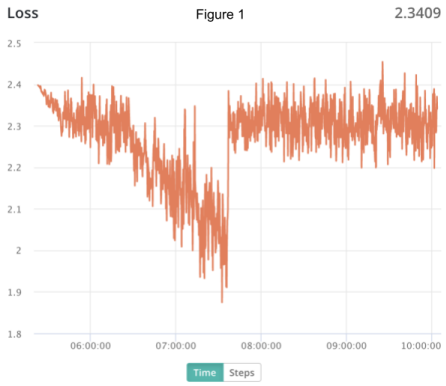
Isaac Shiffman
CS 6476 Fall 2019

Digit Detection and Localization With CNNs

I. Introduction

The goal of this experiment was to gain exposure to working with CNNs for use during image localization and classification. In this experiment three different implementations of the VGG16 algorithm were used. Of the three implementations included, two used the structure already available in Keras. One that had a custom architecture created just for this experiment. The custom CNN, and one of the Keras VGG16's was trained from scratch. For the final CNN pretrained weights were utilized so a comparison could be made to the other methods. Localization in real world images required many techniques beyond CNN classification. I found that a combination of ML and classical CV methods was most effective. To produce my final results, the output from a rule based CV system that excluded parts of the image was fed into a CNN trained on images from the SVHN dataset.

II. Hyperparameters



In all of the models significant hyperparameter tuning was required. Each model involved some analysis steps which led to some final hyperparameter values which provided the most optimal results. For the pretrained VGG16, the hyperparameter process was simplest. I was able to follow along largely with information from research papers and tutorials and eventually settle at a standard learning rate of 1-e3. Because there were fewer choices in the implementation of this model, I found it to be the best way to test how changing hyperparameters affected the outcomes. If the learning rate was too high, the loss and accuracy metrics would jump up and down. I concluded that with a higher learning rate in the order of 0.01, the weights of the model were being updated too much in each epoch. The learning rate of 1-e3 gave the best results in terms of performance and training speed.

I experimented a good deal with the decay of the learning rate as well. The results of the experiment were checked both with and without decay, but eventually it was clear that using a decay rate was beneficial for results and performance. Without decay of the learning rate, the model would run and run without appearing to reach any convergence point. Figure 1 is a good example of this. As the model continued to run, it eventually reached a point where the loss spiked and the results were then useless. Therefore, in all subsequent runs I used a decay rate proportional to the starting learning rate of each model. I had to apply some checking to notify me when the network stopped learning. These checks combined with learning rate decay helped me avoid the overfitting problems that are visible in Fig. 1.

III. Design Decisions

There were several key design decisions that were made during the design process of the networks. These included the choice of optimizer, the loss function choice, the decision to freeze layers, and data loading concerns. All of these decisions had a major impact on the results of the experiment. The choices will be discussed in detail here. The conclusions were made after a combination of research and experimentation.

Research helped me narrow down my optimizer choice to either ADAM and SGD. SGD is a type of randomized gradient descent that only acts on a subset of the data samples, but still produces comparable results and is relatively fast. Adam is a newer optimizer that is based on the optimization of randomized objective functions. I started by using SGD as it seemed simpler to understand. Eventually I switched to ADAM, but I found that the output of my network seemed very randomized. I was worried that using ADAM could be contributing to convergence problems, so after some testing, I used SGD exclusively moving forwards. Because digit classification is a categorical classification problem where no item can belong to multiple classes, categorical cross entropy was immediately chosen as the loss function. This was used in all models. I did not do much experimentation with other loss functions because most of the academic resources I consulted suggested using it. Furthermore, because softmax is the output of VGG16, categorical cross entropy fits perfectly. I had not had a chance to experiment with a softmax output before doing this digit experiment, but I found it helpful throughout many phases of the experiment to be able to compare the probabilities of the different classes. The softmax probability distribution provided me with more intuition about my model than a mere label would have.

One final major decision that was made was the decision to freeze some layers for some of the models. No layers were frozen in my custom implementation of VGG16. In the pre trained Keras VGG16, all layers except the last were frozen. Finally, in the untrained Keras VGG16, 3 layers were frozen. From reading about VGG16, I understood that the first few layers model more general aspects of the images, and the later layers are used to learn the actual aspects of the classes. Because of this, I decided to freeze the first 3 layers of the untrained model, and use imagenet weights for those. This decreased the number of trainable parameters in the Keras VGG16 implementations. The effects of the reduced parameters are discussed in more detail in section V. Most of this model still required significant training so for all purposes it was still “untrained”.

IV. Localization

For localization I did not want to have an implementation that relied too heavily on using the CNN to localize the digits in the image. There were a few reasons for this. Firstly, throughout my testing the results of my CNN were often not perfect. Low accuracy on validation was common, so I decided to try to seperate the localization step as much as

I could. In the end I was able to find an effective way to accomplish localization using MSER detection. This had the added benefit of allowing me to test localization code and CNN code separately, allowing for a more streamlined development process. MSER has much lower computational demand than most CNN techniques, so it was perfect to use with the video frame processing in particular. MSER finds regions of the image that are uniform on the inside and have a higher contrast at the edges. It does this by taking the region distinctiveness and the variance of intensity values inside the region into account.

Beyond simply using MSER, my localization algorithm used a few extra techniques to find digits in the image. First, some preprocessing was done on the image. It was converted to grayscale. Mean subtraction and division by the standard deviation were performed. This preprocessing was added in after some experimentation with the localization results, as it decreased false positives. Next, the MSER algorithm was run on the preprocessed image. I found that this was extremely effective at finding blobs in the image because of the thresholding involved. The unfortunate side effect however was that there were far too many regions returned by MSER. This large number of regions included some which fit perfectly, some overlapping regions which contained digits, and some regions which did not contain the desired sections of the image at all. The first option from here was to simply run the CNN detector on each region, but I knew that this was not a good way to go about solving the problem and there were certainly ways to eliminate processing unnecessary regions with the CNNs. Therefore, I added two additional steps that were extremely effective and produced the results I desired. First, I implemented an aspect ratio technique that I found by reading through some Matlab example documentation. The technique used relies on the fact that for most printed digits, the height of the digit is significantly greater than the width. Usually the numbers are two to three times as tall as they are wide. This is especially true of house numbers for some reason. Therefore, the width and height of each region detected by MSER could be used to compute the aspect ratio and confirm that the detected blob was significantly taller than it was wide. I started by using 2:1, but I found that changing this to 1.3:1 gave me better results when rotation of the camera was taken into account. This did result in more false positives, but the number of regions detected was greatly reduced.

After the aspect ratio technique, the returned bounding boxes were compared with each other to find overlapping regions. If the boxes contained a certain amount of overlap, they were combined with a helper function I wrote into a single bounding box. Finally, the CNN could simply be run on what few bounding boxes remained. In the experimental results the simple reductions to the regions brought the MSER results from about 1000 regions in a typical image down to a filtered result that was usually less than 50. Thorough parameter tuning of MSER along with some experimentation produced very good localization results which could then be passed to the CNN.

V. Training

Not surprisingly, training all of the networks took quite some time. The pretrained VGG16 with frozen layers was the fastest to train, followed by the VGG16 without weights. The custom network implementation was the slowest by far. This was somewhat due to the inherent attributes of each network, but the training time was also affected by the hyperparameters that were used for each model. This is somewhat clear just by looking at the trainable parameters for each model.

Pretrained VGG16 Keras Implementation With Initial Layers Frozen:

```
=================================================================
2019-12-04 03:40:00,121 INFO - Total params: 138,368,555
2019-12-04 03:40:00,121 INFO - Trainable params: 11,011
2019-12-04 03:40:00,121 INFO - Non-trainable params: 138,357,544
2019-12-04 03:40:00,121 INFO - _________________________________________________________________
```

VGG16 Keras Implementation Without Weights or Later Layers Frozen:

```
2019-12-04 04:25:33,892 INFO - =================================================================
2019-12-04 04:25:33,893 INFO - Total params: 138,368,555
2019-12-04 04:25:33,893 INFO - Trainable params: 138,329,835
2019-12-04 04:25:33,893 INFO - Non-trainable params: 38,720
2019-12-04 04:25:33,893 INFO - _________________________________________________________________
```

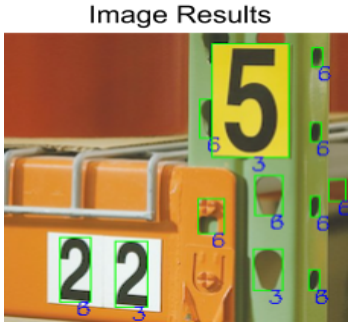
Custom VGG16 Implementation:

```
2019-12-04 04:24:33,429 INFO - =================================================================
2019-12-04 04:24:33,430 INFO - Total params: 134,305,611
2019-12-04 04:24:33,430 INFO - Trainable params: 134,305,611
2019-12-04 04:24:33,430 INFO - Non-trainable params: 0
2019-12-04 04:24:33,430 INFO - _________________________________________________________________
```

As is clear from the above reports on each model, the models with more trainable parameters took far longer overall than the models with fewer. Additionally, some of the hyperparameters affected training time. I found through testing that it was beneficial to have the VGG16 without weights and the custom implementation run through more training epochs. The pretrained implementation however was not affected as much by using higher epoch numbers, so only 2 epochs were performed instead of the 5-10 used for the other models.

VI. Testing

My testing results were somewhat better than I expected for some parts of the project, and worse for others. I was very happy about how well my localization algorithm worked. I wrote my own non-maximum suppression algorithm that used the centers of the squares to choose the best bounding box from similar boxes. For fine tuning of the boxes however, a fully functional CNN was required. For reasons I discuss in my conclusion my CNN had a host of issues, so there were many false positives in most images I ran through my pipeline. Fig. 2 shows the testing accuracy statistics of the runs of my various algorithm implementations.



VI. Testing Video+Image Results (Scale+Rotation+Location+Lighting)

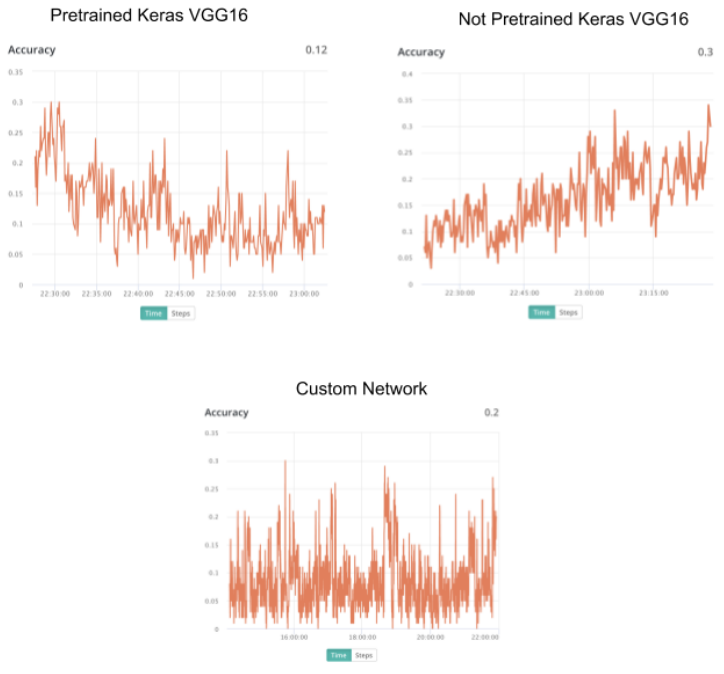
One surprising aspect of the final product was that the results of the video with skew and rotation were better initially than the results of simply running the algorithm on the 5 static images. I predicted before running the output step that the video would perform worse than the static images. Unfortunately, there were some major issues with the output results. Two examples of the output are labeled “Image Results”. Discussion of why there are false positives and why many labels are wrong is given in the conclusion section. On the whole the pipeline performed remarkable well given the issues that arose.

VII. Conclusion

While the results of my digit localization were promising, my models performed poorly. After some examination of the results, I was able to diagnose the areas that were causing the issues with my network. The largest issue involved a data processing and the specifics of the data that I was passing to my model. In addition to the issues that prevented my network from training properly, there are several improvements to my pipeline that would have been beneficial if they were implemented.

One major area that could have been improved was the part of my localization algorithm that relied on the aspect ratio of the detected MSER blob. In my implementation, I simply checked if the height was a certain magnitude greater than the width. This obviously could lead to issues when either there is excessive rotation, or if the digit does not have a typical shape. To fix this issue, a few techniques could be used. First of all, the aspect ratio could be considered in all directions instead of just vertically. This would allow the bounding box to be drawn around rotated digits. Secondly, an adaptive aspect ratio check could be used. Specifically, after classifying a single digit in an image, the aspect ratio for that digit could be recorded. Then the rest of the MSER regions could be filtered based on a known digit in the image. These simple improvements would allow for more robust performance.

Figure 2 - Testing Results



Compared to state of the art methods for image recognition, my classifier did not perform well because of data issues. With a proper data pipeline however, VGG16 can be expected to perform at almost 99% accuracy. This is just as good as the current state of the art methods. Despite this, there are a few drawbacks to a VGG16 approach. First of all, the models used for VGG16 are quite large. The filesize of almost 500MB makes them a little unwieldy to work with. Furthermore, the structure of the VGG16 model is a little

antiquated. Training VGG16 by passing outputs from one layer to the next through all 16 layers takes much longer than many new methods as well. The same is true for making predictions as well. The YOLO image detection method can process 45 frames per second. This being said, while YOLO may be a good detector, VGG16 arguably would perform better on the classification aspects of many problems.

Overall I was very happy with my image pipeline. I thought that the MSER performed especially well, and if my classifier had been better, the final product would have been perfect. The poor classifier was really the weak link that had a major effect though. Essentially, the pipeline of my program provides the classifier with nice digit cutout candidates. It is up to the classifier to reject bad ones however. Because the classifier could not distinguish false positives very well, it was unable to reject several bounding boxes in each image. If the data pipeline issues were fixed, it would fix all the issues with the rest of the program. After hours and hours of testing, I was finally able to find the bug that caused my network to not train properly. I had been cutting out images, but not resizing any of them. They were all placed inside a solid color 224x224 background. When I made the images a more consistent size, the network began to train properly. I assume that the size differences between the images were too much for the network to learn anything about the digits specifically. It was easier for the network to simply model the data distribution. It is unfortunate that such a subtle error could cause so many issues with the whole product. There was sadly not enough time to redo much of the work, but I am proud of my analysis and results regardless.

VIII. Links To Videos

Presentation Video:
https://www.youtube.com/watch?v=sZ8RS6IgNag
Detection Demonstration Video:
https://www.youtube.com/watch?v=N6bJwB6gYvk

IX. Sources

[1] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” ArXiv e-prints, Sep. 2014. https://arxiv.org/pdf/1409.1556.pdf
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” Communications of the ACM, vol. 60, no. 6, pp. 84–90, 2017. https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf
[3] D. Gupta, “Transfer learning & The art of using Pre-trained Models in Deep Learning,” Analytics Vidhya, 01-Jun-2017. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/
[4] S. Lau, “Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning,” Towards Data Science, 29-Jul-2017. [Online]. Available: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1
[5] A. Karpathy, CS231n Convolutional Neural Networks for Visual Recognition. [Online]. Available: http://cs231n.github.io/neural-networks-2/