

Capstone Project

Machine Learning Engineer Nanodegree

Benjamin Chang

November 8th, 2016

Multi-Digit Recognition with CNN

1. Definition

Project Overview

The problem of detecting and reading text in images is a challenging computational task. Although the problem of recognizing characters in text documents or handwritten digits has been widely studied and reliably solved for, recognizing characters reliably in more complex, unconstrained natural photographs is far more difficult. (Netzer, Y., et. al., 2011) In this report, the focus is on recognizing a sequence of digits in photographs of address numbers at street level. An example of a highly known collection of street level photographs is Google's Street View imagery, which comprises of hundreds of millions of geo-located 360 degree panoramic images. In modern-day map making, the ability to automatically transcribe an address number from a geo-located patch of pixels and associate the transcribed number with a known street address helps to pinpoint, with a high degree of accuracy, the location of the building it represents. (Goodfellow, I., et. al., 2014)

Problem Statement

In this project, a deep learning model is trained on the [Street View House Numbers \(SVHN\)](#) dataset (Netzer, Y., et. al., 2011) that is able to decode sequences of digits from natural images. The challenge is to design and build a robust learning model which is able to recognize digits of various form factors including different typefaces, image distortions (scale, translation, skewed planes, rotation), image quality (blurriness, low-contrast), as well as distracting edges and backgrounds. One application for this is a live camera app. The images captured by a smartphone camera is processed by the model, and it outputs the numbers it sees in real time via an Android mobile application interface.

The strategy of the project implementation are as follows:

1. SVHN dataset is obtained and preprocessing of data is done where necessary
2. A learning model architecture based on a Convolutional Neural Network is designed and first tested on single-digit format of SVHN.
3. Once a good architecture is determined, the model is further developed for multi-digit recognition to be trained on the multi-digit SVHN dataset format.
4. The model is tested on its predictive power, and hyperparameter exploration is done to optimize the model. The model's performance is compared with similar academic publications.
5. [Optional] The model is implemented on an Android app, a character and edge detection image algorithm is built and sample photographs are extracted using the smartphone's camera to test the model in real-life circumstances.

Metrics

Accuracy. The model performance is measured by the accuracy of classifying a sequence of digits from the total dataset size.

$$Accuracy = \frac{\text{Number of correct digit sequences}}{\text{Total number of digit sequences in dataset}}$$

For multi-digit label datasets, a prediction is deemed to be classified correctly only when the exact sequence of digits is predicted correctly. For example in a 5-digit sequence, even a misclassification on only one digit when the others are correctly classified will not be considered to have achieved its accuracy goal. If the true label for the digit sequence is [3, 5, 6] and the model predicted [3, 5, 2], for this instance, the model has misclassified the labels, despite getting the last digit incorrectly. Thus, no 'partial correctness' is awarded in order to preserve the goals of the classification task. (Goodfellow, I., et. al., 2014)

The accuracy measure of exact digit sequences is considered in this project, as a basis for comparison with the reference paper by (Goodfellow, I., et. al., 2014). Prediction of all digits correctly in a given sequence is paramount to reduce the chances of incorrect transcription errors, which can be costly and time-consuming to rectify manually. Thus, we essentially consider only the true positives for the accuracy.

To further evaluate the model, confusion matrices are plotted, where the predicted classes versus the actual classes relationship can be visualized. Then, further metrics such as **precision**, **recall** and **F1-score** can be calculated, which provide a better gauge of the model's representation power.

$$Precision = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Positives\ (FP)}$$

$$Recall = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Negatives\ (FN)}$$

$$F1-Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Cross entropy loss. The Cross entropy loss is a standard model evaluation metric of the loss function (or cost function) value in which the optimization algorithm of a particular model is trying to minimize. The loss function is typically represented by logistic regression (logits) components projected over softmax functions to produce a probabilistic classifier output. In a general sense, the Cross entropy loss can be expressed as:

$$L(X, Y) = -\frac{1}{n} \sum_{i=1}^n y^i \ln a(x)^i + (1 - y^i) \ln(1 - a(x)^i)$$

Where $X = \{x^1, \dots, x^n\}$ is the set of input examples in the training dataset and $Y = \{y^1, \dots, y^n\}$ is the corresponding set of labels for those input examples. The $a(x)$ represents the output of the neural network given input x . The labels for each training example i in y^i is either 0 or 1, while the output activation $a(x)$ is typically restricted to the open interval (0, 1) by using a logistic softmax function. A model with lower cross entropy loss usually has better performance than otherwise.

2. Analysis

Data Exploration

The SVHN dataset comes in two obtainable formats: the first containing original images of digit sequences (Format 1), and the second containing cropped images centered on a single digit (Format 2). It contains 630,420 real-world digit images of street numbers obtained by Google Street View. It is split into three datasets: 73,257 digits for training, 26,032 digits for testing and 531,131 additional, somewhat less difficult samples to use as extra training data. Format 1 has the

original images of varying resolutions with character level bounding boxes, while Format 2 are cropped MNIST-like 32-by-32 pixel images centered around a single character (which do contain some distractors at the sides).



SVHN Format 1



SVHN Format 2

The distribution of the digit sequence length compared to the total number of RGB images (248,823) in Format 1 is as below in Table 1 and Figure 1:

Digit Sequence Length	Number of Images	% Total
1	17,005	6.8
2	98,212	39.4
3	117,561	47.2
4	15,918	6.4
5	126	0.05
6	1	0.0004

Table 1: SVHN Digit Sequence Length

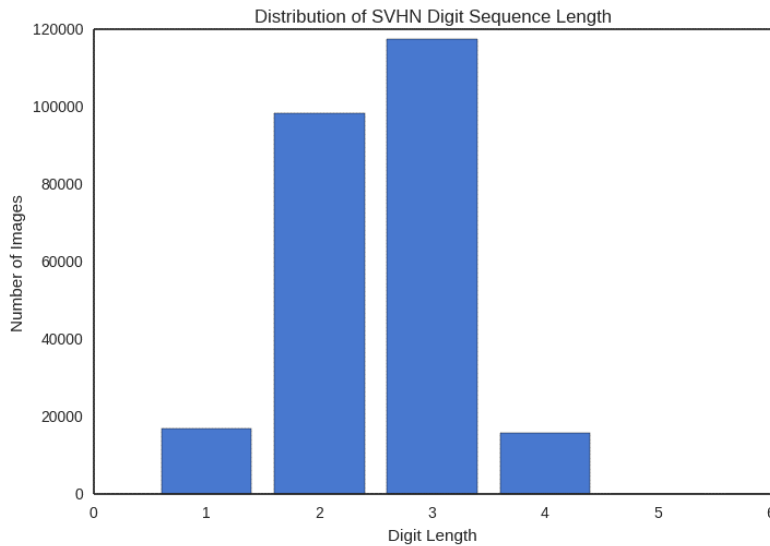


Figure 1: SVHN Digit Sequence Length

From the distribution, 3-digit sequence images have the highest composition at 47.2%, followed by 2-digit sequence images at 39.4%.

Outliers. Sequences of 5 and 6 digits are very low in relation to the rest of the dataset, constituting only 0.05% and 0.0004% respectively. This is an expected outcome as house numbers would typically be in the range from 1 to 3 digits, but rarely exceed beyond 5 digits. Such rare occurrences may be treated as an outlier and thusly omitted from the dataset. We omit the 1 data that is 6-digits in length.

Exploratory Visualization

Below is a distribution plot of the number of digits for the entire 630,420 digit dataset. From the plot, there is an imbalance of occurrences of the digits, as the most common digit is '1' while the least common is '9'. When there are more examples for a particular digit, the learning model is able to learn to recognize the digit easily as compared to lesser examples. Ideally, a dataset with equally distributed classes would mean equal example sizes for the model to learn, but in this case, since house numbers would generally be constrained to 2-3 digit sequence length, and also, lower numbered digits will appear more frequently than others, this is indeed representational of the real-world situation. If the problem to be solved requires recognizing digits of various perturbations (such as different fonts, scale and rotation), then more examples for each digit class will be beneficial for the model to learn from. However, if the model is used to recognize test data similar to SVHN house numbers, then the existing input data will provide a good basis for learning.

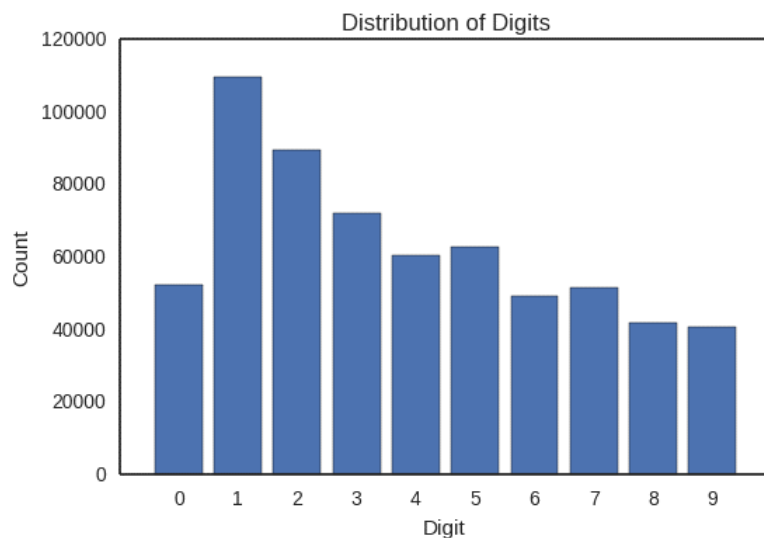


Figure 2: Distribution of digit classes of the SVHN dataset.

Algorithms and Techniques

Deep learning algorithms fall broadly into two main groups: supervised learning and unsupervised learning. With these groupings, it can be structurally broken down into either deep feedforward networks or probabilistic models. Examples of deep feedforward networks are Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) and Autoencoders (AE), whereas Deep Belief Networks (DBN), Restricted Boltzmann Machines (RBM) and Deep Boltzmann Machines (DBM) are probabilistic models. The choice of an algorithm will largely be dependent on the problem description and the nature of the input data to be used for that particular domain. A feedforward network with fully connected layers would be suitable for a supervised learning with fixed-size vectors as input data. While, if input data is of known topological structure (such as images), CNNs have proven to be effective. For modelling sequences, such as speech recognition and text recognition, a RNN will fare better. (Goodfellow, I et. al., 2016)

Since the input data for our digit-recognition problem consists of images, the suitable algorithm to be used is a convolutional neural network. In computer vision academic research, CNNs are the dominant deep neural networks in use which generate state-of-the-art performance for datasets like SVHN. (Szegedy, C. et. al., 2015)

Convolutional Neural Network Terminologies

Structurally, the CNN is a specialized kind of neural network for processing data that has a known, grid-like topology. 1D grid topologies such as time-series data and 2D grid such as image of pixel values are commonly encountered examples. Definition-wise, CNNs employ a specialized kind of linear operation called ‘**convolution**’ in place of general matrix multiplication in at least one of their layers. (Goodfellow, I et. al., 2016) The general convolution operation can be mathematically expressed as the following:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t - a)$$

Where $w(a)$ is a weighting function with a as the age of a measurement of a sensor, which finds the weighted average over the sensor’s outputs $x(t)$ with respect to time index t . More weight is given for recent measurements as they are more relevant. Time is discretized as the sensor provides data at regular intervals and the discrete convolution function $s(t)$ takes the sum of the weighted average values of x , also known as the **input** and function w , also known as the **kernel**. The output of a convolution function is referred to as the **feature map**. The convolution operation is also expressed with an asterisk (*).

For convolution operations over two axes (for processing 2D images), we have the following:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Where I is the 2D image input with horizontal and vertical pixel indices (i, j) and K is the kernel with (m, n) age of measurement. (Goodfellow, I et. al., 2016)

CNNs have important properties called sparse interactions, parameter sharing and equivariant representations that are leveraged for better performance. **Sparse interactions** refers to the connectivity of the neural node units from one layer to another. Traditional neural networks have every input unit from a layer interacting with every output unit of another layer. However, CNNs do not have all units interacting with each other at the same time. This is achieved by making the kernel smaller than the input. For example, when processing an image, the input may have thousands of pixels, but by using a small kernel we can detect small, meaningful features such as edges which only occupy tens of pixels. This allows for fewer parameters to be stored, hence lower memory requirements and also improves statistical efficiency. (Goodfellow, I et. al., 2016)

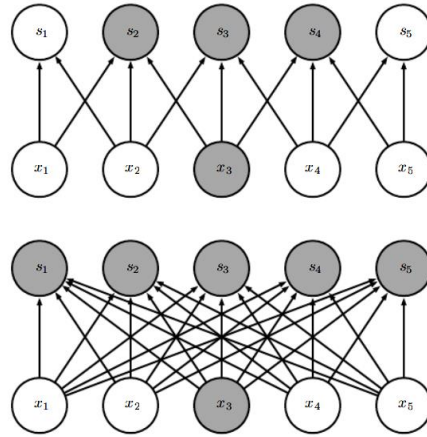


Figure 3: Sparse Interactions of CNN (top) versus traditional neural networks (bottom). The s nodes in grey are the ones interacting and affected by node x . (Goodfellow, I et. al., 2016)

Parameter sharing. This is the concept where nodes in layers can use the same parameter for more than one function in a model. In traditional neural nets, each weight matrix is used only once when computing the output of a layer, whereas in CNNs, a network has tied weights in the sense that the weight applied to one layer is tied to the weight applied elsewhere. This means that rather than learning separate sets of parameters, CNNs only need to learn one set, further reducing the storage requirements to model these parameters.

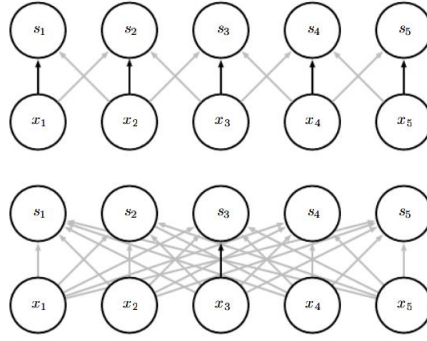


Figure 4: Parameter sharing of CNN (top) versus traditional neural networks (bottom). The black arrows indicate the use of parameters across layers. (Goodfellow, I et. al., 2016)

Equivariance. This refers to when the input changes, the output changes in the same way as well. CNN are equivariant to shifts and translation of the input. This promotes parameter sharing across the input image. However, CNNs are not naturally equivariant to transformations such as rotation and scale of an image. (Goodfellow, I et. al., 2016) There are several ways to solving this, but a simple solution is to add additional rotated and scaled inputs to the training dataset.

Apart from that, a common operation used in CNNs are pooling. These are typically found after the output of nonlinear activation functions (such as a Rectified Linear Unit (ReLU)), once convolutions are performed in a layer. Pooling function modifies the output of the layer further with summary statistics of the nearby outputs. **Max pooling** reports the maximum output within a rectangular neighborhood, while **Average pooling** reports the average similarly. These help to make the representation become approximately invariant to small translations of the input. (Goodfellow, I et. al., 2016)

Convolutional Neural Network Architecture

The design of CNN architectures is sometimes thought of more of an art than science, as varying designs have been shown to work at a comparable performance. However, there are certain heuristics that one may follow to make better choices for their model design. One is by starting off a design based on previously attempted and published models that has shown good performance for the same or similar problem domain and input data. Then, incremental changes can be made to the design against a benchmark. Also, one may employ conventionally used parameter values for convolutional (filter) kernel size, strides, padding, pooling size, number of layers used, feature map size of the final layer and so on. Hyperparameter tuning can be done once a baseline model is set up, with an aim to optimize and improve the model. These include learning rate, regularization, activation function, etc. Significant performance gains may be observed if hyperparameters are tuned appropriately. These are further explored in **3.Refinement**. Available computational resources, time, and network complexity need to be balanced by

setting a clear goal to achieve a certain level of intended performance out of the model.

The final model is presented here after various optimization and improvement tasks, while details of the implementation are further discussed in the later sections.

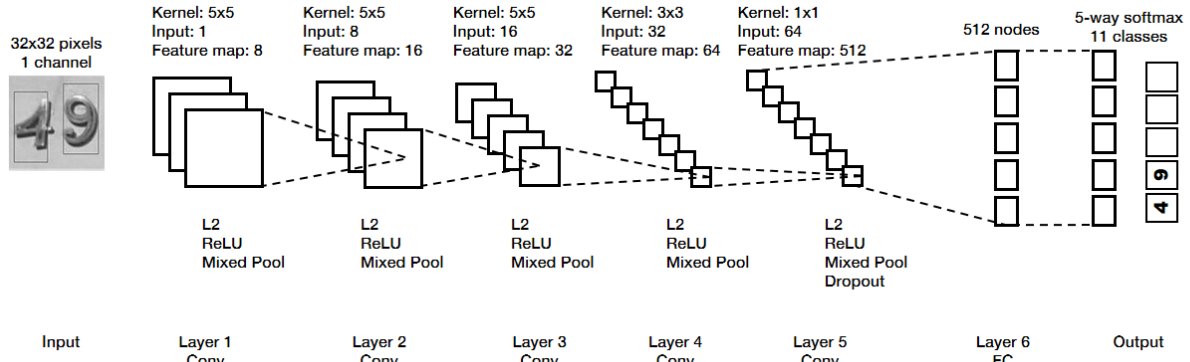


Figure 5: 6-layer CNN with Mixed Max-Avg Pooling.

Based on the above figure, the model consists of 5 convolutional layers with 1 fully (densely) connected layer. The input sizes of the convolutional layers are [1, 8, 16, 32, 64]. Whilst the output (feature map) sizes of the convolutional layers are [8, 16, 32, 64, 512]. Convolution kernel sizes used are [5x5, 5x5, 5x5, 3x3, 1x1], with all convolutional layers set to a stride of 1. Each convolutional layer is stacked onto one another similar to the architecture pioneered by AlexNet. (Krizhevsky. A., et. al., 2012) Every convolutional layer has a ReLU (Rectified Linear Unit) activation function, followed by a Mixed Max-Avg Pooling method as proposed by (Lee, C. Y., Gallagher, P. W. & Tu, Z., 2015), which has been shown to outperform vanilla Max pooling and Average pooling. The mixing factor of the Max and Average Pooling are 50% each, or 0.5. For regularization techniques, L2 and dropout are used. L2 regularization is applied at all convolutional layers with a coefficient of 0.0005, while dropout is only applied at Layer 5 with a probability of keeping nodes set at 0.5. Both L2 and dropout help to reduce the possibility of the model to overfit. Dropout increases the number of iterations required to converge by approximately a factor of 2. (Krizhevsky. A., et. al., 2012). To preserve the representation size of the learning, implicit zero padding is used throughout all convolutional and pooling operations. The fully connected layer is a 5-way matrix multiplication over weights and biases with 512 nodes each, before connecting to a 5-way softmax where the number of softmax operations represent the sequence length of the predicted digits, and each softmax handles a single digit from 11 classes (0,...,9 and blank digits). This structure is a slight modification from (Goodfellow, I., et. al., 2014) where they used a 6-way softmax in which the additional 6th softmax predicts the probability of

the sequence length of the digits. Our intention of omitting it is to simplify the architecture and only focus on predicting a maximum of 5-digit sequences. The loss function is the summation of the sparse cross entropy against the logits over the 5 softmax classifiers. An adaptive moment learning algorithm, Adam (Kingma, D. & Ba, J., 2014), is chosen as the optimizer with a tuned learning rate, γ , of 0.0005 and decay factor of 0.95 over every 10,000 iterations. An optimizer is a function which minimizes the previously defined loss function of our model.

The Adam optimizer (short for Adaptive Moment Estimation) is a stochastic optimization algorithm, created to combine advantages of two other optimizers, namely AdaGrad (which works well for sparse gradients) and RMSProp (which works well in non-stationary settings). The idea is to maintain exponential moving averages of the gradient and its square. A simplified illustration of the algorithm is as below (Cohen, N., 2015):

```

 $M_0 = 0$  (Initialize first moment vector)
 $R_0 = 0$  (Initialize second moment vector)
 $t = 0$  (Initialize timestep)
 $L_t$ : Stochastic objective function with parameters  $W$ 
 $g_t = \nabla_W L_t(W_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

For  $t = 1, \dots, T$ :
   $M_t = \beta_1 M_{t-1} + (1 - \beta_1) g_t$  (first moment estimate)
   $R_t = \beta_2 R_{t-1} + (1 - \beta_2) g_t^2$  (second moment estimate)
   $\hat{M}_t = \frac{M_t}{1 - (\beta_1)^t}$  (first moment bias correction)

   $\hat{R}_t = \frac{R_t}{1 - (\beta_2)^t}$  (second moment bias correction)

   $W_t = W_{t-1} - \gamma \frac{\hat{M}_t}{\sqrt{\hat{R}_t} + \varepsilon}$  (Update rule)
Return  $W_t$ 

```

Where M_t is the first-order gradient, R_t is the second-order gradient, $\beta_1 \in [0, 1]$, is the first moment decay rate, $\beta_2 \in [0, 1]$, is the second moment decay rate, $\gamma > 0$, is the learning rate, and $\varepsilon > 0$, is a numerical constant. This method computes the adaptive learning rates for different parameters, from estimates of the first and second moments of the gradients. (Kingma, D. & Ba, J., 2014). The default parameters of $\beta_1 = 0.001$, $\beta_2 = 0.999$ and $\varepsilon = 1 \times 10^{-8}$ are used for the Adam optimizer as proposed by Kingma et. al.

The Adam algorithm is chosen as it has the best performance over others in our experimentations, and it has advantages of being good for sparse gradients, having

a low memory footprint, and suits a wide-range of non-convex optimization problems. (Kingma, D. & Ba, J., 2014).

Benchmark

SVHN is one of the more common datasets used in academia research as benchmarks for developing new algorithms and architectures, alongside other computer vision domain datasets such as MNIST, CIFAR-10 and CIFAR-100. However, it is important to note that training deep models are very computationally costly: the AlexNet model of 5 convolutional layers and 3 fully connected layers was trained for 5-6 days on two NVIDIA GTX 580 3GB GPUs while the VGG Net model (Simonyan, K. & Zisserman, A., 2015) of simple components but deep architecture (19 layers) necessitated 2-3 weeks to be trained on four NVIDIA Titan Black GPUs. In the reference paper (Goodfellow, I., et. al., 2014), they trained their 8 convolutional layer, 1 locally connected layer and 3 fully connected layer CNN for 6 days.

Hence, in our implementation, we made tradeoffs to maintain a tolerable training time whilst still providing good performance that only require modest hardware capabilities to be trained on. The model is trained on an Amazon Web Services EC2 Instance launched in a virtual server in the cloud (in a compatible [Amazon Machine Images \(AMI\)](#)) and linked to their [c4.xlarge](#) Instance type which is equipped with a quad-core Intel Xeon E5-2666 v3 (Haswell) processor with 7.5GB memory. We chose a compute-optimized CPU approach instead of GPU for a much lower cost-to-performance ratio as we are using the paid version of Amazon's service (about USD 0.30/hour). Training typically took 5 hours on our model, performing early-stopping measures validated against the validation set, for a much more tolerable training time versus several days or weeks it may take otherwise.

With these time, cost and resource constraints, we hypothetically aim for a 96% accuracy (4% error) on the test set as the overall model performance. To give a perspective for comparison, human-level transcription performance for SVHN images is measured at 98% accuracy (2% error), and some of the best published models outperform this by a margin of 0.1-0.3% (Benenson, R., 2016)

3. Methodology

Data Preprocessing

The dataset is preprocessed with two simple methods: converting the images to grayscale and applying Global Contrast Normalization (GCN). The aim is to not place too much effort on preprocessing so that the model learns to be invariant to

the most important features, and is especially true with large datasets. (Goodfellow, I et. al., 2016). Thus, no ZCA Whitening, Local Contrast Normalization (LCN) nor data augmentation is implemented. To prepare the .png images for preprocessing, first, the digit bounding box information is extracted from the digitStruct.mat file from train (train.tar.gz), test (test.tar.gz) and extra (extra.tar.gz) datasets into a h5py python dictionary, which contains information of the boxes' coordinates (in terms of height, left, top, width of the image) and their labels (0,...,9). We then convert the .png images of pixels into numpy arrays and apply the preprocessing steps. **Grayscale**. The RGB images are reduced to grayscale, G_y , expressed as the weighted sum of the Red, Green and Blue channels as below:

$$G_y = 0.2989R + 0.5870G + 0.1140B$$

Then, **Global Contrast Normalization (GCN)** is applied to the images such that the histogram of gray values in an image lie within 1 standard deviation across the entire dataset. This will prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant, s . The formula from (Goodfellow, I et. al., 2016), but simplified for grayscale images is as below:

$$X' = s \frac{X - \bar{X}}{\max\{\epsilon, \sqrt{\lambda + (X - \bar{X})^2}\}}$$

Where X is the input image, X' is the output image, ϵ is an positive number bias to prevent division by 0 errors, and λ is a positive regularization parameter meant to bias the estimate of standard deviation. In most cases, $\lambda = 0$ is the default setting as it is safe to ignore the small denominator problem. ϵ is arbitrarily set to a small number of 1×10^{-4} and the constant, s , is set to 1 to make each individual pixel have standard deviation across examples close to 1. (Goodfellow, I et. al., 2016) The resulting preprocessing steps are illustrated as below.



Figure 6: SVHN images before preprocessing (top) and after preprocessing (bottom). Notice that contrast-normalized images have more defined edges within the digits. Certain digits have poor clarity (top: '9', bottom: '3', '7'), but this allows the model to learn the salient features of the digits.

Once grayscaling and GCN are applied, the images are then cropped by their digit bounding boxes and resized to fit into 32-by-32 pixel images as the final input images. To prepare for the labels, we encode blank digits as '10', and process the labels in the format of an N-digit sequence. Since the original .mat files indexes digit '0' as '10', we encode it to '0' to represent digit '0'. For example, an image containing digit-sequence '120' will have a label format of [1, 2, 0, 10, 10], whilst '4163' will be labelled as [4, 1, 6, 3, 10]. We keep the labels to a maximum length of 5-digit sequences. In the SVHN dataset, there is only one image of a 6-digit sequence. Since the occurrence is very rare (referring to Table 1), we omit this image from the dataset and treat it as an outlier.

Then, the combined dataset from train, test and extra is split into the final training set, validation set, and test set. The method used is as proposed by (Sermanet, P. Chintala, S. & LeCun Y., 2012) as a random shuffling is done to construct the validation set, where 2/3 from the training samples (400 per class) and 1/3 from the extra samples (200 per class) is obtained. This is done to measure success on easy samples (from the extra dataset) but still place an emphasis on more difficult samples (from the train dataset). The resulting split produces a training set of 230,070 images, a validation set of 5684 images and a test set of 13,068 images.

Implementation

The implementation is based on TensorFlow, a deep learning library, with supporting Python libraries including numpy (for mathematical calculations), matplotlib and seaborn (for plotting and visualization). TensorBoard is also used for visualizing the model learning. The code base is implemented in Python 2.7 using a Jupyter notebook in an Ubuntu machine from Amazon Web Services (AWS). The AMI used has all necessary libraries pre-installed to their latest versions.

In this section, the implementation code is presented for the final model as introduced in *Algorithms and Techniques*. However, we do provide the graphical representation of the model at initial implementation and the model after refinement for comparison.

The preprocessed input images are 32-by-32 pixels in dimension, which have been normalized with GCN and converted to grayscale. The preprocessed dataset is saved in a pickle file, to be easily loaded into any model code base. The code which runs the preprocessing resides in 3_preprocess_multi.ipynb. The model implementation code can be found in 4_model_multi.ipynb. First, we import the necessary libraries and load the pickle file.

```
from __future__ import print_function
import numpy as np
import tensorflow as tf
from six.moves import cPickle as pickle
from six.moves import range
```

```

# Load pickle file of preprocessed dataset

pickle_file = 'SVHN_multi_2.pickle'

with open(pickle_file, 'rb') as f:
    save = pickle.load(f)
    train_dataset = save['train_dataset']
    train_labels = save['train_labels']
    valid_dataset = save['valid_dataset']
    valid_labels = save['valid_labels']
    test_dataset = save['test_dataset']
    test_labels = save['test_labels']
    del save # Hint to help gc free up memory

print('Training set', train_dataset.shape, train_labels.shape)
print('Validation set', valid_dataset.shape, valid_labels.shape)
print('Testing set', test_dataset.shape, test_labels.shape)

```

To implement, we first initialized a TensorFlow graph, where all our model components will reside. We can then specify input placeholders for tensors: we initialize the training set input tensor placeholder of shape (batch_size, img_size, img_size, num_channels) for the features, where batch_size refers to the number of batches of data rows to be fed into the tensor, img_size is the width and height of the input image in pixels, and num_channels is the number of channels that represent the input image. For the labels input tensor, we specify a shape of (batch_size, seq_length) where seq_length is the sequence length of the labels (from 1 to 5). In both the validation and testing datasets, a constant tensor is initialized to be later used.

```

# Convolutional Neural Network Architecture

# Define a few constants.
img_size = 32 # Input data is 32x32.
num_channels = 1 # Grayscale
num_labels = 11 # 0 - 9, + blank digits
seq_length = 6

batch_size = 128

# Create a TensorFlow graph.
graph = tf.Graph()

with graph.as_default():

    # Input data.
    # Initialize a placeholder of shape [128, 32, 32, 1] for the train dataset.
    tf_train_dataset = tf.placeholder(
        tf.float32, shape=(batch_size, img_size, img_size, num_channels))
    # Initialize a placeholder of shape [128, 6] for the train labels.
    tf_train_labels = tf.placeholder(tf.int32, shape=(batch_size, seq_length))
    # Initialize a constant for validation dataset.
    tf_valid_dataset = tf.constant(valid_dataset)
    # Initialize a constant for test dataset.
    tf_test_dataset = tf.constant(test_dataset)

```

For a cleaner code, several helper functions are created for convolution and pooling operations. Strides for convolution are set to 1, and all padding are set to zero (padding='SAME'). Max, average and mixed pooling have dimensions of 2x2 and stride 2. Details on Mixed Max-Avg pooling is further explained in section 3.

Refinement.

```
# Helper functions for convolution and pooling
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME') # Stride = 1, Padding = 0

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME') # Size = 2x2, Stride = 2, Padding = 0

def avg_pool_2x2(x):
    return tf.nn.avg_pool(x, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME') # Size = 2x2, Stride = 2, Padding = 0

# Mixed pooling by combining max pool and avg pool. alpha is the mixing factor.
def mixed_max_avg_pool_2x2(x, alpha):
    return (alpha * max_pool_2x2(x) + (1 - alpha) * avg_pool_2x2(x))
```

Then, variables for each layer is defined as below. In each shape represents shape=[kernel size, kernel size, number of input neurons, number of output neurons]. All layers use Xavier initialization for weights and a constant of 0 for biases. L2 Regularization of factor 0.0005 is applied to all convolutional layers. In the fully connected layers (with weights WS1,...,WS5), the shape corresponds to shape=[number of input nodes, number of class labels]. Thus, the feature map of the fully connected layers contain 512 nodes.

```
# Variables.
# Convolution layer variables (W_conv_1, ..., W_conv_5)
# Xavier initializer is used for all weights.
# Initialization constant of 0.0 is used for all biases.
# L2 Regularization is used for all conv layers.
# Dropout in 5th conv layer.

# 5x5 conv kernel, input = 1, output = 8
W_conv_1 = tf.get_variable("W1", shape=[5, 5, 1, 8],
                           initializer=tf.contrib.layers.xavier_initializer_conv2d(),
                           regularizer=tf.contrib.layers.l2_regularizer(0.0005))
b_conv_1 = tf.Variable(tf.constant(0.0, shape=[8]))

# 5x5 conv kernel, input = 8, output = 16
W_conv_2 = tf.get_variable("W2", shape=[5, 5, 8, 16],
                           initializer=tf.contrib.layers.xavier_initializer_conv2d(),
                           regularizer=tf.contrib.layers.l2_regularizer(0.0005))
b_conv_2 = tf.Variable(tf.constant(0.0, shape=[16]))
```



```

# 5x5 conv kernel, input = 16, output = 32
W_conv_3 = tf.get_variable("W3", shape=[5, 5, 16, 32],
initializer=tf.contrib.layers.xavier_initializer_conv2d(),
regularizer=tf.contrib.layers.l2_regularizer(0.0005))
b_conv_3 = tf.Variable(tf.constant(0.0, shape=[32]))

# 3x3 conv kernel, input = 32, output = 64
W_conv_4 = tf.get_variable("W4", shape=[3, 3, 32, 64],
initializer=tf.contrib.layers.xavier_initializer_conv2d(),
regularizer=tf.contrib.layers.l2_regularizer(0.0005))
b_conv_4 = tf.Variable(tf.constant(0.0, shape=[64]))

# 1x1 conv kernel, input = 64, output = 512
W_conv_5 = tf.get_variable("W5", shape=[1, 1, 64, 512],
initializer=tf.contrib.layers.xavier_initializer_conv2d(),
regularizer=tf.contrib.layers.l2_regularizer(0.0005))
b_conv_5 = tf.Variable(tf.constant(0.0, shape=[512]))

# Fully connected (FC) variables.
# Input = 512 nodes, Output = 11 classes
W_S1 = tf.get_variable("WS1", shape=[512, 11],
initializer=tf.contrib.layers.xavier_initializer())
b_S1 = tf.Variable(tf.constant(0.0, shape=[11]))
W_S2 = tf.get_variable("WS2", shape=[512, 11],
initializer=tf.contrib.layers.xavier_initializer())
b_S2 = tf.Variable(tf.constant(0.0, shape=[11]))
W_S3 = tf.get_variable("WS3", shape=[512, 11],
initializer=tf.contrib.layers.xavier_initializer())
b_S3 = tf.Variable(tf.constant(0.0, shape=[11]))
W_S4 = tf.get_variable("WS4", shape=[512, 11],
initializer=tf.contrib.layers.xavier_initializer())
b_S4 = tf.Variable(tf.constant(0.0, shape=[11]))
W_S5 = tf.get_variable("WS5", shape=[512, 11],
initializer=tf.contrib.layers.xavier_initializer())
b_S5 = tf.Variable(tf.constant(0.0, shape=[11]))

```

The model architecture is then designed. 5 convolutional layers followed by a 5-way fully connected layer. Convolution is performed on the input data, then a ReLU activation function is applied for nonlinearity, and finally Mixed Max-Avg pooling is done. Here, the mixing ratio of Max and Avg pooling is 0.5 or 50%. The dropout is implemented at the last convolutional layer, with the configurable probability of keeping the nodes, `keep_prob`. To set up the fully connected layers (S1,...,S5), the output of convolution layer 5 has to be reshaped to include connections to all neurons, accomplished by `reshape = tf.reshape(drop, [shape[0], shape[1] * shape[2] * shape[3]])`. For example, if the convolution tensor outputs a shape of [128, 1, 1, 512], the reshape operation will be: [128, 1x1x512 = 512 nodes].

```

# Model.
# Create a 5 conv layer, apply ReLU activation and mixed pooling.
# Dropout added at layer 5.
def model(data, keep_prob):

```



```

conv = conv2d(data, W_conv_1)
hidden = tf.nn.relu(conv + b_conv_1)
sub = mixed_max_avg_pool_2x2(hidden, 0.5)

conv = conv2d(sub, W_conv_2)
hidden = tf.nn.relu(conv + b_conv_2)
sub = mixed_max_avg_pool_2x2(hidden, 0.5)

conv = conv2d(sub, W_conv_3)
hidden = tf.nn.relu(conv + b_conv_3)
sub = mixed_max_avg_pool_2x2(hidden, 0.5)

conv = conv2d(sub, W_conv_4)
hidden = tf.nn.relu(conv + b_conv_4)
sub = mixed_max_avg_pool_2x2(hidden, 0.5)

conv = conv2d(sub, W_conv_5)
hidden = tf.nn.relu(conv + b_conv_5)
sub = mixed_max_avg_pool_2x2(hidden, 0.5)

drop = tf.nn.dropout(sub, keep_prob)

shape = drop.get_shape().as_list()
print(shape)
# Reshape output of conv layer 5 for input of FC layer
reshape = tf.reshape(drop, [shape[0], shape[1] * shape[2] * shape[3]])

# FC layer. Linear matrix multiplication.
S1 = tf.matmul(reshape, W_S1) + b_S1
S2 = tf.matmul(reshape, W_S2) + b_S2
S3 = tf.matmul(reshape, W_S3) + b_S3
S4 = tf.matmul(reshape, W_S4) + b_S4
S5 = tf.matmul(reshape, W_S5) + b_S5
return [S1, S2, S3, S4, S5]

```

For training computation, the cross entropy loss function is defined as below. Here, `tf.nn.sparse_softmax_cross_entropy_with_logits` is used as the output of the softmax function will be sparse classes, i.e. only one digit class can be represented for each digit sequence slot. For example, the digits '2' and '5' cannot coexist in the same slot in the sequence of maximum $N=5$. By passing in `tf_train_labels[:, (1,..., 5)]` to the softmax function argument, we specify which slot label is meant for which fully connected layer (S1,...,S5).

```

# Training computation.
keep_prob = 0.5 # Probability of keeping nodes for dropout.
[S1, S2, S3, S4, S5] = model(tf_train_dataset, keep_prob)
with tf.name_scope('cross_entropy'):
    # Loss function.
    # Each digit sequence slot is sparse (i.e. two digits cannot occupy the same
slot)
    # Thus, loss is calculated across each digit slot in a 5-way softmax.
    loss = tf.reduce_mean(
        tf.nn.sparse_softmax_cross_entropy_with_logits(S1,
tf_train_labels[:,1])) +\
        tf.reduce_mean(

```

```

        tf.nn.sparse_softmax_cross_entropy_with_logits(S2,
tf_train_labels[:,2])) +\
        tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(S3,
tf_train_labels[:,3])) +\
        tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(S4,
tf_train_labels[:,4])) +\
        tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(S5,
tf_train_labels[:,5]))
    tf.scalar_summary('cross entropy', loss)

```

To minimize our loss function, we employ an optimizer algorithm, Adam. The initial learning rate is set to 0.0005, and it decays by a factor of 0.95 with every 10,000 iterations.

```

# Optimizer.
global_step = tf.Variable(0)
starter_learning_rate = 0.0005
# Decay factor of 0.95 after every 10000 steps.
with tf.name_scope('learning_rate'):
    learning_rate = tf.train.exponential_decay(starter_learning_rate,
global_step, 10000, 0.95)
    tf.scalar_summary('learning_rate', learning_rate)
with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss,
global_step=global_step)

```

The main metric to be defined is accuracy, as below. Here, it is defined as a function to be called at run time. The output of predictions is a set of normalized pixel values for the digit sequence, and the argmax is taken (to get the predicted digit labels) and transposed before comparing with the actual labels in labels. All correct predictions is summed up and divided by the total digit sequence, predictions.shape[1], and subsequently the total number of rows in the dataset prediction.shape[0].

```

# Accuracy calculations
def accuracy(predictions, labels):
    return (100.0 * np.sum(np.argmax(predictions, 2).T == labels) # Compare the
predictions of the model vs true labels.
        / predictions.shape[1] / predictions.shape[0])

```

To produce the confusion matrix, the following code is implemented. The matrix is plotted for each digit sequence slot, N in [N1, N2, N3, N4, N5], thus we get five matrices. Then the precision, recall and F1-scores are averaged over these matrices to get the overall model scores.

```
# Plot confusion matrix.
from pandas_ml import ConfusionMatrix
import matplotlib.pyplot as plt
% matplotlib inline
def conf_mat(y_true, y_pred):
    cm = ConfusionMatrix(y_true, y_pred)
    cm.plot()
    plt.show
    print(cm)
```

We define the prediction runs for training, validation and test datasets as below. The model is fed the respective datasets with the dropout rate. Note that for training, we want to enable dropout with `keep_prob`, however for validation and testing, we would like to turn it off by setting dropout rate = 1, so as to not affect the prediction results at test time. `tf.pack` is used to concatenate all the output of softmax for each digit slot sequence.

```
# Predictions for the training, validation, and test data.
# Feed the the model with training, validation and test datasets respectively.
train_prediction = tf.pack([tf.nn.softmax(model(tf_train_dataset,
keep_prob)[0]),\
                           tf.nn.softmax(model(tf_train_dataset, keep_prob)[1]),\
                           tf.nn.softmax(model(tf_train_dataset, keep_prob)[2]),\
                           tf.nn.softmax(model(tf_train_dataset, keep_prob)[3]),\
                           tf.nn.softmax(model(tf_train_dataset, keep_prob)[4])])

# Turn off dropout (keep_prob = 1.0) for validation.
valid_prediction = tf.pack([tf.nn.softmax(model(tf_valid_dataset, 1.0)[0]),\
                           tf.nn.softmax(model(tf_valid_dataset, 1.0)[1]),\
                           tf.nn.softmax(model(tf_valid_dataset, 1.0)[2]),\
                           tf.nn.softmax(model(tf_valid_dataset, 1.0)[3]),\
                           tf.nn.softmax(model(tf_valid_dataset, 1.0)[4])])

# Turn off dropout (keep_prob = 1.0) for testing.
test_prediction = tf.pack([tf.nn.softmax(model(tf_test_dataset, 1.0)[0]),\
                           tf.nn.softmax(model(tf_test_dataset, 1.0)[1]),\
                           tf.nn.softmax(model(tf_test_dataset, 1.0)[2]),\
                           tf.nn.softmax(model(tf_test_dataset, 1.0)[3]),\
                           tf.nn.softmax(model(tf_test_dataset, 1.0)[4])])
```

An optional but rather useful addition is the use of TensorBoard for visualization purposes. Here, TensorBoard is used to visualize the cross entropy loss, learning rate, summary statistics and histograms for weights and biases for all layers. The implementation is as follows:

```
# Add variable summaries for TensorBoard
# Min, max, mean and std deviation
def variable_summaries(var, name):
    """Attach a lot of summaries to a Tensor."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.scalar_summary('mean/' + name, mean)
```

```

with tf.name_scope('stddev'):
    stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
tf.scalar_summary('stddev/' + name, stddev)
tf.scalar_summary('max/' + name, tf.reduce_max(var))
tf.scalar_summary('min/' + name, tf.reduce_min(var))
tf.histogram_summary(name, var)

```

```

# Tensorboard summaries for weights and biases.
with tf.name_scope('W1'):
    W_conv_1
    variable_summaries(W_conv_1, 'W1_weights')
with tf.name_scope('b1'):
    b_conv_1
    variable_summaries(b_conv_1, 'b1_biases')
with tf.name_scope('W2'):
    W_conv_2
    variable_summaries(W_conv_2, 'W2_weights')
with tf.name_scope('b2'):
    b_conv_2
    variable_summaries(b_conv_2, 'b2_biases')
with tf.name_scope('W3'):
    W_conv_3
    variable_summaries(W_conv_3, 'W3_weights')
with tf.name_scope('b3'):
    b_conv_3
    variable_summaries(b_conv_3, 'b3_biases')
with tf.name_scope('W4'):
    W_conv_4
    variable_summaries(W_conv_4, 'W4_weights')
with tf.name_scope('b4'):
    b_conv_4
    variable_summaries(b_conv_4, 'b4_biases')
with tf.name_scope('W5'):
    W_conv_5
    variable_summaries(W_conv_5, 'W5_weights')
with tf.name_scope('b5'):
    b_conv_5
    variable_summaries(b_conv_5, 'b5_biases')

```

Once the model is designed, it is ready for a training run. We create a session, initialize all variables and prepares the input data in batches. The `offset` creates the batch splitting of the dataset. Here, a batch size of 128 is used, and the model is run through 100,000 iteration steps. A feed dict is created and the training dataset is fed into the model. When the model has reached every 1,000 iterations, the accuracy for training is evaluated. Similarly the validation and test dataset are also fed into the feed dict, and the accuracy of validation and test set is evaluated. For model comparison and plotting with matplotlib, these information on the accuracy and loss are stored into a .csv file. For TensorBoard visualization, all summaries are merged and a summary writer is created. Periodically as well, the model parameters learnt so far is saved as a checkpoint file. This is to ensure model recoverability when training needs to be halted halfway. The TensorBoard event

summaries are saved in the path: /home/ubuntu/temp/trial/t013 while plotting information and checkpoint files are saved in the default directory of the AMI: /home/ubuntu

```
import csv
num_steps = 100001 # Specify training iterations.
# Initialize empty dict for accuracy and loss plot.
train_acc_plot, valid_acc_plot, loss_plot = {}, {}, {}

# Create a TensorFlow session, and initialize all variables.
with tf.Session(graph=graph) as session:
    tf.initialize_all_variables().run()
    saver = tf.train.Saver(tf.all_variables()) # Initialize model saver.
    print('Initialized')
    from time import time
    t0 = time()

    # Merge all the summaries and write them out to /home/... (Amazon AMI EC2
    Instance default dir)
    merged = tf.merge_all_summaries()
    writer = tf.train.SummaryWriter('/home/ubuntu/temp/trial/t013', session.graph)

    for step in range(num_steps):
        # Get offset index to split training dataset into batches.
        offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
        batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
        batch_labels = train_labels[offset:(offset + batch_size), :]
        # Initialize feed_dict as TensorFlow model inputs.
        feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}

        # Runs session for the optimizer to minimize the loss function and produce
        predictions.
        _, l, predictions, summary = session.run(
            [optimizer, loss, train_prediction, merged], feed_dict=feed_dict)
        if (step % 1000 == 0):

            train_accuracy = accuracy(predictions, batch_labels[:,1:6])
            valid_accuracy = accuracy(valid_prediction.eval(), valid_labels[:,1:6])

            print('Minibatch loss at step %d: %f' % (step, l))
            print('Minibatch accuracy: %.1f%%' % train_accuracy)
            print('Validation accuracy: %.1f%%' % valid_accuracy)

            # Store training, validation accuracy and loss data for plotting.
            train_acc_plot[step] = train_accuracy
            valid_acc_plot[step] = valid_accuracy
            loss_plot[step] = l

            # Write training, validation accuracy and loss data to csv file
            with open("fin_9_mixed max avg pool_relu_12 0.0005_lr 0.0005_5 conv_fc
512.csv", "a") as myfile:
                wrt = csv.writer(myfile, delimiter=',')
                wrt.writerow((step, train_accuracy, valid_accuracy, l))

            # Write summaries to TensorBoard
            writer.add_summary(summary, step)

            # Saves model checkpoint
            save_path =
            saver.save(session, 'CNN_model_multi_tmp5.ckpt', global_step=step)
```

```
print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(),
test_labels[:,1:6]))
print("Total run time:", round(((time()-t0)/60), 1), "min")
print("Model saved in file: %s" % save_path)
```

After a complete run, the learning curve and cross entropy loss curve is plotted for evaluation.

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Plot learning curve
sns.set_style("ticks")
fig, ax = plt.subplots()

ax.plot(sorted(train_acc_plot.keys()), train_acc_plot.values(), label='Train')
ax.plot(sorted(valid_acc_plot.keys()), valid_acc_plot.values(), label='Valid')
ax.set_xlabel('Training steps')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Learning Curve')
ax.legend(loc=4)
plt.show()

# Plot loss curve
plt.plot(sorted(loss_plot.keys()), loss_plot.values())
plt.xlabel('Training steps')
plt.ylabel('Loss')
plt.title('Cross Entropy')
plt.show()
```

Challenges and Solutions

During the implementation of the model, several debilitating challenges were encountered. These are discussed below:

Diverging loss values. The issue of loss values gradually increasing with each iteration means that the model optimization is not working. Usually, the culprit is due to a very high learning rate for the particular optimization algorithm in use. A learning rate that is too high will cause exploding gradient and hence a convergence failure. Introducing a learning rate decay over time helps in convergence as well. An optimally tuned learning rate with decay is able to produce great convergence.

Another reason could also be a weight or bias initialization, where if they are set too large, then the signal which passes through the layers will grow until it is too large to be useful. Conversely, if the weights or biases are too small, the signal becomes gradually smaller until it is too tiny to be useful. Apart from uniform distributions, commonly used initializer methods that have seen good performance of training large network are the Xavier method (Glorot, X. & Bengio, Y., 2010) and He method (He, K. et. al., 2015)

Low accuracy. This may have multiple causes, but the most common culprits are the learning rate and the number of training iterations the model is run on. Another important checking to be done is to check for the accuracy metric calculations defined, as mistakes in implementation could cause frustration in troubleshooting. Apart from changing the learning rate and checking the accuracy metric, one could increase the training iteration steps to see any immediate accuracy gains from the learning curve. If the training error is higher than the validation error, one can increase the model capacity by adding more layers or adding more hidden units. (Goodfellow, I et. al., 2016)

Preprocessing error. At times when preprocessing step is incorrectly implemented, this can have a catastrophic effect on model learning and performance. Our implementation was particularly affected by this. We consistently obtained a constant below-par accuracy of 57.5% with a high loss value of around 6.1 despite all aforementioned solutions have been investigated and taken into account. Only when we investigated the preprocessing step, we are able to identify the root cause of the problem. It was caused by an inappropriate implementation of the GCN, where we initially multiplied the denominator standard deviation of the formula with a small constant ϵ of 1×10^{-8} , intended to avoid division-by-zero errors. Our deduction was that this may cause extremely small values for the denominator, and when the numerator is divided by a very small denominator, it may cause an exploded exponential result, which ultimately impeded the model's capability to learn. The solution was to change to a selective substitution for small standard deviation values, where if it is less than 1×10^{-4} , it will be made to 1. However, it is important to note that we are able to achieve good convergence if the former method is applied to SVHN's Format 2 dataset of individual digit labels (1_preprocess_single.ipynb & 2_model_single.ipynb), but not on Format 1 (3_preprocess_multi.ipynb).

```
#im = (im - mean) / (1e-8 * std) # Apply GCN. Multiply by 1e-8 to avoid
division by zero errors.
if std < 1e-4: std = 1
im = (im - mean) / std # Apply normalization
```

Refinement

In this section, various hyperparameter tuning are explored and the optimal values are then used for the final model. Various batches of experiments are run in an attempt to visualize and understand the underlying relationships of the hyperparameters towards achieving better performance out of the model. Due to the number of experiments to be run under time constraints of this project, we used the following baseline architecture and modified related components accordingly for each experiment:

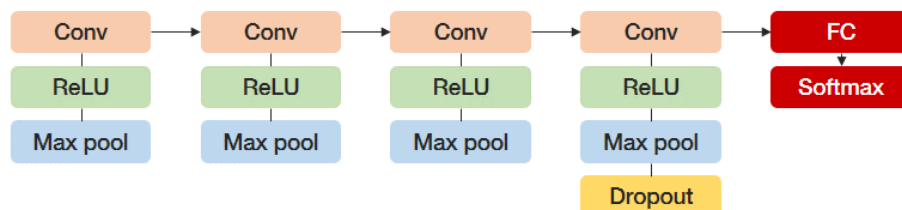


Figure 7: Initial model architecture used for refinement experiments. 4 convolutional layers, 4 ReLU activations and 4 Max pooling operations. Dropout is at the 4th convolution layer.

In each experiment, the model is trained for 15 minutes on average, with the maximum at 1 hour, thus allowing us to explore more hyperparameters experimentation with quicker feedback time. Although this is by no means an exhaustive search for optimal performance, we explore some common hyperparameter and architecture variations.

Learning Rate

The learning rate is regarded as the most important hyperparameter, as it controls the effective capacity of the model. The capacity of the model is optimized when a correct learning rate is used for the optimization problem, not necessarily an especially large or small value. (Goodfellow, I et. al., 2016) Thus, we explore a good learning rate among three values for our optimizer choice, Adam.

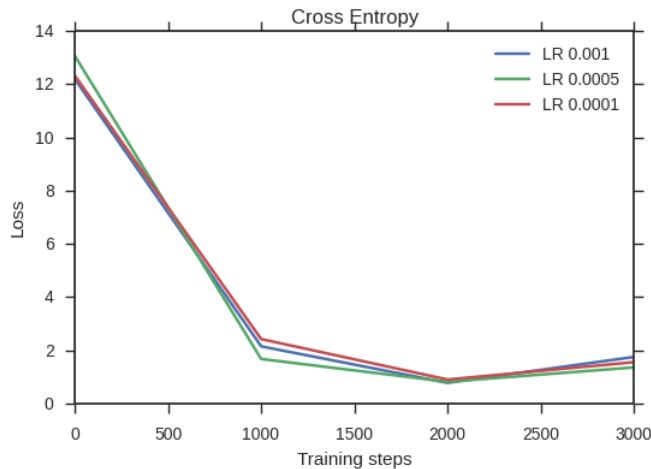


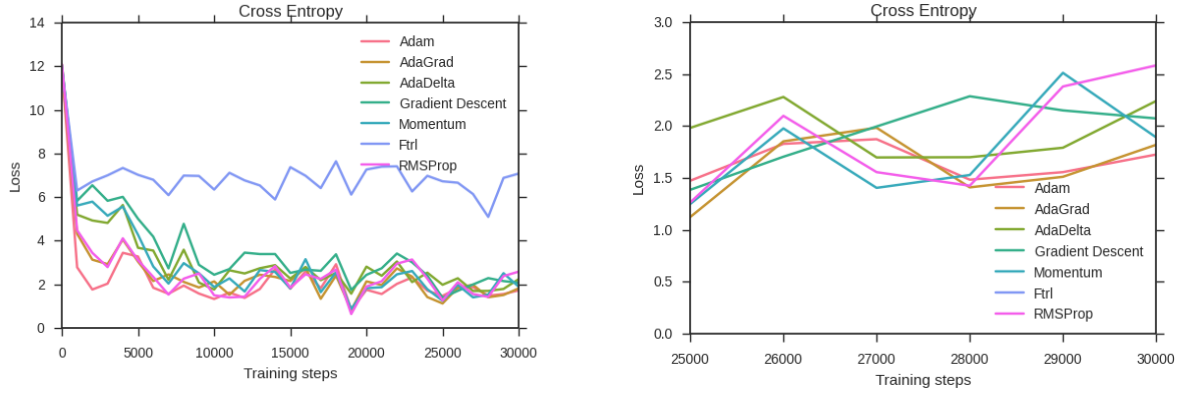
Figure 8: Loss curve at varying learning rates.

Learning Rate	Test Accuracy (%)
LR 0.0001	91.8
LR 0.001	95.0
LR 0.0005	95.4

From the experiments, a learning rate of 0.0005 is the best, as it reduces the loss marginally greater than the rest. For Adam optimizer, initializing learning rates somewhere from 0.001 and below are good choices, any higher will not have good convergence values.

Optimizer

The choice of optimization algorithm is often dependant on the problem domain, thus there will never always be one best algorithm to use in any implementation. Gradient Descent is the most popular optimizer being used for general purposes, due to its simple implementation. Though, it does not guarantee good convergence, and a vigorous tuning is needed for optimal Gradient Descent performance. Later algorithms introduced are often aimed at resolving Gradient Descent's disadvantages, which include, being trapped in local minima of non-convex error functions, and hard-to-choose learning rates. These algorithms, AdaGrad, AdaDelta, Adam, RMSProp, Ftrl and Momentum are explored.



Optimizer	Learning Rate	Test Accuracy (%)
Ftrl	0.01	80.8
RMSProp	0.001	91.9
Gradient Descent	0.005	92.2
AdaDelta	0.9	92.5
AdaGrad	0.05	93.2
Momentum	0.001	93.5
Adam	0.001	93.8

Figure 9: Optimizer performance

The loss curve above shows the behavior of each optimizer minimizing the loss function. The learning rate used varies among the optimizers, but at a good rate so that they are comparable in the time taken to minimize loss. Most of the algorithms are able to reduce the loss to a comparable level, except for the Ftrl algorithm, which performed poorly in this example. The main difference of the models is the convergence time, where Gradient Descent is slower to converge compared to the fastest algorithm, Adam. Comparing the performance of minimizing the loss, Adam, Momentum and AdaGrad achieve the best results, and hence are suitable for this problem.

Regularization

Regularization is done to prevent overfitting issues to the training set, and hence allows the model to better generalize to unseen data. Without any regularization, most deep models will inadvertently overfit to serious degree, and thus some form of regularization is always employed for any deep learning model. We explore the effects of two popular regularization techniques: L2 and Dropout.

L2 Regularization

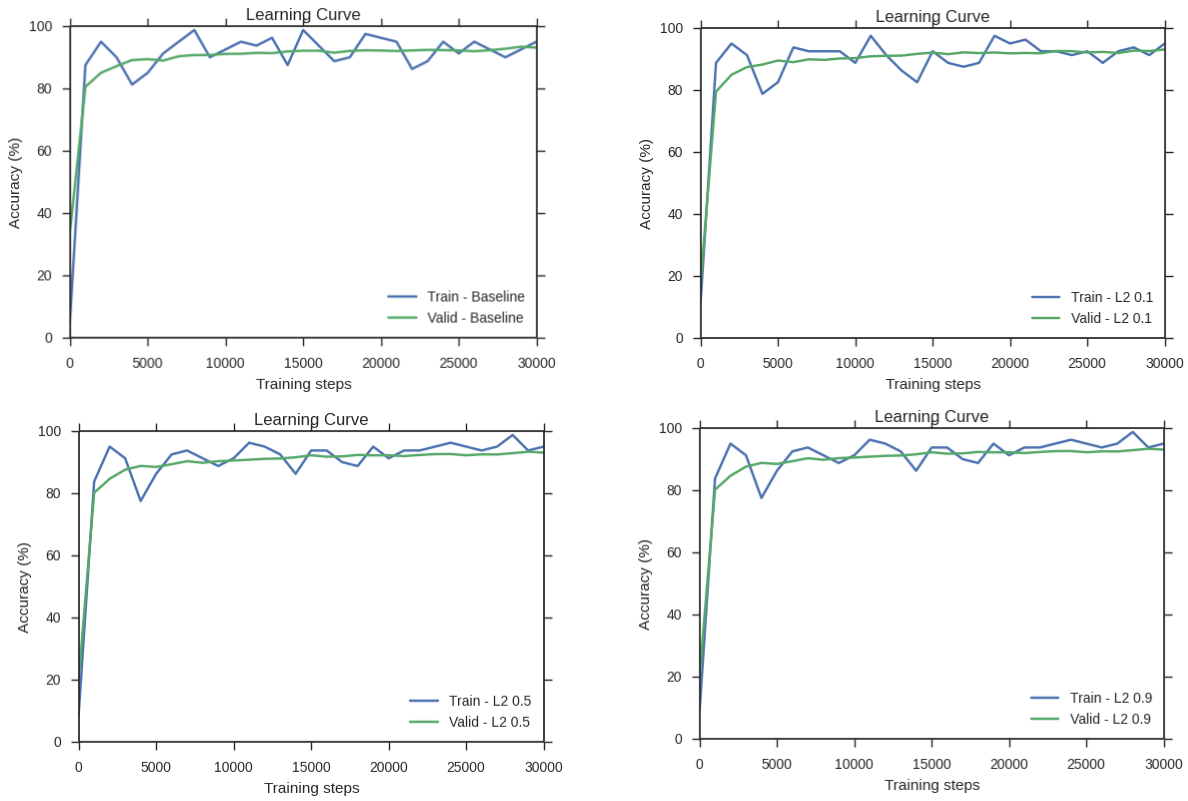


Figure 10: Effects of L2 Regularization strength.

From the figure above, the effects of L2 Regularization can be identified by the gap between the training accuracy and validation accuracy of the learning curve. A larger L2 number indicates a stronger regularization effect, hence less overfitting.

Dropout

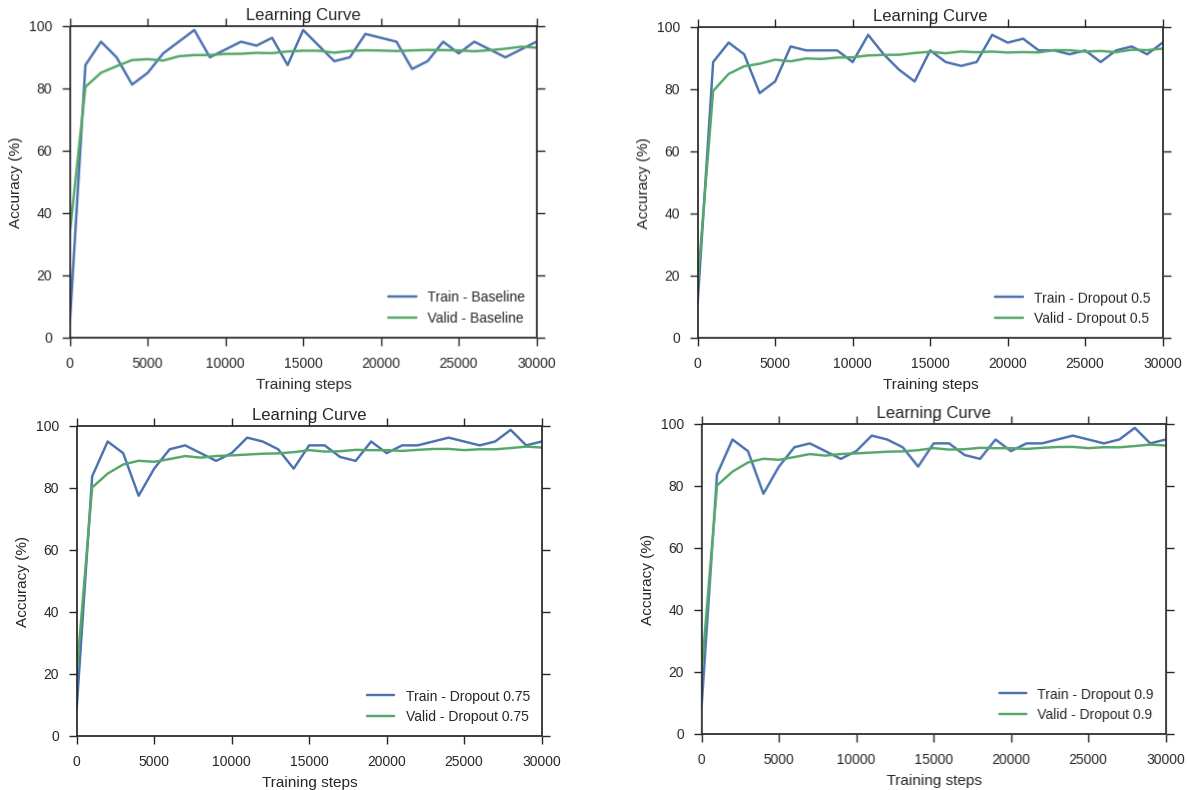
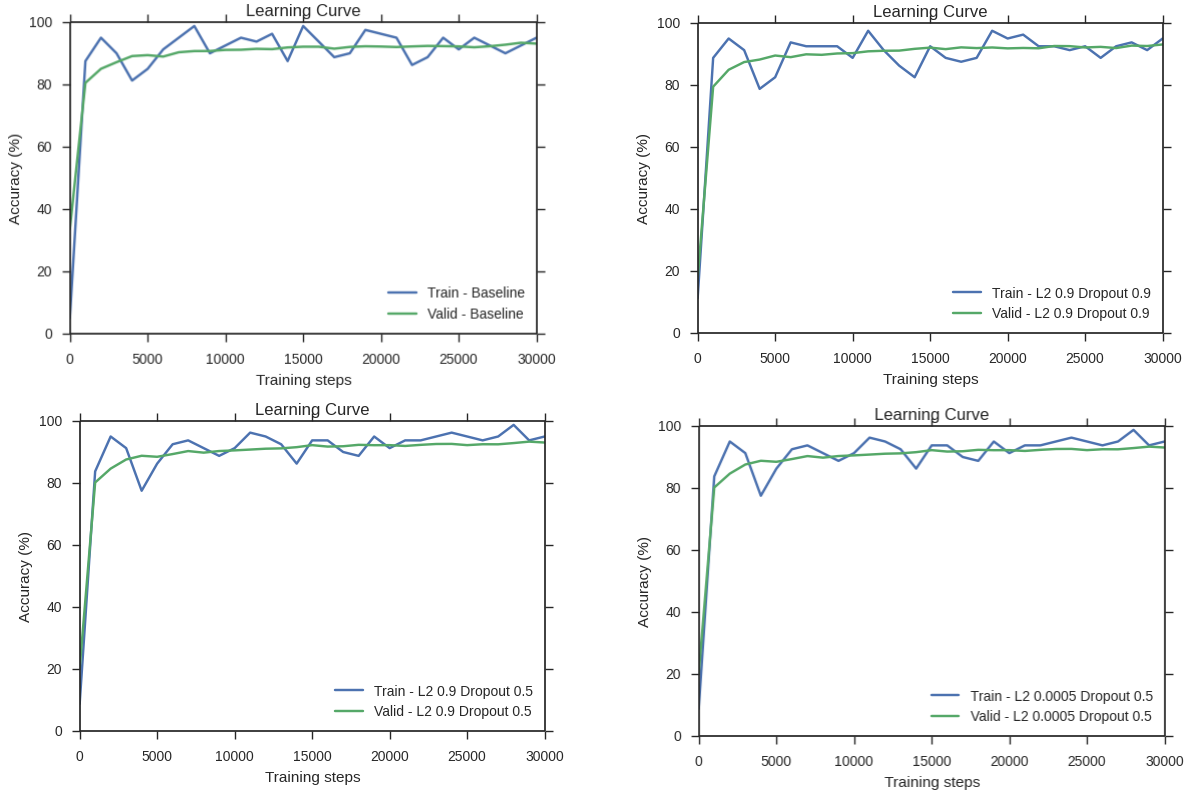


Figure 11: Effects of Dropout

For Dropout, the higher the probability of keeping the nodes, the lesser the effect of regularization. Low probability of keeping nodes means a higher dropout rate, as more nodes are dropped out from the model during training. This reduces the opportunities for nodes to “conspire” with each other to fit to the training set, hence reduces overfitting.

Regularization Refinements

To optimize the regularization, we chose a configuration that allows us the best performance on the test set whilst still maintaining good regularization strength. We combined both L2 and Dropout for the final model.



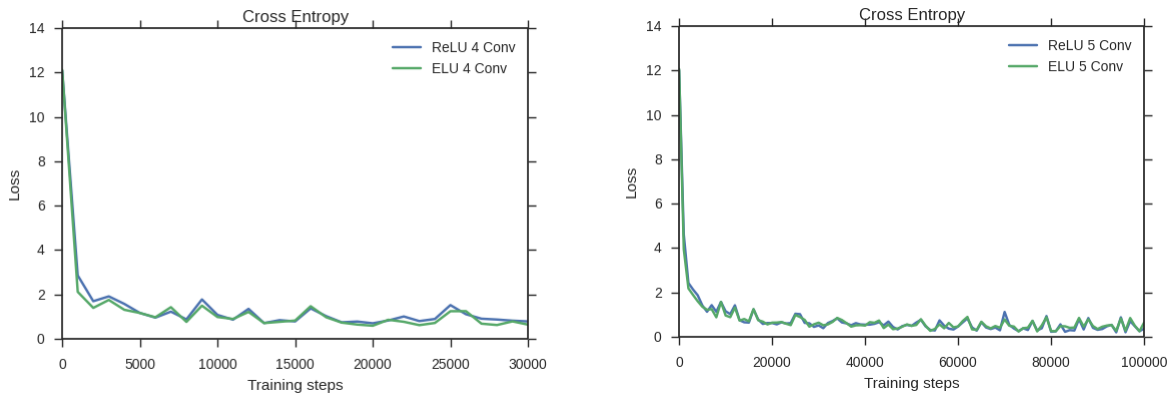
Regularization Configuration	Test Accuracy (%)
Dropout 0.5	93.0
L2 0.9	93.2
Baseline – None	93.5
L2 0.1	93.5
L2 0.5	93.5
Dropout 0.75	93.6
L2 0.9 Dropout 0.9	93.6
L2 0.9 Dropout 0.5	93.8
Dropout 0.9	93.9
L2 0.0005 Dropout 0.5	93.9

Figure 12: Effects of combined L2 and Dropout.

Both the standalone Dropout at 0.9 and L2 0.0005 & Dropout 0.5 combination achieved the highest accuracy scores, however, we chose the latter because it had better overfitting prevention. This configuration also outperformed the baseline model without any regularization by 0.4%. (Srivastava, N., et. al., 2014) has also found that a dropout rate of 0.5 is optimal for most applications, and we chose this recommended parameter value.

Activation Function

Activation functions add non-linearity to a linear deep feedforward network, defining outputs of layers based on input thresholds. Although there exists many activation functions (such as sigmoid, tanh, etc), most of these has gone out of favor. Modern networks mostly employ the use of Rectified Linear Unit (ReLU) due to its superior qualities. We investigate the implementation of ReLU and another fairly new activation function, the Exponential Linear Unit (ELU) introduced by (Clevert, D. A., Unterthiner, T. & Hochreiter, S., 2016)



Activation Function	Test Accuracy (%)
ReLU – 4 Conv	95
ELU – 4 Conv	95.1
ELU – 7 Conv	95.5
ELU – 5 Conv	95.9
ReLU – 5 Conv	96.3

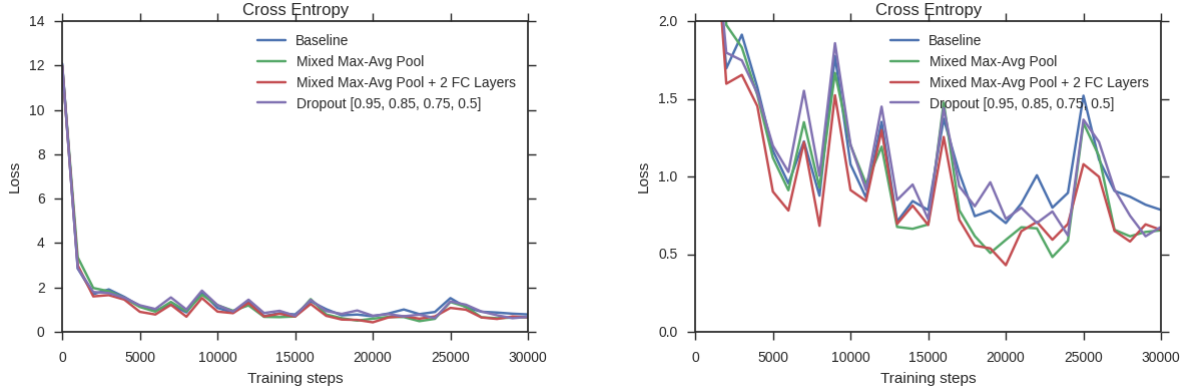
Figure 13: Effects of activation functions.

From the figure above, while ReLU's performance is good, ELU provides some potential, especially as ELU converges faster than ReLU in all cases. However, measuring on the test performance, ELUs don't necessarily produce better results. At 4 convolutional layers, ELU outperformed ReLU by a tiny margin of 0.1%, while for 5 convolutional layers, ReLU clearly outperformed ELU by 0.4%. This is consistent with the findings of (Clevert, D. A., Unterthiner, T. & Hochreiter, S., 2016), where ELUs tend to only outperform ReLUs for network depths of more than 5 layers. ELUs lead not only to faster learning, but significantly better generalization performance than ReLUs on these configurations. Interestingly, for 7 convolutional layers with ELU, it seemed that this performed worse than ELU at 5 convolutional layers. This may be because that when adding more layers in a network, the number of parameters grow greatly, and the computational time required for convergence increases as well. To put this in comparison, we ran the 7-layer ELU for 16 hours with poor results, but it took only 5 hours for the 5 layer ReLU's best results among all configurations. For shallower networks, ReLUs are

an obvious preferred choice. Thus, due to time constraints, we decided to adopt ReLU at 5 layers.

Network Architecture

Here, we investigate the effect of three modifications to the network architecture: network depth, Dropout in all convolutional layers and the Mixed Max-Avg pooling operation.



Architecture	Test Accuracy (%)
Baseline	95.0
Dropout [0.95, 0.85, 0.75, 0.5]	95.3
Mixed Max-Avg Pool	95.5
Mixed Max-Avg Pool + 2 FC Layers	95.8

Figure 14: Performance of various architectures.

From the graphs, all modifications done seemed to outperform the baseline model. In particular, the introduction of a Mixed Max-Avg Pooling proposed by (Lee, C. Y., Gallagher, P. W. & Tu, Z., 2015). In Mixed Max-Avg Pooling, instead of a singular pooling performed after each convolutional layer, both Max and Avg Pooling are used simultaneously, with a mixing factor α . The Mixed Max-Avg Pooling, $f_{mix}(x)$, is expressed as below:

$$f_{mix}(x) = \alpha \cdot f_{max}(x) + (1 - \alpha) \cdot f_{avg}(x)$$

Where $\alpha \in [0, 1]$, represents the proportion of mixing within 0%-100% for both the Max Pooling, $f_{max}(x)$, and Avg Pooling, $f_{avg}(x)$. We chose $\alpha = 0.5$ for the mixing factor as it has shown to produce good results for pooling at each convolutional layer suggested by Lee et. al.

Mixed Max-Avg Pooling has been shown to outperform the baseline, with just a simple modification to the pooling layer. As for network depth, increasing the number of layers increases the capacity and thus the representational power of the model, as evident that the network with two additional fully connected layers outperformed the one without. We explored the option of including Dropout in all convolutional layers, each with increasing dropout rate of [0.95, 0.85, 0.75, 0.5],

similar to AlexNet. We observed that performance increased slightly from the baseline, regularization seemed to be comparable to the L2 & Dropout method, and thus for simplicity, we adopted only one Dropout operation. A multi Dropout method would probably suit a deeper network with much more parameters.

The final model architecture after refinements has been included as below:

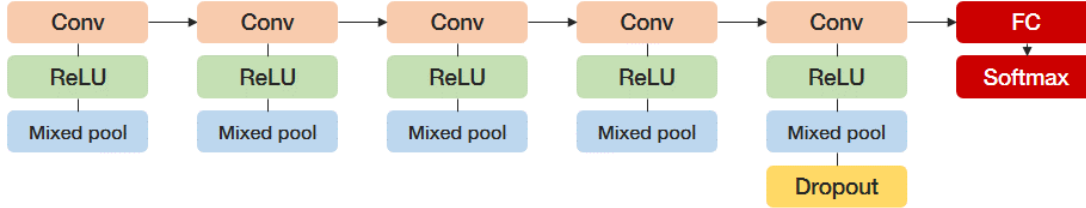


Figure 15: Final model architecture.

The model has a total of 6 layers (5 convolutional and 1 fully connected), where all convolutional layers are subjected to ReLU activation function with Mixed Max-Avg Pooling. Dropout is only applied to the last convolution layer. For regularization, L2 factor of 0.0005 and Dropout rate of 0.5 is used. The Adam optimizer with learning rate 0.0005 with decay is used.

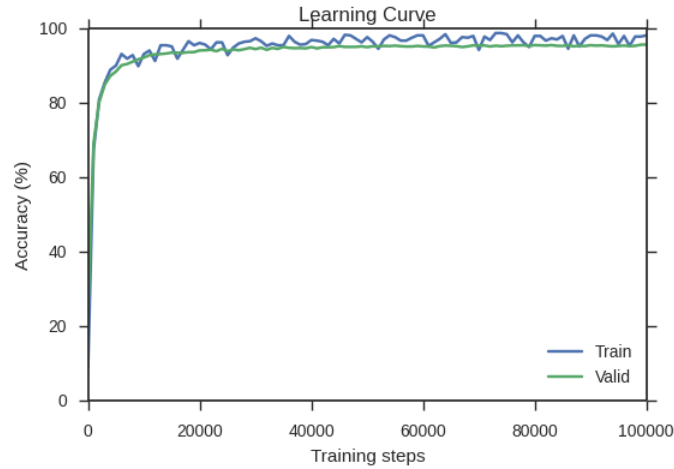
4. Results

Model Evaluation and Validation

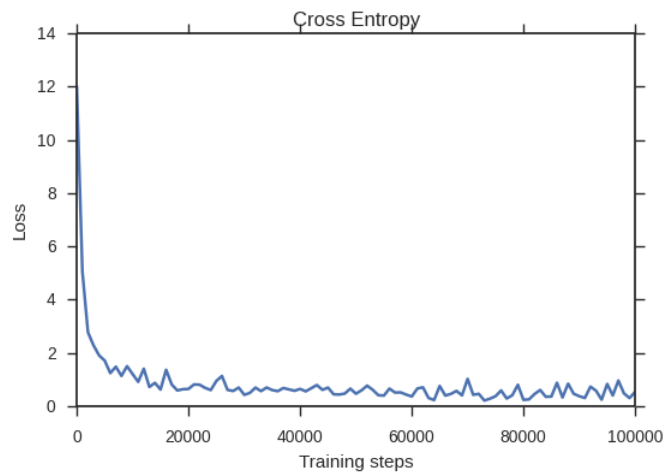
The final model as described earlier was chosen after empirically evaluating various design modifications and tuning hyperparameters, with considerations of heuristics from established research and publications.

After training the model for about 5 hours and about 55 epochs (number of iterations over the entire training set), with 100,000 training iteration steps of batch size of 128, we achieved a test set accuracy of 96.3% (3.7% error), which met our expectations of this model against the benchmark we had set (96% accuracy).

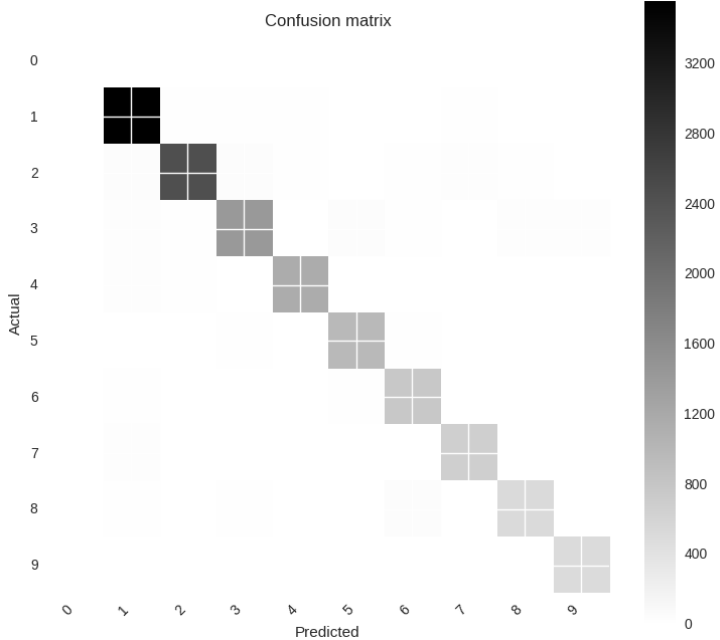
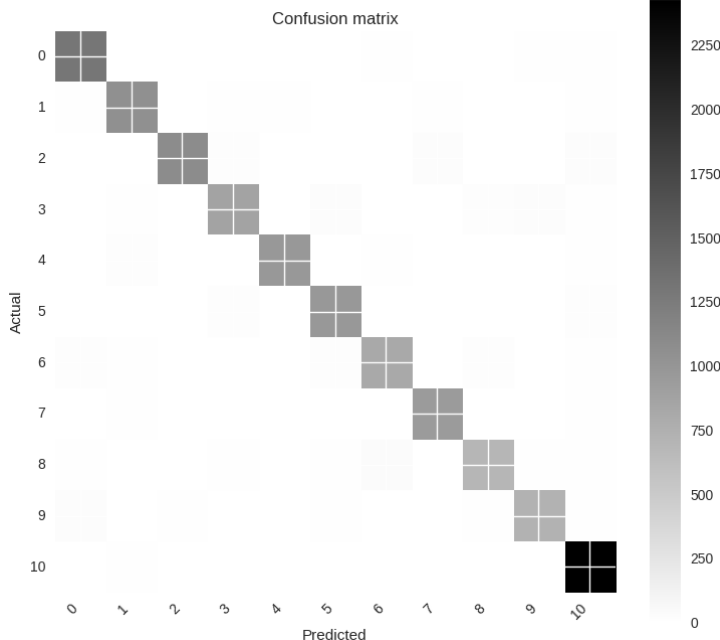
The following learning curve shows a quick rise in accuracy scores for both the training and validation set, before reaching a “plateau” phase, while gradually increasing incrementally.

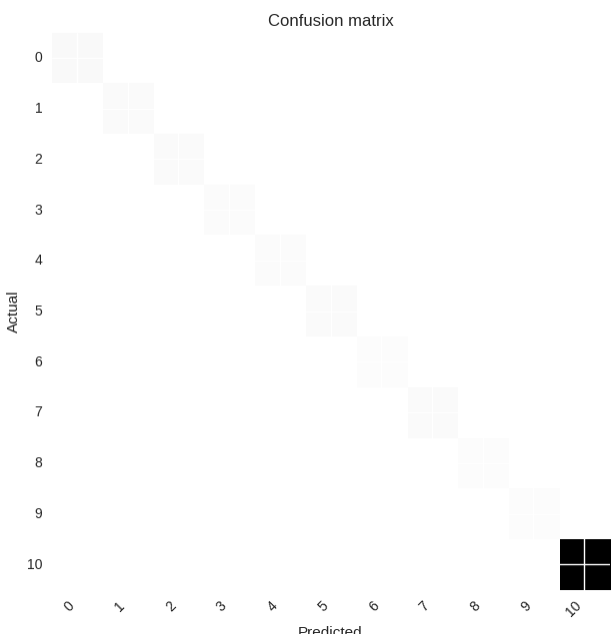
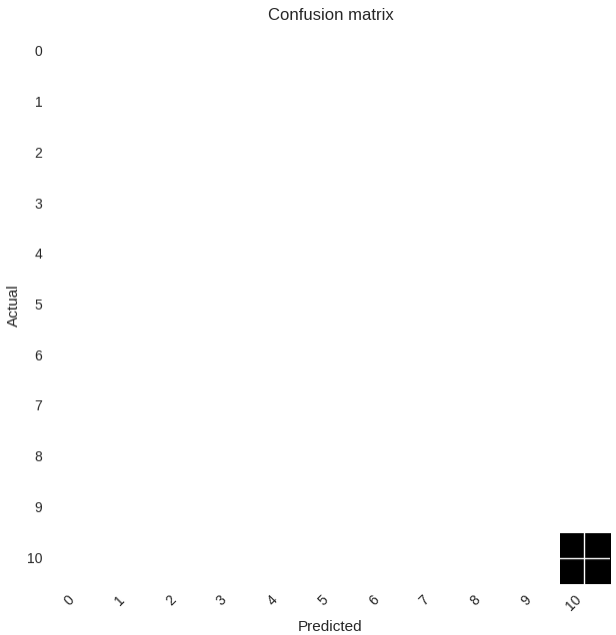


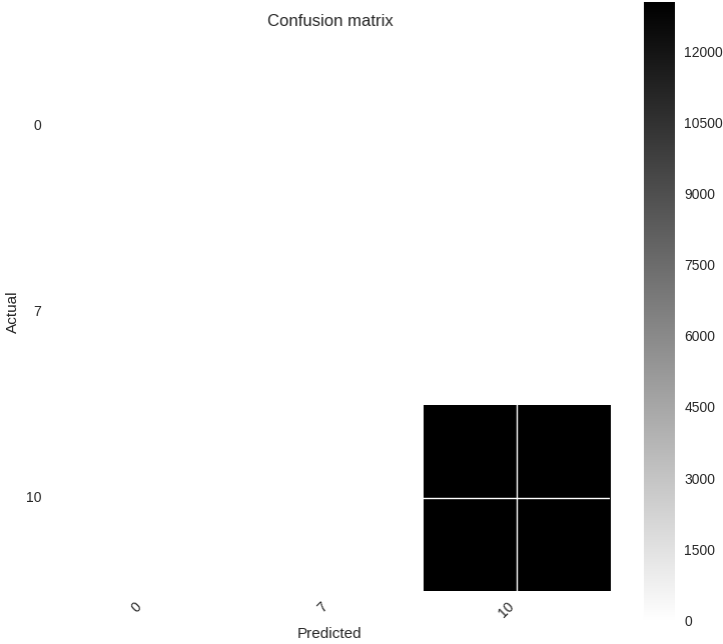
While the cross entropy loss curve follows an exponential curve, reducing drastically for the first 1000 training iterations (upon completing the first epoch) then gradually decreasing in value over time.




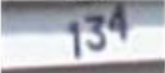


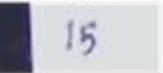
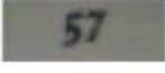






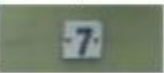
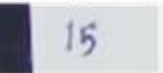
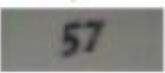




Confusion matrix visualizations is as below. Five matrices are produced over each digit sequence slot (N1, ... N5), and then precision, recall, and F1-scores are calculated by averaging over the results of these five matrices. Overall, the model scored an average precision = 0.9599, recall = 0.9501 and F1-Score = 0.9549.

#	Confusion Matrix	Precision	Recall	F1-Score
1	<p>Confusion matrix</p>  <p>Actual</p> <p>Predicted</p>	0.9233	0.9233	0.9233
2	<p>Confusion matrix</p>  <p>Actual</p> <p>Predicted</p>	0.9145	0.8655	0.8893

3	<p>Confusion matrix</p>  <p>Actual</p> <p>Predicted</p>	0.9655	0.9655	0.9655
4	<p>Confusion matrix</p>  <p>Actual</p> <p>Predicted</p>	0.9965	0.9965	0.9965

5	<p>Confusion matrix</p> 	0.9998	0.9998	0.9998
Average Scores		0.9599	0.9501	0.9549

Visualization of a sample of test data prediction is as follows. The model has classified nine sequences correctly out of ten. The misclassified sequence is '132' where the model has mistaken the '2' digit as '1'.

Actual									
24	134	11	7	15	57	104	132	210	44
									
Predicted									
24	134	11	7	15	57	104	131	210	44
									

Sensitivity Analysis

The model is re-run with a subset of the training dataset to investigate if the model is robust to changes in the input data. We used 90%, 80% and 70% subsets of the dataset.

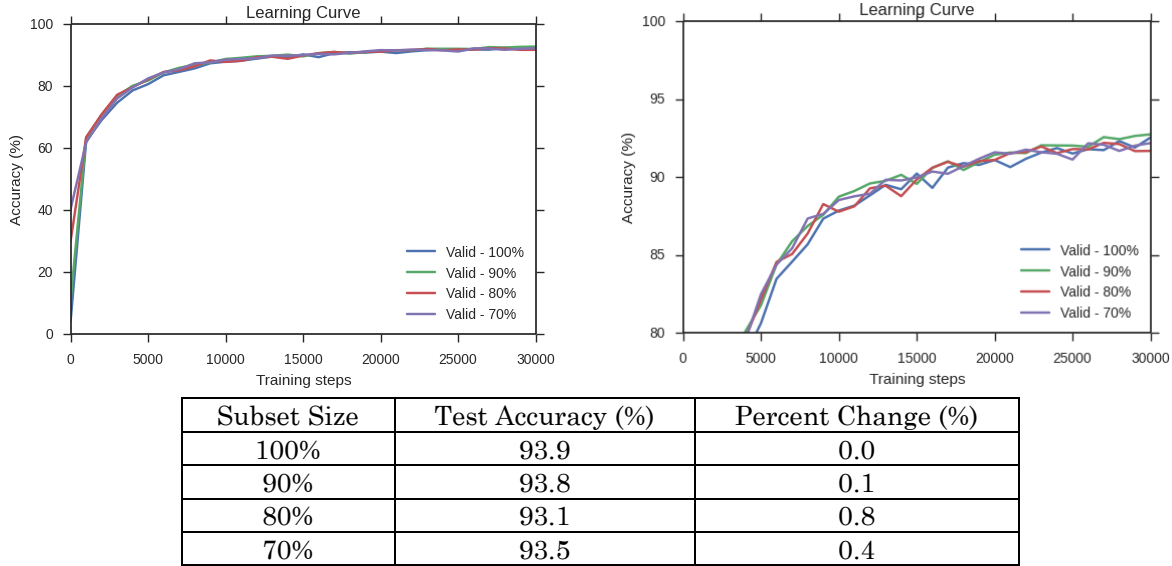


Figure 16: Sensitivity analysis with varying training subsets.

From the results above, the model loses 0.1% accuracy whenever there is a 10% reduction of input data. Up to 30% reduction, the model shows a maximum perturbation of 0.8%. Losing 0.8% can be significant if accuracy tolerance is critical. However, for perturbations within 10% of the training dataset, the model is robust enough.

Justification

Based on the model's performance, this is reasonable as it meets the expectations set earlier. Given the resource and time constraints, as the training time we set for this model is shorter than most publications, whilst has shown good performance. A comparison of the performance of our model against several authors is as below:

Model Architecture	Accuracy (%)	
	Digit	Sequence
Stoch. Pooling (Zeiler, M. D. & Fergus, R., 2013)	97.20	-
Maxout Networks (Goodfellow, I. et. al., 2013)	97.53	-
FitNet (Romero, A. et. al., 2015)	97.58	-
Prob. Maxout (Springenberg, J. & Riedmiller, M., 2014)	97.61	-
NiN	97.65	-

(Lin, M., Chen, Q. & Yan, S., 2013)		
Maxout-11 layers (Goodfellow, I., et. al., 2014)	97.84	96.03
DropConnect (Li, W. et. al., 2013)	98.06	-
DSN (Lee, C. Y. et. al., 2015)	98.08	-
R-CNN (Liang, M. & Hu, X., 2015)	98.23	-
Tree+Max-Avg Pool (Lee, C. Y., Gallagher, P. W. & Tu, Z., 2015)	98.31	-
CNN-6 layers, Max-Avg Pool (Ours)	-	96.30

Table 2: Comparison of published performance on SVHN. Source adapted from Lee et. al. (2015). Note: We included error scores for digit and sequence as our problem closely relates to sequence recognition similar to Goodfellow et. al. (2014), while most academia publications are benchmarked on digit error scores.

For sequence-level accuracy, our model is able to improve slightly on the multi-digit recognition work by (Goodfellow, I., et. al., 2014) (96.3% vs 96.03% accuracy), with a shallower network and less parameters. This performance is attainable may be because of our simplified preprocessing steps of converting to grayscale and applying GCN, and especially restricting our input data dimensions to only 32x32 pixels. For comparison, (Goodfellow, I., et. al., 2014) used 54x54 sized inputs with data augmentation. We simplified our model to work with maximum digit sequences of 5, and utilized efficient and fast-training additions, such as the ReLU and Mixed Max-Avg Pooling. The decision to use 5 convolutional layers with ReLU was to allow fast convergence of shallower networks due to computational power and time constraints. Although the benefits of ELU could not be disregarded, as ELUs can help with the problem of diminishing gradients frequently encountered with training deep networks and improve training times (Clevert, D. A., Unterthiner, T. & Hochreiter, S., 2016), we needed to compromise, and ReLUs used within 5 layers or less seemed sufficient. Given our model has modest capacity, we are still able to slightly improve the performance of our benchmark, and this is considered a good result. However, given more computational resources and increased model capacity, the performance of the resulting model would have shown more potential to perform at human-level or better.

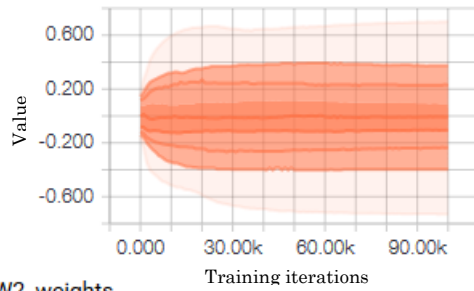
5. Conclusion

Free-Form Visualization

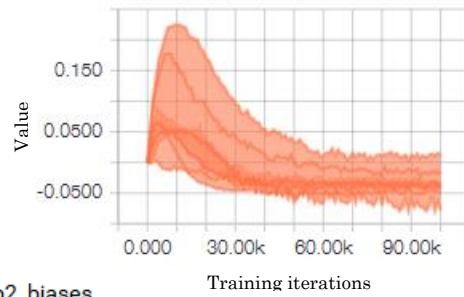
TensorBoard, a visualization suite of TensorFlow, allows us to visualize certain characteristics of the model when learning. Below is the visualization of the weights and biases for all convolutional layers:

Distribution of weights and biases over time. x-axis represents the number of training iterations, y-axis represents the weight/bias value. Darker bands of the graphs indicate that the particular weight/bias value was more frequently in use than others. Most of the weight and bias values are close to 0.

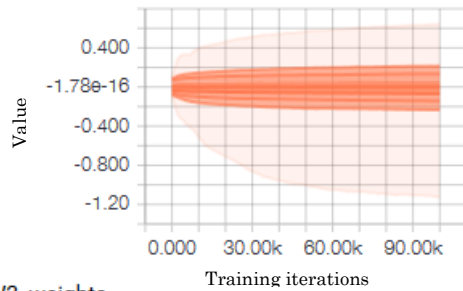
W1_weights



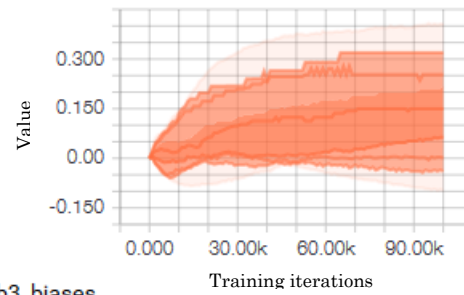
b1_biases



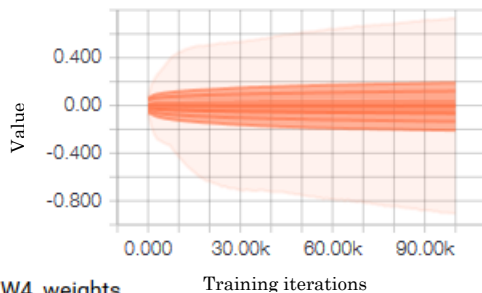
W2_weights



b2_biases



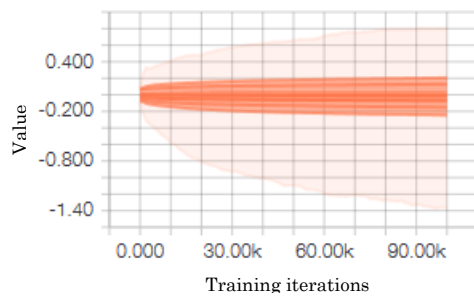
W3_weights



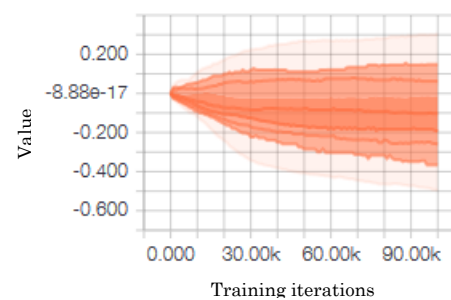
b3_biases



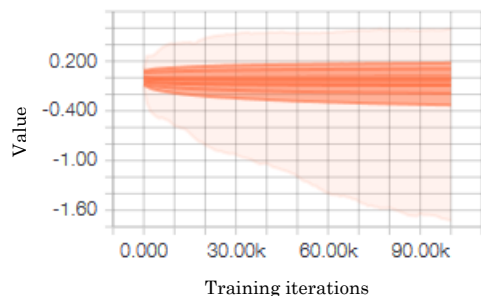
W4_weights



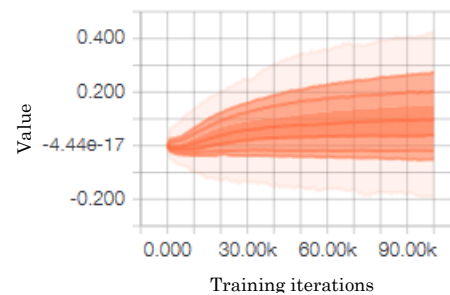
b4_biases



W5_weights



b5_biases



Mean change of weights and biases over time.

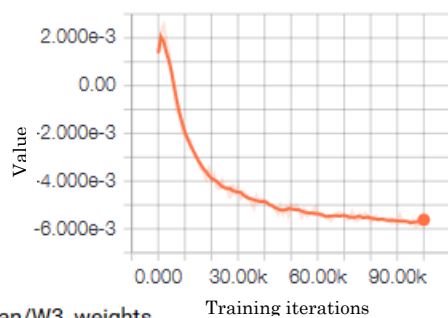
mean/W1_weights



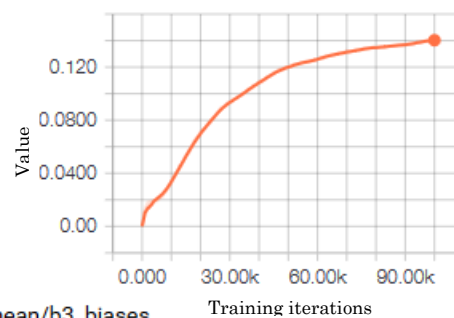
mean/b1_biases



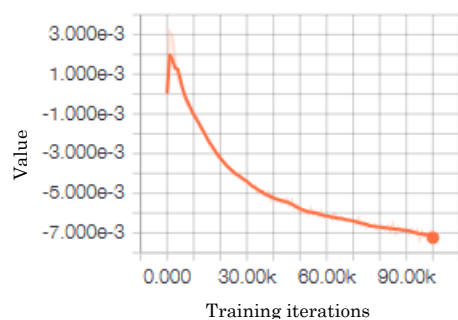
mean/W2_weights



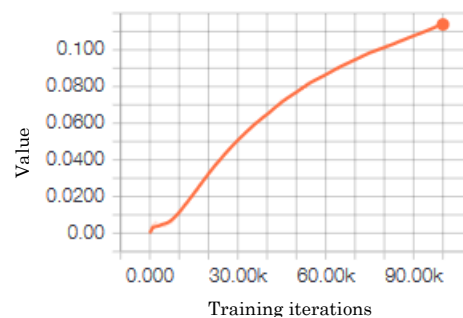
mean/b2_biases

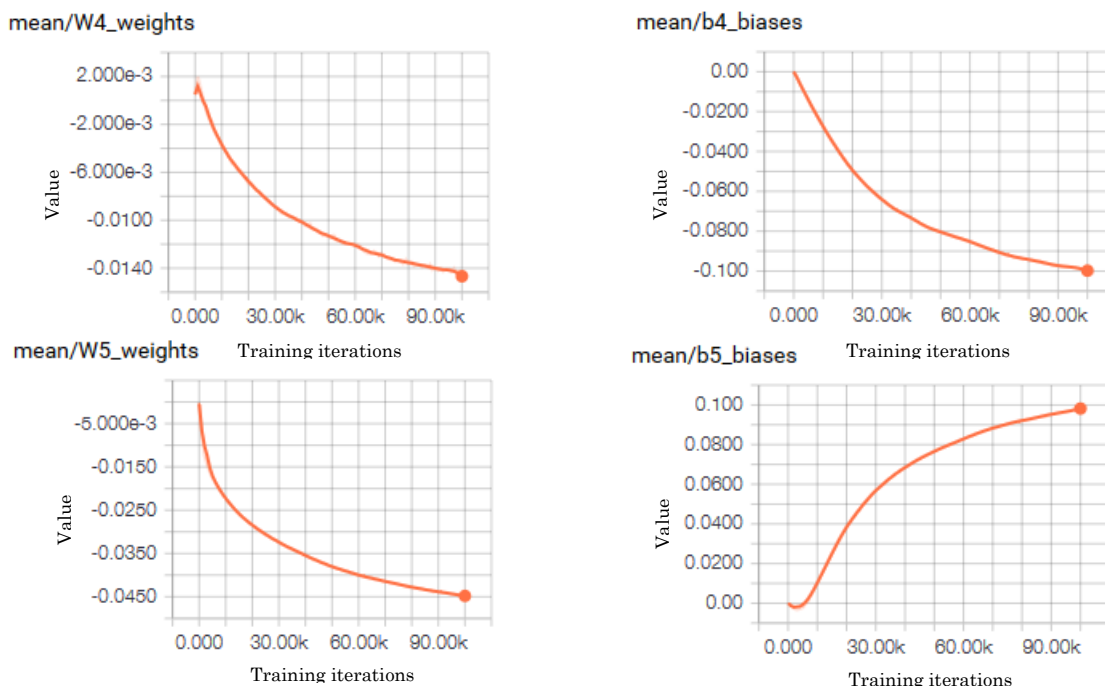


mean/W3_weights



mean/b3_biases





Reflection

Overall, our strategy was to design a fast and efficient deep network incorporating easy-to-implement performance-boosting operations without sacrificing too much representational power for our problem. During the preprocessing step, we implemented simple solutions with GCN and simplified further by converting input images to grayscale to further reduce the computational and memory load of our model. This mantra is carried onto our model architecture design, where we established a simple 4-layer CNN to run quick experiments to tweak our design toward incrementally improving the accuracy scores, while also exploring the various effects of hyperparameter changes toward our model performance. We also researched various implementation ideas in publication, and found that there is no one-size-fits-all kind of solution, but there are general heuristics and progress in the latest research which proved invaluable for our model design. Our final model contained the parameters we found best throughout the experiments coupled with heuristics, and once the base performance has been established, we scaled it up to be trained for 5 hours. Difficult aspects of the project involved figuring out data manipulation in the preprocessing step, as working with image files are unconventional in comparison to structured dataframes. Also, troubleshooting setbacks encountered was a difficult aspect, especially when we thought it the root cause of the problem was in the model, but actually, is caused by a mistake in the preprocessing step. This mistake caused us about two weeks in delay of our project progress. Also, implementing a 5-way softmax for multi digit recognition was one challenge, due to our lack of understanding of how deep neural nets work.

Programmatically was a challenge as well, especially preprocessing implementation and TensorFlow implementation of the model, as we are new to the library and workflow. Optimizing the many model parameters was initially poorly done due to trial-and-error, but progressively gotten better as we applied heuristics. In summary, our model is sufficient for input data similar to SVHN, but for new unseen types of digits, this model may not be suitable if accuracy is critical. It is best to retrain with these new input data for better generalization.

Improvement

In terms of improving the model, there are several parts in which the limitation stems from us being unable to implement due to programmatic challenges and further understanding of the underlying theory. For example, we used Mixed Max-Avg Pooling, which has shown to boost performance with a simple idea of mixing pooling operations. The authors of the work also explored better solutions building on from this, which include a tree-structure mixed pooling, that outperformed vanilla mixed pooling. If we knew how to implement this, then we would expect the performance of our model to increase. There are other concepts that can be explored, such as Inception modules introduced by GoogLeNet (Szegedy, C. et. al., 2015) and Recurrent CNN (R-CNN) (Liang, M. & Hu, X., 2015), however, these are far more time-consuming, especially the latter as it involves a total model architecture modification. Also, hyperparameter tuning has been done manually in this project, and there exists automatic methods which greatly help in tuning them, with the expense of having to tune the automatic method's own hyperparameters. Another interesting improvement to implement is visualizing each convolutional layer's outputs as pioneered by (Zeiler, D. & Fergus, R., 2014) to understand what salient features each convolutional layer is picking up. We would like to explore ELUs too for deeper CNN architecture. As it is today, there definitely exists better solutions for our problem and state-of-the-art performance of published models, even surpassing human-level performance. And if time and expertise permits, the implementation of a live camera, augmented reality app can be explored, where the app reads a digit sequence from a captured image and overlays the output of what the model thinks it is.

Bibliography

- Benenson, R. (2016, February). *What is the class of this image ?* Retrieved from Classification datasets results:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
- Bengio, Y. (2009). *Learning Deep Architectures for AI*. Now Publishers Inc. Retrieved from
[http://sanghvi.com/download/soft/machine%20learning,%20artificial%20intelligence,%20mathematics%20ebooks/ML/learning%20deep%20architectures%20for%20AI%20\(2009\).pdf](http://sanghvi.com/download/soft/machine%20learning,%20artificial%20intelligence,%20mathematics%20ebooks/ML/learning%20deep%20architectures%20for%20AI%20(2009).pdf)
- Clevert, D. A., Unterthiner, T. & Hochreiter, S. (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv:1511.07289v5 [cs.LG]*. Retrieved from <https://arxiv.org/pdf/1511.07289v5.pdf>
- Cohen, N. (2015). Adam: A Method for Stochastic Optimization. *Advanced Seminar in Deep Learning (#67679)*. Retrieved from Advanced :
https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod_resource/content/1/adam_pres.pdf
- Cross entropy*. (n.d.). Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_error_function_and_logistic_regression
- Cross-entropy cost function in neural network*. (2015). Retrieved from Stack Exchange: <http://stats.stackexchange.com/questions/167787/cross-entropy-cost-function-in-neural-network>
- Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *JMLR*. Retrieved from
<http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>
- Goodfellow, I et. al. (2016). *Deep Learning*. MIT Press (In preparation). Retrieved from <http://www.deeplearningbook.org>
- Goodfellow, I. et. al. (2013). Maxout Networks. *ICML*.
- Goodfellow, I., et. al. (2014). Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. *International Conference on Learning Representations (ICLR)*. Retrieved from
<http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf>

- He, K. et. al. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852v1 [cs.CV]*. Retrieved from <https://arxiv.org/pdf/1502.01852v1.pdf>
- Kingma, D. & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980v8 [cs.LG]*. Retrieved from <https://arxiv.org/pdf/1412.6980v8.pdf>
- Krizhevsky, A. et. al. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25 (NIPS)*. Retrieved from <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- Krizhevsky, A., et. al. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25 (NIPS 2012)*. Retrieved from <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- LeCun, Y. et. al. (1998). Efficient BackProp. In G. & Orr, *Neural Networks: tricks of the trade*. Springer. Retrieved from <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>
- Lee, C. Y. et. al. (2015). Deeply Supervised Nets. *AISTATS*.
- Lee, C. Y., Gallagher, P. W. & Tu, Z. (2015). Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree. *arXiv:1509.08985v2 [stat.ML]*. Retrieved from <https://arxiv.org/pdf/1509.08985v2.pdf>
- Li, W. et. al. (2013). Regularization of NNs Using DropConnect. *ICML*.
- Liang, M. & Hu, X. (2015). Recurrent CNNs for Object Recognition. *CVPR*.
- Lin, M., Chen, Q. & Yan, S. (2013). Network in Network. *ICLR*.
- Netzer, Y., et. al. (2011). Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. Retrieved from http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf
- Romero, A. et. al. (2015). FitNets: Hints for Thin Deep Nets. *ICLR*.
- Ruder, S. (2016, January 19). *An overview of gradient descent optimization algorithms*. Retrieved from [sebastianruder.com](http://sebastianruder.com/optimizing-gradient-descent/): <http://sebastianruder.com/optimizing-gradient-descent/>
- Sermanet, P. Chintala, S. & LeCun Y. (2012). Convolutional Neural Networks Applied to House Numbers Digit Classification. *arXiv:1204.3968v1 [cs.CV]*. Retrieved from <https://arxiv.org/pdf/1204.3968v1>

- Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556v6 [cs.CV]*. Retrieved from <https://arxiv.org/pdf/1409.1556v6.pdf>
- Springenberg, J. & Riedmiller, M. (2014). Improving Deep Neural Networks with Probabilistic Maxout Units. *ICLR*.
- Srivastava, N., et. al. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research 15 (2014) 1929-1958*. Retrieved from <http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>
- Szegedy, C. et. al. (2015). Going Deeper with Convolutions. *IEEE*. Retrieved from http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf
- Szegedy, C. et. al. (2015). Rethinking the Inception Architecture for Computer Vision. *arXiv:1512.00567v3 [cs.CV]*. Retrieved from <https://arxiv.org/pdf/1512.00567v3.pdf>
- Szegedy, C. & Ioffe, S. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167*. Retrieved from <https://arxiv.org/pdf/1502.03167v3.pdf>
- The Street View House Numbers (SVHN) Dataset*. (n.d.). Retrieved from <http://ufldl.stanford.edu/housenumbers/>
- Wei, X. S. (2015, October 19). *Must Know Tips/Tricks in Deep Neural Networks*. Retrieved from lamda.nju.edu.cn: <http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html>
- Yang, X. & Pu, J. (2015). *MDig: Multi-digit Recognition using Convolutional Neural Network on Mobile*. Stanford University, CS231m Computer Vision. Retrieved from <http://web.stanford.edu/class/cs231m/projects/final-report-yang-pu.pdf>
- Zeiler, D. & Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. *ECCV*. Retrieved from <http://www.cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>
- Zeiler, M. D. & Fergus, R. (2013). Stochastic Pooling for Regularization of Deep Convolutional Networks. *arXiv:1301.3557*.