CS 470 Final Project

Eric Todd

April 21, 2020

1 Introduction

For my final project, I wanted to use Deep Learning for prediction as part of a pipeline for the word game Boggle. Boggle is a pretty simple game consisting of a 4x4 grid of dice with letters on them. The object of the game is to write down all of the words that you can find such that each letter in the word is adjacent to the previous one, and no letter is repeated in the same word. I built a pipeline that takes an image of a regular boggle board, segments the image to find each letter on the board, and then uses a deep neural network to predict the letters using the segmented images. These predictions are then used to recursively find all the possible words for the given board. This allows for scoring the board, easy checking of any player's words, and the points they might receive for finding those words.

2 Methods

2.1 Building the Dataset

I started with one image for each of the 26 letters (except Q is Qu in boggle, so I had an image of Qu in this case). Using these images as my base, I researched and tested multiple data augmentation techniques, such as random rotations, random translations, adding random noise, random deletions, etc. to increase the size of my dataset to use in training my model. I initially started by creating 5000 images for each letter, and applying many augmentation techniques, creating a total of 130,000 images. However, I found that having so many images significantly slowed the training process, and that fewer images were sufficient for the same purpose. So, I reduced my final dataset to 2000 images per letter (52,000 total), and only used random rotations, random translations, and adding random noise to match more closely the sorts of images that would come from the pipeline. An example of a batch of these images is shown below.

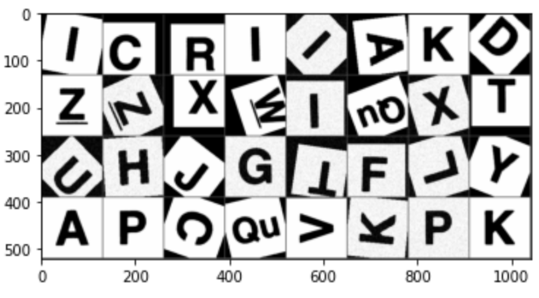


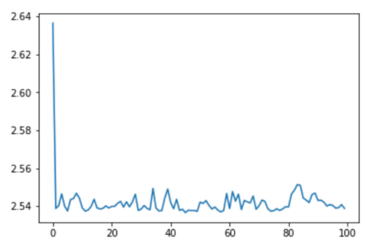
Figure 1: Example Batch of 32 Training Images

2.2 Deep Learning

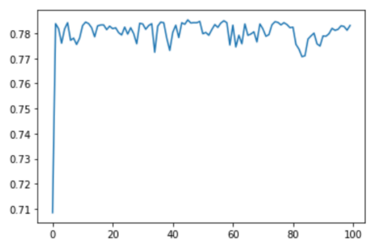
I built my neural network using the framework of PyTorch, and my initial model architecture consisted of multiple convolutional layers, followed by ReLU activations, maxpooling, and then 3 fully connected linear layers with a softmax at the end. The model takes 128x128 images as inputs and returns a vector of 26 probabilities, which represent scores for how likely the model thinks each letter A-Z matches the image.

After building this initial model architecture, I set off to train my model. When I had 5000 images per letter, the training process took somewhere around 30-40 minutes per epoch. However, increasing the number of workers in my DataLoaders and reducing the training set to 2000 per letter allowed me to train my model at the speed of about 3-5 minutes per epoch.

When I first trained my model, I hit a local minimum in the loss landscape that I was unable to overcome with any number of epochs of training or changing the batch size. The model capped at around 80% accuracy. The loss and accuracy plots typical of training at this point looked like the plots below (see Figure 2). I started with a smaller batch size, but then increased it to make sure the batch gradient was more representative of the overall gradient.



(a) Training Loss per Epoch



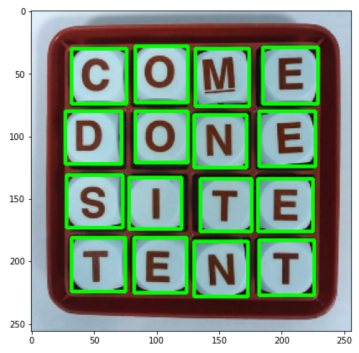
(b) Training Accuracy per Epoch

Figure 2: Training Loss and Accuracy for Initial Model

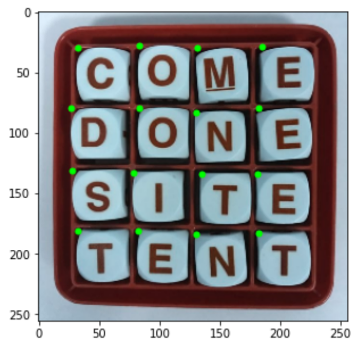
After getting my model up to around 80% accuracy, and since changing the batch size did not have an effect on improving the loss, I decided I would have to modify my model architecture. However, I decided to work on the other parts of the pipeline before trying to improve the model.

2.3 Image Segmentation

I started with figuring out how to segment a boggle board into 16 images, each one containing an image of a letter on the board. I used openCV to find bounding contours in the image, and then subset to all contours that were approximately 1/4 the size of the image, using a threshold on the width and height. This allowed me to grab an approximate square around each letter (see Figure 3a). Since the contours that openCV finds are not ordered, I decided to find the top corner of each of these contours and then sorted by row-value and column-value to sort the letter images properly (see Figure 3b). This allowed my algorithm to provide each letter image in the correct good order for my model to predict them.



(a) Letter Bounding Boxes



(b) Finding the Top Corners

Figure 3: Boggle Board Segmentation Process

By saving only the contour segments, I was able to get 16 images to feed into my model, one for each letter on the board. The separated letters can be seen in Figure 4.

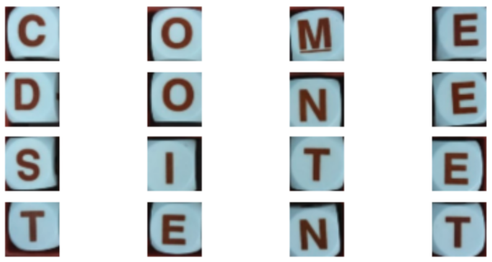
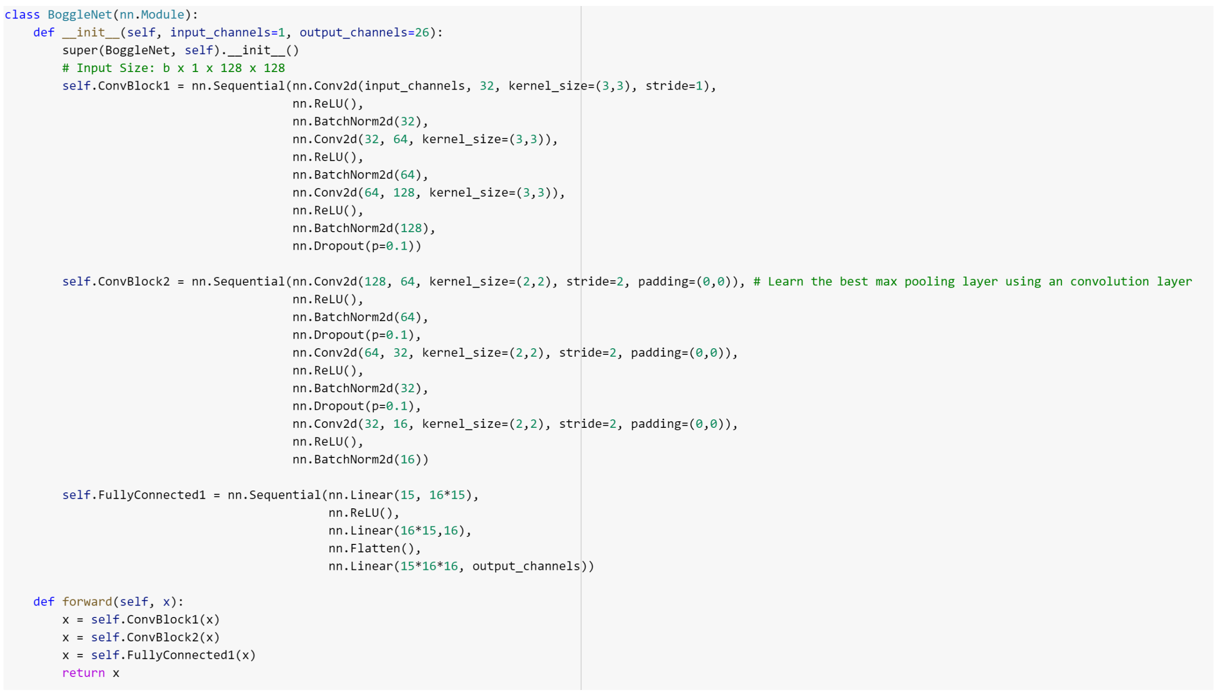


Figure 4: Segmented Images for each Letter

2.4 More Deep Learning

Once I solved the problem of segmenting a boggle board into images that my model could predict on, I went back to trying to fix my model so that it could more accurately identify these letters from the images themselves. After some research, I found that the softmax layer I had included in my architecture was affecting my output, because the CrossEntropyLoss function I was using to compute my model's loss already applies the softmax internally [1]. Thus, I removed the softmax layer, and also added some dropout and batch normalization layers. These served to make the individual neurons more robust to noise, and normalize the inputs for consecutive layers. Another thing that I updated in my model architecture was I replaced the maxpooling layers with convolutional layers with the same size reduction (2x2 kernel, no padding) so that my model could learn the best maxpooling/downsampling instead of being forced to pick the maximum. The weights of these layers would also then be trained along with the rest of the network.



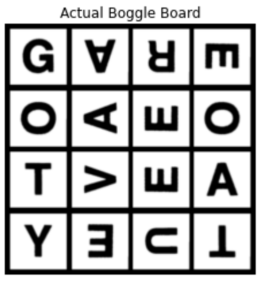
```
class BoggleNet(nn.Module):
    def __init__(self, input_channels=1, output_channels=26):
        super(BoggleNet, self).__init__()
        # Input Size: b x 1 x 128 x 128
        self.ConvBlock1 = nn.Sequential(nn.Conv2d(input_channels, 32, kernel_size=(3,3), stride=1),
                                        nn.ReLU(),
                                        nn.BatchNorm2d(32),
                                        nn.Conv2d(32, 64, kernel_size=(3,3)),
                                        nn.ReLU(),
                                        nn.BatchNorm2d(64),
                                        nn.Conv2d(64, 128, kernel_size=(3,3)),
                                        nn.ReLU(),
                                        nn.BatchNorm2d(128),
                                        nn.Dropout(p=0.1))

        self.ConvBlock2 = nn.Sequential(nn.Conv2d(128, 64, kernel_size=(2,2), stride=2, padding=(0,0)), # Learn the best max pooling layer using an convolution layer
                                        nn.ReLU(),
                                        nn.BatchNorm2d(64),
                                        nn.Dropout(p=0.1),
                                        nn.Conv2d(64, 32, kernel_size=(2,2), stride=2, padding=(0,0)),
                                        nn.ReLU(),
                                        nn.BatchNorm2d(32),
                                        nn.Dropout(p=0.1),
                                        nn.Conv2d(32, 16, kernel_size=(2,2), stride=2, padding=(0,0)),
                                        nn.ReLU(),
                                        nn.BatchNorm2d(16))

        self.FullyConnected1 = nn.Sequential(nn.Linear(15, 16*15),
                                             nn.ReLU(),
                                             nn.Linear(16*15,16),
                                             nn.Flatten(),
                                             nn.Linear(15*16*16, output_channels))

    def forward(self, x):
        x = self.ConvBlock1(x)
        x = self.ConvBlock2(x)
        x = self.FullyConnected1(x)
        return x
```

Figure 5: My Network Architecture in code

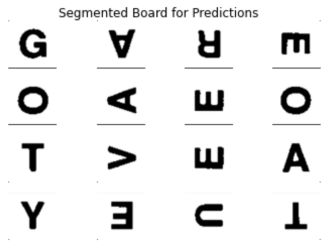
After updating my model, I trained this new architecture and was able to achieve **96.88%** accuracy on the training set in only 8 epochs, and for many batches was getting 100% accuracy. It still took 40 minutes to train this classifier, but the results were much better than the 80% accuracy previous iterations had attained. With my newly trained and improved model, I set out to put all of the pieces together in a Boggle pipeline.

3 Putting it all Together

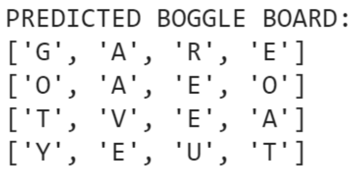
My pipeline starts with taking an image of a Boggle board, and then passing that file to a function that segments the image. The individual letter images are then passed to the trained model for prediction, which returns the guess of the board in the image.



(a) Original Board



(b) Segmented Letters



```
PREDICTED BOGGLE BOARD:
['G', 'A', 'R', 'E']
['O', 'A', 'E', 'O']
['T', 'V', 'E', 'A']
['Y', 'E', 'U', 'T']
```

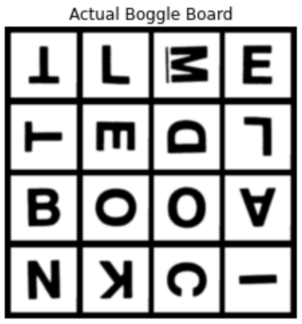
(c) Neural Network Predicted Board

Figure 6: Boggle Pipeline

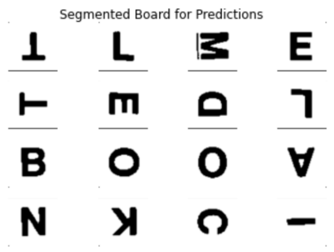
After the model produces a prediction for the board, a BoggleBoard class I built in Python finds all possible words recursively, given each square as the starting point. This class also contains information about each word's point value, and can give the top words to play for the given board. Using the list available from this class, I am able to check the words I found from the boggle board and also see what words I missed. For example, the top 6 words from the board in Figure 6 and their point totals are:

$$overate : 5, goatee : 3, aerate : 3, revote : 3, agave : 2, agate : 2$$

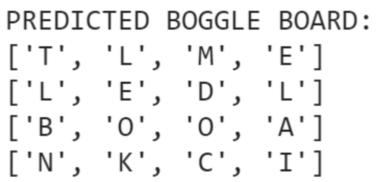
It was exciting to get pretty good results for some boards, but there is still work to be done. One of the problems that I ran into while working on this project is that my model's accuracy is still only around 96%. This may seem like a great success in most other applications, however to be practical, it needs to be 100% accurate. This is because while it is very good, it isn't 100% reliable for Boggle until it correctly classifies each letter. Otherwise, it will predict the wrong board, and the BoggleBoard class will find invalid words that aren't possible using the original board. After looking at some of the cases that my current model is missing, it seems to be mistaking the following letters the most often: X's & Y's, L's & T's, and B's & R's. An example of this is shown below (see Figure 7), where my model predicted an "L" instead of a "T" in the second row, first column.



(a) Original Board



(b) Segmented Letters



```
PREDICTED BOGGLE BOARD:
['T', 'L', 'M', 'E']
['L', 'E', 'D', 'L']
['B', 'O', 'O', 'A']
['N', 'K', 'C', 'I']
```

(c) Neural Network Predicted Board

Figure 7: Boggle Pipeline Mistake - "L" vs. "T"

These are all reasonable mistakes given the augmentation transformations I applied, but I hope that more training, higher percentages of dropout, or larger batch sizes will improve the model's accuracy and clean up some of these mistakes. This will allow the network to be more reliable for use in actual games of Boggle.

4 Conclusion

Overall, this was a very fun project to work on, and allowed me to gain confidence in my ability to build my own successful neural networks. I had taken the CS Deep Learning Class, but felt like the building process was heavily supplemented, and that this is the first deep learning project I built entirely from scratch without guidelines. I was also happy to implement the solving of the boggle board using recursive ideas that we discussed in the first half of the semester (depth-first search, etc.) when we learned about different search algorithms. I am glad I was able to apply the ideas of AI & ML to solve an interesting problem and hope to continue to use them in the future.

References

[1] Slav Ivanov. 37 reasons why your neural network is not working.