CAPSTONE PROJECT

Michele Cavaioni

Machine Learning Engineer Nanodegree

November 30th, 2016

I. Definition

Project Overview

Image recognition has been the focus of researchers for many years. It involves the methodology for a machine to interpret an image and classify it.
The first attempt, implemented with successful results, has been applied to the MNIST[1] dataset, which comprises a set of handwritten numbers. The implementation has been already put to practical use in the post office and in banks in order to use computers to understand human calligraphy and quickly classify handwritten numbers.
Further implementation, with additional complexities, is applied to numbers coming from images. The SVHN (Street View House Numbers)[2] dataset is an example and it comprises images taken from Google Street View. The additional complexity of these data comes from the fact that the images have more noise, due to the different background colors, brightness, blurriness and all sorts of resolutions.

My interest in this topic is fueled by the increasing importance that image detection, such as reading road signs, will play in Self Driving Cars, which will be a critical component of its successful implementation.

In this project I apply machine-learning techniques to both datasets, MNIST and SVHN. In the first one I compare the different results by applying different classifiers. For the SVHN dataset I first apply the same algorithm previously optimized and then find a way to improve it. My goal is to highlight the challenges given by different input data and to experiment several models.

Problem Statement

This project applies several machine learning algorithms to the MNIST and the SVHN datasets. The first one is composed of 50,000 handwritten digits; this study uses several models to obtain an optimal classification of test data. It starts with a simple Support Vector Machine (SVM) algorithm and then applies a Neural Network technique, through a Multi-layer Perceptron (MLP) classifier. The results are based on the classification accuracy metrics. The goal is to find the algorithm that best categorizes the images within the test set.

[1] http://yann.lecun.com/exdb/mnist/ (MNIST dataset is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology).

[2] http://ufldl.stanford.edu/housenumbers/

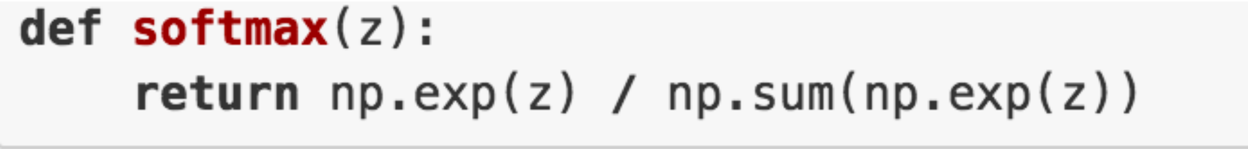
The SVHN dataset is instead evaluated at first using the same MLP classifier, and then improved with a convolutional algorithm, which uses different kernels in order to filter the input images and is finally analyzed with a more complex Convolutional Neural Network (CNN) model. Considerations are given in regards to the performance against this new dataset and solutions for improvements are discussed and implemented. The SVHN set presents different challenges (noise that affects the images) than the previous one and this paper aims to highlight those and to present the best approaches for its analysis.

Metrics

As previously mentioned, the MNIST dataset has first been analyzed through a simple non-neural-network classifier, such as the Support Vector Machine (SVM).
The metric used to evaluate the performance of this model is the accuracy on a given test set. The accuracy is calculated as the number of inputs classified as correct over the total number of inputs.
In our case the number of inputs for the test set was 10,000 units.
The definition of the metric used to evaluate performance is important, but it is also necessary to understand where those obtained results stand in respect to a benchmark.
The place to start is to randomly guess the digits. We have a total of 10 independent digits, so if we guess we have one tenth of getting the right classification.
That's a starting point, although definitely not a particularly clever one. One of the highest records done by Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus is instead classifying with a 99.79% accuracy.
My intention is to play with a simple model, tuning its parameters in order to achieve a result comparable to the last one.

When moving on more complex models, such as the Multi-layer Perceptron (MLP) classifier and the Convolutional Neural Network (CNN), another metric to consider is the cross-entropy error, which gives us a better understanding of the probability associated with each input classification. For the training set performances, I considered both the accuracy and the cross-entropy error. For this set, the last metric gives a more informative overview of the performances.
On the other hand, after training, for the test set I preferred to use classification error to estimate the effectiveness of the neural network, as classification error is what we are ultimately interested in.

The neural network I created uses "Softmax" activation for the output neurons, turning scores (in this case 0 or 1 values for classification) into probabilities.
The "Softmax function" is defined as follows:



```
def softmax(z):
    return np.exp(z) / np.sum(np.exp(z))
```

and it provides the probabilities associated with each input (z) [3].

The cross-entropy function is very useful for evaluating the result coming from a very unbalanced dataset such as the SVHN one.
To understand intuitively why this measure is more useful than a “plain” accuracy classification error, it’s important to see how its formula is built.
The cross-entropy compares the vector from the “Softmax” classifier, consisting of the probabilities of the classes, with the one-hot encoding vector (L). The last one (which will be explained more in details later on, is basically a vector where the index of the label is set to 1 and all other entries to 0).
So, starting from the input X, we apply a linear model (Wx+b, where “W” are the weights and “b” is the bias) and we obtain the scores y (or “logits”).
Applying “Softmax” classifier (S(y)) the above scores get transformed into probabilities. The cross-entropy (D(S,L)) compares the these probabilities with the labels.
Mathematically it is defined as:

$$D(S,L) = - \sum_i Li * log\ (Si)$$

The cross-entropy is the distance between the labels and the results of the classifier.
What we ultimately want is to have weights and bias that provide a low distance for the correct class and a high distance for the incorrect class.
One way to do it is to measure the distance over the entire training set and obtain the loss function. Defined as:

$$Loss = 1/N * \sum_i D(S(w * Xi + b), Li)$$

The goal is to minimize the cross-entropy and therefore have a small value for the loss function.

I have once read a good intuitively way to interpret the cross-function, such as seeing it as a measure of how “surprised” we get when we see the result of our prediction versus the correct label.
If we get a prediction close to the correct class our “surprise” level is very low, but when we get it wrong, we get surprised and the more “far from the truth” we get, the more “surprised” we are.

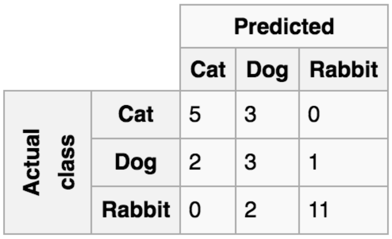
In dealing with a real word scenario, such as the one expressed by the SVHN dataset, knowing how close or far from the correct answer we are is a better explanation than simply calculating how many inputs are correctly interpreted, such as with the accuracy metrics which weighs every sample equally.

Another helpful metric is the “confusion matrix”.
This creates a table where each column represents the instances in a predicted class and each row represents the instances in an actual class. It allows visualization of the performance of an algorithm and it makes it easy to see if the model is confusing and mislabelling two classes.

[3] http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/#Softmax-function

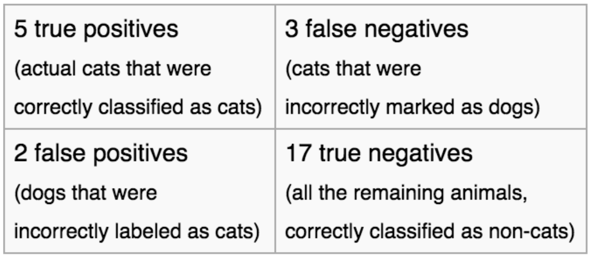
The confusion matrix reports the number of false positives, false negatives, true positives, and true negatives.

For example[4], if a classification system has been trained to distinguish between cats, dogs and rabbits, a confusion matrix will be:



		Predicted		
		Cat	Dog	Rabbit
Actual class	Cat	5	3	0
	Dog	2	3	1
	Rabbit	0	2	11

Its corresponding table of confusion would be represented as:



5 true positives (actual cats that were correctly classified as cats)	3 false negatives (cats that were incorrectly marked as dogs)
2 false positives (dogs that were incorrectly labeled as cats)	17 true negatives (all the remaining animals, correctly classified as non-cats)

[4] https://en.wikipedia.org/wiki/Confusion_matrix

II. Analysis

II - 1.1 Data Exploration (MNIST Dataset)

The MNIST dataset[5] includes a training set of 60,000 handwritten digits and a test set of 10,000 examples, which have been size-normalized and centered in a fixed-size image. Each set has its own label of independent values spanning from 0 to 9.

The input data consists of 28x28 pixel handwritten digits. Below is an example of those:

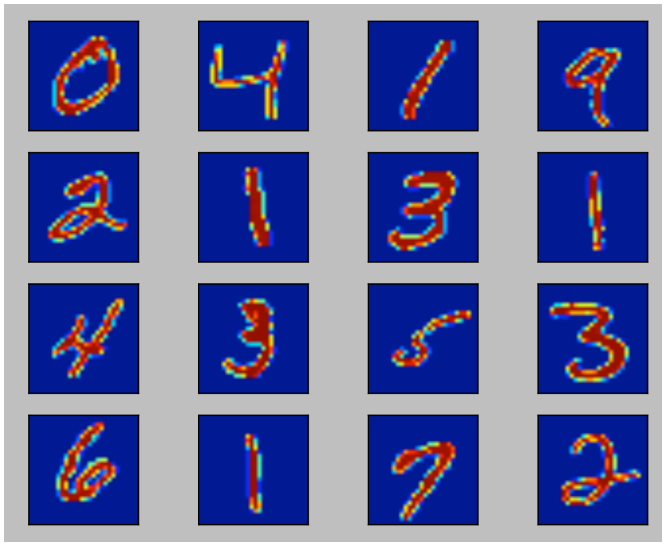
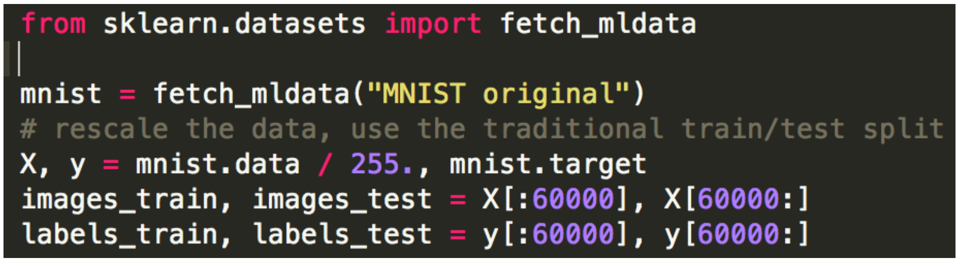


***Fig. 1**: Sample of input images (MNIST dataset)*

The files provided are not in a standard image format and I had to write a specific script in python, in order to read them (ref. code: "mnist_list.py").
Although I discovered that many libraries, such as the "sklearn" library, have their built function to load the well-known MNIST dataset:[6]



```
from sklearn.datasets import fetch_mldata

mnist = fetch_mldata("MNIST original")
# rescale the data, use the traditional train/test split
X, y = mnist.data / 255., mnist.target
images_train, images_test = X[:60000], X[60000:]
labels_train, labels_test = y[:60000], y[60000:]
```

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

[6] http://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_mldata.html#sklearn.datasets.fetch_mldata

I need to preprocess the data before using them as an input into the predictive algorithms.
In particular, the training and testing labels need to be transformed into vectors, generating a vector for each label, where the index of the label is set to `1` and all other entries to `0`.

Also the input data needs to be transformed as well, from 28x28 images sample to a matrix of samples by 784 (= 28pixels * 28 pixels) features. These were normalized by 255 in order to obtain values in a range between 0 and 1.

II - 1.2 Data Exploration (SVHN Dataset)

The SVHN dataset[7] is a more complicated dataset than the previous one. It consists of real images of house numbers taken from Google Street View images.
It contains a much larger dataset, composed of 73,257 digits for training and 26,032 digits for testing. Similar to the MNIST dataset, each input has one label, taken from a total of 10 classes. In this dataset, though, digit '1' has label 1, digit '9' has label 9 and finally digit '0' has label 10. Each image is in a 32x32 pixels format.
These images, taken from a real case scenario, present a variety of peculiarities such as different brightness, different orientation of the digits within the image, disparate blurriness and resolution.
Image processing and filtering are clearly mandatory steps before their usage as inputs.

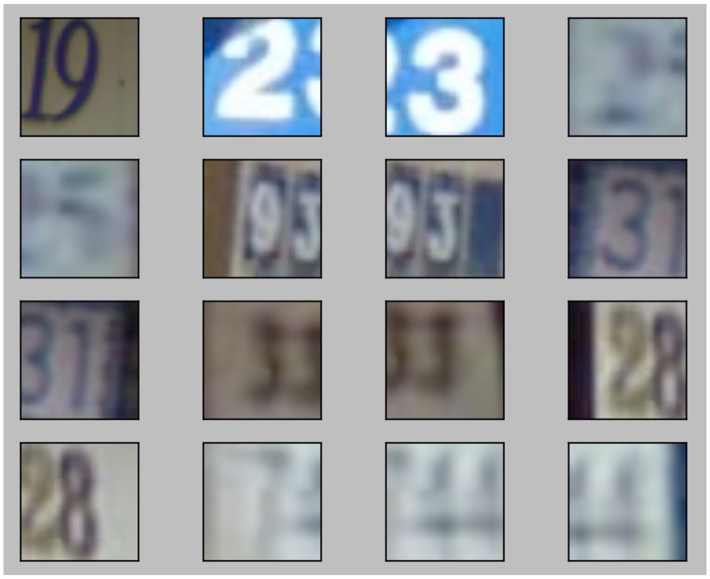


***Fig. 2**: Sample of input images (SVHN dataset)*

[7] http://ufldl.stanford.edu/housenumbers/

Loading the .mat files creates 2 variables: X, which is a 4-D matrix containing the images, and y, which is a vector of class labels. To access the images, X(:,:,:,i) gives the i-th 32-by-32 RGB image, with class label y(i).[8]

For a non-neural-network method, such as the SVM initially implemented, the image-reshaping done in the previous dataset also applies here. The final matrix inserted into the classifier has a shape of n_sample by 1024 (=32pixels * 32 pixels), normalized by 255 in order to obtain values in a range between 0 and 1.

For a neural network algorithm, instead, the input data has been reshaped in the following format:
(n_samples, pixels, pixels, #channels), which in the training set case is (73257, 32, 32, 3).

On the other hand, in both cases the labels are transformed from integers into vectors, where the index of the label is set to '1' and all other entries to '0'. Although prior to the one-hot-encoding I transformed the value 10 into digit 0, to be consistent with prior evaluations.

II - 2.1 Exploratory Visualization (MNIST Dataset)

As mentioned, the label values range from 0 to 9. Plotting the occurrences of each label included in the training set, shows how uniformly distributed they are. This is useful in predicting how balanced the classes will be.

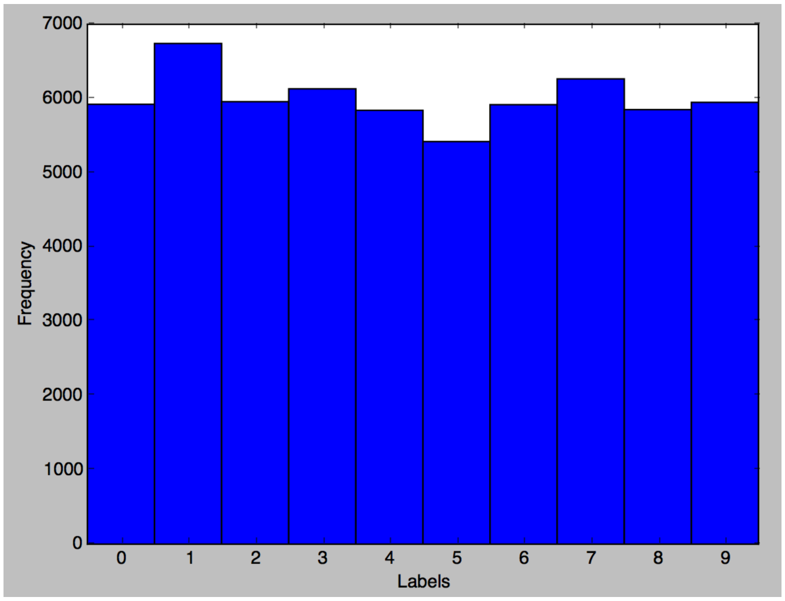


***Fig. 3**: Number of occurrences for labels in training set (MNIST)*

[8] http://ufldl.stanford.edu/housenumbers/

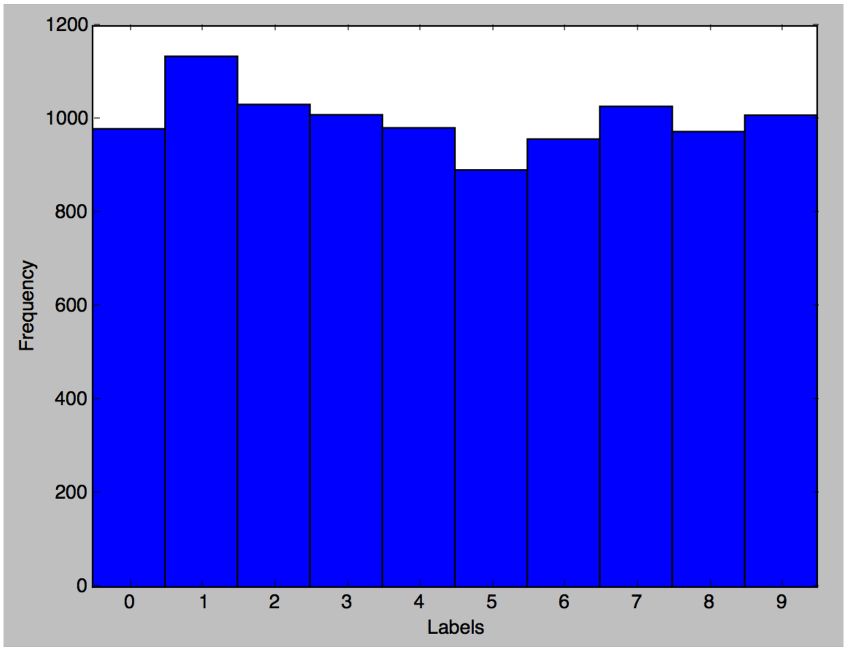


Fig. 4*: Number of occurrences for labels in test set (MNIST)*

II - 2.2 Exploratory Visualization (SVHN Dataset)

As in the previous dataset, I have plotted the number of occurrences for the labels, in order to find out how they are distributed, and therefore balanced.

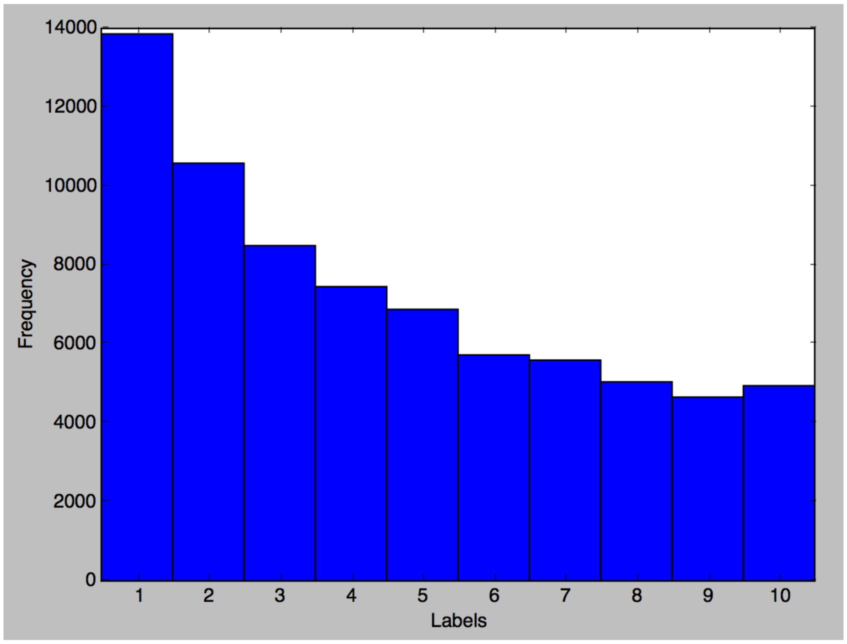


Fig. 5*: Number of occurrences for labels in training set (SVHN)*

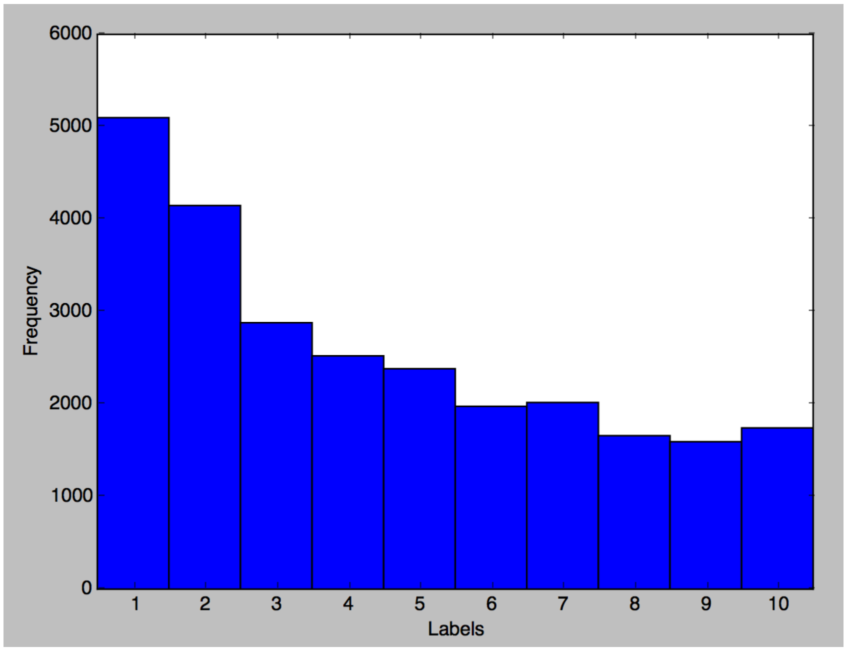


Fig. 6*: Number of occurrences for labels in test set (SVHN)*

The two figures above represent the histograms for the training and test sets, which show clearly a distribution less uniform than the previous dataset.
The skewed distribution is similar in both sets (training and test), therefore I don't foresee any influence in the analysis and final results.

The input data, as previously explained and also seen on *figure 2,* is represented by several variables (color, brightness, resolution) and one of the techniques used to reduce the impact on the final results is to transform the images into a grey scale format.
Additionally, I have implemented several filters in order to highlight relevant features, such as edge detection, which can help the classifier obtain better outcomes.
A thorough visualization and explanation of these methodologies will be discussed further in the Section III.

II - 3.1 Algorithms and Techniques (MNIST Dataset)

The MNIST dataset has been processed using two algorithms:

- Support Vector Machines (SVM)
- Multi-layer Perceptron (MLP)

1- Support Vector Machines

The SVM is one of the most known algorithms and its performance is quite impressive.
The reason why I chose this one versus other common models is the fact that it works well for moderately small datasets, which don't carry a lot of noise.

The following parameters can be tuned to optimize the classifier:

- Kernel (it can take different "values" such as *linear*, *poly*, *ref, ..*).
- C value (it controls the tradeoff between a smooth decision boundary and classifying the points correctly. A large value of C favors a more intricate decision boundary (getting therefore more training points correctly) versus a smoother separator).
- Gamma (defines how much influence a single training example can have. The larger the value of gamma, then the closer other examples must be to be affected).

I have used the "grid_search" function[9] from "sklearn", which considers all parameter combinations among the ones that I have provided:

parameters = {'kernel': ('linear', 'rbf'), 'C': [1,10,100, 1000], 'gamma': [0.001, 0.0001]}

The performance of the algorithm improves while refining these parameters:

Accuracy of test set without grid-search: 94.46%
Accuracy of test set with grid-search: 97.39%

Further implementation could have overtaken (i.e. evaluating the F1-score and optimizing for that), but the goal of this project was to analyze performance among different algorithms and with two different datasets (MNIST and SVHN).

2- Multi-layer Perceptron (MLP)

The second technique I used to analyze the MNIST dataset was a neural network architecture, which consists of sigmoid neurons going from an input layer to the output by passing through a series of hidden layers.

The input layer consists of the preprocessed image data as previously discussed, while the output layer contains a single neuron which classifies a value evaluating it against a threshold (which is 0.5 for the sigmoid function here used).

The output from one layer is the input to the next layer, as a typical characteristic of a "feedforward" network (meaning there is no information fed back, which creates a feedback loop, as in a "recurrent neural network" model).

MLP uses different learning techniques, one of which is back-propagation. The output values are compared with the correct label, to compute the value of the error function, such as the loss function. The error is fed back through the network (back-propagation) and the algorithm, during each iteration, adjusts the weights in order to reduce the error.

I have used the stochastic gradient descent as a method to adjust the weights, because it is one of the parameters that can be chosen in the MLP classifier.

Other critical components that can be tuned are the number of iterations, the learning rate and the number of hidden layers.

[9] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV

Interestingly, we can plot the weight to have a rough idea of how the learning behavior is going, although it's very difficult to discern it. Mainly, if the weight representation looks unstructured, it can give us a hint that the learning rate might be too high. Below is a representation of the first layer of weights for a network with low learning rate, but with short iteration cycle:

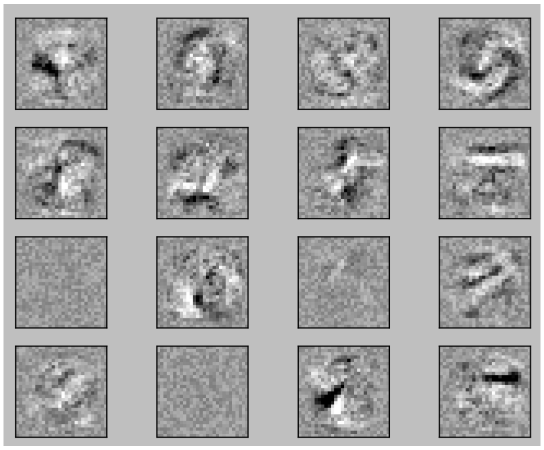


Fig. *7: First layer weights with learning rate = 0.1; hidden_layer=50; iteration=10*

The iteration cycle was indeed too low, as the algorithm did not converge. Increasing the iterations helped reach convergency, providing a fair accuracy on the test set equal to 96.47%. I tried to fine-tune the other parameters as well. I increased the number of hidden layers and I obtained a slight improvement in the performance. On the other hand, when I increased the learning rate to a value of 3.0 the accuracy dropped, proving the fact that the value was too high and some of the weights were not even being used.
Finally the best results I have obtained were with a network consisting of 2 hidden layers with 200 neurons, a learning rate of 0.1 and 2000 iterations:

Accuracy of test set without grid-search: 99.99%
Accuracy of test set with grid-search: 98.11%

Clearly this is a good result, although further fine-tuning might be necessary as the model seams to overfit the training set.
Ultimately, though, increasing excessively the number of layers proved to be counterproductive and did not fix the overfitting.
The current (2013) record is classifying 9,979 of 10,000 images correctly. This was done by Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus.[10]

The following step could be to use a Convolutional Neural Network for the MNIST dataset, but I would like instead to test the MLP algorithm here optimized to the following SVHN dataset.

[10] http://neuralnetworksanddeeplearning.com/chap1.html

Furthermore I am very content with the performance that this simple neural network model has proved, but I know that a more complex dataset will have more difficulties in holding this targets and therefore we will need a more robust architecture to deal with it.

II - 3.2 Algorithms and Techniques (SVHN Dataset)

Training and using the algorithm used for the MNIST dataset revealed to perform extremely poor on the real images data set (a disappointing 7.7% accuracy on the test set).
So, I used a two form approach:

- apply convolutions prior to feeding the input to the MLP classifier
- use deep learning convolutional neural network

<u>1- Image convolutions and MLP classifier</u>

Convolutions are basically "filters" that we can apply on an image in order to "represent" certain "attributes".

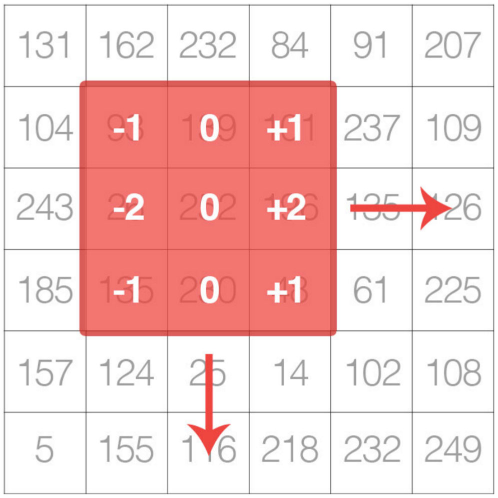


Fig. 8*: A kernel on top of an image matrix (Source: PyImageSearch Gurus)*[11]

Mathematically, it basically consists of an element-by-element multiplication of two matrices and a final addition.
The filter (kernel) has smaller dimensions than the multi-dimensional matrix of the image and it moves through the whole matrix applying this mathematical operation at each coordinate of the image matrix.

The OpenCV library offers a method that allows us to simply apply a desired kernel to an image and perform the convolutions.

[11] https://www.pyimagesearch.com/pyimagesearch-gurus/?src=post-convolutions

The input to the OpenCV function is an image transformed into greyscale.
I have used different kernels, such as “Laplacian” (used as an edge detector), “sharpen” (used to enhance line structures and other details of an image) , SobelX” (used to detect vertical changes in the gradient of the image) , SobelY” (used to detect horizontal changes in the gradient).
After independently applying these filters I fed the data to the MLP classifier previously described.
After few fine-tuning of the parameters on the MLP model, the best results were obtained with the “SobelY” kernel:

Accuracy of test set without grid-search: 92.86%
Accuracy of test set with grid-search: 75.10%

Again, this is not the best performance we can obtain (more time could be spent on tweaking the parameters of the model), but the goal of this paper is to experiment several techniques and evaluate how they compare.
On the other hand, I could dramatically improve the accuracy from a situation where the input data was not pre-processed but simply fed to the algorithm, as previously mentioned at the beginning of this chapter.

2 - Deep learning Convolutional Neural Network

The goal is to create a simple deep neural network architecture in order to see how it performs compared to the previous methodology.
The model is built with TensorFlow, an open source library for numerical computation, using data flow graphs.
The inputs to the network are similar to the previous model, with the exception that in this case TensorFlow is considering also the number of channels, so the data’s new shape is in the form of (73257, 32, 32, 3).
Furthermore, prior to fit the model I have split the initial training set into a training portion and a validation set.

The network includes:

- Convolutional input layer, which learns 16 convolution filters of sizes 5×5. The initial shape is the same as the one of the input images, with height and width of 32 and depth (number channels) of 3.
- Max Pool layer with size 2×2. (Which is a 2 x 2 sliding window that “slides” across the image).
- Convolutional layer with the same characteristics as before. (Note that since this is not the first layer of the network, I don’t need to specify the input shape).
- Max Pool layer with size 2×2. (Which is a 2 x 2 sliding window that “slides” across the image).
- Flatten layer, which takes the output of the preceding MaxPooling2D layer and flattens it into a single vector, while applying dense/fully-connected layers.
- Fully connected layer with 128 units and a rectifier activation function. (A type of layer where every node in the preceding layer connects to every node in the next layer).
- Dropout set 93%.

- Output layer with 10 units and a softmax activation function. The softmax classifier (multinomial logistic regression) will return a list of probabilities, one for each of the 10 class labels.

After creating the network, the model gets compiled using Adam Optimizer as an optimizer, then fitted to the training set and validated against the validation data.
Finally all the training weights get saved, for allowing faster retrieval when the loaded model gets evaluated on the test data.

II - 4 Benchmark

As discussed in the "Metrics" section, I used the accuracy as a measure of performance, particularly in regards to the MNIST dataset.
For this dataset the highest record that I was able to recover was done by Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus, classifying with a 99.79% accuracy.
My goal was to reach at least 95% with a SVM classifier and above 97% for a MLP model.

In regards to the SVHN dataset, I also considered the cross-entropy error, although I ultimately compared the performances against a classification error.
I did not focus on processing time, as I only used the CPU power and I knew that the performances would have not been good, or at least not competitive.

III. Methodology

III - 1.1 Data Preprocessing (MNIST dataset)

After the images were downloaded, I transformed them from a set of images in 28x28 format to a matrix of n_samples (60,000 in the case of the training set) by 784 (= 28pixels * 28 pixels) features.
Furthermore, I normalized them by 255 in order to obtain values in a range between 0 and 1.

The training and testing labels need to be transformed into vectors, generating a vector for each label, where the index of the label is set to `1` and all other entries to `0` (also known as "one-hot encoding").

I have not divided the data into a training set and a validation set, as I wanted to experiment and dedicate more attention to the more complex dataset represented by the Google Street View House Numbers.

III - 1.1 Data Preprocessing (SVHN dataset)

Loading the .mat files creates 2 variables: X which is a 4-D matrix containing the images, and y which is a vector of class labels. To access the images, X(:,:,:,i) gives the i-th 32-by-32 RGB image, with class label y(i).[12]

For a non-neural-network method, such as the SVM initially implemented, the images' reshaping done in the previous dataset applies also here. Furthermore the images are converted into greyscale. The final matrix inserted into the classifier has a shape of n_sample by 1024 (=32pixels * 32 pixels), normalized by 255 in order to obtain values in a range between 0 and 1.

For a neural network algorithm, instead, the input data has been reshaped in the following format:
(n_samples, #channels, pixels, pixels), which in the training set case is (73257, 3, 32, 32).
Additionally, the input set is split and the loss function is calculated against the training and validation data (validation test size = 33%).
Furthermore I normalized the input data with "Min-Max scaling" to a range of [0.1, 0.9] in order to make it easier for the optimizer to perform.

On the other hand, labels are transformed from integers into vectors, where the index of the label is set to '1' and all other entries to '0'.

III - 2.1 Implementation (MNIST dataset)

<u>SVM Algorithm:</u>
After the preprocessing of the input data, as previously discussed, the implementation process is carried out in two steps:

- training the classifier to fit it to the training dataset
- predicting the results of the testing dataset

The first step is further refined identifying and fine tuning certain parameters of the SVM classifier.
In particular, the "kernel", the "C" value and the "gamma".
I chose different values for each parameter and I used "grid-search" to systematically work through combinations of parameters tunes, cross-validating as it goes to determine which tune gives the best performance.
The prediction was carried on the testing set and evaluated against the accuracy metric. The process was repeated until a result was deemed acceptable.

<u>MLP Algorithm:</u>
A similar process as earlier is applicable to this algorithm, where the classifier is trained against the training set and then evaluated against the test dataset.
The accuracy of the test set is the metric I aimed to improve, but I also printed out the loss function for each iteration, in order to have an idea about the "quality" of the network.

[12] http://ufldl.stanford.edu/housenumbers/

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05529723
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
```

The process was iterated several times as I was selecting different values for the number of iterations, the number of hidden layers and the learning rate.
I started with a small number of iterations and a small number/size of hidden layers and I noticed that the algorithm was not converging to a solution. Improving the number of iterations to a 2000 (a good compromise also in terms of hardware speed performance) brought a convergence but not a great accuracy value.
The learning rate was the parameters that needed more critical fine tuning.
Learning rate is used in back propagation to update the value of the weights; a high value might cause the network to diverge.
Low value of learning rate was taking longer to converge and required many more iterations.

III - 2.2 Implementation (SVHN dataset)

Image convolutions and MLP classifier:
With this algorithm I define different kernels (filters), which are going to be applied to the original image.
Each kernel is essentially a small matrix that overlaps on top of the original image matrix and modifies it while moving from left to right, and from top to bottom.
Before this transformation the image gets modified into gray scale.
This process gets applied to all the images in the training and test set and then fed to an MLP classifier as previously described, where fine-tuning of the parameters was necessary.
Each filter brings different results, as each one is focusing on different aspects of image transformation.
Here below an example of an image after applying the "Laplacian" kernel:

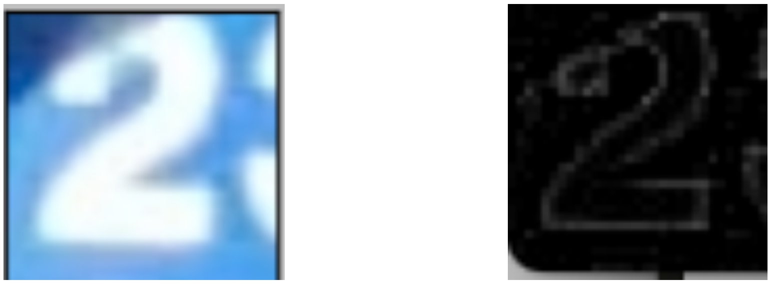


Fig. 9*: Original image (left); "Laplacian" transformation (right)*

The reason why I implemented these filters prior to feeding the data to an MLP classifier was to get rid of some noise in the images and try to highlight features (such as "edge detection" in the case of the "Laplacian" kernel) that would have helped increase the accuracy of the classifier.

Deep learning Convolutional Neural Network Algorithm:

The data fed to this algorithm is split into a training set and a validation set.
Then, using the TensorFlow library, I create a network, which consists of different layers acting on the input data in sub sequential order.
Furthermore, the model is compiled and fit to the training set and finally evaluated against the test set.
Creating a network was not a difficult step, but optimizing it in order to make the machine learn has been quite a intricate task.
Additionally, the computation requires a bit of time for the computer to process as I was only relying on a single CPU power.
A good solution was to save the model prior to evaluating it against test data. That saved a lot of time as I didn't need to train the network every time.

An important factor to take into consideration while monitoring the results of each epoch is the loss function, which gives an idea on how the error gets improved.
In the first models that I created, the loss function remained steady after each epoch. That raised a flag that something wasn't properly working the way that I wanted.
Only focusing the attention on the accuracy for the training set doesn't give a whole picture on the network's performance.

The goal is to create a simple deep neural network architecture in order to see how it performs compared to the previous methodology.
The model that I built consists of Convolutions, Rectified Linear Units (RELUs), MaxPooling and Dropout.
I am going to provide an explanation of those terms before describing the specific of my network.

- Convolution: this is similar to what I explained before related to kernels. It is in fact the application of a filter on top of the original image, in order to detect patterns. The filter is characterized by a height, width and depth and the final result is an image that has been transformed.
- RELU: this takes a logistic classifier and makes it non-linear. It's a type of activation function that "turns off" all the negative weights transforming them to zero and keeping only the positive ones.
- MaxPooling: there are different types of polling (such as max, mean, etc) and in this case I used the "max" type, which is the most common. At every point on the feature map it looks at the small neigh borough (defined by the user) around that point and it computes the maximum of all the responses around it. This layer downsizes the data but it retains the maximum of the information.
- Dropout: this layer is primarily used to avoid over-fitting, dropping, randomly, some units.

The connectivity pattern between the neurons in a Convolutional Neural Network[13] is inspired by how the animal visual cortex works.
Individual cortical neurons respond to stimuli in a restricted region of space known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the visual field. The response of an individual neuron to stimuli within its receptive field can be approximated mathematically by a convolution operation.
The network that I used in this project includes:

- Convolutional input layer, which learns 16 convolution filters of sizes 5×5. The initial shape is the same as the one of the input images, with height and width of 32 and depth (number channels) of 3.
- Max Pool layer with size 2×2. (Which is a 2 x 2 sliding window that "slides" across the image).
- Convolutional layer with the same characteristics as before. (Note that since this is not the first layer of the network, I don't need to specify the input shape).
- Max Pool layer with size 2×2. (Which is a 2 x 2 sliding window that "slides" across the image).
- Flatten layer, which takes the output of the preceding MaxPooling2D layer and flattens it into a single vector, while applying dense/fully-connected layers.
- Fully connected layer with 128 units and a rectifier activation function. (A type of layer where every node in the preceding layer connects to every node in the next layer).
- Dropout set 93%.
- Output layer with 10 units and a Softmax activation function. The Softmax classifier (multinomial logistic regression) will return a list of probabilities, one for each of the 10 class labels.

III - 3.1 Refinement (MNIST dataset)

For the MNIST dataset the SVM algorithm revealed to be a good and "fast" model to get good results.
I started off with the standard settings offered by the "sklearn" toolkit, but then I modified the parameter of "kernel", "C" and "gamma" using the "grid-search" function. That allowed to refine the parameters, and to obtain more competitive results.
The initial accuracy obtained on the test set for the "untouched" parameters was around 94%, while the later adjustments helped increase it to 97.34%.

The MLP classifier, on the other hand, required more careful and long refinements.
I started off with a network consisting of one hidden layer of a size of 50 neurons, a learning rate of 0.001 and 200 iterations. This set up unfortunately brought no results as the algorithm failed to converge. It was clear that the number of iterations was too low. Improving just that (up to 2000) helped achieve 96% of test accuracy.

[13] https://en.wikipedia.org/wiki/Convolutional_neural_network

Then I started to modify the hidden layer's size and number, finding the best results for a network of 2 layers with 200 neurons. This was a good compromise in terms of processing time as well. (which totaled approximately a minute).
The adjustments on the learning rate were more difficult and required more trials and errors. Finally I settled to a 1.0 value for this parameter and I achieved 98% for the test accuracy.

III - 3.2 Refinement (SVHN dataset)

For the SVHN dataset the use of the MLP classifier optimized for the MNIST dataset with no image processing gave me a 7.7% accuracy for the test set (with almost similar value for the training set).
It was clear that the noise present in the real world images was influencing a lot the model.
That is why I implemented different filters, such as:

- "Sharpen": This kernel is used to enhance line structures and other details of an image. The following matrix is applied on top of the image:

  [0, -1, 0]
  [-1, 5, -1]
  [0, -1, 0]

- "SobelX": It is used to detect vertical changes in the gradient of the image. We can clearly see how its matrix structure will modify the image matrix when overlapping with it, as it creates a vertical "gap" allowing to highlight that edge:

  [-1, 0, 1]
  [-2, 0, 2]
  [-1, 0, 1]

- "SobelY": It is similar to the above filter, with the exception that instead of detecting vertical lines it identifies horizontal ones. The matrix I used consists of:

  [-1, -2, -1]
  [0, 0, 0]
  [1, 2, 1]

- "Laplacian": This filter is used to detect edges in images. That matrix is computed as follows:

  [0, 1, 0]
  [1, -4, 1]
  [0, 1, 0]

Most of them (except for the "Sharpening" kernel which gave a 23% test accuracy) improved the results to a value around 70%.

After few fine-tuning of the parameters on the MLP model, the best results were obtained with the “SobelY” kernel (test accuracy of 75%) and with a decrease in the learning rate (0.1 value).

This exercise gave me a good indication on how a Convolutional Neural Network would have helped achieve even higher results.
The first models that I created (with a simple Convolution and Activation layer) proved to give very poor results (around 20% training accuracy).
I tried to lower the learning rate to a very small number (0.000001) but the results were not improving and the loss was steady around 2.5, signaling that the network was not learning.
After careful consideration of the parameters that I used I realized that the initial weight were set as:

weights = { 'layer_1': tf.Variable(tf.truncated_normal([5, 5, 3, layer_width[‘layer_1’]]))

where the default value for the standard deviation in the “truncated_normal” function is set to 1.0.
The standard deviation value determines the order of magnitude of the output at the initial point of the optimization. So setting up large (relatively speaking) values of the weights, as stdev of 1.0 does, caused the error to explode.
Changing the Stdev value to 0.1 improved the performance and made the network reach:

Validation accuracy: 95.1 %
Testing accuracy: 87.5 %

Lowering the value of the Stdev introduces also a form of additional learning for the machine as in the “truncated_normal” function the weights beyond the 2 Stdev are dropped and re-picked.
I trained the network for 10,000 steps as I realized that lower iterations were giving poorer results.

IV. Results

IV - 1.1 Model Evaluation and Validation (MNIST)

The final model applied to the MNIST dataset (MLP classifier) brings acceptable results (98% test accuracy), very close to human accuracy. On the other hand, the slightly tweaked SVM algorithm can reach similar results.
I was able to reach 97.39% after a few adjustments on the parameters. This model also seemed to be a better fit to the training data (94.46% versus an overfitting 99.99% in the MLP model).
Good results on the MLP classifier were achieved with a network of only one hidden layer of 100 neurons, but my final model of two hidden layers with 200 neurons each was computationally faster and provided (even slightly) better results.
The mathematical intuition regarding the performances when compared to a single layer, is that each layer in a feed-forward multi-layer perceptron adds its own level of non-linearity that cannot be contained in a single layer. Each layer's inputs are only linearly combined, and hence cannot produce the non-linearity that can be seen through multiple layers.

Both the MLP and SVM algorithms are not complicated and this can show how a well "manufactured" training dataset can count more that the complexity of the model.

IV - 1.2 Model Evaluation and Validation (SVHN)

A CNN model performs much better than the previous algorithms used even with a not so complicated network such as the one I used.

Although I realized that it takes a bit of practice and trial-error to optimize the network, tweaking some of its parameters.

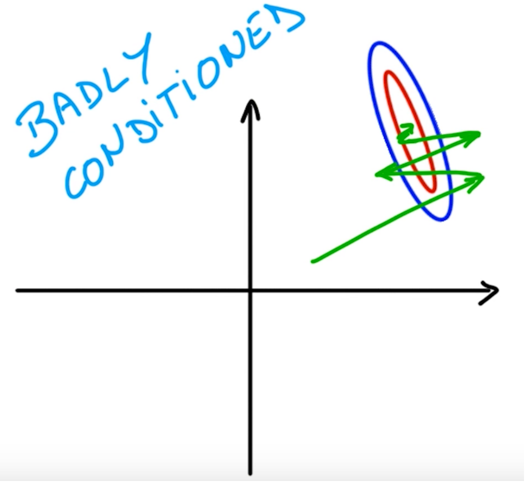
As previously highlighted the value of the initial weights played a critical step in the optimization cycle.

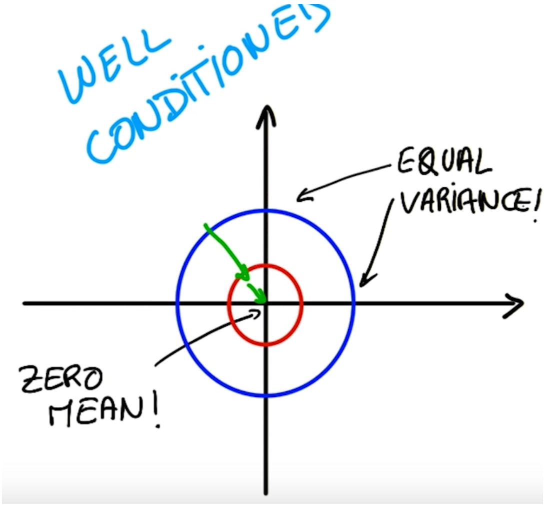
Setting the initial weights to zero is not a good practice as the network might get stuck in a local minima. So it's good to have them set to a random value (as "truncated_normal" function does).

On the other hand if the weights are too big then the back-propagated gradient can amplify too quickly. A good recommendation is to set the standard deviation to 1/sqrt(N), where N is the number of inputs to the given neuron layer.

Also, the pre-processing of the input data plays a crucial role as in a badly conditioned problem the optimizer has to do a lot of searching before finding a good solution.

A good conditioned problem (such as with zero mean and equal variance) , instead, allows the optimizer to reach a solution faster.[14]





Furthermore, perturbation on the training set, such as image orientation can affect results.

Another parameter that I evaluated was the type of optimizer. I initially used the GradientDescent one but, with all the rest of the hyper parameters set the same, the results were poorer compared to the Adam optimizer. (75% test accuracy with SGD vs 87.5% with Adam).

[14] https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/91cc6685-08df-4277-b53d-3a792b02420d/concepts/71191606550923

As mentioned earlier when analyzing the results of the algorithm, the accuracy does not give a whole picture and sometimes might mislead us to superficial conclusions.
Visualizing the certainty of its predictions (using “tf.nn.top_k”) proved very helpful.
In this function we can set the parameter “k” giving us the opportunity to display the Softmax probabilities for each value within a range of k. In addition we can plot the predicted indexes together with their certainty of predictions.
This can give us a certain confidence on how the results are predicted.
The results of the first 10 test images is as follows:

```
[[ 4.88502603e-14  1.46703275e-11  4.52406125e-13  4.61449064e-02
   2.56273031e-10  9.53795552e-01  1.45993890e-05  2.01631951e-06
   4.29712527e-05  1.94639105e-09]
 [ 1.37136567e-06  6.30263855e-07  9.99991417e-01  6.00717885e-06
   2.73057328e-08  7.94036481e-08  4.88468599e-10  5.17070191e-07
   3.08789994e-09  9.46174111e-11]
 [ 1.32689883e-11  9.99999881e-01  2.63460032e-10  9.17061982e-08
   5.66733682e-09  4.41993207e-18  2.09647614e-21  6.09616796e-20
   6.68191618e-18  5.45564893e-22]
 [ 9.06285763e-01  3.14093078e-04  8.59873296e-07  9.27031692e-03
   4.62807659e-09  7.89303519e-03  5.74539676e-02  2.86825924e-10
   1.87504105e-02  3.16723090e-05]
 [ 1.48337795e-05  1.69260369e-03  2.03186912e-09  4.90534221e-05
   2.58913333e-12  8.74037312e-07  4.31830085e-06  9.98238325e-01
   5.70645420e-10  5.96785251e-12]
 [ 8.33535125e-08  9.89628196e-01  1.73912529e-08  1.28574993e-06
   1.03660030e-02  2.75127685e-12  1.76393578e-09  9.36922415e-11
   4.32583283e-06  3.29213913e-11]
 [ 3.66506174e-06  2.17759379e-07  4.25199833e-05  7.86486271e-05
   1.73301451e-05  3.28055321e-04  1.93458982e-05  3.26139329e-08
   2.01671101e-05  9.99490023e-01]
 [ 7.37197412e-13  1.00000000e+00  3.61675208e-14  5.11577068e-15
   2.52129695e-09  5.36611709e-20  1.69876591e-14  1.07084040e-14
   1.56763151e-15  6.67574286e-13]
 [ 1.39159178e-14  9.99235034e-01  2.01694064e-07  7.60406605e-04
   8.89006924e-10  3.14241682e-08  7.46018127e-16  4.31629724e-06
   1.77281990e-13  3.96824961e-13]
 [ 5.80728002e-24  5.22257081e-27  4.49288421e-22  1.13623554e-13
   4.99700196e-20  3.97450482e-18  2.22912689e-07  2.06905909e-27
   9.99999762e-01  2.70204565e-18]]
```

***Fig. 10**: Softmax probabilities (highlighted in yellow the highest value)*

```
[[5 3 6 8 7 9 4 2 0 1]
 [2 3 5 1 0 8 7 6 4 9]
 [1 4 0 3 2 7 8 5 9 6]
 [0 6 8 5 3 1 9 2 4 7]
 [7 1 3 6 5 0 2 8 9 4]
 [1 4 0 3 8 6 2 5 9 7]
 [9 5 4 2 6 3 8 0 1 7]
 [1 4 9 2 0 3 7 6 8 5]
 [1 3 7 2 5 4 9 8 0 6]
[8 6 3 5 9 4 2 0 1 7]]
```

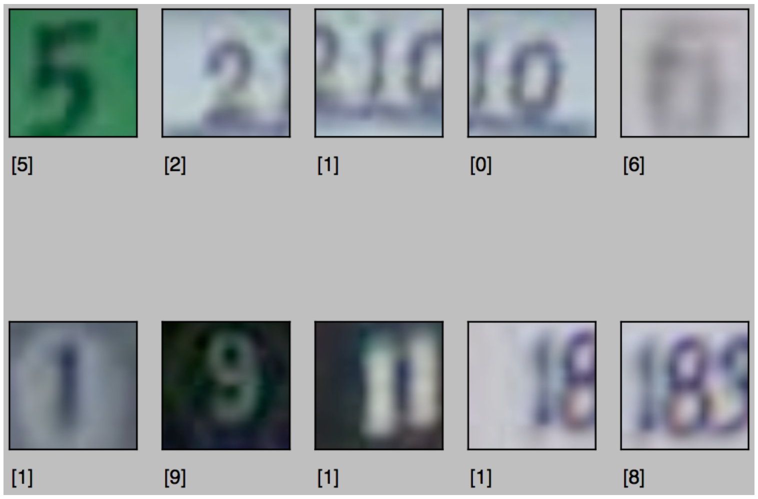


***Fig. 11**: Predicted indices (in order of prediction) (left) vs images and real labels (right)*

```
[[ 14.56169319   7.39169979  -2.11701798  -6.58102894  -8.58360195
   -9.96859837 -14.71340656 -18.65824127 -21.59887314 -24.21249962]
 [ 14.18896675   1.57955813   0.90217799  -3.68015075  -3.71388626
   -3.73167419  -5.46312094  -5.62938786  -9.34714699 -10.42057133]
 [ 23.8605423    3.6127913    3.02724457   3.00351119  -8.87272644
  -15.70703793 -15.85557842 -16.99021339 -17.04851532 -20.33983612]
 [  5.2341423    2.54879737   0.41651297   0.02968544  -0.39148068
   -2.09054828  -5.3107605   -8.54315662 -10.3772192  -16.40671539]
 [  7.13419008   2.12594891  -2.76554751  -3.08811903  -5.03172398
   -5.20023489 -12.3131094  -13.89796352 -15.26828003 -16.08544731]
 [ 15.38036346  10.07276535  -0.29561543  -0.51133513  -2.16351366
   -3.28330421  -9.06376743 -11.61707783 -11.91309738 -12.23480988]
 [  7.8851552    0.866198    -2.28319311  -2.56718278  -2.93514395
   -3.39669156  -3.61748409  -6.72494936  -8.2647028  -11.6579771 ]
 [ 22.11558342   3.95925355  -3.52756405  -7.43588161  -8.32647133
   -8.49642086  -9.87571621 -11.58692455 -12.50385666 -19.57128143]
 [ 16.28534317   8.4971447    1.57733786   1.05957949  -2.74318767
   -4.54402351 -11.75723839 -12.85168648 -14.83203697 -19.36266518]
 [ 26.28301239  10.52411842   2.66665173  -7.56832886 -10.15791607
  -11.2023468  -17.78027153 -20.72381401 -20.83101273 -26.02116966]]
```

***Fig. 12**: Certainty of its predictions*

Justification

The benchmarks for the MNIST dataset are higher than the results that I have achieved with the algorithms applied.

On the other hand, my goal was to see how a well performing model for the MNIST dataset could behave in a more diversified training set such as the SVHN data.

While simple non-neural network algorithms can provide a quick, acceptable solution to a well trained problem, they do not hold in a more complex scenario.

Neural Network reveals to be the most promising way to deal with this type of data.

The use of feed forward flow through different type of filters in the built network and the back propagation methodology require more processing time, but ultimately tangible results.

The final architecture used is the one described in Section II.

V. Conclusion

This project improved my understanding of different algorithms.

Furthermore it proved that a model performing great on a well-defined dataset does not necessarily replicate the same results on a more complex training sample.

Additionally, the importance of having a good dataset and a well-refined preprocessing technique play a crucial part in training an algorithm.

For example a simple “Laplacian” filter applied to the original image of the SVHN data can clearly visualize some characteristics that the model can quickly pick up, without getting disturbed by other noise.

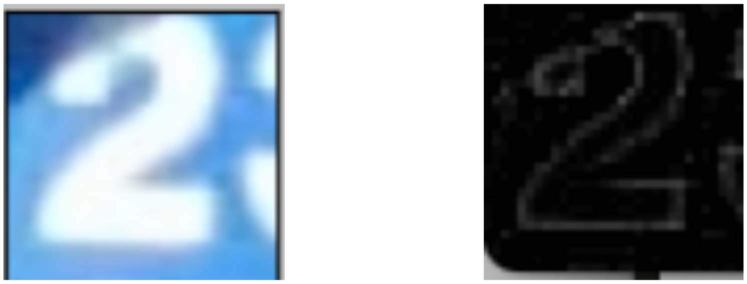


Fig. 13*: Original image (left); “Laplacian” transformation (right)*

I believe that further refinement could be done on images that display the digits in abnormal orientations.

Furthermore dimensionality reduction through PCA could help discard the pixels that are not necessary in predicting the digits from an image.

Improvements can be done using GPU power, instead of solely relying on CPU. This for me was the biggest constraint, as the training time was very long.

In fact increasing the number of training steps will help achieve higher results. I was able to implement 10,000 steps but relying only on CPU power did not allow me to test longer.