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

# Load and flatten MNIST data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
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 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 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

In [ ]:
```python
# Hyperparameters
H = 128  # hidden layer size
rng = np.random.default_rng(0)

# Xavier initialization
W1 = rng.normal(0, 0.02, size=(28 * 28, H)).astype(np.float32)
b1 = np.zeros((H,), dtype=np.float32)
W2 = rng.normal(0, 0.02, size=(H, C)).astype(np.float32)
b2 = np.zeros((C,), dtype=np.float32)
```

Introduces "Hidden Layer" (H) used to learn more complex features. Now a two-layer neural network:

Layer 1 learns features, Layer 2 uses those features to classify digits

In [ ]:
```python
# Activation and softmax
def relu(u):
    return np.maximum(u, 0.0)

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):
    u1 = X @ W1 + b1
    h1 = relu(u1)
    u2 = h1 @ W2 + b2
    yhat = softmax(u2)
    return yhat, (X, u1, h1, u2)

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

relu() introduces nonlinearity, allowing more complex decision surfaces.

softmax() converts final scores to probabilities

Forward pass:

1. Linear transformation from input to hidden
2. ReLU activation
3. Linear transformation from hidden to output

Intermediate results are used again during backpropogation

In [ ]:
```python
def train_epoch(X, Y, lr=0.1, batch=128):
    global W1, b1, W2, b2
    idx = rng.permutation(X.shape[0])

    for i in range(0, len(idx), batch):
        j = idx[i:i+batch]
        Xb, Yb = X[j], Y[j]

        # Forward pass
        yhat, (Xc, u1, h1, u2) = forward(Xb)

        # Compute gradients step by step
```

```
        du2 = (yhat - Yb) / Xb.shape[0]         # dL/du2 derivative of the loss w.r.t. output logits.
        gW2 = h1.T @ du2                        # dL/dW2 gradient for output weights.
        gb2 = du2.sum(axis=0)                   # dL/db2

        dh1 = du2 @ W2.T                        # dL/dh1 gradient flowing backward into the hidden layer.
        du1 = dh1 * (u1 > 0)                    # dL/du1 (ReLU derivative) (1 if >0, else 0).
        gW1 = Xb.T @ du1                        # dL/dW1 gradient for first layer weights.
        gb1 = du1.sum(axis=0)                   # dL/db1

        # Gradient descent update
        W2 -= lr * gW2
        b2 -= lr * gb2
        W1 -= lr * gW1
        b1 -= lr * gb1

# Train for several epochs
epochs = 8
for ep in range(epochs):
    train_epoch(x_train, Y_train, lr=0.1, batch=128)
    P = forward(x_train[:10000])[0]
    acc = accuracy(Y_train[:10000], P)
    print(f"Epoch {ep+1:02d} — Training accuracy: {acc:.4f}")
```

```
Epoch 01 — Training accuracy: 0.9092
Epoch 02 — Training accuracy: 0.9270
Epoch 03 — Training accuracy: 0.9388
Epoch 04 — Training accuracy: 0.9466
Epoch 05 — Training accuracy: 0.9544
Epoch 06 — Training accuracy: 0.9592
Epoch 07 — Training accuracy: 0.9628
Epoch 08 — Training accuracy: 0.9674
```

In [ ]:

```
# Evaluate on the test set
P_test = forward(x_test)[0]
test_acc = accuracy(Y_test, P_test)
print(f"Test Accuracy: {test_acc * 100:.2f}%")
```

```
Test Accuracy: 96.33%
```

Notes: Backpropogation is implemented entirely by hand, every derivative step is shown.

No tensorFlow, no autograd, just matrix operations

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
!jupyter nbconvert --to html '/content/drive/MyDrive/HNR499/HNR499_Model5'
```