Senior Project

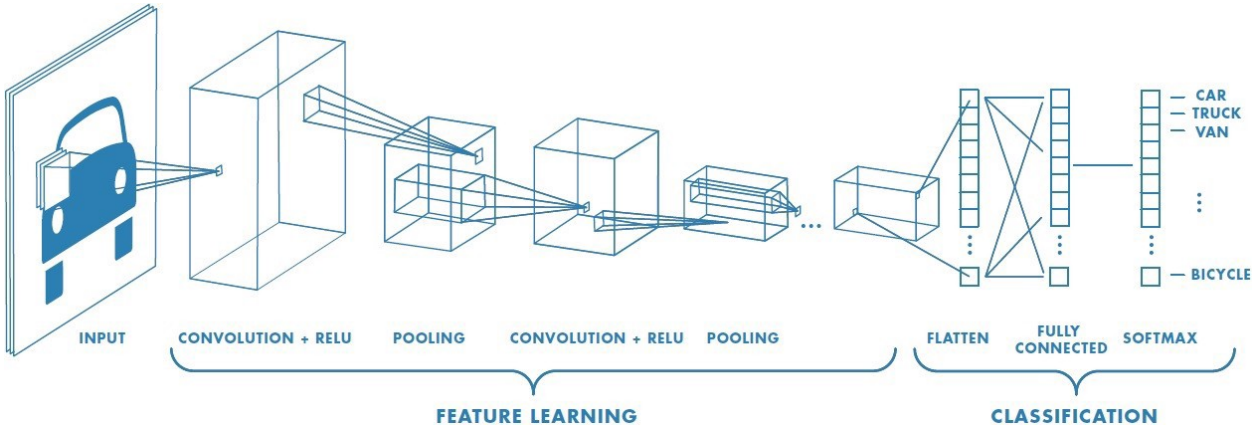
Using Deep Learning to Convert Handwritten Text to Digital

By: Jonathan Bradley

May 5th, 2021

Advisor: Dr. Benjamin Soibam

Advisor Signature:



Abstract

Artificial Intelligence, or AI, is a broad branch of computer science involving the development and implementation of systems that contain human-like characteristics such as the ability to reason and learn from past experiences. AI has been widely implemented in many different applications that are in everyday human life and the goal of this research project is to come to realization and understanding of the technologies and algorithms that make those applications.

This project will use python, anaconda, and keras to utilize the available libraries to create a convoluted neural network that will analyze images, learn their features, and predict the class labels corresponding to what the image contains. The EMNIST dataset will be used during this project as it's a commonly used dataset with a large amount of resources to help understand how these neural networks work. This report will document, explain, and present the results of the research done for this project and the results of the training and testing of the convoluted neural network of the EMNIST dataset.

Acknowledgement

I would first like to thank my advisor, Dr. Benjamin Soibam, who allowed me to do research under his wing. He lent me recommendations for books, shared access to many different lectures, and gave plenty of feedback. During this inconvenient time, all of the things given from him are extremely appreciated and I'm forever grateful for this opportunity.

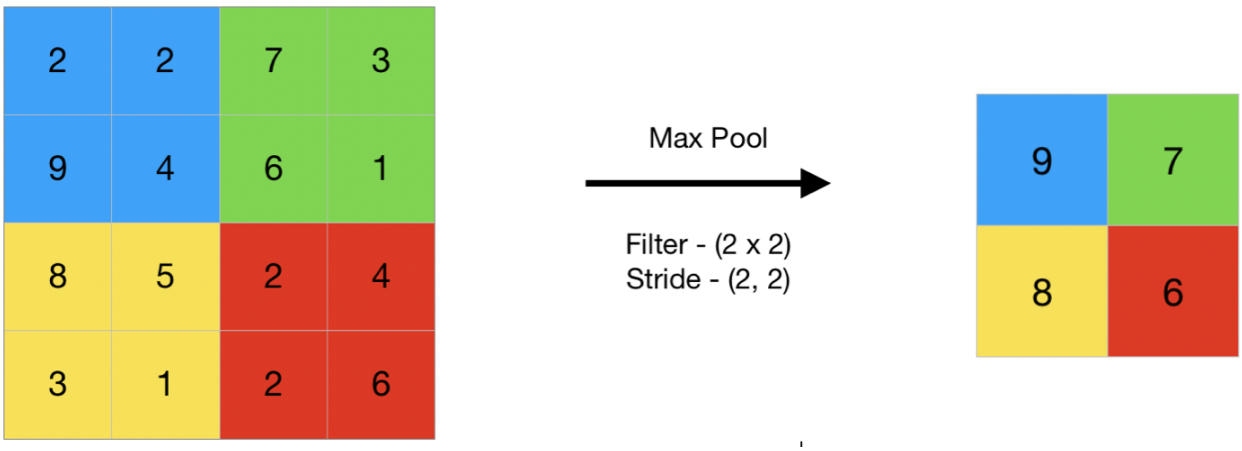
I would also like to thank my parents, Johnny and Fay Bradley, who have stuck with me and kept pushing me throughout all of these years. Without them and their unconditional love and support, there's no way I endure through all my college hardships. Thanks mom and dad.

Table of Contents

Introduction

Convolutional neural networks , or CNN, are composed of multiple layers of connected artificial neurons. These neurons are mathematical functions that calculate the sums of multiple inputs and outputs a value. These networks are best used for image classification, a process that occurs in many different applications such as facial recognition, document analyzing, and object detection.

The CNN takes input of an image, or images, which is stored as a 3D matrix - height by width by depth. Depth indicates the RGB channels, where 3 would indicate all the 3 channels and 1 would indicate a grayscale image. The height and width are an indication of the size (amount of pixels) of the image. The CNN then uses a “kernel” that will detect features of an image, such as an edge, by striding over it and creating weights via pooling . These weights form together to create a feature map that will go through an activation process and possibly other layers that in the end, will be able to classify an image to a given label.



In the image above, provided by geeksforgeeks, we can see the results of what max pooling does as it strides over the original input by 2. This is done to down sample the features of an image so that the neural network is more efficient.

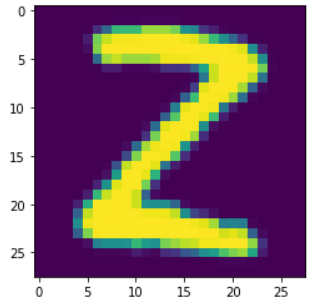
The fully connected or dense layer is then applied and connects all neurons from the previous layer to each neuron to the current layer. The image is finally in a form that is usable to network. The model can now train, test, and be able to distinguish between high and low-level features in the image and classify them using softmax classification.

The Dataset

The dataset used for this project was provided by Kaggle. Similar to the well known and widely used MNIST dataset, the EMNIST (Extended - MNIST) datasets are 6 datasets 'containing handwritten character digits derived from the NIST Special Database 19.' Each image is structured as a 28 by 28 pixel format where, in the CSV file, each row is a separate image, has 785 columns, and the first column is the class_label. The other 784 columns contain a one pixel value. In the graphic below, pyplot is used to print out the shape of the first image in the training portion of the dataset. The shape of the image "Z" can be seen as a 28 by 28 unit square.

```
1 from matplotlib import pyplot as plt
2 plt.imshow(X_train[0])
3 print('Label: ', y_train[0])
```

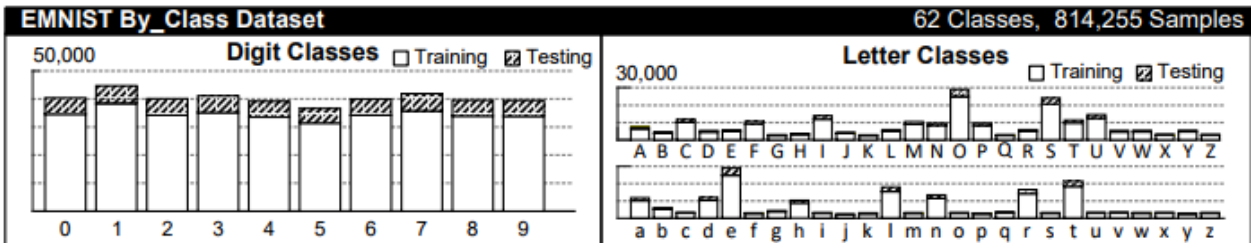
```
Label:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```



The 6 datasets are as listed - ByClass, ByMerge, Balanced, Letters, Digits, and MNIST. The structure of each set is referred to on Kaggle's website but I chose to work with the "ByClass" dataset. This set specifically contains 62 classes - (0-9, 'A-Z', and 'a-z'), 697,932 images to be used for training and 116,323 images to be used for testing, which equates to a total of 814,255

images. The number of digits are greater than the number of letters and the number of letters are based on the frequency of use in the English language.

The numbers can be seen in the graphic shown below. This is the largest dataset in the EMNIST datasets and by far the largest dataset I have personally worked with.



Methodology

Gathering the Dataset

The platform used in this research project is Anaconda as it contains many popular python packages used for data science and machine learning. The Keras library will be used as well, as it provides a well rounded, easy to use neural network interface. The EMNIST datasets are first imported from Keras and already contains functions to list the available datasets that are a part of the EMNIST database and functions that extract the testing and training samples separately.

The use of the sklearn “train_test_split” is not needed, which is a function I have used many times previously. The function is normally used to split a dataset into training and testing sets with an 80/20 split towards the former. In the case of the EMNIST database, the training set and testing set are already defined sets, no need for the “train_test_split”, and there are functions available for extracting those sets into variables.

In the image below, in the first cell, the 6 classes are able to view using the list_datasets() function. In the second cell, the functions extract_test_samples and extract_training_samples are used to assign the data to X_train, y_train, X_test, and y_test respectively as numpy arrays from the ‘byclass’ dataset. We can see the shapes of each variable by using the “.shape” function and as previously stated, the training numbers are 697932 images of a 28 by 28 size, the testing numbers are 116,323 images of a 28 by 28 size and the respective labels match the amount. The max number in the y_train variable is listed as 61 and with the index starting as 0 implies that

there 62 classes which match what was stated as well. The data has been correctly imported in and now needs to be interpreted and shaped.

```
In [1]:
from emnist import list_datasets
import numpy as np
list_datasets()
```

```
Out[1]: ['balanced', 'byclass', 'bymerge', 'digits', 'letters', 'mnist']
```

```
In [2]:
from emnist import extract_test_samples
from emnist import extract_training_samples

X_train, y_train = extract_training_samples('byclass')
X_test, y_test = extract_test_samples('byclass')
max = np.max(y_train)

print('Training Shape: ', X_train.shape)
print('Testing Shape: ', X_test.shape)
print('Training Label Shape: ', y_train.shape)
print('Testing Label Shape: ', y_test.shape)
print('Number of classes: ', max + 1)
```

```
Training Shape:  (697932, 28, 28)
Testing Shape:  (116323, 28, 28)
Training Label Shape:  (697932,)
Testing Label Shape:  (116323,)
Number of classes:  62
```

Shaping the Dataset

When checking the shape of the images, we can see that the training shape was (687832, 28, 28). The lack of a fourth entry in the dimension implies that images are grayscale images. A fourth entry such as (687832, 28, 28, 3) means that the last entry has 3 channels for RGB. The model being built requires the shape to be a 4D tensor. The .reshape() function allows us to do that. Knowing the images are grayscale, we only need to reshape the dimensions of the training and testing to contain one channel in the fourth dimension entry to change. The resulting changes X_train and X_test to (687832, 28, 28, 1) and (116323, 28, 28, 1) respectively.

```
X_train = X_train.reshape(697932, 28, 28, 1)
X_test = X_test.reshape(116323, 28, 28, 1)

print('Data Set Info: \n')
print('Training Shape: ', X_train.shape)
print('Testing Shape: ', X_test.shape)
print('Training Label Shape: ', y_train.shape)
print('Testing Label Shape: ', y_test.shape)

max = np.max(y_train)
print('Number of classes: ', max + 1)
```

```
Data Set Info:

Training Shape:  (697932, 28, 28, 1)
Testing Shape:  (116323, 28, 28, 1)
Training Label Shape:  (697932,)
Testing Label Shape:  (116323,)
Number of classes:  62
```

The pixel values of images are normally unsigned integers that range from 0 to 255. These need to be normalized to be between 0 and 1 as this helps the model's performance during training. This can be done by dividing the values by 255, then testing the values to see if there's a maximum value of 1 and a minimum value of 0.

```
X_train = X_train/255
X_test= X_test/255
print(np.max(X_train))
print(np.min(X_train))
print(np.max(X_test))
print(np.min(X_test))

```

```
1.0
0.0
1.0
0.0
```

Categorizing the Labels

To categorize the labels, keras has a function (.to_categorical()) that will compute a one-hot encoded matrix so that each row is the one-hot ended label of the training sample. For example, in the EMNIST 'byclass' dataset, the number '9' would equate to [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] after categorizing. We categorize both y_train and y_test.

```
print('Shape of y_train and y_test after categorizing: ')
y_train = np_utils.to_categorical(y_train, NB_CLASSES)
y_test = np_utils.to_categorical(y_test, NB_CLASSES)

print(y_train.shape)
print(y_test.shape)
```

```
Shape of y_train and y_test after categorizing:
(697932, 62)
(116323, 62)
```

Building the Convolutional Neural Network Model

As previously stated before in the "Machine Learning Vs. Deep Learning" section, building a CNN model involves multiple parts. The first step is to import the model, Sequential was used here, and the different layers of the model - Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D.

- The "Dense" layer is a regular layer of neurons in the neural network where each input neuron is connected to each output neuron, hence "dense-ly" connected.
- The "Dropout" layer is a technique to help prevent overfitting. This is done by randomly selecting neurons and ignoring them during training.
- The "Activation" layer is a layer where an activation function can be used. A few of the available functions are the 'relu' function which applies the rectified linear unit activation function, 'sigmoid' function which applies the sigmoid activation function, and the 'softmax' function which converts a real vector to a vector of categorical probabilities.
- The "Flatten" layer is used for flattening a multi-dimensional matrix into one. A layer useful so that shaping and mapping data into neurons is much easier (and automated).

- The “Conv2d” is a 2D convolution layer which creates a kernel that is convolved, or combined, with the layer input to produce a tensor of outputs.
- The “MaxPooling2D” layer that down samples the input by taking the value over the window defined size. This was also stated previously in “Machine Learning Vs. Deep Learning” section.

The next step is to start building the model which is done so by the line “model = Sequential()”. The sequential model API allows the model to be built layer by layer which is all that’s needed for this project.

The first layer I used to build on is the Conv2D layer. This is constructed by passing the following arguments -

- **filters:** This parameter determines the number of kernels to combine with the input volume. Early layers learn less of these filters so less are needed. The deeper the layer, the more filters are needed to be learned. 32 was used for the starting layer.
- **Kernel_size:** This parameter is specifying the kernel size or window to be used when parsing through the data. There is no general answer to which size is the best, the most conventional sizes that are used 3 by 3, 5 by 5, and 7 by 7. 3 by 3 was used for this project
- **Activation:** The activation parameter chosen for the first layer was ‘relu’.

- **Input_shape:** This is the shape of the data that is to be read in. We already know the dataset contains 28 by 28 pixel images that have only 1 channel. A variable ‘input_shape’ was created with the last 3 dimensions of X_train’s shape for continuity purposes.

The next layer to add is a MaxPooling2D layer. This layer is used to clean up the data that is formed from the previous layer which has created feature maps that can represent or find a particular feature in the image, such as an edge of a shape. It then down samples the value found by choosing the largest value in the pool size window.

The layers following are a repeat of the first two with a flatten layer used to flatten the pooling data at the end.

The last 2 layers are dropout and dense. The dropout layer is added to help fight overfitting and is normally added right before the densely connected classifier. The dense activation function to be used in the final layer is ‘softmax’ as we are doing a multi-class, single label classification. The model’s architecture can be seen in the figure below.

Model 1

```
model = Sequential()

model.add(Conv2D(32, kernel_size = (3, 3),
                 activation = 'relu',
                 input_shape = input_shape))

model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(64, (3, 3), activation = 'relu'))

model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(128,(3,3), activation = 'relu'))

model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(62, activation = 'softmax'))

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 1, 1, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 62)                7998
=================================================================
Total params: 100,670
Trainable params: 100,670
Non-trainable params: 0
_________________________________________________________________
```

The next two steps are to compile the model and to train the model with the training data and validate the model's performance against the testing data. The parameters passed through in the compiler are the loss function, the optimizer, and the metrics. "Categorical_crossentropy" was chosen for this research project as it computes the loss between labels and predictions, which is needed for classification purposes. The optimizer chosen was "adam", as it's an efficient and

popular algorithm to use. Finally, the metrics chosen is “accuracy” as we’re trying to figure out how well the model performs. When fitting the model, the parameters that are passed through are X_train and y_train - (the training data and training labels), the batch size - (The number of samples that will be propagated, or trained, through the neural network), the epoch size - (the number of periods that the algorithm will work through the entire training dataset), the verbose - (the mode for how you want to see the model’s progress), and finally the validation data - (X_test and y_test). For the sake of saving resources and time, I chose to keep the epoch size at 10 and the verbose at 2.

Model Performance

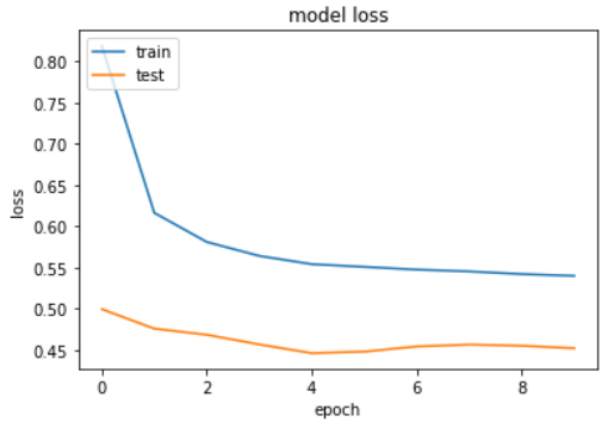
```
test_loss, test_acc = model.evaluate(X_test, y_test)
test_acc
```

```
3636/3636 [==============================] - 10s 3ms/step - loss: 0.4521 - accuracy: 0.8432

0.8432468175888062
```

```
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



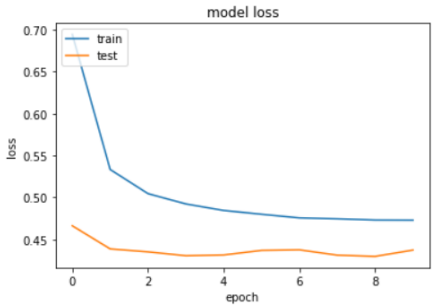
In the figure above, the results of the first model's run is recorded. The model performed "average" with an accuracy of 0.8432. The train and test loss both start to taper off and flatten out indication of a somewhat good fit. To see if I could get the fit closer, I experimented with dozens of models while fine tuning the parameters to see what the different results create. Here are a few of those models.

Model 2

```
model2 = Sequential()

model2.add(Conv2D(64, kernel_size = (3, 3),
                  activation = 'relu',
                  input_shape = input_shape))

model2.add(MaxPooling2D(pool_size = (2, 2)))

model2.add(Conv2D(128, (3, 3), activation = 'relu'))

model2.add(MaxPooling2D(pool_size = (2, 2)))

model2.add(Conv2D(265,(3,3), activation = 'relu'))

model2.add(MaxPooling2D(pool_size = (2, 2)))

model2.add(Flatten())

model2.add(Dropout(0.25))

model2.add(Dense(62, activation = 'softmax'))

model2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_15 (Conv2D)           (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 11, 11, 128)       73856
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 5, 5, 128)         0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 3, 3, 265)         305545
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 1, 1, 265)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 265)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 265)               0
_________________________________________________________________
dense_3 (Dense)              (None, 62)                16492
=================================================================
Total params: 396,533
Trainable params: 396,533
Non-trainable params: 0
_________________________________________________________________
```

```
In [144]:
1 test_loss, test_acc = model2.evaluate(X_test, y_test)
2 test_acc

3636/3636 [==============================] - 21s 6ms/step - loss: 0.4373 - accuracy: 0.8475

Out[144]: 0.8474850058555603

In [145]:
1 plt.plot(history2.history['loss'])
2 plt.plot(history2.history['val_loss'])
3 plt.title('model loss')
4 plt.ylabel('loss')
5 plt.xlabel('epoch')
6 plt.legend(['train', 'test'], loc='upper left')
7 plt.show()
```



For “Model 2”, I adjusted the kernel filter size by increasing the size 2 fold. The model’s learning curve’s fit is a little closer together but the testing curve can be seen increasing on the tail end. This is a sign of overfitting. If I were to add more epochs, the increase in the tail would be compounded. I kept experimenting with different values to see what effect they would have on the model. The next model I increased the batch size from 30 to 50.

Model 3

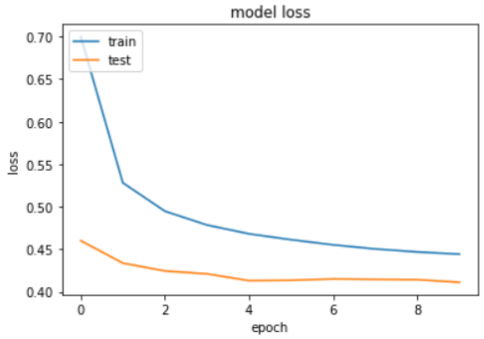
```
model3 = Sequential()

model3.add(Conv2D(64, kernel_size = (3, 3),
                  activation = 'relu',
                  input_shape = input_shape))

model3.add(MaxPooling2D(pool_size = (2, 2)))

model3.add(Conv2D(128, (3, 3), activation = 'relu'))

model3.add(MaxPooling2D(pool_size = (2, 2)))

model3.add(Conv2D(256,(3,3), activation = 'relu'))

model3.add(MaxPooling2D(pool_size = (2, 2)))

model3.add(Flatten())

model3.add(Dropout(0.5))

model3.add(Dense(62, activation = 'softmax'))

model3.summary()
```

```
Model: "sequential_39"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_118 (Conv2D)          (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_104 (MaxPoolin (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_119 (Conv2D)          (None, 11, 11, 128)       73856
_________________________________________________________________
max_pooling2d_105 (MaxPoolin (None, 5, 5, 128)         0
_________________________________________________________________
conv2d_120 (Conv2D)          (None, 3, 3, 256)         295168
_________________________________________________________________
max_pooling2d_106 (MaxPoolin (None, 1, 1, 256)         0
_________________________________________________________________
flatten_25 (Flatten)         (None, 256)               0
_________________________________________________________________
dropout_27 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_27 (Dense)             (None, 62)                15934
=================================================================
Total params: 385,598
Trainable params: 385,598
Non-trainable params: 0
_________________________________________________________________
```

```
test_loss, test_acc = model3.evaluate(X_test, y_test)
test_acc
```

```
3636/3636 [==============================] - 22s 6ms/step - loss: 0.4112 - accuracy: 0.8531

0.8530728816986084
```

```
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



For “Model 3”, the fit is even closer and without the increasing tail of model 2. For model 4, I decided to add another drop out layer.

Model 4

```
model4 = Sequential()

model4.add(Conv2D(64, kernel_size = (3, 3),
                  activation = 'relu',
                  input_shape = input_shape))

model4.add(MaxPooling2D(pool_size = (2, 2)))

model4.add(Dropout(0.25))

model4.add(Conv2D(128, (3, 3), activation = 'relu'))

model4.add(MaxPooling2D(pool_size = (2, 2)))

model4.add(Conv2D(256,(3,3), activation = 'relu'))

model4.add(MaxPooling2D(pool_size = (2, 2)))

model4.add(Flatten())

model4.add(Dropout(0.25))

model4.add(Dense(62, activation = 'softmax'))

model4.summary()
```

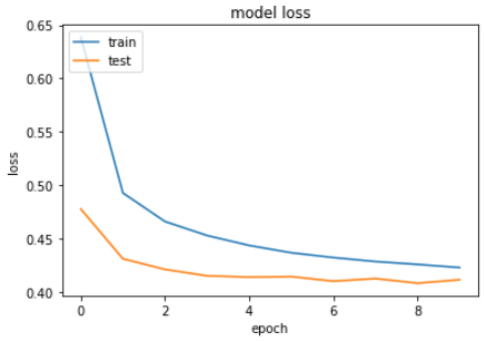
```
Model: "sequential_40"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_121 (Conv2D)          (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_107 (MaxPoolin (None, 13, 13, 64)        0
_________________________________________________________________
dropout_28 (Dropout)         (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_122 (Conv2D)          (None, 11, 11, 128)       73856
_________________________________________________________________
max_pooling2d_108 (MaxPoolin (None, 5, 5, 128)         0
_________________________________________________________________
conv2d_123 (Conv2D)          (None, 3, 3, 256)         295168
_________________________________________________________________
max_pooling2d_109 (MaxPoolin (None, 1, 1, 256)         0
_________________________________________________________________
flatten_26 (Flatten)         (None, 256)               0
_________________________________________________________________
dropout_29 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_28 (Dense)             (None, 62)                15934
=================================================================
Total params: 385,598
Trainable params: 385,598
Non-trainable params: 0
_________________________________________________________________
```

```
test_loss, test_acc = model4.evaluate(X_test, y_test)
test_acc
```

```
3636/3636 [==============================] - 20s 6ms/step - loss: 0.4116 - accuracy: 0.8522
```

```
0.8522390127182007
```

```
plt.plot(history4.history['loss'])
plt.plot(history4.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



For “Model 4”, we can see the fit is even closer than before but with a very slight increase on the tail of the testing curve. I decided to perform experiments with one more model to see what effects it had. In model 5, I decided to drop the kernel size back down to the original sizes (32,64,128 respectively) and keep the extra dropout layer.

Model 5

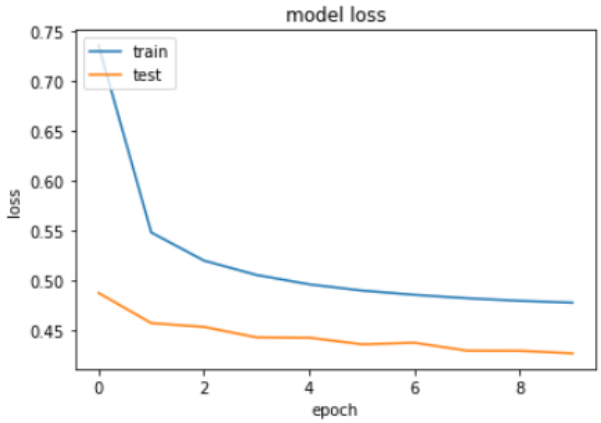
```
model5 = Sequential()

model5.add(Conv2D(32, kernel_size = (3, 3),
                  activation = 'relu',
                  input_shape = input_shape))

model5.add(MaxPooling2D(pool_size = (2, 2)))

model5.add(Dropout(0.25))

model5.add(Conv2D(64, (3, 3), activation = 'relu'))

model5.add(MaxPooling2D(pool_size = (2, 2)))

model5.add(Conv2D(128,(3,3), activation = 'relu'))

model5.add(MaxPooling2D(pool_size = (2, 2)))

model5.add(Flatten())

model5.add(Dropout(0.25))

model5.add(Dense(62, activation = 'softmax'))

model5.summary()
```

```
Model: "sequential_41"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_124 (Conv2D)          (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_110 (MaxPoolin (None, 13, 13, 32)        0
_________________________________________________________________
dropout_30 (Dropout)         (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_125 (Conv2D)          (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_111 (MaxPoolin (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_126 (Conv2D)          (None, 3, 3, 128)         73856
_________________________________________________________________
max_pooling2d_112 (MaxPoolin (None, 1, 1, 128)         0
_________________________________________________________________
flatten_27 (Flatten)         (None, 128)               0
_________________________________________________________________
dropout_31 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_29 (Dense)             (None, 62)                7998
=================================================================
Total params: 100,670
Trainable params: 100,670
Non-trainable params: 0
_________________________________________________________________
```

```
test_loss, test_acc = model5.evaluate(X_test, y_test)
test_acc
```

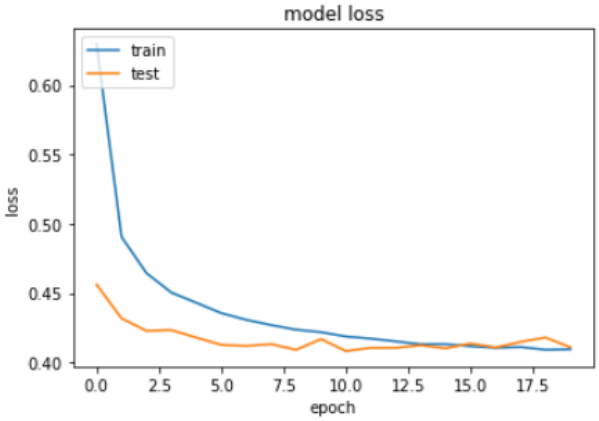
```
3636/3636 [==============================] - 11s 3ms/step - loss: 0.4272 - accuracy: 0.8484

0.8484306335449219
```

```
plt.plot(history5.history['loss'])
plt.plot(history5.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



After looking through the results of all the models listed above, model 4 piqued my interest the most and I decided to increase the epoch size to 20 to see how the model performed over a longer period of time. Here are the results of that model.

```
plt.plot(history6.history['loss'])
plt.plot(history6.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



We can see here the model starts to overfit at around epoch 16-17 and would benefit from early stopping. This model would be a good fit at around epoch 15.

Conclusion

Final Results and Samples

The model I decided to use for sampling was model 4 as it gave me the best fit according to the graph. The time needed to re-run the last model with less epoch was too much for the time and resources that I had. Refer back to the images of the models, model 4 produced an accuracy score of 85%. This number is low due to the structure of the EMNIST 'byclass' dataset. It is unbalanced and there are very similar classes that the neural network has trouble deciphering from such as "Z" and "z" or "0" and "o". Along with similar shaped classes, there's also the problem of the frequency of each class - refer back to the "The dataset" section of this paper. I could resample the data to either increase or decrease the frequencies of each class so that they are equal or just use the EMNIST 'balanced' dataset instead. As you can see these sample images were predicted correctly by model 4.

Final Thoughts

The goal of this research project was to learn more about artificial intelligence and machine learning, more specifically, deep learning was a great introduction to this field. Researching and creating convolutional neural networks was truly an experience. Dozens of hours were spent creating at least 40 models where a lot of experimenting happened to truly understand how a neural network works. This process was fun and I've learned an incredible amount over the semester. Going Forward, I believe I can create better, more efficient networks that will analyze more complex objects. A goal, if time allowed, was to be able to apply this knowledge to a program that can analyze handwriting in real time. I look forward to exploring deeper into the world of artificial intelligence. I would like to thank Dr. Soibam once more for allowing me to work under his guidance and for all of the resources.

References

- Chollet, F. (2018). *Deep learning with Python*. Shelter Island, NY: Manning Publications.
- Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from http://arxiv.org/abs/1702.05373
- Hardest, Larry. "Explained: Neural Networks." *MIT News | Massachusetts Institute of Technology*, 14 Apr. 2017, news.mit.edu/2017/explained-neural-networks-deep-learning-0414.
- Stewart, Matthew PhD Researcher. "Simple Introduction to Convolutional Neural Networks." *Medium*, 30 July 2020, towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac.
- Brownlee, Jason. "How to Use Learning Curves to Diagnose Machine Learning Model Performance." *Machine Learning Mastery*, 6 Aug. 2019, machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance.
- GeeksforGeeks. "CNN | Introduction to Pooling Layer." *GeeksforGeeks*, 26 Aug. 2019, www.geeksforgeeks.org/cnn-introduction-to-pooling-layer.
- Saha, Sumit. "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 Way." *Medium*, 15 Oct. 2020, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.