

Font Classification using Convolutional Neural Networks (CNN)

Co-authors: Deep Patel, Kenneth Hora, Gunjan Neopaney

Text Section (Methods & Results)

Objective and Goal

Convolution Neural Networks (CNN) are a type of feedforward artificial neural network and supervised algorithm that learns a collection of parameters by training on a dataset. The objective of this analysis is to build, explore, and compare different CNN models for classification of font type where inputs are assigned a class.

Data Selection (Step 1)

The table below contains metrics on font data used in this HW. Each case is a representation of a character (20x20 pixels) in a particular font. Each pixel is represented by a number or intensity, 0 to 255, on the gray scale and represents a feature of the data, hence there are 400 features in this data set. The images can be viewed in any color, but we chose to visualize the images in the colormap of grayscale. There are 5 classes of fonts which are described below:

Font	Class	Cases	% of Total Cases	Training Set Cases	Testing Set Cases
Courier	CL1	4262	19.3%	3410	852
Calibri	CL2	4768	21.5%	3814	954
Lucida	CL3	3794	17.1%	3035	759
Sitka	CL4	4500	20.3%	3600	900
Times	CL5	4805	21.7%	3844	961
Total		22129		17702	4427

Table 1: Data Metrics

Total Cases: 22129
Number of Classes: 5
Number of Features: 400

Reshaping and Visualizing Characters (Step 2)

The data in its raw form comes in a .csv file, which is imported into a 2-dimensional numpy array. Specifically, it is an Nx400 array, where N is the number of cases in each font file. To show the images with the 'imshow' function, the data must be reshaped into 20x20 pixel images. This is done with the 'reshape' method in Python. The line of code used is as follows:

```
.reshape(N, 20, 20)
```

The table below shows a sample of a specific character, "B", across the five different fonts. Values of 0 and 255 appear as completely white or black pixels, respectively. As the value increases the pixel color becomes darker.

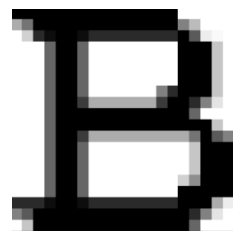
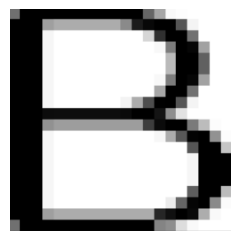
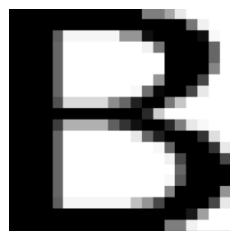
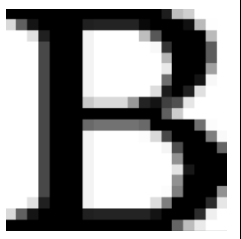
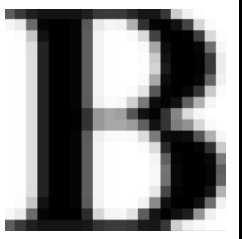
				
Calibri	Courier	Lucida	Sitka	Times

Table 2: 'B' character across the 5 fonts.

For the data to be used in a convolution layer in Tensorflow, it must be reshaped slightly differently; it needs an additional dimension representing the number of layers which make up the image. Color images are made up of three channels; a red, a blue, and a green channel, and hence have a length of 3 for the new dimension. Black and white images are made up only of a grayscale channel, hence the length of the new dimension will be 1. This reshaping is done as follows:

```
.reshape(N, 20, 20, 1)
```

Train/Test Set

As is done with any supervised learning algorithm, our data has been split into a training and testing set. For this analysis the train/test sets are split in a 80%/20% fashion by random selection. The distribution and number of cases resulting from the split of training and test sets is shown in the data metrics section above.

Define CNN Model (Step 3 & 4)

A CNN is made up of various layers which perform unique transformations on cases in a data set. In our model these layers include the following: convolution layers, maxpooling layers, a flattening layer, a hidden layer, and an output layer. These layers and how their transformations work are described in greater detail below:

Convolution layer: A convolution layer takes a 3 dimensional numpy array as its input. In our model, this 3D array is a representation of a pixel image of dimension $20 \times 20 \times 1$, where 20×20 represents the image height and width in pixels and the $\times 1$ represents its single grayscale layer, as it is a black and white image. Next, a window is defined by size $n \times n$, where n is typically an odd number. This window traverses laterally and longitudinally across the input image, though not simultaneously, m pixels at a time, where m is the “step size” or “stride” of the window. This is where CNN gets its namesake; the window snakes around the input image. Each element in the window corresponds to a unique weight. For each position the window occupies on the input image, a value is generated by summing the paired products of weights and pixel values with a bias. This value is then assigned to the state of a new pixel in a new image, called a channel. Typically, multiple channels are created in a convolution layer, each with its own window of unique weights and bias.

Maxpooling layer: Maxpooling layers are used to reduce the amount of information in the output of a convolution layer, keeping only the “most relevant” data. Similar to a convolution layer, a maxpooling layer has a window which snakes around the output channels of a convolution layer. However, there is no overlap of pixels as the window moves about. A typical window size is 2×2 and hence moves two pixels at a time, laterally or longitudinally. The operation the maxpooling layer performs is also much simpler; it merely takes the maximum value of the values inside the window and assigns that value to a new pixel in a new image. Hence, a maxpooling layer will keep the same number of channels as its previous layer, however it will reduce the number of pixels by a factor equal to the product of the window height and width. i.e if you have a $100 \times 100 \times 16$ output of a convolution layer and a 2×2 window in the proceeding maxpooling layer, you will get a $50 \times 50 \times 16$ output.

Flattening layer: a flattening layer takes the 3-dimensional output of a convolution or max pooling layer and transforms into 1-dimension. It does not make mathematical changes to values. This is done so that the data can be fed into a hidden layer of an MLP.

Algorithm Parameters: Values and Definitions

Epoch: 100

Epoch is the number of times the total training cases are run through the algorithm. Our initial epoch size is 100, however this number may be adjusted in order to find training overfit times if needed.

Batch Size: 133

Batch size is the number of training cases the algorithm takes in for each learning iteration. Our initial batch size is chosen based on the calculation $\text{batch} = \sqrt{\text{\# of cases in the training set}}$. Other batch sizes may be experimented with.

H: 90, 150, 200

H is the size of the hidden layer. PCA can be performed to find appropriate maximum and minimum values for H, however the 3 values above will be used as starting points in this analysis.

Loss functions:

The concept of loss indicates how poorly a model is performing in a given instant. Loss functions define how model inaccuracy is measured. Every model attempts to minimize the loss, thereby increasing accuracy.

For classification problems Categorical-Cross-Entropy is used. The equation is defined below:

$$CE = - \sum_i^C t_i \log(f(s)_i)$$
$$CE = - \sum_{i=1}^{C'=2} t_i \log(f(s_i)) = -t_1 \log(f(s_1)) - (1 - t_1) \log(1 - f(s_1))$$

Vector t is a one-hot encoded vector which represents the ground truth (true class) of the case.

When autoencoding Mean-Squared-Error is used, which takes the mean of the squared errors over all samples.

Activation Functions:

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

ReLU:

The ReLU activation function returns the maximum of 0 and z where z is the input.

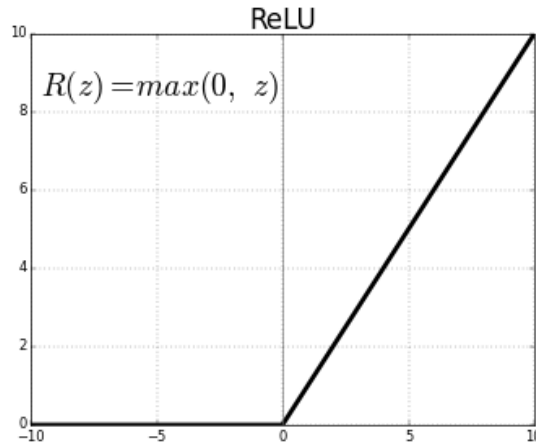


Figure 1: ReLU Activation Graph

Softmax:

The softmax function is a generalization of the logistic function to multiple dimensions. It normalizes the output of a layer to a probability distribution over predicted output classes. The sum of the output layer vector containing the probabilities for each class is 1. The softmax activation function helps determine the predicted class by revealing which class is most likely to be chosen.

$$\frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}} \quad \text{for } i = 1, \dots, J$$

Optimizer:

Optimizers are algorithms or methods used to change the attributes of neural networks such as weights and biases to reduce the loss. The optimizer used in our model was Adam optimizer which is an optimization algorithm that is generally known to be computationally efficient and easier to implement for large amounts of data with better results. In our MLP models, the default learning rate of 0.001 was maintained for all models.

Padding (Conv2D layer parameter)

Padding takes either “valid” or “same” as an argument. If padding is set to “same” then the height and width portions of the output shape will be the same as the input. This will be done by Tensorflow automatically padding the edges of the input image with zeros. If padding is set to “valid” then the output height and width will be reduced based on the window stride and size.

Strides (Conv2D layer parameter)

The strides parameter defines how far a window is translated laterally and longitudinally.

Filters (Conv2D layer parameter)

Filters is a parameter which is taken as a single integer. It represents the dimensionality of the output space (i.e. the number of output channels in the convolution).

Window (kernel size) (Conv2D layer parameter)

The convolution window shape is defined by a tuple of two integers which represent the height and width of the window in pixels.

CNN Architecture

The architecture of our initial CNN is described below:

Layer Type	Input Shape	Output Shape	Trainable Parameters (number of weights and biases)	Argument Settings
Convolution 1	20x20x1	16x16x16	416	Window = 5x5 Stride = (1,1)
Maxpool 1	16x16x16	8x8x16	0	Window = 2x2
Convolution 2	8x8x16	6x6x16	2320	Window = 3x3 Stride = (1,1)
Maxpool 2	6x6x16	3x3x16	0	Window = 2x2
Flatten	3x3x16	144	0	
Hidden	144	H= 90, H= 150, H= 200	13050, 21570, 29000	Activation = ReLU
Output	H= 90, H= 150, H= 200	5	455, 755, 1005	Activation = Softmax

Table 3: Initial CNN architecture

As shown in the table above, our CNN architecture will consist of 2 convolutional layers (both with no padding) which are both followed by a max pooling layer with window size 2x2 (A ReLU activation function is also applied prior to max pooling). After an input processes through the CNN layers, a new output of 3x3 images with 16 channels are produced which are then flattened to create a 144 neuron input for a MLP. The MLP which is attached to the end of the CNN has one hidden layer, and we explore hidden layer sizes, H=90,150 and 200. The hidden layer neurons also undergo the ReLU activation function. Finally, the last output layer consists of 5 neurons representing the 5 classes and a softmax activation function is applied which normalizes this output layer to a probability distribution over the predicted output classes

Due to us exploring three different values of H, three models with different numbers of weights and biases (parameters) in the network are produced. Since all three models have the same architecture before the flattening layer the number of parameters of the convolutional layers stay the same for all the models. That is, since the first convolutional layer has a 16 different filter windows of size 5x5 each with 1 bias associated with it, the

number of parameters in the layer will be $(25+1)*16$ or 416 parameters. The ReLU response functions and maxpooling functions do not have learnable parameters associated with them as they are just a mathematical function applied to the input. So then, the second convolutional layer also has 16 different filter windows but of the size 3x3 each with 1 bias. However, in this convolutional layer it has to take into account the 16 filters that the first conv layer already learned. So the number of parameters in this layer will be $(9*16+1)*16$, or 2320 parameters. So the total number of parameters for the convolutional layers is 2,736.

Now for each of the models, as mentioned before, an MLP is connected right after the convolutional layers. The input of the fully connected MLP is the flattened output of the conv layers, resulting in 144 neurons. The output layer as said before contains 5 neurons. The hidden layer size changes with the three values of H, creating three different models. As the value of H increases, we can expect the total number of weights and biases to also increase in the MLP. These values are also shown in the table of the CNN architecture above. For each of the 3 models, the total number of parameters can be calculated by taking the sum of the parameters in the convolutional layers and the fully connected layers, the values can be seen in the table below.

CNN model	Total number of weights and biases	Infos-parameters ratio
H=90	16,241	5.45
H=150	25,241	3.51
H=200	32,741	2.70

Table 4: Number of weights and biases and Infos-to-parameters ratio

The total number of cases in our training set is 17,702, and since each case brings 5 infos (one-hot encoding of true class), the total number of infos brought by the training set is $17,702*5$ or 88,510. The parameter estimations by automatic learning is generally more stable when one has more infos than parameters. One can calculate the ratio between the number of infos and the number of parameters to get an idea for the robustness, or the capacity to generalize well for the model. The ratios can also be seen on the table above. Generally, a lower ratio can lead to potential overfitting, due to one parameter being responsible for a really small number of infos, and generalizing too well to the data. Similarly, an extremely high ratio can lead to potential underfitting where one parameter can be responsible for a lot of infos and so the parameters fail to completely learn the important details. For the CNN model with H=90, the ratio is about 5.45 infos per parameter, for H=150, the ratio is about 3.51 infos per parameter, and for H=200, the ratio is about 2.70 per parameter. It is important to find a good mix between the number of

infos and the number of parameters. For our case, perhaps an increase in the number of data would be beneficial, as it would allow for the ratio to be a little higher, but with always an option to increase the number of parameters by adjustments to the CNN architecture.

Dropout Option

The dropout option is a technique which is used to reduce overfitting to a training set. Architectures with and without the dropout option will be compared in our analysis. The dropout option temporarily removes neurons from the hidden layer by randomly selecting each neuron with a 50% probability. After each training batch all neurons are added back, then again taken out with the same probability. The probability of dropout can be changed, though 50% is a common starting point.

When training a model with a dropout option, we opted to add one dropout layer to our CNN architecture right after our hidden layer. Here there were the most parameters present. We chose not to add a dropout layer to our input layers including the convolutional layers due to the fact that the input sizes within those layers were not big. Hence, intuitively it would not make much sense to minimize neuron activity within those layers.

Training and Monitoring CNN with $h=90$

The first CNN training model was launched using the architecture described above and the hidden layer size of $h=90$. When training the model, the cross-entropy function and Adam optimizer were used to monitor the performance of the model. As mentioned above, the model will be trained using without the dropout option first and then with the dropout option for effective comparison. To determine the optimal number of epochs, two plots were used but more emphasis was put on the loss vs time plot. First, the cross-entropy loss vs time measured in the number of epochs showed a decreasing curve with the increase in the number of epochs. Secondly, the accuracy plot vs. time measured in the number of epochs showed an increasing curve with the increase in the number of epochs. In interpreting the plots and deciding the optimal number of epochs, two factors were considered which were stability of the curve and overfitting. The stopping value was determined using these two factors.

Without Dropout

When the model was trained without dropout and with 100 epochs, notice in the figure below, the cross entropy loss curve on the testing set has stabilized around 35 epochs and slowly starts to incline afterwards which indicates possible overfitting. Comparatively, the accuracy curve on the testing set flattens out around 35 epochs which also indicates the best number of epochs. The accuracy testing curve is perhaps not the best to analyze

for overfitting as accuracy is more stubborn as predictions need to go over/under a threshold to actually change accuracy. While the loss changes with just raw predictions from the network.

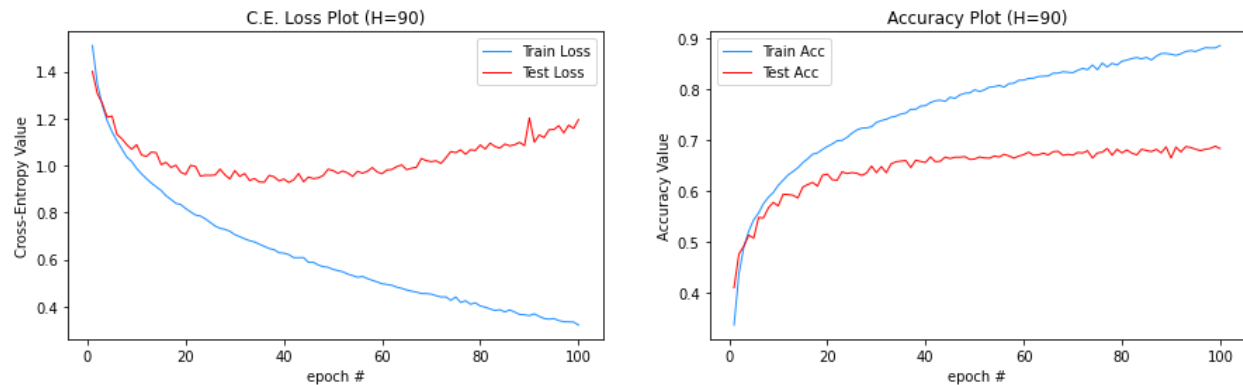


Figure 1: Cross-Entropy Loss and Accuracy Plot for CNN with h=90 (No Dropout)

With Dropout

To reduce the problem of overfitting on cross-entropy loss, the dropout technique was implemented while training the same model with 100 epochs. With the dropout technique, we can see the difference in the stability of the curves in the plots. The testing loss in cross-entropy loss plot seems to flatten out around 40 epochs which is a little higher than the optimal number of epochs determined without dropout technique. However, there is no overfitting observed afterwards. The testing curve of the accuracy plot remained essentially the same throughout the training. It can also be noticed that due to the dropout technique, the training curves on both plots have been affected as well, making its slope much gentler. Although the inclusion of the dropout layer affected the curves, the general loss value and accuracy around their respective best epochs stayed consistent with each other.

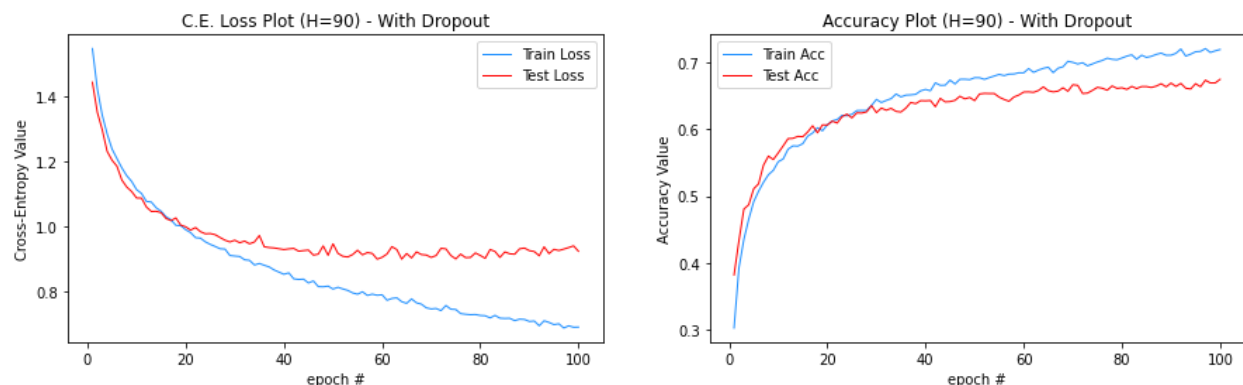


Figure 2: Cross-Entropy Loss and Accuracy Plot for CNN with h=90 (With Dropout)

Training and Monitoring CNN with h=150

The second CNN training model was launched using the hidden layer size of $h=150$. The CNN model architecture and all other parameters remain the same. When the training was launched with 100 epochs on this model without dropout option, we can see the similar behavior we observed in the plots of $h=90$ without dropout. However, the cross-entropy loss plot seems to stabilize early at 20 epochs and then slowly inclines after 40 epochs possibly due to overfitting. Similarly, in the accuracy vs time plot, the testing accuracy also seems to stabilize around 20 epochs indicating the best number of epochs for CNN model with $h=150$.

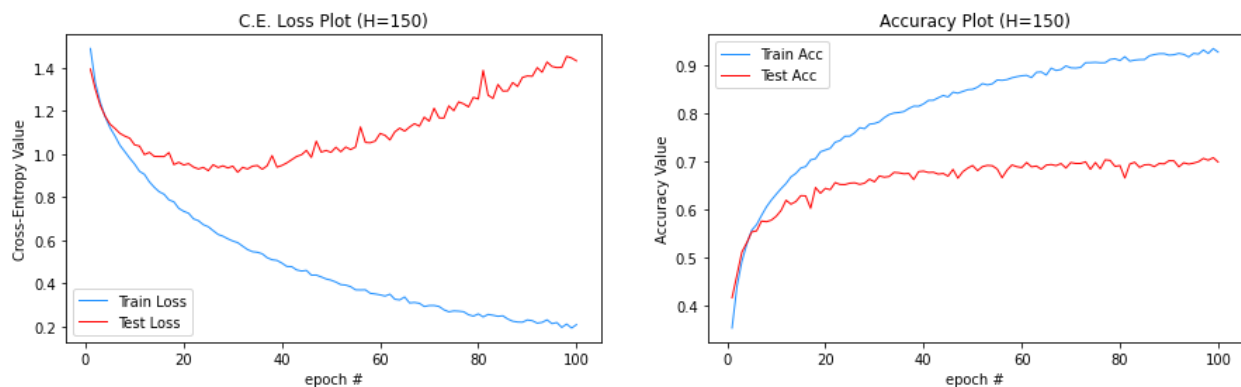


Figure 3: Cross-Entropy Loss and Accuracy Plot for CNN with $h=150$ (No Dropout)

With Dropout

Again, to overcome the problem of overfitting, the dropout technique was implemented on the same model and the model was trained again with 100 epochs. The difference in the curves of the plots is clearly visible when compared to the plots without dropout option. Here, we can see the testing curve in the cross-entropy loss plot seems to stabilize at 40 epochs and the same goes for the testing curve in the accuracy plot. Notice there is no irregular behavior in the cross-entropy plot after the testing curve stabilizes eliminating overfitting.

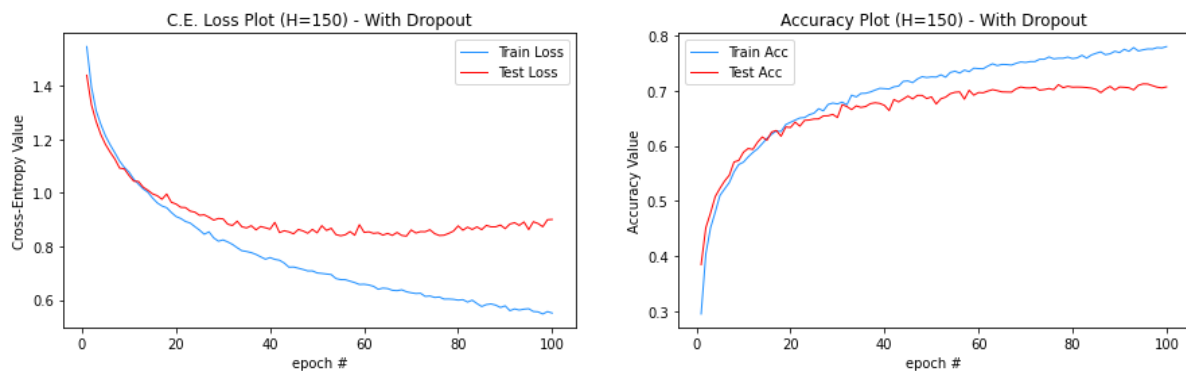


Figure 4: Cross-Entropy Loss and Accuracy Plot for CNN with $h=150$ (With Dropout)

Training and Monitoring CNN with h=200

The training was launched one more time with increased size of hidden layer ($h=200$). Similar to $h=90$ and $h=150$, the same CNN model architecture was used and training was implemented on 100 epochs without dropout option. By observing the plots, the performance behavior was similar to the other two hidden layer sizes without dropout option. However, in the cross-entropy loss plot here, the testing curve seems to barely stabilize and inclines much more rapidly after approximately 30 epochs indicating overfitting. The accuracy plot also shows that testing accuracy stabilizes around 30 epochs.

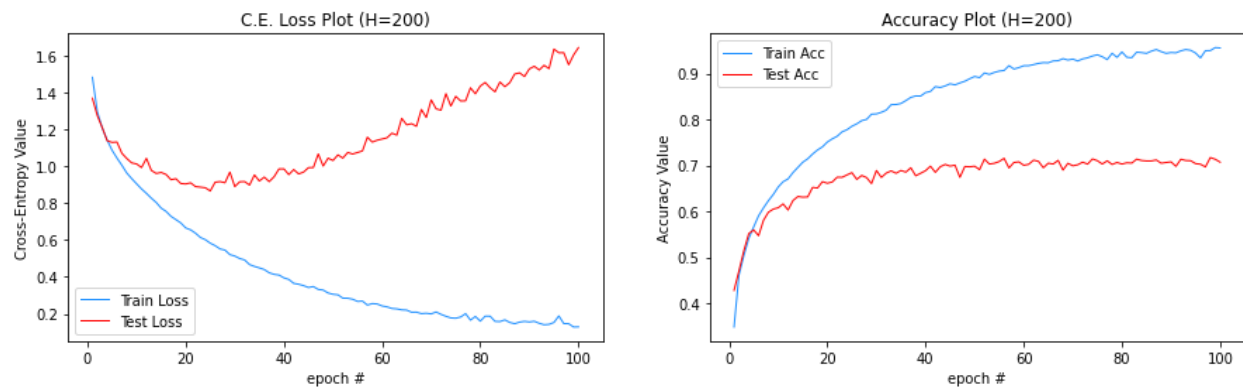


Figure 5: Cross-Entropy Loss and Accuracy Plot for CNN with $h=200$ (No Dropout)

With Dropout

Once again, to reduce or eliminate the overfitting issue, the dropout technique was implemented on the same model and the training was launched again with 100 epochs. As expected, the dropout techniques reduced overfitting from the cross-entropy plot and the test loss curve seems to stabilize around 40 epochs. Similarly, the testing accuracy in the accuracy plot seems to stabilize around 40 epochs.

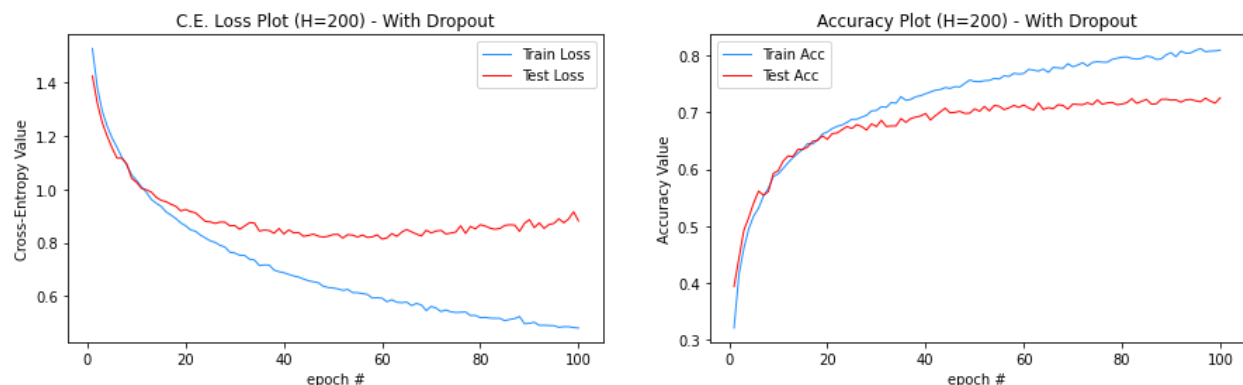


Figure 6: Cross-Entropy Loss and Accuracy Plot for CNN with $h=200$ (With Dropout)

Analysis of Performance (Step 5)

Global Accuracies

On each of the three models described above, two training methods were shown, one with dropout and one without. In general, as mentioned before, the loss and accuracy values around their respective best epochs seemed to be very similar. However, it was clear that training the networks with dropout was preventing overfitting drastically. So when analyzing the performances of each of these models, we chose to include the dropout functionality in the model.

The performance of CNN models was evaluated using the optimal number of epochs as determined using the stability, overfitting and stopping time factors mentioned earlier. The global accuracies of the testing set of each model were used as the first measure of comparing the performances of each CNN model with different hidden layer sizes. The global accuracies of each CNN with h-values are shown below.

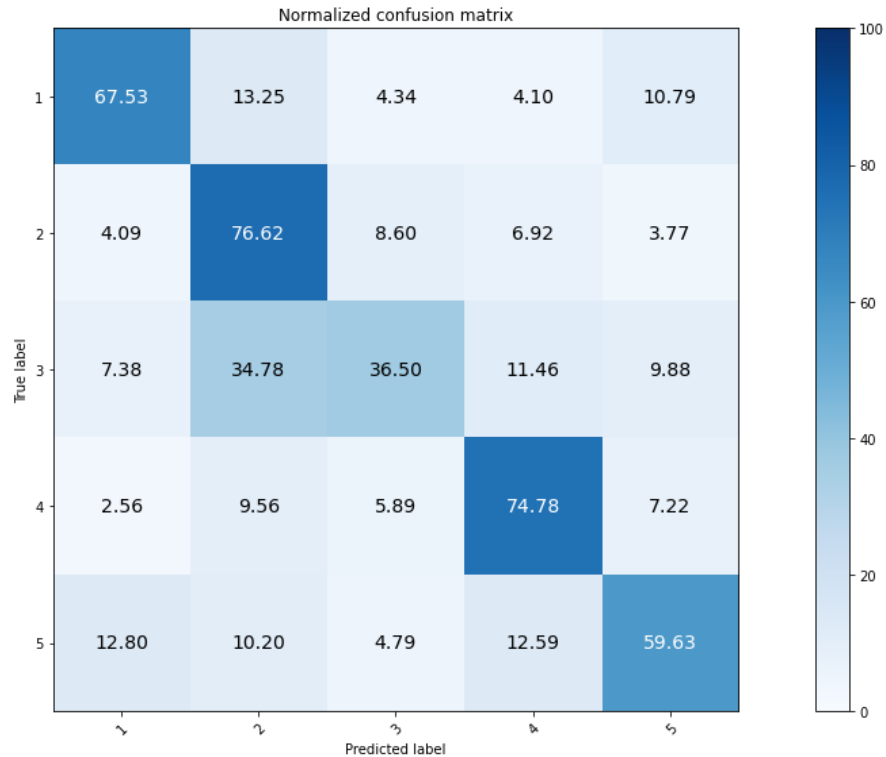
CNN model	Global Accuracies
h= 90	63.93%
h= 150	66.73%
h= 200	69.01%

Table 5: Global Accuracies of each CNN model

As it is clearly noticeable, with the increasing size of the hidden layer, the global accuracy also increases which is not surprising as the performance of the CNN models generally tends to increase with higher dimensions of the hidden layer.

Confusion Matrices for each class

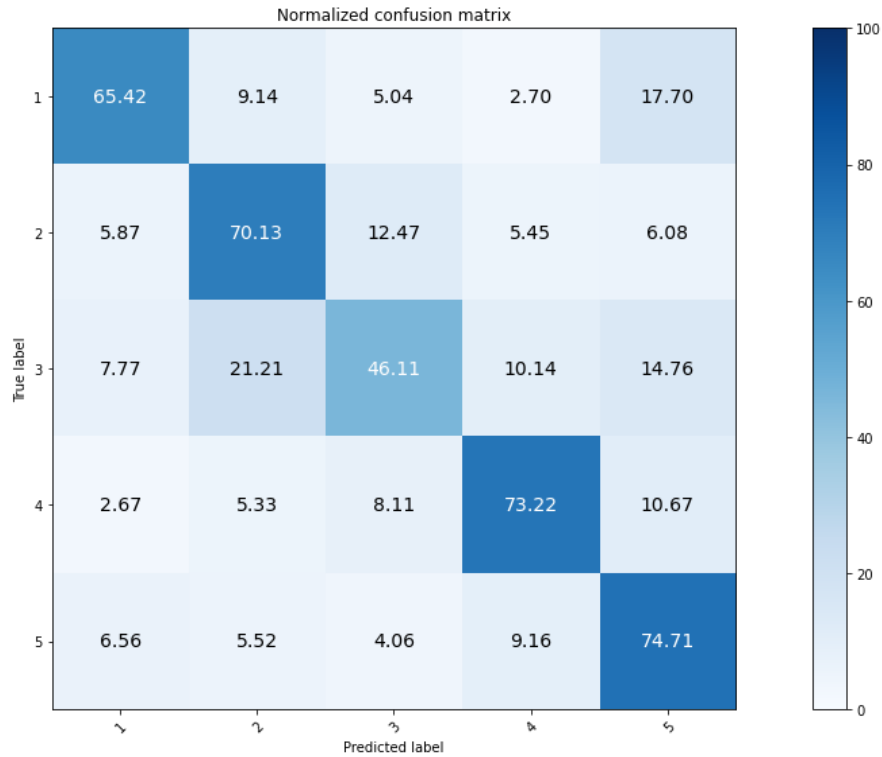
To get deeper insight of the CNN model performance, the confusion matrix was used as a measure of performance. The confusion matrix shows the number of cases correctly classified for each class across the diagonal terms. The non-diagonal terms of the confusion matrix shows the number of cases misclassified. The columns in the confusion matrix shows the predicted number of cases for the corresponding class in the column label and the rows in the confusion matrix shows the true number of cases for the corresponding class in the row label.



Global Accuracy= 63.93%

Figure 7: Confusion Matrix for CNN with h=90 (Test set)

For testing set performance of CNN with h=90, the class 2 and class 4 reported the highest accuracies (76.6% and 74.8%) among other classes. Following class 2 and class 4 were class 1 and class 5 with slight drop in accuracies from the highest class 2 accuracies. Class 3 resulted in significantly lower accuracy (36.5%) than other classes. The major confusion resulting in misclassification was between CL3 and CL2. The other confusion with more than 10% misclassification was between CL1 and CL2, CL5 and CL1, CL5 and CL4, CL3 and CL4, and CL1 and CL5.

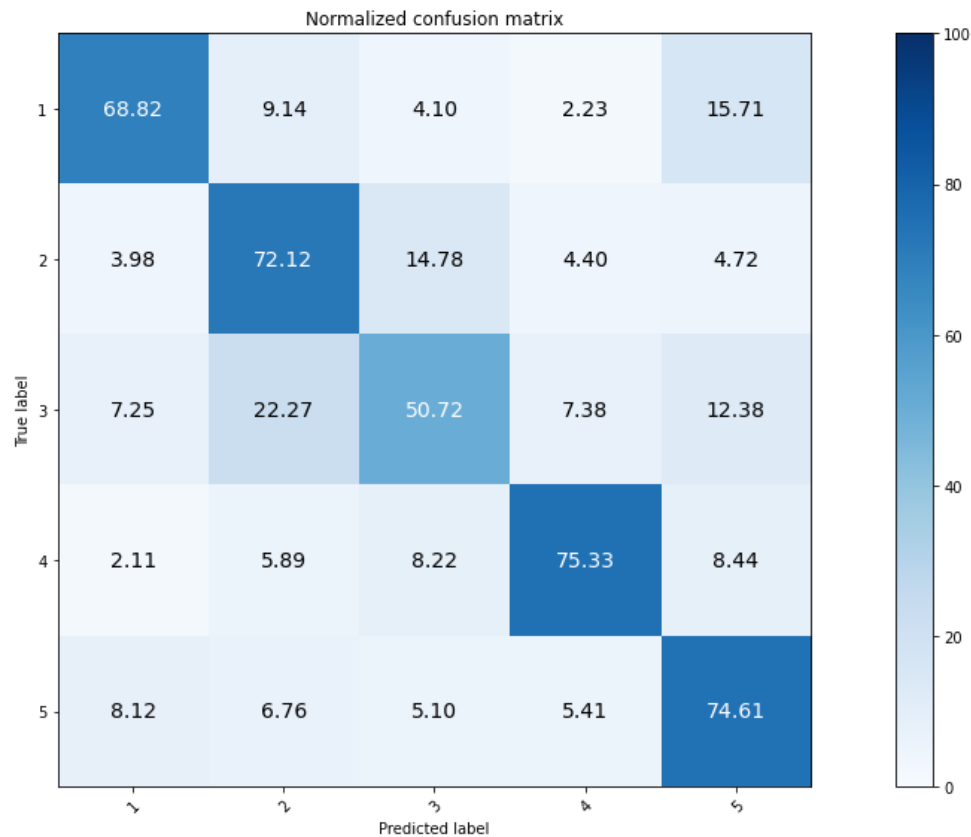


Global Accuracy= 66.73%

Figure 8: Confusion Matrix for CNN with h=150 (Test set)

For testing set performance of CNN with h=150, the class 5 and class 4 reported the highest accuracies (74.7% and 73.2%) followed by class 2 (70.1%) and class 1 (65.4%) with slight drop in accuracies from the highest class 5 accuracies. Class 3 resulted in significantly lower accuracy (46.1%) than other classes. Although CL3 was also the highest misclassified class in CNN with h=90, the class accuracy did improve by roughly 10%. In more comparison with CNN of h=90, the CL5 accuracy improved significantly while CL1, CL2, and CL4 accuracies decreased slightly. The major confusion resulting in misclassification was again between CL3 and CL2. Other than that, the other eye-catching numbers in the confusion matrix were 17.7 which was the percentage of CL1 cases misclassified as CL5, and 14.8 which was the percentage of CL3 misclassified as CL5.

Test set Confusion Matrix

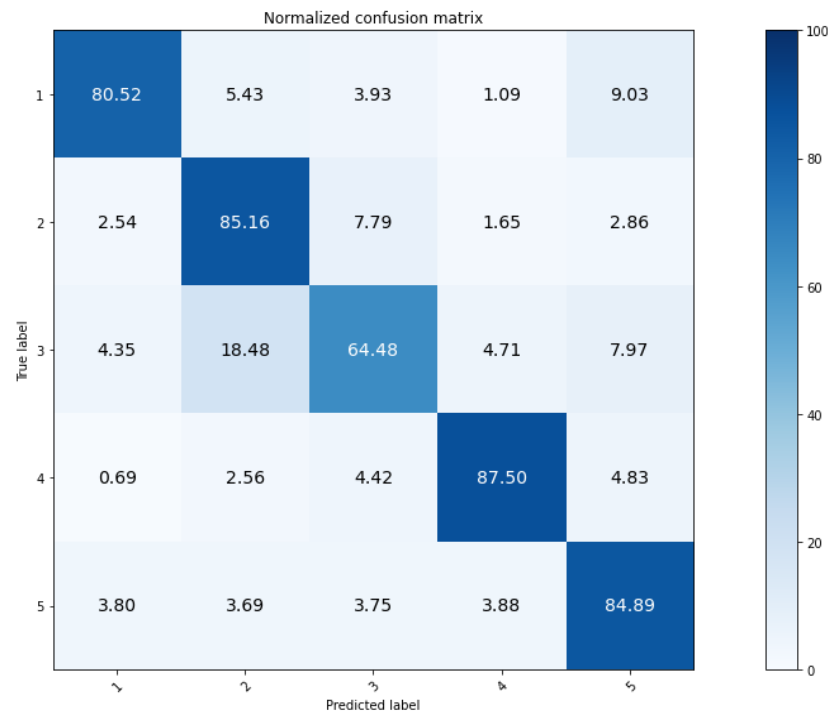


Global Accuracy = 69.01%

Figure 9: Confusion Matrix for CNN with h=200 (Test set)

For testing set performance of CNN with h=200, the class 4 reported the highest accuracies (75.3%) followed by class 5 (74.6%), class 2 (72.1%) and class 1 (68.8%). Class 3 resulted in significantly lower accuracy (50.7%) than other classes although improved slightly than CL3 accuracy (46.1%) for CNN with h=150. In more comparison with CNN with h=150, CL5 accuracy remained nearly similar while CL2, CL3, and CL4 accuracies improved slightly by roughly ~2%. The major confusion resulting in misclassification was again between CL3 and CL2. Other than that, the other eye-catching numbers in the confusion matrix were 15.71 which was the percentage of CL1 cases misclassified as CL5, and 12.38 which was the percentage of CL3 misclassified as CL5. Overall, when leaving out CL3, the confusion matrix of CNN with h=200 has the class accuracies which are not too distinct from each other.

Train set Confusion Matrix



Global Accuracy= 81.14%

Figure 10: Confusion Matrix for CNN with h=90 (Trainset)

To ensure that the best performing CNN test set does not have major discrepancies from the train set, the confusion matrix of the train set was also evaluated. The pattern of the class accuracies were similar to that of testing accuracy with CL4 being the highest followed CL2, CL5, and CL1 with not too distinct accuracies. CL3 in the train set also had the lowest accuracy.

Confidence Intervals

To evaluate if all three CNN models are similar or if there is any significant variation between models, the 95% confidence intervals were computed to observe any overlap or disjointedness. The table below shows the 95% confidence intervals for each CNN.

For h=90				For h=150				For h=200			
Class	Lower Limit	Upper Limit		Class	Lower Limit	Upper Limit		Class	Lower Limit	Upper Limit	
CL1	64.38	70.67		CL1	62.22	68.61		CL1	65.71	71.92	
CL2	73.94	79.31		CL2	67.22	73.03		CL2	69.27	74.96	
CL3	33.07	39.92		CL3	42.57	49.66		CL3	47.17	54.28	
CL4	71.94	77.62		CL4	70.33	76.12		CL4	72.52	78.15	
CL5	56.52	62.73		CL5	71.97	77.46		CL5	71.86	77.36	
Overall	62.51	65.34		Overall	65.34	68.11		Overall	67.65	70.37	

Table 6: 95% Confidence Intervals of each CNN

When global accuracies were compared, there was no major overlap between the accuracies of all three CNN models indicating significant variation between all three CNN models. However, when class accuracies are compared, we can see major overlap for CL4 and CL1 among all three CNNs which indicates there is no significant variation between accuracies of classes CL1 and CL4 among all three models. Between $h=150$ and $h=200$, CL1, CL2, and CL5 also have an overlap indicating no significant variation between these classes for CNN with $h=150$ and $h=200$. There is no major overlap for CL3 across all three CNNs.

Visualizing the convolutional layers of the best model ($h=200$)

To visualize the outputs produced by the CNN, more specifically the convolutional layers itself, we can see the progression of changes and images produced by feeding it an input image. As an example, here we select a case that shows the letter “B”. The left image below in Figure 11 shows the original *Lucida* font for “B”. The right image is actually not an image but the 144 neurons that were generated by 16 channels of 3×3 images.

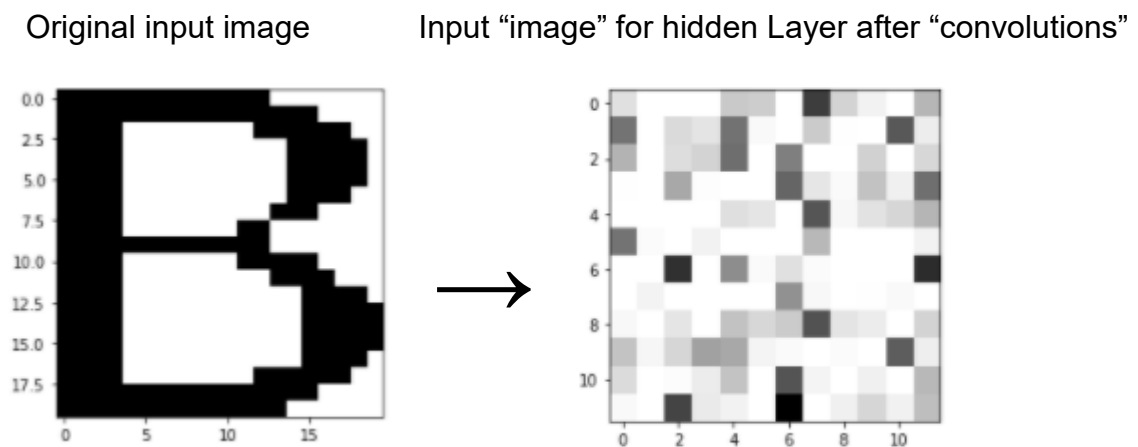


Figure 11: Original Input image and the image after convolutions for Hidden layer

On the CNN architecture specified at the beginning of the paper, it was said that the first convolutional layer receives an input image of the shape $20 \times 20 \times 1$, which is exactly the shape of the image above on the left. Now after applying 16 different windows of size 5×5 with stride 1 on this image, we receive 16 different images or channels of size 16×16 . These channels are shown below in figure 12. It can be seen that each channel looks slightly different than the others, and this is due to each window containing its own sets of learned weights and biases that intend to extract unique features for the network to “understand”. For example, one can say that since higher states/pixel intensities produce

darker spots, channel 5 highlights the left line of the “B” for the network while channel 12 highlights the curves of the “B”.

First convolutional image

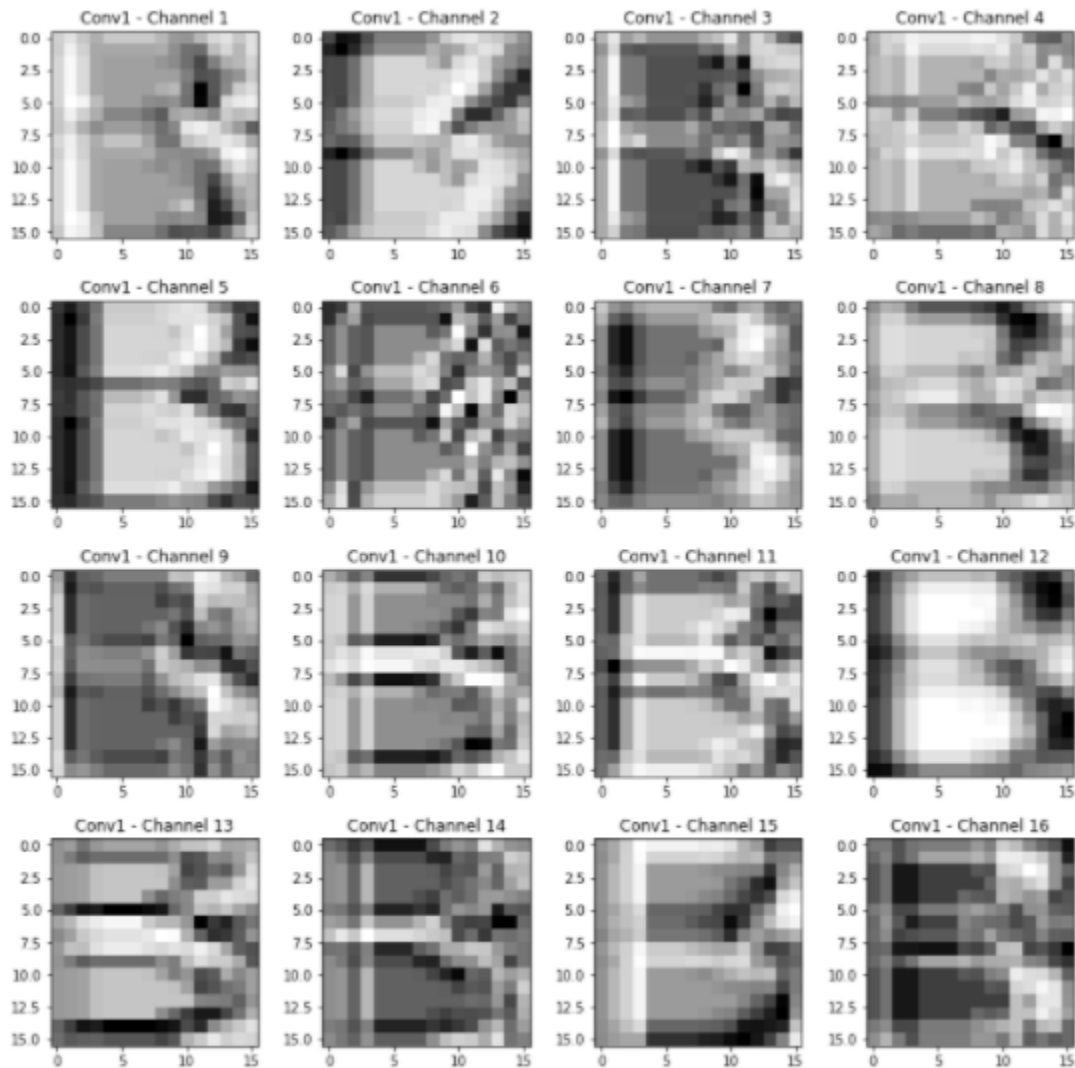


Figure 12: Visualization Example of Images produced in 16 channels of first convolutional layer

After passing the original image through the first convolutional layer, we apply the ReLU response function and the maxpooling layer producing a set of 16, 8x8 images. These images are then fed to the second convolutional layer with 16 different window size 3x3 as intended by our CNN architecture. This produces 16 new channels each with a 6x6 image. Here the images for us are merely just pixels, however the weights and biases that were implemented by the model on the 3x3 windows extracted potentially more details to help the network understand the image.

Second convolutional image

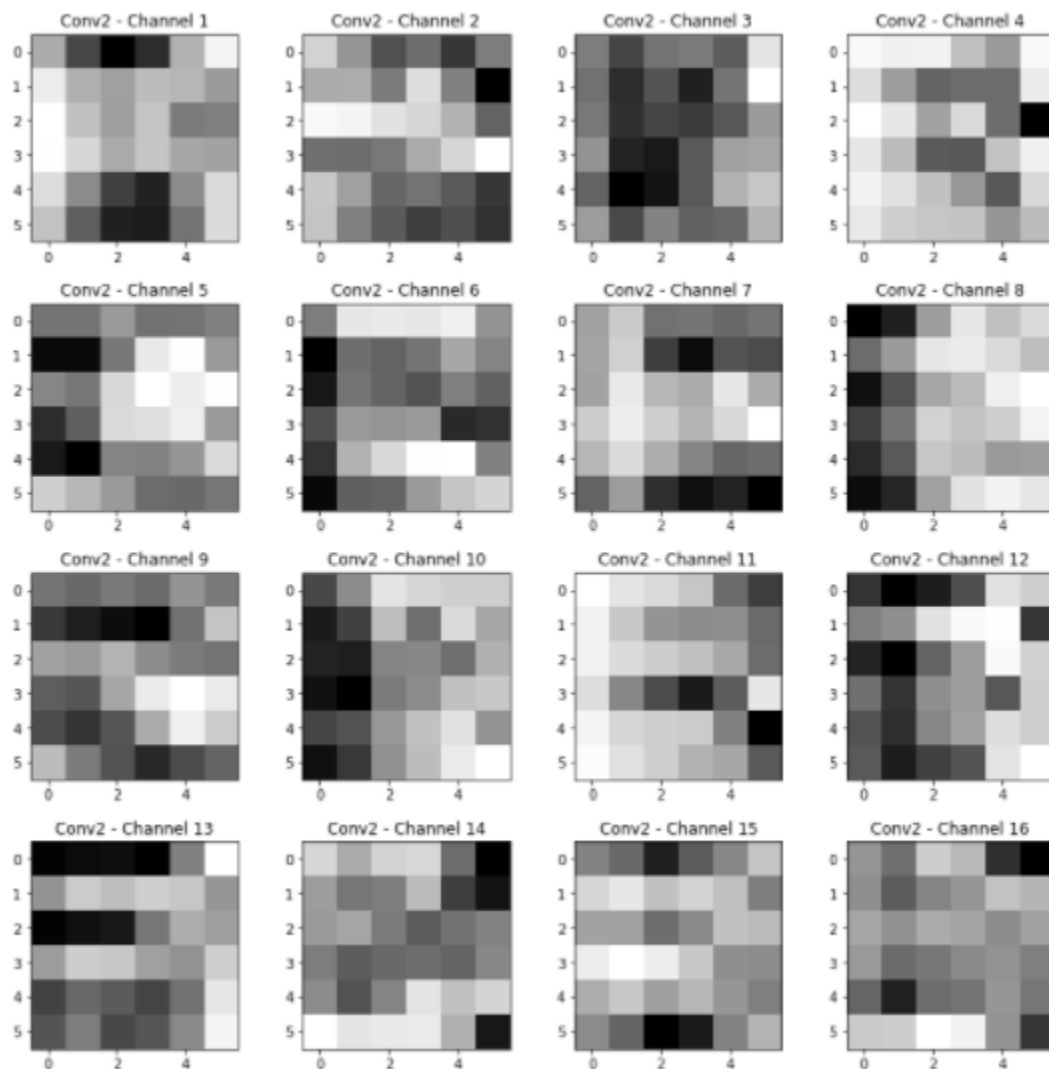
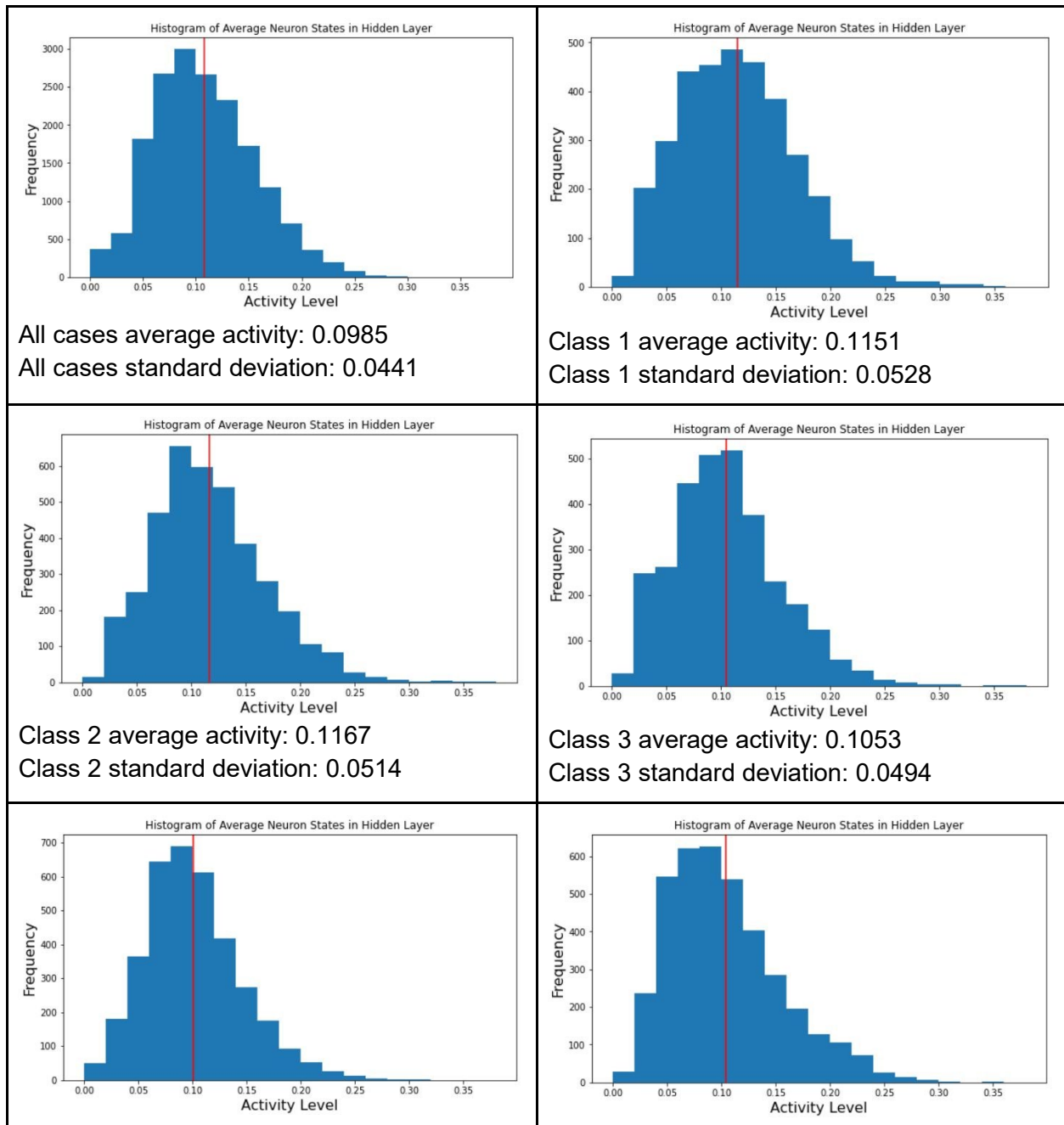


Figure 13: Visualization Example of Images produced in 16 channels of second convolutional layer

Lastly, after this second convolutional layer, again a ReLU function and a max pooling layer is applied, where now the output consists of 16 channels of 3x3 images. And flattening it produces a 144 neuron input for the hidden layer. This image is what was being shown at the beginning of this section. It is important to know however, since our model had a moderate performance of global accuracy ~69%, the weights and biases for these convolutional layers can potentially be adjusted even further to perhaps produce even more detailed/different images.

Average activity (ACTn) on hidden layer

The hidden layer can be analyzed by showing a histogram of average neuron activity level for each case X_n fed into the neural network. (The mean is indicated by the red line). Here we use our best performing CNN mentioned earlier. For every case X_n fed into the MLP, regardless of class separation, average neuron activity in the hidden layer remained between 0.09 and 0.12. Some classes show slightly higher averages and standard deviation, but in general their distributions are very similar.



Class 4 average activity: 0.1011 Class 5 standard deviation: 0.0442	Class 5 average activity: 0.1041 Class 5 standard deviation: 0.0520
--	--

Figure 14: Histograms of average neuron states in hidden layer for best CNN

Class Responders

The histogram below shows the number of neurons in the hidden layer which most greatly respond to each class based on their average activity level when all data is fed into the model. Class 1 has the highest responders, followed by classes 5 & 2, which are nearly equal, followed by classes 3 & 4, which are also nearly equal.

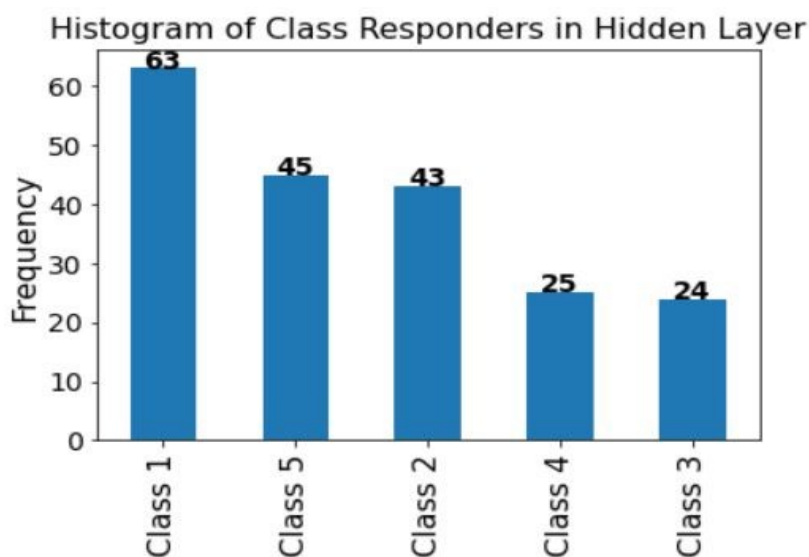


Figure 15: Class responders in hidden layer for best CNN

Attempts at Improvement

Many of the parameters in the model were adjusted in an effort to improve the model. We found that changing parameters did one of three things: decreased, increased, or had no effect on test set accuracy. Different adjustments and their effects are grouped below:

Generally has no effect on test set accuracy

Adjusting batch size (tried 50, 100 & 200)

Adjusting window size of first Conv2D layer (tried 3x3 & 7x7)

Generally decreases test set accuracy

Increasing window stride (tried 2x2 and 3x3)

Generally improves test set accuracy

Increasing the number of channels in both Conv2D layers (tried 24, 32, 40)

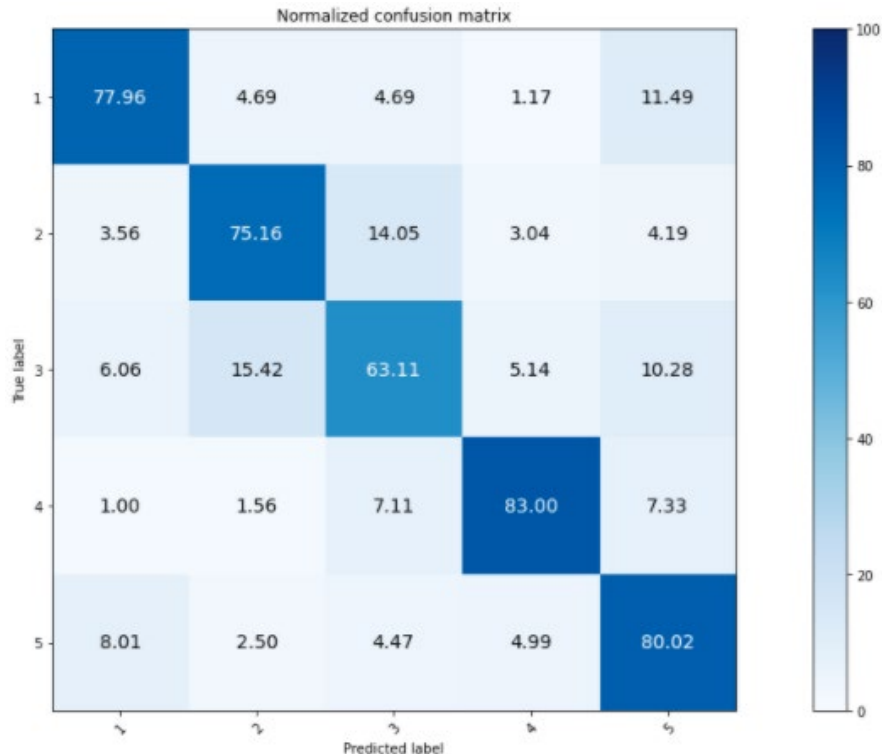
Removing the second Conv2D and maxpooling layers

Adding padding to Conv2D layer

Among these methods, increasing the Conv2D channels to 32 increased accuracy by about 3-4% on any given model, but only when done without padding. Increasing beyond 32 did not result in further improvement. Removing the second Conv2D and maxpooling layers increased accuracy by 1-2%, but only when done without padding. Adding padding increased accuracy by about 7%. It is discussed further below:

With our previous models, the convolutional layers did not have any padding, which resulted in the input image size consecutively getting smaller as it was processed through the hidden layer. However, in an attempt to see if the model would improve, when padding was applied, there was a major increase in model performance. With padding, the input image of a convolutional layer gets “padded” with zeros on the edges so that as the filter window moves through the image and produces new images, the new images are also the same size. With this method, the filter window puts more emphasis on the edges of the images compared to where normally without padding, the filter moves through the edges less frequently than pixels in the middle of the image. This can cause important information if any on the edges of the images to be recognized by the model. By applying padding and also using the 2x2 maxpooling layers, the flattened output produced by the convolutional layer contains 400 neurons. Which is in the same magnitude of size as the original 20x20 image.

We applied padding to our best performing model from our previous attempts which was the CNN consisting of the dropout layer with H=200. In this model, due to padding, the total number of weights and biases also increased significantly, totalling 83,941. Using the number of infos, we can calculate the infos per parameter ratio for this model to be 1.05. After training this model and using early stopping to avoid overfitting, the global accuracy of the testing set on this model was 76.28%, which was more than 7% accurate than our previous best model. The confusion matrix of the predictions made from this model is shown below.



Global Accuracy = 76.28%

Figure 16: Confusion Matrix of CNN (h=200) with dropout and with padding

In the testing set, the class 4 reported the highest accuracies (83%) followed by class 5 (80.02%), class 1 (77.96%) and class 2 (75.16%). Class 3 resulted in a lower accuracy (63.11%) than other classes but significantly improved by more than 13% than any other model. The major confusion resulting in misclassification was again between CL3 and CL2. Overall, the classification accuracy for all the classes were significantly better with more than 5% increase for all the classes, except for CL2, which increased by 2%.

Since the outputs of the 1st convolutional layer produces an image that is 20x20, we can visualize it to compare how different they look to the previous models convoluted image with no padding. The picture below shows the 16 channels of images.

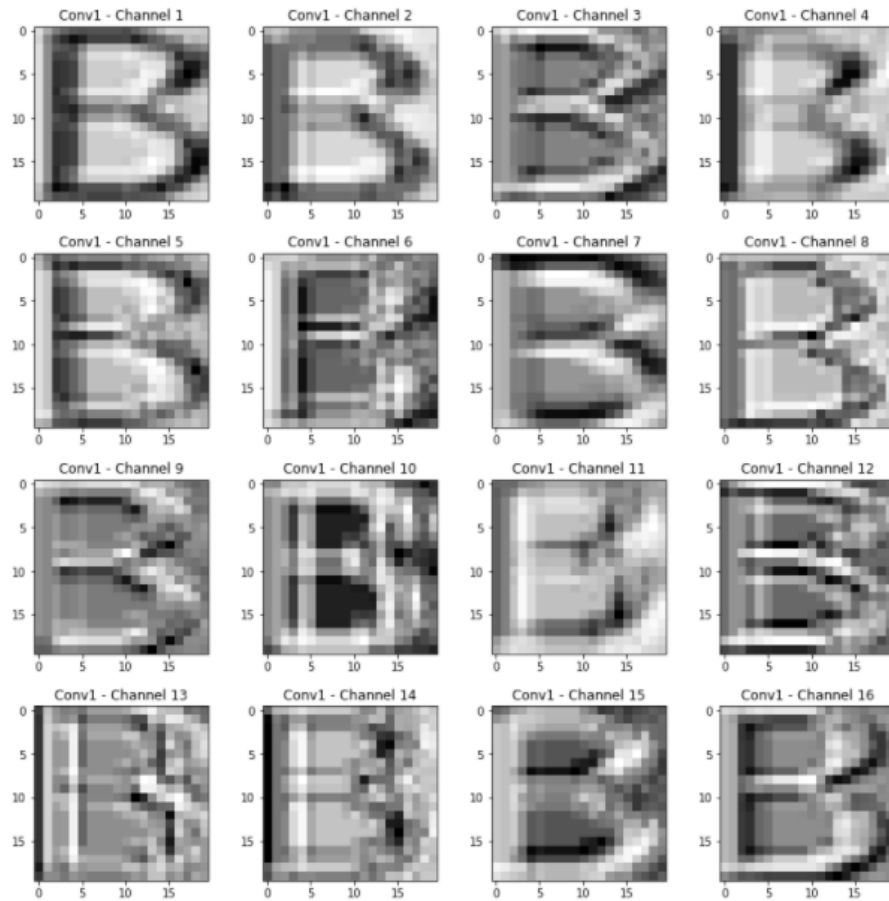


Figure 17: Visualization Example of Images produced in 16 channels of first convolutional layer with padding

It can be noticed when comparing these images (figure 17) to the convolutional images of the previous model (figure 12), that generally the images are less distorted and more clear. This perhaps was also a reason why the accuracy was much greater for this model. In addition, due to the padding, now the model can analyze the pixel intensities that are located on the edges more greatly. As an example, figure 18 shows the characteristics of B which are on the edges with prominence. The weights associated with each convolutional filter could have extracted more information for the model to learn, the characteristics of these images. The figure below (figure 18) shows the 16 output images/channels of the shape 10x10 after the second convolutional layer. Here too when compared to the images in figure 13, slight characteristics of the original image can be seen.

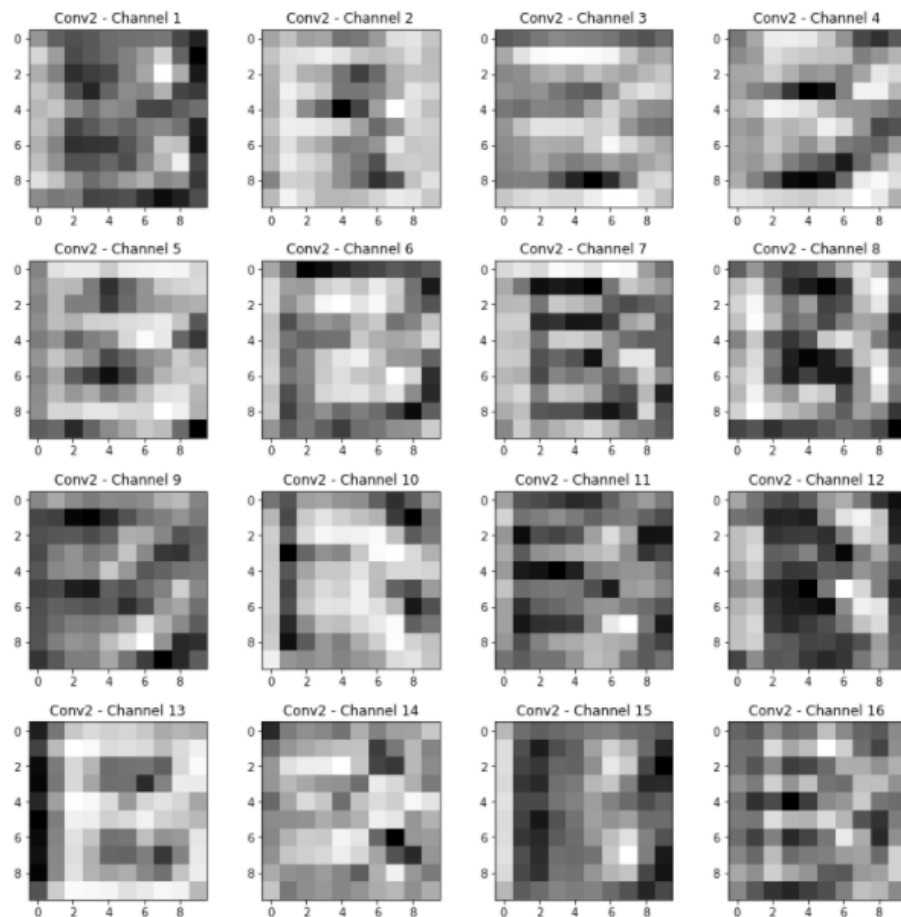
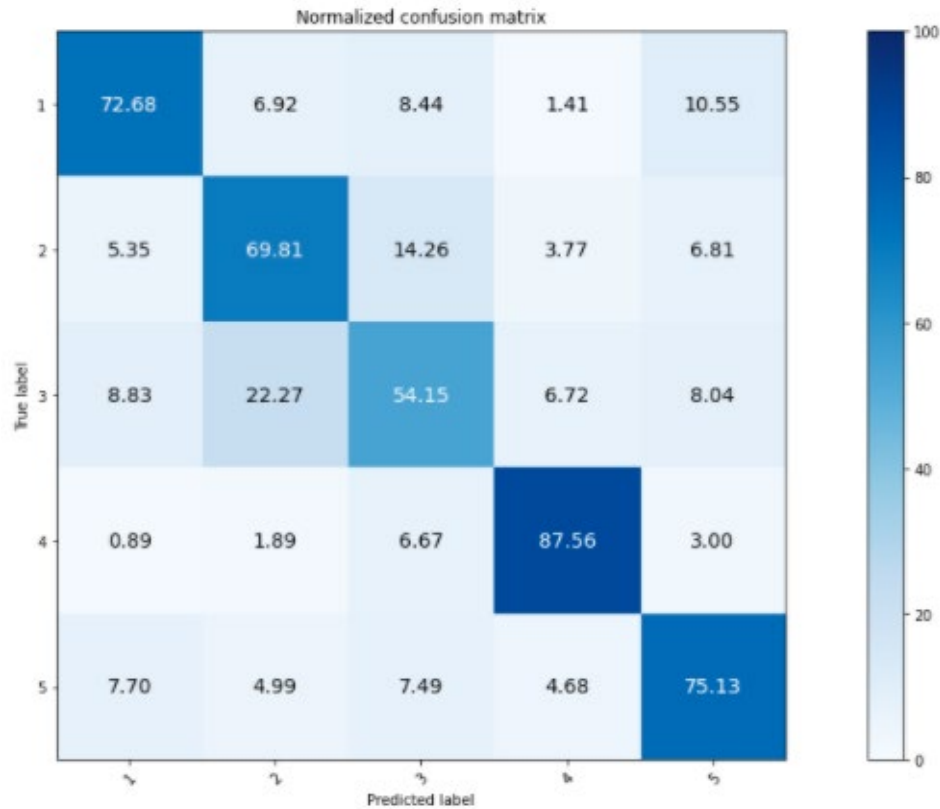


Figure 18: Visualization Example of Images produced in 16 channels of second convolutional layer with padding

We also wanted to see how this data would perform on an ordinary MLP with no convolutional layers. So we proceeded to train an MLP with one hidden layer of size 100, consisting of a relu response function. In total there were 36,545 weights and biases in the model. The performance of the model was actually quite surprising, with a global testing accuracy of 72.44% which is better than some of the CNN models that were discussed here before. The confusion matrix for the testing set is shown below in figure 19. One striking number that we can see is that CL4 has an accuracy of 87.56%. Although CL4 did have the highest accuracy in our CNN models as well, none had an accuracy as high as this MLP produced output.



Global Accuracy = 72.44%

Conclusion

In the initial experimentation with CNN model, simply testing the CNN model with different hidden layer sizes showed that the performance of the model improved with increasing size of hidden layer which is generally expected. The dropout option which is used to reduce overfitting helped to determine the optimal number of epochs which is used to train the model to evaluate the performance of CNN model.

In additional experimentation with CNN models we found that some parameters tend to have more significant impact on model performance than others. The most significant were the number of channels in the Conv2D layers and adding padding to the Conv2D layers. Adding channels improved model performance, however that performance was outmatched by adding padding which obviated additional channels beyond the original 16. We found that both increasing and decreasing the window size of the first Conv2D layer did not have much impact on model performance. Perhaps the image sizes are too small to justify using larger windows. Larger window sizes generally look for texture in images, which is an aspect that is less characteristic of character fonts than contour lines,

which are generally found with smaller window sizes. Increasing window stride hurt model performance, likely due to too much loss of image information.

Many of the characters across the fonts do not actually look remarkably different, which makes it surprising that a neural network can get even that accuracy that we achieved, which is not even high. Better performance can likely be achieved with larger and more varied images, where there are more aspects to differentiate between classes. A rapidly growing use of CNNs is satellite image analysis which is being used to perform various tasks such as predicting crop yields, recognizing climate events, or classifying planetary surface features.

The average computational time that the CNN models took to train are shown below. After running the first model with H=90, the average run time was 9 minutes and 11 seconds, so we chose to run the networks with a GPU accelerator in Google Colab's cloud-based environment. The resulting training time for the first model was 1 minute and 10 seconds, almost 8 times faster than running with no accelerator. This is due to a GPU's ability to excel in parallel computing. We then chose to run with the accelerator for all the models in this paper. The average training time for the models are shown below on the table.

Model	Average Run Time
CNN H=90 no Dropout (100 epochs) with Dropout	1.0 minute 10.9 seconds 1.0 minute 10.5 seconds
CNN H=150 no Dropout (100 epochs) with Dropout	1.0 minute 12.2 seconds 1.0 minute 12.8 seconds
CNN H=200 no Dropout (100 epochs) with Dropout	1.0 minute 12.5 seconds 1.0 minute 14.2 seconds

The training times for the models ran during the attempts for exploration and improvements are shown in the table below.

Model	Average Run Time
CNN H=200 consisting of dropout and padding (100 epochs)	0.0 minute 39.3 seconds
Regular MLP H=100 (100 epochs)	0.0 minute 59.2 seconds

Code Section

```
[1] # These are all the necessary imports
import pandas as pd
import numpy as np
import os
import time
import math
import random
import datetime
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from pandas import DataFrame as df
from numpy import linalg as LA
from matplotlib.ticker import MultipleLocator
from tensorflow import keras
from tensorflow.keras import layers, optimizers, losses, callbacks
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.utils import to_categorical
from scipy.stats import norm
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.utils.multiclass import unique_labels
```

Data Importing

```
[3] # Connecting this colab doc to the shared drive
# Login: gdk6373@gmail.com
# Password: Math6373
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
[4] # Reads in the font files
font1 = pd.read_csv('/content/drive/MyDrive/MATH6373/HW3/COURIER.csv')
font2 = pd.read_csv('/content/drive/MyDrive/MATH6373/HW3/CALIBRI.csv')
font3 = pd.read_csv('/content/drive/MyDrive/MATH6373/HW3/LUCIDA.csv')
font4 = pd.read_csv('/content/drive/MyDrive/MATH6373/HW3/SITKA.csv')
font5 = pd.read_csv('/content/drive/MyDrive/MATH6373/HW3/TIMES.csv')
```

```
[5] # Drops the unnecessary features of the font data sets. Keeps only "font", "strength", "italic" and the pixel gray scales
keep = [0,3,4] + list(range(12,412))
font1 = font1.iloc[:,keep]
font2 = font2.iloc[:,keep]
font3 = font3.iloc[:,keep]
font4 = font4.iloc[:,keep]
font5 = font5.iloc[:,keep]

# Filters the fonts to keep only the non-bold and non-italic characters
font1 = font1[(font1['strength']==0.4) & (font1['italic']==0)]
font2 = font2[(font2['strength']==0.4) & (font2['italic']==0)]
font3 = font3[(font3['strength']==0.4) & (font3['italic']==0)]
font4 = font4[(font4['strength']==0.4) & (font4['italic']==0)]
font5 = font5[(font5['strength']==0.4) & (font5['italic']==0)]
```

Exploring Pixel Images

```
[ ] # Creates the X portion of each font and reshapes them to 20x20.
font1_XX = font1.iloc[:,3:].to_numpy().reshape(font1.shape[0], 20, 20)
font2_XX = font2.iloc[:,3:].to_numpy().reshape(font2.shape[0], 20, 20)
font3_XX = font3.iloc[:,3:].to_numpy().reshape(font3.shape[0], 20, 20)
font4_XX = font4.iloc[:,3:].to_numpy().reshape(font4.shape[0], 20, 20)
font5_XX = font5.iloc[:,3:].to_numpy().reshape(font5.shape[0], 20, 20)

[ ] # Exploring font images (small view, 150 images at a time)

box = font1_XX # change to font1_XX, font2_XX, etc. to view different fonts
cox = 0 # change the index.
dox = 30
for i in range(dox):
    if i == 0:
        fox1 = np.concatenate((box[cox], box[(cox+1)], box[(cox+2)], box[(cox+3)], box[(cox+4)]))
    else:
        fox2 = np.concatenate((box[(cox+i*5)], box[(cox+i*5+1)], box[(cox+i*5+2)], box[(cox+i*5+3)], box[(cox+i*5+4)]))
        fox1 = np.hstack((fox1, fox2))

plt.figure(figsize = (100,4))
plt.imshow((fox1), cmap='Greys', interpolation='nearest')
plt.show
```

Creating X & Y Test & Train sets

```
[8] # Creates the X portion of each font and reshapes them to 1x20x20. This is done so that the X input is the right shape for
font1_X = font1.iloc[:,3:].to_numpy().reshape(font1.shape[0], 20, 20, 1)
font2_X = font2.iloc[:,3:].to_numpy().reshape(font2.shape[0], 20, 20, 1)
font3_X = font3.iloc[:,3:].to_numpy().reshape(font3.shape[0], 20, 20, 1)
font4_X = font4.iloc[:,3:].to_numpy().reshape(font4.shape[0], 20, 20, 1)
font5_X = font5.iloc[:,3:].to_numpy().reshape(font5.shape[0], 20, 20, 1)

# Creates the Y portion of each font. No need to do one hot encoding.
#font1_Y = np.array([[1,0,0,0,0]]*font1_X.shape[0])
#font2_Y = np.array([[0,1,0,0,0]]*font2_X.shape[0])
#font3_Y = np.array([[0,0,1,0,0]]*font3_X.shape[0])
#font4_Y = np.array([[0,0,0,1,0]]*font4_X.shape[0])
#font5_Y = np.array([[0,0,0,0,1]]*font5_X.shape[0])

# Creating Y portion of each font Weiquang's way. Do one hot encoding after making Ytrain & Ytest
font1_Y = np.array([[0]]*font1_X.shape[0])
font2_Y = np.array([[1]]*font2_X.shape[0])
font3_Y = np.array([[2]]*font3_X.shape[0])
font4_Y = np.array([[3]]*font4_X.shape[0])
font5_Y = np.array([[4]]*font5_X.shape[0])

# Creating the TRAIN and TEST sets for each font
font1_Ytrain, font1_Ytest, font1_Xtrain, font1_Xtest = train_test_split(font1_Y, font1_X, train_size=0.8, random_state= 7)
font2_Ytrain, font2_Ytest, font2_Xtrain, font2_Xtest = train_test_split(font2_Y, font2_X, train_size=0.8, random_state= 7)
font3_Ytrain, font3_Ytest, font3_Xtrain, font3_Xtest = train_test_split(font3_Y, font3_X, train_size=0.8, random_state= 7)
font4_Ytrain, font4_Ytest, font4_Xtrain, font4_Xtest = train_test_split(font4_Y, font4_X, train_size=0.8, random_state= 7)
font5_Ytrain, font5_Ytest, font5_Xtrain, font5_Xtest = train_test_split(font5_Y, font5_X, train_size=0.8, random_state= 7)
```

```

# Creating the final TRAIN and TEST sets
Xtrain = np.vstack((font1_Xtrain, font2_Xtrain, font3_Xtrain, font4_Xtrain, font5_Xtrain))
Xtest = np.vstack((font1_Xtest, font2_Xtest, font3_Xtest, font4_Xtest, font5_Xtest))
Ytrain = np.concatenate((font1_Ytrain, font2_Ytrain, font3_Ytrain, font4_Ytrain, font5_Ytrain))
Ytest = np.concatenate((font1_Ytest, font2_Ytest, font3_Ytest, font4_Ytest, font5_Ytest))

Ytest = to_categorical(Ytest, 5)
Ytrain = to_categorical(Ytrain, 5)

# Normalizing the X data
Xtrain=Xtrain/255
Xtest=Xtest/255

# Changing the dtype to 'uint8' because that's what it is in Weiquang's example.
Xtrain = Xtrain.astype('uint8')
Xtest = Xtest.astype('uint8')

print(Xtrain.shape, " ", Xtrain.dtype, " ", type(Xtrain))
print(Xtest.shape, " ", Xtest.dtype, " ", type(Xtest))
print(Ytrain.shape, " ", Ytrain.dtype, " ", type(Ytrain))
print(Ytest.shape, " ", Ytest.dtype, " ", type(Ytest))

```

Built in functions

```

[9] # Function to train NN, returns dictionary with the Monitor history, model summary and model runtime.
# If model_save is 'yes' then dictionary also saves the model of the last epoch.
def train_model(Xtrain, Ytrain,
                turn_dropout_on = 'no',
                H_value = 90,
                epoch = 100,
                save_model = ['no', 'model_name'],
                batch = round((Xtrain.shape[0])**0.5)):

    model_monitor = {}
    ##### Creating the model #####
    model = Sequential()
    # Conv1
    model.add(Conv2D(32, (5, 5), padding='valid', input_shape= Xtrain.shape[1:])) # Number of paramters = ((5x5)+1)*16
    model.add(Activation('relu'))
    # Maxpool1
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # Conv2
    model.add(Conv2D(32, (3, 3), padding='valid')) # Number of paramters = ((3x3)*16+1)*16
    model.add(Activation('relu'))
    # Maxpool2
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # Flatten
    model.add(Flatten())
    # Hidden
    model.add(Dense(H_value))
    if turn_dropout_on == 'yes':
        model.add(Dropout(0.5)) # Dropout Layer, turn on to use
    model.add(Activation('relu'))
    # Output
    model.add(Dense(5))
    model.add(Activation('softmax'))

```



```

model_monitor['model_summary'] = model.summary()

#### Compiling the model ####
opt = Adam(lr=0.001, decay=1e-7)

model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

# checkpointer = ModelCheckpoint(filepath='/content/model_1', monitor='val_accuracy', save_best_only=True)

#### Running the model to check losses for stopping ####
start = time.process_time()
with tf.device('/gpu:0'):
    Monitor = model.fit(Xtrain, Ytrain,
                        batch_size=batch,
                        epochs=epoch,
                        validation_data=(Xtest, Ytest),
                        # callbacks = [checker],
                        shuffle = True)
    if save_model[0] == 'yes':
        model.save(save_model[1])
        model_monitor['model'] = load_model(save_model[1])
model_time = (time.process_time() - start)

#### Adding to the dictionary the model history and run time. ####
model_monitor['run_time'] = (f"Time to run model with H = {H_value}: {model_time//60} minutes {model_time%60} seconds")
model_monitor['History'] = Monitor.history

return(model_monitor)

```

```

[10] # Plot the loss and acc plot
def plot_loss_and_acc(train_loss, val_loss, train_acc, val_acc, title1 = 'C.E. Loss Plot', title2 = 'Accuracy Plot'):
    fig = plt.figure(figsize= (15,4))
    ax1 = fig.add_subplot(1,2,1)
    # Train & Test Loss plot
    plt.plot(range(1,(len(train_loss)+1)), train_loss, color='dodgerblue', linewidth=1, markersize=6) # Train Loss
    plt.plot(range(1,(len(train_loss)+1)), val_loss, color='red', linewidth=1, markersize=6) # Test Loss
    plt.xlabel('epoch #')
    plt.ylabel('Cross-Entropy Value')
    plt.title(title1)
    plt.legend(['Train Loss', 'Test Loss'])

    ax1 = fig.add_subplot(1,2,2)
    # Train & Test Accuracy plot
    plt.plot(range(1,(len(train_acc)+1)), train_acc, color='dodgerblue', linewidth=1, markersize=6) # Train Loss
    plt.plot(range(1,(len(train_acc)+1)), val_acc, color='red', linewidth=1, markersize=6) # Test Loss
    plt.xlabel('epoch #')
    plt.ylabel('Accuracy Value')
    plt.title(title2)
    plt.legend(['Train Acc', 'Test Acc']);

```

```

[11] # Function to plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, classes,
                           normalize=False,
                           title=None,
                           cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
        cm = (cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]) * 100
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    fig, ax = plt.subplots(figsize=(16,8))
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap, vmin=0, vmax=100)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
          yticks=np.arange(cm.shape[0]),
          # ... and label them with the respective list entries
          xticklabels=classes, yticklabels=classes,
          title=title,
          ylabel='True label',
          xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
              rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt), size = 14,
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax

```

```
[12] def confmat_percent(confmat):
    conf_percent = (confmat.astype('float') / confmat.sum(axis=1)[:, np.newaxis])*100
    conf_percent_df= pd.DataFrame(conf_percent)
    conf_percent_df.columns=['CL1','CL2','CL3','CL4','CL5']
    conf_percent_df.index= ['CL1','CL2','CL3','CL4','CL5']
    return conf_percent_df

def confmat_df(confmat):
    conf_df= pd.DataFrame(confmat)
    conf_df.columns=['CL1','CL2','CL3','CL4','CL5']
    conf_df.index= ['CL1','CL2','CL3','CL4','CL5']
    conf_df['Sum']= confmat.sum(axis=1)
    return conf_df

[13] def confidence_interval(cv,confmat):
    cv= cv
    Ntest_CL1= confmat.loc['CL1']['Sum']      #sum of test CL1 values
    Ntest_CL2= confmat.loc['CL2']['Sum']      #sum of test CL2 values
    Ntest_CL3= confmat.loc['CL3']['Sum']      #sum of test CL3 values
    Ntest_CL4= confmat.loc['CL4']['Sum']
    Ntest_CL5= confmat.loc['CL5']['Sum']
    ptest_CL1= (confmat.loc['CL1']['CL1'])/Ntest_CL1
    ptest_CL2= (confmat.loc['CL2']['CL2'])/Ntest_CL2
    ptest_CL3= (confmat.loc['CL3']['CL3'])/Ntest_CL3
    ptest_CL4= (confmat.loc['CL4']['CL4'])/Ntest_CL4
    ptest_CL5= (confmat.loc['CL5']['CL5'])/Ntest_CL5

    #CI for CL1
    sigma_test_CL1= math.sqrt(ptest_CL1*(1-ptest_CL1)/Ntest_CL1)
    lower_test_CL1= (ptest_CL1-(cv*sigma_test_CL1))*100
    upper_test_CL1= (ptest_CL1 + (cv*sigma_test_CL1))*100
    #CI for CL2
    sigma_test_CL2= math.sqrt(ptest_CL2*(1-ptest_CL2)/Ntest_CL2)
    lower_test_CL2= (ptest_CL2-(cv*sigma_test_CL2))*100
    upper_test_CL2= (ptest_CL2 + (cv*sigma_test_CL2))*100

    #CI for CL3
    sigma_test_CL3= math.sqrt(ptest_CL3*(1-ptest_CL3)/Ntest_CL3)
    lower_test_CL3= (ptest_CL3-(cv*sigma_test_CL3))*100
    upper_test_CL3= (ptest_CL3 + (cv*sigma_test_CL3))*100

    #CI for CL4
    sigma_test_CL4= math.sqrt(ptest_CL4*(1-ptest_CL4)/Ntest_CL4)
    lower_test_CL4= (ptest_CL4-(cv*sigma_test_CL4))*100
    upper_test_CL4= (ptest_CL4 + (cv*sigma_test_CL4))*100

    #CI for CL5
    sigma_test_CL5= math.sqrt(ptest_CL5*(1-ptest_CL5)/Ntest_CL5)
    lower_test_CL5= (ptest_CL5-(cv*sigma_test_CL5))*100
    upper_test_CL5= (ptest_CL5 + (cv*sigma_test_CL5))*100
```

```

# Overall CI
N1= confmat['Sum'].sum()
diag_sum= np.trace(confmat)
p1= diag_sum/N1
sigma_p1= math.sqrt(p1*(1-p1)/N1)
lower_limit= (p1-(cv*sigma_p1))*100
upper_limit= (p1+(cv*sigma_p1))*100

CI_data= {'Class':['CL1','CL2','CL3','CL4','CL5','Overall'],
          'Lower Limit':[lower_test_CL1, lower_test_CL2, lower_test_CL3, lower_test_CL4, lower_test_CL5, lower_limit],
          'Upper Limit': [upper_test_CL1, upper_test_CL2, upper_test_CL3, upper_test_CL4, upper_test_CL5, upper_limit]}

CI_table= pd.DataFrame(CI_data, columns=['Class','Lower Limit','Upper Limit'])

return(CI_table.round(2))

```

```

[14] from IPython.display import display_html
from itertools import chain,cycle
def display_side_by_side(*args,titles=cycle([''])):
    html_str=''
    for df,title in zip(args, chain(titles,cycle(['<br>']))):
        html_str+=" <td style='vertical-align:top'>"         html_str+=f"<h2>{title}</h2>"         html_str+=df.to_html().replace('table','table style="display:inline"')         html_str+=" |
```

```

[15] # Function for Hidden Layer Analysis

# Explanation of arguments
# "my_cases" is a set of training cases. Can be Xtrain or font1_Xtrain/255, etc.
# "blurb" is just a print statement saying what classes have been used.
# "model_to_use" is self explanatory
# "layer_to_use" is an integer representing the nth layer in the model. Assumes zero indexing.
def HLA(my_cases, model_to_use, blurb='For...', layer_to_use = 9):
    model = model_to_use
    for i in range(0,(layer_to_use+1)):
        if i == 0:
            aa = model.layers[i](my_cases.astype('float'))
        else:
            aa = model.layers[i](aa)
    ACTn = np.mean(aa.numpy(), axis=1)
    avact = round(sum(ACTn)/len(ACTn), 4)
    per_lhalf = round((sum(ACTn < avact/2)/len(ACTn))*100, 4)
    per_lthird = round((sum(ACTn < avact/3)/len(ACTn))*100, 4)
    print(blurb)
    print(" The average activity of the neurons in the hidden layer is " + str(avact))
    print(" per(1/2): " + str(per_lhalf))
    print(" per(1/3): " + str(per_lthird))
    print("\n")
    print(ACTn.shape)
    print(my_cases.shape)

```

```

# Plots the histogram of ACTn
fig = plt.figure()
plt.figure(figsize=(9,5))
plt.hist(ACTn, bins=list(np.array(list(range(0,20)))/50))
plt.axvline(x=avact, color = 'red')
plt.ylabel('Frequency', fontsize=16)
plt.xlabel('Activity Level', fontsize=16)
plt.title('Histogram of Average Neuron States in Hidden Layer')

```

```
[16] def Class_Responder(my_classes, model, layer_to_use = 9):
    # my_classes = a tuple of your class fonts
    # layer_to_use = an interger representing the hidden layer in your model. assume zero indexing.
    my_h = model.layers[layer_to_use].output_shape[1]
    n = len(my_classes)
    def ACTN_CLASS(CLX, LAYER = layer_to_use): # returns the mean of the hidden layer neuron activities for cases fed in.
        for i in range(0, (LAYER+1)):
            if i == 0:
                aa = model.layers[i](CLX.astype('float'))
            else:
                aa = model.layers[i](aa)
        return np.mean(aa.numpy(), axis=0)

    for i in range(n):
        if i == 0:
            goober = ACTN_CLASS(my_classes[i])
        else:
            goober = np.vstack((goober, ACTN_CLASS(my_classes[i])))

    Responder = [] # A list indicating which class causes the most "activation" in each neuron.
    for i in range(my_h):
        fox = np.where(goober[:,i]==np.max(goober[:,i]))[0][0] + 1
        Responder.append(fox)

    # Creates a data frame of the class responders
    L1, L2 = [], []
    for i in range(n):
        L1.append(('Class ' + str(i+1)))
        L2.append(sum(np.array(Responder)==(i+1)))
    kooky = pd.DataFrame({'category': L1, 'number': L2})
    kooky.set_index('category', inplace=True)
    kooky.sort_values('number', inplace=True, ascending=False)

    # Plots the histogram of class responders
    fig = plt.figure()
    plt.figure(figsize=(7,5))
    kooky.plot(y='number', kind='bar', legend=False)
    plt.ylabel('Frequency', fontsize=16)
    plt.title('Histogram of Class Responders in Hidden Layer', fontsize=16)
    plt.xticks(fontsize = 16)
    plt.yticks(fontsize = 14)
    for i in range(n):
        plt.text((i-0.15), kooky.iloc[i], str(kooky.iloc[i]['number']), color='black', fontweight='bold', fontsize = 14)
```

Training and testing for 3 H values

```
[17] # Use GPU to accelerate the training time
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

No dropout layer training

```
[ ] model = train_model(Xtrain, Ytrain,
                        turn_dropout_on = 'no',
                        H_value = 90,
                        epoch = 100,
                        save_model = ['no', 'model_name'],
                        batch = round((Xtrain.shape[0])**0.5))

[ ] plot_loss_and_acc(model['History']['loss'], model['History']['val_loss'],
                    model['History']['accuracy'], model['History']['val_accuracy'],
                    title1 = 'C.E. Loss Plot (H=90)',
                    title2 = 'Accuracy Plot (H=90)')
```

Dropout layer training

```
[ ] model = train_model(Xtrain, Ytrain,
                        turn_dropout_on = 'yes',
                        H_value = 90,
                        epoch = 100,
                        save_model = ['no', 'model_name'],
                        batch = round((Xtrain.shape[0])**0.5))

[ ] plot_loss_and_acc(model['History']['loss'], model['History']['val_loss'],
                    model['History']['accuracy'], model['History']['val_accuracy'],
                    title1 = 'C.E. Loss Plot (H=90) - With Dropout',
                    title2 = 'Accuracy Plot (H=90) - With Dropout')

[ ] model1['model'].evaluate(Xtest, Ytest)
    y_pred_test = model1['model'].predict(Xtest)
    predlabel_test = np.argmax(y_pred_test, axis=1)
    Ytest_true = np.argmax(Ytest, axis=1)
    classes = np.asarray(['1', '2', '3', '4', '5'])

    cm_h90 = confusion_matrix(Ytest_true, predlabel_test); print(cm_h90)

[ ] plot_confusion_matrix(Ytest_true, predlabel_test, classes=classes, normalize=True,
                        title='Normalized confusion matrix')

    plt.show()
    print('Global acc is:', round(accuracy_score(Ytest_true, predlabel_test)*100, 3))
```

```
1 convlayer1 = model200['model'].layers[0](Xtest.astype('float32')) # Had to convert the input to float32 for this to work.
2 # convlayer1 = model200.layers[0](Xtest.astype('float32')) # Had to convert the input to float32 for this to work.
```

```
1 fig, axes = plt.subplots(4, 4, figsize=(14, 14))
2 for i, x, y in zip(range(16), np.concatenate([([i]*4) for i in [0, 1, 2, 3]], axis=0), list(range(4))*4):
3     axes[x, y].imshow(convlayer1[2489, :, :, i], cmap='Greys', interpolation='nearest')
4     axes[x, y].set_title(f'Conv1 - Channel {i+1}')
5     plt.subplots_adjust(hspace=0.3)
6 plt.show()
```

```
1 # Getting more states of the layers
2 activation_l1 = model200['model'].layers[1](convlayer1)
3 maxpool_l1 = model200['model'].layers[2](activation_l1)
4 convlayer2 = model200['model'].layers[3](maxpool_l1)
5 activation_l2 = model200['model'].layers[4](convlayer2)
6 maxpool_l2 = model200['model'].layers[5](activation_l2)
7 flatten_layer = model200['model'].layers[6](maxpool_l2)
```

```
1 fig, axes = plt.subplots(4, 4, figsize=(14, 14))
2 for i, x, y in zip(range(16), np.concatenate([([i]*4) for i in [0, 1, 2, 3]], axis=0), list(range(4))*4):
3     axes[x, y].imshow(convlayer2[2489, :, :, i], cmap='Greys', interpolation='nearest')
4     axes[x, y].set_title(f'Conv2 - Channel {i+1}')
5     plt.subplots_adjust(hspace=0.3)
6 plt.show()
```