

Udacity Machine Learning Engineer Capstone Project

Srivastava, Shashank

1/3/2017

Table of Contents

Definition	1
Project Overview.....	1
Problem Statement.....	2
Metrics	2
Analysis	3
Data Exploration	3
Data Visualization	4
Algorithm	4
Benchmark	15
Methodology.....	15
Data Preprocessing	15
Implementation	15
Model Training.....	16
CNN Defined.....	16
Model Refinement	16
Results.....	17
Model Evaluation and Validation.....	17
Justification	17
Conclusion.....	20
Free Form Visualization	20
Reflection	20
Improvement	21
References	21

Definition

Project Overview

In this project, we will be working to create a model to automate the digit recognition in an image using deep learning concepts. The model can be applied to solve real word problems such as recognizing

house numbers from google street view, read number from images captured by drones etc. Since house numbers are combination of digits from 0-9, we will be building a classifier that to classify the digits and convolution neural network (CNN) will be used to scan through the images. Dataset used for the project is Street View House Number (SVHN) data set (1).

Problem Statement

We will be developing a deep learning model to accurately predict sequence of digits in images. The output of the prediction model will be each sequence of digits classified in one of 10 possible classes correctly.

For training and validation, we will create synthetic dataset from MNIST dataset. After the model is trained and validated against synthetic dataset, model will be tested against SVHN dataset. Below are steps that will be performed to develop the model:

- 1. Explore SVHN Dataset**

First step is to explore the SVHN dataset to understand the image size, format and other attributes.

- 2. Synthetic Dataset Creation**

We would create a synthetic dataset by concatenating digits in the MNIST dataset. These values would then be split into train, validation and test dataset.

- 3. Data Extraction- SVHN Dataset**

In this step, we will extract data and make it compatible to work with our deep learning model. We will normalize the data images to be of same size i.e. 32 X 32. We will be using Gaussian technique to normalize the data.

- 4. Modelling**

Here, we will develop our deep learning model using CNN and train the model and validate the model using MNIST synthetic dataset created in step 2.

- 5. Model Evaluation**

Here we will evaluate the model's performance on training and validation set.

- 6. Testing with SVHN Dataset**

In this step, we will test the model with SVHN dataset and compare the testing results with training results. We may tweak the model to improve the performance and re-train the model for better results.

Metrics

Credit will be given when model correctly predicts all the sequence of the digits in an image. In-correct prediction of the digits will lead to not finding of house number in production environment. For example: If image contains digits 7048 and model classifies the image as 1048 then no points will be given for the prediction. We will use percentage to measure the performance of the model. Points will only be given when all the digit sequence in an image are classified correctly.

Analysis

Data Exploration

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. The data has been spaced into test, train and extra data sets. There is additional dataset MNIST like of 32x32 images.

- ✚ 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10.
- ✚ 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data
- ✚ Comes in two formats:
 - Original images with character level bounding boxes.
 - MNIST-like 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides).



Figure 1 : SVHN Dataset Representation

These are the original, variable-resolution, color house-number images with character level bounding boxes, as shown in the examples images above. (The blue bounding boxes here are just for illustration

purposes. The bounding box information is stored in digitStruct.mat instead of drawn directly on the images in the dataset.) Each 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.

Data Visualization

As per the SVHN website, all the images are original, colored and have variable resolution with character level bounding boxes.

Character Height

The character is defined as a distance between the top and the bottom of the bounding box.

Dataset	Mean	Median	Standard Deviation
Train	34.366	30.0	19.378
Test	27.898	24	13.459

Above metrics clearly shows that dataset is more difficult as it is more spread-out and images with bigger character height.

Each image in dataset contains sequence of characters; to understand the data more I have plotted the digits in the images to make sure that dataset covers all 10 classes.

Algorithm

I'll be using a Convolution Neural Network (**ConvNets**) to predict recognize a sequence of digits. Convolutional Neural Networks (**ConvNets** or **CNNs**) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. ConvNets have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self-driving cars.

There are four main operations in the ConvNet:

1. Convolution
2. Non Linearity (ReLU)
3. Pooling or Sub Sampling
4. Classification (Fully Connected Layer)

Images are a matrix of pixel values

Essentially, every image can be represented as a matrix of pixel values.

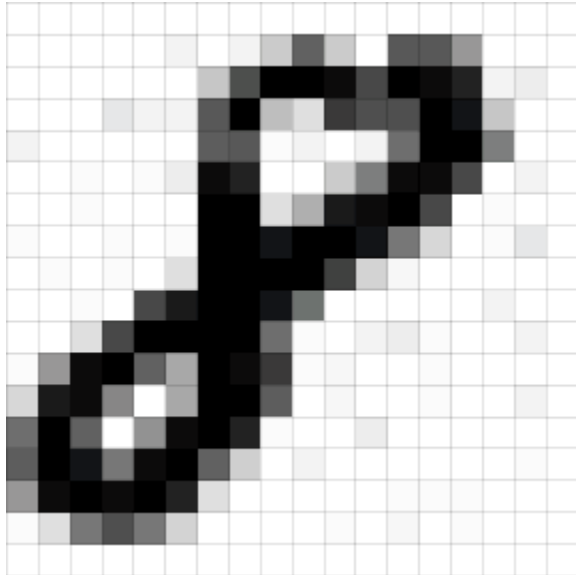


Figure 2: Image

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 12 0 11 39 137 37 0 152 147 04 0 0 0
0 0 1 0 0 0 41 168 258 255 235 162 255 236 206 11 13 0
0 0 0 16 9 9 150 251 45 21 184 159 154 255 233 40 0 0
10 0 0 0 0 0 145 146 3 10 0 11 124 253 255 187 0 0
0 0 3 0 4 15 236 216 0 0 31 109 247 240 169 0 11 0
1 0 2 0 0 0 253 253 23 62 224 241 255 164 0 5 0 0
6 0 0 4 0 3 252 250 228 255 255 234 112 28 0 2 17 0
0 2 1 4 0 21 255 253 251 255 171 31 0 0 1 0 0 0
0 0 4 0 163 225 251 255 229 128 0 0 0 0 0 11 0 0
0 0 21 162 255 255 254 255 126 6 0 10 14 6 0 0 9 0
3 79 242 255 141 66 255 245 189 7 0 0 0 5 0 0 0 0
16 221 237 98 0 67 251 255 144 0 0 0 0 7 0 0 11 0
125 255 141 0 87 244 255 288 3 0 0 13 0 1 0 1 0 0
145 240 228 116 235 255 141 34 0 11 0 1 0 0 0 1 3 0
85 237 253 246 255 210 21 1 0 1 0 0 6 2 4 0 0 0
6 23 112 157 114 32 0 0 0 0 2 0 0 0 7 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 3: Image as Matrix of Pixel

Channel is a conventional term used to refer to a certain component of an image. An image from a standard digital camera will have three channels – red, green and blue – you can imagine those as

three 2d-matrices stacked over each other (one for each color), each having pixel values in the range 0 to 255.

This is largely due to the network being able to identify images at different scales. Another advantage here is that a convnet is able to directly work on raw pixels. We'll be using an architecture similar to the one made popular by the tensor flow website to solve the CIFAR-10 dataset.

The Convolution Step

ConvNets derive their name from the "convolution" operator. The primary purpose of Convolution in case of a ConvNet is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. We will not go into the mathematical details of Convolution here, but will try to understand how it works over images.

As we discussed above, every image can be considered as a matrix of pixel values. Consider a 5 x 5 image whose pixel values are only 0 and 1 (note that for a grayscale image, pixel values range from 0 to 255, the green matrix below is a special case where pixel values are only 0 and 1):

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Also, consider another 3 x 3 matrix as shown below:

1	0	1
0	1	0
1	0	1

Then, the Convolution of the 5 x 5 image and the 3 x 3 matrix can be computed as shown in the animation in figure 4 below:

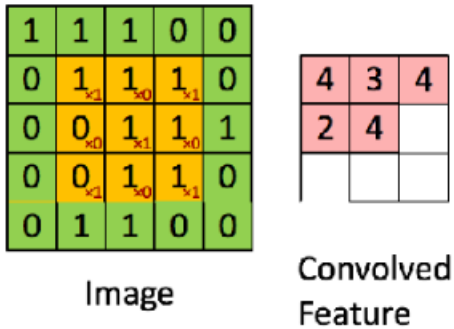


Figure 4: The Convolution operation. The output matrix is called Convolved Feature or Feature Map



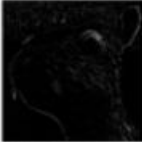
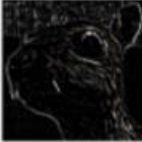



We slide the orange matrix over our original image (green) by 1 pixel (also called ‘stride’) and for every position, we compute element wise multiplication (between the two matrices) and add the multiplication outputs to get the final integer which forms a single element of the output matrix (pink). Note that the 3×3 matrix “sees” only a part of the input image in each stride.

In CNN terminology, the 3×3 matrix is called a ‘**filter**’ or ‘kernel’ or ‘feature detector’ and the matrix formed by sliding the filter over the image and computing the dot product is called the ‘Convolved Feature’ or ‘Activation Map’ or the ‘**Feature Map**’. It is important to note that filters acts as feature detectors from the original input image.

It is evident from the animation above that different values of the filter matrix will produce different Feature Maps for the same input image. As an example, consider the following input image



In the table below, we can see the effects of convolution of the above image with different filters. As shown, we can perform operations such as Edge Detection, Sharpen and Blur just by changing the numeric values of our filter matrix before the convolution operation [8] – this means that different filters can detect different features from an image, for example edges, curves etc.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Another good way to understand the Convolution operation is by looking at the figure below

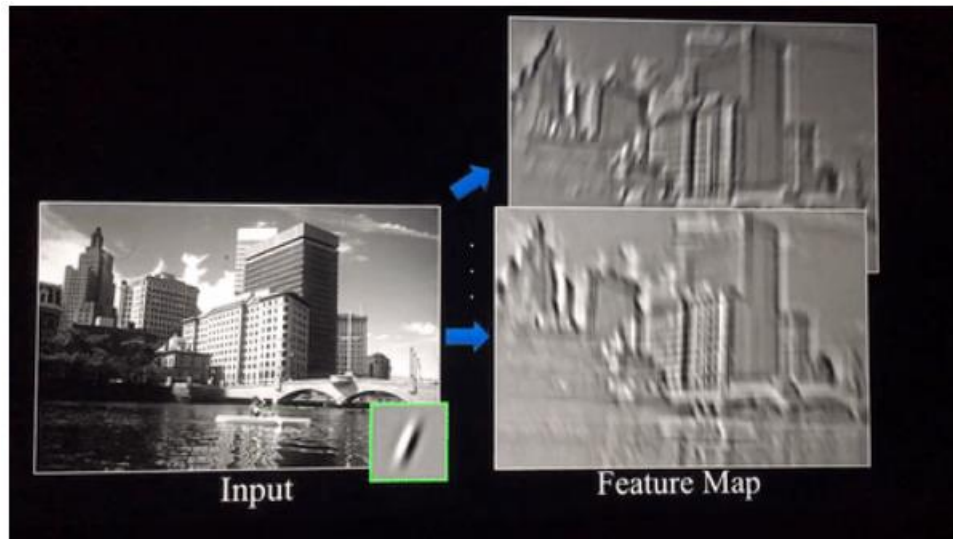


Figure 5: The Convolution Operation

A filter (with green outline) slides over the input image (convolution operation) to produce a feature map. The convolution of another filter (with the green outline), over the same image gives a different feature map as shown. It is important to note that the Convolution operation captures the local dependencies in the original image. Also notice how these two different filters generate different feature maps from the same original image. Remember that the image and the two filters above are just numeric matrices as we have discussed above.

In practice, a CNN *learns* the values of these filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size, architecture of the network etc. before the training process). The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

The size of the Feature Map (Convolved Feature) is controlled by three parameters that we need to decide before the convolution step is performed:

- **Depth:** Depth corresponds to the number of filters we use for the convolution operation. In the network shown in Figure 6, we are performing convolution of the original boat image using three distinct filters, thus producing three different feature maps as shown. You can think of these three feature maps as stacked 2d matrices, so, the 'depth' of the feature map would be three.

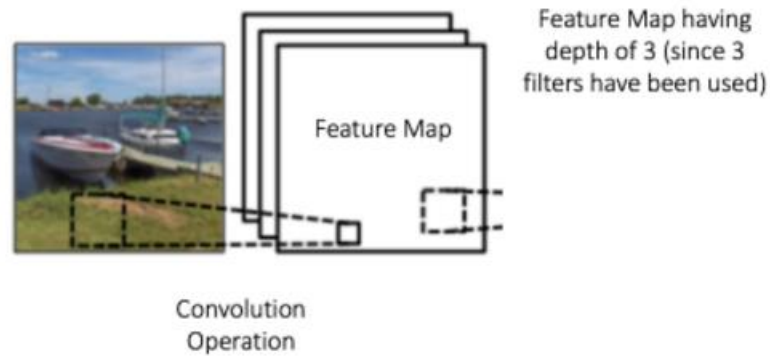


Figure 6: Feature Map with Depth

- **Stride:** Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.
- **Zero-padding:** Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution.

Introducing Non Linearity (ReLU)

An additional operation called ReLU has been used after every Convolution operation. ReLU stands for Rectified Linear Unit and is a non-linear operation. Its output is given by:

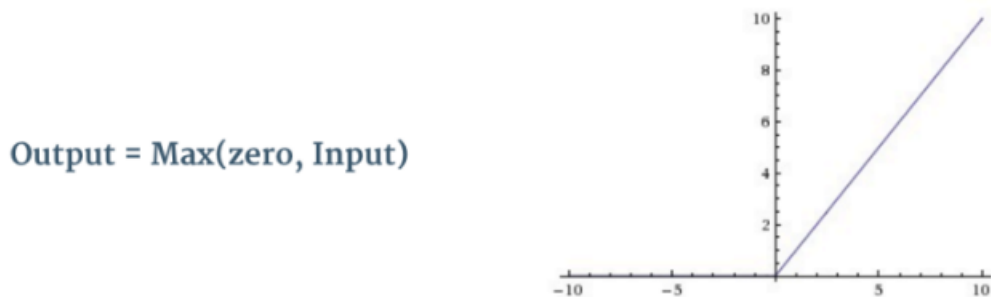


Figure 7 : The ReLU operation

ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear

operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU).

The ReLU operation can be understood clearly from figure 8 below. It shows the ReLU operation applied to one of the feature maps obtained in Figure 5 above. The output feature map here is also referred to as the 'Rectified' feature map.

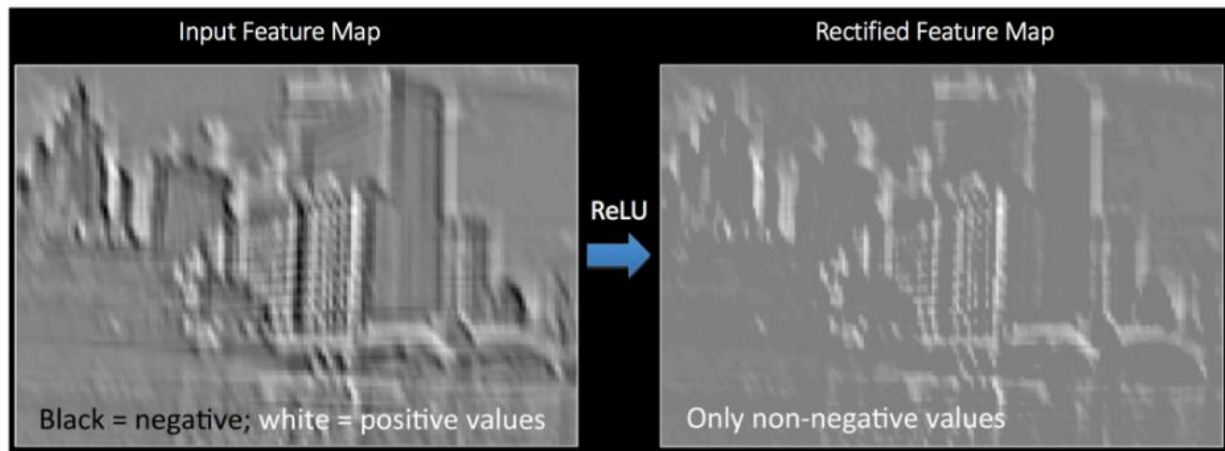


Figure 8: ReLU operation

The Pooling Step

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Figure 9 shows an example of Max Pooling operation on a Rectified Feature map (obtained after convolution + ReLU operation) by using a 2×2 window.

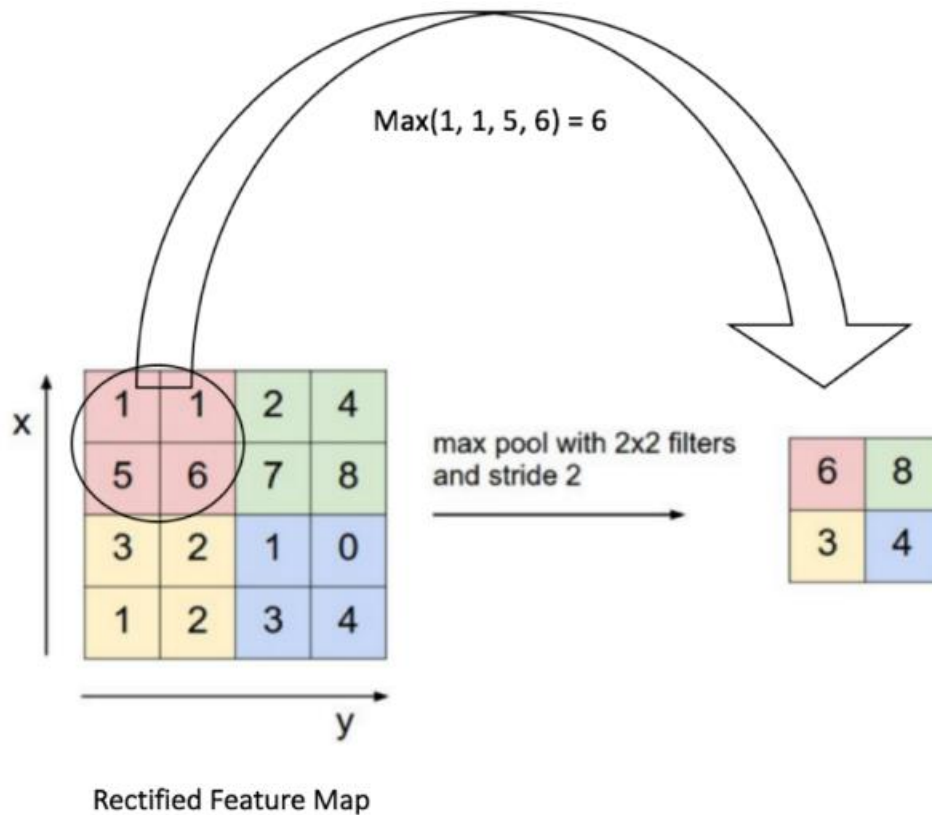


Figure 9: Max Pooling

We slide our 2 x 2 window by 2 cells (also called 'stride') and take the maximum value in each region. As shown in Figure 9, this reduces the dimensionality of our feature map.

In the network shown in Figure 10, pooling operation is applied separately to each feature map (notice that, due to this, we get three output maps from three input maps).

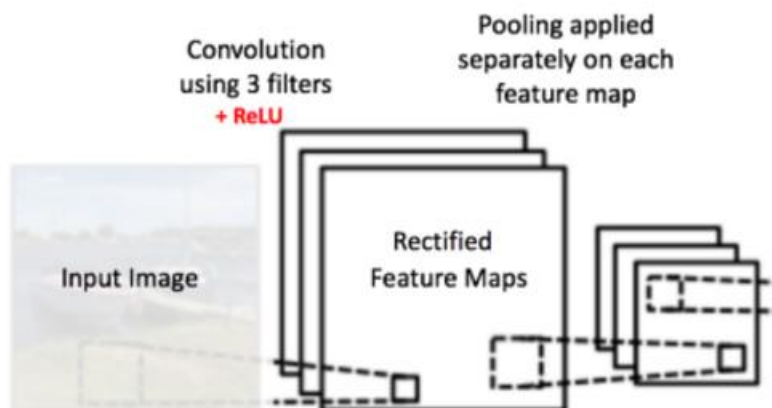


Figure 10: Pooling applied to Rectified Feature Maps

Figure 11 shows the effect of Pooling on the Rectified Feature Map we received after the ReLU operation in Figure 8 above.

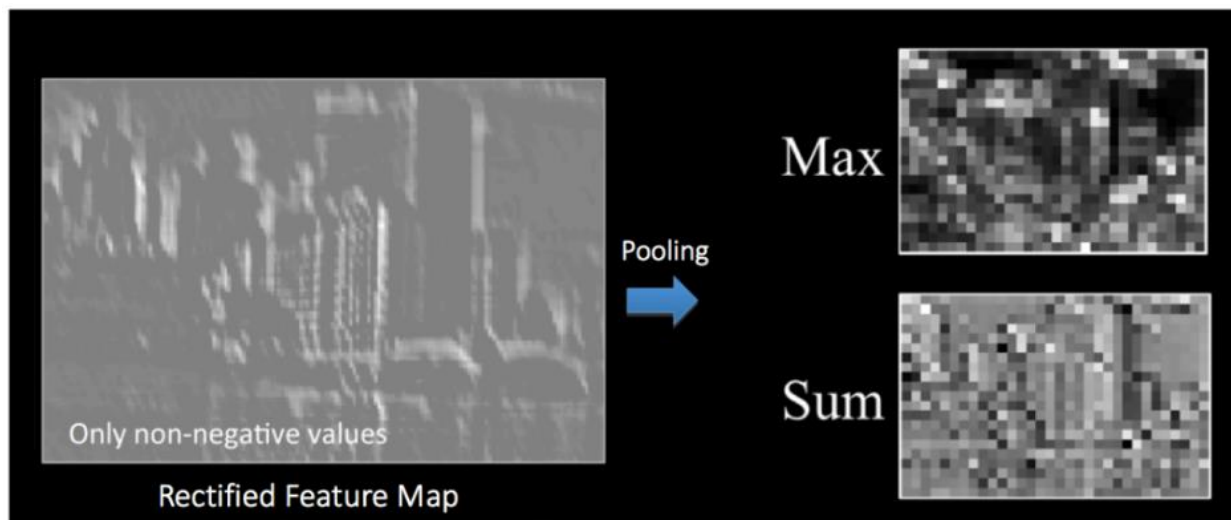


Figure 11: Pooling

The function of Pooling is to progressively reduce the spatial size of the input representation. In particular, pooling

- Makes the input representations (feature dimension) smaller and more manageable
- Reduces the number of parameters and computations in the network, therefore, controlling overfitting.
- Makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).
- Helps us arrive at an almost scale invariant representation of our image (the exact term is “equivariant”). This is very powerful since we can detect objects in an image no matter where they are located.

So from start to finish, our whole five-step pipeline looks like this:

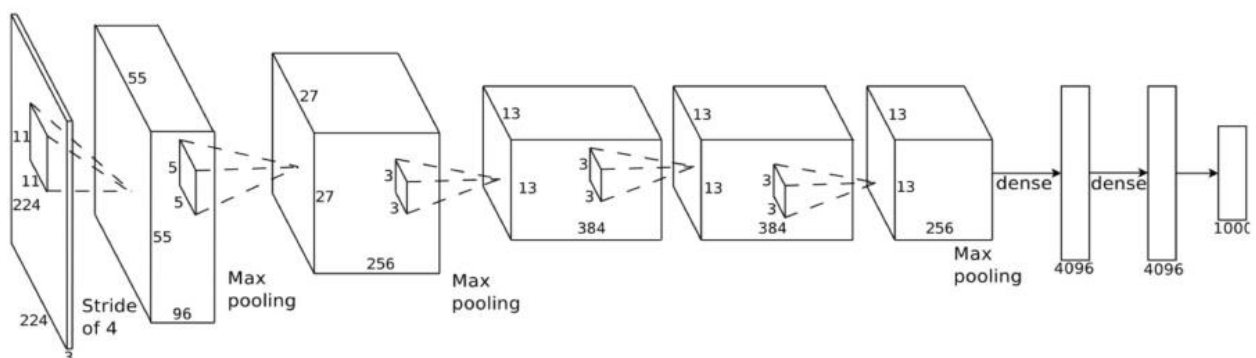


Figure 12: Convolution Neural Network

So far we have seen how Convolution, ReLU and Pooling work. It is important to understand that these layers are the basic building blocks of any CNN. As shown in **Figure 13**, we have two sets of Convolution, ReLU & Pooling layers – the 2nd Convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps. ReLU is then applied individually on all of these six feature maps. We then perform Max Pooling operation separately on each of the six rectified feature maps.

Together these layers extract the useful features from the images, introduce non-linearity in our network and reduce feature dimension while aiming to make the features somewhat equivariant to scale and translation.

The output of the 2nd Pooling Layer acts as an input to the Fully Connected Layer.

Fully Connected Layer

The Fully Connected layer is a traditional Multi Layer Perceptron that uses a softmax activation function in the output layer (other classifiers like SVM can also be used, but will stick to softmax in this post). The term “Fully Connected” implies that every neuron in the previous layer is connected to every neuron on the next layer. I recommend reading this post if you are unfamiliar with Multi Layer Perceptrons.

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset. For example, the image classification task we set out to perform has four possible outputs as shown in Figure 13 below (note that Figure 13 does not show connections between the nodes in the fully connected layer)

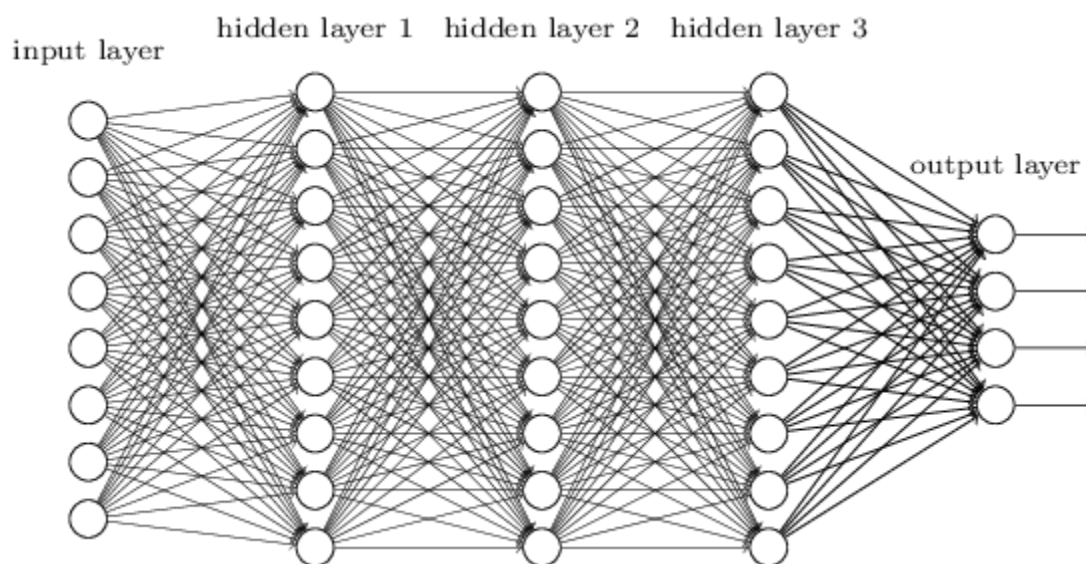


Figure 13: Fully Connected Layer -each node is connected to every other node in the adjacent layer

Apart from classification, adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.

The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the Softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

Benchmark

The primary benchmark here would be the accuracy of prediction. During our data exploration experiments, our classifier was able to predict single digits with only 59% accuracy because of overfitting. Accuracy score is the percentage of correctly predicted data. As per the paper published by Goodfellow et al [6] in 2014 states that experiment achieved 96.03% accuracy in predicting sequence is digits in an image.

Methodology

Data Preprocessing

Followings steps were performed for data pre-processing:

- Extract information from the digitStruct.mat file and save it in a python friendly format.
- Generate new images using the bounding boxes.
- Resize the new images to 32x32 pixels.
- Generate test, train and validation dataset. The validation set is selected by sampling the extra dataset.

Implementation

Implementation is broken down into three steps, and covered in 5 jupyter notebook files.

1. DownloadAndExtractData.ipynb

Here, we pull download the SVHN dataset. We make sure the data is downloaded and extracted without any error. We also make sure that file **digitStruct.mat** is present.

2. DataExploration.ipynb

Here we explore the dataset and generate visualizations such as a histogram of character count / height.

3. DataPreparation.ipynb

Here we Go through our dataset for train, test and extra, load up the images, crop the bounding boxes and the resize the image to 32x32px. We also save pickle our dataset so it can be reused later.

4. Training.ipynb

Here, model is implemented with accuracy function. Weights, bias and classifier are implemented to recognize the digits in images. Model is trained in batches because of limited computing capacity. Trained model is saved to run the testing data for final prediction.

5. Testing.ipynb

This notebook redefines our model, but instead of training, it loads already saved model. We run model against few samples to make the prediction. We run the model against testing set for the accurate predictions.

Dropout Layer

Learning Rate

Model Training

1. Dataset is downloaded and extracted and pre-processed
2. CNN is defined and datasets are loaded in memory
3. The weights are initialized
using Xavier initialization - a tensor flow function that ensures that weights are balanced randomly based on the number of neurons
4. Loss and Accuracy function is defined
5. Model is trained and accuracy and loss is calculated and tracked
6. Trained is saved on disk and loaded in memory for testing and prediction

CNN Defined

1. C1: convolutional layer, batch_size x 28 x 28 x 16, convolution size: 5 x 5 x 1 x 16
2. S2: sub-sampling layer, batch_size x 14 x 14 x 16
3. C3: convolutional layer, batch_size x 10 x 10 x 32, convolution size: 5 x 5 x 16 x 32
4. S4: sub-sampling layer, batch_size x 5 x 5 x 32
5. C5: convolutional layer, batch_size x 1 x 1 x 64, convolution size: 5 x 5 x 32 x 64
6. Dropout F6: fully-connected layer, weight size: 64 x 16
7. Output layer, weight size: 16 x 10

To train, we read the already preprocessed data into the model and train it in batches.

During the training, we keep a track of loss and accuracy to measure the performance of the model.

Once the model is trained, it is saved on disk and can be used in production environment for prediction.

Model Refinement

The initial model produced an accuracy of 82.1% with 300000 training data points. Because of the limited computation capacity of the machine, model was only trained over 300000 data points.

Following steps were taken to improve the accuracy of the model:

1. Dropout is added before the fully connect layer to prevent the model from over-fitting.
2. Learning rate was changes to exponential decay, instead of keep low learning rate.
Model initially learns fast and then learns slowly over the time as more is fed to the model.

After implementing above two features accuracy increased from 82.1% to 85.9% for 30000 training data points.

Once more training data is fed; model should perform better than current accuracy of 85.9%.

Results

Model Evaluation and Validation

At the end of the testing following results were achieved:

Minibatch loss: 1.077957

Minibatch accuracy: 93.8%

Validation accuracy: 75.3%

Test accuracy: 85.9%

According to our benchmark, this model produced much more accurate predictions when compared with the base classifier, although not as good as humans do.

The final model is made up of the following:

1. Weights are initialized using Xavier Initializer
2. Max pooling is implemented at hidden layers
3. Convolutions is implemented for depths at 16, 32 and 64
4. Five classifiers / logits were created
5. Learning rate was kept at 0.05
6. AdagradOptimizer was used as an optimizer for the model
7. Dropout was just before the fully connected layer with 0.9375 keep probability
8. Accuracy score is as our benchmark

Justification

Our benchmark [6] is 96.03% while that of a naive classifier was 59%.

Our model was able to achieve 85.9% accuracy while training on testing set. This is much better than the naive classifier. Since we have over 200,000 more images to train on, we expect the accuracy to continue to improve moving closer to human recognition. According to the paper written by on this model, one can achieve over 97% accuracy with this model.

Why Xavier Initializer?

When working with deep neural networks, initializing the network with the right weights can be the difference between the network converging in a reasonable amount of time and the network loss function not going anywhere even after hundreds of thousands of iterations.

If the weights are too small, then the variance of the input signal starts diminishing as it passes through each layer in the network. The input eventually drops to a really low value and can no longer be useful. And that is a big problem. Let's consider the sigmoid function:

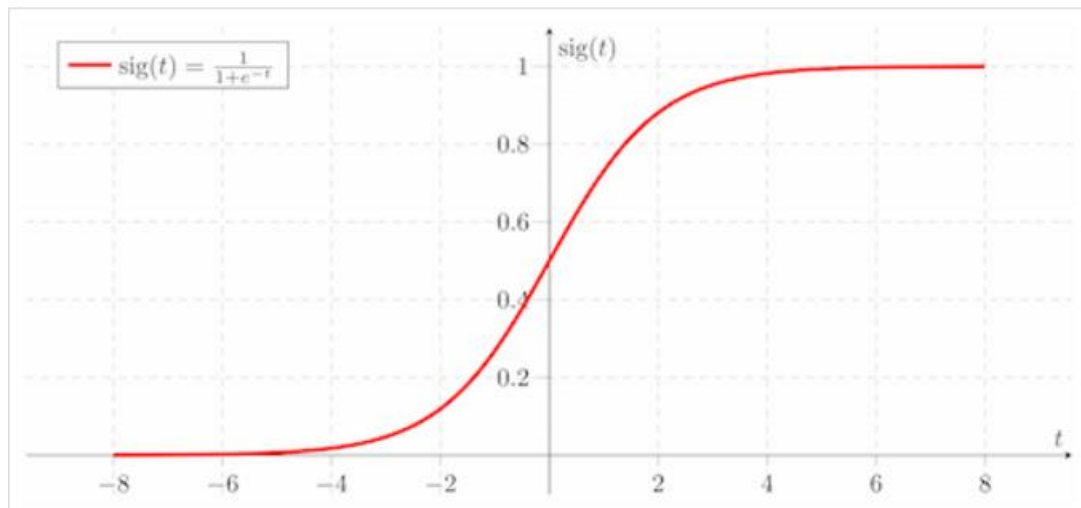


Figure 14: Sigmoid Function

If we use this as the activation function, then we know that it is approximately linear when we go close to zero. This basically means that there won't be any non-linearity. If that's the case, then we lose the advantages of having multiple layers.

If the weights are too large, then the variance of input data tends to rapidly increase with each passing layer. Eventually it becomes so large that it becomes useless. Weights become useless because the sigmoid function tends to become flat for larger values, as we can see the graph above. This means that our activations will become saturated and the gradients will start approaching zero.

Initializing the network with the right weights is very important if you want your neural network to function properly. We need to make sure that the weights are in a reasonable range before we start training the network. This is where Xavier initialization comes into picture.

What exactly is Xavier initialization?

Assigning the network weights before we start training seems to be a random process, right? We don't know anything about the data, so we are not sure how to assign the weights that would work in that particular case. One good way is to assign the weights from a Gaussian distribution. Obviously this distribution would have zero mean and some finite variance. Let's consider a linear neuron:

$$y = w_1x_1 + w_2x_2 + \dots + w_Nx_N + b$$

With each passing layer, we want the variance to remain the same. This helps us keep the signal from exploding to a high value or vanishing to zero. In other words, we need to initialize the weights in such a way that the variance remains the same for x and y . This initialization process is known as Xavier initialization.

How to perform Xavier initialization?

Just to reiterate, we want the variance to remain the same as we pass through each layer. Let's compute the variance of y :

$$\text{var}(y) = \text{var}(w_1x_1 + w_2x_2 + \dots + w_Nx_N + b)$$

Let's compute the variance of the terms inside the parentheses on the right hand side of the above equation. If you consider a general term, we have:

$$\text{var}(w_ix_i) = E(x_i)^2\text{var}(w_i) + E(w_i)^2\text{var}(x_i) + \text{var}(w_i)\text{var}(x_i)$$

Here, $E()$ stands for expectation of a given variable, which basically represents the mean value. We have assumed that the inputs and weights are coming from a Gaussian distribution of zero mean. Hence the " $E()$ " term vanishes and we get:

$$\text{var}(w_ix_i) = \text{var}(w_i)\text{var}(x_i)$$

Note that ' b ' is a constant and has zero variance, so it will vanish. Let's substitute in the original equation:

$$\text{var}(y) = \text{var}(w_1)\text{var}(x_1) + \dots + \text{var}(w_N)\text{var}(x_N)$$

Since they are all identically distributed, we can write:

$$\text{var}(y) = N * \text{var}(w_i) * \text{var}(x_i)$$

So if we want the variance of y to be the same as x , then the term " $N * \text{var}(w_i)$ " should be equal to 1. Hence:

$$N * \text{var}(w_i) = 1$$

$$\text{var}(w_i) = 1/N$$

We arrived at the Xavier initialization formula. We need to pick the weights from a Gaussian distribution with zero mean and a variance of $1/N$, where N specifies the number of input neurons. This is how it's implemented in the Caffe library. In the original paper [8], the authors take the average of the number input neurons and the output neurons. So the formula becomes:

$$\text{var}(w_i) = 1/N_{\text{avg}}$$

$$\text{where } N_{\text{avg}} = (N_{\text{in}} + N_{\text{out}}) / 2$$

The reason they do this is to preserve the back propagated signal as well. But it is more computationally complex to implement. Hence we only take the number of input neurons during practical implementation.

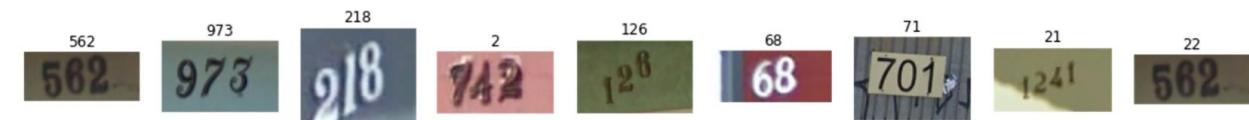
Conclusion

Free Form Visualization

To test how well our model is working, we created random images and used it for our prediction. Here are the actual images and their labels:



These images were then passed through our model and here is what our model predicted the values to be



We can see that our model did a pretty good job at predicting data and it did so with a training set of just over 30000. If we continue to train with the larger set of 600,000 images on a GPU, we will achieve even better results.

Reflection

Deep learning is an interesting field of machine learning that can be applied to many exciting problems. The problem we have applied it to is housing number digit recognition.

While deep learning is capable of solving many problems, the part that I find most challenging is the specialized compute resource required to get accurate results.

The initial solution was to synthetically create a new dataset by combining digits together from either MNIST or SVHN 32x32 dataset. My earlier experiments worked when predicting one character and 5 characters in a sequence but performed poorly on a real dataset. So I used the bounding boxes instead.

Another thing here was to resize images into 32x32px images. My earlier model used 800x600px images to simulate what a phone screen might look like. I was unable to train it because of the sheer amount of memory needed to process that dataset. I rented a 50GB virtual machine in the cloud and I have over 30GB of memory used up on the

first 2000 images using this architecture.

I also converted images to grayscale by averaging the pixels. This was to further reduce the memory footprint. Some other scholars believe there is a certain ratio of Green, Red and Blue that makes an image appear closer to human perspective, however, I did not explore that option.

Improvement

There are several ways this implementation can be improved.

One of which is to run this model on a GPU. Scholars have written on how this can provide up to 10x improvement in the training time for deep neural networks.

Another improvement here would be to explore Recurrent neural networks. Using RNN should be able to predict beyond 5 digits at once. My implementation makes use of 5 classifiers need to be changed when implementing RNN.

References

1. SVHN Dataset - <http://ufldl.stanford.edu/housenumbers/>
2. Google Tensorflow - <https://www.tensorflow.org/versions/r0.11/tutorials/mnist/beginners/>
3. Image Classifier - <https://research.googleblog.com/2016/03/train-your-own-image-classifier-with.html>
4. Reading digits in Neural Network - http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf
5. Deep Learning - <http://deeplearning.net/tutorial/lenet.html>
6. Goodfellow et al 2014 - <http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf>
7. Udacity Deep Learning Course
8. Xavier Initialization - <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>