



SVHN Sequence Detector

Machine Learning Nanodegree Capstone Project

Summer 2016

Sergio Gordillo

@Sergio_Gordillo

github.com/archelogos/sequence-detector

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

In order to simplify the explanation, the desired result is shown in the following picture.

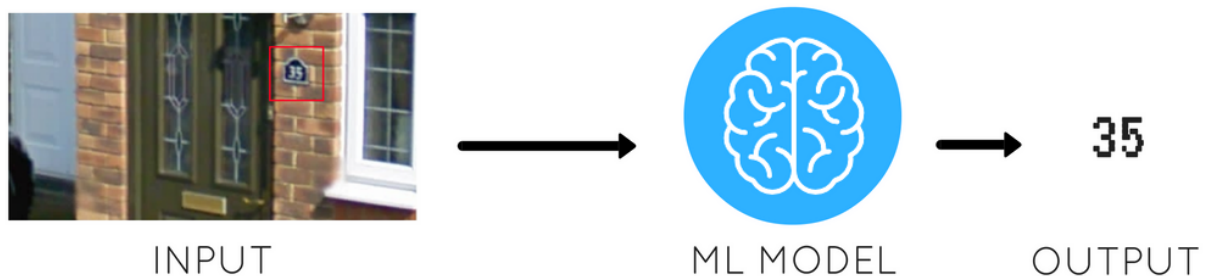


Image 1: Project goal overview

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

This is a real-world problem studied for many years and it wasn't until the application of neural networks to the images recognition problems when it could be considered as solved. At this moment it can be seen real products and apps that are focused on solve this problem (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.

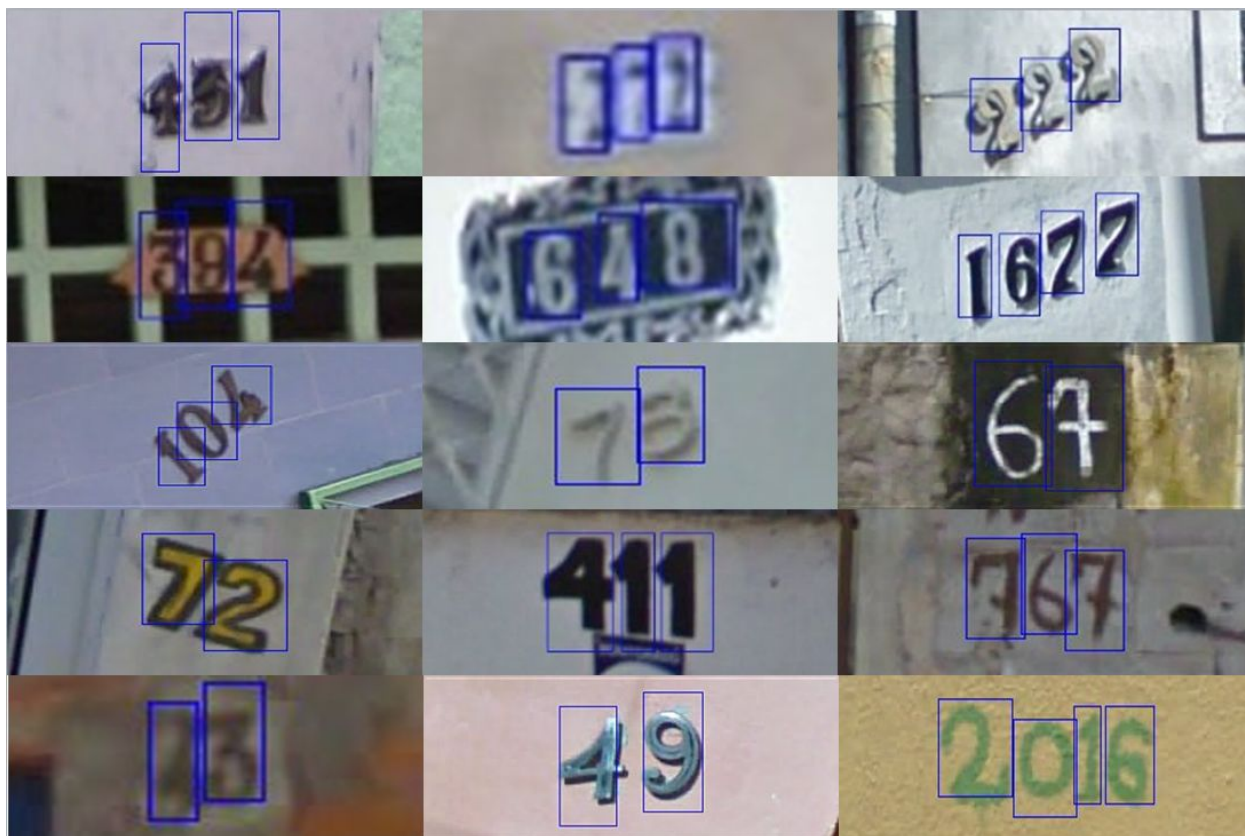


Image 2: Samples of images from the dataset

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

The model begins being as basic as it's possible based on a simple [Logistic Regression](#). Along the project, the model is getting more complicated and sophisticated step by step, becoming a Multilayer [ConvNet](#) production-ready model at the end of the project.

In the process, different techniques, algorithms and solutions are 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-digit-MNIST** dataset.
2. Modify the previous model and apply it to a **single-digit-SVHN** dataset.
3. Improve the model and apply it to a **sequence-MNIST** dataset.
4. Modify the previous model and apply it to a **sequence-SVHN** dataset.
5. Define a stable and consistent 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 resources. For that reason the first steps of the project are developed in ipython notebooks, running small training per each session. Once it is clear that the model works fine, it was moved to the Public Cloud (GCP) to train it in a better way (highcpu machines running efficient py scripts).

Anyway it's important to notice that the purpose of this project is not to improve the current best models built for this problem but it's to study and build an able and production-ready model which can perfectly obtain a 96% of accuracy identifying sequences from 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 defined for this problem.

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

A prediction it's 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 not correct.

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's always estimate 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
```

Image 3: Pseudo code of accuracy function used

Analysis

Data Exploration

As it was explained, the project itself and the main objective of it is focused on the SVHN dataset. However, there are other interesting dataset that can be used at the first steps of the project in order to be more practical studying the model and delaying until the end of the project the preprocessing functionality.

One of these dataset is the MNIST one and it consists of 70.000 examples of **handwritten** digits. The size of the representation of each number is 28x28 pixels, and they are normalized. Each digit has 784 features, (28 by 28) and 1 channel-depth corresponding to the grayscale (28x28x1).

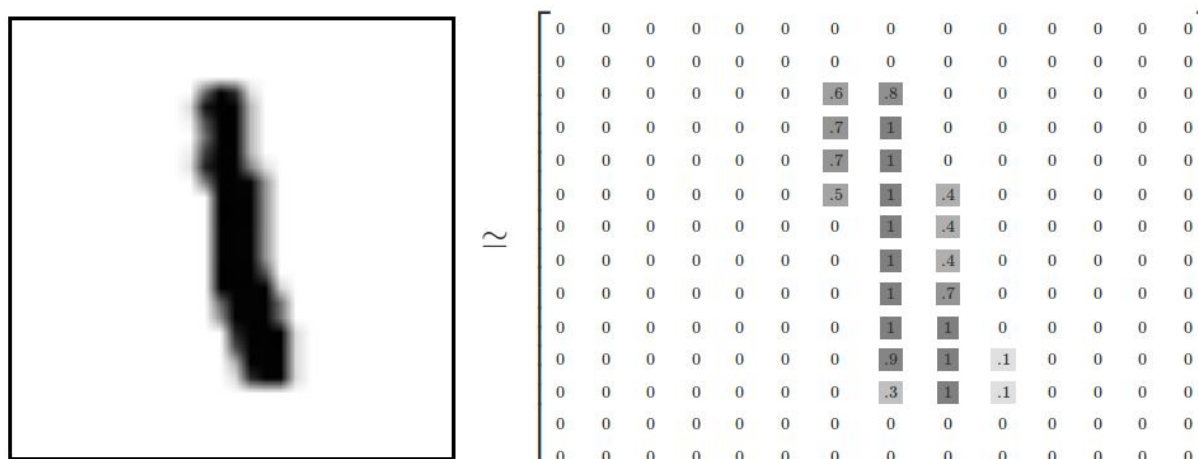


Image 4. MNIST Digit representation

This data is downloaded from the [Yann LeCun Website](https://yann.lecun.com/ex/ex2/) and after that, the files are properly extracted and reformatted creating the different dataset (training, validation and test).

The labels are not downloaded in [One-Hot](#) encoding so the label of each digit is the value of the own number and it's not other kind of representation.

Once the model is working for the single-digit-MNIST dataset, the next step is apply it to the SVHN dataset. This dataset has two different formats: the first one is compound by all the real images from Google Street View and the metadata that contains the bounding boxes and the labels for each digit in the picture. This format requires that the images are preprocessed. The details of how they were preprocessed will be explained and

discussed later because for this step it was used the second format from the SVHN dataset.

The second format provides preprocessed images with a size of 32 by 32 pixels. The labels don't correspond with the labels of the entire sequence but they represent the central digit on the image. The data is given in a 4D Tensor so the previous build model can be applied just converting the images to grayscale and modifying the variables of our model that are related with the image size (from 28x28 to 32x32).

Technically the model works at the same way in both cases, using MNIST or SVHN data.

The next logical step was to transform the MNIST single digits into sequences, and these were just built concatenating randomly digits and creating sequences of variable length. After that, the new image was reformatted to a size of 28x28. Obviously, the model had to change to be able to detect and identify more than one digit.

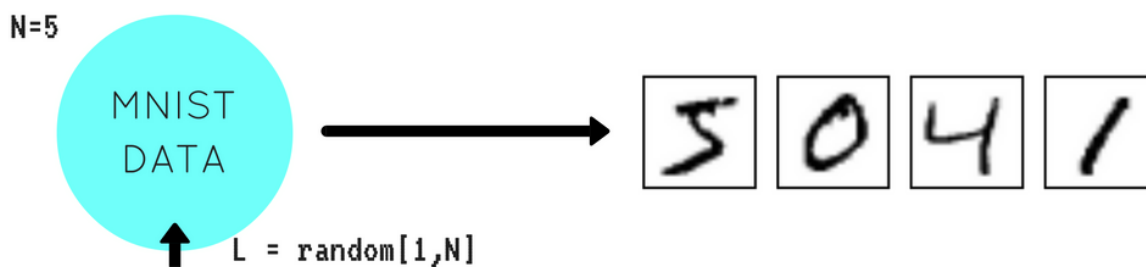


Image 5: Creating MNIST Sequences

Finally, once the model was built and tested in the MNIST sequences, it was necessary to think in how to preprocess the SVHN images. Thanks to some Python libraries this step wasn't so hard as it could be. Each file was downloaded and extracted and using the available metadata it was possible to open each image and crop each digit from it. Knowing perfectly how many digits were in each image and the bounding boxes of each of them, the new images were created concatenating the digits according to the length of the sequence and properly resized to 32 by 32 pixels.

Labels were also extracted and all the data was converted in Tensors. After that it was properly saved in a pickle file which makes easier to access the data.

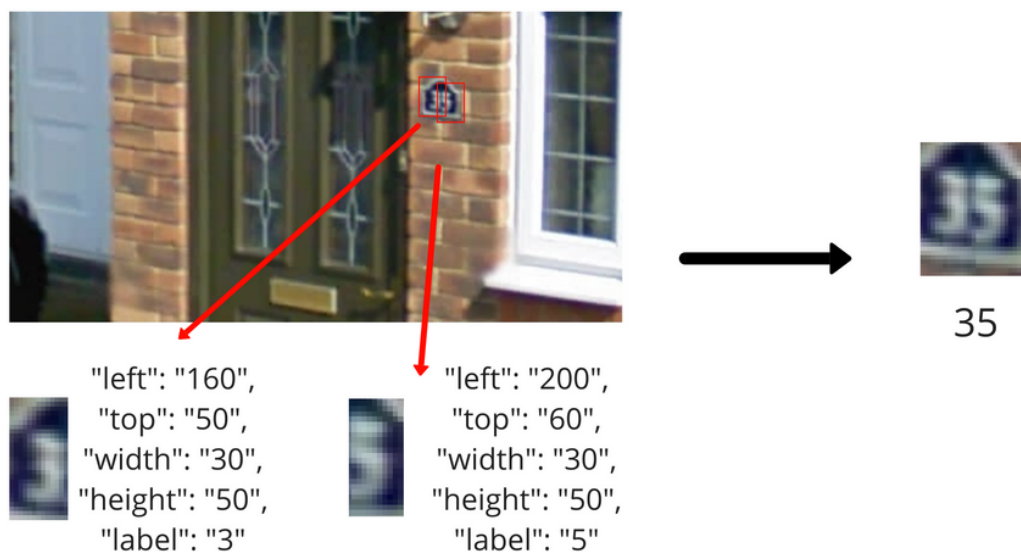


Image 6: SVHN Preprocessing

Exploratory Visualization

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

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

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

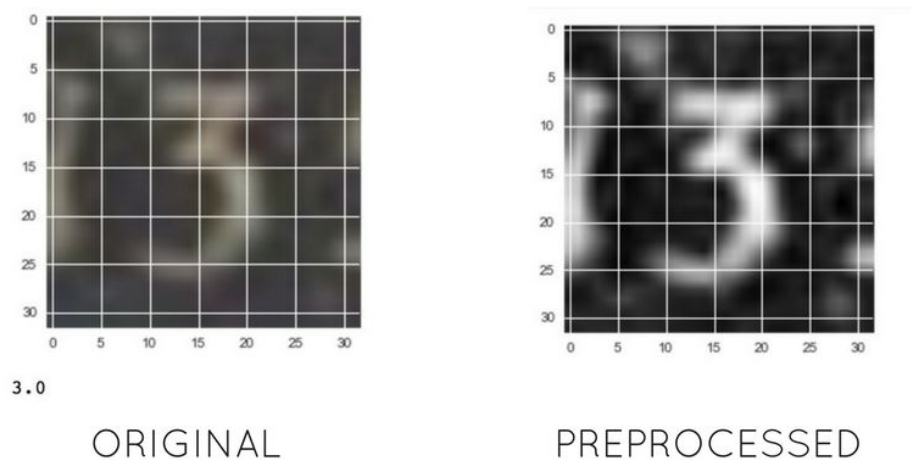


Image 7: Preprocessing SVHN single digits

In the case of the MNIST **sequences**, the images were resized and the digits were a little bit distorted, but it didn't alter strength of the CNN model and the digits were properly identified. This fact was analyzed in terms of information loss but, as it was just said, the model worked very fine detecting the shapes of the numbers in any case. Besides that, it's important to point out that as soon as the digits were horizontally concatenated, the proportion of the image was lost in one dimension.

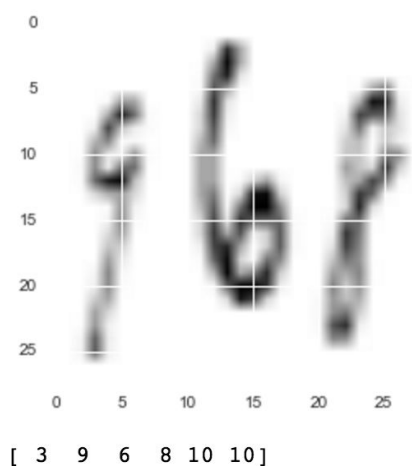


Image 8: Artificial MNIST sequences

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

Once the files were downloaded and extracted, the metadata was obtained from the .mat file and each digit on each image were 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 at the end a 4D Tensor for the images and a 2D Tensor for the labels.

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

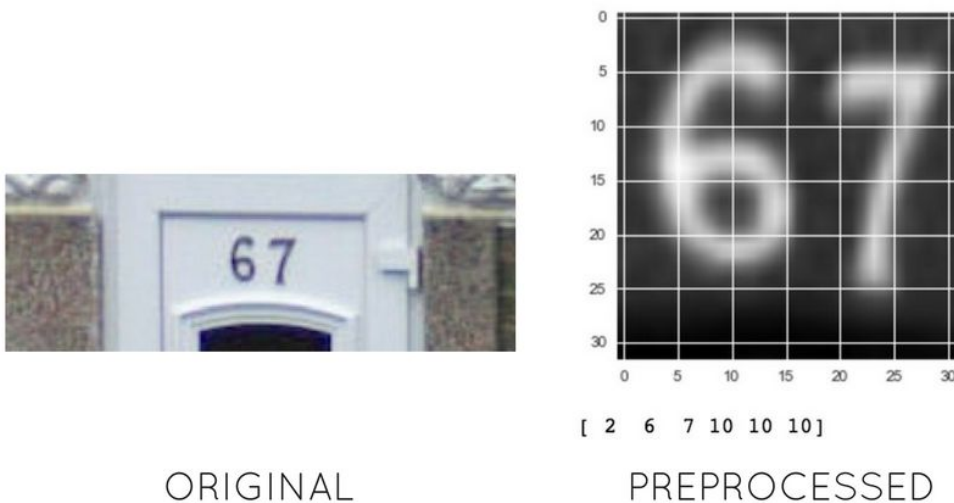


Image 9: Preprocessing SVHN images

Algorithms and Techniques

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

Focusing on the SVHN-sequences, that is the main objective of the project, the result of the preprocessing step was 2 Tensors per each set as it was pointed out.

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 converted 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 it. The absence of a digit it's represented by a **10**.

An example of this is shown in the Image 9:

- Real number: 67
- Labels: [2, 6, 7, 10, 10, 10] (length of the array 6, first element 2 corresponding with the length of the sequence, the digits, and 3 numbers 10 filling the array. According to this, the number of different classes is 11 [0-10])

The same explanation applied to the validation and test set.

The final architecture consists of **three convolutional locally connected hidden layers**. The connections of the convolutions 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 looking for a better feature extraction. However this modifications can increase drastically the computation cost of the training.

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.

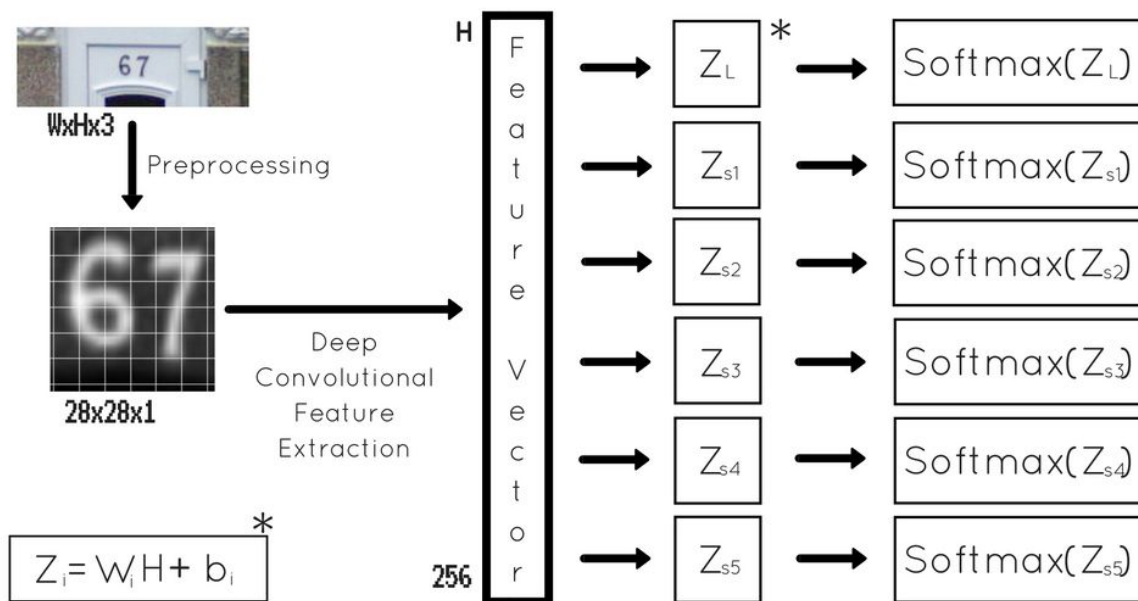


Image 10: Model Overview

The result of this softmax function can be considered the prediction of our model. It could be defined as the probability of be 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 summation of all individual losses.

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

Summing up the hyperparameters initial setup is:

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 and the success or fail of the project would depend basically on the understanding of the problem.

For that reason it was decided not to focus on complicated regularization or processing techniques but it was important to understand the data and step-by-step improved 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 multiples linear regressions. It was important to understand how hard is to train a ConvNet properly. Deep Neural Networks need large dataset and long periods of training which means expensive computation resources and time. Don't forget that this kind of models trying to solve extremely hard problems as Pattern Recognition.

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

At the end of the process, the model has a stable Test Accuracy of 96% and of course modifying some hyperparameters like the convolutional depths or applying advanced regularization techniques, the final result could be improved.

Anyhow these improvements will be discussed in the "Improvements" section.

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 dataset corresponding with the previous explanation. The data given was divided and compressed into three different files (train, test and extra).

The first step was download and extract the data. For that commitment were coded two different functions.

```
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
```

Image 11: Extract and download functions

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

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

```
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
```

Image 12: Get Metadata Function

The next function shows in detail how an image is preprocessed. The function receives the path 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 the 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
```

Image 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, 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 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 ten’s that were actually “0” to “0”.

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

[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
```

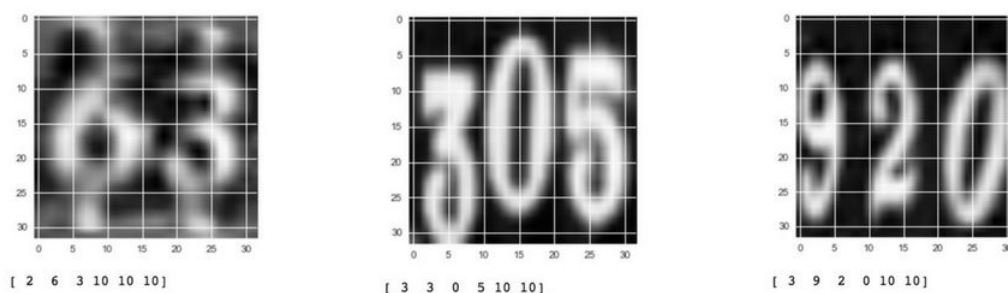
Image 14: Label Processing Function

Mentioned that, the next step was to define a function that corrects that inconsistency in the labels. This function just transforms the ten’s from the sequence into zeros.

```
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
```

Image 15: Label Zero Function

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



After a single step of reformatting and data randomization, all the data was saved in a pickle file in order to make easier to load the data several times.

The final result of this preprocessing were the 4D Tensor (num_samples, 32, 32, 1) and the 2D Tensor (num_samples, N+1). The final size of the sets are:

Training set: 160.755 samples

Validation set: 30.000 samples

Test set: 58.068 samples

Seen this, It's hard to think in face overfitting problems later 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))
```

Image 17. Placeholders

Here **tf_train_dataset** and **tf_train_labels** aren't 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]. The 64 corresponds to the size of the batch, in other words, the number of samples passed 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 evaluate periodically 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]))
```

Image 19. Variables

To create this model, It was necessary to define the weights and biases. It's 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
```

Image 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, the first and third convolution use VALID padding in order to reduce the output dimensionality.

The first convolutional 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 is the number of input channels, and the last 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, it's applied a normalization function that could be considered as a "local contrast normalization" of the features of the image. The function compute the vector given 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 in the sequence.

Next, the logits that represents 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, it was applied the Softmax function to the estimated logits and their labels. Once the entire function loss is calculated, 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** it's useful in this situation because in this case an object can only belong to one class, so it's not necessary to convert the logits into One-Hot format.

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

In this example it was set 20k steps for a partial training of the model. The first thing that it could be seen, is 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))

```

Image 22. Session and training step

If it existed a saved model, the first step was to restore it. Then the process of training is quite simple. Each step takes random data from the training set and this data is passed

to the model. As it can be seen, it was set a series of minibatch evaluations to check that the model was working properly.

At the end of each session it was estimated the performance of the model, 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 it was mentioned before, this type of ConvNet model can be very hard to train in a standard computer. They usually take a great computing resources and can take a lot of time to train a stable and consistent model.

In the actual scenario, it was fastly proved that the models works very fine, at least in terms of performance-complexity tradeoff.

The model is not extremely complicated 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 in the Public Cloud.

Said that, that scripts were divided as follows:

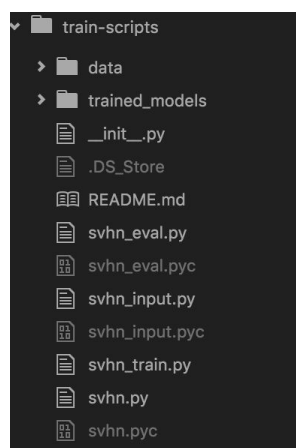


Image 23. Training scripts

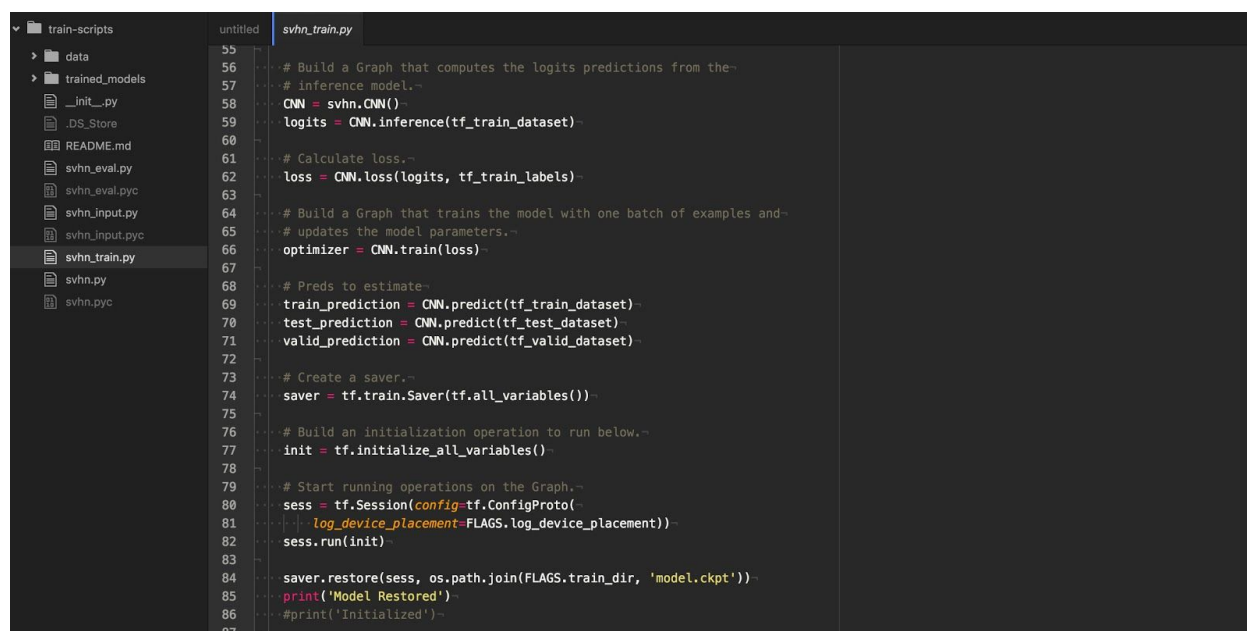
svhn.py: this script is the core and define a CNN class that allows to create ConvNets Model objects. This class has the following implemented functions: **inference**, **loss**, **train** and **predict**. Besides that, this script acts like an interface between the svhn_eval, svhn_input and the svhn_train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

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 the model and save the results properly taking advantage of the Public Cloud Computation Capacity.

svhn_eval.py: the only function that is defined in this script is the accuracy function. In this case it wasn't defined the periodic validations. 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. 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

```

Image 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. It was downloaded and installed Python and TensorFlow on the machine.

Then, the scripts were 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%**.

The screenshot shows the Google Cloud Platform console with the 'Compute Engine' section selected. The 'VM instances' page displays the details for a specific instance. The instance was created on July 27, 2016, at 2:44:11 PM. It has no tags and is using the 'n1-highcpu-16' machine type, which provides 16 vCPUs and 14.4 GB of memory. The CPU platform is 'Intel Haswell'. The instance is located in the 'us-central1-b' zone. Its external IP is 130.211.138.242 (ephemeral) and its internal IP is 10.128.0.2. IP forwarding is turned off. The boot disk is named 'tensorflow', has a size of 100 GB, is a standard persistent disk, and is in 'Boot, read/write' mode.

Name	Size (GB)	Type	Mode
tensorflow	100	Standard persistent disk	Boot, read/write

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

The screenshot shows the Google Cloud Platform console with the 'Compute Engine' section selected. The 'VM instances' page displays a graph of CPU utilization for the 'tensorflow' instance. The graph shows the percentage of CPU usage over time, with a significant spike occurring around July 28, 6:38 PM. The current CPU utilization is 0.117. Below the graph, a table lists the instance details.

Name	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	Connect
tensorflow	us-central1-b	16 vCPUs, 14.4 GB			10.128.0.2	130.211.138.242	SSH

Image 26. The VM instances running the script

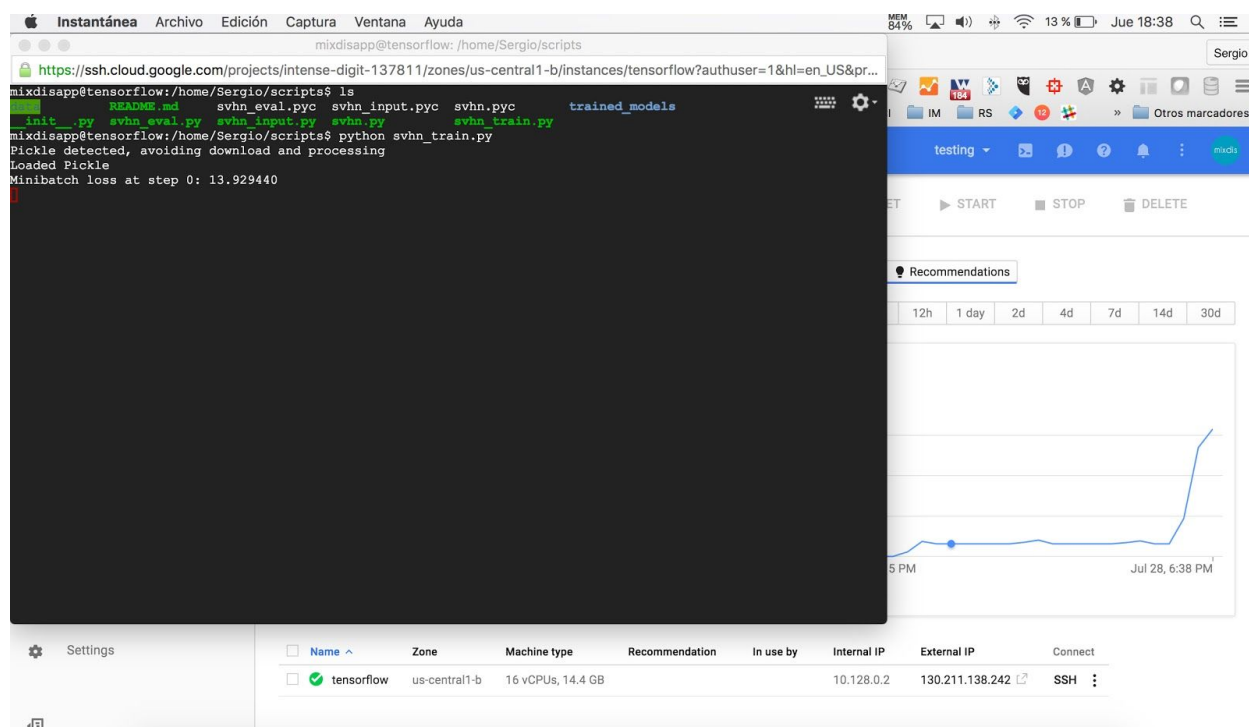


Image 27. First steps of the training

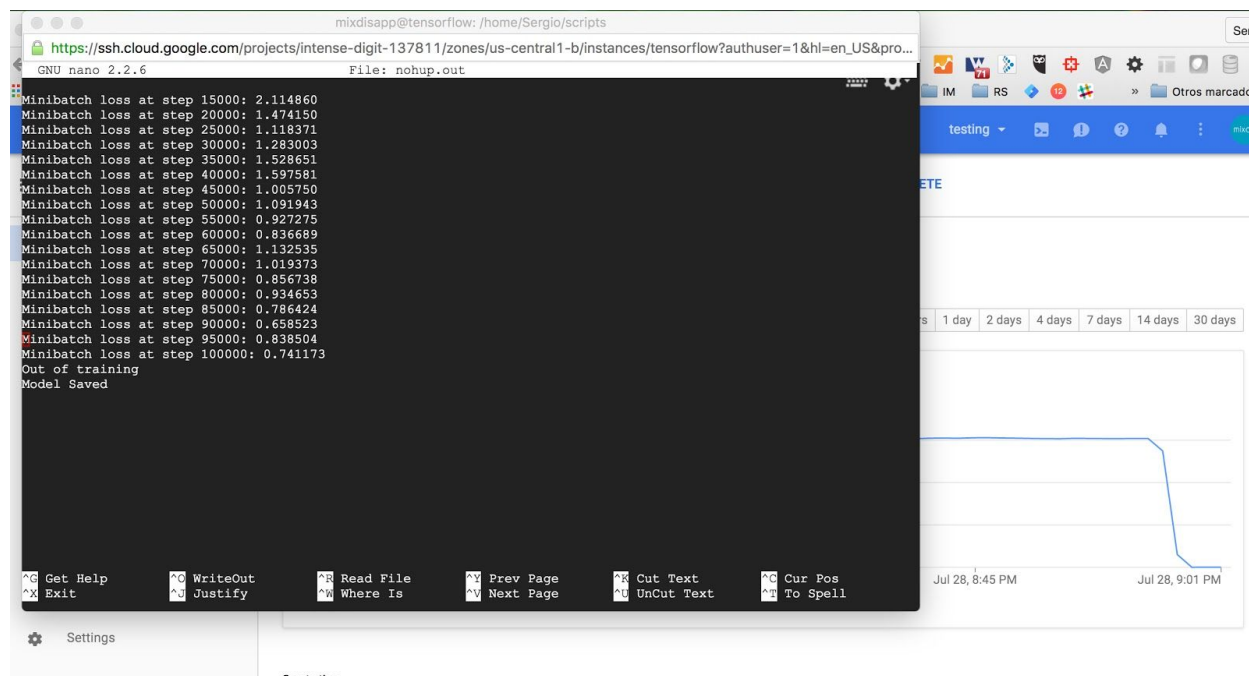


Image 28. The end of the first 100k steps. The model was trained 200k steps.

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

Build and Deploy a Web App on Heroku

At the moment the model was well trained on Google Cloud Platform, some tests and evaluations were implemented with the objective of evaluate the final performance of the model.

Besides that it's interested to convert this project in something more practical, that could show in a better way the main goal of it.

For that reason it was built a web application 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. To process an image (from the SVHN dataset) in real time, and predict the real number of the sequence in the image based on the model explained in this project.

The first step was to define correctly the limits of the web app. It doesn't allow to upload any kind of image and instead of that, it actually shows two hundred images from the test dataset in the web app.

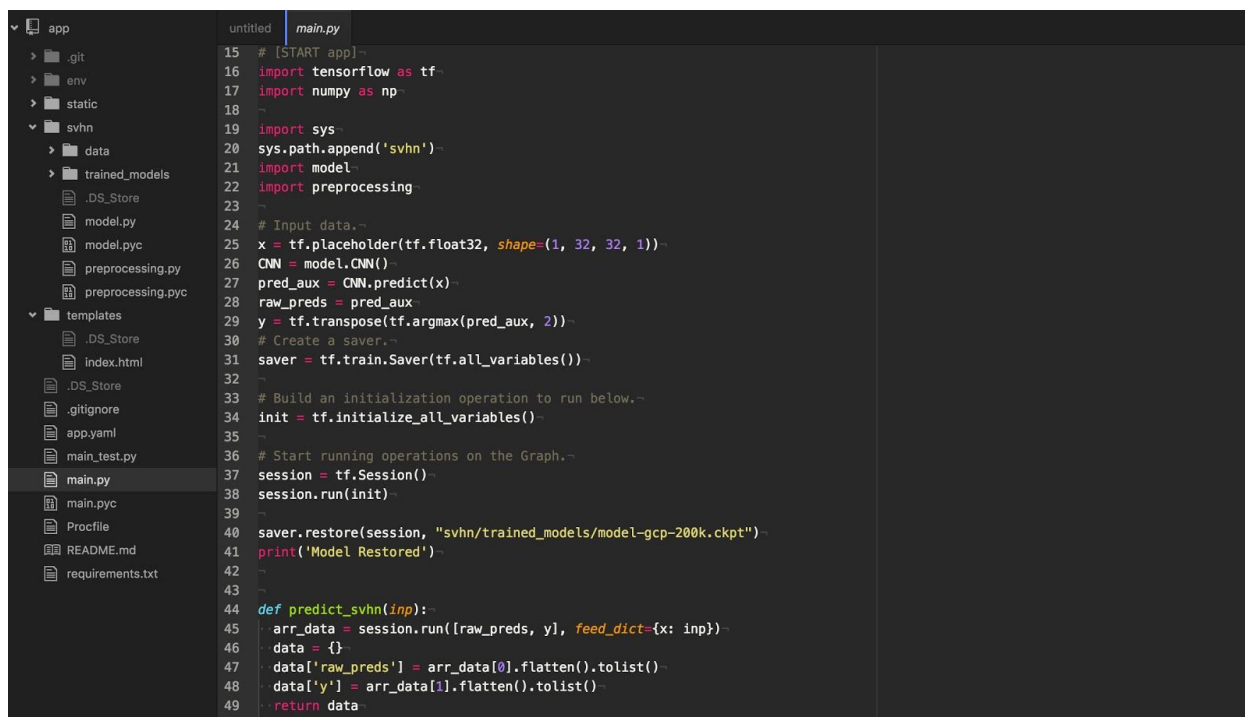
The user should select one of them and requests the prediction. In real time, the Backend gets the metadata information of the selected image from a preprocessed pickle, and makes a prediction.

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 was properly preprocessed, it was the time to think in how to built the web app.

It was used Flask 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.



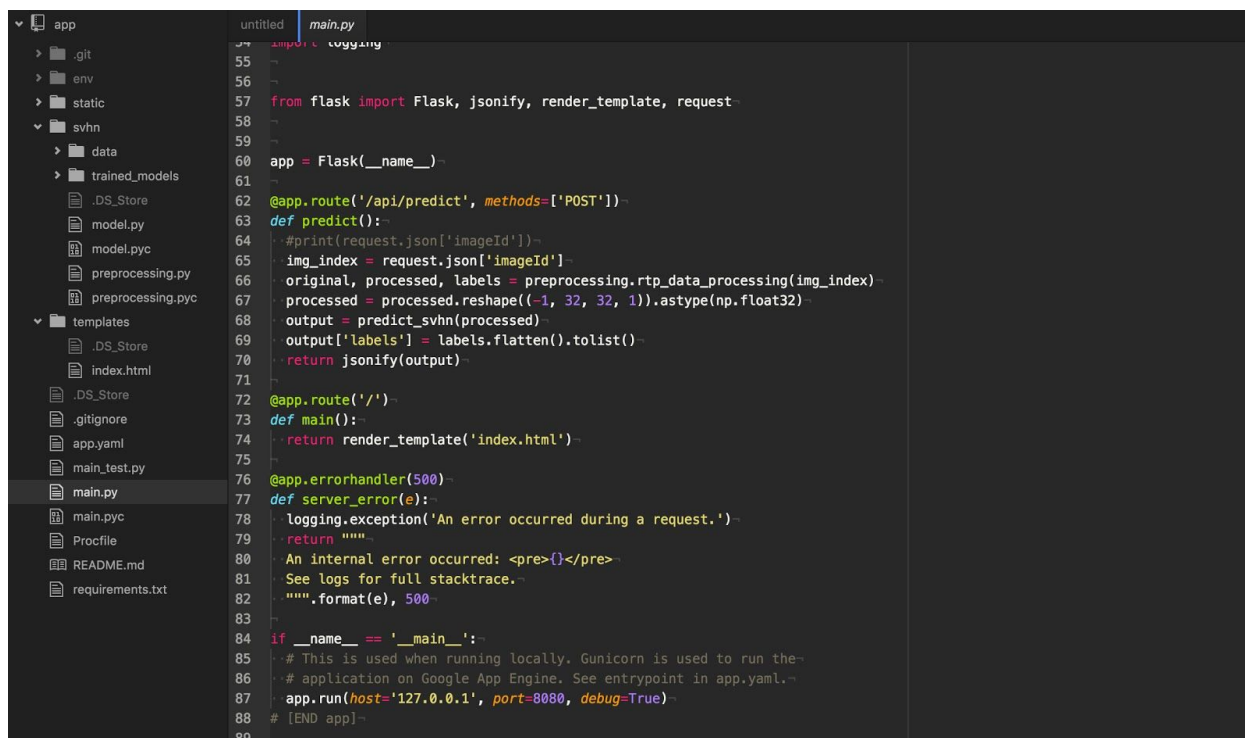
```

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_svhv(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

```

Image 29. main.py script (TF code)

It was created an endpoints that exposes the prediction functionality. In this case the graph only receives one sample at each time representing only the image selected by the user.



```

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_svhv(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     """~
83     .format(e), 500~
84 ~
85 if __name__ == '__main__':~
86     # This is used when running locally. Gunicorn is used to run the~
87     # application on Google App Engine. See entrypoint in app.yaml.~
88     app.run(host='127.0.0.1', port=8080, debug=True)~
89 ~
90 # [END app]~

```

Image 30. main.py script (Flask code)

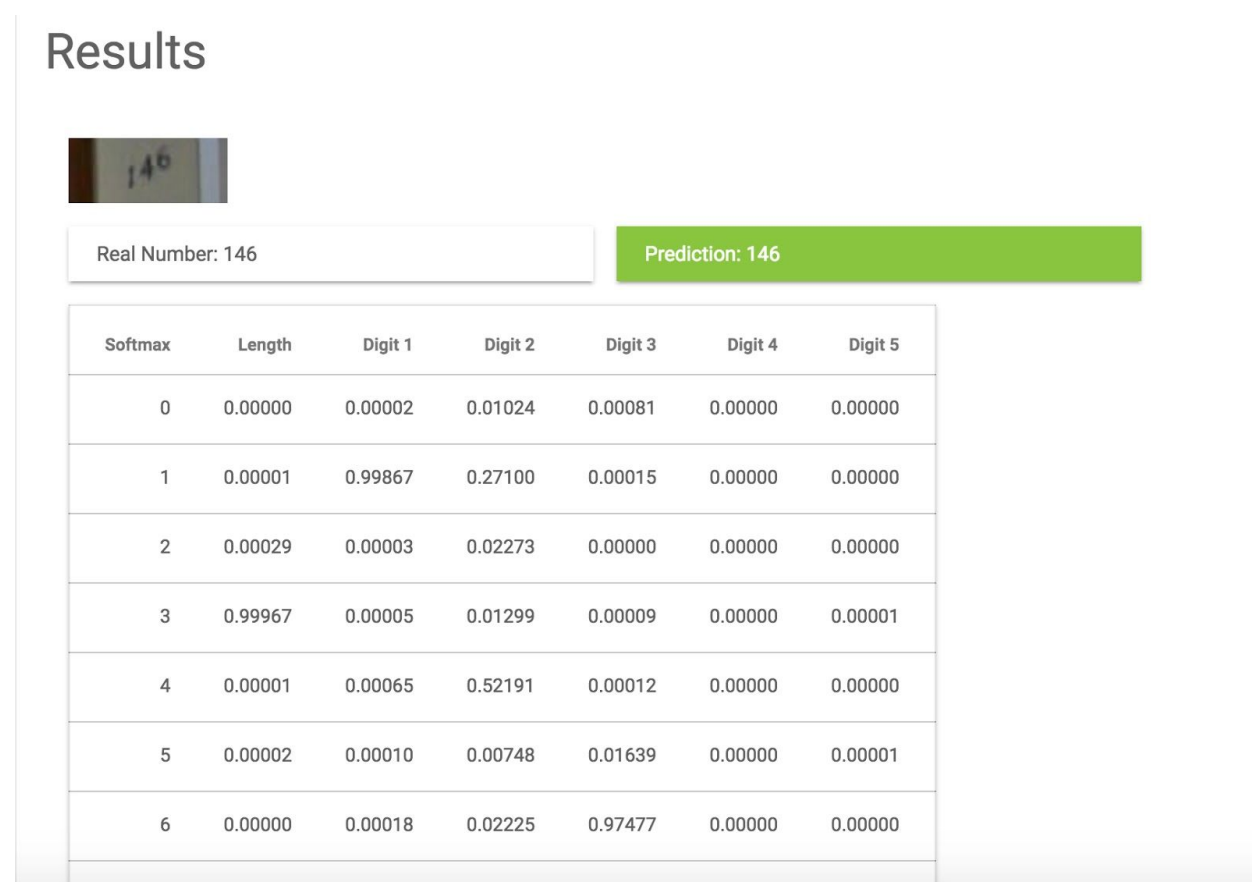
Of course it was necessary to define and code the preprocessing and the svhn modules. It's not very interesting to explained it in detail because the important concepts are properly explained in the previous sections. However the code is entirely public.

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 it's available in the following url:

<https://immense-dusk-10380.herokuapp.com/>



The screenshot shows a web application interface for digit recognition. At the top, the word "Results" is displayed. Below it, there is a small image of a handwritten digit "146". Underneath the image, there are two boxes: "Real Number: 146" and "Prediction: 146". Below these boxes is a table showing the softmax probabilities for each digit from 0 to 6.

Softmax	Length	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5
0	0.00000	0.00002	0.01024	0.00081	0.00000	0.00000
1	0.00001	0.99867	0.27100	0.00015	0.00000	0.00000
2	0.00029	0.00003	0.02273	0.00000	0.00000	0.00000
3	0.99967	0.00005	0.01299	0.00009	0.00000	0.00001
4	0.00001	0.00065	0.52191	0.00012	0.00000	0.00000
5	0.00002	0.00010	0.00748	0.01639	0.00000	0.00001
6	0.00000	0.00018	0.02225	0.97477	0.00000	0.00000

Image 31. Web app results page

Results

Model Evaluation and Validation

In this section, the final model and any supporting qualities should be evaluated in detail. It should be clear how the final model was derived and why this model was chosen. In addition, some type of analysis should be used to validate the robustness of this model and its solution, such as manipulating the input data or environment to see how the model's solution is affected (this is called sensitivity analysis). Questions to ask yourself when writing this section:

- *Is the final model reasonable and aligning with solution expectations? Are the final parameters of the model appropriate?*
- *Has the final model been tested with various inputs to evaluate whether the model generalizes well to unseen data?*
- *Is the model robust enough for the problem? Do small perturbations (changes) in training data or the input space greatly affect the results?*
- *Can results found from the model be trusted?*

Justification

In this section, your model's final solution and its results should be compared to the benchmark you established earlier in the project using some type of statistical analysis. You should also justify whether these results and the solution are significant enough to have solved the problem posed in the project. Questions to ask yourself when writing this section:

- *Are the final results found stronger than the benchmark result reported earlier?*
- *Have you thoroughly analyzed and discussed the final solution?*
- *Is the final solution significant enough to have solved the problem?*

Conclusion

Free-Form Visualization

In this section, you will need to provide some form of visualization that emphasizes an important quality about the project. It is much more free-form, but should reasonably support a significant result or characteristic about the problem that you want to discuss. Questions to ask yourself when writing this section:

- *Have you visualized a relevant or important quality about the problem, dataset, input data, or results?*
- *Is the visualization thoroughly analyzed and discussed?*
- *If a plot is provided, are the axes, title, and datum clearly defined?*

Reflection

In this section, you will summarize the entire end-to-end problem solution and discuss one or two particular aspects of the project you found interesting or difficult. You are expected to reflect on the project as a whole to show that you have a firm understanding of the entire process employed in your work. Questions to ask yourself when writing this section:

- *Have you thoroughly summarized the entire process you used for this project?*
- *Were there any interesting aspects of the project?*
- *Were there any difficult aspects of the project?*
- *Does the final model and solution fit your expectations for the problem, and should it be used in a general setting to solve these types of problems?*

Improvement

In this section, you will need to provide discussion as to how one aspect of the implementation you designed could be improved. As an example, consider ways your implementation can be made more general, and what would need to be modified. You do not need to make this improvement, but the potential solutions resulting from these

changes are considered and compared/contrasted to your current solution. Questions to ask yourself when writing this section:

- *Are there further improvements that could be made on the algorithms or techniques you used in this project?*
- *Were there algorithms or techniques you researched that you did not know how to implement, but would consider using if you knew how?*
- *If you used your final solution as the new benchmark, do you think an even better solution exists?*

Before submitting your report, ask yourself...

Does the project report you've written follow a well-organized structure similar to that of the project template?

Is each section (particularly Analysis and Methodology) written in a clear, concise and specific fashion? Are there any ambiguous terms or phrases that need clarification?

Would the intended audience of your project be able to understand your analysis, methods, and results?

Have you properly proof-read your project report to assure there are minimal grammatical and spelling mistakes?

Are all the resources used for this project correctly cited and referenced?

Is the code that implements your solution easily readable and properly commented?

Does the code execute without error and produce results similar to those reported?