

# report

January 23, 2017

## 1 Machine Learning Engineer Nanodegree

### 1.1 Capstone Project - Street View House Numbers

Thomas Wieczorek January 21th, 2017

#### 1.2 I. Definition

##### 1.2.1 Project Overview

This capstone-project aims to solve the problem to correctly classify housenumber, extracted from Street View images. A rule-based approach ("Look for a certain color or pattern...") is not very promising, because the house numbers look differently: They come in all colors, different fonts, sometimes they are even one underneath the other.



Example of Google Street View in Berlin [Source](#)



Close up of House Number [Source](#)

The dataset was collected by Stanford University and is described as follows: *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.* (from <http://ufldl.stanford.edu/housenumbers/>)

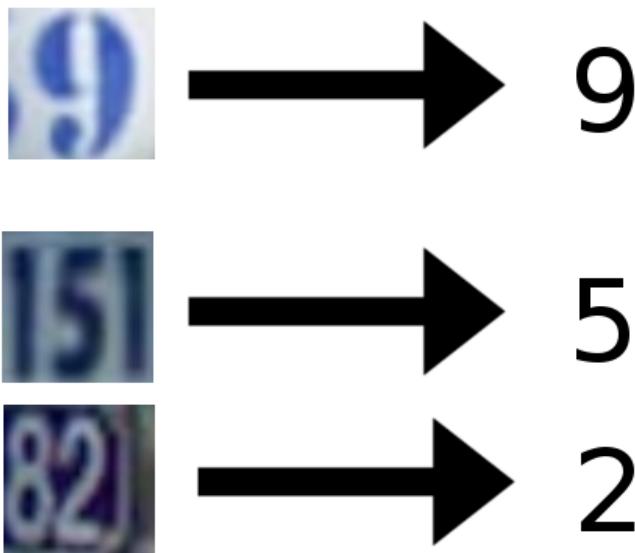
To solve this Computer Vision problem, the domain of Machine Learning is advisable. Especially Deep Learning has been very successful for solving Computer Vision problems like these in the latest years.

### 1.2.2 Problem Statement

The problem statement is to identify house numbers out of images. The images from the SVHN have been resized to a fixed resolution of 32-by-32 pixels. The file type is .mat. In addition to the images, we have the correct labels of every image.

However, if the house number contains more than one digit (for example 123), the images are cropped. It is possible, that several digits are visible in the 32x32 image, the wanted digit is always centered. So the classifier only has to classify one digit.

These images illustrate several examples:



3. Example of correct predictions.



1      1      3      7

4. Example of Cropping a 4-digit house number.

To solve this problem, my strategy is to: - Analyze the data, to get a better understanding - Preprocess the images, for example with [Histogram equalization](#) to optimize very dark and bright images. - Using Machine Learning, we can classify the images. Especially **CNN (Convolutional Neural Networks)** have been very useful for solving image classification. - If the issue of multiple digits will be a problem, cutting out the borders could be a useful approach

### 1.2.3 Metrics

The metrics used for this project were **Accuracy** and the **Confusion Matrix**. The Formula for the **Accuracy** is:

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} \quad (1)$$

(2)

$tp$  : *TruePositive* (3)

$tn$  : *TrueNegative* (4)

$fp$  : *FalsePositive* (5)

$fn$  : *FalseNegative* (6)

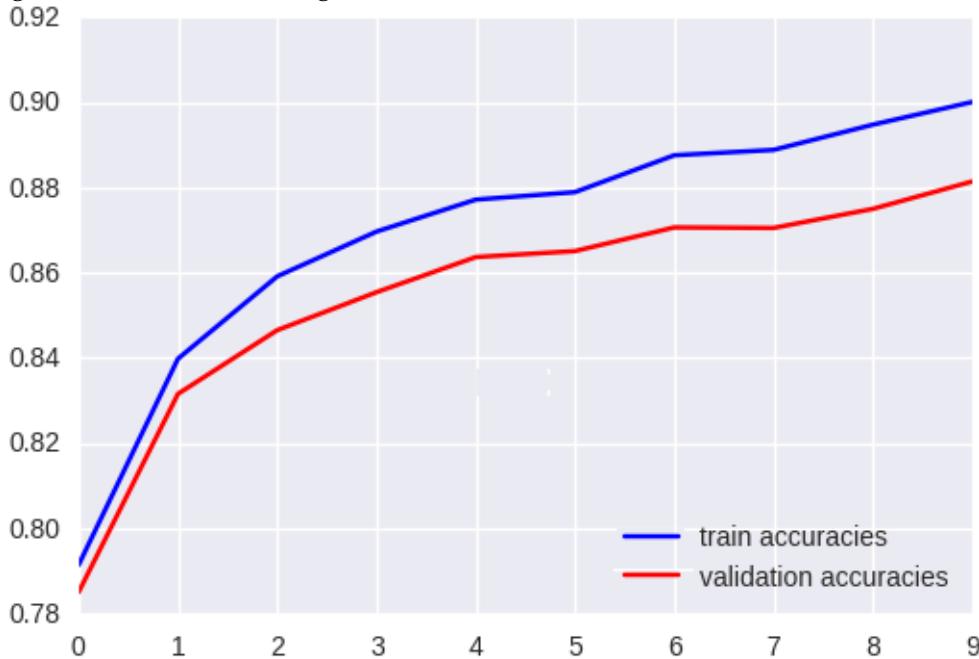
(7)

For classification tasks like this, the accuracy or error-rate(which is *1-accuracy*) are very common metrics, see this [overview of the best classification techniques for SVHN-Dataset](#).

The reason why I also chose this metric, is because it gives an easy but powerful way of calculating, how powerful and reliable the classifier will be. An important thing to analyze for using accuracy: When asking the question “Is this image a zero?”, there is no real difference in severity of errors. False positives are as severe as false negatives. If there is no difference, using accuracy is fine (for example when classifying AIDS-tests, accuracy would be questionable).

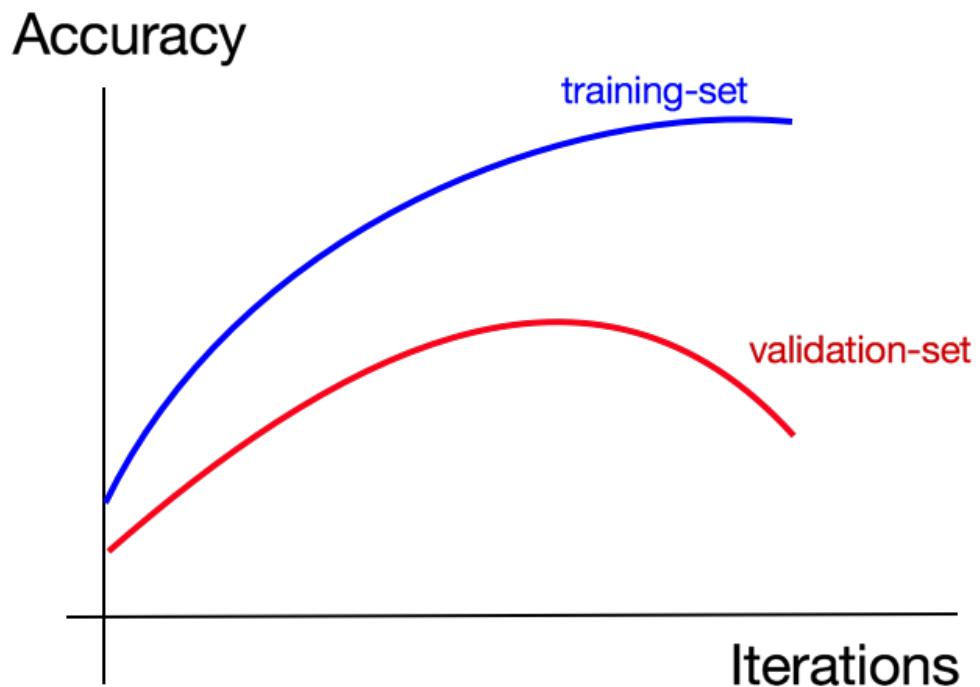
In addition to that, the **Confusion Matrix** is also an interesting metric. It allows to see, which classes are wrongly predicted and provides interesting insights into the system.

For my research, I plotted the Accuracy of the **Training Data**, as well as of the **Validation Data** to prevent [Overfitting](#). Overfitting describes the effect, if the system learns too much and “memorize” the training data. The result ist, that the system does not generalize very well and is not good with different images.



4. Accuracy of Train-Data and Validation-Data.

Overfitting can be detected by comparing the accuracy of the training and validation-set. If the validation line would be declining while the training line still increasing, it is a very good sign to detect Overfitting. The best model is usually, where the validation accuracy peaks. See this figure:



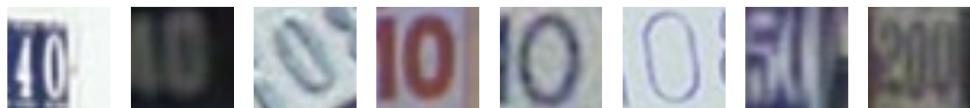
5. Overfitting explanation [Source](#)

## 1.3 II. Analysis

### 1.3.1 Data Exploration

In the following images, 8 random examples of every house number are presented:

0.



1.



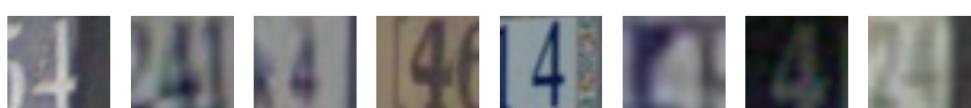
2.



3.



4.



5.



6.



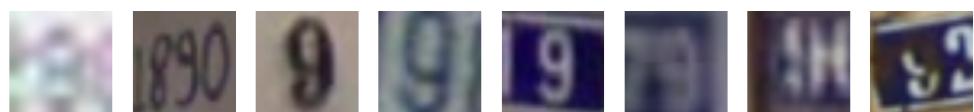
7.



8 .



9 .



Examples of the house numbers

The images of the data-sets have a shape of: (<width>, <height>, <channels>, <batchsize>), for example (32, 32, 3, 73257).

For easier usage in Tensorflow, I changed the shape to (<batchsize>, <width>, <height>, <channels>), for example (73257, 32, 32, 3). In further documentation we will abbreviate the images with **X**

The labels of the data-sets have one-dimensional shape, for example [1,5,7,8,3,6,7...], where every number represents the correct house number of the index.

Dataset	Size (batchsize, width, height, channel)
X_train	(73257, 32, 32, 3)
y_train	(73257,)
X_test	(26032, 32, 32, 3)
y_test	(26032,)

image alt >

Number of training examples = 73257 Number of testing examples = 26032 Image data shape = (32, 32, 3) Number of classes = 10

There were no abnormalities or characteristics found, however some images are very blurry, and it is even hard for humans to identify them.

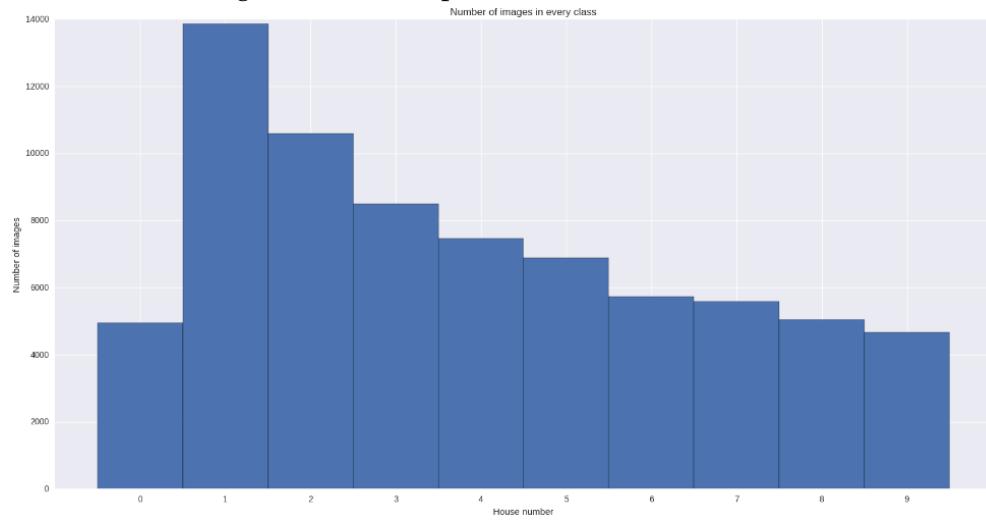


image alt >

Example of Blurry Image

### 1.3.2 Exploratory Visualization

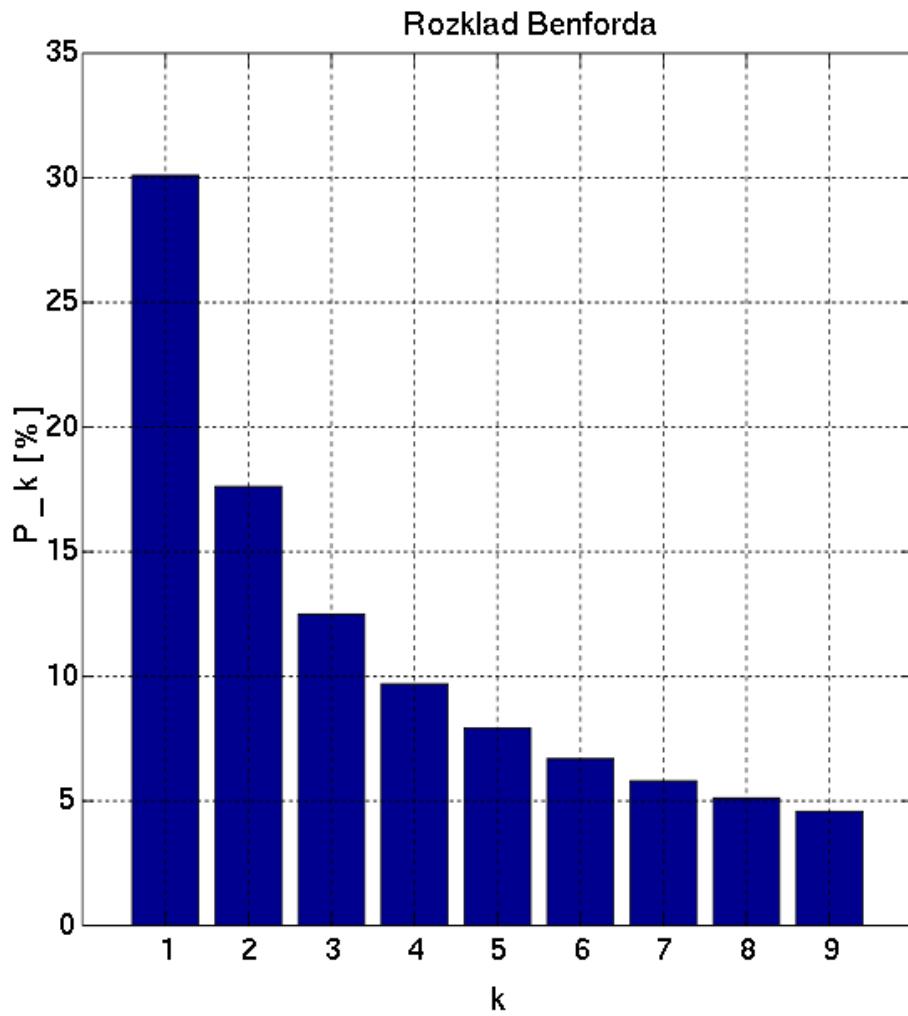
There are 10 classes with the digits from 0-9. Originally, the zeros were labeled as '10'. To remove any irritations, I changed it to '0'. ([https://www.wikiwand.com/en/Benford's\\_law](https://www.wikiwand.com/en/Benford's_law)).



Histogram of the labels

The classes are not distributed equally: The number one is the most common and the quantity is decreasing, the higher the number. The number 0 (originally labelled as 10) is the least common number.

Interestingly, the distribution seems to follow [Benford's Law]([https://en.wikipedia.org/wiki/Benford's\\_law](https://en.wikipedia.org/wiki/Benford's_law)), "also called the **first-digit law**, is an observation about the frequency distribution of leading digits in many real-life sets of numerical data. The law states that in many naturally occurring collections of numbers, the leading significant digit is likely to be small."

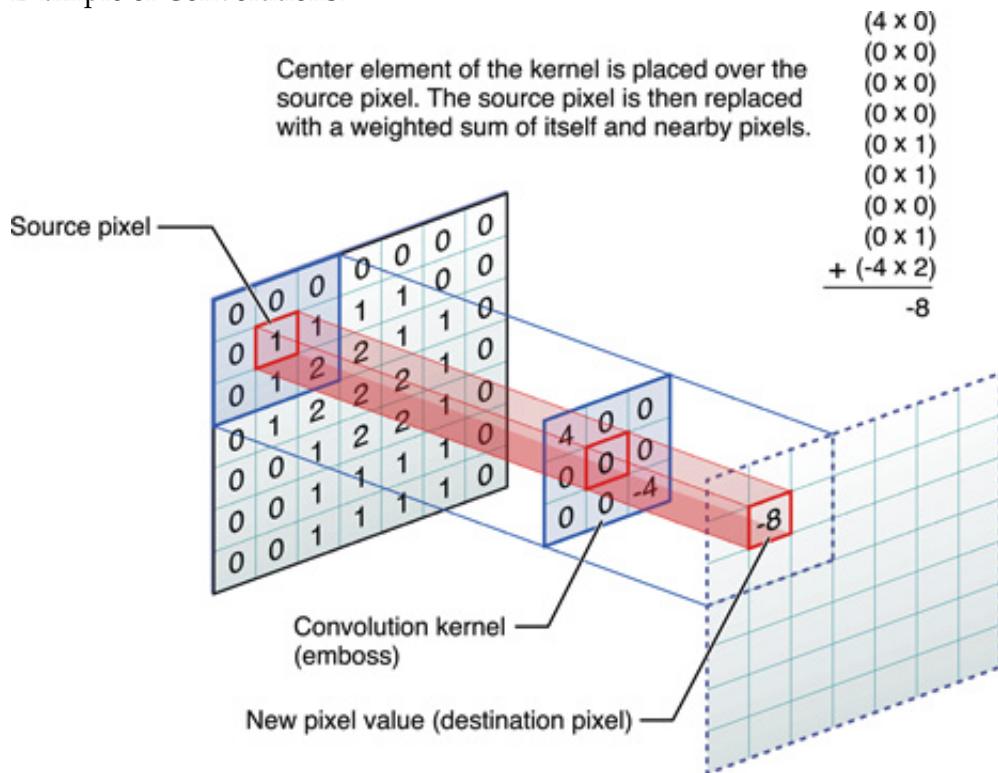


Benford's Law for comparision [Source]([https://en.wikipedia.org/wiki/Benford's\\_law](https://en.wikipedia.org/wiki/Benford%27s_law))

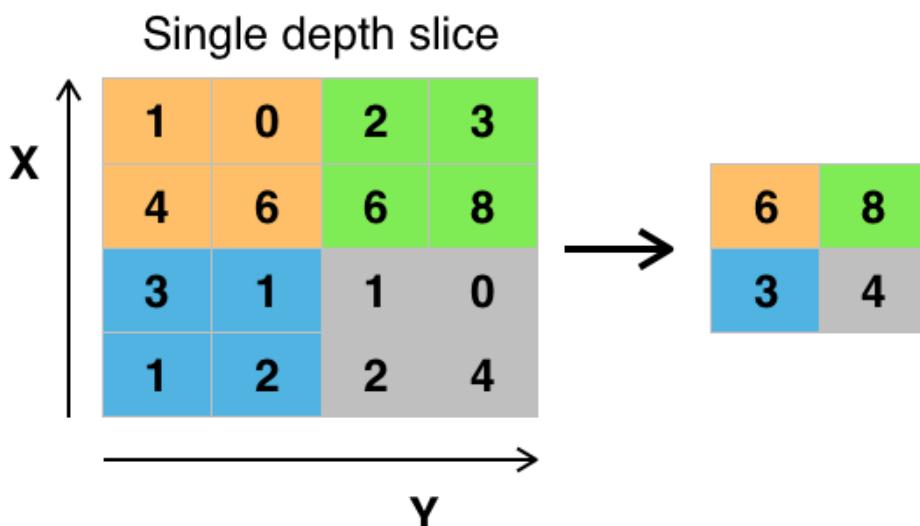
### 1.3.3 Algorithms and Techniques

For solving the classification problem I will use **Deep Learning techniques**, to be more precisely: **Convolutional Neural Networks (CNN)**. CNN's are very common for image recognition, which is the reason why they were used in this project. CNN's are based on different layers of different types. In our architecture, three layer types are used: - Convolutions - Subsampling - Fully connected Layer

Example of Convolutions:



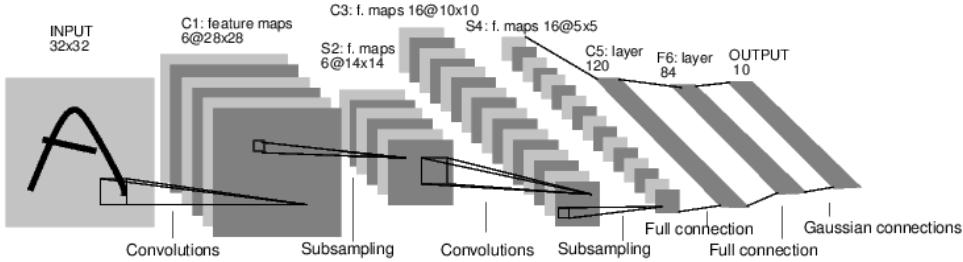
Example of Convolutions [Source](#)



Example of Pooling (it is Max pooling): [Source](#)

Finally, the high-level reasoning in the neural network is done via fully connected layers.

As architecture I will use [LeNET](#), which achieves good results for computer vision classification problems.



#### LeNet Architecture [Source](#)

As Optimizer, I will use the [AdamOptimizer](#). The benefit of AdamOptimizer is, that it controls the learning rate itself, which improves in speed and finding an optimum [See Paper](#).

To reduce Overfitting, I will use [Dropout](#). The idea behind Dropout is interesting: If you deactivate some neurons during the learning process, other neurons have to compensate for the missing neurons. This results in a system, which is very stable.

**Default Parameter:** - Normalizing = Off - AdamOptimizer with Learning Rate = 0.001 - EPOCHS = 10 - BATCH\_SIZE = 50 - Dropout Rate = 0.5

#### 1.3.4 Benchmark

As described, I will use **Accuracy** of the test set as my metric. For my benchmark I am using the original paper from Stanford/Google by Andrew Ng et al

ALGORITHM	SVHN-TEST (ACCURACY)
HOG	85.0%
BINARY FEATURES (WDCH)	63.3%
K-MEANS	90.6%
STACKED SPARSE AUTO-ENCODERS	89.7%
HUMAN PERFORMANCE	98.0%

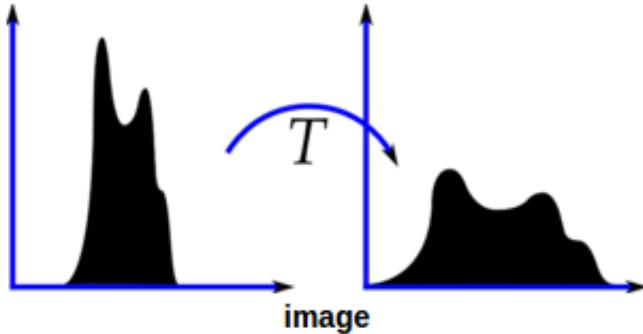
Benchmark from original paper from Stanford/Google by Andrew Ng et al

## 1.4 III. Methodology

### 1.4.1 Data Preprocessing

For preprocessing, I used [Histogram equalization](#).

Consider an image whose pixel values are confined to some specific range of values only. For eg, brighter image will have all pixels confined to high values. But a good image will have pixels from all regions of the image. So you need to stretch this histogram to either ends (as given in below image, from wikipedia) and that is what Histogram Equalization does (in simple words). This normally improves the contrast of the image. (from [http://docs.opencv.org/3.1.0/d5/daf/tutorial\\_py\\_histogram\\_equalization.html](http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html))



Example of Blurry Image Source

In addition to that, I used [One Hot Encoding](#) for the labels. For every possible house number, there will be one additional column with a boolean value.

The number 0 is represented as: [1,0,0,0,0,0,0,0,0]

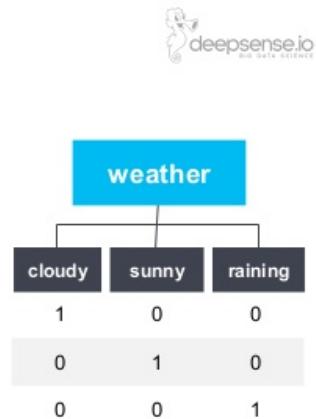
The number 3 is represented as: [0,0,0,1,0,0,0,0,0]

The number 9 is represented as: [0,0,0,0,0,0,0,0,1]

The following picture explains One-hot encoding with a different setting:

## One-hot encoding

day	temperature [F]	weather	bike rentals
3	76	Cloudy	543
4	72	Raining	173
5	78	Sunny	674
6	68	Raining	124



Example of Blurry Image Source

As another way to prevent Overfitting, I splitted the training set into 2 seperate sets: Training and Validation set.

---

<b>Dataset</b>	<b>Size (&lt;batchsize&gt;, &lt;width&gt;, &lt;height&gt;, &lt;channel&gt;)</b>
Train size:	(49082, 32, 32, 3)
Validation size:	(24175, 32, 32, 3)
Test size:	(26032, 32, 32, 3)

---

## 1.4.2 Implementation

The whole implementation can be found [here](#).

The implementation was done in Python with jupyter notebook. The most important frameworks were - cv2 - scipy - sklearn - tensorflow - and several more

The **process** for which metrics, algorithms, and techniques were implemented are inspired by the [Cross Industry Standard Process for Data Mining \(CRISP-DM\)](#).



Process diagram showing the relationship between the different phases of CRISP-DM [Source](#)

First, the “**Business Understanding**” was established (Chapter Business Overview, Problem Statement). What is the problem? How are the data collected?

The **Data Understanding** was done by looking at samples of images and analyzing the proportions of the data. The first two steps were done by turns

The step Data Understanding revealed, that there are some dark and light images. To simplify the classification, in the step **Data Preparation** the data was normalized by using Histogram equalization

Using Deep Learning, the classifier was build in the step **Modeling**. First, the LeNet-Architecture was implemented and using TensorFlow, the model was trained. Using the Validation Data the model was evaluated. However, the results were only mediocre. To improve the system, different parameters were implemented by using trial and error. The Training Accuracy was notably improved, still the gap between Training Accuracy and Validation Accuracy was significant. I interpreted this as Overfitting. To improve the system I implemented Overfitting

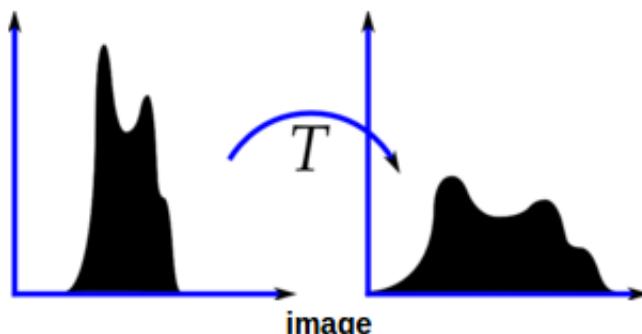
After final enhancements, I applied the Test Set on the model in the step **Evaluation**

The most important parts and algorithms of the implementations are the following:

## 1.4.3 Normalizing, using Histogram equalization

```
In [ ]: def normalize_YUV(img):
    yuv = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    yuv[:, :, 0] = cv2.equalizeHist(yuv[:, :, 0])
    return cv2.cvtColor(yuv, cv2.COLOR_YUV2RGB)
```

Using [Histogram equalization](#) to improve the preprocessing.



Example of Blurry Image [Source](#)

#### 1.4.4 LeNet architecture, including dropout

In [ ]: `def LeNet(x, keep_prob):`

```

# 28x28x6
conv1_W = tf.Variable(tf.truncated_normal([5, 5, 3, 6], stddev = 0.01))
conv1_b = tf.Variable(tf.zeros(6))
conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID')

conv1 = tf.nn.relu(conv1)

# 14x14x6
conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],

# 10x10x16
conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16),stddev =
conv2_b = tf.Variable(tf.zeros(16))
conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VAL

conv2 = tf.nn.relu(conv2)

# 5x5x16
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],

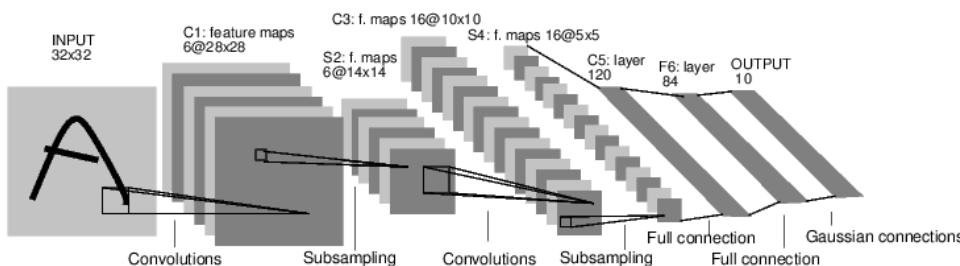
# Flatten
fc1 = flatten(conv2)
# (5 * 5 * 16, 120)

fc1_W = tf.Variable(tf.truncated_normal([400,120],stddev = 0.01))
fc1_b = tf.Variable(tf.zeros(120))
fc1 = tf.matmul(fc1, fc1_W) + fc1_b
fc1 = tf.nn.relu(fc1)

#Dropout
fc1 = tf.nn.dropout(fc1, keep_prob)

fc2_W = tf.Variable(tf.truncated_normal(shape=(120, n_classes), stddev
fc2_b = tf.Variable(tf.zeros(n_classes))
return tf.matmul(fc1, fc2_W) + fc2_b

```



LeNet Architecture [Source](#)

#### 1.4.5 Using Tensorflow for Training:

```
In [ ]: if __name__ == '__main__':
    if LEARN_MODUS:
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            steps_per_epoch = len(X_train) // BATCH_SIZE
            num_examples = steps_per_epoch * BATCH_SIZE

            # Train model
            for i in range(EPOCHS):
                for step in range(steps_per_epoch):
                    #Calculate next Batch
                    batch_start = step * BATCH_SIZE
                    batch_end = (step + 1) * BATCH_SIZE
                    batch_x = X_train[batch_start:batch_end]
                    batch_y = y_train[batch_start:batch_end]

                    #Run Training
                    loss = sess.run(train_op, feed_dict={x: batch_x, y: batch_y})

                    #Calculate Training Loss and Accuracy
                    train_loss, train_acc = eval_data(X_train, y_train)
                    print("EPOCH {} ...".format(i+1))
                    print("Training loss = {:.3f}".format(train_loss))
                    print("Training accuracy = {:.3f}".format(train_acc))
                    train_losses.append(train_loss)
                    train_accuracies.append(train_acc)

                    #Calculate Validation Loss and Accuracy
                    val_loss, val_acc = eval_data(X_val, y_val)
                    print("EPOCH {} ...".format(i+1))
                    print("Validation loss = {:.3f}".format(val_loss))
                    print("Validation accuracy = {:.3f}".format(val_acc))
                    val_losses.append(val_loss)
                    val_accuracies.append(val_acc)

                    #Calculate Test Loss and Accuracy (Should only be done once)
                    test_loss, test_acc = eval_data(X_test, y_test)
                    print("EPOCH {} ...".format(i+1))
                    print("Test loss = {:.3f}".format(test_loss))
                    print("Test accuracy = {:.3f}".format(test_acc))
                    test_losses.append(test_loss)
                    test_accuracies.append(test_acc)

    try:
        saver
    except NameError:
        saver = tf.train.Saver()
```

```
saver.save(sess, 'foo')
print("Model saved")
```

One difficulty that occurred during coding were several Out of Memory Errors, which are very cryptic in TensorFlow and hard to debug. I implemented batch learning and batch validation, which improved the stability of the system.

### 1.4.6 Refinement

As already described, the initial solution had the parameter:

- Normalizing = Off
- AdamOptimizer with Learning Rate = 0.001
- EPOCHS = 10
- BATCH\_SIZE = 50
- Dropout Rate = 0.5

The result on Training accuracy is: 0.890 The result on Validation accuracy is: 0.862

The process to further refine the system and the parameter is **trial and error**. I changed the parameters and looked if the accuracy of the validation set improved.

#### 1) EPOCHS

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
10	0.001	50	0.5	ON	0.857
25	0.001	50	0.5	ON	0.884
50	0.001	50	0.5	ON	0.890
75	0.001	50	0.5	ON	0.908
100	0.001	50	0.5	ON	0.904

**Result: Overfitting is visible above 75 epochs.**

#### 2,3) Learning Rate

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
10	<b>0.0005</b>	50	0.5	ON	0.865
25	<b>0.0005</b>	50	0.5	ON	0.857
10	<b>0.0002</b>	50	0.5	ON	0.887
25	<b>0.0002</b>	50	0.5	ON	0.919

**Result: Learning Rate: The lower, the better.**

#### 4,5) Batch\_Size

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
10	0.0002	<b>25</b>	0.5	ON	0.913
25	0.0002	<b>25</b>	0.5	ON	0.951
10	0.0002	<b>75</b>	0.5	ON	0.904
25	0.0002	<b>75</b>	0.5	ON	0.934

**Result: Batch Size: The lower, the better.**

**6,7) Dropout Rate**

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
10	0.0002	25	<b>0.25</b>	ON	0.850
25	0.0002	25	<b>0.25</b>	ON	0.867
10	0.0002	25	<b>0.5</b>	ON	0.913
25	0.0002	25	<b>0.5</b>	ON	0.951
10	0.0002	25	<b>0.75</b>	ON	0.922
25	0.0002	25	<b>0.75</b>	ON	0.960

**Result: Dropout 0.5 was best.**

**8) Normalisation**

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
10	0.0002	25	0.5	<b>OFF</b>	0.886
25	0.0002	25	0.5	<b>OFF</b>	0.924
10	0.0002	25	0.5	<b>ON</b>	0.913
25	0.0002	25	0.5	<b>ON</b>	0.951

**Result: Without Normalisation is worse.**

The final parameters are:

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
25	0.0002	25	0.5	<b>ON</b>	0.951

## 1.5 IV. Results

### 1.5.1 Model Evaluation and Validation

\*\* Final Parameter\*\*

EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
25	0.0002	25	0.5	ON	0.951

The final parameter were found by trial and error. Another (more structured way) to find the best parameter is [grid search](#). However, grid search takes a lot of time and computer power, that is why it was not used for this. For the same reason, I only used EPOCHS=25. In addition to faster computation, I reduce overfitting by limiting EPOCHS to 25.

#### Test-Set

To evaluate the model, I used the test-set for the first time. The precision for the test set is: **0.901**

#### Further Model Evaluation and Validation

To get a better estimation for the test accuracy, the evaluation was run **five** times with the same parameters.

testrun	EPOCHS	Learning_Rate	BATCH_SIZE	Dropout Rate	Normalisation	Training Acc
#1	25	0.0002	25	0.5	ON	0.951
#2	25	0.0002	25	0.5	ON	0.926
#3	25	0.0002	25	0.5	ON	0.944
#4	25	0.0002	25	0.5	ON	0.944
#5	25	0.0002	25	0.5	ON	0.944
<b>Average</b>	25	0.0002	25	0.5	ON	0.942

The notebooks with the results were saved in the capstone.ipynb to capstone5.ipynb. The first testrun was better than all the others. One reason for this could be simply due to luck, while testing a lot of different parameters. This bias is called [Survivorship Bias](#). A way to improve the validation and the statistical significance is to use [cross-validation](#). However, this takes a lot of time and computer power, which is why it is not used here.

**Second run:** EPOCH 25 ... Training loss = 0.247 Training accuracy = 0.926 EPOCH 25 ... Validation loss = 0.379 Validation accuracy = 0.890 EPOCH 25 ... Test loss = 0.573 Test accuracy = 0.835

**Third run:** EPOCH 25 ... Training loss = 0.186 Training accuracy = 0.944 EPOCH 25 ... Validation loss = 0.342 Validation accuracy = 0.901 EPOCH 25 ... Test loss = 0.545 Test accuracy = 0.854

**Forth run:** EPOCH 25 ... Training loss = 0.195 Training accuracy = 0.944 EPOCH 25 ... Validation loss = 0.347 Validation accuracy = 0.904 EPOCH 25 ... Test loss = 0.557 Test accuracy = 0.848

**Fifth run:** EPOCH 25 ... Training loss = 0.191 Training accuracy = 0.944 EPOCH 25 ... Validation loss = 0.343 Validation accuracy = 0.905 EPOCH 25 ... Test loss = 0.546 Test accuracy =

0.849

**Sixth run:** EPOCH 25 ... Training loss = 0.235 Training accuracy = 0.932 EPOCH 25 ... Validation loss = 0.371 Validation accuracy = 0.894 EPOCH 25 ... Test loss = 0.585 Test accuracy = 0.834

### 1.5.2 Justification

ALGORITHM	SVHN-TEST (ACCURACY)
HOG	85.0%
BINARY FEATURES (WDCH)	63.3%
K-MEANS	90.6%
STACKED SPARSE AUTO-ENCODERS	89.7%
HUMAN PERFORMANCE	98.0%

Benchmark from [original paper from Stanford/Google by Andrew Ng et al](#)

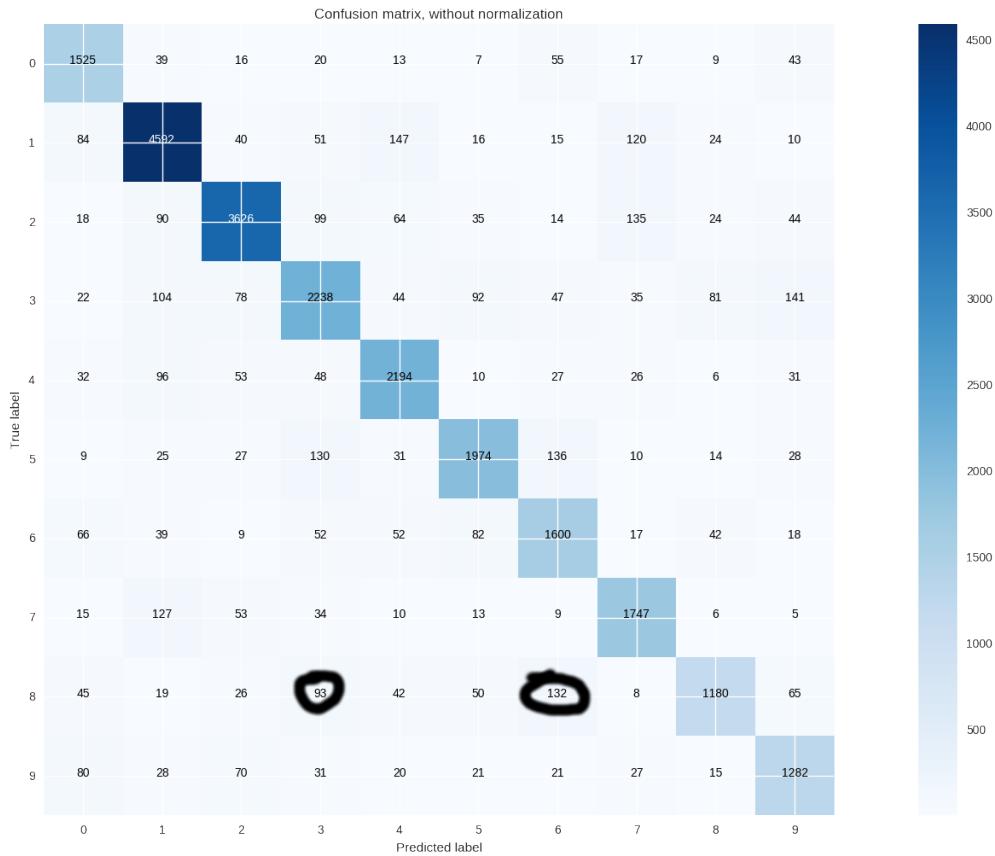
With an accuracy of 85.5%, the results are stronger than the algorithms "Binary Features (WDCH) and "HOG", however it is worse than K-Means and Stacked Sparse Auto-Encoders. It is well below Human Performance.

## 1.6 V. Conclusion

### 1.6.1 Free-Form Visualization

To get a better understanding of the model, it is interesting how the model would perform on singular classes. Are there any numbers, which are mixed up regularly?

One way to get a better understanding of the performance is a **Confusion Matrix**. This helps to identify problems with certain classes (for example the number 3 always get mixed up with 8, because of the similarity).



### 6. Confusion Matrix

The Confusion Matrix gives a detailed overview about misclassified house numbers. House numbers with the true label '8' are most often confused with the numbers 6 and 3. The reason is, because these numbers look similar, only a few strokes different.

Another fact that can be derived from this chart, is that most of the time, the results are correct. This can be seen in the diagonal line, where "True label" is the same as "Predicted label".

## 1.6.2 Reflection

First, to get a basic understanding of the problem, the data was thoroughly **analyzed**: samples of the house-number pictures, the distribution, the labels. After this similar tasks and papers were examined, to get a benchmark and additional knowledge.

After this, the data was **preprocessed**: the label 10 was changed to 0, because pictures of this labels displayed a 0. In addition to this, the labels were one-hot encoded. Then the pictures were optimized for machine learning, using histogram equalization. Finally, the data was splitted into Train-, Test- and Validationset

As a classifying algorithm, **Convolutional Neural Network** (Deep Learning) based on Tensorflow was used. As architecture **LeNET** was implemented, which achieves good results for computer vision classification problems and does not take too much time to learn.

To optimize the parameters, I used trial and error on the validation set and resulted in a system with an accuracy of **85.5 on the testset**.

One interesting aspect was the analysis of mistakes: Similar to humans, the model sometimes confused "similar numbers", like 3, 8 or 6. A difficult aspect (and very time consuming) is fine tuning the parameters.

In summary, the model fit my expectation for the problem and this approach is recommended for these types of problems.

## 1.6.3 Improvement

There are a lot of ways, to improve the described system:

- Generating more data could be a good way to improve the accuracy. One way is to increase the dataset by rotating, stretching or distorting the images. By this way, the imbalance of the occurrence of the house numbers could be fixed.
- Using grid search and cross-validation for improving the parameters
- A more complex CNN-architecture probably would improve the accuracy. Examples would be [AlexNet](#) or [GoogleNet](#).

One technique that would be very interesting is the detection of the position of house numbers in high-resolution images. I am currently enrolled in the (Self Driving Car-Nanodegree)[<https://de.udacity.com/course/self-driving-car-engineer-nanodegree-nd013/>], where these techniques will be taught. By that, one could detect and classify house numbers in random Google Streetview images.

The problem is often used for Machine Learning papers and there are a lot of systems, which outperform my system. Using the above mentioned AlexNet and GoogleNet, the highest result in the SHVN-dataset was achieved by (Chen-Yu Lee, Patrick W. Gallagher and Zhuowen Tu)[<https://arxiv.org/pdf/1509.08985v2.pdf>] with an accuracy of **98.31**.