Digit Recognition Project

Project Overview

- Train neural network with the existing images from the MNIST dataset to predict what digit is on new images.
- Use convolution-based classifier to figure out what digit is in a photo.
- Use probability distribution over 0-9 to predict what digit it is.

- Pipeline: Preprocessing -> Batching -> Training through gradient descent -> Evaluation

Prerequisites before we start

- Neural Networks
  - Tensors
  - Convolutional Neural Networks
  - Loss Functions
  - Optimizers
- PyTorch (Python Library)

- The content of this project is a little more complicating, but a lot of the details in the slide you just need a high-level understanding!

Neural Networks

- “A neural network is a machine learning model that stacks simple "neurons" in layers and learns pattern-recognizing weights and biases from data to map inputs to outputs.” [1]
- The network is made of the input layer, the ‘hidden’ layers and the output layer.
- The input layer just holds the initial values.
- The hidden layers transform the input by a factor of their biases and weights. This results in a linear transformation.
- Throughout the model, we perform a nonlinear activation, which makes the network not just a massive linear equation. The model would only learn linearly, only picking up straight line relationships, without this.

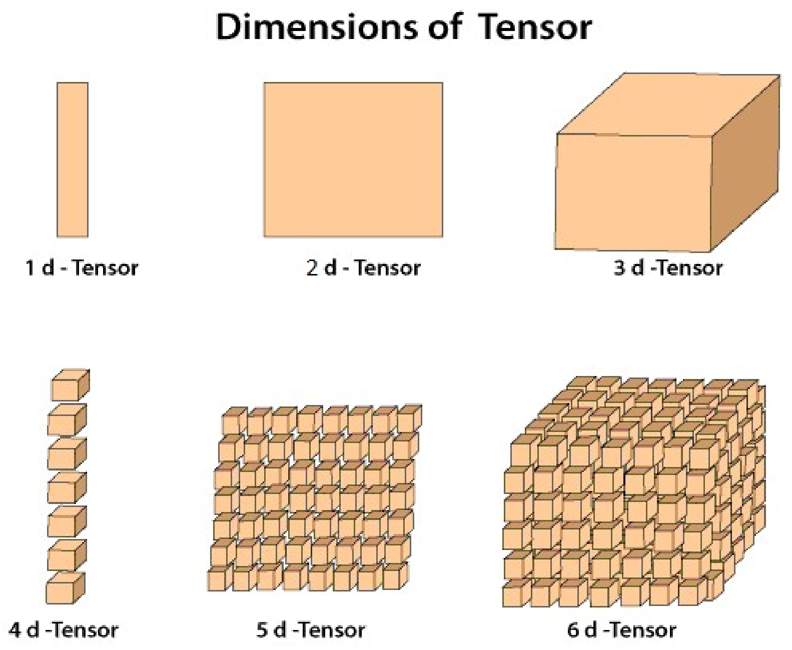
[1] IBM, “Neural Networks,” IBM Think.

PyTorch

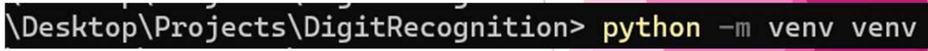
- PyTorch is a machine learning framework which helps with building neural networks. It is used in applications related to reinforcement learning, deep learning research, and computer vision.
- It has a very useful feature called **autograd,** which automatically calculates gradients when operations are performed on tensors. (more later)

Tensors

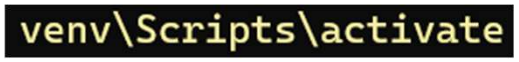
- A tensor is a multi-dimensional array that PyTorch uses as it’s main data structure.
- It is useful as it can do matrix math very quickly along with some other benefits.



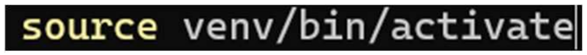
Setup Project!

- Make a folder (somewhere you'll remember) with the project name, like "DigitRecognition"
- Right-click the folder, and '**Open in terminal**'. → 

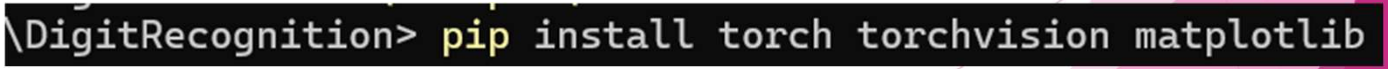
```
\Desktop\Projects\DigitRecognition> python -m venv venv
```

- Create a virtual environment
  - Windows → 

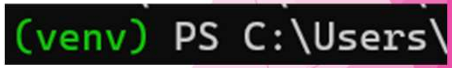
```
venv\Scripts\activate
```

  - Mac/Linux → 

```
source venv/bin/activate
```

- Install libraries → 

```
\DigitRecognition> pip install torch torchvision matplotlib
```



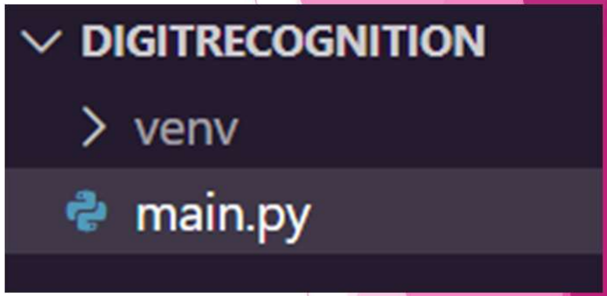
```
(venv) PS C:\Users\
```

Should see this after running this (Windows)

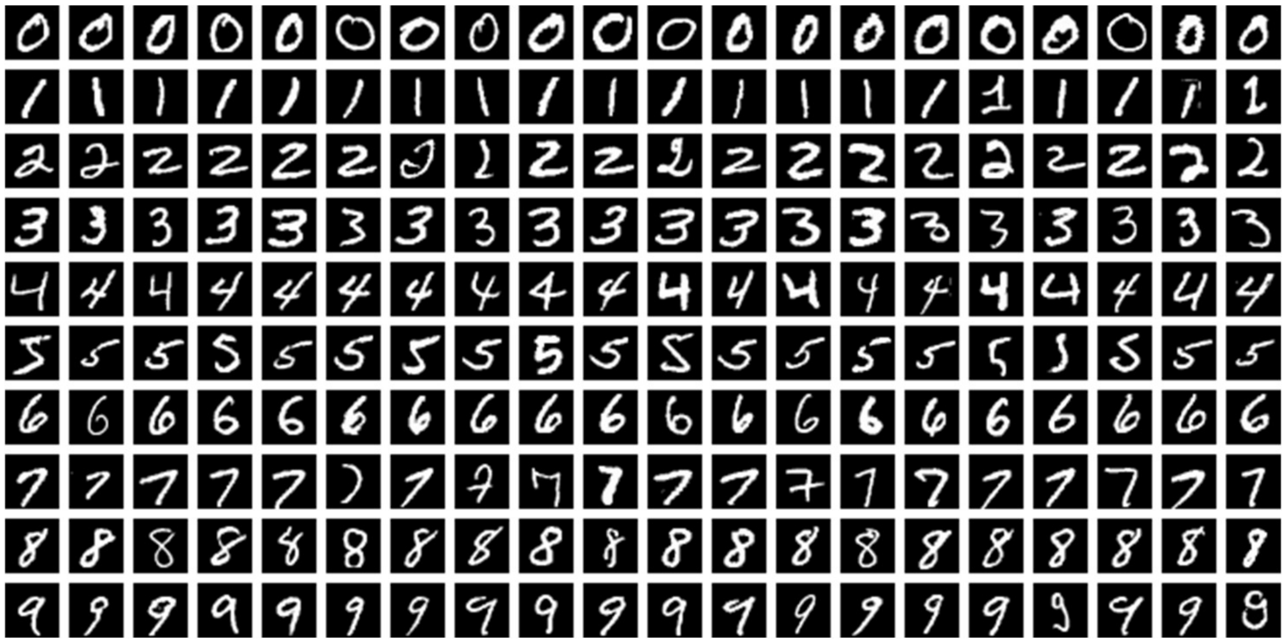
Setting up

- Open up your favorite IDE, I’ll be using VS Code.
- Open the project in that IDE and create a ‘main.py’ file.

Your directory should look like this



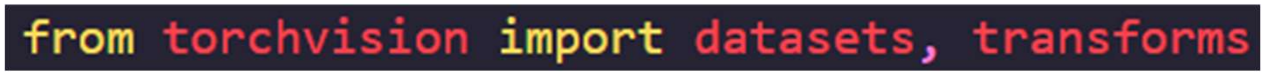
```
DIGITRECOGNITION
  > venv
  main.py
```



MNIST Dataset

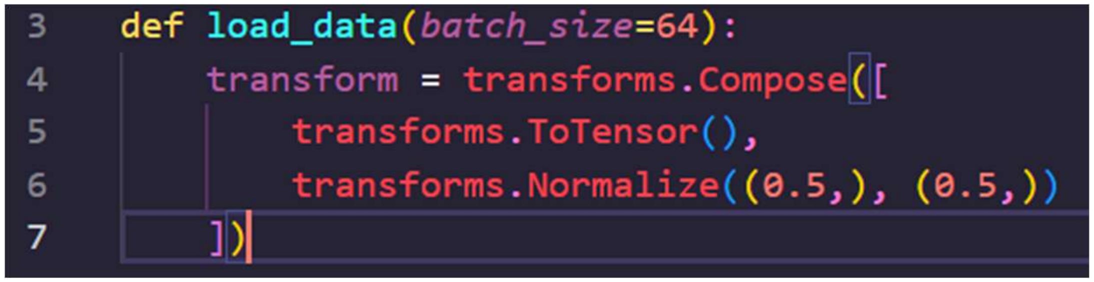
60,000 labeled grayscale images with 10,000 test images.

Let’s load this dataset!

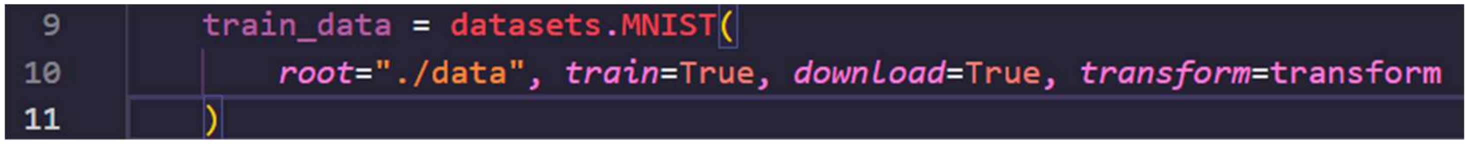


```python
from torchvision import datasets, transforms
```

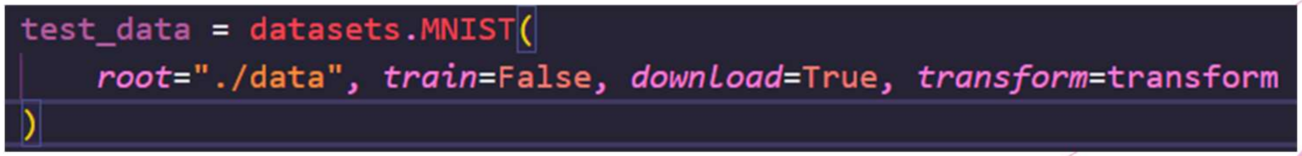
- We need to preprocess the data, converting the image to a numeric format that PyTorch can work with (tensor) and normalize the data a little to make it easier to work with.



```python
def load_data(batch_size=64):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
```



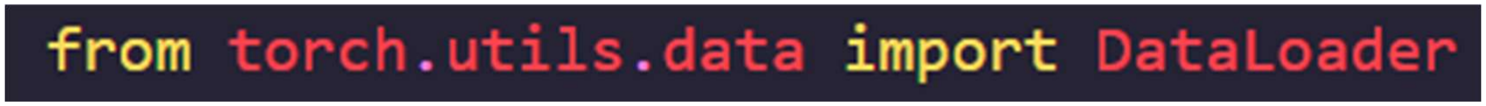
```python
    train_data = datasets.MNIST(
        root="./data", train=True, download=True, transform=transform
    )
```



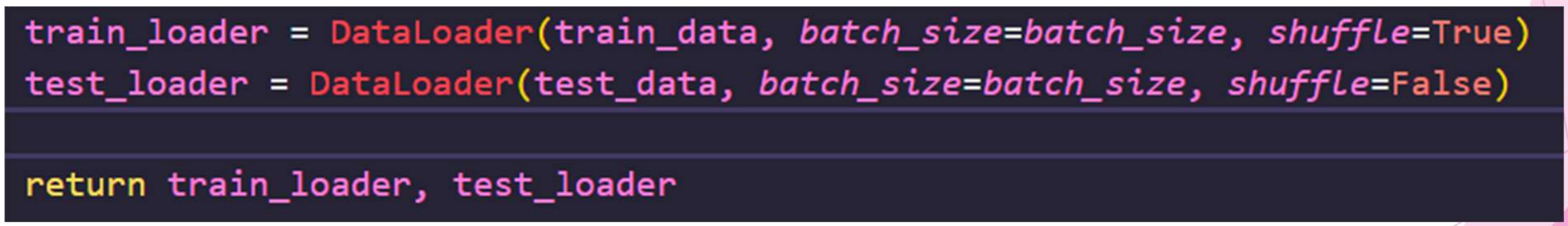
```python
test_data = datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)
```

Let’s batch the data

- Data Loader batches our data which makes it usable with our model.



```
from torch.utils.data import DataLoader
```



```
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

return train_loader, test_loader
```

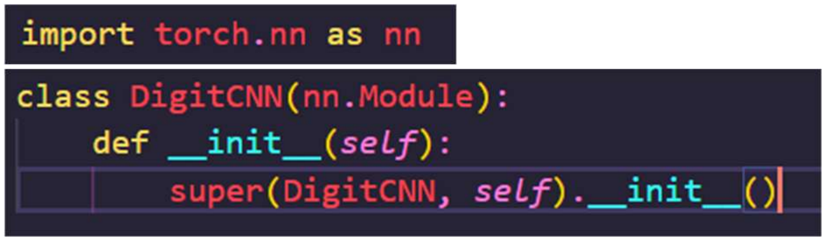
Convolutional Neural Network (CNN)

- A type of neural network that is especially effective in pattern recognition, which works perfectly for our case.
- It works by stacking linear ‘filters’ to find patterns in our data.
- A convolution layer takes a tiny grid of numbers called a filter. It slides that filter over the image. At each position it multiplies the filter with the pixels underneath and sums the result. That produces one number.
- Doing this everywhere produces a feature map. Using many filters produces many maps. Reusing the same filter everywhere forces the network to learn patterns that work no matter where they appear.

CNN Continued

- Let’s say our image is an object in a box in a factory that identifies what an object is. A CNN will travel through the factory stopping at specialized inspecting stations, each looking for a specific part.
- Each inspector looks for a specific pattern (our filters), like a line, curve, or corner, sliding their ‘window’ over the object and marking where they see that pattern.
- This results in the feature map, showing where those patterns were found.
- We then simplify these maps into smaller maps (called pooling), keeping the strongest features.
- We finally use these maps to decide based on what the inspectors found on their maps

Let’s create our CNN structure!



```
import torch.nn as nn
```

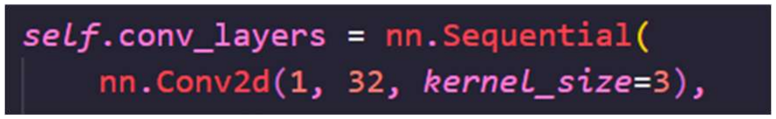
```
class DigitCNN(nn.Module):
    def __init__(self):
        super(DigitCNN, self).__init__()
```

- This initializes the nn.module(), setting up the internal workings of PyTorch that tracks all the layers, weights, and gradients in the network, making training work correctly.

CNN Structure continued

This should go right under the super(…) line.

- nn.Sequential is callable module that has each layer in our NN in order.

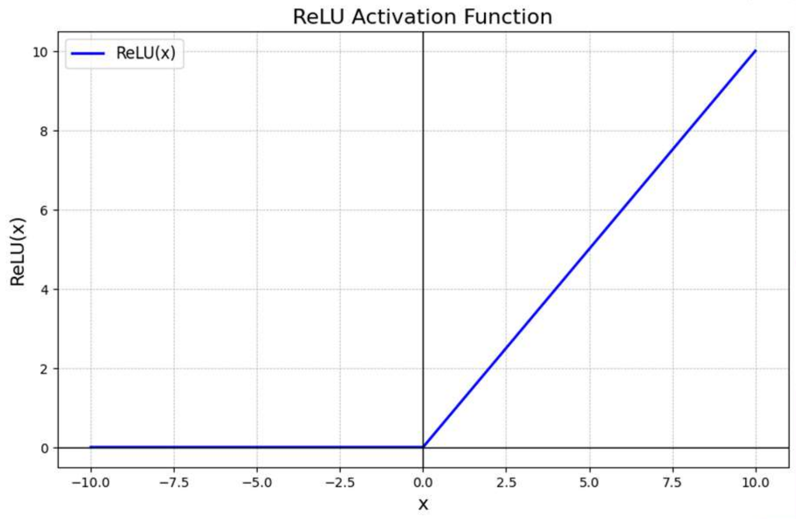


```
self.conv_layers = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),
```

- First, the Conv2d is the convolutional layer, applying our filter to the image data to detect the features.
- The 1 represents the input channels: only having 1 means the images must be in grayscale.
- The 32 means that we create 32 different filters looking for different features. This makes our feature maps!
- The kernel_size=3 represents how big our sliding window across the image is, 3 representing a 3x3 pixel window.

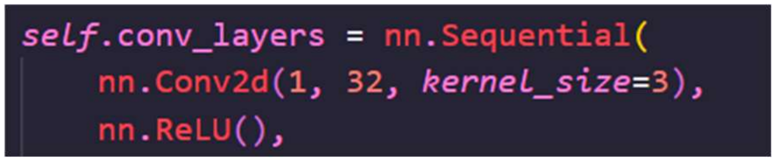
CNN Continued (Nonlinear Activation)

- We then do the nonlinear activation we talked about earlier, in our case ReLU (Rectified Linear Unit).
- Without this, each layer is just a linear transformation and stacking these would result in one big linear transformation. This nonlinear activation adds a break in those linear lines.
- ReLU does this very simply with f(x) = max(0, x), keeping positive values unchanged and changing negative values to 0. There are some advantages and disadvantages to this function, but for our purposes it works well.



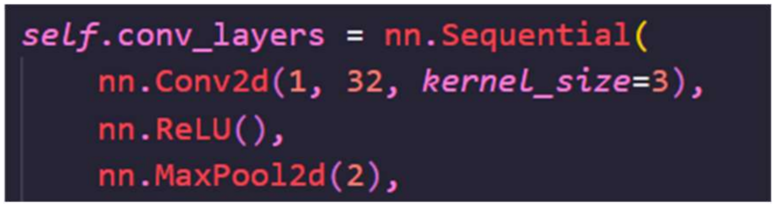
CNN Structure continued

- Next, we will add this nonlinear activation



```
self.conv_layers = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),
    nn.ReLU(),
```

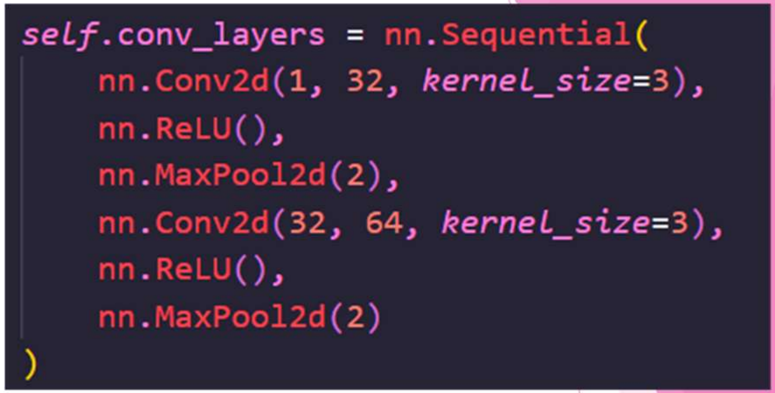
- Again, this converts all negative values to 0 and keeps the positive values as is.



```
self.conv_layers = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(2),
```

- This line is the simplification step we talked about earlier, where we keep the strongest value from each 2x2 cluster.

CNN Structure continued



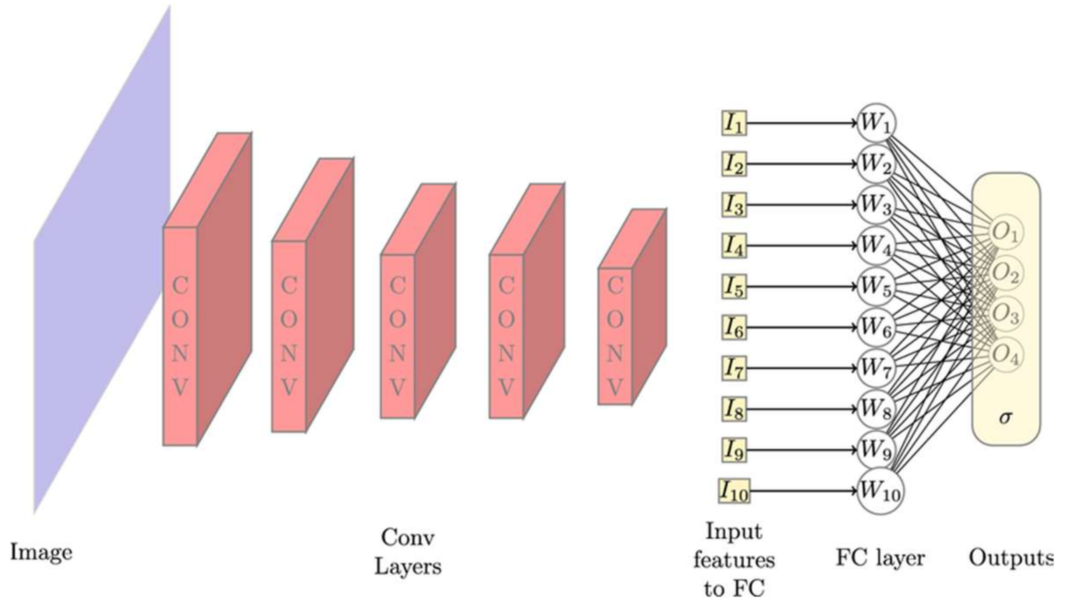
```python
self.conv_layers = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Conv2d(32, 64, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
```

- We do these steps again:
- We take the 32 filters as the channels
- We get 64 feature maps from these
- This is to detect more details and high-level shapes.

- We repeat ReLU and MaxPool2d for the same reasons as the first time.
- We have now made our stack of convolutional layers!

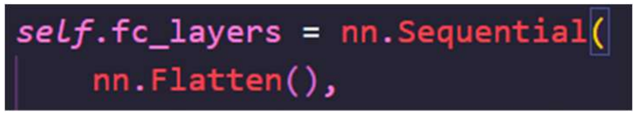
Fully Connected Layers (fc_layers)

- Once the convolutional layers are set and we have our feature maps, we take the compressed feature maps and flatten them into a single vector, integrating all the features into the predictions.


CONV
CONV
CONV
CONV
CONV
I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 I_9 I_{10}
W_1 W_2 W_3 W_4 W_5 W_6 W_7 W_8 W_9 W_{10}
O_1 O_2 O_3 O_4
σ
Image
Conv Layers
Input features to FC
FC layer
Outputs

Let’s make these fc_layers

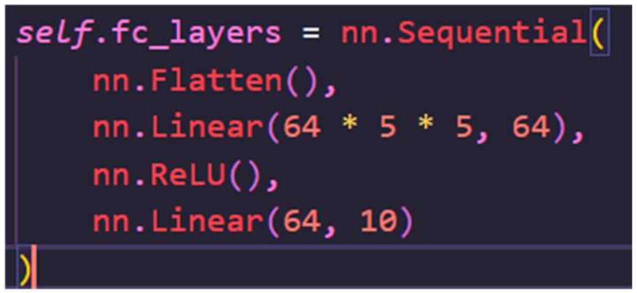
- Right under our previous code, add



```
self.fc_layers = nn.Sequential(
    nn.Flatten(),
```

- The output from the convolutional layers has a ‘shape’ of
- (**batch**, [how many images are being processed at the same time]
- **64**, [# of channels]
- **5**, [height]
- **5** [width])
- **Flatten** converts it to (batch, 64 * 5 * 5), converting to a vector that is compatible with our fc layer.

Fc_layer continued

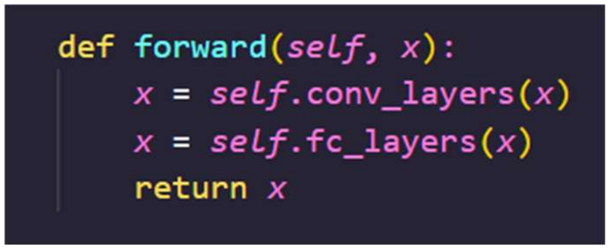


```
self.fc_layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(64 * 5 * 5, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

- **Linear** takes the flattened vector and outputs a vector of size 64, combining the extracted features into a meaningful representation that the network can work with.
- **ReLU** here makes it nonlinear as before, helping with learning more complex combinations of features.
- The 2nd **Linear** Outputs 10 scores for each digit.
- That’s it for initializing our neural network!

The final connection

- We need a way to actually use our functions!
- Add this function to our class, make sure it's **inline** with __init__, and not inside of.



```
def forward(self, x):
    x = self.conv_layers(x)
    x = self.fc_layers(x)
    return x
```

- We get a batch of images, x.
- We put it through our convolutional layers. X is now a stack of feature maps.
- Finally, we put it through our fully connected layers, so x is now a list of our probabilities.
- We never directly call this function, it runs when we do model(images) [later]

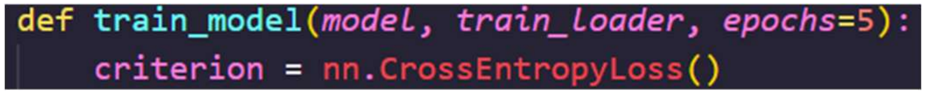
Training (Loss function)

- Now that we have our model setup to take training data, we need to figure out how to adjust the weights of our model as we show it each image.
- First, we need to figure out how to determine how far the model's predicted scores are from the correct ones.
- We will use CrossEntropyLoss as our loss function. This basically penalizes high confidence wrong answers harshly and rewards high confident correct answers heavily.
- The general training pipeline is:
- Prediction -> Loss -> Gradient -> Weight Change

  We are here ^

Let’s start the training function

- **Outside** of our class, create a new function,



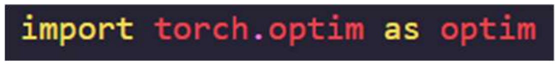
```
def train_model(model, train_loader, epochs=5):
    criterion = nn.CrossEntropyLoss()
```

- Our function will take our model, the training data loader, and we will train over our data 5 times (5 epochs).

Training (Optimization)

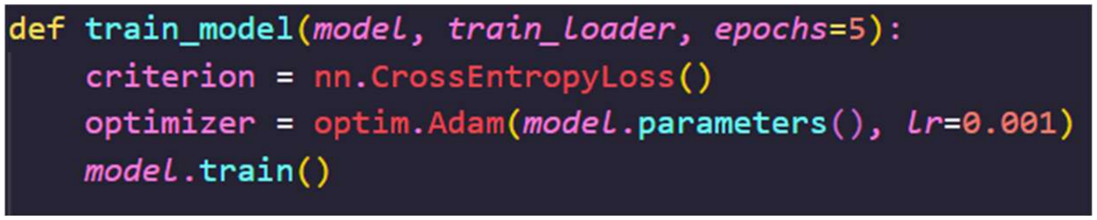
- Loss functions go hand in hand with optimizers, together helping improve a model’s accuracy.
- We will use the Adam (Adaptive Moment Estimation) optimizer, which takes the gradients from the loss function and adjusts the change in weights based on past gradients.

Let’s add the optimizer



```
import torch.optim as optim
```

- We will be using the model’s parameters and a fixed learning rate of 0.001.



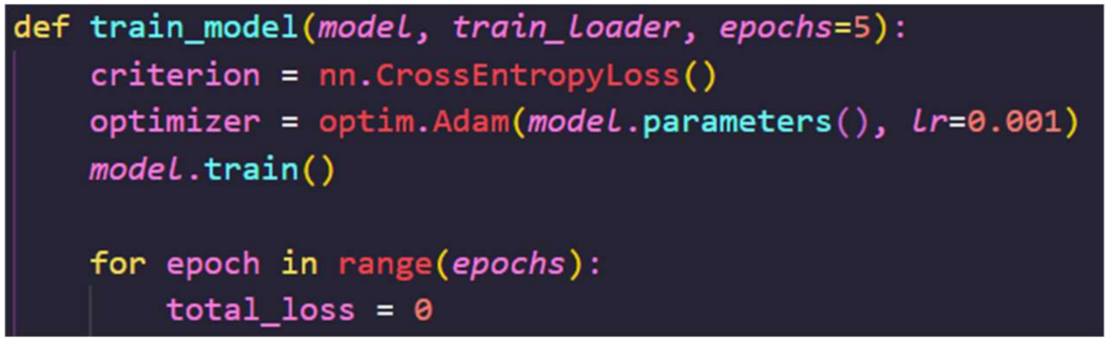
```
def train_model(model, train_loader, epochs=5):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    model.train()
```

- The way Adam works is it updates the weights based on the past gradients of the parameters.
- A gradient is vector that points towards where the greatest increase in a function is.
- The learning rate is the scale factor for each weight update, so the distance we change our parameters each time isn’t too intense.
- We will also switch the network to training mode.

Training continued

- With each epoch, the images must be processed through our convolutional layers and ReLU functions.
- We then calculate the gradients of the loss, which indicate how much each parameter contributes to the overall error. We also learn which direction to change the loss.
- The optimizer then takes these gradients and updates the parameters based on the Adam function.

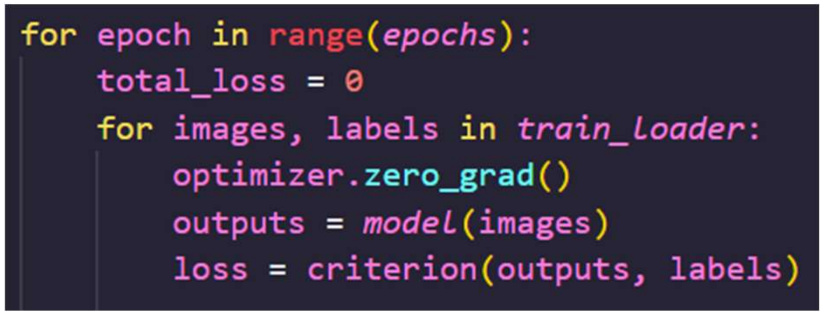
Let’s start iterating over our data



```
def train_model(model, train_loader, epochs=5):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    model.train()

    for epoch in range(epochs):
        total_loss = 0
```

- total_loss is going to keep track of total batch loss per epoch. That is the error found from calculating the loss function for each batch in the epoch.
- Instead of the calculating the total loss of the entire dataset, we will process a small batch of images and measure the accuracy for that.

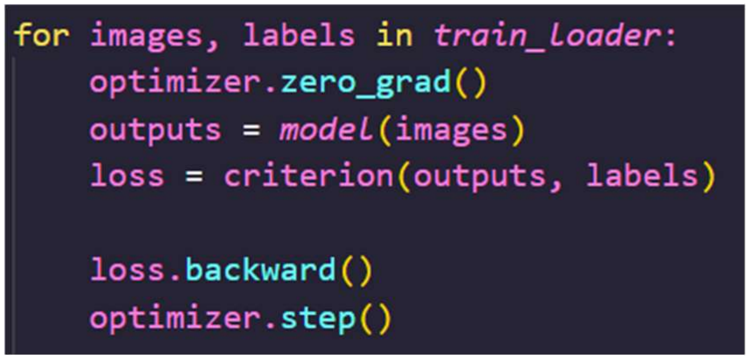
Let’s use the batch



```
for epoch in range(epochs):
    total_loss = 0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
```

- Zero_grad clears the previous gradients so they don’t keep stacking.
- We then feed the batch of images into our model (calling forward()).
- We then also use our loss function to find our loss for this batch.

A few more things

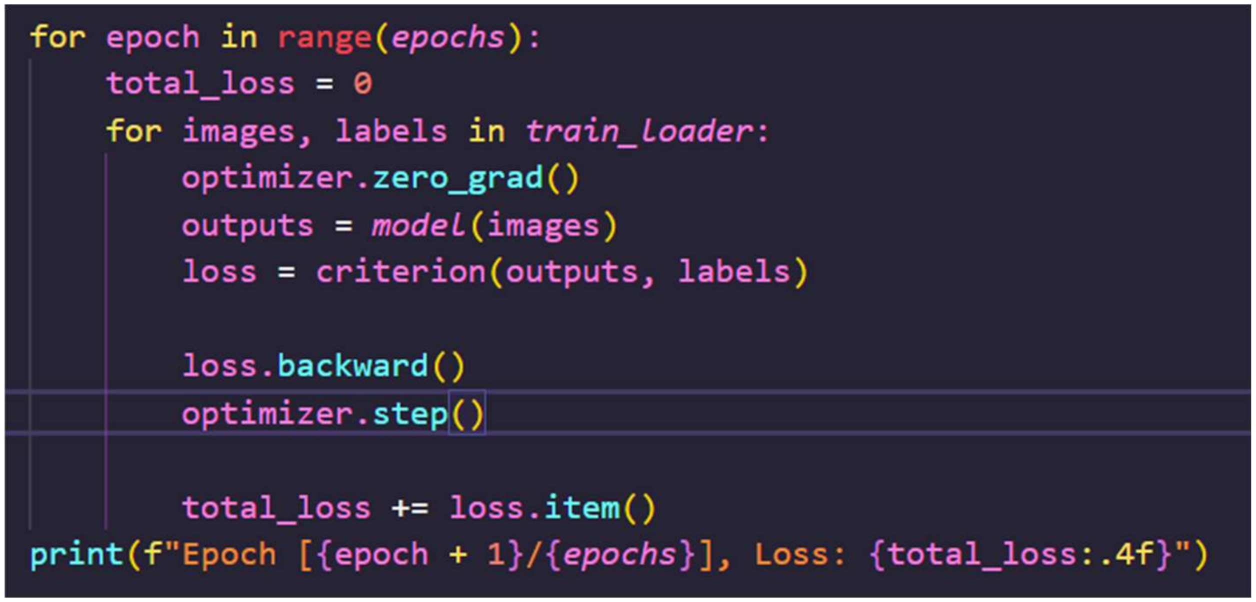


```
for images, labels in train_loader:
    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs, labels)

    loss.backward()
    optimizer.step()
```

- Loss.backward() calculates the gradient accounting for all the parameters in the network, giving the optimizer the weights contributed to the error from each parameter. It basically tells the optimizer which direction to move each weight.
- .step() applies those calculations.

And a few more…

- This should round out our training function



```
for epoch in range(epochs):
    total_loss = 0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)

        loss.backward()
        optimizer.step()

        total_loss += loss.item()
print(f"Epoch [{epoch + 1}/{epochs}], Loss: {total_loss:.4f}")
```

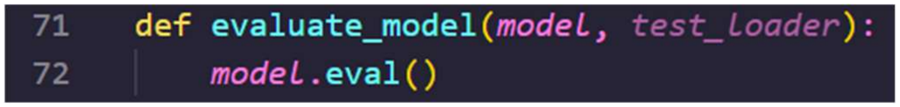
- Add the batch’s loss to the overall loss.

Evaluation

- To test the accuracy of our model, we will create an environment without gradient calculations to save system resources, as we don’t need gradients when making predictions.
- We then run the model on our test data, taking the max of the output vector which represents what digit the model is most confident in.

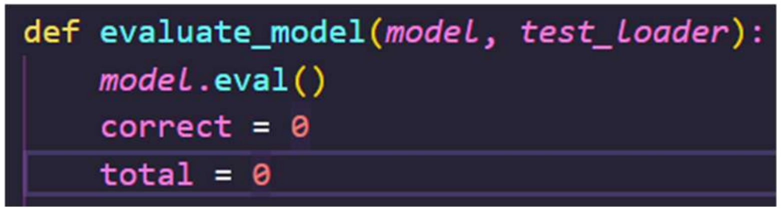
Let’s make the evaluation function

- Make sure the function is not tabbed and is outside the other functions / class.



```python
def evaluate_model(model, test_loader):
    model.eval()
```

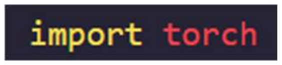
- We will put the model in evaluation mode.



```python
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
```

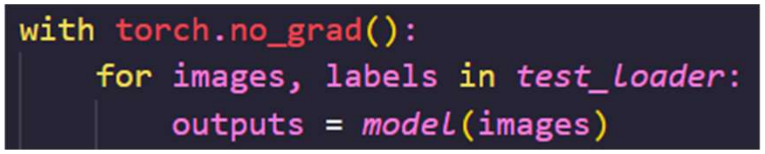
- We will count the # of images predicted correctly, and the total images we process from the test set.

Let’s iterate through our test data



```
import torch
```

- Right under our total variable, let’s add:

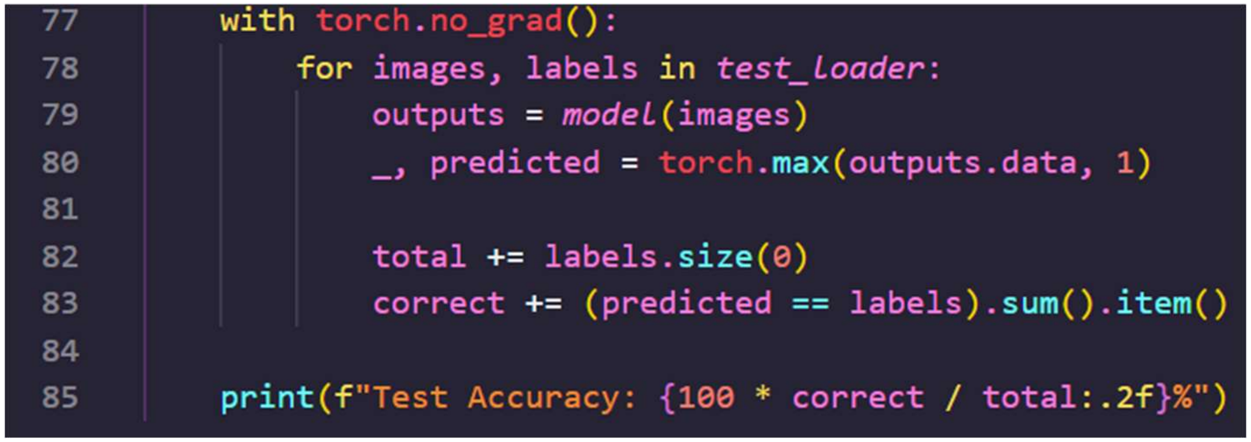


```
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
```

- Torch.no_grad creates an environment where we don’t have gradient tracking, which saves memory and increases speed. We don’t need to calculate gradients to see if our predictions are accurate.
- We run model(images) again to get our tensors of logits for our batch of images.

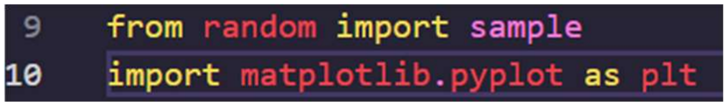
Let’s verify what our model gives us

- The underscore line gets the max value from the list of probabilities of numbers. We don’t care about the actual max value, so we ignore it with the underscore. Predicted is the digit.
- We add to our total count by the batch size, and we add to correct how many the model got correct from the batch.



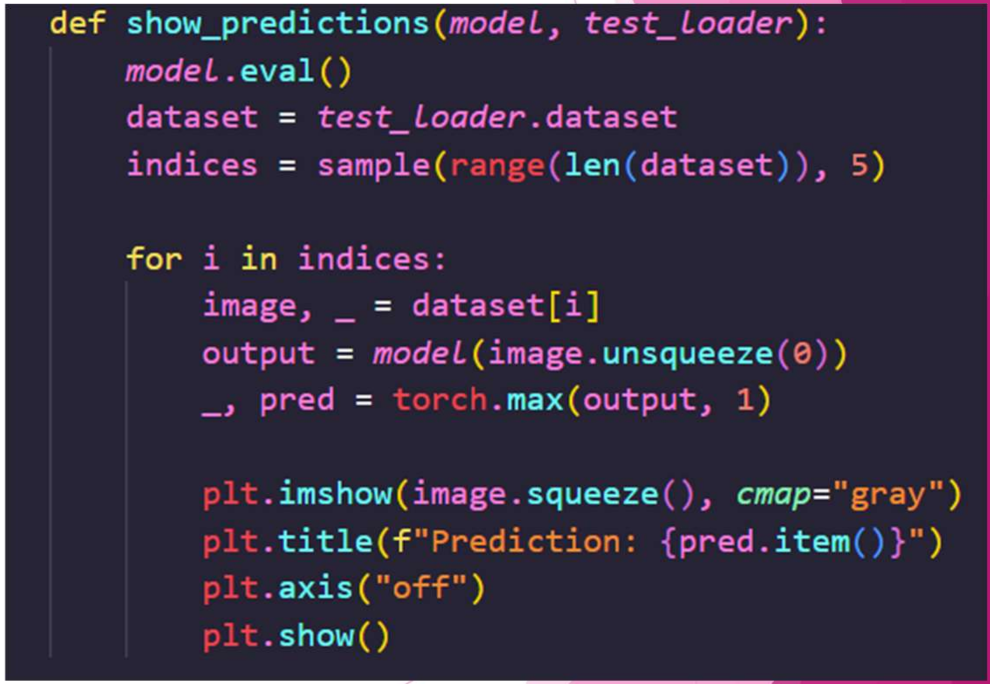
```python
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

Let’s visualize some predictions



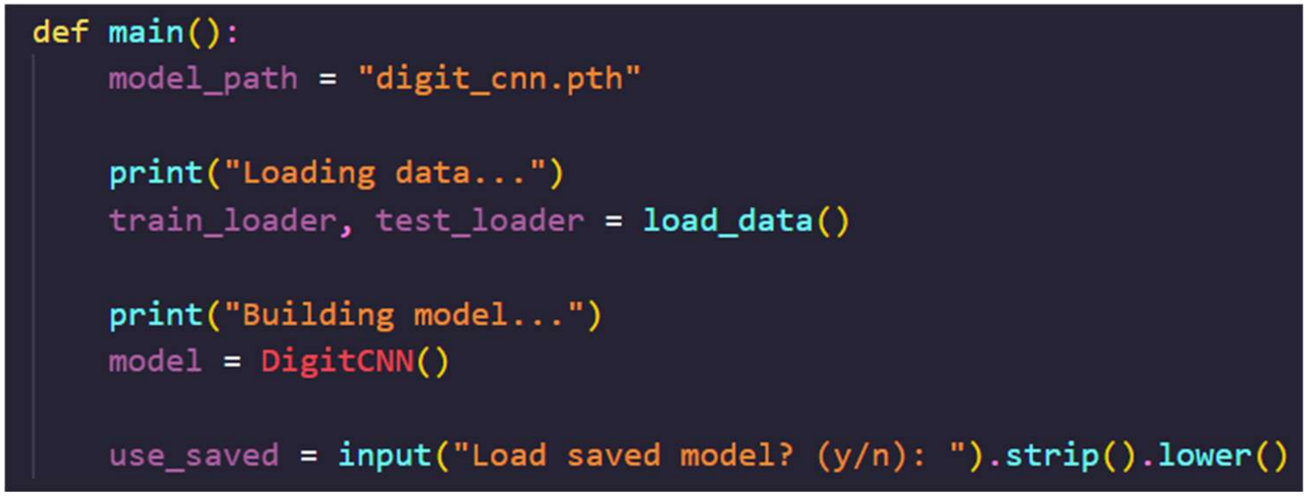
```python
from random import sample
import matplotlib.pyplot as plt
```

- We first confirm the model is in evaluation mode.
- We then use Matplotlib, which is a very popular graphing library, to show the image and the prediction we have.



```python
def show_predictions(model, test_loader):
    model.eval()
    dataset = test_loader.dataset
    indices = sample(range(len(dataset)), 5)

    for i in indices:
        image, _ = dataset[i]
        output = model(image.unsqueeze(0))
        _, pred = torch.max(output, 1)

        plt.imshow(image.squeeze(), cmap="gray")
        plt.title(f"Prediction: {pred.item()}")
        plt.axis("off")
        plt.show()
```

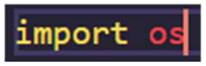
Finally, our main method!



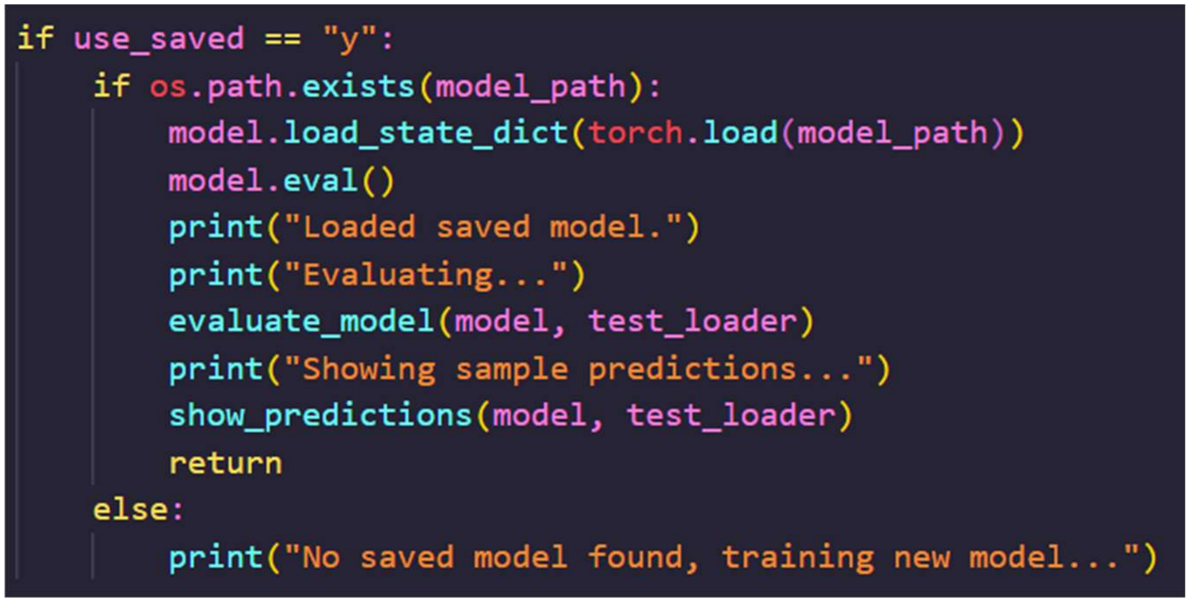
```
def main():
    model_path = "digit_cnn.pth"

    print("Loading data...")
    train_loader, test_loader = load_data()

    print("Building model...")
    model = DigitCNN()

    use_saved = input("Load saved model? (y/n): ").strip().lower()
```

- Make model path what you want to save your model as. .pth is a PyTorch model file here.
- We’ll ask if we want to reuse a model or retrain it from scratch.

Main method continued



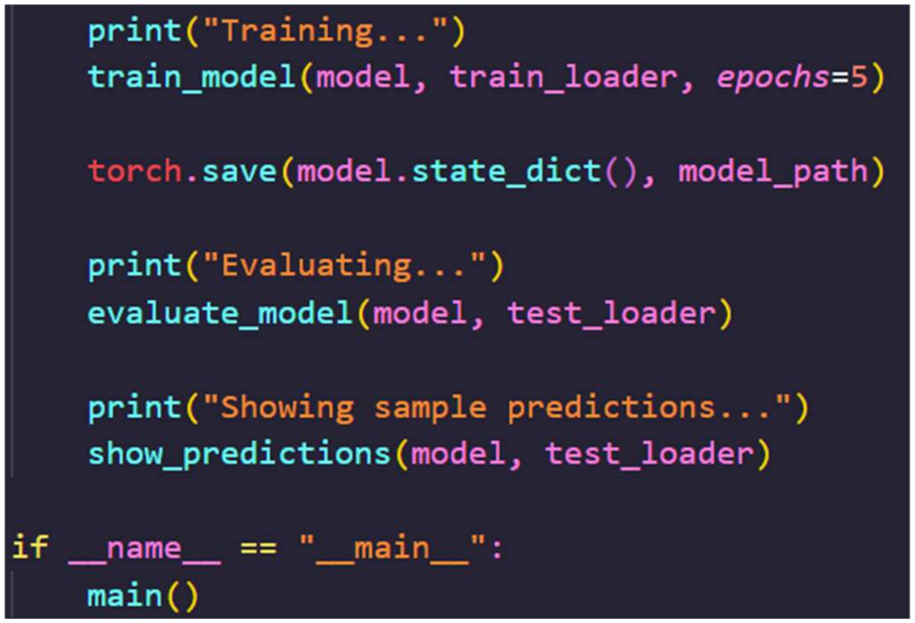
```
import os
```



```
if use_saved == "y":
    if os.path.exists(model_path):
        model.load_state_dict(torch.load(model_path))
        model.eval()
        print("Loaded saved model.")
        print("Evaluating...")
        evaluate_model(model, test_loader)
        print("Showing sample predictions...")
        show_predictions(model, test_loader)
        return
    else:
        print("No saved model found, training new model...")
```

- If we want to reuse, load the model, put it into evaluation mode, and then just run our eval method and show samples
- If there isn’t a model, then we train from scratch

Main method continued



```
    print("Training...")
    train_model(model, train_loader, epochs=5)

    torch.save(model.state_dict(), model_path)

    print("Evaluating...")
    evaluate_model(model, test_loader)

    print("Showing sample predictions...")
    show_predictions(model, test_loader)

if __name__ == "__main__":
    main()
```

- This code runs if we don’t want to reuse the model.
- Make sure the if __name__ … block is outside the main method. This is for making sure we only run our main method and don’t retrain the model every time.

What else can this model do?

- With a little refactoring, this code can be used with other datasets, especially those with simple images with distinct shapes. Our architecture of Conv2d -> ReLU -> Pool is very standard in terms of feature extraction, so we can use it with other datasets easily. For example, the following can be used without any / minimal changes:
  - Fashion-MNIST is a dataset with a bunch of different types of clothing, like sneakers, t-shirts, shoes, etc.
  - EMNIST is a dataset of handwritten letters A-Z.
  - A game like Google's Quick, Draw! uses a CNN, you would have to source your own data, but our code is a good skeleton of this project.

What else **can't** this model do?

- This model cannot tell us **where** an object is in a picture, only the probability of it being an object.
  - This leads to the 'Picasso Problem'. If you showed a CNN a picture of a face with the nose on the forehead or the eyes on the chin, it would still classify it as a face.
- This can also be extended to rotation or scale, as the model was trained on all right side up data.
- CNNs are generally biased towards textures, as it is an easier feature to classify than shape is. For example. If you have a dog with the texture of a zebra, it'll classify it as a zebra.