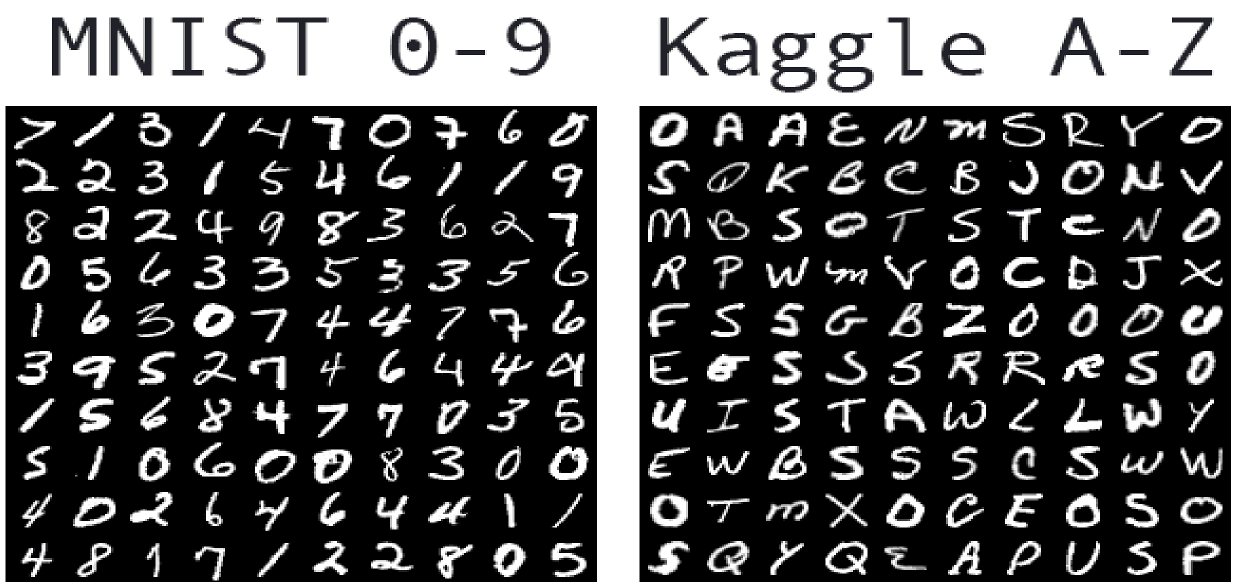
Optical Character Recognition (OCR) from scratch using PyTorch and OpenCV

I. Project datasets



In order to train our custom PyTorch model, we'll be using two datasets:

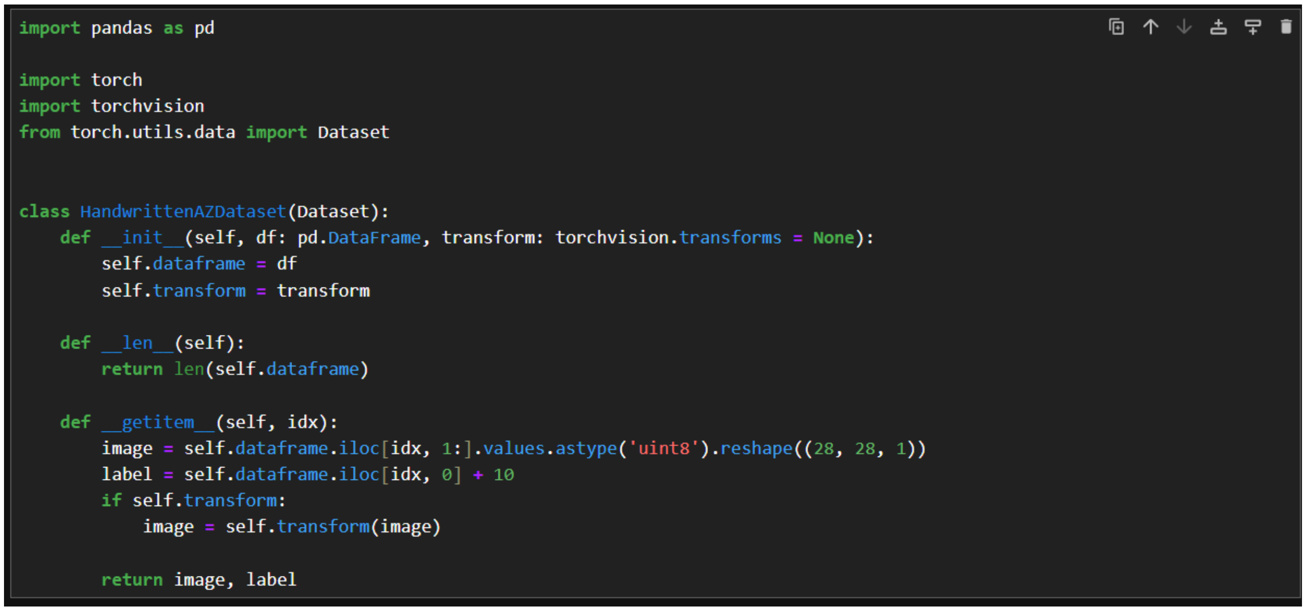
- The standard MNIST 0-9 dataset by LeCun et al.
- The Kaggle A-Z dataset by Sachin Patel, based on the NIST Special Database 19

The standard MNIST 0-9 dataset is built into popular deeplearning frameworks, including Keras, TensorFLow, PyTorch, etc. The MNIST 0-9 dataset will allow us to regconize the digits 0 – 9. Each image is a 28 x 28 gray scale image. You can read more about MNIST here.

As the MNIST dataset does not contain letters from A-Z. So, we decided to combine it with the Kaggle A-Z Handwritten dataset which contains images of capital letters from A to Z. Each image is a 28 x 28 gray scale image. The dataset was saved under the format of a CSV file which is very easy to use. You can file the dataset here.

II. Writing a custom dataset for A-Z Handwritten Data

In order to load the data from csv file and make it compatible with out PyTorch model, we will need to use torch.utils.data.Dataset to create one Dataset object for future reference with it when creating dataloader and Pandas to support interfering with CSV file data. You can learn more about how to create a custom dataset here.



```python
import pandas as pd

import torch
import torchvision
from torch.utils.data import Dataset


class HandwrittenAZDataset(Dataset):
    def __init__(self, df: pd.DataFrame, transform: torchvision.transforms = None):
        self.dataframe = df
        self.transform = transform

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        image = self.dataframe.iloc[idx, 1:].values.astype('uint8').reshape((28, 28, 1))
        label = self.dataframe.iloc[idx, 0] + 10
        if self.transform:
            image = self.transform(image)

        return image, label
```

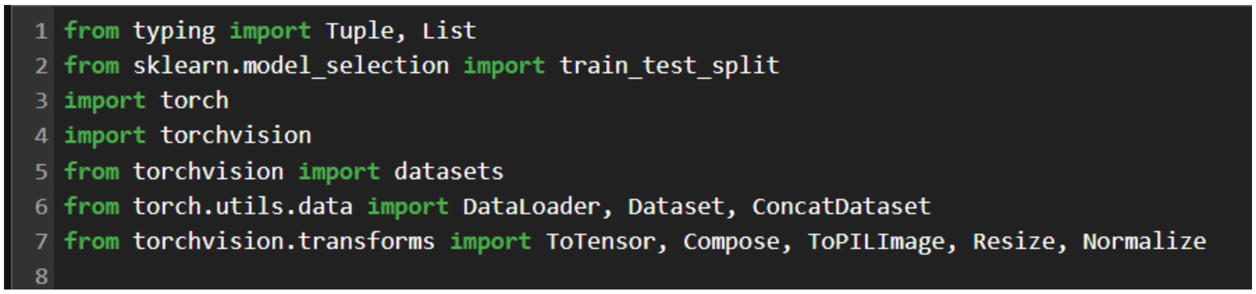
We will need to create a torch.utils.data.Dataset object that inherited from the torch.utils.data.Dataset class. The derived class have to contain the __init__ methods and escpecially two overrided methods __len__ and __getitem__.

- __init__: save the dataframe and the transform
- __len__: return the number of samples in our dataset
- __getitem__: load and return a sample at the given index. Given the index, it loads the sample and return a tuple of (image, label) and it loads image by the following ways:
    - image: Pixel values of the image will be saved in the file in the index row and from the second columns to the end. Then use .value to get the numpy array version of it, turn it into type uint8 and then finnaly, reshape it to (28, 28, 1), which is a 1 channel (gray scale), 28x28 image.
    - label: The label is the value at the first column at the given index row. Notice, because we are going to conbine this with the MNIST dataset, which will be label as the number (0-9) so we need to add an addition 10 to our label so it starts at 10 and not overlap the MNIST dataset labels.

III. Combining datasets and creating dataloaders for training

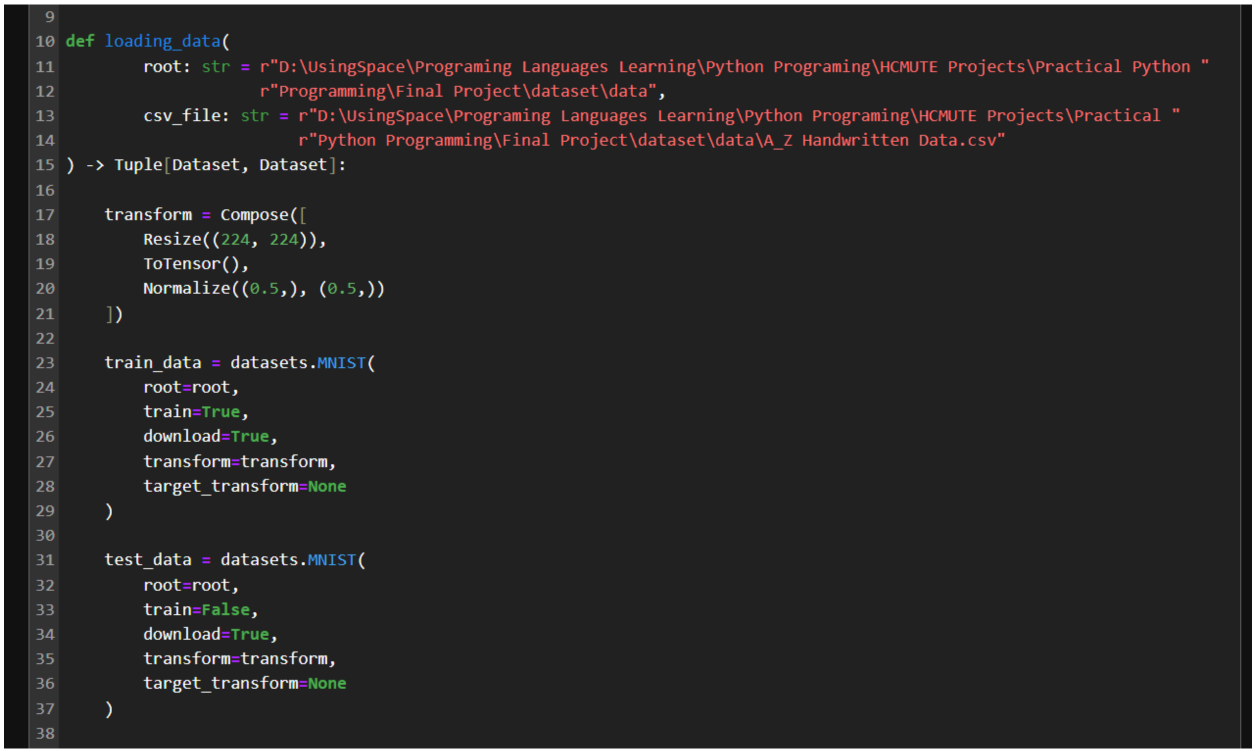
1. *Combine the two datasets*

Dependecies needed for this section.



```python
from typing import Tuple, List
from sklearn.model_selection import train_test_split
import torch
import torchvision
from torchvision import datasets
from torch.utils.data import DataLoader, Dataset, ConcatDataset
from torchvision.transforms import ToTensor, Compose, ToPILImage, Resize, Normalize

```

To combine the datasets, first we need to load the MNIST dataset, which was built in PyTorch. We will be using torchvision.datasets.MNIST, you can read more how to implement this here.



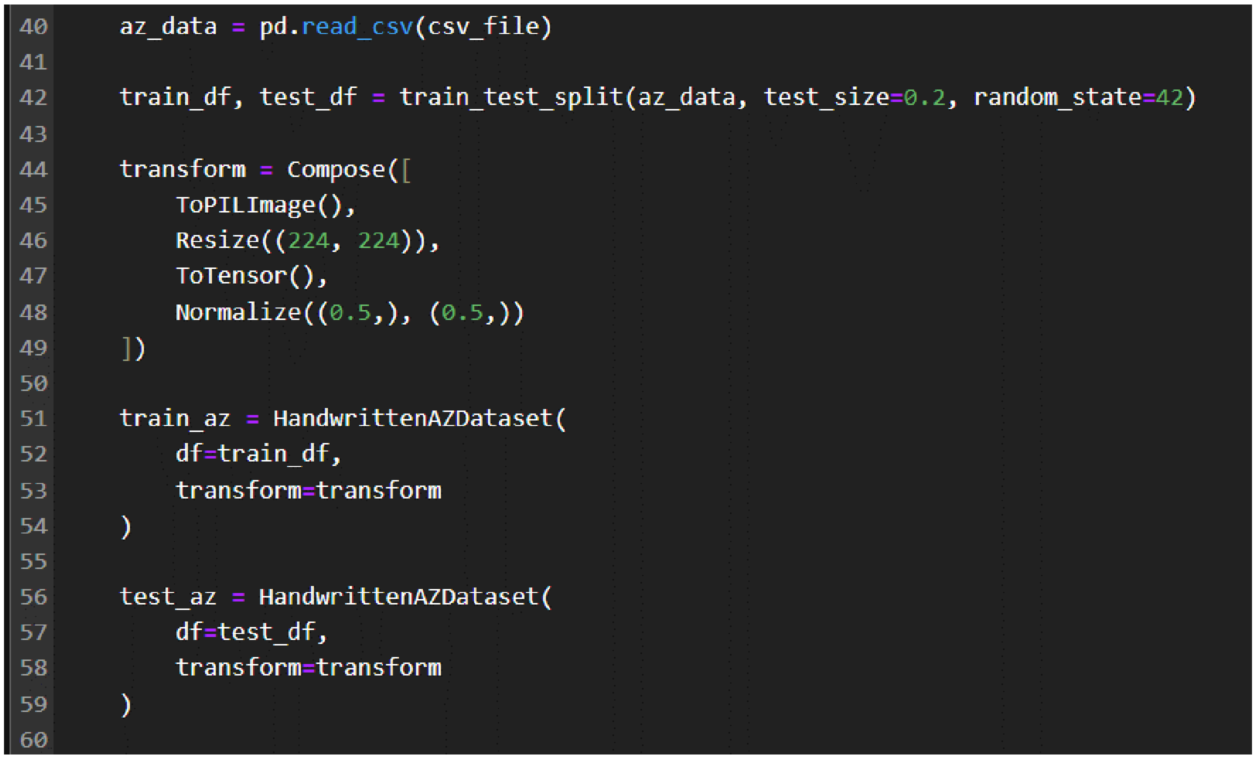
```python

def loading_data(
        root: str = r"D:\UsingSpace\Programing Languages Learning\Python Programing\HCMUTE Projects\Practical Python "
                    r"Programming\Final Project\dataset\data",
        csv_file: str = r"D:\UsingSpace\Programing Languages Learning\Python Programing\HCMUTE Projects\Practical "
                        r"Python Programming\Final Project\dataset\data\A_Z Handwritten Data.csv"
) -> Tuple[Dataset, Dataset]:

    transform = Compose([
        Resize((224, 224)),
        ToTensor(),
        Normalize((0.5,), (0.5,))
    ])

    train_data = datasets.MNIST(
        root=root,
        train=True,
        download=True,
        transform=transform,
        target_transform=None
    )

    test_data = datasets.MNIST(
        root=root,
        train=False,
        download=True,
        transform=transform,
        target_transform=None
    )

```

The transform object created using torchvision.transforms.Composed, which used to stack transformations for the image. For the MNIST dataset, we will be doing three transformations:

- torchvision.transforms.Resize: Scale the image from 28 x 28 to 224 x 224, since the model we are about to use designed for 224 x 224 inputs. Do this will help the model learns small details better than redesign a smaller model.
- torchvision.transforms.ToTensor: It turn the image to a numerical torch.Tensor object which PyTorch uses to calculate.
- torchvision.transforms.Normalize: Scale the input so it to the range [-1, 1] which will help speed up the convergence during training and model learn better (models tend to learn better if inputs are in similar scales). The mean = 0.5 and std = 0.5 is because this is a gray scale image input so it only has 1 channel. See explaination here.

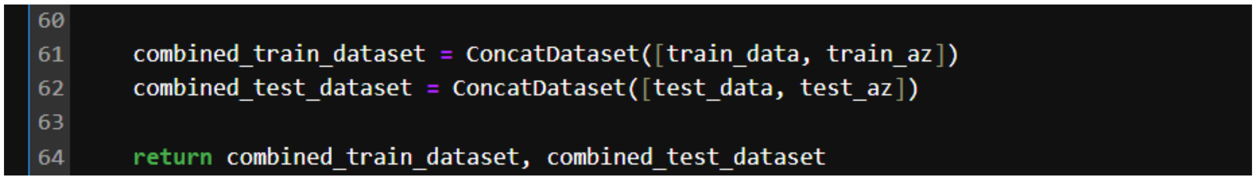
Load the training set of MNIST to train_data (set the parameter train=True), and the testing set of MNIST to test_data (set the parameter train=False).

Next, we will load the A-Z dataset using the HandwrittenAZDataset object we created earlier.



```python
    az_data = pd.read_csv(csv_file)

    train_df, test_df = train_test_split(az_data, test_size=0.2, random_state=42)

    transform = Compose([
        ToPILImage(),
        Resize((224, 224)),
        ToTensor(),
        Normalize((0.5,), (0.5,))
    ])

    train_az = HandwrittenAZDataset(
        df=train_df,
        transform=transform
    )

    test_az = HandwrittenAZDataset(
        df=test_df,
        transform=transform
    )

```

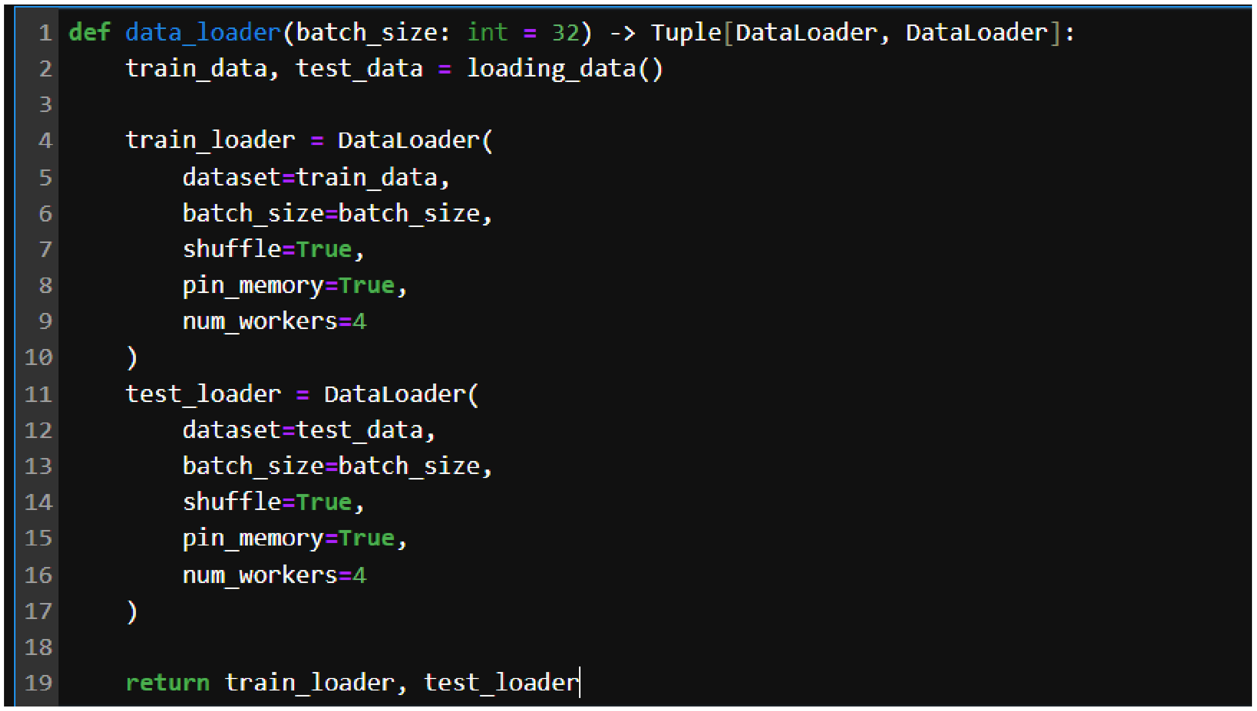
First, we read the CSV file using pd.read_csv(), pass in the csv_file which is the path to the CSV file you downloaded.
Then, we can use sklearn.model_selection.train_test_split() to split our data into 80% for training and 20% for testing by specify the test_size=0.2. Also set the random_state=42 so it will implement randomly the same every time you run.
Then create the transform object similar to what we did at the MNIST dataset. But onething different, I want to add ToPILImage to turn the numpy array we got from the class HandwrittenAZDataset to image before resize, turn it to tensor and normalize.
Finally, we create train and test A-Z dataset by passing in the train dataframe and the test dataframe from the train_test_split, along with the transform.



```python

    combined_train_dataset = ConcatDataset([train_data, train_az])
    combined_test_dataset = ConcatDataset([test_data, test_az])

    return combined_train_dataset, combined_test_dataset
```

After got all the datasets ready, we use ConcatDataset to combine out datasets. Then return the combined train dataset and combined test dataset.

2. *Create dataloader*



```python
def data_loader(batch_size: int = 32) -> Tuple[DataLoader, DataLoader]:
    train_data, test_data = loading_data()

    train_loader = DataLoader(
        dataset=train_data,
        batch_size=batch_size,
        shuffle=True,
        pin_memory=True,
        num_workers=4
    )
    test_loader = DataLoader(
        dataset=test_data,
        batch_size=batch_size,
        shuffle=True,
        pin_memory=True,
        num_workers=4
    )

    return train_loader, test_loader
```

For convience, we created a function to create dataloaders, given the batch_size and return train and test dataloaders.
First we load the combined training and testing data using the load_data() function we have created earlier.
Then pass it to DataLoader object and get the train_loader and test_loader.

- dataset: where you pass in the dataset
- batch_size: batch size
- shuffle: shuffle the data every time we load or not, the test_loader shuffle could be optional
- pin_memory and num_workers: use to make GPU use the best of it and increase calculating speed

IV. Define model

1. *Model introduction*

The model we use for this project was a self build model base on the architecture of ResNet34. It takes in the input shape of (224, 224, 1) as which 1 is for the gray scale image.

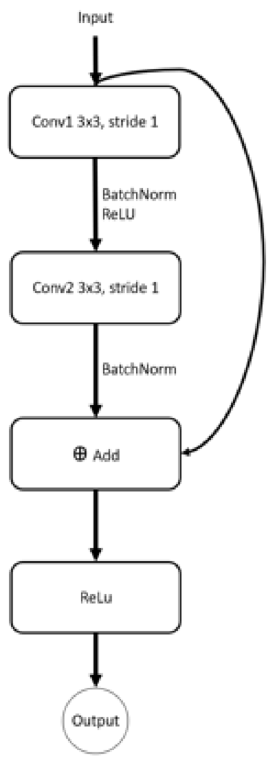
ResNet architectures are well-known for their residual blocks, which add the input of the block to the output of that block to prevent gradient vanishing when the model got too deep, so it works really well on really deep model. You can learn more about Residual Neural Network here.

The table and images underneat demonstraight how this model is constructed, the residual blocks and the model in general.

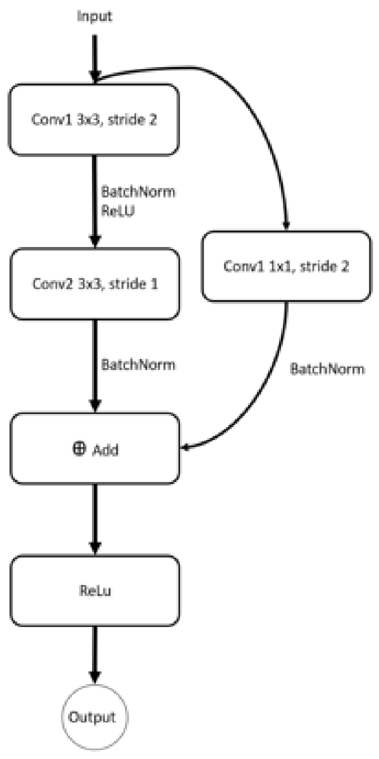
The model summary in table format

Layers name	Output size	Specific layers
conv1	112 x 112	7x7, 64, stride 2
pool1	56 x 56	3x3 max pool, stride 2
layer0	56 x 56	[3 x 3, 64, 3 x 3, 64] x 3
Layer1	28 x 28	[[3 x 3, 64, 3 x 3, 64] x 4
Layer2	14 x 14	[3 x 3, 64, 3 x 3, 64] x 6
Layer3	7 x 7	[3 x 3, 64, 3 x 3, 64] x 3
	1 x 1	Average pool, 36-d fully-connected , softmax

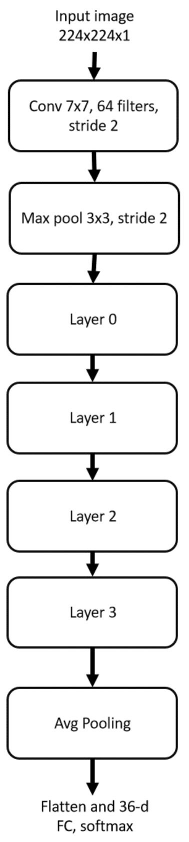
Model summary in image format



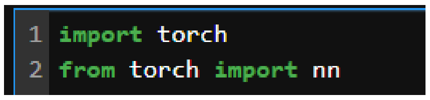
Stride 1



Stride 2

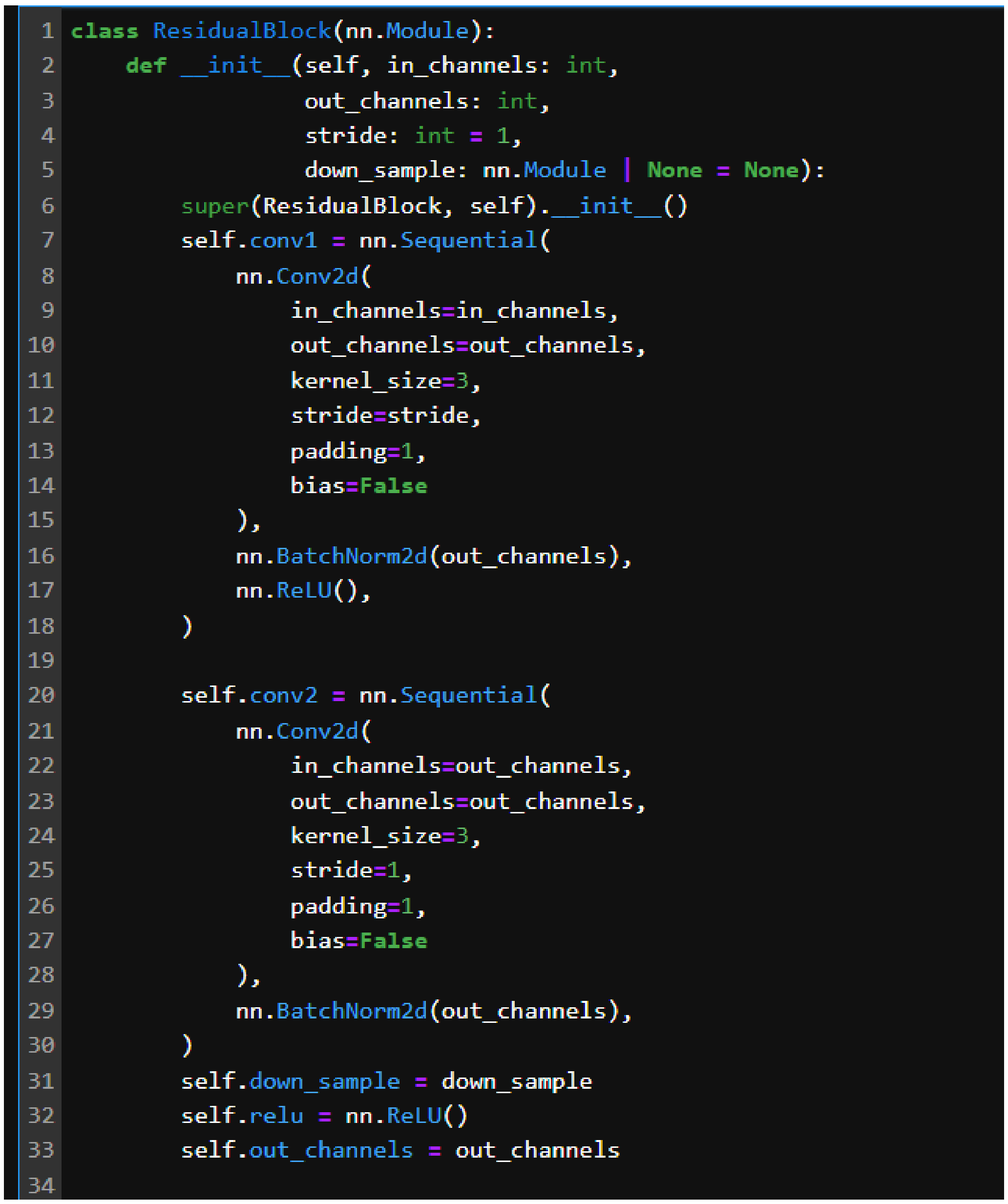


2. *<u>Create Residual Block</u>*



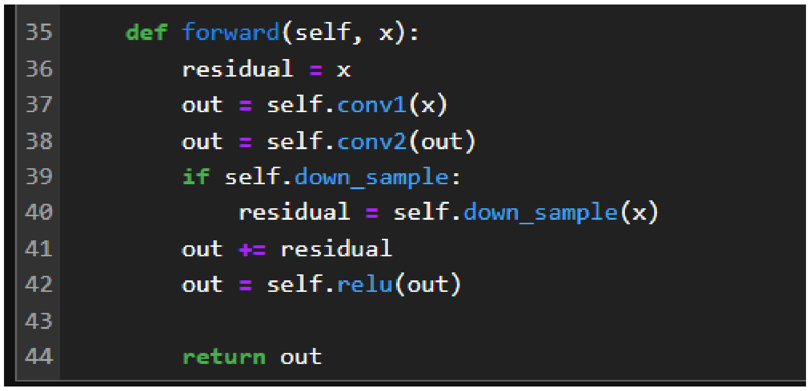
```python
import torch
from torch import nn
```

To create a layer, we can use subclassing method. By creating a class inherited from torch.nn.Module and that class has to have two methods __init__ and forward.



```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels: int,
                 out_channels: int,
                 stride: int = 1,
                 down_sample: nn.Module | None = None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=in_channels,
                out_channels=out_channels,
                kernel_size=3,
                stride=stride,
                padding=1,
                bias=False
            ),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=out_channels,
                out_channels=out_channels,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=False
            ),
            nn.BatchNorm2d(out_channels),
        )
        self.down_sample = down_sample
        self.relu = nn.ReLU()
        self.out_channels = out_channels

```

The __init__ method is used for creating layers. Always have to use super().__init__() to inititalize attribtutes in the base class torch.nn.Module then start creating layers. Here specificly, like in the image we showed you, the block has two convolutional layers. The attribute down_sample will be None by default but it uses as a saved convolutional layer for us to pass in if we need to apply pointwise convolutional to match the size of layers.



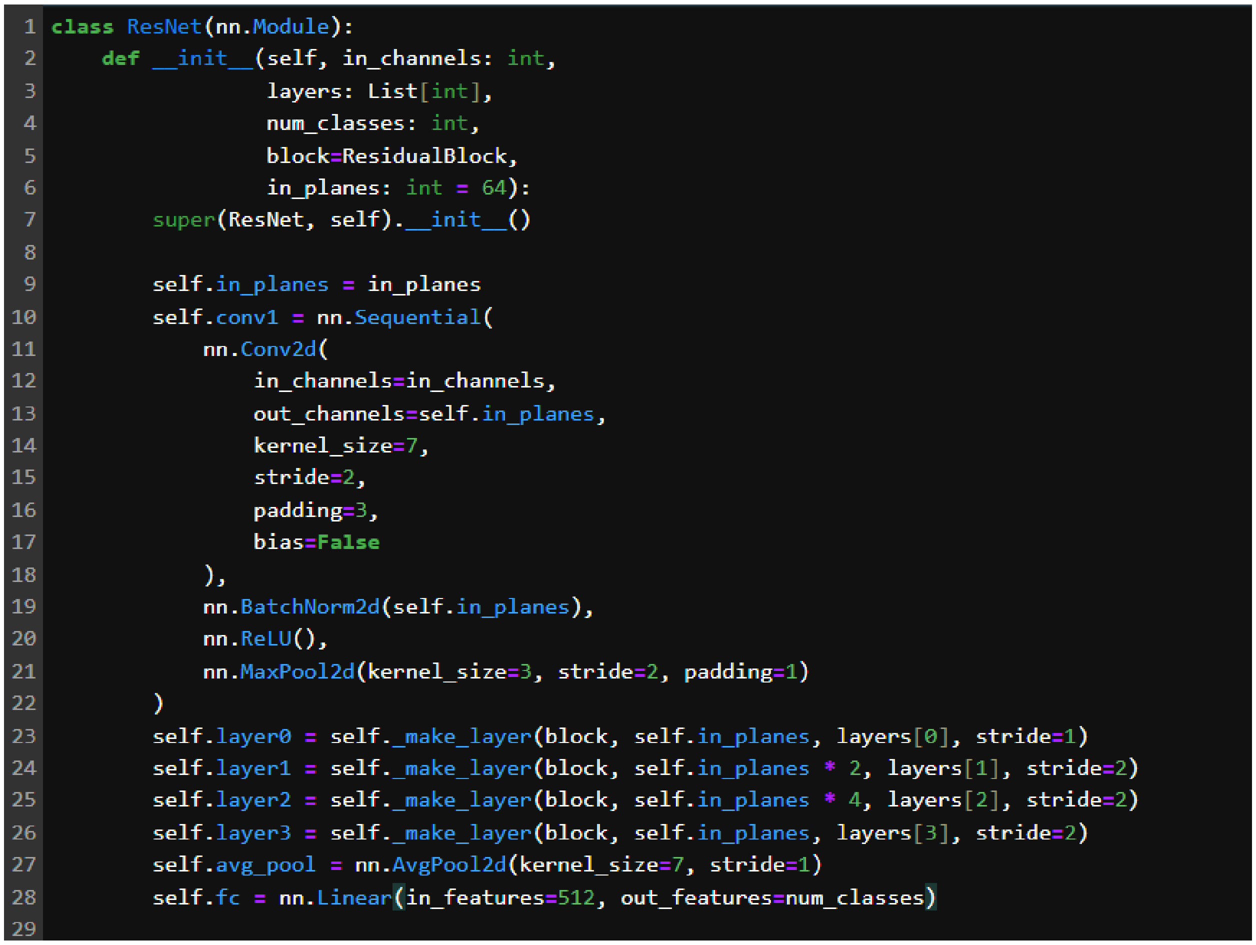
```python
    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.conv2(out)
        if self.down_sample:
            residual = self.down_sample(x)
        out += residual
        out = self.relu(out)

        return out
```

The self.forward() method used to connect layers. You can just pass in the previous layer to connect them.

We also check if self.down_sample is None or not, then we will use point wise to match the layers required size if it's required.

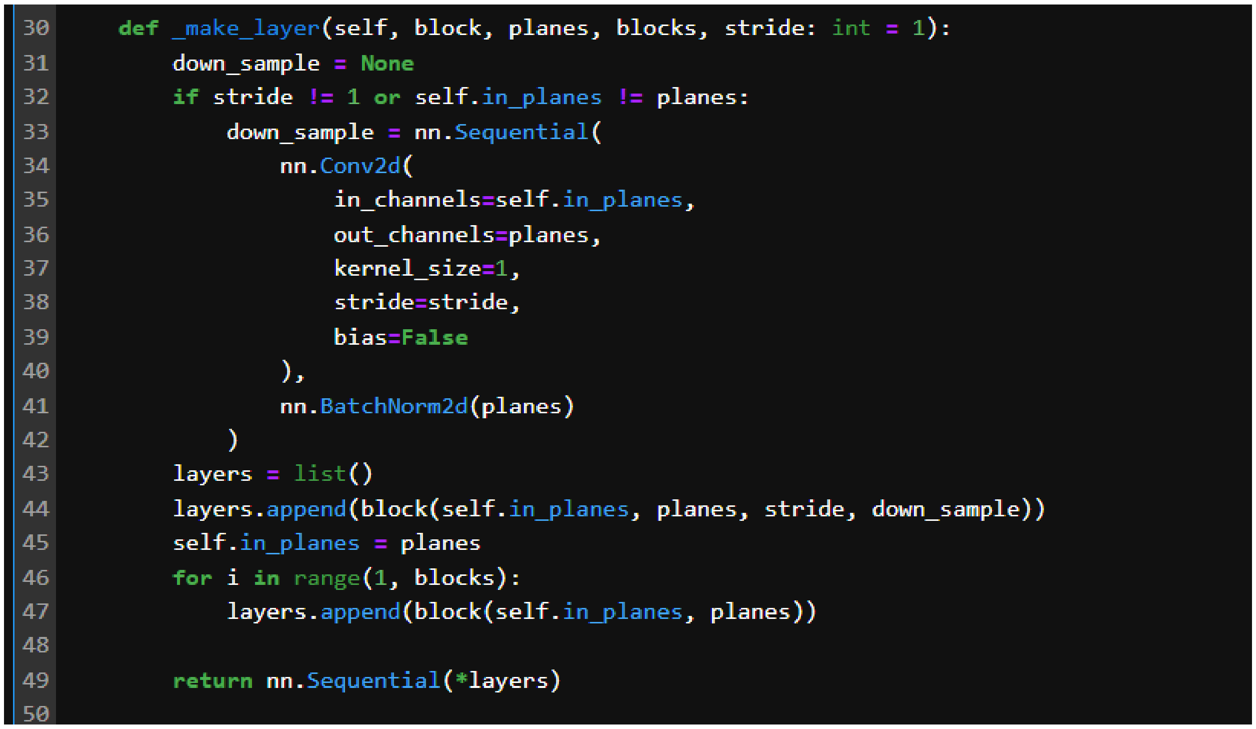
3. *Create ResNet Model*

All that above is just a layer, but we can create a model simmilarly to creating a layer since the model is just layers connected logically.



```python
class ResNet(nn.Module):
    def __init__(self, in_channels: int,
                 layers: List[int],
                 num_classes: int,
                 block=ResidualBlock,
                 in_planes: int = 64):
        super(ResNet, self).__init__()

        self.in_planes = in_planes
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=in_channels,
                out_channels=self.in_planes,
                kernel_size=7,
                stride=2,
                padding=3,
                bias=False
            ),
            nn.BatchNorm2d(self.in_planes),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.layer0 = self._make_layer(block, self.in_planes, layers[0], stride=1)
        self.layer1 = self._make_layer(block, self.in_planes * 2, layers[1], stride=2)
        self.layer2 = self._make_layer(block, self.in_planes * 4, layers[2], stride=2)
        self.layer3 = self._make_layer(block, self.in_planes, layers[3], stride=2)
        self.avg_pool = nn.AvgPool2d(kernel_size=7, stride=1)
        self.fc = nn.Linear(in_features=512, out_features=num_classes)

```

At the __init__, just creating layers similarly to what we did in the creating residual block. But there is one different thing, the self._make_layer() method. That method help us create residual blocks easier. Now let’s take a closer look to that method.



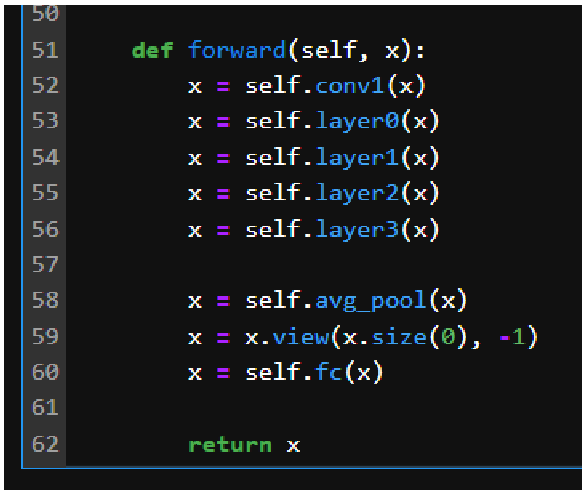
```python
    def _make_layer(self, block, planes, blocks, stride: int = 1):
        down_sample = None
        if stride != 1 or self.in_planes != planes:
            down_sample = nn.Sequential(
                nn.Conv2d(
                    in_channels=self.in_planes,
                    out_channels=planes,
                    kernel_size=1,
                    stride=stride,
                    bias=False
                ),
                nn.BatchNorm2d(planes)
            )
        layers = list()
        layers.append(block(self.in_planes, planes, stride, down_sample))
        self.in_planes = planes
        for i in range(1, blocks):
            layers.append(block(self.in_planes, planes))

        return nn.Sequential(*layers)

```

First we initialize down_sample = None in case we don’t need to use it, it will always be None.

The case stride != 1 is for countering input mismatch shape after a down sample. Resnet model will always have a stride 2 down sample every first block since the second layer, so at these, there will be mismatch shape, so a pointwise convolution layer can fix that.
The case where self.in_planes != planes is when the filters increase every beginning of a new layer, here we will need to use the pointwise convoluitional layer to match the filters from the current and previous layers.

One big layer of the model contains many residual blocks, it will get store in a list. The first block needs to use different stride and need down sample so we need to append it seperately. Then we can reset the self.in_planes so that it matches the next big layer.
The other layers can be append using a for loop because those are all stride 1 and required no down sample.

After all, we use nn.Sequential to stack all the layers we stored in the list and return that.

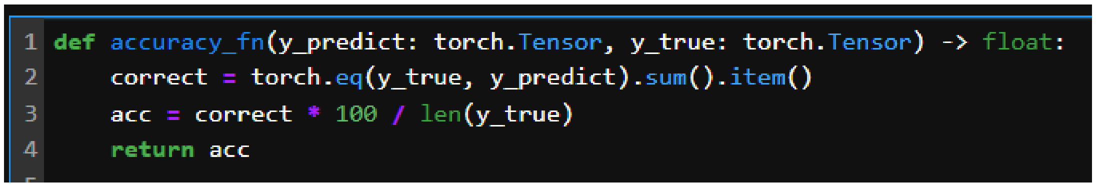


```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Now we only need to connect layers using the method forward(). After the big layers, we need an avg_pool (average pooling) for calulate the total average pool of all filters. And after the avg_pool layer, we need to flatten it before feed it forward to the fully connected layer. So we can use x.view() to change the shape of it. You can do it similarly with nn.Flatten()

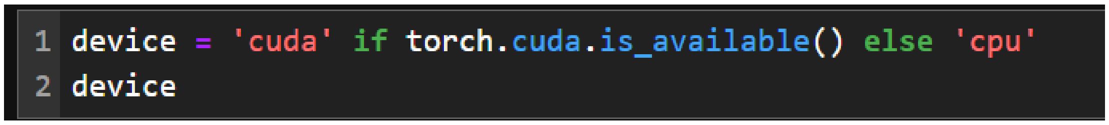
V. Training model using PyTorch

Before training model, we need first setup accuracy function, loss function and optimizer. For accuracy function, we can create a function like this.



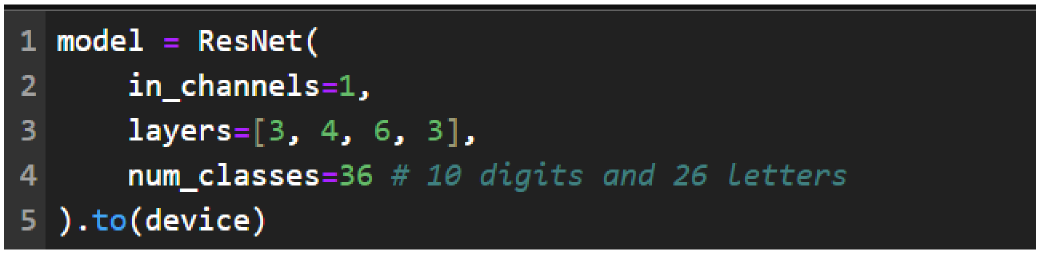
```python
def accuracy_fn(y_predict: torch.Tensor, y_true: torch.Tensor) -> float:
    correct = torch.eq(y_true, y_predict).sum().item()
    acc = correct * 100 / len(y_true)
    return acc
```

The torch.eq() will compare the two tensors element to element and return True if they are equal, False if they're not equal. Then the sum() method will calculate total of True as Trues are counted as 1. And the item() method will take out the value.
Then calculate and return the accuracy
Set up the device for training



```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
```

Then create a model for the training process to begin



```python
model = ResNet(
    in_channels=1,
    layers=[3, 4, 6, 3],
    num_classes=36 # 10 digits and 26 letters
).to(device)
```

For loss function and optimizer, we can use PyTorch built-in objects

```python
    optimizer = torch.optim.SGD(model.parameters(), lr=0.2)
    loss_fn = torch.nn.CrossEntropyLoss()
```
We try different options of optimizer and learning rate. Results showed that Stochastic Gradient Descent (SGD) optimizer with the learning rate 0.2 is the best option by far for this model. You can learn more about how Stochastic Gradient Descent works here.

The training process contains 2 main parts: training and validating. And for future convinience, we should make these 2 process functions.

- *Training*

  You will need to import these two more for the training function.

```python
from torch.cuda.amp import autocast
from torch.cuda.amp import GradScaler
```
  Those are just for optimizing the training speed.

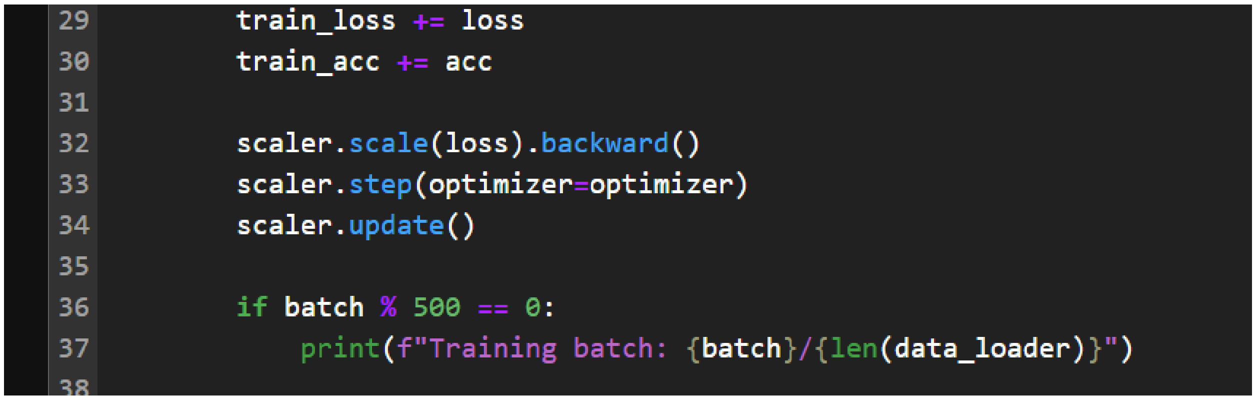
```python
def train(data_loader: DataLoader,
          model: nn.Module,
          device: torch.device,
          loss_fn: Any,
          accuracy_fn: Any,
          optimizer: torch.optim) -> Tuple[float, torch.tensor]:
    model.to(device)
    model.train()

    train_loss = train_acc = 0

    scaler = GradScaler()

    for batch, (X, y) in enumerate(data_loader):

        # optimizer.zero_grad()
        for param in model.parameters():
            param.grad = None

        X, y = X.to(device), y.to(device)

        with autocast():
            y_logit = model(X)
            y_predict = torch.softmax(y_logit, dim=1).argmax(dim=1)

            loss = loss_fn(y_logit, y)
            acc = accuracy_fn(y_predict, y)
```
  The train model will need data loader, model, device, loss function, accuracy function and optimizer. The function will return the accuracy and the loss. The model parameters will automatically update itself so we don’t need to be worried.

  First we need to set the model to device and set it to trainning mode using model.train().
  Set the train_loss and train_acc to 0
  Create a Scaler object, this will help optimizing the training speed.

  Training data will get loaded to train the model in batches, so we need a for loop to loop over all the batches to train it.

You can either use the code we showed in the picture or use optimizer.zero_grad() to reset gradient. Mine are just used to optimize training speed.
Set the image and label to the same device as the model. Use with auto_cast () will also help increasing the training speed.

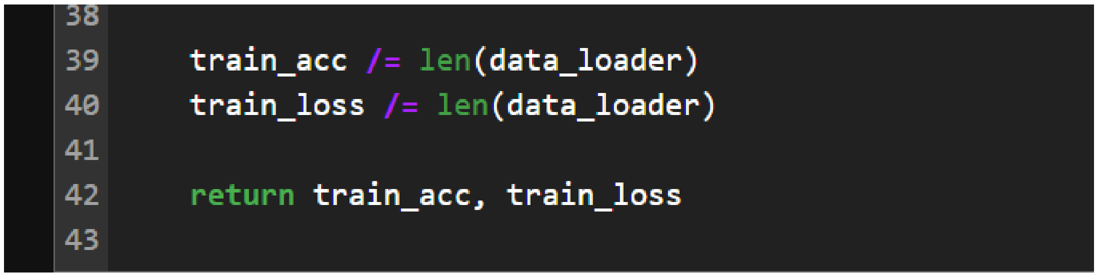
Now we can get the model's predicted logits, this is for calculating loss.
And we can also get the probability of each classes using torch.softmax() (Notice, remember to specify the dim=1 because dim=0 is just the batch size), but we combine this with argmax() method to get the predicted label as the argmax(dim=1) will return the index of the highest value in the tensor of predicted probabilites.
Calculate the batch loss by passing in y_logit and label to the loss function
Similarly with the batch accuracy, but for the accuracy, we use the y_prediction for the label since the accuracy function compares the predicted labels with the ground truth labels.



```
        train_loss += loss
        train_acc += acc

        scaler.scale(loss).backward()
        scaler.step(optimizer=optimizer)
        scaler.update()

        if batch % 500 == 0:
            print(f"Training batch: {batch}/{len(data_loader)}")
```

After calculating all the loss and the accuracy, we add up to the total train_loss and train_acc.
To understand what we're doing next, you will need to learn about backpropagation, and you can do that here. To simplify it, it's just a way to automatically derivative the function and apply Gradient Descent.
We perform back propagation by backward() method. Again, you can just simply use loss.backward().
Then perform gradient step, which is optimizing parameters (also, you can just use optimizer.step())
Notice, if you do not use scaler, there is no need for scaler.update()

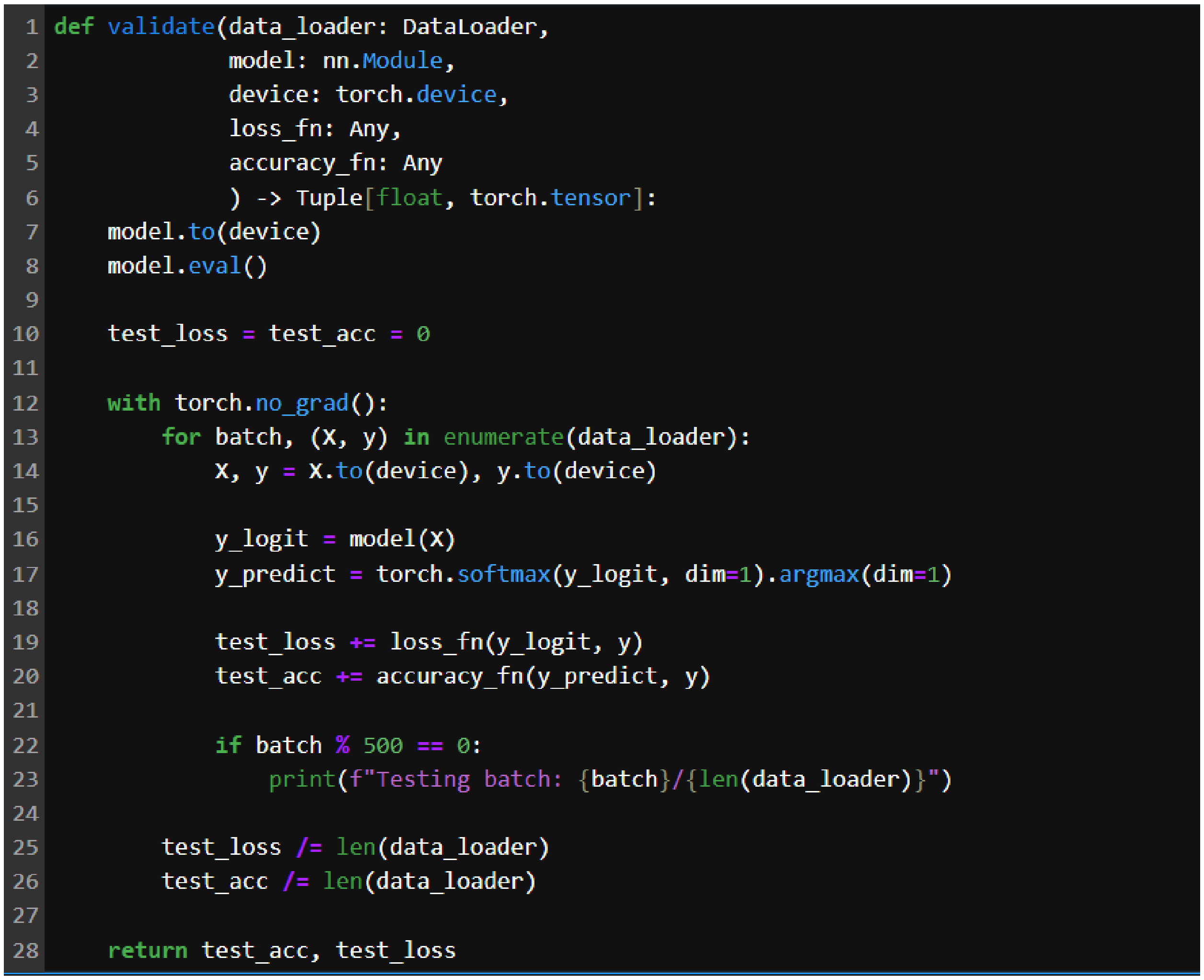
And if we have looped through 500 batch, print out the number of batch so we can check if the training is still going on or not.



```

    train_acc /= len(data_loader)
    train_loss /= len(data_loader)

    return train_acc, train_loss

```

After looping through all the batches, we devive total train_acc and train_loss to number of batches to get the average train_acc and train_loss of the training process. And return train_acc, train_loss.

- *Validating*
  The validating process is much easier. Because now no difficult calculation is involved.

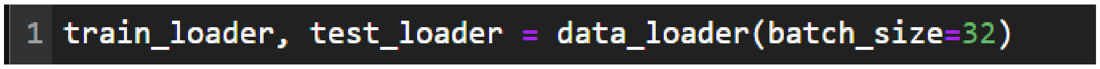


```python
def validate(data_loader: DataLoader,
             model: nn.Module,
             device: torch.device,
             loss_fn: Any,
             accuracy_fn: Any
             ) -> Tuple[float, torch.tensor]:
    model.to(device)
    model.eval()

    test_loss = test_acc = 0

    with torch.no_grad():
        for batch, (X, y) in enumerate(data_loader):
            X, y = X.to(device), y.to(device)

            y_logit = model(X)
            y_predict = torch.softmax(y_logit, dim=1).argmax(dim=1)

            test_loss += loss_fn(y_logit, y)
            test_acc += accuracy_fn(y_predict, y)

            if batch % 500 == 0:
                print(f"Testing batch: {batch}/{len(data_loader)}")

        test_loss /= len(data_loader)
        test_acc /= len(data_loader)

    return test_acc, test_loss
```

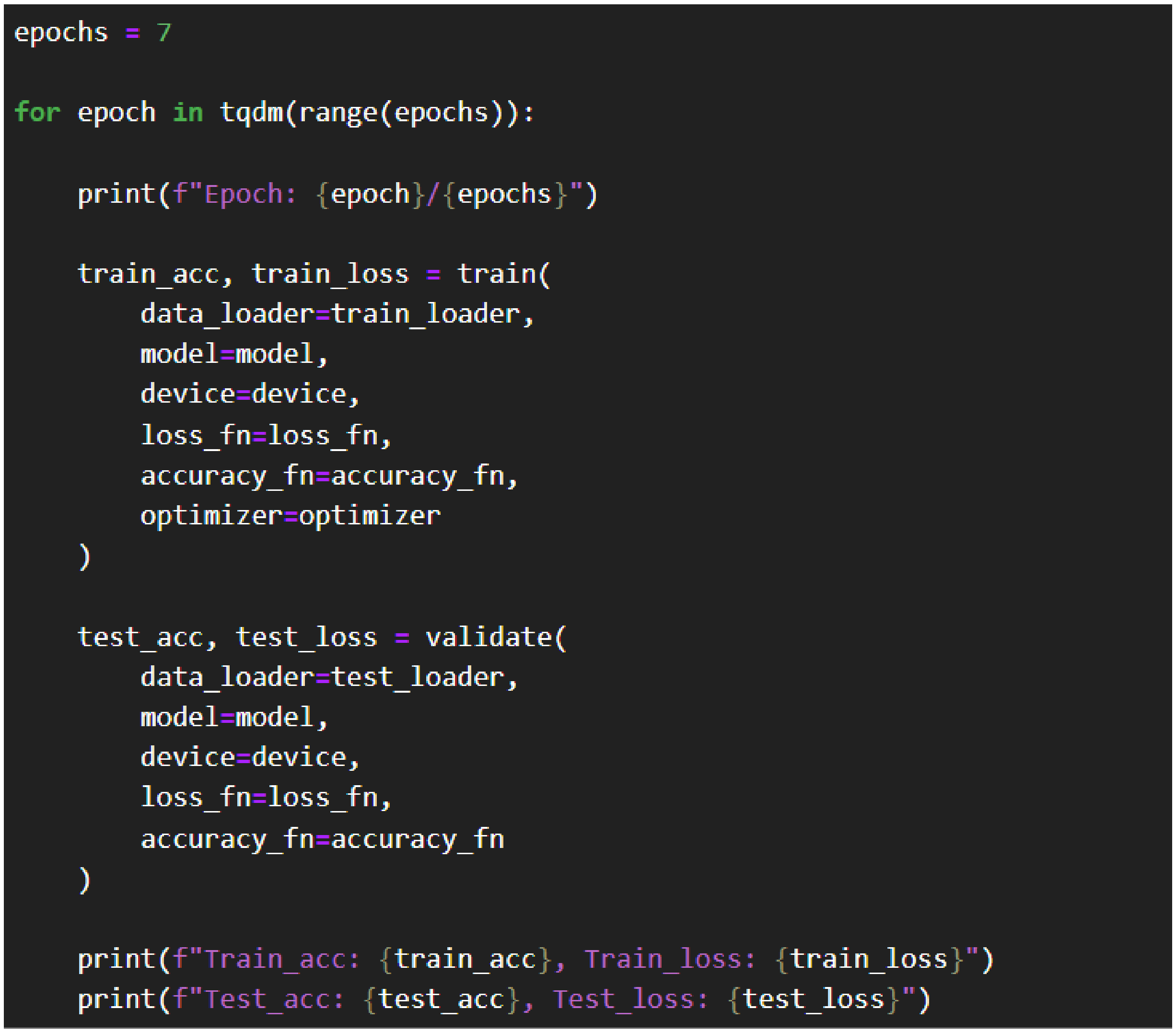
In this function , we no longer need optimizer. Still set model to the device and set it to evaluation mode by model.eval()

Put everything in size a torch.no_grad() or you can try torch.inference_mode(). What ever you feel comfortable as the two of them just use for the same purpose that just don't calculate and update the gradient.
Still calculating the test_loss and test_acc like how we did in the train function.

Now you can get the data loaders using the functions you created before.



```python
train_loader, test_loader = data_loader(batch_size=32)
```

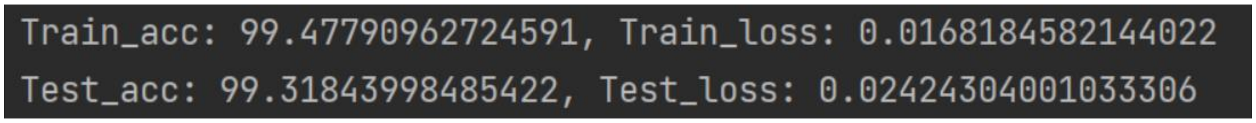


```python
epochs = 7

for epoch in tqdm(range(epochs)):

    print(f"Epoch: {epoch}/{epochs}")

    train_acc, train_loss = train(
        data_loader=train_loader,
        model=model,
        device=device,
        loss_fn=loss_fn,
        accuracy_fn=accuracy_fn,
        optimizer=optimizer
    )

    test_acc, test_loss = validate(
        data_loader=test_loader,
        model=model,
        device=device,
        loss_fn=loss_fn,
        accuracy_fn=accuracy_fn
    )

    print(f"Train_acc: {train_acc}, Train_loss: {train_loss}")
    print(f"Test_acc: {test_acc}, Test_loss: {test_loss}")
```

Now we can just use those two functions in a for loop. Here I trained it for 7 epochs but you can change it as how much you want it to be.

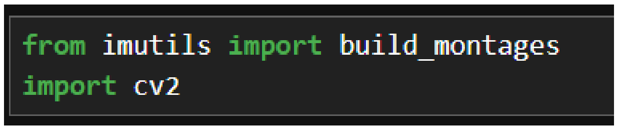
VI. Training result

The final epoch reached



```
Train_acc: 99.47790962724591, Train_loss: 0.0168184582144022
Test_acc: 99.31843998485422, Test_loss: 0.02424304001033306
```

Plot it to visualize how our model is performing. And to do that, we need opencv and build_montages from imutils. The function build_montages help create Montag, which is images in a table, you need pass in list of image, image shape, montage shape.



```python
from imutils import build_montages
import cv2
```

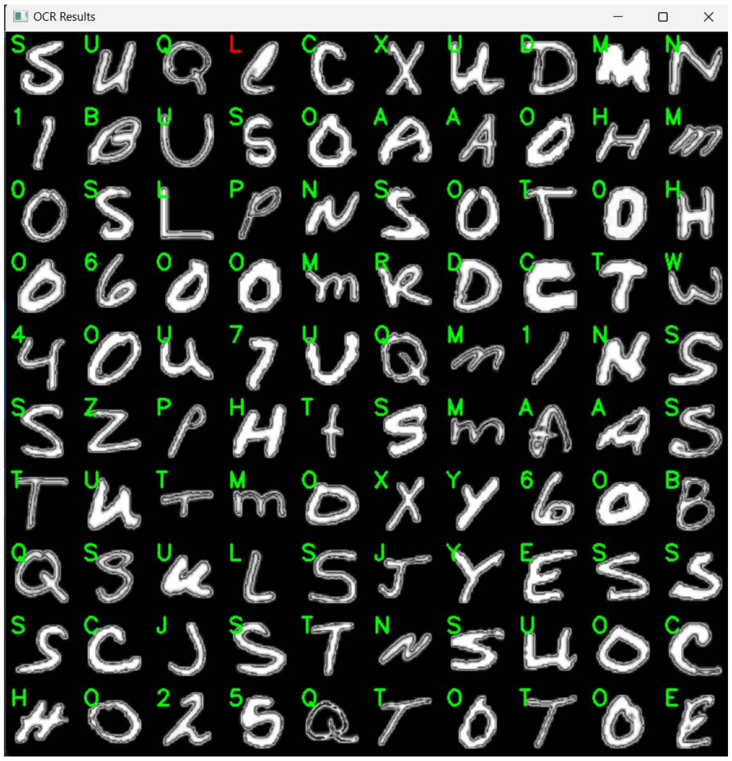
And the function use for doing the ploting is this

```python
def plot_vals(dataloader, model, device, label_names):
    images = list()
    side = 10

    for i, (img, label) in enumerate(dataloader):
        if i >= side * side:
            break
        img, label = img.to(device), label.to(device)

        with torch.inference_mode():
            probs = model(img)
            prediction = probs.argmax(dim=1).item()

            label_name = label_names[prediction]

            image = (img.squeeze().cpu().detach().numpy() * 255).astype("uint8")
            color = (0, 255, 0) if prediction == label.item() else (0, 0, 255)

            image = cv2.merge([image] * 3)
            image = cv2.resize(image, (75, 75), interpolation=cv2.INTER_LINEAR)
            cv2.putText(image, label_name, (5, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.75, color, 2)

            images.append(image)

    montage = build_montages(images, (75, 75), (side, side))[0]

    cv2.imshow("OCR Results", montage)
    cv2.waitKey(0)
```

Now you need to load the model and apply the function.

```python
train_loader, test_loader = data_loader(batch_size=1)
plot_vals(
    dataloader=test_loader,
    model=model,
    device=device,
    label_names=labelNames
)
```

The result:



You can see that our model is performing quite well, a minor mistake is C and L, but even us human could crash into that mistake.

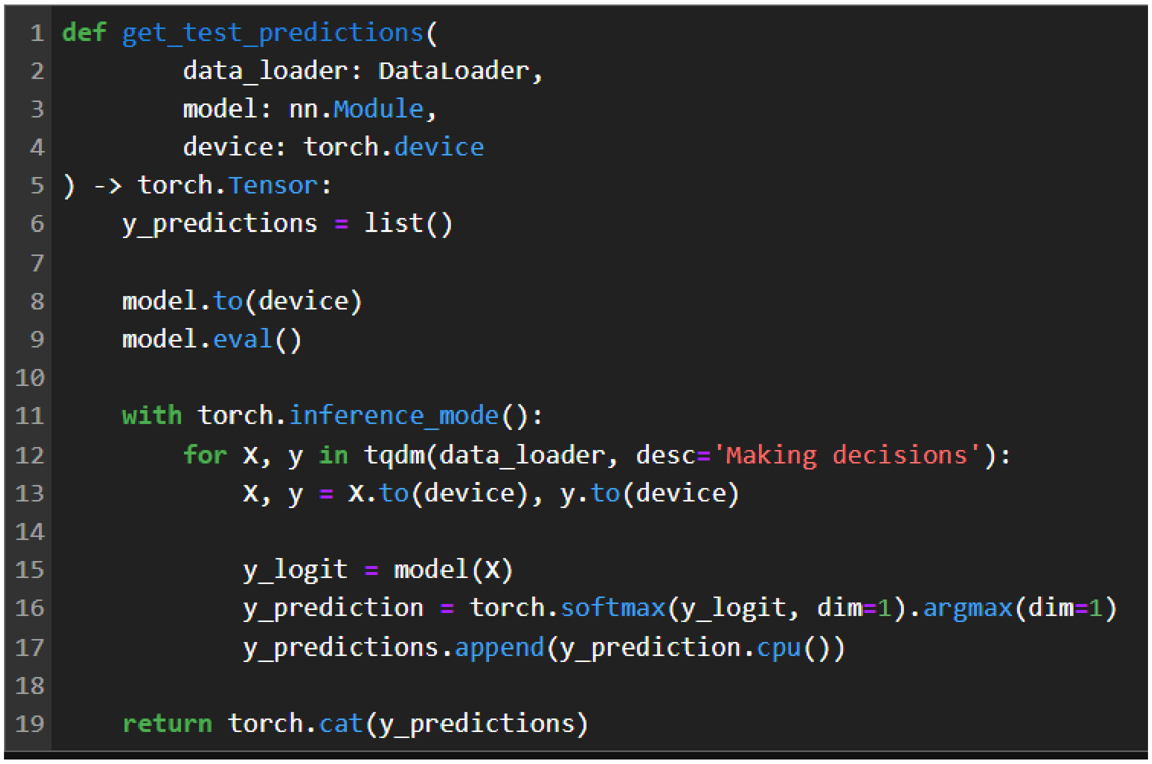
VII. Check performance using confusion matrix

Load the dependencies. The function confusion_matrix is used for create a confusion matrix and the function ConfusionMatrixDisplay help display is, you can use plt.save_fig() to save the confusion matrix if you want to.

```python
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
```

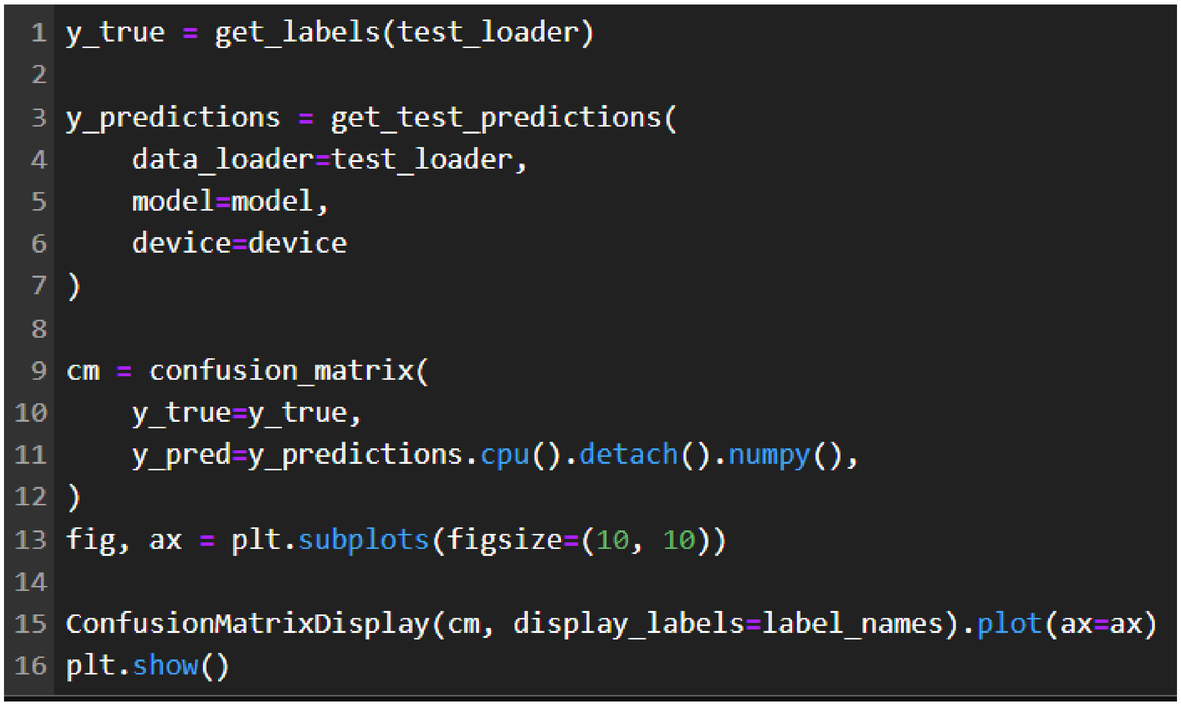
Creating some supporting functions

```python
def get_labels(dataloader: DataLoader) -> List[int]:
    labels = list()

    for _, label in dataloader:
        labels.extend(label.tolist())

    return labels
```



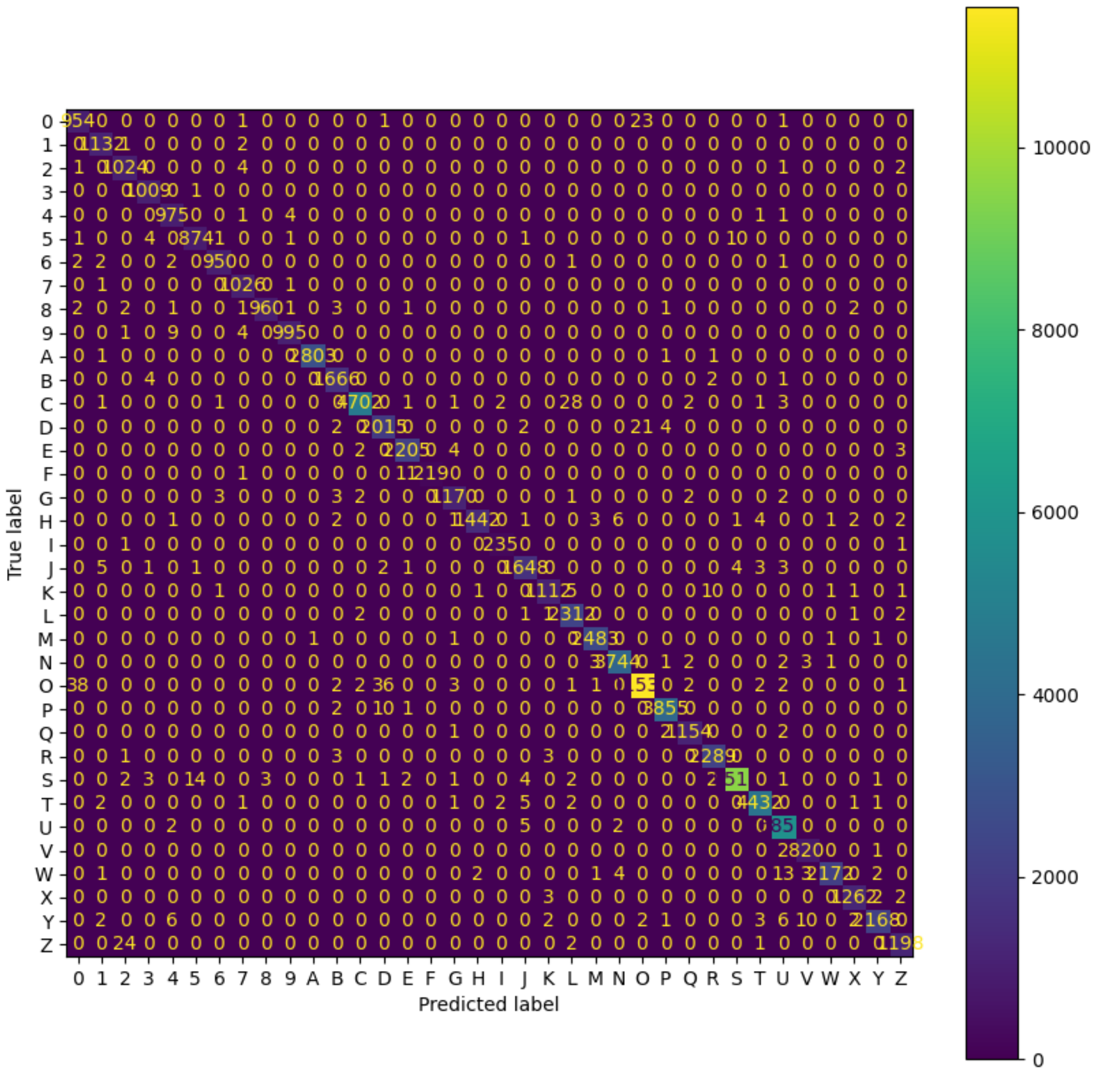
```python
def get_test_predictions(
        data_loader: DataLoader,
        model: nn.Module,
        device: torch.device
) -> torch.Tensor:
    y_predictions = list()

    model.to(device)
    model.eval()

    with torch.inference_mode():
        for X, y in tqdm(data_loader, desc='Making decisions'):
            X, y = X.to(device), y.to(device)

            y_logit = model(X)
            y_prediction = torch.softmax(y_logit, dim=1).argmax(dim=1)
            y_predictions.append(y_prediction.cpu())

    return torch.cat(y_predictions)
```

And finally, plot the confusion matrix on the test dataset



```python
y_true = get_labels(test_loader)

y_predictions = get_test_predictions(
    data_loader=test_loader,
    model=model,
    device=device
)

cm = confusion_matrix(
    y_true=y_true,
    y_pred=y_predictions.cpu().detach().numpy(),
)
fig, ax = plt.subplots(figsize=(10, 10))

ConfusionMatrixDisplay(cm, display_labels=label_names).plot(ax=ax)
plt.show()
```

The result:



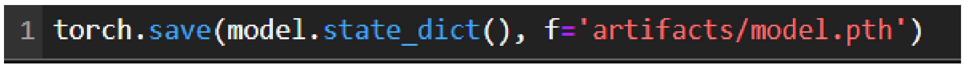
Here in the confusion matrix, the model works quite well, but there were still some main confusions between:

- 0 and O
- O and D
- 5 and S
- L and C

These are because of some handwriting of these are very confusing, something we can be mistaken too.

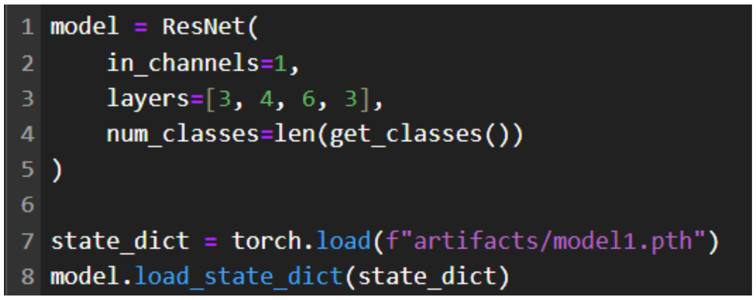
VIII. Save and loaded model

The model is working quite well, so for future use, we need to save it. Here in PyTorch, what we save is the state dict, which means we are saving the trained weights.



```
torch.save(model.state_dict(), f='artifacts/model.pth')
```

We have successfully saved the model's state dict, so now when ever we need to use the model, we can just create the model again and load the state dict.

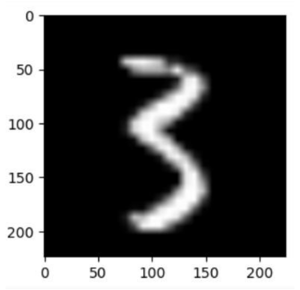


```python
model = ResNet(
    in_channels=1,
    layers=[3, 4, 6, 3],
    num_classes=len(get_classes())
)

state_dict = torch.load(f"artifacts/model1.pth")
model.load_state_dict(state_dict)
```

IX. More thorough look into the model layers

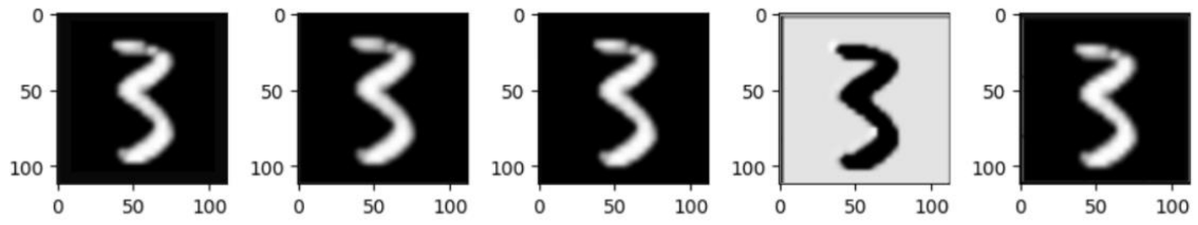
We trained the model, but what is inside of it, let's have a close look. We will be using another notebook from now on, load the model and try to learn what the layers do.
Let's sample with one image, I will just cover some significant layers. And I will be ploting

<u>Input</u>



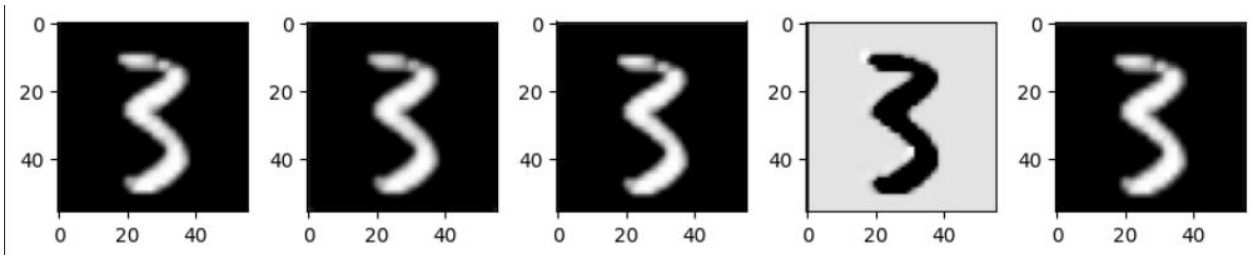
The input we sample this time is a picture of the number 3, which has the shape 224x224x1.

<u>Conv 7x7, 64, stride 2</u>



This help down sample the image, it runs over the image, keep the important information, that's why image still look pretty much the same although it size has reduced almost in half, the shape now is 112x112x64. The number 64 is the number of filters.

<u>Maxpooling 3x3 stride 2</u>

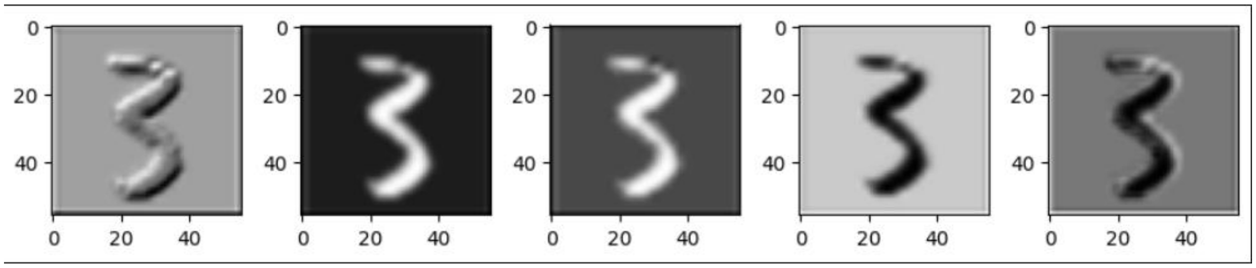


The max pool layer help reduce the image size in half (56x56x64), by taking returning the max value in each kernel. This help keep important informations.

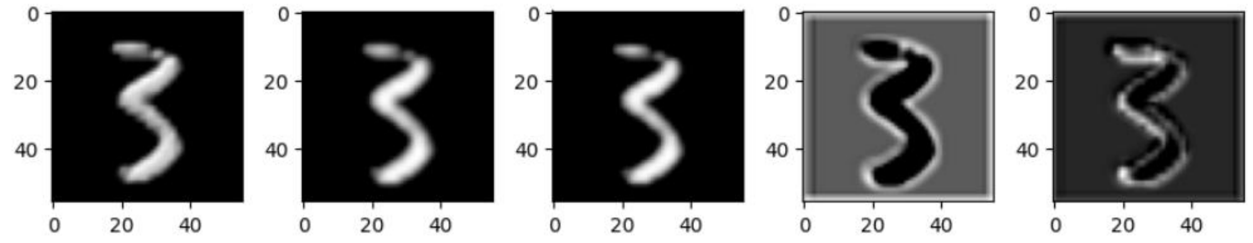
Layer0

Residual 0

Before add with the residual

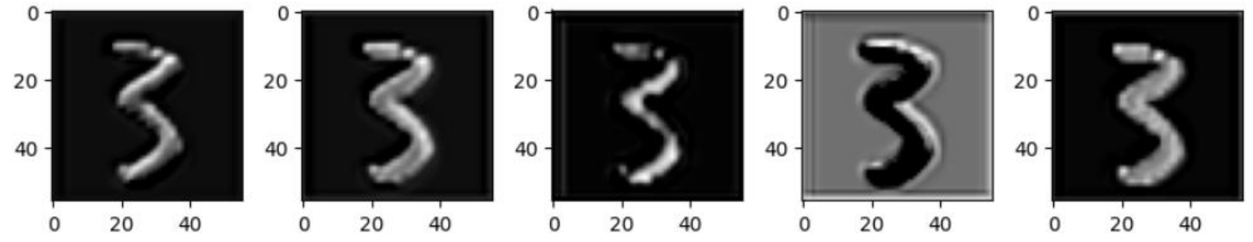


After added

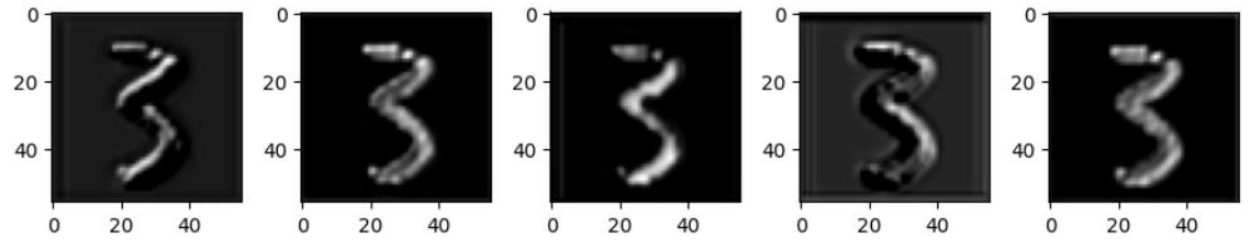


What we can see is after added using residual technique, the number is darker, more significant. It help prevent gradient vanishing (like we mentioned, and the image before adding is lighter, because some pixels has vanished)

Residual 1

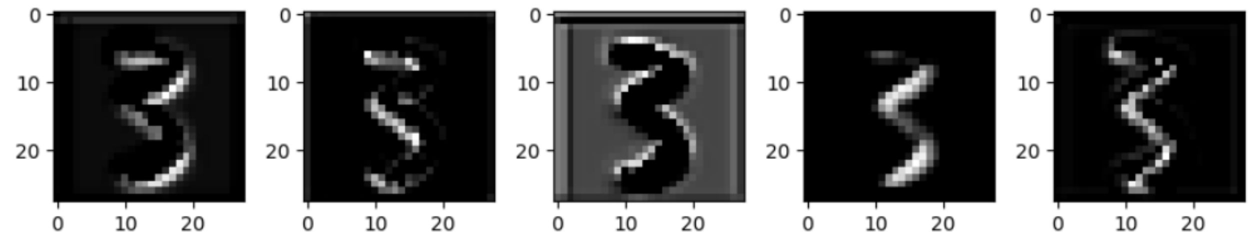


Residual 2

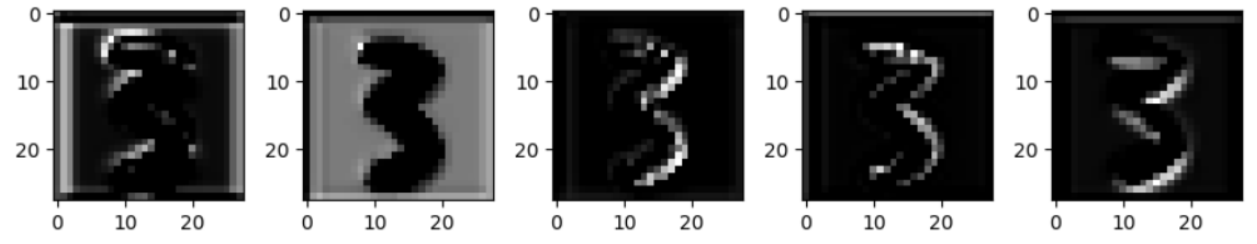


Layer 1

Residual 0

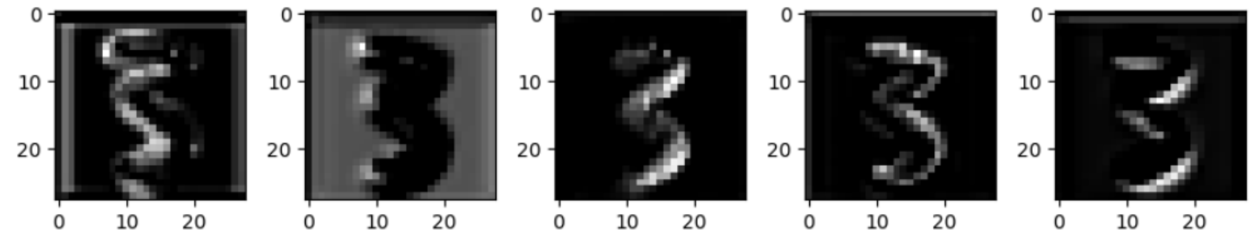


As I was mention at the theory section, since the second big layer, there will be some down sampling. This is the image at the first convolutional layer of the first residual block, which has the stride of 2 and number of filters now 128. Output of this residual block become 28x28x128.

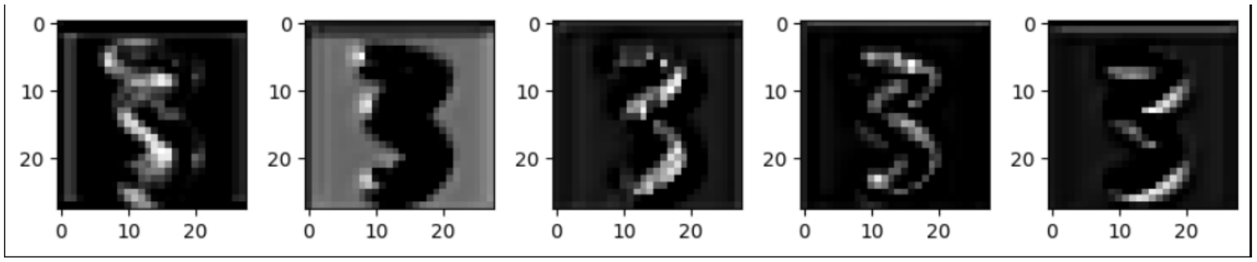


The image above is the output of the first residual block, you can see now it only highlights some important curves of the input we passed in.

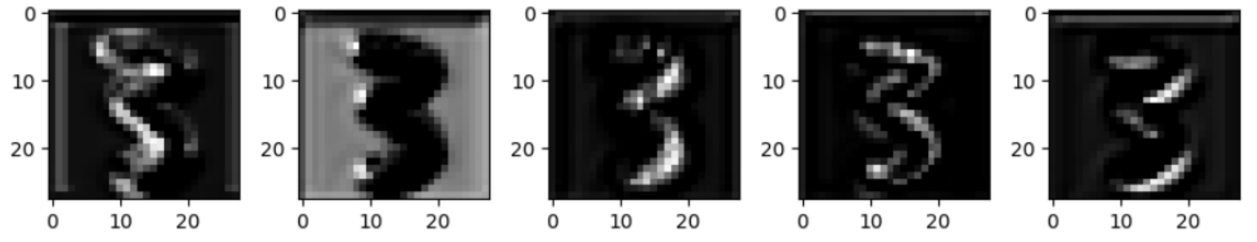
Residual 1



Residual 2

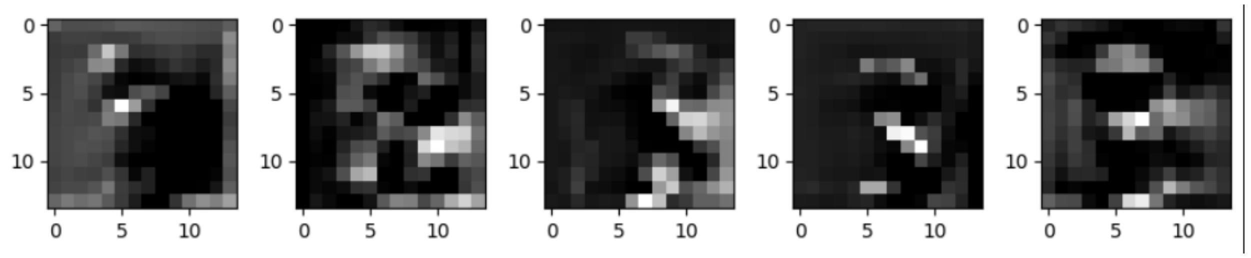


Residual 3



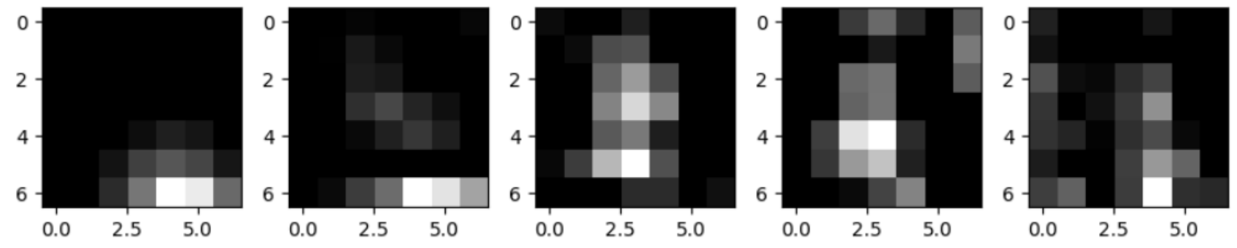
I will be going through faster from now on, I will only pass in as big layers level so we can see how the output turned out

Layer 2



The output became very hard to read since it has been downsampled again, now it’s 14x14x256. But some of it you can still recognize the curves that make the number 3. That what’s is important for our model to give predictions.

Layer 3

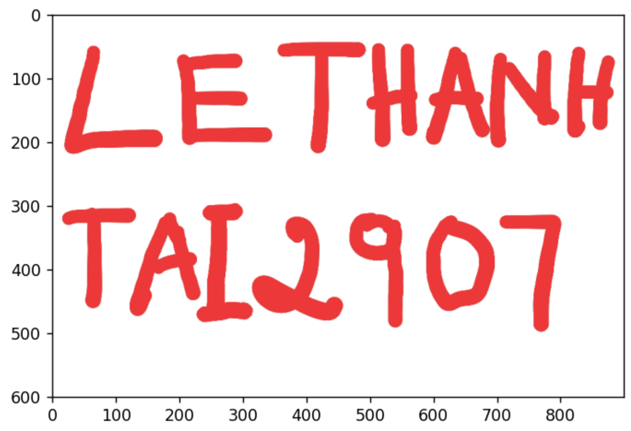


At these final big layers, it’s now 7x7x512 and can’t be understand by human any more, but take a closer look, those are curves that usually appears in the number 3. This is what the model needs for giving predictions, it findS special curves that make the subject unique and give predictions.

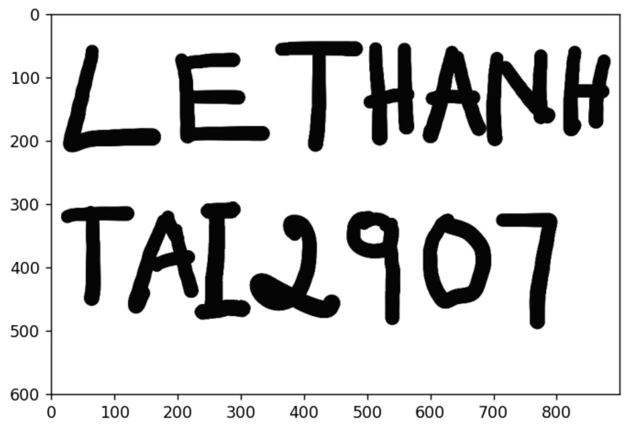
After all that, we just need to get the average pooling and flatten it then pass it to the fully connected layers and softmax to get the probability of the classes.

X. Implementing handwritten recognition OCR script with OpenCV and PyTorch

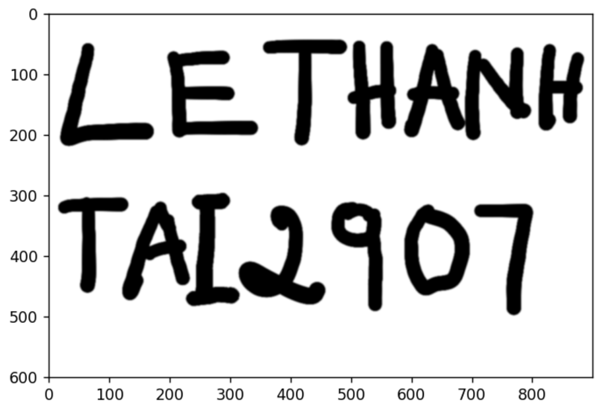
First, of course we will need to load the model. And today we will work on this image.



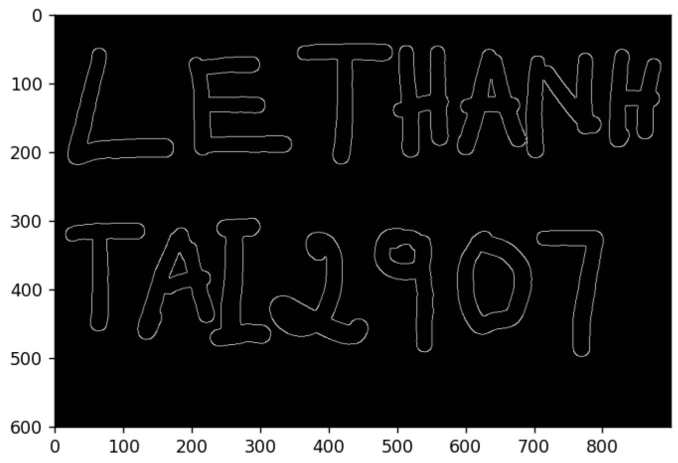
Since the model we learnt from the data of gray scale images. So, we need to make sure the input is also gray scale.



Apply GassianBlur to reduce noise. All little noise will be ignored and keep the important informations.



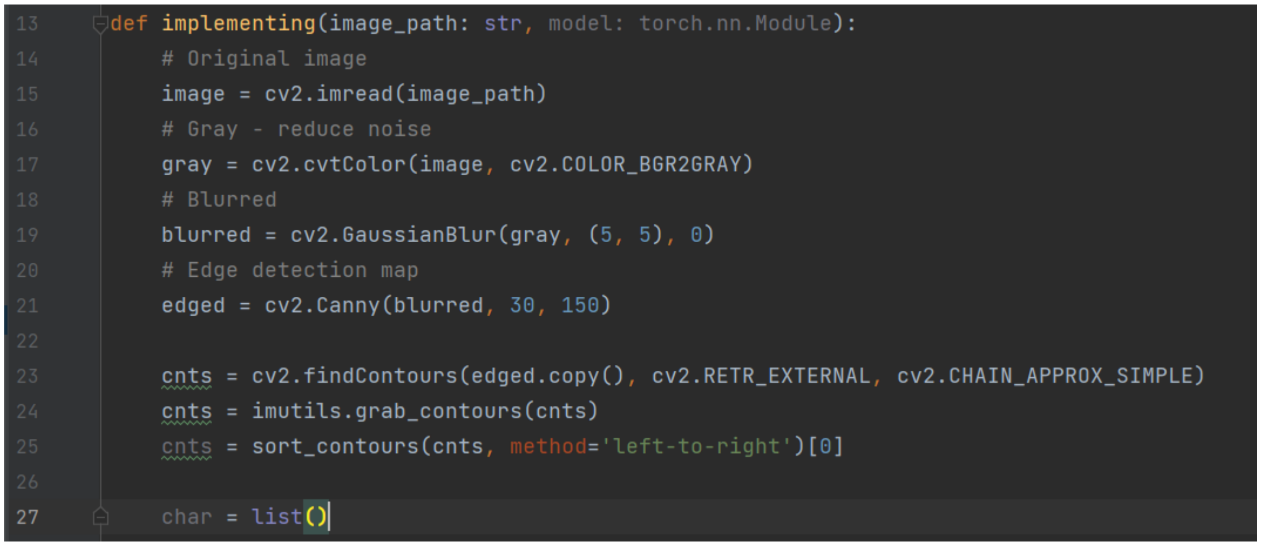
Then get the edges of the letters, this is for getting contours for creating bouding boxes.



And finally step in our preprocessing phrase, find the contours, the contours will look like this and OpenCV will use these for calculating the bouding boxes.

LETHANH
TAI2907

Those images are just for visualing and give you a better understanding on how the preprocessing step work, now let's get into our main function. This part of the project I prefer using a python file than using a notebook.

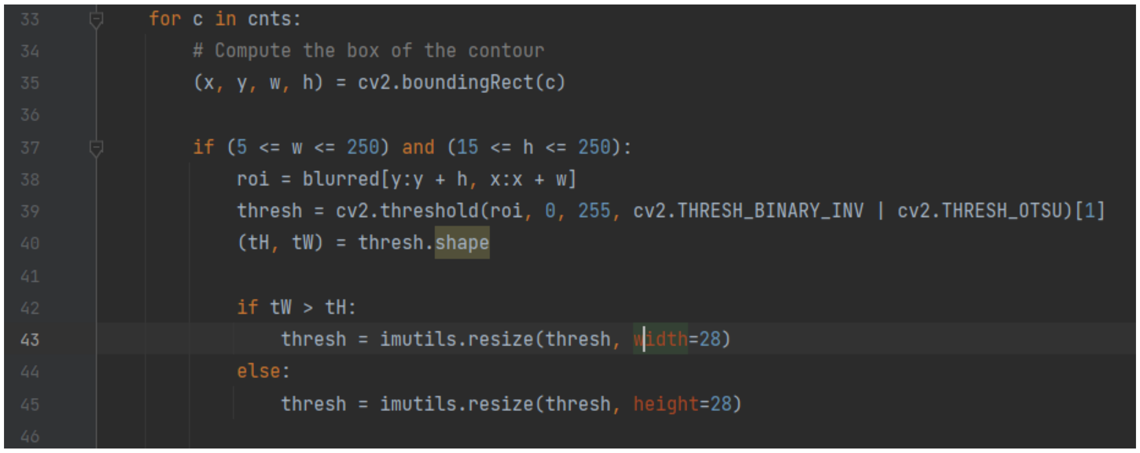


```python
13  def implementing(image_path: str, model: torch.nn.Module):
14      # Original image
15      image = cv2.imread(image_path)
16      # Gray - reduce noise
17      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
18      # Blurred
19      blurred = cv2.GaussianBlur(gray, (5, 5), 0)
20      # Edge detection map
21      edged = cv2.Canny(blurred, 30, 150)
22
23      cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
24      cnts = imutils.grab_contours(cnts)
25      cnts = sort_contours(cnts, method='left-to-right')[0]
26
27      char = list()
```

At first, we apply the preprocessing process to our image. And to locate the contours for each character, we apply contour detection. We find contours from the edged map and resulting contours from left to right (line 23-25). Learn how contour detection works here.

The chars list initialize will soon hold each and every single character image and associated bounding boxes.

Our next step will involve a large contour processing loop. Let's break that down in more detail, so that it is easier to get through.

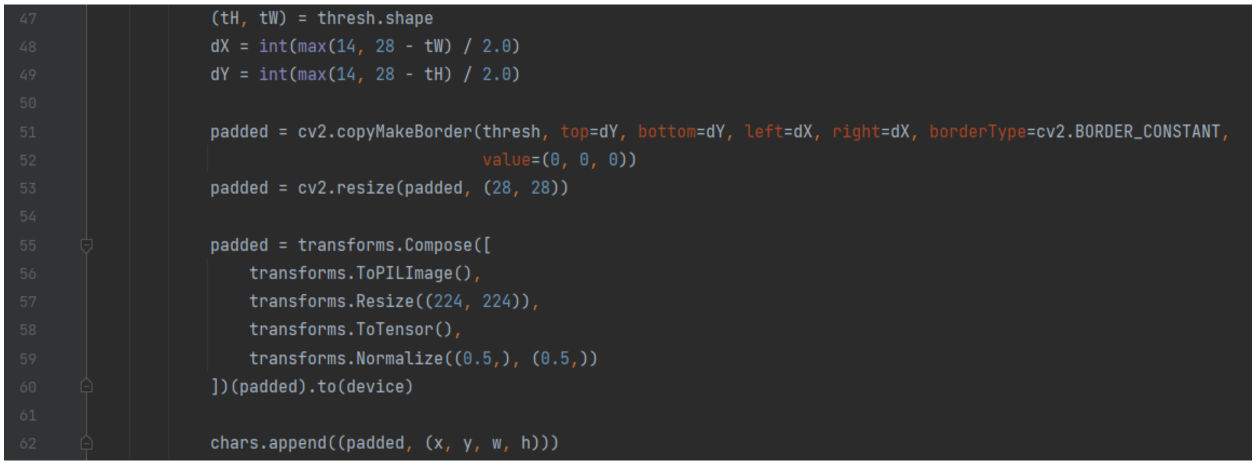


```
33  for c in cnts:
34      # Compute the box of the contour
35      (x, y, w, h) = cv2.boundingRect(c)
36
37      if (5 <= w <= 250) and (15 <= h <= 250):
38          roi = blurred[y:y + h, x:x + w]
39          thresh = cv2.threshold(roi, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
40          (tH, tW) = thresh.shape
41
42          if tW > tH:
43              thresh = imutils.resize(thresh, width=28)
44          else:
45              thresh = imutils.resize(thresh, height=28)
46
```

We loop over the contours, computing the bouding box of the contour using cv2.boundingRect(), we pass in the contour and it will return the bounding box's x, y, width and height.
Then we will filter the bounding boxes, ensuring they are neither too small nor to large (line 33).
We will only process the contours that have bouding box which has the width in the range of [5, 150] and height in the range of [15, 120].
And for each bouding box meeting our size criteria, we will extract the region of interest (ROI) associate with the characte, line 34.

Clean up the image using a threshold algorithm, in this case, we use Otsu's binary thresholding method to the ROI (line 35, 36). You can learn more about the Otsu's binary thresholding algorithm here.
That result in a binary image consisting of a with character on a black background.

Next, we size every character to a 28x28 pixel image with a border, depending on whether the width is greater than the height or the height is greater than the width, we resize the threshold character ROI accordingly (line 38-42).

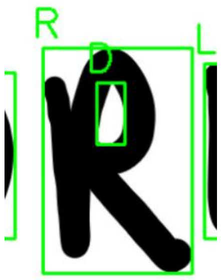


```
47              (tH, tW) = thresh.shape
48              dX = int(max(14, 28 - tW) / 2.0)
49              dY = int(max(14, 28 - tH) / 2.0)
50
51              padded = cv2.copyMakeBorder(thresh, top=dY, bottom=dY, left=dX, right=dX, borderType=cv2.BORDER_CONSTANT,
52                                          value=(0, 0, 0))
53              padded = cv2.resize(padded, (28, 28))
54
55              padded = transforms.Compose([
56                  transforms.ToPILImage(),
57                  transforms.Resize((224, 224)),
58                  transforms.ToTensor(),
59                  transforms.Normalize((0.5,), (0.5,))
60              ])(padded).to(device)
61
62              chars.append((padded, (x, y, w, h)))
```

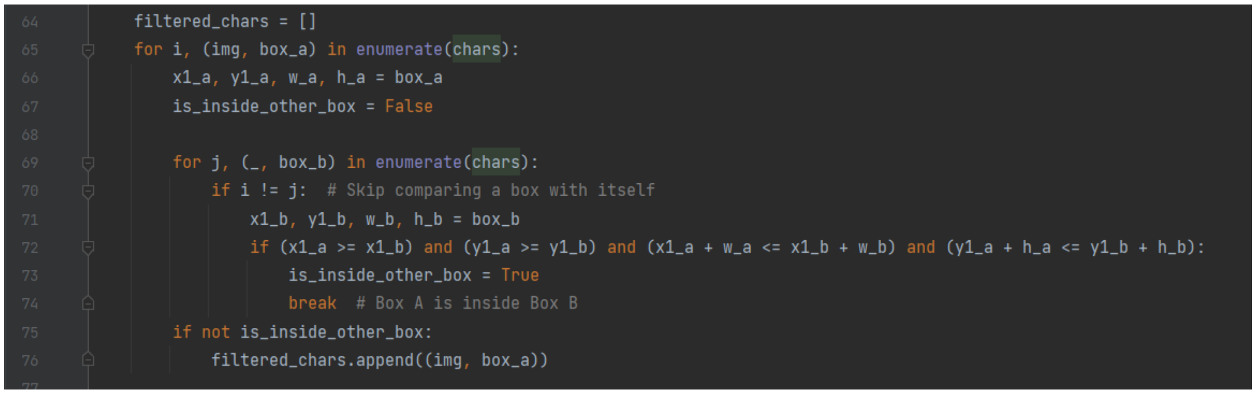
We continue to loop with padding actions. First, we re-grab the image dimensions (now that it has been resized (line 44-46).
Then we compute the necessary padding, line 48, 49. The minimum padding has to be around 14 as you can check again the input image we used to train, there always a lot of blank space

surround the letter, but the image we got from the bounding box is very tight and have none of those blank space, that will affect our result. After calulating, we then add padding to the image and force it to stay in the size of 28x28. The preprocess it before adding to a list for future model predictions. We use the same transform as we used in the training process. Then append it to chars list we created earlier.

This is a special case because some space inside a big letter could be given a bouding box, it will look some thing like this.

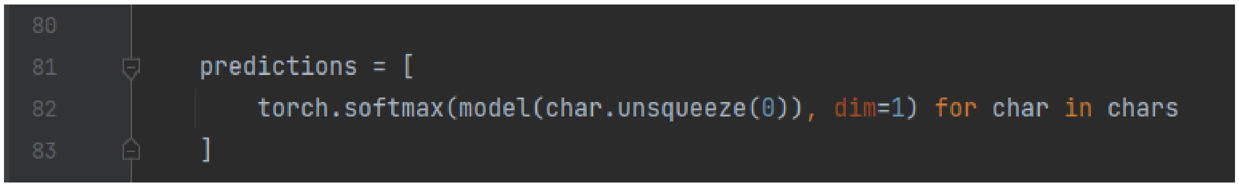


And we don't want that "D"'s box, so we need to eliminate the images and boxes that are inside other boxes.



```
    filtered_chars = []
    for i, (img, box_a) in enumerate(chars):
        x1_a, y1_a, w_a, h_a = box_a
        is_inside_other_box = False

        for j, (_, box_b) in enumerate(chars):
            if i != j:  # Skip comparing a box with itself
                x1_b, y1_b, w_b, h_b = box_b
                if (x1_a >= x1_b) and (y1_a >= y1_b) and (x1_a + w_a <= x1_b + w_b) and (y1_a + h_a <= y1_b + h_b):
                    is_inside_other_box = True
                    break  # Box A is inside Box B
        if not is_inside_other_box:
            filtered_chars.append((img, box_a))
```

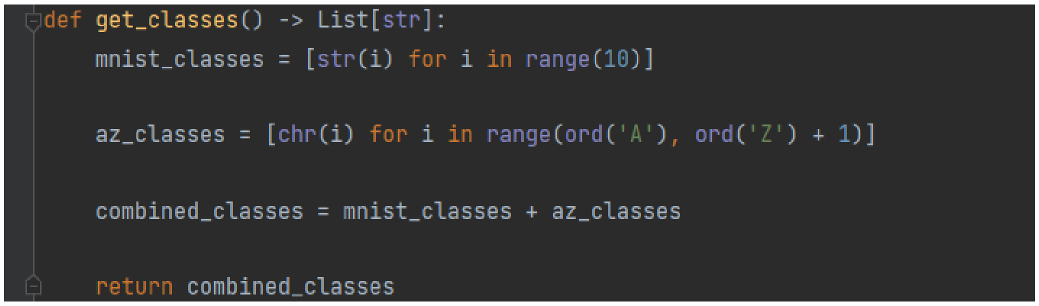
We create a new list for filtered chars, then we loop through all the chars, then compare its box to all other boxes, check and append if that is not inside another box.



```
    predictions = [
        torch.softmax(model(char.unsqueeze(0)), dim=1) for char in chars
    ]
```

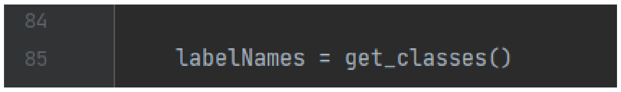
The create a list of predictions, here I use list comprehesion for faster speed, but if you're not familiar, you can also write a for loop to append all the predictions of the model to a list.

Before continue, you should code a get_classes() function to get label when you needed.

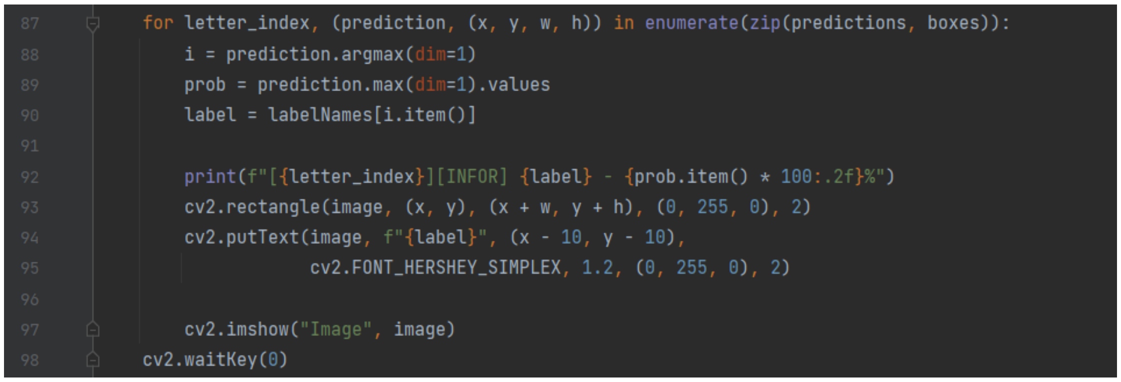


```python
def get_classes() -> List[str]:
    mnist_classes = [str(i) for i in range(10)]

    az_classes = [chr(i) for i in range(ord('A'), ord('Z') + 1)]

    combined_classes = mnist_classes + az_classes

    return combined_classes
```

Then continue our unfinished function with this, this will get the label names and save it in a list for ploting purpose.



```python

    labelNames = get_classes()
```

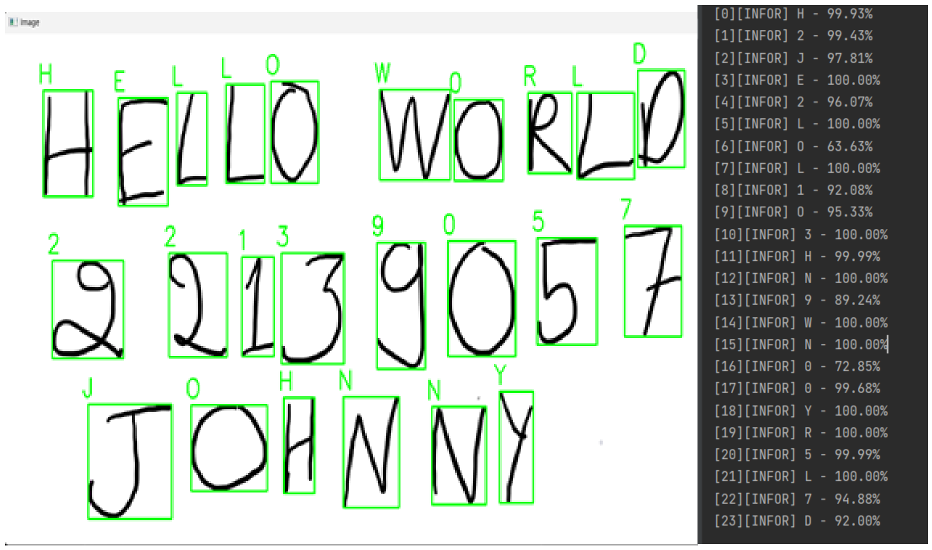


```python
    for letter_index, (prediction, (x, y, w, h)) in enumerate(zip(predictions, boxes)):
        i = prediction.argmax(dim=1)
        prob = prediction.max(dim=1).values
        label = labelNames[i.item()]

        print(f"[{letter_index}][INFOR] {label} - {prob.item() * 100:.2f}%")
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
        cv2.putText(image, f"{label}", (x - 10, y - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, 0), 2)

        cv2.imshow("Image", image)
    cv2.waitKey(0)
```

Now we loop through all the predictions and their boxs location.
The variable i here is the index of the highest probability return by the softmax activation function, that will also be the label. But the label we can get using argmax() method is just a number from 0 -35. But that is also the index of the labelNames list, so we use the item() method to get value of it and indexing the labelNames to get the corresponding label name.
The probability cat be get from the prediction using the max() method instead of argmax(), the max() method will return the max value in the tensor.
Now we can print out in the terminal the percentage of each letter from left to right to see the confidence of our predictions.
Then we use cv2.rectangle() to plot the bounding boxes. And use cv2.putText() to plot the predicted labels. Then use imshow to display everything

XI. Handwriting recognition OCR result

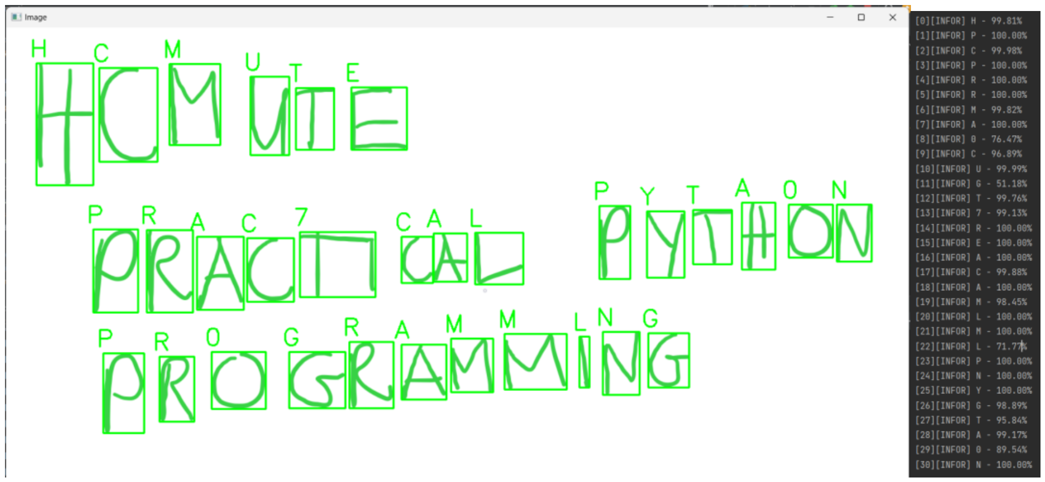
The output will lool something like this.



```
[0][INFOR] H - 99.93%
[1][INFOR] 2 - 99.43%
[2][INFOR] J - 97.81%
[3][INFOR] E - 100.00%
[4][INFOR] 2 - 96.07%
[5][INFOR] L - 100.00%
[6][INFOR] O - 63.63%
[7][INFOR] L - 100.00%
[8][INFOR] 1 - 92.08%
[9][INFOR] O - 95.33%
[10][INFOR] 3 - 100.00%
[11][INFOR] H - 99.99%
[12][INFOR] N - 100.00%
[13][INFOR] 9 - 89.24%
[14][INFOR] W - 100.00%
[15][INFOR] N - 100.00%
[16][INFOR] 0 - 72.85%
[17][INFOR] 0 - 99.68%
[18][INFOR] Y - 100.00%
[19][INFOR] R - 100.00%
[20][INFOR] 5 - 99.99%
[21][INFOR] L - 100.00%
[22][INFOR] 7 - 94.88%
[23][INFOR] D - 92.00%
```

In this example, we try to OCR the handwritten text "HELLO WORLD 22139057 JOHNNY". The model is perfoming quite well except one minor mistake which is it confused the letter "O" with the digit "0".

Here is another example. Now we try to recognize letters in the text "HCMUTE PRACTICAL PYTHON PROGRAMMING"



```
[0][INFOR] H - 99.81%
[1][INFOR] P - 100.00%
[2][INFOR] C - 99.98%
[3][INFOR] P - 100.00%
[4][INFOR] R - 100.00%
[5][INFOR] R - 100.00%
[6][INFOR] M - 99.82%
[7][INFOR] A - 100.00%
[8][INFOR] 0 - 76.47%
[9][INFOR] C - 96.89%
[10][INFOR] U - 99.99%
[11][INFOR] G - 51.18%
[12][INFOR] T - 99.76%
[13][INFOR] 7 - 99.13%
[14][INFOR] R - 100.00%
[15][INFOR] E - 100.00%
[16][INFOR] A - 100.00%
[17][INFOR] C - 99.88%
[18][INFOR] A - 100.00%
[19][INFOR] M - 98.45%
[20][INFOR] L - 100.00%
[21][INFOR] M - 100.00%
[22][INFOR] L - 71.77%
[23][INFOR] P - 100.00%
[24][INFOR] N - 100.00%
[25][INFOR] Y - 100.00%
[26][INFOR] G - 98.89%
[27][INFOR] T - 95.84%
[28][INFOR] A - 99.17%
[29][INFOR] 0 - 89.54%
[30][INFOR] N - 100.00%
```

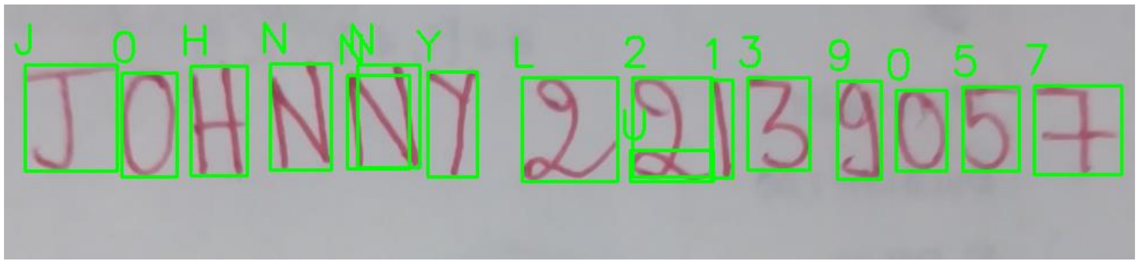
We can see the model made a few minor mistakes here:

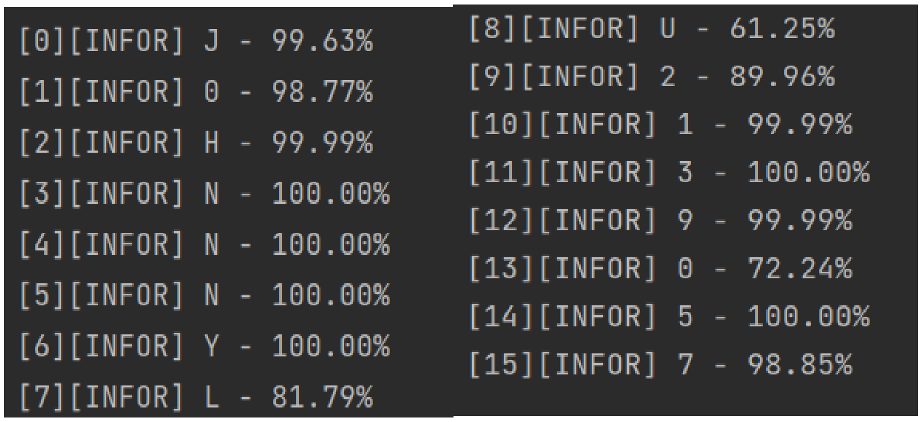
- "I" and "L"
- "H" and "A"
- "O" and "0"

Those are very understandable so it's not worth concerning. Because find the ultimate correctness in machine learning is impossible. So, if there are some reasonable mistakes, that could be pass.

But there is a crucial mistake that can't be ignored. The "T" and "I" letters have got combined and recognized as one single character "7". Why is that? That will be explained in the next section.

The two previous examples are just using paint 3D software on the computer to draw, so those are quite close to the dataset use to train the model, now let's try some real life handwriting recognition.





```
[0][INFOR] J - 99.63%
[1][INFOR] 0 - 98.77%
[2][INFOR] H - 99.99%
[3][INFOR] N - 100.00%
[4][INFOR] N - 100.00%
[5][INFOR] N - 100.00%
[6][INFOR] Y - 100.00%
[7][INFOR] L - 81.79%
[8][INFOR] U - 61.25%
[9][INFOR] 2 - 89.96%
[10][INFOR] 1 - 99.99%
[11][INFOR] 3 - 100.00%
[12][INFOR] 9 - 99.99%
[13][INFOR] 0 - 72.24%
[14][INFOR] 5 - 100.00%
[15][INFOR] 7 - 98.85%
```

I was trying to use it to OCR the "JOHNNY 22139057" but the result turned out to be a little bit troublesome. I mistaken the letter "O" with "0" and the digit "2" with "L" again.

But the new error happened is that there are 2 boxes for "N" and a smaller box part of the digit "2" and got recognized as "U". That's because we only remove boxes that completely inside another box but haven't got a solution for removing boxes which are overlapped mostly.

And the reason it got those boxes in the beginning is that the ink was not as smooth as the image we drawed on paint 3D, some of it got blank space into it and make the function of OpenCV got confused.

XII. Limitation, drawbacks and next steps

1. *Limitation, drawbacks*

Although those are very reasonable mistake but our model reached 99% on our testing dataset, shouldn't it be working well on our real dataset?

That is because of the datasets, MNIST and A-Z Handwritten datasets are cleaned and preprocessed for us before. But the realworld characters can never be that "clean". As you can see in the examples, some of its line might twist make it look like another letter and make the model confused.

Additionally, our handwriting recognition methods requires characters to be individually segmented. That may be possible for some characters, but many of us (especially cursive writers) connect characters when writing quickly. This confused our model into thinking a group of characters is actually a single character, which ultimately leads to the incorrect results.

Finally, our model is too simplistic. While our model performed well on the training and testing set, the architecture – combined with training dataset it selft is not robust enough to generalize as an “off-the-shelf” handwriting recognition model.

2. *Next steps*

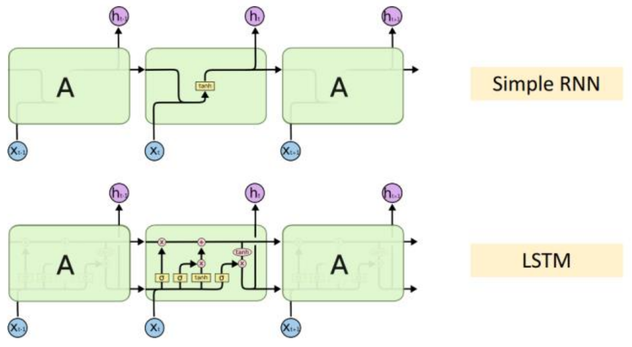
For now, our model can just recognize the uppercase characters and digits, so int the future, we will increase the recognition ability by adding more training data by searching and adding lowercase letters to training datasets.

Apply de-skewing to our model, so it can perform OCR on images that were taken using the camera from a non-direct angle.

Split the data in lines, this is the premise for identify the text in “word” level. Remove the characters that are partly overlapped.

Apply training from model real-world data, use this more widely and take the data in the implementing process back for training more. Make the model more powerful. And combined with the techniques using the newly trained model periodicly retrained the model with the original data so that it won’t forget the past data.

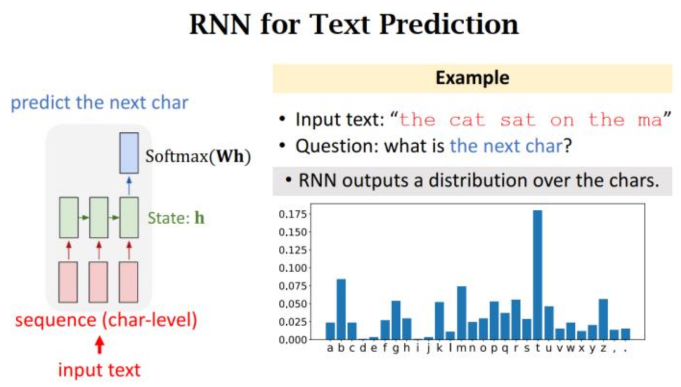
Finally, the most powerful method to improve this, using Long-Short Term Memory. That is an improvement from the original Recurrent Neural Network, it just basically a RNN but has stronger ability to memorize the words based on the importance of the words.



Figures are from Christopher Olah's blog: Understanding LSTM Networks.

So how can we apply this?

We apply one of the applications of the RNN models.



RNN for Text Prediction

Example

- Input text: "the cat sat on the ma"
- Question: what is the next char?
- RNN outputs a distribution over the chars.

RNN model can help predict the possibilty of the next possible character which will appears in the sentence and when will words end.
Although simple RNN could do this quite well aready, but we prefer LSTM since it's more efficient and more accurate.

And the result of combining this with the current OCR model could result in a total completion of such an OCR application.