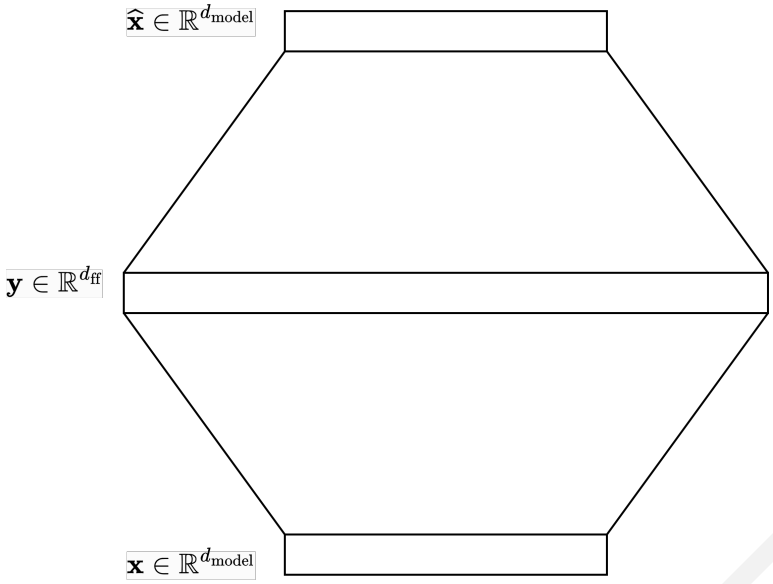


Figure 11.3: The position-wise FFN is implemented by mapping the input vector $\in \mathbb{R}^{d_{\text{model}}}$ to a higher-dimensional space $\in \mathbb{R}^{d_{\text{ff}}}$, applying a non-linear activation function, and then projecting it back to a vector $\in \mathbb{R}^{d_{\text{model}}}$.

11.4 The Transformer Model

The Transformer model [37], revolutionized the field of natural language processing (NLP) by relying entirely on self-attention mechanisms and dispensing with recurrent and convolutional layers. To grasp the Transformer model, it's essential to tackle sequential modeling tasks like language modeling. We'll start by introducing language modeling and then explore the encoder-decoder architecture, including the Transformer.

11.4.1 N-gram Language Modeling

An n-gram language model is a probabilistic model used for predicting the next word in a sequence based on the previous $n-1$ words. An n-gram is simply a contiguous sequence of n words from a given text or speech. The model calculates the probability of a word sequence based on these n-grams.
A bigram language model uses a context of one word to predict the next word. For a sequence of words $w_1, w_2, \ldots, w_m$, the probability of the sequence according to the bigram model is given by:

$$P(w_1, w_2, \ldots, w_m) = P(w_1) \prod_{i=2}^{m} P(w_i \mid w_{i-1}), \tag{11.9}$$

where $P(w_i \mid w_{i-1})$ is the probability of word w_i occurring given the previous word w_{i-1}.
Consider a simple corpus with the following sentences:

1. "I like pizza"
2. "I like pasta"
3. "I eat pizza"

We can compute the bigram probabilities using counts:

$$P(\text{like} \mid \text{I}) = \frac{\text{Count("I like")}}{\text{Count("I")}} = \frac{2}{3}$$

$$P(\text{eat} \mid \text{I}) = \frac{\text{Count("I eat")}}{\text{Count("I")}} = \frac{1}{3}$$

$$P(\text{pizza} \mid \text{like}) = \frac{\text{Count("like pizza")}}{\text{Count("like")}} = \frac{1}{2}$$

Given the sentence "I like pizza", the probability can be computed as:

$$P(\text{"I like pizza"}) = P(\text{I}) \cdot P(\text{like} \mid \text{I}) \cdot P(\text{pizza} \mid \text{like})$$

The probability of a unigram like "I" is:

$$P(\text{I}) = \frac{\text{Count(I)}}{N}$$

where Count(I) is the number of times the word "I" appears in the corpus and N is the total number of words in the corpus.
If $P(\text{I})$ is $\frac{3}{9}$, then:

$$P(\text{"I like pizza"}) = \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{1}{2} = \frac{1}{9}$$

A trigram language model uses a context of two words to predict the next word. For a sequence of words $w_1, w_2, \ldots, w_m$, the probability of the sequence according to the trigram model is given by:

$$P(w_1, w_2, \ldots, w_m) = P(w_1)P(w_2 \mid w_1) \prod_{i=3}^{m} P(w_i \mid w_{i-2}, w_{i-1}), \tag{11.10}$$

where $P(w_i \mid w_{i-2}, w_{i-1})$ is the probability of word w_i occurring given the two preceding words w_{i-2} and w_{i-1}.
Consider the same corpus:

1. "I like pizza"
2. "I like pasta"
3. "I eat pizza"

We can compute the trigram probabilities:

$$P(\text{pizza} \mid \text{I like}) = \frac{\text{Count("I like pizza")}}{\text{Count("I like")}} = \frac{1}{2}$$

$$P(\text{pasta} \mid \text{I like}) = \frac{\text{Count("I like pasta")}}{\text{Count("I like")}} = \frac{1}{2}$$

Given the sentence "I like pizza", the probability can be computed as:

$$P(\text{"I like pizza"}) = P(\text{I}) \cdot P(\text{like} \mid \text{I}) \cdot P(\text{pizza} \mid \text{I like})$$

Using the previous example:

$$P(\text{"I like pizza"}) = \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{1}{2} = \frac{1}{9}$$

For an n-gram language model, the general formula for a sequence $w_1, w_2, \ldots, w_m$ is:

$$P(w_1, w_2, \ldots, w_m) = P(w_1) \cdot P(w_2 \mid w_1) \cdot \ldots \cdot P(w_m \mid w_{m-n+1}, \ldots, w_{m-1}) \quad (11.11)$$

This approach is used to approximate the probability of sequences in natural language processing tasks.

Smoothing in N-gram Language Models

In n-gram language models, smoothing is a technique used to handle the problem of zero probabilities for unseen n-grams. When building a language model, we often encounter n-grams that do not appear in the training data, resulting in a zero probability for those n-grams. This can be problematic, especially for sequences that are rare or entirely absent in the training corpus.
Smoothing adjusts the estimated probabilities to account for these unseen events, ensuring that no n-gram has a zero probability.

Laplace smoothing, also known as "add-one smoothing", is a simple technique that adds a small amount (usually 1) to each count to prevent zero probabilities. For a bigram language model, suppose we want to estimate the probability $P(w_i \mid w_{i-1})$, where w_i is the current word and w_{i-1} is the preceding word.
Without smoothing, the maximum likelihood estimate (MLE) of the bigram probability is:

$$P(w_i \mid w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})},$$

where:

- $\text{Count}(w_{i-1}, w_i)$ is the number of times the bigram (w_{i-1}, w_i) appears in the training data.
- $\text{Count}(w_{i-1})$ is the number of times the word w_{i-1} appears in the training data.

Laplace smoothing modifies this estimate by adding 1 to all bigram counts and adjusting the denominator accordingly:

$$P_{\text{Laplace}}(w_i \mid w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i) + 1}{\text{Count}(w_{i-1}) + V}$$

where V is the size of the vocabulary (total number of unique words in the training data). This modification ensures that every bigram has a non-zero probability, even if it did not appear in the training data.
Consider a small corpus:

1. "I like pizza"
2. "I like pasta"
3. "I eat pizza"

Assume our vocabulary $V = \{\text{I}, \text{like}, \text{eat}, \text{pizza}, \text{pasta}\}$, so $V = 5$. Let's compute the probability $P(\text{pizza} \mid \text{like})$ with Laplace smoothing:

$$P_{\text{Laplace}}(\text{pizza} \mid \text{like}) = \frac{\text{Count(like, pizza)} + 1}{\text{Count(like)} + V} = \frac{1+1}{2+5} = \frac{2}{7}$$

Without smoothing, if we had an unseen bigram, its probability would be zero, which Laplace smoothing avoids.
Given an n-gram model, the general formula for Laplace smoothing is :

$$P_{\text{Laplace}}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \frac{\text{Count}(w_{i-n+1}, \ldots, w_i) + 1}{\text{Count}(w_{i-n+1}, \ldots, w_{i-1}) + V}, \tag{11.12}$$

where Count($w_{i-n+1}, \ldots, w_i$) is the count of the n-gram, Count($w_{i-n+1}, \ldots, w_{i-1}$) is the count of the (n-1)-gram preceding w_i, and V is the size of the vocabulary.
While Laplace smoothing is straightforward, it is not always the most effective method because it adds the same amount (1) to all counts, regardless of their frequency. This approach can overly penalize more frequent n-grams, leading to less accurate probability estimates. Other smoothing techniques, such as Good-Turing smoothing [39] or Kneser-Ney smoothing [40, 41], are often preferred in practice.

11.4.2 Neural Language Modeling

A "neural language model" uses neural networks to predict the probability of a sequence of words in a sentence. Unlike traditional n-gram models, neural language models can handle larger contexts and capture more complex patterns in the data by using word embeddings and neural networks to represent and learn relationships between words.
Given a sequence of words $w_1, w_2, \ldots, w_{n-1}$, the goal is to predict the next word w_n using a neural network. The probability of a sequence of words $w_1, w_2, \ldots, w_n$ is defined as:

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_1, w_2) \cdot \ldots \cdot P(w_n \mid w_1, w_2, \ldots, w_{n-1}) \tag{11.13}$$

A neural language model aims to compute each conditional probability $P(w_i \mid w_1, w_2, \ldots, w_{i-1})$ using a neural network.

A Feed-Forward Language Model

The most common architecture for a neural language model is a feedforward neural network or a recurrent neural network (RNN). Let's describe a simple feedforward neural language model:

1. Word Embedding Layer: Convert each word into a fixed-size vector representation (embedding) using an embedding matrix. Suppose we have a vocabulary of size V and each word is represented by an embedding of size d. The embedding matrix E is of size $V \times d$.

   For a context window of size 3, given words w_1, w_2, w_3, we first obtain their embeddings:

   $$\mathbf{e}_1 = E[w_1], \quad \mathbf{e}_2 = E[w_2], \quad \mathbf{e}_3 = E[w_3]$$

   where $\mathbf{e}_i \in \mathbb{R}^d$.

2. Hidden Layer: Use a neural network layer with non-linear activation (e.g., ReLU or sigmoid) to process the concatenated embeddings. We start by concatenating the embeddings to form a single input vector:

$$\mathbf{x} = [\mathbf{e}_1; \mathbf{e}_2; \mathbf{e}_3] \in \mathbb{R}^{3d}$$

Apply a hidden layer transformation:

$$\mathbf{h} = \tanh(\mathbf{W}_h \mathbf{x} + \mathbf{b}_h)$$

where $\mathbf{W}_h \in \mathbb{R}^{m \times 3d}$ is the weight matrix for the hidden layer, $\mathbf{b}_h \in \mathbb{R}^m$ is the bias vector, and $\mathbf{h} \in \mathbb{R}^m$ is the hidden layer output.

3. Output Layer: Apply a softmax function to compute the probability distribution over the vocabulary for the next word. Compute the scores for each word in the vocabulary:

$$\mathbf{z} = \mathbf{W}_o \mathbf{h} + \mathbf{b}_o,$$

where $\mathbf{W}_o \in \mathbb{R}^{V \times m}$ is the output weight matrix, $\mathbf{b}_o \in \mathbb{R}^V$ is the output bias vector, and $\mathbf{z} \in \mathbb{R}^V$ are the scores for each word in the vocabulary.

Apply the softmax function to get the probability distribution over the vocabulary:

$$P(w_{n+1} \mid w_1, w_2, \ldots, w_n) = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)}$$

where z_i is the score corresponding to the word w_i.

Let's walk through a numerical example with a simple neural language model. The vocabulary size $V = 5$ (words: w_1, w_2, w_3, w_4, w_5), embedding dimension $d = 2$, and hidden layer size $m = 3$.
Let the embedding matrix E be:

$$E = \begin{bmatrix} 0.1 & 0.3 \\ 0.4 & 0.2 \\ 0.5 & 0.8 \\ 0.6 & 0.9 \\ 0.7 & 0.1 \end{bmatrix}$$

For the input context (w_2, w_3, w_4), the embeddings are:

$$\mathbf{e}_1 = E[w_2] = [0.4, 0.2], \quad \mathbf{e}_2 = E[w_3] = [0.5, 0.8], \quad \mathbf{e}_3 = E[w_4] = [0.6, 0.9]$$

Concatenate the embeddings:

$$\mathbf{x} = [\mathbf{e}_1; \mathbf{e}_2; \mathbf{e}_3] = [0.4, 0.2, 0.5, 0.8, 0.6, 0.9]$$

Suppose the weight matrix $\mathbf{W}_h$ and bias vector $\mathbf{b}_h$ are:

$$\mathbf{W}_h = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 & 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 \end{bmatrix}, \quad \mathbf{b}_h = [0.1, 0.2, 0.3]$$

Compute the hidden layer output:

$$\mathbf{h} = \tanh(\mathbf{W}_h \mathbf{x} + \mathbf{b}_h)$$

Plugging in the values:

$$\mathbf{W}_h \mathbf{x} = \begin{bmatrix} (0.1 \cdot 0.4 + 0.2 \cdot 0.2 + 0.3 \cdot 0.5 + 0.4 \cdot 0.8 + 0.5 \cdot 0.6 + 0.6 \cdot 0.9) \\ (0.7 \cdot 0.4 + 0.8 \cdot 0.2 + 0.9 \cdot 0.5 + 0.1 \cdot 0.8 + 0.2 \cdot 0.6 + 0.3 \cdot 0.9) \\ (0.4 \cdot 0.4 + 0.5 \cdot 0.2 + 0.6 \cdot 0.5 + 0.7 \cdot 0.8 + 0.8 \cdot 0.6 + 0.9 \cdot 0.9) \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}$$

Calculate:

$$\mathbf{W}_h \mathbf{x} \approx \begin{bmatrix} 1.49 \\ 1.56 \\ 2.71 \end{bmatrix}, \quad \mathbf{h} = \tanh([1.49, 1.56, 2.71]) \approx [0.90, 0.91, 0.99]$$

Assume:

$$\mathbf{W}_o = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \\ 1.0 & 1.1 & 1.2 \\ 1.3 & 1.4 & 1.5 \end{bmatrix}, \quad \mathbf{b}_o = [0.1, 0.1, 0.1, 0.1, 0.1]$$

Compute the output scores:

$$\mathbf{z} = \mathbf{W}_o \mathbf{h} + \mathbf{b}_o \approx [0.669, 1.509, 2.349, 3.189, 4.029]$$

Apply the softmax function:

$$P(w_i \mid w_2, w_3, w_4) = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)}$$

For example:

$$P(w_1) = \frac{\exp(0.669)}{\exp(0.669) + \exp(1.509) + \exp(2.349) + \exp(3.189) + \exp(4.029)} \approx 0.020$$

This is a simplified example to illustrate the mechanics of neural language modeling. The key advantage is that neural models, especially those with recurrent or self-attention mechanisms, can capture longer-range dependencies in text compared to traditional n-gram models.

An RNN Language Model

RNNs are particularly useful for modeling sequential data because they maintain a hidden state that captures information about the past inputs. Hence, they are commonly used for language modeling.
An "RNN language model" uses the hidden state of the network to predict the next word in a sequence. Given a sequence of words $w_1, w_2, \ldots, w_n$, the RNN computes the probability of each word given the previous words:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, w_2, \ldots, w_{i-1})$$

The hidden state at each step t is updated based on the current input word and the previous hidden state:

$$h_t = f(W_h \cdot x_t + U_h \cdot h_{t-1} + b_h),$$

where $h_t \in \mathbb{R}^m$ is the hidden state at time step t, $x_t \in \mathbb{R}^d$ is the embedding of the input word w_t, $W_h \in \mathbb{R}^{m \times d}$ is the weight matrix for the input, $U_h \in \mathbb{R}^{m \times m}$ is the weight matrix for the hidden state, $b_h \in \mathbb{R}^m$ is the bias vector, and $f(\cdot)$ is a non-linear activation function (e.g., tanh).
The output layer computes the probability of the next word:

$$o_t = V \cdot h_t + b_o$$

$$P(w_{t+1} \mid w_1, w_2, \ldots, w_t) = \text{softmax}(o_t)$$

where $V \in \mathbb{R}^{V \times m}$ is the weight matrix for the output layer, and $b_o \in \mathbb{R}^V$ is the bias vector for the output layer.

Masked Self-Attention Language Model

A "Masked Self-Attention Based Language Model" predicts the next word in a sequence by leveraging self-attention mechanisms that are restricted to focus only on the preceding words in the sequence. This approach ensures that the model does not have access to "future" words when predicting the next word, which is crucial for causal (auto-regressive) language modeling. The key steps and equations for this process are described below:
Given an input sequence of words $[w_1, w_2, \ldots, w_n]$, the first step is to convert these words into their corresponding embeddings. Let:

$$X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{n \times d}$$

where $x_i \in \mathbb{R}^d$ is the embedding of word w_i and d is the dimensionality of the embedding.
For each word w_i, we compute the masked self-attention score by masking the future words (i.e., setting the attention weights for future words to zero). Let Q, K, and V represent the Query, Key, and Value matrices:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ are learnable weight matrices and d_k is the dimensionality of the Key/Query space. The attention scores are computed using the scaled dot-product:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

where $M \in \mathbb{R}^{T \times T}$ is a masking matrix such that $M_{ij} = -\infty$ if $j > i$ (to prevent attending to future words), and $M_{ij} = 0$ otherwise. Please note that the Softmax is applied row-wise.
The attention weights are computed as:

$$A = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)$$

So the masked self-attention output Z is:

$$Z = AV = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

The output Z of the self-attention layer is then used to predict the next word. For next-word prediction, a linear layer followed by a softmax function is used. Let $W_{\text{out}} \in \mathbb{R}^{d \times V}$ be the output weight matrix where V is the vocabulary size. The final

softmax layer gives the probability of each word in the vocabulary being the next word in the sequence:

$$P(v) = \frac{\exp(ZW_{\text{out}})}{\sum_{j=1}^{V} \exp(ZW_{\text{out}})}$$

This is a basic application of masked self-attention for language modeling. In the following section, we will explore the Transformer model, which is designed for sequence-to-sequence tasks. The Transformer consists of an encoder and a decoder and serves as an example of conditional language modeling.

11.4.3 Conditional Language Modeling

Conditional language modeling involves predicting the next word in a sequence given both the previous context and an additional condition, such as another sequence. Mathematically, this can be represented as finding the probability of a target sequence $Y = (y_1, y_2, \ldots, y_n)$ given a source sequence $X = (x_1, x_2, \ldots, x_m)$. The objective is to maximize the conditional probability:

$$P(Y \mid X) = P(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_m)$$

Using the chain rule, this can be decomposed as:

$$P(Y \mid X) = \prod_{t=1}^{n} P(y_t \mid y_1, y_2, \ldots, y_{t-1}, X)$$

Here, each word y_t in the target sequence Y is predicted based on both the preceding words in Y and the entire source sequence X.

An elementary encoder-decoder model [42] utilizes an RNN to encode the input sequence into a hidden representation. The final hidden state of the encoder is then passed to the decoder as its initial state. The decoder, which is also an RNN, generates the output sequence step by step. Each output is conditioned on the previous output and the last hidden state from the encoder. This structure allows the decoder to leverage both the encoded input and prior generated tokens. The encoder-decoder pair works together to produce the final output sequence as shown in Figure 11.4.

When the input sequence is particularly long, the encoder must compress all the information into a single vector, which increases the likelihood of losing important details. As the encoder tries to capture everything in one vector, it can struggle to retain the full context of the sequence. This often leads to the model forgetting critical information from earlier in the input. This issue is especially prominent in RNN-based encoder-decoder models. The bottleneck of relying on one hidden state to store all information results in degraded performance on longer sequences.

Addressing this limitation is crucial for improving the model's accuracy on complex, lengthy inputs.

A solution to the issue of forgetting in encoder-decoder models is to introduce a cross-attention layer between the encoder and decoder. This allows the decoder to focus on specific tokens from the source sequence that are most important for each step of the output. At every generation step, the decoder can attend to different parts of the input sequence, ensuring it captures relevant information. By examining the attention weights, we can see which tokens the decoder emphasizes during its decision-making process. This mechanism helps the decoder leverage important source tokens dynamically. Overall, cross-attention improves the model's ability to handle longer and more complex sequences. We will explore cross-attention in more detail when discussing the Transformer model, where it plays a key role in allowing the decoder to attend to the encoder's output more effectively at each generation step.

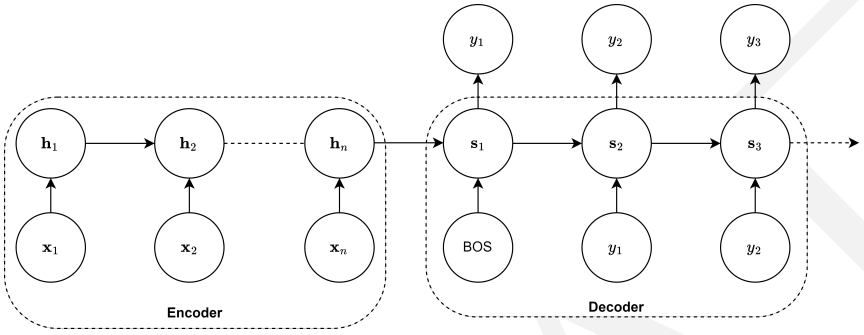


Figure 11.4: An elementary encoder-decoder model uses an RNN to transform the input sequence into a hidden representation. The encoder's final hidden state is transferred to the decoder as its starting point. The decoder, also an RNN, generates the output sequence in a step-by-step manner, with each output relying on the previous one and the encoder's last hidden state. This setup enables the decoder to incorporate both the encoded input and previously generated outputs to produce the final sequence.

The Transformer model is an example of a conditional language model that uses self-attention mechanisms to capture dependencies in both the source sequence X and the target sequence Y. The Transformer consists of two main parts: an encoder and a decoder.

The encoder processes the source sequence X and outputs a sequence of context-aware representations. Let's denote the encoder's input embeddings as $\mathbf{E}^X = (e_1^X, e_2^X, \ldots, e_m^X)$. The encoder applies multiple layers of self-attention and feed-forward networks to transform these embeddings into hidden states:

$$\mathbf{H}^X = \text{Encoder}(\mathbf{E}^X)$$

where $\mathbf{H}^X = (h_1^X, h_2^X, \ldots, h_m^X)$ are the context-aware representations of the input sequence.

The decoder generates the target sequence Y by predicting each word y_t based on its own previous words and the encoded representations from the source sequence. The decoder takes two inputs: the encoder's outputs $\mathbf{H}^X$ and the embeddings of the target sequence up to step $t-1$, denoted by $\mathbf{E}_{<t}^Y = (e_1^Y, e_2^Y, \ldots, e_{t-1}^Y)$.

The decoder also consists of multiple layers of self-attention and feedforward networks, and its output at step t is:

$$\mathbf{H}_t^Y = \text{Decoder}(\mathbf{E}_{<t}^Y, \mathbf{H}^X)$$

where $\mathbf{H}_t^Y$ is the hidden state for time step t. Finally, the conditional probability of the next word y_t is computed as:

$$P(y_t \mid y_1, y_2, \ldots, y_{t-1}, X) = \text{softmax}(\mathbf{W_o}\mathbf{H}_t^Y)$$

where $\mathbf{W_o}$ is the output projection matrix.

For example, consider a translation task where we want to translate the English sentence "I love you" (source sequence X) into French ("Je t'aime"). The encoder processes the source sentence to create context-aware representations. Then, the decoder starts generating the target sequence "Je", "t'", "aime" step-by-step. At each step t, the decoder uses both the previously generated words and the encoder's output to predict the next word until the full translation is generated.

The Transformer Architecture

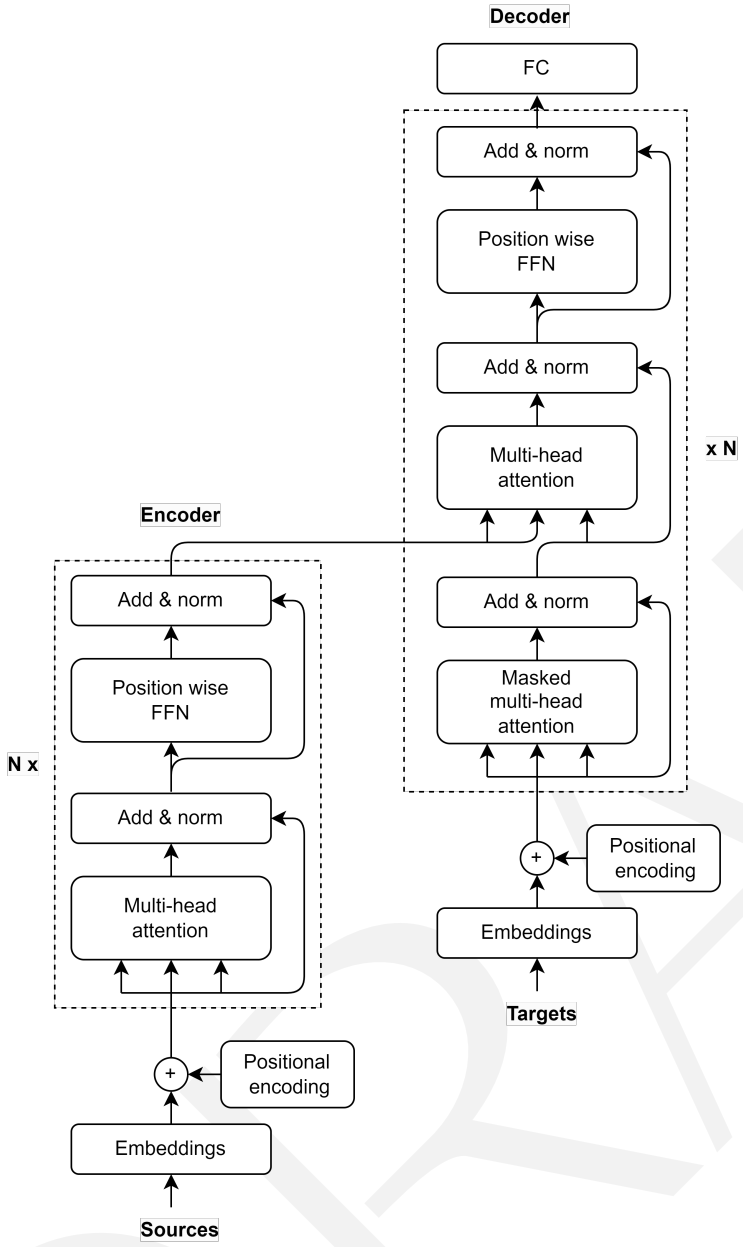


Figure 11.5: The Transformer encoder-decoder model consists of two main components: an encoder that processes input sequences using self-attention and feed-forward networks, and a decoder that generates output sequences while utilizing masked self-attention and encoder-decoder attention (i.e. cross-attention). Both the encoder and decoder are composed of multiple stacked layers, with positional encodings added to embeddings to maintain token order.

The Transformer model consists of an encoder-decoder architecture as shown in Figure 11.5. Both the encoder and decoder are composed of multiple identical layers. Since the Transformer model doesn't have any recurrence or convolution, it lacks a way to handle the order of the sequence. To overcome this, positional encodings are added to the input embeddings to give the model information about the position of the tokens in the sequence[2]:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$

where $\mathrm{PE}(pos, i)$ is the positional encoding at position pos and dimension i, d_{model} is the dimensionality of the model, pos is the position of the token in the sequence, and i is the dimension index (0 to $d_{model} - 1$). Figure 11.6 illustrates how sequence order information is incorporated into the token embeddings to produce the final sequence embeddings used by the Transformer model. Figure 11.7, presents an example of sinusoidal positional embeddings. These embeddings use sine and cosine functions to encode position information into the sequence. The sinusoidal functions ensure that each position has a unique representation. This approach helps maintain sequence order in the model's input.

[2]For more information about positional encoding, please check this page https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

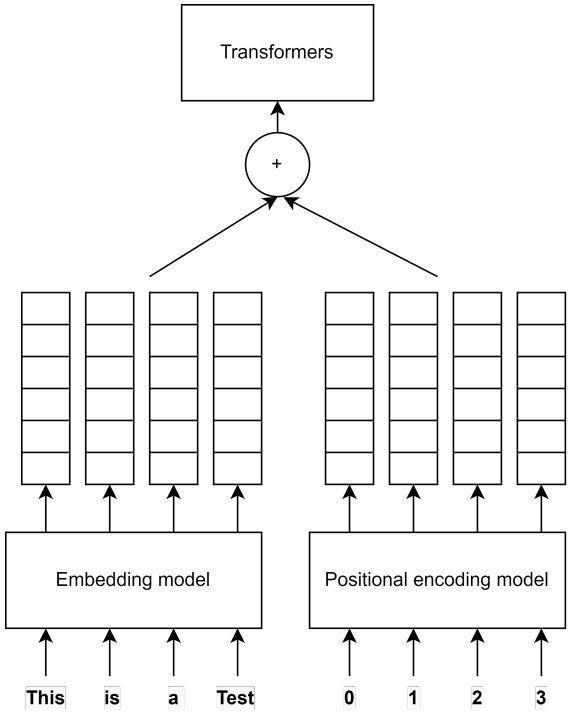


Figure 11.6: Positional encoding is used to incorporate sequence order information, as Transformers do not inherently capture the order of elements in a sequence.

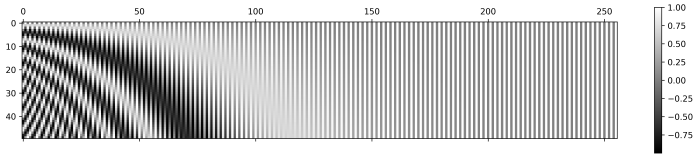


Figure 11.7: The sinusoidal positional embedding matrix represents a sequence of length 50 with a dimensionality of 256. Each position in the sequence is uniquely encoded using sine and cosine functions. These embeddings provide position-related information to the model, helping it understand the order of elements in the sequence. The resulting matrix has 50 rows, each containing a 256-dimensional embedding that corresponds to a specific position in the sequence.

The Transformer architecture given by

- Encoder

Input Embedding + Positional Encoding
for each layer l :
$\hat{X}^l = \text{LayerNorm}(X^{l-1} + \text{MultiHead}(X^{l-1}, X^{l-1}, X^{l-1}))$
$X^l = \text{LayerNorm}(\hat{X}^l + \text{FFN}(\hat{X}^l))$
Output of Encoder $= X^N$

- Decoder

Input Embedding + Positional Encoding
for each layer l :
$\hat{Y}^l = \text{LayerNorm}(Y^{l-1} + \text{MaskedMultiHead}(Y^{l-1}, Y^{l-1}, Y^{l-1}))$
$\tilde{Y}^l = \text{LayerNorm}(\hat{Y}^l + \text{MultiHead}(\hat{Y}^l, X^N, X^N))$
$Y^l = \text{LayerNorm}(\tilde{Y}^l + \text{FFN}(\tilde{Y}^l))$
Output of Decoder $= Y^N$

- Final Linear and Softmax Layer

  The output of the decoder is transformed into the final output probabilities through a linear layer followed by a softmax function:

$$\text{Output Probabilities} = \text{softmax}(Y^N W_{\text{final}} + b_{\text{final}})$$

The architecture has three new ideas:

1. Add & Norm: involves applying a residual connection followed by layer normalization. It is given by:

$$\text{LayerNorm}(X + \text{SubLayer}(X)) \tag{11.14}$$

   Residual connections, also known as skip connections shown in Figure 11.8, are a crucial component in deep neural networks, including the Transformer architecture. They were introduced by He et al. in their work on ResNet [43]. In general, gradients can become extremely small during backpropagation in very deep networks, making it difficult for the network to learn. This is known as the vanishing gradient problem. Residual connections allow gradients to flow more directly through the network, helping to mitigate this issue. By providing an alternative path for the gradient, they ensure that the learning signal can reach earlier layers more effectively. Moreover, the Transformers rely on positional encodings to maintain the order of the sequence. Hence,

residual connections help preserve this positional information throughout the network.

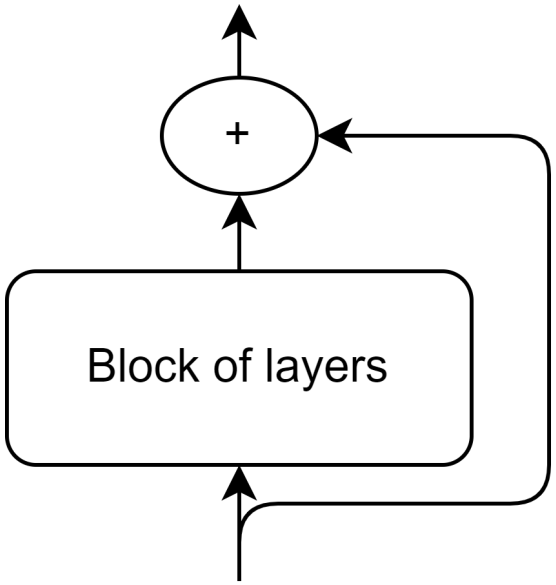


Figure 11.8: The residual connections allow the original input to bypass one or more layers and then add it back to the output of those layers, which helps mitigate the problem of vanishing gradients and improves model convergence.

To demonstrate the mathematical utility of residual connections, let's calculate the derivative with respect to the input of a residual network. We start with the loss function E defined over the output $\mathbf{y}$. Let's denote the loss as a function of the output:

$$E = E(\mathbf{y}, \mathbf{y}_{\text{true}}), \tag{11.15}$$

where $\mathbf{y}_{\text{true}}$ is the true output.

Given the residual network defined as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \mathbf{W}) + \mathbf{x} \tag{11.16}$$

We can apply the chain rule to compute the gradient of the loss with respect to the input $\mathbf{x}$:

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \tag{11.17}$$

Hence,

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{J}_{\mathcal{F}} + \mathbf{I}, \tag{11.18}$$

where $\mathbf{J}_{\mathcal{F}} = \frac{\partial \mathcal{F}(\mathbf{x},\mathbf{W})}{\partial \mathbf{x}}$ is the Jacobian matrix of the residual function and $\mathbf{I}$ is the identity matrix.

Thus, we can rewrite the derivative of the loss with respect to the input as:

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \cdot (\mathbf{J}_{\mathcal{F}} + \mathbf{I}) \tag{11.19}$$

This equation highlights how the gradients from the loss propagate through both the residual function and the identity mapping, improving the flow of gradients during backpropagation in deep networks.

2. In Layer Normalization[3], each input x is normalized by subtracting the mean and dividing by the standard deviation, and then scaled and shifted by learnable parameters γ and β.

   Given an input vector x with dimensionality d:

   Compute the Mean:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

   Compute the Variance:

$$\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$$

[3]Another form of normalization is Batch Normalization (BN) which normalizes the input to a layer over a mini-batch of data. This is done by using the mean and variance statistics computed from the mini-batch during training. This normalization is given by:

$$\hat{x}_i = \frac{x_i - \frac{1}{m}\sum_{i=1}^{m} x_i}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 + \epsilon}},$$

where m is the batch size, and x_i are the inputs in the batch. Since BN relies on batch statistics, it can be problematic in sequence models due to variable sequence lengths and smaller batch sizes for longer sequences. This leads to noisy estimates of the mean and variance, resulting in less effective normalization. On the other hand, layer normalization is more suited for sequence models because it normalizes across the features for each sequence element independently, avoiding issues related to batch statistics.

Normalize the Input:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Here, ϵ is a small constant added to the variance to avoid division by zero.

Scale and Shift:

$$y_i = \gamma \hat{x}_i + \beta$$

Here, γ and β are learnable parameters. After normalization ($\hat{x}_i$), the normalized values $\hat{x}_i$ have a mean of 0 and a variance of 1. The final output $y_i = \gamma \hat{x}_i + \beta$ will have the mean and variance determined by γ and β. Specifically: The scaling by γ and shifting by β do not directly enforce a specific mean or variance on y. Instead, they allow the model to adjust the normalized output flexibly.

3. Cross-attention (i.e. Encoder-Decoder Attention): This layer allows the decoder to focus on appropriate parts of the input sequence, using the encoder's output. It was given by

$$\text{MultiHead}(\hat{Y}^l, X^N, X^N)),$$

where the K and V are from one sequence and Q is from another sequence. It was originally introduced in [44, 45] for a machine translation application but it can be used for other applications where the encoder is from speech or image. At different stages, the decoder may need to focus on various source tokens that are most relevant to that particular step. By looking at the attention weights, we can identify which source tokens the decoder uses, as illustrated in Figure 11.9. The two examples demonstrate that the attention mechanism has established a (soft) alignment between source and target tokens, showing that the decoder concentrates on the source tokens being translated at that moment.

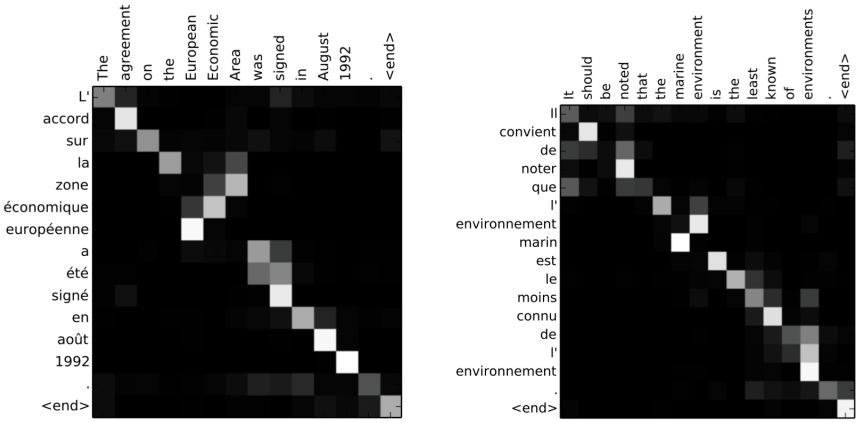


Figure 11.9: The attention weights in the cross-attention module can be used to identify the alignments between target tokens and source tokens. In this example, we have one attention head, and an attention score is calculated by applying a multi-layer perceptron (MLP) to the encoder and decoder state vectors. Specifically, the score is computed as $\text{score}(\mathbf{q}, \mathbf{k}) = \mathbf{w}_2^T \tanh(\mathbf{W}_1[\mathbf{q}, \mathbf{k}])$ [44]. This is referred to as Bahdanau's attention scoring function, while in Transformers, we utilize a scaled dot-product attention scoring function.

This detailed architecture highlights the key components and processes involved in the Transformer model, showcasing its ability to effectively handle long-range dependencies in sequences through self-attention mechanisms.
Attention mechanisms and Recurrent Neural Networks (RNNs) are two powerful approaches in the field of deep learning, particularly for sequence modeling tasks such as natural language processing. Table 11.1, shows a detailed comparison of attention mechanisms and RNNs:

11.5 Training and Inference for Encoder-Decoder Framework

In a Transformer model, the training objective is typically designed for sequence-to-sequence tasks such as machine translation, text generation, or other conditional language modeling tasks. The objective is to minimize the difference between the predicted output sequence and the ground truth sequence.

11.5.1 Cross-Entropy Loss for Transformer Models

Given a dataset with input sequences X and corresponding target sequences Y, the model is trained to minimize the cross-entropy loss between the predicted probability distribution and the true distribution over possible output tokens.

Let $X = (x_1, x_2, \ldots, x_m)$ represent the input sequence and $Y = (y_1, y_2, \ldots, y_n)$ represent the corresponding target sequence, where n is the length of the sequence and y_i represents the token at position i in the target sequence.
The Transformer predicts the probability distribution over the vocabulary $P(y_i \mid X, y_{1:i-1})$ for each time step i by using the self-attention mechanism and feedforward neural networks. The goal is to maximize the likelihood of the correct sequence Y given the input sequence X.
The training objective can be formulated as minimizing the cross-entropy loss:

$$E = -\sum_{i=1}^{n} \log P(y_i \mid X, y_{1:i-1}), \tag{11.20}$$

where $P(y_i \mid X, y_{1:i-1})$ is the probability predicted by the model for the token y_i at position i, conditioned on the input sequence X and the previously generated tokens $y_{1:i-1}$ and n is the length of the target sequence.
The model is trained using backpropagation to minimize the cross-entropy loss E. The gradients with respect to the model parameters (e.g., weights in the attention layers and feedforward networks) are computed and updated using an optimizer like Adam.
In summary, the training objective for a Transformer model is to minimize the cross-entropy loss over the predicted probability distributions for the target sequence, ensuring that the model generates the most likely correct sequence given the input.

11.5.2 Inference for Transformer Models

The transformer model uses either greedy search or beam search during inference. Greedy search selects the highest probability token at each step to generate the sequence, which is fast but may not yield the best result. Beam search, on the other hand, explores multiple possible sequences by keeping the top-k candidates at each step, leading to better-quality results. However, beam search is more computationally intensive compared to greedy search. Inference in the model halts when it reaches a predetermined maximum number of steps. Alternatively, the generation can also terminate when the model outputs the "EOS" token, signaling the end of the sequence. This mechanism ensures that the output remains concise and relevant. By using these criteria, the model effectively manages the length of the generated sequences.
In Encoder-Decoder or sequence-to-sequence (Seq2Seq) models, given an input sequence $X = [x_1, x_2, \ldots, x_m]$, the goal is to generate an output sequence $Y = [y_1, y_2, \ldots, y_n]$. Greedy search is the simplest approach where, at each time step, the word with the highest probability is chosen. This can be represented mathematically as:

$$y_i = \arg\max_{y_i} P(y_i \mid y_1, y_2, ..., y_{i-1}, X) \tag{11.21}$$

The probability $P(y_i \mid y_1, ..., y_{i-1}, X)$ is computed by the decoder at each time step. This approach may lead to suboptimal solutions because it doesn't explore other sequences that may have higher overall probability.

Beam search is an improvement over greedy search. Instead of selecting the best word at each step, beam search maintains the top-k candidate sequences (beam width) at each time step. At every time step, for each of the top-k candidates, it expands the next possible tokens and selects the top-k from the expanded list based on the cumulative probability. Figure 11.10 depicts the beam search algorithm with a beam width of 2 and a sequence length of 3. It shows how the algorithm explores potential sequences. The limited width allows for a focused search. Overall, the sequence length is set to 3, indicating the number of steps in the search process.

Let $S^{(l)} = [y_1^{(l)}, y_2^{(l)}, ..., y_i^{(l)}]$ represent the candidate sequences at time step i, where l is the index for beam search (i.e., there are k candidate sequences). The beam search objective is to maximize:

$$S^{(i)} = \arg\max_{S^{(i)}} \sum_{j=1}^{i} \log P(y_j^{(l)} \mid y_1^{(l)}, y_2^{(l)}, \ldots, y_{j-1}^{(l)}, X) \tag{11.22}$$

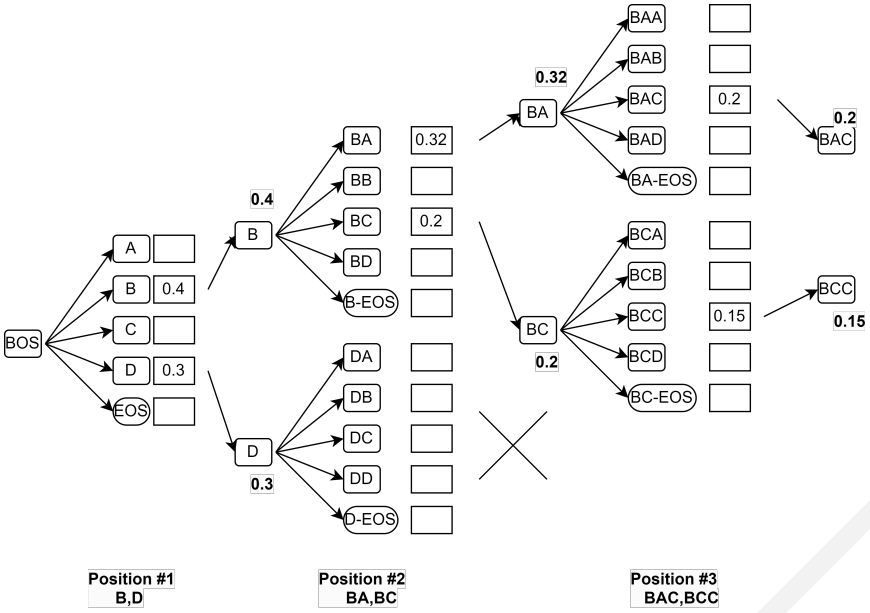


Figure 11.10: An illustration of a beam search algorithm featuring a sequence length of 3 and a beam width of 2.

Greedy search can get stuck in local optima because it only considers the immediate next step. Beam search mitigates this by maintaining multiple candidates at each step, but at the cost of increased computation. A comparison between the greedy search and beam search is summarized in Table 11.2.

11.6 Assignment

Implement a neural machine translation algorithm using the Transformer model using the Keras deep learning library and Google colab. To load the dataset, please visit https://github.com/SamirMoustafa/nmt-with-attention-for-ar-to-en/blob/master/ara_.txt.

Aspect	Recurrent Neural Networks (RNNs)	Attention Mechanisms
Overview	- Sequential Processing: Processes input sequences one element at a time. - Recurrence: Maintains a hidden state capturing information about previous elements. - Variants: Includes LSTMs and GRUs for long-range dependencies.	- Direct Interaction: Accesses all elements of the input sequence at each time step. - Weighted Sum: Computes a weighted sum of input elements using a learned attention mechanism.
Pros	1. Naturally suited for temporal sequences. 2. Encodes entire sequence into a compact hidden state.	1. Fully parallelizable, leading to faster training times. 2. Handles long-range dependencies effectively. 3. Provides insights into model focus, enhancing interpretability.
Cons	1. Cannot be easily parallelized; slow training. 2. Challenges with long-range dependencies. 3. Susceptible to gradient issues (vanishing/exploding).	1. Computationally expensive for long sequences. 2. High memory usage for storing attention weights.
Performance and Efficiency	- Slower training speed due to sequential computation. - More compact memory footprint for hidden states.	- Much faster training due to parallelization. - Requires more memory for attention weights.
Flexibility and Usability	- Good for sequential data applications. - Limited interpretability compared to attention mechanisms.	- Contextually aware of the entire input sequence. - Natural way to visualize model focus, aiding interpretability.
Conclusion	- Suited for tasks where sequential data is crucial, but struggles with long-range dependencies.	- Excel at capturing long-range dependencies and context, becoming the go-to approach for many NLP tasks.

Table 11.1: Comparison of Attention Mechanisms and RNNs.

Table 11.2: Comparison of Greedy Search and Beam Search

Method	Pros	Cons
Greedy Search	Simple, fast, and computationally efficient.	Suboptimal in many cases since it doesn't explore alternative sequences.
Beam Search	Explores multiple possible sequences, providing better quality results.	More computationally expensive; still not guaranteed to find the global optimum.

12. State Space Models

Finally, we make some remarks on why linear systems are so important. The answer is simple: because we can solve them!

— Richard Feynman

State Space Models (SSMs) employ linear recurrence mechanisms, a characteristic that contrasts with the non-linear recurrent structures found in the Recurrent Neural Networks (RNNs) described in Chapter 10. In this chapter, we explore structured state space models (S4) and selective state space models (S6). These models enhance the ability to capture long-range dependencies in time-series and text data.

12.1 Discrete-Time State Space Model

The Structured State Space Sequence Model (S4) is a deep learning architecture designed for efficient long-sequence modeling. It combines continuous-time state space models (SSMs) with structured parameterizations and HiPPO (High-order Polynomial Projection Operators) theory. The core of S4 is a linear time-invariant (LTI) system described by:

$$\frac{dh(t)}{dt} = Ah(t) + Bx(t) \tag{12.1}$$

$$y(t) = Ch(t) + Dx(t), \tag{12.2}$$

where:
- $h(t) \in \mathbb{R}^N$ is the hidden state at time t,
- $x(t) \in \mathbb{R}$ is the input at time t,
- $y(t) \in \mathbb{R}$ is the output at time t,
- $A \in \mathbb{R}^{N \times N}$ is the state transition matrix,
- $B \in \mathbb{R}^{N \times 1}$ is the input matrix,

- $C \in \mathbb{R}^{1\times N}$ is the output matrix,
- $D \in \mathbb{R}$ is the skip connection (optional).

To be able to use a discrete input sequence, the bilinear transformation discretizes the continuous-time state-space model:

$$\frac{dh(t)}{dt} \approx \frac{h_k - h_{k-1}}{\Delta}, \tag{12.3}$$

where Δ is the discretization step size (sampling interval).
Substituting this into the state equation:

$$\frac{h_k - h_{k-1}}{\Delta} = A\left(\frac{h_k + h_{k-1}}{2}\right) + B\left(\frac{x_k + x_{k-1}}{2}\right) \tag{12.4}$$

Rearranging terms to solve for h_k:

$$h_k - h_{k-1} = \frac{\Delta}{2}A(h_k + h_{k-1}) + \frac{\Delta}{2}B(x_k + x_{k-1}) \tag{12.5}$$

Collecting h_k terms on the left-hand side:

$$h_k - \frac{\Delta}{2}Ah_k = h_{k-1} + \frac{\Delta}{2}Ah_{k-1} + \frac{\Delta}{2}B(x_k + x_{k-1}) \tag{12.6}$$

Factor out h_k:

$$\left(I - \frac{\Delta}{2}A\right)h_k = \left(I + \frac{\Delta}{2}A\right)h_{k-1} + \frac{\Delta}{2}B(x_k + x_{k-1}) \tag{12.7}$$

Solving for h_k:

$$h_k = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(I + \frac{\Delta}{2}A\right)h_{k-1} + \left(I - \frac{\Delta}{2}A\right)^{-1}\frac{\Delta}{2}B(x_k + x_{k-1}) \tag{12.8}$$

We approximate $x_k \approx x_{k-1}$:

$$h_k = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(I + \frac{\Delta}{2}A\right)h_{k-1} + \left(I - \frac{\Delta}{2}A\right)^{-1}\Delta Bx_k \tag{12.9}$$

Hence:

$$h_k = \overline{A}h_{k-1} + \overline{B}x_k, \tag{12.10}$$

and

$$y_k = \overline{C}h_k + Dx_k, \tag{12.11}$$

where $\overline{A} = \left(I - \frac{\Delta}{2}A\right)^{-1}\left(I + \frac{\Delta}{2}A\right)$, $\overline{B} = \left(I - \frac{\Delta}{2}A\right)^{-1}\Delta B$ and $\overline{C} = C$.

In addition to bilinear discretization, the zero-order hold (ZOH) discretization can be utilized as well. It assumes that the sampled signal remains constant between consecutive sampling points. For the ZOH discretization, the $\overline{A}$, $\overline{B}$, $\overline{C}$ are given by $\overline{A} = e^{A\Delta}$, $\overline{B} = (\Delta A)^{-1}(\overline{A} - I)\Delta B$, and $\overline{C} = C$.

12.1.1 Training SSMs

Instead of processing sequences step-by-step (like RNNs), SSMs exploit their linear time-invariant (LTI) property to convert the recurrent state-space equations into a global convolution kernel. This kernel allows parallel computation of outputs across the entire sequence, drastically speeding up training.
The convolution kernel K is derived from the discretized system given in Equation 12.10 and Equation 12.11:
- At $k = 0$: $h_0 = \overline{B} \cdot x_0$, $y_0 = C\overline{B}x_0$
- At $k = 1$: $h_1 = \overline{AB}x_0 + \overline{B}x_1$, $y_1 = C\overline{AB}x_0 + C\overline{B}x_1$
- At $k = m$: $h_m = \overline{A}^m\overline{B}x_0 + \overline{A}^{m-1}\overline{B}x_1 + \cdots + \overline{B}x_m$, $y_m = C\overline{A}^m\overline{B}x_0 + C\overline{A}^{m-1}\overline{B}x_1 + \cdots + C\overline{B}x_m$

Thus, the sequence of outputs $\{y_0, y_1, y_2, \dots\}$ is given by, excluding Dx_k (handled separately):

$$y = x * K, \tag{12.12}$$

where

$$K = \left[C\overline{B},\ C\overline{AB},\ C\overline{A}^2\overline{B},\ \dots,\ C\overline{A}^{L-1}\overline{B}\right], \tag{12.13}$$

where L is the sequence length.
The convolution $x * K$ is computed efficiently using the Fast Fourier Transform (FFT):

$$y = \mathcal{F}^{-1}\left(\mathcal{F}(K) \odot \mathcal{F}(x)\right) + Dx, \tag{12.14}$$

reducing complexity from $O(L^2)$ (naive convolution) to $O(L \log L)$.
Unlike RNNs, during the training phase of the S4 model, a convolutional architecture is employed to optimize the learning process, enabling parallel computation across input sequences for enhanced efficiency. By processing entire sequences simultaneously through convolutional operations, the model accelerates parameter updates and captures long-range dependencies effectively without gradient vanishing/explosion. This approach leverages the inherent parallelism of convolutional setups to reduce training time while maintaining robust performance. However, during inference, the model transitions to a recurrent formulation, which processes inputs step-by-step rather than in bulk. This recurrent mode allows for real-time or sequential task execution, as it generates outputs incrementally with minimal computational overhead at each step. The shift from convolutional training to recurrent inference

ensures both efficient learning and streamlined deployment, balancing speed during training with adaptability in practical applications. On the other hand, precomputing K for long sequences requires significant memory (mitigated by chunking).

12.2 HiPPO Initialization for S4 Models

A standard state-space model (SSM) demonstrates weak empirical performance and struggles to capture long-range dependencies effectively. One possible explanation is that linear first-order ordinary differential equations (ODEs) inherently solve to exponential functions, leading to gradients that scale exponentially with sequence length. This issue also arises from their formulation as a linear recurrence, where repeatedly applying a recurrent matrix results in the well-known vanishing or exploding gradient problem commonly observed in recurrent neural networks (RNNs). The HiPPO framework initializes A to optimally project historical input $x(t)$ onto polynomial bases (e.g., Legendre polynomials). The HiPPO-LegS [46] matrix for scaled Legendre basis is:

$$A_{nk} = -\begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k, \\ n+1 & \text{if } n = k, \\ 0 & \text{otherwise.} \end{cases}$$

This allows the state $h(t)$ to compress the history of $x(t)$ into coefficients of an orthogonal polynomial basis.
To reduce computational complexity, A is constrained to a diagonal plus low-rank (DPLR) structure:

$$A = \Lambda - PP^*,$$

where $\Lambda \in \mathbb{C}^{N\times N}$ is diagonal, and $P \in \mathbb{C}^{N\times r}$ is low-rank. Hence, this enables $O(N)$ parameters instead of $O(N^2)$ and efficient computation via conjugate symmetry and FFT.
Earlier research demonstrated that replacing a randomly initialized state-space model (SSM) matrix A with the HiPPO matrix significantly enhanced performance on the sequential MNIST classification benchmark, increasing accuracy from 60% to 98%.

12.3 Selective State Space (S6) Models

In traditional SSMs (e.g., S4), the convolution form arises because the system is linear time-invariant (LTI), meaning A, B, and C are fixed and do not depend on the input or time. Selective State Space (S6) Models [47], or Mamba models, introduces a selection mechanism to make the SSM more flexible and efficient. In particular,

B and C are input-dependent, breaking the LTI assumption in Mamba models. This means the convolution form must be adapted to account for the time-varying nature of B and C. The selection mechanism modifies the matrices B and C based on the input x_k. This is achieved through learned projections:

$$B_k = B \cdot \text{Linear}_B(x_k)$$
$$C_k = C \cdot \text{Linear}_C(x_k)$$

where Linear_B and Linear_C are learned linear transformations and B_k and C_k are input-dependent matrices. Hence, the Mamba model is given by

$$h_k = \overline{A} h_{k-1} + \overline{B}_k x_k \tag{12.15}$$
$$y_k = C_k h_k + D x_k \tag{12.16}$$

Because B_k and C_k vary with time, the convolution kernel K is no longer fixed. Instead, the output y can be expressed as a time-varying convolution:

$$y_k = \sum_{i=1}^{k} C_k \overline{A}^{k-i} \overline{B}_i x_i \tag{12.17}$$

Here, the kernel K is implicitly defined by the time-dependent terms C_k and $\overline{B}_i$. This formulation is more complex than the traditional SSM convolution because the kernel depends on both the current time step k and the past time steps i.
Mamba computes the hidden states $\{h_t\}$ efficiently using a parallel scan algorithm (similar to prefix sums), enabling parallel training despite the sequential recurrence:

$$h_k = \overline{A} \cdot h_{k-1} + \overline{B}_k \cdot x_k.$$

This avoids $\mathcal{O}(L^2)$ complexity and leverages modern hardware (GPUs/TPUs).[1]
The parallel scan algorithm leverages the associative property of linear recurrence to hierarchically compose transformations:

[1] Consider the linear recurrence relation in Mamba:

$$h_t = \overline{A} h_{t-1} + \overline{B}_t x_t$$

For a sequence of length L, the hidden states can be unrolled explicitly:

$$h_t = \overline{A}^t h_0 + \sum_{k=1}^{t} \overline{A}^{t-k} \overline{B}_k x_k$$

A naive parallel implementation would attempt to compute each h_t independently by:

1. Precomputing all powers of $\overline{A}$: Computing $\overline{A}^1, \overline{A}^2, \ldots, \overline{A}^L$ requires L matrix multiplications. However, to avoid sequential computation, one might compute all $\overline{A}^{t-k}$ terms directly, leading to $\mathcal{O}(L^2)$ operations since there are $L \times t$ terms for $t = 1, \ldots, L$.
2. Summing over all terms for each h_t: For each h_t, the sum $\sum_{k=1}^{t} \overline{A}^{t-k} \overline{B}_k x_k$ involves t matrix-

- Associative Reformulation: The recurrence can be viewed as applying a linear operator $T_k = (\overline{A}, \overline{B}_k x_k)$ to the state h_{k-1}. These operators compose associatively:

$$T_i \circ T_j = (\overline{A}, \overline{B}_i x_i) \circ (\overline{A}, \overline{B}_j x_j) = (\overline{A}^2, \overline{AB}_i x_i + \overline{B}_j x_j),$$

  where $(T_1 \circ T_2) \circ T_3 = T_1 \circ (T_2 \circ T_3)$.[2]

- Tree-Based Computation: The scan algorithm proceeds in $\log L$ layers: Layer 1: Compose adjacent pairs $(T_1 \circ T_2), (T_3 \circ T_4), \ldots$ in parallel. Layer 2: Compose results from Layer 1 into chunks of 4 steps, etc. After $\log L$ layers, the full sequence is covered.

- Work Complexity: Each layer performs $\mathcal{O}(L)$ compositions. With $\log L$ layers, the total work is $\mathcal{O}(L \log L)$.

Example for $L = 4$

1. Step 1: Compute local transformations:

$$T_1, \quad T_2, \quad T_3, \quad T_4$$

2. Step 2: Compose adjacent pairs in parallel:

$$T_{1:2} = T_1 \circ T_2, \quad T_{3:4} = T_3 \circ T_4$$

3. Step 3: Compose results to cover the full sequence:

$$T_{1:4} = T_{1:2} \circ T_{3:4}$$

---

vector multiplications. Across all L states, this results in $\sum_{t=1}^{L} t = \frac{L(L+1)}{2} = \mathcal{O}(L^2)$ operations. Please note for sequential recurrence, the complexity is $\mathcal{O}(L)$.

Thus, the naive approach is quadratic in sequence length due to redundant computations.

[2] Actually, we represent each step as a linear operator T_k that maps $(h_{k-1}, 1)$ to $(h_k, 1)$:

$$T_k = \begin{pmatrix} \overline{A} & \overline{B}_k x_k \\ 0 & 1 \end{pmatrix}, \quad T_k \begin{pmatrix} h_{k-1} \\ 1 \end{pmatrix} = \begin{pmatrix} h_k \\ 1 \end{pmatrix}$$

The "1" is a dummy dimension to handle the affine term.

These operators compose associatively:

$$T_i \circ T_j = \begin{pmatrix} \overline{A} & \overline{B}_i x_i \\ 0 & 1 \end{pmatrix} \circ \begin{pmatrix} \overline{A} & \overline{B}_j x_j \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \overline{A}^2 & \overline{AB}_i x_i + \overline{B}_j x_j \\ 0 & 1 \end{pmatrix}$$

4. Output: Extract all h_t by combining partial results. To recover all h_t (not just h_4), we store intermediate results: $h_1 = T_1 h_0$, $h_2 = T_{1:2} h_0$, $h_3 = T_{1:2} \circ T_3 h_0$, and $h_4 = T_{1:4} h_0$.

   Total operations: 4 (Step 1) + 2 (Layer 1) + 1 (Layer 2) = 7 compositions.

The parallel scan avoids $\mathcal{O}(L^2)$ by reusing intermediate computations (like dynamic programming), exploiting associativity to hierarchically merge chunks, and leveraging parallelism with $\mathcal{O}(L \log L)$ work instead of brute-force $\mathcal{O}(L^2)$.
Selective SSM blocks can be integrated as independent transformation units within a neural network, similar to how RNN cells such as LSTMs or GRUs are utilized. The complete structure of a Mamba block extends beyond just the SSM module discussed earlier. It includes additional components such as linear projections, convolutions, and non-linear activation functions that work alongside the SSM block within the broader Mamba architecture. The Mamba block is shown in Figure 12.1.

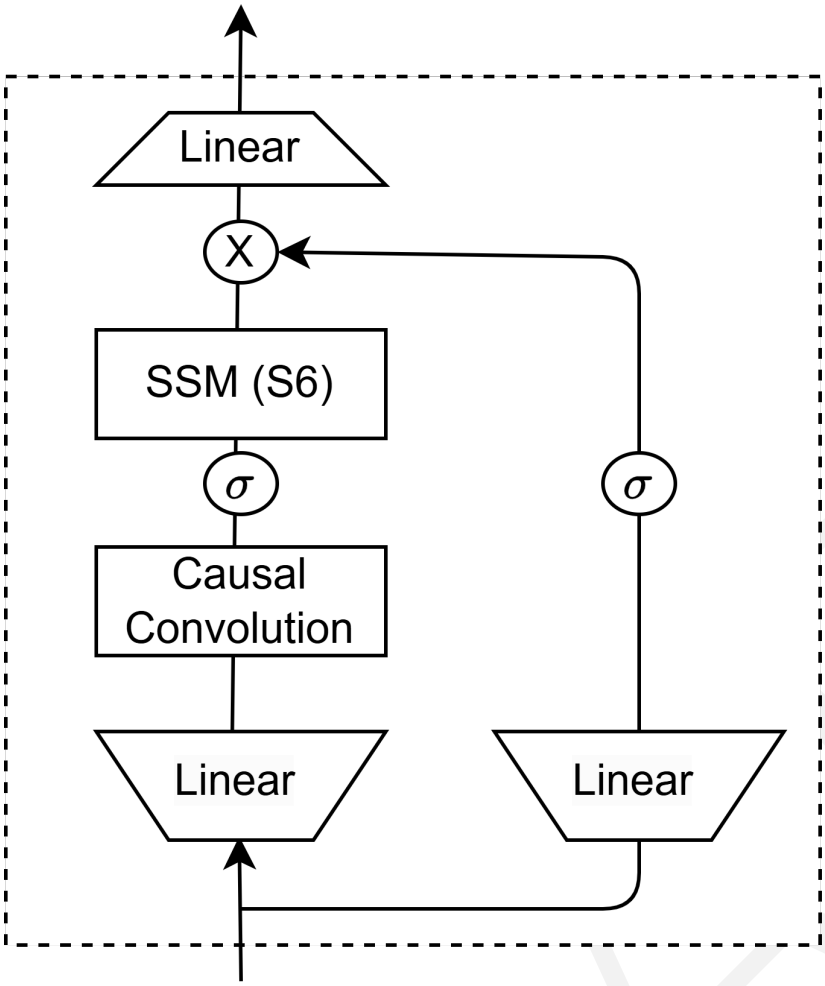


Figure 12.1: A Mamba block [47].

Transformers rely on attention mechanisms to model long-range dependencies, whereas Mamba is based on structured state-space models (SSMs), making it more efficient for sequential tasks. Transformers have high computational complexity due to the quadratic scaling of self-attention, whereas Mamba significantly reduces complexity by achieving linear scaling with sequence length. In terms of inference speed, Transformers require $\mathcal{O}(L)$ operations due to attention computations, while Mamba operates in constant time $\mathcal{O}(1)$, making it more efficient for long sequences. Training Transformers involves quadratic complexity $\mathcal{O}(L^2)$, making them resource-intensive, whereas Mamba benefits from linear training complexity $\mathcal{O}(L \log L)$, lead-

ing to faster convergence and reduced memory usage. While Transformers excel in capturing global dependencies through attention, Mamba leverages selective state-space modeling to process sequential information efficiently without explicit attention mechanisms. Overall, Mamba presents a promising alternative to Transformers, particularly in scenarios where efficiency and scalability are critical, such as processing extremely long sequences. A comparison between Transformer and Mamba block is shown in Table 12.1.

12.3.1 Bidirectional Mamba models

Bidirectional Mamba (Bi-Mamba) extends the strengths of the Mamba architecture (efficiency, selectivity) to bidirectional contexts, making it a powerful tool for tasks requiring full-sequence understanding. While it sacrifices some autoregressive capabilities, it outperforms Transformers and RNNs in memory efficiency and long-context modeling for non-autoregressive applications. The basic structure of Bi-Mamba has two blocks of Mamba: one block handles the past context and the second block handles the flipped input (i.e. future context). This is similar to Bidirectional RNNs discussed in Chapter 10.

$$y = \text{Mamba}(x) + \text{flip}(\text{Mamba}(\text{flip}(x))) \tag{12.18}$$

Feature	Transformer	Mamba
Architecture	Attention-based	SSM-based
Complexity	High	Lower
Inference Speed	$\mathcal{O}(L)$	$\mathcal{O}(1)$
Training Speed	$\mathcal{O}(L^2)$	$\mathcal{O}(L \log L)$
Memory Usage	$\mathcal{O}(L^2)$	$\mathcal{O}(L)$

Table 12.1: Comparison between Transformer and Mamba.

12.4 Improvements based on Mamba

The Jamba Block is a key architectural component of the Jamba model [48], an advanced hybrid neural network that efficiently combines state-space models and attention mechanisms. The Jamba block is designed to leverage the long-range dependency modeling of SSMs while incorporating the context-awareness of attention mechanisms. Unlike traditional Transformers, which suffer from quadratic complexity in sequence length, the Jamba block introduces a more efficient representation.
Jamba blocks (see Figure 12.2) combine Mamba's selective structured state space models (SSMs) with Mixture-of-Experts (MoE) layers to enhance model capacity

and computational efficiency. The MoE layer Scales model capacity efficiently by routing tokens to specialized experts and it consists of N experts $\{E_i\}_{i=1}^{N}$ and a trainable gating network G. For an input token $\mathbf{x} \in \mathbb{R}^d$:
The gating network computes weights for expert selection:

$$\mathbf{g}(\mathbf{x}) = \text{TopK}\left(\text{Softmax}\left(\frac{W_g\mathbf{x} + \mathbf{b}_g}{\tau}\right)\right), \tag{12.19}$$

where $W_g \in \mathbb{R}^{N\times d}, \mathbf{b}_g \in \mathbb{R}^N$: Gating parameters, and τ: Temperature hyperparameter (often omitted, $\tau = 1$). The TopK: Retains only the top k values (others set to 0), followed by re-normalization.[3]
Each expert E_i is a feed-forward network (FFN):

$$E_i(\mathbf{x}) = \mathbf{W}_{2,i} \cdot \sigma\left(\mathbf{W}_{1,i}\mathbf{x} + \mathbf{b}_{1,i}\right) + \mathbf{b}_{2,i}, \tag{12.20}$$

where σ: Activation function (e.g., GeLU, SiLU) and $\mathbf{W}_{1,i} \in \mathbb{R}^{h\times d}, \mathbf{W}_{2,i} \in \mathbb{R}^{d\times h}$: Expert parameters.
The final output combines selected experts via gating weights:

$$\mathbf{y} = \sum_{i=1}^{N} g_i(\mathbf{x}) \cdot E_i(\mathbf{x}) \tag{12.21}$$

The Jamba block integrates selective attention to enhance the capabilities of State Space Models (SSMs) by capturing dynamic token interactions efficiently. Instead of applying full self-attention, it strategically incorporates attention only where necessary, reducing computational overhead while preserving expressivity. While SSMs excel at handling long-range dependencies, attention refines both local and global interactions, improving sequence modeling. This hybrid approach enables Jamba to balance structured memory processing with adaptive token relationships, ensuring both scalability and strong performance in NLP and speech tasks. By leveraging attention in a controlled manner, Jamba maintains efficiency while outperforming pure SSM-based models like Mamba. Ultimately, attention in Jamba plays a crucial role in optimizing information flow, making it a powerful alternative to traditional Transformer architectures. Based on a sequence of 4 Jamba blocks, the Jamba language model was successfully trained on context lengths of up to 1M tokens.

[3]The top k values (e.g., $k = 2$) experts process each token, limiting compute costs.

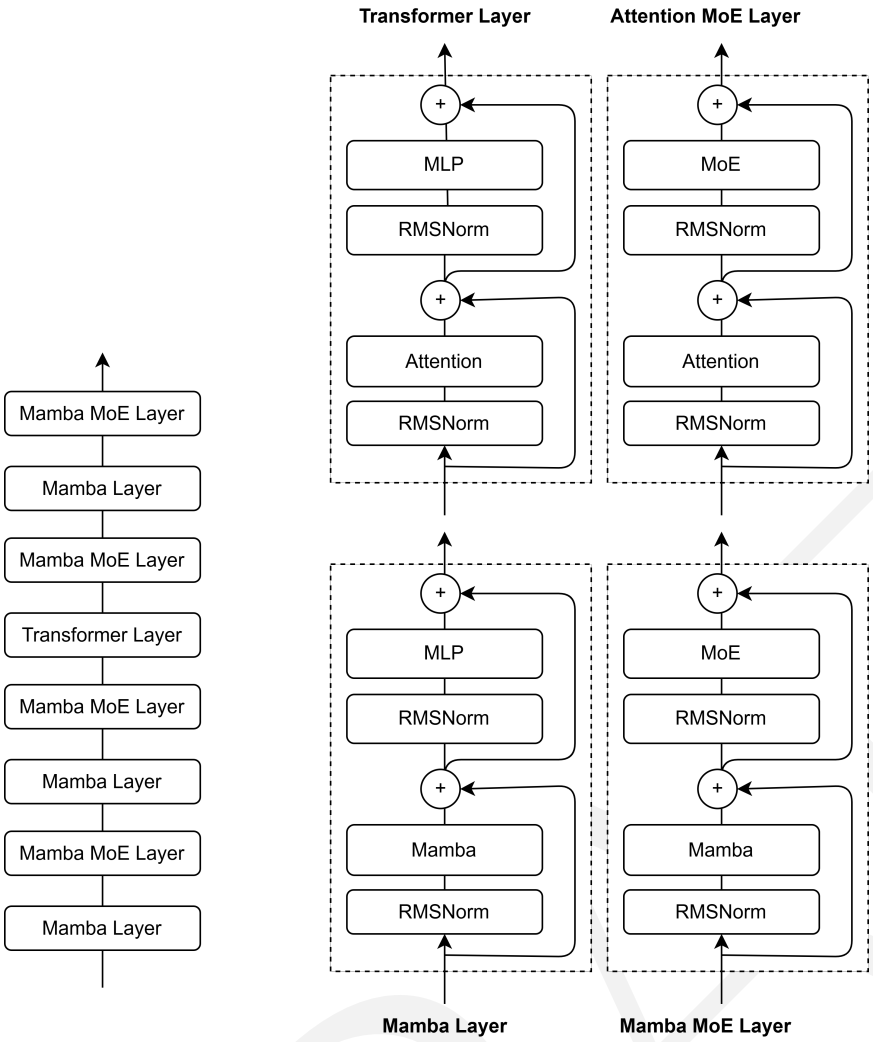


Figure 12.2: A single Jamba block [48].

13. Probablistic Learning

Although initially introduced and studied in the late 1960s and early 1970s, statistical methods of Markov source or hidden Markov modeling have become increasingly popular in the last several years. There are two strong reasons why this has occurred. First the models are very rich in mathematical structure and hence can form the theoretical basis for use in a wide range of applications. Second the models, when applied properly, work very well in practice for several important applications.

— Lawrence R. Rabiner

This chapter presents probabilistic learning methods. These approaches are widely used due to their simplicity and effectiveness. They are computationally efficient, making both training and inference straightforward. One well-known example is Hidden Markov Models (HMMs), which played a crucial role in speech recognition for decades. Before deep learning innovations, HMMs were the dominant method in this field. They were eventually replaced by more advanced techniques like recurrent neural networks (RNNs) and transformers. A key feature of probabilistic models is their reliance on strong mathematical assumptions. These assumptions often enable efficient derivations, reducing computational complexity. Additionally, they frequently lead to closed-form solutions, simplifying optimization. Despite their limitations, probabilistic learning remains valuable in many applications.

13.1 Naïve Bayes Multiclass Classification

The Naïve Bayes classifier is a probabilistic model based on Bayes' theorem with the naïve assumption that features are independent given the class label. Despite this strong assumption, it performs well in many applications, such as text classification and spam detection. For discrete features, it is known as multinomial Naïve Bayes classifier.

Bayes' theorem states that for two events A and B:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{13.1}$$

In the context of Multiclass classification, let C be the class label (e.g., spam or not spam) and $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ be the feature vector.

Then, the posterior probability of class C_k given the features is:

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k)P(C_k)}{P(\mathbf{x})} \tag{13.2}$$

Since $P(\mathbf{x})$ is the same for all classes, we use the proportional form:

$$P(C_k|\mathbf{x}) \propto P(\mathbf{x}|C_k)P(C_k) \tag{13.3}$$

The Naïve Bayes assumption simplifies the computation by assuming feature independence, meaning:

$$P(x|C_k) = \prod_{i=1}^{d} P(x_i|C_k) \tag{13.4}$$

Thus, the posterior becomes:

$$P(C_k|\mathbf{x}) \propto P(C_k) \prod_{i=1}^{d} P(x_i|C_k) \tag{13.5}$$

The classifier assigns $\mathbf{x}$ to the class with the highest posterior probability:

$$\hat{C} = \arg\max_{C_k} P(C_k) \prod_{i=1}^{d} P(x_i|C_k) \tag{13.6}$$

For discrete features, such as text classification (word counts), we model word occurrences using:

$$P(x_i|C_k) = \frac{N_{i,k}}{N_k}, \tag{13.7}$$

where $N_{i,k}$ is the count of word x_i in class C_k and N_k is the total count of words in class C_k.
To prevent zero probabilities when a word is missing from a class, we apply Laplace Smoothing (Laplace Correction):

$$P(x_i|C_k) = \frac{N_{i,k} + 1}{N_k + V}, \tag{13.8}$$

where V is the vocabulary size (total number of unique words).
Example: Assume we classify emails as spam (S) or not spam ($\neg S$), based on words ("free", "win").

Class	"free" count	"win" count	Total words
Spam (S)	4	3	10
Not Spam ($\neg S$)	2	1	8

Table 13.1: Word occurrences in spam and non-spam emails

Using Laplace smoothing ($V = 3$ since we consider three words: "free", "win", and an unseen word):

$$P(\text{"free"}|S) = \frac{4+1}{10+3} = \frac{5}{13} \tag{13.9}$$

$$P(\text{"win"}|S) = \frac{3+1}{10+3} = \frac{4}{13} \tag{13.10}$$

For non-spam:

$$P(\text{"free"}|\neg S) = \frac{2+1}{8+3} = \frac{3}{11} \tag{13.11}$$

$$P(\text{"win"}|\neg S) = \frac{1+1}{8+3} = \frac{2}{11} \tag{13.12}$$

Given $P(S) = 0.6$ and $P(\neg S) = 0.4$, the posterior probabilities for a new email containing "free win" are:

$$P(S|\text{"free win"}) \propto P(S)P(\text{"free"}|S)P(\text{"win"}|S) \tag{13.13}$$

$$= 0.6 \times \frac{5}{13} \times \frac{4}{13} \tag{13.14}$$

$$P(\neg S|\text{"free win"}) \propto P(\neg S)P(\text{"free"}|\neg S)P(\text{"win"}|\neg S) \tag{13.15}$$

$$= 0.4 \times \frac{3}{11} \times \frac{2}{11} \tag{13.16}$$

Computing these values, we choose the class with the highest probability.
Naïve Bayes is simple and efficient, even with its strong independence assumption. It works well in many practical scenarios, especially text classification and spam filtering.

13.2 Gaussian Models

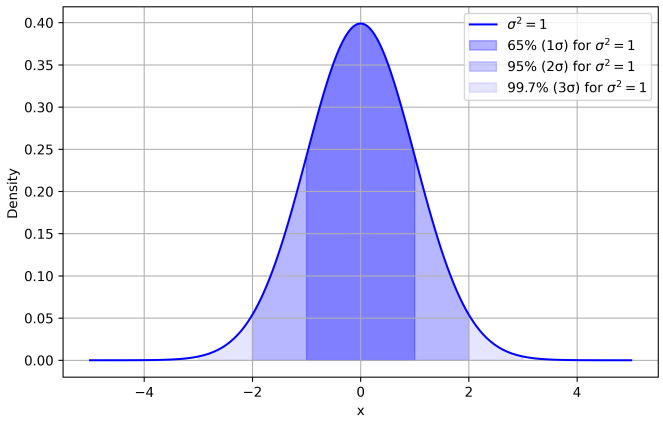


Figure 13.1: 1D Gaussian density with variance =1.

The probability density function (PDF) of a 1-dimensional Gaussian distribution[1] (also known as the normal distribution) is given by (see Figure 13.1):

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{13.17}$$

where x is the continuous random variable, μ is the mean (center of the distribution), σ^2 is the variance (spread of the distribution), σ is the standard deviation (square root of variance), π is the mathematical constant approximately equal to 3.14159, $\exp(\cdot)$ denotes the exponential function.
The Gaussian function assigns higher probability density to values of x that are closer to the mean and lower probability to values in the tails (far from μ). It is symmetric about the mean μ, meaning the left and right sides of the distribution are mirror images. The total area under the curve of $p(x)$ is 1, which ensures that it is a valid probability distribution:

[1]The Gaussian distribution or model is also called the bell curve due to its characteristic shape, which is smooth and unimodal.

$$\int_{-\infty}^{\infty} p(x)\,dx = 1 \tag{13.18}$$

The empirical rule explains the distribution of data in a normal distribution, indicating that almost all values are within three standard deviations of the mean. More precisely, around 68% of the data lies within one standard deviation, 95% falls within two standard deviations, and nearly 99.7% is encompassed within three standard deviations from the mean. When the variance σ^2 increases, the curve widens (spreads out), indicating more uncertainty or variation in the data. If the variance is small, the Gaussian curve is narrow and peaked, indicating that most values are close to the mean.

13.2.1 Properties of a Gaussian model

The Gaussian distribution has key properties, such as its normalized density function, which can be calculated along with its mean and variance. To derive the mean and variance of a Gaussian random variable x with probability density function (PDF) (see Equation 13.17):
The mean (expected value) of x is defined as:

$$\mathbb{E}[x] = \int_{-\infty}^{\infty} xp(x)dx. \tag{13.19}$$

Substitute the Gaussian PDF into this integral:

$$\mathbb{E}[x] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} xe^{-\frac{(x-\mu)^2}{2\sigma^2}} dx. \tag{13.20}$$

Substitution: Let $y = x - \mu$. Then $x = y + \mu$, $dx = dy$, and the limits remain $-\infty$ to ∞:

$$\mathbb{E}[x] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (y+\mu)e^{-\frac{y^2}{2\sigma^2}} dy. \tag{13.21}$$

Split the integral into two terms:

$$\mathbb{E}[x] = \frac{1}{\sigma\sqrt{2\pi}} \left[\underbrace{\int_{-\infty}^{\infty} ye^{-\frac{y^2}{2\sigma^2}} dy}_{\text{Odd function: } 0} + \mu \int_{-\infty}^{\infty} e^{-\frac{y^2}{2\sigma^2}} dy \right], \tag{13.22}$$

where the first integral is zero because $ye^{-\frac{y^2}{2\sigma^2}}$ is an odd function and the second integral is the Gaussian integral:

$$\int_{-\infty}^{\infty} e^{-\frac{y^2}{2\sigma^2}} dy = \sigma\sqrt{2\pi}. \tag{13.23}$$

Thus:

$$\mathbb{E}[x] = \frac{1}{\sigma\sqrt{2\pi}} \cdot \mu \cdot \sigma\sqrt{2\pi} = \mu. \tag{13.24}$$

The variance is defined as:

$$\text{Var}(x) = \mathbb{E}\left[(x-\mu)^2\right] = \mathbb{E}[x^2] - (\mathbb{E}[x])^2. \tag{13.25}$$

First, compute $\mathbb{E}[x^2]$:

$$\mathbb{E}[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} x^2 e^{-\frac{(x-\mu)^2}{2\sigma^2}}\, dx. \tag{13.26}$$

Substitution: Again, let $y = x - \mu$, so $x = y + \mu$, $dx = dy$:

$$\mathbb{E}[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (y+\mu)^2 e^{-\frac{y^2}{2\sigma^2}}\, dy. \tag{13.27}$$

Expand $(y+\mu)^2$:

$$\mathbb{E}[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} \left(y^2 + 2\mu y + \mu^2\right) e^{-\frac{y^2}{2\sigma^2}}\, dy. \tag{13.28}$$

Split into three integrals:

$$\mathbb{E}[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \left[\int_{-\infty}^{\infty} y^2 e^{-\frac{y^2}{2\sigma^2}}\, dy + 2\mu \underbrace{\int_{-\infty}^{\infty} y e^{-\frac{y^2}{2\sigma^2}}\, dy}_{\text{Odd function: 0}} + \mu^2 \int_{-\infty}^{\infty} e^{-\frac{y^2}{2\sigma^2}}\, dy \right], \tag{13.29}$$

where the second integral is zero (odd function), the third integral is $\mu^2 \cdot \sigma\sqrt{2\pi}$, and for the first integral, we use the known result for the second moment of $Y \sim \mathcal{N}(0, \sigma^2)$:

$$\int_{-\infty}^{\infty} y^2 e^{-\frac{y^2}{2\sigma^2}}\, dy = \sigma^3\sqrt{2\pi}. \tag{13.30}$$

Substitute these results:

$$\mathbb{E}[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \left[\sigma^3\sqrt{2\pi} + 0 + \mu^2\sigma\sqrt{2\pi}\right] = \sigma^2 + \mu^2. \tag{13.31}$$

Finally, compute the variance:

$$\text{Var}(x) = \mathbb{E}[x^2] - (\mathbb{E}[x])^2 = (\sigma^2 + \mu^2) - \mu^2 = \sigma^2. \tag{13.32}$$

Hence, the parameter σ^2 in the Gaussian PDF is the variance of x.

13.2.2 Multivariate Gaussian models

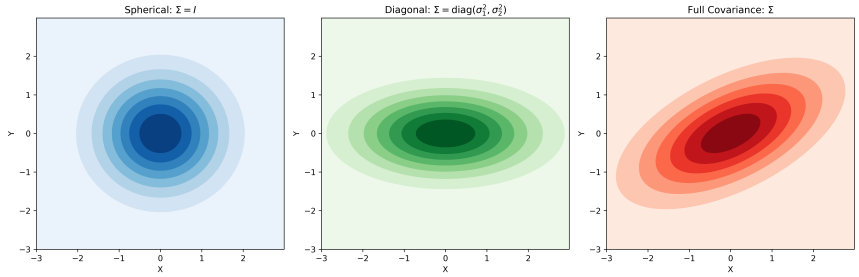


Figure 13.2: Gaussian densities in 2D with different covariance types.

The probability density function (PDF) for a d-dimensional Gaussian distribution is[2]:

$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{13.33}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the random vector, $\boldsymbol{\mu} \in \mathbb{R}^d$ is the mean vector, $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$ is the covariance matrix (symmetric positive definite), $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$, and $\boldsymbol{\Sigma}^{-1}$ is the inverse of $\boldsymbol{\Sigma}$.

The type of covariance matrix determines the shape of the contour plot for a Gaussian density. A spherical Gaussian has equal variances in all directions, with off-diagonal elements set to zero, resulting in circular contour plots. In contrast, a diagonal covariance matrix allows different variances along each axis while maintaining zero off-diagonal elements, leading to elliptical contours aligned with the coordinate axes. A full covariance matrix, however, includes nonzero off-diagonal elements, indicating correlations between dimensions, which causes the elliptical contours to be rotated. Figure 13.2 illustrates the contour plots for these three covariance structures.

From a linear algebra perspective, eigenvalues and eigenvectors of a Gaussian's covariance matrix describe its shape and orientation. Given a covariance matrix Σ, the eigen-decomposition is:

$$\boldsymbol{\Sigma} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T, \tag{13.34}$$

where $\mathbf{Q}$ contains the eigenvectors as columns and $\boldsymbol{\Lambda}$ is a diagonal matrix with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_d$ on the diagonal. The eigenvectors determine the principal axes (directions) of the Gaussian's elliptical contours. The eigenvalues define the variance along those directions. In 2D, the contours of the Gaussian are ellipses, and the lengths of the semi-axes are proportional to $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$. As shown in

[2]The Gaussian density can serve as an activation function when applied to an input vector.

Figure 13.3, these values control how stretched or compressed the Gaussian is in each principal direction.

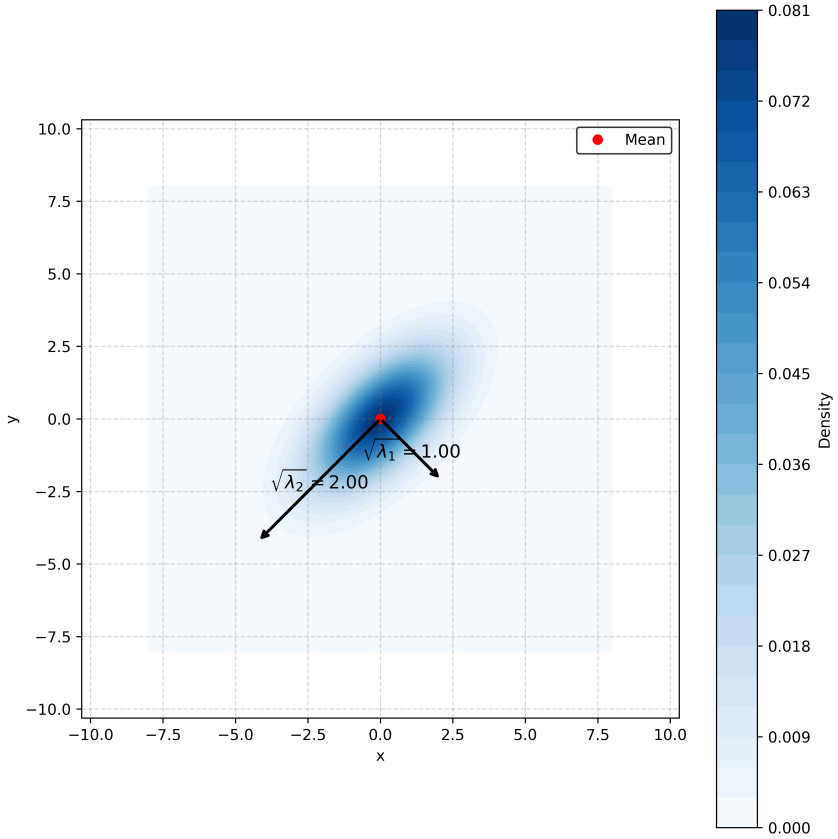


Figure 13.3: Relation between Covariance matrix and eigenvectors and eigenvalues.

By definition, the mean $\boldsymbol{\mu}$ is the expected value of $\mathbf{x}$:

$$\boldsymbol{\mu} = \mathbb{E}[\mathbf{x}] = \int_{\mathbb{R}^d} \mathbf{x} \cdot p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})\, d\mathbf{x}, \tag{13.35}$$

and the covariance matrix $\boldsymbol{\Sigma}$ is defined as:

$$\boldsymbol{\Sigma} = \mathbb{E}\left[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\right] = \int_{\mathbb{R}^d} (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T \cdot p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})\, d\mathbf{x}. \tag{13.36}$$

The covariance matrix $\boldsymbol{\Sigma}$ can alternatively be expressed as:

$$\boldsymbol{\Sigma} = \mathbb{E}[\mathbf{x}\mathbf{x}^T] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{x}]^T \tag{13.37}$$

which simplifies to:

$$\boldsymbol{\Sigma} = \mathbb{E}[\mathbf{x}\mathbf{x}^T] - \boldsymbol{\mu}\boldsymbol{\mu}^T \tag{13.38}$$

13.2.3 Learning Problem

Maximum Likelihood Estimation (MLE) is a method used to determine the parameters of a Gaussian distribution that best fit the data. It finds the values of the mean and covariance that maximize the likelihood of observing the given dataset. Given a dataset of N i.i.d.[3] samples $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, we estimate the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ using Maximum Likelihood Estimation (MLE).
The likelihood function for N independent samples is:

$$L(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=1}^{N} p(\mathbf{x}_i) \tag{13.39}$$

Taking the log-likelihood:

$$\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{N} \log p(\mathbf{x}_i|\boldsymbol{\mu}, \boldsymbol{\Sigma}). \tag{13.40}$$

Substituting the Gaussian PDF:

$$\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{Nd}{2}\log(2\pi) - \frac{N}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}). \tag{13.41}$$

Maximizing with Respect to $\boldsymbol{\mu}$ and using the matrix identity[4]:

$$\frac{\partial \mathbf{x}^T\mathbf{A}\mathbf{x}}{\partial \mathbf{x}} = 2\mathbf{A}\mathbf{x}. \tag{13.42}$$

Take the derivative of $\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to $\boldsymbol{\mu}$:

$$\frac{\partial \mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1}\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu}), \tag{13.43}$$

and set the derivative to zero:

$$\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu}) = 0 \implies \boldsymbol{\mu}_{\text{MLE}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i. \tag{13.44}$$

[3] i.i.d. stands for independent and identically distributed where each sample doesn't affect or depend on the others and all variables follow the same probability distribution. This assumption breaks down when dealing with sequential data, where each observation often depends on the ones before it.

[4] The Matrix Cookbook: https://www2.imm.dtu.dk/pubdb/edoc/imm3274.pdf.

Maximizing with Respect to $\boldsymbol{\Sigma}$: Let the scatter matrix $\mathbf{S} = \sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu}_{\text{MLE}})(\mathbf{x}_i - \boldsymbol{\mu}_{\text{MLE}})^T$ and using the matrix calculus identities:

$$\frac{\partial \log |\boldsymbol{\Sigma}|}{\partial \boldsymbol{\Sigma}} = \boldsymbol{\Sigma}^{-1}, \tag{13.45}$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. Given the cyclic property of Trace: For matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, we have $\text{tr}(\mathbf{ABC}) = \text{tr}(\mathbf{CAB})$ (if dimensions match). Applying this:

$$\text{tr}\left((\mathbf{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})\right) = \text{tr}\left(\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T\right), \tag{13.46}$$

and the trace of a sum is the sum of traces (i.e. linearity of Trace), so:

$$\sum_{i=1}^{N} \text{tr}\left(\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T\right) = \text{tr}\left(\boldsymbol{\Sigma}^{-1}\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T\right). \tag{13.47}$$

Therefore,

$$\frac{\partial}{\partial \boldsymbol{\Sigma}}\left(\text{tr}(\boldsymbol{\Sigma}^{-1}\mathbf{S})\right) = -\boldsymbol{\Sigma}^{-1}\mathbf{S}\boldsymbol{\Sigma}^{-1}. \tag{13.48}$$

Combining the Identities: The log-likelihood $\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ for a multivariate Gaussian (up to constants) is:

$$\mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{N}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}\text{tr}\left(\boldsymbol{\Sigma}^{-1}\mathbf{S}\right). \tag{13.49}$$

Taking the derivative with respect to $\boldsymbol{\Sigma}$ and applying the identities:

$$\frac{\partial \mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\Sigma}} = -\frac{N}{2}\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathbf{S}\boldsymbol{\Sigma}^{-1} = 0. \tag{13.50}$$

Solving for $\boldsymbol{\Sigma}$: multiply through by $2\boldsymbol{\Sigma}$ (from the left and right) to eliminate the inverses:

$$-N\boldsymbol{\Sigma} + \mathbf{S} = 0 \implies \boldsymbol{\Sigma} = \frac{1}{N}\mathbf{S}. \tag{13.51}$$

This gives the maximum likelihood estimate (MLE) for the covariance matrix:

$$\boldsymbol{\Sigma}_{\text{MLE}} = \frac{1}{N}\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu}_{\text{MLE}})(\mathbf{x}_i - \boldsymbol{\mu}_{\text{MLE}})^T. \tag{13.52}$$

The MLE covariance $\boldsymbol{\Sigma}_{\text{MLE}}$ is biased (divided by N) and consistent (converges to the true parameter as $N \to \infty$).[5] An unbiased estimator uses $N - 1$ instead of N.[6]

[5] A biased estimator in statistics is one whose expected value (mean) does not equal the true parameter it is estimating. Mathematically, let $\hat{\theta}$ be an estimator of a parameter θ. The bias of $\hat{\theta}$ is defined as $\text{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$ where the estimator $\hat{\theta}$ is biased if $\mathbb{E}[\hat{\theta}] \neq \theta$ else if $\mathbb{E}[\hat{\theta}] = \theta$, the estimator is unbiased.

[6] The Bessel correction is the use of $(n-1)$ instead of n in the denominator of the sample variance

13.2.4 Gaussian Naïve Bayes Classifier

For continuous features, we assume that each feature follows a normal distribution:

---

formula to make it an unbiased estimator of the population variance. Let $x_1, x_2, \ldots, x_n$ be independent and identically distributed (i.i.d.) 1D random variables with the true mean: $\mu = \mathbb{E}[x_i]$ and true variance: $\sigma^2 = \text{Var}(x_i)$. To estimate the biased sample variance $\hat{\sigma}^2_{\text{biased}}$ from the data, Since expectation is linear:

$$\mathbb{E}[\bar{x}] = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} x_i\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[x_i] = \frac{1}{n}\cdot n\mu = \mu.$$

So, $\bar{x}$ is an unbiased estimator of μ. By expanding the squared differences $(x_i-\bar{x})^2 = (x_i-\mu+\mu-\bar{x})^2 = (x_i-\mu)^2 + (\mu-\bar{x})^2 + 2(x_i-\mu)(\mu-\bar{x})$, the biased sample variance:

$$\mathbb{E}[\hat{\sigma}^2_{\text{biased}}] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[(x_i-\bar{x})^2].$$

Substituting the expansion:

$$\mathbb{E}[(x_i-\bar{x})^2] = \mathbb{E}[(x_i-\mu)^2] + \mathbb{E}[(\mu-\bar{x})^2] + 2\mathbb{E}[(x_i-\mu)(\mu-\bar{x})].$$

Calculating each term: $\mathbb{E}[(x_i-\mu)^2] = \sigma^2$ (population variance) and $\mathbb{E}[(\mu-\bar{x})^2] = \text{Var}(\bar{x}) = \text{Var}\left(\frac{1}{n}\sum x_i\right) = \frac{1}{n^2}\sum \text{Var}(x_i) = \frac{\sigma^2}{n}$ (variance of the sample mean). For the cross-term:

$$\mathbb{E}[(x_i-\mu)(\mu-\bar{x})] = \mathbb{E}\left[(x_i-\mu)\left(\mu - \frac{1}{n}\sum_{j=1}^{n} x_j\right)\right] = \mathbb{E}\left[-\frac{(x_i-\mu)^2}{n}\right] = -\frac{\sigma^2}{n},$$

because $\mathbb{E}[(x_i-\mu)(x_j-\mu)] = 0$ for $i \neq j$ (independent samples).

Substituting these into the expression:

$$\mathbb{E}[(x_i-\bar{x})^2] = \sigma^2 + \frac{\sigma^2}{n} + 2\left(-\frac{\sigma^2}{n}\right) = \sigma^2 - \frac{\sigma^2}{n}.$$

Thus, the expected value of the biased estimator is:

$$\mathbb{E}[\hat{\sigma}^2_{\text{biased}}] = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma^2 - \frac{\sigma^2}{n}\right) = \frac{1}{n}\cdot n\left(\sigma^2 - \frac{\sigma^2}{n}\right) = \sigma^2 - \frac{\sigma^2}{n} = \frac{n-1}{n}\sigma^2.$$

The expected value of the biased sample variance estimator is:

$$\mathbb{E}[\hat{\sigma}^2_{\text{biased}}] = \frac{n-1}{n}\sigma^2.$$

This shows that the biased estimator underestimates the true population variance σ^2 by a factor of $\frac{n-1}{n}$. To correct this bias, we use Bessel's correction, multiplying by $\frac{n}{n-1}$ to obtain the unbiased sample variance estimator:

$$\hat{\sigma}^2_{\text{unbiased}} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2.$$

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_{k,i}^2}} \exp\left(-\frac{(x_i - \mu_{k,i})^2}{2\sigma_{k,i}^2}\right) \tag{13.53}$$

where $\mu_{k,i}$ and $\sigma_{k,i}^2$ are the sample mean and variance of feature x_i for class C_k. By applying Bayes' rule as indicated in Equation 13.6, we select the class that has the highest probability. The process of generative estimation for Gaussian densities (models) is highly efficient and can be completed in a single pass through the training data. We partition the training data according to their respective classes and independently calculate the mean and variance or covariance for each class. It's important to note that generative training does not differentiate between classes. As a result, it does not make optimal use of the parameters compared to discriminative training. Thus, while generative training is effective, it may lack the efficiency of its discriminative counterpart.

13.3 Gaussian Mixture Models

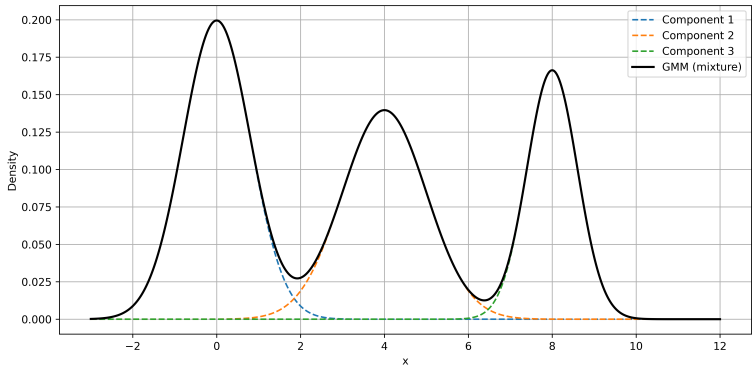


Figure 13.4: 1D Gaussian Mixture Model with 3 Components (Multimodal Density).

The Gaussian Mixture Model (GMM) is a powerful extension of the single Gaussian density function, allowing us to model more complex, multimodal data distributions. While a single Gaussian assumes the data is generated from one symmetric, bell-shaped distribution centered around a mean, real-world data often exhibits multiple clusters or modes. GMM addresses this limitation by combining several Gaussian components, each capturing different subpopulations or structures in the data. This makes GMM especially useful in tasks like clustering, density estimation, and unsupervised learning, where the assumption of a single Gaussian is too restrictive.

Moreover, GMM can approximate any continuous distribution given enough components, making it a flexible and expressive model. By estimating each component's mean, covariance, and mixing weight, GMM provides a probabilistic framework that captures uncertainty and overlapping clusters in data. A GMM represents a probability distribution as a weighted sum of K Gaussian components:

$$p(\mathbf{x}|\Theta) = \sum_{k=1}^{K} c_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{13.54}$$

where c_k is the mixing coefficient (weight) of the k-th component ($\sum_{k=1}^{K} c_k = 1$), $\boldsymbol{\mu}_k$ is the mean vector of the k-th component, $\boldsymbol{\Sigma}_k$ is the covariance matrix of the k-th component, $\Theta = \{c_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^{K}$ represents all parameters, and $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the multivariate Gaussian density.
As illustrated in Figure 13.4, the Gaussian Mixture Model (GMM) can represent any arbitrary multimodal probability distribution.

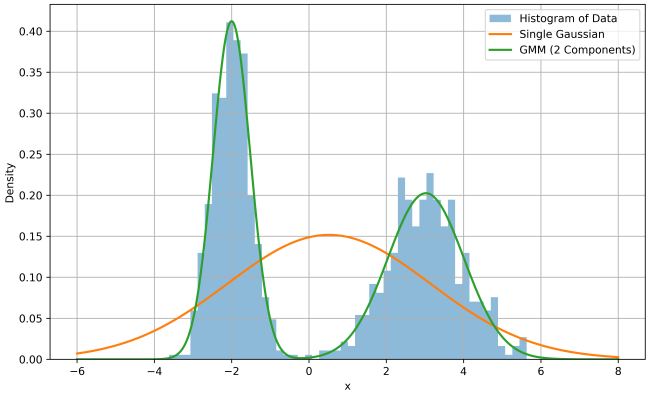


Figure 13.5: 1D Gaussian Mixture Model with 3 Components (Multimodal Density).

Figure 13.5 demonstrates how using a single Gaussian to approximate complex, multimodal data results in an inaccurate density estimation and leads to incorrect modeling assumptions.

13.3.1 Learning Problem

The Expectation-Maximization (EM) algorithm is a powerful iterative method for finding maximum likelihood estimates in probabilistic models with latent variables. For Gaussian Mixture Models (GMMs), EM provides an efficient way to learn the parameters of the mixture components.

Given data $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and a probabilistic model with latent variables $\mathbf{Z} = \{z_1, \ldots, z_N\}$, we want to maximize:

$$\log p(\mathbf{X}; \theta) = \sum_{n=1}^{N} \log \sum_{\mathbf{z}_n} p(\mathbf{x}_n, z_n \mid \theta) \tag{13.55}$$

This is hard because of the log-sum over latent variables $z_n \in \{1, \ldots, K\}$. We introduce a variational distribution $q(z_n)$ and apply Jensen's inequality[7] to the log-sum [49, 50, 51]:

$$\log \sum_{z_n} p(\mathbf{x}_n, z_n \mid \theta) = \log \sum_{z_n} q(z_n) \frac{p(\mathbf{x}_n, z_n \mid \theta)}{q(z_n)} \geq \sum_{z_n} q(z_n) \log \frac{p(\mathbf{x}_n, z_n \mid \theta)}{q(z_n)} \tag{13.56}$$

This gives a lower bound $\mathcal{L}(q, \theta)$:

$$\mathcal{L}(q, \theta) = \sum_{z_n} q(z_n) \log \frac{p(\mathbf{x}_n, z_n \mid \theta)}{q(z_n)} \tag{13.57}$$

This is the Evidence Lower Bound (ELBO):

$$\log p(\mathbf{x}_n \mid \theta) \geq \mathbb{E}_{q(z_n)}[\log p(\mathbf{x}_n, z_n \mid \theta)] + \mathsf{H}(q(z_n)) \tag{13.58}$$

Where $\mathsf{H}(q(z_n))$ is the entropy of $q(z_n)$. We now alternate:
E-Step (Optimize q): Set $q(z_n) = p(z_n \mid \mathbf{x}_n; \theta^{\text{old}})$, i.e., the posterior over latent variables using old parameters.

$$q(z_n = k) = \gamma(z_{nk}) = \frac{c_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} c_j \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \tag{13.59}$$

This is the responsibility of component k for point $\mathbf{x}_n$.
M-Step (Optimize θ): Maximize the ELBO w.r.t. $\theta = \{c_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$, using the fixed $q(z_n) = \gamma_{nk}$:
We maximize the expected complete-data log-likelihood. The joint probability of observations $\mathbf{X}$ and latent variables $\mathbf{Z}$ factorizes as:

$$p(\mathbf{X}, \mathbf{Z}) = \prod_{n=1}^{N} p(\mathbf{x}_n, z_n) = \prod_{n=1}^{N} p(z_n)\, p(\mathbf{x}_n \mid z_n), \tag{13.60}$$

where $p(z_n = k) = c_k$ is the prior probability of cluster k (mixing coefficient), and $p(\mathbf{x}_n \mid z_n = k) = \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is the Gaussian likelihood.

[7] Jensen's Inequality (general form) for a concave function (like the logarithm) and a random variable $\mathbf{x}$ is $f(\mathbb{E}[\mathbf{x}]) \geq \mathbb{E}[f(\mathbf{x})]$ where equality holds if $\mathbf{x}$ is deterministic (no randomness).

Thus:

$$\mathbb{E}_{q(z)}[\log p(\mathbf{X}, \mathbf{Z})] = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \log [c_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)], \tag{13.61}$$

where:

$$\gamma_{nk} = p(z_n = k \mid \mathbf{x}_n) = \text{responsibility}$$

This splits into 3 terms. We optimize each one w.r.t. $c_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ in the M-step.

1. Mixing Coefficients c_k

   We want to maximize:

   $$\mathcal{L}_c = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \log c_k \quad \text{subject to} \quad \sum_{k=1}^{K} c_k = 1$$

   Use Lagrange multipliers:

   $$\mathcal{L} = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \log c_k + \lambda \left(1 - \sum_{k=1}^{K} c_k\right)$$

   Take derivative w.r.t. c_k:

   $$\frac{\partial \mathcal{L}}{\partial c_k} = \sum_{n=1}^{N} \frac{\gamma_{nk}}{c_k} - \lambda = 0 \Rightarrow c_k = \frac{1}{\lambda} \sum_{n=1}^{N} \gamma_{nk}$$

   Normalize:

   $$\sum_{k=1}^{K} c_k = 1 \Rightarrow \lambda = N \Rightarrow c_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_{nk}$$

   Update:

   $$\boxed{c_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_{nk}}$$

2. Means $\boldsymbol{\mu}_k$

   We optimize:

$$\mathcal{L}_{\boldsymbol{\mu}} = \sum_{n=1}^{N} \gamma_{nk} \log \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Focus on the log of the Gaussian (ignoring constants):

$$\log \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = -\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

Take derivative w.r.t. $\boldsymbol{\mu}_k$:

$$\frac{\partial \mathcal{L}_{\boldsymbol{\mu}}}{\partial \boldsymbol{\mu}_k} = \sum_{n=1}^{N} \gamma_{nk} \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0$$

Solve:

$$\sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n = \boldsymbol{\mu}_k \sum_{n=1}^{N} \gamma_{nk} \Rightarrow \boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n}{\sum_{n=1}^{N} \gamma_{nk}}$$

Update:

$$\boxed{\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n}{\sum_{n=1}^{N} \gamma_{nk}}}$$

3. Covariance Matrices $\boldsymbol{\Sigma}_k$

   Maximize:

$$\mathcal{L}_{\boldsymbol{\Sigma}} = \sum_{n=1}^{N} \gamma_{nk} \log \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

   Log Gaussian includes:

$$\log \mathcal{N} = -\frac{1}{2} \log |\boldsymbol{\Sigma}_k| - \frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

   Take derivative w.r.t. $\boldsymbol{\Sigma}_k$ and set to zero:

$$\boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} \gamma_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T}{\sum_{n=1}^{N} \gamma_{nk}}$$

Update:

$$\boxed{\boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} \gamma_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T}{\sum_{n=1}^{N} \gamma_{nk}}}$$

4. Compute the log-likelihood:

$$\log p(\mathbf{X}|\Theta) = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} c_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Stop if the change in log-likelihood is below a threshold ϵ or after a maximum number of iterations.

The EM Algorithm Pseudocode for GMM:

Algorithm 13.1 EM Algorithm for GMM

Initialize $\Theta^{(0)} = \{c_k^{(0)}, \boldsymbol{\mu}_k^{(0)}, \boldsymbol{\Sigma}_k^{(0)}\}_{k=1}^K$, using either random values or a clustering algorithm like K-means (see Section 13.3.3).
$t \leftarrow 0$
repeat
  E-Step:
  for $n = 1$ to N **do**
    for $k = 1$ to K **do**
      Compute $\gamma(z_{nk}) = \frac{c_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} c_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$ using current parameters $\Theta^{(t)}$
    end for
  end for
  M-Step:
  for $k = 1$ to K **do**
    Compute $N_k = \sum_{n=1}^{N} \gamma(z_{nk})$
    Update $c_k^{(t+1)} = N_k/N$
    Update $\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n$
    Update $\boldsymbol{\Sigma}_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})^T$
  end for
  $t \leftarrow t + 1$
  Compute $\log p(\mathbf{X}|\Theta^{(t)}) = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} c_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$
until convergence ($|\log p(\mathbf{X}|\Theta^{(t)}) - \log p(\mathbf{X}|\Theta^{(t-1)})| < \epsilon$ or $t > t_{max}$)

13.3.2 Gaussian Mixture Naïve Bayes Classifier

A more accurate Naive Bayes classifier for continuous features can be implemented by modeling each class as a Gaussian Mixture Model (GMM), which captures multi-modal distributions better than a single Gaussian. By representing each class with a mixture of Gaussians, the classifier improves flexibility while maintaining the Naive Bayes assumption of feature independence. This approach allows for more precise probability density estimation, enhancing classification performance on complex datasets. The parameters of the GMM for each class can be efficiently learned using the Expectation-Maximization (EM) algorithm.
The Naïve Bayes assumption enforces that features are conditionally independent given the class and mixture component. Thus, the covariance matrix $\mathbf{\Sigma}_{k,m}$ is diagonal:

$$\mathbf{\Sigma}_{k,m} = \text{diag}(\boldsymbol{\sigma}^2_{k,m,1}, \boldsymbol{\sigma}^2_{k,m,2}, \ldots, \boldsymbol{\sigma}^2_{k,m,d}) \tag{13.62}$$

So, the class-conditional density simplifies to:

$$p(\mathbf{x}|C = k) = \sum_{m=1}^{M_k} c_{k,m} \prod_{j=1}^{d} \mathcal{N}(\mathbf{x}_j|\boldsymbol{\mu}_{k,m,j}, \boldsymbol{\sigma}^2_{k,m,j}) \tag{13.63}$$

Using Bayes' theorem, the posterior probability of class k given $\mathbf{x}$ is:

$$P(C = k|\mathbf{x}) = \frac{p(\mathbf{x}|C = k)\, P(C = k)}{p(\mathbf{x})} \tag{13.64}$$

where the marginal likelihood $p(\mathbf{x})$ is:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}|C = k)\, P(C = k) \tag{13.65}$$

The predicted class $\hat{C}$ is the one that maximizes the posterior probability:

$$\hat{C} = \arg\max_k P(C = k|\mathbf{x}) = \arg\max_k p(\mathbf{x}|C = k)\, P(C = k) \tag{13.66}$$

13.3.3 Connection with K-means

K-Means is an iterative clustering algorithm that partitions data into K distinct clusters by minimizing the within-cluster variance.[8] The objective function (Inertia):

$$J = \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} ||\mathbf{x}_i - \boldsymbol{\mu}_k||^2 \tag{13.67}$$

[8] The K-Means standard algorithm was first proposed by Stuart Lloyd.

where $\mathbf{x}_i \in \mathbb{R}^d$ is a data point, $\boldsymbol{\mu}_k$ is the centroid of cluster k. The $r_{ik} \in \{0, 1\}$ is an indicator:

$$r_{ik} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ is assigned to cluster } k, \\ 0 & \text{otherwise.} \end{cases} \tag{13.68}$$

The Lloyd's algorithm is usedAlgorithm Steps:

1. Initialize K centroids randomly.
2. Assignment Step: Assign each point to the nearest centroid:

$$r_{ik} = \begin{cases} 1 & \text{if } k = \arg\min_j ||\mathbf{x}_i - \boldsymbol{\mu}_j||^2, \\ 0 & \text{otherwise.} \end{cases}$$

3. Update Step: Recompute centroids as the mean of assigned points:

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} r_{ik}\mathbf{x}_i}{\sum_{i=1}^{N} r_{ik}}$$

4. Repeat until convergence (centroids stabilize).

The K-Means algorithm can be viewed as a constrained version of Gaussian Mixture Models (GMM) with specific simplifying assumptions. While GMM employs probabilistic soft assignments $r_{ik} = \frac{c_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j c_j \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$, K-Means uses deterministic hard assignments ($r_{ik} \in \{0, 1\}$) where each point belongs exclusively to one cluster. Furthermore, GMM typically allows for general covariance matrices, whereas K-Means effectively operates with isotropic, spherical clusters of equal variance. Another key difference lies in cluster weighting: GMM learns individual weights for each component, while K-Means treats all clusters as equally important. These restrictions make K-Means a special case that emerges when GMM is constrained to uniform weights $c_k = 1/K$, hard assignments, and spherical covariances with vanishing variance. The key distinctions between K-Means and GMM are concisely presented in Table 13.2. This comparison highlights their fundamental differences in assignment methods, covariance structures, and cluster weighting approaches.

13.3.4 Connection with Deep Neural Networks

To connect the Gaussian Mixture Model (GMM) with deep neural networks, we reformulate it in exponential form, which simplifies interpretation and integration. This representation expresses the GMM's components using linear and quadratic terms, making the model's structure more transparent. By aligning the GMM with the exponential family, we enable seamless compatibility with neural network frameworks.

Table 13.2: Comparison of K-Means and GMM.

Aspect	K-Means	GMM
Assignments	Hard ($r_{ik} \in \{0, 1\}$)	Soft (posterior probabilities)
Covariance	Implicitly $\sigma^2\mathbf{I}$	General $\mathbf{\Sigma}_k$
Cluster Shape	Spherical	Elliptical
Optimization	Lloyd's algorithm	Expectation-Maximization (EM)

Assuming diagonal covariance matrices, $\mathbf{\Sigma}_k = \text{diag}(\mathbf{\Sigma}_{k1}^2, \ldots, \mathbf{\Sigma}_{kd}^2)$, the component k of the Gaussian Mixture Model (GMM) can be expressed in the following form:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \mathbf{\Sigma}_k) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \mathbf{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)\right) \tag{13.69}$$

Expanding the quadratic term:

$$= \exp\left(\sum_{i=1}^{d}\left(\frac{\boldsymbol{\mu}_{ki}}{\mathbf{\Sigma}_{ki}^2}x_i - \frac{1}{2\mathbf{\Sigma}_{ki}^2}\mathbf{x}_i^2\right) - \frac{1}{2}\sum_{i=1}^{d}\left(\frac{\boldsymbol{\mu}_{ki}^2}{\mathbf{\Sigma}_{ki}^2} + \log(2\pi\mathbf{\Sigma}_{ki}^2)\right)\right). \tag{13.70}$$

Then, we express each Gaussian component in exponential form:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \mathbf{\Sigma}_k) = \exp\left(\mathbf{w}_k^T T(\mathbf{x}) - A(\mathbf{w}_k)\right), \tag{13.71}$$

where $\mathbf{w}_k$ = exponential parameters for component k, $T(\mathbf{x}) = [1, x_1, \ldots, x_d, x_1^2, \ldots, x_d^2]^T$ (sufficient statistics), $A(w_k)$ = log-partition function for component k.
Now, match to $\exp(\mathbf{w}_k^T T(x) - A(\mathbf{w}_k))$:

$$\mathbf{w}_k^T T(\mathbf{x}) = w_{k0} \cdot 1 + \sum_{i=1}^{d} w_{ki}x_i + \sum_{i=1}^{d} w_{k(d+i)}x_i^2 \tag{13.72}$$

Hence, the $\mathbf{w}_k$ parameters for each component

1. Bias term (1):
$$w_{k0} = -\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\mu_{ki}^2}{\Sigma_{ki}^2} + \log(2\pi\Sigma_{ki}^2)\right)$$
2. Linear terms (x_i):
$$w_{ki} = \frac{\mu_{ki}}{\Sigma_{ki}^2}, \quad i = 1, \ldots, d$$
3. Quadratic terms (x_i^2):
$$w_{k(d+i)} = -\frac{1}{2\Sigma_{ki}^2}, \quad i = 1, \ldots, d$$

4. Log-partition function:

$$A(\mathbf{w}_k) = -w_{k0}$$

The GMM can be written:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \exp\left(\mathbf{w}_k^T T(\mathbf{x}) - A(\mathbf{w}_k) + \log c_k\right). \tag{13.73}$$

It is possible to recover the original parameters from $\mathbf{w}_k$ ror each component k:

$$\mu_{ki} = -\frac{w_{ki}}{2w_{k(d+i)}}, \quad \Sigma_{ki}^2 = -\frac{1}{2w_{k(d+i)}}, \quad i = 1, \ldots, d$$

The mixture weights c_k remain unchanged (they are not part of w_k).
In the context of multiclass classification, the predicted class $\hat{C}$ is the one that maximizes the posterior probability:

$$\begin{aligned}\hat{C} &= \arg\max_i P(C = i|\mathbf{x}) = \arg\max_k p(\mathbf{x}|C = i)\, P(C = i) \\ &= \arg\max_i \log(p(\mathbf{x}|C = i)) + \log(P(C = i)) \\ &= \arg\max_i \log(\sum_{k=1}^{K} \exp\left(\mathbf{w}_{ik}^T T(\mathbf{x}) - A(\mathbf{w}_{ik}) + \log c_{ik}\right)) + b_i,\end{aligned} \tag{13.74}$$

where the i is the class index and $b_i = \log(P(C = i))$ may be considered as a bias term for each class.
Following Equation 13.74, we visualize the Gaussian Mixture Naïve Bayes Classifier with its single hidden layer structure in Figure 13.6. This contrasts fundamentally with deep neural networks (see Chapter 8), which utilize multiple hidden layers to enable hierarchical feature extraction. While The GMM classifier offers computational efficiency through its shallow architecture, DNNs achieve greater modeling flexibility by composing learned representations across layers. The depth advantage allows DNNs to capture complex nonlinear patterns that often yield superior accuracy, albeit requiring more training data and computational resources. This trade-off positions our single-layer probabilistic model as an interpretable baseline against deeper, more expressive alternatives.

13.4 Sequential Modeling using Hidden Markov Models

Historically, speech recognition systems have frequently utilized Hidden Markov Models (HMMs) to process input speech signals, which are structured as sequences of frames, as outlined in Chapter 3. HMMs are well-suited for modeling the temporal structure of speech data. Their flexibility makes them essential for translating

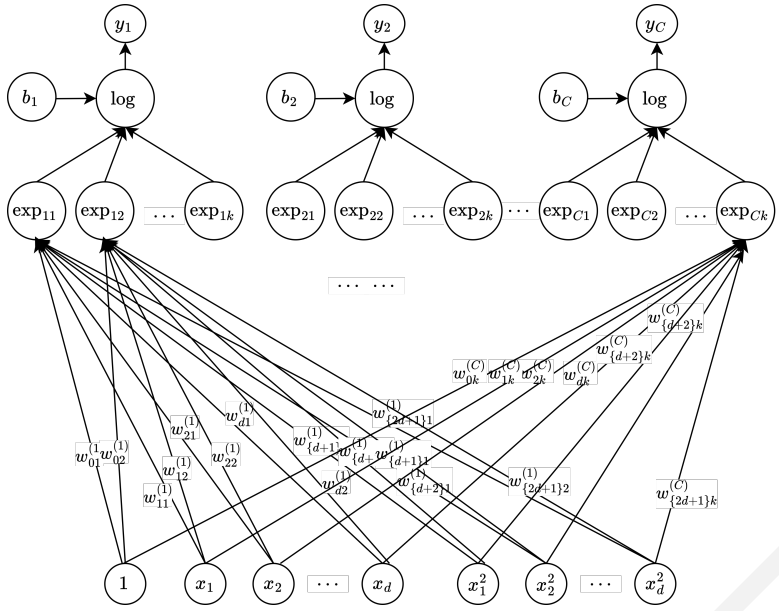


Figure 13.6: The Gaussian Mixture Naïve Bayes Classifier operates in a comparable way to a neural network architecture. The architecture features a single hidden layer with exponential activations, where each class's log prior probability serves as the bias component. Unlike standard neural networks with d-dimensional inputs, our feature vector incorporates both raw features and their squared terms, resulting in twice the dimensionality ($2d$).

spoken input into text. This section provides a summary of HMMs, emphasizing their role in speech recognition. Gaining insight into how they operate in this setting helps us recognize their value in handling audio signals. Therefore, we examine HMMs through the lens of speech recognition to demonstrate their real-world usefulness.

Automatic speech recognition systems, also known as speech-to-text systems, are the core technology for man-machine interface. These systems aim to find the most likely word sequence given acoustic observations collected from a speech signal. Using statistical methods [52], speech recognition can be defined as a problem of choosing a word sequence $\hat{\mathbf{W}}$ with the *maximum a posterior* (MAP) criterion given a time sequence of speech frames T or acoustic observations associated an utterance $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T) \in \mathbb{R}^{TxD}$ where D is the dimensionality of the acoustic vector:

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{W}} P(\mathbf{W}|\mathbf{X}) \tag{13.75}$$

Using Bayes's rule, Equation 13.75 can be written as

$$P(\mathbf{W}|\mathbf{X}) = \frac{p(\mathbf{X}|\mathbf{W})P(\mathbf{W})}{p(\mathbf{X})} \propto p(\mathbf{X}|\mathbf{W})P(\mathbf{W}) \tag{13.76}$$

where $p(\mathbf{X}|\mathbf{W})$ is the likelihood of the acoustic observations given by an *acoustic model* and $P(\mathbf{W})$ is the likelihood of the hypothesized word sequence given by a *language model* (see Chapter 11).

A Hidden Markov Model is a stochastic finite state machine [5]. An example of an HMM with left-to-right transition topology, which is used to model a phone in an acoustic model, is shown in figure 13.7. This model has one entry state, three emitting states, and one exit state. The left-to-right topology imposes prior information, where speech production is sequential in time.

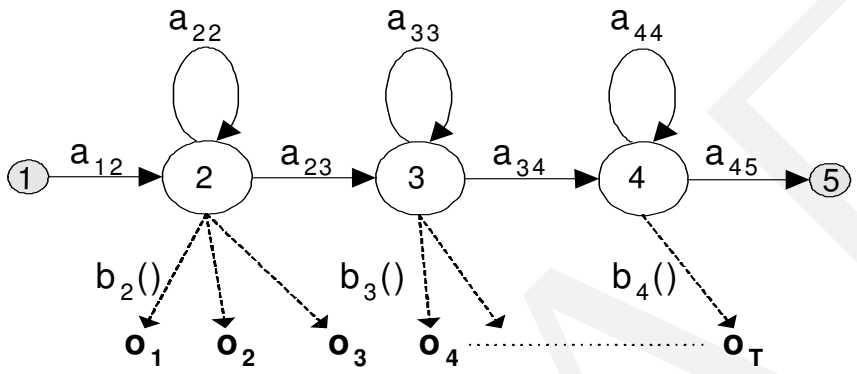


Figure 13.7: A typical Hidden Markov Model for a phone.

For every observation at time t, a jump from the current state i to some new state j is allowed with a transition probability:

$$a_{ij} = P(\mathbf{s}_{t+1} = j|\mathbf{s}_t = i) \tag{13.77}$$

where $\sum_j^N a_{ij} = 1$, N is the number of states in the HMM model. An acoustic feature vector $\mathbf{x}_t$ may be generated, with an output probability density function $b_j(\mathbf{x}_t)$, which is associated with state j. A mixture of Gaussian distributions is typically used to model the output distribution for each state,

$$b_j(\mathbf{x}_t) = \sum_{m=1}^{M} c_{jm}\mathcal{N}(\mathbf{x}_t; \mu_{jm}, \Sigma_{jm}) \tag{13.78}$$

where M is the number of mixture components, c_{jm} is the component weight and $\sum_m^M c_{jm} = 1$. μ_{jm} and Σ_{jm} are the component specific mean vector and covariance matrix respectively. If the acoustic features are statistically independent, then diagonal covariance matrices are used to compute the likelihood of a Gaussian model,

$$\mathcal{N}(\mathbf{x}_t; \mu_{jm}, \Sigma_{jm}) = \prod_{d=1}^{D} \frac{1}{\sqrt{(2c)\sigma_{jmd}}} \exp - \frac{(\mathbf{x}_{td} - \mu_{jmd})^2}{2\sigma_{jmd}^2} \tag{13.79}$$

where σ_{jmd} is the variance element of the Gaussian component m for dimension d. An HMM can be written in terms of a set of parameters Λ,

$$\Lambda = \{a_{ij}, c_{jm}, \mu_{jm}, \Sigma_{jm}\} \tag{13.80}$$

HMM model estimation is based on *two* assumptions that lead to a tractable inference when computing the likelihood $p(\mathbf{X}|\mathcal{M})$ of the observation sequence, $\mathbf{X}$, given a model $\mathcal{M}$. Although the HMM is successful as an acoustic model because of these assumptions, they are also its main limitations. The first assumption is the *Markov* assumption, which approximates, or factorizes, the probability of the *hidden* state sequence $\mathbf{S} = \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_T$ given a model $\mathcal{M}$ by a first order Markov chain:

$$P(\mathbf{S}|\mathcal{M}) = \prod_{t=1}^{T} P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \prod_{t=1}^{T} a_{\mathbf{s}_t \mathbf{s}_{t-1}} \tag{13.81}$$

The second assumption is the *conditional independence* assumption, where the probability of an observation sequence, $\mathbf{X}$, given a state sequence, $\mathbf{S}$ and a model $\mathcal{M}$ is given by

$$p(\mathbf{X}|\mathbf{S}, \mathcal{M}) = \prod_{t=1}^{T} p(\mathbf{x}_t|\mathbf{s}_t) = \prod_{t=1}^{T} b_{\mathbf{s}_t}(\mathbf{x}_t), \tag{13.82}$$

Since the state sequence is hidden, the *total probability* or likelihood of the acoustic observations $p(\mathbf{X}|\mathcal{M})$ is expressed as a sum over all possible state sequences:

$$p(\mathbf{X}|\mathcal{M}) = \sum_{\mathbf{S}} p(\mathbf{X}|\mathbf{S}, \mathcal{M}) P(\mathbf{S}|\mathcal{M}), \tag{13.83}$$

which can be efficiently computed using a dynamic programming algorithm given the factorizations in Equation 13.81 and Equation 13.82. The summation over all possible state sequences in Equation 13.83 can be approximated by a maximum operation to find the best state sequence $\hat{\mathbf{S}}$:

$$\hat{\mathbf{S}} = \arg\max_{\mathbf{S}} p(\mathbf{X}|\mathbf{S}, \mathcal{M}) P(\mathbf{S}|\mathcal{M}) \tag{13.84}$$

which is known as the *Viterbi* path. This gives the best alignment of acoustic observations with the states of an HMM. The Viterbi algorithm (i.e. Algorithm 13.2)is a dynamic programming approach used to find the most likely sequence of hidden states $\hat{\mathbf{S}}$ in a Hidden Markov Model (HMM) given an observation sequence $\mathbf{X}$ and model parameters $\mathcal{M}$.

Algorithm 13.2 Viterbi Algorithm for Left-to-Right HMM

Require: Observation sequence $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$
Require: HMM $\mathcal{M} = (\mathbf{A}, \mathbf{B}$ with:
- Non-emitting start state s_0
- Non-emitting end state s_{N+1}
- Left-to-right transitions ($a_{ij} = 0$ for $j < i$)

Ensure: Most likely state sequence $\hat{\mathbf{S}} = (s_1^*, \ldots, s_T^*)$

Initialization:
for all emitting states i **do**
  $\delta_1(i) \leftarrow \cdot b_i(\mathbf{x}_1)$
  $\psi_1(i) \leftarrow s_0$
end for
Forward Pass:
for $t \leftarrow 2$ **to** T **do**
  for all emitting states j **do**
    $\delta_t(j) \leftarrow \max_i[\delta_{t-1}(i) \cdot a_{ij}] \cdot b_j(\mathbf{x}_t)$
    $\psi_t(j) \leftarrow \text{argmax}_i[\delta_{t-1}(i) \cdot a_{ij}]$
  end for
end for
Termination:
$P^* \leftarrow \max_j[\delta_T(j) \cdot a_{j,s_{N+1}}]$
$s_T^* \leftarrow \text{argmax}_j[\delta_T(j) \cdot a_{j,s_{N+1}}]$
Backtracking:
$\hat{\mathbf{S}} \leftarrow [s_T^*]$
for $t \leftarrow T - 1$ **downto** 1 **do**
  $s^* \leftarrow \psi_{t+1}(\hat{\mathbf{S}}[0])$
  $\hat{\mathbf{S}} \leftarrow [s^*] + \hat{\mathbf{S}}$
end for
return $\hat{\mathbf{S}}$

13.4.1 Generative Parameter Estimation

Parameters of HMMs can be estimated using the maximum likelihood estimate (MLE) framework. For R training observations $\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_r, \ldots, \mathbf{X}_R\}$ with cor-

responding transcriptions $\{w_r\}$, the MLE objective function is given by

$$\mathcal{F}_{\text{MLE}}(\Lambda) = \sum_{r=1}^{R} \log p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_{w_r}) \tag{13.85}$$

where $\mathcal{M}_{w_r}$ is the composite model corresponding to the reference word sequence w_r.

The parameters can be estimated using iterative Baum-Welch algorithm, also known as the *forward-backward* algorithm [19]. The Baum-Welch algorithm is a special case of the Expectation-Maximization (EM) algorithm, which is an efficient iterative procedure to perform MLE in the presence of hidden variables [53]. The inference of an HMM is based on computing the forward and backward probabilities. The forward probabilities (see Algorithm 13.3) can be computed recursively:

$$\alpha_j(t) = p(\mathbf{x}_1, \ldots, \mathbf{x}_t, \mathbf{s}_t = j|\mathcal{M}) = \Big(\sum_{i=2}^{N-1} \alpha_i(t-1) a_{ij} \Big) b_j(\mathbf{x}_t) \tag{13.86}$$

with initial conditions $\alpha_1(1) = 1$ and $\alpha_j(1) = a_{1j} b_j(\mathbf{x}_1)$ for $1 < j < N$ and a final condition $\alpha_N(T) = \sum_{i=2}^{N-1} \alpha_i(T) a_{iN}$.

Algorithm 13.3 Forward Algorithm for HMM Probability Computation

Require: Observation sequence $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$
Require: HMM $\mathcal{M} = (\mathbf{A}, \mathbf{B})$ with:
- Non-emitting start state s_1
- Non-emitting end state s_N
- N total states (including non-emitting states)

Ensure: Total observation probability $P(\mathbf{X}|\mathcal{M})$

Initialization:
$\alpha_1(1) \leftarrow 1$ ▷ Start state probability
for $j \leftarrow 2$ **to** $N-1$ **do** ▷ Emitting states
    $\alpha_j(1) \leftarrow a_{1j} \cdot b_j(\mathbf{x}_1)$
end for
Recursive Computation:
for $t \leftarrow 2$ **to** T **do**
    for $j \leftarrow 2$ **to** $N-1$ **do** ▷ For each emitting state
        $\alpha_j(t) \leftarrow \left(\sum_{i=2}^{N-1} \alpha_i(t-1) \cdot a_{ij}\right) \cdot b_j(\mathbf{x}_t)$
    end for
end for
Termination:
$\alpha_N(T) \leftarrow \sum_{i=2}^{N-1} \alpha_i(T) \cdot a_{iN}$ ▷ End state probability
$p(\mathbf{X}|\mathcal{M}) \leftarrow \alpha_N(T)$ ▷ Total observation probability
return $p(\mathbf{X}|\mathcal{M})$

Similarly, the backward probabilities can be computed see Algorithm 13.4:

$$\beta_j(t) = p(\mathbf{x}_{t+1}, \ldots, \mathbf{x}_T | \mathbf{s}_t = j, \mathcal{M}) = \sum_{i=2}^{N-1} a_{ji} b_j(\mathbf{x}_{t+1}) \beta_i(t+1) \tag{13.87}$$

with initial conditions $\beta_i(T) = a_{iN}$ for $1 < i < N$ and a final condition $\beta_1(1) = \sum_{j=2}^{N-1} a_{1j} b_j(\mathbf{x}_1) \beta_j(1)$.

Algorithm 13.4 Backward Algorithm for HMM

Require: Observation sequence $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$
Require: HMM $\mathcal{M} = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$ with:
- Non-emitting start state s_1
- Non-emitting end state s_N
- N total states (including non-emitting states)

Ensure: Backward probabilities $\beta_j(t)$ for all states and times

Initialization:
for $i \leftarrow 2$ **to** $N-1$ **do** ▷ Emitting states
  $\beta_i(T) \leftarrow a_{iN}$ ▷ Transition to end state
end for
$\beta_1(T) \leftarrow 1$ ▷ Start state at time T
$\beta_N(T) \leftarrow 1$ ▷ End state at time T
Recursive Computation:
for $t \leftarrow T-1$ **downto** 1 **do**
  for $j \leftarrow 1$ **to** N **do** ▷ All states
    $\beta_j(t) \leftarrow \sum_{i=2}^{N-1} a_{ji} \cdot b_i(\mathbf{x}_{t+1}) \cdot \beta_i(t+1)$
  end for
end for
Final Computation:
$\beta_1(1) \leftarrow \sum_{j=2}^{N-1} a_{1j} \cdot b_j(o_1) \cdot \beta_j(1)$
return β ▷ All backward probabilities

To describe transitions between state i at time t and state j at time $t+1$, we define:

$$\xi_{ij}(t) = P(s_t = i, s_{t+1} = j \mid \mathbf{X}; \mathcal{M}) = \frac{\alpha_i(t) \cdot a_{ij} \cdot b_j(\mathbf{x}_{t+1}) \cdot \beta_j(t+1)}{P(\mathbf{X} \mid \mathcal{M})}, \tag{13.88}$$

where a_{ij} is the transition probability from state i to j and $b_j(\mathbf{x}_{t+1})$ is the observation likelihood in state j at time $t+1$. The transition matrix $A = [a_{ij}]$ is updated by estimating the expected number of transitions from state i to j over the expected number of transitions from state i to any state:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)} \tag{13.89}$$

In addition, the frame-state alignment probability γ_j, denoting the probability of being in state j at some time t can be written in terms of the forward probability $\alpha_j(t)$ and the backward probability $\beta_j(t)$:

$$\gamma_j(t) = P(\mathbf{s}_t = j|\mathbf{X}; \mathcal{M}) = \frac{p(\mathbf{X}, \mathbf{s}_t = j|\mathcal{M})}{p(\mathbf{X}|\mathcal{M})} = \frac{\alpha_j(t)\beta_j(t)}{p(\mathbf{X}|\mathcal{M})} \tag{13.90}$$

where

$$p(\mathbf{X}|\mathcal{M}) = \alpha_N(T) = \beta_1(1) \tag{13.91}$$

and a component specific alignment probability can be derived:

$$\gamma_{jm}(t) = P(\mathbf{s}_t = j, \mathbf{m}_t = m|\mathbf{X}; \mathcal{M}) = \gamma_j(t)\frac{c_{jm}b_{jm}(\mathbf{x}_t)}{b_j(\mathbf{x}_t)} \tag{13.92}$$

where the m^{th} component is associated with the j^{th} state.
Consequently, the accumulators of the sufficient statistics $\mathcal{C}_{jm}(1) = \gamma_{jm}$, $\mathcal{C}_{jm}(\mathbf{X})$, and $\mathcal{C}_{jm}(\mathbf{X}^2)$ are calculated as follows:

$$\mathcal{C}_{jm}(1) = \sum_{r=1}^{R}\sum_{t=1}^{T_r}\gamma_{jm}^r(t) \tag{13.93}$$

$$\mathcal{C}_{jm}(\mathbf{X}) = \sum_{r=1}^{R}\sum_{t=1}^{T_r}\gamma_{jm}^r(t)\mathbf{x}_t \tag{13.94}$$

$$\mathcal{C}_{jm}(\mathbf{X}^2) = \sum_{r=1}^{R}\sum_{t=1}^{T_r}\gamma_{jm}^r(t)\mathbf{x}_t^2 \tag{13.95}$$

Hence, the Baum-Welch re-estimation formulae for the mean and covariance of state j and component m of an HMM are given by

$$c_{jm} = \frac{\mathcal{C}_{jm}(1)}{\sum_m \mathcal{C}_{jm}(1)} \tag{13.96}$$

$$\mu_{jm} = \frac{\mathcal{C}_{jm}(\mathbf{X})}{\mathcal{C}_{jm}(1)} \tag{13.97}$$

$$\Sigma_{jm} = \frac{\mathcal{C}_{jm}(\mathbf{X}^2)}{\mathcal{C}_{jm}(1)} - \mu_{jm}^2 \tag{13.98}$$

The transition probabilities between states are also estimated by calculating the forward and backward probabilities.
Generative training of HMM models leads to models that may be useful for generating speech, which is useful for speech synthesis. Using Bayes rule as in Equation 13.75, these generative models can be used for speech recognition. Although HMM models should be trained to discriminate between speech classes, it is not uncommon that generative training is the basic training method in speech recognition. This is related to the basic fact that generative training of HMM models is fast and efficient because:

- The Maximization step of the EM algorithm for Gaussian models inherits the closed form of Gaussian's mean and variance estimation from the data. This

attractive property may be the main reason behind the widespread of HMM models.

- Maximum likelihood generative training accumulates statistics from the correct class only (i.e. it does not use out-of-class data for discrimination). This leads to a fast training for speech recognition, where the data is split according to classes and trained independently. However, this advantage is a common property of generative training.

- Controlling the number of Gaussian mixtures usually leads to coarse generative modeling, which is usually very effective for modeling the spectral information related to discrimination. Modeling the fine structure of the spectrum may lead to poor discrimination.

HMMs trained by a generative training procedure maximize the likelihood between the data and the underlying distributions. However, if the true underlying distribution that generated the data is an HMM, given sufficient data, the Bayes classification based on the HMM models, will minimize the probability of classification/recognition error [54]. Practically, the decision boundaries constructed after the generative training are not optimal and generative HMMs are not optimal models for speech recognition applications. One way to address this problem within the HMM framework is to utilize the parameters efficiently to improve the discrimination between speech classes via discriminative training for HMM models [55, 56].

13.4.2 Discriminative Parameter Estimation

HMM models trained using the EM algorithm are very effective for coarse generation of data. Unfortunately, generative training does not address the classification problem, where the objective is to discriminate between the classes and hence to reduce the misclassification error. To address this problem, the Gaussians of an HMM can be rotated and shifted in the feature space to increase the discrimination between classes via a discriminative training procedure.
The Conditional Maximum Likelihood (CML) criterion, defined by Equation 13.99, aims to maximize the log of posterior probability of the correct word sequence given the observations,

$$\begin{aligned}\mathcal{F}_{\text{CML}}(\Lambda) &= \sum_{r=1}^{R} \log P_{\Lambda}(\mathcal{M}_{w_r}|\mathbf{X}_r) \\ &= \sum_{r=1}^{R} \log \frac{p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_{w_r})P(w_r)}{\sum_{\hat{w}} p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_{\hat{w}})P(\hat{w})} \\ &\approx \sum_{r=1}^{R} \log p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_r^{\text{num}}) - \log p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_r^{\text{den}})\end{aligned} \tag{13.99}$$

where $\mathcal{M}_w$ is a composite model corresponding to the word sequence w and $P(w)$ is the probability of this sequence as determined by a language model. This discriminative training aims to maximize a term related to the probability of the correct models (known as the numerator) $p_{\Lambda}(\mathbf{X}_r|\mathcal{M}^{\text{num}})$, which is identical to the ML objective function, and simultaneously minimize a term related to all incorrect models probabilities (known as the denominator term) $p_{\Lambda}(\mathbf{X}_r|\mathcal{M}^{\text{den}}) \approx \sum_{\hat{w}} p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_{\hat{w}})P(\hat{w})$. $\sum_{\hat{w}} p_{\Lambda}(\mathbf{X}_r|\mathcal{M}_{\hat{w}})P(\hat{w})$, which is the summation over all possible word sequences $\hat{w}$ allowed in the task, is computationally expensive for LVCSR systems. As a result, $p_{\Lambda}(\mathbf{X}_r|\mathcal{M}^{\text{den}})$ is an approximation to the denominator term, which is computed by N-best lists [57] or lattices [58, 59] generated from a decoding pass based on MLE trained models.

Extended Baum-Welch (EBW) algorithm is the state-of-the-art discriminative training algorithm that maximize the CML criterion for HMMs.[9] It was introduced for discriminative training for discrete distributions in [63]. Using a discrete approximation to the Gaussian distribution [64], it was shown that the mean of a particular dimension of the Gaussian for state j, mixture component m, μ_{jm} and the corresponding variance, σ_{jm}^2 (assuming diagonal covariance matrices) can be reestimated by

$$\hat{\mu}_{jm} = \frac{\mathcal{C}_{jm}^{\text{num}}(\mathbf{X}) - \mathcal{C}_{jm}^{\text{den}}(\mathbf{X}) + D\mu_{jm}}{\gamma_{jm}^{\text{num}} - \gamma_{jm}^{\text{den}} + D} \tag{13.100}$$

$$\hat{\sigma}_{jm}^2 = \frac{\mathcal{C}_{jm}^{\text{num}}(\mathbf{X}^2) - \mathcal{C}_{jm}^{\text{den}}(\mathbf{X}^2) + D(\mu_{jm}^2 + \sigma_{jm}^2)}{\gamma_{jm}^{\text{num}} - \gamma_{jm}^{\text{den}} + D} \tag{13.101}$$

In these equations, D is a smoothing constant that controls the degree of deviation of the new parameters with respect to the old parameters. The superscripts *num* and *den* refer to the model corresponding to the correct word sequence, and the recognition model for all word sequences, respectively. Figure 13.8 shows the

[9] For numerical optimization based methods see [60, 61]. Given an appropriate setting for learning parameters and smoothing terms, the EBW and gradient ascent algorithms can be equivalent [62].

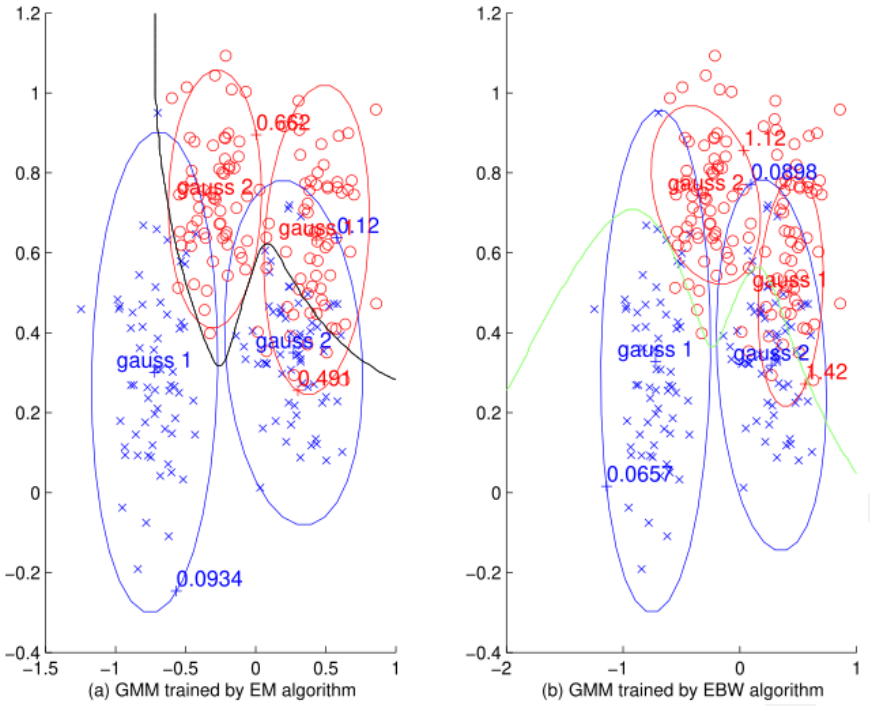


Figure 13.8: Two class classification problem. (a) decision boundary is constructed with EM generative training (b) decision boundary is constructed by EBW discriminative training.

decision boundary for a simple two class classification problem, where the Gaussians are shifted and rotated to improve the discrimination between classes. This may explain the basic idea behind the EBW update for Gaussian models. It may be important to mention that Gaussian models estimated by discriminative training are not generative models. They are simply activation functions that have the same functional form of a Gaussian generative model and the same probabilistic constraint (i.e. $\int_{\mathbf{x}} f(\mathbf{x}|\mu, \Sigma)d\mathbf{x} = 1$). Similarly, HMM models trained using discriminative procedures are not generative models or distributions.
Setting the optimal value for D is the subject of extensive research and it is usually set per-Gaussian level, D_{jm}, given the formula

$$D_{jm} = \max\{2D_{jm}^{min}, E\gamma_{jm}^{den}\}, \tag{13.102}$$

where D_{jm}^{min} is a necessary value to ensure positive variances and E is a global constant set to 1 or 2 [65]. It has been shown that there is a value of D, which proves the convergence of the algorithm [66] and [67]. Using the reverse Jensen inequality

for e-family distributions [68], a closed form expression for D_{jm} was derived and the heuristic in Equation 13.102 was justified [69]. The discriminative training of HMM models is usually initialized by ML generative training. For historical reasons, CML discriminative training for HMMs is known as Maximum Mutual Information Estimation (MMIE) in speech recognition domain. The two criteria lead to the same results because the language model parameters are not optimized during the training.

Discriminative training based on the CML objective function does not directly measure the expected WER criterion. Instead, the Overall Risk Criterion Estimation (ORCE) [70] directly minimizes the expected word or phone error rates by refining the model parameters based on a measure of risk related to recognition error. The update equations of the parameters for ORCE was shown to be very similar to the EBW update equations described above for CML [71]. Minimum Phone Error (MPE) criterion may be considered as a particular realization of ORCE and it is given by

$$\mathcal{F}_{\text{MPE}}(\Lambda) = \sum_{r=1}^{R} \sum_{w} P_{\Lambda}(\mathcal{M}_w|\mathbf{X}_r) A(w, w_r) \tag{13.103}$$

where $A(w, w_r)$ is the raw phone transcription accuracy of the sentence w, given the reference sentence w_r. It has been reported that ORCE based on MPE criterion gives a small improvement over ORCE based on Minimum Word Error (MWE) criterion [72, 73]. Alternatively, the Minimum Classification Error (MCE) criterion [74] may be used to update the parameters of HMMs [75, 76, 77, 78]. Some discriminative criteria have been compared in a unified framework for some tasks [79, 80]. To match the training and decoding criteria, ORCE based criteria can also be used for decoding tasks since they directly minimize the expected word error rate [81, 82].

Recognition accuracy can be significantly increased by increasing the number of hidden states in the HMM. We refer to this process as *augmenting the state space*, which aims to increase the capacity of observation distributions. This is usually done by using context-dependent HMM models like tri-phone, quad-phone, or penta-phones, which use a window of left and right neighboring phones. The process of augmenting the state space increases dramatically the number of parameters, which need to be robustly estimated given the limited amount of training data and unseen context. Parameter tying allows acoustically similar units to share the same parameters. Extensive research has been done on clustering the augmented state space based on tied, context-dependent phonetic units to reduce model complexity given limited training data [83, 84, 85, 86].

13.5 Hybrid NN/HMM models

In hybrid NN/HMM speech recognition systems [87], Neural Networks (NN) models are used as flexible discriminant classifiers to estimate a scaled likelihood. In particular, the emission probability score is given by

$$b_j(\mathbf{x}_t) = \frac{P_\Lambda(\mathbf{s}_j|\mathbf{x}_t)}{P(\mathbf{s}_j)} \tag{13.104}$$

where $P_\Lambda(\mathbf{s}_j|\mathbf{x}_t)$ is the posterior probability of a phonetic state estimated by a NN estimator [88] and $P(\mathbf{s}_j)$ is estimated from the labeled data. In addition to discriminative training, if the posterior probability $P_\Lambda(\mathbf{s}_j|\mathbf{x}_t)$ is sensitive to acoustic context, $b_j(\mathbf{x}_t)$ score may help to overcome conditional independence assumption and improve the overall recognition performance without changing the basic HMM framework.

14. Generalization

So clearly we don't actually care about training error, we don't really care about making accurate predictions on the training set, or at a least that's not the ultimate goal. The ultimate goal is how well it makes – generalization – how well it makes predictions on examples that we haven't seen before.

— Andrew Ng

This chapter discusses various techniques to enhance a model's ability to generalize to new data. Methods like regularization, cross-validation, and dropout are highlighted for reducing overfitting. Additionally, techniques such as batch normalization and layer normalization are explored to improve training stability and performance. These approaches help models maintain accuracy on unseen data while preventing them from fitting noise in the training set. By employing these methods, the chapter shows how to build more robust models. Each technique is explained in terms of its contribution to better generalization.

14.1 Model Complexity

Model complexity refers to the capacity of a machine learning model to capture patterns in data, which typically increases with the number of parameters or the flexibility of the model. A simple model, like a linear regression with only a few parameters, is considered low in complexity. In contrast, more complex models, such as deep neural networks or high-degree polynomial regressions, can capture intricate relationships due to their flexibility and a large number of parameters. While increasing model complexity allows the model to better fit the training data, it also increases the risk of overfitting, which may hinder the model's ability to

generalize to new, unseen data. Thus, finding the right level of complexity is crucial for building models that perform well on both training and test sets.

Underfitting occurs when a model is too simple and fails to capture the underlying structure or patterns in the data. This typically happens when the model has too few parameters or is not complex enough to accommodate the true relationships within the dataset. In this case, the model produces high errors on both the training and testing datasets because it cannot adequately describe the data, leading to poor performance. An underfitting model is characterized by a large bias, where it assumes that the relationship between input and output is overly simplistic, resulting in missed nuances or variations that the model should be able to capture.

Overfitting happens when a model becomes too complex and starts to learn not only the patterns in the training data but also the noise and minor fluctuations that are irrelevant to the true underlying relationships. In this situation, the model fits the training data almost perfectly, resulting in very low training error, but it struggles to generalize to new data, leading to higher test errors. Overfitting is common in models with too many parameters or in cases where the model is overly sensitive to the specifics of the training dataset (i.e. high variance where a model learns not only the underlying patterns in the training data but also the noise and random fluctuations.). The key issue with overfitting is that while the model appears to perform well during training, it fails to maintain this performance when applied to unseen examples because it has essentially "memorized" the training set rather than learning a generalizable pattern.

Figure 14.1 illustrates the phenomena of underfitting and overfitting. It visually demonstrates how these issues affect model performance in relation to complexity.

To avoid both underfitting and overfitting, it's crucial to find an optimal balance of model complexity. A well-balanced model should be complex enough to capture the important patterns in the data but not so complex that it starts modeling noise. Techniques like cross-validation, regularization (such as Ridge or Lasso) are often used to control the complexity of models and ensure they generalize well. This involves testing the model's performance on both training and validation datasets to detect overfitting and underfitting issues and adjust the model accordingly. The goal is to select a model with sufficient complexity to accurately capture the relationships in the data while ensuring it performs well on unseen data, achieving a good trade-off between bias and variance.

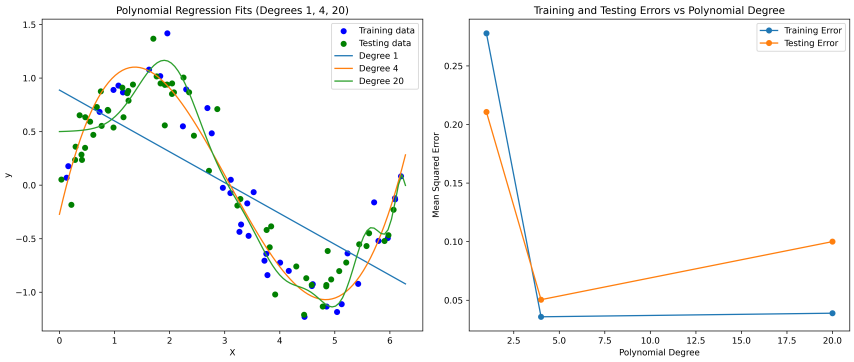


Figure 14.1: There are two side-by-side plots: one illustrating polynomial regression fits for degrees 1, 4, and 20, and the other displaying the corresponding training and testing errors. In the first plot, degree 1 represents **underfitting**, where the model is too simple (low complexity) and fails to capture the underlying pattern, leading to high error on both training and testing data. The degree 4 model demonstrates an appropriate balance between complexity and fit, capturing the data's pattern well without overfitting. The degree 20 model, however, shows **overfitting**, where the high complexity allows the model to fit noise in the training data, producing a very complex curve. In the second plot, the training error decreases as the model's complexity increases, but the testing error initially decreases and then increases for higher degrees, showing poor generalization in the overfitting case. This behavior emphasizes that a model that is either too simple or too complex can result in poor predictive performance.

Mathematically, assume we have a dataset with input features x and a corresponding output Y that we want to predict. The true relationship between x and Y can be represented as:

$$Y = f(x) + \epsilon \tag{14.1}$$

where $f(x)$ is the true underlying function that maps input x to the output Y and ϵ is the noise term, representing the irreducible error in our observations, has a mean of zero $\mathbb{E}[\epsilon] = 0$ and $\text{Var}(\epsilon) = \sigma^2$.
Now, if we take the expected value of Y:

$$\mathbb{E}[Y] = \mathbb{E}[f(x) + \epsilon] = f(x) + \mathbb{E}[\epsilon] = f(x)$$

This shows that the expected output Y is equal to the true function $f(x)$.
When we evaluate a model $\hat{f}(x)$ that attempts to predict Y, we can express the mean squared error (MSE) for a linear regression problem as:

$$\text{MSE} = \mathbb{E}[(Y - \hat{f}(x))^2]$$

Substituting $Y = f(x) + \epsilon$ into the MSE gives us:

$$\text{MSE} = \mathbb{E}[((f(x) + \epsilon) - \hat{f}(x))^2]$$

Expanding this expression, we have:

$$\text{MSE} = \mathbb{E}[(f(x) + \epsilon - \hat{f}(x))^2]$$

Using the identity $(a + b)^2 = a^2 + 2ab + b^2$, we get:

$$\text{MSE} = \mathbb{E}[(f(x) - \hat{f}(x))^2] + \mathbb{E}[\epsilon^2] + 2\mathbb{E}[(f(x) - \hat{f}(x))\epsilon]$$

Evaluating the Terms

1. First Term:

$$\begin{aligned}
\mathbb{E}[(f(x) - \hat{f}(x))^2] &= \mathbb{E}\Big[\big(f(x) - \mathbb{E}[\hat{f}(x)] + \mathbb{E}[\hat{f}(x)] - \hat{f}(x)\big)^2\Big] \\
&= \mathbb{E}\Big[(f(x) - \mathbb{E}[\hat{f}(x)])^2\Big] + \mathbb{E}\Big[(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))^2\Big] + 2\mathbb{E}\Big[(f(x) - \mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))\Big] \\
&= f(x)^2 + 2f(x)\mathbb{E}[\hat{f}(x)] + \mathbb{E}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + 2\mathbb{E}\Big[(f(x) - \mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))\Big] \\
&= (f(x) - \mathbb{E}[\hat{f}(x)])^2 + \text{Var}[\hat{f}(x)] + 2\mathbb{E}\Big[(f(x) - \mathbb{E}[\hat{f}(x)])(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))\Big] \\
&= \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + 2\mathbb{E}\Big[(f(x)\mathbb{E}[\hat{f}(x)] - f(x)\hat{f}(x) - \mathbb{E}[\hat{f}(x)]^2 + \mathbb{E}[\hat{f}(x)]\hat{f}(x)\Big] \\
&= \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + 2f(x)\mathbb{E}[\hat{f}(x)] - 2f(x)\mathbb{E}[\hat{f}(x)] - 2\mathbb{E}[\hat{f}(x)]^2 + 2\mathbb{E}[\hat{f}(x)]^2 \\
&= \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + 0 \\
&= \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)],
\end{aligned} \tag{14.2}$$

   where $\text{Bias}[\hat{f}(x)]^2 = (f(x) - \mathbb{E}[\hat{f}(x)])^2$ and $\text{Var}[\hat{f}(x)] = \mathbb{E}\Big[(\mathbb{E}[\hat{f}(x)] - \hat{f}(x))^2\Big] = \mathbb{E}\Big[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\Big]$.

2. Second Term: The expected value of the squared noise is:

$$\mathbb{E}[\epsilon^2] = \sigma^2$$

3. Third Term: The covariance term $\mathbb{E}[(\hat{f}(x) - f(x))\epsilon]$ is zero, assuming the noise is independent of the model's predictions:

$$\mathbb{E}[(\hat{f}(x) - f(x))\epsilon] = 0$$

Putting it all together, we have:

$$\text{MSE} = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2 \tag{14.3}$$

Thus, the irreducible noise error, represented by σ^2, is the part of the total error that cannot be reduced by improving the model. This derivation highlights the role of noise in the context of expected mean squared error in a predictive model.

14.2 Regularization

This technique combats overfitting by adding a penalty term to the model's loss function, discouraging overly complex models and promoting simpler and more generalized representations. Techniques like L0, L1, and L2 regularization help to control model complexity and prevent overfitting.

14.2.1 L0 Regularization

L0 regularization directly influences the number of features or weights that remain active, forcing many weights to be exactly zero. Unlike L1 or L2 regularization, which provide smooth, differentiable loss functions, L0 regularization is inherently "non-differentiable" because it operates based on counting non-zero elements, not their magnitudes.

Because of this non-differentiability, standard gradient-based optimization techniques, such as Stochastic Gradient Descent (SGD), are not applicable. Instead, finding an optimal solution for L0 regularization involves combinatorial optimization, where the goal is to search through all possible combinations of which weights should be kept or pruned.

In mathematical terms, the objective is to solve an optimization problem like:

$$\text{E}_{\text{reg}}(\mathbf{w}) = \text{E}(\mathbf{w}) + \lambda \sum_{i=1}^{n} \mathbf{1}(w_i \neq 0)$$

Here, we have to evaluate many possible subsets of non-zero weights, which leads to an exponential search space. The number of possible subsets of features or weights is 2^n for n weights, making this problem NP-hard. Each possible subset of weights needs to be evaluated in terms of how it contributes to minimizing the overall loss function.

Consider a simple model with 3 weights $\mathbf{w} = [w_1, w_2, w_3]$. The L0 regularization problem involves determining whether each of these weights should be included in the model (non-zero) or pruned (set to zero). The possible combinations of non-zero weights are:

$[w_1, w_2, w_3]$, $[w_1, w_2, 0]$, $[w_1, 0, w_3]$, $[0, w_2, w_3]$, $[w_1, 0, 0]$, $[0, w_2, 0]$, $[0, 0, w_3]$, $[0, 0, 0]$.

This example shows how the number of possible configurations grows exponentially with the number of weights. For each of these combinations, the model needs to evaluate the loss function, making the optimization process extremely computationally intensive.
In conclusion, L0 regularization requires combinatorial optimization because the problem involves selecting the best combination of non-zero weights from an exponentially large set of possibilities.

14.2.2 L1 Regularization (Lasso)

L1 regularization encourages sparsity in the weight vector by penalizing the sum of the absolute values of the weights. The modified loss function with L1 regularization is given by::

$$E_{reg}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_{i=1}^{n} |w_i|$$

To compute the gradient of the L1 regularizer, we need to differentiate the absolute value term. The gradient of $|w_i|$ with respect to w_i is:

$$\frac{d}{dw_i}|w_i| = \begin{cases} 1 & \text{if } w_i > 0 \\ -1 & \text{if } w_i < 0 \\ \text{undefined} & \text{if } w_i = 0 \end{cases}$$

Thus, the gradient of the L1 regularization term is:

$$\frac{\partial}{\partial w_i}(\lambda \mathbf{w}_1) = \lambda \cdot \text{sign}(w_i)$$

Where $\text{sign}(w_i)$ is the sign function that outputs 1, -1, or 0 based on whether w_i is positive, negative, or zero, respectively.

14.2.3 L2 Regularization (Ridge)

L2 regularization adds a penalty proportional to the square of the weights. The regularized loss function for L2 is given by:

$$E_{reg}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \sum_{i=1}^{n} w_i^2$$

The gradient of the L2 regularization term with respect to w_i is:

$$\frac{\partial}{\partial w_i}\left(\frac{\lambda}{2} \sum_{i=1}^{n} w_i^2\right) = \lambda w_i$$

Thus, the gradient of L2 regularization is proportional to the weight value itself. A summary of Regularizers

Regularization	Effect on Weights	Feature Selection	Sparsity
L0	Limits number of non-zero weights	Yes	Yes
L1 (Lasso)	Drives weights to zero	Yes	Yes
L2 (Ridge)	Shrinks weights but keeps all features	No	No

In conclusion, L1, L2, and L0 regularizations each provide a method to control model complexity, with different trade-offs in promoting sparsity and computational stability. L1 encourages sparsity, L2 shrinks weights while retaining all features, and L0 selects a minimal subset of features.

14.2.4 Label Smoothing

Label smoothing is a regularization technique used in classification tasks to improve model generalization by modifying the target labels. Instead of using hard labels (0s and 1s), label smoothing assigns a small probability to incorrect classes, effectively softening the target distribution. By mathematically redistributing the label's probability mass, it enhances the model's ability to generalize to unseen data.
Let's explore the mathematics of label smoothing in detail, focusing on the formulation, the loss function, and the implications for training. In a multiclass classification problem with C classes, we represent the true label using a one-hot encoded vector $\mathbf{t}$:

$$\mathbf{t} = [0, 1, 0, \ldots, 0]$$

Label smoothing modifies this one-hot vector to create a smoothed target vector $\tilde{\mathbf{t}}$:

$$\tilde{\mathbf{t}} = (1 - \epsilon)\mathbf{t} + \frac{\epsilon}{C}\mathbf{1},$$

where ϵ is the smoothing parameter (e.g., 0.1) and $\mathbf{1}$ is a vector of ones of size C.
For Example, Assuming $C = 3$ (three classes) and $\epsilon = 0.1$:

$$\mathbf{t} = [0, 1, 0],$$

and after applying label smoothing:

$$\tilde{\mathbf{t}} = (1 - 0.1)[0, 1, 0] + \frac{0.1}{3}[1, 1, 1] = [0 + 0.033, 0.9, 0 + 0.033] = [0.033, 0.9, 0.033]$$

The categorical cross-entropy loss for a predicted probability distribution $\mathbf{y}$ is defined as:

$$E(\mathbf{t}, \mathbf{y}) = -\sum_{i=1}^{C} t_i \log(y_i),$$

and when we use the smoothed labels, this loss function becomes:

$$E(\tilde{\mathbf{t}}, \mathbf{y}) = -\sum_{i=1}^{C} \tilde{t}_i \log(y_i)$$

Substituting the smoothed label $\tilde{t}_i$:

$$E(\tilde{\mathbf{t}}, \mathbf{y}) = -\sum_{i=1}^{C} \left((1-\epsilon)t_i + \frac{\epsilon}{C}\right) \log(y_i)$$

This can be separated into two terms:

$$E(\tilde{\mathbf{t}}, \mathbf{y}) = -\sum_{i=1}^{C} (1-\epsilon)t_i \log(y_i) - \sum_{i=1}^{C} \frac{\epsilon}{C} \log(y_i)$$

The first term resembles the standard cross-entropy loss but is scaled by $(1-\epsilon)$, which reduces the contribution of the true class. The second term adds a penalty for all classes, distributing some loss across incorrect classes, effectively encouraging the model to assign non-zero probabilities to them. During training, if the model becomes overly confident in predicting the distribution, the first term will approach zero while the second term will rise significantly. Thus, label smoothing effectively acts as a regularizer, helping to prevent the model from making overly confident predictions. In other words, the second term minimizes the entropy between a uniform distribution over classes and the predicted distribution, encouraging less confident predictions. By softening the target labels, label smoothing reduces the model's confidence in its predictions, which can help prevent overfitting to noise in the training data.

Models trained with label smoothing often produce more calibrated probability distributions, meaning their predicted probabilities better reflect the true likelihoods of the classes. In addition, the modified loss function results in smoother gradients, which can lead to more stable training dynamics, especially in complex models like deep neural networks. This technique has been shown to be effective in improving the performance of models on various tasks.

14.3 Dropout Regularization

Dropout is a regularization technique used to improve the generalization performance of deep neural networks by preventing overfitting [90]. It works by randomly

"dropping out" units (neurons) from the neural network during training. This means that, at each training step, a subset of neurons is ignored, and the network is forced to adapt without relying on specific neurons. As a result, the network becomes more robust and generalizes better to unseen data. Dropout introduces noise into the training process, which can be viewed as a form of regularization, as highlighted in Bishop's work [91]. This noise, introduced by randomly deactivating neurons, forces the network to generalize better by preventing it from relying too heavily on specific neurons. This mechanism of adding noise helps avoid overfitting, making dropout an implicit regularizer for neural networks.

In a fully connected neural network, each neuron in a given layer is connected to every neuron in the next layer as shown in Figure 14.2. During training, dropout randomly disables a fraction p (referred to as the dropout rate) of neurons in each layer with probability p, which effectively means these neurons do not participate in forward or backward propagation during that iteration. This is typically done by multiplying the activations of these neurons by 0.

Mathematically, if $\mathbf{h}_i$ is the output of neuron i in some layer during training, the dropout step can be described as:

$$\mathbf{h}_i = r_i \cdot \tilde{\mathbf{h}}_i, \tag{14.4}$$

where r_i is a random binary variable (1 with probability $1-p$ and 0 with probability p) and $\tilde{\mathbf{h}}_i$ is the original activation of the neuron before dropout.

During training, the network learns not to rely on any specific neuron but instead to distribute the learned representations across many neurons. This reduces overfitting because, during inference (when dropout is not applied), the model can perform well on unseen data by leveraging all neurons.

During the inference phase (after training), dropout is no longer applied. To account for the fact that more neurons are active during inference than during training, the weights of the neurons are scaled by a factor of $1-p$. This ensures that the overall activations remain approximately the same as during training. If $\mathbf{w}_i$ is the weight of neuron i, then during inference, we use:

$$\mathbf{h}_i = (1-p) \cdot \tilde{\mathbf{h}}_i \tag{14.5}$$

This compensates for the fact that neurons were randomly dropped during training. Dropout helps improve generalization by:

- Reducing Overfitting By randomly dropping units, dropout prevents the network from over-relying on specific neurons, forcing it to learn more robust features that generalize well to unseen data.
- Implicit Ensemble Method Dropout can be viewed as training a large number of different networks (by randomly dropping neurons) and averaging them, which

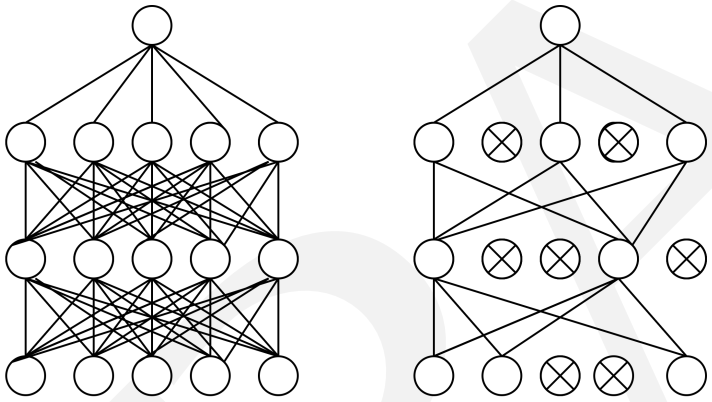
acts as a form of model ensembling. This boosts the network's robustness.

- Improving Neuron Independence: By ensuring that no neuron can fully dominate the decision-making process, dropout promotes independence between neurons. This leads to learning representations that are less correlated and more useful for generalization.

Consider a toy neural network with three input neurons and two hidden layers, each with four neurons. Applying dropout with $p = 0.5$ means that, at each training iteration, approximately half of the neurons in each hidden layer will be randomly dropped. This leads to various subnetworks being trained at each iteration, preventing over-reliance on particular pathways.
The overall impact of dropout can be mathematically summarized by modifying the loss function with dropout applied:

$$\mathrm{E}_{\text{dropout}} = \mathbb{E}_r[\mathrm{E}(\mathbf{w}, r)] \tag{14.6}$$

Where r represents the random dropout masks applied to the neurons. The expectation over all possible dropout masks ensures that the network learns a more generalized solution. In conclusion, dropout is an effective regularization technique that enhances a model's ability to generalize by preventing overfitting, effectively acting as an implicit ensemble of networks.



(a) Standard feedforward neural network (b) After applying dropout

Figure 14.2: Dropout regularization: On the left, a conventional feedforward neural network with two hidden layers is shown. On the right, a reduced version of the same network is depicted, where dropout has been applied. The crossed-out neurons indicate the units that have been removed during training. The illustration closely resembles the one shown in the referenced paper [90].

14.4 Normalization

The Batch Normalization (BatchNorm) and Layer Normalization (LayerNorm) also introduce slight variations in the input values, which can be seen as introducing noise during training. This added noise acts as a form of regularization [91], helping to improve the model's generalization ability by preventing overfitting in a manner similar to other regularization techniques. They already discussed in Chapter 11.

14.4.1 Batch Normalization

Batch Normalization is applied during training to normalize the activations of each layer across a mini-batch of data. It does this by adjusting the mean and variance of the activations, which helps in stabilizing and speeding up training. Mathematically, BatchNorm normalizes a batch of data as:

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sigma_{\text{batch}}}, \tag{14.7}$$

where μ_{batch} is the mean of the batch and σ_{batch} is the standard deviation of the batch.
It then introduces learnable parameters, γ (scale) and β (shift), allowing the network to maintain the capacity to learn features that are beneficial to the task at hand.
Batch normalization helps smooth the optimization surface, making the training process more stable and allowing the model to converge more quickly. In addition, BatchNorm introduces some noise to the network due to the small fluctuations between mini-batches. This acts as a regularizer, similar to dropout, reducing the chances of overfitting. Moreover, by normalizing the activations, BatchNorm reduces the issue of vanishing or exploding gradients, which can help deeper networks generalize better by stabilizing the training dynamics.

14.4.2 Layer Normalization

Layer Normalization normalizes across the features of a single sample rather than across the batch [92]. It's more commonly used in recurrent neural networks (RNNs) and Transformers because it doesn't depend on batch size, making it more suited for sequential data.
The normalized output for LayerNorm is:

$$\hat{x}_i = \frac{x_i - \mu_{\text{layer}}}{\sigma_{\text{layer}}}, \tag{14.8}$$

where μ_{layer} is the mean of the activations across all the features in a layer and σ_{layer} is the standard deviation of the activations across all the features in a layer.

Since LayerNorm doesn't depend on the batch size, it provides consistent normalization regardless of how data is split into batches. This consistency can improve generalization, especially in settings like natural language processing (NLP) where input sequence lengths vary. For sequential data like in RNNs and Transformers, LayerNorm is better suited than BatchNorm as it ensures stable gradients throughout the sequence. This stability helps the network generalize better across various sequence lengths. The difference between Layer Normalization and Batch Normalization is illustrated in Figure 14.3.

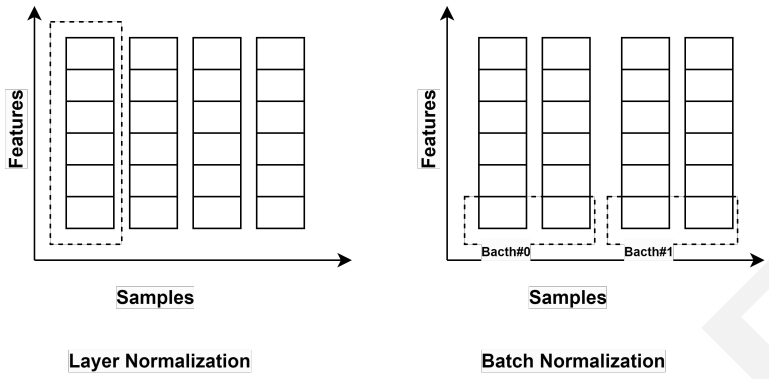


Figure 14.3: Layer Normalization and batch Normalization both stabilize and speed up neural network training, but they differ in how they normalize inputs. Layer Normalization normalizes across the feature dimension for each individual sample, making it more suited for tasks where batch size is small or variable. In contrast, Batch Normalization normalizes across the batch dimension, meaning it computes the mean and variance for each feature across a mini-batch, which introduces dependencies between samples in the batch.

In conclusion, BatchNorm tends to improve generalization more effectively in convolutional neural networks (CNNs) and feed-forward networks by reducing internal covariate shifts[1] and helping the network learn more robust features. LayerNorm, on the other hand, is effective in NLP models, such as Transformers, where data dependencies are sequential or vary in size. It improves generalization by ensuring smoother and more stable training dynamics.

[1]Internal Covariate Shift refers to the phenomenon where the distribution of inputs to each layer of a neural network changes during training, due to updates in the parameters of the preceding layers. This shift happens because as weights are updated in each training step, the output of a layer (which is the input to the next layer) changes its distribution. This shift can slow down training since the model has to constantly adapt to the changing input distribution [93].

14.4.3 Root Mean Square Normalization

Root Mean Square Layer Normalization (RMSNorm) is often used as a replacement for LayerNorm (Layer Normalization) because it simplifies the normalization process and reduces computational overhead [94].
RMSNorm simplifies this process by skipping the mean subtraction and using only the root mean square (RMS) of the input. We first compute the Root Mean Square:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}, \tag{14.9}$$

where d is the dimension of $\mathbf{x}$ vector. Then, the input is normalized by dividing each element by the RMS value:

$$\hat{x}_i = \frac{x_i}{\text{RMS}(\mathbf{x})} \tag{14.10}$$

Similar to LayerNorm, a learnable scale parameter γ is applied to the normalized values:

$$\mathbf{y}_i = \gamma \hat{x}_i \tag{14.11}$$

RMSNorm is computationally more efficient than LayerNorm since it skips the calculation of the mean and variance, reducing the number of operations involved in normalization. In addition, by focusing only on the magnitude (via the RMS), RMSNorm simplifies the regularization process, potentially improving training dynamics in certain models (e.g., Transformers). Moreover, RMSNorm introduces less variance in the gradients during training, which can help improve stability in some cases.
The gradient stability of RMSNorm with respect to LayerNorm can be explained as follows: LayerNorm normalizes across the features of a single training example by subtracting the mean and dividing by the standard deviation. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, the normalized output $\hat{x}_i$ for each element x_i is computed as:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{14.12}$$

where $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ is the mean of the input vector, $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$ is the variance, and ϵ is a small constant added for numerical stability.
The gradient of μ with respect to x_j is:

$$\frac{\partial \mu}{\partial x_j} = \frac{1}{d}, \tag{14.13}$$

and the gradient of Variance σ^2:

$$\frac{\partial \sigma^2}{\partial x_j} = \frac{\partial \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}{\partial x_j} = \frac{2}{d}(x_j - \mu) \tag{14.14}$$

Using the quotient rule, the gradient of $\hat{x}_i$ with respect to x_j involves both the mean and variance:

$$\frac{\partial \hat{x}_i}{\partial x_j} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \delta_{ij} - \frac{x_i - \mu}{(\sigma^2 + \epsilon)^{3/2}} \frac{\partial \sigma^2}{\partial x_j} - \frac{1}{\sqrt{\sigma^2 + \epsilon}} \frac{\partial \mu}{\partial x_j} \tag{14.15}$$

Substitute the gradients of μ and σ^2:

$$\begin{aligned} \frac{\partial \hat{x}_i}{\partial x_j} &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \delta_{ij} - \frac{(x_i - \mu)(x_j - \mu)}{n(\sigma^2 + \epsilon)^{3/2}} - \frac{1}{n\sqrt{\sigma^2 + \epsilon}} \\ &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \left(\delta_{ij} - \frac{1}{n} - \frac{(x_i - \mu)(x_j - \mu)}{n(\sigma^2 + \epsilon)} \right) \end{aligned} \tag{14.16}$$

This expression has additional complexity because of the terms involving the input mean μ and variance σ^2, introducing correlations between the input dimensions. On the other hand, the gradient of RMS with respect to x_j is:

$$\frac{\partial \text{RMS}(\mathbf{x})}{\partial x_j} = \frac{x_j}{n \cdot \text{RMS}(\mathbf{x})} \tag{14.17}$$

Using the quotient rule, the gradient of the normalized output, $\hat{x}_i$, with respect to x_j is:

$$\frac{\partial \hat{x}_i}{\partial x_j} = \frac{\partial}{\partial x_j} \left(\frac{x_i}{\text{RMS}(\mathbf{x})} \right) = \frac{1}{\text{RMS}(\mathbf{x})} \delta_{ij} - \frac{x_i}{\text{RMS}^2(\mathbf{x})} \cdot \frac{\partial \text{RMS}(\mathbf{x})}{\partial x_j} \tag{14.18}$$

Substituting the gradient of RMS:

$$\begin{aligned} \frac{\partial \hat{x}_i}{\partial x_j} &= \frac{1}{\text{RMS}(\mathbf{x})} \delta_{ij} - \frac{x_i x_j}{n \cdot \text{RMS}^3(\mathbf{x})} \\ &= \frac{1}{\text{RMS}(\mathbf{x})} \left(\delta_{ij} - \frac{x_i x_j}{n\,\text{RMS}^2(\mathbf{x})} \right) \end{aligned} \tag{14.19}$$

The gradient stability is clearer in RMSNorm where only the RMS is used for normalization, which simplifies the gradient. The absence of the mean subtraction results in less interaction between the input elements, thus reducing the variance in the gradients. This can lead to more stable training, especially in deeper networks. In LayerNorm, the presence of the mean μ and variance σ^2 introduces more complex interactions between the elements of the input, leading to higher gradient variance, especially in deep networks. The gradient contains terms involving both the input mean and the variance.

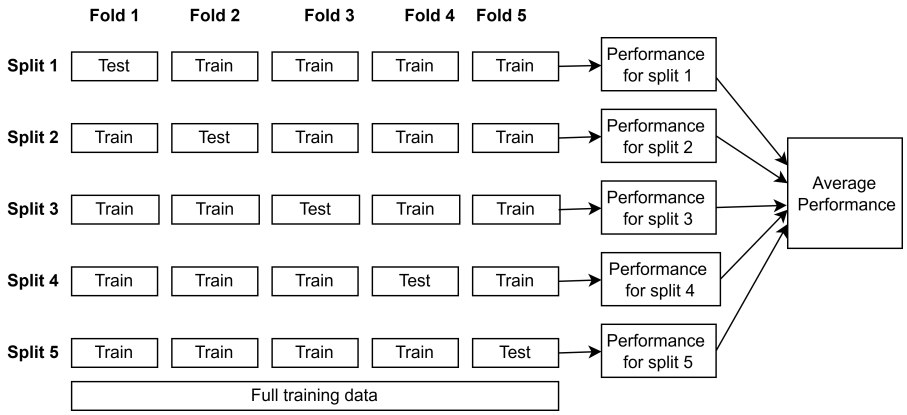


Figure 14.4: In K-fold cross-validation, the dataset is divided into k equal parts or folds. The model is trained on k-1 folds while the remaining fold is used for evaluation. This process is repeated k times, each time using a different fold as the validation set. Finally, the performance scores from all iterations are averaged to obtain the overall model performance.

14.5 Cross-Validation

Cross-Validation is a powerful technique used to estimate a model's performance on unseen data. It involves splitting the available data into multiple subsets, training the model on a portion of the data, and evaluating its performance on the remaining test set. Cross-validation provides a more robust estimation of the model's generalization ability, helping in model selection and hyperparameter tuning.
A common approach of cross-validation is of K-fold cross-validation which is a statistical technique used to assess a model's generalization performance by partitioning the dataset into several subsets (or folds) and then performing multiple training and validation rounds. This approach provides a more robust estimate of model performance than a simple train-test split, especially when the dataset is small or imbalanced. Here's a detailed description, followed by the mathematical basis for why it improves generalization.
The K-fold cross-validation is implemented as follows (see Figure 14.4):

1. Partitioning The dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ is divided into K approximately equal-sized folds: $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$.

2. Training and Validation: For each fold k (where $k = 1, 2, \ldots, K$):
   - Use fold $\mathcal{D}_k$ as the validation set.
   - Use the remaining $K - 1$ folds as the training set.

3. Evaluation: Compute the performance (e.g., accuracy, loss) for each validation fold $\mathcal{D}_k$, resulting in K performance scores.

4. Average Performance: The overall performance of the model is estimated by averaging the K validation scores:

$$\text{Performance}_{\text{avg}} = \frac{1}{K}\sum_{k=1}^{K}\text{Performance}_k.$$

This approach allows each data point to serve as part of both training and validation sets, providing a more comprehensive measure of model performance.
Let $\hat{R}_k(\mathbf{w})$ be the empirical risk (e.g., the validation error) on fold k of the K-fold cross-validation. Then, the overall K-fold cross-validation estimate $\hat{R}_{\text{CV}}(\mathbf{w})$ is:

$$\hat{R}_{\text{CV}}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\hat{R}_k(\mathbf{w}). \tag{14.20}$$

The variance of $\hat{R}_{\text{CV}}(\mathbf{w})$ is given by:

$$\text{Var}(\hat{R}_{\text{CV}}(\mathbf{w})) = \text{Var}\left(\frac{1}{K}\sum_{k=1}^{K}\hat{R}_k(\mathbf{w})\right). \tag{14.21}$$

By the properties of variance, if we assume the risks $\hat{R}_k(\mathbf{w})$ have a common variance σ^2 and pairwise covariances $\rho\sigma^2$ (where ρ is the correlation coefficient between different folds), the variance can be expanded as:

$$\text{Var}(\hat{R}_{\text{CV}}(\mathbf{w})) = \frac{1}{K^2}\sum_{k=1}^{K}\text{Var}(\hat{R}_k(\mathbf{w})) + \frac{1}{K^2}\sum_{i\neq j}\text{Cov}(\hat{R}_i(\mathbf{w}), \hat{R}_j(\mathbf{w})). \tag{14.22}$$

This becomes:

$$\text{Var}(\hat{R}_{\text{CV}}(\mathbf{w})) = \frac{1}{K}\sigma^2 + \frac{K(K-1)}{K^2}\rho\sigma^2. \tag{14.23}$$

Simplifying, we get:

$$\text{Var}(\hat{R}_{\text{CV}}(\mathbf{w})) = \frac{\sigma^2}{K}(1 + (K-1)\rho). \tag{14.24}$$

If $\rho = 0$ (folds are independent), this reduces to $\frac{\sigma^2}{K}$, showing the ideal variance reduction. But in practice, folds are not independent, so ρ typically has a positive value, and the actual reduction in variance is less than $\frac{\sigma^2}{K}$.
In summary, K-fold cross-validation enhances generalization by reducing the variability in model performance estimates by averaging over multiple folds (i.e. variance reduction) and providing a comprehensive assessment of model performance by ensuring each data point is used in both training and validation, allowing the model to generalize better to unseen data (i.e. more robust estimation or bias reduction).

14.6 Data Augmentation

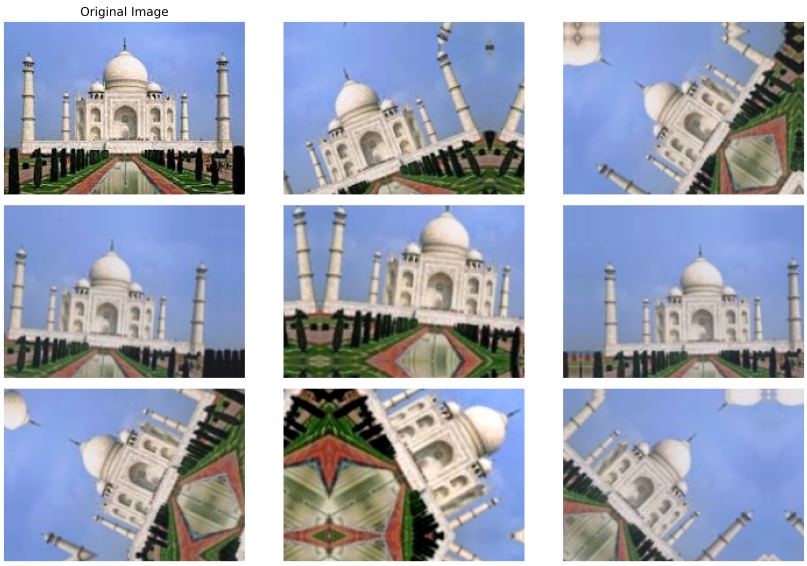


Figure 14.5: Image augmentation involves applying a series of random transformations to the original image, such as horizontal flipping, rotation, zooming, translation, and contrast adjustment.

Data augmentation is a technique used to improve generalization in machine learning models by artificially increasing the diversity of the training data. Mathematically, it can be viewed as a way to reduce overfitting by expanding the dataset to better approximate the true data distribution.
Consider a simple linear model $f(x; \mathbf{w}) = \mathbf{w}^T x$ with a mean squared error (MSE) loss. For a dataset augmented with Gaussian noise, where $\tilde{x}_i = x_i + \epsilon$ and $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, the loss over the augmented data can be written as:

$$E_{\text{aug}}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I)} \left[(y_i - \mathbf{w}^T (x_i + \epsilon))^2 \right]. \tag{14.25}$$

Expanding this expression:

$$E_{\text{aug}}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \left((y_i - \mathbf{w}^T x_i)^2 + \sigma^2 \mathbf{w}^T \mathbf{w} \right). \tag{14.26}$$

This augmented loss now includes a regularization term $\sigma^2 \mathbf{w}^T \mathbf{w}$, which prevents large weights and promotes generalization (see Section 14.2.3).

In summary, data augmentation enhances generalization by introducing an implicit regularization term in the loss function, reducing the variance of the model's predictions without adding significant bias, and more closely approximating the true data distribution, thus lowering the risk of overfitting. A sample of data augmentation for an image is shown in Figure 14.5.

14.7 Ensemble Methods

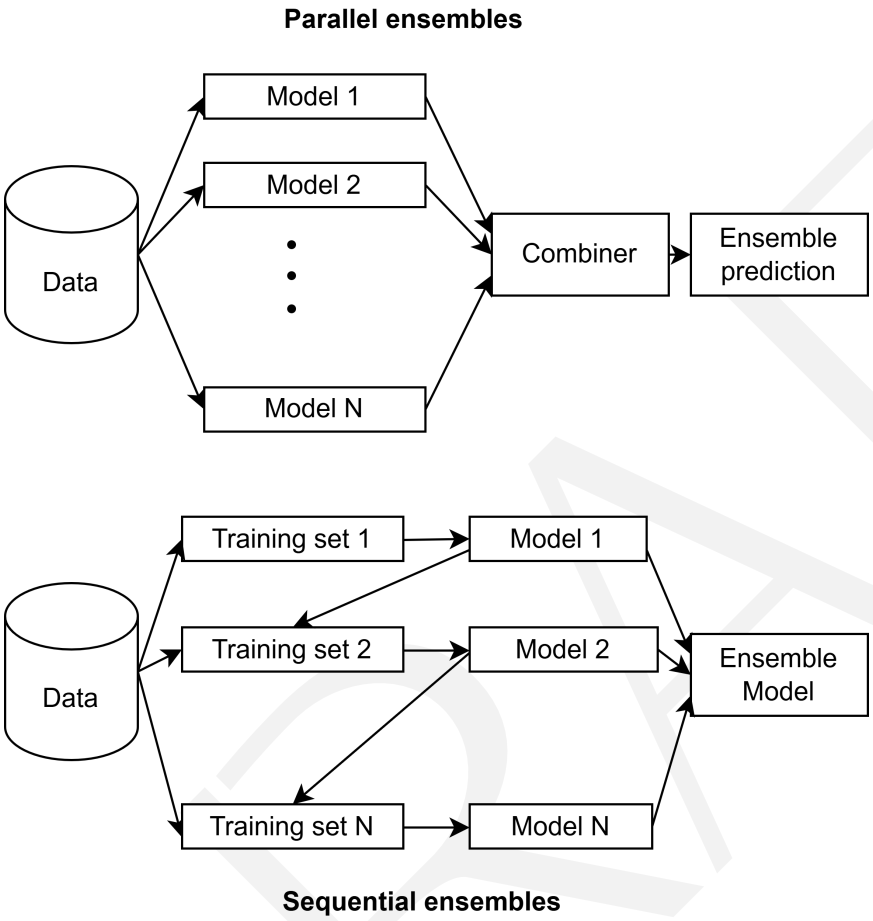


Figure 14.6: Parallel methods train models independently and simultaneously, while sequential methods build each new model based on the errors of the previous model in a step-by-step process.

Ensemble methods improve generalization by combining predictions from multiple models to reduce variance, bias, or both, leading to a more robust predictor. The key intuition is that aggregating the outputs of several models minimizes the risk of overfitting to particular patterns in a dataset, as each model provides a unique perspective. As shown in Figure 14.6, Ensemble methods can be classified into two types: parallel methods (e.g. bagging), which train models independently and simultaneously, and sequential methods (e.g. boosting), which build each new model incrementally by addressing the errors of the previous model.

In bagging (i.e. Bootstrap Aggregating), several models are trained independently on different bootstrap samples[2] of the training data, and their outputs are averaged or voted upon to produce a final prediction. Majority voting aggregates predictions from multiple models and selects the most common prediction as the final output. Hence, this approach enhances predictive accuracy by leveraging the strengths of multiple models.

Suppose we have M independent models $f_1(x), f_2(x), \ldots, f_M(x)$ trained on bootstrapped samples of the dataset, each predicting $\hat{y}_i$ for an input x. For regression, the ensemble prediction is the mean:

$$\hat{y}_{\text{ensemble}} = \frac{1}{M} \sum_{m=1}^{M} f_m(x). \tag{14.27}$$

Assuming each model has variance σ^2 and they are uncorrelated, the variance of the ensemble prediction is given by:

$$\text{Var}(\hat{y}_{\text{ensemble}}) = \frac{1}{M} \sigma^2. \tag{14.28}$$

This reduction in variance improves generalization, as the ensemble is less sensitive to fluctuations in any single training set compared to individual models.

Boosting reduces bias by training a sequence of models where each model attempts to correct errors made by the previous ones. This can be particularly beneficial for weak models, such as a simple linear regression model that can be weak for complex relationships. Let the final model $F(x)$ be a weighted sum of M models $f_1(x), f_2(x), \ldots, f_M(x)$:

$$F(x) = \sum_{m=1}^{M} \alpha_m f_m(x), \tag{14.29}$$

where α_m are weights that depend on the model's performance in predicting the data. For each iteration m, the model $f_m(x)$ is trained to minimize the loss on the

[2] Bootstrap samples are randomly drawn subsets of a dataset created by sampling with replacement. When generating bootstrap samples, each original data point has a chance to be selected multiple times, or not at all, in a single sample.

residuals from the previous model. This iterative correction reduces bias at each step. Hence, boosting aims to achieve a lower overall error, improving generalization on unseen data. By combining models, ensemble methods reduce variance (bagging) or bias (boosting), thereby improving generalization. Each model's prediction errors complement each other, resulting in a more stable and accurate ensemble model.

Bibliography

[1] R. G. Gallager. *Information Theory and Reliable Communication*. Wiley, 1968.

[2] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1991.

[3] David MacKay. *Information Theory, Pattern Recognition, and Neural Networks*. Cambridge University Press, 2003.

[4] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, jul , oct 1948.

[5] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. of IEEE*, 77(2):257–286, February 1989.

[6] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[7] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics, 2015.

[8] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[9] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis, 2021.

[10] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of IEEE*, 86(11):2278–2324, 1998.

[12] Ning Qian. On the momentum term in gradient descent learning algorithms. In *Neural networks: the official journal of the International Neural Network Society*, volume 12, pages 145–151, 1999.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7):2121–2159, 2011.

[14] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a: overview of mini-batch gradient descent, 2012.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[16] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven C. H. Hoi, and Weinan E. Towards theoretically understanding why SGD generalizes better than ADAM in deep learning. *CoRR*, abs/2010.05627, 2020.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[18] S.B. Davis and P. Mermelstein. Comparison of parametric representation for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, 1980.

[19] Lawrence R. Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[20] Ben Gold and Nelson Morgan. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. Wiley, 1999.

[21] S. Furui. Speaker independent isolated word recognizer using dynamic features of the speech spectrum. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(1):52–59, Feb 1986.

[22] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[23] E.T. Jaynes. On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, 70(9):939–952, 1982.

[24] Cs231n convolutional neural networks for visual recognition. 2018. https://cs231n.github.io/convolutional-networks/.

[25] A comprehensive tutorial to learn convolutional neural networks from scratch (deeplearning.ai course #4). 2018. https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/.

[26] Machine learning practical course. 2020. https://www.inf.ed.ac.uk/teaching/courses/mlp/2019-20/lectures/mlp08-cnn2.pdf.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[29] Brett Koonce and Brett Koonce. Resnet 50. *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*, pages 63–72, 2021.

[30] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[31] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[32] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[33] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In *International Conference on Artificial Neural Networks*, pages 799–804. Springer, 2005.

[34] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[35] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[36] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn, 2018.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[39] Irving J Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4):237–264, 1953.

[40] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

[41] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

[42] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

[44] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[45] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[46] Albert Gu. *Modeling Sequences with Structured State Spaces*. Stanford University, 2023.

[47] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[48] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024.

[49] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.

[50] Jeff A Bilmes et al. A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. *International computer science institute*, 4(510):126, 1998.

[51] Maya R Gupta, Yihua Chen, et al. Theory and use of the em algorithm. *Foundations and Trends® in Signal Processing*, 4(3):223–296, 2011.

[52] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, 1997.

[53] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[54] A. Nadas. A decision theoretic formulation of a training problem in speech recognition and a comparison of training by unconditional versus conditional maximum likelihood. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 31(4):814–817, 1983.

[55] Lalit R. Bahl, Peter F. Brown, Peter V. de Souza, and Robert L. Mercer. Maximum mutual information estimation of hidden Markov model parameters for speech recognition. In *Proc. IEEE ICASSP*, pages 49–52, Tokyo, Japan, 1986.

[56] Peter F. Brown. *The Acoustic-Modelling Problem in Automatic Speech Recognition*. PhD thesis, Carnegie Mellon University, 1987.

[57] Yen-Lu Chow. Maximum Mutual Information estimation of HMM parameters for continuous speech recognition using the N-best algorithm. In *Proc. IEEE ICASSP*, pages 701–704, Albuquerque, NM, April 1990.

[58] Y. Normandin, R. Lacouture, and R. Cardin. MMIE training for large vocabulary continuous speech recognition. In *Proc. ICSLP*, pages 1367–1370, Yokohama, Japan, 1994.

[59] V. Valtchev, J. J. Odell, P. C. Woodland, and S. J. Young. MMIE training of large vocabulary speech recognition systems. *Speech Communication*, 22(4):303–314, 1997.

[60] S. Kapadia, V. Valtchev, and S.J. Young. MMI training for continuous phoneme recognition on the TIMIT database. In *Proc. IEEE ICASSP*, volume 2, pages 491–494, Minneapolis, MN, USA, April 1993.

[61] Sadik Kapadia. *Discriminative Training of Hidden Markov Models*. PhD thesis, University of Cambridge, 1998.

[62] R. Schlüter, W. Macherey, S. Kanthak, H. Ney, and L. Welling. Comparison of optimization methods for discriminative training criteria. In *Proc. EUROSPEECH*, pages 15–18, Rhodes, Greece, 1997.

[63] P. S. Gopalakrishnan, D. Kanevsky, A. Nadas, and D. Nahamoo. An inequality for rational function with applications to some statistical estimation problems. *IEEE Transactions on Information Theory*, 37(1):107–113, Jan 1991.

[64] Y. Normandin. *Hidden Markov Models, Maximum Mutual Information Estimation and the Speech Recognition Problem*. PhD thesis, McGill University, 1991.

[65] P. Woodland and D. Povey. Large scale discriminative training for speech recognition. In *ISCA ITRW Automatic Speech Recognition: Challenges for the Millenium*, pages 7–16, 2000.

[66] A. Gunawardana and W. Byrne. Discriminative speaker adaptation with conditional maximum likelihood linear regression. In *Proc. EUROSPEECH*, pages 1203–1206, Aalborg, Denmark, 2001.

[67] Dimitri Kanevsky. Extended Baum transformations for general functions. In *Proc. IEEE ICASSP*, volume 5, Montreal, Canada 2004.

[68] Tony Jebara. *Discriminative, Generative, and Imitative Learning*. PhD thesis, Massachusetts Institute of Technology, 2002.

[69] Mohamed Afify. Extended Baum-Welch reestimation of Gaussian mixture models based on reverse Jensen inequality. In *Proc. INTERSPEECH*, pages 1113–1116, Lisbon, Portugal, 2005.

[70] K. Na, B. Jeon, D. Chang, S. Chae, and S. Ann. Discriminative training of hidden Markov models using overall risk criterion and reduced gradient method. In *Proc. EUROSPEECH*, pages 97–100, Madrid, Spain, September 1995.

[71] Janez Kaiser, Bogomir Horvat, and Zdravko Ka. Overall risk criterion estimation of hidden Markov model parameters. *Speech Communication*, 38(3-4):383–398, 2002.

[72] D. Povey and P.C. Woodland. Minimum phone error and I-smoothing for improved discriminative training. In *Proc. IEEE ICASSP*, volume 1, pages 105–108, Orlando, FL, May 2002.

[73] D. Povey. *Discriminative Training for Large Vocabulary Speech Recognition*. PhD thesis, University of Cambridge, 2004.

[74] B.H. Juang and S. Katagiri. Discriminative learning for minimum error classification. *IEEE Transactions on Signal Processing*, 40(12):3043– 3054, December 1992.

[75] W. Chou, B. H. Juang, and C. H. Lee. Segmental GPD training of HMM based speech recognizer. In *Proc. IEEE ICASSP*, volume 1, pages 473–476, San Francisco, CA, USA, 1992.

[76] W. Chou, C. H. Lee, and B. Juang. Minimum error rate training based on N-best string models. In *Proc. IEEE ICASSP*, volume 2, pages 652–655, Minneapolis, MN, USA, April 1993.

[77] W. Reichl and G. Ruske. Discriminative training for continuous speech recognition. In *Proc. EUROSPEECH*, volume 1, pages 537–540, Madrid, Spain, 1995.

[78] S. Katagiri, Biing-Hwang Juang, and Chin-Hui Lee. Pattern recognition using a family of design algorithms based upon the generalized probabilistic descent method. *Proc. IEEE*, 11:2345–2373, Nov 1998.

[79] Wolfgang Macherey, Lars Haferkamp, Ralf Schlöuter, and Hermann Ney. Investigations on error minimizing training criteria for discriminative training in automatic speech recognition. In *Proc. INTERSPEECH*, pages 2133–2136, Lisbon, Portugal, 2005.

[80] Ralf Schlüter, Wolfgang Macherey, Boris Müller, and Hermann Ney. Comparison of discriminative training criteria and optimization methods for speech recognition. *Speech Communication*, 34(3):287–310, 2001.

[81] L. Mangu, E. Brill, and A. Stolcke. Finding consensus among words: Lattice-based word error minimization. In *Proc. EUROSPEECH*, pages 495–498, Budapest, Hungary, 1999.

[82] L. Mangu, E. Brill, and A. Stolcke. Finding consensus in speech recognition: Word error minimization and other applications of confusion networks. *Computer Speech and Language*, 14(4):373–400, 2000.

[83] K.-F. Lee. *Large Vocabulary Speaker-independent Continuous Speech Recognition: The SPHINX System*. PhD thesis, Carnegie Mellon University, 1988.

[84] M. Hwang and X. Huang. Acoustic classification of phonetic hidden Markov models. In *Proc. EUROSPEECH*, Genova, Italy, 1991.

[85] L. Bahl, P. deSouza, P. Gopalakrishnan, D. Nahamoo, and M. Picheny. Decision trees for phonological rules in continuous speech. In *Proc. IEEE ICASSP*, volume 1, pages 185– 188, Toronto, Canada, 1991.

[86] S. Young and P. Woodland. State clustering in HMM-based continuous speech recognition. *Computer Speech and Language*, 8(4):369–384, 1994.

[87] N. Morgan and H. Bourlard. Continuous speech recognition: An introduction to the hybrid HMM/connectionist approach. *IEEE Signal Processing Magazine*, 12(3):25–42, May 1995.

[88] E. Trentin and M. Gori. A survey of hybrid ANN/HMM models for automatic speech recognition. *Neurocomputing*, 37(1-4):91–126, April 2001.

[89] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[90] Chris M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, 1995.

[91] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[92] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[93] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019.