Deep Learning
SVHN Number Recognition By CNN

Zongchang Chen

October 2016

1 Introduction

This project focuses on classifying the main digit on cropped pictures from the Street View House Numbers (SVHN) Dataset [1] with convolutional neural networks. The main architecture will be explicitly illustrated in details at the rest parts of this report. As mentioned above, in this project, only cropped digits with size of 32×32 are used, excluding the extra dataset.

2 Overview

Since this is apparently a classification problem, some supervised learning algorithms are naturally brought up to the desk. Initially, logistic regression and support vector machine are utilized to do the first several trials, but the results are not quite ideal. Hence, convolutional neural networks are deployed to train the dataset with their complex non-linear properties. But the procedure is also rusty. It takes a great amount of huge time cost trials to eventually determine the decent range of hyper-parameters and appropriate optimizer. Finally, another more complicated CNN architecture is used for a deeper exploration of the dataset. The precision is slightly higher with the deeper network, but the improvement is not significant.

3 Data

3.1 Exploration

In this project, we use the cropped single digit images from the Street View House Numbers (SVHN) as our dataset. SVHN includes 73257 digits for training and 26032 digits for testing, and each image has a size of 32×32 pixels. Additional dataset will not be used in this project.

Next, let's take a peek at the digit distribution of this whole dataset.

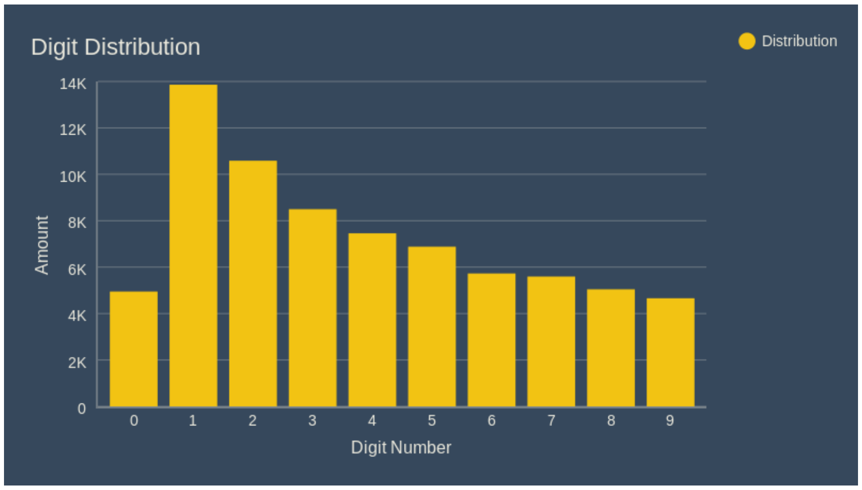

Figure 1: Train Digit Distribution

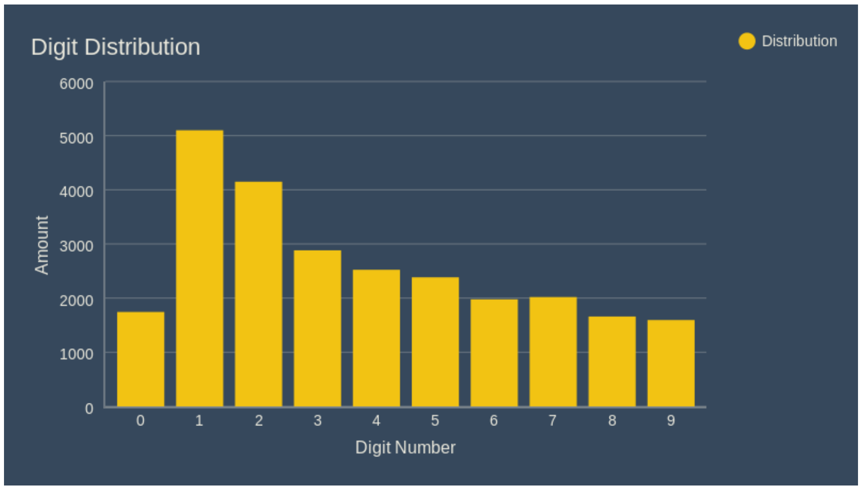

Figure 2: Test Digit Distribution

The distributions of the 10 digits in train and test dataset are roughly the same, so the initial effectiveness of this project can be guaranteed.

3.2 Pre-Processing

The only pre-process made on the dataset is normalization [2].

$$X = X/128 - 1.0 \tag{1}$$

By applying normalization, we make the value of each pixel within range $(-1, 1)$ as type $float32$. This will help gradient descent work better since numbers are generally smaller.

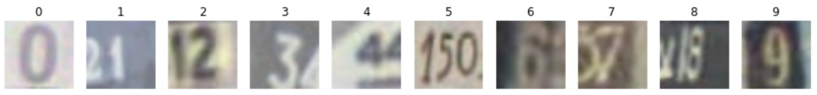

Figure 3: Original images from 0 to 9

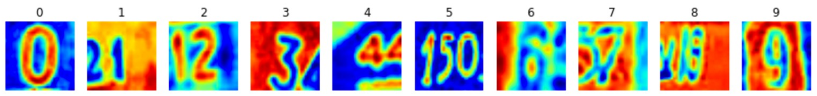

Figure 4: Normalized images from 0 to 9

4 Training

4.1 Logistic Regression

At the very beginning, let's see how logistic regression can model the dataset.

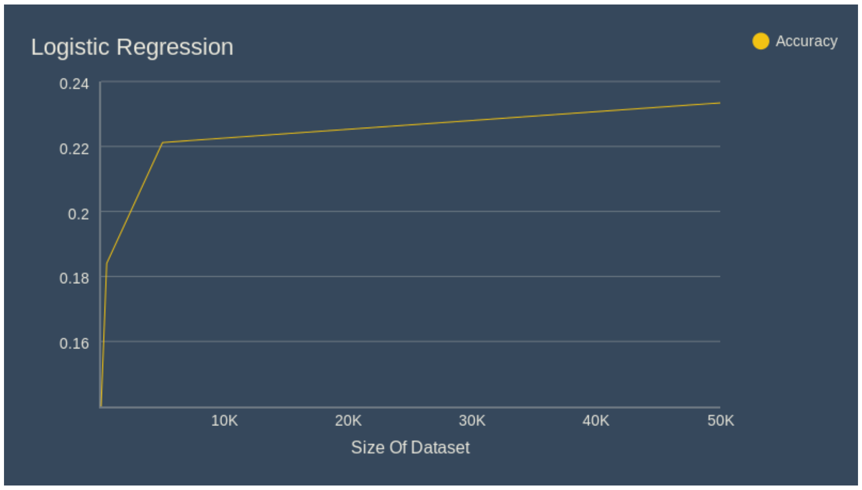

Figure 5: Logistic regression model

From the figure above, we can clearly see that the test accuracy grows slightly higher each time when the size of dataset is timed by 10. But even with the stable growth, the accuracy is still extremely low when the dataset includes 50000 data. Unfortunately, after several tries on parameter tuning, the result is still not drastically changed.

On the other hand, it actually makes sense that the map of this classification problem is not just simply linear. So in the next experiment, we will use a more complex model to train the dataset.

4.2 Support Vector Machine

Here, we apply SVM model to our dataset. With using 'rbf' kernel, SVM can map the current data into a higher dimensional space, and then manage to split the data by a linear hyperplane.

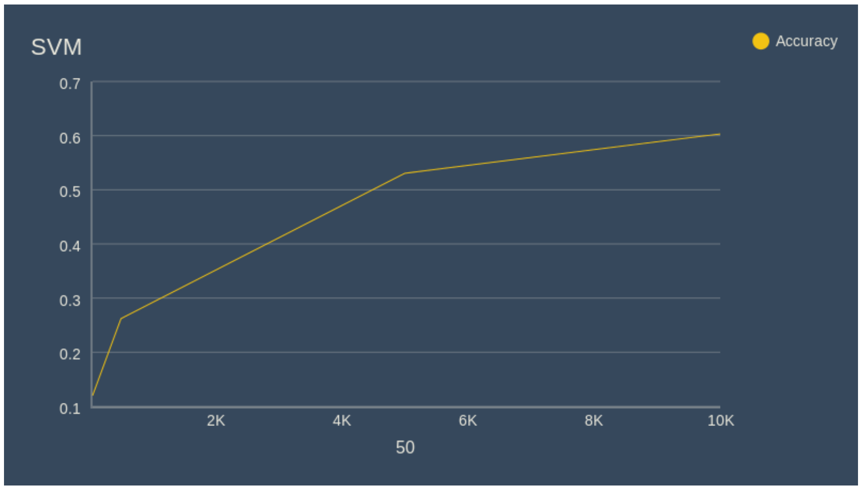

Figure 6: SVM model

SVM gives us a much better result. With 50000 data for training, we get a model whose accuracy on the test set is over 73.0%. Moreover, we can give a very confident guess that if more dataset are used for this training, the final accuracy could probably reach 80.0% or even higher.

SVM has shown a great improvement comparing to simple logistic regression model, but the result accuracy of 73.0% is still not ideal. So even mapped into a higher dimensional space, the dataset is still not linear separable. Therefore, we have to deploy some algorithms with higher complexity and thus CNN is introduced in the following section.

4.3 Convolutional Neural Network

The next approach we are going to implement is CNN. CNN is so widely used in computer vision domain in recent years. Thus, due to the success of empirical experiments, it's definitely worth a try to this problem.

4.3.1 Architecture

- Default

Input
32×32
$\Downarrow$
Convolution
5×5 patch & same padding
$\Downarrow$
ReLU
$\Downarrow$
Max Pooling
2×2 stride & same padding
$\Downarrow$
Convolution
5×5 patch & same padding
$\Downarrow$
ReLU
$\Downarrow$
Max Pooling
2×2 stride & same padding
$\Downarrow$
Dropout
keep rate 0.9
$\Downarrow$
Fully Connected
$\Downarrow$
Fully Connected
$\Downarrow$
Softmax
$\Downarrow$
Output

The reason why this net is named by `Default` is that it performs very well on the dataset MNIST which is also a typical number recognition dataset for CNN research. More hyper-parameter details will be discussed in the next section.

- Deeper

Input
32×32
$\Downarrow$
Convolution
5×5 patch & valid padding
$\Downarrow$
ReLU
$\Downarrow$
LRN
$\Downarrow$
Max Pooling
2×2 stride & same padding
$\Downarrow$
Convolution
5×5 patch & valid padding

⇓

ReLU

⇓

LRN

⇓

Convolution

5×5 patch & valid padding

⇓

ReLU

⇓

Max Pooling

2×2 stride & same padding

⇓

Dropout

keep rate 0.9

⇓

Fully Connected

⇓

Fully Connected

⇓

Softmax

⇓

Output

`LRN` here is abbreviation of `Local Response Normalization`. It's first introduced by Krizhevsky in his original ImageNet classification paper[3]. Also here is a quote from `Caffe` main page, "The local response normalization layer performs a kind of 'lateral inhibition' by normalizing over local input regions." [4]

4.3.2 Metrics

The main tool to measure our model is accuracy, but in the meanwhile, we also make some observations on the loss of our models to see the speed of convergence, where loss is defined as the cross entropy of softmax function,

$$L_i = -\lg \frac{e^{f_{yi}}}{\Sigma_j e^{f_j}}. \tag{2}$$

Moreover, in this project, l_2 regularization is added to the loss computation, so

$$L_i' = L_i + \lambda \sum_k (w_{ik}^2 + b_{ik}^2). \tag{3}$$

Our aim is to minimize the loss so that we could obtain a fully approximated model to the real and unknown map to this problem.

4.3.3 Hyper-Parameter Tuning

In this section, we will mainly discuss the hyper-parameter setting to the first default ConvNet. We will apply the similar strategy to our second deeper net.

- **Benchmark**

$$\begin{aligned}
batch\_size &= 128 \\
conv1\_depth &= 16 \\
conv2\_depth &= 32 \\
hidden\_layer1\_size &= 64 \\
hidden\_layer2\_size &= 16 \\
patch\_size &= 5 \\
start\_learning\_rate &= 0.001 \\
regularization\_lambda &= 0.001 \\
dropout\_keep\_rate &= 0.9 \\
learning\_rate\_decay\_rate &= 0.95 \\
optimizer &= SGD
\end{aligned}$$

This set of hyper-parameters looks quite normal at the first glance. However, the result of prediction is surprisingly bad.

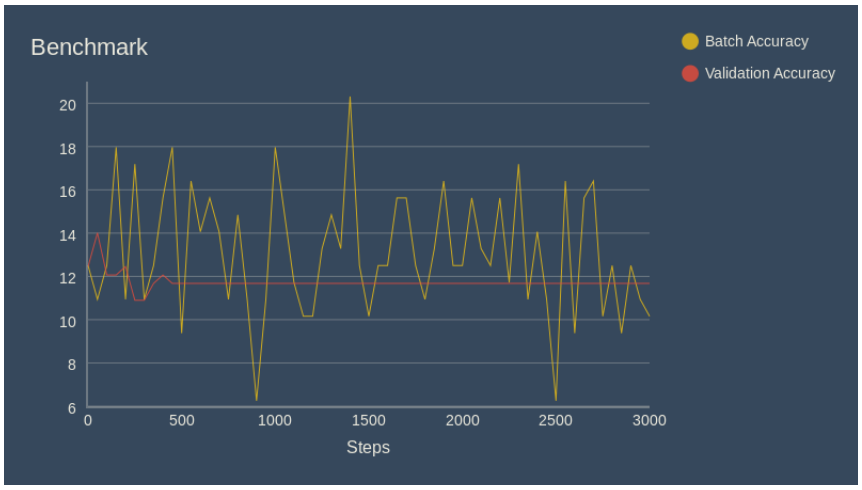

Figure 7: Accuracy of default ConvNet with benchmark hyper-parameters

The batch accuracy oscillates within 6% and 25% heavily in the first 3000 steps. The validation accuracy is even worse since it doesn't change after 1000*th* step. The loss is stuck at around 5.0, which is also very huge.

After several trials on tuning *batch_size*, *layer_size*, *layer_depth*, *learning_rate*, and *dropout_keep_rate*, the prediction accuracy is still unacceptable.

Therefore, the last thing that we could adjust is the optimizer. After some researches [5], it becomes clear that simple stochastic gradient descent has lots of shortcomings (we will discuss that in the next analysis section). Thus, some new optimizers are used in experiments, including "Momentum", "Adagrad" and "Adam".

- **Adam Optimizer**
  Through some more experiments on new optimizers, it turns out that "Adam" is the best candidate to this problem. Thus, with almost the same hyper-parameters setting with benchmark, now we instead use Adam optimizer in the new training.

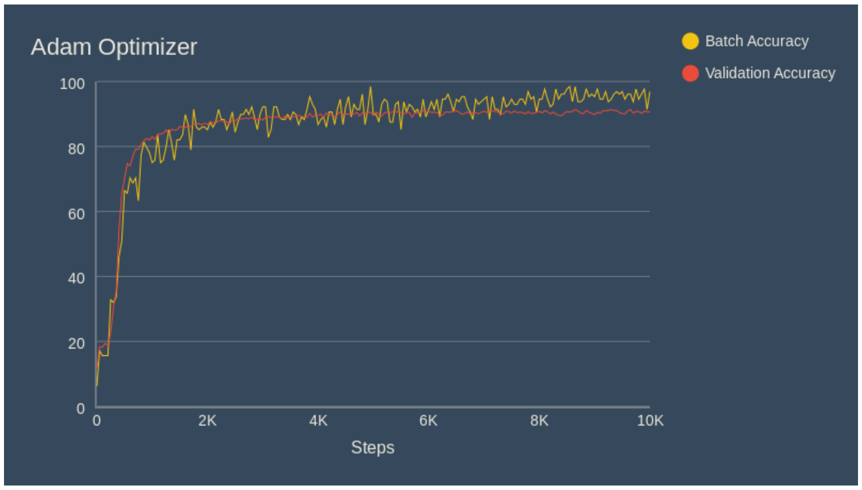

Figure 8: Accuracy of default ConvNet with Adam optimizer

  Now, we indeed have a valid model. The validation accuracy converges at 90.60% at the end of 10000-step training. The final test accuracy of this model is 89.59% and the final loss is between 0.20 and 0.40(not included in the plot), which is a huge hop from previous benchmark experiment.

- **Dropout Keep Rate & Learning Rate**
  After a number of more experiments and trials, it turns out that the *size* and *depth* of the network do not have significant impacts to our final test accuracy. Thus, we will mainly focus on the affect that different *dropout_keep_rate* and *learning_rate* bring to our model.

  Dropout layer is firstly introduced in this paper [6]. It could speed up our training process because it randomly drops out some weights, turns them into 0, so that our network becomes simpler. Also, it helps avoid overfitting since for each batch training, we actually have a distinct stochastic model. So, to some extent, our whole training process is an ensemble training which is a typical way to prevent the model from being overfitting.

First, let's take a look at the case with no dropout applied.

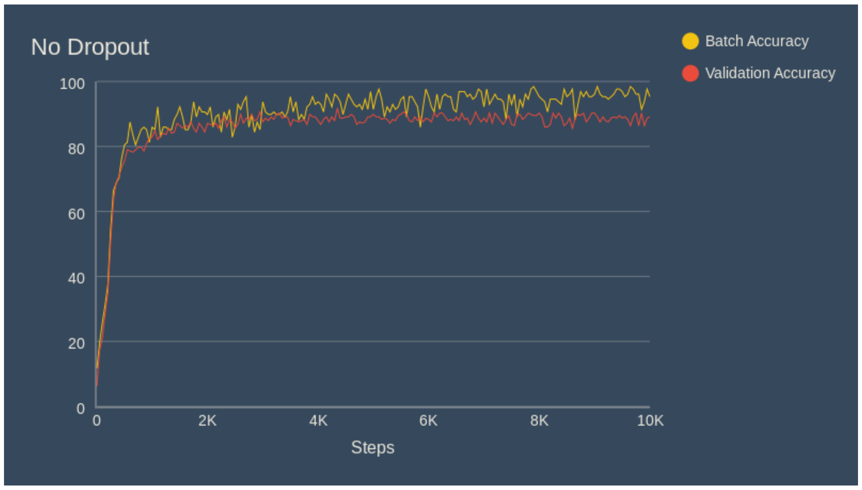

Figure 9: Accuracy of default ConvNet with no dropout

Without dropout layer, the accuracy stops growing at 87.35%, almost 3% below that of our previous model. Hence, appropriate dropout rate can indeed enhance the ability of generalization of the model.

Next, we switch the keep rate to 0.5, to see what happens to our model if half of information are dropped.

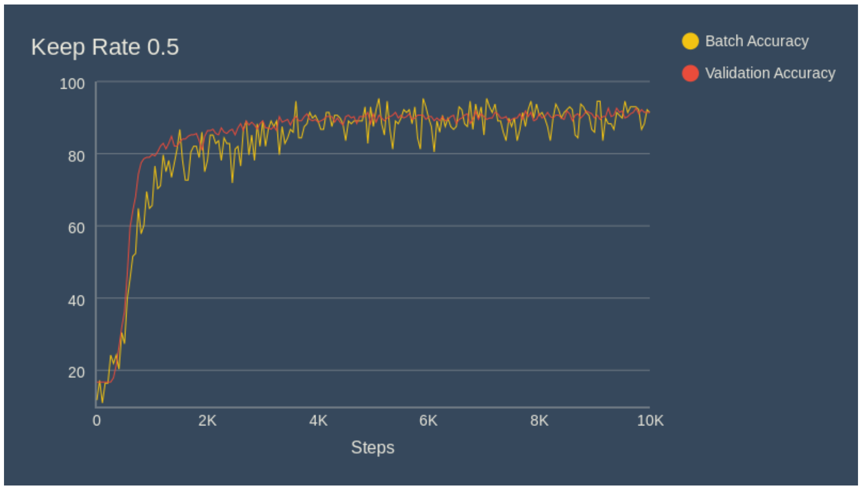

Figure 10: Accuracy of default ConvNet with dropout keep rate 0.5

With keep rate of 0.5, we actually drop out half of features in the first fully connected layer, which is probably slightly cross the threshold. Thus, the model accuracy falls down to around 88.0%, and this leads to a slow convergence as well. This model is not optimal but it also performs not devastatingly bad, since it greatly eliminates the probabilities that certain feature relies on others feature structures. This operation is somehow similar to Naive Bayes and regularization.

0.5 might be a appropriate watershed of dropout keep rate in this circumstance. While using 0.3, the model accuracy is stuck at only 20.0%, which is highly likely caused by underfitting because too many information and work are lost in the dropout layer.

Then, let's switch our attention to *learning_rate* tuning. Since we already select Adam optimizer, its own attributes help us adjust the real time *learning_rate* during the training. Thus, we only need to tune the base learning rate in this step.

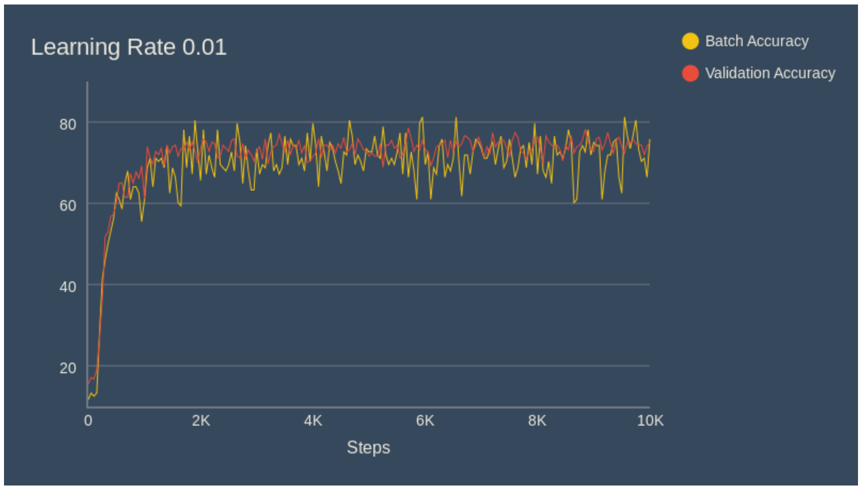

Figure 11: Accuracy of default ConvNet with learning rate 0.01

We increase the *learning_rate* from 0.001 to 0.01. With this higher base learning rate, we can clearly see that the convergence is faster than previous model, but the validation accuracy only oscillates around 70.0%. So our base learning rate here is probably too big so that although the model learns quite fast at the first several steps, it needs a lot more steps to decay its learning rate to search for the local min loss. Hence, if one sets a relatively high base learning rate, the loss may need a lot more epochs to converge to its minima.

To sum up, we now have a decent set of hyper-parameters for our first model.

$$\begin{aligned}
batch\_size &= 64 \\
conv1\_depth &= 16 \\
conv2\_depth &= 32 \\
hidden\_layer1\_size &= 64 \\
hidden\_layer2\_size &= 16 \\
patch\_size &= 5 \\
start\_learning\_rate &= 0.001 \\
regularization\_lambda &= 0.001 \\
dropout\_keep\_rate &= 0.9 \\
learning\_rate\_decay\_rate &= 0.95 \\
optimizer &= Adam
\end{aligned}$$

Let's move on to the second ConvNet with deeper architecture, and see if the parameter set can well fit to our deeper model.

- **Deeper ConvNet**

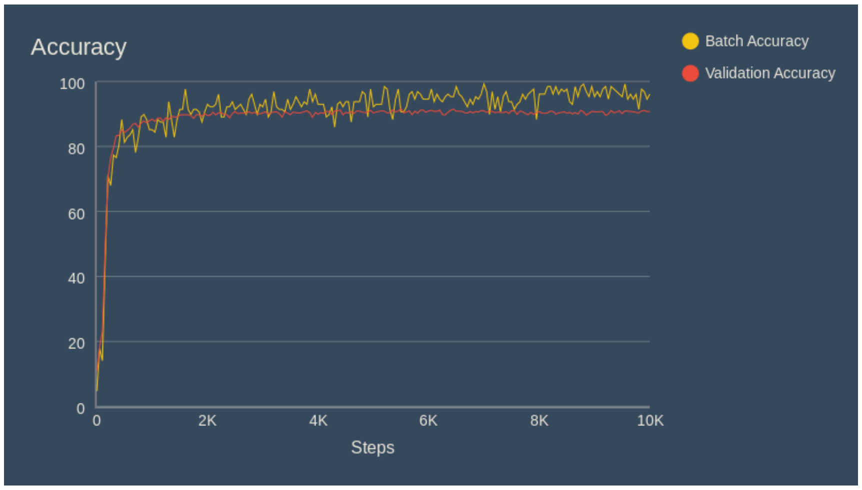

Figure 12: Accuracy of deep ConvNet

First of all, the deep ConvNet has a really fast convergence. This may credit to LRN layers because of its empirical verifications. Then, the test accuracy ends up with 90.85%, which is a just slightly improvement according to the previous model. Therefore, this scale of depth extension is not significantly helpful.

5 Analysis

The core improvement is caused by switching the optimizer to Adam, so in this section, we will concentrate on exploring the differences between Adam and SGD optimizer.

- **Stochastic Gradient Descent**
  Ideally, at each epoch of our training, we can use all training dataset for the next update to parameters. And it turns out that this fashion of training can end up with a very good convergence on local minima, and with few hyper-parameters to tune, it's more convenient and easy to implement as well. This method is named as Batch Gradient Descent. However, in practice, making use of full dataset at each iteration is almost impossible due to its extravagant time cost and memory consumption. Moreover, BGD cannot receive new data during the training process, so its incompatibility with online training mode is also an intractable issue.

Hence, SGD is introduced to solve these problems.

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^i, y^i) \tag{4}$$

, where x^i, y^i is a pair of data and corresponding label of the dataset [7].

Using SGD, we only randomly pick a set amount of data as a single mini-batch for each iteration. Since our goal is to find a direction towards the local minima according to the negative gradient, but not strictly the accurate path, SGD can only use a relatively much smaller time to discover an approximate direction to our target. On the other hand, since the data we pick each time is absolutely stochastic, adding new data into the base dataset will not affect the randomness of this algorithm, thus SGD solves the online issue as well.

SGD can converge very fast if the learning rate, i.e. step length, is big, but it may also fluctuate heavily around the local minima because "if the objective has the form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum." [7] On the other hand, if we set the learning rate a very small value, SGD can indeed converge to the local or global minimum, but the convergence is too slow to endure. Thus, this leads to two optimization subjects: **Ravines navigation** and **Learning rate adaptation**.

- **Momentum**
  Momentum is invented to navigate ravines and let the parameters quit oscillating around the irrelevant dimensions. It accelerates and rectifies the SGD along the shallow ravine.

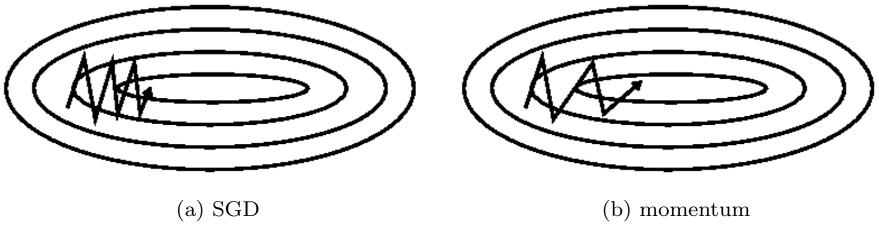
(a) SGD (b) momentum

Figure 13: SGD without/with momentum. Source [5]

Momentum implements this functionality by adding a coefficient γ as weight to the previous vector to update the current step vector. Thus, SGD with momentum is defined as

$$v = \gamma v + \eta \nabla_\theta J(\theta; x^i, y^i) \tag{5}$$
$$\theta = \theta - v \tag{6}$$

- **Adaptive Moment Estimation**
  Last but not the least, Adaptive Moment Estimation (Adam) algorithm here is to give us a better solution to learning rate adaptation. Adam optimizer keeps an exponential decay average of past squared gradient v and moving average of the parameter m, which in some senses, the momentum.

$$m = \beta_1 m + (1 - \beta_1) g \tag{7}$$
$$v = \beta_2 v + (1 - \beta_2) g^2 \tag{8}$$
$$\theta = \theta - \frac{\eta}{\sqrt{v} + \epsilon} m \tag{9}$$

  , where β_1 and β_2 are decay rates for m and v respectively[5].

  With this update, Adam adapts the learning rate according to the parameters in previous moments, it performs a larger update for infrequent parameters and if the parameter is updated with a very large scale at last moment, the change will be trivial.

  In this much smarter and automated fashion, Adam works very well based on many practical attestments, and it indeed performs as the best optimizer in our model.

6 Result

6.1 Accuracy

With deeper ConvNet and optimal hyper-parameters, we have a model with best accuary of 90.80%, it's much higher than the benchmark model. Since the test result is generally lower than the training dataset accuracy, overfitting does not take place in our training process.

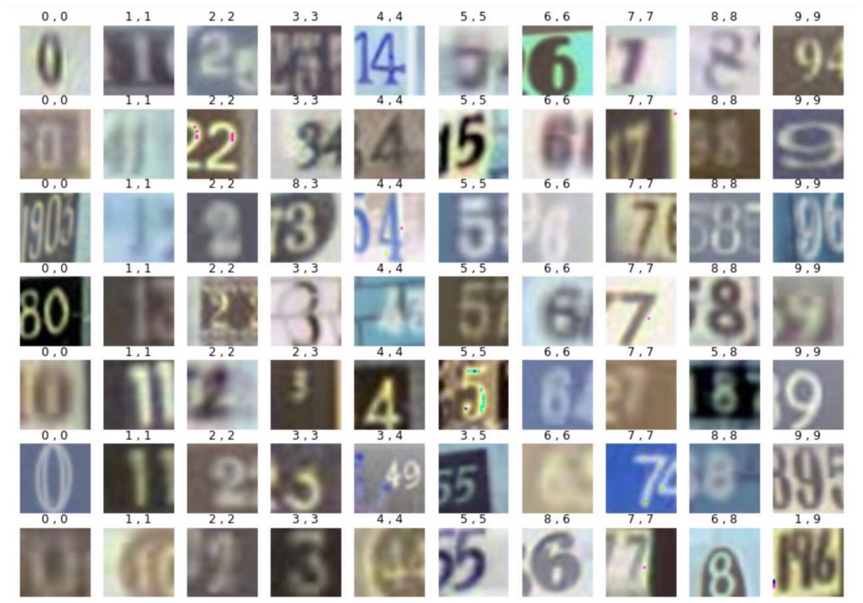

Figure 14: Visualized Results. Format of the label at top of each image: "prediction, true label"

We can see that our model has good predictions on vague inputs, but there are still a lot more improvements needed.

6.2 Robustness

To prove the robustness of this model, several additional experiments are carried out. But this time, the size of the input dataset is truncated to only 30000, more than half of the original data are randomly removed from the input. The result is as following:

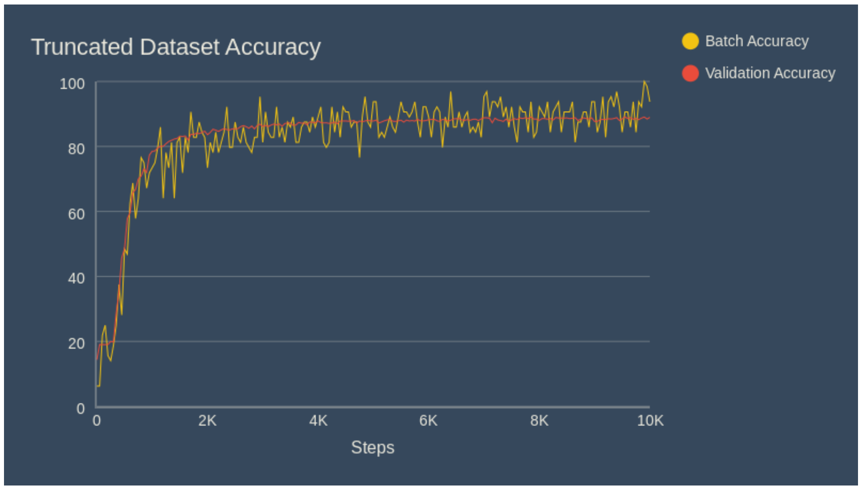

Figure 15: Accuracy of truncated input dataset

With only half of data, the prediction accuracy of this model is still able to reach 87.59%. Though it is about 2% lower than that of the previous model trained with the full dataset, it can prove that small perturbations do not considerably affect the result. In addition, this occurrence also implies the potential growth of accuracy of the ConvNet if we could include more huge amount of data. So if the `extra.mat` is used in the training phase, the model is very likely to have a much better fit and accuracy. This is also an improvement that we will pursue.

7 Reflection & Improvement

This project is a good practice to understand the principle of convolutional neural networks, especially the core algorithm gradient descent and its optimization.

The interesting part of this project is actually using Tensorflow to deploy the ConvNet structure. The well designed API really saves a lot of efforts of coding and the concept of graph and flow are very trending and interesting. But the most challenging and struggle part is definitely discovering the significance of proper optimizer. At the very beginning, without any prior knowledge, it's very hard to realize that the optimizer difference actually plays a considerably important role in loss convergence. Another challenge is hyper-parameter tuning. Due to some hardware restriction, the training is only able to be proceeded on CPU, thus the time cost for each training is tremendous. So tuning hyper-parameter is quite painful in this method.

The most essential gain from this project is the deeper understanding of various gradient descent optimizers, including their improvements and the mathematical deductions. Searching broad information and references is also a great experience in this project.

There are mainly four aspects that need more refinements and improvements.

- **More Image Pre-Processing**
  In this project, only normalization is implemented in image pre-processing phase. By empirical trials, image normalization is viewed as a must before sending them into the ConvNet model, but there are still a variety of pre-processing methods that we can utilize, such as whitening, thresholding and etc.

- **Locate Digits & Filter Out Outliers**
  As we can see from the dataset, there are a great number of extremely vague data in the training dataset, which are hardly to be recognized by human. Therefore, it is necessary to filter out the outlier. This can be also viewed as a type of pre-process, but it will be implemented on the whole dataset instead of image-wise.

- **Scale Up Model**
  Due to the constraint of the hardware gear, the architecture of the model and the size of the dataset are limited. If a beefy GPU could be provided (hopefully), we can include the extra dataset in the training phase, and largely increase the number of training steps, which are both effective ways to generate a more accurate model proved by practical researches.

- **Architecture Improvement**
  There are a lot more approaches we could make a use of during the training process or the architecture design. For example, Batch Normalization and DropConnect are new techniques introduced to practical experiments that have performed great improvements on CNN models. Follow some more complicated techniques to build a more accurate model is clearly my next goal.

References

[1] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng. *Reading Digits in Natural Images with Unsupervised Feature Learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning.* 2011.

[2] Stanford CS231n. `http://cs231n.github.io/neural-networks-2/#datapre`

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks.* NIPS 2012.

[4] Caffe: Local Response Normalization (LRN),
`http://caffe.berkeleyvision.org/tutorial/layers.html`

[5] Sebastian Ruder. *An overview of gradient descent optimization algorithms*,
`http://sebastianruder.com/optimizing-gradient-descent/index.html`

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research 15 1929-1958, 2014.

[7] UFLDL: Optimization Stochastic Gradient Descent,
`http://ufldl.stanford.edu/tutorial/`
`supervised/OptimizationStochasticGradientDescent/`