Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks

GR5242 Advanced Machine Learning
Section1: Lin Du ld2770
Section 2: Kehui Zhu kz2293, Yuchen Ma ym2644, Weibo Zhang wz2353
Columbia University

Abstract

In this report, we address a hard problem of recognizing multi-digit numbers at street level with a high accuracy with different models. We present an implementation of a deep Convolutional Neural Network with a very small receptive field applied to The Street View House Numbers (SVHN) Dataset which achieved over 93.68% accuracy based on 5 attemptions of different model structure. And an elaborate comparison after exploring the performances of different tuned parameters was built up.

1. Introduction

Recognition of multi-digit numbers on photographs taken at street level is one of the crucial issues in modern-day map making. However, the ability to recognize destinations automatically is a hard problem caused by wide variability in the visual appearance of text. The captured texts vary in aspects of fonts, colors, styles, orientations, and character arrangements. Also, this problem becomes more complicated due to environmental factors as well as image acquisition factors [1]. A recent study on this issue using a deep Convolutional Neural Network (Goodfellow et al.,2014) had reached a 96% accuracy which is an unprecedented level.

To achieve a good balance between efficiency and accuracy, we implemented models inspired by ConvNets[2]. A 93.68% accuracy was achieved in 10 hours with Google Cloud CPUs and GPUs. We also did a comparison between models with different convolutional layer filter size and different optimizers. Google Cloud CPUs and GPUs(16 vCPUs, 60 GB memory, and 2x NVIDIA Tesla K80) were used as the computational platform to implement all the experiments.

2 Related work

Instead of applying traditional localization, segmentation, and recognition steps of image preprocessing, a unified approach that integrates these three steps via the use of a deep convolutional neural network has been proposed by Goodfellow et al in 2014. In the paper, probabilistic model of sequences given images, which maximized log P(S | X) by using stochastic gradient descent, was employed on three different datasets (SVHN, Internal street view data, and CAPTCHA). The best architecture consisted of nine convolutional hidden layers, one locally connected hidden layer, and two densely connected hidden layers. The best model obtained a sequence transcription accuracy of 96.03%.

An investigation of the effect of the convolutional network depth on its accuracy in the large-scale image recognition demonstrated that the representation depth is beneficial for the classification accuracy(Karen Simonyan et al.,2015). A convolution neural network with 16 - 19 layers can be achieved using an architecture with very small (3x3) convolution filters.

3. Methodology

We implemented a deep neural network with eleven convolutional layers with small receptive fields on the Street View House Numbers (SVHN) Dataset.

3.1 Description of Dataset

The Street View House Numbers (SVHN) dataset is a real-world image dataset obtained from house numbers in Google Street View images. It shares similarities with MNIST where the images are of small cropped digits, but SVHN incorporates an order of magnitude more labeled data (over 600,000 digit images). Each image is of 64-pixels long and 64-pixels wide and has three RGB channels, and it was transformed into 54*54*3 tensor form . It is also a significantly hard problem compared to MNIST, because the images lack any contrast normalization, contain overlapping digits and distracting features.
The dataset consists of 73,257 digits for training and 26,032 digits for testing. In addition, 531,131 somewhat less difficult samples are also provided to be used as extra training data. In this project, we used the full number format dataset.

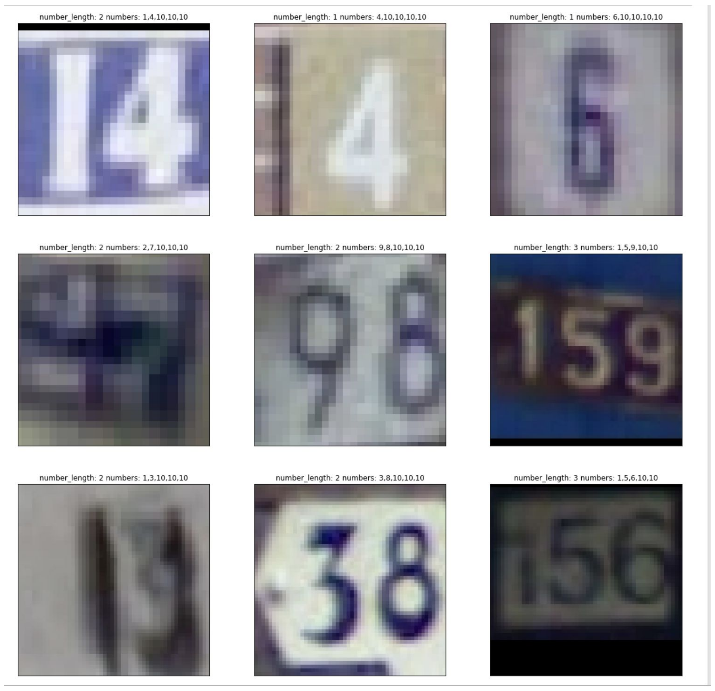

number_length: 2 numbers: 1,4,10,10,10
number_length: 1 numbers: 4,10,10,10,10
number_length: 1 numbers: 6,10,10,10,10
number_length: 2 numbers: 2,7,10,10,10
number_length: 2 numbers: 9,8,10,10,10
number_length: 3 numbers: 1,5,9,10,10
number_length: 2 numbers: 1,3,10,10,10
number_length: 2 numbers: 3,8,10,10,10
number_length: 3 numbers: 1,5,6,10,10

Figure 1: Training dataset example

3.2 Problem Formulation

Let S represents the output sequence and X represents the input image. Our goal is to learn a model of $P(S|X)$ by maximizing $logP(S|X)$ on the training set.

To model S, assume that the identities of the separate digits are independent of each other and add random variable L to represent the length of the sequence(0,...,5 and >5), so that the probability of a specific sequence $s = s_1, ..., s_n$, where s_i has 10($0, ..., 9$) possible digits, is given by

$$P(S = s|X) = P(L = n|X) \prod_{i=1}^{n} P(S_i = s_i|X).$$

At train time, we use a Convolutional Neural Network. The first layer can extract the features that split the digits and localize it at the same time. After two fully-connected layers, each softmax model (the model for L and each Si) can use the same backprop learning rule as for when training an isolated softmax layer, except that a digit classifier softmax model backprop nothing on examples for which that digit is not present.
At test time, we predict

$$s = (l, s_1, ..., s_l) = argmax_{L,S_1,...,S_L} logP(S|X).$$

The argmax function for each character can be computed independently.

3.3 Optimization methods

3.3.1 Momentum

Momentum[2] is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

3.3.2 Nesterov accelerated gradient

Nesterov[3] accelerated gradient gives us an approximation of the next position of the parameters.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

3.3.3 Adam

Adam[4] stories an exponentially decaying average of past squared gradients, also keeps an exponentially decaying average of past gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

M and V are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

3.3.4 Nadam

Nadam (Nesterov-accelerated Adaptive Moment Estimation) [5] thus combines Adam and Nesterov accelerated gradient. For Nesterov accelerated gradient:

$$g_t = \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1})$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

Modify Nesterov accelerated gradient:

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t)$$

Expanding the second equation:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

4. Parameter Selection

We start by implementing a neural network with 7 convolutional stacks, which is recommended in the original paper. For the first four layers, it contains one convolutional layer with batch normalization and relu activation, the activation passing through a pooling layer and dropout by a certain probability. The fifth and sixth layer consist of two convolutional layers and a pooling layer with same padding for output. The last layer contains three convolutional layers and a pooling layer. Digits data go through this network then reshape into vectors, and then the vectors will be calculated through full-connection layer and passed to digit classifier.

Since there are too many parameters to alter in this deep neural network, we begin do it step by step.

Firstly, we used different numbers of dense layers to explore the function of dense layer(Figure 1). We also tried different convolutional kernel sizes and channel filters. Also, we can test how different pooling sizes, as well as strides, making effect on out model.

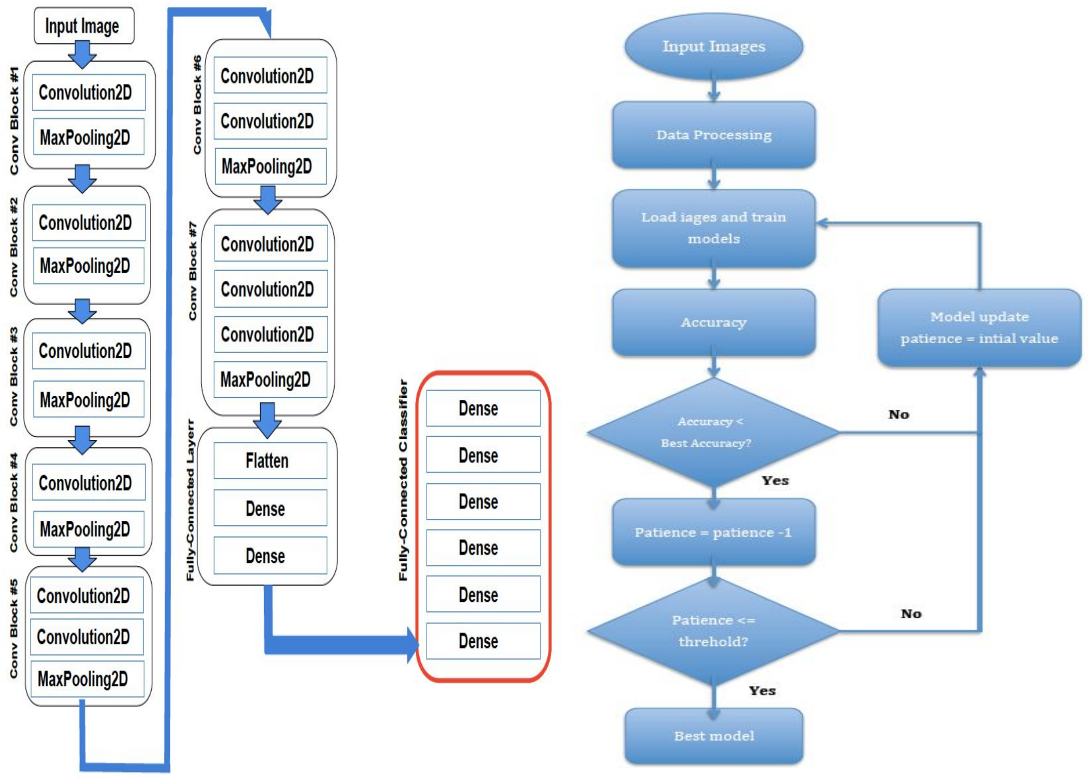
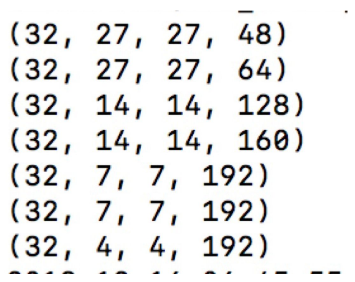


Figure 2: The structure of model

4.1.Training algorithm details

By default, we put the 54 × 54 RGB image to the first hidden layer, using same pooling, the number of neurons is the same as the previous one unless pooling stride bigger than one. Neurons tend to reduce as layer deeper, which may lead to loss of useful information. Thus we enlarged our channel filters from 3 to 48, 64, 128, 160, and 192. To simplify, we set all convolution kernel as 5*5 , pooling size 2*2 , and pooling strides staggering between 1 and 2 through layers. In this model, tensor in each layer is as follows:



```
(32, 27, 27, 48)
(32, 27, 27, 64)
(32, 14, 14, 128)
(32, 14, 14, 160)
(32, 7, 7, 192)
(32, 7, 7, 192)
(32, 4, 4, 192)
```

Figure 3: The structure of tensors through each hidden layer

The original model has 7 convolutional hidden layers and 2 dense layers and dense layer has 3072 units. We deduced one dense layer to explore the performance of these two models.
Next, we tried a new structure which was inspired by ConvNet[2]. The image is passed through stacks of convolutional layers. The first four stacks have the same structure which is one convolutional layer

and one max pooling layer. The next two stacks have the same structure, both of which have two blocks using two convolutional layers connect to each other with one max pooling layer. The last stack has three convolutional layers connecting with each other with one max pooling layer. The convolution stride is fixed to 1 pixel. Max-pooling is performed over a 2 × 2 pixel window, and the stride changes from 2 to 1 in every other layer. We used zero padding to the border to keep the output has the same size as input size in each layer. We also used batch normalization and dropout to improve the performance and to avoid the overfitting. All hidden layers are equipped with the ReLU activation function [6] to fulfill the non-linearity.

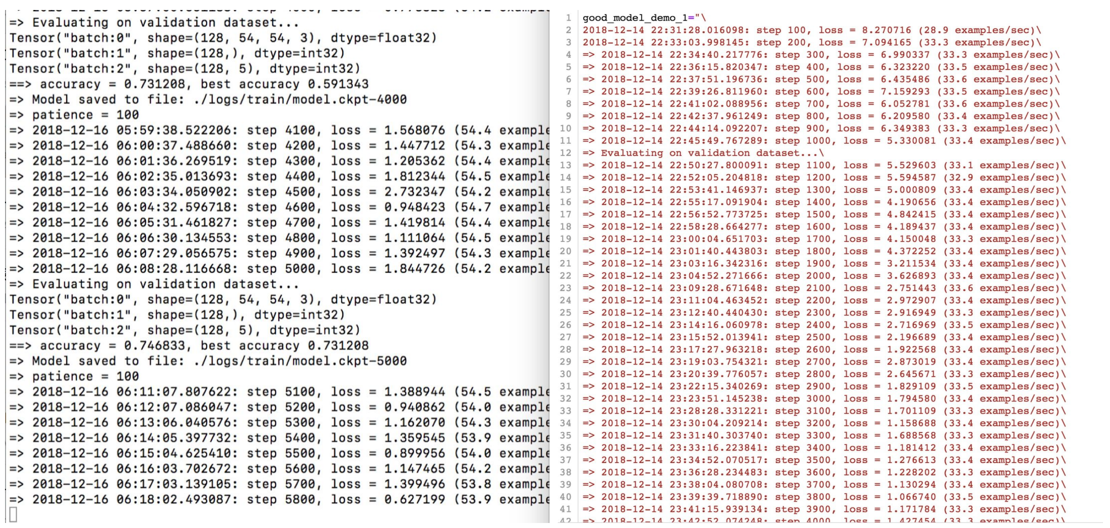
We tried three convolutional filters sizes. The first one is 5 × 5 which is implemented in the original paper. The second one is 3 × 3 which is inspired by ConvNets[2]. The 3 × 3 small receptive field is the smallest size to capture the notion of left/right, up/down the center.

We even tried different max pooling kernel size with corresponding strides. At last, we tried different optimizers such as Gradient Descent, Gradient Descent with Momentum, Adam, Nadam, and Nesterov accelerated gradient in the model which has 3 × 3 filter size.

4.2. Extracting loss and accuracy rate from outputs

We trained the model with input data, at the same time we set a parameter named patience to control when to stop the algorithm, and a parameter named global step to control when to print the outcome and when to evaluate the accuracy. After each 100 global steps, we printed an outcome with a step number and a loss. Every 1000 global steps we evaluated the accuracy with the best accuracy we got before. If this accuracy is higher than the best accuracy, this model will be saved and the best accuracy will be updated, otherwise, it will not be saved and the patience will be decreased by 1. Because about 92% of the accuracy has been reached in 8 hours, with the considering of the limitation of our computing capacity, we used the early stop at 10 hours with a accuracy of 93.68%. When we compared different optimizers, we stop the training at 10000 global steps as more than 80% accuracy in 10000 global steps. (Figure 2)

Since our model is run on Linux command line, the output cannot be used directly to visualize in python. Thus we placed all the output in a txt file, read txt as a string and use regular expression to catch the loss and accuracy rate, the we can use numpy and pyplot to visualize and evaluate each model .



```
=> Evaluating on validation dataset...
Tensor("batch:0", shape=(128, 54, 54, 3), dtype=float32)
Tensor("batch:1", shape=(128,), dtype=int32)
Tensor("batch:2", shape=(128, 5), dtype=int32)
==> accuracy = 0.731208, best accuracy 0.591343
=> Model saved to file: ./logs/train/model.ckpt-4000
=> patience = 100
=> 2018-12-16 05:59:38.522206: step 4100, loss = 1.568076 (54.4 example
=> 2018-12-16 06:00:37.488660: step 4200, loss = 1.447712 (54.3 example
=> 2018-12-16 06:01:36.269519: step 4300, loss = 1.205362 (54.4 example
=> 2018-12-16 06:02:35.013693: step 4400, loss = 1.812344 (54.5 example
=> 2018-12-16 06:03:34.050902: step 4500, loss = 2.732347 (54.2 example
=> 2018-12-16 06:04:32.596718: step 4600, loss = 0.948423 (54.7 example
=> 2018-12-16 06:05:31.461827: step 4700, loss = 1.419814 (54.4 example
=> 2018-12-16 06:06:30.134553: step 4800, loss = 1.111064 (54.5 example
=> 2018-12-16 06:07:29.056575: step 4900, loss = 1.392497 (54.3 example
=> 2018-12-16 06:08:28.116668: step 5000, loss = 1.844726 (54.2 example
=> Evaluating on validation dataset...
Tensor("batch:0", shape=(128, 54, 54, 3), dtype=float32)
Tensor("batch:1", shape=(128,), dtype=int32)
Tensor("batch:2", shape=(128, 5), dtype=int32)
==> accuracy = 0.746833, best accuracy 0.731208
=> Model saved to file: ./logs/train/model.ckpt-5000
=> patience = 100
=> 2018-12-16 06:11:07.807622: step 5100, loss = 1.388944 (54.5 example
=> 2018-12-16 06:12:07.086047: step 5200, loss = 0.940862 (54.0 example
=> 2018-12-16 06:13:06.040576: step 5300, loss = 1.162070 (54.3 example
=> 2018-12-16 06:14:05.397732: step 5400, loss = 1.359545 (53.9 example
=> 2018-12-16 06:15:04.625410: step 5500, loss = 0.899956 (54.0 example
=> 2018-12-16 06:16:03.702672: step 5600, loss = 1.147465 (54.2 example
=> 2018-12-16 06:17:03.139105: step 5700, loss = 1.399496 (53.8 example
=> 2018-12-16 06:18:02.493087: step 5800, loss = 0.627199 (53.9 example
```

```
good_model_demo_1="\
2018-12-14 22:31:28.016098: step 100, loss = 8.270716 (28.9 examples/sec)\
2018-12-14 22:33:03.998145: step 200, loss = 7.094165 (33.3 examples/sec)\
=> 2018-12-14 22:34:40.217776: step 300, loss = 6.990337 (33.3 examples/sec)\
=> 2018-12-14 22:36:15.820347: step 400, loss = 6.323220 (33.5 examples/sec)\
=> 2018-12-14 22:37:51.196736: step 500, loss = 6.435486 (33.6 examples/sec)\
=> 2018-12-14 22:39:26.811960: step 600, loss = 7.159293 (33.5 examples/sec)\
=> 2018-12-14 22:41:02.088956: step 700, loss = 6.052781 (33.6 examples/sec)\
=> 2018-12-14 22:42:37.961249: step 800, loss = 6.209580 (33.4 examples/sec)\
=> 2018-12-14 22:44:14.092207: step 900, loss = 6.349383 (33.3 examples/sec)\
=> 2018-12-14 22:45:49.767289: step 1000, loss = 5.330081 (33.4 examples/sec)\
=> Evaluating on validation dataset...\
=> 2018-12-14 22:50:27.800091: step 1100, loss = 5.529603 (33.1 examples/sec)\
=> 2018-12-14 22:52:05.204818: step 1200, loss = 5.594587 (32.9 examples/sec)\
=> 2018-12-14 22:53:41.146937: step 1300, loss = 5.000809 (33.4 examples/sec)\
=> 2018-12-14 22:55:17.091904: step 1400, loss = 4.190656 (33.4 examples/sec)\
=> 2018-12-14 22:56:52.773725: step 1500, loss = 4.842415 (33.4 examples/sec)\
=> 2018-12-14 22:58:28.664277: step 1600, loss = 4.189437 (33.4 examples/sec)\
=> 2018-12-14 23:00:04.651703: step 1700, loss = 4.150048 (33.3 examples/sec)\
=> 2018-12-14 23:01:40.443803: step 1800, loss = 4.372252 (33.4 examples/sec)\
=> 2018-12-14 23:03:16.342316: step 1900, loss = 3.211534 (33.4 examples/sec)\
=> 2018-12-14 23:04:52.271666: step 2000, loss = 3.626893 (33.4 examples/sec)\
=> 2018-12-14 23:09:28.671648: step 2100, loss = 2.751443 (33.6 examples/sec)\
=> 2018-12-14 23:11:04.463452: step 2200, loss = 2.972907 (33.4 examples/sec)\
=> 2018-12-14 23:12:40.440430: step 2300, loss = 2.916949 (33.3 examples/sec)\
=> 2018-12-14 23:14:16.060978: step 2400, loss = 2.716969 (33.5 examples/sec)\
=> 2018-12-14 23:15:52.013941: step 2500, loss = 2.196689 (33.4 examples/sec)\
=> 2018-12-14 23:17:27.963218: step 2600, loss = 1.922568 (33.4 examples/sec)\
=> 2018-12-14 23:19:03.754321: step 2700, loss = 2.873019 (33.4 examples/sec)\
=> 2018-12-14 23:20:39.776057: step 2800, loss = 2.645671 (33.3 examples/sec)\
=> 2018-12-14 23:22:15.340269: step 2900, loss = 1.829109 (33.5 examples/sec)\
=> 2018-12-14 23:23:51.145238: step 3000, loss = 1.794580 (33.4 examples/sec)\
=> 2018-12-14 23:28:28.331221: step 3100, loss = 1.701109 (33.3 examples/sec)\
=> 2018-12-14 23:30:04.209214: step 3200, loss = 1.158688 (33.4 examples/sec)\
=> 2018-12-14 23:31:40.303740: step 3300, loss = 1.688568 (33.3 examples/sec)\
=> 2018-12-14 23:33:16.223841: step 3400, loss = 1.181412 (33.4 examples/sec)\
=> 2018-12-14 23:34:52.070517: step 3500, loss = 1.276613 (33.4 examples/sec)\
=> 2018-12-14 23:36:28.234483: step 3600, loss = 1.228202 (33.3 examples/sec)\
=> 2018-12-14 23:38:04.080708: step 3700, loss = 1.130294 (33.4 examples/sec)\
=> 2018-12-14 23:39:39.718890: step 3800, loss = 1.066740 (33.5 examples/sec)\
=> 2018-12-14 23:41:15.939134: step 3900, loss = 1.171784 (33.3 examples/sec)\
```

Figure 4: Output get from cmd and restore in python

After we got outcomes, we built up an API to extract data from outcomes. It includes all the computing history of all models which can allows us to reach the outcomes from every dimension. It also helps us to draw plots to compare the accuracy and loss between two models.

5. Results

5.1. Project Results

Our project did the processing of choosing parameters to get the best model. Firstly, we adjusted the original paper building up a model with 8 convolutional layers and 1 dense layer. The result is 0.939 while the original model got a accuracy of 0.938(Figure 5). We concluded that dense layer does not influence the outcome very much.

Secondly, we implemented a deep neural network model with 11 convolutional layers, 1 flatten layer and 2 dense layers. There are seven stacks, and in last two stacks, we connected two convolutional layers with each other followed by one pooling layer, and in the last stack, we connected three convolutional layers with each other then we used one pooling layer. After the last pooling layer, there is 1 flatten layer and 2 dense layers. We tried two different filter size about this structure. The model with 5×5 size filter got to the 0.9124 and the model with 3×3 size filters got to the 0.899 at 17000 global step (Figure 6,7). Because 3×3 filter size model has a small receptive field, it can be computed 77 examples per second, while 5×5 filter size model can be computed 33 examples per second using the same computation ability: Google Cloud CPUs and GPUs (16 vCPUs, 60 GB memory, and 2x NVIDIA Tesla K80). 3×3 filter size model was computed much faster than 5×5 filter size model. Even though we used a deep neural network model, our computation speed is not slow if we used 3×3 filter size. Therefore this model fits into the broader literature.

We explored different pooling kernel size with accordingly strides. We tried 3×3 max pooling kernel size with the first pooling layer strides are 3. This model got a bad outcome which is 0.86. We concluded that this model does not capture enough information with strides equals to 3. We compared it with the original model which has 2×2 max pooling kernel size with 2 strides (Figure 8). We didn't use average pooling because it is not as good as max pooling.

In the last part, under 1e-4 learning rate, we adjusted four optimization methods based on the model with 3×3 size model because it is faster. With this model, 80% accuracy will be obtained in 10000 global steps, so we set the threshold at 10000 global steps and compared the outcome of models. Adam and Nadam optimizer have a good performance but Momentum and Nesterov can not learn. Adam model reached 0.8654 at 10000 global steps and Nadam model got 0.8672. However, Momentum and Nesterov got stuck at 7%. We also tried different learning rate, because we found that Adam model can not learn under the learning rate under 1E-2 learning rate.

Finally, we chose our best model. We implemented 11 convolutional neural network in the stacks structure with 5×5 kernel size. The pooling kernel size is 2 and pooling strides are 1 and 2 in every other pooling layer. We used Adam optimizer and 1E - 4 learning rate.

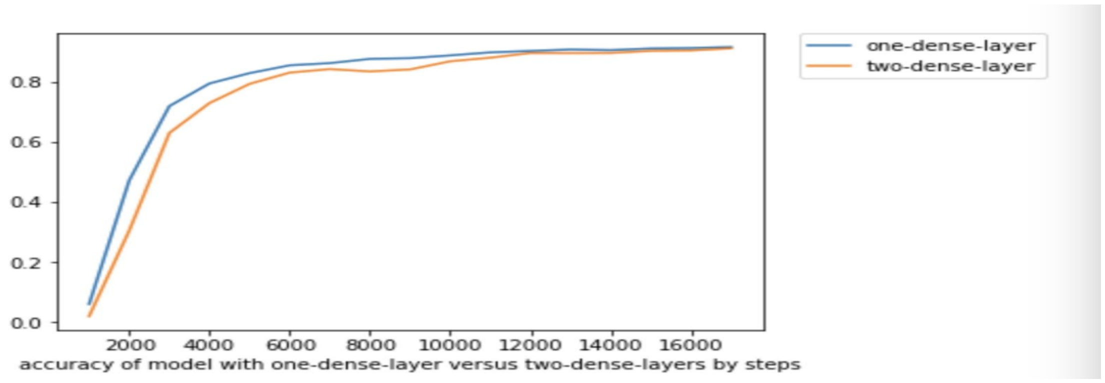


Figure 5

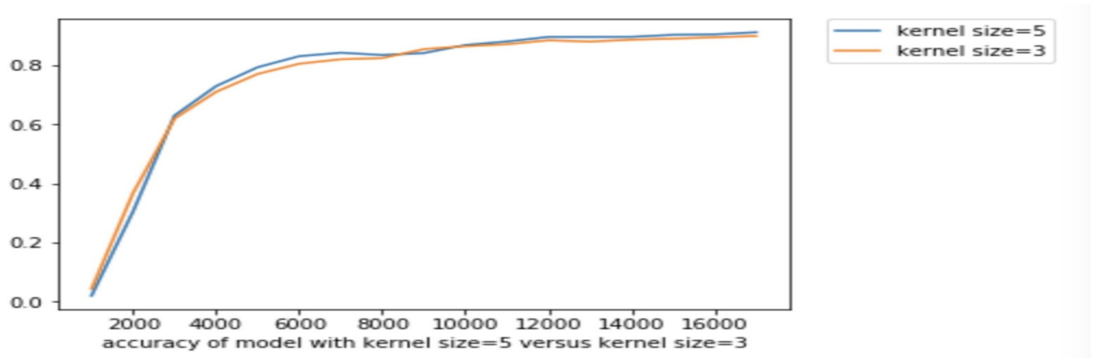


Figure 6

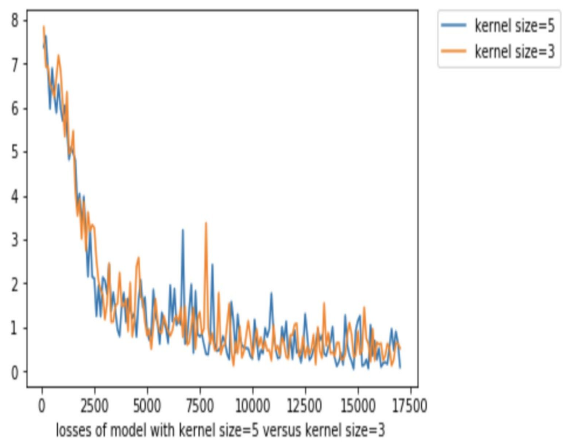


Figure 7

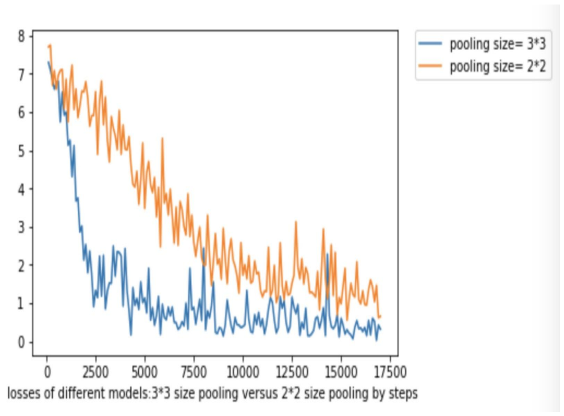


Figure 8

Structure	**Accuracy**
2 dense layers	0.938
1 dense layer	0.939

3 × 3 filter size	0.9124
5 × 5 filter size	0.899

3 pooling kernel size	0.938
2 pooling kernel size	0.86

Optimization	**Adam**	**Nadam**	**Momentum**	**Nesterov**
Accuracy	0.8654	0.8672	0.007	0.008

Under the 1E-4 learning rate, we found that Nadma and Adam optimizer had the best accuracy. The performance of SGD with Momentum can not study at all. We also noticed that the larger learning rate will be bad for Adam optimizer. For the reason of Momentum and Nesterov can not learn, it would be the 1E-4 learning rate doesn't fit these two optimization methods.

5.3. Discussion of Insights Gained

We have tuned convolutional kernel size, pooling kernel size and stride, dense layer, optimizers and learning rate to find the best model on this problem. The model we chose has 11 convolutional layers, 1 flatten layer, and 2 dense layers. The layers are in 7 stacks. The structures in every stack are different. In the last three stacks, there are two or three convolutional layers stack up with Relu activation function and batch normalization followed up. Therefore, we incorporated three non-linear rectification layers instead of a single one, which makes the decision function more discriminative.

Instead of using very large receptive fields, we used very small 3 × 3 receptive fields in our neural network. A stack of two 3 × 3 convolution layers has an effective receptive field 5 × 5, and three 3 × 3 convolution layers have a 7 × 7 effective receptive field. By using 3 × 3 effective receptive we decrease the number of parameters. That is why even though we have a deep neural network structure, the model was computed fast.

We can conclude that dense layer does not influence the outcome so much on this dataset. when Max_pooling kernel size is 2, the model is much better than the model with 3 Max_pooling kernel size. In terms of the optimizer, We may use Adam optimizer to adaptively select a separate learning rate for each parameter. Nadam combines Adam and Nesterov. Learning rate is another important parameter for deep learning model. Large learning rate may lead to being unable to learn, while small learning rate may lead to slow coverage. However, the tuning of learning rate should cooperate with the optimization method, a good learning rate for one optimizer is not necessarily suitable for another optimizer.

6. Conclusion

In this project, we implement several deep neural networks to recognize multi-digit numbers from street view pictures. We have built different models with different model structures, optimizers, learning rates and we achieved 93.68% accuracy with our best model.

Inspired by ConvNets, we implement our model which has many convolutional layers stack up with a small receptive field. Through the comparison of different models, the understanding of tuning parameters was confirmed. We should carefully choose our parameters in the beginning, especially the learning rate, the layer number of neural network and kernel size.

Due to the limitation of the capability of computing engine (Google Cloud) of this project, we did not achieve the same accuracy as the original model. In the future, hopefully, there will be more computing capability to develop this model with more adjustment of this model. We could improve the generalization of a model by randomly shifting, rotating existing images, and other data augmentation methods given more time in the future.

7. Acknowledgment

We would like to thank Prof. Peter Orbanz, Prof. John Cunningham and TAs for all the helpful suggestions on our project.

8. References

[1] 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.
[2] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151.
[3] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.
[4] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.
[5] Karen Simonyan, Andrew Zisserman. (2015) Very deep convolutional networks for large-scale image recognition
[6] Alex, K. Ilya S. Geoffrey, H. (2012). ImageNet Classification with Deep Convolutional Neural Networks.
[7] Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1), 2013–2016.