Übungsblatt: Backpropagation

Rashad Gasimov

09/01/2026

NN.Backprop.01 – Gewichtsupdates für versteckte Schichten

Vorwärtsrechnung

Die Eingabe sei $A^{[0]} = X$. Für jede Schicht l gilt:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}, \quad A^{[l]} = \sigma(Z^{[l]})$$

Die Netz-Ausgabe ist $A^{[3]} = \hat{Y}$.

Sigmoid-Funktion

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$$

Fehlerfunktion

$$L = -\Big(Y \log \hat{Y} + (1-Y)\log(1-\hat{Y})\Big)$$

Für die Ausgabeschicht ergibt sich:

$$\frac{\partial L}{\partial Z^{[3]}} = \hat{Y} - Y$$

Backpropagation

$$\delta^{[3]} = \hat{Y} - Y$$

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \odot \sigma'(Z^{[2]})$$

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \odot \sigma'(Z^{[1]})$$

Gradienten

$$\frac{\partial L}{\partial W^{[1]}} = \delta^{[1]} X^T, \quad \frac{\partial L}{\partial b^{[1]}} = \delta^{[1]}$$

Gewichtsupdates

Mit Lernrate η:

$$W^{[1]} \leftarrow W^{[1]} - \eta \frac{\partial L}{\partial W^{[1]}}, \quad b^{[1]} \leftarrow b^{[1]} - \eta \frac{\partial L}{\partial b^{[1]}}$$

NN.Backprop.02: Forward- und Backpropagation

Forward Pass

$$x = 0, \quad y_T = 0.5$$

Versteckte Zelle

$$z_h = -1 \cdot 0 + 1 = 1, \quad h = \sigma(1) \approx 0.7311$$

Ausgabezelle

$$z_y = 1 \cdot 0.7311 + 2 \cdot 0 - 2 = -1.2689, \quad y = \sigma(-1.2689) \approx 0.2195$$

Fehler

$$E = \frac{1}{2}(y - y_T)^2 \approx 0.0393$$

Backpropagation

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

Fehlerterme

$$\delta_y = (y - y_T)\sigma'(z_y) \approx -0.0481$$
$$\delta_h = \delta_y \cdot 1 \cdot \sigma'(z_h) \approx -0.00945$$

Gradienten

$$\frac{\partial E}{\partial w_2} = \delta_y \cdot h \approx -0.0352$$
$$\frac{\partial E}{\partial b_2} = \delta_y \approx -0.0481$$
$$\frac{\partial E}{\partial b_1} = \delta_h \approx -0.00945$$

Updates

Lernrate $\alpha = 0.01$:

$$w_2 \leftarrow 1 + 0.000352, \quad b_2 \leftarrow -2 + 0.000481, \quad b_1 \leftarrow 1 + 0.0000945$$

NN.Backprop.03: MLP und Backpropagation

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# --- Aktivierungsfunktionen ---
def relu(z): return np.maximum(0, z)
def relu_deriv(z): return (z > 0).astype(float)
def sigmoid(z): return 1 / (1 + np.exp(-np.clip(z, -50, 50)))

# --- Loss: Cross-Entropy ---
def cross_entropy(y_true, y_pred):
    eps = 1e-9
    y_pred = np.clip(y_pred, eps, 1-eps)
    return -np.mean(np.sum(y_true*np.log(y_pred), axis=1))

# --- MLP mit 2 Hidden Layers ---
class MLP2Hidden:
    def __init__(self, input_dim, hidden1, hidden2, output_dim, lr=0.01, seed=0):
        rng = np.random.default_rng(seed)
        # He Init f r ReLU
        self.W1 = rng.normal(0, np.sqrt(2/input_dim), (input_dim, hidden1))
        self.b1 = np.zeros((1, hidden1))
        self.W2 = rng.normal(0, np.sqrt(2/hidden1), (hidden1, hidden2))
        self.b2 = np.zeros((1, hidden2))
        self.W3 = rng.normal(0, np.sqrt(1/hidden2), (hidden2, output_dim))
        self.b3 = np.zeros((1, output_dim))
        self.lr = lr

    def forward(self, X):
        self.X = X
        self.Z1 = X @ self.W1 + self.b1
        self.A1 = relu(self.Z1)
        self.Z2 = self.A1 @ self.W2 + self.b2
        self.A2 = relu(self.Z2)
        self.Z3 = self.A2 @ self.W3 + self.b3
        self.A3 = sigmoid(self.Z3)
        return self.A3

    def backward(self, Y):
        delta3 = self.A3 - Y
        dW3 = self.A2.T @ delta3
        db3 = delta3
        delta2 = (delta3 @ self.W3.T) * relu_deriv(self.Z2)
        dW2 = self.A1.T @ delta2
        db2 = delta2
        delta1 = (delta2 @ self.W2.T) * relu_deriv(self.Z1)
        dW1 = self.X.T @ delta1
        db1 = delta1
        # Update
        self.W3 -= self.lr * dW3
        self.b3 -= self.lr * db3
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1

    def train(self, X, Y, epochs=2000):
        losses = []
        n = X.shape[0]
        for ep in range(epochs):
            idx = np.random.permutation(n)
            Xs, Ys = X[idx], Y[idx]
            epoch_loss = 0
            for xi, yi in zip(Xs, Ys):
                xi = xi.reshape(1, -1)
                yi = yi.reshape(1, -1)
                self.forward(xi)
                epoch_loss += cross_entropy(yi, self.A3)
                self.backward(yi)
            epoch_loss /= n
            losses.append(epoch_loss)
        return losses

    def predict(self, X):
        probs = self.forward(X)
        return np.argmax(probs, axis=1), probs

# --- Daten laden und vorbereiten ---
df = pd.read_csv("iris.csv")
X = df.iloc[:, :-1].values.astype(float)
y_raw = df.iloc[:, -1].values

# Standardisierung
X = (X - X.mean(axis=0)) / (X.std(axis=0) + 1e-9)

# One-Hot Encoding
classes = np.unique(y_raw)
Y = np.zeros((len(y_raw), len(classes)))
for i, c in enumerate(classes):
    Y[y_raw == c, i] = 1

# --- Netzwerk initialisieren und trainieren ---
mlp = MLP2Hidden(input_dim=4, hidden1=16, hidden2=8, output_dim=3, lr=0.01)
losses = mlp.train(X, Y, epochs=2000)

# --- Evaluation ---
y_pred, _ = mlp.predict(X)
y_true = np.argmax(Y, axis=1)
acc = (y_pred == y_true).mean()
print(f"Final Loss: {losses[-1]:.6f}, Accuracy: {acc*100:.2f}%")

# --- Trainingsfehler plotten ---
plt.plot(losses)
plt.xlabel("Epoche")
plt.ylabel("Cross-Entropy Loss")
plt.title("Trainingsfehler  ber  Epochen (2 Hidden Layers)")
plt.grid(True)
plt.show()
```

Mit 2 Hidden Layers (z.B. 16 + 8 Neuronen) und ausreichender Lernrate sinkt der Trainingsfehler nach etwa 1000–2000 Epochen fast auf Null. Bei einem kleineren Netzwerk (z.B. nur 1 Hidden Layer) kann es deutlich länger dauern oder der Fehler erreicht nicht exakt Null.