

Capstone project: Machine learning engineer nanodegree

Digit sequence recognition

I. Definition

I.1. Introduction

Deep learning is a hot topic nowadays. Almost every day we hear about new deep learning algorithms achievements in different fields like games (poker, chess, Go), cancer research, translation.

Though the history of the neural networks goes back to 1950s and many of key breakthrough occurred in the 1990s, it has just recently gained its popularity because of combination of computational power and huge datasets that became available to us not that long time ago.

Deep learning has always fascinated me. It is the whole new world with its own rules that requires to think differently about known problems. I decided to start getting familiar with deep learning algorithm with image recognition problem on the Street View House Numbers Dataset (<http://ufldl.stanford.edu/housenumbers/>).

I.2. Problem statement

SVHN dataset contains real-world images of house numbers (essentially, sequences of digits). The dataset consists of training dataset, test dataset and extra dataset with RGB pictures of size 64x64. For each picture labels and the position of each digit (bounding boxes) are known.

The objective is to recognize the house numbers on the pictures (having the bounding boxes) as good as possible (with the highest accuracy). In order to do that I will be training a Convolutional Neural Network and implementing it using Tensorflow.

The project consists of multiple steps:

1. Get familiar with the theoretical foundations of the Convolutional Neural Networks and implement a simple convolutional network on a single-digit MNIST dataset.
2. Create a sequence of MNIST digits and implement a convolutional network on a multi-digit MNIST dataset
3. Download, analyse and modify the SVHN dataset
4. Implement a convolutional neural network on a multi-digit SVHN dataset
5. Make predictions for new house numbers

I.3. Metrics

Goodfellow et al (2014) use accuracy as the metrics for model performance. They define an input image to be predicted correctly when each element in the sequence is predicted correctly. In other words, there is no "partial credit" for getting individual digits of the sequence correct.

I will be using the same definition of accuracy in the project. Dummy code for defining accuracy:

```
def accuracy(prediction,true_labels):  
    return (np.sum([np.min for a in prediction == true_labels])/len(prediction)
```

Another metrics we could consider would be accuracy based on number correctly predicted digits. For instance, if we have two house numbers 90 and 567 and our model predicts 80 and 561 instead, we would say that we predicted 3 out of 5 digits correctly, therefore have accuracy of 60%.

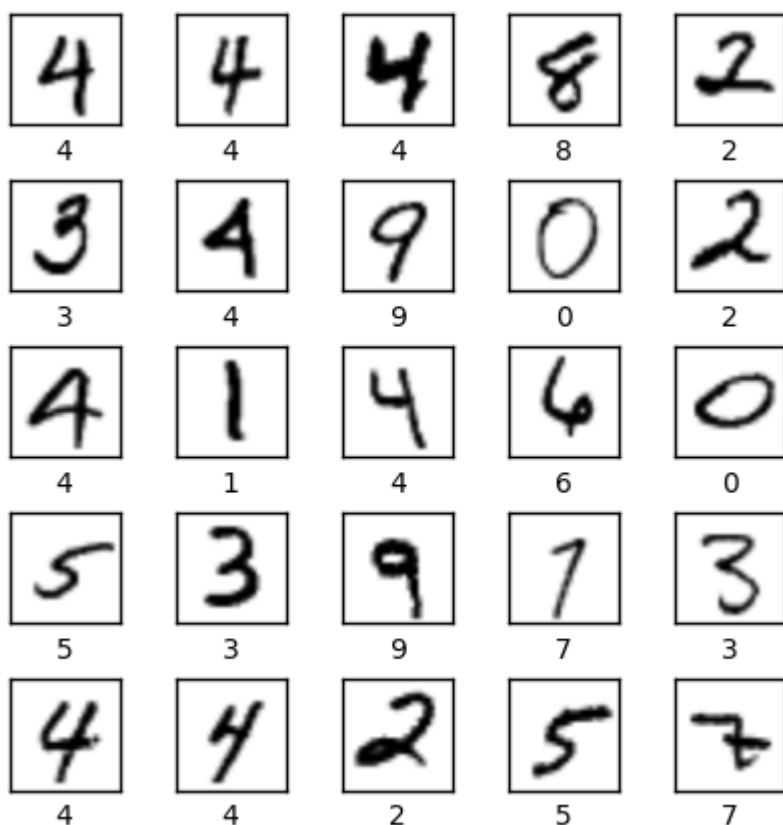
However, this would not be a logical approach. The model can be used for automatic map generation, and if at least 1 digit in a number is wrong, it would cause a lot of misunderstanding. While using accuracy metrics defined above, we say that house number is accurate only if all digits in the sequence are predicted correctly. In the example above we would therefore get accuracy of 0%, what is completely justified.

II. Analysis

II.1. Data exploration

In this project two different datasets were used: MNIST dataset (which was loaded directly from tensorflow) and SVHN dataset (from <http://ufldl.stanford.edu/housenumbers/>).

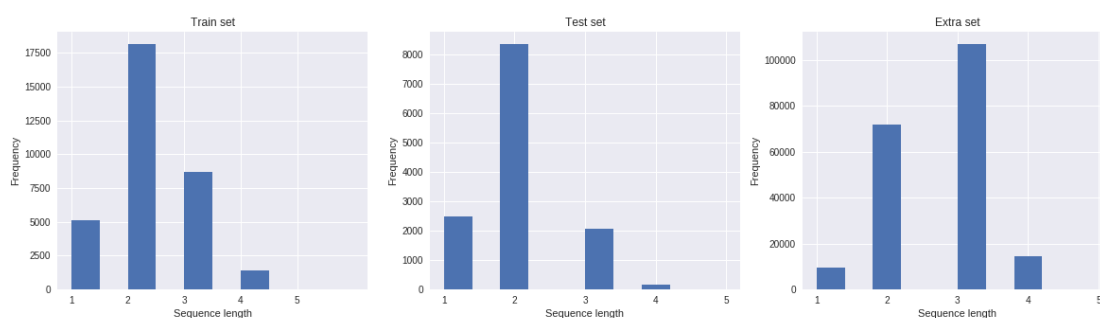
MNIST dataset available in Tensorflow contains 55000 training examples, 10000 test examples and 5000 validation examples that contain images of size 28x28. The data is very clean (all images have the same size, digits are very clear, have approximately the same size and are not rotated) and easy to start with for a novice. You will not find such a clean dataset in real life. Here are some examples of the data from the test set with labels:



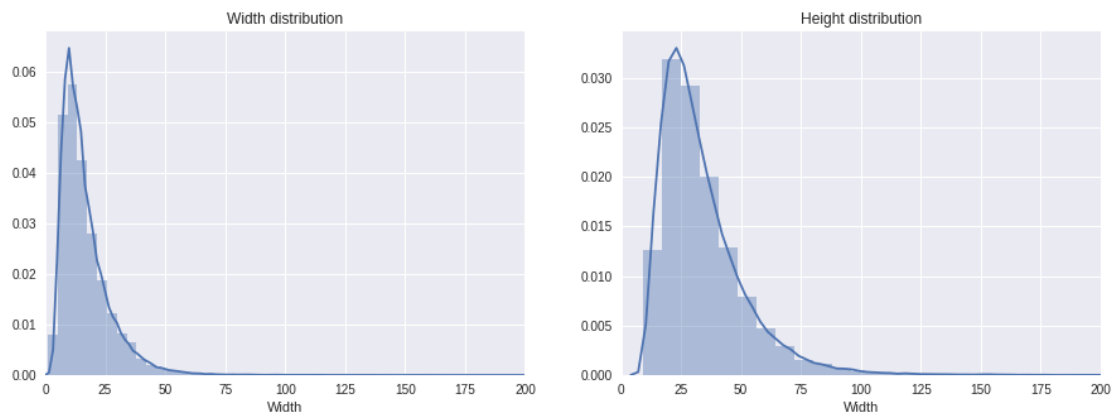
SVHN dataset, on the other hand, is a real life dataset, which is more complex. First of all, it is a multi-digit dataset. Predicting sequence of digit is a more difficult task. Secondly, pictures have different shapes, digits are rotated, not always clear and have different sizes. Also, the dataset has 3 dimensions (RGB). Here are some examples of the data with labels and bounding boxes:



The following plot shows the distribution of house number length for training, test and extra dataset. We can see that most house numbers in test and training set have length of 2. Distribution for extra set looks a bit different and has mode of 3. Note: SVHN dataset contains a small number of house numbers longer than 5 digits, these images were excluded from the dataset.



The following plot shows the distribution of digit widths and heights for the training set:



II.2. Algorithms and techniques

II.2.1 General concepts

Convolutional neural network (CNN) is a typical algorithm to use for an image recognition problem. In order to start working with CNNs, it is important to have basic understanding of the following concepts:

- Convolution
- Activation function
- Pooling
- Backpropagation

Brandon Rohrer in [his blog](#) gives great explanation of how basic convolutional neural networks work. As an example he takes a task to determine whether an image is of an X or an O. As input we have a greyscale image where black pixel has value of -1 and a white pixel has value of +1.

This task would be trivial if X's and O's always had the same shape, which is not the case. Identifying smaller features (in this case mini-images of size 3x3 with elements like diagonal lines, crosses etc), comparing it piece by piece with original images and calculating the match is a typical thing to do to solve the problem. The math we use is called **convolution**, what gives the name to the algorithm (Convolutional Neural Networks). Convolutional layers can be explained intuitively. We know that each image is composed of smaller features which we can recognize. If we think about digits, digits may consist of vertical lines, horizontal lines, circles etc. Those could be the meaningful features the filters of the first convolutional layer could detect.

Convolutional layers use **activation functions** as Maxout or ReLu introduce non-linearities in the neural network in order to make the extracted features more meaningful. ReLu simply turns all negative input values into 0 what helps the neural network to keep learned values from getting stuck near 0. However, problem of "dead" ReLu may occur in situations when a large negative bias term is learned. In this case ReLu always outputs the same value and takes no role in discriminating between inputs.

Maxout introduced by Goodfellow et al is a generalization of ReLu that solves "dead" ReLu problem. Maxout neuron computes the following function (in case of ReLu the first argument is 0):

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Fully connected layers typically use softmax as activation function.

Pooling layers are usually used convolutional layers. Pooling takes the output of the convolutional layer and uses kernels of given size (for example, 2x2) in order to take maximum value within the kernel (max pooling) or average value (average pooling). When using pooling, we have to specify the stride with which we slide the kernel. When stride > 1, we will reduce output volumes and simplify the information from the convolutional layer.

Typically, multiple convolutional layers and pooling steps are stacked together in a CNN. As an output of all these layers we get the final number of features which will be used further in a neural network as we would do it with a normal dataset.

Backpropagation is a way to compute weights in the neural network. First we initialize the weights and forward propagate to get the output, compare it to actual value and compute the error. To minimize the error, we are backpropagating through the neural network, what results in a new set of weights that lead to a smaller error. We yield new weights through a process of optimization via gradient descent.

II.2.2 Example of solving SVHN classification problem

Goodfellow et al train a convolutional neural network that predicts the set of parameters: length of the sequence of digits (max 5) and digit at each position in the sequence. This approach is different from previous works where authors were concentrating on splitting the sequence into digits and recognizing it separately. Final architecture of the model used in their work can be represented in the following way:

```
INPUT [54, 54, 3]
CONV1-48-5 -> SUBNORM (R=3) -> MAXOUT -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 2, 2, 1] -> DROPOUT
CONV2-64-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 1, 1, 1] -> DROPOUT
CONV3-128-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 2, 2, 1] -> DROPOUT
CONV4-160-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 1, 1, 1] -> DROPOUT
CONV5-192-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 2, 2, 1] -> DROPOUT
CONV6-192-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 1, 1, 1] -> DROPOUT
CONV7-192-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 2, 2, 1] -> DROPOUT
CONV8-192-5 -> SUBNORM (R=3) -> RELU -> MAXPOOL : ksize [1, 2, 2, 1], strides [1, 1, 1, 1] -> DROPOUT
LC -> -> RELU -> DROPOUT
FC-3092 -> RELU -> DROPOUT
FC-3092 -> RELU -> DROPOUT
```

II.3. Benchmark

Goodfellow et Al (2014) can achieve accuracy of 97,84% on SVHN dataset. The architecture they use is quite complex (exact architecture is discussed in the next section), and model training took approximately 6 days. I do not have that much time and computational resources and will train a simpler model with less iterations. Therefore accuracy of 80-85% would be satisfactory achievement. I will discuss how the performance can be improved by adjusting model architecture and increasing the training time.

III. Methodology

III.2.1 Data preprocessing

From the previous section we can see that we face some challenges with SVHN dataset. Therefore data preparation is needed before we start training the model.

First of all, the model requires the pictures of equal size. Therefore we would have to crop the pictures. Goodfellow et Al (2014) proposed the following methodology:

We preprocess the dataset in the following way – first we find the small rectangular bounding box that will contain individual character bounding boxes. We then expand this bounding box by 30% in both the x and the y direction, crop the image to that bounding box and resize the crop to 64×64 pixels. We then crop a 54×54 pixel image from a random location within the 64×64 pixel image. This means we generated several randomly shifted versions of each training example, in order to increase the size of the dataset

I did not follow the proposed methodology because of concerns regarding the training set getting much bigger and computation power increase needed because of that.

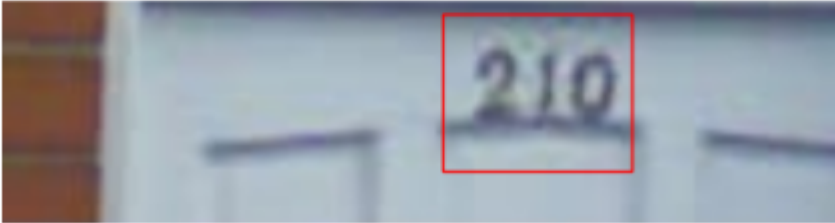
Instead, I followed the following approach. I decided to create one bounding box for each picture that contains all individual bounding boxes, expand it by 20%, crop the image to that new bounding box and crop the image to 48×48 . This image size is easier to use in convolutions when I use ksize [1, 2, 2, 1] and strides [1, 2, 2, 1] in each convolutional layer. After cropping the image and subtracting the mean I also greyscaled because I believe that keeping the images RGB does not really help identifying the digits correctly, but costs a lot in terms of computing power.

This is an example how the proposed approach works on a real picture:

Input image



Input image with new bounding boxes



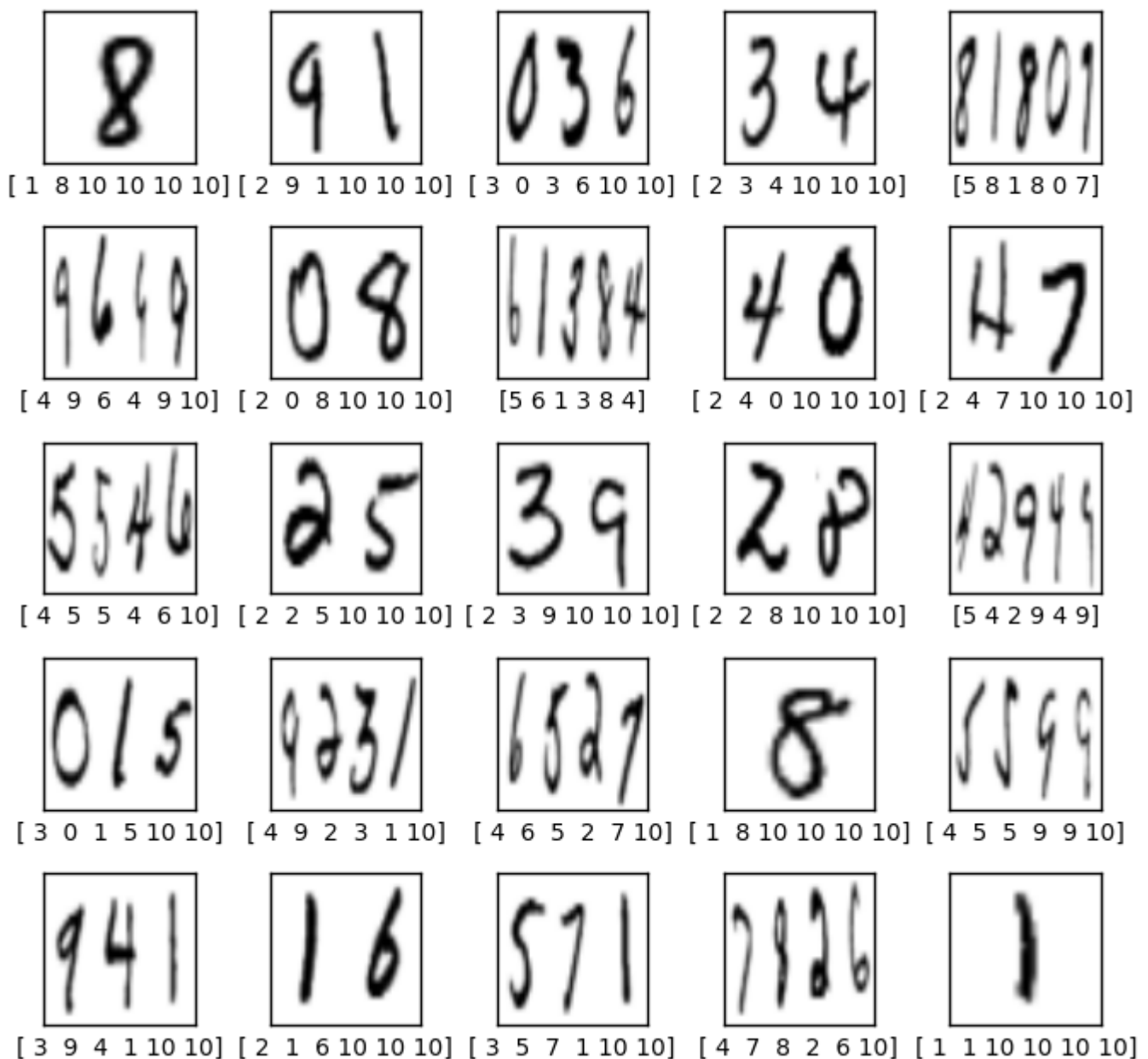
Input image cropped and greyscaled



Next to SVHN dataset transformation I also created a syntetic dataset from the MNIST dataset in the following way:

In order to generate 1 sequence: 1. a random number L from interval (0;5] is selected 2. L random images are selected from the dataset 3. The selected images are stacked together and reshaped to format 48x48 (which I chose to use for SVHN dataset)

This procedure is repeated amount of times that is the length of the dataset so that the size of training, test and validation set stays the same. The result of data transformation with lables in the following way: first digit represents the length of the sequence, the rest represents digit at each place in the sequence. If the digit is absent, lable 10 is given for that digit.



III.2.2 Implementation

**** MNIST 1 digit ****

I started the project with implementing logistic regression on single-digit MNIST dataset using Tensorflow. Even with logistic regression (and only 2000 iterations) I could achieve about 90% accuracy on the test set.

I improved the result by implementing a simple neural network with one hidden layer and after that a convolutional network with architecture proposed in Hvas-Labs tutorials (<https://github.com/Hvas-Labs/TensorFlow-Tutorials>), which resulted in accuracy of 96,4% after 2000 iterations:

```
INPUT [28, 28, 1]
CONV1-16-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
```

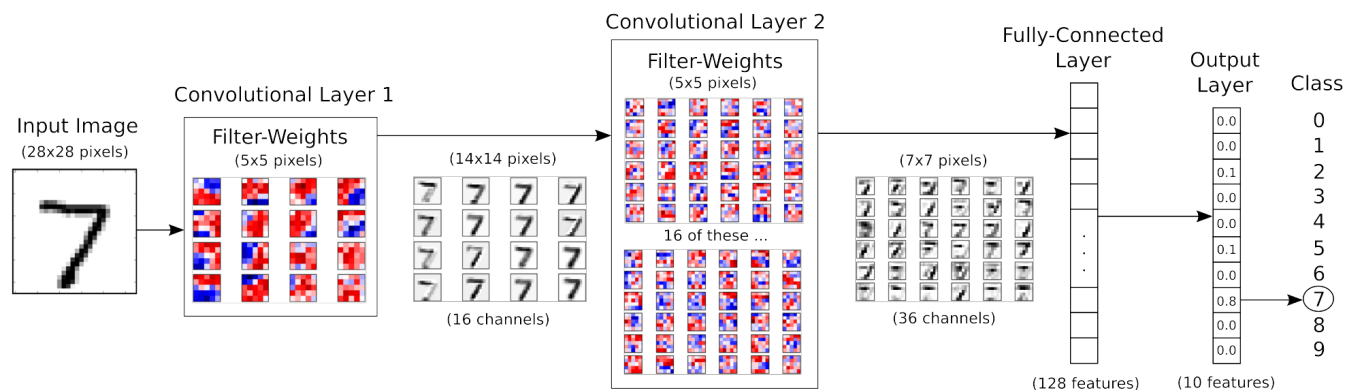


```

CONV2-36-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
LC
FC-128 -> RELU

```

Input images have 1 channel (greyscaled) and are of size 28x28. Examples of images are shown above.



The input image is goes to the first convolutional layer that uses 16 filters. As a result, we get 16 new images from 1 input image. These images are downsized to 14x14 after maxpooling (with kernel size 2x2 and stride 2). These 16 images go to the second convolutional layer with 36 filters, which results is $16 \times 36 = 576$ convolutions in the second convolutional layer. The output of the second convolutional layer is 36 images of 7x7 pixels each (after maxpooling with kernel size 2x2 and stride 2).

These images are flattened to a single vector of length $7 \times 7 \times 36 = 1764$, which is used as the input to a fully-connected layer with 1 hidden layer with 128 neurons. Output layer contains 10 neurons, one for each of the classes.

**** MNIST multi-digit ****

After excercising with 1-digit MNIST dataset I moved to a more complex problem: recognizing the sequence of MNIST digits (artificially created dataset using the methodology described above). As well as Goodfellow et Al, I decided to build an algorithm that focuses on recognizing all digits in the sequence simultaneously.

I used a simpler architecture for a multi-digit MNIST dataset than Goodfellow er Al did:

```

INPUT [48, 48, 1]
CONV1-16-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
CONV2-32-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
CONV2-64-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
LC
FC-1024 -> RELU -> DROPOUT

```

This resulted in accuracy of about 88% already after 10000 steps. Important to mention that this result was achieved because of correct weights initialization and dropout.

In the previous exercise I used `truncated_normal` command (that outputs random values from a truncated normal distribution) from `tensorflow` module in order to initialize the weights. I used standard deviation of 0.1, and it worked fine for a 1-digit MNIST dataset. This kind of initialization gave horrible results for a multi-digit MNIST problem. I switched to Xavier initialization, which worked much better in this case.

In Xavier initialization we pick the weights from a Gaussian distribution with zero mean and a variance of $1/(\text{average of the number input neurons and the output neurons})$. If we look at our network, we standard deviation would be approximately 0.2 for second convolutional layer and 0.14 for third convolutional layer, which differs quite a lot from 0.1 we used initially.

For this problem I used dropout of 0.8. Dropout is a method being used to prevent the neural network from overfitting. While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. Here is some explanation from the Dropout paper by Srivastava et Al:

If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p . This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time.

**** SVHN multi-digit ****

I applied the same architecture to the SVHN dataset, which resulted in a much worse accuracy after 50000 steps (about 76%). After some refinement I managed to achieve satisfactory accuracy. Adjustments will be discussed in the next section

III.2.3 Refinement

My SVHN multi-digit model required some refinement. With current arcitecture I could not achieve satisfactory results. I decided to change architecture in the following way:

```
INPUT [48, 48, 1]
CONV1-16-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
CONV2-32-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
CONV2-64-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
CONV2-128-5 -> RELU -> MAXPOOL: ksize [1, 2, 2, 1], strides [1, 2, 2, 1]
LC
FC-1024 -> RELU -> DROPOUT
FC-512 -> RELU -> DROPOUT
```

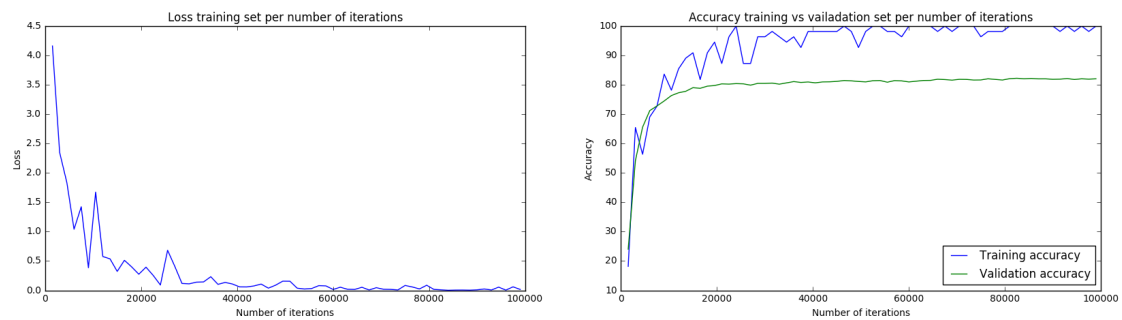
I added extra convolutional layer with 128 filters and extra fully connected layer with 512 neurons. In this way I could achieve satisfactory for my purposes result of 82.5%. I tried adding extra layers and varying the number strides in the fashion like Goodfellow et Al did, but it did not bring better results after 40000 steps, so I decided to keep the described architecture.

I also tried dropout rates of 0.6 and 0.7, and did not achieve better results after 40000 steps.

III. Results and conclusions

III.1. Model quality

The final model has accuracy 82,5% on the test set, which is satisfactory. On the other hand, we see that the model suffers from overfitting. Clearly, we have quite a big gap between training and validation/ test accuracy.



Let's take a look at the sequences that were predicted correctly:



True: 474,
Pred: 474



True: 165,
Pred: 165



True: 50,
Pred: 50



True: 116,
Pred: 116



True: 11,
Pred: 11



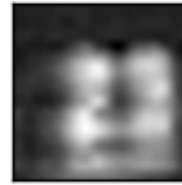
True: 124,
Pred: 124



True: 197,
Pred: 197



True: 9,
Pred: 9



True: 23,
Pred: 23



True: 387,
Pred: 387



True: 23,
Pred: 23



True: 619,
Pred: 619



True: 68,
Pred: 68



True: 15,
Pred: 15



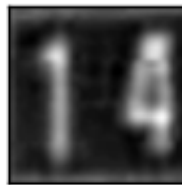
True: 119,
Pred: 119



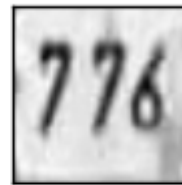
True: 93,
Pred: 93



True: 712,
Pred: 712



True: 14,
Pred: 14



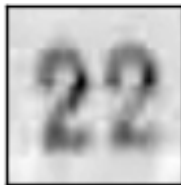
True: 776,
Pred: 776



True: 174,
Pred: 174



True: 14,
Pred: 14



True: 22,
Pred: 22



True: 27,
Pred: 27



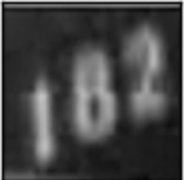















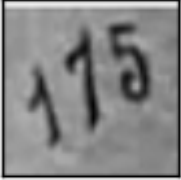







True: 30,
Pred: 30



True: 410,
Pred: 410

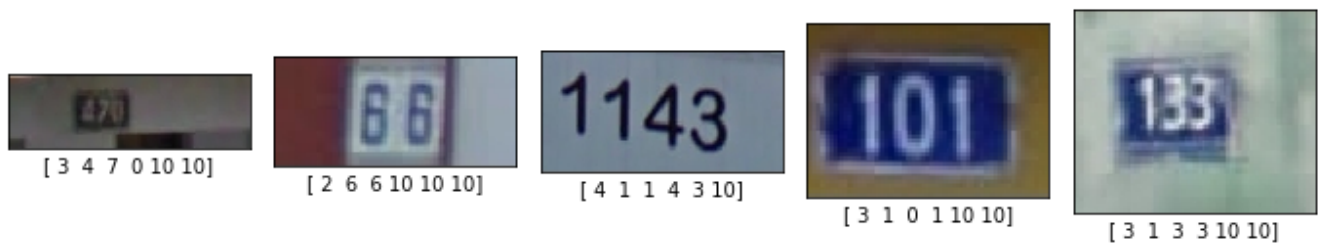
We can see that pretty complicated cases were recognized correctly (when digits are rotated or are in the frame, what can possibly mislead the algorithm). We can also see that model "got lucky" when predicting sequence 23 (4th in the second row). Human could not possibly recognize that sequence.

Let's take a look at the sequences that were not predicted correctly:

				
True: 162, Pred: 102	True: 305, Pred: 308	True: 51, Pred: 91	True: 74, Pred: 24	True: 222, Pred: 22
				
True: 34, Pred: 39	True: 45, Pred: 15	True: 1003, Pred: 1815	True: 587, Pred: 367	True: 33, Pred: 99
				
True: 9, Pred: 8	True: 62, Pred: 662	True: 41, Pred: 17	True: 2000, Pred: 1000	True: 39, Pred: 29
				
True: 1710, Pred: 1749	True: 175, Pred: 115	True: 61, Pred: 611	True: 1254, Pred: 1264	True: 720, Pred: 420
				
True: 89, Pred: 39	True: 39, Pred: 32	True: 256, Pred: 255	True: 22, Pred: 24	True: 38, Pred: 36

Almost all of these sequences could be predicted correctly by human (except maybe number 45, 2nd in the 2nd row and number 74, 3rd in the 1st row). Also, we see that number 1003 is predicted as 1815, which is quite a big mistake. There is definitely space for improvement.

Let's use the model we built and use it for making new predictions for not used images from "extra" dataset:



First we have to transform the pictures in the way described above. The code can be found in `notebook 6. New predictions SVHN dataset`. This is the result of transformation:



After the transformation is done, we can feed it into model saved in `notebook 5. Neural net SVHN sequence`. All predictions were accurate:



III.2. Justification

As I mentioned above, the final model has accuracy 82,5% on the test set, which falls within the benchmark of 80-85% I established earlier. The purpose of the project was not to build a model which can be perfectly used for house number recognition in Google maps, but to learn how this kind of models can be built and what kind of techniques the stated problem requires. For this purpose the model result is perfectly fine. However, the model would need improvement if we really want to solve the problem of house number recognition (this would require human accuracy (around 98%). This would require much more time and computing resources.

III.3. Reflections

This project was an important step for me to get understanding of Convolutional Neural Networks

and how this algorithm can be used to solve the problems of image recognition.

I learned the logics and theory behind the algorithm and adapted the theory on the real-world problem. The model is built with one of the most advanced opensourced ML libraries (Tensorflow), which makes it quite easy for a data scientist to work with neural networks. Layers can be constructed in a clear logical way, what helps you to concentrate on the model quality rather than complexity of coding.

I also learned the basics of image transformations in Python and understood the importance of it while working with image recognition problems. The quality of your model can change drastically when you choose for the right way to transform the input pictures. SVHN dataset had bounding boxes available, what helped a lot with image processing. I also "cheated" by assuming that this information is known while making predictions for random pictures in the end. However, the bounding boxes can be added to real pictures by using some libraries like OpenCV. This means that my code would also work on any picture of house number taken from someone's mobile camera if we add an OpenCV step to it.

My model in the end is not a perfect model. It has accuracy of 82,5%, overfits and makes some obvious mistakes (like shown above). The model could get better if we adjust model architecture by adding more convolutional layers, add L2 regularization and use Decay Learning Rate. These improvements are relatively easy to implement, but it is not so easy to evaluate their improvement in the final performance. This is basically because each training stage of 100k steps is not cheap at all and takes time and computation resources.

The project I chose was quite challenging for me and I learned a lot. I was "stuck" in some places like getting metadata for SVHN dataset (never worked with Matlab data files before), understanding the bounding boxes (what "top", "left", "right", "bottom" exactly means) and it took me some time to figure it out. It was interesting to understand how images are represented and how you can modify the images by manipulating values in arrays that represent the image in different ways (gave me some feeling about how image editors work).

I was somehow familiar with Neural network algorithm before (but was always applying it to more standard datasets), so it was not that challenging to understand the fully connected layer part of CNN's. Convolutional layer had to do a lot with image transformations that I learned during the project and in the end it all seemed intuitive to me. Interesting part in CNN's was understanding that if you add extra factor (like sequence length) as a factor that model has to predict, the model gets better. Tweaking the model was also quite challenging and time-consuming.

In the end I am happy I've done the project and am able to apply the knowledge to other problems I may face (for instance, crowd-recognition).

References

1. The Street View House Numbers (SVHN) Dataset : <http://ufldl.stanford.edu/housenumbers/>
2. 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.

<http://static.googleusercontent.com/media/research.google.com/fr//pubs/archive/42241.pdf>

3. How convolutional neural networks work.

http://brohrer.github.io/how_convolutional_neural_networks_work.html

4. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting.

<http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>