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

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

# Normalize and reshape
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# One-hot encode labels
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

In [ ]:
```python
# Glorot/Xavier initializer
def glorot(shape):
    fan_in = np.prod(shape[:-1])
    fan_out = shape[-1] * np.prod(shape[:-2])
    limit = np.sqrt(6.0 / (fan_in + fan_out))
    return tf.Variable(tf.random.uniform(shape, -limit, limit), trainable=True)

# Convolution and dense layer weights
W1 = glorot((3, 3, 1, 32))   # conv1: 1 input channel -> 32 filters
b1 = tf.Variable(tf.zeros([32]), trainable=True)
W2 = glorot((3, 3, 32, 64))  # conv2: 32 -> 64 filters
b2 = tf.Variable(tf.zeros([64]), trainable=True)


# Fully connected layers
W3 = glorot((1600, 64))
b3 = tf.Variable(tf.zeros([64]), trainable=True)
W4 = glorot((64, num_classes))
b4 = tf.Variable(tf.zeros([num_classes]), trainable=True)
```

Instead of keras creating weights for us, we manually define them as trainable variables.

Weight tensors initialized with Glorot, a technique to keep values in range that trains efficiently.

W1 and W2: Convolutions kernels (filters)

W3 and W4: Dense weight matracies

b1, b2, b3, b4: Biases added after each layer

Ultimately shows that neaural networks are big matricies that have operations performed on them (multiplication, addition)

In [ ]:
```python
def relu(x): # Non linearity, allowing network to learn complex patterns
    return tf.nn.relu(x)

def maxpool2x2(x): # reduces dimensions of data, keeps strongest variation captured
    return tf.nn.max_pool2d(x, ksize=2, strides=2, padding="VALID")

def flatten(x): # reshapes matracies into vectors
    return tf.reshape(x, [tf.shape(x)[0], -1])

# Forward pass
def forward(x, training=True):
    z1 = tf.nn.conv2d(x, W1, strides=1, padding='VALID') + b1
    a1 = relu(z1)
    p1 = maxpool2x2(a1)

    z2 = tf.nn.conv2d(p1, W2, strides=1, padding='VALID') + b2
    a2 = relu(z2)
    p2 = maxpool2x2(a2)

    f = flatten(p2)
    h = relu(tf.matmul(f, W3) + b3)
    logits = tf.matmul(h, W4) + b4
    return tf.nn.softmax(logits), logits

    # shows each step manually, from detecting edges, to matrix multiplication to
    # proaability conversion for classification
```

In [ ]:
```python
# Prepare optimizer and loss
optimizer = keras.optimizers.Adam()
```

```
loss_fn = keras.losses.CategoricalCrossentropy()

# Dataset batches
batch_size = 64
train_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(1024).batch(batch_size)
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size)

# Training loop
epochs = 2
for epoch in range(epochs):
    acc_metric = keras.metrics.CategoricalAccuracy()
    for xb, yb in train_ds:
        with tf.GradientTape() as tape:
            y_pred, _ = forward(xb, training=True)
            loss = loss_fn(yb, y_pred)
        grads = tape.gradient(loss, [W1, b1, W2, b2, W3, b3, W4, b4])
        optimizer.apply_gradients(zip(grads, [W1, b1, W2, b2, W3, b3, W4, b4]))
        acc_metric.update_state(yb, y_pred)

    print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {acc_metric.result().numpy():.4f}")
```

```
Epoch 1/2 - Accuracy: 0.9471
Epoch 2/2 - Accuracy: 0.9836
```

Controlling the process:

Computing precitions with our forward pass, calculating loss, using it to calculate gradients, applying them to update the weight tensors.

In [ ]:
```
# Evaluate on test data
test_acc_metric = keras.metrics.CategoricalAccuracy()
for xb, yb in test_ds:
    y_pred, _ = forward(xb, training=False)
    test_acc_metric.update_state(yb, y_pred)

print("Test accuracy:", test_acc_metric.result().numpy())
```

```
Test accuracy: 0.9878
```

What's new?: Manually defined every weight tensor

Convolution, pooling, ReLU, flattening, and dense layers are all coded with tf.nnops

Nearly everything is visible math, openly followable

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
!jupyter nbconvert --to html '/content/drive/MyDrive/HNR499/HNR499_Model3'
```