Learning to recognise multiple digits in real-world images with Convolutional Neural Networks

Machine Learning Engineer Nanodegree

Thomas Almenningen

1. Introduction

PROJECT OVERVIEW

Much of the hype in the data science field is centred around deep learning or deep neural networks and its uses in many modern applications and image vision. ConvNets have improved dramatically since the ImageNet challenge was started in 2010. ImageNet is a large scale visual recognition challenge where the contestants are given a dataset containing 1,000 categories and 1.2 million. Their job is to make the best possible classifier to recognise 1,000 different objects in a test set consisting of 150,000 images. The best models going from a classification error of 28% in 2010 to 3.57% in 2015, which was won using Deep Residual Learning [3].

This project explores how convolutional neural networks (ConvNets) can be used to identify series of digits in natural images taken from The Street View House Numbers (SVHN) dataset [1]. The dataset contains real-world images of street numbers annotated with bounding boxes and labels. Learning to recognise sequences of digits or characters in real-world images is a fairly general problem with many real-world use-case such as reading the number and text from photographs such as e.g. reading the number plates of cars. A previous paper written by Goodfellow et al. [4] which has influenced this work is used a ConvNet to automatically geo-tag some addresses currently in Google Maps.

PROBLEM STATEMENT

The goal of the project is to train a ConvNet on the original SVHN dataset containing multi digit house numbers and measure how it performs on previously unseen images. The model must be able to identify sequences of multiple digits with a high accuracy (>90%) in natural images containing various distortions including varying viewing angles, rotations, background noise etc. Since i was starting from scratch in terms of TensorFlow and Deep Learning i decided to plot my work into three separate steps. The first two steps can be seen as learning exercises while the third is what is discussed in this paper.

1. The first part of the project describes my approach for preprocessing and building a ConvNet to classify single digits on the simpler SVHN 32 x 32 dataset. This dataset consists of over 600,000 32 x 32 RGB images of single printed digits cropped from the original dataset.
2. In the second part of the project I create a synthetic multi-digit dataset of 64 x 64 images using the MNIST dataset and implement a ConvNet architecture that is able to recognise sequences in this simple dataset.
3. In the third and last part of the project I preprocess the original SVHN by cropping the images based on their bounding boxes as described in [1] and converting them to 32 x 32 greyscale images. Here i experiment with different architectures and try to implement the general recommendations in terms of network architecture and hyper parameter settings to achieve the highest possible classification accuracy.

METRICS

To be able to benchmark my model against that of the original paper i will use to same evaluation metric to evaluate my models. The use-case in the original paper requires that every single digit in the house number to be correct as they prefer not to mislead GoogleMaps users. We therefore evaluate our model using accuracy where we give no partial credit for getting parts of the number correct.

PROGRAMMING LANGUAGE AND ENVIRONMENT

The code in this project has been written down and documented in Jupyter notebooks using Python 2.7 using standard libraries for data and statistics and TensorFlow library to build and optimise the ConvNets. All the models have been trained on the CPU of my MacBook Pro 15" with 16GB of memory. Given my computational constraints i do not expect beating or coming close to any of the benchmark scores.

2. Analysis

DATA EXPLORATION

The SVHN dataset is a real-world images of street addresses. It can be seen as similar to the MNIST dataset [5] but it incorporates over 250,000 images and comes from a significantly harder real-world problem of recognising sequences of digits in natural images of different sizes containing various distortions, viewing angles, rotations, background noise etc. There are 10 classes in the dataset, 1 for each digit and in total there are over 630,000 individual digits in the dataset.

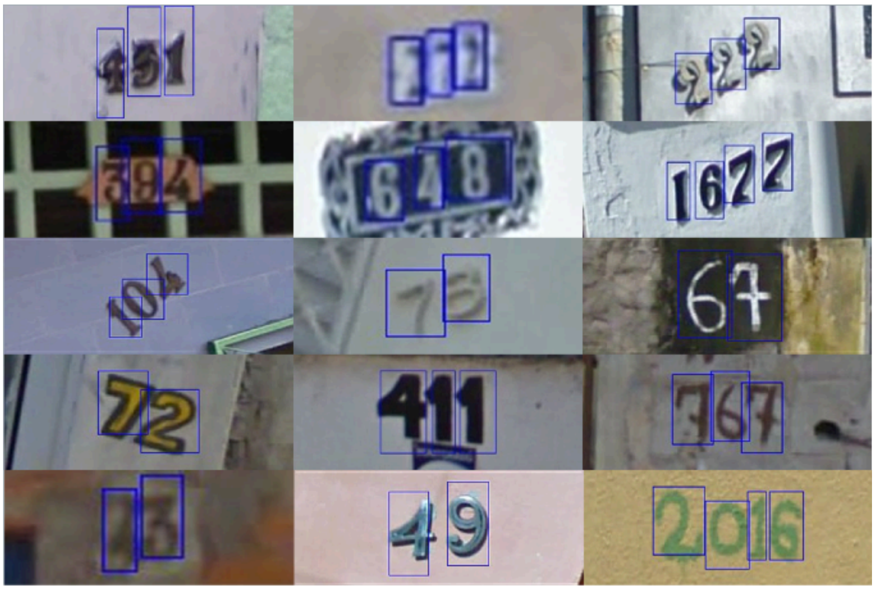


Figure 1: Examples of the raw digits and bounding boxes contained in the dataset.

The dataset consists of 33,401 images for training, 13,068 images for testing and an additional 202,353 images in the extra set gives us roughly 250,000 images in total. There are separate files containing the label information and bounding box information of each individual image. The images are of varying sizes and contain from 1-6 digits.

EXPLORATORY VISUALISATION

There are over 630,000 individual digits in the dataset. The following plot shows the class distribution of the individual digits in the original datasets.

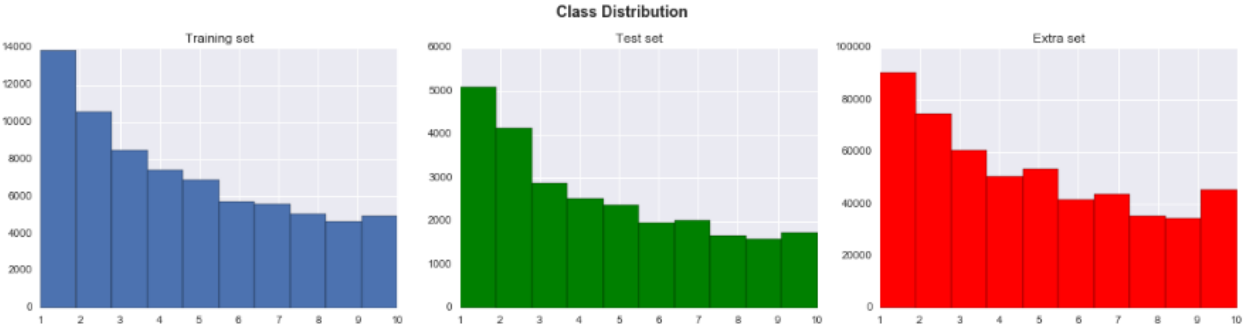


Figure 2: Class distribution in the original dataset

As we can see our distributions have a positive skew meaning that we have an overweight of smaller values which makes intuitive sense given our domain. If we look at the sequence length distribution we get the following plot.

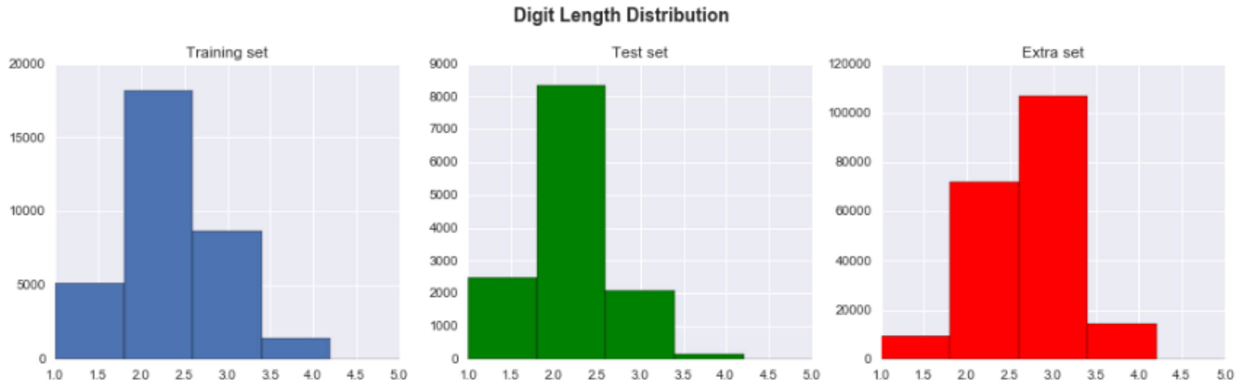


Figure 3: Sequence length distribution in the original datasets

We can see that most images contain 1-3 digits and that 2 and 3 digits are the most common and we have fewer examples of images with 4 or more digits. We even have 1 image containing 6 digits but this is discarded in our experiment.

For a more detailed analysis including sample images and more see the supplied notebooks.

ALGORITHMS AND TECHNIQUES

ConvNets are very similar to ordinary Neural Networks. However, ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture.If you are reading about computers recognising things in videos or images it probably involves a ConvNet. There are three main types of layers used to build ConvNet architectures: Convolutional Layer, RELU layer, Pooling Layer and a Fully-Connected Layer. We stack these together in an architecture.

The ***Input Layer*** (containing the image) typically has three dimensions. The width, height and the number of color channels in the image. A 32 x 32 color image will have a shape or (32 x 32 x 3) while greyscale images only have one color channel (32 x 32 x 1). The width and height should preferably be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

The ***Convolutional Layer*** (CONV) layer is the core building block of a ConvNet. The CONV layer's parameters consists of a set of learnable filters. Each filter is small spatially (along width and height), but extends through the full depth of the input volume. A typical filter of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because the image have depth 3, the color channels). During the pass we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. This means that the network will learn filters that activate when they see a particular visual feature in the image such as e.g. an edge. If you stack multiple CONV layers with 3x3x3 filters on top of each other (with non-linearities in between) each neural on the first CONV layer will have e.g. a 3x3 view of the input volume, the neuron in the second CONV layer will have a 5x5 view of the the input layer and so on. This filters are typically memory intensive and computationally expensive. It is generally preferred to stack small filters than having a single layer with big filters as this allows us to express more powerful features. However this is computationally expensive and might require more memory. The CONV layers should be using small filters (e.g. 3x3 or at most 5x5), a stride of 1 and zero padding.

The **RELU layer** Applies an element-wise activation function, such as max(0, x) thresholding at zero. This leaves the size of the volume unchanged.

The **Pooling layer** perform downsampling of the spatial dimensions in the input. The most common setting is to use max-pooling with 2x2 receptive fields and with stride 2. This will discard 75% of the activations in an input volume. This means that if the input volume is [32, 32, 12] the resulting volume will be [16, 16, 12].

In the **Fully-Connected Layer t**he neutrons in a fully connected layer will have connections to all activations in the previous layer as seen in regular neural networks. The last fully-connected layer is called the "output layer" or the "softmax layer" and in classification settings it represents the class scores.

The most common form of ConvNet architecture stacks a few CONV-RELU layers followed by POOL layers, and repeats this pattern has been merged spatially to a small size. At some point it is common to transition to fully-connected layers and the last fully connected layer will contain the class scores. ConvNet architectures generally follow the following pattern:

*INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC*

BENCHMARK

In the original paper Goodfellow et al. [4] achieved an accuracy of 96.03% on the entire dataset and they reported gaining a half percentage increase from augmenting their training set by randomly cropping the images. There are also a few papers using Recurrent Neural Networks (RNN). Ba et al. [12] report achieving an accuracy of 96.1% using a RNN approach.

3. Methodology

PREPROCESSING

The original SVHN images is preprocessed by finding a rectangular bounding box containing all the individual digits, we then expand this box by 30% in both direction and crop the image contained by the bounding box and resizing it to 32 x 32 pixels. Because of the differing numbers of digits in the image this introduces some scale variability. For single digit images the resulting bounding box will have a large height and small width, cropping the image will therefore squeeze the image vertically. For images containing e.g., 5 digits the bounding boxes will have a greater width than height and cropping the images will squeeze the images horizontally. We also remove the images containing 6 or more digits (1 image in total) and subtract the mean from every image. After the images have been preprocessed they are stored to a HDF5 file.

MODEL

My final architecture is of the form

*INPUT -> [[CONV -> RELU]*2 -> POOL]*2 -> [FC -> RELU]*2 -> OUTPUT*

With dropout applied to the input layer and the fully connected layers. The following table gives an detailed breakdown of my ConvNet architecture.

Layer	Description
Input layer	We input a batch of 64 images (64, 32, 32 ,1)
Dropout layer	Dropout layer with p = 0.9
Convolutional layer 1	64 filters, 5x5 filter size, stride 1, zero padding
RELU	Rectified Linear Unit Activation
Pooling layer	2x2 receptive field, stride 2
Dropout layer	Dropout layer with p = 0.75
Convolutional layer 2	32 filters, 3x3 filter size, stride 1, zero padding
RELU	Rectified Linear Unit Activation
Pooling layer	2x2 receptive field, stride 2
Dropout layer	Dropout layer with p = 0.75
Convolutional layer 3	64 filters, 3x3 filter size, stride 1, zero padding
RELU	Rectified Linear Unit Activation
Dropout	Keep probability of 0.5
Fully-connected layer	128 nodes
RELU	Rectified Linear Unit Activation
Softmax layer	5 softmax activation layers (1 for each digit)

As you can see from the figure the final architecture contains 3 convolutional layers, 4 dropout layer applied between all layers except the softmax layer and a single fully connected layer feeding into a parallel softmax layer, one for each individual digit.

In training deep networks it is usually helpful to anneal the learning rate over time. There are three common ways of implementing learning rate decay: step decay, exponential decay and 1/t decay. In this project i have implemented exponential decay with a starting learning rate of 7.5e-4 and we decay the a decay rate of 0.95 every 5,000 steps using a step function. For my optimiser i used Adam which is an adaptive learning rate method. In practice Adam is currently recommended as the default algorithm to use, and is shown to often work slightly better than RMSProp.

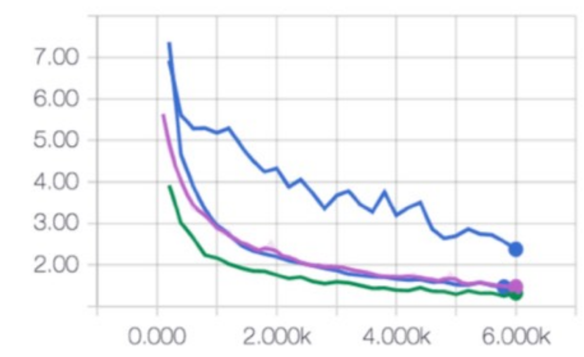
Dropout is one of several ways to control overfitting in neural networks and was introduced by by Srivastava et al. [7]. While training, dropout is implemented by only keeping a neuron active with some probability p, or setting it to zero otherwise. In practice it is most common to combine a single, global L2 regularisation strength that is cross-validation combined with dropout applied after all layers. A reasonable default for p is 0.5.

The weights are initialised using Xavier Initialization proposed by Gloret et al. [10] and the biases are all initialised to zero.

REFINEMENTS

I spent a lot of time tweaking my architecture and hyper-parameters such as learning rate and the different dropout rates in my network. To document all my runs and babysit the learning process i used TensorBoard which ended up being really helpful. For my initial parameter settings i read through the CS231n notes and implemented their recommendations such as starting with a learning rate of 1e-3 and keeping the keep probability of the dropout layer around 0.5, and if applied to the input layer keeping it over 0.75.

Initially i started out with four different model architectures, all following the layer patterns proposed in the CS231n notes and compared them after running 6000 batches on the training set using the TensorBoard graphs. It is recommended to do training for 1-5 epochs or 3600 - 18000 iterations to find the wide hyper parameter ranges and many more for narrower ranges. I therefore decided to test all my architectures running 6,000 batches, evaluate them and build on the best model.


loss/loss

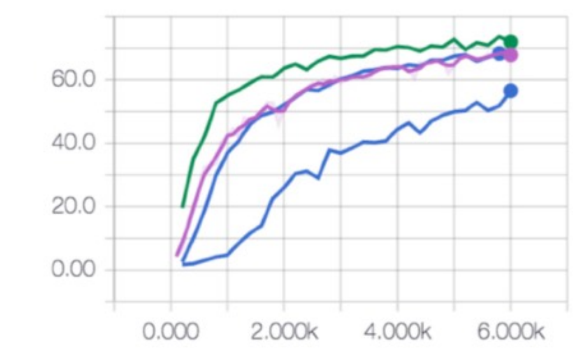

accuracy/accuracy

Figure 4: Loss and accuracy graphs from TensorBoard of the four initial models for the first 6000 batches.

The best initial model was both the fastest and achieved the highest accuracy on both the validation and testset (75.33%) which is far from my goal of getting ~90% accuracy. I then made some additional refinements to the best model such as changing the number of filters in the different layers, changing the dropout rates and experimenting with different learning rates making an additional 8 runs up to 6,000 batches before settling on a final model that i ran overnight.

4. Results

My final model achieved an accuracy of 85.05% on the test set and a 91.5% single digit accuracy. For more detailed results such as the performance on different sequence lengths and the confusion matrix for the individual digits see the notebooks. This is much lower than what i was hoping to achieve and far from the published benchmarks. The following figure shows some of the images they classifier was able tocorrectly classify.

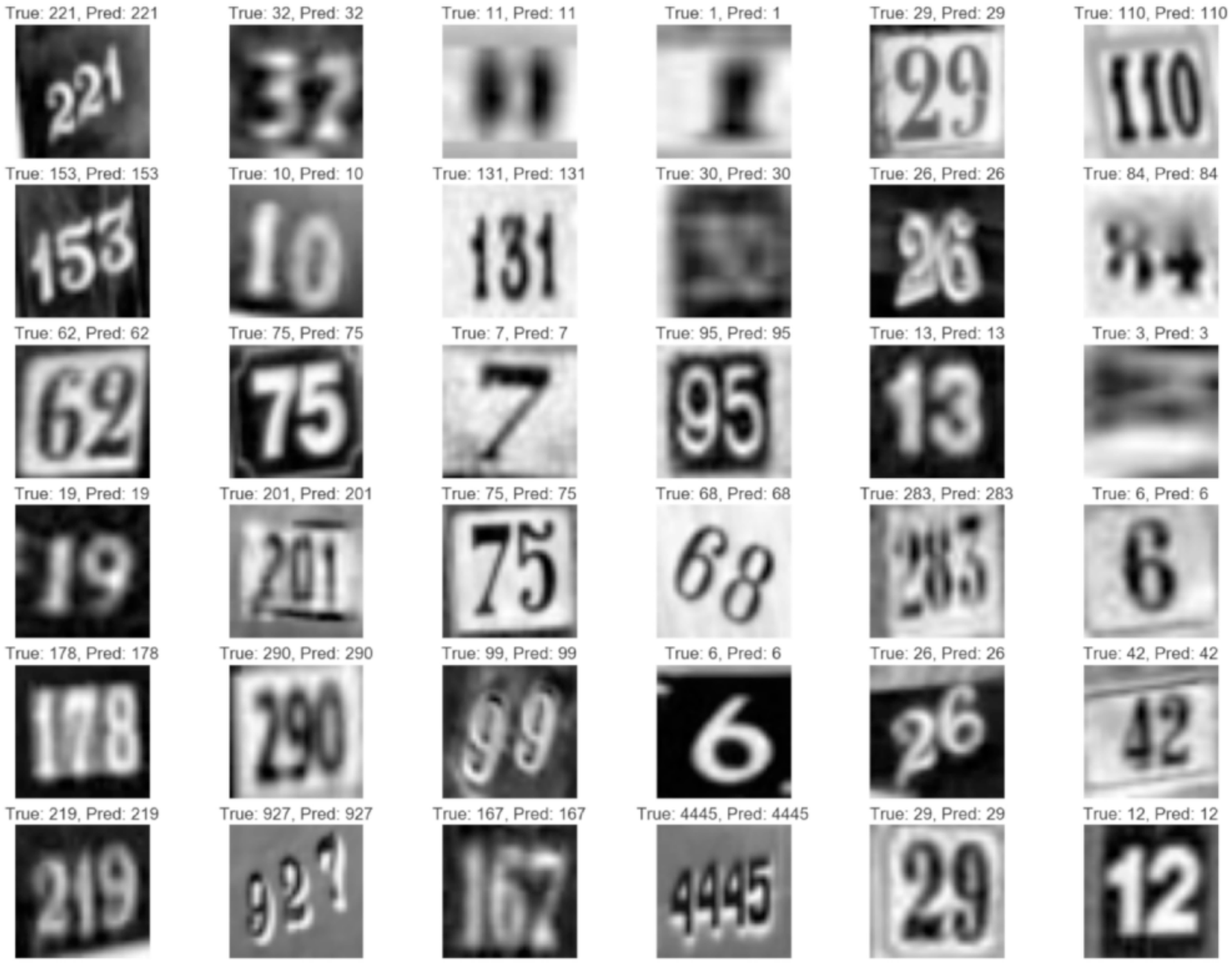


Figure 5: Correct classifications

As we can see from the images the classifier is able to classify some fairly difficult images. The following table shows some incorrectly classified examples.

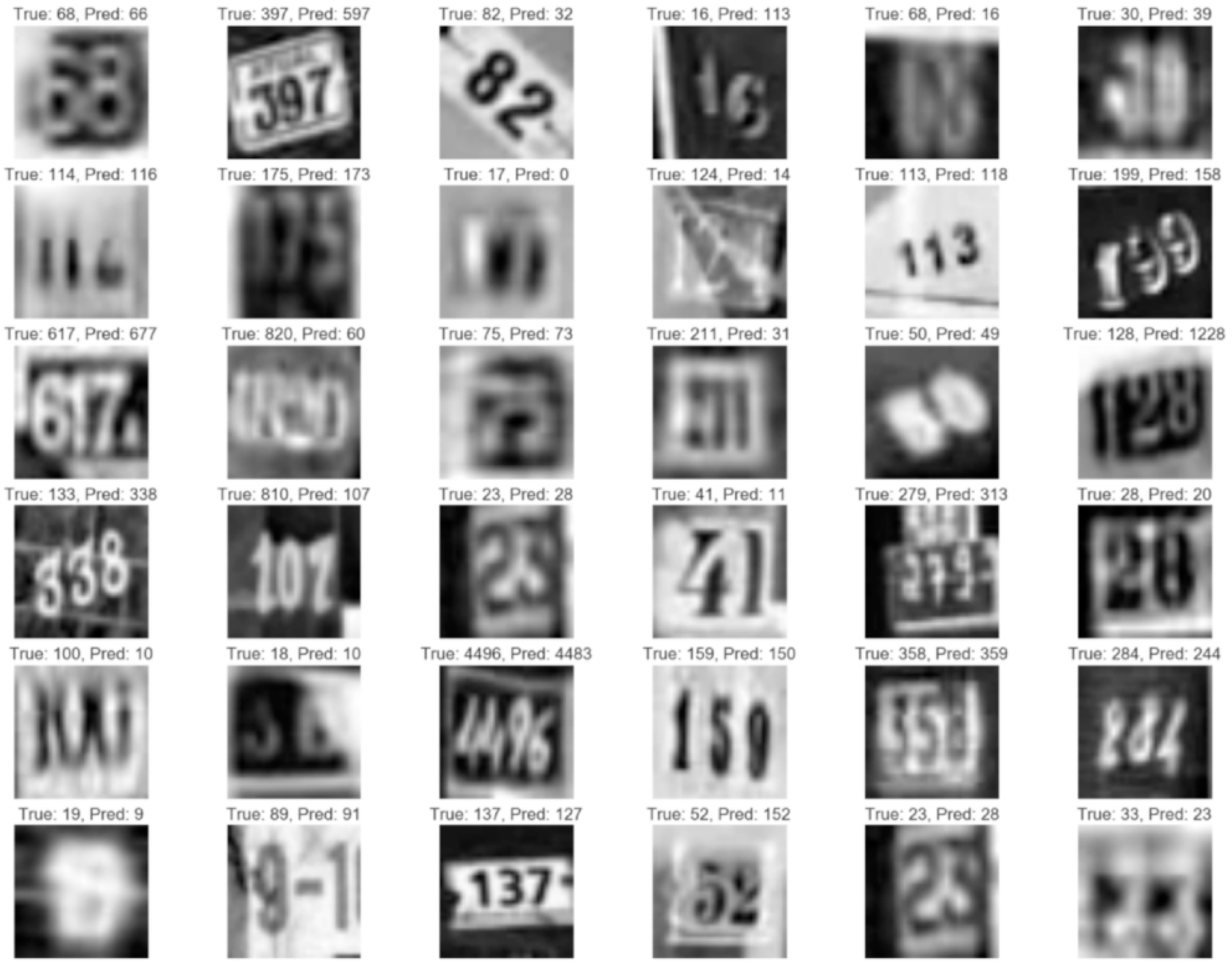


Figure 6: Incorrectly classified examples

Among the incorrectly classified examples there are some really difficult examples that are hard to read, even for me. But it also contains some fairly easy examples which is should be able to classify correctly. The following figure shows the loss and accuracy for the entire training duration for my final model.

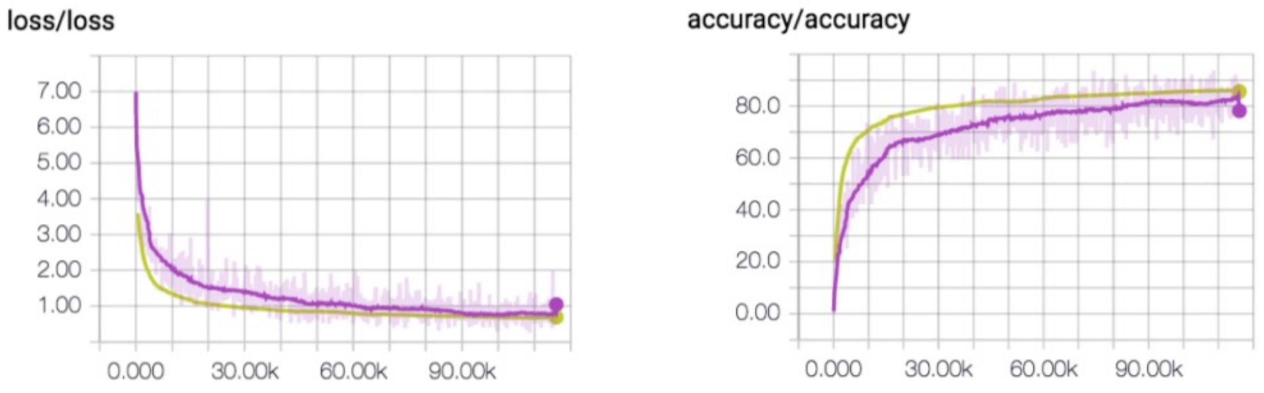


Figure 7: Training and loss graphs from TensorBoard for the final model.

As we can see the training graph the loss the graph looks fairly good but is maybe flattening out a bit too early indicating a too high learning rate. However, both the loss and accuracy kept improving until the last iteration meaning that we probably could have improved the performance slightly be running more iterations.

5. Conclusion

As we have seen I have successfully implemented a simple ConvNet on my laptop that is able to transcribe the digit sequences in the test set by an accuracy of xx.xx% which is lower than the best published results. It is therefore a lot of room for improvement in terms of much much better the model can be made scaling the model up and implementing other smart tricks. I'm glad i Using TensorFlow allowed me to understand fairly low-level detains on ConvNets and not jump as straight into it as i would have done if i had decided to use e.g., Keras. I also think doing this project on my laptop was a big mistake as i ended up spending a lot of time waiting for my runs to finish. I would have saved a lot of time if i had spun up a few instances in Amazon EC2 and performed the runs in parallel. I should also have written the code to be run on a GPU as it was painfully slow to run on a CPU.

As a general rule the experts seem to recommend to "not be a hero"' when it comes to making any decisions related to ConvNet architectures if you are a general user. You should take whatever model works best on ImageNet and tune the three main hyperparameters, and if possible - use a pre-trained model. These models are generally very deep and cannot be trained on your average laptop. Since i ran this experiment locally i tried to follow the general recommendations i could find in the Stanford CS231n notes and create a model that fit into my computational budget. This meant scaling down the images to 32 x 32 pixels, lowering the volume of the CONV layers and using fewer layers than what has been used to achieve state-of-the-art results.

As for future work I'm hoping to be able to implement some of the following improvements to by project:

- Scaling up the model or training a state-of-the-art model such as e.g. ResNet-18 or similar and run it on multiple GPUs instance on Amazon
- Experiment with various data augmentation techniques to introduce random noise in the training images
- Experiment with Recurrent Neural Networks using an approach inspired by the work of Liang et al. [6] and Ba et al [11].
- Improving the cropping procedure to keep the images as square as possible to reduce the amount of scale invariance

- Learn to use TensorBoard more efficiently and reading the learning graphs more efficiently to make more educated guesses on the optimal hyper-parameters
- Move the code to a Python project to simplify the process of running it remote.

References

- [1] - The Street View House Numbers (SVHN) Dataset
- [2] - Large Scale Visual Recognition Challenge (ILSVRC)
- [3] - Deep Residual Learning for Image Recognition
- [4] - Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks
- [5] - MNIST dataset
- [6] - Recurrent Convolutional Neural Network for Object Recognition
- [7] - Dropout: A Simple Way to Prevent Neural Networks from Overfitting
- [9] - Adaptive Subgradient Methods for Online Learning and Stochastic Optimization
- [10] - Very Deep Convolutional Networks for Large-Scale Visual Recognition
- [11] - Multiple Object Recognition With Visual Attention