Optical Character Recognition of the Sorani Dialect of the Kurdish Language Using Deep Convolutional Neural Networks

A capstone project

presented to

Dr. Bruno R. Andriamanalimanana,

Department of Computer Information Science,

College of Engineering,

State University of New York Polytechnic Institute,

Utica, New York

In partial fulfillment

of the requirements of the

Bachelor of Science Degree

in

Computer & Information Science

By:

Zachary Aaron Clark

May 2020

Declaration

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

[illegible]

Zachary Aaron Clark

SUNY Polytechnic Institute

Department of Computer Information Science

Approved and recommended for acceptance
as a capstone project in partial fulfillment of the requirements
for the degree of Bachelor of Science in
Computer & Information Science

____
DATE

____
Dr. Bruno R. Andriamanalimanana
Capstone Advisor

Abstract

Through thorough testing and trials, a neural network model was created that is able to identify which of 29 letters in the Sorani Kurdish alphabet is depicted in a 32x32 black and white image. This neural network is able to perform this task with a greater than 90% accuracy and a loss of only approximately 0.3195. Through rigorous testing by modifying independent variables individually, it was discovered that, for this particular problem, Convolutional layers generally perform much better than standard Dense layers, although Dense layers are still useful as a supplement to Convolutional layers. It was also determined that dropout rate has only a small effect on the accuracy and loss of the neural network unless the rate is very high or very low. Some other minor observations include: Max Pooling layers seem to be largely ineffective for this type of problem; deeper and more complex neural networks are usually, but not always, more effective at categorizing these images, and the “relu” activation, “categorical crossentropy” loss function, and “RMSprop” optimizer all appear to be the most effective for this problem.

Acknowledgments

A special thanks in no particular order to:

Ethan Gray, John Harlan, Elizabeth Green, and Mackenzie Dahlem for keeping me sane.

Aya Hamdi and Zulfa Rachmaheni for helping me identify Sorani Kurdish letters.

Susan Mahoney-Clark for proofreading my paper.

https://sourceforge.net/projects/bulkimageconver/, https://bulkresizephotos.com/en, and http://www.highmotionsoftware.com/products/imbatch for making it much easier to format images for my dataset.

And

3Blue1Brown (Grant Sanderson) and CGP Grey for explaining neural networks and machine learning in an intuitive way.

Table of Contents

List of Figures

Chapter 1:
Introduction

1.1: Optical Character Recognition

Optical Character Recognition (OCR) is the process of identifying characters or numerals of a particular language using a software program. The best OCR programs are able to differentiate between the different characters using one of various machine learning algorithms, including the genetic breeding model, regression, Q learning, and neural networks.

1.2: Neural Networks

Neural networks are a subset of machine learning that can be trained to perform a particular task, in this case, identifying characters. A neural network can almost be thought of like a function, except instead of a few inputs, maybe dozens of internal variables, and a single output, a neural network can have hundreds or thousands of inputs, with hundreds of thousands or millions of internal variables, and as many outputs as there are desired solutions.[1] Each internal variable is like a neuron, and all the neurons are arranged into layers. Each neuron is connected to all the neurons of the layer above it and to all of the neurons in the layer below it. There are three primary ways in which a neural network can be taught to perform a task: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the neural network is given a dataset with labels on each element that includes the desired outcome. The neural network makes its best guess given the input and is graded against the labeled correct answers. Unsupervised learning allows the neural network to learn from unlabeled data and is useful in situations where labeled data is difficult to acquire or when the designer does not know the exact desired outcome. In reinforcement learning, the neural network

[1] Sanderson, Grant, director. *But What Is a Neural Network? | Deep Learning, Chapter 1*. *Youtube*, 3Blue1Brown, 5 Oct. 2017, www.youtube.com/watch?v=aircAruvnKk.

is trying to find the best solution, rather than the correct solution and is reinforced the closer it gets to an optimal solution.[2]

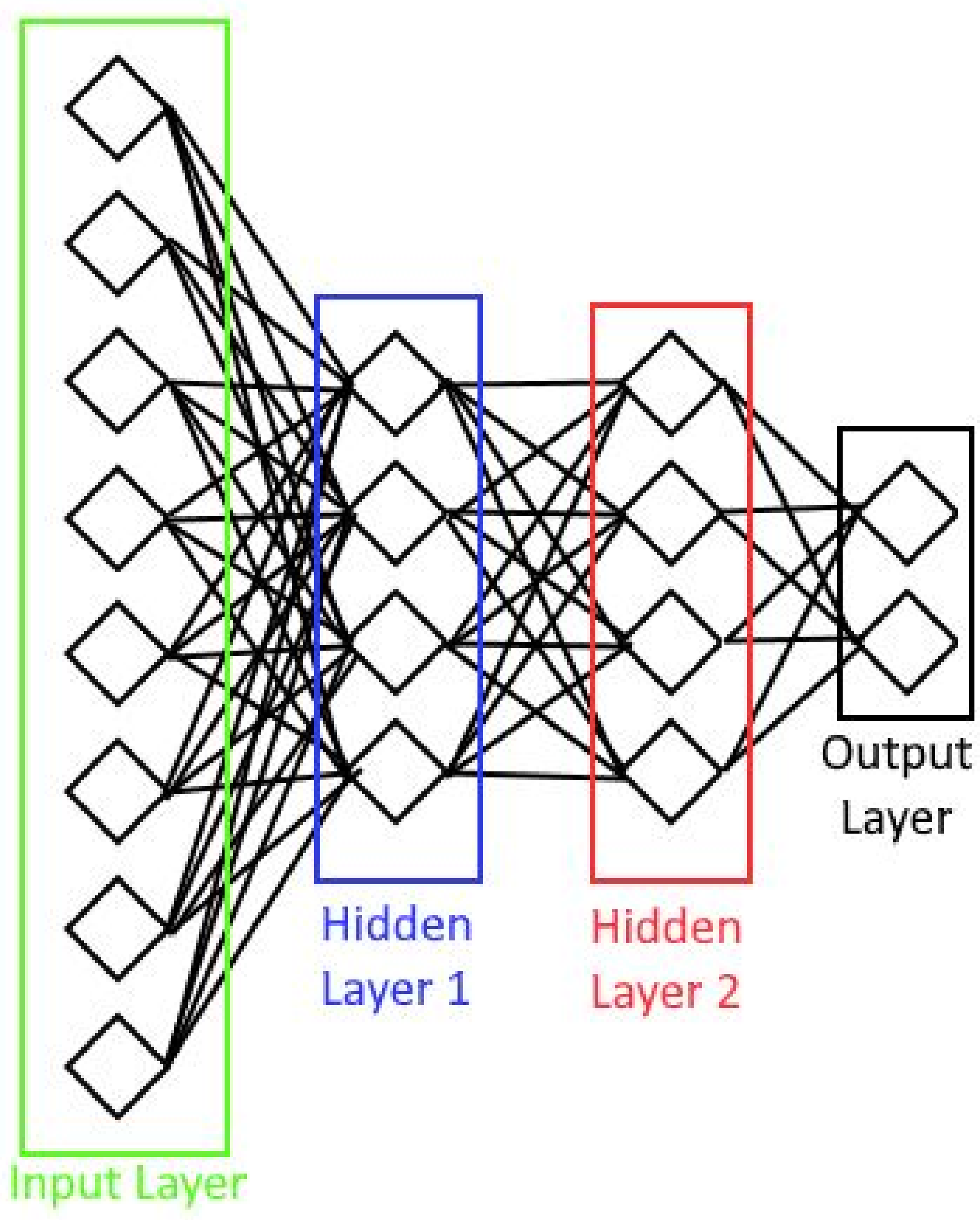


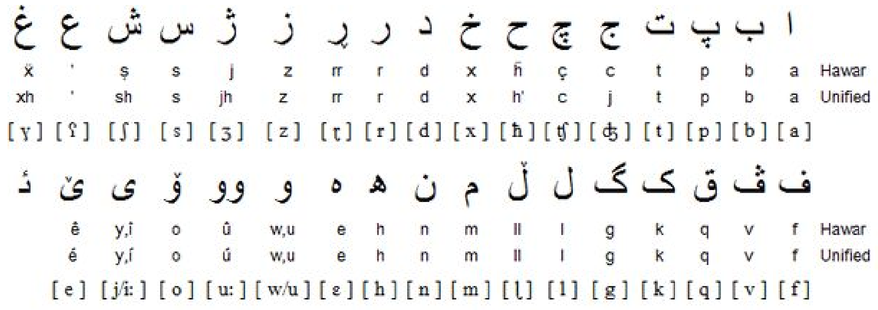
Fig. 1.2: A basic neural network showing different layers and connections between layers

[2] Salian, Isha. "SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?" *The Official NVIDIA Blog*, Nvidia, 20 Aug. 2019, blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/.

Chapter 2:
Background

2.1: Sorani Kurdish

Sorani is a dialect of Kurdish spoken primarily in Northern Iraq. Like many other languages in the region, Sorani has adopted the Arabic script as its writing system, albeit in a modified form. While Arabic uses this script as an abjad, it only writes consonants and implies vowel sounds with diacritical marks, Sorani uses the script as a true alphabet, writing both consonants and vowels. Most of Sorani's 33 (or 34, depending on what is counted as a letter) letters are borrowed from the Arabic script, but a significant portion are also borrowed from the Persian script, itself a modified Arabic script, and a few letters are entirely unique. Since the Sorani writing system is based on the Arabic script, it is written exclusively in cursive, as such each letter has several forms, one each for when the letter is: at the beginning of a word, in the middle of a word, at the end of a word, and on its own.[3]



غ	ع	ش	س	ژ	ز	ڕ	ر	د	خ	ح	چ	ج	ت	پ	ب	ا	
ẍ	'	ş	s	j	z	rr	r	d	x	ḧ	ç	c	t	p	b	a	Hawar
xh	'	sh	s	jh	z	rr	r	d	x	h'	c	j	t	p	b	a	Unified
[ɣ]	[ʕ]	[ʃ]	[s]	[ʒ]	[z]	[ɽ]	[r]	[d]	[x]	[ħ]	[ʧ]	[ʤ]	[t]	[p]	[b]	[a]	
ئ	ێ	ی	ۆ	وو	و	ە	ه	ن	م	ڵ	ل	گ	ک	ق	ڤ	ف	
	ê	y,î	o	û	w,u	e	h	n	m	ll	l	g	k	q	v	f	Hawar
	é	y,í	o	ú	w,u	e	h	n	m	ll	l	g	k	q	v	f	Unified
	[e]	[j/iː]	[o]	[uː]	[w/u]	[ɛ]	[h]	[n]	[m]	[ɫ]	[l]	[g]	[k]	[q]	[v]	[f]	

Fig. 2.1[4]: The full Sorani Kurdish Alphabet

2.2: Dataset

Due to Sorani being a relatively obscure dialect of a small language in a region of the world not known for high-tech industry, gathering a complete dataset of the script is an

---

[3] Hassanpour, A. "History of Kurdish Orthography." *Kurdish Academy*, Kurdish Academy of Language, 1992, kurdishacademy.org/?p=233.

[4] "Sorani Alphabet." *Parsendow Translation*, Sharepoint, parsendowtranslation.sharepoint.com/siteimages/sorani%20alphabet.png.

extremely challenging task. A task so difficult, that this project was only created on a dataset of 29 out of the 33/34 letters in the writing system, and only in their isolated forms. The dataset includes 500 32x32 black and white PNG images of each character and is divided randomly into a training set, a validation set, and a testing set of 300, 100, and 100 samples respectively. This data was gathered from two sources. The characters that are borrowed from the Arabic script were taken from the "ahcd1" dataset, publically available under the Open Database Contents License[5]. The characters borrowed from Persian were taken from the "Comprehensive Database for Offline Persian Handwritten Recognition" dataset, which is available under a custom license that lasts three years upon request, for research purposes.[6] The data from the latter set was modified to resize the image, convert from grayscale to black and white, and convert from the TIFF file format to the PNG file format. This project was unable to acquire data for handwriting recognition of the letters unique to Sorani Kurdish or any of the characters in forms other than the isolated form.

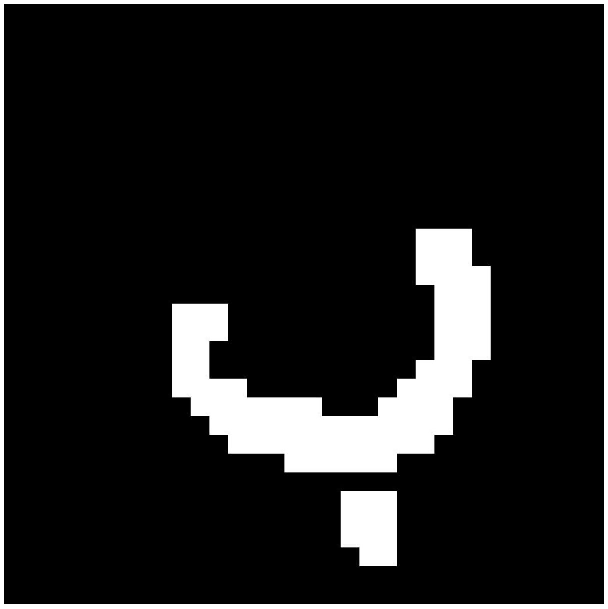


Fig. 2.2[5]: A typical image in the dataset, taken from the training data

[5] Loey, Mohamed. "Arabic Handwritten Characters Dataset." *Kaggle*, 22 June 2017, www.kaggle.com/mloey1/ahcd1.

[6] Sadri, Javad, et al. "A Novel Comprehensive Database for Offline Persian Handwriting Recognition." *Pattern Recognition*, vol. 60, Dec. 2016, pp. 378–393., doi:10.1016/j.patcog.2016.03.024.

2.3: Setup and Dependencies

To run this project an environment must be set up in Anaconda using Python 3.6.x, as well as the latest versions of TensorFlow for Nvidia GPUs, Keras, Pillow, Pydot, and Graphviz. TensorFlow is an API that allows for the creation of neural networks on GPUs and requires CUDA version 10.1 as well as the latest version of cuDNN. Keras is a high-level library that runs on top of TensorFlow, and Pillow is an image processing library used by it. Pydot and Graphviz are used to optionally generate an image of the model. In order to run the program, the program and the “data” folder must be located in the same folder within the properly set up Anaconda environment.

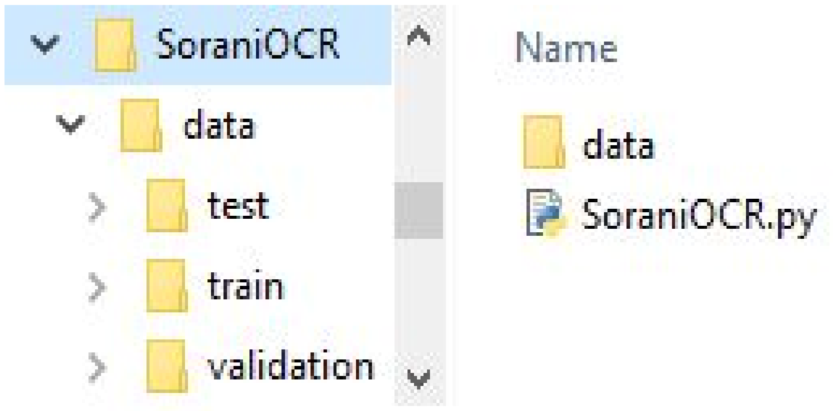


Fig. 2.3: A screenshot showing the correct directory layout within the Anaconda environment

Chapter 3:
Design

3.1: Layer Types

Keras has several different layer types, not all of which would be considered a layer of a neural network. Some such layers are the “Flatten” and “Dropout” layers. These layers perform operations on the neural network itself, rather than on the data to arrive at an answer. While these layer types are invaluable for efficiency, it is the other type of layers, the ones that would constitute their own layers of a neural network, that will receive the most testing. Some such layers include: Dense layers, Convolutional layers, Pooling layers, and Recurrent layers.[7]

Dense layers are a standard type of neural network layer in which each neuron in the layer is connected to each neuron in the layers directly above and below it, and which generally perform matrix multiplication with the input and a calculated bias.[7]

Convolutional layers are generally connected to the layers directly adjacent to them in the same way as Dense layers, except instead of performing matrix multiplication, they perform mathematical convolution on the input. Convolutional layers may or may not use biases to assist in the calculations.[7]

Pooling layers are separated into two types, Max Pooling and Average Pooling, and are connected to surrounding layers in the same way as Dense and Convolutional layers but instead, downsample the input to allow for more efficient feature extraction.[7]

Recurrent layers are any layer type that, rather than outputting only to the layer below them, are able to output back to themselves or to a layer above them.[7]

3.2: Neural Network Design

The program that creates and tests the neural network starts by loading in the data from the training, validation, and testing sets. It then specifies the type of neural network, in this case,

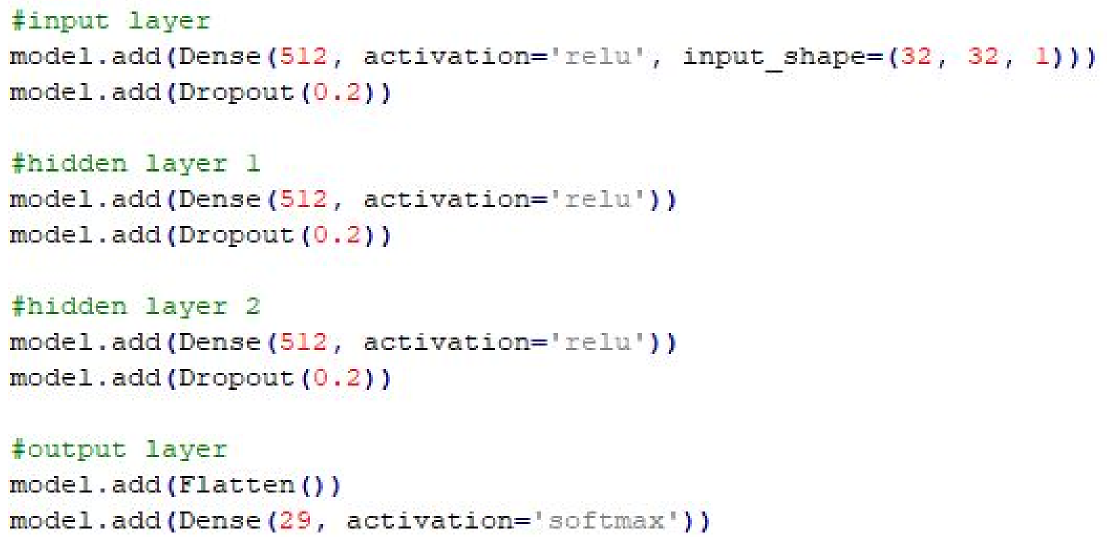
[7] “Keras: The Python Deep Learning Library.” *Home - Keras Documentation*, keras.io/.

a sequential neural network, before defining the different layers that will make up the network. It then compiles the neural network so that it can be given data to process. Next, it begins training the neural network by having it process each input tensor into one of the 29 different outputs. It then grades each epoch using the validation dataset to check for overfitting. Training is complete after the model runs through the training data for the specified number of epochs, at which time, the program will begin to test the neural network using the testing dataset and report back its accuracy and loss.

Chapter 4:
Implementation

4.1: Overview

In order to most accurately measure the effects of the variable being tested, the neural network used in each test will be created as described in section 3.2: Neural Network Design, with the layers as shown in Fig. 4.1. The only changes to the layers will be those that are relevant to the variable being tested and only the hidden layers will be modified. To strike a balance between training time and effectiveness, each test is run for ten epochs. Each configuration of layers will be tested three times, and the average of those scores will be compared to the average scores of the other configurations. The baseline average accuracy and loss are 0.58207 and 1.28235 respectably.

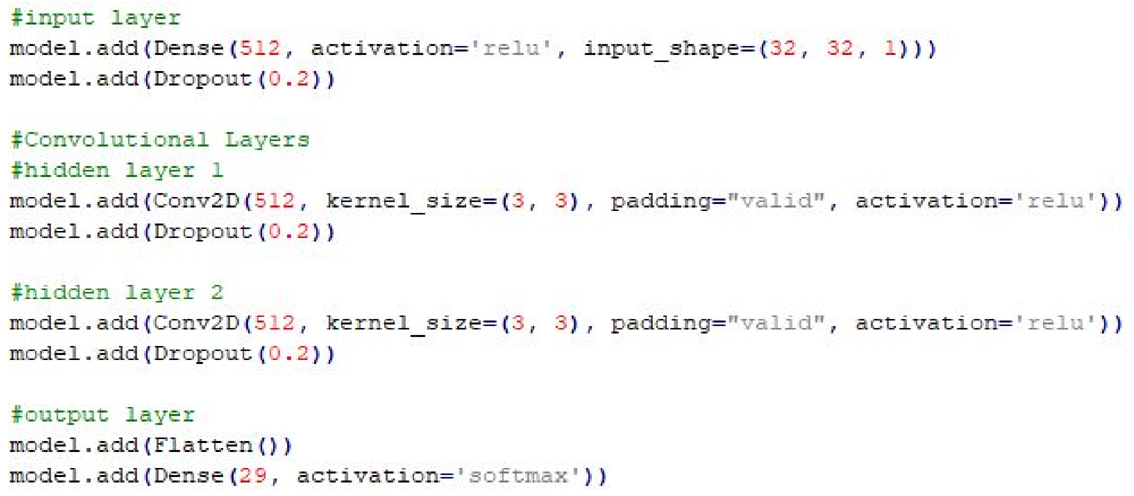


```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.1: The default layers that will be modified for testing

4.2: Layer Type Testing

This phase of testing compared Dense, Convolutional, and Max Pooling layers against each other. The Dense layers were configured as in Fig. 4.1; the Convolutional layers were configured with 512 filters and a filter size of 3x3; and the Max Pooling layers were set to downsample by a factor of two each layer. An example of this is shown in Fig. 4.2a below. The data from this test clearly show that Convolutional layers are most effective for this problem. The results for all three layer types can be seen below in Fig. 4.2b.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#Convolutional Layers
#hidden layer 1
model.add(Conv2D(512, kernel_size=(3, 3), padding="valid", activation='relu'))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Conv2D(512, kernel_size=(3, 3), padding="valid", activation='relu'))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.2a: The layers as used for the basic Convolutional testing

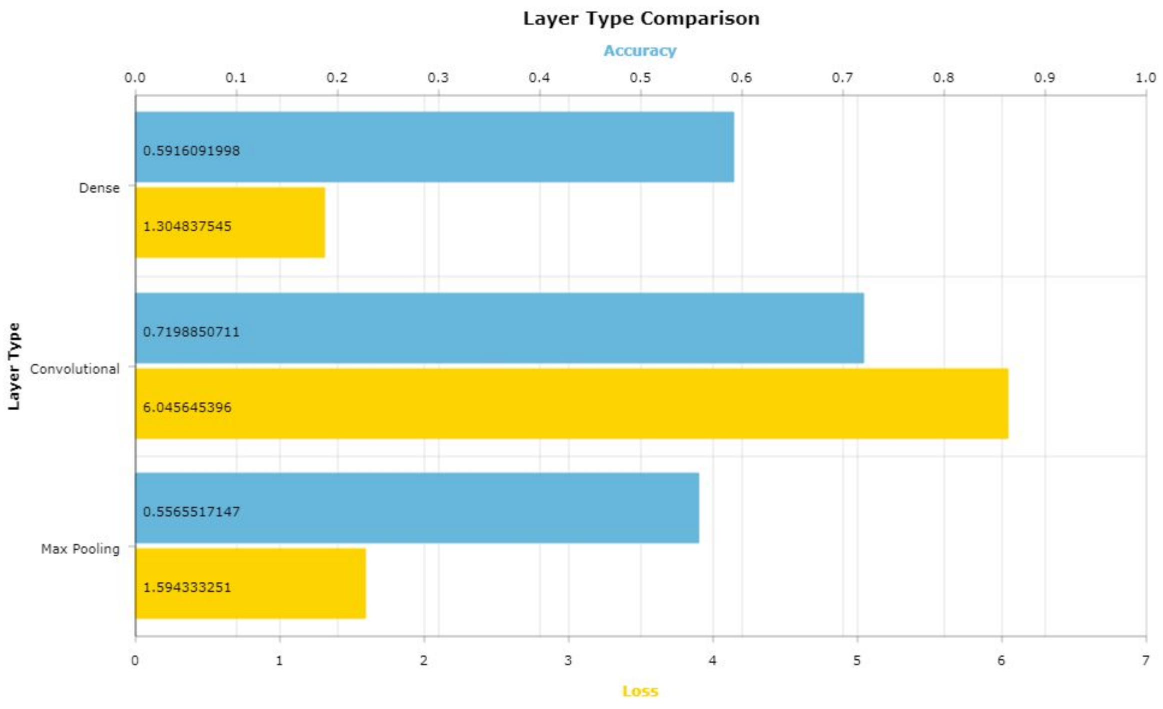
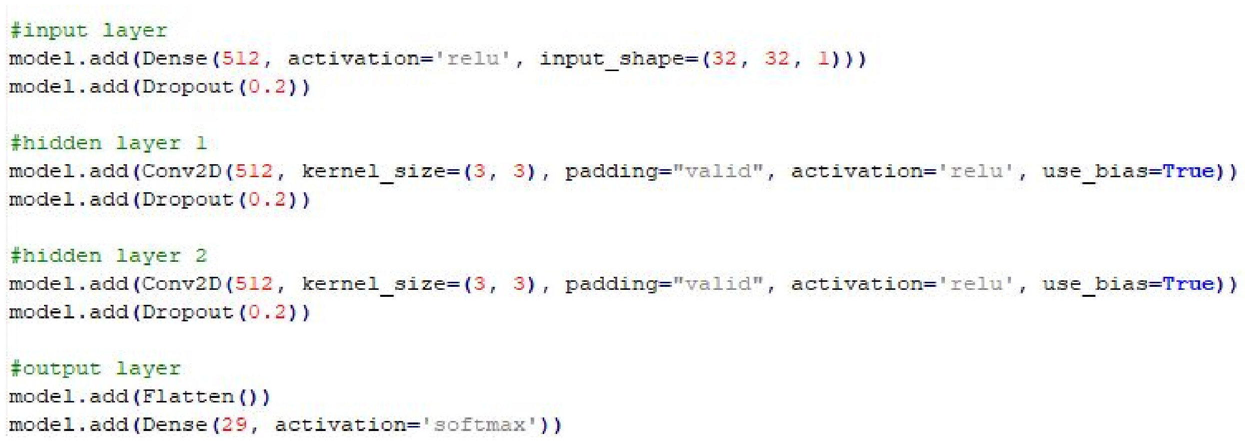


Fig. 4.2b: Graph showing the results of layer type testing

4.3: Biased Filtering Testing

The next phase of testing compared Convolutional layers with bias against Convolutional layers without bias. The coded Convolutional layers with bias can be seen in Fig. 4.3a. From this testing, it is determined that bias has little or no effect on accuracy, but significantly lowers loss for this particular problem. The full results can be seen in Fig. 4.3b.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Conv2D(512, kernel_size=(3, 3), padding="valid", activation='relu', use_bias=True))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Conv2D(512, kernel_size=(3, 3), padding="valid", activation='relu', use_bias=True))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.3a: The layers as used for the basic Convolutional testing with bias

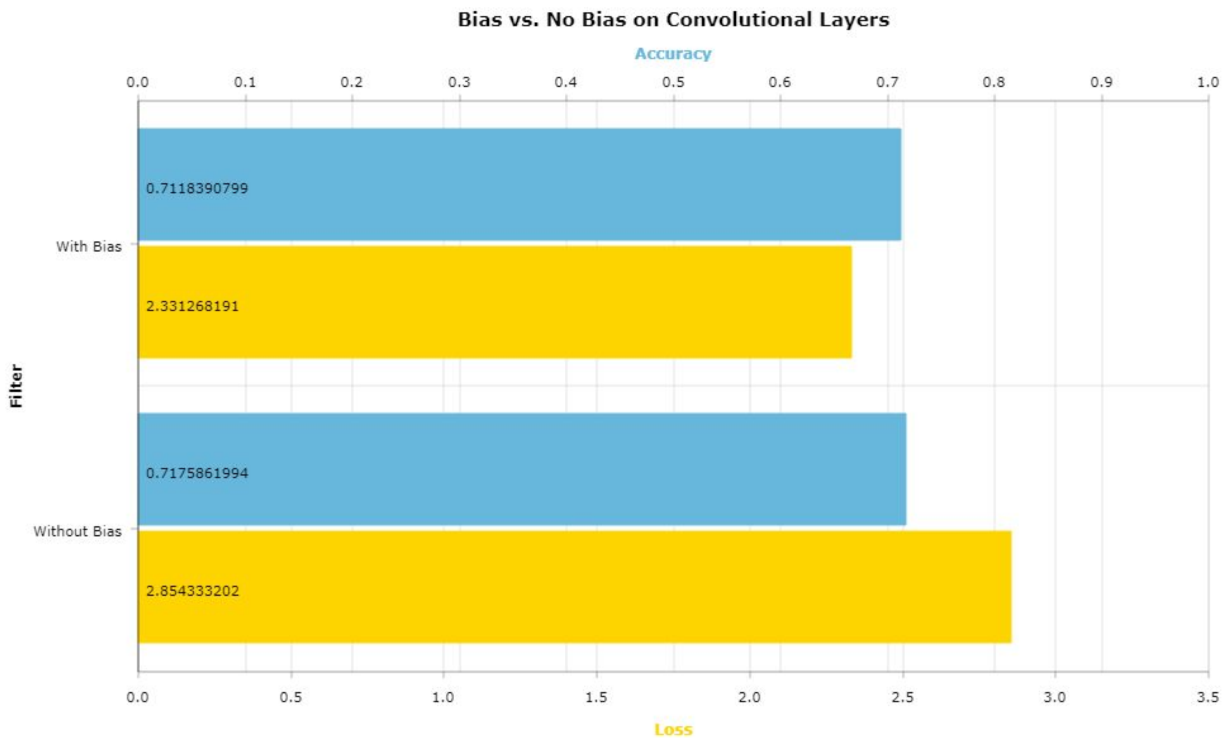
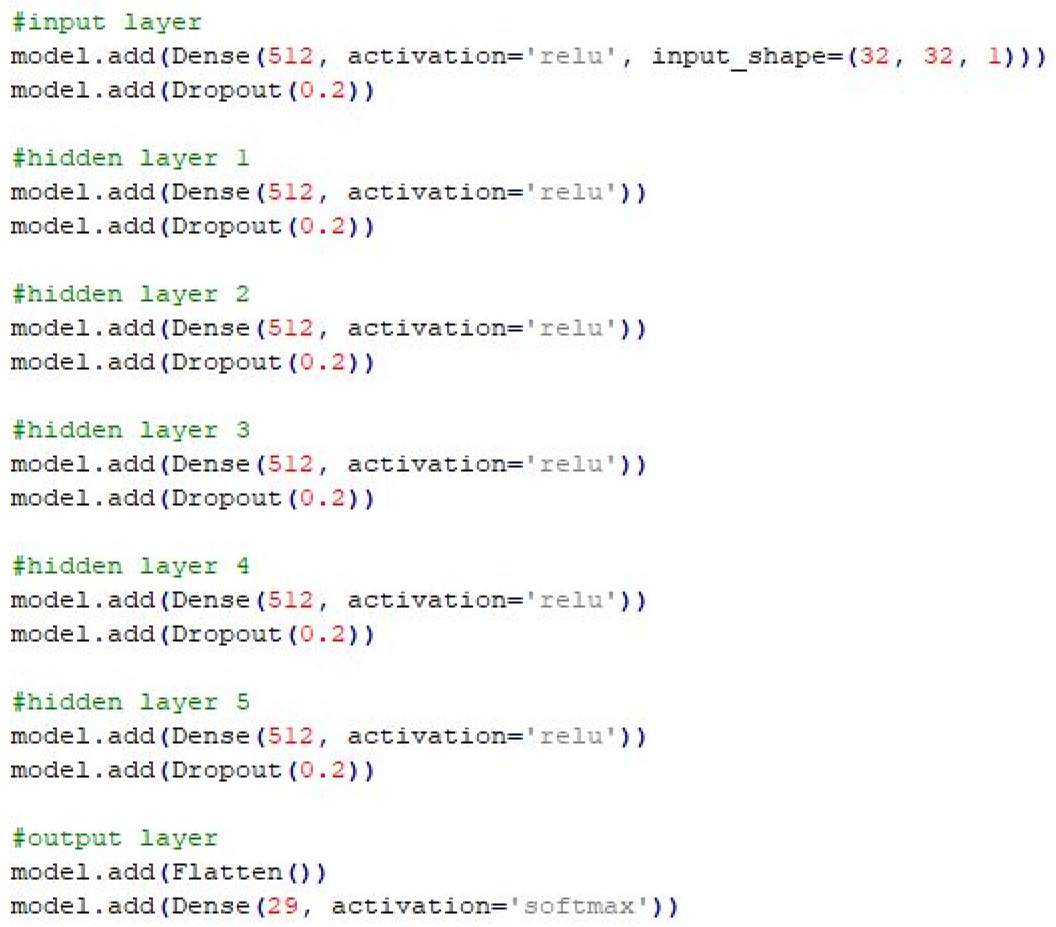


Fig. 4.3b: Graph of results comparing Convolutional layers with and without bias

4.4: Layer Amount Testing

In this phase, the effect of the number of hidden layers was tested. Dense, Convolutional, and Max Pooling layer types were all tested in amounts from zero hidden layers to five hidden layers. An example of the layers is shown in Fig. 4.4a. Adding additional Dense hidden layers seems to have little impact on the accuracy or loss of the neural network. Every additional Convolutional layer added greatly increased the accuracy, and after rising for the first few layers, also decreased the loss. Each Max Pooling layer added after the first one decreased the accuracy and increased loss, to the point that with five hidden Max Pooling layers, the accuracy was at pure chance. The full results of each layer type can be seen in Figs. 4.4b-d.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 3
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 4
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 5
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.4a: The layers as used for testing the number of hidden Dense layers

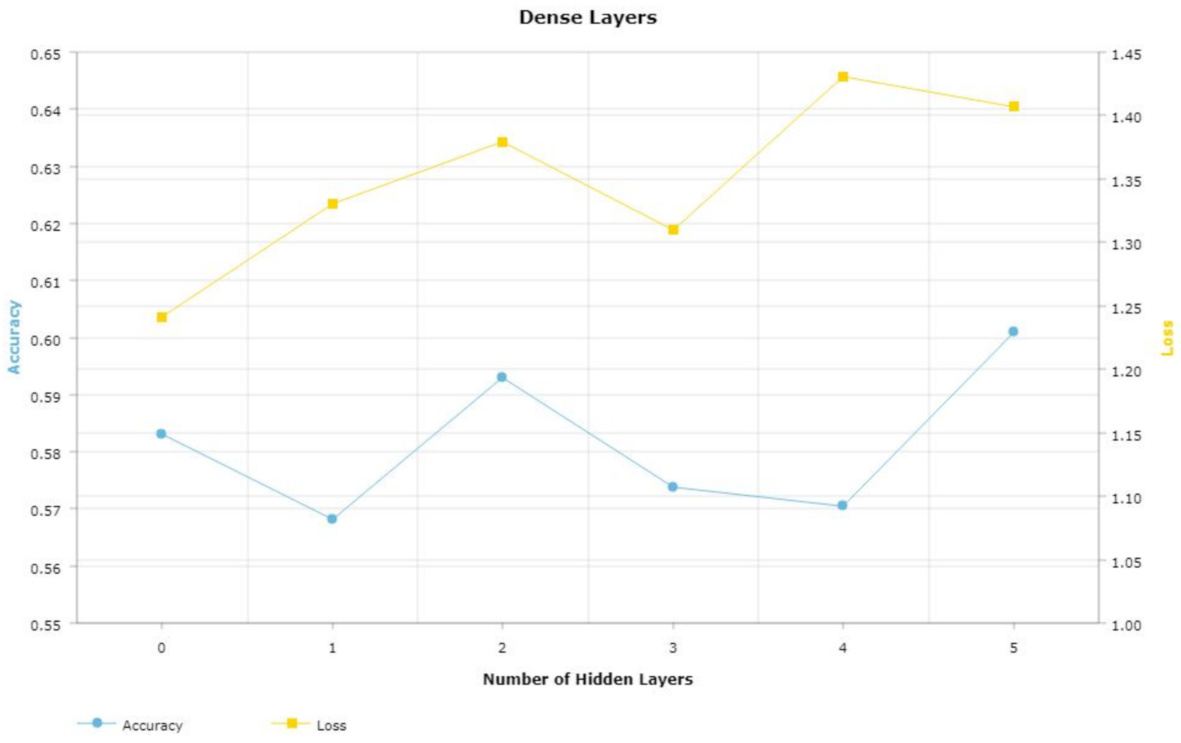


Fig. 4.4b: Graph of results comparing the effects of different amounts of Dense layers

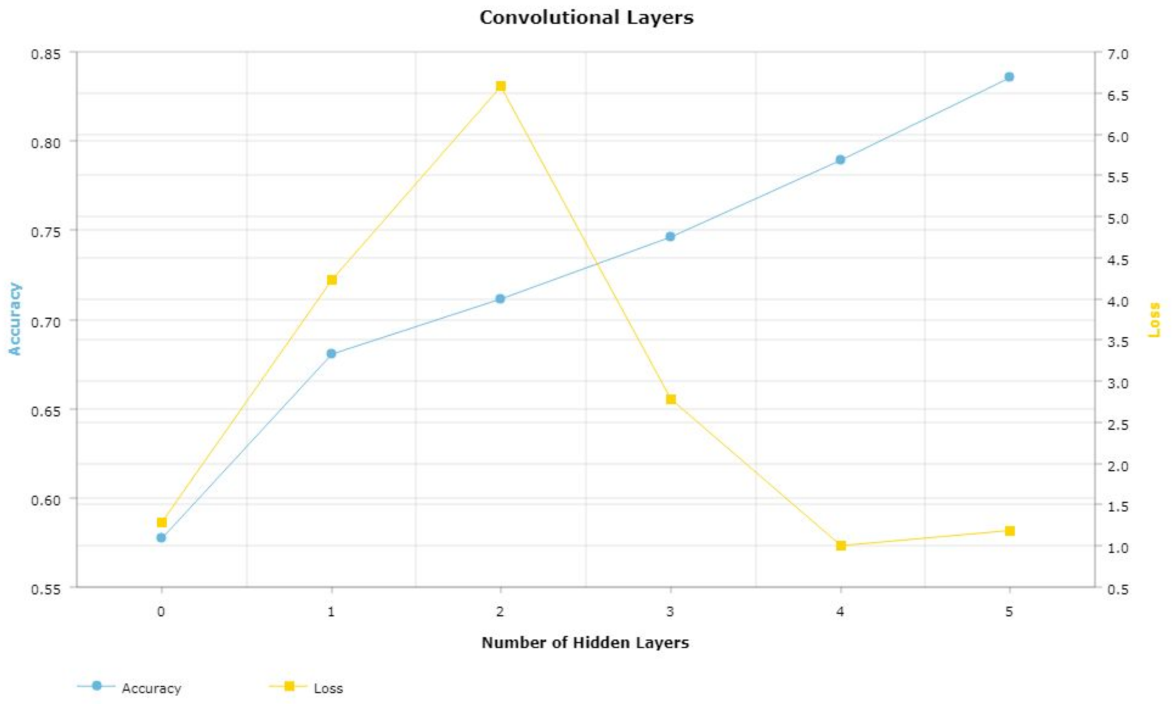


Fig. 4.4c: Graph of results comparing the effects of different amounts of Convolutional layers

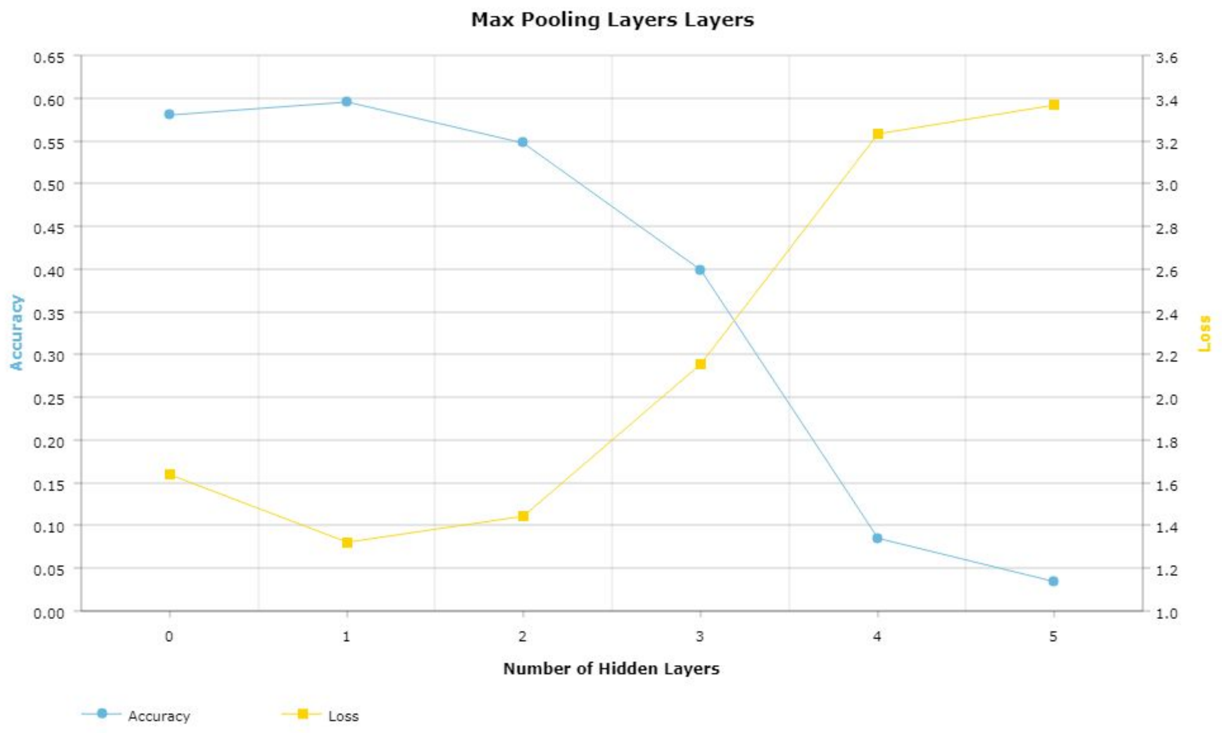
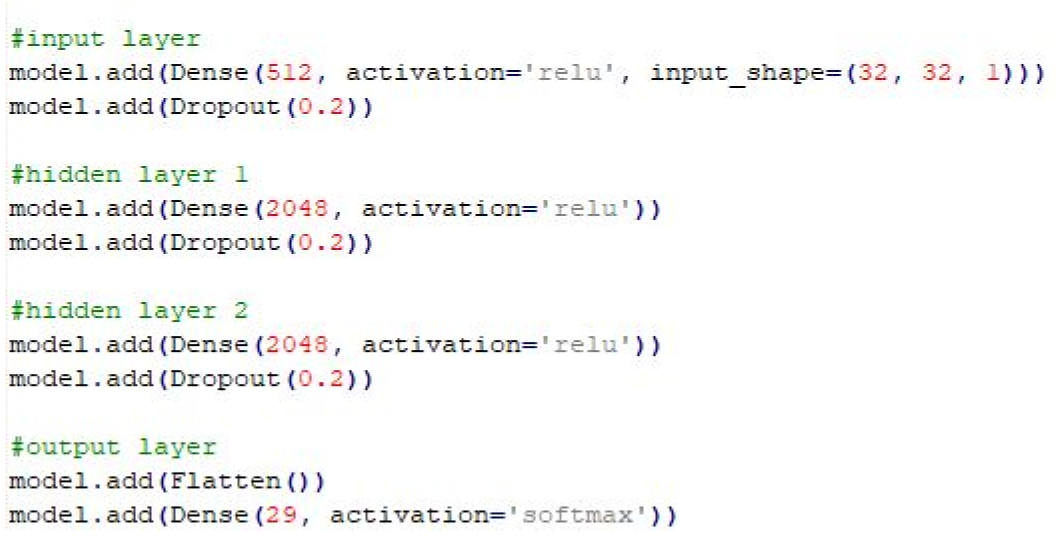


Fig. 4.4d: Graph of results comparing the effects of different amounts of Max Pooling layers

4.5: Layer "Size" Testing

This section of testing was used to determine the effects of changing the "size" of each type of layer. The term "size" is used loosely, as what that means for each layer type is different. For Dense layers, the size refers to the number of units used to create each layer; for Convolutional layers, there are two "sizes" that were tested, the first is the number of filters used and the second is the size of the filters; and for Max pooling layers, the "size" refers to the rate at which they downsample the image. An example of the layers used for this testing is shown in Fig. 4.5a. Dense layers had accuracy peaks and loss valleys at 16 and 512 units; Convolutional layers accuracy peaks at 64 and 1024 filters, and with a filter size of 9x9; and Max Pooling layers performed worse in accuracy and loss for every increase in size. The full results for each "size" test are shown in Figs. 4.5b-e.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.5a: The layers as used for testing the size of hidden Dense layers

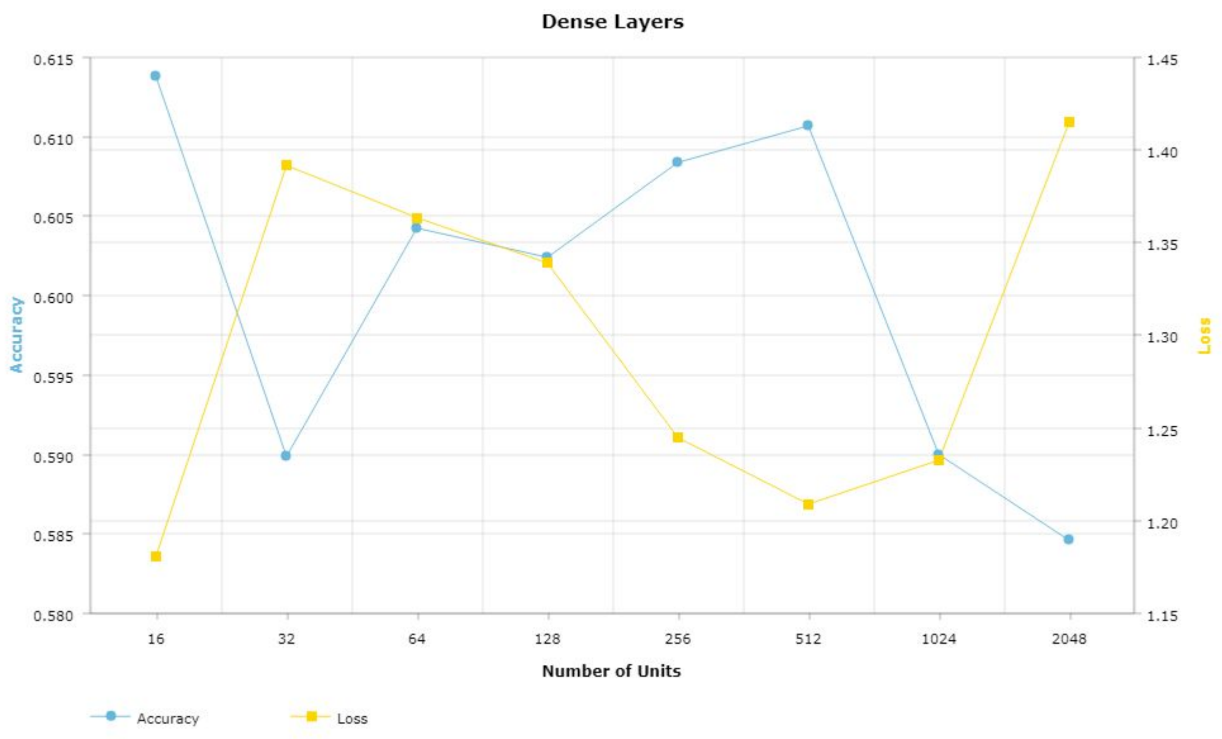


Fig. 4.5b: Graph of results comparing the effects of different sizes of Dense layers

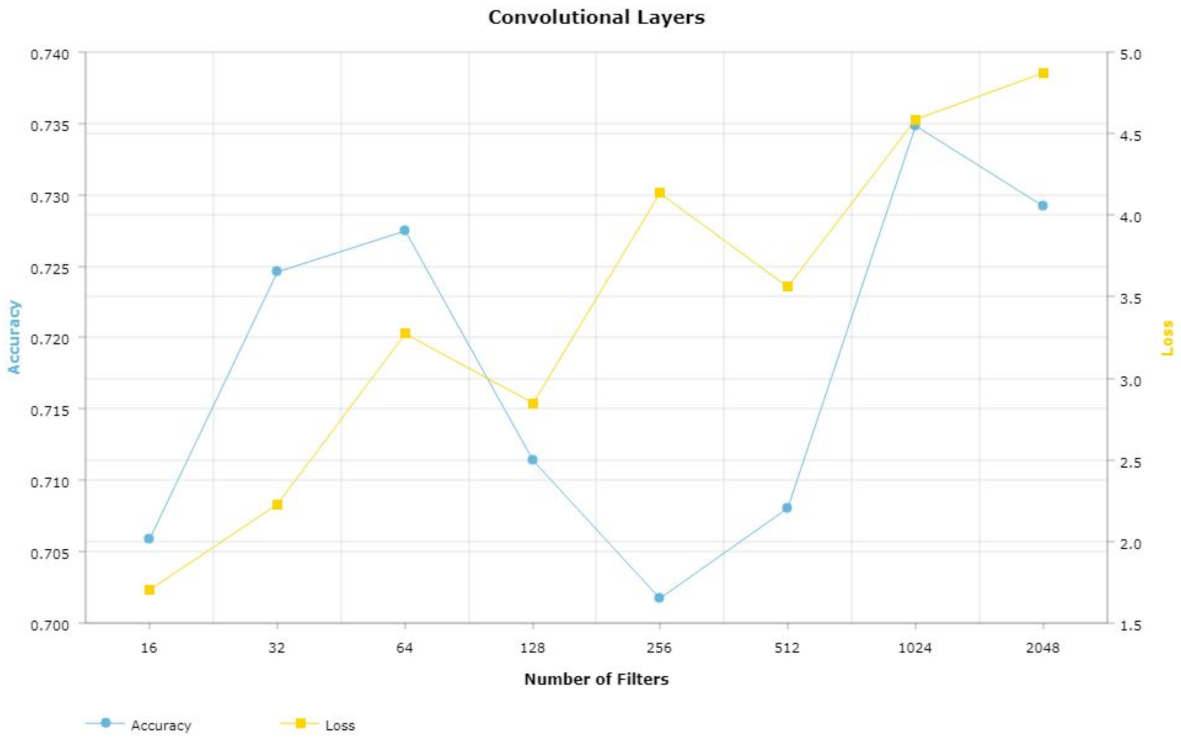


Fig. 4.5c: Graph of results comparing the effects of different numbers of filters

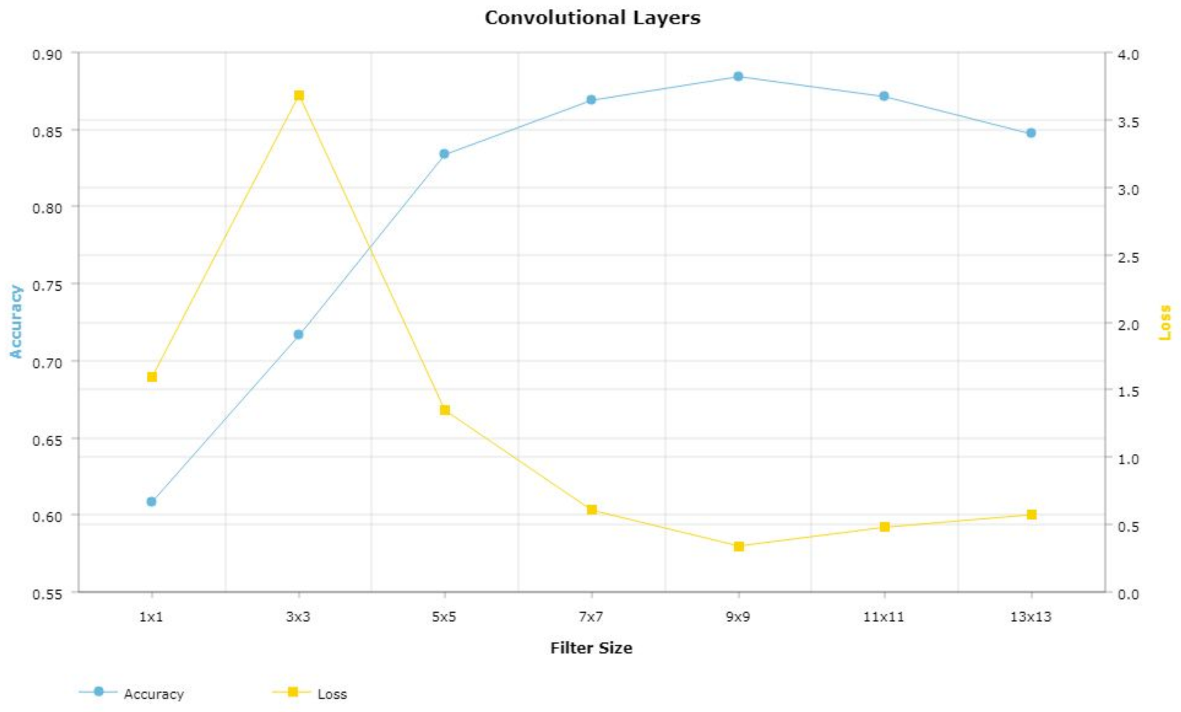


Fig. 4.5d: Graph of results comparing the effects of different filter sizes

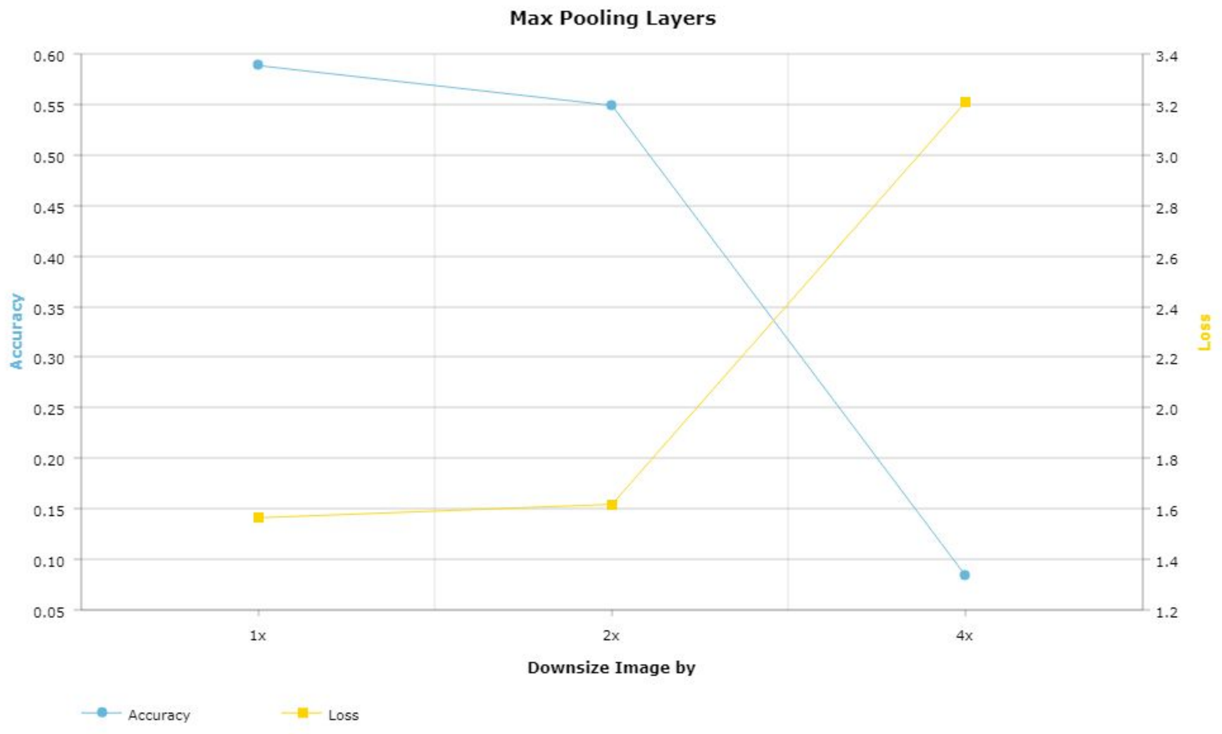
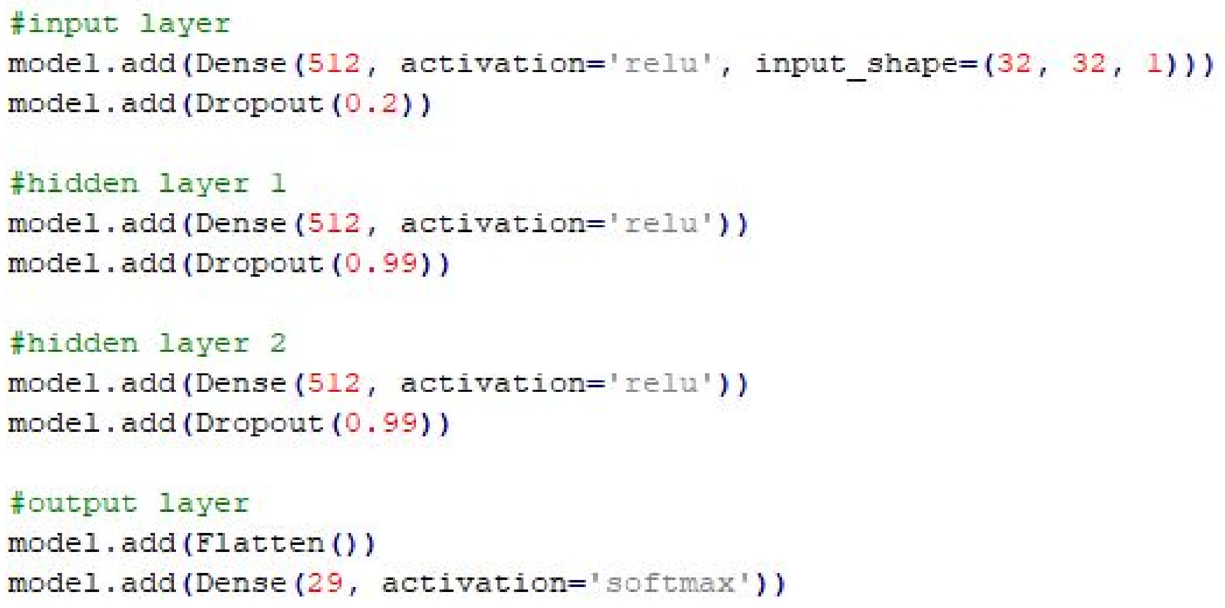


Fig. 4.5e: Graph of results comparing the effects of different sizes of Max Pooling layers

4.6: Dropout Testing

This section tested the effects of different dropout rates. An example of how the baseline layers were modified is shown in Fig. 4.6a. Overall, dropout rate had little effect on accuracy and loss of the neural network until a sharp drop at 0.95 and 0.99. The full results are shown in Fig. 4.6b. Not shown in this data is how the dropout rate affected the delta between the training accuracy and loss, and the testing accuracy and loss. Higher dropout rates decreased this delta, suggesting that training would be effective for more epochs.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.99))

#hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.99))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.6a: The layers as used for testing dropout rate

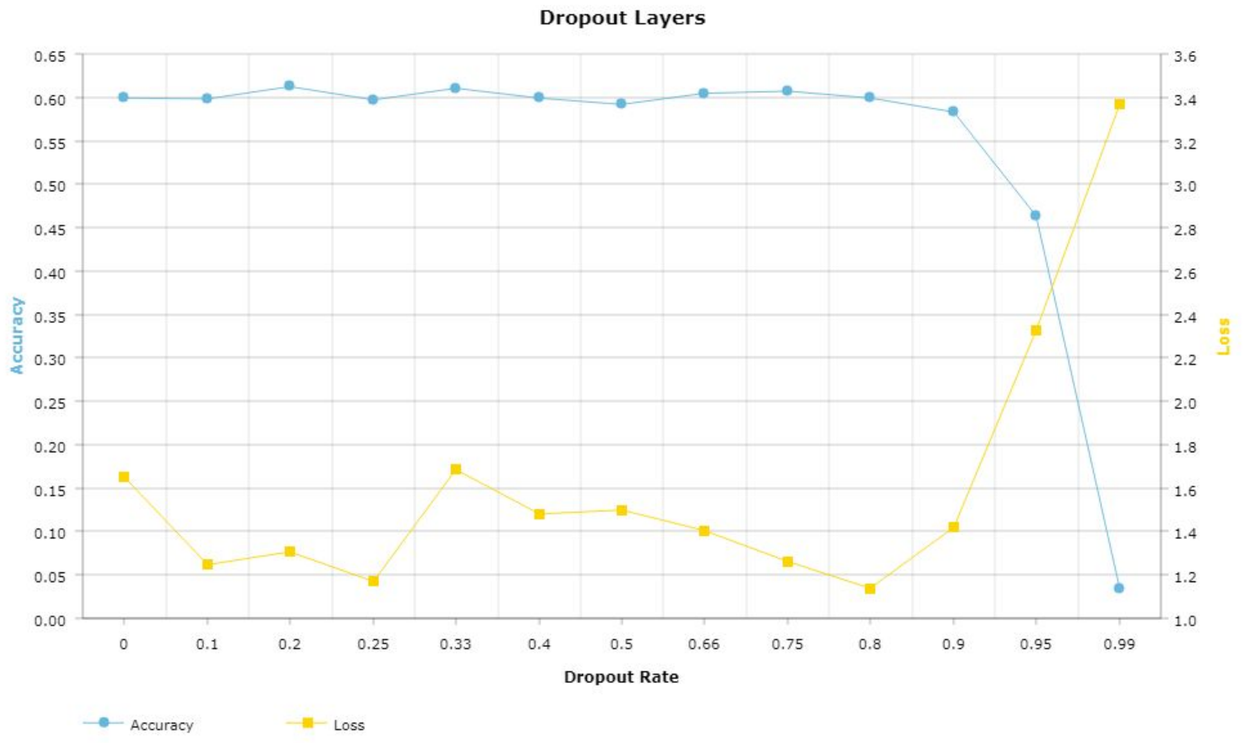
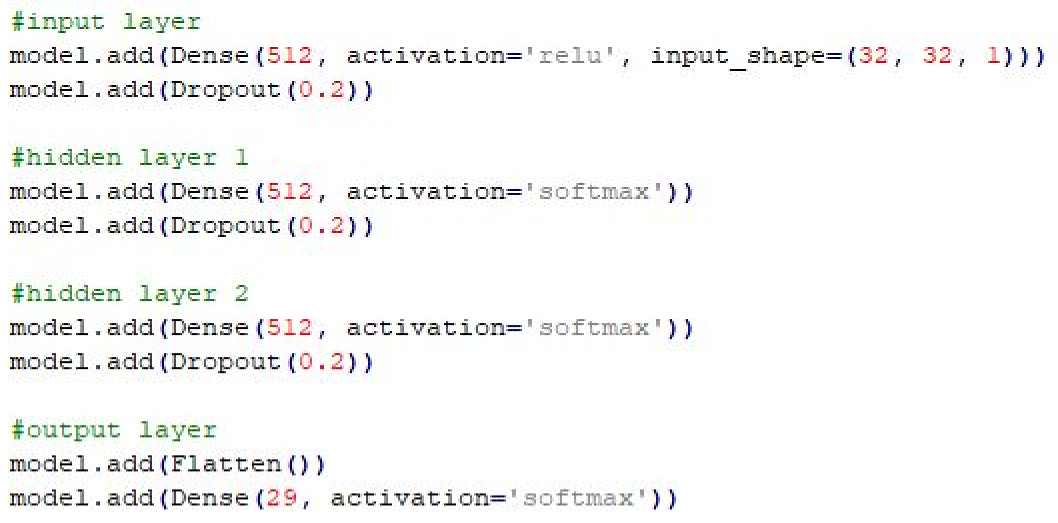


Fig. 4.6b: Graph of the effects of increasing dropout rate

4.7: Activation Testing

In this phase, four different activations for the hidden layers were tested: relu, tanh, sigmoid, and softmax. An example of the layers used is shown in Fig. 4.7a. From this testing, relu performed the best in both accuracy and loss. The full results are shown in Fig. 4.7b.



```
#input layer
model.add(Dense(512, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.2))

#hidden layer 1
model.add(Dense(512, activation='softmax'))
model.add(Dropout(0.2))

#hidden layer 2
model.add(Dense(512, activation='softmax'))
model.add(Dropout(0.2))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 4.7a: The layers as used for testing different activations

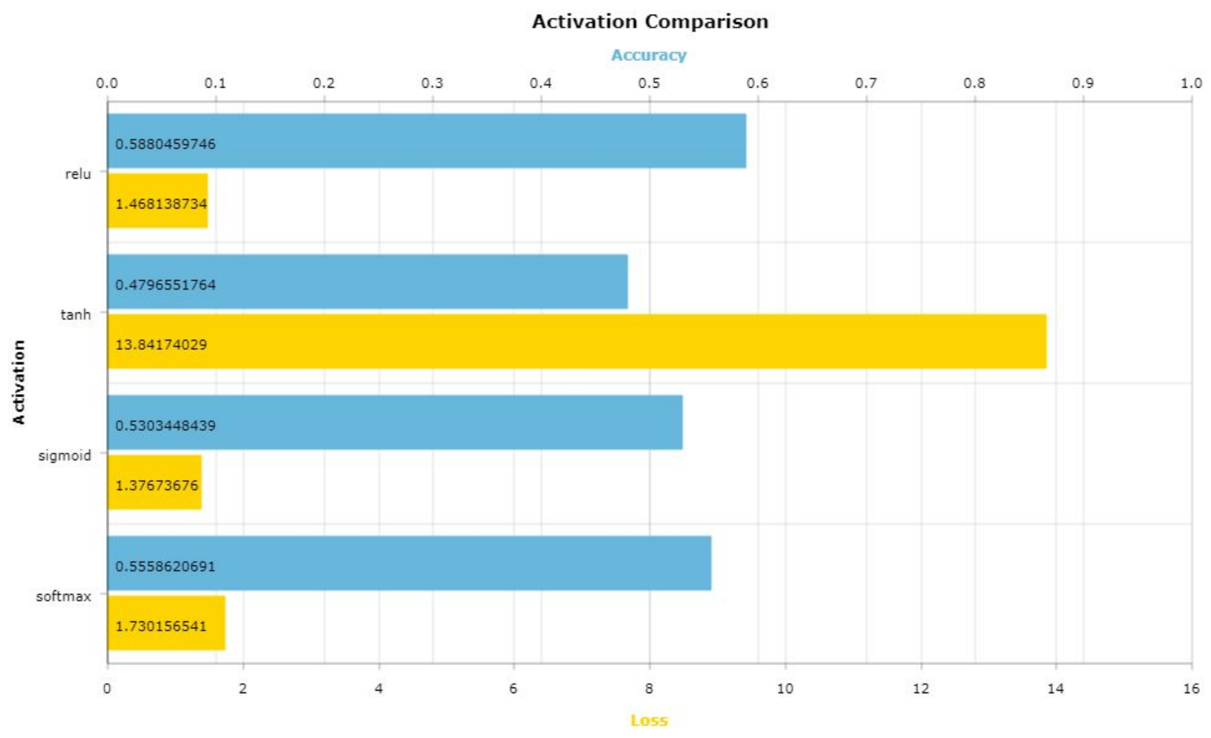


Fig. 4.7b: Graph of the effects of different activations

4.8: Loss Function Testing

In this section two different categorical loss functions were tested, Categorical Crossentropy and Kullback-Leibler Divergence. The data from this testing show that Categorical Crossentropy is far superior for this problem. The full results are shown below in Fig. 4.8.

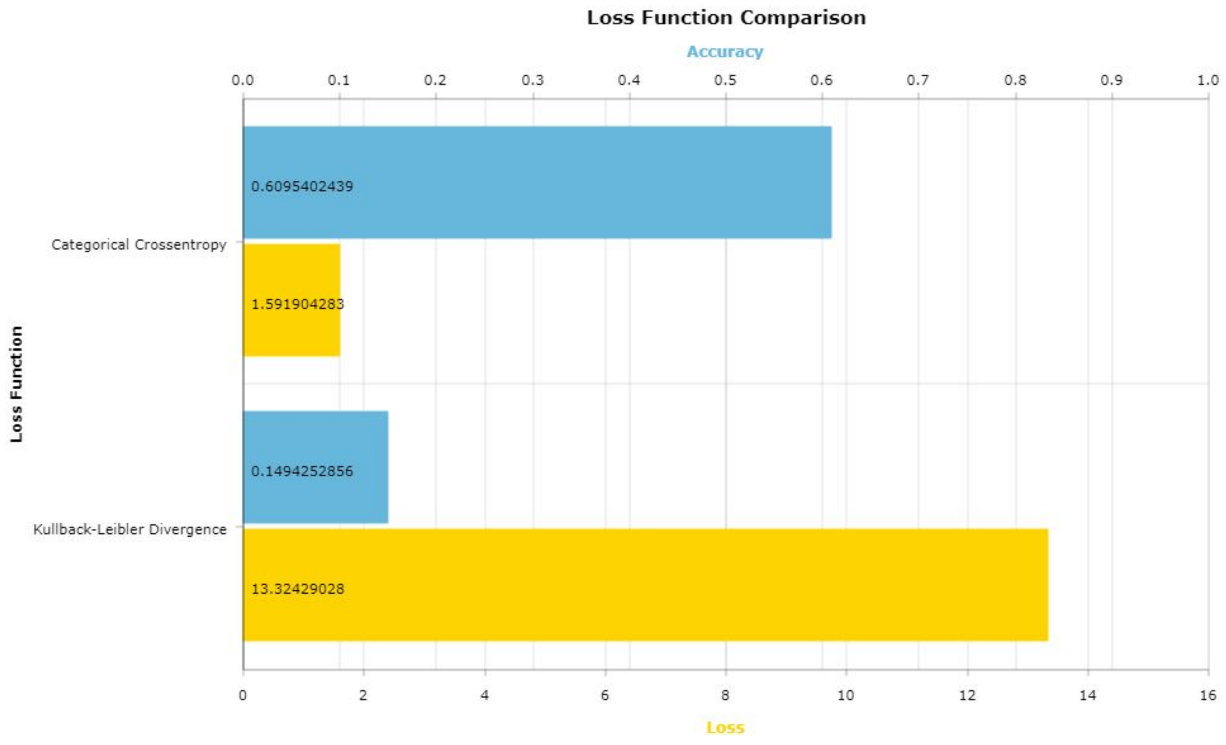


Fig. 4.8: Graph of the effects of different loss functions

4.9: Optimizer Testing

In this final test, different optimizers were compared. RMSprop, the optimizer used for all of the other testing was compared to: SGD, Adam, Adamax, and Adadelta. They all performed significantly worse than RMSprop, with the one exception of Adadelta, which performed similarly in accuracy, but had marginally lower loss. The full results are shown in Fig. 4.9.

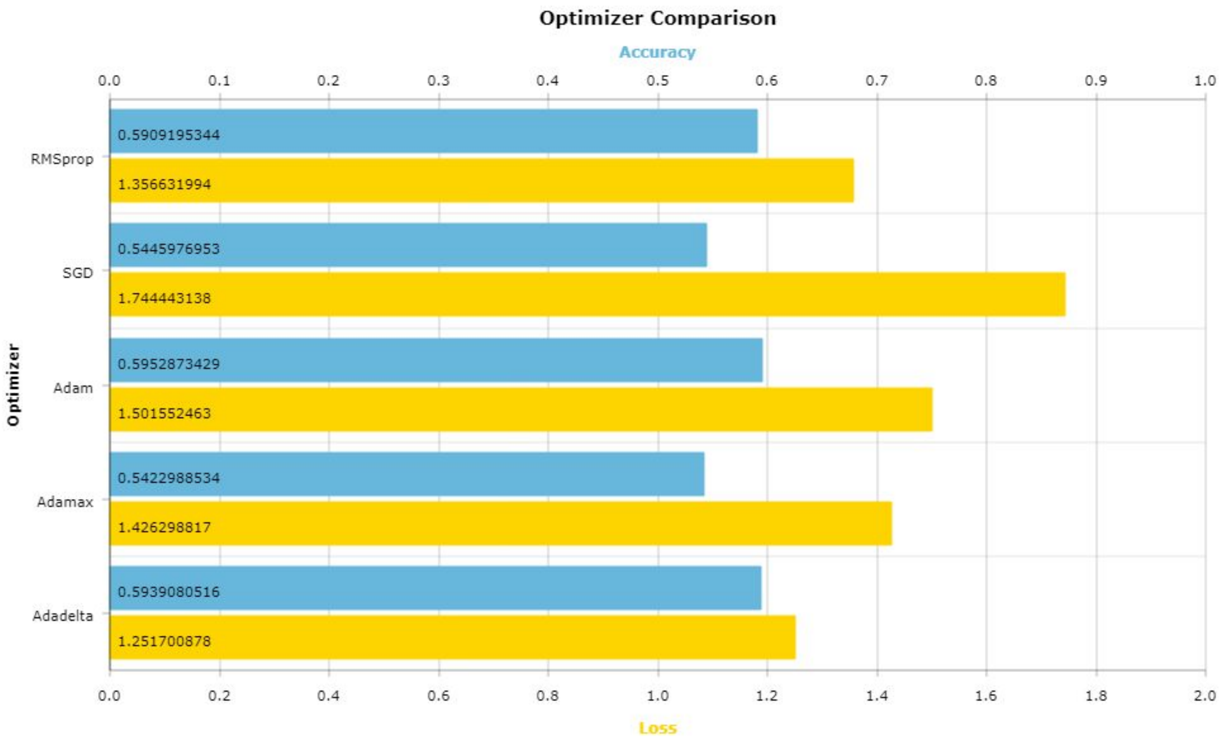
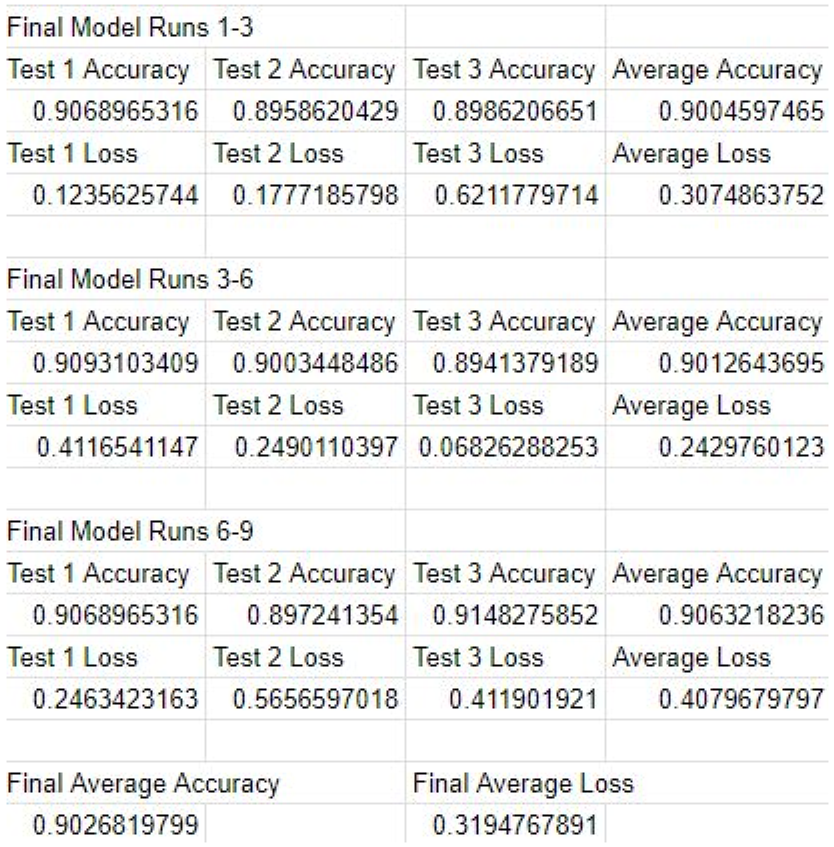


Fig. 4.9: Graph of the effects of different optimizers

Chapter 5:
Conclusions

5.1: Results

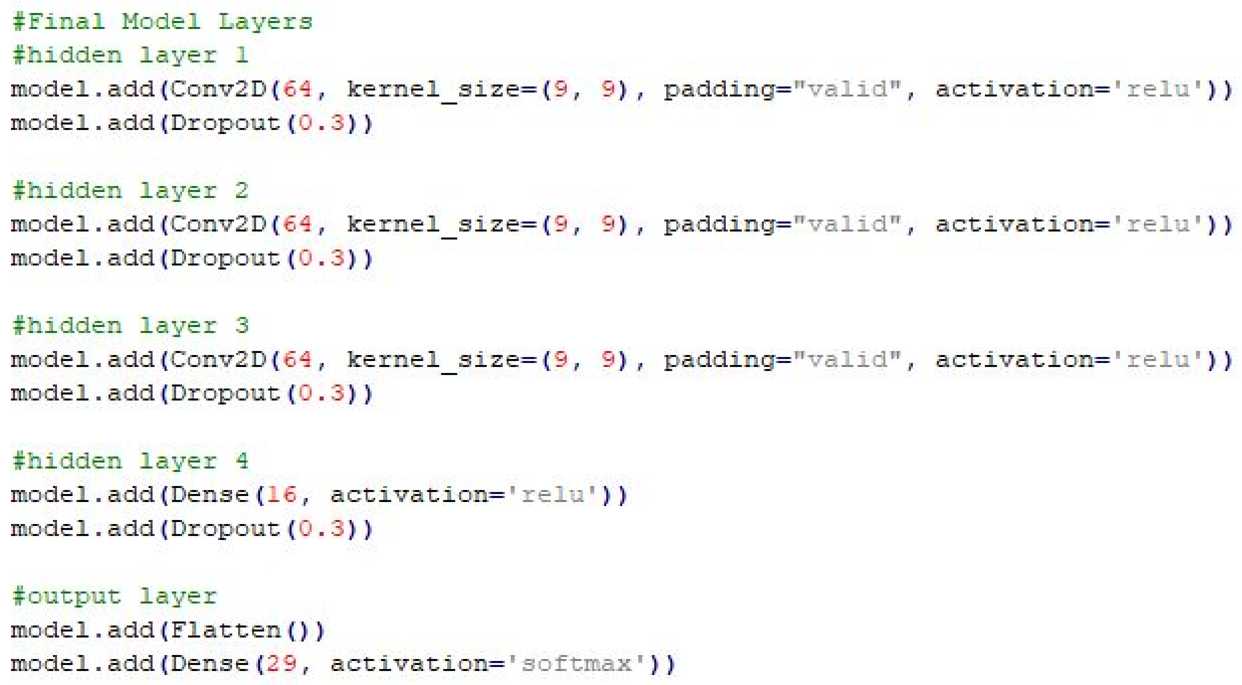
Using what was learned during the research presented in Chapter 4, and with some additional tweaking and fine-tuning, a model was created with an accuracy of greater than 0.9 in each of three averages of three runs, with an overall nine-run average accuracy of 0.9027 and a loss of 0.3195. The full results of all nine runs can be seen in Fig. 5.1a.



Final Model Runs 1-3			
Test 1 Accuracy	Test 2 Accuracy	Test 3 Accuracy	Average Accuracy
0.9068965316	0.8958620429	0.8986206651	0.9004597465
Test 1 Loss	Test 2 Loss	Test 3 Loss	Average Loss
0.1235625744	0.1777185798	0.6211779714	0.3074863752
Final Model Runs 3-6			
Test 1 Accuracy	Test 2 Accuracy	Test 3 Accuracy	Average Accuracy
0.9093103409	0.9003448486	0.8941379189	0.9012643695
Test 1 Loss	Test 2 Loss	Test 3 Loss	Average Loss
0.4116541147	0.2490110397	0.06826288253	0.2429760123
Final Model Runs 6-9			
Test 1 Accuracy	Test 2 Accuracy	Test 3 Accuracy	Average Accuracy
0.9068965316	0.897241354	0.9148275852	0.9063218236
Test 1 Loss	Test 2 Loss	Test 3 Loss	Average Loss
0.2463423163	0.5656597018	0.411901921	0.4079679797
Final Average Accuracy		Final Average Loss	
0.9026819799		0.3194767891	

Fig. 5.1a: A table of the full results of testing the final model

This final model was made with a Dense input layer, followed by three hidden convolutional layers with filter sizes of 9x9, a fourth hidden Dense layer, and a Dense output layer. The code for the final layers can be seen in Fig. 5.1b, and a flowchart-style graphic of the layers can be seen in Fig. 5.1c.



```
#Final Model Layers
#hidden layer 1
model.add(Conv2D(64, kernel_size=(9, 9), padding="valid", activation='relu'))
model.add(Dropout(0.3))

#hidden layer 2
model.add(Conv2D(64, kernel_size=(9, 9), padding="valid", activation='relu'))
model.add(Dropout(0.3))

#hidden layer 3
model.add(Conv2D(64, kernel_size=(9, 9), padding="valid", activation='relu'))
model.add(Dropout(0.3))

#hidden layer 4
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))

#output layer
model.add(Flatten())
model.add(Dense(29, activation='softmax'))
```

Fig. 5.1b: The layers that make up the final model

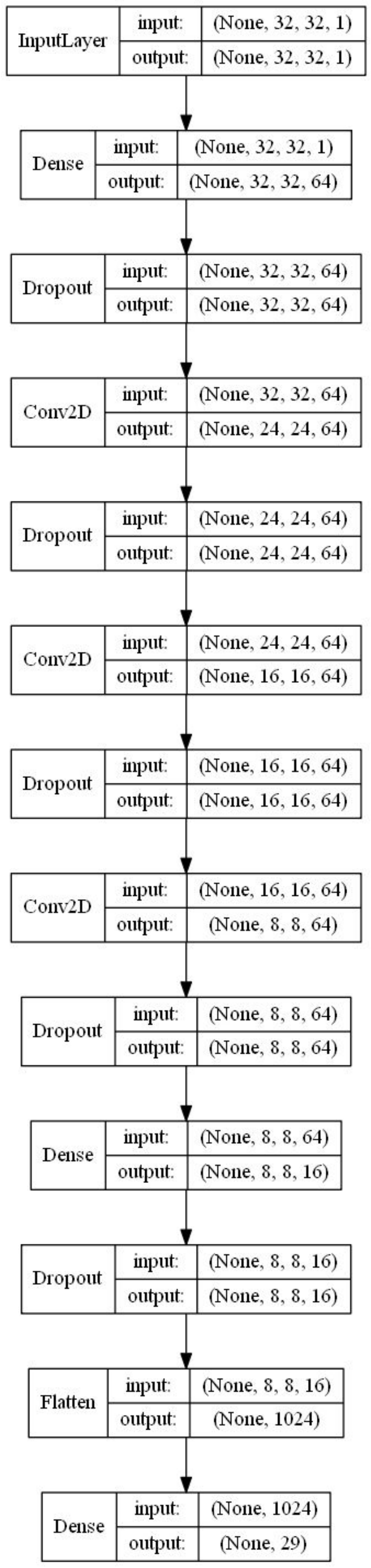


Fig. 5.1c: Flowchart of the layers in the final model

5.2: Shortcomings

The largest shortcoming of this project is the dearth of data. The Sorani Kurdish alphabet has 33 (or 34, depending on what is counted as a letter) letters, but samples were only able to be obtained on 29 of them. For those 29 letters, only 500 samples of each are available, and must be split between the training, validation, and testing sets. With more data, the accuracy and usability of the model would surely improve.

An additional shortcoming is that, occasionally during training, the model will drop from a training accuracy of approximately 0.95 and a validation accuracy of approximately 0.89 to training and validation accuracies of 0.345, which is pure chance. This problem was difficult to address due to its inconsistency and no solution was able to be found. If this occurs it is recommended to run the model again until it succeeds.

5.3: Applications

Assuming that the shortcomings listed in section 5.2: Shortcomings are addressed, this model could be paired with other neural networks and software to convert a standard photo of Sorani Kurdish text into a text file on a computer. Since this model has inputs of 32x32 black and white images of isolated characters, other models would be needed to break the image into chunks of isolated characters and words, and output the characters to this model, and another model able to recognize the words, whose characters are slightly different due to the cursive nature of this script.

5.4: Further Research

While this project did test many variables in a neural network for this particular problem, further testing would be needed to see the precise effects of modifying the input and output layers, as well as how different types of layers in the same model interact. Recurrent layers were also not tested in this project and should be looked into in the future.

References

Hassanpour, A. “History of Kurdish Orthography.” *Kurdish Academy*, Kurdish Academy of Language, 1992, kurdishacademy.org/?p=233.

“Keras: The Python Deep Learning Library.” *Home - Keras Documentation*, *keras.io*/.

Loey, Mohamed. “Arabic Handwritten Characters Dataset.” *Kaggle*, 22 June 2017, www.kaggle.com/mloey1/ahcd1.

Sadri, Javad, et al. “A Novel Comprehensive Database for Offline Persian Handwriting Recognition.” *Pattern Recognition*, vol. 60, Dec. 2016, pp. 378–393., doi:10.1016/j.patcog.2016.03.024.

Salian, Isha. “SuperVize Me: What’s the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?” *The Official NVIDIA Blog*, Nvidia, 20 Aug. 2019, blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/.

Sanderson, Grant, director. *But What Is a Neural Network? | Deep Learning, Chapter 1*. *Youtube*, 3Blue1Brown, 5 Oct. 2017, www.youtube.com/watch?v=aircAruvnKk.

“Sorani Alphabet.” *Parsendow Translation*, Sharepoint, parsendowtranslation.sharepoint.com/siteimages/sorani%20alphabet.png.

Appendix

```
#SoraniOCR.py

#A program to create a deep convolutional neural network
#to perform optical character recognition on Sorani Kurdish
#using TensorFlow, Keras, Pillow, Pydot, and Graphviz
#Made by Zachary Aaron Clark
#as part of a capstone project for a B.S. in Computer Science
#from SUNY Polytechnic Institute
#Last edit 4/29/2020

#The created model will achieve an average of approximately
#0.907 accuracy and 0.319 loss after 20 epochs of training
#Each epoch takes approx 6-10 seconds on a GTX 1080

#For directory management
import os
#For creation of the neural network
import keras
#To parse the data files and prepare them for training
from keras.preprocessing.image import ImageDataGenerator
#The model type being used
from keras.models import Sequential
#Different layers types that were tested or used in the final model
from keras.layers import Flatten, Dense, Conv2D, MaxPooling2D, Dropout
#Different optimizers that were tested
from keras.optimizers import RMSprop, Adam, Adamax, Adadelta, SGD
#To create a visualization of the model
from keras.utils import plot_model
```

```
#Number of epochs to train for before testing
epochs = 20

#Create data generator
datagen = ImageDataGenerator()

#Load and iterate training dataset
train_it = datagen.flow_from_directory(os.path.dirname(__file__) + '/data/train/',
target_size=(32,32), class_mode='categorical', batch_size=50, color_mode='grayscale')
#Load and iterate validation dataset
val_it = datagen.flow_from_directory(os.path.dirname(__file__) + '/data/validation/',
target_size=(32,32), class_mode='categorical', batch_size=50, color_mode='grayscale')
#Load and iterate test dataset
test_it = datagen.flow_from_directory(os.path.dirname(__file__) + '/data/test/',
target_size=(32,32), class_mode='categorical', batch_size=50, color_mode='grayscale')

#Create model
model = Sequential()

#Input layer
model.add(Dense(64, activation='relu', input_shape=(32, 32, 1)))
model.add(Dropout(0.3))

#Baseline
'''
#Hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
```

```
#Hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
'''

#Layer Type Testing
'''
#Dense Layers
#Hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Convolutional Layers
#Hidden layer 1
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Pooling Layers
#Hidden layer 1
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))
'''

#Biased Filtering Testing
'''
#Convolutional Layer with Bias
#Hidden layer 1
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu', use_bias=True))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu', use_bias=True))
model.add(Dropout(0.2))
'''

#Layer Amount Testing
'''
#Dense Layers
#Hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
```

```
#Hidden layer 3
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 4
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 5
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))

#Convolutional Layers
#Hidden layer 1
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 3
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 4
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))
```

```
#Hidden layer 5
model.add(Conv2D(512, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Pooling Layers
#Hidden layer 1
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 3
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 4
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 5
model.add(MaxPooling2D(pool_size=(2, 2), padding='valid'))
model.add(Dropout(0.2))
'''

#Layer 'Size' Testing
'''
```

```
#Dense Layers
#Hidden layer 1
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.2))

#Convolutional Layers
#Filter Amount
#Hidden layer 1
model.add(Conv2D(2048, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Conv2D(2048, kernel_size=(3, 3), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Filter Size
#Hidden layer 1
model.add(Conv2D(512, kernel_size=(13, 13), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Conv2D(512, kernel_size=(13, 13), padding='valid', activation='relu'))
model.add(Dropout(0.2))

#Pooling Layers
```

```
#Hidden layer 1
model.add(MaxPooling2D(pool_size=(4, 4), padding='valid'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(MaxPooling2D(pool_size=(4, 4), padding='valid'))
model.add(Dropout(0.2))
'''

#Dropout Testing
'''
#Hidden layer 1
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.99))

#Hidden layer 2
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.99))
'''

#Activation Testing
'''
#Hidden layer 1
model.add(Dense(512, activation='softmax'))
model.add(Dropout(0.2))

#Hidden layer 2
model.add(Dense(512, activation='softmax'))
model.add(Dropout(0.2))
```

```
'''

#Final Model Layers
#Hidden layer 1
model.add(Conv2D(64, kernel_size=(9, 9), padding='valid', activation='relu'))
model.add(Dropout(0.3))

#Hidden layer 2
model.add(Conv2D(64, kernel_size=(9, 9), padding='valid', activation='relu'))
model.add(Dropout(0.3))

#Hidden layer 3
model.add(Conv2D(64, kernel_size=(9, 9), padding='valid', activation='relu'))
model.add(Dropout(0.3))

#Hidden layer 4
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))

#Output layer
model.add(Flatten())
#"29" represents the number of output categories. After 04/17/2023 the license for 10
#classes of the data will expire and must be deleted. To ensure the program still works
#after this date, change "29" to "19".
model.add(Dense(29, activation='softmax'))

#Print a summary of the model to the screen
model.summary()
```

```
#Save a visualization of the model to a file
#plot_model(model, show_shapes=True, show_layer_names=False,
#                              to_file='C:\ProgramData\Anaconda3\envs\capstone\capstone\model.png')

#Compile model
model.compile(loss='categorical_crossentropy', optimizer=RMSprop(), metrics=['accuracy'])

#Train model
history = model.fit(train_it, epochs=epochs, verbose=1, validation_data=val_it)

#Test model
score = model.evaluate(test_it, verbose=0)
print('Test accuracy:', score[1])
print('Test loss:', score[0])
```