In [ ]:
```
import numpy as np
from tensorflow import keras

# Load MNIST data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Flatten the 28×28 images into 784-length vectors
x_train = x_train.reshape(-1, 28 * 28).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28 * 28).astype('float32') / 255.0

# One-hot encode the labels
C = 10
def one_hot(y, C):
    Y = np.zeros((y.shape[0], C), dtype=np.float32)
    Y[np.arange(y.shape[0]), y] = 1.0
    return Y

Y_train = one_hot(y_train, C)
Y_test = one_hot(y_test, C)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

First step is essentially reshaping data from 28 x 28 to a single vector of 784.

In [ ]:
```
# Initialize weights and biases randomly (small values)
rng = np.random.default_rng(42)
W = rng.normal(0, 0.01, size=(28 * 28, C)).astype(np.float32)
b = np.zeros((C,), dtype=np.float32)
```

W = weight matrix that transforms 784 input features into 10 output classes (guesses).

b = bias vector added to each output.

These values start random so the model can learn from scratch.

In [ ]:
```
# Softmax function: converts raw scores into probabilities
def softmax(z):
    z = z - z.max(axis=1, keepdims=True)  # numerical stability
    expz = np.exp(z)
    return expz / expz.sum(axis=1, keepdims=True)

# Forward pass
def forward(X):
    return softmax(X @ W + b)

# Cross-entropy loss
def cross_entropy(Y, P):
    eps = 1e-12
    return -np.mean(np.sum(Y * np.log(P + eps), axis=1))

# Accuracy
def accuracy(Y, P):
    return np.mean(np.argmax(Y, axis=1) == np.argmax(P, axis=1))
```

Softmax: Converts raw scores into probabilities, where the statistics come into data interpretation

Forward Pass: Prediction = Input (X) * Weights (W) + bias (b)

Cross Entropy Loss: Measures how far off predictions are off from true value: what we are trying to minimize

In [ ]:
```
# Hyperparameters
lr = 0.1
batch = 256
epochs = 8

# Training loop
for epoch in range(epochs):
    idx = rng.permutation(x_train.shape[0])  # shuffle data
    for i in range(0, len(idx), batch):
        j = idx[i:i+batch]
        Xb, Yb = x_train[j], Y_train[j]
        Pb = forward(Xb)

        # Gradient computation
        grad_logits = (Pb - Yb) / Xb.shape[0]
        gW = Xb.T @ grad_logits
        gb = grad_logits.sum(axis=0)

        # Weight update
```

```
        W -= lr * gW
        b -= lr * gb

    # Monitor training progress
    P = forward(x_train[:10000])
    L = cross_entropy(Y_train[:10000], P)
    A = accuracy(Y_train[:10000], P)
    print(f"Epoch {epoch+1:02d} — Loss: {L:.4f}, Accuracy: {A:.4f}")
```

```
Epoch 01 — Loss: 0.4578, Accuracy: 0.8858
Epoch 02 — Loss: 0.3865, Accuracy: 0.8969
Epoch 03 — Loss: 0.3568, Accuracy: 0.9029
Epoch 04 — Loss: 0.3398, Accuracy: 0.9070
Epoch 05 — Loss: 0.3275, Accuracy: 0.9101
Epoch 06 — Loss: 0.3183, Accuracy: 0.9123
Epoch 07 — Loss: 0.3132, Accuracy: 0.9131
Epoch 08 — Loss: 0.3065, Accuracy: 0.9167
```

In [ ]:
```
# Final test accuracy
P_test = forward(x_test)
test_acc = accuracy(Y_test, P_test)
print(f"Test Accuracy: {test_acc * 100:.2f}%")
```

```
Test Accuracy: 91.58%
```

What's New?: Every step, from forward pass to weight update, is done manually. Purely linear algebra and calculus.

Much slower, but transparent and surprisingly accurate.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
!jupyter nbconvert --to html '/content/drive/MyDrive/HNR499/HNR499_Model004'
```

```
Mounted at /content/drive
```