Capstone Project

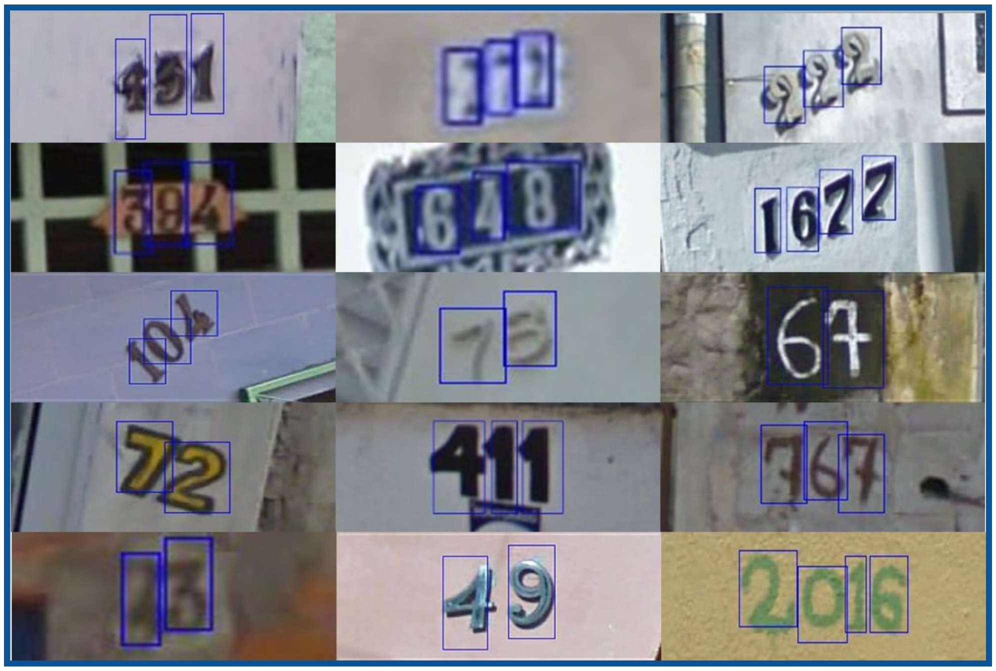
Machine Learning Engineer Nanodegree

Definition

Project Overview

This project aims at developing the skills for solving a Computer vision problem using Deep Learning approach. One of the problems in computer vision is to identify the digits and numbers from the images. This problem have a lot of practical applications from automated assistance for driving(reading speed limits etc.), vehicle localization based on surrounding images, extracting telephone number from visiting cards etc.

The particular problem being worked on in this project is to identify the house numbers from the images taken from a vehicle on road. The dataset we are using is called “The Street View House Numbers (SVHN) Dataset” available at http://ufldl.stanford.edu/housenumbers/. This dataset has been created from images captured for Google street view. All the images have been hand labeled for the house number and a bounding box for individual digits. A few of the images from dataset looks are shown below.

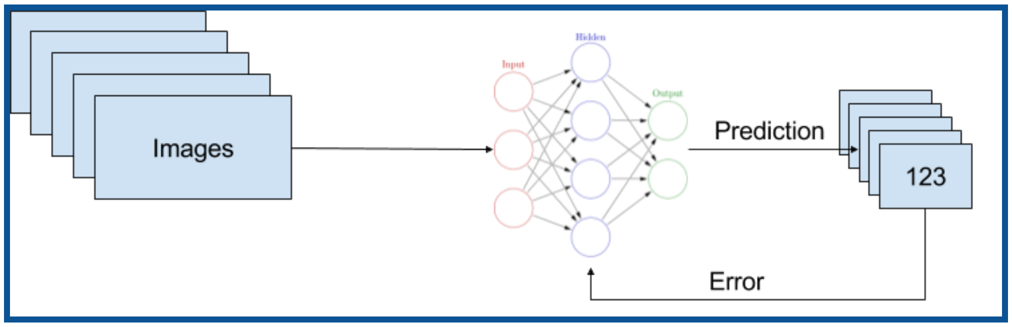


I am hugely interested in AI and Computer Vision is a great stepping stone to start. This the reason I picked up a Computer Vision problem. Moreover, the problem is neither too large like Imagenet that I need to spend weeks just to do an iteration of training hence limiting learning opportunities for debugging or improving the results. Nor is the problem as small as MNIST that gives a extremely good result for even a low capacity model.

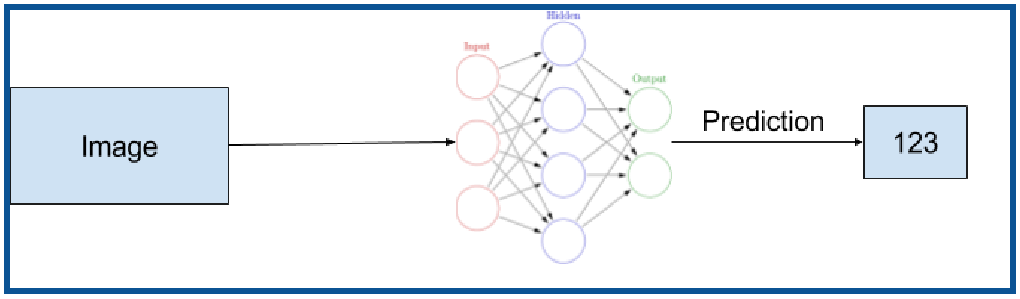
Problem Statement

As discussed in the above section, given the images with house number from the dataset, the project aims to extract the house number automatically without human supervision. As the images show, the style, the orientation and texture of number is quite diverse, it requires a computer vision algorithm to recognise the numbers. Our approach is to use Deep learning technique called Convolution neural network (CNN) to automatically “learn” to discriminate the digits from the background (gates, lawns etc) and also to be able to put the digits in order to form a number.

The system for achieving the goal of automatically recognizing digits consists of two phases on high level. First phase is “learning” phase, where use a statistical learning technique to learn some parameters for a model. Learning happens by first applying the input image and getting a prediction, the difference in the predicted and actual actual value is calculated using a loss function. This error is propagated back to the model to adjust its parameters. This process is applied iteratively multiple times which makes the parameters to converge to the values such that model can differentiate the images to generate correct label (to an extent). Then in the “inferencing” phase, we use the model with the learned parameters to predict the number in a given image.



Learning phase



Inferencing Phase

Metrics

Since the applications for reading house number would require the number to be correct completely, we need to use the metric which considers the result to be correct only when the model predicts all the digits to be correct (and correct number of digits). Further, a histogram for the digits at each position in the house numbers showed that the digits are almost uniformly distributed. This suggests that accuracy will be an appropriate metric and we don't need to go for F1 score (or precision and recall).

To measure the performance of the learned model, we need to use a metric which gives measure of how different the predicted output of model is from the actual number in the images. We use on 0-1 marking rule for a particular image i.e. if the prediction completely matches the real result, it is awarded 1 mark. In case the prediction is incorrect (even for a single digit), it is awarded 0 marks. We using this marking scheme over a large set of images and calculate the aggregated accuracy as

Accuracy = (Sum of marks based on above scheme) / (Total number of test cases)

For example if the Following are the predicted and actual digits
Predicted	Actual	Marks
123	123	1
125	123	0
12	123	0
23	23	1

Ground truth	Predicted	Marks
123	123`	1
123	125	0

123	12	0
23	23	1

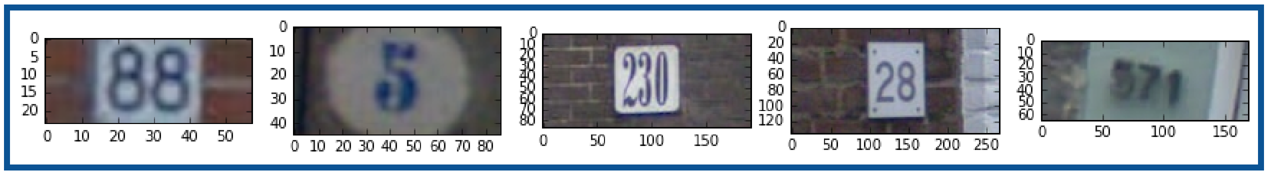
Accuracy for this case = 2/4 = 0.5

Analysis

Data Visualization and Exploration

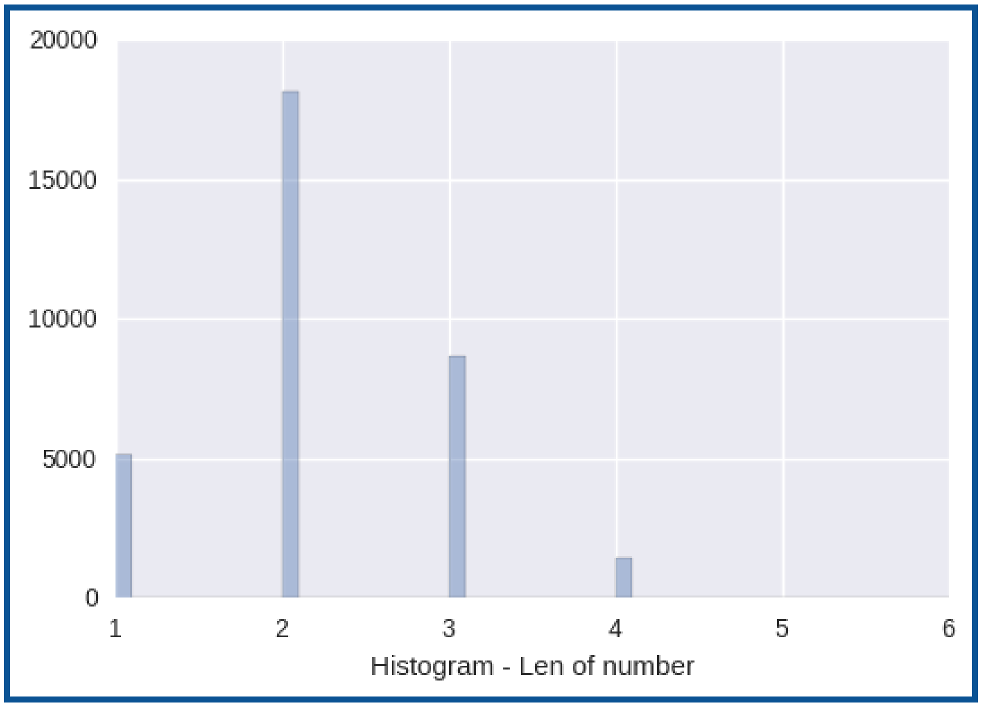
Before starting the actual implementation and tuning of Convolutional Neural Networks, we looked at the various statistics and characteristics of the dataset. It includes the visualization of the dataset.

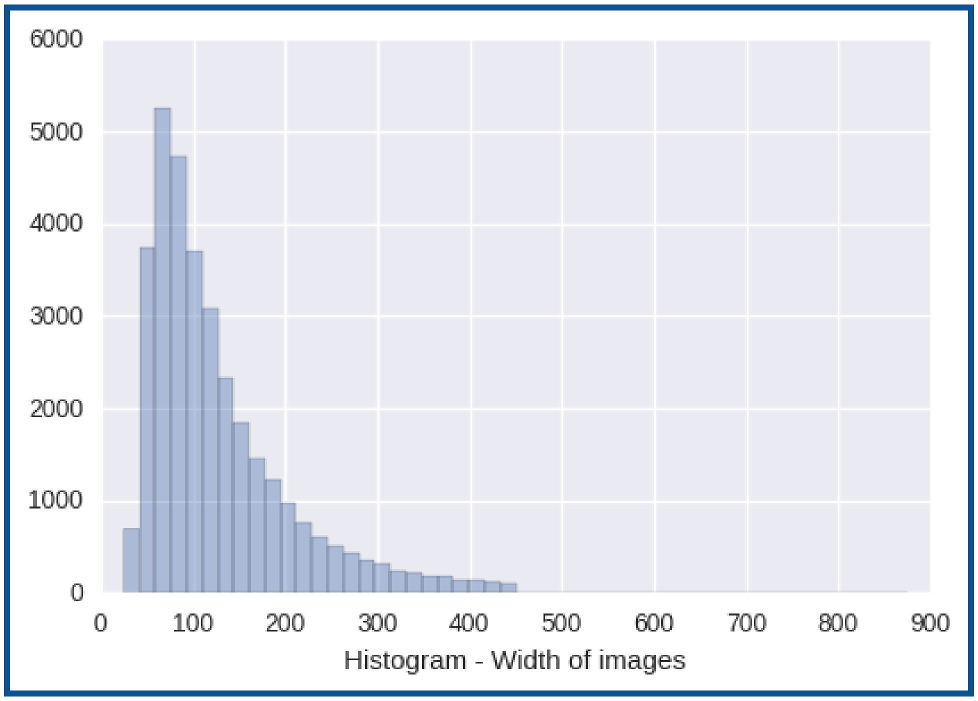
First step is to visually inspect the images and its labelled data. The code used to visualize the dataset is available in my github repo at (https://goo.gl/6jDDZp). 5 random images picked and visualized as

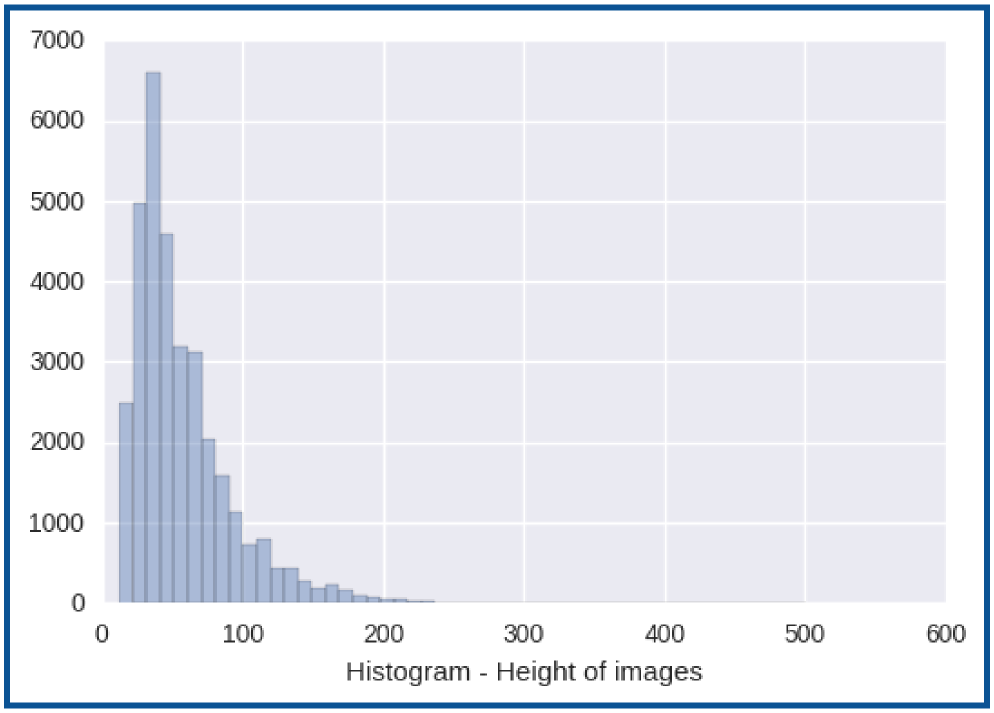


This shows the great variation in the texture, lighting conditions, width and height for the images. These variations make it harder to extract the numbers automatically. For CNNs, all the images are generally scaled to same width and height.

Further, to investigate the attributes of the images and the labels in the dataset, we plotted histograms for length of numbers in label, width of the images and the height of the images. The code for this analysis is shared at (https://goo.gl/Yax9AE). Length of the numbers (number of digits in the number) directs helps to find the number of digits we should try to target to recognise from the images. As most of the images contain numbers with less than 4 digits (~99.7%), we will predict upto 4 digits only.


20000
15000
10000
5000
0
1
2
3
4
5
6
Histogram - Len of number

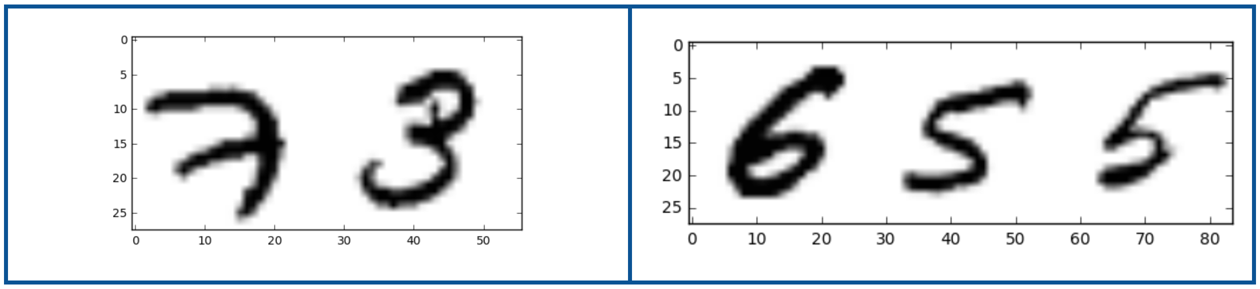

6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
700
800
900
Histogram - Width of images



As shown in the histograms for width and height of images, there is appreciable variation in the values. However, in the both cases, the distribution has a central peak. Hence we would use the values at mean for both of quantities and resize all the images to the values. The mean for width and height for the images in dataset is - (128.3 and 57.2).

Synthetic Benchmark

To get a feet about the nature of problem and how good the CNN can work on the problem, we created a synthetic data set using MNIST[1] dataset. This allows to create a dataset with numbers with controllable parameters (number of digits, presence of a blank in digits, variable/fixed size number images). Also this allows us to create arbitrary number of data samples to experiment with. The source code for generation of the dataset and its analysis is available at https://goo.gl/GbLB7m.



Sample generated images of width 2 and 3 respectively

Above figures show the some of the randomly generated images with fixed width (2 and 3 digits resp.).

Algorithms and Techniques

Given the huge success of Convolutional Neural Network for image classification problems, it makes CNNs to be ideal choice for this problem. The architecture and hyperparameters for the model will be determined using experimentation using cross validations.

Convolutional neural networks are a class of neural networks where each neuron takes inputs from a fixed, small window of neurons from previous layer, spatially located near each other. This helps each neuron to capture an arbitrary function of the features which are correlated spatially. One such use case is images. Images have a high degree of spatial information. This is the reason CNNs have been extremely successful in image related classification and regression tasks (e.g. all top entries in ImageNet competition are based on different architectures of CNNs).

The typical type of layers involved in Convolutional Neural Networks are

- Convolutional layers - These layers perform a 2D(or 3D) convolution over the layers and each convolution operation leads to the linear combination of spatially located neurons. We use 3X3 convolutional filters for these layers (based on VGG architecture) and its philosophy that 3X3 applied repeatedly in consecutive layers is effectively 5X5 and 7X7 kernels with more interleaved non-linearities[3].

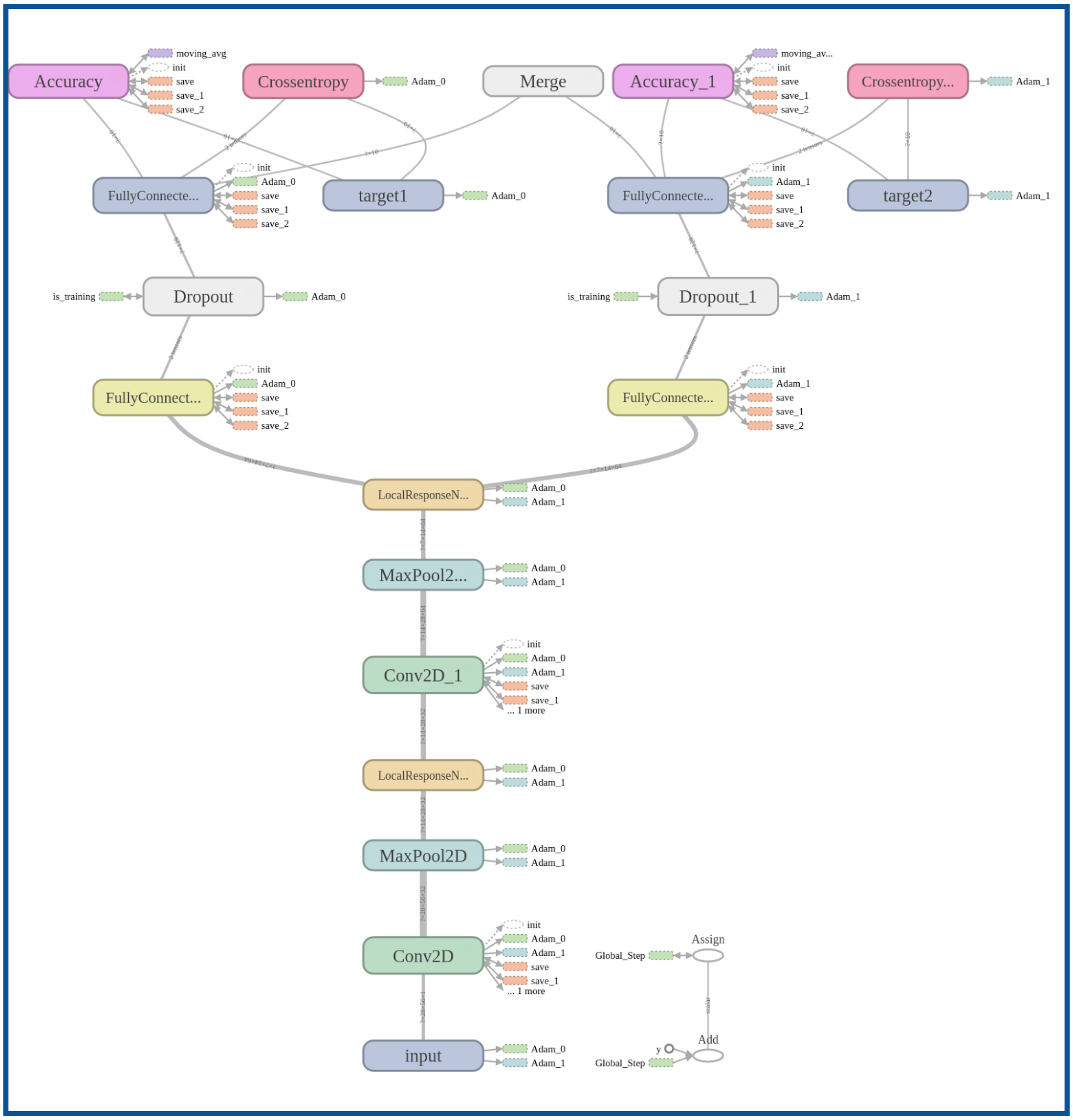
- Activation layers - These are layers which follow the convolutional layers and introduce nonlinearity to allow the system to capture non-linear relationships between neurons. We use ReLU activation layers which show a great performance without the diminishing gradient problems common in Sigmoid and TanH functions. Thus allows deep models to be learned

- Max Pooling layers - These layers downsample from the activation layers by selecting a neuron with highest activation value from the a typically 2X2 window. These layers help to reduce overfitting and reducing parameters and computational requirements for a model.

- Fully connected layers + Softmax layer - These layers are used for classification of the image based on the features it receives from the complete network of earlier mentioned layers. In these layers, each neuron from a level is connected to every neuron from earlier layer. The activation generated by these fully connected layers is passed through a set of softmax neurons which maps the input activation to probability for each target class. We use the argmax of these probabilities to assign the class to the input image.

For optimization of the Convolution Neural Network, we use variant of gradient descent algorithm. Adam (Adaptive Moment Estimation) is a variant of gradient descent where momentum and learning rate are tracked and decreased exponentially automatically. Adam is less sensitive to the learning rate set. Although Adam works great with most of the learning problems, no optimizer is universally better than the other. So we tried and evaluaged SGD and RMSProp during hyperparameter tuning too.

Following figure shows a sample CNN configuration used with synthetic dataset with fixed width of 2. The graph was created using Tensorflow's visualization tool Tensorboard. The CNN configuration uses 2 set of (Conv, MaxPool and Normalization layers). The resulted tensor generated captures the "features" in the image which can be used to classify the digits. This tensor is fed to two separate sets of 2 layer deep-fully connected layers which use softmax activation with 10 classes. The Loss function captures cross entropy loss across the classes. Each "head" created with the help of these two layers are optimized independently using Adam Optimizer.



Dataset consists of a set of input images which are converted to fixed size according to the statistics discussed in earlier section. The target value is provided as tuple of numpy ndarray where each ndarray at ith position in tuple represents the digit in the ith place in the target number. Each digit is converted to one-hot vector encoding to enable to calculate cross entropy for each digit separately.

Creating a simple histogram of digits used in the house numbers revealed that all digits are almost uniformly distributed and hence accuracy would be appropriate metric to use.

Benchmark

[2] reports the accuracy achieved to be 96% accuracy. I wish to achieve 92% with my exploration using CNNs. (I would love to reach 96%, however considering that the results are from Ian Goodfellow, who is the author of the book I am using to learn CNNs, I would rather be more realistic in goals).

Methodology

Data Preprocessing

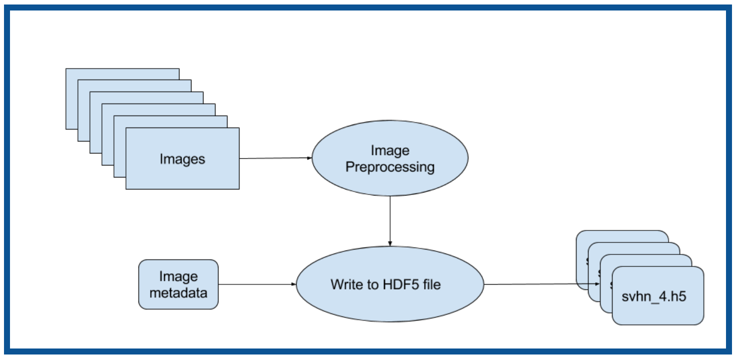
As described in [2], to improve the performance of system, we extract the portions from image containing the number. To do so, we take the union of the bounding boxes of individual digits and crop the image to the resulting bounding box. All the resulting images are resized to 32X32 pixels and are converted to gray scale.

We applying following transformation per pixel.
I(i, x, y) = (I(i, x, y) - mean(x,y)) / std (x,y)
This normalizes the the images which helps in better optimization using gradient.

We further apply rotation preprocessing (max 30 degree) to allow rotational invariance for learned model. This is important as the dataset seems to have a lot of numbers which are not horizontally aligned. We did not apply image whitening and PCA as they have not shown to improve results for CNN in recent research.

Data Pipeline setup

To use the data effectively for repeated experimentation, we created following pipeline for handling data. All the image files are loaded and sorted according to the number of digits then cropped, converted to gray scale and stored back into a hdf5 record. This helps to speed up the experimentation as compared to preprocessing directly images. The code for data reading/dumping utilities are shared at https://goo.gl/3Fxk1q.



Implementation

As mentioned above, we use CNNs to solve the problem. For the CNN implementation, we used TFLearn.org library, which is a wrapper around the TensorFlow library for deep learning. Using TFLearn makes code less verbose and more readable.

We first used a simple configuration with small number of layers to experiment and debug the basic problems (shown in the "Algorithms and Techniques" section. This allowed us to quickly remove problems other than the CNN architecture. For the real experimentation, we used a VGG like architecture and created multiple variants based on the number of layers it supported. Following table shows the architecture configurations used (conv_2d(n) represents a conv layer with n output activation layers, max_pool_2d(n) represent max pooling layer with nXn strides.

SVHN-1	SVHN-2	SVHN-3	SVHN-4	SVHN-5

conv_2d(32)
conv_2d(32)
max_pool_2d(2)
	conv_2d(32)
	conv_2d(32)
	max_pool_2d(2)
	
	conv_2d(64)
	conv_2d(64)
	max_pool_2d(2)
		conv_2d(32)
		conv_2d(32)
		max_pool_2d(2)
		
		conv_2d(64)
		conv_2d(64)
		max_pool_2d(2)
		
		conv_2d(128)
		conv_2d(128)
		conv_2d(128)
		max_pool_2d(2)
			conv_2d(32)
			conv_2d(32)
			max_pool_2d(2)
			
			conv_2d(64)
			conv_2d(64)
			max_pool_2d(2)
			
			conv_2d(128)
			conv_2d(128)
			conv_2d(128)
			max_pool_2d(2)
			
			conv_2d(256)
			conv_2d(256)
			conv_2d(256)
			max_pool_2d(2)
				conv_2d(32)
				conv_2d(32)
				max_pool_2d(2)
				
				conv_2d(64)
				conv_2d(64)
				max_pool_2d(2)
				
				conv_2d(128)
				conv_2d(128)
				conv_2d(128)
				max_pool_2d(2)
				
				conv_2d(256)
				conv_2d(256)
				conv_2d(256)
				max_pool_2d(2)

				conv_2d(256)
				conv_2d(256)
				conv_2d(256)
				max_pool_2d(2)

All convolutional layers in above configurations used ReLU activation layers and filter of size 3X3. We used “Adam” as learning algorithm and 0.001 as learning rate.

Refinement

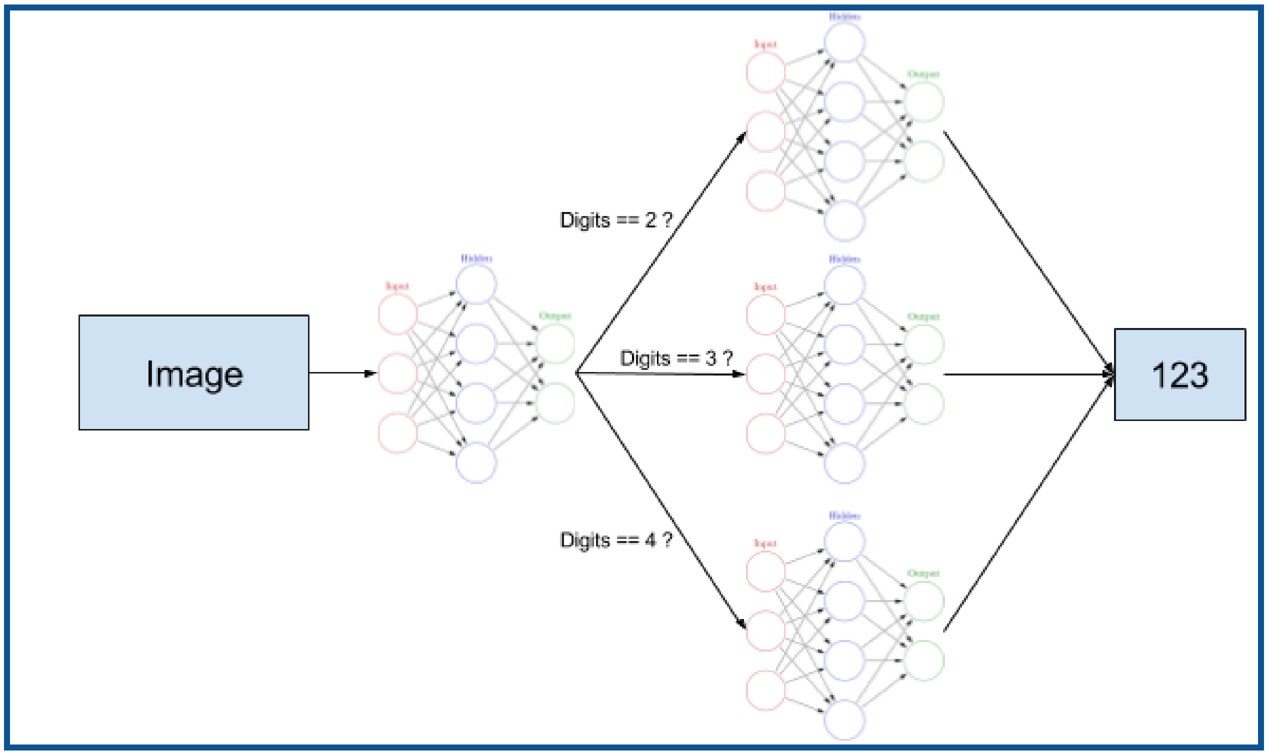
We use following two approaches for creating the CNN pipeline.

1. Use a two stage pipeline using weak CNNs to boost results. The first stage predicts the number of digits in the number (length of number) followed by choosing of a CNN based trained to handle the specific number of digits.
2. End to end trained CNN to directly predict the number.

- *Two stage pipeline* - The two stages pipeline (as shown in the diagram below) is based on the methodology discussed in [2]. It first classifies the image to number of digits present in the image in the first stage. In the second stage, a particular model is picked based on the number of digits predicted in first stage. The softmax function on the last layer of the models in second stage effectively calculates the probability

  $$p(S|X) = p(N|X) * P(S|N)$$

  X represents an input image
  S represents the number in the image
  N represents the number of digits in the image



Implementation code for this methodology is available at https://goo.gl/3Aw8Em. A quick analysis for the trained system based on above models showed that prediction for number of digits was around (~97%) and max accuracy of all the models in second stage was (~95%) (more details in “Results” section). This bounds the max accuracy that we would achieve is (0.93 X 0.95 = ~88%) which is way below our expected benchmark. We did not spend more time trying to tune this network further due to limited resources and low expected ROI.

- *End to End training* - In this approach we directly learn reporting the house number. We assume the number of digits to be less than equal to five (and will be misclassified in case of higher number of digits). This does not affect the result much as the number of such cases in extremely less as shown in histogram in analysis section. Also we use the number 10 to represent the missing value (always appended at the end). As we only assume the prediction to be correct if all the digits are predicted correctly, a good measure to calculate the final accuracy is given by min of accuracy for individual digits.

The work required multiple iteration of debugging and experimenting to get to the working phase. Following issues were faced during the coding and debugging.

- Two stage pipeline approach - During this approach after trying a lot simple CNN models, the accuracy remained at 10% (same as random chance). The problem later on turned out be that I was using full images to find the house numbers and the simple models were not able to capture the variance in the large dataset and was thus underfitting. Cropping the images to the Bounding boxes of the numbers solved this problem.

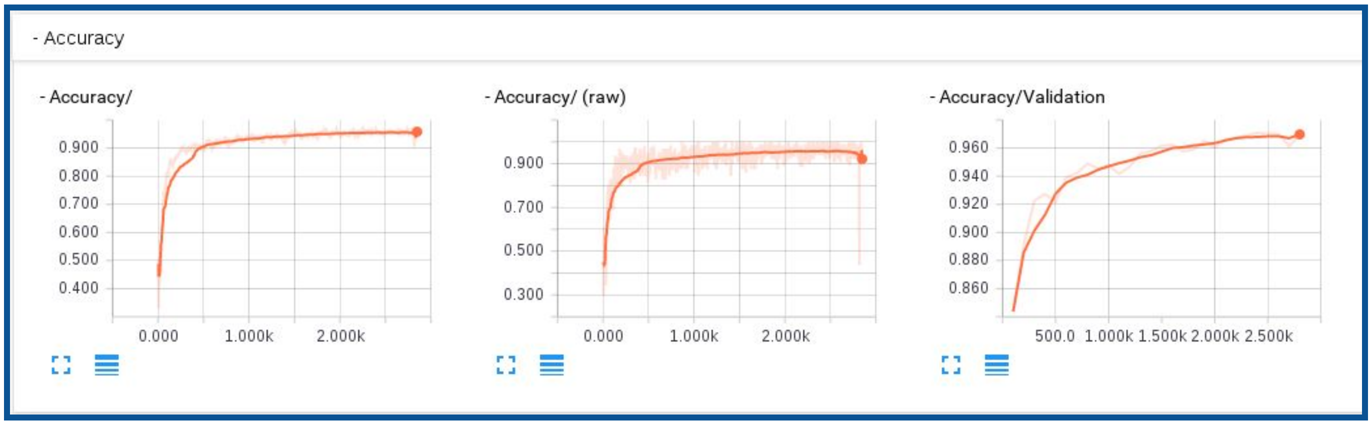
- End to End approach - A similar problem occured while using VGG network where the model was stuck at 10%. The problem this time was high learning rate (0.1). Interestingly, while working on simpler model, this problem did not occur even with high learning rate and although the accuracy was stuck at ~70% (underfitting due to limited model capacity), it was better than getting stuck at 10%. The problem just disappeared when the learning rate was set at 0.01.

Results

Model Evaluation and Validation

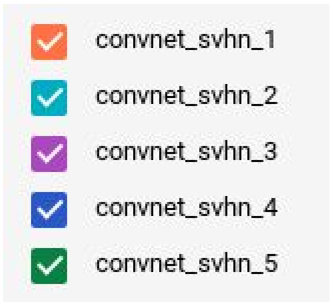
Two stage pipeline -

Following two figures show the accuracy achieved on dataset for first stage (predicting the number of digits in the input. As shown, the validation accuracy achieved is (~ 96%).

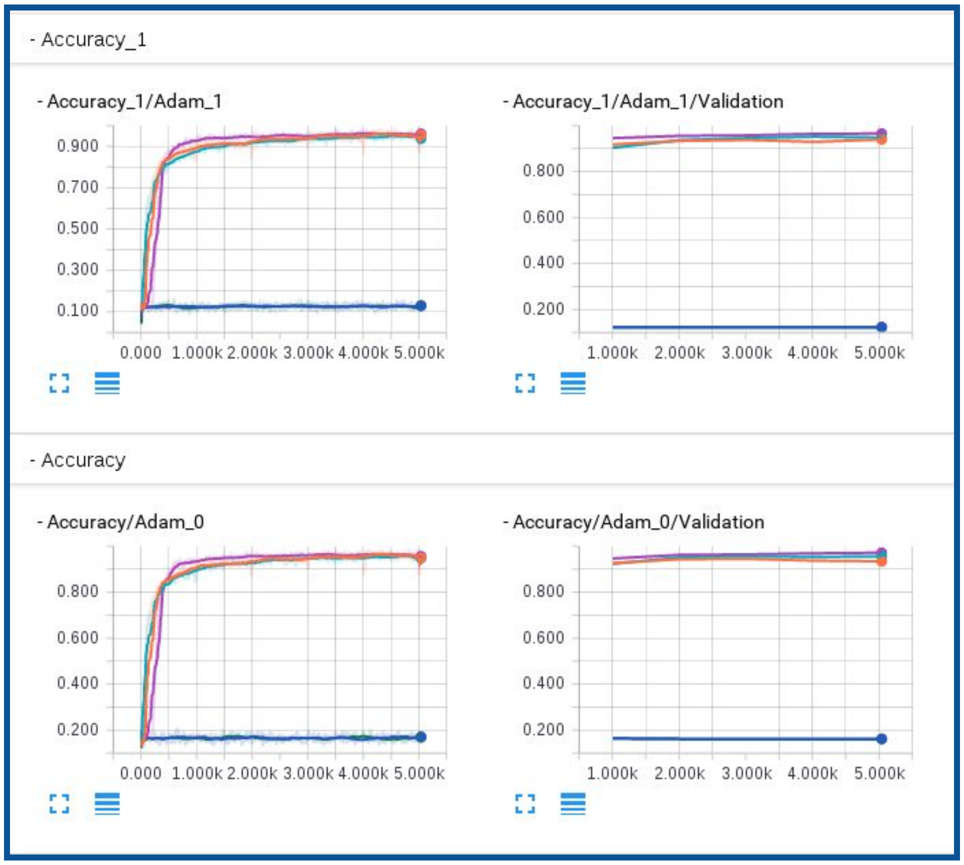


Results of 1st stage for Two stage pipeline method

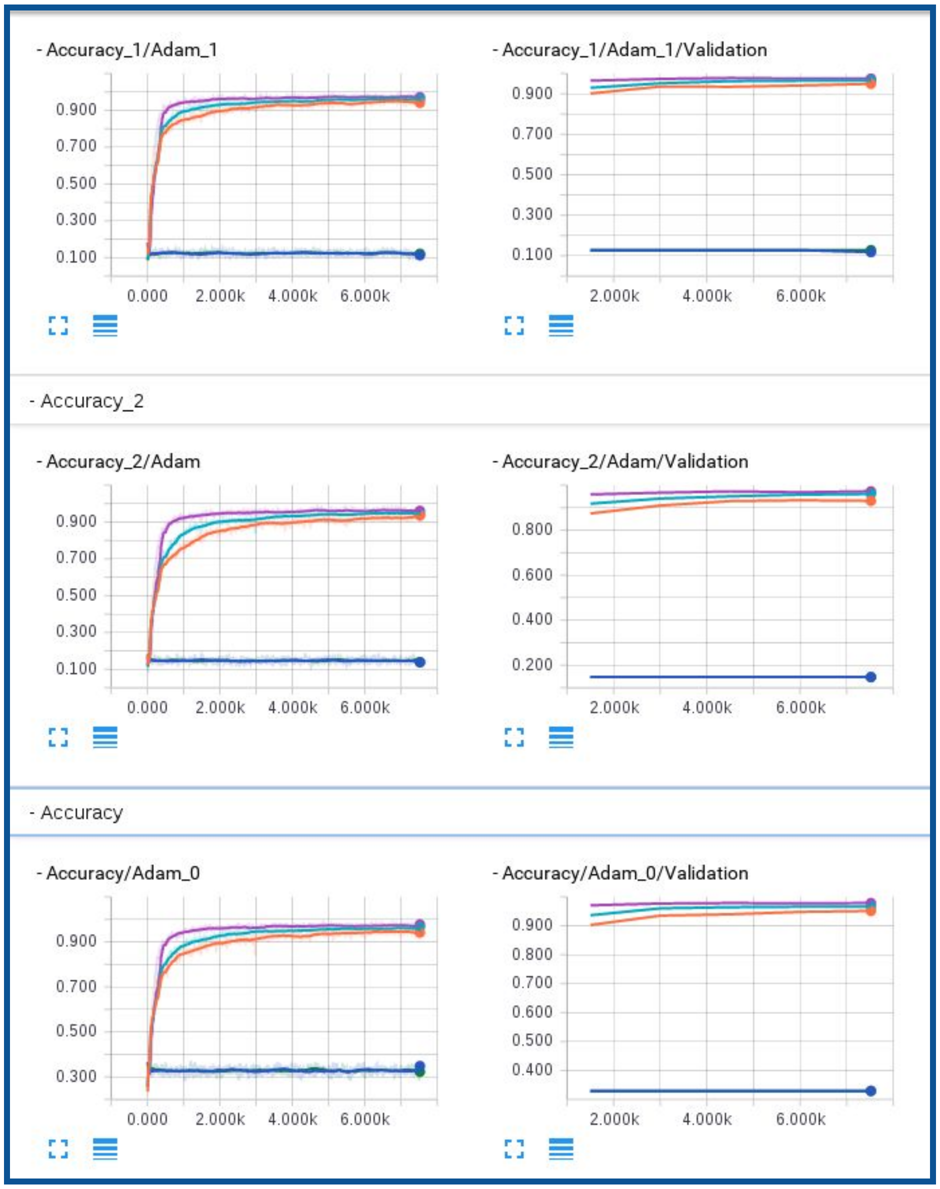
Following figures show the performance of the second stage for different num of digits. Each experiment tried the earlier listed VGG like networks. The legends (color coding) used in all results is as follows



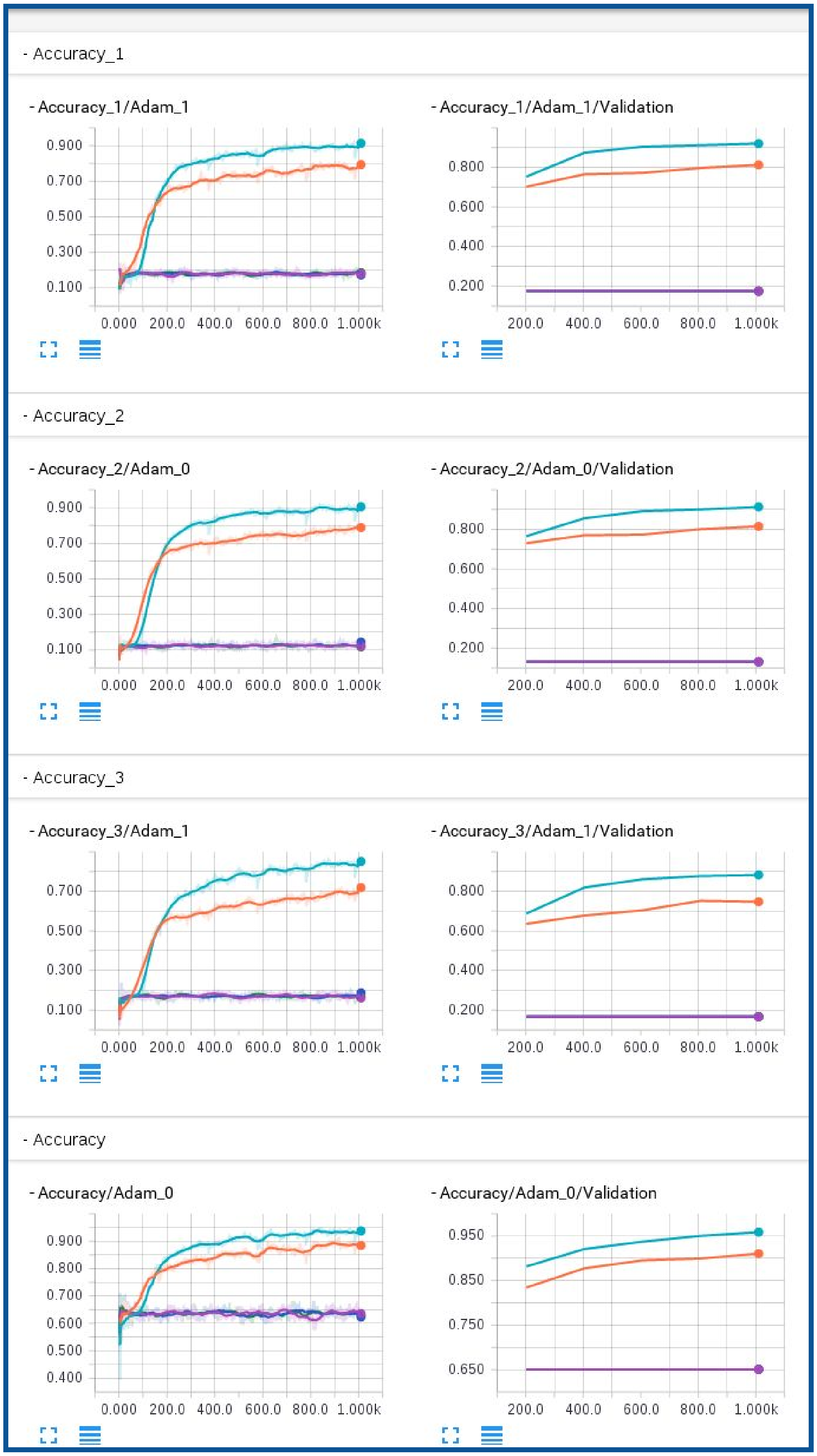
🟧 convnet_svhn_1
🟦 convnet_svhn_2
🟪 convnet_svhn_3
🟦 convnet_svhn_4
🟩 convnet_svhn_5



Second stage results for 2 digits classifier



Second stage results for 3 digits classifier


- Accuracy_1
- Accuracy_1/Adam_1
0.900
0.700
0.500
0.300
0.100
0.000 200.0 400.0 600.0 800.0 1.000k
- Accuracy_1/Adam_1/Validation
0.800
0.600
0.400
0.200
200.0 400.0 600.0 800.0 1.000k
- Accuracy_2
- Accuracy_2/Adam_0
0.900
0.700
0.500
0.300
0.100
0.000 200.0 400.0 600.0 800.0 1.000k
- Accuracy_2/Adam_0/Validation
0.800
0.600
0.400
0.200
200.0 400.0 600.0 800.0 1.000k
- Accuracy_3
- Accuracy_3/Adam_1
0.700
0.500
0.300
0.100
0.000 200.0 400.0 600.0 800.0 1.000k
- Accuracy_3/Adam_1/Validation
0.800
0.600
0.400
0.200
200.0 400.0 600.0 800.0 1.000k
- Accuracy
- Accuracy/Adam_0
0.900
0.800
0.700
0.600
0.500
0.400
0.000 200.0 400.0 600.0 800.0 1.000k
- Accuracy/Adam_0/Validation
0.950
0.850
0.750
0.650
200.0 400.0 600.0 800.0 1.000k

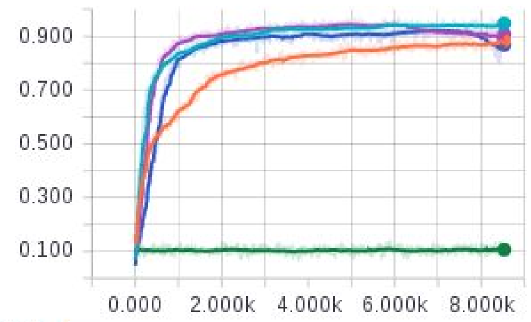
Second stage results for 4 digits classifier

Based on the above results, we chose the svhn-2, svhn-3, svhn-3 configurations for 2 digit, 3 digit, 4 digit classifiers respectively. Also, as the training accuracy closely follows the validation accuracy, these models did not show the over fitting problem.
Another observation is that most of the models were unable to learn anything significant for svhn-4 and svhn-5 case. The probable reason is that the size of activation maps decreases to half in height and width with application of a pooling layer in the models above. Since VGG is designed for 224X224 input image size and we are using much smaller (32X32 image size). So the later layers leaves the activation maps so small that it is unable to capture the input features necessary for distinction. We also tried to reduce the learning rate to check if the problem was divergence in learning, however it did not change the results, reinforcing the logic mentioned above.

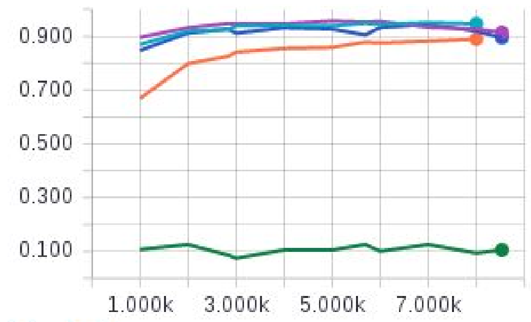
End to End Learning - This case reported much better accuracy. The max accuracy was achieved 92% which is near to our expected accuracy (for svhn-3 configuration) . In this case, however, only svhn-5 showed the poor learning performance. Svhn-4 was still able to achieve appreciable results. The models did not show any signs of overfitting even without any specific efforts for the same (drop-outs, batch normalization etc.)

- Accuracy_1

- Accuracy_1/Adam_1

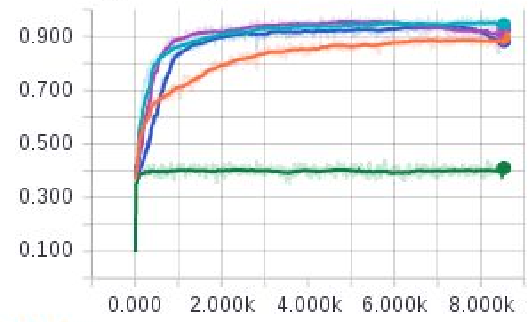


- Accuracy_1/Adam_1/Validation

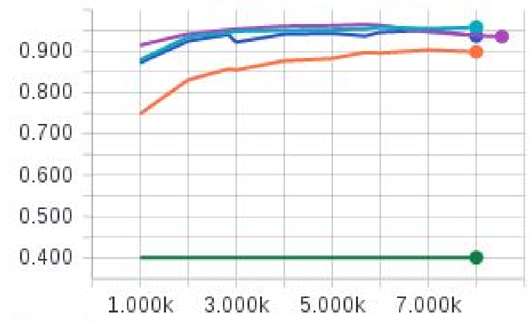


- Accuracy_2

- Accuracy_2/Adam_0

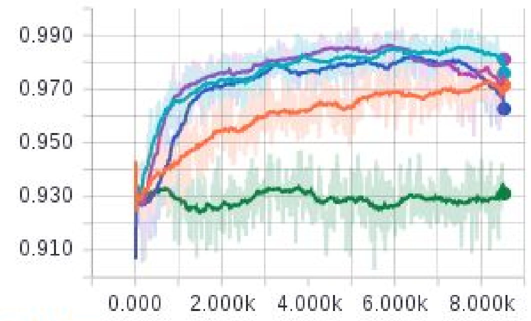


- Accuracy_2/Adam_0/Validation

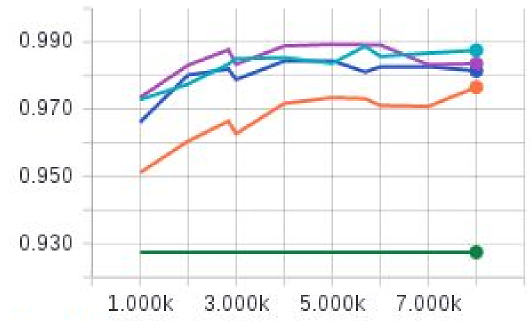


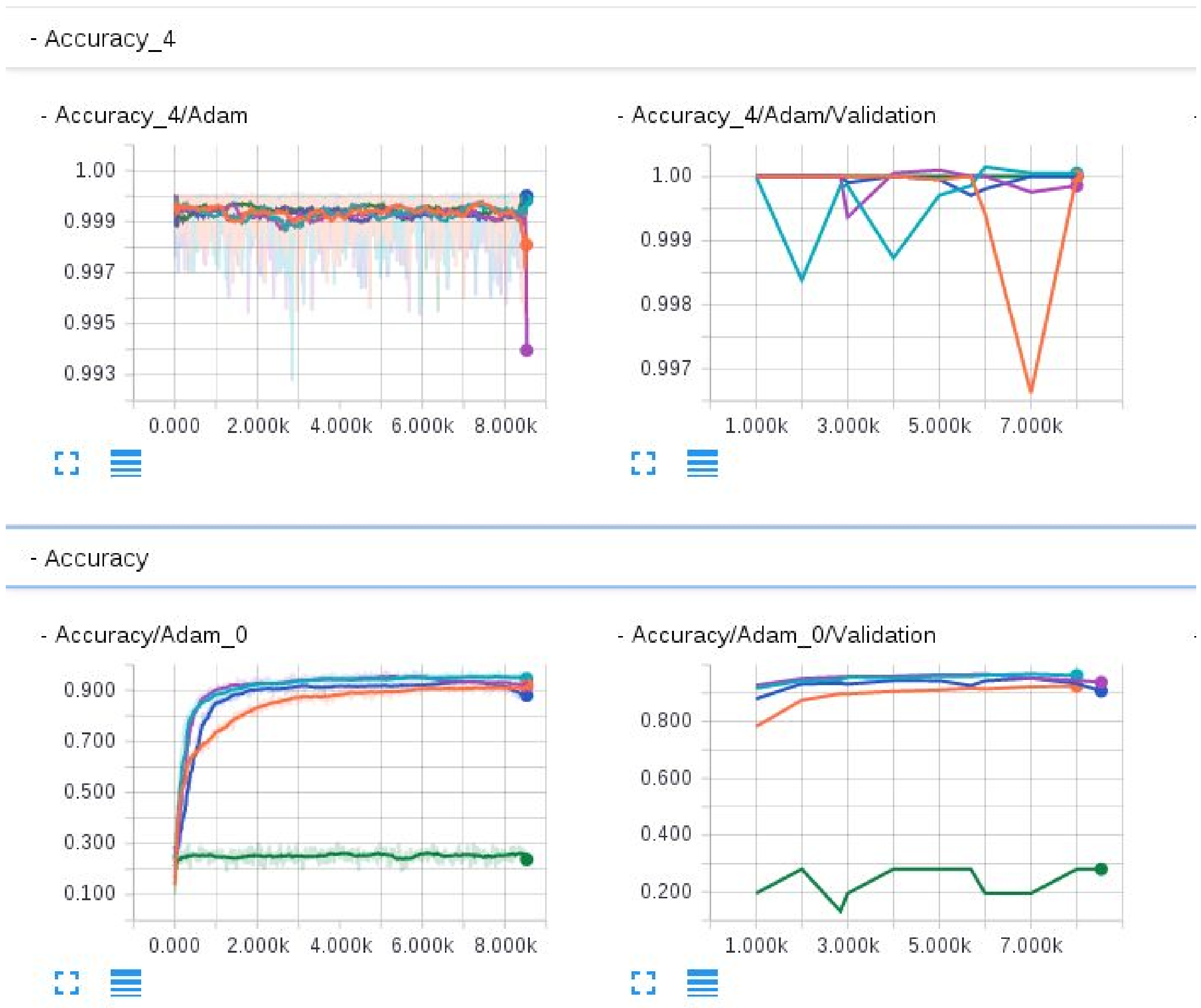
- Accuracy_3

- Accuracy_3/Adam_1



- Accuracy_3/Adam_1/Validation



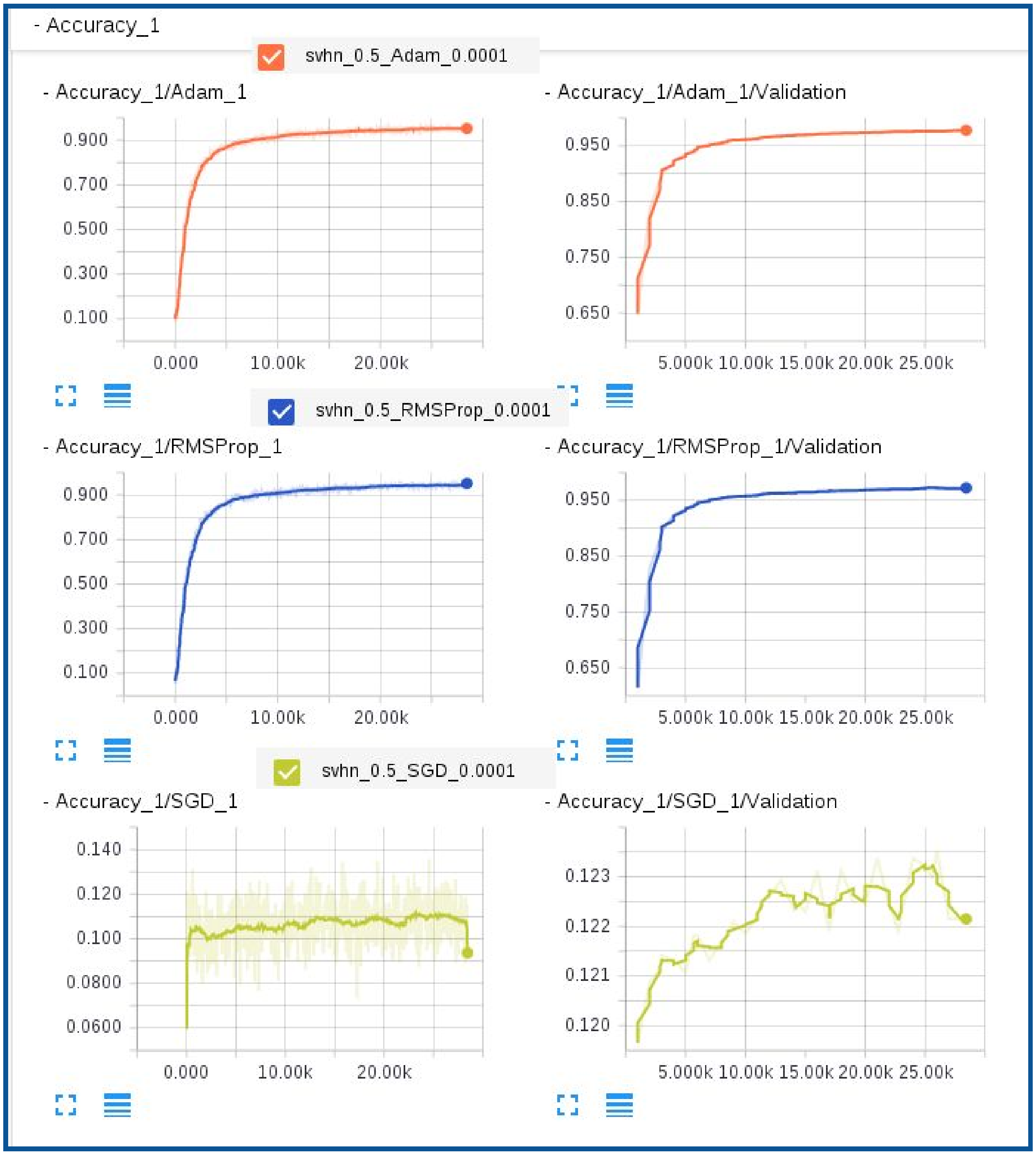


Hyper-parameter optimization

We performed search over a small set of hyper parameter for end-to-end learning model and use it as guiding stone to improve performance further. We performed search over following hyper-parameters

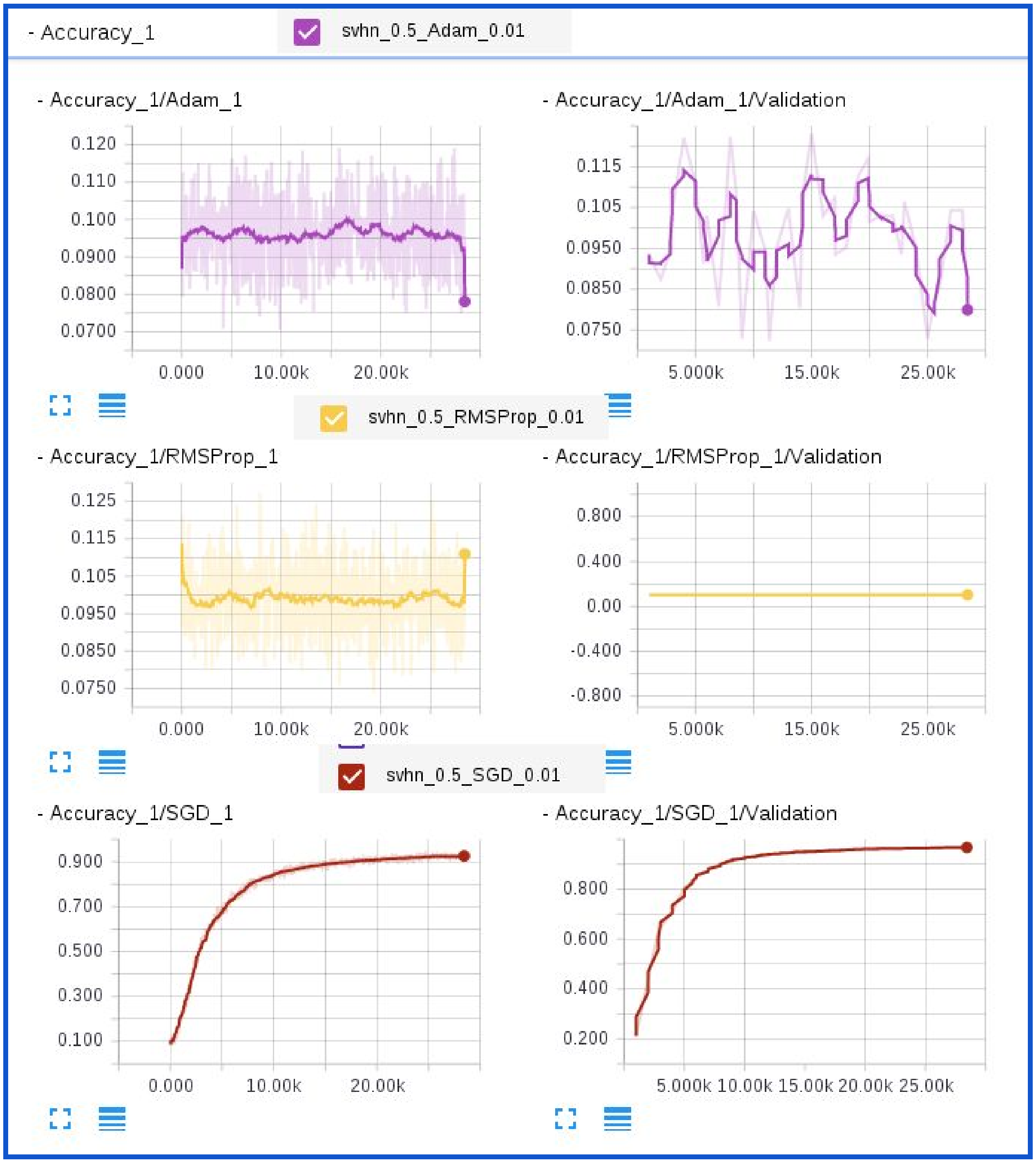
1. Dropout ratio - 0.5, 0.7, 0.9
2. Optimizer algorithm - Adam, RMSProp, SGD
3. Learning rate - 0.0001, 0.01, 0.1

Some of the noticeable trends are shared below



Hyperparameter optimization - Dropout = 0.5, Learning Rate = 0.0001

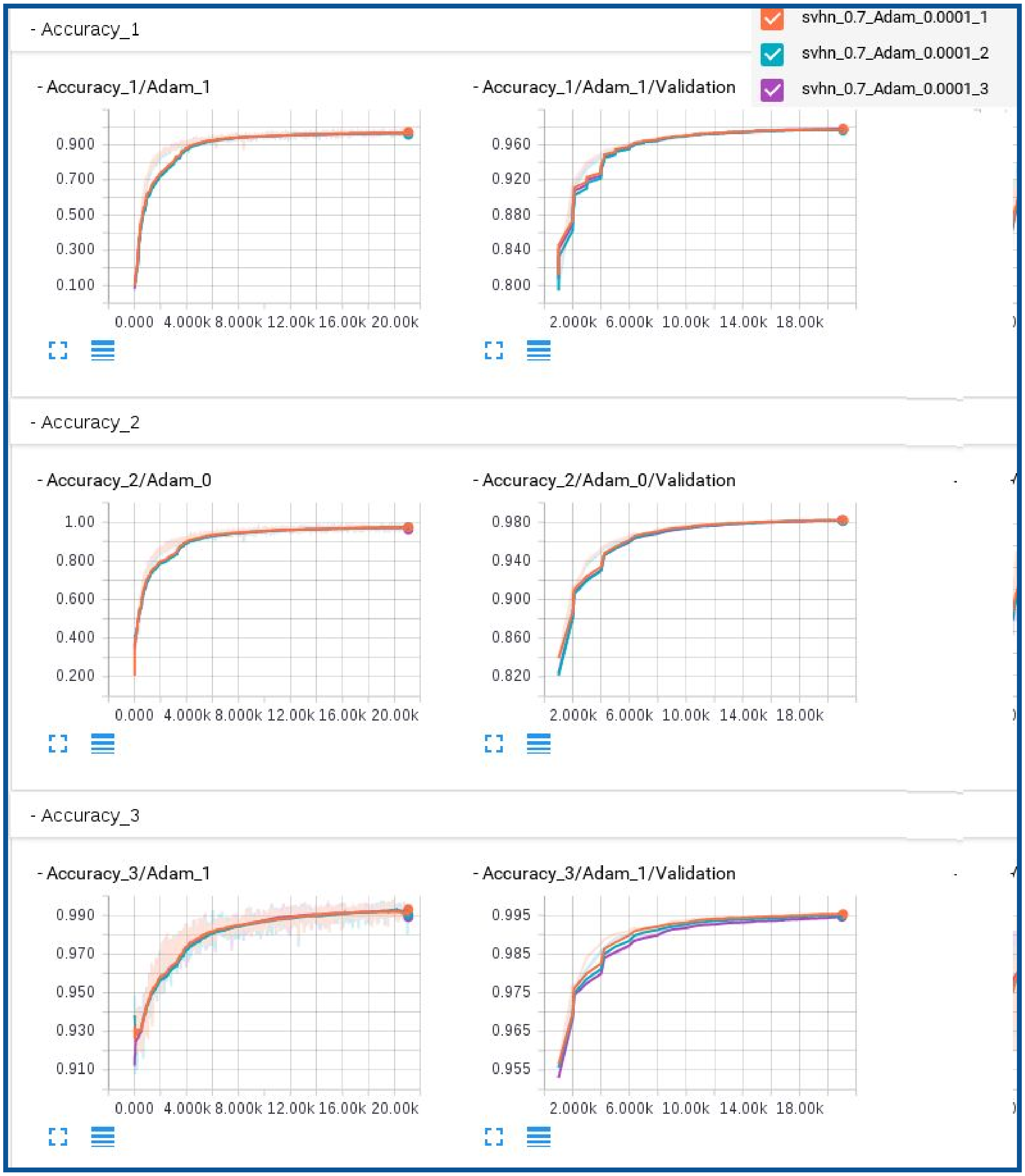
As shown above, at very low learning rate, both Adam and RMSProp achieve appreciable results. However SGD struggles to learn. We tried to decrease learning rate to 0.01 and try again. Following figure summarizes the results for the same.



Hyperparameter optimization - Dropout = 0.5, Learning Rate = 0.01

These results at high learning rate of 0.01, Adam and RMSProp struggle. However, SGD is able to appreciable learn with this configuration. However the results were a bit inferior to Adam and RMSProp in general (which showed almost similar results).

Since there was almost no signs of overfitting, we used lower Dropout ratio (0.9) for further results. Keeping the learning rate as 0.0001, we used Adam as optimizer for final resulting model after hyper parameter search. To improve the confidence in the model, we used 3-fold cross validation run for 10 epochs and capture the final result as shown in the figure below (for first three digits).



Final results with 3-Fold CV. Dropout = 0.9, Optimizer = Adam. Learning Rate= 0.0001

As shown above, the all three 3 fold runs track each other very closely installing confidence in the model. Also, the training and test accuracy tends to be almost same, hinting to no overfitting.

The final accuracy achieved with the above configuration is ~97.5% , which is way higher than benchmark value we planned.

Justification

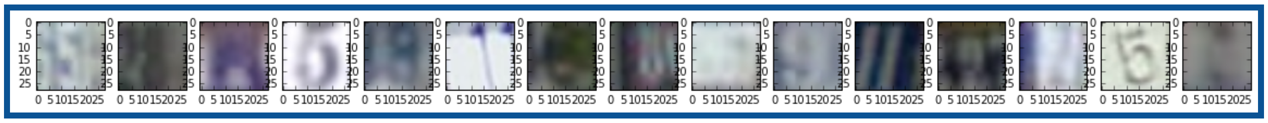
The final results, beats the benchmark results. The possible reason for such high performance number can be that [2] did not use the "extra" dataset available at SVHN dataset home. The use of this extra data could have provided us the extra boost. This reinforces the common wisdom in deep learning that more and better data is generally more important than the models itself.

Also to be emphasised that the current problem solved is much harder than the single digit dataset problem which is solved in most of the papers stating the state of art results (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#5356484e).

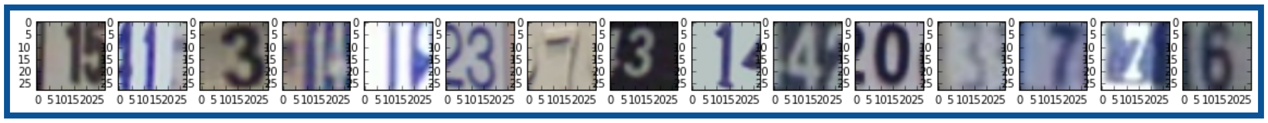
Conclusion

Free-Form Visualization

To further visualize and find out what are the images which are harder to classify than the others, I implemented a simple classifier which works on the single digit dataset. The dataset is a collection of single digit cutouts from the original dataset and is available along with the SVHN dataset. The model used is a CNN implementation using Tensorflow library. The code is available at https://goo.gl/dVFviR. This code is derived from a sample examples available with tensorflow documentation. The CNN is a modest (3 conv layer + 1 fully connected layer) model, learned with Adagrad optimizer and learning rate of 0.01. We did not do any hyperparameter tuning for the model. This model uses softmax layer at the end to predict the class probabilities. If the model is able to assign a probability of more than 0.8 to a label for an image, we consider the image to be easy to classify else it is considered harder to classify. Following figure shows the difference in the hard to classify vs the other images.



Images - hard to classify


Images - easier to classify

As shown above the hard to classify images are either if poor quality or too much blended in the background which can challenge even a human to classify the image correctly.

Reflection

We worked through two schemes using CNN to solve the house number detection problem. We found that the end-to-end learning approach provided better results. One possible reason for low accuracy in two stage approach might be the division of dataset into multiple smaller datasets (for each subproblem of classification for fixed num of digits). While in the case of end to end learning, whole dataset contributed to the learning of appropriate filters for Convolutional layer to extract appropriate features.

Most difficult part I encountered in project was the RNN approach. Recent papers have hinted that using RNNs can be advantageous for image recognitions problems too. Which I tried using for building a simple LSTM based classifier (https://goo.gl/GAvPa1) for single digit recognition problem. However, could not get it working. I decided to master the CNNs first and then later learn to mix CNN and LSTM to improve the results (some day !!!), similar to approach in [5]. Another issue which I should have had resolved earlier was using a pipeline to preprocess the data and dumping into a intermediate HDF5 file (rather than reading from the image file every time). This speeded up my experiments by at least 2-3 times faster and helped to do much more experiments.

The most interesting part was debugging using the tensorboard which allowed to debug the problems by visual inspection. TFlearn.org library generates a lot of default graphs and histograms which hints to the layer which has problem. It gave me a lot of insight into the CNNs that would have not been possible by just text log descriptions. This problem has set a confidence level for me to take up real life (or rather a bit refined Kaggle) problems related to computer vision.

Improvement

We were not able to fine tune the architecture using hyperparameter search space. It is due to lack of resources (high end GPU) available currently and the size of the datasets makes it harder to do the search effectively.

Another technique that could be explored is use of RNNs. RNNs have recently been applied to computer vision and as RNNs learn the sequences, it would be interesting to see the use of RNN to learn the digits moving from left to right in an image.

References

[1] LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[2] Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., & Shet, V. (2013). Multi-digit number recognition from street view imagery using deep convolutional neural networks. arXiv preprint arXiv:1312.6082.

[3] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] He, Pan, et al. "Reading scene text in deep convolutional sequences." arXiv preprint arXiv:1506.04395 (2015).