Multiclass Classification Networks

Softmax Regression, Cross-Entropy, and Linear Decision Boundaries

Yasser Hifny

Faculty of Computers and Artificial Intelligence
Capital University

Roadmap

1. Multiclass softmax model
2. One-hot target representation
3. Categorical cross-entropy learning objective
4. Softmax Jacobian and gradient derivation
5. Gradient-descent parameter updates
6. Classification decision and linear boundaries
7. Three-class numerical/Python example
8. Assignment: handwritten-digit recognition

Multiclass Classification

- Multiclass classification assigns an input sample to one of $K \geq 3$ classes.
- We introduces the probabilistic **softmax** (maximum-entropy) classifier.
- Inputs may be float, binary, or integer features.
- Outputs are categorical class labels represented using one-hot vectors.

Goal

Learn parameters **W** and **b** so that the predicted class probabilities match the training targets.

One-Hot Representation

For $K = 3$ classes, a target vector contains exactly one 1:

$$A = [1, 0, 0], \qquad B = [0, 1, 0], \qquad C = [0, 0, 1].$$

- Each vector identifies the target class.
- The components sum to one.
- This representation is directly compatible with the softmax probability vector.

The Multiclass Network

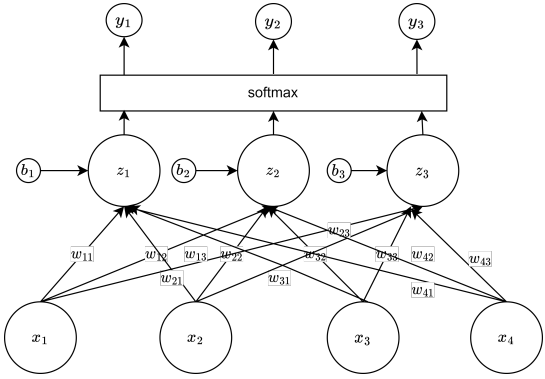


Figure: Four input variables and three output classes.

Linear Multiclass Model

For four inputs and three classes:

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

Compactly,

$$\boxed{\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}}$$

where $\mathbf{W} \in \mathbb{R}^{K \times d}$ and $\mathbf{b} \in \mathbb{R}^K$.

Softmax Output

The logits are transformed into normalized class probabilities:

$$\boxed{y_i = \frac{e^{z_i}}{\sum_{c=1}^{K} e^{z_c}}, \qquad i = 1, \ldots, K}$$

- $0 < y_i < 1$.
- $\sum_i y_i = 1$.
- $\mathbf{y} = \text{softmax}(\mathbf{z})$ is therefore a probability distribution over classes.
- The parameters $\mathbf{W}, \mathbf{b}$ are learned from the training data.

Numerically Stable Softmax

A useful implementation subtracts the largest logit before exponentiation:

$$\mathbf{z}' = \mathbf{z} - \max_i z_i.$$

Then

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i'}}{\sum_c e^{z_c'}}.$$

Why?

Subtracting the same constant from all logits does not change the softmax probabilities, but reduces the risk of numerical overflow in e^{z_i}.

Learning Problem

Given training samples

$$\{(\mathbf{x}_n, \mathbf{t}_n)\}_{n=1}^{N},$$

where $\mathbf{t}_n \in \{0,1\}^K$ is one-hot, minimize the average categorical cross-entropy:

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

with

$$\boxed{E^n = -\sum_{k=1}^{K} t_k^n \log y_k^n.}$$

Why Cross-Entropy?

- The model outputs probabilities, while the targets are one-hot categorical distributions.
- Cross-entropy measures the discrepancy between the target distribution and predicted distribution.
- For a one-hot target, only the probability assigned to the true class contributes directly to the loss.

$$E^n = -\log y^n_{\text{true}}.$$

Learning objective

Find **W** and **b** that minimize the average training loss.

Softmax Jacobian Matrix

The softmax function maps the input vector $\mathbf{z} \in \mathbb{R}^K$ to the probability vector $\mathbf{y} \in \mathbb{R}^K$. Its derivative with respect to the inputs is a $K \times K$ matrix called the **Jacobian matrix**.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_K} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial z_1} & \frac{\partial y_K}{\partial z_2} & \cdots & \frac{\partial y_K}{\partial z_K} \end{pmatrix}.$$

For the softmax function,

$$y_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}},$$

the diagonal elements ($i = k$) are

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

For the off-diagonal elements ($i \neq k$),

$$\frac{\partial y_k}{\partial z_i} = -y_i y_k.$$

Softmax Derivative: Compact Form

Combining the diagonal and off-diagonal cases gives the general softmax derivative:

$$\boxed{\frac{\partial y_k}{\partial z_i} = y_k(\delta_{ik} - y_i)} \tag{1}$$

Here, δ_{ik} is the Kronecker delta:

$$\delta_{ik} = \begin{cases} 1, & i = k, \\ 0, & i \neq k. \end{cases}$$

The Jacobian can also be expressed as the difference between a diagonal matrix and an outer product:

$$\boxed{\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \operatorname{diag}(\mathbf{y}) - \mathbf{y}\mathbf{y}^{\mathsf{T}}}$$

$$= \begin{pmatrix} y_1 & 0 & \cdots & 0 \\ 0 & y_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_K \end{pmatrix} - \begin{pmatrix} y_1 y_1 & y_1 y_2 & \cdots & y_1 y_K \\ y_2 y_1 & y_2 y_2 & \cdots & y_2 y_K \\ \vdots & \vdots & \ddots & \vdots \\ y_K y_1 & y_K y_2 & \cdots & y_K y_K \end{pmatrix}.$$

Cross-Entropy Gradient with Respect to Weights

For training example n, the cross-entropy loss is

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

Using the chain rule and the softmax derivative,

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

Here, $z_i^n = \sum_j w_{ij} x_j^n + b_i$ and $\partial z_i^n / \partial w_{ij} = x_j^n$.

Simplifying the Cross-Entropy Gradients

Since the target vector is one-hot (or a probability distribution), its components satisfy

$$\sum_{k=1}^{K} t_k^n = 1.$$

Therefore, the weight gradient simplifies to

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n \sum_{k=1}^{K} t_k^n - \sum_{k=1}^{K} t_k^n \delta_{ik}\right) x_j^n \\ &= \frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)x_j^n.\end{aligned}$$

Similarly, the gradient with respect to the bias is

$$\boxed{\frac{\partial E}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)}$$

Thus, both gradients depend on the prediction error $(y_i^n - t_i^n)$.

Gradient Descent Updates

Using gradient descent, the weights and biases are updated as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{1}{N} \sum_{n=1}^{N} (y_i^n - t_i^n) x_j^n,$$
$$b_i^{(t+1)} = b_i^{(t)} - \eta \frac{1}{N} \sum_{n=1}^{N} (y_i^n - t_i^n). \tag{2}$$

- η is the learning rate.
- $(y_i^n - t_i^n)$ is the prediction error for class i.
- The weight gradient scales this error by input feature x_j^n.
- Mini-batch stochastic gradient descent estimates the gradients from small batches, enabling efficient training on large datasets.

The resulting optimization procedure follows the same general principle as gradient-based linear regression.

Vectorized Gradient Form

For a batch represented by matrix X:

$$Z = XW + b, \qquad Y = \text{softmax}(Z).$$

The gradients can be written compactly as

$$\boxed{\nabla_W E = \frac{1}{m} X^T (Y - T)}$$

$$\boxed{\nabla_b E = \frac{1}{m} \sum_{n=1}^{m} (y^n - t^n)}.$$

This is the form used naturally in NumPy/PyTorch implementations.

Classification Decision

After learning **W** and **b**, prediction is made by selecting the class with the largest probability:

$$\boxed{\hat{y} = \arg \max_{1 \le k \le K} y_k}$$

Because softmax is monotonic with respect to each individual logit,

$$\boxed{\hat{y} = \arg \max_{1 \le k \le K} z_k}.$$

Thus prediction can be made directly from the logits.

Linear Decision Boundaries

For three classes and two input variables:

$$z_i = w_{i1}x_1 + w_{i2}x_2 + b_i.$$

The boundary between classes 1 and 2 occurs where their logits are equal:

$$z_1 = z_2.$$

Therefore,

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

This is a straight line in the two-dimensional feature space.

Three-Class Decision Boundaries

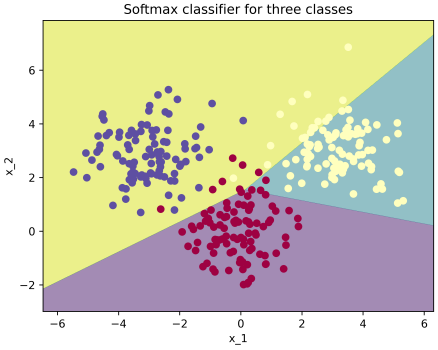


Figure: Linear decision boundaries between three classes.

Three-Class Example: Data Generation

The example generates three Gaussian clusters:

- Class 0: mean $[0, 0]$
- Class 1: mean $[3, 3]$
- Class 2: mean $[-3, 3]$
- 100 samples per class
- One-hot targets for the three classes

$$X = \text{concatenate}(X_0, X_1, X_2), \qquad T = \text{concatenate}(T_0, T_1, T_2).$$

Python: Softmax and Cross-Entropy

```
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    exp_x = np.exp(x)
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def cross_entropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-12, 1 - 1e-12)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

Python: Gradient Descent

```
def gradient_descent(X, y_true, y_pred, w, b, learning_rate):
    m = X.shape[0]
    dw = (1/m) * np.dot(X.T, (y_pred - y_true))
    db = (1/m) * np.sum(y_pred - y_true, axis=0)
    w = w - learning_rate * dw
    b = b - learning_rate * db
    return w, b
```

Core computation

The implementation is exactly the vectorized form of the derived gradients.

Python: Training Loop

```
w = np.random.randn(2, 3)
b = np.random.randn(3)
epochs = 100
learning_rate = 0.01

for epoch in range(epochs):
    y_pred = softmax(np.dot(X, w) + b)
    loss = cross_entropy(y_true, y_pred)
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
    w, b = gradient_descent(
        X, y_true, y_pred, w, b, learning_rate)
```

Python: Decision Regions

```
xx, yy = np.meshgrid(
    np.arange(x_min, x_max, .01),
    np.arange(y_min, y_max, .01))

Z = np.argmax(
    softmax(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))
plt.show()
```

Example: What the Training Learns

- The model starts with randomly initialized weights and biases.
- Each iteration computes class probabilities with softmax.
- Cross-entropy measures the prediction error.
- The gradients adjust W and b toward lower loss.
- The resulting model creates linear regions separated by linear decision boundaries.

Key point

For this single-layer softmax model, the learned decision boundaries are linear in the input feature space.

Summary

1. Softmax regression extends linear classification to K classes.
2. Class labels are represented as one-hot vectors.
3. Softmax converts logits into a probability distribution.
4. Categorical cross-entropy provides the learning objective.
5. The softmax Jacobian leads to a simple $Y - T$ gradient.
6. Gradient descent learns W and b.
7. Prediction selects the largest logit/probability.
8. The single-layer classifier produces linear decision boundaries.

Assignment

Handwritten-Digit Recognition

Implement optical recognition of handwritten digits as a multiclass classification task using Python from scratch, using the Keras deep-learning library, and Google Colab.

- Load the Optical Recognition of Handwritten Digits dataset.
- Encode the digit labels for multiclass classification.
- Train a suitable classifier.
- Evaluate the classification performance.
- Visualize or analyze the predictions.

Dataset source:
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

Thank You

Questions?