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

# Load and preprocess MNIST
(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)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

In [ ]:
```
# Define a CNN by subclassing keras.Model for more control
class CNN(keras.Model):
    def __init__(self, num_classes=10):
        super().__init__()
        self.conv1 = layers.Conv2D(32, 3, activation='relu') # Detects edges and shapes
        self.pool1 = layers.MaxPooling2D()  # reduces spatial size, keeping important features
        self.conv2 = layers.Conv2D(64, 3, activation='relu') # Detects edges and shapes
        self.pool2 = layers.MaxPooling2D() # same here
        self.flatten = layers.Flatten() #reshapes matrix into a vector
        self.fc1 = layers.Dense(64, activation='relu')
        self.out = layers.Dense(num_classes, activation='softmax') # turns outputs into probabilities

    def call(self, x, training=False):  #defines how data flows through the network
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = self.fc1(x)
        return self.out(x)

# Create model instance
model = CNN(num_classes=num_classes)
```

Essentially, we are creating our own model class by subclassing keras.Model.

In [ ]:
```
# Loss and optimizer

# Measures how far off predictions are from true labels, thing network is trying to
# minimize
loss_fn = keras.losses.CategoricalCrossentropy()
optimizer = keras.optimizers.Adam()

# Create TensorFlow Datasets
batch_size = 64

# shown below, batches and shuffles data, making training faster/more stable
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)
```

Adam: adaptive gradient-based optimizer that automatically adjusts the learning rate for each parameter.

In [ ]:
```
# Manual training loop with gradient tape
epochs = 3
for epoch in range(epochs):
    print(f"\nEpoch {epoch + 1}/{epochs}")
    acc_metric = keras.metrics.CategoricalAccuracy()
    total_loss = 0.0
    steps = 0

    for step, (x_batch, y_batch) in enumerate(train_ds):
        with tf.GradientTape() as tape:
            predictions = model(x_batch, training=True)
            loss = loss_fn(y_batch, predictions)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        acc_metric.update_state(y_batch, predictions)
        total_loss += loss
```

```
        steps += 1

        if step % 100 == 0:
            print(f"Step {step}: loss = {loss:.4f}")

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

```
Epoch 1/3
Step 0: loss = 2.3556
Step 100: loss = 0.1351
Step 200: loss = 0.3009
Step 300: loss = 0.0883
Step 400: loss = 0.1522
Step 500: loss = 0.0713
Step 600: loss = 0.0815
Step 700: loss = 0.2047
Step 800: loss = 0.1178
Step 900: loss = 0.0979
Epoch 1 - Loss: 0.1783, Accuracy: 0.9474

Epoch 2/3
Step 0: loss = 0.0239
Step 100: loss = 0.0522
Step 200: loss = 0.1498
Step 300: loss = 0.0544
Step 400: loss = 0.0409
Step 500: loss = 0.0146
Step 600: loss = 0.0166
Step 700: loss = 0.0321
Step 800: loss = 0.0679
Step 900: loss = 0.0241
Epoch 2 - Loss: 0.0545, Accuracy: 0.9830

Epoch 3/3
Step 0: loss = 0.0250
Step 100: loss = 0.0131
Step 200: loss = 0.0301
Step 300: loss = 0.0353
Step 400: loss = 0.0751
Step 500: loss = 0.0596
Step 600: loss = 0.0177
Step 700: loss = 0.0330
Step 800: loss = 0.0571
Step 900: loss = 0.0185
Epoch 3 - Loss: 0.0382, Accuracy: 0.9882
```

Each batch does four things:

Forward pass: makes predictions from current weights

Loss calculation: measures how far off those precitions are

Backward pass (backpropogation):uses imported function to calculate gradients

Weight update: Use optimizer to adjust weights and reduce loss on the next iteration

In [ ]:
```
# Evaluate manually on the test set
test_acc_metric = keras.metrics.CategoricalAccuracy()
for x_batch, y_batch in test_ds:
    preds = model(x_batch, training=False)
    test_acc_metric.update_state(y_batch, preds)

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

```
Test accuracy: 0.9863
```

What's new: Explicitly compute gradients with tf.GradientTape().

Weight updates controlled with optimizer.apply_gradients()

Shows how learning happens

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
!jupyter nbconvert --to html '/content/drive/MyDrive/HNR499/HNR499_Model2'
```