Basic Explanation for Handwritten Digit to Text Using the CNN Model

- The first thing that we do is we load and train the mnist dataset.

- The mnist dataset is a large dataset that is composed of handwritten alphanumeric characters that are commonly used to train image processing softwares.

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images.shape
train_labels.shape
test_images.shape
test_labels.shape
```

- We now have created a training set and a test set.

- Training set:

    - The training set has 60,000 images in which each image is a 28x28 pixel grayscale image of a character.

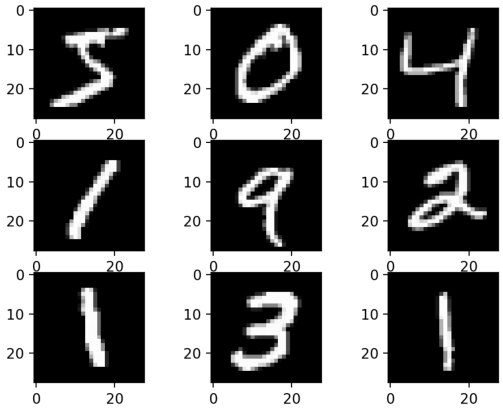
    - The training set also has 60,000 labels that define the digit that a respective image represents.

- Test set:

    - The test set has 10,000 images in which each image is a 28x28 pixel grayscale image of a character.

    - The test set also has 10,000 labels that define the digit that a respective image represents.

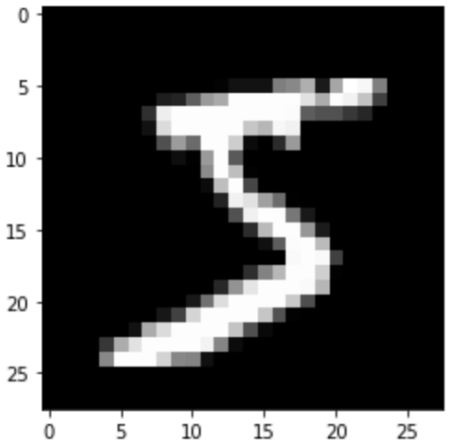
- A visualization of the first nine digits of the mnist dataset is shown below:



- Outputting just the first digit of the mnist dataset (or the 0th training sample) we can see that the digit resembles a handwritten 5.

```
import matplotlib.pyplot as plt
plt.gray()
plt.imshow(train_images[0])
```

- Output:



- We can see that the 0th label corresponds to the 0th digit because it outputs a value of 5.

```
train_labels[0]
```

MLP Model

- One way to create our handwriting to text converter is using the multilayer perceptron (MLP) model. MLP is a class of artificial neural network. They consist of at least three layers of nodes: the input layer, a hidden layer, and an output layer.cnn

- The input shape that we input is a column vector that corresponds to one column of our 28x28 image for a digit.

- The first hidden layer has 512 neurons and has a rectified linear unit (ReLU) activation function. The ReLU activation function is a piecewise linear function that will output the input directly if it is positive and 0 otherwise.

- Our output layer has 10 neurons that correspond to the 0 to 9 digits. It uses the softmax activation function. The softmax activation function essentially just converts a vector of numbers into a vector of probabilities. In our case, we use the softmax function to output the probability that a given digit is predicted as each of the 10 digits in the mnist dataset.

```
from keras import models
from keras import layers
model_mlp = models.Sequential()
model_mlp.add(layers.Dense(512, activation='relu', input_shape=(28*28,)))
model_mlp.add(layers.Dense(10, activation='softmax'))
model_mlp.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

- We then define the optimizer function, loss function, and the metrics that are used for the model.

```
model_mlp.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

- The optimizer is a function that modifies the attributes of the neural network such as weights and learning weight. The purpose of this function is to reduce the overall loss and improve the accuracy of the neural network.
    - The optimizer that we use implements the RMSprop algorithm. This algorithm maintains a moving average of the square of gradients and divides the gradient by the root of this average.
- The loss function shows a measure of model performance. They are used to train a machine learning model.
    - The loss function that we use is the categorical crossentropy algorithm. This algorithm is well suited for classification tasks. The loss that this function calculates is a good measure to distinguish two discrete probability distributions from each other.
- Metrics are used to monitor and measure the performance of a model during training and testing. They tell you if you are making progress and give it a value.
    - We use the classification accuracy metric. It is defined as the number of correct predictions divided by the total number of predictions multiplied by 100.
- Before we feed data into our model, we need to process the data. Processing the data consists of:
    - Changing the images to column vector form: 28x28 → 784x1 to match the model's input expectations.
    - Changing the vector values from int to float.
    - Scaling the vector values to be in the [0, 1] interval. This way our model will see all samples with equal weightage as the range for all samples are the same (gray scale values are from 0 to 255 so dividing the float by 255 will give us the values in our desired interval).

- The code for processing train_images is shown below:

```
# original image
train_images.shape
# output: (60000, 28, 28)


# after reshaping
train_images_mlp = train_images.reshape(60000, 28*28)
train_images_mlp.shape
# output: (60000, 784)

# type is currently an int
# chosing a random non-zero value element in our column vector
type(train_images_mlp[0][350])
# output: numpy.uint8
train_images_mlp[0][350]
# output: 70

# converting type to float
train_images_mlp = train_images_mlp.astype('float32') / 255
# type is now a float
type(train_images_mlp[0][350])
# output: numpy.float32
train_images_mlp[0][350]
# output: 0.27450982
```

- Implementing the above code for test_images is similar:

```
test_images_mlp = test_images.reshape(10000,28*28)
test_images_mlp = test_images_mlp.astype('float32') / 255
```

- We also have to process the data labels. Processing the data labels consists of:

    - Representing the label as a 10 bit value where the bit corresponding to the digit value will be 1 and the rest of the 9 bits will be 0. For example, if the label = 1, we will change it to a 10 bit value as 0100000000.

- The code for processing train_labels is shown below:

```
# label before processing
train_labels[0]
# output: 5

# processing labels
from keras.utils import to_categorical
train_labels_mlp = to_categorical(train_labels)
tain_labels_mlp[0]
# output: array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

- Implementing the above code for test_labels is similar:

```
test_labels_mlp = to_categorical(test_labels)
```

- After processing, we are able to train the model. We will use a batch size of 120 where 1 epoch is 60000/120 = 500 batches. An epoch is 1 complete run of all train samples for training the model. We will have a total of five epochs.

```
model_mlp.fit(train_images_mlp, train_labels_mlp, epochs = 5, batch_size = 120)
# output:
#   Epoch 1/5
#   60000/60000 [==============================] - 5s 83us/step - loss: 0.2532 - accuracy:
#   Epoch 2/5
#   60000/60000 [==============================] - 5s 75us/step - loss: 0.1022 - accuracy:
#   Epoch 3/5
#   60000/60000 [==============================] - 5s 76us/step - loss: 0.0677 - accuracy:
#   Epoch 4/5
#   60000/60000 [==============================] - 5s 75us/step - loss: 0.0495 - accuracy:
#   Epoch 5/5
#   60000/60000 [==============================] - 5s 78us/step - loss: 0.0371 - accuracy:
#   <keras.callbacks.callbacks.History at 0x13528c2c198>
```

- We can now test our model's performance with the test data:

```
test_loss_mlp, test_acc_mlp = model_mlp.evaluate(test_images_mlp, test_labels_mlp)
print('test accuracy:', (test_acc_mlp*100))
# output: test accuracy: ~98.0%
```

CNN Model

- Our CNN model is a little different from the MLP model. CNN stands for convolutional neural network. It is a class of artificial neural network that is most commonly applied to analyze visual imagery. CNNs are regularized versions of MLPs. MLPs consist of fully connected networks, meaning that each neuron in one layer is connected to all neurons in the next layer. Due to the "full connectivity" of these networks, they are prone to overfitting data. CNNs take advantage of the hierarchical pattern in data and assemble patterns of increasing complexity using smaller and simpler patterns.

```
from keras import models
from keras import layers
model_cnn = models.Sequential()
```

- Our first layer is a 2 dimensional convolution layer with 32 filters/kernels. The kernel size is 3x3. Similar to the MLP model, we use the ReLU activation function. Our input shape is a 28x28 matrix with one channel.

```
model_cnn.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
```

- Our second layer is a downsample of the output of the previous layer. It takes the max value for every 2x2 window.

```
model_cnn.add(layers.MaxPooling2D(2,2))
```

- Our next layer is a 2 dimensional convolution layer with 64 kernels. Everything else is the same as our first layer.

```
model_cnn.add(layers.Conv2D(64, (3,3), activation='relu'))
```

- Our next layer is another downsample of the output from the previous layer. It also takes the max value for every 2x2 window.

```
model_cnn.add(layers.MaxPooling2D(2,2))
```

- Our next layer is another 2 dimensional convolution layer with 64 kernels of size 3x3 and with the same ReLU activation function.

```
model_cnn.add(layers.Conv2D(64, (3,3), activation='relu'))
```

- At this point our data is in matrix form so we will convert it into vector form in order to feed a fully connected network.

```
model_cnn.add(layers.Flatten())
```

- We will design for 64 outputs with the ReLU activation function

```
model_cnn.add(layers.Dense(64, activation = 'relu'))
```

- The final layer has 10 outputs corresponding to the 10 digits (0 to 9). The activation function that we use here is softmax (as explained in the MLP model).

```
model_cnn.add(layers.Dense(10, activation = 'softmax'))
```

- This is what our layers look like:

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

- Similar to the MLP model, we also need to process our data before we feed it into our model.

```
# image before processing
train_images.shape
# output: (60000, 28, 28)

# CNN needs another dimentsion for this channel. Because our image is greyscale, it only needs 1 channel.
train_images_cnn = train_images.reshape(60000, 28, 28, 1)
train_images_cnn.shape
# output: (60000, 28, 28, 1)

# We need to change our element values from integer to decimal.
# Similar to the MLP model, we limit values between the interval [0, 1] so that the model treats each sample with equal weightage.
train_images_cnn = train_images_cnn.astype('float32') / 255
test_images_cnn = test_images.reshape(10000, 28, 28, 1)
test_images_cnn = test_images_cnn.astype('float32') / 255
```

- We also need to process our labels. Like in the MLP model, we convert the labels to 10 bit values.

```
from keras.utils import to_categorical
train_labels_cnn = to_categorical(train_labels)
test_labels_cnn = to_categorical(test_labels)
```

- We define the optimizer function, loss function and metrics the same way as the MLP model.

```
model_cnn.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

- We are now able to train the model using train_images and train_labels. We use a batch size of 60 where 1 epoch = 60000/60 = 10000 batches. We will have a total of 5 epochs.

```
model_cnn.fit(train_images_cnn, train_labels_cnn, epochs = 5, batch_size = 60)
# output:
#   Epoch 1/5
#   60000/60000 [==============================] - 38s 636us/step - loss: 0.1731 - accuracy: 0.9459
#   Epoch 2/5
#   60000/60000 [==============================] - 38s 639us/step - loss: 0.0477 - accuracy: 0.9853
#   Epoch 3/5
#   60000/60000 [==============================] - 39s 647us/step - loss: 0.0327 - accuracy: 0.9898
#   Epoch 4/5
#   60000/60000 [==============================] - 40s 665us/step - loss: 0.0243 - accuracy: 0.9925
#   Epoch 5/5
#   60000/60000 [==============================] - 39s 645us/step - loss: 0.0198 - accuracy: 0.9941
#   <keras.callbacks.callbacks.History at 0x1352a775c18>
```

- We can now test the model's performance with the test data

```
test_loss_cnn, test_acc_cnn = model_cnn.evaluate(test_images_cnn, test_labels_cnn)
print('test accuracy:', (test_acc_cnn*100))
# output: test accuracy: ~99.3%
```