# STREET VIEW HOUSE NUMBERS RECOGNITION

12/31/2016

Thomas Tilli Capstone Project: Deep Learning

This report describes the approach to build a deep learning classifier for the recognition of street view house numbers.

# 1 TABLE OF CONTENT

# Street View House Numbers Recognition

**THOMAS TILLI CAPSTONE PROJECT: DEEP LEARNING**

## 2 DEFINITION

### 2.1 Project Overview

Recognition of house numbers is an important task in computer vision. the introduction of Google Street View, a large collection of data has been made available for improving the accuracy of maps. The street view images taken of buildings can be analyzed to extract house numbers which can be included in maps. Before this was done manually, which is slow and expensive, limiting the quantity of data that can extracted from these images. With recent advances in automated recognition methods, much larger volumes of Google Street View images can be used in map making.

For research purposes the so-called Street View House Numbers (SVHN) Dataset of Stanford University is available http://ufldl.stanford.edu/housenumbers/ . This images of this dataset were obtained from house numbers in Google Street View Images.

*"SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images."* [1]

SVHN exist in two formats:

[1] http://ufldl.stanford.edu/housenumbers/ 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.* (PDF)

1. Full images: Original images with character level bounding boxes. ach tar.gz file contains the original images in png format, together with a digitStruct.mat file, which can be loaded using Matlab. The digitStruct.mat file contains a struct called digitStruct with the same length as the number of original images. Each element in digitStruct has the following fields: 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. The training set contains 33401 pictures, the testing set 13068 and there is an extra dataset of 202353 additional samples described as somewhat less difficult.

Figure 1: Full Images with Bounding Boxes

2. Cropped digits: 32-by-32 images centered around a single character

Figure 2: Cropped Digits

Each format is divided in 3 part, the train set, the test set and the extra set, which is comprised of less difficult samples, to use as training examples.

## 2.2 Problem Statement

In this project I focus on the recognition of a full house number, which can be a sequence of digits $s= s_1,s_2,..s_n$, $n =(1,2,3,4,5)$. Therefore, I will use the full dataset with the original images with character level bounding boxes. To avoid to find the location of a house number in an image and to recognize it simultaneously, I will use the character level bounding boxes to crop the images to a small area containing only the house numbers. These images will be used to train a classifier for house number recognition. The problem locating a house number could be addressed in different ways, two simple approaches are:

1. Build a regression model for the bounding box around a house number.
2. Use the trained classifier and move a window over the whole image and for each window run that part of the image through the classifier. That rectangle which gives the highest confidence in number recognition is then the bounding box of the house number. This approach is time consuming.

I use deep convolution neural networks (deep CNN, or CNN for short) for the house number recognition. I will implement the CNNs with Tensorflow. I want to experiment with a number of network architectures in order to find the best one for this problem and I want to learn to program Tensorflow low level programming as well.

The tasks involved in this projects are:

1. Download the full data sets.
2. Preprocess the data sets to get the images, the labels and the individual bounding boxes.
3. Compute an overall bounding box for each image which includes all the individual bounding boxes of a house number.
4. Crop the images to the overall bounding box to size 32x32 and for exploration of very deep networks to size 54x54. For regression models resizes the images to size 128x128.
5. Change the cropped images to gray scale, since I believe that for this problem the color is not important.
6. Merge the train set and extra set into one dataset and then split it into a train and a validation data set.
7. Save all data sets (32x32 or 54x54 images) with their labels for classification models.
8. Save all data sets (128x128 images) with the corresponding overall bounding boxes for regression models.
9. Explore the data sets.
10. Build deep CNN networks for classification and explore what model gives the highest accuracy for house number recognition. Save the best classification model.
11. Build deep CNN networks for regression and explore what model delivers the best accuracy to localize the house number on an image. Save the best regression model.
12. Use the best regression model to localize a house number and crop the corresponding area to 32x32 (or 54x54) size and feed it into the best classification model to recognize the house number.

## 2.3 Metrics

For classification accuracy of house number recognition, I will use two metrics:

1. Correct classification of the house number. This means that a house number and the number of digits of a house number is correct at all. For example, the true house number is 357 with 3 digits. If the prediction is 356 with prediction of 3 numbers than the prediction is wrong. If the prediction is 357 with 2 numbers, the prediction is also wrong. This is a quite hard accuracy definition, but a wrong digit in a house number is quite misleading. Therefore, the accuracy metric is defined as:

$$Accuracy = \left(\frac{Number\ of\ correct\ house\ number\ predictions}{Number\ of\ all\ house\ number\ predictions}\right) * 100$$

2. Correct classification of the numbers. This means that the correct prediction of single numbers is measured. This metric is defined as:

$$AccuracyCharLevel = \left(\frac{Number\ of\ correct\ single\ number\ predicitions}{Number\ of\ all\ single\ number\ predictions}\right) * 100$$

For regression accuracy of bounding boxes, I use the Intersect Over Union (IoU) between the prediction and the ground truth bounding boxes[2]:

$$Accuracy = \frac{\sum IoU(precdicted\ BBox, trueBBox)}{Number\ of\ all\ predictions} * 100$$

# 3 ANALYSIS

## 3.1 Data Exploration

The full data set (format 1) consists of 33401, 13068 and 202353 images, colored images (3 color channels). All images have different dimensions. Labels are created during preprocessing and are composed first by the number of digits forming the number, the number itself and filling the rest by the symbol 10, to indicate any eventual blank. Figure 1shows some example images of the full data set. The orientation of the house number is not only horizontal, other directions are there as well. The quality of the images shows a large variation, sometimes the digits are very easy recognizable, sometimes they are very difficult to recognize even for humans. There is a strong variation of the background color and the background texture. That all makes the recognition task difficult.

For further processing the images are converted to grayscale and cropped to 32x32 or 54x54 sizes around the overall calculated bounding boxes around the house number for the classification tasks and for regression tasks the image sizes is rescaled to 224x224 without cropping. The following Figure 3 shows some of the preprocessed images for classification with their corresponding labels:

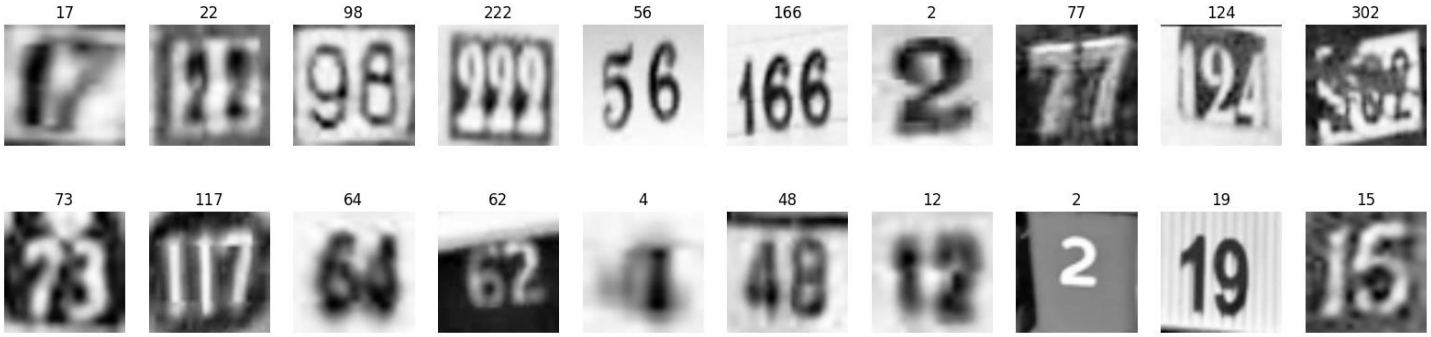



Figure 3: Some preprocessed images with their corresponding labels

This illustrates a further difficulty for classification: the size of the numbers depends on the number of digits, but even for the same number of digits the size of the digit is quite different (e.g “2” in row one and “2” in row two). Sometimes the digits are very blurry and difficult to recognize. This shows that the SVHN dataset is quite difficult to handle by a classifier.

[2] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/object_localization_and_detection.html

For further processing a validation set was created. Following Sermanet et al. (2012)[3], to build a validation set, we select 400 samples per class from the training set and 200 samples per class from the extra set. The remaining digits of the train and extra sets are used for training. This results in a train dataset of 230070 samples, a validation data set of 5684 samples and a test data set of 13068 samples.

## 3.2 Exploratory Visualization

An important characteristic of the data sets is the distribution of the number of digits. The

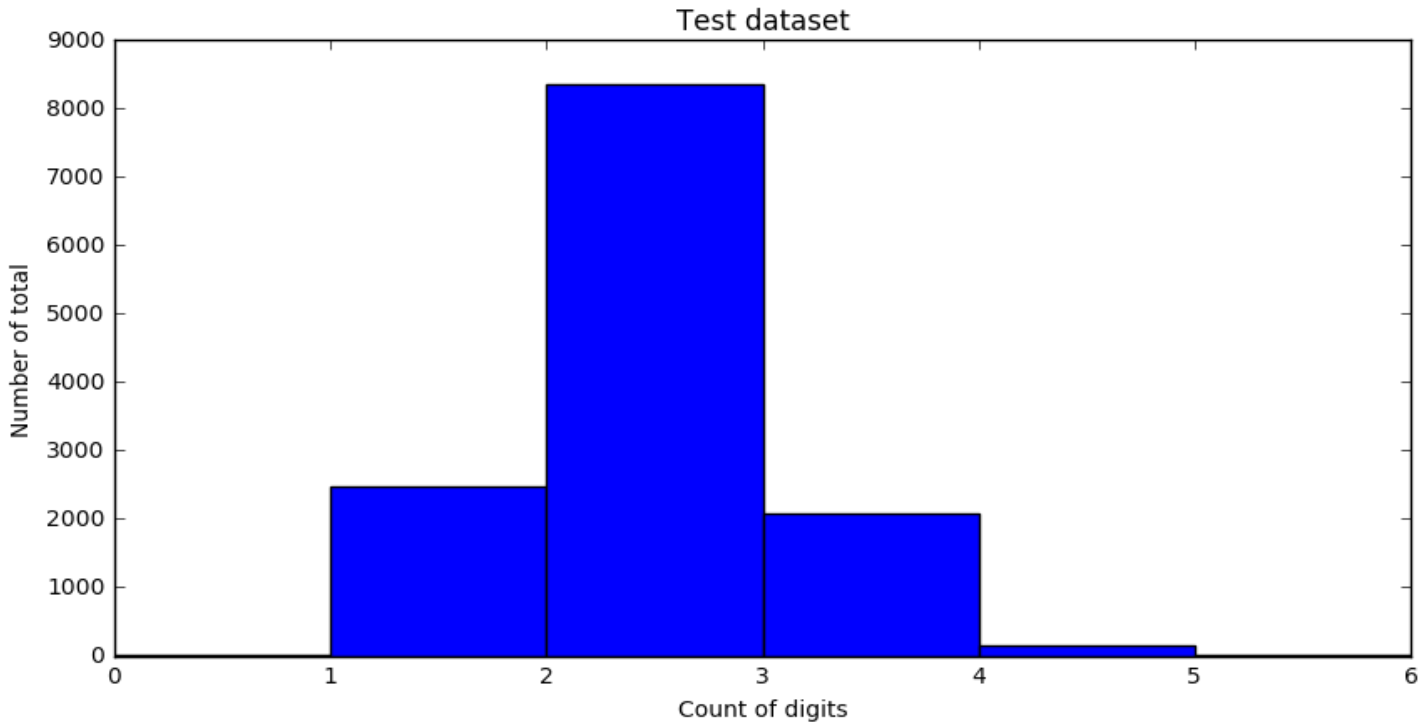
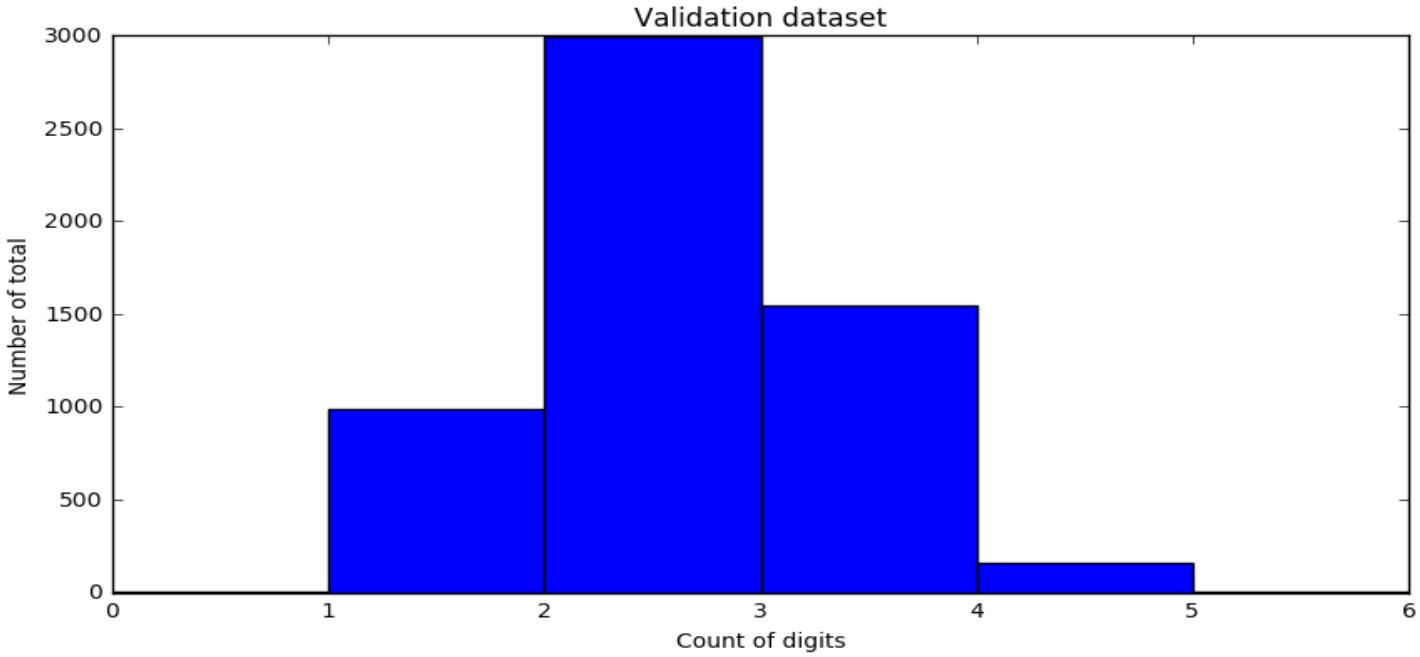



Figure 4 shows the distribution of numbers for the training, validation and test data set.

[3] Sermanet, Pierre, Chintala, Soumith, and LeCun, Yann. Convolutional neural networks applied to house numbers digit classification. In International Conference on Pattern Recognition (ICPR 2012), 2012


Training dataset
Number of total
Count of digits
0
5000
10000
15000
20000
0
1
2
3
4
5
6


Validation dataset
Number of total
Count of digits
0
500
1000
1500
2000
2500
3000
0
1
2
3
4
5
6

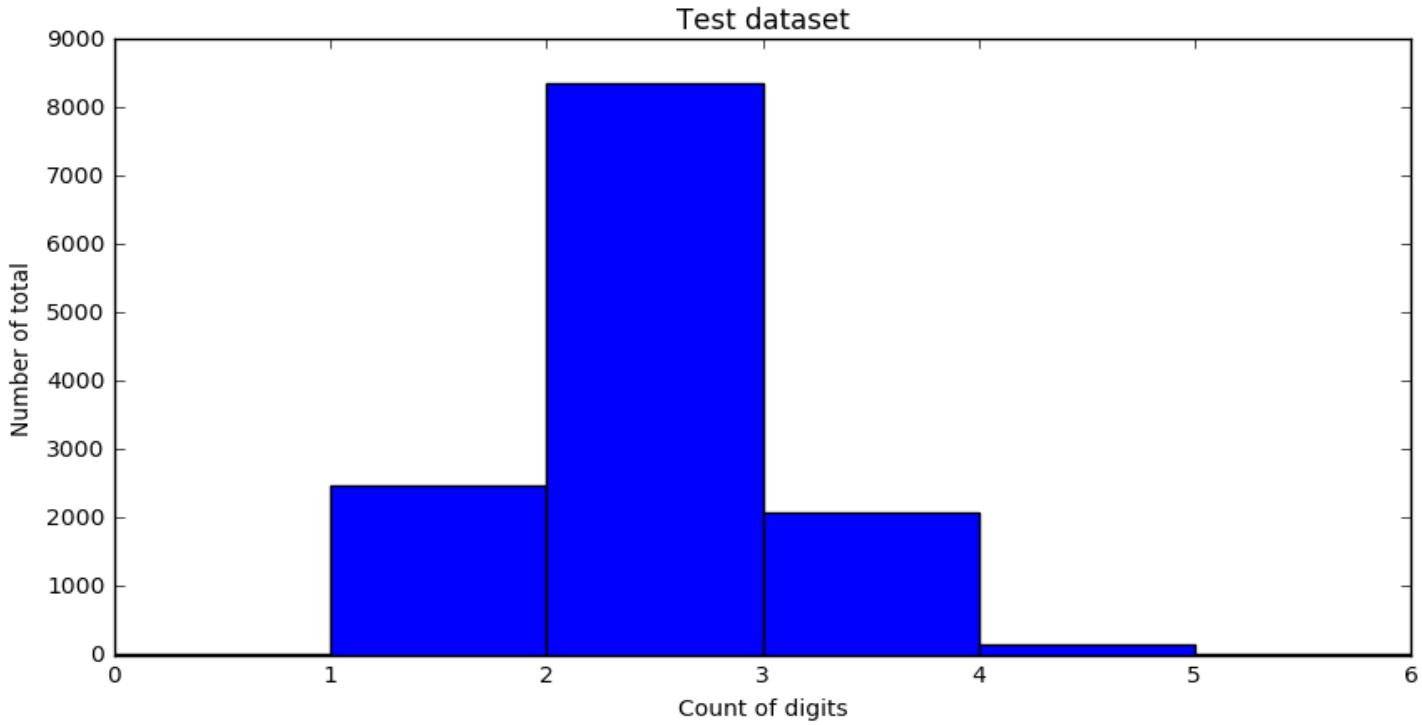



Figure 4: Distribution of numbers in the training, validation and test data set.

The figures show that most house numbers have 2 digits, and house numbers with 1 or 3 digits are quite common too. House numbers with 4 digits are rare. The training data set consists one image with a house number with 5 digits which was removed in the data preparation step.

## 3.3 Algorithms and Techniques

This project follows to some extent the approach of Goodfellow et. Al. 2014[4]. They use a deep convolutional network for the house number recognition. Her architecture consists of eight convolutional hidden layers, one locally connected hidden layer, and two densely connected hidden layers. The first hidden layer contains maxout units (not available in Tensorflow), all other layers use rectifier units. Each convolutional layer includes max pooling and subtractive normalization. The max pooling window size is 2x2. The fully connected layers contain 3072 units each. Drop out was applied to all hidden layers but not the input.

They use data augmentation to increase the size of the dataset by random translations of the images. They obtain an accuracy of 96.03% for the whole numbers and a character lever accuracy of 97.84%. They see a considerable overfitting but they not provide detailed numbers.

I will use the following data preprocessing steps:

1. Extracting images, individual bounding boxes for every digit and label from the training, extra and test data sets.
2. Calculate an overall bounding box around the individual bounding boxes and make it a bit larger on all sides
3. Crop the images to this overall bounding boxes.
4. Convert the images to gray scale
5. Shuffle the training and extra data set.
6. Build a validation set: selecting 400 samples per class from the training set and 200 samples per class from the extra set.
7. Add the remaining extra data set to the training data set.

[4] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet (2014). Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks https://arxiv.org/pdf/1312.6082

8. Augment the training data set with the following operations:
    a. Do N random rotations in the range -20°to 20°
    b. Don N random movements in x and y direction in the range of -5 to 5 pixels.
    c. Do N random sharpness operations
    d. Do N random brightness operations
    e. Do N random contrast operations.

    I use N=2 which results in an augmented training data set of 3681120 images. I do it in memory for simplicity but this requires at lot of main memory - 64 GByte of main memory is required.
9. Shuffle training, test and validation data set.
10. Normalize each data set bet subtracting the mean of each image from each image and divide then by the standard deviation of the image.

For this project, I will test several CNN architectures to build the classifier and the regression model for the bounding box detection. I will start with quite deep networks – at least with 5 hidden layers since the complexity of the problem suggest that networks with a small number of hidden layers will not deliver much test accuracy.

Since deep CNN architectures are not quite common I will provide some details of the building blocks I will use in this project. A very good introduction with many details and applications can be found in the Stanford University course CS231n: Convolutional Neural Networks for Visual Recognition.

### 3.3.1 CNN Overview

Figure 5 shows an example of a regular neural network with one input layer, two fully connected hidden layer and one fully connected output layer. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The problem with regular neural networks for image recognition tasks is, that they do not scale well to larger images. For example, an image of size 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Furthermore, full connectivity is wasteful and such an huge number of parameters results in overfitting.

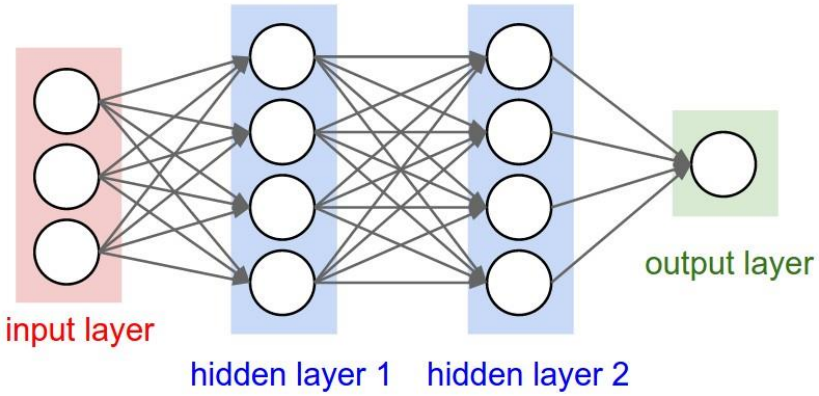



Figure 5 A regular 3-layer Neural Network, source: CS231n

Figure 6 shows an example of a very simple CNN. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. Like other neural networks a CNN is composed of multiple layers. Each layer has a simple function, it transforms a 3D volume of inputs with some differentiable function to an output 3D volume. Unlike a regular Neural Network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. For example, the input images for our problem are an input volume of activations, and the volume has dimensions 32x32x1 (width, height, depth respectively) since we will use gray scaled images.

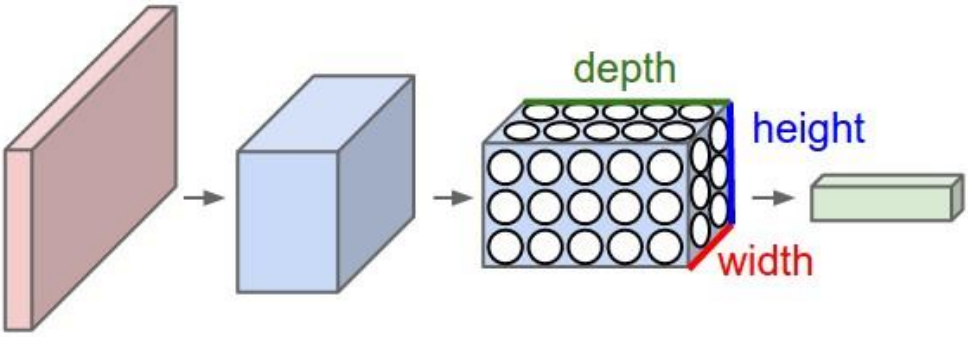



Figure 6: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels), source: CS231n

### 3.3.2 Layers to build CNNs

There are a few distinct types of layers to build a CNN. The most commonly used are CONV (Convolution), FC (Fully Connected), RELU (Rectified Linear Unit), POOL (Pooling) and DROP (Dropout) layers. For example, a simple CNN for CIFAR-10 classification could have the architecture [INPUT-CONV-RELU-POOL-DROP-FC]:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R, G, B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. Mathematically this is a convolution operation. This may result in volume such as [32x32x16] if we decided to use 16 filters.
- RELU layer will apply an elementwise activation function, such as the $\max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x16]). Other activation functions can be used to, like the ELU (Exponential Linear Unit)[5]. The authors claim that CNN with ELUs outperform CNN with RELUs and learning is much faster.
- POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x16]. This layer's function is to summarize and progressively reduce the spatial size of the activations to reduce the number of parameters and computation in the network; correctly setting the hyper parameters also offers some control of overfitting.
- DROP: a simple method to regularize neural networks and to reduce overfitting. This layer is only applied to the training network and keeps only random neurons active. The key idea is to randomly drop units (along with their connections) from the network to prevent overfitting and to force the network to generalize more.
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In summary[5]:

[5] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) . Conference paper at ICLR 2016

- A CNN architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

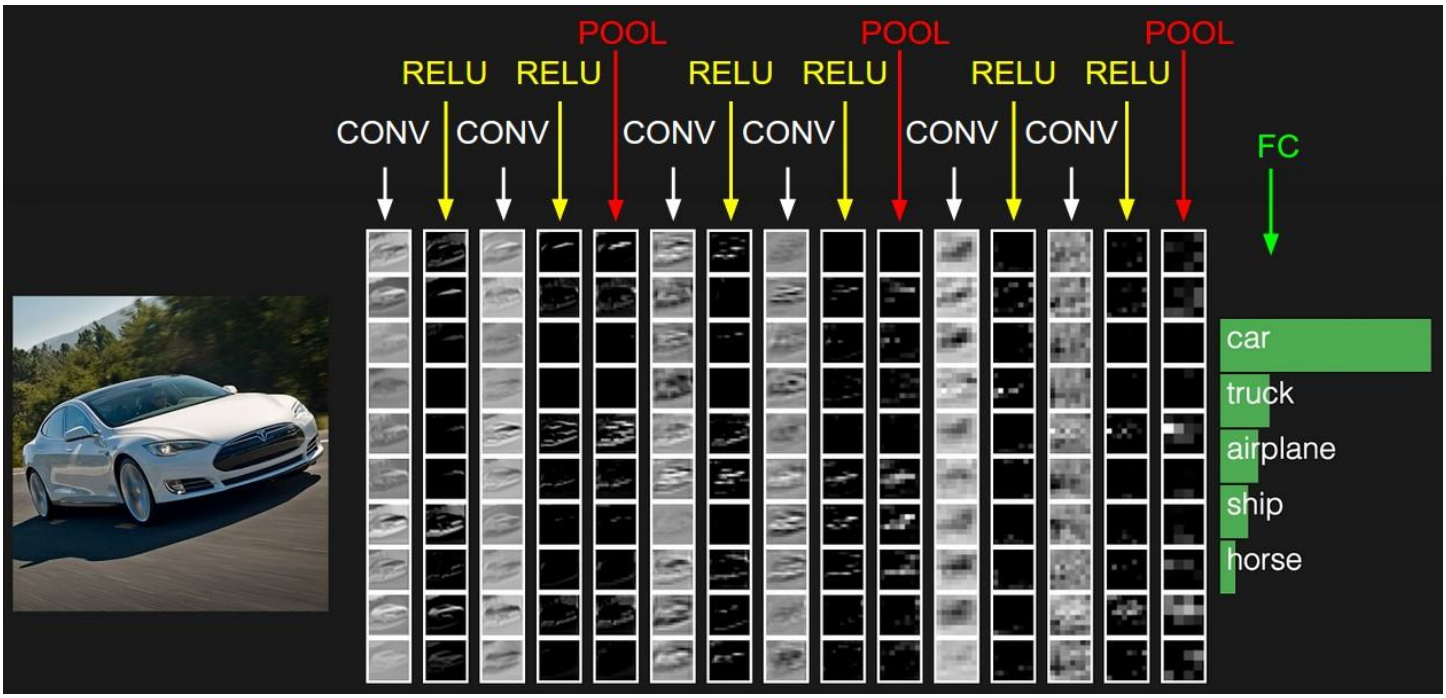



Figure 7: The activations of an example CNN architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one. Source: CS231n.

For a regression model the output layer will not compute a class score but a L2 distance function.

### 3.4 Benchmark

I will use the above-mentioned paper of Goodfellow et. Al. 2014 as benchmark. I not expect to achieve a test level accuracy of 96.03% for the whole numbers and a character lever accuracy of 97.84%. My personal benchmark will be at least 92% test level accuracy and at least a character lever accuracy of 96%. An overview of the best results for the SVHN data set can be found here:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#5356484e

Comparing the results can be misleading, since most authors don't use the full data set but only the cropped data set. This is a single digit recognition task which is much simpler and therefore results in much better accuracy results.

## 4 METHODOLOGY

## 4.1 Data Preprocessing

The first step was to download the source files train.tar.gz, test.tar.gz and extra.tar.gz of the Street View House Number (SVHN) images from the Internet using the URL http://ufldl.stanford.edu/housenumbers/. The code checks if this files exists, if not it downloads the files.

The second step is to extract the downloaded files into three folders: train, test and extra. After extraction, each folder contains a number of images in PNG format and two additional files:

- digitStruct.mat
- see_bboxes.m

The code check if these folders already exist, if they exist, the extraction step is skipped.

I used only the digitStruct.mat file. As described on the SVHN Dataset website:

> The digitStruct.mat file contains a struct called digitStruct with the same length as the number of original images. Each element in digitStruct has the following fields: 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.

For my purposes, I needed a bounding box around the entire house number and not for each digit. I also needed a label dataset for the true house number values of each image.

I used some helper functions to extract the informations form the digitStruct.mat file. See the following python code:

```
def getPictureName(digitStruct,index):
    x=digitstruct[digitstruct['digitStruct']['name'][index][0]].value.tostring()
    name="".join(map(chr, x)).replace("\x00", "")
  #  return digitStruct[digitStruct['digitStruct']['name'][index][0]].value.tostring().replace("\x00", "")
    return name

def getBBoxAttr(digitstruct, index, attr):
    attribute = digitstruct[digitstruct['digitStruct']['bbox'][index][0]][attr].value.squeeze()
    if attribute.dtype == 'float64':
        return attribute.reshape(-1)
    else:
        return np.array([digitstruct[x].value for x in attribute]).squeeze()

def getBBox(digitstruct,index):
    bbox = {}
    bbox['height'] = getBBoxAttr(digitstruct, index,'height')
    bbox['label'] = getBBoxAttr(digitstruct, index,'label')
    bbox['left'] = getBBoxAttr(digitstruct, index,'left')
    bbox['top'] = getBBoxAttr(digitstruct, index,'top')
    bbox['width'] = getBBoxAttr(digitstruct, index,'width')
    return bbox

def getImageInfo(folder,digitstruct,index):
    info=getBBox(digitstruct,index)
    info['name']=getPictureName(digitstruct,index)
    fullname = os.path.join(folder, info['name'])
    im = Image.open(fullname)
    info['imgwidth']=im.size[0]
    info['imgheight']=im.size[1]
    return info
```

```
def getAllImageInfos(folder):
    digitstruct=h5py.File(os.path.join(folder,'digitStruct.mat'),'r')
    return [getImageInfo(folder,digitstruct,i) for i in range(len(digitstruct['digitStruct']['name']))]
```

After extracting this information, I joined the single digits of a house number into a list. The number of digits of a house number is the first digit in the list. The digit "0" is denoted as 10 in the label dataset, I converted it to "0". All house numbers are coded as list with six digits. Non-existing digits are coded as "10". The house number 23 will be coded as [2,2,3,10,10,10], the house number 533 as [3,5,3,3,10,10]. Then I calculate on overall bounding box bases on the bounding boxes of the single digits and make it 20% larger:

```
        #coordinates of bounding boxes
         top=imgInfos[i]['top']
         height=imgInfos[i]['height']
         left=imgInfos[i]['left']
         width=imgInfos[i]['width']
         #calculate bounding box around all digits
         #bounding box shall be little bit larger and shall preserve image proportions
         im_top = np.amin(top)
         im_left = np.amin(left)
         im_height = np.amax(top) + height[np.argmax(top)] - im_top
         im_width = np.amax(left) + width[np.argmax(left)] - im_left
         im_top = int(np.floor(im_top - 0.1 * im_height))
         im_left = int(np.floor(im_left - 0.1 * im_width))
         im_bottom = int(np.floor(np.amin([np.ceil(im_top + 1.2 * im_height), im.size[1]])))
         im_right = int(np.floor(np.amin([np.ceil(im_left + 1.2 * im_width), im.size[0]])))
```

The next step is to crop the images to this bounding boxes and resize them to size 32x32 or 54x54. The last operation is converting the images to gray scale.

As next step, I shuffled the data of the train and extra dataset and then used both to create a validation data set with 400 examples of each class is taking from the train data set and 200 examples of each is taken for the extra data set as described in[3]. Finally, I saved the data sets as pickle files.

I used additional data preprocessing steps for the classification. I applied data augmentation as described above with the following operations:

1. Do N random rotations in the range -20°to 20°
2. Don N random movements in x and y direction in the range of -5 to 5 pixels.
3. Do N random sharpness operations
4. Do N random brightness operations
5. Do N random contrast operations.

These operations are defined in a function:

```
def generateAugmentedData(npimg,label,images,labels,n=1):

    #first append original image
    images.append(npimg)
    labels.append(label)
    im=array2image(npimg).convert("L")
    # do n random rotations
    for i in range(0,n):
        rot=np.random.uniform(-20,20)
        img=im.rotate(rot,Image.BILINEAR)
        images.append(image2Array(img))
        labels.append(label)
```

```
    #do n random movements
    for i in range(0,n):
        dx=np.random.randint(-5,5)
        dy=np.random.randint(-5,5)
        img=ImageChops.offset(im,dx,dy)
        images.append(image2Array(img))
        labels.append(label)
    #do n random sharpness operations
    enhancer = ImageEnhance.Sharpness(im)
    for i in range(0,n):
        en=np.random.uniform(-2,3)
        img=enhancer.enhance(en)
        images.append(image2Array(img))
        labels.append(label)

   #do n random brightness operations
    enhancer = ImageEnhance.Brightness(im)
    for i in range(0,n):
        en=np.random.uniform(0.01,2)
        img=enhancer.enhance(en)
        images.append(image2Array(img))
        labels.append(label)
    #do n random contrast operations
    enhancer = ImageEnhance.Contrast(im)
    for i in range(0,n):
        en=np.random.uniform(0.01,3)
        img=enhancer.enhance(en)
        images.append(image2Array(img))
        labels.append(label)
    return images,labels
```

These function is applied to the whole training data set. I use N=2 which results in an augmented training data set of 3681120 images. I do it in memory for simplicity but this requires a lot of main memory - 64 GByte of main memory is required. The following images shows some examples of the data augmentation:


Figure 8: Some examples results of data augmentation.

The last operation before feeding the data into the CNN is shuffling the training, test and validation data set and then normalizing each data set bet subtracting the mean of each image from each image and divide then by the standard deviation of the image.

## 4.2 Implementation and Refinement of CNN Classifier

I experimented with a lot of CNN models with 54x54 and 32x32 image sizes. For some strange reasons, which are not further explored, Tensorflow can't manage several CNN models within one IPython notebooks maybe due to memory management issues with the GPU. I got either a memory allocation error or a kernel died message and then I have to restart Python. The only way to manage several models in one IPython notebook is to define the whole CNN model inside a method but for all CNN models the same method name must be used (I have no idea what is going on here). Furthermore, I tried to use Tensorboard for visualization but several times I got the "kernel died" message too. This costs me a lot of time and I decided not to use Tensorboard at all.

**Remark**: I run the tests for CNNs for 54x54 and for 32x32 images sizes in parallel, therefore the order of the following descriptions is not the order of the experiments performed.

### 4.2.1 General CNN parameters and elements

Adam-Optimizer was used for almost all CNN trainings.

Padding='SAME' for convolutional and pooling layers. All pooling layers use max pooling.

Convolution layer strides: [1,1,1,1]

Calculation of accuracy:

```
#how much house numbers are correct -> alle digits of a house number must be correct
def accuracy(predictions, labels,show_details=False,n=100):
  if show_details:
        for i in range(0,n):
            print(labels[i],'  ',predictions[i],'  ',np.all(labels[i] == predictions[i]))

  return (100.0 * np.sum(np.all(predictions == labels,axis=1)))/ predictions.shape[0]

#define character level accuracy measuere

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

Calculation of logits:

```
  logits0 = tf.matmul(final2, s0_w) + s0_b  #lenght of house number
  logits1 = tf.matmul(final2, s1_w) + s1_b  #1. digit
  logits2 = tf.matmul(final2, s2_w) + s2_b  #2. digit
  logits3 = tf.matmul(final2, s3_w) + s3_b  #3. digit
  logits4 = tf.matmul(final2, s4_w) + s4_b  #4. digit
return [logits0, logits1, logits2, logits3, logits4]
```

Calculation of loss is defined as the sum of losses if each softmax classifiers:

```
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits0, tf_train_labels[:,0])) +\
    tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits1, tf_train_labels[:,1])) +\
    tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits2, tf_train_labels[:,2])) +\
    tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits3, tf_train_labels[:,3])) +\
    tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits4, tf_train_labels[:,4]))
```

Calculation of the predictions:

```
   [logits0, logits1, logits2, logits3, logits4] = model(tf_train_dataset, 1.0)
   train_prediction = tf.pack([tf.nn.softmax(logits0),\
                               tf.nn.softmax(logits1),\
                               tf.nn.softmax(logits2),\
                               tf.nn.softmax(logits3),\
                               tf.nn.softmax(logits4)])

    [logits0, logits1, logits2, logits3, logits4] = model(tf_valid_dataset, 1.0)
    valid_prediction = tf.pack([tf.nn.softmax(logits0),\
                               tf.nn.softmax(logits1),\
                               tf.nn.softmax(logits2),\
                               tf.nn.softmax(logits3),\
                               tf.nn.softmax(logits4)])
```

```
    [logits0, logits1, logits2, logits3, logits4] = model(tf_test_dataset, 1.0)
    test_prediction = tf.pack([tf.nn.softmax(logits0),\
                               tf.nn.softmax(logits1),\
                               tf.nn.softmax(logits2),\
                               tf.nn.softmax(logits3),\
                               tf.nn.softmax(logits4)])
```

Helper functions to construct the CNNs:

```
# Helper functions
def create_weight_conv(name,shape):
    return tf.get_variable(name, shape,initializer=tf.contrib.layers.xavier_initializer_conv2d())

def create_weight(name,shape):
    return tf.get_variable(name, shape,initializer=tf.contrib.layers.xavier_initializer())

def create_bias(name,shape):
    return tf.Variable(tf.constant(1.0, shape=shape),name=name)


def create_conv_layer(name,data, weights,padding,conv_stride):
    return tf.nn.conv2d(data, weights, [1, conv_stride, conv_stride, 1], padding)

def create_pool_layer(name,data,pool_window_size,padding,pool_stride):
    return tf.nn.max_pool(data, pool_window_size, pool_stride, padding)

def get_conv_shape(data_shape, weights_shape,padding,conv_stride):
    if padding == 'VALID':
        new_height = int(np.ceil((1.0*(data_shape[1] - weights_shape[0] + 1)) / conv_stride))
        new_width = int(np.ceil((1.0*(data_shape[2] - weights_shape[1] + 1)) / conv_stride))
    else:
        new_height = int(np.ceil(1.0*data_shape[1]) / conv_stride)
        new_width = int(np.ceil(1.0*data_shape[2] / conv_stride))
    return (data_shape[0], new_height, new_width, weights_shape[3])

def get_pool_shape(data_shape,pool_stride):
    new_height = int(np.ceil(1.0*data_shape[1] / pool_stride[1]))
    new_width = int(np.ceil(1.0*data_shape[2] / pool_stride[2]))
    return (data_shape[0], new_height, new_width, data_shape[3])
```

### 4.2.2 CNN Architectures 54x54 image size

4.2.2.1 CNN ARCHITECTURE 1 (54X54 IMAGES), 5 CONV LAYERS, 2LRN BEFORE RELU
Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> LRN -> RELU
- CONV2(patchsize1, depth2) -> LRN -> RELU -> POOL (-> DROPOUT)
- CONV3(patchsize2, depth3) -> LRN -> RELU
- CONV4(patchsize2, depth4) -> LRN -> RELU -> POOL (-> DROPOUT)
- CONV5(patchsize2, num_hidden1) -> LRN -> RELU -> POOL
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT

- logits

Parameters used for training:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=64*4
topology_params['num_hidden2']=64*6

learning_params['num_steps']=200001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

Results:

| **Dropout layers in CONV Layers** | **Data Augmentation** | **Test accuracy** | **Test Accuracy Character Level** |
|---|---|---|---|
| No | No | 91.48% | 97.33% |
| No | Yes | 93.52% | 97.89% |
| Yes | Yes | 94.12% | 98.06% |

These results are not very good. The test accuracy character level is quite similar in all three cases but the test accuracy for the whole numbers are quite different. The data augmentation of the training data improves the test accuracy from 91.48% to 93.52% - more than 2%! Therefore, I used data augmentation for all other trainings. Adding 2 dropout layers increases the accuracy further to 94.12%. This is about 2% lower than the benchmark results (96.01%). The character level accuracy 98.06% is better than the benchmark (97.84%).

For the last configuration, the plot of the losses and validation and training accuracy is shown in Figure 9 and Figure 10.

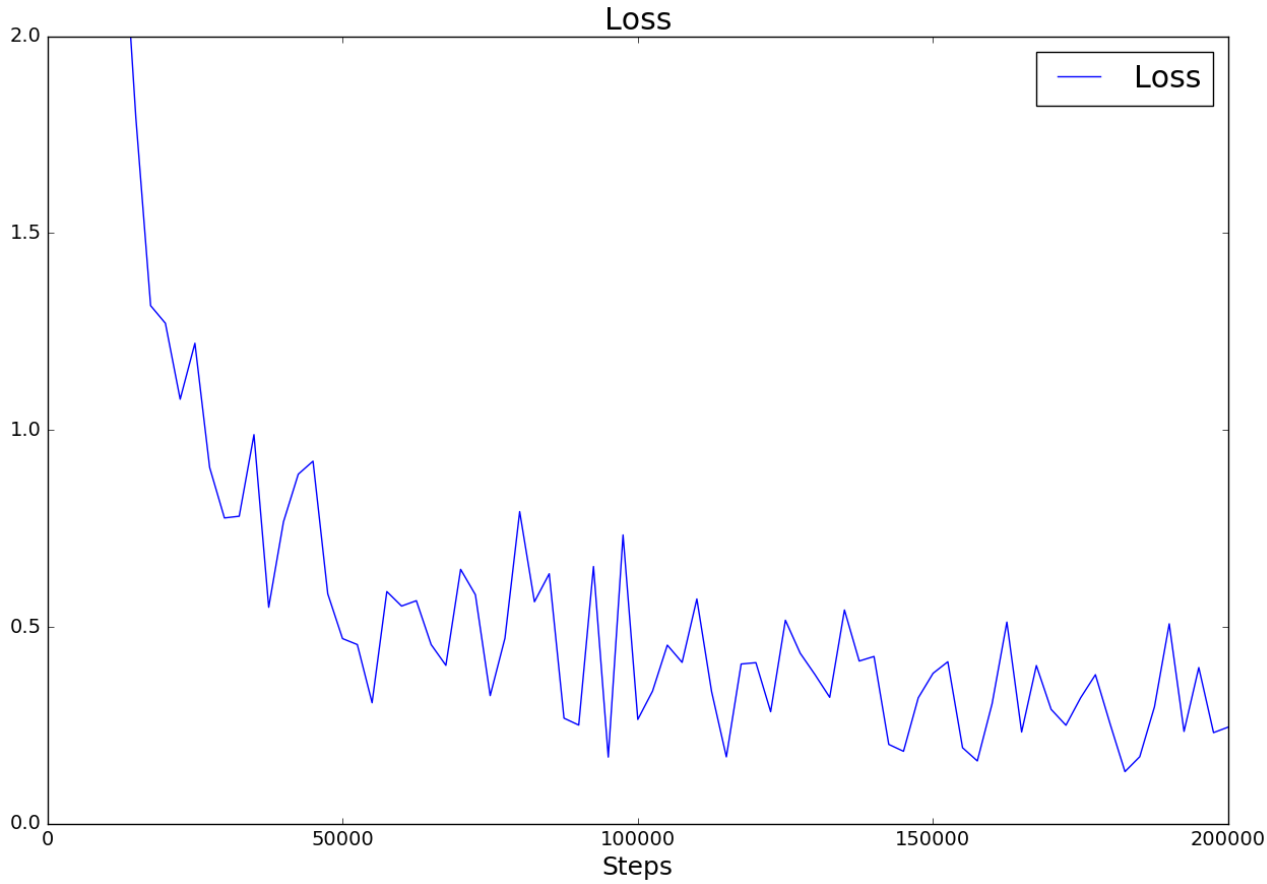



Figure 9: Loss Function for CNN Architecture 1

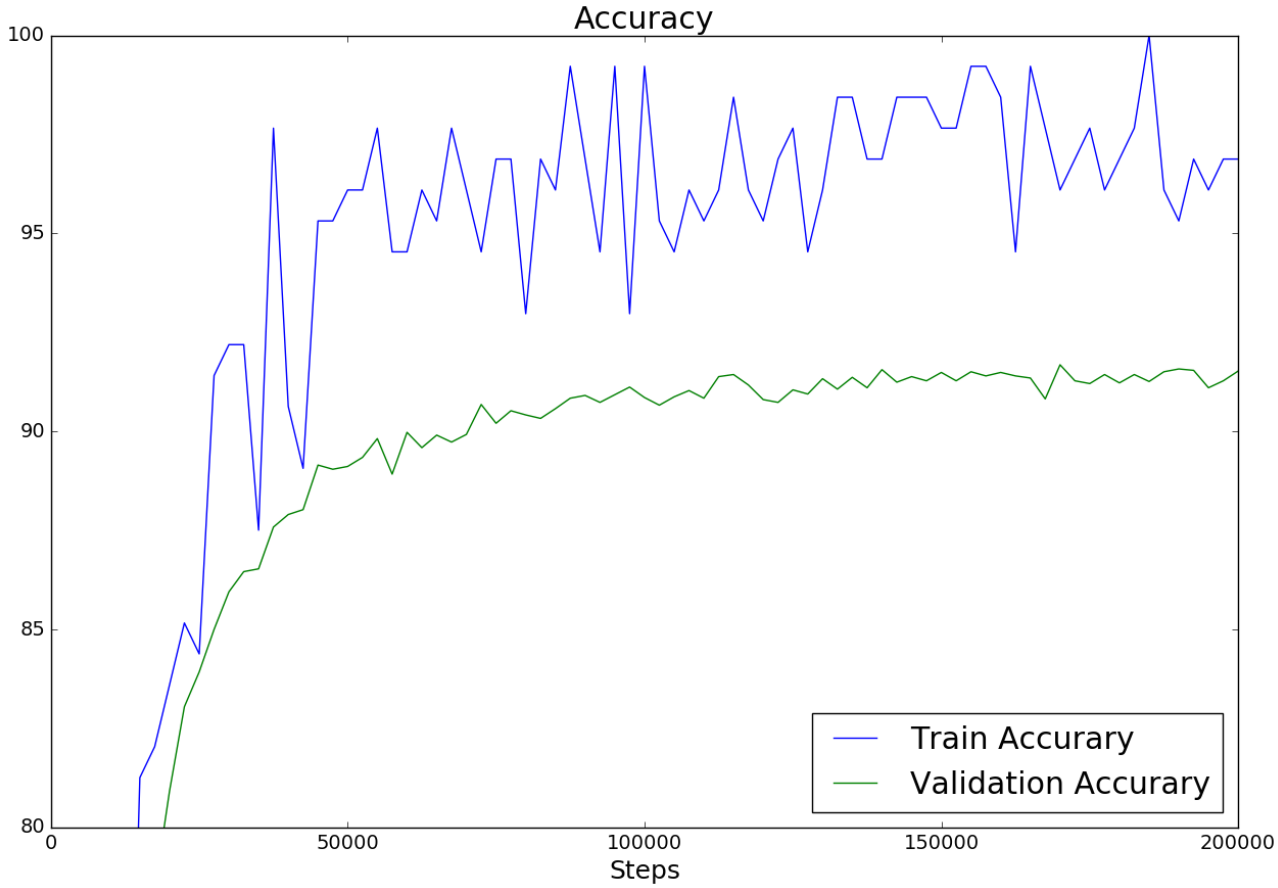



Figure 10. Training and Validation accuracy CNN Architecture1

The training accuracy is quite higher than the validation accuracy (~96% to 91.5%), which shows some overfitting, but test accuracy is 94.12%, therefore the network generalizes quite well.

#### 4.2.2.2 CNN ARCHITECTURE 2 (54X54 IMAGES), 5 CONV LAYERS, 2 FC LAYERS, LRN BEFORE RELU, DROPOUT IN ALL CONV LAYERS

To reduce overfitting, dropout layers where added to all convolution layers.

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> RELU-> LRN -> DROPOUT
- CONV2(patchsize1, depth2) -> LRN -> RELU -> POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> LRN -> RELU -> DROPOUT
- CONV4(patchsize2, depth4) -> LRN -> RELU -> POOL -> DROPOUT
- CONV5(patchsize2, num_hidden1) -> LRN -> RELU -> POOL -> DROPOUT
- FC(num_hidden1) -> DROPOUT

- FC(num_hidden2) -> DROPOUT
- logits

Parameter set 1 used for training:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=64*4
topology_params['num_hidden2']=64*6

learning_params['num_steps']=200001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

Parameter set 2 used for training:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=64*12
topology_params['num_hidden2']=64*8

learning_params['num_steps']=200001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

Results

| **Parameter Set** | **Test accuracy** | **Test Accuracy Character Level** |
|---|---|---|
| 1 | 93.92% | 98.04% |
| 2 | 93.86% | 97.99% |

Adding more dropout layer reduces the test accuracy but not reduces overfitting, but even increases overfitting.

#### 4.2.2.3 CNN ARCHITECTURE 3 (54X54 IMAGES), 5 CONV LAYERS, 2 FC LAYERS, INTERCHANGE RELU AND LRN LAYERS

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> RELU - > LRN
- CONV2(patchsize1, depth2) -> RELU - > LRN -> POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> RELU - > LRN
- CONV4(patchsize2, depth4) -> RELU - > LRN -> POOL -> DROPOUT
- CONV5(patchsize2, num_hidden1) -> RELU - > LRN -> POOL
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Parameter set 1:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=64*4
topology_params['num_hidden2']=64*6

learning_params['num_steps']=200001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

| **Parameter Set** | **Test accuracy** | **Test Accuracy Character Level** |
|---|---|---|
| 1 | 92.17% | 97.32% |

This is a very interesting result!! The only difference to CNN Architecture 1 is that the RELU layer is now before the LRN layer and that reduces the test accuracy by 2%!

#### 4.2.2.4 CNN ARCHITECTURE 4 (54X54 IMAGES), NO LRN LAYERS

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> RELU (-> DROPOUT)
- CONV2(patchsize1, depth2) -> RELU - > POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> RELU (-> DROPOUT)
- CONV4(patchsize2, depth4) -> RELU - > POOL -> DROPOUT
- CONV5(patchsize2, num_hidden1) -> RELU -> POOL (-> DROPOUT)

- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Parameter set:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=64*4
topology_params['num_hidden2']=64*6

learning_params['num_steps']=200001
learning_params['batch_size']=128
```

Results:

| DROPOUT in all CONV layer | Test accuracy | Test Accuracy Character Level |
|---|---|---|
| No | 93.85% | 98.04% |
| Yes | 94.03% | 98.03% |

This results are very similar to the results of CNN architecture 1. The LRN layer does not improve the accuracy much, but training will take much longer time.

#### 4.2.2.5 CNN ARCHITECTURE 5 (54X54 IMAGES), 8 CONV LAYERS, 2 FC LAYERS, LRN BEFORE RELU

This architecture is similar to the CNN described in the benchmark paper.

Padding='SAME' for convolutional and pooling layers.

Pooling strides:

- Stride1: [1,1,1,1]
- Stride2: [1,2,2,1]

- CONV1(patchsize1, depth1) -> LRN -> RELU -> POOL(Stride2)
- CONV2(patchsize1, depth2) -> LRN -> RELU -> POOL(Stride1) -> DROPOUT
- CONV3(patchsize1, depth1) -> LRN -> RELU -> POOL(Stride2)
- CONV4(patchsize1, depth2) -> LRN -> RELU -> POOL(Stride1) -> DROPOUT
- CONV5(patchsize1, depth1) -> LRN -> RELU -> POOL(Stride2)
- CONV6(patchsize1, depth2) -> LRN -> RELU -> POOL(Stride1) -> DROPOUT
- CONV7(patchsize1, depth1) -> LRN -> RELU -> POOL(Stride2)

- CONV8(patchsize1, depth2) -> LRN -> RELU -> POOL(Stride1)
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Parameter set 1:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=32
topology_params['depth2']=32
topology_params['depth3']=64
topology_params['depth4']=64
topology_params['depth5']=64*2
topology_params['depth6']=64*2
topology_params['depth7']=64*3
topology_params['depth8']=64*3
topology_params['num_hidden1']=64*6
topology_params['num_hidden2']=64*6


learning_params['num_steps']=200001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

| Parameter Set | Test accuracy | Test Accuracy Character Level |
|---|---|---|
| 1 | 92.54% | 97.49% |

This results were quite discouraging. Due to that and the long training times even on a Titan X I not further explore this architecture.

#### 4.2.2.6 CNN ARCHITECTURE 6 (54X54 IMAGES), 8 CONV LAYERS, 2 FC LAYERS, ELU INSTEAD OF RELU ACTIVATION

Padding='SAME' for convolutional and pooling layers.

Pooling strides:

- Stride1: [1,1,1,1]
- Stride2: [1,2,2,1]

- CONV1(patchsize1, depth1) -> ELU -> POOL(Stride2)
- CONV2(patchsize1, depth2) -> ELU -> POOL(Stride1) -> DROPOUT
- CONV3(patchsize1, depth1) -> ELU -> POOL(Stride2)
- CONV4(patchsize1, depth2) -> ELU -> POOL(Stride1) -> DROPOUT
- CONV5(patchsize1, depth1) -> ELU -> POOL(Stride2)
- CONV6(patchsize1, depth2) -> ELU -> POOL(Stride1) -> DROPOUT

- CONV7(patchsize1, depth1) -> ELU -> POOL(Stride2)
- CONV8(patchsize1, depth2) -> ELU -> POOL(Stride1)
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Parameter sets and results:

```
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

| Parameter | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 |
|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | 5 |
| patch_size1 | 5 | 5 | 5 | 5 | 5 |
| depth1 | 32 | 32 | 32 | 32 | 32 |
| depth2 | 32 | 32*2 | 32 | 32 | 32 |
| depth3 | 64 | 64*2 | 64 | 64 | 64 |
| depth4 | 64 | 64 | 64 | 64 | 64 |
| depth5 | 64*2 | 64*3 | 64*2 | 64*2 | 64*2 |
| depth6 | 64*2 | 64*3 | 64*2 | 64*2 | 64*2 |
| depth7 | 64*3 | 64*4 | 64*2 | 64*4 | 64*2 |
| depth8 | 64*3 | 64*4 | 64*3 | 64*4 | 64*2 |
| num_hidden1 | 64*6 | 64*8 | 64*8 | 64*12 | 64*16 |
| num_hidden2 | 64*6 | 64*8 | 64*8 | 64*12 | 64*16 |
| num_steps | 200000 | 200000 | 200000 | 200000 | 200000 |
| | | | | | |
| Test accuracy | 93.50% | 93.09% | 93.63% | 93.01% | 93.28% |
| Test accuracy character level | 97.96% | 97.65% | 97.86% | 97.76% | 97.84% |

These results are not bad, but the best CNN architecture for 54x54 image size is CNN architecture 1.

### 4.2.3 CNN Architectures 32x32 image size

#### 4.2.3.1 CNN ARCHITECTURE 1 (32X32 IMAGES), 5 CONV LAYERS, 1 FC LAYERS, RELU BEFORE LRN

This is my first CNN for the SVHN dataset.

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> RELU -> LRN
- CONV2(patchsize1, depth2) -> RELU -> LRN -> POOL
- CONV3(patchsize2, depth3) -> RELU -> LRN
- CONV4(patchsize2, depth4) -> RELU -> LRN -> POOL
- CONV5(patchsize2, num_hidden1) -> RELU -> LRN -> POOL
- FC(num_hidden1) -> DROPOUT
- logits

Parameter sets and results:

```
learning_params['batch_size']=128
```

Optimizer:

1. AdaDelta with exponential decay (0.01, global_step, 20000, 0.95)
2. Adam with exponential decay (1.0E-4, global_step, 20000, 0.95)
3. Adam with constant learning rate 1.0E-4

| Parameter | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 | Set 6 |
|---|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | 5 | 5 |
| patch_size1 | 3 | 5 | 5 | 5 | 5 | 5 |
| depth1 | 16 | 16 | 16*2 | 16*2 | 16*2 | 16*2 |
| depth2 | 32 | 32 | 32*2 | 32*2 | 32*2 | 32*2 |
| depth3 | 64 | 64 | 64*2 | 64*2 | 64*2 | 64*2 |
| depth4 | 128 | 128 | 128*2 | 128*2 | 128*2 | 128*2 |
| depth5 | 256 | 256 | 256*2 | 256*2 | 256*2 | 256*2 |
| num_hidden1 | 64*16 | 64*16 | 64*16 | 64*16 | 64*16 | 64*16 |
| num_steps | 300000 | 300000 | 500000 | 500000 | 200000 | 200000 |

| Optimizer | AdaDelta | AdaDelta | AdaDelta | AdaDelta | Adam with decay | Adam without decay |
|---|---|---|---|---|---|---|
| Data Augmentation | Yes | Yes | Yes | No | Yes | Yes |
| Test accuracy | 91.12% | 92.50% | 93.13% | 90.73% | 92.92% | 92.50% |
| Test accuracy character level | NA | NA | NA | 97.17% | 97.78% | 97.59% |

Here again data augmentation helps a lot to improve the test accuracy – see parameter set 3 and 4 – the difference is about 2.4%! Therefore, as stated above I will always use data augmentation for all further experiments. Furthermore, I will only use Adam-Optimizer since this optimizer works best without playing around with a number of optimizer parameters.

#### 4.2.3.2 CNN ARCHITECTURE 1 (32X32 IMAGES), 5 CONV LAYERS, 2 FC LAYERS, LRN BEFORE RELU, DROPOUT IN 2 AND ALL CONV LAYERS

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> LRN -> RELU (-> DROPOUT)
- CONV2(patchsize1, depth2) -> LRN -> RELU -> POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> LRN -> RELU (-> DROPOUT)
- CONV4(patchsize2, depth4) -> LRN -> RELU -> POOL -> DROPOUT (-> DROPOUT)
- CONV5(patchsize2, num_hidden1) -> LRN -> RELU -> POOL (-> DROPOUT)
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Optimizer:

1. Adam with exponential decay (1.0E-4, global_step, 20000, 0.95)
2. Adam with constant learning rate 1.0E-4

Parameter sets and results:

```
learning_params['batch_size']=128
```

| Parameter | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 | Set 6 |
|---|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | 5 | 5 |

| patch_size1 | 5 | 5 | 5 | 5 | 5 | 5 |
|---|---|---|---|---|---|---|
| depth1 | 16*2 | 16*2 | 16*2 | 16*2 | 16*2 | 16*2 |
| depth2 | 32*2 | 32*2 | 32*2 | 32*2 | 32*4 | 32*4 |
| depth3 | 64*2 | 64*2 | 64*2 | 64*2 | 64*4 | 64*4 |
| depth4 | 64*2 | 64*3 | 64*2 | 64*2 | 64*4 | 64*4 |
| depth5 | 64*2 | 64*3 | 64*2 | 64*2 | 64*4 | 64*4 |
| num_hidden1 | 64*6 | 64*12 | 64*16*3 | 64*8 | 64*12 | 64*12 |
| num_hidden2 | 64*6 | 64*12 | 64*16 | 64*8 | 64*12 | 64*12 |
| num_steps | 300000 | 300000 | 200000 | 300000 | 200000 | 200000 |
| Dropout CONV Layer | Two CONV layers | Two CONV layers | Two CONV layers | All CONV layers | All CONV layers | All CONV layers |
| Dropout parameter | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.6 |
| Optimizer | Adam with decay | Adam with decay | Adam with decay | Adam with decay | Adam with decay | Adam with decay |
| Data Augmentation | Yes | Yes | Yes | Yes | Yes | Yes |
| Test accuracy | 94.28% | 94.09% | 93.92% | 94.28% | 94.71% | 94.84% |
| Test accuracy character level | 98.08% | 98.03% | 98.03% | 98.08% | 98.23% | 98.22% |

| Parameter | Set 7 | Set 8 | Set 9 | Set 10 | **Set 11** | Set 12 |
|---|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | **5** | |
| patch_size1 | 5 | 5 | 5 | 5 | **5** | |
| depth1 | 16*2 | 16*2 | 16*2 | 16*2 | **16*2** | |
| depth2 | 32*4 | 32*4 | 32*4 | 32*4 | **32*4** | |
| depth3 | 64*4 | 64*4 | 64*4 | 64*4 | **64*4** | |
| depth4 | 64*4 | 64*4 | 64*4 | 64*4 | **64*4** | |
| depth5 | 64*4 | 64*4 | 64*4 | 64*4 | **64*4** | |

| num_hidden1 | 64*12 | 64*12 | 64*12 | 64*32 | **64*12** | |
|---|---|---|---|---|---|---|
| num_hidden2 | 64*12 | 64*12 | 64*12 | 64*32 | **64*12** | |
| num_steps | 200000 | 200000 | 200000 | 200000 | **500000** | |
| Dropout CONV Layer | All CONV layers | All CONV layers | All CONV layers | All CONV layers | **All CONV layers** | |
| Dropout parameter | 0.45 | 0.7 | 0.55 | 0.55 | **0.5** | |
| Optimizer | Adam with decay | Adam with decay | Adam with decay | Adam with decay | **Adam with decay** | |
| Data Augmentation | Yes | Yes | Yes | Yes | **Yes** | |
| Test accuracy | 94.42% | 94.62% | 94.71% | 94.26% | **95.46%** | |
| Test accuracy character level | 98.07% | 98.21% | 98.15% | 98.08% | **98.41%** | |

I experimented first with some different topology parameters (set 1 to set 5) and then I varied the dropout parameter for a fixed topology (set 6 to set 9). With set 10 I tried a network with a larger number of hidden neurons but with the same depth values for the CONV layers. This leads to a lower accuracy, so finally I get parameter set 5 a new chance but with much more steps (500,000). This takes about 36h on a Titan X GPU card.

The Figure 11shows the loss and Figure 12 training and validation accuracy of the best CNN model.

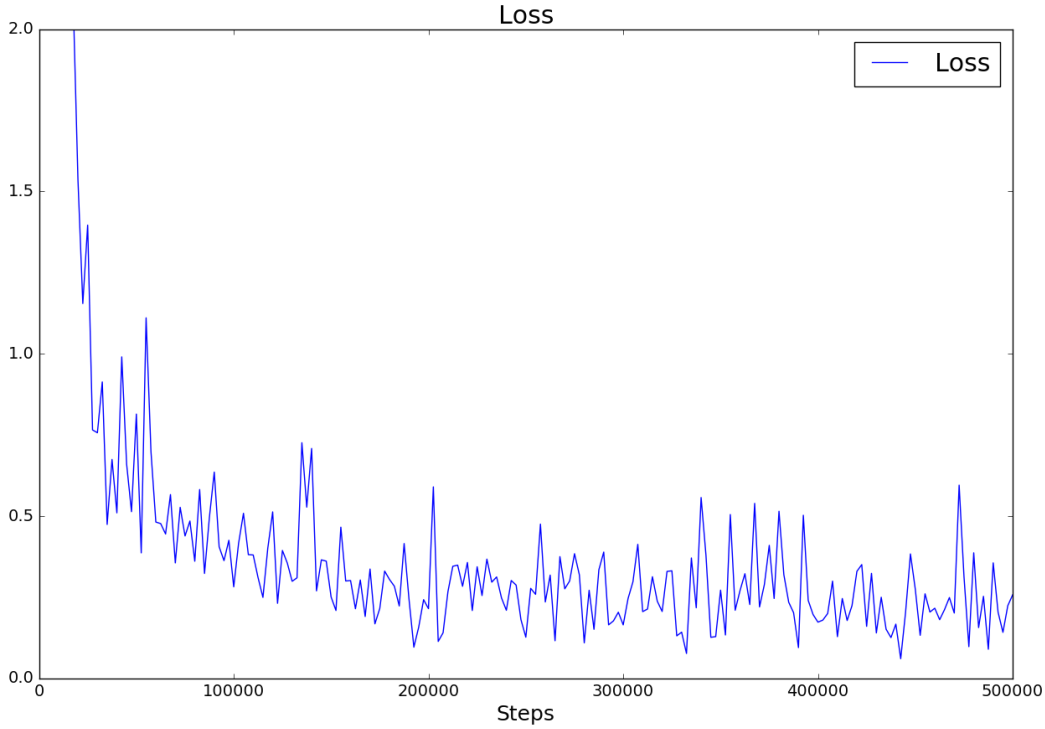



Figure 11: Loss Function of the best CNN Model

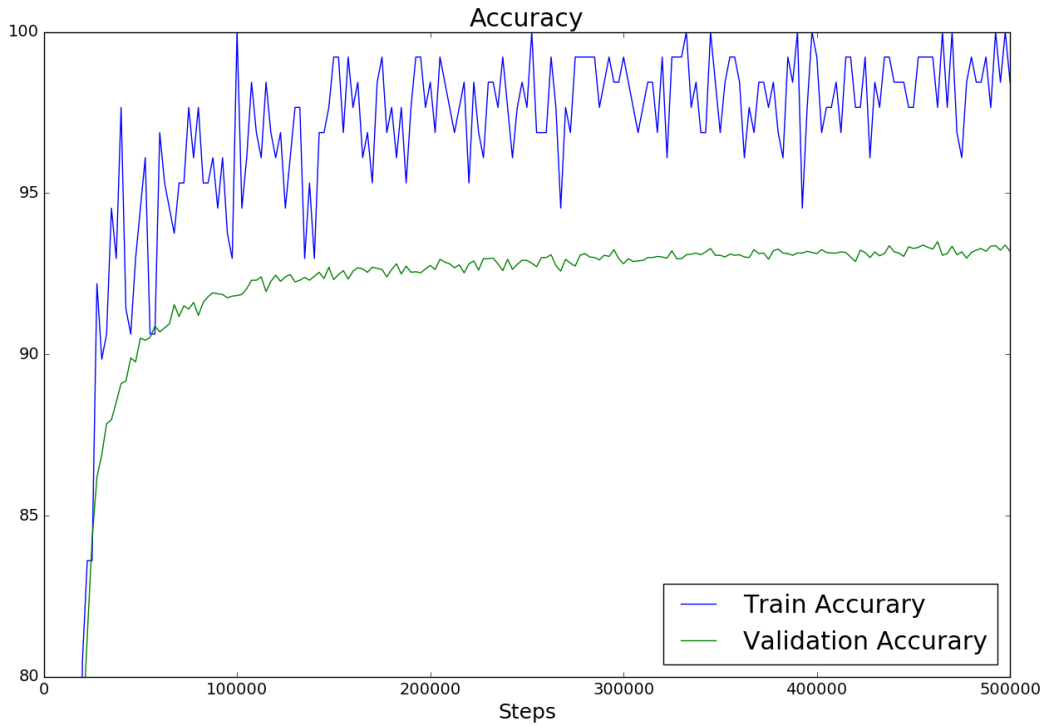



Figure 12: Training and Validation Accuracy of the best CNN Model

The best model shows some overfitting but comparing the final training accuracy of about 98% with the test accuracy of 95.48% indicates some overfitting to, but in my opinion the model generalizes quite well. Interestingly for all

models test accuracy is higher than validation accuracy. In the bench mark paper the authors see a "considerable overfitting" too without providing any numbers.

**This produced the highest accuracy with a test accuracy of 95.46% for the whole numbers and a test accuracy character level of 98.41%! This test accuracy is not far away from the bench mark result of 96.03% for the whole numbers and beats the bench mark results at character lever accuracy of 97.84%.**

#### 4.2.3.3 CNN ARCHITECTURE 1 (32X32 IMAGES), 5 CONV LAYERS, 2 FC LAYERS, ELU INSTEAD OF RELU, DROPOUT IN ALL CONV LAYERS

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> ELU -> DROPOUT
- CONV2(patchsize1, depth2) -> ELU -> POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> ELU -> DROPOUT
- CONV4(patchsize2, depth4) -> ELU -> POOL -> DROPOUT -> DROPOUT
- CONV5(patchsize2, num_hidden1) -> ELU -> POOL -> DROPOUT
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Optimizer:

3. Adam with exponential decay (1.0E-4, global_step, 20000, 0.95)
4. Adam with constant learning rate 1.0E-4

Parameter sets and results:

```
learning_params['batch_size']=128
```

| Parameter | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 | Set 6 |
|---|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | 5 | 5 |
| patch_size1 | 5 | 5 | 5 | 5 | 5 | 5 |
| depth1 | 16*2 | 16*2 | 16*2 | 16*2 | 16*2 | 16*2 |
| depth2 | 32*2 | 32*2 | 32*4 | 32*4 | 32*4 | 32*4 |
| depth3 | 64*2 | 64*2 | 64*4 | 64*4 | 64*4 | 64*4 |
| depth4 | 64*2 | 64*4 | 64*4 | 64*4 | 64*4 | 64*4 |
| depth5 | 64*2 | 64*4 | 64*4 | 64*8 | 64*4 | 64*4 |
| num_hidden1 | 64*6 | 64*12 | 64*12 | 64*8 | 64*12 | 64*12 |

| num_hidden2 | 64*6 | 64*12 | 64*12 | 64*8 | 64*12 | 64*12 |
|---|---|---|---|---|---|---|
| num_steps | 300000 | 300000 | 300000 | 300000 | 100000 | 100000 |
| Dropout parameter | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.6 |
| Optimizer | Adam without decay | Adam without decay | Adam without decay | Adam without decay | Adam without decay | Adam without decay |
| Data Augmentation | Yes | Yes | Yes | Yes | Yes | Yes |
| Test accuracy | 93.50% | 94.19% | 94.68% | 94.18% | 93.77% | 94.25% |
| Test accuracy character level | 97.77% | 98.00% | 98.18% | 98.06% | 97.91% | 98.12% |

| Parameter | Set 7 | Set 8 | Set 9 | Set 10 | Set 11 | Set 12 |
|---|---|---|---|---|---|---|
| patch_size1 | 5 | 5 | 5 | 5 | 5 | |
| patch_size1 | 5 | 5 | 5 | 5 | 5 | |
| depth1 | 16*2 | 16*2 | 16*2 | 16*2 | 16*2 | |
| depth2 | 32*4 | 32*4 | 32*4 | 32*4 | 32*4 | |
| depth3 | 64*4 | 64*4 | 64*4 | 64*4 | 64*4 | |
| depth4 | 64*4 | 64*4 | 64*4 | 64*4 | 64*4 | |
| depth5 | 64*4 | 64*4 | 64*4 | 64*4 | 64*4 | |
| num_hidden1 | 64*12 | 64*12 | 64*12 | 64*12 | 64*12 | |
| num_hidden2 | 64*12 | 64*12 | 64*12 | 64*12 | 64*12 | |
| num_steps | 100000 | 100000 | 200000 | 300000 | 200000 | |
| Dropout parameter | 0.7 | 0.65 | 0.65 | 0.65 | 0.6 | |
| Optimizer | Adam without decay | Adam without decay | Adam without decay | Adam without decay | Adam without decay | |
| Data Augmentation | Yes | Yes | Yes | Yes | Yes | |
| Test accuracy | 94.17% | 94.34% | 94.76% | 94.64% | 94.49% | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Test accuracy character level | 98.02% | 98.10% | 98.22% | 98.20% | 98.14% | |

I experimented first with some different topology parameters (set 1 to set 5) and then I varied the dropout parameter for a fixed topology (set 6 to set 11) and the number of learning steps too. The results are like the CNN with RELU and LRN, but training is much faster since the LRN layers requires much computation. Maybe training the CNN with best results for 500,000 steps with Adam optimizer with decay would result in even higher test accuracy but I run out of time – I spent already about 3 month for playing around with different CNNs and parameter sets.

## 4.3 Results for best CNN Classifier

### 4.3.1 CNN Structure and Parameters

I experimented a lot with different CNNs for 32x32 and 54x54 image sizes and with different parameter settings. I got the highest accuracy with this CNN model for 32x32 image size:

Padding='SAME' for convolutional and pooling layers.

Pooling strides: [1,2,2,1]

5 convolution layers, 2 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> LRN -> RELU (-> DROPOUT)
- CONV2(patchsize1, depth2) -> LRN -> RELU -> POOL -> DROPOUT
- CONV3(patchsize2, depth3) -> LRN -> RELU (-> DROPOUT)
- CONV4(patchsize2, depth4) -> LRN -> RELU -> POOL -> DROPOUT (-> DROPOUT)
- CONV5(patchsize2, num_hidden1) -> LRN -> RELU -> POOL (-> DROPOUT)
- FC(num_hidden1) -> DROPOUT
- FC(num_hidden2) -> DROPOUT
- logits

Optimizer: Adam with exponential decay (1.0E-4, global_step, 20000, 0.95)

Parameters:

```
#dropout 0.5
#adam with decay
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*4
topology_params['depth3']=64*4
topology_params['depth4']=64*4
topology_params['depth5']=64*4

topology_params['num_hidden1']=64*12
```

```
topology_params['num_hidden2']=64*12

learning_params['num_steps']=500001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
```

### 4.3.2 Results

This model produced the highest accuracy with a test accuracy of 95.46% for the whole numbers and a test accuracy character level of 98.41%! This test accuracy is not far away from the bench mark result of 96.03% for the whole numbers and beats the bench mark results at character lever accuracy of 97.84%.

### 4.3.3 Justification

The best model I built so far reaches 95.46%, while the human's performance on this data set reaches 98%. This is not a big difference and I think with some improved data augmentation, artificial generated vertical aligned house numbers and some CNN optimization more than 96% accuracy can be achieved.

## 4.4 Implementation and Refinement of CNN Regression

### 4.4.1 Preprocessing

The images are only normalized by their mean and standard deviation. The following images shows some of the images with labels and bounding boxes:

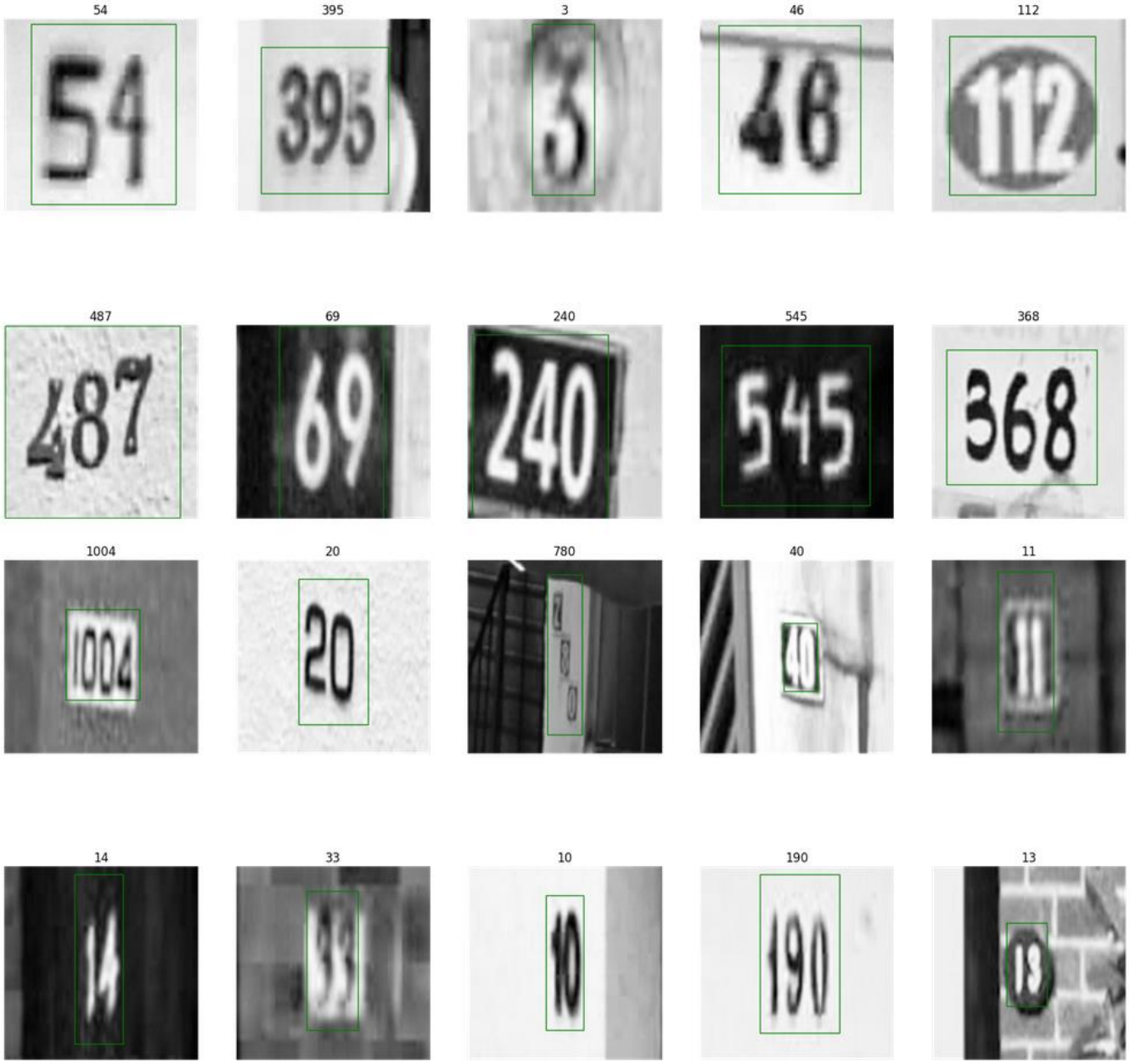



Figure 13: Preproccessed images with labels and Bounding Boxes

### 4.4.2 General CNN parameters and elements

Adam-Optimizer was used for almost all CNN trainings.

Padding='SAME' for convolutional and pooling layers. All pooling layers use max pooling.

Convolution layer strides: [1,1,1,1]

Calculation of accuracy (intersection of union)

```
def bb_intersection_over_union(boxP, box):
    #boxP: predicted Box
    #box: true Box
```

```
    # determine the (x, y)-coordinates of the intersection rectangle
    xA = max(boxP[0], box[0])
    yA = max(boxP[1], box[1])
    xB = min(boxP[2], box[2])
    yB = min(boxP[3], box[3])

    # compute the area of intersection rectangle
    interArea = (xB - xA + 1) * (yB - yA + 1)

    # compute the area of both the prediction and ground-truth
    # rectangles
    boxPArea = (boxP[2] - boxP[0] + 1) * (boxP[3] - boxP[1] + 1)
    boxArea = (box[2] - box[0] + 1) * (box[3] - box[1] + 1)

    # compute the intersection over union by taking the intersection
    # area and dividing it by the sum of prediction + ground-truth
    # areas - the interesection area
    iou = interArea / float(boxPArea + boxArea - interArea)

    # return the intersection over union value
    return iou

#test
examples = [[[39, 63, 203, 112], [54, 66, 198, 114]],\
            [[49, 75, 203, 125], [42, 78, 186, 126]],\
            [[31, 69, 201, 125], [30, 68, 205, 130]],\
            [[31, 69, 201, 125], [31, 69, 201, 125]]]
for i in range(len(examples)):
    print(bb_intersection_over_union(examples[i][0],examples[i][1]))

def accuracy(predBoxes,trueBoxes):
    sum=0.0
    for i in range(len(predBoxes)):
        pbox=np.zeros(4)
        for j in range(4):
            pbox[j]=predBoxes[j,i]

        sum+=bb_intersection_over_union(pbox,trueBoxes[i])
    acc= sum/ len(predBoxes)
 #   print(acc)
    return acc*100.0
print("Success")
```

Calculation of logits:

```
  #logits
  logitsl = tf.matmul(final1, s0_w) + s0_b  #left
  logitst = tf.matmul(final1, s1_w) + s1_b  #top
  logitsr = tf.matmul(final1, s2_w) + s2_b  #right
  logitsb = tf.matmul(final1, s3_w) + s3_b  #bottom
```

Calculation of loss is done by calculating the L2 distance:

```
  loss = tf.sqrt(tf.reduce_sum(tf.square(logitsl-tf_train_boundingboxes[:,0])) + \
                     tf.reduce_sum(tf.square(logitst-tf_train_boundingboxes[:,1]))+ \
                     tf.reduce_sum(tf.square(logitsr-tf_train_boundingboxes[:,2]))+\
                     tf.reduce_sum(tf.square(logitsb-tf_train_boundingboxes[:,3])))
```

Calculation of the predictions:

```
    # Predictions for the training, validation, and test data.
     train_prediction = tf.pack([model(tf_train_dataset, 1.0)[0],\
                                 model(tf_train_dataset, 1.0)[1],\
                                 model(tf_train_dataset, 1.0)[2],\
                                 model(tf_train_dataset, 1.0)[3]])
     valid_prediction = tf.pack([model(tf_valid_dataset, 1.0)[0],\
                                 model(tf_valid_dataset, 1.0)[1],\
                                 model(tf_valid_dataset, 1.0)[2],\
                                 model(tf_valid_dataset, 1.0)[3]])
     test_prediction = tf.pack([model(tf_test_dataset, 1.0)[0],\
                                model(tf_test_dataset, 1.0)[1],\
                                model(tf_test_dataset, 1.0)[2],\
                                model(tf_test_dataset, 1.0)[3]])
```

Helper functions to construct the CNNs:

```
# Helper functions
def create_weight_conv(name,shape):
    return tf.get_variable(name, shape,initializer=tf.contrib.layers.xavier_initializer_conv2d())

def create_weight(name,shape):
    return tf.get_variable(name, shape,initializer=tf.contrib.layers.xavier_initializer())

def create_bias(name,shape):
    return tf.Variable(tf.constant(1.0, shape=shape),name=name)


def create_conv_layer(name,data, weights,padding,conv_stride):
    return tf.nn.conv2d(data, weights, [1, conv_stride, conv_stride, 1], padding)

def create_pool_layer(name,data,pool_window_size,padding,pool_stride):
    return tf.nn.max_pool(data, pool_window_size, pool_stride, padding)

def get_conv_shape(data_shape, weights_shape,padding,conv_stride):
    if padding == 'VALID':
        new_height = int(np.ceil((1.0*(data_shape[1] - weights_shape[0] + 1)) / conv_stride))
        new_width = int(np.ceil((1.0*(data_shape[2] - weights_shape[1] + 1)) / conv_stride))
    else:
        new_height = int(np.ceil(1.0*data_shape[1]) / conv_stride)
        new_width = int(np.ceil(1.0*data_shape[2] / conv_stride))
    return (data_shape[0], new_height, new_width, weights_shape[3])

def get_pool_shape(data_shape,pool_stride):
    new_height = int(np.ceil(1.0*data_shape[1] / pool_stride[1]))
    new_width = int(np.ceil(1.0*data_shape[2] / pool_stride[2]))
    return (data_shape[0], new_height, new_width, data_shape[3])
```

#### 4.2.3.3 CNN REGRESSION ARCHITECTURE 1, 5 CONV LAYERS, 1 FC LAYERS, ELU INSTEAD OF RELU, DROPOUT IN ALL CONV LAYERS

At first I used the CNN with ELU activation (see 4.2.2.6) and I removed one FC layer and replaced the classification output layer with a L2 regression layer (regression head instead classification head).

5 convolution layers, 1 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> ELU -> DROPOUT
- CONV2(patchsize1, depth2) -> ELU -> POOL -> DROPOUT

- CONV3(patchsize2, depth3) -> ELU -> DROPOUT
- CONV4(patchsize2, depth4) -> ELU -> POOL -> DROPOUT -> DROPOUT
- CONV5(patchsize2, num_hidden1) -> ELU -> POOL -> DROPOUT
- FC(num_hidden1) -> DROPOUT
- logits

I tried several parameter combinations:

```
topology_params['patch_size1']=5
topology_params['patch_size2']=5
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2

topology_params['num_hidden1']=16*4


learning_params['num_steps']=50001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-5
#dropout=1.0
#after 2000 iterations not decrease of loss and no increase in validation accuracy!
#Minibatch accuracy: 63.73%
#Validation accuracy: 54.22%
#====> Test accuracy: 28.07%
#dropout=0.5


topology_params['patch_size1']=11
topology_params['patch_size2']=11
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2
topology_params['num_hidden1']=16*4


learning_params['num_steps']=10001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
#Validation accuracy: 51.51%
#====> Test accuracy: 27.93%

topology_params['patch_size1']=15
topology_params['patch_size2']=15
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2
topology_params['num_hidden1']=16*4

learning_params['num_steps']=10001
```

```
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
#Minibatch accuracy: 61.51%
#Validation accuracy: 55.99%
#====> Test accuracy: 29.00%
topology_params['patch_size1']=19
topology_params['patch_size2']=19
topology_params['depth1']=16*2
topology_params['depth2']=32*2
topology_params['depth3']=64*2
topology_params['depth4']=64*2
topology_params['depth5']=64*2
topology_params['num_hidden1']=16*4

learning_params['num_steps']=10001
learning_params['batch_size']=128
#Minibatch accuracy: 59.81%
#Validation accuracy: 56.11%
#====> Test accuracy: 29.25%
```

Regardless of the parameters the train accuracy goes up after some thousand iterations to about 60% to 70% and then not improves anymore. The highest test accuracy is about 29% - very low. The visualization of the predictions shows the reason for that. If the background of the house number has no texture and no other significant pattern, the predicted bounding box is not much different from the true bounding box. In all other cases the house number is located but some other structures as well. This makes it impossible for the network to focus on the house numbers only. Maybe some clever image preprocessing can remove this background structures or reduce it enough to make the location of the house numbers in most cases possible. But within this project it was not possible to explore this.

Figure 14: Predicted (red rectangles) and true (green rectangles) bounding boxes for randomly selected test images.

The training time was quite high due to the high resolution of the images (128x128). Therefore, I tried a much simpler CNN regression model as well.

#### 4.4.2.1 CNN REGRESSION ARCHITECTURE 1, 2 CONV LAYERS, 1 FC LAYERS, ELU INSTEAD OF RELU, DROPOUT IN ALL CONV LAYERS

At first I used the CNN with ELU activation (see 4.2.2.6) and I removed one FC layer and replaced the classification output layer with a L2 regression layer (regression head instead classification head).

5 convolution layers, 1 fully connected layers, logits:

- CONV1(patchsize1, depth1) -> ELU -> DROPOUT
- CONV2(patchsize1, depth2) -> ELU -> POOL -> DROPOUT
- • FC(num_hidden1) -> DROPOUT
- logits

I tried several parameter combinations:

```
topology_params['patch_size1']=11
topology_params['patch_size2']=11
topology_params['depth1']=16*2
topology_params['num_hidden1']=16*4

learning_params['num_steps']=10001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
# conv_stride = 2
#Minibatch accuracy: 61.45%
#Validation accuracy: 56.18%
#====> Test accuracy: 29.29%



topology_params['patch_size1']=21
topology_params['patch_size2']=21
topology_params['depth1']=16*4
topology_params['num_hidden1']=16*8
learning_params['num_steps']=10001
learning_params['batch_size']=128
learning_params['learning_rate']=1.0E-4
# conv_stride = 2
#Validation accuracy: 55.71%
#====> Test accuracy: 29.04%
```

The results are very similar only the training times were considerable shorter. I used the predictions of this model to crop the test images to the predicted bounding boxes and resized them to 32x32 size

### 4.4.3 Cropping test data images to predicted bounding boxes

This is performed with the following code:

```
#crop a single image
im_size=32
from PIL import Image
# Helper functions to convert images from numpy to PIL and back
```

```
def array2image(fa):
    return Image.frombytes("F", (fa.shape[1], fa.shape[0]), fa.tostring())

def image2Array(img):
    ar=np.array(img)*1.0
    return ar

def crop_to_boundbox(npim,box):
     image=array2image(npim).convert("L")
     im_left=box[0]
     im_top=box[1]
     im_right=box[2]
     im_bottom=box[3]
     img = image.crop((im_left, im_top, im_right, im_bottom)).resize([im_size,im_size], Image.ANTIALIAS)
     npimg=image2Array(img)
     return npimg
#process all test images
len_predict=13056
cropped_test_dataset = np.ndarray([len_predict,im_size,im_size,1], dtype='float32')
cropped_test_dataset.shape
print(cropped_test_dataset.shape)

for idx in  range(len_predict):

    im=crop_to_boundbox(test_dataset[idx,:,:,0],predboxes[idx])
    cropped_test_dataset[idx,:,:,0] = im
print("done")
```

The following images shows some examples of the resulting images with their labels:

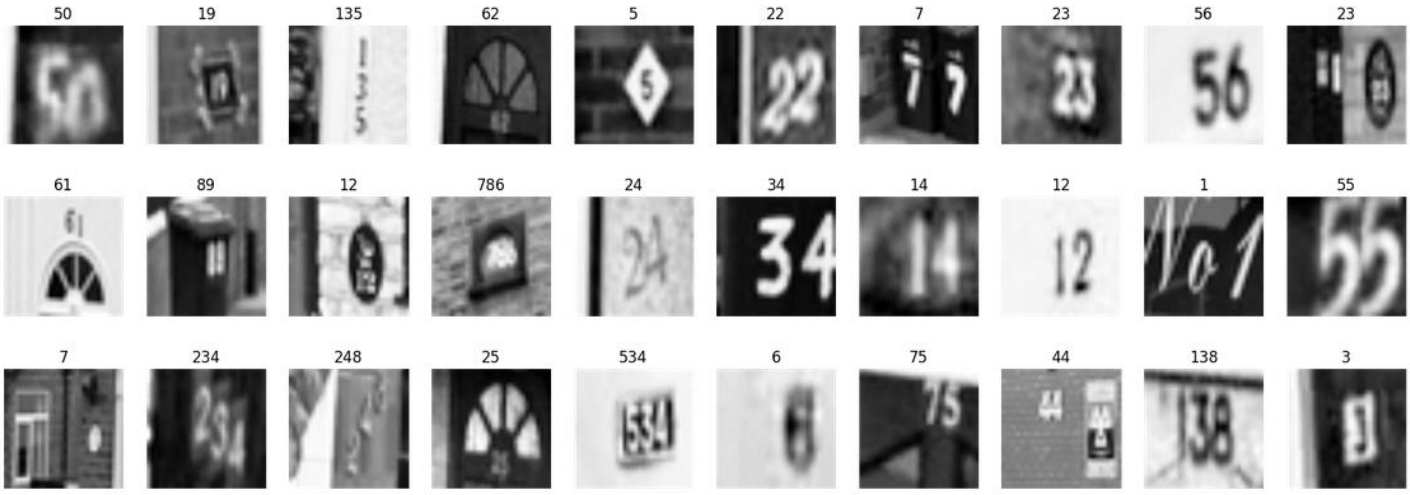



Figure 15: Test Images after Cropping to Predicted Bounding Boxes and resizing

The resulting images were saved with their labels to a hdf5 file.

### 4.4.4 Model Evaluation and Validation

Using the saved best CNN classification model, I loaded the images produced by the regression model and calculated the accuracy of the house number recognition. As expected with this images the test accuracy was much lower: 38.70% for the full house numbers and 69.69% for the character level accuracy. The main reason is, that the images as shown in Figure 15 very often not only contains the house number but many other stuff as well.

### 4.4.5 Justification

I used a CNN regression model to predict the bounding boxes around house numbers. For a background without any texture and other structures it worked quite well, but in all other cases it fails completely. Clearly either some other approach is required or some "clever" image preprocessing approach which removes most of the background textures and all structures which are not digits is required. This is outside the scope of this project.

# • CONCLUSION

## 4.5 Free-Form Visualization best CNN model

Using the saved best CNN model, I applied it to the preprocessed test data set again and selected random samples and displayed them. The test data set is the one were the bounding box was calculated based on the manually generated individual bounding boxes of the original test data set. The following images shows some examples with the true and the predicted labels wrong predictions are marked with a red frame):


Figure 16: Some House Numbers with true and predicted Labels

One of the wrong predictions is truly wrong the two other wrong predictions matches the number in then image but the true label is wrong!

Some more examples (wrong predictions have red titles):


Figure 17: Some more House Numbers with true and predicted Labels

Here are four wrong predictions of 90 numbers. One maybe is wrong labeled (6 versus 0 predicted), three are truly wrong.

Some more examples:


Figure 18: Some more House Numbers with true and predicted Labels

The results are very good, it seems that vertical aligned digits cause problems. Maybe there are not much examples of such house numbers in the datasets. Maybe artificial generated house numbers with vertical aligned digits would help to classify them better.

## 4.6 Free-Form Visualization best CNN model with images from the CNN regression model

Using the saved best CNN classifier model, I applied it to the test data set generated by the CNN regression model and selected random samples and displayed them. The following images shows some examples with the true and the predicted labels wrong predictions are marked with a red frame):


Figure 19: Some more House Numbers with true and predicted Labels, Images cropped by predicted Bounding Boxes

Most predicted labels are wrong but if the background has no structure and other significant structures the predictions are correct in most cases.

## 4.7 Reflection

The experiments demonstrate that CNN classifiers can achieve a high level of accuracy to predict the correct house numbers in most cases. It's worth noting that in this project our approach the models trained fully supervised only, and we use the digits' boundaries in the preprocess stage to remove redundant information. Also, even many digits' boundaries provided by the dataset are transcribed by the AMT workers. To find this bounding boxes without human help is difficult as my simple approach of a CNN regression model shows. With a more advanced approach this shall be possible with high accuracy too.

## 4.8 Improvement

### 4.8.1 Classifier model

The best CNN classifier model I built so far reaches 95.46%, while the human's performance on this data set reaches 98%. This is not a big difference and I think with some improved data augmentation, artificial generated vertical aligned house numbers and some CNN optimization more than 96% accuracy can be achieved.

### 4.8.2 Localization of the house number in an image

Here much improvement is necessary. Optimization of the CNN regression model will not help, maybe some clever image preprocessing steps are necessary or a completely different approach. One possible approach would be to train a CNN classifier by single house numbers with 32x32 images and then use this classifier for the localization of all numbers in an image by using a sliding window with size 32x32 moving over the whole image. For all cases where the classifier predicts with high confidence a number the bounding boxes will be the corresponding 32x32 window. Such an approach was used by Overfeat - Winner of ILSVRC 2013 localization challenge[6].

# 5 REFERENCES

Aaa

[1] http://ufldl.stanford.edu/housenumbers/ 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] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/object_localization_and_detection.html

[3] Sermanet, Pierre, Chintala, Soumith, and LeCun, Yann. Convolutional neural networks applied to house numbers digit classification. In International Conference on Pattern Recognition (ICPR 2012), 2012

[4] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet (2014). Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks https://arxiv.org/pdf/1312.6082

[5] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) . Conference paper at ICLR 2016

[6] Sermanet et al, "Integrated Recognition, Localization and Detection using Convolutional Networks", ICLR 2014

# 6 APPENDIX: SET-UP

Requirements:

- Anaconda Python 2.7
- Python Pillow Image Procession Library https://github.com/python-pillow/Pillow
- Google Tensorflow r0.12
- Fast computer with at least 64GB RAM and a GPU board (at least a GTX1080)

[6] Sermanet et al, "Integrated Recognition, Localization and Detection using Convolutional Networks", ICLR 2014