Comp540 Term Project Report

Donnie Kim (dk17) & Martin Zhou (cz16): Finished_in_TH

April 19th, 2017

1 Introduction

SVHN is a real-world image dataset collected from house numbers in Google Street View images. It can be compared to MNIST, a collection of hand-written digits, but has much more labeled data and comes from a significantly more challenging real world problem; unlike MNIST which has no background, SVHN asks users to recognize digits and numbers in natural scene images. Each SVHN image is 32 by 32 in pixels and has a label ranging from 1 to 10. The training and testing data set sizes are the following:

- Training data: 73,257 labeled digits
- Testing data: 26,032 labeled digits
- Extra Training data: 531,131 labeled digits

Extra Training data is there for more images for user to utilize.

2 Data Visualization

All the data sets came as a csv file with format of (W,H,C,B) where W = width, H = height, C = channels, and B = Batch (Data) size that have been flattened (or lineariezd). For instance, training data is a $73,257 \times 3072$ matrix where each row vector is in the following fashion:

$$px0\_0\_R, px0\_0\_G, px0\_0\_B, px0\_1\_R, \ldots, px31\_31\_R, px31\_31\_G, px31\_31\_b \tag{1}$$

where the format is in pxH_W_C (H = height, W = width, C = channels). After quick analysis, we learned that intensity of each channel ranges from 0 to 255 (8bit image).

In order to ensure that the file was indeed indeed in the right format, we transformed the flattened vector into 32 by 32 by 3 image and visualized it. Fig 1 shows some examples of the visualized images:

Figure 1: Examples of SVHN data

3 Choice of the model: Convolutional Neural Network

For MNIST, Yann LeCun built the very first convolutional neural network (CNN), LeNet-5, that achieved a test set error rate of 0.95% [1]. Knowing that the SVHN data set is quite comparable to that of MNIST, we assumed that a top performing CNN model for MNIST would also work fairly well for SVHN data set. In order to build a CNN, we decided to use Tensor Flow [2], an open source software for machine learning/deep learning.

4 Modeling of CNN

Here, we documented our evolution of our model from Tensor Flow MNIST tutorial to our final SVHN model.

4.1 Intensity normalization

Before we fed our training data set (or extra data set) into the architecture, we normalized the intensity of images by dividing by 128 and subtracted by 1 such that all the intensities fell between -1 and 1. We considered subtracting the mean from each image, but this is only valid under the assumption that the statistics for each data dimension follow the same distribution. However, each pixel has 3 channels and it is generally not true that RGB follow the same distribution. Hence, we simply applied a "simple" scaling scheme (Please refer to Data Pre-processing under Technical Reference).

4.2 Model 1: CNN from Tensor Flow MNIST tutorial

We started out with the model implemented in the Tensor Flow tutorial for MNIST CNN model. The architecture of this model is the follwoing:

$$\text{Conv1} - \text{Pool1} - \text{Conv2} - \text{Pool2} - \text{FC} - \text{Dropout} - \text{Softmax}$$

- Conv1: 5x5 convolution filter with stride of 1 in 3 channels. 32 feature maps are produced. Padded the input image to keep the input dimension.

- Pool1: Max pooling with a window of 2x2. After the pooling, the dimension is in 16x16 for all 32 features.
- Conv2: 5x5 convolution filter with stride of 1 in 3 channels. 64 feature maps are produced. Padded the input image to keep the input dimension.
- pool2: Max pooling with a window of 2x2. After the pooling, the dimension is in 8x8 for all 64 features.
- FC: Fully connected layer with 1024 nodes. 8x8 64 features are flattened and connected to this FC layer.
- Dropout: Drop the connections on Fully connected layer with probability of 0.5.
- Softmax: Our loss function.
- Activation function: ReLu.
- Bias: Gaussian with standard deviation of 0.1
- Learning Algorithm: Adam Optimizer with learning rate of 1e-4.
- batch size: 64 for training. For validation, we just loaded all (no batch).

We also implemented one-hot-encoding such that all the classes are in matrix format. This does not affect the accuracy of the model, but it helped us to organize and understand our code.

For train and validation split, we used first *70,000 for train data set*, and the remaining *3,257 images for validation* data set.

After running for 10 epochs (of the training data set), our model achieved an accuracy of

- training: 0.89062
- validation: 0.88437
- test: 0.89436

Training accuracy is based on the batch, hence it fluctuates depends on what kind of batch the machine is looking at. This is true for all the other models we developed below.

As expected, the model for MNIST yielded a reasonable accuracy for SVHN data set as well due to similarity in their classification problem.

Note: This implementation was built before the first draft due date (Feb 10th), but we incorrectly implemented one-hot-encoding for labels such that we kept getting really low accuracies on both train and validation set. We initially suspected that our network was too shallow such that the architecture cannot learn enough features to classify the digits. We also did a lot of different hyper-parameter adjustments, such as adding more convolutional layers, increasing filter numbers, increasing filter size, etc., yet none of them could give us accuracy higher than 5% which was very alarming. Then, we finally figured out that we simply messed up our one-hot-encoding on late March. Once fixed, we could obtained the aforementioned accuracies.

Once we observed that the architecture can actually classify in a reasonable fashion, we increased our number of epochs to 50 with decay learning rate added on our learning algorithm (0.97 every 10,000 steps). The purpose of decay is that once the model finds the minimum, by reducing the learning rate, it can further go down the valley, hence yields a lower cost. We also tried different filter sizes of 3, 5, and 7 (5 did best) and also implemented local response normalization layer (which did not improve both training and validation set accuracy not much). With this set up, our model achieved an accuracy of

- training: 0.91046
- validation: 0.90802
- test: 0.90276

4.3 Model 2: VGG variation 1

Because we could not improve the accuracy further by tuning hyper-parameters and making minor changes to the code, we looked into ways of making our neural network deeper. We tried to implement an architecture that resembles the VGG 16 model [3]. The key characteristics of VGG19 we tried to mimic was *convolving twice or even three times before max pooling*. The new architecture can be seen below:

Conv1 – Conv2 – Pool1 – Conv3 – Conv4 – Pool2 – FC – Dropout – Softmax

We kept the same hyper-parameter as model 1 except, for additional convolutional layers, we simply kept the same parameters until max-pooled (i.e. All convolutional filters are in size of 5x5 with a stride of 1. Conv1/2 yields 32 and Conv3/4 yields 64 feature mappings).

With the above architecture, we achieved the following results:

- training: 0.91304
- validation: 0.91031
- test: 0.90483

Compared to Model 1, we saw slight improvement on test set by 0.00207

4.4 Model 3: VGG variation 2

Upon the realization that adding one more convolution layer before max pooling does not significantly improve, we decided to go deeper instead. While keeping the conv-conv-pool scheme the same, we simply added one more of it:

Conv1 – Conv2 – Pool1 – Conv3 – Conv4 – Pool2 – Conv5 – Conv6 – Pool3 – FC – Dropout – FC – Softmax

With the above architecture, we obtained the following:

- training: 0.925908
- validation: 0.92209
- test: 0.91501

Compared to Model 2, we saw improvement on test set by 0.01018, which is a big jump (1% improvement). At this stage, we wanted to see if the *local batch normalization* suggested by Krizhevsky et al.[4] would further fine tune/regularize our model. But with local response normalization, the test accuracy was actually *lower* than before (0.90238).

4.5 Model 4: Training on Extra data

So far we have only been training on the training data, which contains 73,257 data points. This time, we decided to train the same model from model 3 with larger number of data by using "*extra.csv*", which contains 531,131 images. Since the data set is 7 fold larger than training data set, we ran for *10 epochs* first for faster computation and yielded the following results:

- training: 1
- validation: 0.98001
- test: 0.95347

We split the "*extra.csv*" into 520,000 for training and the remaining for validation. Compared to Model 3, test set accuracy improved by 0.03846 (3.846%) which is a significant jump. Upon the realization of importance of larger data size, we simply ran for more epochs, 50 and 100. The test set accuracies are the following:

- 50 epochs: 0.95952
- 100 epochs: 0.95625

With 50 epochs, we obtained close to 96% mark while 100 epochs actually yielded a lower accuracy. Yet, we think that all 10,50,100 epochs yield comparable test set results/accuracies and the variation simply comes from some random error (such as different random initialization in weight filters, etc).

4.6 Model 5: VGG variation 3

Even with further hyper-tuning, we could not yield any better results. Hence, we decided to go even further deep with the network:

$$\text{Conv1} - \text{Conv2} - \text{Pool1} - \text{Conv3} - \text{Conv4} - \text{Pool2} - \text{Conv5} - \text{Conv6} - \text{Pool3} - \text{Conv7} - \text{Conv8} - \text{FC} - \text{Dropout} - \text{FC} - \text{Softmax}$$

Similar to Model 3 with one extra conv-conv-pool layer. However, surprisingly, we obtained the following results:

- training: 0.20238
- validation: 0.19692
- test: 0.19443

It was surprising that having one more conv-conv-pool layer drastically decreased our model accuracy across all data sets. We suspect that with too many max pooling layer, the architecture down-sampled too much, hence possibly lost information in the process. Or, the network became too deep such that it faced a "vanishing gradient problem". Nonetheless, with such a poor accuracy, this model was immediately abandoned.

4.7 Model 6: Training with cropped images

So far, the biggest booster in accuracy is introducing "extra.csv" file which contains 7 times more data points than "train.csv". In other words, if we can augment our data, we can artificially increase the number of images and may result in higher accuracy. But we had to be careful how we augmented. We did not want to flip left to right or up and down because in that case, true spatial information (i.e. asymmetry) in digits would be lost. For instance, digit "1" or "8" might be okay with flipping left to right, but all the other digits (especially digit 10) would lose their asymmetric spatial information. Same goes with rotation: if we rotate "6" 180 degrees, we will get 9 with label of 6, which is not a good idea for machine to learn.

Considering these scenarios, we decided to crop the images instead in 4 corners and the center by 28x28 pixels. We chose 28 by 28 so that we can ensure all the digit spatial information is still in the cropped image. In other words, if we did crop in 16 by 16 pixel size for instance, then we only get a quarter of the original image, thus, the cropped image most likely no longer contains the needed original information.

In order to test our hypothesis, we first cropped the "train.csv" in 5 corners as well as "test.csv". Now, we could no longer use Model 4 because our image size is in 28 by 28, hence we could only perform 2x2 max pooling twice (which down-samples our image to 7 by 7). In addition, when we predicted, we used *mode prediction* by predicting a class using 5 crops per image, then taking a mode of the predictions. If there were ties, then we simply flipped a coin and assigned whatever the class came out to be. Our new pipeline is the following:

$$\text{Conv1} - \text{Conv2} - \text{Pool1} - \text{Conv3} - \text{Conv4} - \text{Pool2} - \text{FC} - \text{Dropout} - \text{FC} - \text{Softmax} - \text{Mode Prediction}$$

And the accuracies are:

- training: 0.98475
- validation: 0.97736
- test: 0.95054

Compared to Model 3 (best model so far with only original "train" data set), the 5 crop train data set boosted accuracy by **0.03553 or 3.553%**. Once we confirmed our hypothesis of boosting accuracy with augmenting the data, we cropped "*extra.csv*" and evaluated:

- training: 1
- validation: 0.98447
- test: 0.96408

Comparing with Model 4, the best model so far with accuracy of 0.95952, we increased by **0.00456 or 0.456%**. The effect of cropping "*extra.csv*" was not as drastic as "*training.csv*" case, yet it still boosted our model in a meaningful way.

4.8 Model 7: Ensemble

Now that we have 5 different crops, we attempted to utilize an ensemble method. Instead of a single architecture to see all 4 corners and the center, we made 5 individual model per crop, predicted based on each model, and took the majority vote.

The classification pipeline looks like this:

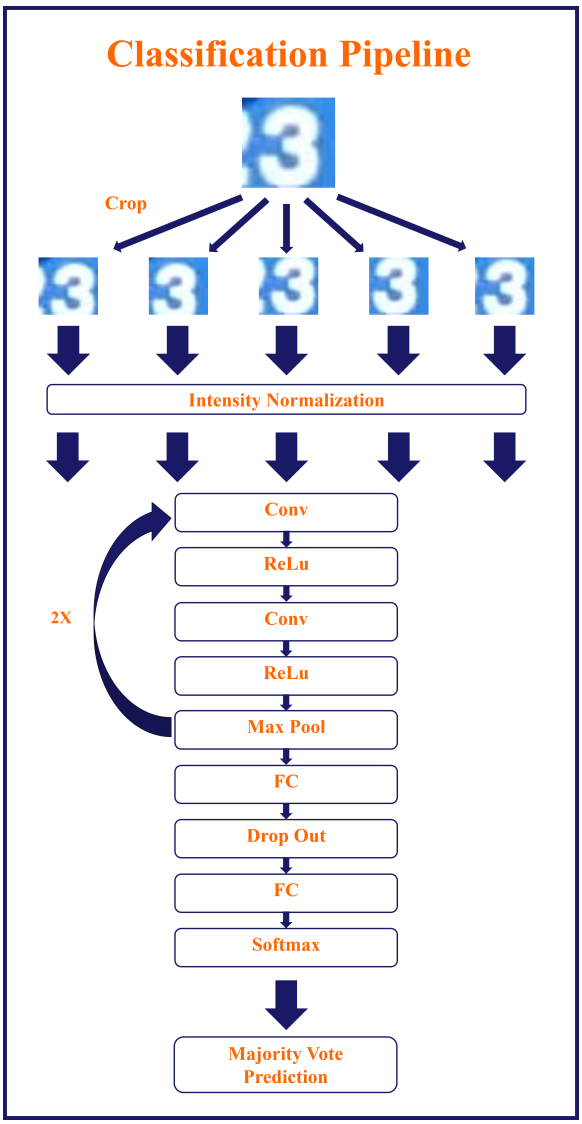

Figure 2: Ensemble digit classification using 5 crops

- training: 1-ish for all 5 models
- validation: 0.981-ish for all 5 models
- test: **0.96427** as a majority vote among 5 models

The reported values for training and validation are approximate values for all 5 models. For test, we took the majority vote among 5 predictions from 5 models.

Compared to Model 6, the increase was quite minuscule, 0.00019, which indicates that there is no difference between a single network seeing all four corners and center and 5 individual networks seeing 5 different crops. It is also noteworthy that all individual networks could achieve 100% training accuracy and 98.1% validation accuracy.

Among the 7 models we have developed so far, ***this one is the most advanced/accurate model for SVHN classification task***, hence we decided to further fine-tune hyper-parameters: filter size, number of filters, number of fully connected nodes, learning rate, batch size, drop out rate. Below is the summary of hyper-parameter tuning with train and validation set accuracy:

Convolution filter size	Number of Convolution filters	Number of Fully Connected Nodes	Drop out probability	Learning Rate	Batch size	Train accuracy	validation accuracy
3	32,32,64,64	1024	0.5	1.00E-04	64	1	0.912865
5	32,32,64,64	1024	0.5	1.00E-04	64	0.984375	0.913724
7	32,32,64,64	1024	0.5	1.00E-04	64	1	0.882714
5	16,16,32,32	1024	0.5	1.00E-04	64	1	0.922934
5	32,32,64,64	1024	0.5	1.00E-04	64	1	0.917102
5	64,64,128,128	1024	0.5	1.00E-04	64	0.984375	0.913724
5	32,32,64,64	524	0.5	1.00E-04	64	1	0.915566
5	32,32,64,64	1024	0.5	1.00E-04	64	0.984375	0.913724
5	32,32,64,64	2048	0.5	1.00E-04	64	1	0.90697
5	32,32,64,64	1024	0.25	1.00E-04	64	1	0.905124
5	32,32,64,64	1024	0.5	1.00E-04	64	0.984375	0.913724
5	32,32,64,64	1024	0.75	1.00E-04	64	0.953125	0.89868
5	32,32,64,64	1024	0.5	1.00E-03	64	1	0.914952
5	32,32,64,64	1024	0.5	1.00E-04	64	0.984375	0.913724
5	32,32,64,64	1024	0.5	1.00E-05	64	0.8125	0.853546
5	32,32,64,64	1024	0.5	1.00E-04	32	1	0.913417
5	32,32,64,64	1024	0.5	1.00E-04	64	0.984375	0.913724
5	32,32,64,64	1024	0.5	1.00E-04	128	0.992188	0.907891

From this table, we could deduce that the following hyper-parameters yield the best validation set result:

- Convolution filter size: 3
- Number of Convolution filters: 32,32,64,64
- Number of Fully connected Nodes: 1024
- Drop Out probability: 0.5
- Learning rate: 1e-4
- Batch Size: 64

The only difference between our initial ensemble model and best hyper-parameter was the Convolution filter size (5 vs 3). However, ironically, our best validation set settings did not perform better than what we did with the initial findings with ensemble approach: the accuracy with best validation set hyper-parameters was 0.96067. Hence, we went back to our initial settings, filter size of 5.

Below graphs are the accuracy and loss over time of our best model:

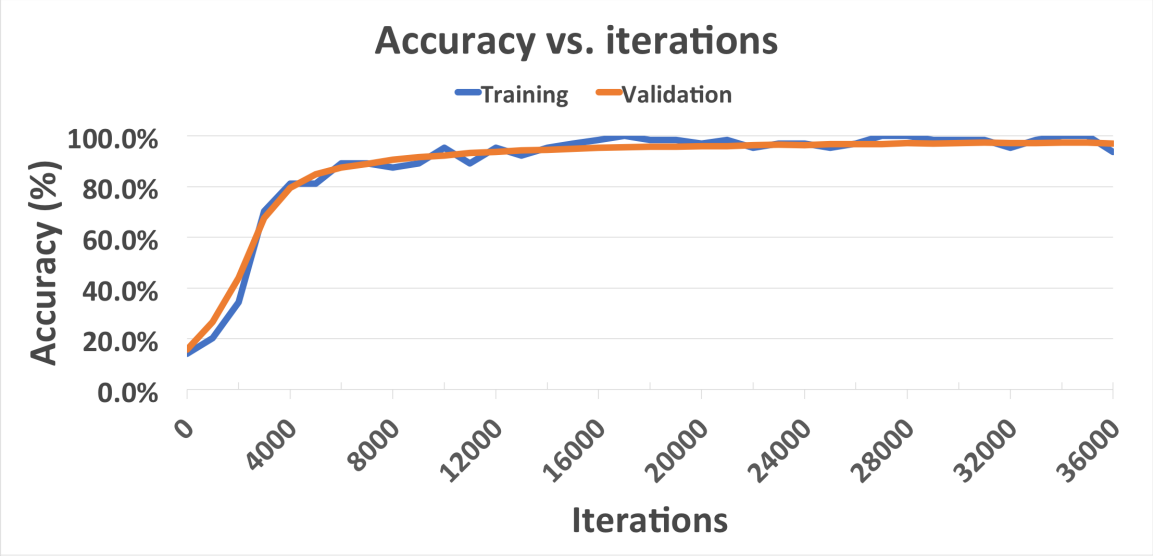

Figure 3: Accuracy over iterations

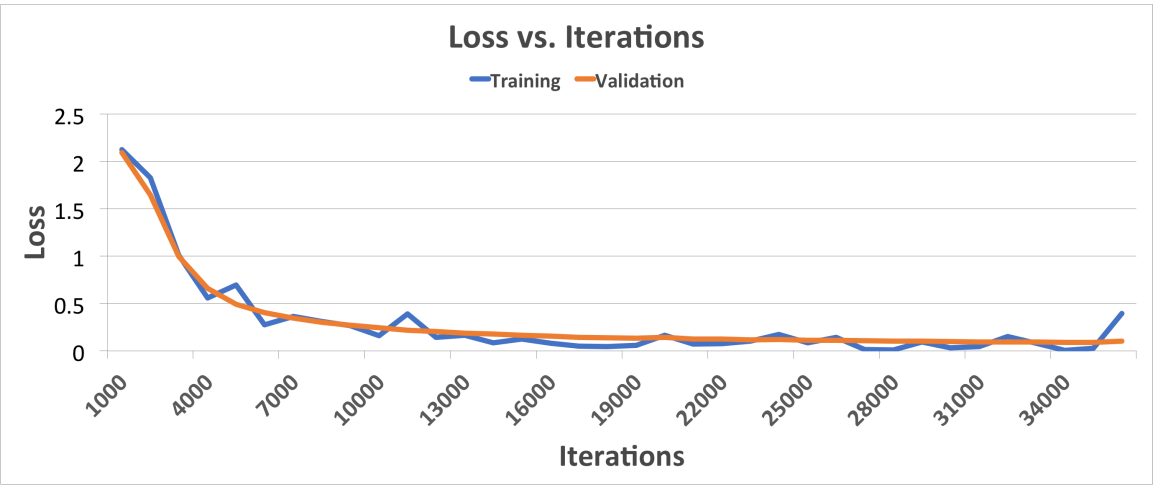

Figure 4: Loss over iterations

One can see that training accuracy and loss are quite jiggly due to the batch: a different batch is fed at every 1000th step, hence the model yields a different loss and accuracy every time it evaluates. Below, we reported the normalized confusion matrix of the model of the validation set:

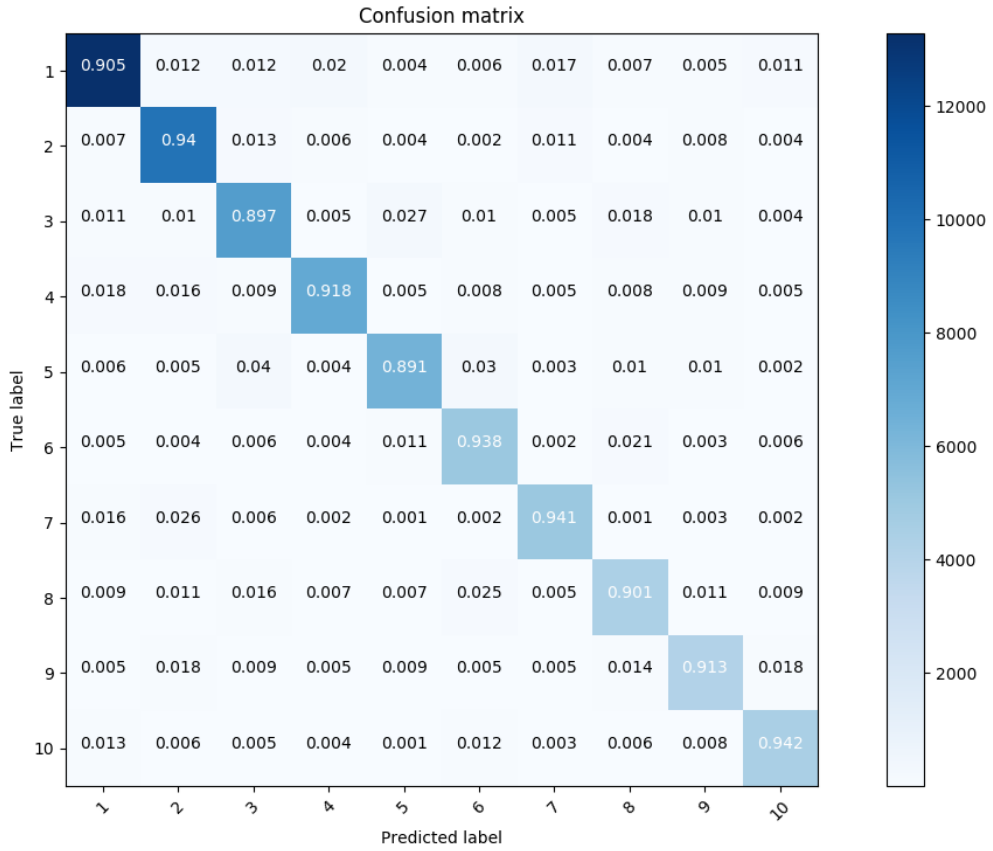

Figure 5: Normalized Confusion Matrix of Validation Set

It is interesting to see that digit 5 was particularly hard for the model to classify (0.891) as digit 5 was most confused with digit 3 (0.027). Considering that both numbers have a half round shape on the bottom half, we wonder whether the model paid too much attention to it and got confused. Digit 10 was the easiest one to classify with robust 0.942 accuracy. This is kind of expected as digit 10 contains both number "1" and "0", hence probably least confusing for the model to predict.

5 Summary Table of 7 models and Kaggle submission accuracy graph

Here, we report a summary table for 7 models. We also report the Test set accuracy vs. Kaggle submission.

Model number	Training set Accuracy	Validation Set Accuracy	Test Set Accuracy
Model 1	0.89062	0.88437	0.89437
Model 2	0.91304	0.91031	0.90483
Model 3	0.925908	0.92209	0.91501
Model 4	1	0.98001	0.95952
Model 5	0.20238	0.19692	0.19443
Model 6	1	0.98447	0.96408
Model 7	1	0.981	**0.96427**

Figure 6: Summary of 7 models with corresponding train/validation/test accuracy

- Model 1: Conv1 - Pool1 - Conv2 - Pool2 - FC - Dropout - Softmax
- Model 2: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - FC - Dropout - Softmax
- Model 3: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - Conv5 - Conv6 - Pool3 - FC - Dropout - FC - Softmax
- Model 4: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - Conv5 - Conv6 - Pool3 - FC - Dropout - FC - Softmax with EXTRA data set
- Model 5: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - Conv5 - Conv6 - Pool3 - Conv7 - Conv8 - FC - Dropout - FC - Softmax
- Model 6: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - FC - Dropout - FC - Softmax - Mode Prediction with EXTRA data set cropped (SINGLE ARCHITECUTRE)
- Model 7: Conv1 - Conv2 - Pool1 - Conv3 - Conv4 - Pool2 - FC - Dropout - FC - Softmax - Mode Prediction with EXTRA data set cropped (ENSEMBLE ARCHITECUTRE)

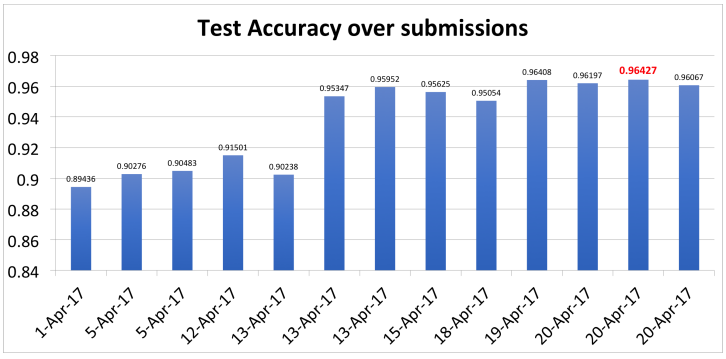

Figure 7: Accuracy over submissions

NOTE: I excluded the 8 layer convolution one as that one produced way too low of an accuracy, hence skew the entire graph down. For accurate representation, I dropped that particular case.

6 Discussions

As of *Thursday 20th, 2017*, our model is **ranked 17th with accuracy of 0.96427**. Considering that the human performance on SVHN classification is at 98% [5], we are slihgtly worse than what humans can perform.

6.1 Feature selection and construction

In retrospect, we could have done SVM or logistic regression to solve SVHN classification problem, but knowing the best method (CNN) in advance, thanks to Elec 677, we simply did not find any reason to pursue this. In addition, convolutional neural network is just a multiple stack of logit units that can perform SVM, logistic regression, etc. We simply happened to stack a lot of logit units with convolutional and fully connected layers, and used softmax regression at the end to classify. And via convolutional layers, we produced many feature mappings and hoped that the architecture would learn some good representations of the images.

6.2 Experiments and our model selection evolution

The base line architecture we started was simply adopting an architecture for MNIST dataset. Since MNIST and SVHN are very similar in their nature (digit classification), we expected a good performance from it, which was true (0.89436 test accuracy). But even after doing all hyper-parameter optimization for MNIST model, we could not break 91% accuracy.

Hence, we adopted some ideas of VGG network and learned that adding more convolutional layers before max pooling can yield a better performance (91.5%). However, if you max pool way too many times, then it is possible that the network can significantly under-perform and yield a poor result. We think that information in feature mappings can be potentially lost due to too much down-sampling. Yet, more in-depth analysis is surely needed to confirm this. We also tried a local response normalization method to see if this can regularize/improve the network, but did not see any meaningful improvement.

The biggest improvement was increasing the size of the training set we into network. By utilizing extra data images, our network accuracy jumped from 91.5% to 95.95%. But one must be careful about adding more data into the network because "quality" of the data set is very important. We have to remind ourselves the lesson of "Garbage in, garbage out". Fortunately, the SVHN data set was of high quality and our network benefited from this.

Knowing that more data points can yield a significantly better performance, we augmented our data by cropping 4 corners and the center of the image. By generating 5 times more images, we expected a better performance. This was true with training data set: By augmenting training data set, we saw the test set accuracy improved from 91.1% to 95.1%. However, cropping extra training set did not yield that big of an improvement (from 95.95% to 96.4%). This tells us that once there are enough data points, the architecture simply does not learn any better from augmentation.

The last method we did was ensembling our model. By building a single model per crop, we predicted from 5 different architectures and took a majority vote. The improvement was very small (0.019% increase in test accuracy), which tells us that seeing 5 crops via a single architecture is not that different from seeing 5 crops via 5 different models.

6.3 Model assessment (preventing overfit)

It was essential that we split our data into two: train and validation. By doing so, we could always get some feedback on the model accuracy on unseen data (validation set) and fine-tune our hyper-parameters.

While we trained our model, over-fitting was definitely a concern. In order to prevent it, we implemented drop-out on our fully connected layer so that not all the nodes are activated while training. We tried 3

different strength of drop-out and found that drop out rate of 0.5 gave us the best result on validation set. With lower drop out rate (0.25), we were able to converge fast (as all the nodes were connected to learn), but the accuracy on validation set was low (compared to training), indicating that there is a over-fit problem. With higher drop out rate (0.75), the model converged much slower as the layers were not connected more often. In addition, because we penalized our model too much, the accuracy on both training and validation set yielded poorly, which indicates that the middle value (0.5) is the best number to regularize our model. Yet, compared to test set accuracy, our validation set accuracy was always a little higher than our best test set result (98% vs 96.4%), which is a sign of possible over-fit. But the difference is only minuscule 1.6%, and we might be able to get away with it.

6.4 Final model

Our final model is the **Ensemble CNN with 5 crops on extra.csv (Model 7)**. The selection is based on the test set accuracy (highest one). But a single network seeing all 5 crops (Model 6) would do equally good as the difference is only 0.019% in test set accuracy.

6.5 Lessons learned from machine learning and Big Data

The biggest roadblock we felt doing this project was dealing with memory issue. We had a GeForce 1070 with 8 GB RAM to deal with long computation time, but we sometimes ran into out of memory problems. Fortunately, with 8 GB RAM, we could load both the data and the network altogether on the GPU to train the architecture. Yet, memory management was always the biggest enemy to overcome. One of the solutions we found is to split the data set, train the model on different sessions, and then output the result.

By looking at how much our accuracy increased when we utilized extra data, we also came to realize the importance of the amount of data. Thus, we decided to crop one image into five regions (4 corners and the middle in 28 by 28) to get more data as well as go deeper into the model training.

We also faced the problem of training efficiency. For example, when we trained the model with filter size of 64, the accuracy slightly increase, but the training time is way longer than before and it gives out of memory error. Thus, although having more filter size might be better, we decided to stick with a reasonable number, which is 32, so that we can train our model efficiently and still have a good accuracy.

The process of tuning parameters also taught us how different parameters affect the accuracy of the model. The rules of thumb for hyper-parameter that we learned are the followings:

- You can increase number of layers or the number of feature mappings, but considering the efficiency, the gain is not that substantial after certain level. You will saturate eventually.
- There is always a too much vs too little problem. You must find a sweet spot (for instance, in our case, finding drop out rate of 0.5).

It is also interesting how one can pre-process the data in different ways. Here, we pre-processed by dividing by 128 and subtracting by 1 to put the intensity values into [-1,1]. We could have done transforming our images into gray scale or applying different normalization methods (like subtracting mean intensity). Or we could have tilted our images in some angles and see if that brings any accuracy improvement. Also, we could have explored different learning algorithm, such as Adagrad, RMS prop, or Adadelta, and different activation functions, like tanh, leaky Relu, sigmoid, etc. All those were not explored in this project, but will be left to as future work.

7 Technical Reference

The following links are websites we visited to help us process data and build the model with tensorflow.

- pandas.read_csv: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html
- Digit recognition from Google Street View images: https://experimentationground.wordpress.com/tag/tensorflow/
- An Introduction to Implementing Neural Networks using TensorFlow: https://www.analyticsvidhya.com/blog/2016/10/an-introduction-to-implementing-neural-networks-using-tensorflow/
- Digit Classifier Using Convolution Neural Network: http://courseprojects.souravsengupta.com/cds2016/digit-classifier-using-convolution-neural-network/
- Data Pre-processing: http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing#Simple_Rescaling

8 Structure Reference

Followings are blogs and papers we read through to gain insights on structures of models other people implemented for SVHN data.

- Tensorflow tutorial on deep cnn:https://www.tensorflow.org/tutorials/deep_cnn
- Deep MNIST for Experts: https://www.tensorflow.org/get_started/mnist/beginners
- Convolutional Neural Networks Applied to House Numbers Digit Classification: http://yann.lecun.com/exdb/publis/pdf/sermanet-icpr-12.pdf
- Very Deep Convolutional Networks for Large-Scale Visual Recognition: http://www.robots.ox.ac.uk/~vgg/research/very_deep/

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: http://tensorflow.org/.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[5] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification," in *Pattern Recognition (ICPR), 2012 21st International Conference on*, IEEE, 2012, pp. 3288–3291.