Deep learning on the Streetview Housenumber Dataset **Michael Wartmann**
Udacity Nanodegree 12/31/2016
Machine Learning Engineer

Preamble

I had started the Nanodegree in July 2016 and aimed at completing within this year. I always wanted to do deep learning for my capstone and the street view house number capstone seemed perfect for this. It is a well structured and bench-marked problem with a bit more guidance available also through the Deep Learning Udacity course by Vincent van Houcke and other resources on the web.

Definition

Pattern recognition in images has recently evolved to one of the most important disciplines of machine learning. Being able to recognize characters, classify pictures automatically has gained enormous importance in the digital age. Even in the early 2000's, 20% of US checks could have been read by a deep neural network already but being confined to simple character recognition [1]. Neural networks have been studied and used since the early 70's while most of the work and applications had purely academic applications. With the advent of vast computational power and large amounts of digitally stored and easily accessible data starting during the 2000's, neural networks have returned to play a significant role in academia and in practice.

Until the early 2010's, the amount of data needed to model and fit deep neural networks with large numbers of parameters was not practical or even possible in many cases and hence confined to simple applications such as 2D character recognition. In combination with using graphics processing units for computation, very large and deep networks could be trained with millions of images efficiently in particular with the parallelization of neural network training using GPU's [2]. A breakthrough in imagine recognition had been the performance of Alexnet in 2012. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton created a deep convolutional neural network that was used to win the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) [3]. AlexNet's performance was well ahead of most competitors and the performance at the previous ILSVCR event in 2010.

Project Overview

The work investigated in this capstone project has been to train an artificial convolutional neural network to perform digit recognition on real image data. The work can be attributed to the domain of image/character recognition in the field of computer vision. In many applications nowadays, relevant information can be extracted from digital images automatically via advanced algorithms. In the recent past, much work has been carried out starting with simple applications such as recognizing handwritten digits [9], to recognizing number sequences [4] on photographs. This work aims at showing how a standardized test data set such as the Google Street View House Number data set [4], can be used to effectively train an neural network to recognize number sequences in images. This project is the standard capstone of the Udacity deep learning course by Vincent van Houcke which served me a well defined starting point to learn about deep learning and Google's Tensorflow. Related datasets for character recognition are for example MNIST [9] with 70000 handwritten characters.

The Street View House Numbers (SVHN) dataset [4] is a dataset of about 600k digits total. The images come along with a matlab files describing the bounding boxes of individual digits.

Problem Statement

The machine learning problem I intend to solve is an image classification/optical character recognition problem. The inputs are png images of different sizes, which have to be pre-processed and converted into a training dataset. The machine learning problem considered is an image classification task. The ultimate goal is to classify a housenumber, i.e., a sequence of digits in a digital image correctly. In a first step, the goal is to recognize single digits in an image and later upgrade to recognizing a sequence of images correctly. The image is correctly classified, if the entire number is correctly predicted.

To solve the task, I propose to build and train a convolutional deep neural network using Google's Tensorflow framework. The goal is a classification performance in which the result reach levels achieved in Ian Goodfellow's paper [13] (within 5-10% deviation).

I intend to use the dataset optimally to learn how to build a convolutional neural network. With the ulitmate goal of being able to recognize a sequence of numbers in mind, we subdivide the project into the following 4 steps.:

1. Pre-processing of the SVHN single image data into a Pandas dataframe. Images are being read from a matlab datafile and prepared into an array format such that they can be fed to Tensorflow.

2. Building a convolutional neural network (convnet) using all relevant components to achieve acceptable performance.

3. Preparing images and bounding box information on the large house number dataset and transforming into a Pandas data-frame.

4. Building a convnet to be able to classify a sequence of digits effectively without feeding the neural network more information than simply the pictures, i.e., learn how to classify a sequence from scratch with no information about length or location of the digits and string of digits.

Metrics

The metric used for this project is the test accuracy of an independent test dataset that has not been used for training. Comparing the achieved accuracy to the performance of several papers like Goodfellow et al. [13] and Sermanet et al. [14] serves as a benchmark for the quality of the work. My expectation is to be in the approximate "layperson" ballpark of the test accuracy of those papers, i.e., within 5-10% accepted deviation.

For both the single digit and the multi digit classification, I have defined the accuracy such that each individual label counts in the accuracy score.

$$Accuracy = \frac{length + digit1 + ... + digit5}{numberoflabels} \tag{1}$$

where each of the variables in the numerator is a binary variable which is 0 if not correctly predicted and 1 if correctly predicted. For a perfect accuracy score, each individual digit and the length have to be correctly predicted, however, if individual digits are correctly predicted, then the accuracy score reflects this.

Figure 1: Images of the 32x32 cropped single digits.

Analysis

The SVHN dataset has 10 classes - 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. The dataset on the 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data. It can be downloaded at *http://ufldl.stanford.edu/housenumbers*.

The images come in two different formats:

1. Images of single digits in already prepared 32x32 cropped format. These images have been provided in a matlab matfile.

2. Images have been provided in all sizes, however they have bounding boxes around the single digits in the sequence of digits. These images were cut out of the full house number and contain distractors, i.e., other digits within the same image off the center.

Data Exploration

The image data is displayed in Fig. 1. Digits have been cropped to a fixed size of 32-by-32 pixels. Square windows have been created without altering the aspect ratio. Nevertheless this pre-processing introduces some distracting digits to the sides of the digit of interest. Loading the .mat files with the python scipy.io package creates 2 variables: X which is a 4-D matrix containing the images, and y which is a vector of class labels.

In Fig. 2 are the original, variable-resolution, color house-number images with character level bounding boxes, as shown in the examples images above. The bounding box information is captures in a matlab mat file and not directly being drawn on the picture. In this work, we will use the bounding box information only to crop the picture to a uniform size - also a 32x32 square image. Each element in digitStruct has the following items: name which is a string containing the filename of the corresponding image. bbox which is a struct array that contains the position, size and label of each digit bounding box in the image. Eg: digitStruct(300).bbox(2).height gives height of the 2nd digit bounding box in the 300th image.

Figure 2: Images of variable size resolution format data.

Figure 3: Images of variable size resolution format data.

Exploratory Visualization

To visualize the data, we can extract a sample of ten images from the existing image matfile, see Fig. 3. Many of the number images are fairly hard to decipher, blurry, distractors are present, different fonts, angles, colors, and distortions. The distribution of data across the different number labels from 0 to 9 is given in the histogram of Figure 4 as a further visualisation of what the most common digits are and if the dataset is balanced. Please note that visualisation of the test and training data histogram has been carried out after the dataset have been randomly split and shuffled. It is apparent that the frequency of the number 1 is highest decreasing towards higher digits. Minority classes hereby are the number 5 and the number 7. Benford's law is clearly recognizable in the dataset, i.e., frequency of the digit 1 is higher than all consecutive digits and follows a decaying trend. Benford's law for base 10 is given as

$$P(d) = log_{10}(1 + \frac{1}{d}) \tag{2}$$

where the probability P of the digit occurring can be calculated as a function of the respective digit d.

Algorithms and Techniques

The algorithm used is Google's open source Tensorflow software library for machine learning. Starting in 2011, Google Brain built DistBelief as their first-generation, proprietary, machine learning system. TensorFlow provides a Python API, as well as a less documented C++ API.

Deep neural networks have recently become interesting as compared to classical neural networks. In a classical neural network, there is (only) one hidden layer which basically allows for one (non-)linear transformation, see 6. Having many more hidden layers allows the nesting of nonlinear activation functions which allow to build very complex functional behavior into the neural network. Until ca. the mid 2000s, it was very hard to train neural networks without running into issues computing the gradients. As gradients are computed using the chain rule,

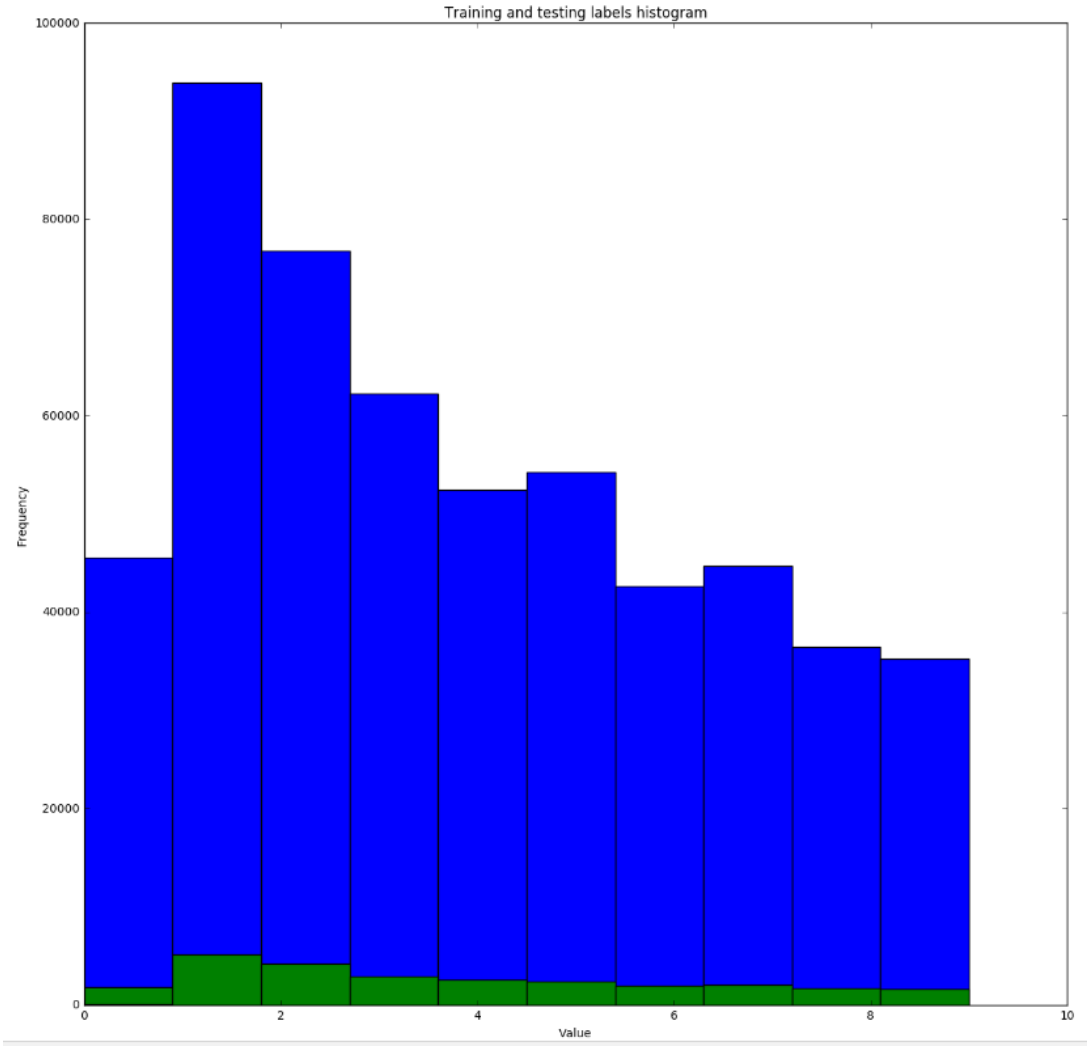

Figure 4: Histogram of the digit frequency distribution for all 10 digits.

deeper layers lead to the so called vanishing gradient problem, i.e., the front layers of the network receive little of the error information. In most recent years, it has been possible to make important steps forward to allow stochastic gradient descent to be used for training deep neural networks. Momentum acceleration in low curvature areas of the search space and random initialization to break symmetry have been crucial in making deep neural network training work. Visual recognition tasks are as such very complex and require an algorithm to learn many detailed visual features for which deep neural network provide the necessary complexity to capture the patterns.

Tensorflow uses data flow graphs, where nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between the nodes. The flexible architecture allows to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed for conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

The following functions of the Tensorflow library have been used in this work:

1. **tf.nn.conv2d**: A key component of a deep neural network for image recognition are

d	$P(d)$	Relative size of $P(d)$
1	30.1%	
2	17.6%	
3	12.5%	
4	9.7%	
5	7.9%	
6	6.7%	
7	5.8%	
8	5.1%	
9	4.6%	

Figure 5: Table of Benford's law for digits of base 10.

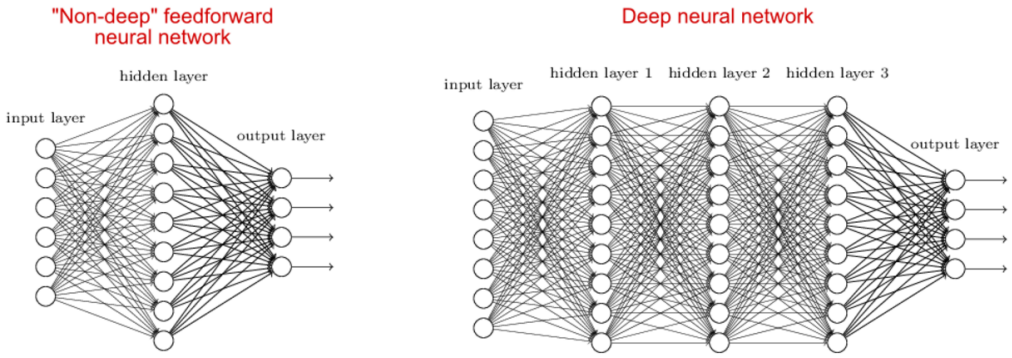

Figure 6: Neural networks: Deep versus conventional neural network.

the convolutional layers. Tensorflow allows simple custom definition of a convolutional layer defining the patch size, stride, and padding. Convolutions are biologically inspired and work similar to the visual cortex - receptive fields (kernels) scan the visual field and convolute the information into a deeper representation (patterns, shapes). Each patch is then represented by one neuron while patches overlap and tile the visual field.

2. **tf.nn.relu**: Rectified linear units (ReLU) serve as activation function in our deep neural network and introduce the non-linearity in the neural network. In the past, a classical activation function has been the sigmoid function. Each node or also called neuron in a neural network has input from a number of edges multiplied by weights added a bias and then passed through an activation function. The advantages of a ReLU unit lies in being a hard max activation function, i.e., it induces sparsity in the hidden units and is computationally advantageous. A ReLU does not face gradient vanishing problem as with sigmoid and tanh function. It has been shown that deep networks can be trained efficiently using ReLU even without pre-training.

3. **tf.nn.local_response_normalization** Normalization is useful to prevent neurons from saturating when inputs may have varying scale, and to aid generalization. Typically, ReLU's do not saturate - any positive input to a ReLU triggers learning in that particular neuron. Krizhevsky et al. [3] still find that local response normalization aids generalization. This sort of response normalization implements a form of lateral inhibition inspired

by the type found in real neurons, creating competition for big activities amongst neuron outputs computed using different kernels.

4. **tf.nn.max_pool**: Pooling is a form of non-linear down-sampling of the given resolution of a convolutional layer where max pooling is the most common one. Essentially, the layer is divided into non-overlapping patches and the maximum value of each patch is passed forward. The intuition is that once a feature in a layer has been found, its exact location is not as important as its rough location relative to other features. The function of the pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control for over-fitting. It is common to periodically insert a pooling layer in-between successive convolutional layers in a deep neural network.

5. **tf.nn.dropout** Dropout is a regularization technique used to avoid overfitting in neural networks. A certain fractions of signals is dropped randomly before the fully connected layer. As such, this forces the neural network during training to create redundancy in the signal transfer and therefore create a more balanced and generalized representation. Neurons which are dropped do not participate in forward and back-propagation and hence the technique reduces the complex, tightly fitted interactions between nodes, leading them to learn more robust features which better generalize to new data. Using dropout extends the number of iterations until convergence roughly by factor 2.

6. **tf.matmul** This is a simple but very efficient matrix multiplication function used by Tensorflow based on its flow graph architecture. It efficiently links matrix operations with calculating derivatives for backpropagation.

7. **tf.nn.softmax** The softmax function is used to map the output of the fully connected last layer in a neural network to a vector of probability for the given labels. Softmax serves as a way of normalizing the neural network output to a set of propabilities that add up to 1.

8. **tf.nn.sparse_softmax_cross_entropy_with_logits** This function measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). The cross entropy is used as a metric to evaluate the prediction of the neural network against the actual label of the data. It serves as an objective function to be minimized by tuning the weights of the neural network during learning/back-propagation.

9. **tf.train.AdagradOptimizer** In this work, an adaptive gradient optimizer is used to minimize the cross entropy. Recent advances in stochastic optimization have enabled the use of proximal functions to control the gradient steps the algorithm takes to improve on the objective function.Using the geometry of the problem at hand is key and the algorithm performs better than a standard gradient descent algorithm.

In total, the algorithm uses the different functions of Tensorflow in sequential form. In most convnets, functions 1-4 are being repeated from the initial image resolution until a desired depth is reached. In the next step, dropout takes care of regularization and in the final step a fully connected layer transforms the layer into label output signals that are then fed into the softmax function. A schematic and the detailed configuration chosen in this work is described in the implementation section and shown in Fig. 8.

Benchmark

According to the paper [14], human performance on the street view house number dataset is 98% accuracy. Goodfellow et al. used a neural network configuration to achieve an accuracy of 96%. Obviously, it is not reasonable to try beat the performance by Goodfellow et al. (cite) for a beginner, but it would be desirable to achieve a performance within 5% of their result and not less than 10% then human performance. As a minimum target, we set 90% (I don't stop trying until this performance is reached), and a maximum target is 93% (after which I would stop trying to get better).

Methodology

Data Pre-processing

Pre-processing is significantly different between the matlab file based test set of the single digits and the actual real png files stored in the tar.gz files. To avoid complexity, I will avoid writing each path separately and integrate the data processing in a more general procedure with remarks where deviations occur.

1. **Download and extraction:** The first step in the data processing is the downloading of the relevant datasets. I have used the functions download_progress_hook and maybe_download from the deep learning class of Vincent van Houcke. In the case of the tar.gz file of the png picture, we untar and extract the images into 3 folders train, text, and extra. For the matfile, we already have a structured format, so we can immediately separate out image and label data and save them in the appropriate array format.

2. **Creating dictionary with image information**: I used the h5py framework to convert the digitstruct h5 data into a python-based dictionary. I have found the pre-processing functions on www.a2ialab.com

3. **Processing into arrays for images and labels:** For both types of input data, I have processed the input into an array of the shape (#ofimages, pixelheight, pixelwidth, imagedepth). According to a number of resources [7], grayscale images have certain advantages when trying to find edges and keep requirements and code simple when training the neural network. As a result, I have converted the RGB image into grayscale with a simple vector multiplication with [0.2989, 0.5870, 0.1140]. For the labels, we have used the digistruct dictionary information to store the labels as a array of the shape (#ofimages, [length,digit1,digit2,digit3,digit4,digit5], i.e., I trained the neural network the length and five digits of the sequence.

4. **Shuffle and pickle:** In the final step of pre-processing, the data is shuffled using the sk.learn train_test_split function such that we can create a large dataset for training from the extra data and set aside a dataset for validation during training and a test set for benchmarking performance after training is complete. The datasets are then stored in a pickle file such that pre-processing only has to be carried out a single time and the pickle file can be used a starting point for repeated training runs.

Implementation

In a first step after pre-processing the arrays are rebuilt from the pickle files and loaded into the tensorflow required format. In the next step a number of helper functions are being defined.

The helper functions serve as a simplifier to be able to build the convnet in simple lines of code. A few of the properties are pre-define as standard the way I programmed the convnet such as the stridelength and padding for the convolutions, the initialization of the weight and bias variables, the kernel size of the max pooling step (2x2), etc. After those pre-definitions, the building of the convnet layers in the main model looks more intuitive and less cryptic. Also, the accuracy function is pre-defined which is used for evaluation after every batch during training.

```
# helper methods for creating CNN
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d_same(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def conv2d_valid(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1],
    padding='SAME')

def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == labels)
          / predictions.shape[0])
```

The initial implementation has been to build a very basic model based on the MNIST tensorflow tutorial. The code of the model is schematically displayed below (this code would actually not compile, it only captures the key elements).

```
def model(data, keep_prob, shape):
        # C1: convolutional layer, batch_size x 28 x 28 x 16
        convolution size: 5 x 5 x 1 x 16
        conv =conv2d_valid(data, W_C1)
        hconv = tf.nn.relu(conv + b_C1)

        # S2: sub-sampling layer, batch_size x 14 x 14 x 16
        maxpool = max_pool_2x2(lrn)

        # C3: convolutional layer, batch_size x 10 x 10 x 32
        convolution size: 5 x 5 x 16 x 32
        conv =conv2d_valid(maxpool, W_C2)
        hconv = tf.nn.relu(conv + b_C2)

        # S4: sub-sampling layer, batch_size x 5 x 5 x 32
        maxpool = max_pool_2x2(lrn)

        # C5: convolutional layer, batch_size x 1 x 1 x 64
        convolution size: 5 x 5 x 32 x 64
        conv =conv2d_valid(maxpool, W_C3)
        hconv = tf.nn.relu(conv + b_C3)

        # Dropout
        dropout = tf.nn.dropout(hconv, keep_prob)

        # F6: fully-connected layer, shape: 64 x 16
        FC = tf.nn.relu(tf.matmul(flattened, W_C4) + b_C4)

        # F7: Output layer, weight size: 16 x 10
        return tf.matmul(FC, W_C5) + b_C5

    # defining logits and loss function
```

```
    logits = model(tf_train_dataset, 0.9, shape)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits, tf_train_labels))

    # Defining learning rate and optimizer
    global_step = tf.Variable(0)
    learning_rate = 0.1
    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss,
    global_step=global_step)

    # Predictions for the training, validation, and test data.
    train_prediction = tf.nn.softmax(model(tf_train_dataset, 1.0, shape))
    valid_prediction = tf.nn.softmax(model(tf_valid_dataset, 1.0, shape))
    test_prediction = tf.nn.softmax(model(tf_test_dataset, 1.0, shape))

# starting session and run computation
num_steps = 200001
with tf.Session(graph=graph,config=tf.ConfigProto(log_device_placement=True)) as session:
    tf.initialize_all_variables().run()
    print('Initialized')
    #training batch by batch
    for step in range(num_steps):
        offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
        batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
        batch_labels = train_labels[offset:(offset + batch_size)]
        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction], feed_dict=feed_dict)
```

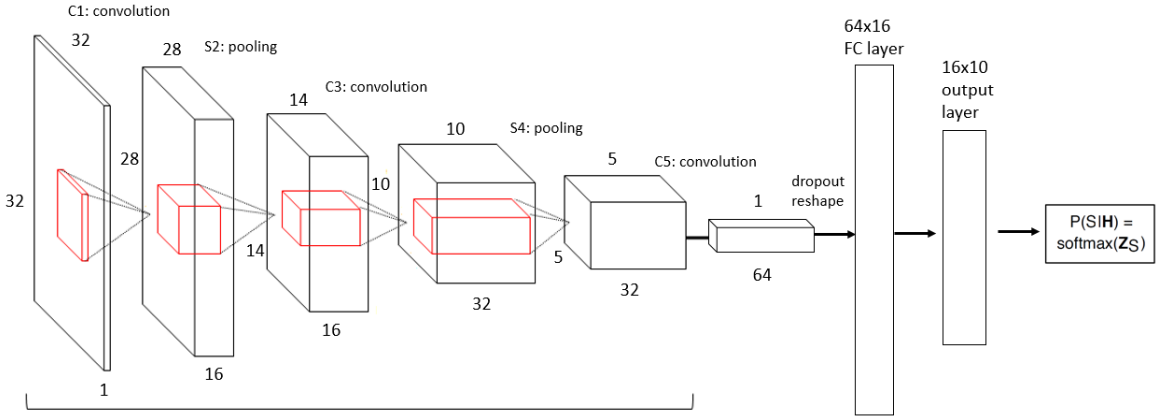

Figure 7: Convnet built for the single digit training.

In first steps, I had trained the model on my personal laptop using the Docker container from the Udacity class. Although having 8 GB RAM and and a reasonable processor with two Intel Core i5 CPU's I very quickly ran both into memory and processing power limitations. Even data processing could already take a very long time or even crash for the full datasets. As such, I have spent very little time further trying to train on my laptop (which is also used for other professional purposes) and used the opportunity to teach myself how to use an Amazon EC2 instance for the training.

The best option was to use the Bitfusion Ubuntu 14 TensorFlow Image. Bitfusion instances come pre-installed with all the relevant tools such as Nvidia Drivers, Cuda 7.5 Toolkit, cuDNN 5.1, TensorFlow 0.11, scikit-learn, Python, Pandas, NumPy, SciPy, Matplotlib, h5py, and Jupyter. I chose an Amazon EC2 P2 Instance together with the Bitfusion image. The p2.xlarge instance has an NVIDIA K80 GPU, 4 CPUs and 61 GiB RAM. As such, I had a guarantee that the hardware requirements would be met and on top of it, I was able to also learn how to use a cloud computing instance which was also one of my training goals.

Refinement

Below, I present a list of refinements and their effects on the performance. The starting point is the above neural network for the single digit learning. The project started out with making the simplest possible convnet work as given in [6]. The basis was using the basic MNIST neural net as a starting point from the tensorflow tutorial and plugging in SVHN data. In a number of steps, I have improved the neural network to incorporate more layers, use AdaGrad optimizer, local response normalization and generally converting into gray-scale and using normalizing in the data pre-processing. I have to admit, that I did not check the impact of each of the refinements on the algorithmic or model side versus actual computational time or predictive performance for a simple reason - every time I would have to start my Amazon instance and pay a dollar, which then lead to that I have implemented a number of improvements at once without knowing which one lead to the step up explicitly. In an environment where my resources are paid for, I would have taken these steps to be sure they are effective.

1. A step of local response normalization has been added after every max pooling operation. I have found this in the Alexnet paper [3]. The impact has been little and inconsistent, it seems that ReLU's are already robust enough. If measureable at all, then in the below 1% accuracy improvement range.

2. Upgrade from gradient descent optimizer to AdaGrad optimizer. The improvement could be resulting in improved training time and more robust performance. I have not explicitly measured the improvement but it seemed tangible.

3. Using an exponential decay learning rate versus a constant decay rate. This seem to have direct influence on the convergence and final performance. With constant rate, it seems that the loss function bounces around too much and cannot improve any further.

4. During pre-processing, I have converted the RGB images into grayscale and normalized the picture data. This also seemed to improve the accuracy slightly (but not consistently measurably).

5. Implementation of the softmax split into 5 digits and length according to the label structure. This is necessary to learn the sequence and align the label information with the model. This set-up is borrowed from Ian Goodfellow's paper [13].

In the next step, I explain how the final model has been built. I have build a convnet starting with a first convolution that maps 32x32 images of depth 1 (RGB had earlier been converted to gray via vector multiplication) to 28x28 of depth 16 (C1) due to valid padding, followed by a max pooling layer reducing the resolution to 14x14 (S2). Then it follows another convolution to 10x10 with valid padding to depth 32 (C3) and max pooling (S4) to 5x5. A last convolution transforms into a fully connected layer after dropout and reshaping. The layer then sees another matrix operation which allows to build a model honoring as labels the length of the house number and the first 5 digits (any further digits are discarded and also do not really occur). After, we feed this to 6 individual softmax functions, 1 for length and 5 for the digits. In the single-digit deep neural network, I essentially use the same structure only that the fully connected layer is not split but fed as a single input to the softmax function. It is remarkable that this model does not use any of the boundary box information. All I do is define the structure of the information, i.e., the structure of the labels and the neural network has to figure out everything else from scratch.

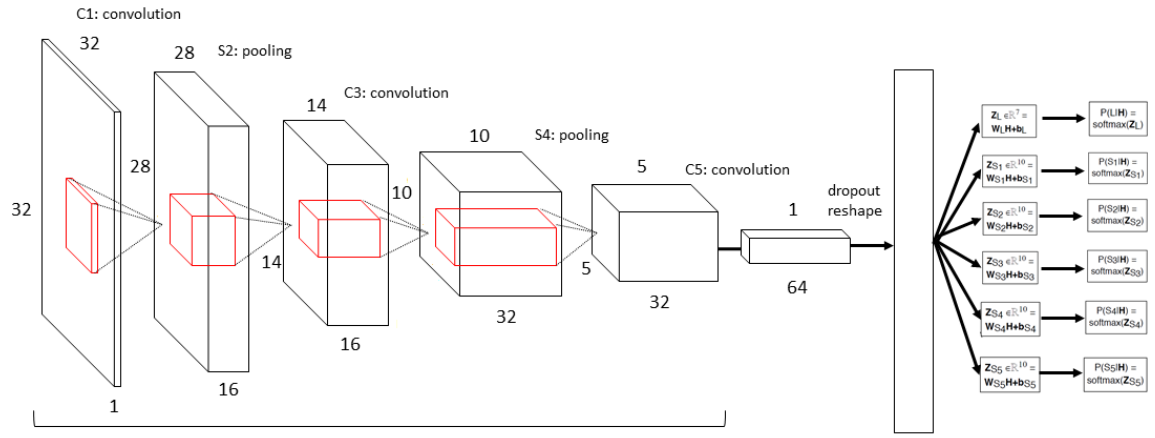

Figure 8: Convnet built for the sequence training.

Results

This section summarizes the results I have achieved using the Amazon P2 instance. My primary goals was to develop my skills in understanding all important steps being carried out for deep learning. I combine my machine learning nanodegree with a PhD in industrial automation (2010) and experience in R&D management - my primary learning goal is to understand the individual tools and applicability of machine learning and their business value in my domain rather than becoming a full-time machine learning engineer. As a result, while having relatively quickly achieved a sufficient performance, I have focused my energy on writing a consistent report and move on rather than long experimentation with the available parameters. Typical rule of thumb values for parameters as a starting point have worked very well and can be readily found in different papers (i.e. 50% for dropout probability according to sexual reproduction theory [8]). For the learning rate, I wanted to use a decaying learning rate and experimented a little but essentially used a parameter set I borrowed from several papers and the tensorflow docu [11].

Model Evaluation and Validation against Benchmark

The first results concern the performance of the single digit convnet. The final model was trained using 20000 images and the training took a few minutes on the P2 Amazon instance. Training started with a learning rate of 0.05 decaying with a factor of 0.95 every 10000 steps. For the dropout keep probability, 0.9 and larger have been shown to be optimal. I have carried out screening for the different training variables as given in the table below (Fig.) for the multi-digit training.

After 20000 steps, the single digit convnet performance indicates an accuracy of 94.1% on the dedicated test set (ca. 5-10 % of image data) with a validation accuracy during training of 96.8%, see below the learning curves of validation and minibatches, see Fig. 10. This seemed sufficient proof of concept to move on to the multi-digit training.

For the multi-digit training, we have a training dataset of ca. 200000 images. To test different parameter combinations, I have varied the initial learning rate, the decay rate, and dropout for 50000 steps training each, while comparing the test accuracies. Each training took ca. 10 minutes - the table in shows the variation in the performance versus the parameters I varied. From the data, it becomes clear that run number 3 gave the best performance .

Run	Data	learning rate start	decay factor	decay step	dropout	val accuracy	test accuracy	duration
1	50000	0.1	0.9	10000	0.5	93.6	93.3	10
2	50000	0.1	0.9	10000	0.8	94.6	93.7	10
3	50000	0.1	0.9	10000	0.95	95.4	94.5	10
4	50000	0.1	0.8	10000	0.95	95.1	94.1	10
5	50000	0.1	0.95	10000	0.95	95.2	94.3	10
6	50000	0.15	0.9	10000	0.95	94.8	94	10
7	50000	0.05	0.9	10000	0.95	95	94.1	10
8	212000	0.1	0.9	40000	0.95	95.8	94.7	43

Figure 9: Training with different parameters. The green line highlights the parameter combination with the best performance.

The evaluation shows that with a dropout keep probability of 0.95, an initial learning rate of 0.1, and a decay factor of 0.9, the model performs best after 50000 images have been trained. Clearly, the parameters could have been further optimized, however, I have reached a performance sufficient versus my initial benchmark and take the approximate optimal parameters as a starting point for the final training with all 200000+ images. It has to be noted that the decay rate has been adjusted to fit the larger training set, i.e., instead of 10000 steps I have set a learning rate update only after each 40000.

The final training shows only small improvements versus the already good performance after 50000 training images. The final best test accuracy is 94.7%.

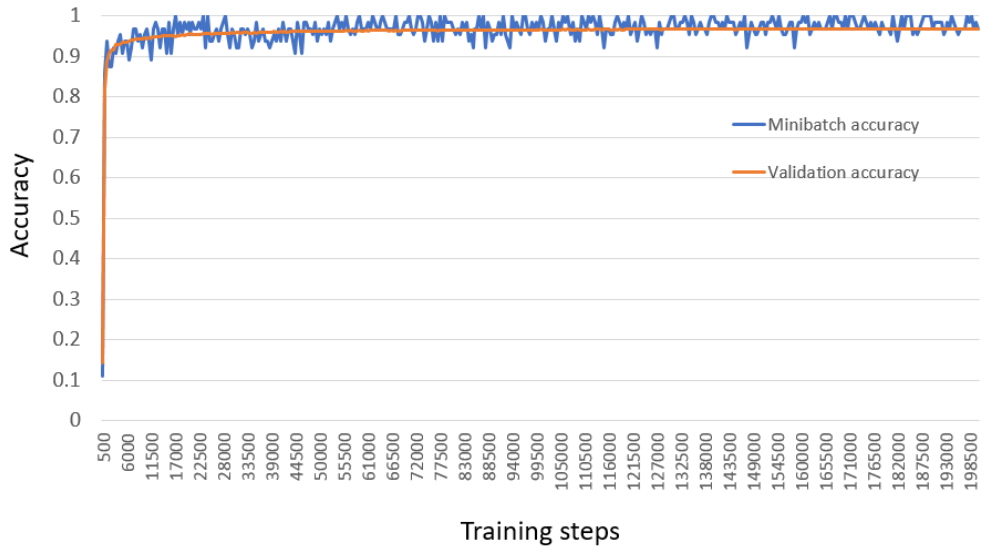

Figure 10: Accuracy score, i.e. learning curves for mini-batches during stochastic gradient descent and the validation set.

Justification

For proper analysis and comparison against the benchmark, formally the t-statistic should be calculated and compared against the null hypothesis of the accuracy of the model being 90% as initially proposed. Calculating the test statistic would require to have an estimate of the variance which cannot be estimated with only one training set. However, if we used e.g. k-fold cross validation, we could probably make a statistically more sound evaluation. Further,

reserving more test data or dividing the test data into smaller batches would also allow us to get an idea of the variance.

Here, the performance of the model lies well above the 90% threshold. A best guess on the basis of the variation on the mini-batch accuracy shows that the standard deviation lies approximately, within 1-2%. To reject the null hypothesis of the model performing no better than 90%, we would need to be at least 95% significant, i.e., roughly two times the standard deviation. We can hence reject the null hypothesis if we have a performance more than 93% as had been stated earlier also. Since the model achieved a performance of 94.7 %, we are safely better than the initial benchmark of 90% (even by 3 standard deviations which would reject the null hypothesis with ¿99% significance).

Conclusions

The pictures in 11 have been chosen to test the accuracy of the model and its predictions on a real set of images. It is great to see how well the model performs even on difficult pictures. One can also see that pictures with lower likelihood of occurrence, such as house numbers with more than two digits have been predicted well.

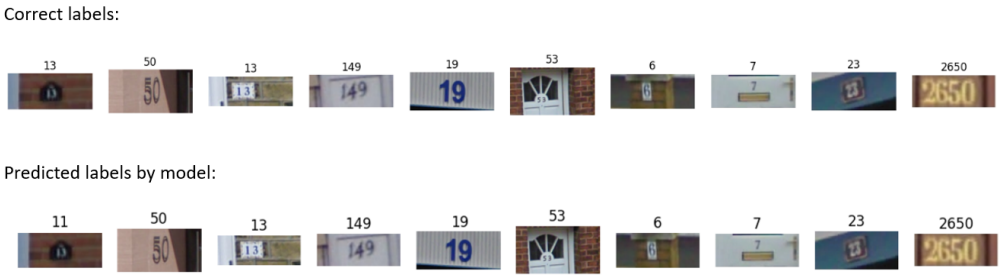

Figure 11: Comparison of labeled picture with predictions by the model

In order to further improve on the performance, we would either need more training images or run several epochs of the same training data to train especially for the house numbers with more digit or funny distortions. Another option is also to create synthetic datasets with distortions and rare types of cases as Goodfellow et al. have done in [4]. Further, to improve the model one could share weight parameters across the digits or even including the training data from the single digit training (if that is helpful). Parameter sharing is essentially passing information of spacial location to the network, i.e., it would work just as if we employ bounding box information to located the digits which essentially makes the problem a sequential quasi single-digit recognition task. The entire fun of this capstone is to have the neural network learn from scratch and not give it any spatial information.

In this work, a set of ca. 250000 images of house number have been used to train a deep neural network to recognize and read the house numbers with accuracy larger than 90%. The task has been accomplished using Google's tensorflow deep learning framework building a convolutional deep neural network with 7 layers involving convolution, max pooling among other techniques. The neural network has been build to predict the length and single digits of a house number up to the length of 5 digits using 6 softmax classifiers being feed with input from a fully connected layer. Dropout has been used as a technique for regularization and a keep probability of 95%. A exponentially decaying learning rate has been used to optimize the stochastic gradient descent based AdaGrad optimizer. The neural network has been trained on an Amazon P2 GPU instance.

The problem was enjoyable to work on. I found it very difficult to get all the infrastructure to work but also learned a lot on the way about tensorflow, docker containers, Linux command line, etc. Further, it was great to learn about the differences and similarities between how humans do image recognition versus computers and that a neural network sounds so very fancy and like a little brain and you start feeling all like Frankenstein, while in fact you are simply using a very advanced algorithm with super-efficient ways of multiplying large matrices and computing large sets of gradients efficiently.

It is still marvelous how the optimization of such a large scale nonlinear system can converge so elegantly. In the world of chemical engineering, parameter fitting is a nightmare since problems are highly nonlinear and sparse making the problems almost singular and ill-defined. As a result, getting any good (meaningful) parameters is a challenge and neural networks might be very valuable also in the context of chemical engineering. It is cool to see how many building blocks there exist in building up convnets with all the layers and it reminds me a lot of building with Lego. In the future, building your own deep neural nets will become even more intuitive than now and who knows even teens and children will be able to experiment around with them. The major difficulty I had was to code out the different detailed steps in pre-processing and making sure I get a good model in tensorflow. I am still not a very good Python coder, but it is nice to see how if you put your mind to it and use the internet efficiently, you can actually piece and patch it together. My dream for the future would be to work a few problems in deep reinforcement learning, if I have time, and that was also my initial goal. This does not fit the nanodegree as a starter task in deep learning but should rather be a succession project. It would be fantastic to have a neural network blow through all my old Atari games from when I was 8.

References

[1] Y. LeCun, L. Bottou and Y. Bengio: Reading Checks with graph transformer networks, International Conference on Acoustics, Speech, and Signal Processing, 1:151-154, IEEE, Munich, 1997.

[2] Adam Coates, Paul Baumstarck, Quoc V. Le, Andrew Y. Ng: Scalable learning for object detection with GPU hardware. IROS 2009: 4287-4293.

[3] Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton: ImageNet Classification with Deep. Convolutional Neural Networks. University of Toronto, NIPS 2012.

[4] 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.

[5] John Duchi, Elad Hazan, Yoram Singer: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Journal of Machine Learning Research 12 (2011) 2121-2159.

[6] https://www.tensorflow.org/tutorials/mnist/beginners/

[7] http://stackoverflow.com/questions/12752168/why-we-should-use-gray-scale-for-image-processing

[8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 15(Jun):19291958, 2014, Journal of Machine Learning Research 15 (2014) 1929-1958.

[9] http://yann.lecun.com/exdb/mnist/

[10] https://www.cs.toronto.edu/ kriz/cifar.html

[11] https://www.tensorflow.org/api_docs/python/train/decaying_the_learning_rate

[12] http://cs231n.github.io/neural-networks-3/#baby

[13] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet: Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks, ICLR2014.

[14] Pierre Sermanet, Soumith Chintala and Yann LeCun: Convolutional Neural Networks Applied to House Numbers Digit Classification, ICPR 2012.

[15] Andrej Karpathy: Connecting Images and Natural Language. Stanford University. PhD Thesis. 2016

https://github.com/hangyao