



SVHN Sequence Detector

Machine Learning Nanodegree Capstone Project

Summer 2016

Sergio Gordillo
@Sergio_Gordillo

github.com/archelogos/sequence-detector

<https://sequence-detector.herokuapp.com/>

Definition

Project Overview

[Deep Learning](#) is currently one of the most interesting fields in Machine Learning. The combination of [Neural Networks](#) and powerful computation systems is extremely useful to solve complicated [Pattern Recognition](#) or Text Classification problems.

The main goal of this project is, thanks to Deep Learning techniques, to be able to detect and identify sequences of digits in a random picture. Specifically in this project, the images correspond to houses and the number sequences correspond to their house numbers.

The desired and summarized result is shown in the following picture.

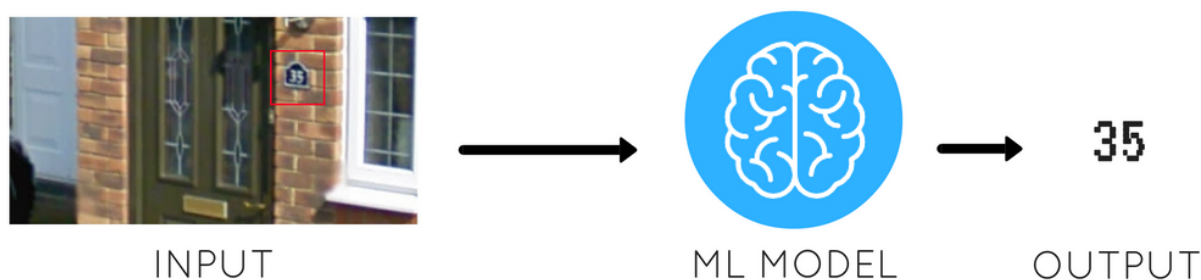


Figure 1: Project goal overview

The data used to train the model is part of the [SVHN dataset](#), which is a real-world image dataset for developing machine learning and object recognition problems and it is obtained from house numbers in Google Street View images.

This is a real-world problem studied for many years. It wasn't until the application of neural networks to the images recognition problems that it was considered solved. At this moment real products and apps that are focused on solve this problem feature Deep Learning (i.e. <http://questvisual.com/>).

Problem Statement

As it was said, the main goal is to define, build and train a stable and consistent ML Model which can identify sequences of house numbers in a particular image.

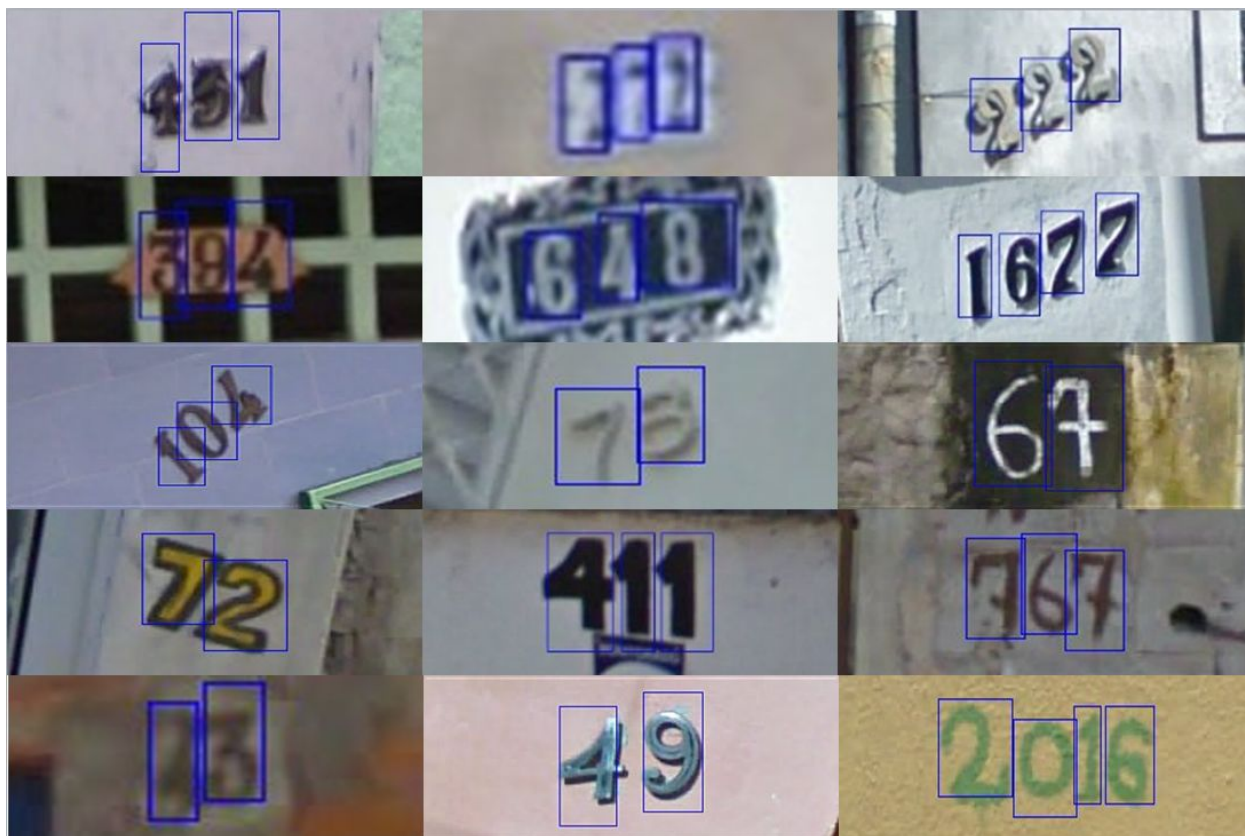


Figure 2: Samples from the dataset

To reach this goal, the mentioned dataset was used to create a ML model based on a Neural Network. The final model was built taking different ideas from investigation papers, tutorials and examples.

Initially, the model was as basic as possible, based on a simple [Logistic Regression](#). Throughout the project, the model got more complicated and sophisticated step by step, becoming a Multilayer [ConvNet](#) production-ready model by the end of the project.

In that process, different techniques, algorithms and solutions were tried and properly discussed.

The project is divided in 7 steps:

1. Familiarize with [TensorFlow](#), applying from a Logistic Regression to a Multilayer ConvNet model to a **single-MNIST-digit** dataset.
2. Modify the previous model and apply it to a **single-SVHN-digit** dataset.
3. Improve the model and apply it to a **MNIST-sequences** dataset.
4. Make some modifications to the previous model and apply it to a **SVHN-sequences** dataset.
5. Define and evaluate a stable and consistent final model.

6. Train the final model in [Google Cloud Platform](#) in order to boost its performance.
7. Build and deploy a web app which demonstrates how the model works.

The human ability to correctly identify the numbers from the SVHN dataset is approximately 98% and the most important solutions to this problem given by companies like Google have almost reached that goal.

Convolutional Neural Networks aren't extremely efficient and they are not easy to train properly because they are expensive in terms of computation cost. For that reason the first steps of the project were developed in ipython notebooks, running small training stages in each session. Once it was clear that the model worked fine, it was moved to the Public Cloud (GCP) to train it in a better way (high-cpu machines running efficient python scripts).

It's important to notice that the purpose of this project is not to improve the best solution for this problem but to study and build a **reliable and production-ready** model which can perfectly obtain a **90+% of accuracy** identifying sequences from SVHN pictures (which in the field of Neural Networks is a huge difference with the mentioned 98%).

Metrics

As it was said, the accuracy is the main metric to be measured in this problem.

It can be defined as the probability of coincidence between the real number in the picture and the number predicted by the model.

A prediction is considered correct if all the digits in the sequence and the length of the sequence are exactly the same, in other cases the prediction will be considered incorrect.

To check that the model is working properly, and the solution fits the problem, a group of periodic checks were set on the **training** and **validation** dataset along the training.

At the end of each training session it is always estimated the current accuracy of the model on the entire **test** dataset.

```
## Pseudo python
def accuracy(labels, predictions):
    >> n_accuracy = summation(predictions == labels) / length(labels)
    >> p_accuracy = 100 * n_accuracy
    >> return p_accuracy
```

Figure 3: Pseudocode of the accuracy function

Analysis

Data Exploration

As it was explained, the project itself and the main objective of it are focused on the SVHN dataset. However, there are other interesting datasets that can be used in the first steps of the project in order to be more practical studying the model. That way, the preprocessing function can be delayed to the end.

MNIST is one of these datasets and it has approximately 70.000 examples of **handwritten** digits. The size of the representation of each number is 28 by 28 pixels. Each digit has 784 features, (28 by 28) and 1 channel depth corresponding to a grayscale picture (28x28x1).

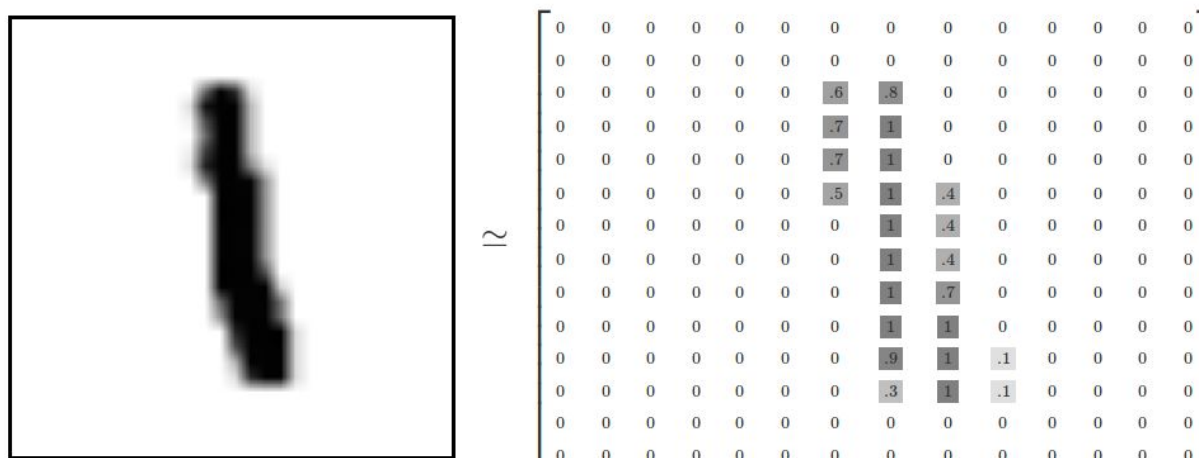


Figure 4. MNIST Digit representation

This data is downloaded from the [Yann LeCun Website](#) and after that, the files are properly extracted and reformatted creating different datasets (training, validation and test).

The labels are not downloaded in [One-Hot](#) encoding so they are the value of the number itself and not other type of numerical representation.

Once the model is working for the single-digit-MNIST dataset, the next step is apply to it to the SVHN dataset. This dataset has two different formats. The first one is compounded by all the real images from Google Street View and the metadata that contains the bounding boxes and the labels of each digit in the picture. This format requires a

preprocessing phase. The details of how the images were preprocessed will be explained and discussed later.

The second format provides preprocessed images with a size of 32 by 32 pixels. The labels do not correspond with the labels of the entire sequence but they represent the central digit of the image. The given data is a 4D Tensor so the MNIST model can be applied straightaway just converting the images to grayscale and modifying some variables of the model.

Technically, the model works similarly in both cases, using MNIST or SVHN data.

The next logical step was to transform the MNIST single digits into sequences. They were built by a random concatenation of digits to create variable length sequences. After that, the new image was resized to 28 by 28 pixels.

At the same time, the model had to change in order to be able to detect and identify more than one digit.

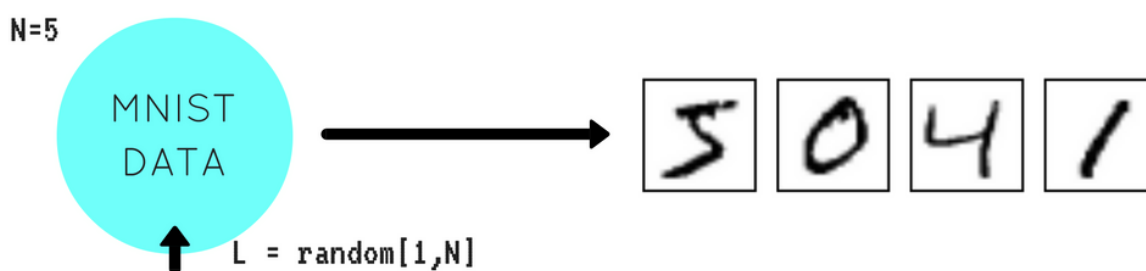


Figure 5: Creating MNIST Sequences

Finally, once the model was built and tested with the MNIST sequences, the next step was to consider how to preprocess the SVHN images. Each file was downloaded and extracted then, using the available metadata, each image was opened and cropped to get each digit from it.

Knowing perfectly how many digits were in each image and the bounding boxes of each one, the new images were created concatenating the digits according to the length of the sequence. Once created, they were properly resized to 32 by 32 pixels.

Labels were also extracted and all the data was converted into [Tensors](#). After that, it was properly saved in a pickle file which makes it easier to access the data.

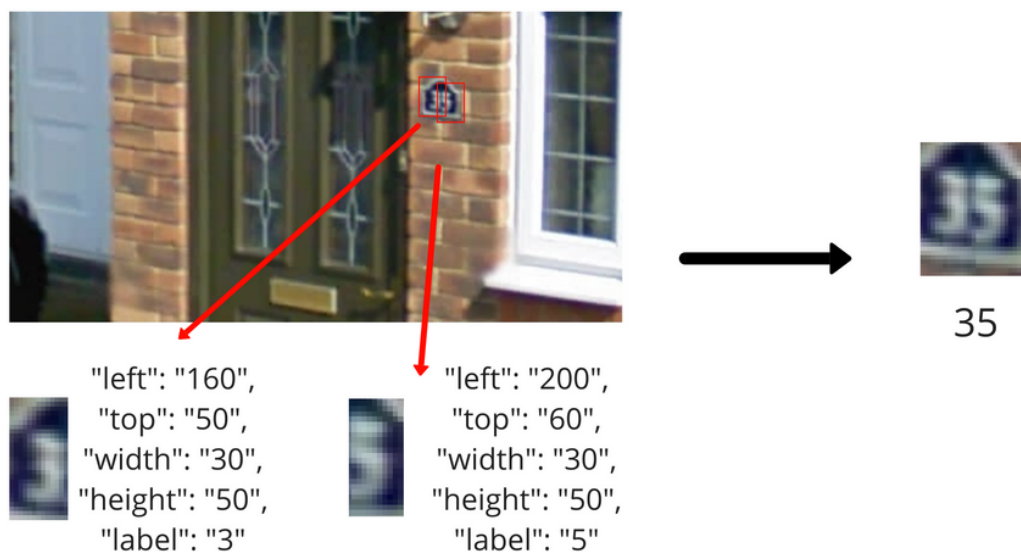


Figure 6: SVHN Preprocessing

Exploratory Visualization

The most important point to cover in the data visualization was to be safe and confident about any step done in the preprocessing functions.

For the MNIST dataset it was not necessary any kind of preprocessing stage as it was mentioned. In the case of SVHN, with the objective of simplifying the final model without losing information, the images were transformed into grayscale. Furthermore, a Global Contrast Normalization was applied to get better shapes of the numbers.

It was really important to notice that all the different transformations didn't alter the original information.

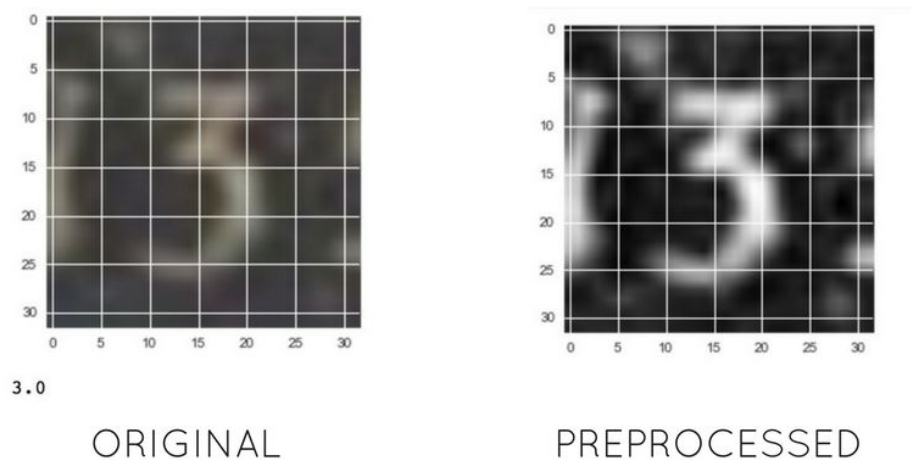


Figure 7: Preprocessing SVHN single digits

In the case of the **MNIST sequences**, the images were resized and the digits were a little distorted. This distortion didn't make the strength of the CNN model unstable and the digits were properly identified. This fact was analyzed in terms of **information loss** but, as it has been said, the model worked very well detecting the shapes of the numbers. Besides that, it's relevant to point out that as soon as the digits were horizontally concatenated, the original ratio of the images was lost.

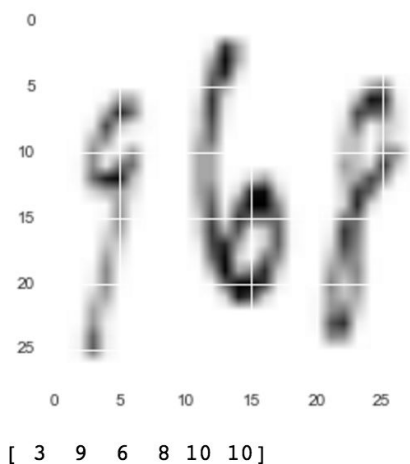


Figure 8: Artificial MNIST sequences

Finally, for the SVHN images the required preprocessing was a bit harder than the previous one.

The metadata was obtained from the .mat file. Then, each digit on each image was preprocessed (taking the bounding box from the metadata) and concatenated with the numbers of the same sequence.

In a parallel process, the labels were correctly extracted and processed, forming a 4D Tensor for the images and a 2D Tensor for the labels.

This process will be explained in detail in the next section.

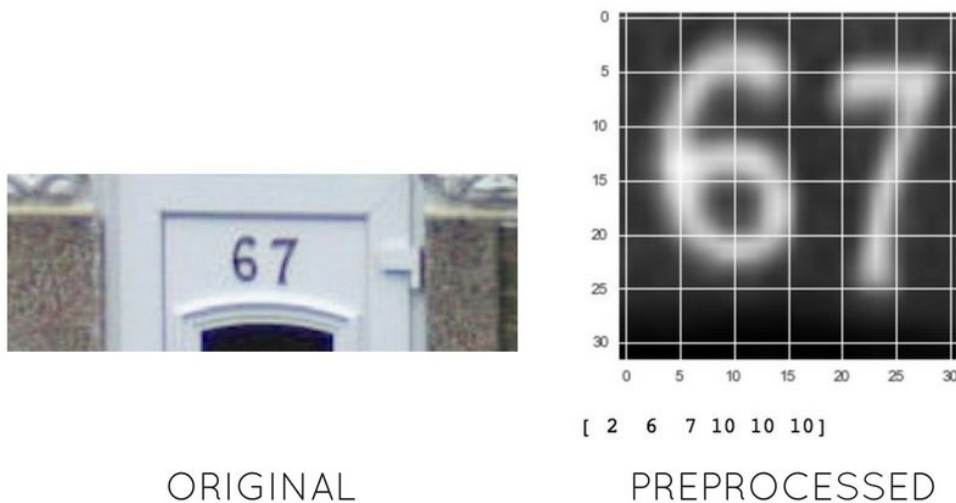


Figure 9: Preprocessing SVHN images

Algorithms and Techniques

At the beginning of the project a single-layer softmax regression model was built in order to create its foundations. Later, it was improved creating a softmax regression with a multilayer convolutional network.

Focusing on the SVHN-sequences, that is actually the main objective of the project, the output of the preprocessing step was 2 Tensors for each dataset as it was explained before.

Training set: (160755, 32, 32, 1) (160755, 6)

Test set: (58068, 32, 32, 1) (58068, 6)

Validation set: (30000, 32, 32, 1) (30000, 6)

Analyzing the Training set, the first Tensor (4D) represents the data resulted from the images:

- 160755 **samples**
- **Width:** 32 pixel
- **Height:** 32 pixel
- **Depth:** 1 channel (grayscale)

The second Tensor (2D) contains the information of the labels:

- 160755 labels (matching with the number of samples)
- 6 labels (5+1) **N=5 is the maximum number of digits in a sequence**. The first label of each sequence is always the length of the sequence. The absence of a digit it's represented by a **10**.

An example of this is shown in the Figure 9:

- Real number: 67
 - Labels: [2, 6, 7, 10, 10, 10] (the first element is 2 corresponding to the length of the sequence, the digits 6 and 7 are the house numbers, and there are three numbers 10 filling the array.)
- According to this, a label can take 11 different values [0-10].

The same explanation applies to the validation and test set.

About the model, the final architecture consists of **three convolutional locally connected hidden layers**. The connections of the convolution layers are feedforward and go from one layer to the next. The number of units at each spatial location in each layer is [16, 32, 64] respectively. All convolution kernels are of size 5×5 .

Each convolutional layer uses a stride of one and includes max pooling and subtractive normalization. The max pooling window size is 2×2 . The second convolution uses zero padding on the input to preserve representation size. The first and third convolution use VALID padding in order to reduce the output dimensionality.

The result of the convolutions (the feature vector **H**) is **fully connected to a N+1 Linear Regressions**. Each regression extracts the features corresponding to each digit in the sequence. The regression has to compute **256** values according to the size of the feature

vector, and of course, the size of each regression output matches with the number of classes for each digit [0-10] (**11**)

This model could be easily improved adding a fourth convolutional layer or increasing the number of units at each spatial location in each layer to look for a better feature extraction. However these modifications can drastically increase the computation cost of the training stage.

After that, the logits $z_{s_i} = w_{s_i}H + b_{s_i}$ are computed applying the [Softmax](#) algorithm and obtaining the probabilities that are assigned to each digit.

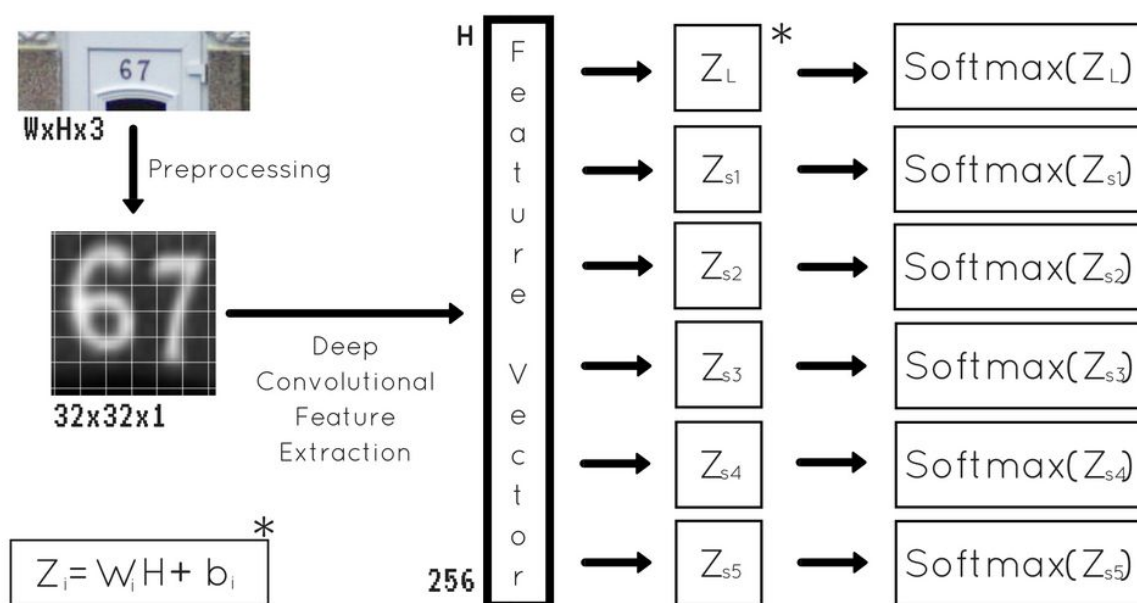


Figure 10: Model Overview

The result of the softmax functions can be considered the prediction of our model. It could be defined as the probability of belonging to a particular value. These logits are computed with the labels applying the [Cross Entropy Function](#).

Each digit has its own label and, in consequence, its own loss function. The total loss of the entire model is just the arithmetic sum of all individual losses.

In order to train the model, the [AdagradOptimizer](#) was applied with an initial value of 0.01 for the learning rate.

Summing up, the hyperparameters were initialized as follows:

IMAGE_SIZE = 32 (32 by 32 pixels each image)

NUM_CHANNELS = 1 (grayscale images, it would be 3 if the images were in color)

NUM_LABELS = 11 (0-10 both included)

N = 5 (maximum number of digits in a sequence)

BATCH_SIZE = 64 (number of samples of each batch on the training step)

PATCH_SIZE = 5 (5 x 5 convolution kernels)

DEPTH_1 = 16 (the first convolution compute 16 features for each 5x5 patch)

DEPTH_2 = 32 (the second convolution compute 32 features for each 5x5 patch)

DEPTH_3 = 64 (the third convolution compute 64 features for each 5x5 patch)

REGRESSION_INPUT_SIZE = 256 (the Regression compute the 256 features of the Feature Vector)

Benchmark

At the beginning of the project, the most important milestone was to reach a stable, well-analyzed and consistent model with a minimum accuracy of 90%.

Based on different papers and tutorials, it was obvious that with advanced libraries like TensorFlow, it was possible to model and create strong Convolutional Networks. The success or failure of the project would depend basically on the understanding of the data and the problem.

For that reason it was decided not to focus on complex regularization or processing techniques instead it was important to understand the data and step-by-step continue improving the model as soon as new requirements appeared.

The model starts being a Logistic Regression, then a 2NN, after that a 2L ConvNet, and finally a Multilayer ConvNet with multiple linear regressions. It was important to understand how hard it is to train properly a ConvNet. Deep Neural Networks need in general, large datasets and long periods of training which means expensive computation resources and a lot of time.

Once the model was clearly well-built, the final performance depended on the training stage. For that reason it was decided to move the training phase to Google Cloud Platform.

At the end of the process, the model should have a stable Test Accuracy of **90+%**.

Methodology

Data Preprocessing

Different datasets were used in this project, but the most important data preprocessing functionality was implemented for the SVHN dataset. As it was mentioned for the other problems it was enough with basic transformation and rescaling.

It was necessary to get a 4D Tensor corresponding with the previous explanation. The given data was divided and compressed into three different files (train, test and extra).

The first step was downloading and extracting the data. For that assignment two different functions were coded.

```
def maybe_download(filename, expected_bytes, force=False):
    """Download a file if not present, and make sure it's the right size."""
    downloaded = False
    if force or not os.path.exists(FOLDER + filename):
        print('Attempting to download:', filename)
        filename, _ = urlretrieve(URL + filename, FOLDER + filename, reporthook=download_progress_hook)
        downloaded = True
        print('\nDownload Complete!')
    if downloaded:
        statinfo = os.stat(filename)
    else:
        statinfo = os.stat(FOLDER + filename)
    if statinfo.st_size == expected_bytes:
        print('Found and verified', filename)
    else:
        raise Exception(
            'Failed to verify ' + filename + '. Can you get to it with a browser?')
    if downloaded:
        return filename
    else:
        return FOLDER + filename
```

```
def maybe_extract(filename, force=False):
    root = os.path.splitext(os.path.splitext(filename)[0])[0] # remove .tar.gz
    if os.path.isdir(root) and not force:
        # You may override by setting force=True.
        print('%s already present - Skipping extraction of %s.' % (root, filename))
    else:
        print('Extracting data for %s. This may take a while. Please wait.' % root)
        # Extract tar in the folder where it is placed
        sr = root.split('/')
        sr1 = root.split('/')[0:len(sr)-1]
        sr2 = "/".join(sr1)
        tar = tarfile.open(filename)
        sys.stdout.flush()
        tar.extractall(sr2)
        tar.close()
        print('Completed!')
    data_folders = root
    print(data_folders)
    return data_folders
```

Figure 11: Extract and download functions

The first function checks if the data is already downloaded and the second one checks if it is properly extracted. The data is extracted into three different folders (one for each file), containing the images (.png) and one file with the metadata (**digitStruct.mat**).

To continue with the process, it was defined a function that receives the name of the file that contains the metadata and returns a map (or dict) with all the extracted data.

```
def get_metadata(filename):
    f = h5py.File(filename)

    metadata= {}
    metadata['height'] = []
    metadata['label'] = []
    metadata['left'] = []
    metadata['top'] = []
    metadata['width'] = []

    def print_attrs(name, obj):
        vals = []
        if obj.shape[0] == 1:
            vals.append(obj[0][0])
        else:
            for k in range(obj.shape[0]):
                vals.append(f[obj[k]][0][0][0])
            metadata[name].append(vals)

    for item in f['/digitStruct/bbox']:
        f[item[0]].visititems(print_attrs)
    return metadata
```

Figure 12: Get Metadata Function

The next function shows in detail how an image is preprocessed. The function receives the name of the image file and the metadata associated with that image. The function iterates L times (L is the length of the current sequence) and extracts each digit thanks to the boundary box of each one. After that, the function resizes each digit and transforms it into grayscale.

Once all digits are available, each one is horizontally stacked and resized again obtaining a 32-by-32 pixels size.

```
def image_processing(image, metadata):
    original = Image.open(image)
    L = len(metadata['label'])
    aux = []
    for i in range(L):
        left = metadata['left'][i]
        top = metadata['top'][i]
        right = metadata['left'][i] + metadata['width'][i]
        bottom = metadata['top'][i] + metadata['height'][i]
        cropped = original.crop((left, top, right, bottom)) # crop with bbox data
        pix = np.array(cropped)
        pix_resized = imresize(pix, (IMAGE_SIZE, IMAGE_SIZE)) # resize each digit
        pix_gs = np.dot(pix_resized[...,:3], [0.299, 0.587, 0.114]) # grayscale
        aux.append(pix_gs)

    sequence = np.hstack(aux) # horizontal stack
    sequence_resized = imresize(sequence, (IMAGE_SIZE, IMAGE_SIZE)) # resize
    sequence_resized = sequence_resized.reshape((IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS)).astype(np.float32) # 1 channel
    return sequence_resized
```

Figure 13: Image Processing Function

There is also a function that receives all the metadata related with an image and returns the sequence of labels. As it can be seen in the following Figure, the default value in the array is 10, which means “no digit”. Iterating again L times it is possible to create the sequence of labels being always the first element of the array the value of L .

It is important to notice that the digits that are zeros in a SVHN image are by default labeled as “10”. In this case the model interprets 10 as “no digit” so after this step it was necessary to transform all tens that were actually zeros to zeros.

In other words a sequence **203** would be represented at this step as follows:

[3 2 **10** 3 10 10] and the model needs [3 2 **0** 3 10 10]

```
def label_processing(metadata):
    L = len(metadata['label'])
    seq_labels = np.ones([N+1], dtype=int) * 10
    seq_labels[0] = L # labels[i][0] = L to help the loss function. 6xLinearModels: s0...s5 and L
    for i in range(1,L+1):
        seq_labels[i] = metadata['label'][i-1]
    return seq_labels
```

Figure 14: Label Processing Function

To solve that, a function was defined that corrects that inconsistency in the labels. This function transforms the 10s from the sequence into 0s.

```
def label_zero(labels):
    for label in labels:
        #print(label)
        L = label[0]
        for i in range(1,L+1):
            if label[i] == 10:
                label[i] = 0
    return labels
```

Figure 15: Label Zero Function

As soon as all the data was preprocessed, a visualization stage was defined to ensure that all the images were well-transformed and suitably defined.

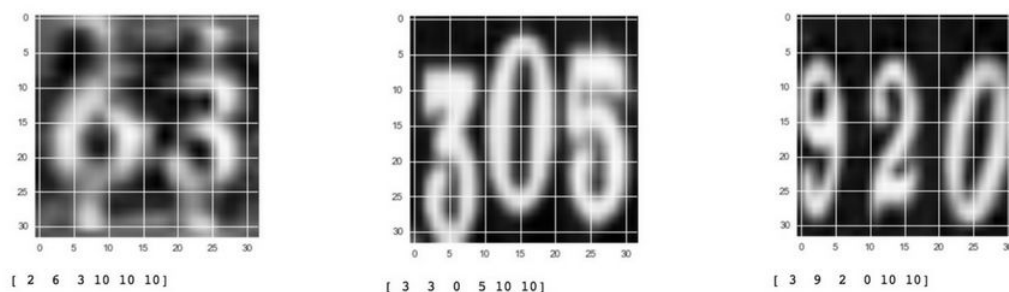


Figure 16: Image Preprocessing check

After a single step of reformatting and data randomization, all the data was saved in a pickle file in order to make it easier to load the data at any time.

The final output of this preprocessing phase was the 4D Tensor (num_samples, 32, 32, 1) and the 2D Tensor (num_samples, N+1). The final sizes of the sets are:

Training set: 160.755 samples

Validation set: 30.000 samples

Test set: 58.068 samples

Having said that, it's hard to think of overfitting problems with this amount of data.

Implementation

The implementation started building the computation graph by creating nodes for the input images and target output classes.

```
# Input data.
tf_train_dataset = tf.placeholder(tf.float32, shape=(BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS))
tf_train_labels = tf.placeholder(tf.int64, shape=(BATCH_SIZE, N+1))
```

Figure 17. Placeholders

Here **tf_train_dataset** and **tf_train_labels** are not specific values. Rather, they are each a placeholder -- a value that it will be input when the script asks TensorFlow to run a computation.

The input images **tf_train_dataset** consists of a 4D tensor of floating point numbers. It was assigned to the placeholder a shape of [64, 32, 32, 1]. Number 64 corresponds to the size of the batch, in other words, the number of samples processed on each step.

The other values are referenced to the size of the image, 32 by 32 pixels and 1 channel depth. That is exactly what it was obtained in our preprocessing stage.

The target output classes **tf_train_labels** consists of a 2D tensor where each element is a sequence of labels corresponding with the image.

The shape argument to placeholder is optional, but it allows TensorFlow to automatically catch bugs stemming from inconsistent tensor shapes.

Then, two constants were defined referencing the testing and validation set. They were used to periodically evaluate the performance in terms of accuracy.

```
tf_valid_dataset = tf.constant(valid_dataset)
tf_test_dataset = tf.constant(test_dataset)
```

At this point, Weights W and biases b for the model were defined as variables. A Variable is a value that lives in TensorFlow's computation graph. It can be used and even modified by the computation.

```
# Variables.
# 5x5 Filter, depth 16
conv1_weights = tf.Variable(tf.truncated_normal([PATCH_SIZE, PATCH_SIZE, NUM_CHANNELS, DEPTH_1], stddev=0.1))
conv1_biases = tf.Variable(tf.constant(1.0, shape=[DEPTH_1]))

# 5x5 Filter, depth 32
conv2_weights = tf.Variable(tf.truncated_normal([PATCH_SIZE, PATCH_SIZE, DEPTH_1, DEPTH_2], stddev=0.1))
conv2_biases = tf.Variable(tf.constant(1.0, shape=[DEPTH_2]))

# 5x5 Filter, depth 64
conv3_weights = tf.Variable(tf.truncated_normal([PATCH_SIZE, PATCH_SIZE, DEPTH_2, DEPTH_3], stddev=0.1))
conv3_biases = tf.Variable(tf.constant(1.0, shape=[DEPTH_3]))

# Linear
N1_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
N1_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))

N2_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
N2_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))

N3_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
N3_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))

N4_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
N4_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))

N5_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
N5_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))

NL_weights = tf.Variable(tf.truncated_normal([NODES, NUM_LABELS], stddev=0.1))
NL_biases = tf.Variable(tf.constant(1.0, shape=[NUM_LABELS]))
```

Figure 19. Variables

It is a good idea to generally initialize weights with a small amount of noise for symmetry breaking and to prevent 0 gradients. Since it was used ReLU neurons, it is also good practice to initialize them with a slightly positive initial bias to avoid "dead neurons".

Biases were initialized as one's (1.0) and Weights were initialized getting values from a truncated normal distribution with standard deviation of 0.1.

After that, the model was defined as follows.

```
def model(data):

    kernel1 = tf.nn.conv2d(data, conv1_weights, strides=[1, 1, 1, 1], padding='VALID')
    conv1 = tf.nn.relu(tf.nn.bias_add(kernel1, conv1_biases))
    pool1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    norm1 = tf.nn.local_response_normalization(pool1)

    kernel2 = tf.nn.conv2d(norm1, conv2_weights, strides=[1, 1, 1, 1], padding='SAME')
    conv2 = tf.nn.relu(tf.nn.bias_add(kernel2, conv2_biases))
    pool2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    norm2 = tf.nn.local_response_normalization(pool2)

    kernel3 = tf.nn.conv2d(norm2, conv3_weights, [1, 1, 1, 1], padding='VALID')
    conv3 = tf.nn.relu(tf.nn.bias_add(kernel3, conv3_biases))
    norm3 = tf.nn.local_response_normalization(conv3)
    pool = tf.nn.max_pool(norm3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    #pool = norm3

    pool_shape = pool.get_shape().as_list()
    reshape = tf.reshape(pool, [pool_shape[0], pool_shape[1] * pool_shape[2] * pool_shape[3]])

    # Linear Extraction for each component
    logits_L = tf.nn.bias_add(tf.matmul(reshape, NL_weights), NL_biases)
    logits_1 = tf.nn.bias_add(tf.matmul(reshape, N1_weights), N1_biases)
    logits_2 = tf.nn.bias_add(tf.matmul(reshape, N2_weights), N2_biases)
    logits_3 = tf.nn.bias_add(tf.matmul(reshape, N3_weights), N3_biases)
    logits_4 = tf.nn.bias_add(tf.matmul(reshape, N4_weights), N4_biases)
    logits_5 = tf.nn.bias_add(tf.matmul(reshape, N5_weights), N5_biases)

    return logits_L, logits_1, logits_2, logits_3, logits_4, logits_5
```

Figure 20. ConvNet Model

Each convolutional layer uses a stride of one and includes max pooling and subtractive normalization. The max pooling window size is 2×2 . The second convolution uses zero padding on the input to preserve representation size. However the first and third convolution use VALID padding in order to reduce the output dimensionality.

The first convolution computes 16 features for each 5×5 patch. Its weight tensor has a shape of $[5, 5, 1, 16]$. The first two dimensions are the patch size. The next one is the number of input channels and the last one is the number of output channels. It was also defined a bias vector with a component for each output channel.

After the max pooling stage, a normalization function is applied that could be considered as a “local contrast normalization” of the image features. The function computes the given vector dividing each component by the weighted, squared sum of inputs.

The second and third layer work in the same way just computing 32 and 64 features of their inputs for each 5×5 patch.

The result is a 256-feature vector that can be processed for the next 6 linear regression in order to extract the features for each component of the sequence.

Next, the logits that represent the features of each component are connected to the readout layer.


```

logits_L, logits_1, logits_2, logits_3, logits_4, logits_5 = model(tf_train_dataset)

lL = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_L, tf_train_labels[:,0]))
l1 = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_1, tf_train_labels[:,1]))
l2 = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_2, tf_train_labels[:,2]))
l3 = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_3, tf_train_labels[:,3]))
l4 = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_4, tf_train_labels[:,4]))
l5 = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits_5, tf_train_labels[:,5]))

loss = lL+l1+l2+l3+l4+l5

# Optimizer.
optimizer = tf.train.AdagradOptimizer(0.01).minimize(loss)

```

Figure 21. Readout Layer

As it was mentioned, the Softmax function was applied to the estimated logits and their labels. Once the entire function loss is calculated, it is passed to the AdagradOptimizer in order to find the variables that minimize that function.

The function **tf.nn.sparse_softmax_cross_entropy_with_logits** is useful in this situation because in this case an object can only belong to one class, so it is not necessary to convert the logits into One-Hot format.

To train and evaluate the model it was necessary to define a number of training steps that could take a while depending on the Graph and the computation resources.

In this example 20k steps were set for a partial training of the model. In the picture below, it's shown the session definition and the variables initialization.

```

NUM_STEPS = 20001

with tf.Session(graph=graph) as session:
    tf.initialize_all_variables().run()
    sp = saver.restore(session, "../tmp/SVHN-CNN-sequence-L2.ckpt")
    print('Model Restored')
    print('Initialized')
    for step in range(NUM_STEPS):
        offset = (step * BATCH_SIZE) % (train_labels.shape[0] - BATCH_SIZE)
        batch_data = train_dataset[offset:(offset + BATCH_SIZE), :, :, :]
        batch_labels = train_labels[offset:(offset + BATCH_SIZE),:]
        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
        _, l, predictions = session.run(
            [optimizer, loss, train_prediction], feed_dict=feed_dict)
        if (step % 10000 == 0):
            print('Minibatch loss at step %d: %f' % (step, l))
            print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_labels))
            print('Validation accuracy: %.1f%%' % accuracy(valid_prediction.eval(), valid_labels))
    sp = saver.save(session, "../tmp/SVHN-CNN-sequence-L2.ckpt")
    print('Model Saved')
    print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(), test_labels))

```

Figure 22. Session and training stage

The first step is to restore a model if it actually exists. Then, the process of training is quite simple. Each step takes random data from the training set that it is passed to the model.

It was set a series of minibatch evaluations to evaluate that the model was working properly.

At the end of each session, the final performance of the model is estimated, predicting the entire test set (+58k samples) and calculating the accuracy of that prediction as it was previously defined.

Implementation on Google Cloud Platform

As previously mentioned, this type of ConvNet model can be very hard to train in a standard computer. It usually takes a lot of computing resources and can take a lot of time to train a stable and consistent model.

In the actual scenario, it was quickly proven that the models worked very fine, at least in terms of performance-complexity tradeoff.

The model is not extremely complex but as soon as it's trained 20k steps it's possible to reach a 90% stable test accuracy which is not bad at all, taking into account the complexity of the problem.

In order to deliver a confident model it was necessary to implement a set of python-scripts based on the explained model and train it on the Public Cloud.

Having said that, that scripts were divided as follows:

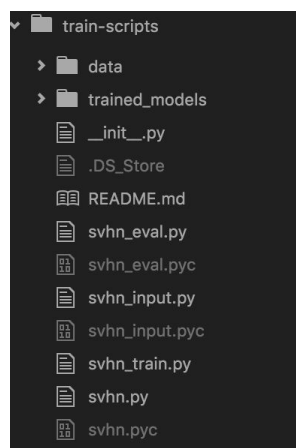


Figure 23. Training scripts

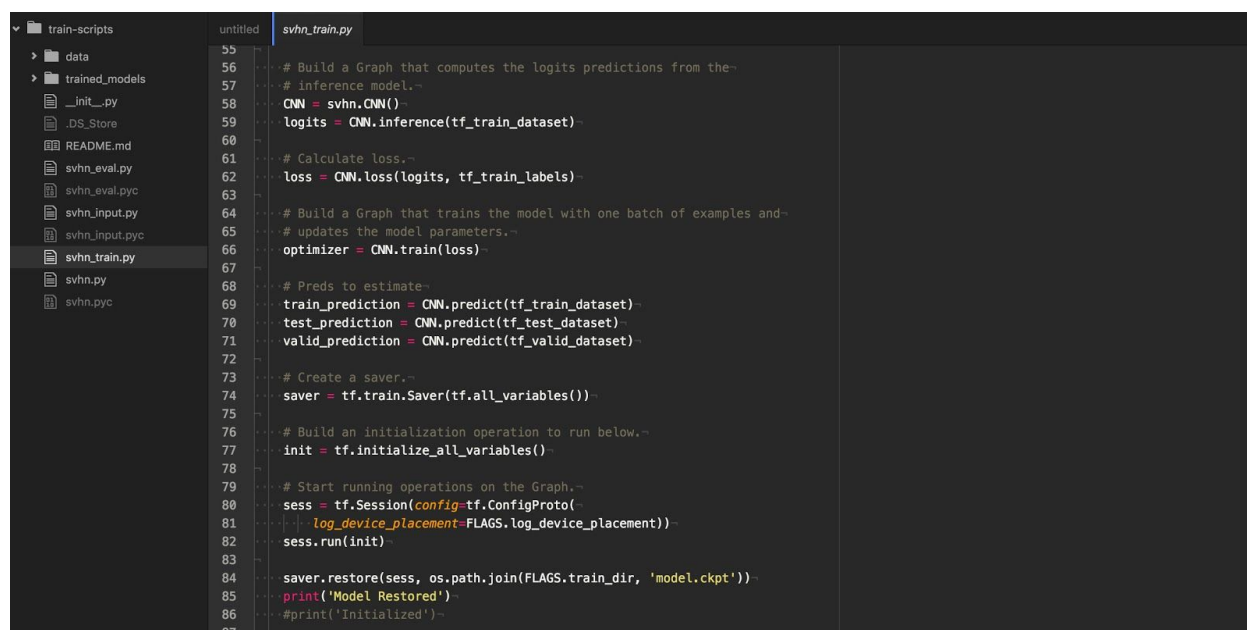
svhn.py: this script is the core and define a CNN class that allows to create ConvNets Model objects. This class has the following implemented functions: **inference**, **loss**, **train** and **predict**. Besides that, this script acts like an interface between the svhn_eval, svhn_input and the svhn_train scripts.

It's important to notice that the main script for this purpose is the `svhn_train` one. This implementation was made just to train for long sessions and save the results taking advantage of the Public Cloud computation power.

svhn_eval.py: in this script it's defined the accuracy function. In this case the periodic validations on the training and validation sets were not defined. The partial evaluations were just made in the minibatch training data.

svhn_input.py: this script loads the data from the pickle file. There is no extra functionality related to the preprocessing stage here. The pickle file was also uploaded into the **data** folder.

svhn_train.py: in this script was defined the TensorFlow Graph using the CNN class coded in the `svhn.py` script and it was set up a 200k steps training stage.



```

55
56 # Build a Graph that computes the logits predictions from the
57 # inference model.
58 CNN = svhn.CNN()
59 logits = CNN.inference(tf_train_dataset)
60
61 # Calculate loss.
62 loss = CNN.loss(logits, tf_train_labels)
63
64 # Build a Graph that trains the model with one batch of examples and
65 # updates the model parameters.
66 optimizer = CNN.train(loss)
67
68 # Preds to estimate.
69 train_prediction = CNN.predict(tf_train_dataset)
70 test_prediction = CNN.predict(tf_test_dataset)
71 valid_prediction = CNN.predict(tf_valid_dataset)
72
73 # Create a saver.
74 saver = tf.train.Saver(tf.all_variables())
75
76 # Build an initialization operation to run below.
77 init = tf.initialize_all_variables()
78
79 # Start running operations on the Graph.
80 sess = tf.Session(config=tf.ConfigProto(
81     log_device_placement=FLAGS.log_device_placement))
82 sess.run(init)
83
84 saver.restore(sess, os.path.join(FLAGS.train_dir, 'model.ckpt'))
85 print('Model Restored')
86 #print('Initialized')
87

```

Figure 24. svhn_train.py snippet

The service chosen to running this implementation in GCP was Compute Engine.

It was created a [n1-highcpu-16](#) machine on Compute Engine running Ubuntu 14.04.

Once Python and TensorFlow were installed on the machine, the script was executed accessing directly from shell and running **nohup python svhn_train.py &**

In the first session after **100k steps**, the Test Accuracy was approximately **94%**

After **200k** the performance was improved from **94% to almost 96%**.

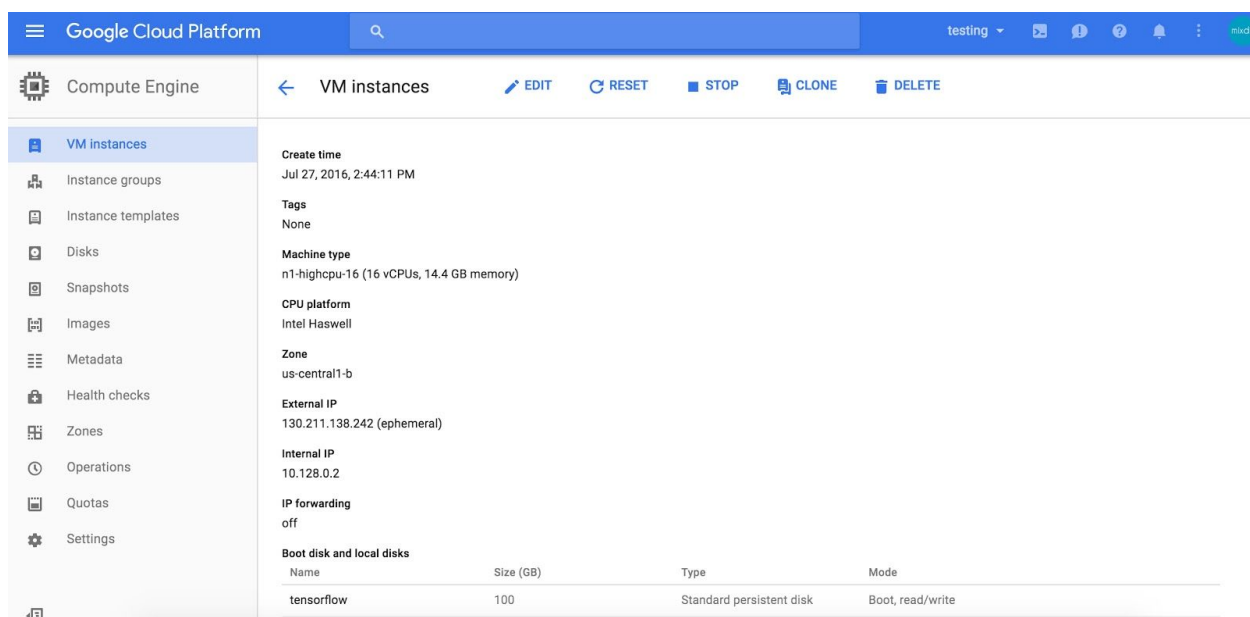


Figure 25. Creating a n1-highcpu-16 on Compute Engine

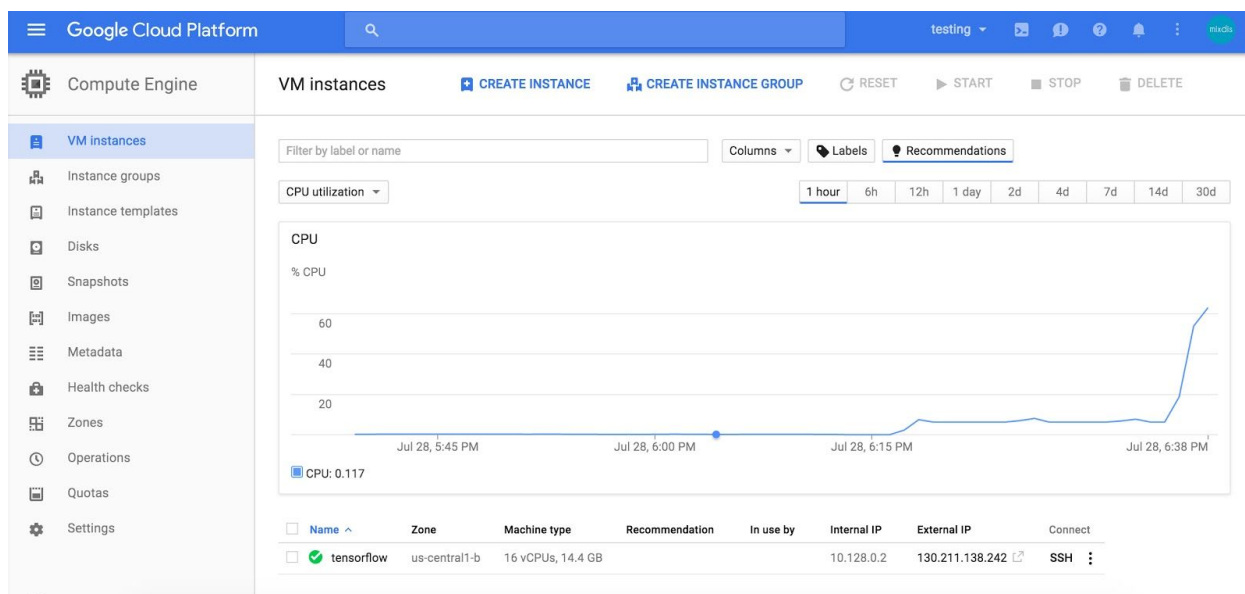


Figure 26. The VM instances running the script

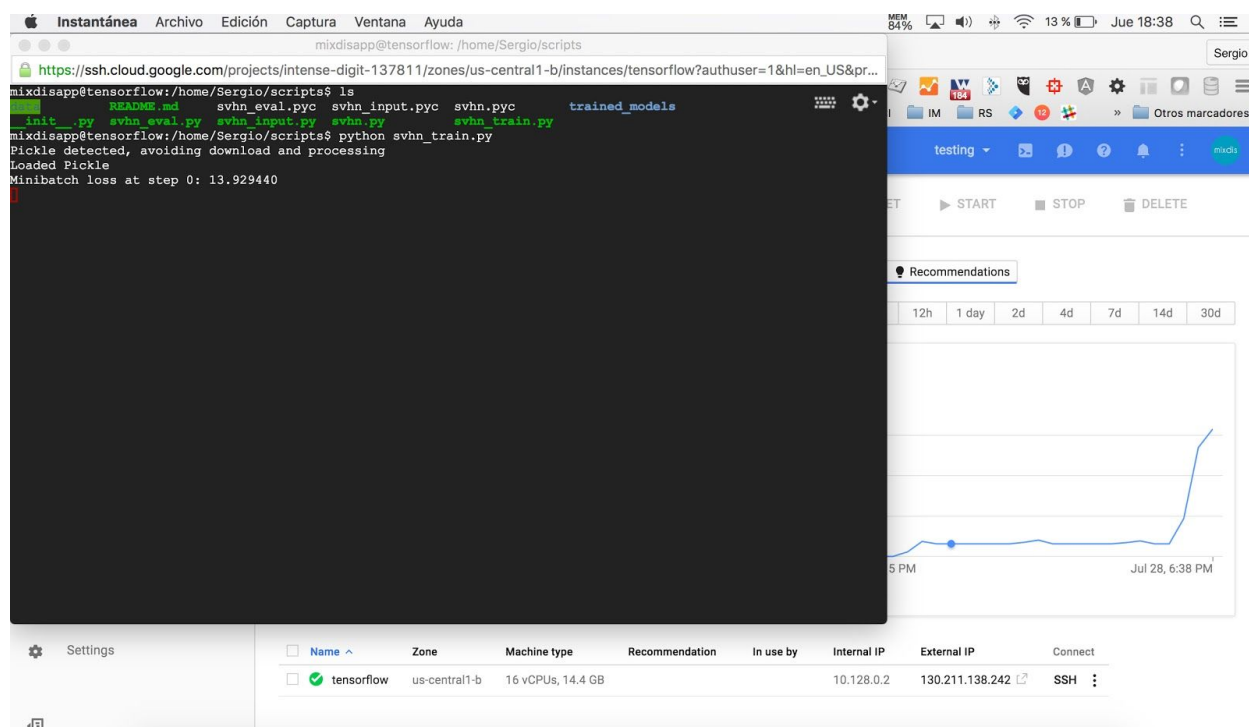


Figure 27. First steps of the training

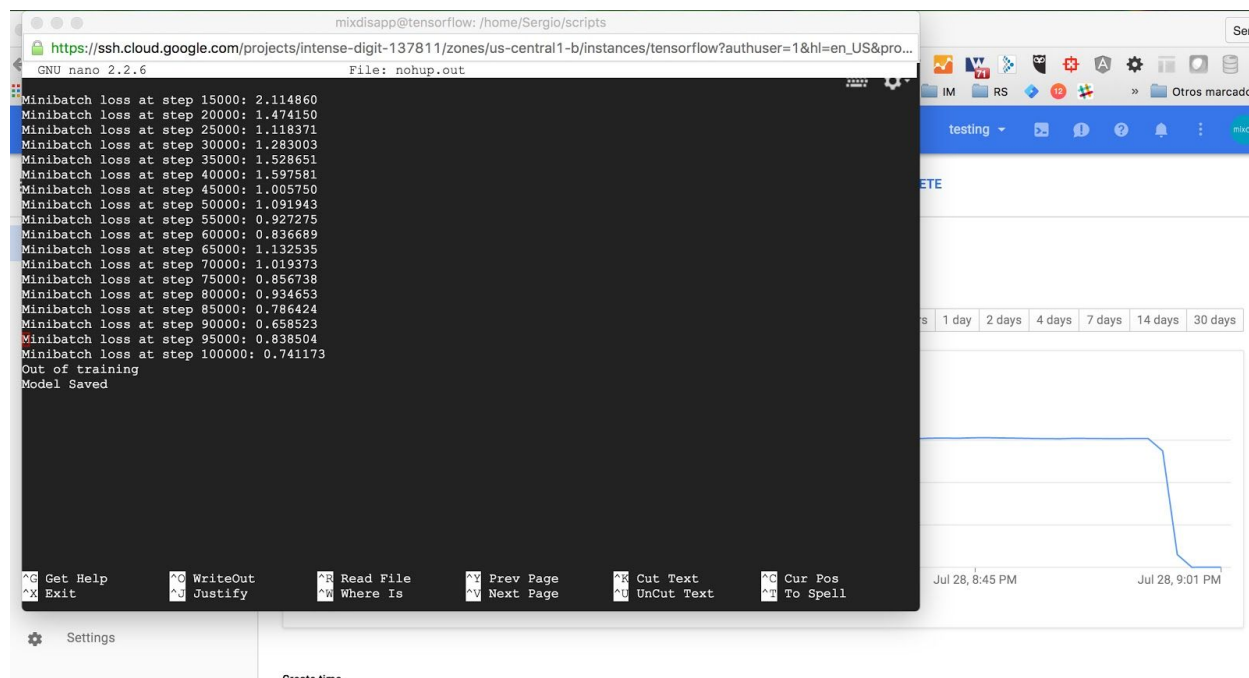


Figure 28. The end of the first 100k steps. The model was trained 200k steps.

Once the model was trained, it was suitably saved to a .ckpt file and downloaded. Then the VM instances were destroyed.

Build and Deploy a Web App on Heroku

Once the model was well-trained on Google Cloud Platform, some tests and evaluations were implemented with the objective of evaluating the final performance of the model.

Besides that, it seemed interesting to convert this project into something more practical. Something that could show in a better way the main goal of it.

For that reason a web application was built and deployed to Heroku.

To reach that, it was necessary to think in a Backend built in Python (as soon as it must run TensorFlow code), with two clear features. Be able to process an image (from the SVHN dataset) in real time, and to predict the real number based on the explained model.

The first step was to correctly define the limits of the web app. It doesn't allow the user to upload any kind of image, but shows two hundred images from the test dataset.

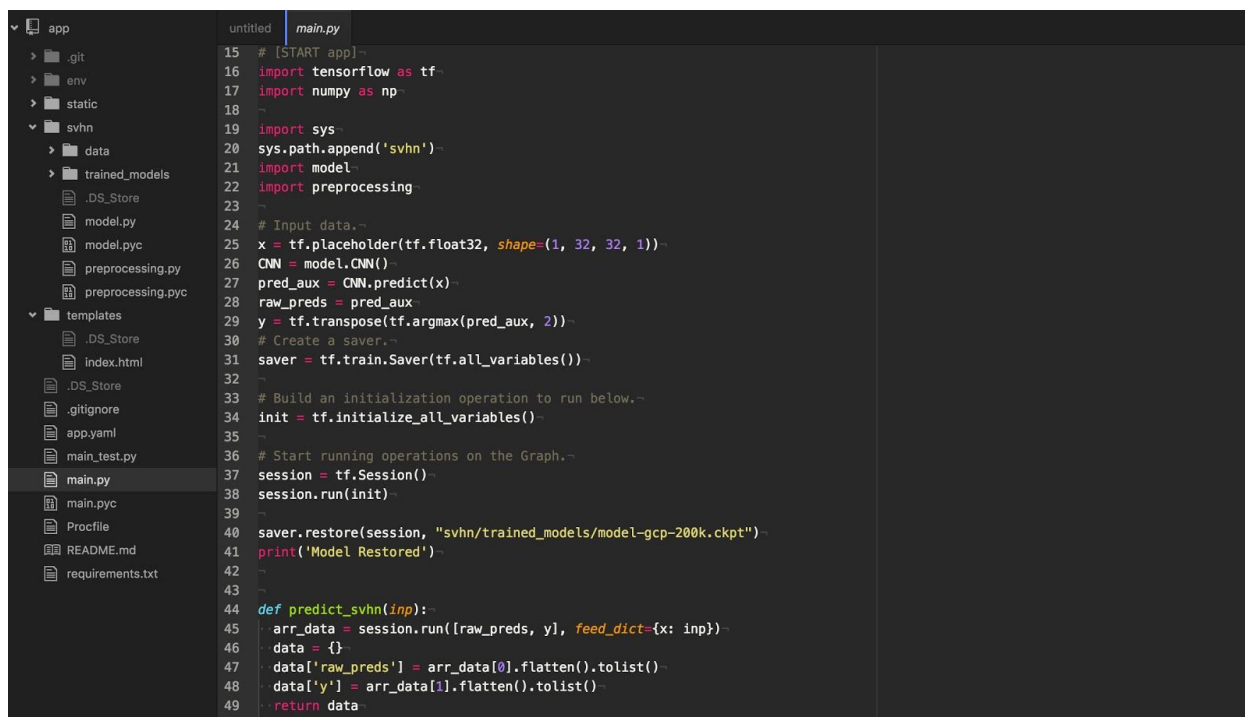
The user should select one of them and requests the prediction. In real time, the Backend gets the metadata information of the selected image from a preprocessed pickle, and makes a prediction.

Furthermore, the app not only gives the final prediction, but it also shows the **66** probabilities returned by the softmax function (11 labels per each digit).

Once the images and the metadata were properly preprocessed, it was the time to think of how to built the web app.

[Flask](#) was used to code the endpoints of the app and also to serve the static files (html, png, js or css).

The main script builds the TensorFlow graph, restores the model trained on Google Cloud Platform and creates a session which is able to make predictions.



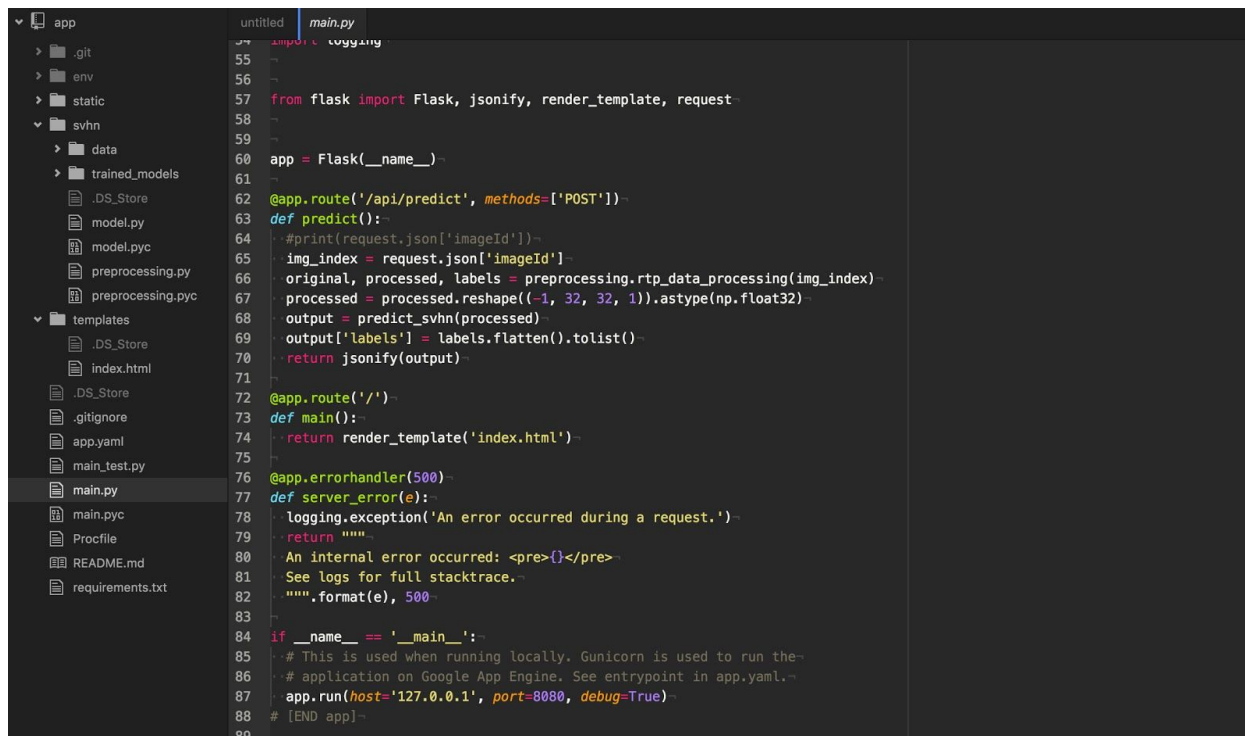
```

15 # [START app]~
16 import tensorflow as tf~
17 import numpy as np~
18 ~
19 import sys~
20 sys.path.append('svhn')~
21 import model~
22 import preprocessing~
23 ~
24 # Input data.~
25 x = tf.placeholder(tf.float32, shape=(1, 32, 32, 1))~
26 CNN = model.CNN()~
27 pred_aux = CNN.predict(x)~
28 raw_preds = pred_aux~
29 y = tf.transpose(tf.argmax(pred_aux, 2))~
30 # Create a saver.~
31 saver = tf.train.Saver(tf.all_variables())~
32 ~
33 # Build an initialization operation to run below.~
34 init = tf.initialize_all_variables()~
35 ~
36 # Start running operations on the Graph.~
37 session = tf.Session()~
38 session.run(init)~
39 ~
40 saver.restore(session, "svhn/trained_models/model-gcp-200k.ckpt")~
41 print('Model Restored')~
42 ~
43 ~
44 def predict_svhn(inp):~
45     arr_data = session.run([raw_preds, y], feed_dict={x: inp})~
46     data = {}~
47     data['raw_preds'] = arr_data[0].flatten().tolist()~
48     data['y'] = arr_data[1].flatten().tolist()~
49     return data~
50

```

Figure 29. main.py script (TF code)

Endpoints were created that exposed the prediction functionality. In this case, the graph only receives one sample at each time representing the image selected by the user.



```

54 import logging~
55 ~
56 ~
57 from flask import Flask, jsonify, render_template, request~
58 ~
59 ~
60 app = Flask(__name__)~
61 ~
62 @app.route('/api/predict', methods=['POST'])~
63 def predict():~
64     #print(request.json['imageId'])~
65     img_index = request.json['imageId']~
66     original, processed, labels = preprocessing.rtp_data_processing(img_index)~
67     processed = processed.reshape((-1, 32, 32, 1)).astype(np.float32)~
68     output = predict_svhn(processed)~
69     output['labels'] = labels.flatten().tolist()~
70     return jsonify(output)~
71 ~
72 @app.route('/')~
73 def main():~
74     return render_template('index.html')~
75 ~
76 @app.errorhandler(500)~
77 def server_error(e):~
78     logging.exception('An error occurred during a request.')~
79     return ""~
80     An internal error occurred: <pre>{}</pre>~
81     See logs for full stacktrace.~
82     """format(e), 500~
83 ~
84 if __name__ == '__main__':~
85     # This is used when running locally. Gunicorn is used to run the~
86     # application on Google App Engine. See entrypoint in app.yaml.~
87     app.run(host='127.0.0.1', port=8080, debug=True)~
88 ~
89 # [END app]~

```

Figure 30. main.py script (Flask code)

It was necessary to define and code the preprocessing and the svhn modules. It is not very interesting to explain it in detail because the important concepts were explained in the previous sections. However the [code](#) is entirely public.

The preprocessed pickle file was uploaded into the data folder and the trained model file was uploaded too into the trained_models folder.

At the same time it was necessary to build a state-machine frontend logic which was able to manage the web application requests and responses.

The final result is available in the following url:

<https://sequence-detector.herokuapp.com/>

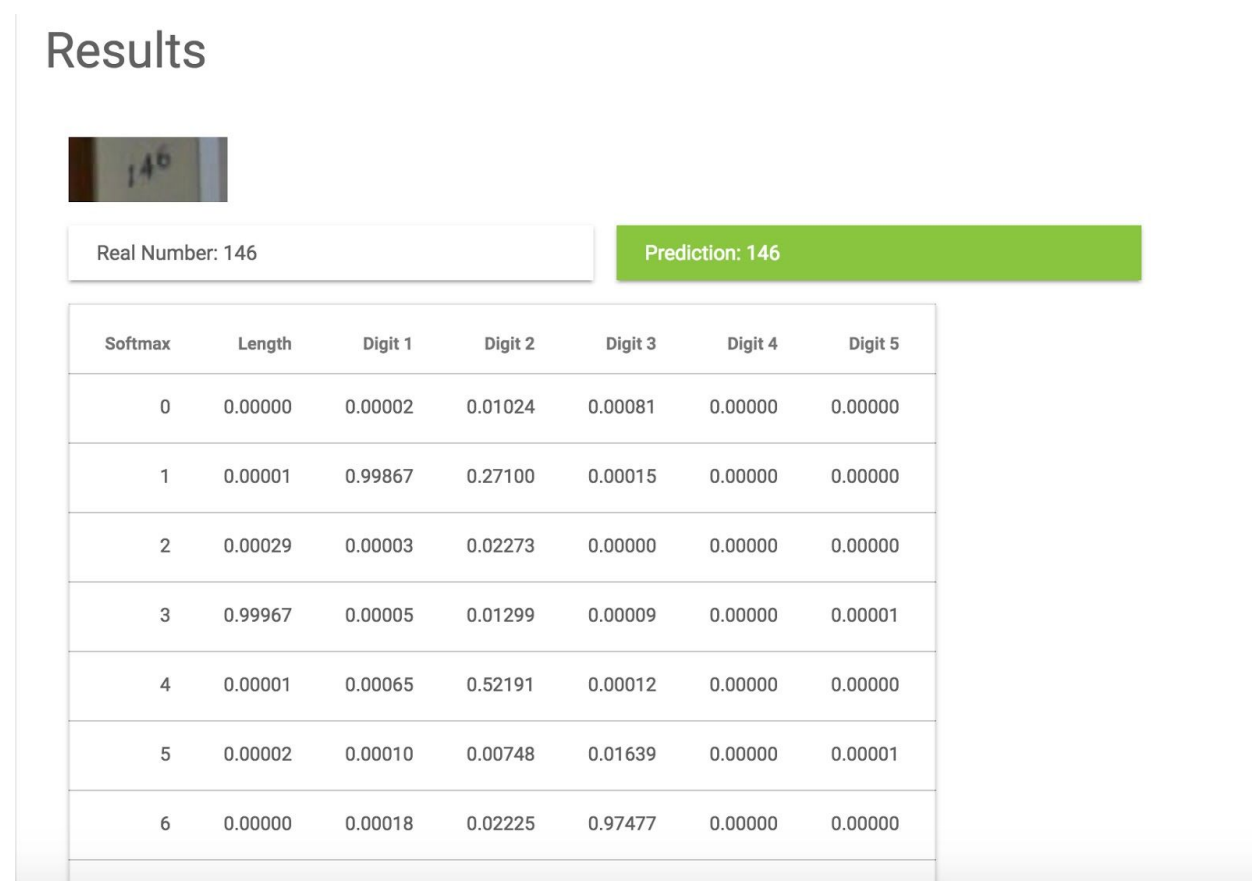


Figure 31. Web app results page

The requirements for the app are: **Python 2.7**, [PIP](#), [VirtualEnv](#) and the app modules specified in the **requirements.txt**

Results

Model Evaluation and Validation

The final model was trained several times in different environments to ensure that the model was actually stable and robust.

Once the final MultiLayer ConvNet was built for the SVHN data, the model was trained twice in a personal computer, generating two temporary models (**/tmp/SVHN-CNN-sequence-L.ckpt** and **/tmp/SVHN-CNN-sequence-L2.ckpt**).

In all training attempts the model reaches relatively fast the test accuracy of 90%. Anyhow the model did not improve from that performance just adding a few number of steps to the training stage. In these cases the final test performance, that means the success or failure predicting the entire test set (more than 58k samples), was between **91% and 92%**.

In these two training stages, the model was trained **20k steps** approximately.

In order to improve the model significantly, it was exported to python scripts and run it on GCP as it was explained.

After the training phase on GCP, there were also generated two different models.

The first one was trained 100k steps and the second one 200k. The first training stage took 75 minutes to run the 100k steps, and the second one took roughly the double.

In this case, the model improved to a 94% and 96% respectively. Both models are consistent and stable and they can be restored, retrained and tested at any time.

(**/trained_models/model-gcp-100k.ckpt** and **/trained_models/model-gcp-200k.ckpt**)

At the same time, different evaluation snippets were defined to check the final performance.

Loading the 200k-steps model, and taking 64 random samples from the test set, a **98.4% of accuracy** can be obtained. This snippet can be run at any time and the final results are always between **97.0% and 98.5%**.

```
with tf.Session(graph=graph) as session:
    tf.initialize_all_variables().run()
    ## 200k steps GCP trained model WIN-WIN!
    sp = saver.restore(session, "../trained_models/model-gcp-200k.ckpt")
    print('Model Restored')
    print('Initialized')
    offset = random.randint(0, test_dataset.shape[0]-SAMPLES)
    test_samples = test_dataset[offset:offset+SAMPLES]
    test_samples_labels = test_labels[offset:offset+SAMPLES]
    test_predictions = session.run(test_predictions, feed_dict={tf_test_dataset : test_samples})
    check_predictions(test_samples, test_samples_labels, test_predictions)
```

Figure 32. Evaluating the model

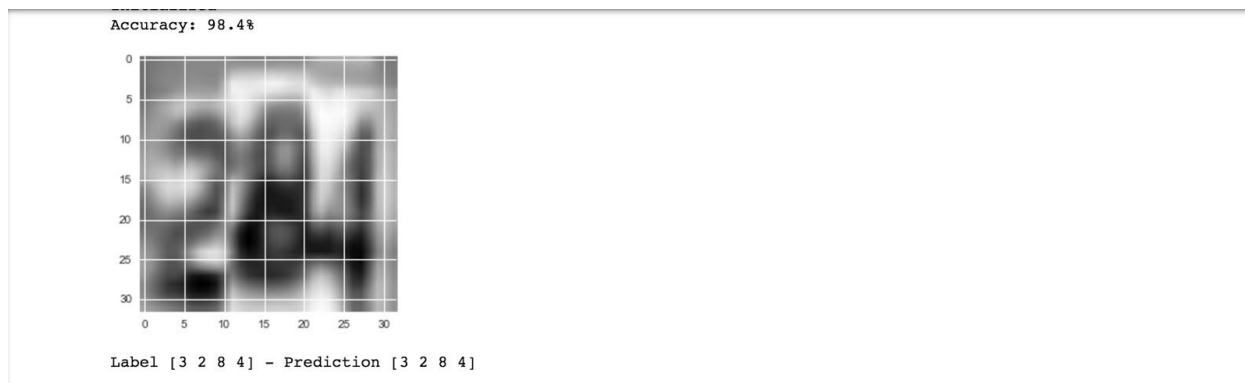
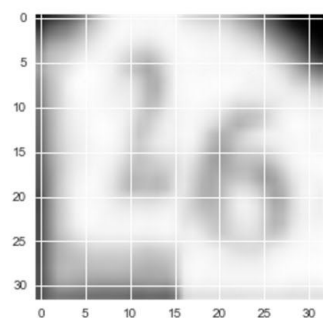


Figure 33. Prediction example

In addition, another snippet was defined which was the foundation of the web app functionality. This snippet does not directly load the data from the pickle but it takes 64 random images from the test dataset and preprocess them in “real time”.

After that, it estimates the accuracy of the model predicting the entire validation set (30k unseen samples) in order to represent better the reliability, independency and stability of the final model.

In the example the accuracy of prediction 64 unseen samples is 97.4% and the validation accuracy is 96.9% so in general it could be said that the model generalizes pretty well. It could be improved applying the mentioned regularization techniques or maybe by setting longer training stages.



Label [2 2 6] - Prediction [2 2 6]
Validation accuracy: 96.9%

Figure 34. “Real Time” Prediction and Validation Accuracy

Justification

It could be perfectly said that the initial goals are actually reached. One of the most important objectives were to be able to transform a basic Linear Function in a complex ConvNet that predicts, at least confidently, the numbers in a SVHN image.

The reasoning followed along the project was clear, explaining any point of the model and justifying why it was necessary or not. At the same time, different techniques were explained that could easily improve the model.

And last but not least, the predictions made by the model are completely trusted, stable and easily evaluated.

The model not only predicts confidently, but it does it correctly with a stable 97-98% of accuracy which means that the created model fits in this problem very well.

Conclusion

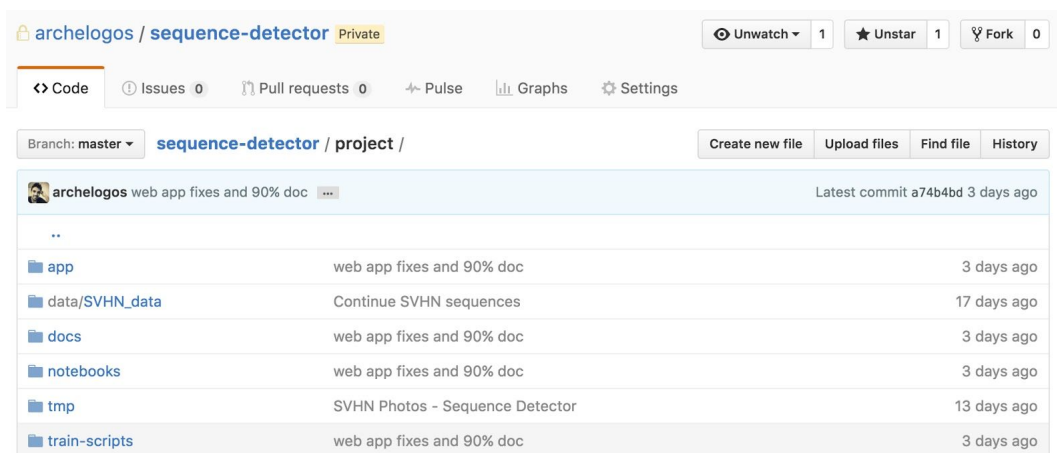
Free-Form Visualization

The current document has a great number of visualization examples that clarify the followed process and help to understand the different datasets used in the project.

This project has some available resources to complete the understanding of it.

The public repository (<https://github.com/archelogos/sequence-detector>) contains all the code, notebooks and documents that were necessary to complete the project.

In the “project” folder of the repo the following structure can be found:



File/Folder	Commit Message	Time Ago
..		
app	web app fixes and 90% doc	3 days ago
data/SVHN_data	Continue SVHN sequences	17 days ago
docs	web app fixes and 90% doc	3 days ago
notebooks	web app fixes and 90% doc	3 days ago
tmp	SVHN Photos - Sequence Detector	13 days ago
train-scripts	web app fixes and 90% doc	3 days ago

Figure 35. Repository Structure

The ipython notebooks created and the exported html from each one are in the “notebooks” folder.

The “train-script” folder contains the scripts for training the model in GCP.

The “app” folder contains the scripts of the web app deployed on Heroku.

The other ones contain docs and papers, the used data, and the temporary saved models

Reflection and Improvement

This project has been important to understand better the different steps in a Machine Learning problem.

- Understand the data. It is very important to spend all the necessary time to understand the available data.
- Understand the complexity of creating and analyzing a model that fits in real-world problem.
- Be able to use an Open Source library like TensorFlow and adapt the theory to a production-ready environment.
- Evaluate any step along the process in order to ensure that each one is in the correct direction.
- Pay attention to the typical problems building machine learning models (overfitting, bias-variance tradeoff, etc)

At the end, the model is built with one of the most advanced ML libraries. This is perfect to understand how Neural Networks are used in a real-world problem.

It is important to notice that this problem has many different points of view. One of them could be to focus the model in the “raw” images of the house numbers instead of in the preprocessed data.

The mentioned scenario could have been more interesting in order to create an application that could predict pictures taken straightaway from a mobile camera. This is actually one of the planned improvements of the project.

In the other hand, there are some improvements directly related with the actual project and its model. Some of them have been mentioned in the previous section:

- L2 Regularization
- Dropout
- Decay Learning Rate

These improvements are relatively easy to implement in the current model but it is not so easy to evaluate their improvement in the final performance. This is basically because each training stage of 200k steps is not cheap at all and takes time and computation resources.

Anyway these are some of the planned improvements to the future: Improve the current model with the mentioned techniques, analyze how to transform the model to process raw images and not cropped digits, or think in a useful real-world application where this model and its results may be applied.