Deep Learning
Build a Live Camera App

Table of Contents

Software and Libraries

Ipykernel (4.3.1) Ipython (4.2.0) Ipython-genutils (0.1.0) Ipywidgets (5.1.4) Jupyter (1.0.0) Jupyter-client (4.2.2) Jupyter-console (4.1.1)	Jupyter-core (4.1.0) Matplotlib (1.3.1) Notebook (4.2.0) Numpy (1.11.0) Pandas (0.18.1) Pathlib2 (2.1.0) Pickleshare (0.7.2)	Pillow (2.3.0) Pyparsing (2.0.1) Python-apt (0.9.3.5ubuntu2) Python-dateutil (1.5) Scikit-learn (0.17.1) Scipy (0.13.3) Six (1.10.0)	Tensorflow (0.8.0)

Environment and Installation

After failing to install the necessary infrastructure on a Windows Virtual Machine (i.e. getting Docker to work). I decided to use a cloud solution to run Python and Tensorflow. For more details on the insulation steps and cost, please refer to Appendix A. The final technical environment specification is summarised in T*able0* below.

RAM	2.5GB
CPU	X1
Disk	10GB

Table0 – Environment specifications

Definition of Terms

Scalar: This is a single number. It could be of type *integer,float, decimal*, e.g.100.
Vector: This is an array of numbers, e.g. [1,2.3].
Matrix: This is a 2-D array of numbers, e.g. [[1,2,3],[4,5,6]].
Tensor: This is an array with more than two axes, e.g. [[[1],[2],[3]], [[4],[5],[6]]]
Dimension: In the reminder of this document, the term *n*-D is used interchangeably with the term *n*-dimension. They both refer to the dimension of a vector space. In case of a Python array, the term dimension is related to the concept of rank. A 2-dimensional (2-D) array is an array of rank 2. In the case of an array defined of shape(2,3) , e.g. [[0,0,0], [1,1,1]], the rank is set to 2. This is equivalent to stating the length of the array is 2.
In Tensorflow, the rank/dimension has a different meaning to the one of an array. The rank tensor rank represents the number of dimensions of a tensor.

Summary

Rank/Dimension	**Math entity**	**Example**
0	Scalar	s = 10
1	Vector	V = [1,2,3]
2	Matrix	M =[[1,2,3], [4,5,6]]
3	3-Tensor	T= [[[1],[2],[3]], [[4],[5],[6]]]
N	n-Tensor	T= [[[1],[2],[3]], [[4],[5],[6]], ……., [[100,101,102], …]

The Shape of tensor is defined as follows

Rank	**Shape**	**Dimension number**	**Example**	**Description**
0	[]	0-D	s= 1	A 0-D tensor, a scalar of value1
1	[D0]	1-D	t= [1]	A 1-D tensor, with a shape [1]
2	[D0,D1]	2-D	t= [1,2]	A 2-D tensor, with a shape [1,2]
3	[D0,D1,D3]	3-D	t= [1,2,3]	A 3-D tensor, with a shape [1, 3]
N	[D0,D1,…, Dn-1]	n-D	t= [D0,…, Dn-1]	A n-D tensor, with a shape [1, Dn-1]

Definition

Project Overview

The task is to build a live camera application simulator that interprets number strings in real-world images. The numbers are assumed to be positive integers. The aim of this project is to *Train* a model that can decode sequences of digits from natural images and create an application that prints the numbers, it sees in real time. The *Training* and *Test* datasets used in this paper come from *The Street View House Number* (a.k.a. *SVHN*). It was originally manufactured by Netzer Y. et al (2011). Due to time constraints and extra complexities relating to the implementation of a fully functional user interface, this project simply focuses on implementing the application in an iPython Notebook. For ease of development and deployment, the code was developed in the Cloud9 virtual environment. However, the user can install all required dependencies on another dedicated environment and run the iPython code.

Project Statement

Identifying multi-digit numbers in pictures is an important task for modern robots, such as self-driving cars or package delivery drones. They need to promptly identify geo-located sign where number are present, turn the image into pixels representations, crop sequence of digits from the image and 'translate' the pixel sequence into positive integers. This paper solely focuses on the last step that is the prediction of a positive integer given an image (assuming the image contains at least one digit). One of the challenges resides in the image quality (e.g. shadow, blur, tilt etc.), the presence of non-standard fonts or broken outlines.
A number of solutions to this problem are available in the literature. In their research, Netzer Y et al. (2011) compare a number of approaches using supervised machine learning algorithms. They conclude that the K-Means algorithm is the most accurate, with 90.6% success rate. The Tensorflow (2016) group uses a multinomial logistic classifier (*softmax* regression) and achieve 92% of success rate. Goodfellow, Ian J., et al. (2013) present of a deep *convolutional* neural solution which operates directly on the whole image pixels. This design reaches over 94% accuracy in recognizing complete street numbers.
The solution proposed in this paper does not follow a Deep Neural Network design, as:

- It has some intrinsic scalability limitation, and
- The CNN is relatively hard to *Train*, and have many more parameters than fully connected networks with the same number of hidden units (c.f. UFDDL Tutorial).

Instead, the proposed solution implements as 4-layer *Convolutional Neural Network* (CNN). In a nutshell, the CNN consists of 3 *convolutional* layers feeding into a fully connected layer. The outline of the architecture is shown in *Fig.0* below:

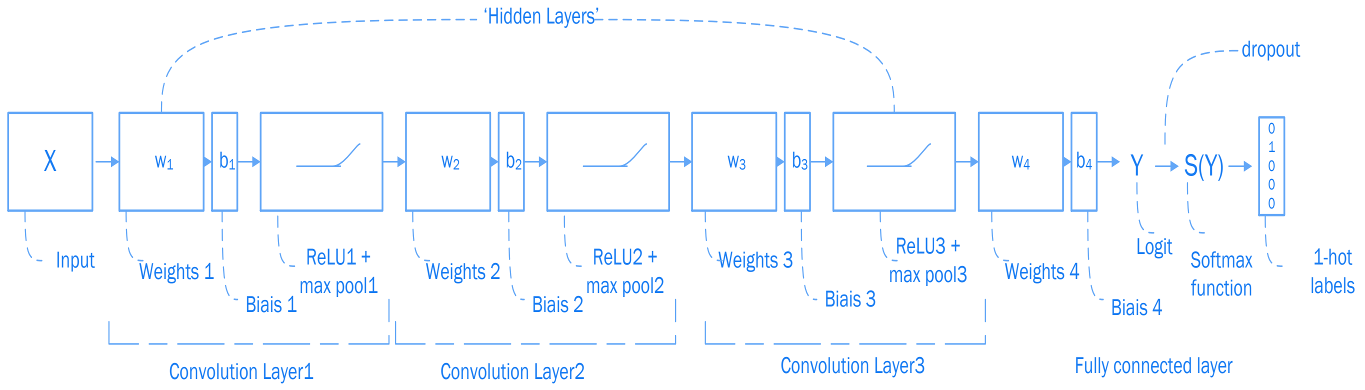

Fig.0: The Convolutional Neuronal Network (CNN) design

- The input data, in this case an *m* **x** *m* **x** *r* images, denoted by X, is fed in the first layer of the pipeline. In this context, *m* represents the height and width of the image, and *r* represents the number of channels. In the *SVHN* sample, the height and width are set to 32 and *r* is set to 3, as defined by the RGB colouring scheme.
- The input data is multiplied by a set of weights (*Weights1*) to which biases (*Bias1*) are added. These weights (a.k.a. the kernel filter) are of size *n* **x** *n* **x** *q*, where *n* is smaller than the dimension of the image and *q* can either be the same as the number of channels *r,* or smaller, and may vary for each kernel. The output of the kernel filtering is then passed through a rectified linear unit (*RelU1*), also described as a *hidden layer*. The ReLU makes the output non-linear and add a new point of tuning. As explained by Standford (2016), the advantages of using a *ReLU* function over *Sigmoid* or *Tanh* functions is that it was found to greatly accelerate the convergence of *stochastic gradient descent*. Furthermore, it usually involves inexpensive operations as the *ReLU* can be implemented by simply thresholding a matrix of activations at zero. The result of this operation produces a feature map of size *m* **-** *n* **+** 1**.** A subsample is then computed using a *max 2* **x** *2 pooling*. In this experiment, the pooling groups two contiguous maps to produce an output of the shape of the map that contains the maximum value for equivalent regions in each map. This is presented in *Fig.1* which was originally produced in UFDDL Tutorial.

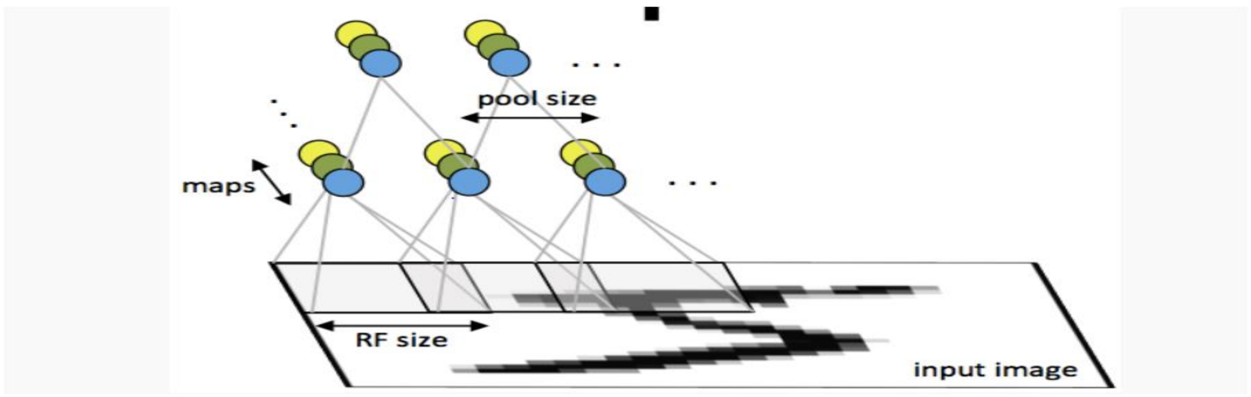

Fig.1: First layer of a Convolutional Neural Network with pooling. Units of the same colour have tied weights. Units of different colour represent different filter maps.

- The second layer consists of weights (*Weights2*) and biases (*Bias2)* applied to the layer 1 intermediate outputs, followed by *ReLU2* and *max pool2*.
- The third layer consists of weights (*Weights3*) and biases (*Bias3*) applied to the layer 2 intermediate outputs.
- The final layer (a.k.a. the fully connected layer) applies a regularisation filter over the third layer outputs. It is implemented by a *dropout* function. The purpose of the *dropout* function is to reduce *overfitting*. These results are then fed to a *softmax* function to generate the final label predictions.

Metrics

As described by Netzer Y et al. (2011), with rapid advances in mobile phone cameras and computation capabilities, there is a requirement for algorithms to detect objects (here numbers) with a degree of correctness at least equal to the one of human beings. Therefore, the accuracy, that is the mean of correct predictions, will be the main metrics that is used to measure the performance of the model.

Analysis

Data Exploration

The Street View House Numbers (*SVHN*) dataset was obtained from the http://ufldl.stanford.edu/housenumbers/ website. The source of this data was constructed from a large number of Street View images using a combination of automated algorithms, and the Amazon Mechanical Turk (AMT) framework. The AMT was used to localise and record single digits for the purpose of labelling. Netzer Y et al. (2011) "downloaded a very large set of images from urban areas in various countries. From these randomly selected images, the house-number patches were extracted using a dedicated sliding window house-numbers detector (…) using a low threshold
on the detector's confidence score in order to get a varied, unbiased dataset of house-number signs.
These low precision detections were screened and transcribed by AMT workers."
For the purpose of this paper's experiment, we use a subset of the original 600,000 labelled characters:

- Full Numbers *Training* and *Test* datasets (train.tar.gaz/test.tar.gaz) - this shows the images corresponding to the original, variable-resolution and colour of the house-number images as they appeared in the image file. These images are not being used by the predictor algorithm directly, due to the heterogeneous nature of the image size. It is solely be used by the user interface to display the original image to the end user. A sample of these images can be seen in *Fig.2* below:

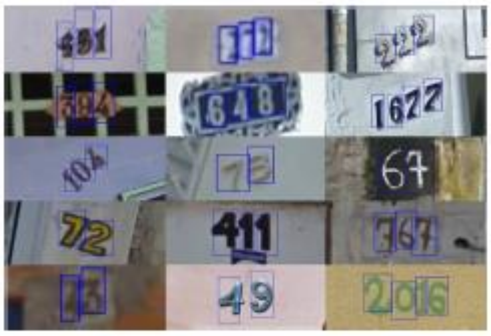

Fig.2: Sample SVHM *images*

Alongside each *Training* and *Test* sets, a *Training* and *Test* mapping table is present. For more details on the generation of this file, please refer to Appendix C. The file lists the following attributes, for each record in the mapping table:
  - the file name of the original image
  - the label each digit presents in the image
  - the left and top position of the digit on the image
  - the width and height of the digit

A record consists of the image name, the digit label, the position and the size. As shown in *Table1*, the image name *2.png* containing the number 210, highlighted in grey. This number is recorded on three rows. Each row contains the digit size and position details against the same image name (*2.png*).
As indicated by Netzer Y et al. (2011), it is interesting to note that, digits from *1* to *9* are set to label *1* to label 9. The digit 0 is reserved for label 10. This exception has no impact on the algorithm as, as we describe in a later section, the labels are turned into *one-hot* labels (i.e. a collection of digits set to *0* or *1*). An example is provided below:

FileName,	DigitLabel,	Left,	Top,	Width,	Height	
1.png,	5,	43,	7,	19,	30	=> image showing number 5
2.png,	2,	99,	5,	14,	23	=> image showing number *210*
2.png,	1,	114,	8,	8,	23	=> image showing number *210*
2.png,	10,	121,	6,	12,	23	=> image showing number *210*

Table1 – Extract of the Imagine Mapping table

- Cropped Digits – each digit of each image has been collected and resized to a 32**x**32 pixels format (MNIST-like format), ensuring no aspect ratio distortions. For the purpose of this experiment, we only retain the following two subsets:
  - *SVHN Training* (train_32x32.mat) contains 73,257 digits for training, which corresponds to 33,402 images (c.f. Appendix B).
  - *SVHN Test* (test_32x32.mat) contains 26,032 digits for testing, which corresponds to 13,068 images (c.f. Appendix B).

The proposed CCN algorithm uses these datasets for establishing its predictions.

As pointed out by the above mentioned authors: "The *SVHN* dataset, compared to many existing benchmarks, hits an interesting test point: since most digits come from printed signs, the digits are artificial and designed to be easily read, yet they nevertheless show vast intra-class variations and include complex photometric distortions that make the recognition problem challenging for many of the same reasons as general-purpose object recognition or natural scene understanding. "

Table2 shows the statistics for all *Training* images. The left and top positions of a number in an image showing respectively a mean of 58.40px (with a standard deviation of 41.90px) and 11.88 px (with a standard deviation of 14.46 px). This implies a large bias in the data. This is reinforced by the min (left =41.90 px, top=0.00 px) and max (left =618.00 px, top=219.00 px) values compared to the 25% (left =30.00 px, top=4.00 px) and 75% (left =73.00 px, top=15.00 px) percentiles which are very much apart. Furthermore, the left and top box plots show a large number of outliers. The same analysis and conclusion can be drawn from Table 2 relative to the height and width. They show respectively a mean of 33.86px and 16.65px for standard deviations of 18.60px and 10.68px. Their minimum and maximum values are far away from the 25% and 75% percentiles. *Fig.3* also shows a large number of outliers in both instances. This behaviour is to be expected as this is the main difficulty relating to this problem domain. The challenge lies in the ability of the algorithm to translate images of different shape, title, quality into computer readable integers.

	Left Position	**Top Position**	**Height**	**Width**
Count	73257.00	73257.00	73257.00	73257.00
Mean	58.40	11.88	33.86	16.65
Median	46.0	7.00	29.00	14.00
Std	41.90	14.46	18.60	10.68
Min	-1.00	0.00	9.00	1.00
25%	30.00	4.00	21.00	10.00
50%	46.00	7.00	29.00	14.00
75%	73.00	15.00	41.00	21.00
Max	618.00	219.00	403.00	207.00

Table2 – The SVHN Training image list statistics (all values unit - bar the count - are pixels)

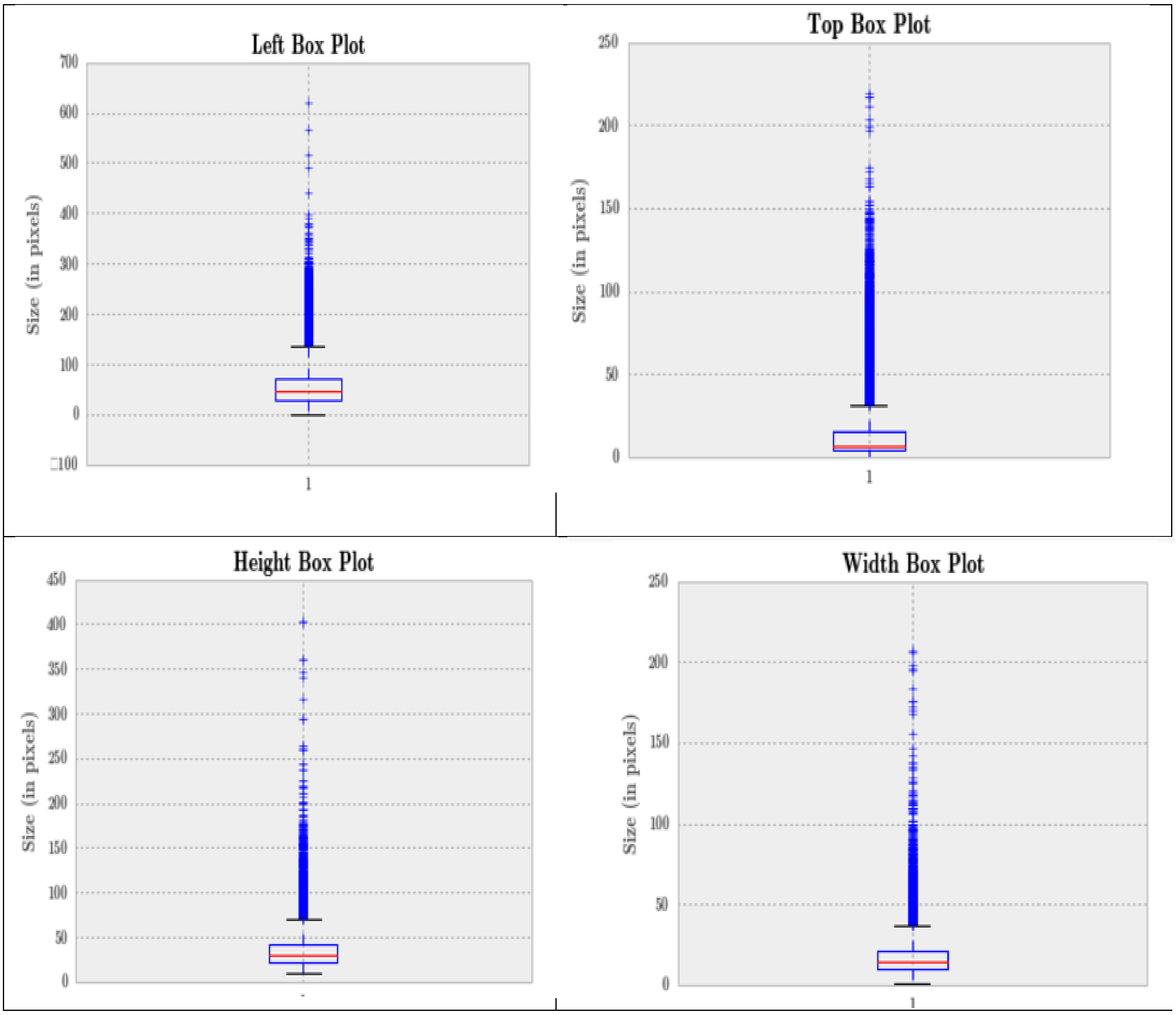

Fig.3: SVHM *Training images box plots.*

A thorough automated analysis of the data has been carried on both the *Training* image list and the *Training* mapping file.

- Step1 - The code checks for any empty or null data in the mapping file. It also verifies that the positions and dimensions data columns are integer.
- Step2 - The code runs a sanity check on the *Training* image list presents.

In both instances the code does not return any abnormality in the data. Therefore, no piece of data was rejected for this analysis. The code is available in the IPython note book. A cut down version is available below.

```
print("Start Data Prepocessing Validation...")
 (...)
"""Check for csv incorrect data abnormalities"""
try:
   print("Start checking for incorrect data in the csv file") 🡨 Step 1 - Check for empty or null data
   with open(train_raw_csv_directory + '/'+ train_csv, 'rb') as csvfile:
      spamreader = csv.reader(csvfile, delimiter=',', quotechar='|')
      #skip the header
      spamreader.next()
      for row in spamreader:
         for i in range(len(row)):
            if (i== 0 and isBlank(row[i])):
               print("Row:" +  str(row) + "is showing a empty string in the first column")
            if (i> 0):
               # if data is not castable to int (and implicity is empty or null) => then en exception is raised
               data = type(int(row[i]))
               if (i> 4):
                  data = type(int(row[i]))
                  if (data == 0):
                     print("Row:" +  str(row) + "is showing a width and/or height set to 0. This is not correct")

except ValueError as vae:
   print("This value contained in - col:" + i + " is not an integer", vae)
except Exception as e:
   print("An unexpected error occured", e)
print("End checking for incorrect data in the csv file")

"""Check for image abnormalities"""
try:
   print("Start for image abnormalities") 🡨 Step 2 - Sanity checks
   images = os.listdir(train_raw_img_directory)
   for index in range(len(images)):
      im=Image.open(train_raw_img_directory + '/' + images[index])
except IOError as ioe:
   print ("This is not a valid image: " + train_raw_img_directory + '/' + images[index], ioe)
except Exception as ex:
   print ("An unexpected error occured: " + train_raw_img_directory + '/' + images[index], ex)
print("End for image abnormalities")
print("End Data Prepocessing Validation...")
```

Exploratory Visualization

Fig.4 bottom histograms show that most of the *Training* dataset is composed of small images. The median heights and widths are respectively 29px and 14px, as displayed in *Table2* above. As mentioned by Netzer Y. et al. (2011), this is an important feature, as humans are remarkably good at identifying digits in low-resolution. Therefore, the algorithm should aim at mimicking human recognition performance.
Fig.4 top histograms imply that most of the digits are centred in the image. The median of the left and top positions are respectively 46px and 7px. This should help in increasing the algorithm *Training* accuracy.

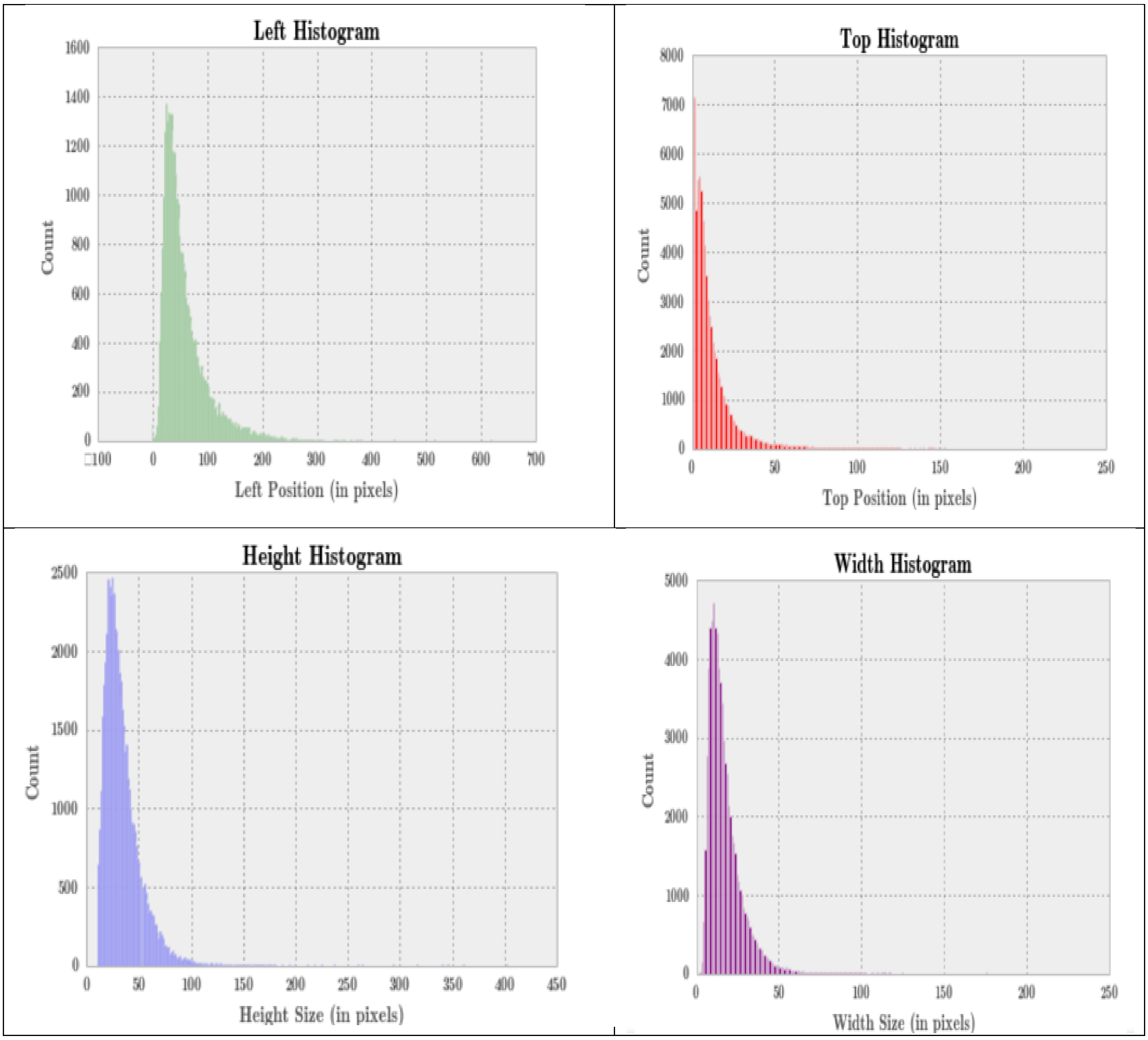

Fig.4: The SVHM *Training images histograms.*

Algorithms and Techniques

The following section describes the standard algorithms and techniques used in CNNs. In a nutshell, the input data (the *SVHN* images) are manipulated (*Fig.5*) to produce a set consumable by the *convolution* layers (*Fig.6*). The *convolution* layer starts with the *convolution* step, then generates the rectified linear units (*ReLUs*) and terminates with the pooling calculation. The later produce a set consumed by the final layer (*Fig.7*). The final layer is composed of a regularisation step and a *softmax* step. At the end of this pipeline, the *Training* step occurs. The remainder of this section details the mathematical representation of each of these steps.

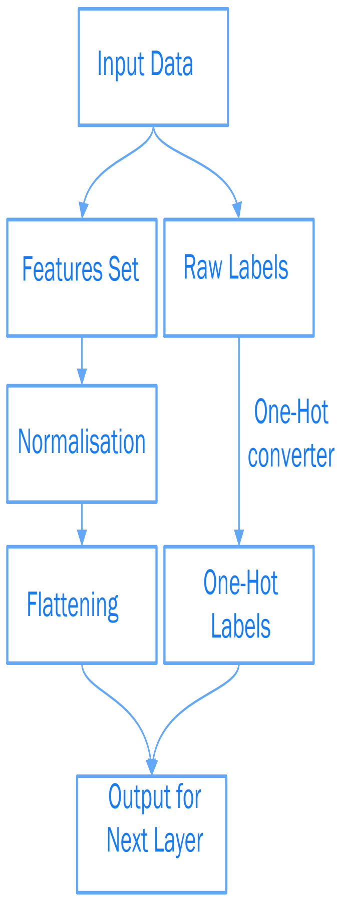

Fig.5: Input data conversion

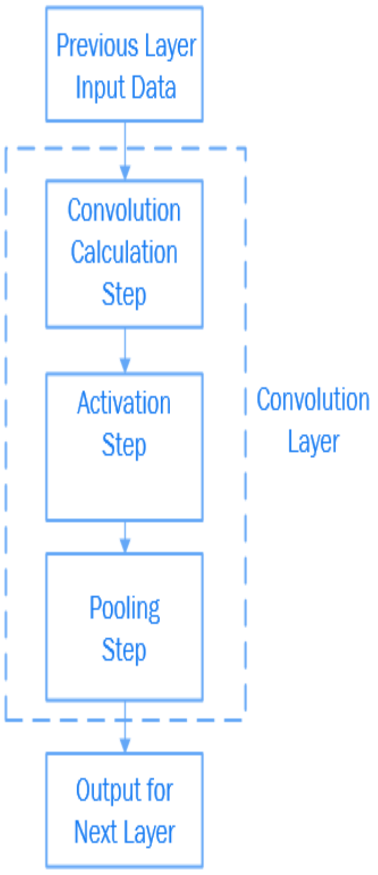

Fig.6: Steps involved in a convolution layer

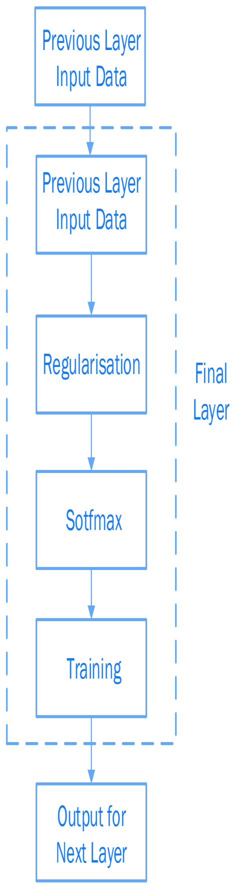

Fig.7: Steps involved in the final layer

Input Data Manipulation Step

As shown in Fig. 5 the input is broken in two categories:

- The feature set – This is a 3-D images representation. Each dimension corresponds to a number in pixels ranging from 0 to 255. The dataset is first normalised, where the *RBG* colours are divided by 255, then it is flattened into an array of 3*32*32 = 3072 numbers. There is a loss in terms of information relation to the 2-D image. However, the *softmax* does not use this data to establish its probability. Therefore, this simplification step is acceptable. The result is a tensor of a shape [73257, 3072] of integers. 73257 is the number of images.
- The raw labels – Each image label is *one-hot* labels converted, i.e. transformed into a list of 10 items set to 0 or 1. This process is required to improve prediction accuracy. This produces a list of labels of a shape [73257,10].

Convolution Step

At the heart of the *Convolutional* neural network (LeCun,1989) lies the *convolution* mathematical formula (Goodfellow, Ian J., et al, 2016). It is typically denoted as follows:

$s(t) = (x * w)(t) = \int x(a)\, w(t-a)\, da$

(x*w)(t) is a compact representation of the formula using a primitive.

S(t) :	The output of the *convolution equation,* a.k.a. the *feature map.*
x * w:	The *convolution* of x and w is written x * w, using an asterisk. It is defined as the integral of the product of the two functions after one is reversed and shifted (Wikipedia, *Convolution*).
x:	It represents the multidimensional array of input data (in this case the images pixels matrix 32*32*3).
w:	This is the *kernel*. This is a multidimensional array of parameters that are adapted by the learning algorithm. These arrays are known as tensors. It refers to a valid probability function, *w* needs to be set to 0 for all negative arguments, or it will look in the future. *W* represents the weighted average of the function *x(a)* at the moment *t*, where the weighting is given by *w(t)*, simply shifted by amount *a* . As *t* changes, the weighting function emphasizes different parts of the input function (Wikipedia, *Convolution*).
a:	This is the age of the measurement.
t:	This is the current time.

The motivation for using a *convolution* algorithm is three folds (Goodfellow, Ian J., et al, 2016):

- Sparse interaction – this is obtained by making the kernel smaller than the input. In the case of a 3072 pixels image (32*32*3), it reduces the amount of data to focus on small amount of meaningful features (e.g. the edges). This in turn reduces the number of parameters, reduces the memory requirement and improves the algorithm speed.
- Parameter sharing – it refers to using the same parameter for more than one function in a model. In other words, instead of learning from many locations, we just learn from one. This also has a positive impact on the memory requirements.
- Equivariant representation – this means that when the input changes, the output changes in the same way. This is represented mathematically as f(g(x)) = g(f(x))

Parameter Name	***Default***	***Justification***
w -> std	0.1	This is the standard deviation of the probability function. It is set to 0.1to break potential hidden symmetry.

Convolutional Layer #	convolutional patch dimensions	input channels #	features computed #	Bias vector
1	5x5	1	32	32
2	5x5	32	64	64
3	5x5	64	128	128

Activation Step

An "activation function of a node defines the output of that node given an input or set of inputs." (Wikipedia, Activation Function). There are a number of action functions available (Academia, 2014), such as the *Binary Threshold* or the *Logistic Sigmoid* function. However, the most popular approach is the *ReLU*. It is defined as follows:

$f(x) = \max(0, x)$ where x represents the input data.

There is a number of implementation of the *ReLU* that are smooth approximation of the *ReLU* (such as the *Noisy ReLU, Leaky ReLU*, etc.). We will not look into the advantages and drawbacks of using these different methods. The *ReLU* can be graphed as shown in *Fig.8*. There are two main advantages of using the *ReLU*. First, it was found to greatly accelerate the convergence of *stochastic gradient descent* compared to the *Sigmoid/Tanh* functions (Krizhevsky et al., 2012). Second, compared to other activation functions, the *ReLU* is implemented by a straight forward mathematical formula that simply thresholds the matrix activation at zero.

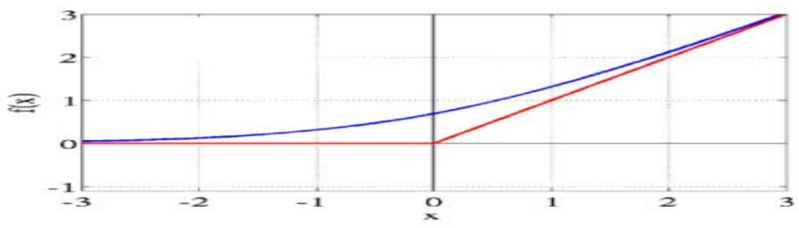

Fig.8: ReLU (in red) and approximation function (in blue)

Pooling Step

Once the non-linear activations have been produced by the *ReLU* (a.k.a. the detector stage), the pooling step replaces the *ReLU* input at a certain location with a summary statistic of its neighbourhood. There are mainly three popular pooling algorithms, the *average*, the *L2-norm* and *max pooling*. According to the literature (Wikipedia, *Convolutional Neural Network*) the *max pooling* is currently the most currently used. Fig. 9, borrowed from Wikipedia (*Convolutional Neural Network*), shows the *max pooling* in action. The filter is of size 2x2. The stride is set by 2 down samples at every depth slice in the input, and by 2 along both width and height. This removes 75% of the activations with no impact on the depth.

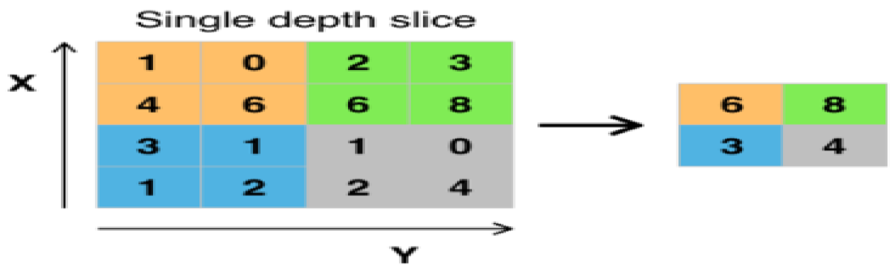

Fig.9: Max pooling with a 2x2 filter and stride = 2

Parameter Name	***Default***	***Justification***
stride	1	This number of pixel shifted each time the filter is moved on the image. This is the vanilla version.
padding	0	No padding
Pooling	2x2	2x2 is used across all layers
This set-up ensures the image output is the same size as the image input.		

The regularisation Step
As described by Srivastava N. et al (2014), there are a number of approaches to perform the *regularisation* step, such as “stopping the *Training* as soon as performance on a validation set starts to get worse, introducing weight penalties of various kinds such as *L1* and *L2* regularization and soft weight sharing (…)”. In this paper, we focus on the *dropout* technique as “it prevents *overfitting* and provides a way of approximately combining exponentially many different neural network architectures efficiently.” It randomly drops units (along with their connections) from the neural network during *Training*. This prevents units from co-adapting too much”.

Softmax Step
This step is necessary to calculate the probability of an image being one of several different labels. Tensorflow (2016) defines the *softmax* equation is as follows:

$$y = softmax(\sum_j Wi,j \;\; xi + bi\,)$$

Wi,j:	It represents the weight sum of the pixels intensities. A negative weight means the pixel has a high intensity; therefore the evidence goes against the image being in that class. Positive evidence goes in favour.
Xj:	It refers to the input data.
Bi:	It corresponds to bias.
Softmax:	It converts the evidence into probabilities.

The *softmax* function is written as:
$softmax = normalise((\exp(x))$

Training Step
The *Training* is the final step in the process.
It uses the *cross-entropy* (Tensorflow (2016) to measure how inefficient the predictions are for describing the truth.
The mathematical formula is written as:
$$Hy'(y) = -\sum_i y'i \;\; \log(yi\,)$$

y:	This is the predicted probability distribution
y’:	This is the true probability distribution coming from the *one-hot* vector labels input

Once, the algorithm knows what is expected, it can use a *backpropagation* algorithm to efficiently determine how the variables affect the cost that needs minimisation. At the end, we can apply an optimization algorithm to modify the variables and reduce the cost. There are a large number of them in the literature. Here we focus on the *gradient descent*. The gradient descent of a new point is represented as:

$x' \,=\, x -\, \varepsilon\forall x\, f(x)$

x’:	This is the position of the new point.
x:	This is the position of the current point.
ε:	This is step size or learning rate.
f(x):	This is the loss function.

Parameter Name	***Default***	***Justification***
ε	0.0001	This parameter determines how fast or slow we move towards the optimal weights. If the λ is very large the optimal solution is skipped. When is too small, too many iterations are required to converge to the optimal values. This is a user modifiable parameter that helps tuning the algorithm.

The graphical representation of the gradient descent is shown in *Fig.10* borrowed from Goodfellow, Ian J., et al (2016), p83.

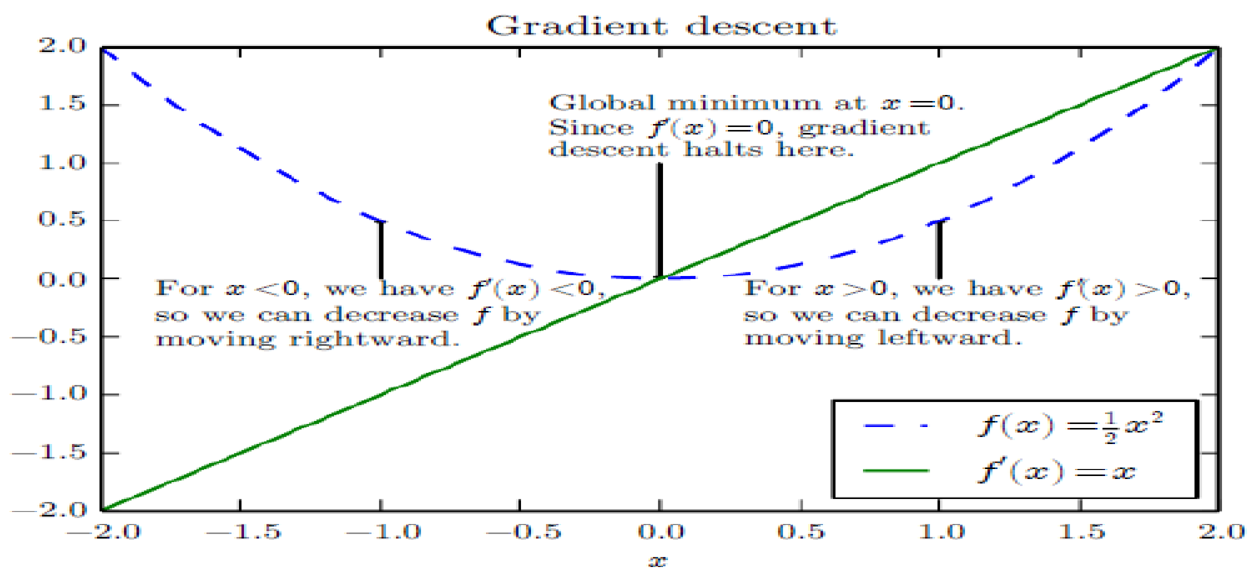

Fig.10: Gradient Descent

Benchmark

The K-means approach devised by Netzer Y et al. (2011) on this dataset achieved an accuracy of 90.6%. A more recent research from Sermanet P., Chintala S. and LeCun Y (2012), using a similar architecture to the one proposed in this paper, achieved 94.85% of accuracy. As Neural Networks goal is to mimic human brain cells interaction. It is therefore natural to use human recognition performance as benchmark for digit recognition. According to Netzer Y et al. (2011), the human brain accuracy this data is 98%. Consequently, the aim of this paper is to get a close as possible to the 98% limit and hopefully perform better than the other previously mentioned algorithms.

Algorithm	**Reference**	**Test Accuracy**
Binary Feature (WDCH)	Kimura F.,Wakabayashi T., Tsuruoka S., and Miyake Y. (1997)	63.30%
HOG	Dalal N. and Triggs B. (2005)	85.00%
Stacked Sparse Auto-Encoders	Coates A., Lee H., and Ng A. Y (2011)	89.70%
K-means	Netzer Y et al. (2011)	90.60%
ConvNet / MS / Average	Sermanet P., Chintala S. and LeCun Y (2012)	90.75%
ConvNet / MS / L2 (Smaller training)	Sermanet P., Chintala S. and LeCun Y (2012)	91.55%
ConvNet / SS / L2	Sermanet P., Chintala S. and LeCun Y (2012)	94.28%
ConvNet / MS / L12	Sermanet P., Chintala S. and LeCun Y (2012)	94.76%
ConvNet / MS / L4	Sermanet P., Chintala S. and LeCun Y (2012)	94.85%
Human Brain Accuracy	Netzer Y et al. (2011)	98.00%

Box1 – Benchmark list

Methodology

Data Pre-processing

As described in the above section *Input Data Manipulation Step*, the data flow is defined as follows.
First, the data is loaded from the store (held on the cloud VM). This happens in the *loadmat()* function shown in (1), in the below code snippet. Then, the *Training* and *Test* data are normalised (i.e. divided by 255). This happens in the function *rearrange()* shown in (2). The shape of the *Training/est* data remains unchanged [73257,32,32,3]. The first element corresponds to the length of the dataset, the others are the width /height and number of channels.
Second, the *Training and Test* labels arrays of arrays are collapsed into a 1-D array. As original labels are encoded from 1 to 10, this causes a problem for the *one-hot* labels generation. Therefore, labels are converted to integers between 0 and 9, prior to *one-hot* conversion. The new label 0 represents the original label 10; all the other labels between 1 and 9 remain unchanged. This is shown in (3). Finally, the *one-hot* labels are produced, see (4). Each image label is represented by a list of 10 items set to 0 or 1. This produces a list of labels of a shape [73257,10].

```
print("Start - Loading Massaged Trained and Test data ...")
(...)
def rearrange(x): 🡐 Detailed logic for (2)
  n = x.shape[-1]
  out = np.zeros((n, 32, 32, 3), dtype=np.float32)
  for i in xrange(n):
    for j in xrange(3):
      out[i, :, :, j] = x[:, :, j, i]
  return out / 255

def flatten_keeping_order(list_of_lists): 🡐 Detailed logic for (3)
  flattened  = np.asarray([val for sublist in list_of_lists for val in sublist])
  return flattened

def hot_encode_labels(labels): 🡐 Detailed logic for (4)
  n = len(labels)
  one_hot_labels = np.zeros((n, 10))
  for i in xrange(n):
    one_hot_labels[i, labels[i]] = 1
  return one_hot_labels.astype(np.float32)

def get_train_features_and_one_hot_labels(train_file_full_path):
  train = loadmat(train_file_full_path) #73257 🡐 (1) Load data into memory
  x_train = rearrange(train['X'])  🡐 (2) Normalisation of the training dataset
  y_train_original = train['y']
  y_train = flatten_keeping_order(train['y']) -1 🡐 (3) Label flattening and reduced by 1
  del train #remove ref to train (not the content)
  y_train_one_hot = hot_encode_labels(y_train) 🡐 (4)One-hot label encoding
  return x_train,y_train_original, y_train, y_train_one_hot

def get_test_features_and_one_hot_labels(test_file_full_path):
  test = loadmat(test_file_full_path) 🡐 (1) Load data into memory
  x_test  = rearrange(test['X']) 🡐 (2) Normalisation of the test dataset
  y_test  = flatten_keeping_order(test['y']) -1 🡐 (3) Label flattening and reduced by 1
  del test  #remove ref to test (not the content)
  y_test_one_hot  = hot_encode_labels(y_test) 🡐 (4) One-hot label encoding
  return x_test,y_test,y_test_one_hot
(...)
print ('Get the features and one hot labels...') 🡐 Extract training/test data and labels
x_train,y_train_original, y_train, y_train_one_hot = get_train_features_and_one_hot_labels('svhn_data/train_32x32.mat')
x_test,y_test,y_test_one_hot = get_test_features_and_one_hot_labels('svhn_data/test_32x32.mat')
print("End - Loading Massaged Trained and Test data ...")
```

Implementation

The above section dealt with the creation of *Training/Test* data and labels compatible with the generation of CNNs in Tensorflow. This section details the technical implementation of the 4-layers CNNs pipeline in Tensorflow. One point to note: each *convolution* layer is composed of a *convolution*, activation and a max pooling step. The variable corresponding to these calculations are noted respectively conv1, conv2, conv22 / conv1_relu, conv2_relu, conv22_relu max_pool_1, max_pool_2, max_pool_22. In the remainder of this section, only the conv1 / conv1_relu and max_pool_1 are explained. The others follow the same principle.

Parameters Initialisation

Prior to entering the *convolutional* pipeline, the data and placeholders variables need to be set for the experiment. Step (1) consists of resizing the *Training/Test* data to the length required for the experiment. There are currently 73,257 *Training* data and 26,032 *Test* data. These are user definable variables that can be increased or decreased to fit with the experiment and/or environmental specification at hand.

x_train	A 4-D tensor of shape [73257, 32, 32, 3]
y_test	A 4-D tensor of shape [73257, 32, 32, 3]
y_train_one_hot	A 2-D tensor of shape [73257, 10]
x_test	A 4-D tensor of shape [26032, 32, 32, 3]
y_test	A 1-D tensor of shape [26032]
y_test_one_hot	A 2-D tensor of shape [26032, 10]

Step (2) relates to the instantiation and initialisation of the Weights and Biases for each *convolutional* layer. The Weights initialisations (e.g. w1_init) are 4-D tensors for each CNN. They represent the width, length, colour channel and number of features map. The shapes are [5,5,3,32], [5,5,3,64] and [5,5,3,128] respectively for each of the CNN layer.
The Biases initialisations are 1-D tensors of shaped [32],[64] and [128] respectively for each of the CNN layer.
Step (3) keeps the data and label tensor dimension constant (*batch_sz =500*). Consequently, the *Training* and prediction steps happen on sample instead of the full selected size. See section *Training and Evaluation Steps* below for more information. This is a workaround to avoid RAM running out of memory.
The input images (*x*) consist of 4-D tensors of length *batch_sz* (500), with an image of width and height of 32 and 3 colour channels. The labels (*t*) are 2-D tensors where each row is a *one-hot* 10-D vector, indicating which digit class the corresponding *SVHN* image belongs to.
Step (4) defines the Weights (*w*) and Biases (*b*) for each *convolution* layer. The Weight variable is a 4-D tensor of width and height of 32 pixels, number of channels set to 3 and a length of 73,257. It is initialised to zero. The bias is a 1-D tensor with a shape [32], [64] and [128] depending on the CNN layer.

Step (5) flattens the third *convolution* layer output prior to being consumed by the final layer. Weights are initialised to random values, and biases are initialised to of zero.

Step (6) defines the Weight (*w*) and Bias (*b*) for the *Softmax* layer.

```
print("Start - Initialising and Defining the Model...")

"""Helpers"""
def init_filter(shape, poolsz):
   w = np.random.randn(*shape) / np.sqrt(np.prod(shape[:-1]) + shape[-1]*np.prod(shape[:-2] / np.prod(poolsz)))
   return w.astype(np.float32)

def conv_shape_weight_biais_init(filter_width, filter_height, num_color_channels, num_feature_maps, poolsz): ← Logic for (2)
   W_shape = (filter_width, filter_height, num_color_channels, num_feature_maps)
   W_init = init_filter(W_shape, poolsz)
   b_init = np.zeros(W_shape[-1], dtype=np.float32) # one bias per output feature map
   return W_shape, W_init,b_init

def init_weight_and_biais_variable(w_init, b_init): ← Logic for (4)
   w = tf.Variable(w_init.astype(np.float32))
   b = tf.Variable(b_init.astype(np.float32))
   return w,b
(…)

print ('Resize the training and test sets...')  ← (1) Resize the training/test dataset and labels to the required length
x_train = x_train[:train_max_size,]
x_test = x_test[:test_max_size,]
y_test = y_test[:test_max_size]
y_test_one_hot = y_test_one_hot[:test_max_size,]
y_train_one_hot = y_train_one_hot[:train_max_size,]
one_hot_len = len(y_test_one_hot)
one_hot_train_len = len(y_train_one_hot)

print ('Init placeholders and variables...')
k = len (y_test_one_hot.T)
n = x_train.shape[0]
n_batches = int(n / batch_sz)
# conv shape, weights and biais init ← (2) Instantiation and initialisation convolutional layer Weights and Biases
w1_shape, w1_init,b1_init = conv_shape_weight_biais_init (conv1_filter_width, conv1_filter_height, conv1_num_color_channels, 
conv1_num_feature_maps, poolsz)
w2_shape, w2_init,b2_init = conv_shape_weight_biais_init (conv2_filter_width, conv2_filter_height, conv2_num_color_channels, 
conv2_num_feature_maps, poolsz)
w22_shape, w22_init,b22_init = conv_shape_weight_biais_init (conv22_filter_width, conv22_filter_height, conv22_num_color_channels, 
conv22_num_feature_maps, poolsz)
# vanilla ANN weights init
w3_init = np.random.randn(w2_shape[-1]*8*4, m) / np.sqrt(w2_shape[-1]*8*4 + m) ← (5)
b3_init = np.zeros(m, dtype=np.float32)  ← (5)
w4_init = np.random.randn(m, k) / np.sqrt(m + k) ← (6)
b4_init = np.zeros(k, dtype=np.float32) ← (6)
# define variables and expressions
# using None as the first shape element takes up too much RAM unfortunately
x = tf.placeholder(tf.float32, shape=(batch_sz, 32, 32, 3), name='x') ← (3) The imagine tensor place holder
t = tf.placeholder(tf.float32, shape=(batch_sz, k), name='t') ← (3) The label tensor place holder
w1, b1 = init_weight_and_biais_variable (w1_init, b1_init) ← (4) Weight and Bias initialisation for the 1st convolution layer
w2, b2 = init_weight_and_biais_variable (w2_init, b2_init) ← (4) Weight and Bias initialisation for the 2nd convolution layer
w22, b22 = init_weight_and_biais_variable (w22_init, b22_init) ← (4) Weight and Bias initialisation for the 3rd convolution layer
w3, b3 = init_weight_and_biais_variable (w3_init, b3_init) ← (5) Weight and Bias initialisation for the final convolution layer
w4, b4 = init_weight_and_biais_variable (w4_init, b4_init) ← (6) Weight and Bias initialisation for the softmax layer

(…)
print("End - Initialising and Defining the Model...")
```

Convolution Step

The *convolution* layer starts with the *convolution* step implemented by the *conv2d* function in the code below. This function delegates into the Tensorflow *tf.nn.conv2d* function, which takes the following parameters:

- x: a tensor of shape [batch, in_height, in_width, in_channels]. In our case, the batch corresponds to the data shape dimension divided by the *batch_size*. For example 73,257 pieces of data divided by 500 (i.e. 73 batches) .The *in_height* and *in_width* represent the image height and width, both set to 32. The number of channels, *in_channels* is set to 3.
- w: the kernel tensor, a.k.a. the filter . The first layer contains 32 features for each 5x5 patch with a colour channel of 3. The shape is [5, 5, 1, 32]. Each successive layer doubles the number of features. Therefore, the second layer has 64 features (i.e. [5, 5, 1, 64]), and the third has 128 features (i.e. [5, 5, 1, 128]). All the rest being constant.
- strides: a 4-D tensor. It represents the stride of the sliding window for each dimension of input. The strides are set to [1, 1, 1, 1].
- Padding: a string that can be set to *SAME* or *VALID*. It represents the type of padding algorithm to use. Here it is set to *SAME*.

A bias is added to the output of the *tf.nn.conv2d* function. For more information on the bias, please see step (4) in the previous section named *Parameters Initialisation*. The result is a 4-D tensor output of shape [batch, in_height, in_width, in_channels], which serves as the input for the activation step (a.k.a. the ReLU).

```
print("Start - Initialising and Defining the Model...")
(…)
def conv2d(x, w, b, conv_strides, conv_padding):
   conv = tf.nn.conv2d(x, w, strides=conv_strides, padding=conv_padding)
   conv = tf.nn.bias_add(conv, b)
   return conv
(…)
conv1 =  conv2d (x, w1, b1, conv_strides, conv_padding)  🡐 Generate Convolution with bias
(…)
print("End - Initialising and Defining the Model...")
```

Activation Step

The *ReLU* computes the max of a 4-D Tensor of features of shape [batch, in_height, in_width, in_channels] or 0.

```
print("Start - Initialising and Defining the Model...")
(…)
def relu(conv):
   return tf.nn.relu(conv)
(…)
conv1_relu =  relu(conv1) 🡐 Produce the ReLU
 (…)
print("End - Initialising and Defining the Model...")
```

Pooling Step

This is the last step of a *convolution* layer. It calculates the *max pooling* on a *convolution* tensor. The parameters are the following:

- conv: a 4-D Tensor with shape [batch, height, width, channels] and type tf.float32.
- ksize: a list of integers with a length >= 4. This is the size of the window for each dimension of the tensor input. In this implementation, it is set to [1, 2, 2, 1] for each *convolution* layer.
- strides: a list of integers with length >= 4. This is the stride of the sliding window for each dimension of the tensor input. In our implementation, it is set to [1, 2, 2, 1] for each *convolution* layer.
- padding: a string, either *VALID* or *SAME*. In our implementation, it is set to *SAME*.

It returns a *max pooled* 0–D Tensor of type *tf.float32* . This is the final output of a *convolutional* layer.

```
print("Start - Initialising and Defining the Model...")
(...)
def max_pool(conv, pool_kernel_size, pool_strides, pool_padding  ):
   max_pool = tf.nn.max_pool(conv, ksize=pool_kernel_size, strides=pool_strides, padding=pool_padding)
   return max_pool
(...)
max_pool_1 = max_pool (conv1_relu, pool_kernel_size, pool_strides, pool_padding) 🡐 Max Pool Output
 (...)
print("End - Initialising and Defining the Model...")
```

As mentioned above, each *convolutional* layer output serves as an input for the next a *convolutional* layer. The full linkage of *convolutional* layer is shown below.

```
print("Start - Initialising and Defining the Model...")
(...)

print ('Define convolution layers...')
conv1 =  conv2d (x, w1, b1, conv_strides, conv_padding)
conv1_relu =  relu(conv1)
max_pool_1 = max_pool (conv1_relu, pool_kernel_size, pool_strides, pool_padding)  🡐 Convolutional Layer 1 Output

conv2 =  conv2d (max_pool_1, w2, b2, conv_strides, conv_padding)
conv2_relu = relu(conv2)
max_pool_2 = max_pool (conv2_relu, pool_kernel_size, pool_strides, pool_padding)  🡐 Convolutional Layer 2 Output

conv22 =  conv2d (max_pool_2, w22, b22, conv_strides, conv_padding)
conv22_relu = relu(conv22)
max_pool_22 = max_pool (conv22_relu, pool_kernel_size, pool_strides, pool_padding)  🡐 Convolutional Layer 3 Output

(...)
print("End - Initialising and Defining the Model...")
```

The final step consists of resizing the final *convolution* output into a shape that is consumable by the final layer.

```
print("Start - Initialising and Defining the Model...")
(...)

max_pool_22_shape = max_pool_22.get_shape().as_list()
max_pool_22_reshape = tf.reshape(max_pool_22, [max_pool_22_shape[0], np.prod(max_pool_22_shape[1:])]) 🡐 Convolutional Layer 3
                                                                                                       Output Reshape
 (...)
print("End - Initialising and Defining the Model...")
```

The original 32x32 image size has been reduced to a 4x4 shape as shown below:

- Height = initial Height / (filter Height)$^{\text{NumberOfConvolutedLayer}}$ => $4 = 32/(2^3)$
- Width = initial Width / (filter Width)$^{\text{NumberOfConvolutedLayer}}$ => $4 = 32/(2^3)$

The result is a 4x4 image size, which is processed in the final layer with 1024 neurons (1024=32*32) to allow processing of the entire image. For that, the previously reshaped *convolutional* output tensor is multiplied by a weight matrix, to which a bias is added. For more information on the weight and bias, see step 5 in the above section *Parameters Initialisation*.

```
print("Start - Initialising and Defining the Model...")
(…)
conv_final = tf.matmul(max_pool_3_reshape, w3) + b3 ← Final Convolutional Layer Output
 (…)
print("End - Initialising and Defining the Model...")
```

The regularisation Step
To reduce *overfitting*, the *dropout* is applied prior to the *softmax* function. The *dropout* placeholder, named *keep_prob*, is a variable that can be modified during the training or prediction phases. It represents the probability that a neuron's output is kept during *dropout*. For the training phase it is set to 0.5. In other words, during training, only 50% of the output is kept. For the *Training* and *Test* predictions, it is set to 1.0. The result is a 0-D Tensor of type *tf.float32*.

```
print("Start - Initialising and Defining the Model...")
(…)
keep_prob = tf.placeholder(tf.float32)
drop_out = tf.nn.dropout(conv_final, keep_prob) ← Regulaisation Output
 (…)
print("End - Initialising and Defining the Model...")
```

Softmax Step
The last step in the pipeline is the generation of the prediction regression line. For this, the Tensorflow native *dt.nn.softmax()* function is called. It takes the sum of the regularised input images multiplied by the weight matrix *w4* and add the *b4* bias. For more information on the weight and bias, see step 6 of section *Parameters Initialisation*. This computes the *softmax* probabilities that are assigned to each class. The result is a regression line.

```
print("Start - Initialising and Defining the Model...")
(…)
y_prediction = tf.nn.softmax(tf.matmul(drop_out, w4) + b4) ← Prediction Output
 (…)
print("End - Initialising and Defining the Model...")
```

Training and Evaluation Steps

Once the predictive regression line is built, the model is trained and then evaluated against expected results. The aim of the *Training* phase is to minimise the cost (a.k.a. loss) function using gradient descent. The model 'learns' and adjusts the *hyperparameters* during each *Training* iteration. The remainder of this section details the steps involved in generating loss and accuracy measures.

Step (1) – It defines he loss function as the *cross-entropy* between the logits and the model's label prediction. This is implemented by the Tensorflow tf.nn.softmax_*cross_entropy_with_logits()* function. It takes two parameters:

- y_prediction: unscaled log probabilities.
- t: the labels

It returns a total loss tensor.

Step (2) – The Tensorflow *RMSPropOptimizer ()* function belongs to the family of the *gradient descent* algorithm. According to the Tensorflow (2016) documentation, it operates by dividing the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. It takes three parameters:

- learning_rate: the weight of the negative gradient
- decay : the discount factor for the history/coming gradient
- momentum: the weight of the previous update. By smoothing the weight updates across iterations, the momentum tends to make deep learning with *RMSPropOptimizer* both more stable and faster.

Step (3) – This defines the prediction function.

The remainder of the implementation relies on a number of loops to produce the *Training* data prediction, the *Test* data evaluation as well the loss.

- The first loop (a top-level outer loop) relates to the number of iterations that need to be run in order to train the data. It is set to 20. In this case, the iteration and epoch are interchangeable. Each iteration uses the entire *Training* data set.
- The second loop (inner loop) relates to the number of batches (73,256/500=147) that need to be run through to train on all the *Training* set.
- The third loop iterates through the list of features length divided by the batch size (*batch_sz)*. As the features length increases by the batch size *(500)* for each iteration its parent loop, the ratio starts at 1 and get incremented by 1 (e.g. 1,2,3,4,….). This produces a row of result showing the error rate for each feature sub-batch (batch_sz =500*)*, in each batch (0,1, 2,3, …,147) for a given epoch.

Step (4) – The *Training* and *one-hot* label are resized to the *batch_sz* length (500). For each batch iteration, the size is kept constant and slides to the next 500 items. The first iteration uses the first 500 elements, the second the next 500, etc.

Step (5) – The *Training* data and loss data are generated. The *keep_prob* variable is set to 0.5. This means only 50% of the neurons' output are kept.

Step (6) – The loss data is store into a *.csv* file. This log is used at a later stage for statistics gathering and plotting purpose.

Step (7) – It calls the *get_predictions()* function twice. The first time on the *Training* data, the second time on the *Test* data. The accuracy of predicted labels is stored into a *.csv* file. These results serves as the basis for the model validation analysis.

Step (7.1) – It generates the predicted labels for a batch sample of the *Training* data
Step (7.2) – It produces the accuracy measure (i.e. 1-error) by comparing the predicted labels to the true label for the batch.

```
print("Start - Training and Measuring Model Accuracy...")

"""Helpers"""
def error_rate(p, t):
   return np.mean(p != t)

 (...)

def get_predictions(run_type, batch_sz, batch_sz_1, features, labels, labels_one_hot, predict_op_param):
   start_index = 0
   end_index = 0
   one_hot_len = len(labels_one_hot)
   while (start_index+batch_sz_1) <= one_hot_len:
      #due to RAM limitations we need to have a fixed size input  🡐 c.f. below section Implementation Complications and Workarounds
      #so as a result, we have this ugly total cost and prediction computation
      #we also need to have a fixed size test iput and then we average.
      end_index = start_index+batch_sz_1
      features_sample = features[start_index:end_index]
      labels_sample = labels[start_index:end_index]
      labels_one_hot_sample = labels_one_hot[start_index:end_index,]
      test_cost = 0
      prediction = np.zeros(len(features_sample))
      for kk in xrange(int(len(features_sample) / batch_sz)): 🡐 Third Loop
         features_batch = features_sample[kk*batch_sz:(kk*batch_sz + batch_sz),]
         labels_batch = labels_one_hot_sample[kk*batch_sz:(kk*batch_sz + batch_sz),]
         prediction[kk*batch_sz:(kk*batch_sz + batch_sz)] =
                                    session.run(predict_op_param, feed_dict={x: features_batch, keep_prob: 1.0})  🡐 (7.1) Get label prediction
      err = error_rate(prediction, labels_sample) #error calculation 🡐 (7.2) Calculate error
      print_info(run_type, results_directory, results_file_name, i,j,start_index, err,batch_start_time, full_debug_info) 🡐 Log data to CSV
      start_index = start_index + batch_sz_1 #reset the start_index
   return prediction, start_index

"""Training"""
#print the timing information on training completion
train_start_time = datetime.now()
train_start_time_display = train_start_time.strftime("%Y-%m-%d %H:%M:%S")
print ("\t Start time - " + train_start_time_display)
write_to_csv(results_directory, timings_file_name, ('Start_Time', 'End_Time'), (train_start_time_display, ''), 'a')
write_to_csv(results_directory, results_file_name, ('Type','Iteration_Id', 'Batch_Id','Test_Start_Index','Error_%','Accuracy_%','Start_Time',
'Current_Time','Elapsed_Time'), ( ), 'a')

#define the cost function that needs minimising during training
loss_op = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(y_prediction, t)) 🡐  (1) Loss Function
train_op = tf.train.RMSPropOptimizer(learning_rate=the_learning_rate,
               decay=the_decay, momentum=the_momentum).minimize(loss_op) 🡐 (2) Training / Loss Minimisation function
#define the prediction function
predict_op = tf.argmax(y_prediction, 1) 🡐 (3) Prediction function

init = tf.initialize_all_variables()
session = tf.Session()
session.run(init)
```

```
for i in xrange(max_iter): ← First Loop
   print ("\t Nb of batches: "+ str(n_batches))
   for j in xrange(n_batches): ← Second Loop
      batch_start_time = datetime.now()
      x_batch = x_train[j*batch_sz:(j*batch_sz + batch_sz),] ← (4) Training dataset resize
      y_batch = y_train_one_hot[j*batch_sz:(j*batch_sz + batch_sz),] ← (4) Label dataset resize
      if len(x_batch) == batch_sz:
         res = session.run([train_op,loss_op], feed_dict={x: x_batch, t: y_batch, keep_prob: drop_out_rate})
         loss = res[1]
         if j % print_period == 0:
            #print the iteration and batch id
            print_start_training_info(i,j)
            #store the loss info
            print_loss_info("Loss", results_directory, results_file_name, i,j,loss,batch_start_time, full_debug_info) ← (6) Loss data storage
            #calc train predictions and store training accuracy
            prediction_train, train_start_index =
                 get_predictions("Train", batch_sz, batch_sz, x_train, y_train, y_train_one_hot, predict_op) ← (7) Predicted label for a given
                                                                                                                  iteration and batch size
                                                                                                                  against training data

            #calc test predictions and store test accuracy
            prediction, test_start_index =
                get_predictions("Test", batch_sz, test_batch_sz, x_test, y_test, y_test_one_hot, predict_op) ← (7) Predicted label for a given
                                                                                                                  iteration and batch size
                                                                                                                  against test data
(…)
print("End - Training and Measuring Model Accuracy...")
```

The User Interface

At the end of a complete run, a minimalistic user interface is produced in the last section of the Python code. The code for this section can be found in Appendix G. In a nutshell, the program loops through a subset of the Training data, matches the corresponding image, print the 'True' label and the expected label (generated by the algorithm). The user interface also prints 'match', under the *Result* column header, when there is a match between the 'True' label and the predicted label and 'no match' when they are different. The below screen shot1[1] shows example success rates for image containing 1,2 and 3 digits.

```
*****************************************************************************************
```

```
Idx      Image Name      True Label      Predicted Label         Result
74       35.png                   6                 6                    match
Success Rate: 100.00%

*****************************************************************************************


*****************************************************************************************
```

```
Idx      Image Name      True Label      Predicted Label         Result
56       27.png                   5                 6                    no match
57       27.png                   6                 6                    match
Success Rate: 50.00%

*****************************************************************************************


*****************************************************************************************
```

```
Idx      Image Name      True Label      Predicted Label         Result
65       31.png                   6                 6                    match
66       31.png                   3                 6                    no match
67       31.png                   8                 6                    no match
Success Rate: 33.33%

*****************************************************************************************
```

[1] The success rates in this example are low because this report was produced from the algorithm trained with 5000 Training/1000 Test data and 1 epoch. The aim of this screen shot was solely to show the capability of the User Interface.

Implementation Complications and Workarounds
The inherent limitation of the compute power of this virtual machine, 1CPU of 2.5GB, meant that I was facing a design issue related to memory constraint. One of the weaknesses of CNNs is the generation of multiple large matrices for each layer computation. The more layers and complexity within the layers, the more data is held in memory by Tensorflow. My first implementation did not have the concept of nested loop for batching data. Without a sliding the window over the data, loading relatively small *Training* and *Test* sets of respectively 10,000 points and 5,000 points, broke the application with an *Out Of Memory Exception*.
Batching data has the advantage of *Training* and evaluating the model on a larger data training/test dataset without facing memory constraint and reducing *overfitting*. However, this approach as a number of drawbacks:

- It means the code is harder to read and maintain.
- The *Training/Test* accuracy and loss values need to be averaged across all batches, for each iteration (epoch), to produce the final accuracy and loss values. In our case, the averaging for each epoch is implemented as part of the validation exercise further down in this report.

The CPU limitation means it can take a few days for the model to *Train* and evaluate on a *Training* dataset of 73,257 elements. Therefore, it is more sensitive to potential network connection or server outage during a *Training* phase.

Refinement

Scenario runs description

Due to the large number of data, the limited amount of compute resource, and the time required to complete a scenario; I had to carry a number of experiments to optimise the scenario run parameters. The aim was to find the right balance between the 'optimised size of the *Training/Test* data set' and the 'optimised number of epochs'. In this context the 'optimised size of the *Training/Test* data set' means the highest possible *Training* accuracy given a number of epochs. The 'optimised number of epochs' means finding the smallest epochs range where the maximum accuracy is found, and then starts to plateau. I found that with a *Training/Test* size of respectively 18,250 and 6,500 the *Training* accuracy averaged 86% for 86 epochs. Slightly better results were obtained with a *Training/Test* size of respectively 36,628 and 13,016 with 20 epochs. Significantly better results were obtained using the full set of *Training/Test* data with 20 epochs. The results in this case were above the 90% mark. Therefore, a *Training/Test* size of respectively 36,628 and 13,016 with 20 epochs were chosen for the remainder of the analysis.

The scenario results

The process of refinement involves a selection of the most sensitive *hyperparameters*; namely the learning rate, the decay coefficient and the dropout strength, as well as as the kernel filters. I have also modified the current implementation, moving the design from a 4-CNN to a 3-CNN and 5-CNN layers. The purpose was to establish whether there could be a worthwhile 'time to complete vs accuracy' trade off. However, no attempt was made to replace the convolution/activation, polling or the regularisation algorithms implementation. This would have been an interesting piece of analysis. However, this was out scoped due to the potentially large impact on the current Python implementation.
Table4 summarises the scenario list selection. *Table6.1/6.2* show the impact of each scenario on the *Training/Testing* prediction accuracy, compared to the initial results. Each scenario implements the change of one *hyperparameter* at a time, keeping all other parameters in their initial state. *Table8* shows the final impact where the model runs with *hyperparameter* optimisation, all other things being equal.

Family	**Scenario Name**	**Description**	**Reason**
Base Case	Base Case	The initial case against which all other scenario results will compare against	
Scenario Cases	LR_0.001	Set *Learning Rate* to 0.001	An improper Learning Rate results in a model with low effective capacity.
	LR_0.01	Set *Learning Rate* to 0.01	
	LR_0.5	Set *Learning Rate* to 0.5	
	DC_0.5	Set *Weight Decay* coefficient to 0.9	The decay factor determines how long old gradients are kept for. The smaller the decay factor, the shorter the effective window.
	DC_0.9	Set *Weight Decay* coefficient to 0.5	
	DO_0.2	Set *Dropout Rate* to 0.2	Dropping units less often tends to increase *overfitting*.
	DO_0.7	Set *Dropout Rate* to 0.7	
	FLT_2	Set Kernel Height/Wight to 2	A Wider kernel results in a narrower output dimension capacity, reducing model capacity. Wider kernels require more memory for parameter storage.
	CNN_3	Implement a 3 CCN architecture	The time to completion of the 20 epochs and the accuracy rate rise with the increase number of *convoluted neurons* in the network.
	CNN_5	Implement a 5 CCN architecture	

Table4 – Scenario list

The Scenario logs

Running the scenario produces a number of log files:

- params_settings_[*scnearioName*]: It lists all the parameters for a given scenario in a key/value format. The below box shows a cut down version of the *params_settings_base_case* file:

```
Key,Value
full_debug_info,False
train_max_size,36628
test_max_size,13016
max_iter,20
(…)
conv_strides,"[1, 1, 1, 1]"
conv_padding,SAME
pool_kernel_size,"[1, 2, 2, 1]"
pool_strides,"[1, 2, 2, 1]"
pool_padding,SAME
(..)
conv1_num_color_channels,3
conv1_num_feature_maps,32
(…)
the_learning_rate,0.0001
the_decay,0.99
the_momentum,0.9
```

- results_[scenario*Name*]: This is the most important log file as it contains the loss and accuracy level for each sub-batch given an epoch (c.f. step (3) in *Training* and Evaluation Steps for more details). A cut down version of the *results_base_case* file is shown in the box below. The log file schema is described in *Table5*.

```
Type,Iteration_Id,Batch_Id,Test_Start_Index,Error_%,Accuracy_%,Start_Time,Current_Time,Elapsed_Time
Loss,0,0,-1,1150.63,-1,2016-08-08 06:48:22,2016-08-08 06:48:36,0:00:14.613426
Train,0,0,0,85.4,14.6,2016-08-08 06:48:22,2016-08-08 06:48:41,0:00:19.424868
 (…)
Train,0,130,72500,73.6,26.4,2016-08-08 11:41:45,2016-08-08 11:58:00,0:16:14.405743
Test,0,130,0,70.2,29.8,2016-08-08 11:41:45,2016-08-08 11:58:10,0:16:25.001471
 (…)
Test,0,140,25000,68.6,31.4,2016-08-08 12:06:43,2016-08-08 12:29:21,0:22:37.609526
Loss,1,0,-1,1075.68,-1,2016-08-08 12:30:44,2016-08-08 12:30:58,0:00:13.990246
Train,1,0,0,68.4,31.6,2016-08-08 12:30:44,2016-08-08 12:31:03,0:00:18.590291
```

Name	**Description**
Type	Type set to *Loss* corresponds to the row details relative to the Loss information. In this case the loss amount is recorded under the column heading *Accuracy_%.* Type set to *Train/Test* corresponds to the row details relative to *Training/Test* data.
Iteration_id	It corresponds to the epoch unique id, from 0 to 19, implying the use of 20 epochs.
Batch_id	The batch unique id
Test_Start_Index	The sub-batch unique id
Error_%	In the case of Type = *Train* or Type = *Test*, it represents the prediction error rate.
Accuracy_%	In the case of Type = *Train* or Type = *Test*, it is calculated as 100%-*error_%*
Start_Time	The start time of the calculation
Current_Time	The end time of the calculation
Elapsed_Time	The difference between the *Start_Time* and Current_Time

Table5 – Results log file description

- timings_[scenario *Name*]: this files records the start date/time of a scenario. It can be discarded.

The scenario s result

The aim of this section is to analyse the results produced by the scenario runs in order to optimise the final solution. The results_[scenario*Name]* *log files* contain the loss, accuracy results and timings for each sub-batch run. They were produced by running each scenario in parallel on different VM. The code version on each VM only differed by either:

- the hyperparameter modification, or
- in the case of the 3-CNN and 5-CNN layers, the code had to be extended to accommodate for the removal or addition of a *convolutional* neuron.

The results_[scenario*Name]* *log file*s were used to produce the below *Table6.1/6.2* and *Table8* . The code for generating these tables is described in Appendix D.

These results tables are divided in two sections. The top section contains the list of *hyperparameters* for each scenario. The *Base Case* corresponds to the version with the initial parameters set. The other columns represent a scenario where one of the *hyperparameters* was modified. There are highlighted in blue in the tables.

The bottom section lists the results obtained after each run. It displays the following results:

- Time to Complete: The time required to run the Loss and Training/Test accuracy calculations.
- *Training/Test* Accuracy (%): The *Training/Test* accuracy figures represented respectively by the first and second values in the comma separated range.
- Overfitting /Underfitting (%): The difference between the *Training* and *Test* accuracy (%). A positive (negative) number indicates overfitting (underfitting).
- *Training/Test* Delta (%)): The first (second) number in the comma separated range represents the difference between the *Training (Test)* accuracy of the scenario case (e.g. D0_0.2) compared to the Base Case. A green (red) figure indicates an improvement (decline) of accuracy with a scenario case (e.g. D0_0.2).
- Time to Complete Delta: A green (red) figure shows a time improvement (deterioration) with the scenario case compared to the Base Case (e.g. FLT_2).

Hyperparameter	**Base Case**	**LR_0.001**	**LR_0.01**	**LR_0.5**	**DC_0.5**	**DC_0.9**	**DO_0.2**	**D0_0.7**	**FLT_2**
Initial Pool Size	2,2	2,2	2,2	2,2	2,2	2,2	2,2	2,2	2,2
Convolution Strides	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1
Convolution Padding	SAME	SAME	SAME	SAME	SAME	SAME	SAME	SAME	SAME
Pooling Kernel Size	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1
Pooling Strides	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1	1,2,2,1
Pooling Padding	SAME	SAME	SAME	SAME	SAME	SAME	SAME	SAME	SAME
Kernel Height/Wight	5,5	5,5	5,5	5,5	5,5	5,5	5,5	5,5	2,2
Learning Rate	0.0001	0.001	0.01	0.5	0.0001	0.0001	0.0001	0.0001	0.0001
Dropout rate	0.5	0.5	0.5	0.5	0.5	0.5	0.2	0.7	0.5
Weight Decay Coefficient	0.99	0.99	0.99	0.99	0.5	0.9	0.99	0.99	0.99
Momentum	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
Results									
Time to Complete	4 days, 19:26:01	10 days, 22:40:08	10 days, 13:17:21	11 days, 5:57:29	11 days, 20:26:52	10 days, 13:58:01	7 days, 3:15:50	9 days, 9:27:00	4 days, 12:11:59
Training/Test Accuracy (%)	[90.80, 88.63]	[18.93, 19.58]	[18.92, 19.56]	[18.92, 19.58]	[59.78, 59.76]	[82.59, 80.60]	[91.63, 89.30]	[89.35, 86.71]	[62.89, 61.67]
Overfitting /Underfitting (%)	2.17	-0.65	-0.64	-0.66	0.02	1.99	2.33	2.64	1.22
Training/Test Delta (%))		**[-71.87 -69.05]**	**[-71.88, -69.07]**	**[-71.88, -69.03]**	**[-31.02, -28.87]**	**[-8.21, -8.03]**	**[0.83, 0.67]**	**[-1.45, -1.92]**	**[-27.91, -26.96]**
Time to Complete Delta		**>+ 6 days**	**>+ 5 days**	**>+ 7 days**	**>+ 7 days**	**>+ 5 days**	**>+3 days**	**>+ 5days**	**<- 7h**

Table6.1 – Accuracy/Time impact under different scenario based on 3 Conv. Layers - Training size = 73,257 / Test size = 26,032 / Epoch = 20, all other parameters being equal.

Hyperparameter	**Base Case**	**CNN_3**	**CNN_5**
# of CNN layers	4	3	5
Comments		The 3 CNN code implementation is available in Appendix E	The 5 CNN layers code implementation is available in Appendix F
Results			
Time to Complete	4 days, 19:26:01	10 days, 22:40:08	9 days, 18:27:28
Training/Test Accuracy (%)	[90.80, 88.63]	[74.93, 72.11]	[90.78, 88.78]
Overfitting /Underfitting (%)	2.17	2.82	2.0
Training/Test Delta (%))		**[-15.87, -16.52]**	**[-0.02, 0.15]**
Time to Complete Delta		**>+ 6 days**	**>+ 5 days**

Table6.2 – Accuracy/Time impact under different scenario based on 2 and 4 Conv. Layers - Training size = 73,257 / Test size = 26,032 / Epoch = 20, all other parameters being equal.

The results analysis

This paragraph focuses on the analysis of the results in order to produce an optimised version of the *Base Case* scenario.

Family	**Scenario Name**	**Comments**
Scenario Cases	LR_0.001	The *Training/Test* accuracy values have deteriorated significantly, a loss of approximately 70%, so these cases are discarded for optimisation.
	LR_0.01	
	LR_0.5	
	DC_0.5	The *Training/Test* accuracy value has deteriorated significantly, a loss of approximately 8%, so this case is discarded for optimisation.
	DC_0.9	The *Training/Test* accuracy value has deteriorated significantly, a loss of approximately 31%, so this case is discarded for optimisation.
	DO_0.2	This is an interesting case where the *Training/Test* accuracy values have raised by respectively 0.83% and 0.67%, with an increase of overfitting of only 0.16% (from 2.17% to 2.33%). The only drawback is the time to completion, as it took an extra 3 days compared to the *Base Case*. Despite the time to completion increase, I would argue to use a drop out of 0.2 instead of the initial 0.5.
	DO_0.7	The Training/Test accuracy value has deteriorated significantly, a loss of approximately 1.5-2%, so this case is discarded for optimisation.
	FLT_2	The Training/Test accuracy value has deteriorated significantly, a loss of more than 25%, so this case is discarded for optimisation.
	CNN_3	The time to completion is not improved, one the contrary, it has been increased by 50%. This is not expected. This could be associated with the load on the VM environment at the time of the run. The training and test accuracy delta show a loss of 16% of accuracy. Therefore, this case is rejected for optimisation.
	CNN_5	The overfitting improved slightly from 2.17 to 2.0 (a 8.5% improvement) and the *Training* accuracy is relatively unchanged. There is an improvement of the *Test* data accuracy by 0.15%. However, the time to completion is approximately twice as long as in the *Base Case.* In this case the trade-off between the accuracy gain and time to completion is not significant enough. Therefore, this case is rejected for optimisation.

Table7 – Result Analysis

The optimised solution

From the results shown in *Table7,* it looks like the best improvement was to reduce the *dropout* rate to 0.2. This results in an extra overfitting of 0.16% (2.33%-2.17%), which is a marginal price to pay for an increase of 0.83% and 0.67% of respectively the *Training* and *Test* accuracy. The slight improvement of 0.15% of *Test* accuracy and 0.17% of overfitting reduction in scenario *CNN_5* was not retained as the improvement was not significant enough for the extra time required to complete the analysis (+5days). *Table8* summarises the parameters and results of the final proposed solution.

Hyperparameter	**Base Case**	**Final State**
# of CNN layers	3	3
Initial Pool Size	2,2	2,2
Convolution Strides	1,1,1,1	1,1,1,1
Convolution Padding	SAME	SAME
Pooling Kernel Size	1,2,2,1	1,2,2,1
Pooling Strides	1, 2, 2, 1	1, 2, 2, 1
Pooling Padding	SAME	SAME
Kernel Height/Wight	5,5	5,5
Learning Rate	0.0001	0.0001
Dropout Rate	0.5	0.2
Weight Decay Coefficient	0.99	0.99
Momentum	0.9	0.9
Results		
Time to Complete	4 days, 19:26:01	7 days, 3:15:50
Training/Test Accuracy (%)	[90.80, 88.63]	[91.63, 89.30]
Overfitting /Underfitting (%)	2.17	2.33
Training/Test Delta (%))		**[0.83, 0.67]**
Time to Complete Delta		**>+3 days**

Table8 – Final accuracy improvement based on 3 Conv. Layers - Training size = 73,257 / Test size = 26,032 / Epoch = 20, all other parameters being equal.

Results

Model Evaluation and Validation

The final model is reasonable and aligns with the initial expectations. Only the dropout rate needed to be amended to 20%, from its original setup at 50%. This by definition increases *overfitting*. However, as proven in this large and diversified dataset, the increase of *overfitting* is four times smaller (+0.16%) than the increase in *Test* accuracy (+0.67%). As demonstrated in the *Data Exploration* section, the large variance and amount of outliners relating to *Training* images height/width and left/right position is related to the diversity of translated images of different shape, title, quality into computer readable integers.

Table9 shows equivalent statistical measure for the *Test* images. The left and top positions of a number in an image show respectively a mean of 80.63px (with a standard deviation of 63.10px) and 22.62px (with a standard deviation of 26.98px). This implies a large bias in the *Test* data. This is reinforced by the min (left =-3.00px, top=0.00px) and max (left =832.00px, top=283.00px) values compared to the 25% (left =41.00px, top=8.00px) and 75% (left =98.00px, top=26.00px) percentiles which are very much apart. Furthermore, the left and top box plots show a large number of outliers. The same analysis and conclusion can be drawn from *Table9* relative to the height and width. They show respectively a mean of 27.96px and 15.52px for standard deviations of 13.46px and 8.24px. Their minimum and maximum values are far away from the 25% and 75% percentiles. *Fig.11* also shows a large number of outliers in both instances.

Table10 shows the difference of statistics obtain from the *Training* and *Test* images. It is clear from these results that the mean, median, minimum and maximum statistics for the Left/Top position and Height/Weight indicates that images sets are very different as they display wide differences in statistics.

From these results, we can conclude that the *Test* data is varied in terms of image shape, title and quality and that the Test and Training images are very different. Although the model could be further improved, a 89.30% Test accuracy result is deemed robust enough for this problem. Small and large changes in the *Training* and *Test* data sets do not affect greatly the results. Therefore, this model can be trusted at this level of expectations. The code that generates these statistics and graphs can be found in section *'print("Start - Data Vizualisation...") '* in the IPython notebook.

	Left Position	**Top Position**	**Height**	**Width**
Count	260.32.00	260.32.00	260.32.00	260.32.00
Mean	80.63	22.62	27.96	15.52
Median	60.0	13.00	24.00	13.00
Std	63.10	26.98	13.46	8.24
Min	-3.00	0.00	10.00	3.00
25%	41.00	8.00	19.00	10.00
50%	60.00	13.00	24.00	13.00
75%	98.00	26.00	32.00	18.00
Max	832.00	283.00	403.00	133.00

Table9 – The SVHN Test image list statistics (all values unit - bar the count - are pixels).

	Left Position	**Top Position**	**Height**	**Width**
Mean Delta	22.23	10.74	-5.90	-1.13
Median Delta	14.00	6.00	-5.00	-1.00
Std Delta	21.20	12.52	-5.14	-2.44
Min Delta	-2.00	0.00	1.00	2.00
25% Delta	1.00	4.00	-2.00	0.00
50% Delta	14.00	6.00	-5.00	-1.00
75% Delta	25.00	11.00	-9.00	-3.00
Max Delta	214.00	64.00	0.00	-74.00

Table10 – The Difference (deltas) between the SVHN Training image and Test images statistics.

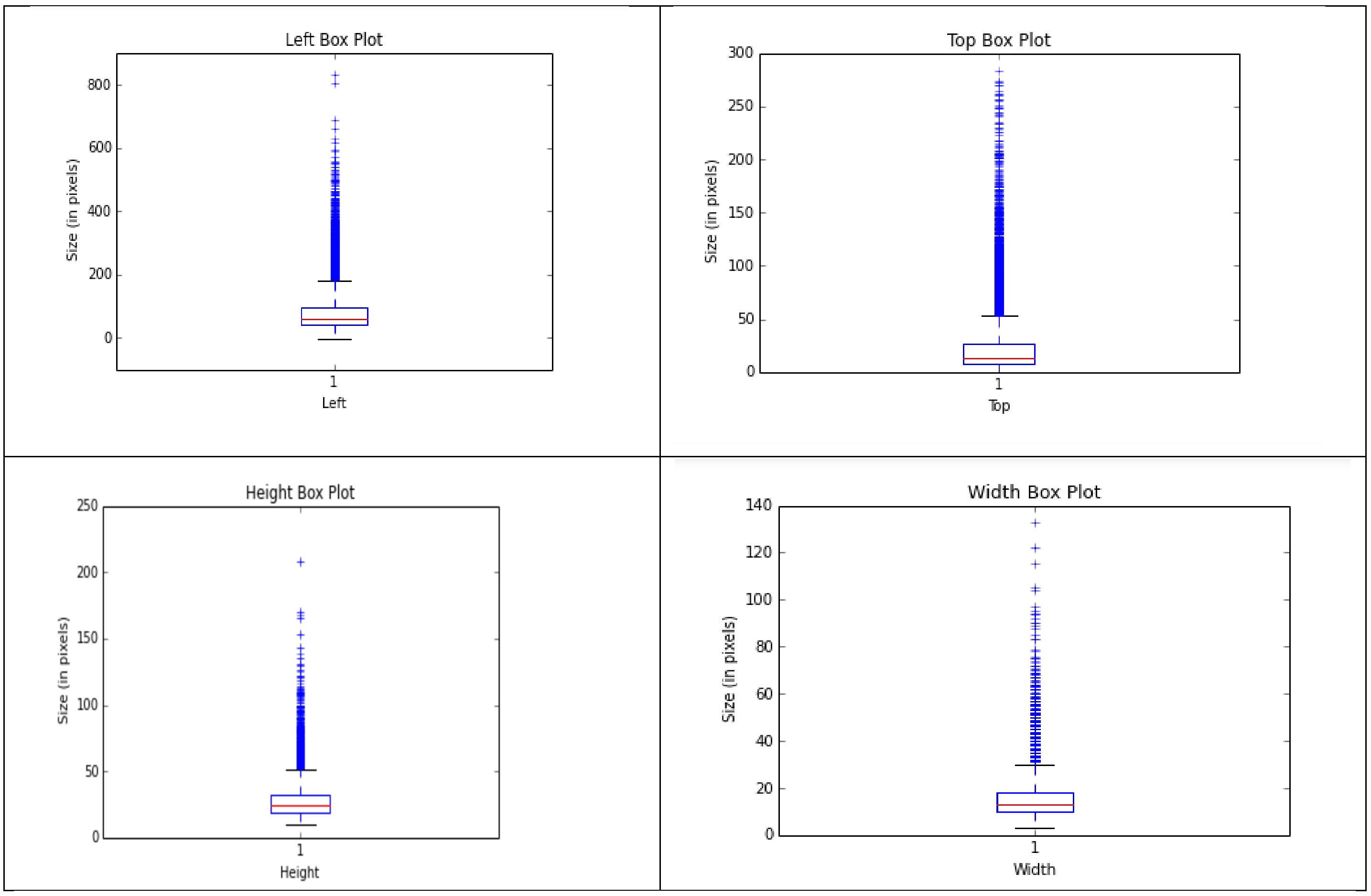

Fig.11 The SVHM *Test images box plots.*

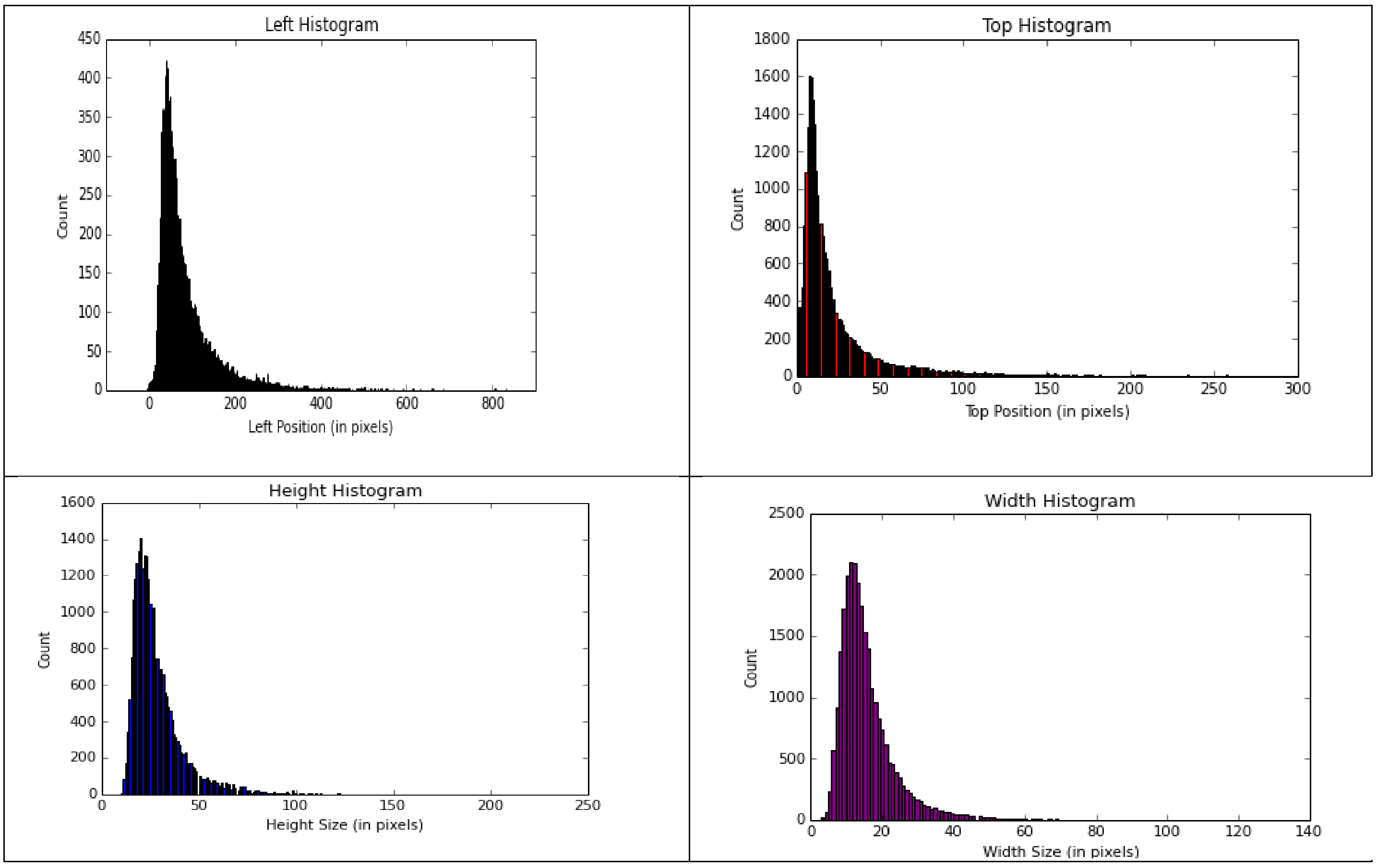

Fig.12: The SVHM Test images histograms.

Justification

The average models performance shown in *Table11* is 89.28% (with a standard deviation of 0.1%). Test accuracy result obtained by the 4-layer Convolutional Neural Network (CNN) of 89.30%, places it as an average performer. It is far better than a *Binary Feature (WDCH)* model and a 5% less performant than the *ConvNet / MS / L4.*
This result is disappointing as it does not outperform the *K-means and/*or the *ConvNet / MS / Average* models, let alone the performance of human brain accuracy.

Algorithm	**Reference**	**Test Accuracy**
Binary Feature (WDCH)	Kimura F.,Wakabayashi T., Tsuruoka S., and Miyake Y. (1997)	63.30%
HOG	Dalal N. and Triggs B. (2005)	85.00%
4-layer Convolutional Neural Network (CNN)		89.30%
Stacked Sparse Auto-Encoders	Coates A., Lee H., and Ng A. Y (2011)	89.70%
K-means	Netzer Y et al. (2011)	90.60%
ConvNet / MS / Average	Sermanet P., Chintala S. and LeCun Y (2012)	90.75%
ConvNet / MS / L2 (Smaller training)	Sermanet P., Chintala S. and LeCun Y (2012)	91.55%
ConvNet / SS / L2	Sermanet P., Chintala S. and LeCun Y (2012)	94.28%
ConvNet / MS / L12	Sermanet P., Chintala S. and LeCun Y (2012)	94.76%
ConvNet / MS / L4	Sermanet P., Chintala S. and LeCun Y (2012)	94.85%
Human Brain Accuracy	Netzer Y et al. (2011)	98.00%

Table11 – Result comparison

However this models comes with a number of strengths.

- Although this architecture alone does not provide a better accuracy to deep *convolution* network design, it is important to note that it does not suffer from the intrinsic scalability limitation of a deep *convolution* network design, and hold fewer *hyperparameters,* making it easier to train.
- It still shows a *Test* accuracy superior to 89%, which is belongs to the higher scoring algorithms listed in *Table9*.

Conclusion

Free-Form Visualization

Fig13 a) shows that for a relatively large number of *Training and Test* data (respectively 73,257 and 26,032), the optimum *Test* accuracy level is reached in 8 epochs, at a level of 89.30%. It also shows that prior to 5 epochs there was close to no overfitting or underfitting. After that, overfitting remains small and relatively constant around 2%. So it shows the model is robust, and matches *Test* data at a 89.30% confidence level. *Fig13* b) shows that the *loss* curve for a 20% dropout is much smoother than the when the *dropout* is set to 70% or 50% (base case). The learning improvement is therefore much more linear.

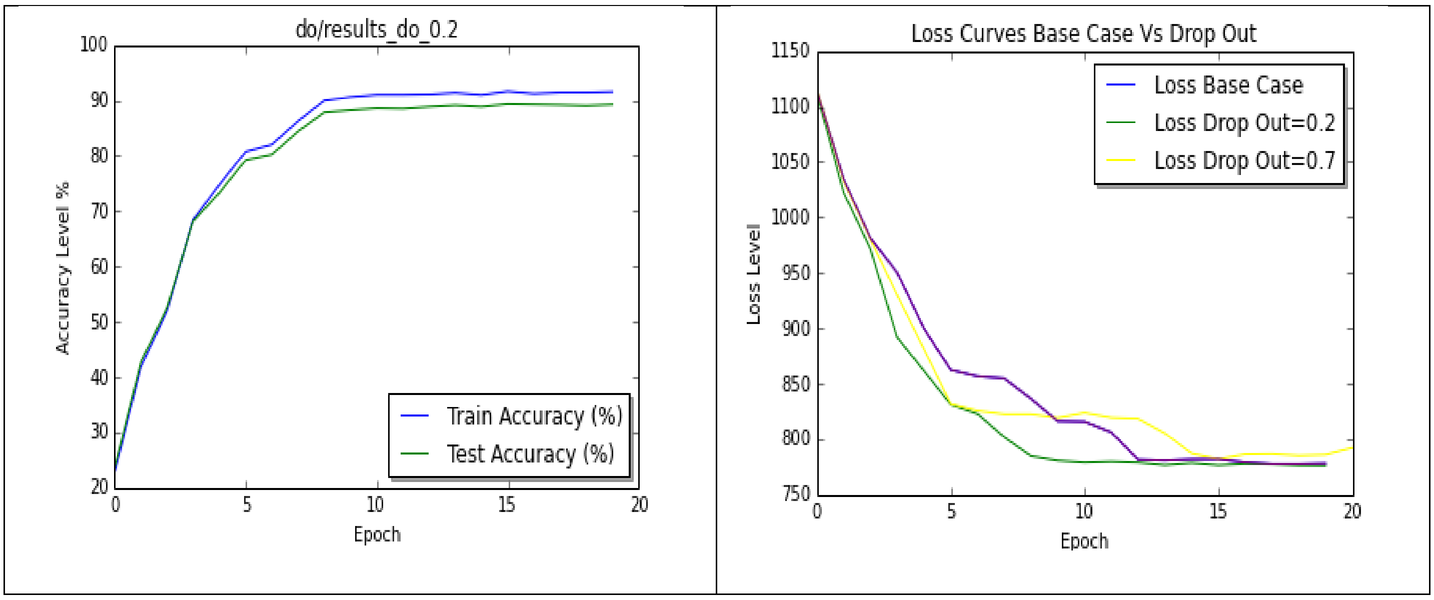

Fig.13: a)The Train/Validation Accuracy % and b) Loss functions

Reflection

The task was to build a live camera app that interprets digits in real-world images. These images presented some identification difficulties such as tilt, shadow, and other image quality degradations. The analysis was carried out on a 'read life' complex image dataset named *SVNH*. The data source contained an image bank converted into a matrix of length 73,257 and shape X(:,:,:,i). The first three columns listed the 8bits value of each of the RGB primary colour model. The last column referred to the label. The dataset contained 33,402 *Training* images and 13,068 *Test* images converted into 73,257 *Training* and 26,032 *Test* data (c.f. Appendix B).

The chosen prediction algorithm was architected with a 4-layer CCN. The first step in the process was to make the *Training* and *Test* data/label compatible with the CNN internal algorithms. The *Training* data was normalised and flattened to a 4-D tensor, while the labels were transformed into *one-hot* labels. From this point on, both the data and labels flew through a pipeline of *Convoluted Layers* to reduce the original image size from 32x32 to 4x4, whilst keeping the most significant piece of information, to produce the best possible label predictions. The prediction line was generated in the last *Convolution Layer*. Once the model was in place, it then had to be trained over 20 iterations (a.k.a. epochs).

The first challenge was to find the right balance between optimising the *Training* data volume and the number of iterations. The aim was to keep the largest *Training* set possible to reduce *overfitting*, coupled with the largest number of iterations to optimise the algorithm learning. The third factor taken into account was the time to complete Training and evaluation. With longer learning phase, there was a greater risk of network outage or the virtual machine unavailability. Outages had the effect of delaying information gathering by several days.

Experiments showed that the optimised *Training* data set, *Test* data set and batch size was respectively: 73,257 and 26,032 and 500, for a 20 epochs running period.

The second challenge concerned the optimisation of the model prediction power, given the chosen *Training/Test* dataset. For that, a number of scenarios were identified for *hyperparameters* tuning (c.f. *Table6.1/6.2*). As a result, the optimised *hyperparameters* were selected as per *Table8*.

The interesting parts of the project came with:

- the study of existing approaches to resolve a closely related problem,
- the selection of an architecture that was proven to be performant on a dataset containing simpler images,
- the tweaking of the existing solution to get as close as possible to the set goal, given the technical environment constrains,
- the creation of a simple user interface that shows images containing street number and the predicted /true labels.

There were a number of difficulties relating to this project. The first one was to understand how to implement and train CNNs architecture in Python. A large part of the project focused on reading existing literature to get an idea on how to implement the initial solution. Second, the Tensorflow deployment was an issue on Windows 10. After failing to install it on a local Ubuntu VM environment and making it work with Python, I decided to rent a pre-installed version of Python/Tensorflow VM on the Cloud. This had a cost implication of approximately GBP20 per month. But without this basic installation, I would have had to abandon the project. This solution came with a limitation in terms of available CPU (set to 1), and a limited RAM (capped to 2.5GB). These limitations had two implications: i) the time required completing the training on large datasets and ii) the reaching of the RAM limits provoking *Out Of Memory Exceptions*. No work around were available to reduce the training completion time, as the number of CPU/memory could not be increased above the existing configuration. GPU computation was not available either on the VM. However, I managed to work around the RAM limitations by batching the *Training* and *Test* data, at the cost of increasing maintenance complexity. The third difficulty related to the inability to generate Tensorflow plots. The rented environment did not support some of the Python/Tensorflow third party libraries relating to tables, graphs and plots generation. Therefore, I had to revert to using the Panda library and basic plotting facility based on the cropped *Training/Test* data, stored into CSV files.

As a whole, the model did fit my expectations. The model achieved Test accuracy greater than 90% with a small level of overfitting (approximately 2%). Nonetheless, it is clear that the model is not fit for an industry due to the large potential error (10%) which would have a cost and reputation implication. However, the model has shown to be resilient to perturbation in *Training* data and scalable, as well as having relatively small amount of *hyperparameters* to tune. Consequently, I think that if some of the inner algorithms used by this model were optimised, the next generation of *n-CNN*s could fully answer this type of problems.

Improvements

First, the Python code shows a number of sections, as an attempt to clarify the different steps in the process of gathering, analysing, training/predicting and displaying image list versus their true and predicted labels. Nonetheless, the code has not been written in a way that enables reuse and flexibility. It is a monolithic piece of code that is sufficient as a prototype. However, it does not abide to good design patters and practices. The lack of interfaces and plugins does not allow for easy unit testing and extensibility. The first action would be to organise the code into classes and packages. Then, a number of interfaces should be created to represent the objects behaviours. To facilitate code reuse and unit testing, *inversion of control* (a.k.a. *IoC*) should be implemented, where interfaces are passed as function members instead of using object references.

The proposed solution, in this paper, could be improved by swapping the current *max pooling* algorithm by the Lp4 polling introduced by Sermanet P., Chintala S. and LeCun Y (2012). "Lp pooling is a biologically inspired pooling layer modelled on complex cells" (c.f. *Equation1). G* represents a Gaussian kernel, *I* denotes the input feature map and O represents the output feature map. "It can be imagined as giving an increased weight to stronger features and suppressing weaker features (…). Two special cases of Lp pooling are notable. P = 1 corresponds to a simple Gaussian averaging, whereas P = ∞ corresponds to max-pooling." Their analysis showed that with P=4, L4 pooling increased the authors' CNN accuracy by 4.25% from 90.6% to 94.85%. It would be interesting to see by how much an L4 pooling would increase the CNN solution proposed in this paper.

$$O = (\sum_{i}\sum_{i} I(i,j)^{P} * \; G(i,j))^{P}$$

Equation1 – Lp polling

Second, as shown in the Table 6.1/6.2, it takes on average 4.5 days to produce an end-to-end run. This is partially due to the amount of data produced by the 4-D RGB-coloured images tensor input. Time to complete a run could be reduced by compressing this data into a 2-D tensor. The colour could be averaged for each image following this simple calculation: (R+G+B) / 3. This would also reduce the memory requirement significantly.

Appendix A – Tensorflow and Python Installation

Due to a number of difficulties associated with the installation of the Tensorflow software on Windows Virtual Machine, I decided to rent a pre-built cloud environment on the c9.io platform. It had a cost of approximatively £20 per month for 1CPU, 2.5GB of RAM and a 10GB disk. This was not ideal, as this solution required ideally a minimum of 8GB RAM. However, only a maximum of 2.5GB RAM was available. The steps to install Tensorflow and run the CNN python code is as follows:

- Login to www.c9.io and purchase a plan that provide the above listed capacity requirements
- Follow the steps explained in the video hosted on https://www.youtube.com/watch?vh=ReaxoSIM5XQ&feature=youtu.be
- Load the python code to run in the older of your choice and run it.

Appendix B – Training and Test Images Count

The *Training* and *Test* images counts were generated from the train.csv and test.csv files (c.f. Appendix C). I created a pivot Table with the *fileName* column on the *Y-axis,* and the count of *filename* records on the *X-axis*. By counting the number of unique *filename* records, I could then deduce the number of images used for the experiment. Please see the 'extra_files/train_analysis.xlsx' and 'extra_files/test_analysis.xlsx' files for the implementation details.

Appendix C – *.mat* to *.csv* Conversion

I had to generate human readable files from the *SVHN Training* and *Test .mat* ' files. The output can be found in 'extra_files/train.csv' and 'extra_files/test. csv' files. To convert the files, I borrowed the *.mat* parsing Python code from the web site https://github.com/prijip/Py-Gsvhn-DigitStruct-Reader/blob/master/digitStruct.py . To build and run the code, the *h5py* library needed to be installed and referenced in the Python program (via *pip install h5py*). Unfortunately, it was not possible to install this dependency on the rented Virtual Machine environment (c.f. Appendix C). Therefore, I ran the following code on a separate box. The code can found in 'extra_files/digitStruct.py'. The user simply needs to follow these simple steps to produce the .csv files:

- Download and unzip the 'train.tar.gz' and 'test.tar.gz' files present in http://ufldl.stanford.edu/housenumbers/
- Have a Python IDE ready and install h5py via *pip install h5py*
- Load the 'digitStruct.py' into the environment
- Change the training and test source file paths in *main* method
- Run
- Copy paste the training and test outputs into two different csv files, named 'train.csv' and 'test.csv'

Appendix D – Scenario Results Implementation

```
print("Start - Learning Process Analysis ...")

main_directory     = 'svhn_data'
results_directory  = 'results'

results_directory = main_directory + '/' + results_directory


def show_train_validation_accuracy( result_file_name, start_time_idx, end_time_idx,):
   print ("")
   print ("*************************************************************************************")
   print ("***********************" + result_file_name + "*****************************************")
   print ("*************************************************************************************")
   print ("")
   df = pd.read_csv(results_directory + "/" + result_file_name)
   start_time= datetime.strptime(df.loc[start_time_idx, 'Start_Time'],"%Y-%m-%d %H:%M:%S")
   end_time= datetime.strptime(df.loc[end_time_idx, 'Current_Time'], "%Y-%m-%d %H:%M:%S")
   time_to_completion = end_time - start_time
   train_df = df.loc[df['Type'] == 'Train']
   train_accuracy_avg = train_df.groupby('Iteration_Id',as_index=False)['Accuracy_%'].mean()
   test_df = df.loc[df['Type'] == 'Test']
   test_accuracy_avg = test_df.groupby('Iteration_Id',as_index=False)['Accuracy_%'].mean()
   print (">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Statistics (Training Accuracy) >>>>>>>>>>>>>>>>>>>>>>>>>>>")
   print ("Training Aggregated Data: " + str(train_accuracy_avg.ix[:,1:3].describe))
   print ("Min Training Accuracy_%: " +  str(train_accuracy_avg['Accuracy_%'].min()))
   print ("Median Training Accuracy_%: " +  str(train_accuracy_avg['Accuracy_%'].median()))
   print ("Max Training Accuracy_%: " +  str(train_accuracy_avg['Accuracy_%'].max()))
   print ("")
   print (">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Statistics (Test Accuracy) >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
   print ("Test Aggregated Data: " + str(test_accuracy_avg.ix[:,1:3].describe))
   print ("Min Test Accuracy_%: " +  str(test_accuracy_avg['Accuracy_%'].min()))
   print ("Median Test Accuracy_%: " +  str(test_accuracy_avg['Accuracy_%'].median()))
   print ("Max Test Accuracy_%: " +  str(test_accuracy_avg['Accuracy_%'].max()))
   print ("")
   print (">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Time to Completion>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")

   print ("Time to completion: (start:" + str(start_time) + " - end:" + str(end_time) + ") -> " +  str(time_to_completion))

   print (">>>>>>>>>>>>>>>>>>>>>>>>>>> Graph Training Vs Test Accuracy >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
   plt.title(result_file_name)
   plt.ylabel('Accuracy Level %')
   plt.xlabel('Epoch')
   plt.gca().set_color_cycle(['blue','green'])
   plt.plot(train_accuracy_avg['Iteration_Id'], train_accuracy_avg['Accuracy_%'], label='Train Accuracy (%)')
   plt.plot(test_accuracy_avg['Iteration_Id'], test_accuracy_avg['Accuracy_%'], label='Test Accuracy (%)')
   plt.legend(loc="lower right", shadow=True)
   plt.show()
   print ("*************************************************************************************")
```

```
def show_loss(title, label_base_case, df_base_case,
                label_sim_1, df_sim1,
                label_sim_2, df_sim2,
                label_sim_3, df_sim3):
    loss_avg_base_case = df_base_case.loc[df_base_case['Type'] == 'Loss'].groupby('Iteration_Id',as_index=False)['Error_%'].mean()
    loss_avg_sim_1 = df_sim1.loc[df_sim1['Type'] == 'Loss'].groupby('Iteration_Id',as_index=False)['Error_%'].mean()
    loss_avg_sim_2 = df_sim2.loc[df_sim2['Type'] == 'Loss'].groupby('Iteration_Id',as_index=False)['Error_%'].mean()
    loss_avg_sim_3 = df_sim3.loc[df_sim3['Type'] == 'Loss'].groupby('Iteration_Id',as_index=False)['Error_%'].mean()
    plt.title(title)
    plt.ylabel('Loss Level')
    plt.xlabel('Epoch')
    plt.gca().set_color_cycle(['blue','green','yellow', 'purple'])
    #add more graph below
    plt.plot(loss_avg_base_case['Iteration_Id'], loss_avg_base_case['Error_%'], label=label_base_case)
    plt.plot(loss_avg_sim_1['Iteration_Id'], loss_avg_sim_1['Error_%'], label=label_sim_1)
    plt.plot(loss_avg_sim_2['Iteration_Id'], loss_avg_sim_2['Error_%'], label=label_sim_2)
    plt.plot(loss_avg_sim_3['Iteration_Id'], loss_avg_sim_3['Error_%'], label=label_sim_3)
    plt.legend(loc="upper right", shadow=True)
    plt.show()

try:
    #display train vs test accuracy graphs
    #base
    show_train_validation_accuracy("base_case/results_base_case", 0,51877)
    #lr simulation
    show_train_validation_accuracy("lr/results_lr_0.001", 0,51877)
    show_train_validation_accuracy("lr/results_lr_0.01", 0,51877)
    show_train_validation_accuracy("lr/results_lr_0.5", 0,51877)
    #dc simulation
    show_train_validation_accuracy("dc/results_dc_0.5", 0,51877)
    show_train_validation_accuracy("dc/results_dc_0.9", 0,51877)
    #do simulation
    show_train_validation_accuracy("do/results_do_0.2", 0,51877)
    show_train_validation_accuracy("do/results_do_0.7", 0,51877)
    #fltr simulation
    show_train_validation_accuracy("fltr/results_fltr_2", 0,51877)
    #cnn simulation
    show_train_validation_accuracy("cnn/results_ccn_2_layers", 0,51891)
    show_train_validation_accuracy("cnn/results_ccn_4_layers", 0,51894)
    #show_train_validation_accuracy("results_M5000_3CL_itr100_train_36628_test_13016_DO_09")
    #add more

    #display loss graphs
    df_base_case = pd.read_csv(results_directory + "/" + "base_case/results_base_case")
    df_lr_0_001 = pd.read_csv(results_directory + "/" + "lr/results_lr_0.001")
    df_lr_0_01 = pd.read_csv(results_directory + "/" + "lr/results_lr_0.01")
    df_lr_0_5 = pd.read_csv(results_directory + "/" + "lr/results_lr_0.5")
    show_loss(  "Loss Curves Base Case Vs Learning Rates",
            'Loss Base Case',
            df_base_case,
            'Loss Learning Rate=0.001',
            df_lr_0_001,
            'Loss Learning Rate=0.01',
            df_lr_0_01,
            'Loss Learning Rate=0.5',
            df_lr_0_5)
    df_do_0_2 = pd.read_csv(results_directory + "/" + "do/results_do_0.2")
    df_do_0_7 = pd.read_csv(results_directory + "/" + "do/results_do_0.7")
    show_loss(  "Loss Curves Base Case Vs Drop Out",
              'Loss Base Case',
              df_base_case,
              'Loss Drop Out=0.2',
              df_do_0_2,
              'Loss Drop Out=0.7',
              df_do_0_7,
              '', #dummy
              df_base_case)  #dummy

    df_dc_0_5 = pd.read_csv(results_directory + "/" + "do/results_dc_0.5")
    df_dc_0_9 = pd.read_csv(results_directory + "/" + "do/results_dc_0.9")
    show_loss(  "Loss Curves Base Case Vs Drop Out",
              'Loss Base Case',
              df_base_case,
              'Loss Drop Out=0.5',
              df_dc_0_5,
              'Loss Drop Out=0.9',
              df_dc_0_9,
              '', #dummy
              df_base_case)  #dummy

    df_cnn_2 = pd.read_csv(results_directory + "/" + "cnn/results_ccn_2_layers")
    df_cnn_4 = pd.read_csv(results_directory + "/" + "cnn/results_ccn_4_layers")
    show_loss(  "Loss Curves Base Case Vs Different CNN Implmentations",
              'Loss Base Case',
              df_base_case,
              'Loss CNN layers=2',
              df_cnn_2,
              'Loss CNN layers=4',
              df_cnn_4,
              '', #dummy
              df_base_case)#dummy

except Exception as e:
    print("An unexpected error occured", e)

print("End - Learning Process Analysis ...")
```

Appendix E – 2-CCN Layer Code

The only change is the removal of the last *convolution* and the reference of the second *convolution* into the final *convolutional* layer.
The full code version is available in the file ‘.../code/MultiDigitNumberRecognitionMultiLayer_3CNN .ipynb’.

```
print("Start - Initialising and Defining the Model...")
 (…)

print ('Define convolution layers...')
conv1 =  conv2d (x, w1, b1, conv_strides, conv_padding)
conv1_relu =  relu(conv1)
max_pool_1 = max_pool (conv1_relu, pool_kernel_size, pool_strides, pool_padding)

conv2 =  conv2d (max_pool_1, w2, b2, conv_strides, conv_padding)
conv2_relu = relu(conv2)
max_pool_2 = max_pool (conv2_relu, pool_kernel_size, pool_strides, pool_padding)
max_pool_2_shape = max_pool_2.get_shape().as_list()
max_pool_2_reshape = tf.reshape(max_pool_2, [max_pool_2_shape[0], np.prod(max_pool_2_shape[1:])])

🡐 Start  removal of the third Convolutional Layer
#conv22 =  conv2d (max_pool_2, w22, b22, conv_strides, conv_padding) #conv2
#conv22_relu = relu(conv22) #conv2_relu
#max_pool_22 = max_pool (conv22_relu, pool_kernel_size, pool_strides, pool_padding) #max_pool_2
#max_pool_22_shape = max_pool_22.get_shape().as_list()
#max_pool_22_reshape = tf.reshape(max_pool_22, [max_pool_22_shape[0], np.prod(max_pool_22_shape[1:])])
🡐  End removal of the third Convolutional Layer

conv_final = tf.matmul(max_pool_2_reshape, w3) + b3  🡐  The final convolution now references the second Convolutional Layer

keep_prob = tf.placeholder(tf.float32)
drop_out = tf.nn.dropout(conv_final, keep_prob)

y_prediction = tf.nn.softmax(tf.matmul(drop_out, w4) + b4)

(…)
print("End - Initialising and Defining the Model...")
```

Appendix F – 4-CCN Layer Code

The full code version is available in the file '.../code/ MultiDigitNumberRecognitionMultiLayer_5CNN .ipynb'.

First the fourth *convolution hyperparameters* are added to the model:

```
print("Start - Setting Model Parameters ...")
(...)

🡐 Start hyperparameters definition for the fourth layer
conv23_filter_width = 4
print_and_store(results_directory, params_settings_file_name, 'conv23_filter_width',conv23_filter_width)
conv23_filter_height = 4
print_and_store(results_directory, params_settings_file_name, 'conv23_filter_height',conv23_filter_height)
conv23_num_color_channels = conv22_num_feature_maps
print_and_store(results_directory, params_settings_file_name, 'conv23_num_color_channels',conv23_num_color_channels)
conv23_num_feature_maps = 256
print_and_store(results_directory, params_settings_file_name, 'conv23_num_feature_maps',conv23_num_feature_maps)
🡐 End  hyperparameters definition for the fourth layer

(...)
print("End - Setting Model Parameters ...")
```

Then, there is an addition of a fourth *convolution* as well the referencing of the fourth *convolution* into the final *convolutional* layer.

```
print("Start - Initialising and Defining the Model...")
(...)

🡐 Start  weight and bias instantiations for the fourth layer
w23_shape, w23_init,b23_init = conv_shape_weight_biais_init (conv23_filter_width, conv23_filter_height, conv23_num_color_channels,
conv23_num_feature_maps, poolsz)
(...)
w23, b23 = init_weight_and_biais_variable (w23_init, b23_init)
(...)
🡐 End  weight and bias instantiations for the fourth layer

(...)

🡐 Start addition of the fourth Convolutional Layer
conv23 =  conv2d (max_pool_22, w23, b23, conv_strides, conv_padding)
conv23_relu = relu(conv23)
max_pool_23 = max_pool (conv23_relu, pool_kernel_size, pool_strides, pool_padding)
max_pool_23_shape = max_pool_23.get_shape().as_list()
max_pool_23_reshape = tf.reshape(max_pool_23, [max_pool_23_shape[0], np.prod(max_pool_23_shape[1:])])
🡐 End addition of the fourth Convolutional Layer

conv_final = tf.matmul(max_pool_23_reshape, w3) + b3  🡐 The final convolution now references the fourth Convolutional Layer

keep_prob = tf.placeholder(tf.float32)
drop_out = tf.nn.dropout(conv_final, keep_prob)

y_prediction = tf.nn.softmax(tf.matmul(drop_out, w4) + b4)

(...)
print("End - Initialising and Defining the Model...")
```

Appendix G – The User Interface Code

```
print("Start - Number Prediction Visual Representation...")

"""Helpers"""
# maxCount = the max number of rows to show image vs true label vs predicted label for
# true_labels =  the list of true labels (labels attached against an picture from the source dataset)
# predicted_labels =  the list of predicted labels (i.e. labels guessed by the model)
# predicted_labels_offset =  An offset on the predicted labels, as they have been reduced by one during the flattening exericse to enable
the one-hot label generation
def visual_representation(maxCount,true_labels, predicted_labels,predicted_labels_offset = 1):
    main_directory           = 'svhn_data'
    train_raw_csv_directory    = 'raw_train_csv'
    train_csv               = 'train.csv'
    train_raw_img_directory    = 'raw_train_img'
    train_raw_csv_directory = main_directory + '/' + train_raw_csv_directory
    train_raw_img_directory = main_directory + '/' + train_raw_img_directory

    try:
        #open the csv file
        csv_reader = open(train_raw_csv_directory + '/'+ train_csv, 'r')
        lines = csv_reader.readlines()
        i = 0
        temp_image_name = "temp";
        image_name = "";
        match = "no match"
        count_match = 0.0
        total = 0.0
        #display image name /labels and result
        for index in range(len(predicted_labels)):
            line_splits = lines[index+1].split(",")
            image_name = line_splits[0] #first is the image name
            true_label = line_splits[1] #second is the true label
            predicted_label = str(int(predicted_labels[index]) + predicted_labels_offset)#thirst is the predicted label

            #display image and description headers
            if (temp_image_name != image_name):
                if(index > 0):
                    success_rate = "%0.2f" % (100.0 * (count_match/total))
                    print ("Success Rate: " + str(success_rate) + "%")
                    total = 0.0
                    count_match = 0.0
                from IPython.display import display, Image #required else it breaks...
                img= Image(filename=train_raw_img_directory + "/" + image_name)
                print("")

               print("**************************************************************************************************")
                display(img)
                print("Idx \t Image Name \t True Label \t Predicted Label \t Result")
            #count digit matches and total number of digits per image
            if (true_label == predicted_label):
                match = "match"
                count_match = count_match + 1.0
            total = total + 1.0

            #display description
            print(str(index) + " \t " + image_name + " \t\t " + true_label + " \t\t " + str(predicted_label) + " \t\t\t " + match )

            #reset params
            match = "no match"
            i = i + 1
            temp_image_name = image_name
            if (i == maxCount):
                break;
    except Exception as e:
        print("An unexpected error occured", e)

print("End Number Prediction Visual Representation...")

"""View Results"""
visual_representation(500, y_train, prediction_train,1 )
```

References

Academia (2014), Some Intuition about Activation Functions in Feed-Forward NeuralNetworks, http://www.academia.edu/7826776/Mathematical_Intuition_for_Performance_of_Rectified_Linear_Unit_in_Deep_Neural_Networks, [Accessed 05 June 2016]

Alex Krizhevsky et al. (2012), ImageNet Classification with Deep Convolutional Neural Networks, *University of Toronto.*

Coates A., Lee H., and Ng A. Y (2011). An Analysis of Single-Layer Networks in Unsupervised Feature Learning, *AI and Statistic*.

Dalal N. and Triggs B. (2005). Histograms of oriented gradients for human detection, *CVPR*, pp 886–893.

Kimura F.,Wakabayashi T., Tsuruoka S., and Miyake Y. (1997), Improvement of handwritten japanese character-recognition using weighted direction code histogram, *Pattern Recognition 30(8)*,pp1329–1337.

Goodfellow, Ian J., et al. (2013), 'Multi-digit number recognition from street view imagery using deep convolutional neural networks.', *arXiv preprint arXiv:1312.6082*.

Goodfellow, et al. (2016), 'Deep Learning', *Book in preparation for MIT Press,* pp331-373 and 424-455

LeCun, Y. (1989), 'Generalization and network design strategies. Technical Report', *University of Toronto*, pp331-351

Netzer Y., et al. (2011),' Reading Digits in Natural Images with Unsupervised Feature Learning', *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*

Nitish Srivastava et al. (2014), Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research Research 15 (2014) 1929-1958*

Nowlan S. J and Hinton G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, *4(4)*

Sermanet P., Chintala S. and LeCun Y (2012), 'Convolutional Neural Networks Applied to House Numbers Digit Classification', *The Courant Institute of Mathematical Sciences - New York University*

Stanford (2016), CS231n Convolutional Neural Networks for Visual Recognition, *http://cs231n.github.io/neural-networks-1/,* [Accessed 15 May 2016]

Tensorflow (2016), MNIST For ML Beginners, *https://www.tensorflow.org/versions/r0.8/tutorials/mnist/beginners/index.html#mnist-for-ml-beginners*, [Accessed 01 June 2016]

UFDDL Tutorial, Convolutional Neural Network, *http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/*, [Accessed 01 May 2016]

Wikipedia, Convolution, *https://en.wikipedia.org/wiki/Convolution*, [Accessed 03 June 2016]

Wikipedia, *Convolutional Neural Network*, *https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer*, [Accessed 03 June 2016]

Wikipedia, Activation Function, *https://en.wikipedia.org/wiki/Activation_function*, [Accessed 03 June 2016]