Digit Classification using Convolutional Neural Networks

Andrew Toulis (aptoulis@uwaterloo.ca) and Baha Nurlybayev (bnurlyba@uwaterloo.ca)
Department of Management Sciences, University of Waterloo

Abstract—**In this paper, state-of-the-art architectures and techniques for training convolutional neural networks are explored on a relatively challenging domain. The task at hand is to classify digits from Google Street View images. Experimenting with different architectures, the objective of this study is to highlight the impact of each change made on classification performance. A list of best practices for initialization, normalization and regularization is compiled, validating and summarizing prior literature. Following these guidelines, 97% accuracy was obtained compared to 76% in a baseline model. As complex architectures in neural networks become increasingly popular, reviews of this nature will become critical for clarifying and validating theoretical and experimental research.**

I. Introduction

For over a decade, image recognition algorithms have attained above 99% accuracy on MNIST, a dataset of handwritten digits [17]. Successes have sparked research interest in applying computer vision techniques to real-world imagery [3]. While tasks based on real-world data are generally challenging, the dataset of choice in this paper is well-studied, with a state-of-the-art accuracy of 98% [5]. The best performing models are highly complex. By compiling and utilizing several simple techniques from literature, an accuracy of 97% was obtained, demonstrating their effectiveness.

The goal of this research is to highlight key components of high-performing convolutional neural networks (CNNs) on a sufficiently difficult domain. In addition to achieving strong final results, the contribution of this paper is to summarize how specific changes in architecture impacted performance. This is in effort to validate and consolidate existing literature.

Specifically, the dataset under study is called Street View House Numbers (SVHN) [21]. While the dataset is similar to MNIST, it is much larger and contains significantly more variability in contrast, orientation and style of digits. The ground-truth for each image is included, enabling supervised learning algorithms to be applied.

The remainder of the report is structured as follows:

1) State-of-the-art CNNs in the image classification domain are reviewed in Section II;
2) The dataset and pre-processing steps applied are described in Section III;
3) The methodology is outlined in Section IV;
4) In Section V, each model configuration is explained and performance is tabulated;
5) Finally, the advantages and disadvantages of different configurations are highlighted in the Section VI.

II. Literature Review

A. Classification using Convolutional Neural Networks

CNNs are only a subset of successful approaches in computer vision. For example, a Support Vector Machine approach from Decoste et al achieved 99.5% accuracy on MNIST in 2002 [4]. However, both the popularity and complexity of CNNs has increased, calling for a clear review of best practices for training them.

Essential components found in state-of-the-art models are reviewed in detail in this section. To summarize, the most common neural network layers used in practice are convolutional, batch normalization, and fully connected.

1) Convolutional Layer: The convolutional layer is the distinguishing layer of a CNN. Its primary function is to replace the sophisticated procedure of extracting features from input images. A particularly nice property of convolutional layers is that they can be stacked to obtain more abstract representations of images [12], a task which becomes increasingly difficult in a semi-manual pipeline.

The convolutional layer has been given significant theoretical attention. It has been predicted and experimentally verified that random weights in a convolutional layer can extract meaningful features such as edges [13]. As will be emphasized in the Results section, initialization of random weights is critical for training a CNN, with a popular approach called Xavier initialization [14]. Recent theoretical work and experimentation has pointed towards orthogonal (unitary) initialization of weights, which can reduce the vanishing/exploding gradient problem due to the length-preserving nature of orthogonal transformations [15].

The derived features are fed into a classifier that can be trained separately or in an end-to-end model. In an unsupervised setting, the goal is often to reconstruct the input, often with some regularization conditions. A popular unsupervised CNN is the convolutional autoencoder [11]. In the end-to-end supervised setting, a final classification layer is used to predict the true labels. The error is computed after the feedforward stage by comparing the predicted outputs to the true outputs. Since the operations in a convolutional layer are differentiable, backpropagation is commonly used for credit assignment. As a result, the features in the convolutional layers can be learned procedurally, rather than through careful hand-crafting.

A convolutional layer consists of several filters. Each filter is applied on a small spatial area of an image (typically 5x5 or 3x3 pixels [10]), where the size of the region is known as the receptive field of the filter. The operation performed is discrete convolution, which is used in image processing to extract specific features from images such as edges. Filters are always applied across the whole depth of the layer's input (for example, all three colour channels in the input). The same filter is applied across all patches in an input layer, which is known as weight sharing or locally connectivity. This significantly reduces the number of parameters that must be learned, and enables shared learning from several pixels of an image simultaneously.

An arbitrary number of filters can be learned per layer, although for computational reasons this is usually limited between 32 and 256, where powers of two are often used for computational optimization. The number of features is sometimes called the depth of the layer or the number of output channels. Knowing this, the total weights in each convolutional layer can be stored as a tensor of dimension: [input channels x receptive field height x receptive field width x output channels].

How each filter is applied is an important aspect of the configuration of a convolutional layer. Stride dictates how much the filter slides from one subregion of the image to another. A stride of two will reduce the size of the output, but is often too destructive [10]. Hence, most results presented will use a stride of one for convolution.

Furthermore, zero-padding allows for control of the output size by padding the outside of the input subregions with zeros. Often, this is done to keep the length of the input the same as the outputs, which is known as "same" padding. Alternatively, a "valid" padding will not add any padding, and hence subregions of the image that are too close to the ends will not be convolved on.

It is computationally efficient to apply this large matrix to a batch of input images, rather than a single one. Hence the output of a convolutional layer is a tensor of dimension: [batch size x output spatial height x output spatial width x output channels].

2) Batch Normalization: Batch normalization is employed to tackle the shifting covariates problem [7], which is the issue where the distribution of each hidden layer changes over time during training. Since neural activation functions assume the data is centered and distributed in a particular manner, shifting distributions can be a significant issue with training neural networks. Batch normalization combats this issue and is thus known for improving stability of convergence and enabling larger learning rates.

During training, batch statistics are calculated in order to shift the current output distribution of a convolutional layer to zero mean and unit variance. Larger batches are better since they tend to improve the calculated mean and variance. By applying normalization at the batch level, the layer can be implemented efficiently. Two learnable parameters are then used to rescale and shift the distribution to an appropriate place for the following activation function or next layer. Note that the literature originally intended for batch normalization to be applied before activations, although gains have been reportedafter activations an activation.

At test time, an exponential moving average of the batch means and variances is used to estimate an effective population statistic for normalization.

3) Activation Function: Rectified Linear Unit: Rectified linear unit (ReLU) activation functions are typically applied after the convolution and batch normalization layers as a simple truncation of negative pre-activations. They have nice properties during training, including a simple derivative and a tendency to easily become activated, significantly speeding up training time [2]. State-of-the-art has moved towards using ReLU instead of sigmoidal or other activations, except in the final layer [16].

4) Pooling: A common layer which reduces the spatial size of the representations between the convolutional layers is pooling. The most common approach is a 2x2 max pooling with a stride of two [10]. This operates under the assumption that in every square of size 2x2, the most important piece of information is the maximum activation, thereby removing 3/4 of the input that the following layer would need to process. This also mitigates phenomena such as each side of an edge being detected in nearby pixels (whereas average pooling could accidentally average each side of an edge). Larger pooling sizes are typically considered too destructive for learning [10], although a 3x3 pool with an overlapping stride of two is also used in practice.

While pooling can greatly reduce redundancy, performance can be significantly degraded by pooling, and research effort is attempting to remove this layer [10]. In this paper, due to computational limitations, it was not possible to thoroughly test the effect of not pooling on various architectures. Significant gains have been reported by others on the same dataset [19].

5) Fully Connected Layer: Fully connected layers appear after the last convolutional layer of the CNN architecture and before the classifier layer. The purpose of fully connected layers is to combine the activations of the final convolutional layer into a set of typical neurons with activations. The layer often consists of anywhere from 256 to 4096 neurons, where again powers of two are used for computational reasons. The outputs of a fully connected layer assist the classification layer in utilizing the features derived by the convolutional layers.

In contrast to the the convolutional layer, instead of local connectivity, every neuron of the fully connected layer is connected to every input in the previous layer. In the final fully connected layer, known as the output or classifier layer, typically a softmax (sigmoidal, logistic) function is used to convert pre-activations to a probability. These final activations are used for prediction, calculating errors, and credit assignment in backpropogation.

State-of-the-art convolutional networks typically have more than one hidden fully connected layer before the final output layer [1]. Increasing the number of fully connected layers can significantly increase the number of parameters and can also lead to over-fitting.

6) Regularization: Deep neural networks are extremely powerful models which are thus prone to over-fitting data. One simple way to avoid this is to have more training data, which is demonstrated in this study. On the other hand, it is not always possible or computationally feasible to train on very large datasets. Hence, regularization is an important component in most models in order to avoid this problem.

There are two forms of regularization commonly applied in neural networks. The first one to be discussed is dropout [20]. The simple idea is to restrict neurons from activating (and hence receiving gradients) at a fixed random rate in a given neural layer. This is done during the training process to discourage neurons from becoming too dependent. Theoretically, this training process is thought to be equivalent to training an exponential number of slightly weaker networks. At test time, an approach is to simply multiply each neuron's activation by 0.50, while others simply do not use dropout during testing.

Weight decay is equivalent to typical regularization of weights in classifiers in other settings. Commonly, an L2 penalty is applied to punish large squared values of individual weights. Also popular, L1 penalties punish absolute values of weights, which is often used to enforce sparsity in a model. This is since near-zero elements do not square to a very small number and hence are penalized persistently.

In addition, the batch normalization layer has a regularization effect, sometimes eliminating the need for these other types of regularization.

B. Extensions

Goodfellow et al. approached an even more difficult task of sequence of digits classification [1]. They demonstrate that a single CNN can recognize sequences of digits. Prior state-of-the-art used a pipeline of models that localize, segment and then finally detect digits in images. The specific architecture they used had eight convolutional hidden layers, which is too deep to reproduce. There is a locally connected hidden layer and two densely connected hidden layers at the end, where adding another densely connected layer performed much better than a single one. Each convolutional layer includes max pooling and subtractive normalization (see Krizhevsky et al [2] for a review of these techniques in practice). Batch normalization has replaced subtractive normalization in practice, although these procedures have similar purposes and effects. The team trained with dropout applied to all hidden layers except the first one.

Two notable convolutional architectures not covered include Competitive Multi-scale Convolution [18] and Residual Connections [16]. There are also generalized convolutions and pooling functions that can learn different features [5]. These extended models achieve better performance, but are not in the scope of this study due to computational complexity.

III. Data

The dataset consists of 600k (604,388) training examples and 26k (26,032) testing examples. Images have been cropped at a fixed resolution of 32x32, and the digit of interest are already centered in each image.

Originally, 531k (531,131) of the training examples were not included since they are less difficult [21], although this significantly limited performance on the testing set. In particular, models could not exceed 67% accuracy without this extra training data. The labels of the data range from '0'' to '9'. Class imbalance was present in the official dataset, with the digit '1'' being most common. The literature that was reviewed did not treat this and hence no changes were made to the distribution of classes.

Since this is a dataset of real house numbers, there are several intra-class variations in font, colour, texture, background and orientation. This calls for a sophisticated computer vision approach that is robust to such variations. When given the right data, CNNs are capable of becoming invariant to many of these properties. To speed up processing and to reduce the burden on the model, simple and commonly performed pre-processing was performed. For completeness, models were trained on the raw input data and performance was very low (sub 50% accuracy).

Prior to normalization, the input was grayscaled based on a mapping derived by Grundland et al [8], which enhances contrast. Following a similar pre-processing approach from Sermanet et al [6], per image global contrast normalization was performed next, which is simply Z-score normalization in the terminology of other domains. The idea is that different images are taken in different levels of lighting, and hence the mean pixel value is not informative. Furthermore, normalizing the per image standard deviation to 1.0 helps with convergence due to assumptions made in the Xavier initialization technique. Figure 1 demonstrates the effect of this pre-processing on two sample inputs. These examples also highlight the difficulty of the dataset due to the presence of neighbouring digits, colour variation, distortions, and approximate cropping.

The chaining of these processing steps follows the work of Hang Yao on this same dataset, and the code used was written by him [9]. Hang also applied a vanilla Support Vector Machine to the problem to demonstrate its difficulty [9]. In particular, on the processed data, 22% accuracy was obtained using a vanilla SVM, showing this is a fairly difficult task.

IV. Methodology

Along the lines of prior work [1, 2, 3], the general approach taken in this paper is to train a feedforward neural network using backpropogation. Given an input image, the network is trained to correctly predict the ground-truth output label. All activations used were ReLUs, except the output layer which used a softmax activation. To take advantage of parallel computation, the feedforward stage is performed on batches of 100 input samples. This also aids with stable training, as discussed in the review of batch normalization. During training, after every 500 batches, a validation sample of 4000 images is fed through the network to visualize accuracy and loss metrics in real-time. This was the maximum sample size that the server's GPU could fit into memory without exhausting resources. Across several batches, the entire test

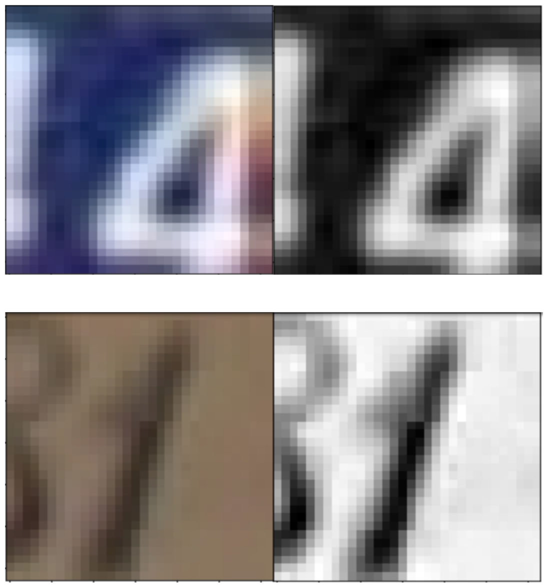


Fig. 1. Pre-processing applied to two sample images ('4' on the top and '1' on the bottom). On the left is the original data. On right, the pre-processing steps (grayscale conversion and global contrast normalization) are applied.

set of 26k was fed through the network after training finished to report final performance metrics.

The objective function is to minimize the cross-entropy between predicted and true outputs, as commonly done. When used, weight decay also plays a part in the objective function, where a multiplied parameter is used to control its relative importance. The input is a vector of length 1024, which is reshaped into a 32x32 matrix to be convolved on. Between the last convolutional layer and first fully connected layer is a reshaping operation.

A. Experimental Setup

The machine learning framework used is Tensorflow [23]. The most recent release was used, TensorFlow 1.0. The neural network layers discussed in the literature review are implemented in TensorFlow. To aid with visualizing performance, TensorBoard was used [24]. Code can be found in the appendix and is available online [26].

AWS EC2 servers were used for training the models and tuning parameters through repeated experimentation. P2.xlarge instance was chosen in particular, as it provided high performance while being relatively affordable. The instance contained a Tesla K80 GPU with 12GB memory and costed $0.9 per hour. This was extremely important as it enabled more than a 10-fold performance boost compared to a CUDA enabled GPU in Macbook Pro, and allowed the team to experiment rapidly with larger batches and more complex model architectures. Amazon's Deep Learning AMI's (Amazon Machine Image) made infrastructure setup simple and ready to train.

B. Training Parameters

Non-architectural parameters that were used across all experiments, unless otherwise specified, are outlined in this section. As standard, stochastic gradient descent was used for training models. A learning rate of 0.001 was used, since higher learning rates would not converge due to oscillations. The optimizer used is Adam [22], which estimates exponentially-weighted first and second moments of the gradient. It then performs a normalized gradient descent update, which is thus an adaptive learning rate. This update uses the exponential average of gradients rather than a point estimate, giving it an effective momentum.

For fine tuning, a learning rate of 0.002 was used, with a smooth exponential decay that effectively halves after completing each epoch (full dataset of 600k seen images). This was done to speed up earlier training while keep a low learning rate in later epochs.

Furthermore, due to an abundance of data, dropout was not performed for most of the experimentation. However, for the final model a dropout probability of 25% was used. Xavier initialization of weights was used [14]. Without this initialization models performed poorly. For biases, a bias of zero was simply used due to lack of effect on performance. Larger initial biases diminished performance significantly.

V. Results

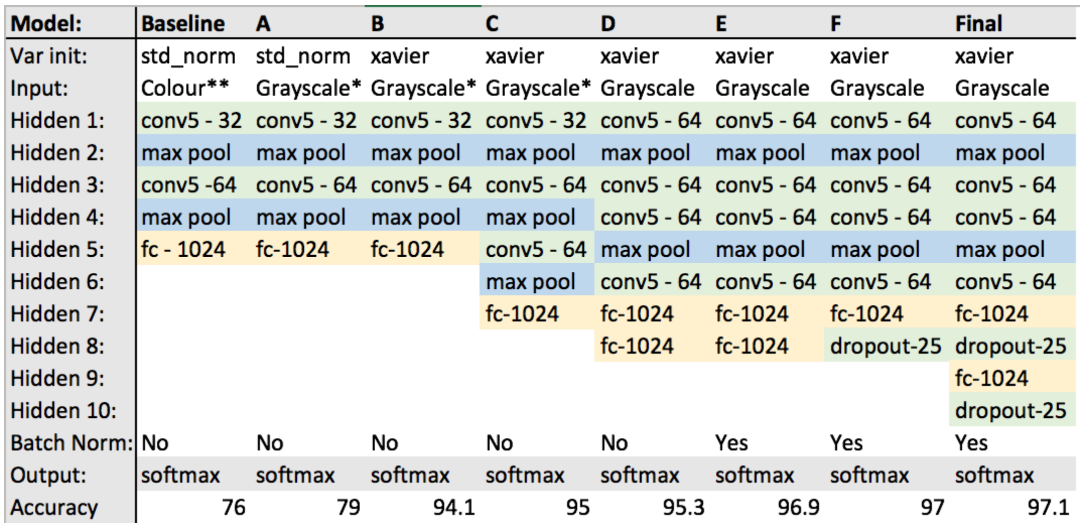
The final model achieved 97.1% accuracy, which is 1.2% below the current state-of-the-art for this dataset. It is significantly simpler and contains several core features that will be highlighted. The steps made to achieve this performance are described in detail in this section.

A. Benchmark Model

The benchmark network architecture is inspired from the Tensorflow MNIST tutorial [25]. The model included two convolution layers, with 32 and 64 filters respectively. Both convolutional layers had a receptive field of 5x5 and were followed by 2x2 max pooling. Batch normalization was not used. A fully connected layer with 1024 neurons followed the last convolutional layer, and was connected to the 10-class output layer. The initial benchmark was performed on the original 3-channel RGB images. Critical pre-processing steps were taken to normalize pixel values between zero and one, and to remove the per-image mean. This benchmark achieved a testing accuracy of 76%.

B. Final Model

The final model included four convolution layers, each with 64 filters and 5x5 receptive field. 2x2 max pooling was applied on 1st and 3rd convolution layers. Further pooling would significantly decrease performance, since after two pooling operations the output size was 8x8. Two fully connected layers were stacked after the last convolution layer, each trained with a dropout of 25%. A TensorBoard computational graph of the final model can be seen in Figure 4, which highlights the feedforward nature of the model. The final model was trained and tested on pre-processed images (normalized per image, with a single grayscale channel), as outlined in Section III. Figure 3 displays the accuracy and loss of the model over time during training.



Model:	Baseline	A	B	C	D	E	F	Final
Var init:	std_norm	std_norm	xavier	xavier	xavier	xavier	xavier	xavier
Input:	Colour**	Grayscale*	Grayscale*	Grayscale*	Grayscale	Grayscale	Grayscale	Grayscale
Hidden 1:	conv5 - 32	conv5 - 32	conv5 - 32	conv5 - 32	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64
Hidden 2:	max pool	max pool	max pool	max pool	max pool	max pool	max pool	max pool
Hidden 3:	conv5 -64	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64
Hidden 4:	max pool	max pool	max pool	max pool	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64
Hidden 5:	fc - 1024	fc-1024	fc-1024	conv5 - 64	max pool	max pool	max pool	max pool
Hidden 6:				max pool	conv5 - 64	conv5 - 64	conv5 - 64	conv5 - 64
Hidden 7:				fc-1024	fc-1024	fc-1024	fc-1024	fc-1024
Hidden 8:					fc-1024	fc-1024	dropout-25	dropout-25
Hidden 9:								fc-1024
Hidden 10:								dropout-25
Batch Norm:	No	No	No	No	No	Yes	Yes	Yes
Output:	softmax	softmax	softmax	softmax	softmax	softmax	softmax	softmax
Accuracy	76	79	94.1	95	95.3	96.9	97	97.1

Fig. 2. Summary of the various architectures tested and their performance. "Final Model" had the highest performance of 97.1% accuracy. "Colour" refers to the initial 3-channel dataset; "Grayscale" refers to the pre-processed data. The first two models on left were initialized with normally distributed values (standard deviation = 1), and the rest of the models were initialized using Xavier initialization.

C. Configurations Tested

Table 2 summarizes the architectures tested. The neural layers are described in detail in the literature review (Section II). The most significant configurations that made improvements are discussed below.

*1) **Weight Initialization**:* Using Xavier weight initialization, whereby following layers effectively receive inputs with approximately unit standard deviation, had the most significant impact on performance. Up to 15% improvement in accuracy was observed, with much faster convergence.

*2) **Batch Normalization**:* After improving initialization, batch normalization improved performance by an additional 1.6%. In addition, it improved the rate and stability of convergence.

*3) **Increased Depth**:* The final adjustment was to add extra layers of convolution. Two extra layers caused a modest improvement in accuracy of 1.2%. Notice that when extending convolutional depth, max pooling cannot be applied on every layer. In particular, the best configuration had alternating pooling, with the first layer always pooling to reduce computational complexity. An extra fully connected layer had statistically insignificant improvements in accuracy (0.1% gain), despite literature indicating performance boosts could be attained [1].

*4) **Regularization**:* Employing dropout of 25% along with batch normalization helped to reduce oscillation during training and improved accuracy by a further 0.2% compared to the same configuration with no dropout. Since the dataset was very large, regularization did not have a significant impact but would likely be a required component on smaller datasets.

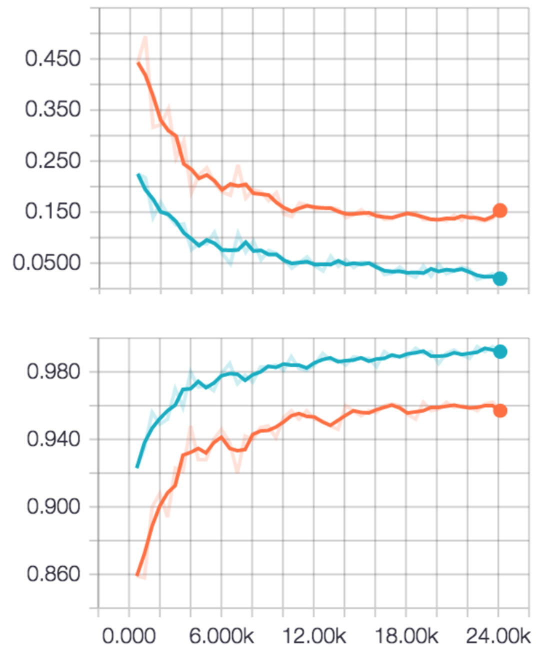


Fig. 3. Test (orange) and train (teal) cross-entropy loss (top) and accuracy (bottom) of the final model performed over four epochs. The x-axis is the number of steps (a batch of 100 examples).

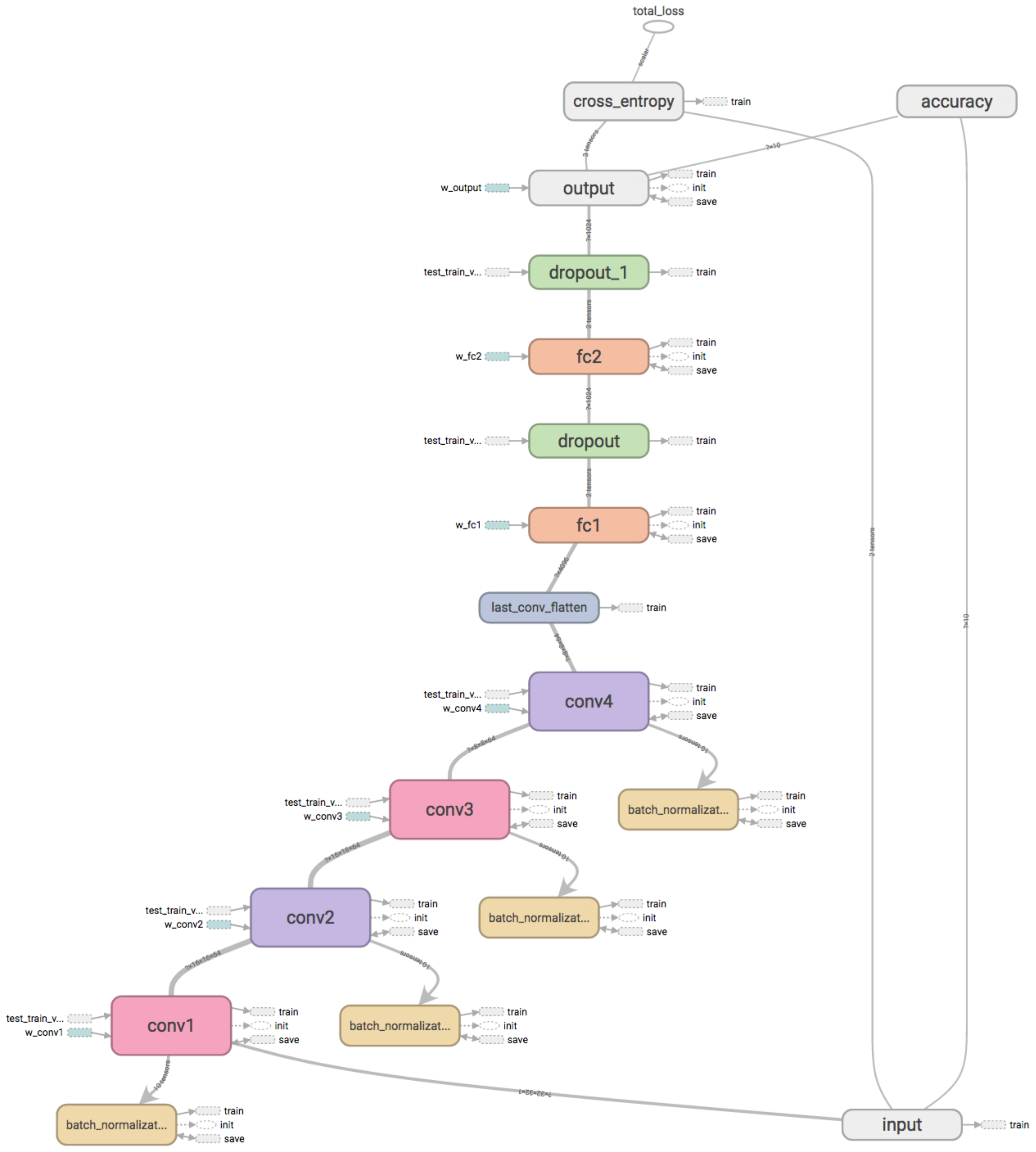


Fig. 4. The computation graph of the final CNN, rendered by TensorBoard. The model is read from bottom to top: input, convolutional layers (pooling is coloured pink), fully connected layers, and finally the output layer which is fed into the loss and accuracy.

VI. Discussion

Reported performance differences validate the literature reviewed and stress the need for careful model training. In particular, below is a summary of best practices for future researchers training neural networks:

1) **Apply pre-processing**: At a minimum, subtract the image mean. Models could not converge if this step was missing.
2) **Initialization is critical**: Consider Xavier initialization [14] and ensure biases do not overwhelm the activation function. This was critical for convergence.
3) **Normalize each layer**: To avoid shifts in the distributions of each layer during training, it is essential to employ normalization. Batch Normalization is the modern approach to this and is highly efficient.
4) **Pool infrequently**: Pooling should only be applied a maximum number of times while extending depth of the convolutional neural network.

There are limitations with this study. While the results were highly useful for this dataset, future studies would need to perform these benchmarking experiments on several different domains to fully validate the findings. Furthermore, not every combination of the tested configurations was performed. For example, dropping out only on one fully connected layer was not tested. Furthermore, batch normalization was applied either globally or not at all for each convolutional layer. Data-level issues include class imbalance and a few cases that were mislabeled or non-digits.

Future work could explore whether batch normalization should be applied before or after activation and pooling operations. Furthermore, due to computational limitations, lack of pooling could not be tested and likely would improve performance. Finally, the effect of weight decay was not significant in this case, but it is believed that a special initialization could be used to appropriately utilize this regularizer.

VII. Appendix: Code

```
import cPickle as pickle
import gzip
import numpy as np
import os
from datetime import datetime
from time import time
import tensorflow as tf

with
    gzip.open("data/svhn_grayscale_gcn.pkl",
    "rb") as f:
  svhn_gray = pickle.load(f)

svhn_train_data =
    np.load("data/svhn_grayscale_gcn_train.np.npy")
svhn_train_labels = svhn_gray["train_labels"]
svhn_test_data = svhn_gray["test_data"]
svhn_test_labels = svhn_gray["test_labels"]

print svhn_train_data.shape,
    svhn_train_labels.shape
print svhn_test_data.shape,
    svhn_test_labels.shape


def reformat(data):
  data = data.reshape((-1, 32, 32,
      1))#.astype(np.float32)
  return data

svhn_train_data = reformat(svhn_train_data)
svhn_test_data = reformat(svhn_test_data)
print svhn_train_data.shape,
    svhn_train_labels.shape
print svhn_test_data.shape,
    svhn_test_labels.shape


### Model Wrappers ###
def weight_variable(shape, name):
  weight_name = "w_" + name
  W = tf.get_variable(weight_name,
      shape=shape,
      initializer=tf.contrib.layers.xavier_initializer())
  return W

def bias_variable(shape, default_bias=0.0):
  initial = tf.constant(default_bias,
      shape=shape)
  return tf.Variable(initial)

def conv_layer(layer_name, input_tensor,
    receptive_field, channels_in,
    channels_out,
           padding='SAME', stride=1,
               act=tf.nn.relu, decay=0.0,
           pool=True, pooler=tf.nn.max_pool,
               pool_size=2, pool_stride=2,
               pool_padding='SAME',
           batch_norm=False, training=True,
               default_bias=0.0):

  with tf.name_scope(layer_name):
   with tf.name_scope('weights'):
     weights =
         weight_variable([receptive_field,
         receptive_field, channels_in,
         channels_out], layer_name)

     if decay > 0:
        weight_decay =
            tf.multiply(tf.nn.l2_loss(weights),
            decay, name='weight_decay')
        tf.add_to_collection('losses',
            weight_decay)

   with tf.name_scope('biases'):
     biases = bias_variable([channels_out],
         default_bias)

   with tf.name_scope('W_conv_x_plus_b'):
     preactivate = tf.nn.conv2d(input_tensor,
                weights, strides=[1, stride,
                    stride, 1],
                    padding=padding) + biases

   if batch_norm:
     with tf.name_scope('batchnorm'):
       normed =
           tf.layers.batch_normalization(preactivate,
           training=training)
     activations = act(normed,
         name='activation')
   else:
     activations = act(preactivate,
         name='activation')

   if pool:
     max_pool = pooler(activations, ksize=[1,
         pool_size, pool_size, 1],
                strides=[1, pool_stride,
                    pool_stride, 1],
                padding=pool_padding)
     return max_pool
   else:
     return activations


def dense_layer(layer_name, input_tensor,
    input_dim, output_dim, act=tf.nn.relu,
    decay=0.0, default_bias=0.0):
  with tf.name_scope(layer_name):
   with tf.name_scope('weights'):
     weights = weight_variable([input_dim,
         output_dim], layer_name)

     if decay > 0:
       weight_decay =
           tf.multiply(tf.nn.l2_loss(weights),
           decay, name='weight_decay')
       tf.add_to_collection('losses',
           weight_decay)

   with tf.name_scope('biases'):
     biases = bias_variable([output_dim],
         default_bias)
   with tf.name_scope('Wx_plus_b'):
     preactivate = tf.matmul(input_tensor,
         weights) + biases
   activations = act(preactivate,
       name='activation')
   return activations
```

```
def flat_dimension(tensor):
  dim = 1 # Compute how many numbers we have,
      ignoring the batch size
  for d in tensor.get_shape()[1:].as_list():
    dim *= d
  return dim

# Normalize by subtracting per image, per
     channel means
def normalize_batch(batch):
  per_img_ch_means = batch.mean(axis=1)
  return batch - per_img_ch_means[:,
      np.newaxis, :]

# hacky shuffle for hacky next batch
shuffled_indices = []
for epoch in xrange(10):
  idx = np.arange(len(svhn_train_data))
  np.random.shuffle(idx)
  shuffled_indices.append(idx)
shuffled_indices

# hacky next_batch
def grab_next_train_batch(batch_num, data,
     labels, batch_size):
  assert len(labels) == len(data)
  total_images = len(data)
  assert batch_size <= total_images
  assert batch_num <= total_batches

  start = (batch_num * batch_size) %
      total_images
  epoch = int((batch_num * batch_size) /
      total_images)
  current_idx = shuffled_indices[epoch]
  end = start + batch_size

  next_batch_idx = current_idx[start:end]
  next_batch = data[next_batch_idx],
      labels[next_batch_idx]
  return next_batch

def grab_next_test_batch(data, labels,
     batch_size):
  idx = np.arange(len(data))
  np.random.shuffle(idx)
  idx = idx[:batch_size]

  next_batch = data[start:end],
      labels[start:end]
  return next_batch


def run():
  # RESET TF GRAPH, just in case
  tf.reset_default_graph()

  ### Place holders ###

  with tf.name_scope('test_train_variables'):
    batch_norm_train_mode =
        tf.placeholder(tf.bool) # for
        batch_norm mode
    tf.add_to_collection('batch_norm_train_mode',
        batch_norm_train_mode)
    keep_prob = tf.placeholder(tf.float32) #
        for drop out
    tf.add_to_collection('keep_prob',
        keep_prob)

    # Optionally track that place holders are
        correctly set at test and train tme
    tf.summary.scalar('batch_norm_train_mode',
        tf.to_int32(batch_norm_train_mode,
        name='ToInt32'))
    tf.summary.scalar('dropout_keep_probability',
        keep_prob)


  with tf.name_scope('input'):
    x = tf.placeholder(tf.float32,
        shape=[None, 32, 32, 1],
        name="x-input")
    y_ = tf.placeholder(tf.float32,
        shape=[None, 10], name="y-input")
    tf.add_to_collection('x', x)
    tf.add_to_collection('y_', y_)

  ###################
  ##### Network #####
  ###################

  conv1 = conv_layer(layer_name='conv1',
      input_tensor=x, receptive_field=5,
                  channels_in=1,
                      channels_out=64,
                      pool=True, pool_size=2,
                      pool_stride=2,
                  batch_norm=True,
                      training=batch_norm_train_mode,
                      default_bias=0.0)

  conv2 = conv_layer(layer_name='conv2',
      input_tensor=conv1, receptive_field=5,
                channels_in=64,
                    channels_out=64,
                    pool=False, pool_size=2,
                    pool_stride=2,
                batch_norm=True,
                    training=batch_norm_train_mode,
                    default_bias=0.0)

  conv3 = conv_layer(layer_name='conv3',
      input_tensor=conv2, receptive_field=5,
                channels_in=64,
                    channels_out=64,
                    pool=True, pool_size=2,
                    pool_stride=2,
                batch_norm=True,
                    training=batch_norm_train_mode,
                    default_bias=0.0)

  conv4 = conv_layer(layer_name='conv4',
      input_tensor=conv3, receptive_field=5,
                channels_in=64,
                    channels_out=64,
                    pool=False, pool_size=2,
                    pool_stride=2,
                batch_norm=True,
                    training=batch_norm_train_mode)

  last_conv = conv4

  with tf.name_scope('last_conv_flatten'):
```

```
  conv_reshaped = tf.reshape(last_conv,
      [-1, flat_dimension(last_conv)])

fc1 = dense_layer(layer_name='fc1',
    input_tensor=conv_reshaped,
    input_dim=flat_dimension(last_conv),
              output_dim=1024,
                   decay=fc_decay,
                   default_bias=0.0)
dropped1 = tf.nn.dropout(fc1, keep_prob)

fc2 = dense_layer(layer_name='fc2',
    input_tensor=dropped1, input_dim=1024,
    output_dim=1024, decay=fc_decay)
dropped2 = tf.nn.dropout(fc2, keep_prob)

last_fc = dropped2

# Do not apply softmax activation yet! use
    the identity
logits = dense_layer(layer_name='output',
    input_tensor=last_fc, input_dim=1024,
    output_dim=10, act=tf.identity)
tf.add_to_collection('logits', logits)

print conv1.shape
print conv2.shape
print conv3.shape
print conv4.shape
print conv_reshaped.shape
print fc1.shape
print fc2.shape

### Losses and Accuracy ###
# Cross-Entropy Loss
with tf.name_scope('cross_entropy'):
  diff =
      tf.nn.softmax_cross_entropy_with_logits(
  labels=y_, logits=logits)
  with tf.name_scope('total'):
    cross_entropy = tf.reduce_mean(diff)
    tf.add_to_collection('losses',
        cross_entropy)
tf.summary.scalar('cross_entropy',
    cross_entropy)

# Total loss (weight decay + cross-entropy)
total_loss =
    tf.add_n(tf.get_collection('losses'),
    name='total_loss')

with tf.name_scope('train'):
  global_step = tf.Variable(0)
  learning_rate =
      tf.train.exponential_decay(
  learning_rate_init, global_step,
      decay_steps, decay_rate)
  tf.summary.scalar('learning_rate',
      learning_rate)
  optimizer =
      tf.train.AdamOptimizer(learning_rate).
  minimize(total_loss,
      global_step=global_step)


# Other metrics
with tf.name_scope('accuracy'):
    correct_prediction =
        tf.equal(tf.argmax(logits, 1),
        tf.argmax(y_, 1))
    accuracy =
        tf.reduce_mean(tf.cast(correct_prediction,
        tf.float32))
tf.summary.scalar('accuracy', accuracy)

# Might be needed for batch norm
extra_update_ops =
    tf.get_collection(tf.GraphKeys.UPDATE_OPS)
# Initializing the variables
init = tf.global_variables_initializer()

merged_summaries = tf.summary.merge_all()

# for saving the model in the end
saver = tf.train.Saver()

ts = datetime.now().strftime('%Y%m%d_%H%M')
logs_path = "logs/{}/".format(ts)
print "-"* 70
pwd = os.getcwd()+"/"
print("Run the following to start
    tensorboard server:\n" \
     "tensorboard
         --logdir=/{}{}".format(pwd,
         logs_path))

# Fill in the place holders depending on
    the context (training? testing?)
def feed_dict(batch_num, mode):
  if mode == 'train':
    batch_x, batch_y =
        grab_next_train_batch(batch_num,
        svhn_train_data, svhn_train_labels,
        batch_size)
    keep_proba = train_keep_prob
    training_mode = True

  elif mode == 'train_no_dropout':
    batch_x, batch_y =
        grab_next_train_batch(batch_num,
        svhn_train_data, svhn_train_labels,
        test_batch_size)
    keep_proba = 1.0
    training_mode = False

  elif mode == 'test_no_dropout':
    batch_x =
        svhn_test_data[:test_batch_size]
    batch_y =
        svhn_test_labels[:test_batch_size]
    keep_proba = 1.0
    training_mode = False

  elif mode == "test_all":
    start = batch_num * 4000
    end = start + 4000
    batch_x = svhn_test_data[start:end]
    batch_y = svhn_test_labels[start:end]
    keep_proba = 1.0
    training_mode = False

  batch_x =
      batch_x#normalize_batch(batch_x) #
      Subtract per image mean
  return {x: batch_x, y_: batch_y,
      keep_prob: keep_proba,
      batch_norm_train_mode: training_mode}
      # can't name values same as keys

###########################################
##                             ##
##        Launch the graph     ##
##                             ##
###########################################

with tf.Session() as sess:
  t_start = time()
  sess.run(init)
  train_writer =
      tf.summary.FileWriter(logs_path +
      '/train', sess.graph)
  test_writer =
      tf.summary.FileWriter(logs_path +
      '/test')
  # Training loop
  for epoch in xrange(training_epochs):
    print "Learning Rate: ",
        sess.run(learning_rate)
    for batch_num in xrange(total_batches):
      if batch_num % test_every ==
          test_every - 1:

        # Record summaries and accuracy on
            the *test* set
        summary, acc =
            sess.run([merged_summaries,
            accuracy],
            feed_dict=feed_dict(batch_num,
            mode='test_no_dropout'))
        test_writer.add_summary(summary,
            epoch * total_batches + batch_num)

        # To compare against *training* set
            (apples to apples comparison)
        summary = sess.run(merged_summaries,
            feed_dict=feed_dict(batch_num,
            mode='train_no_dropout'))
        train_writer.add_summary(summary,
            epoch * total_batches + batch_num)

        # Print occasional progress
        print('Validation accuracy at epoch
            %s: batch %s: %s' % (epoch,
            batch_num, acc))
      else:
        sess.run([optimizer,
            extra_update_ops],
            feed_dict=feed_dict(batch_num,
            mode='train'))

  train_writer.close()
  test_writer.close()

  t_end = time()
  elapsed_mins = (t_end - t_start) / 60.0
  print "\nOptimization Finished! in {}
      minutes".format(elapsed_mins)

  ### Test on the full test set ###
  test_batches = (len(svhn_test_data) /
      4000) + 1
  accuracies = []
  for batch_num in range(test_batches):
    acc = sess.run([accuracy],
        feed_dict=feed_dict(batch_num,
        mode='test_all'))
    accuracies.append(acc)

  print accuracies
  print np.mean(accuracies)

  # Save down the current model
  if not os.path.exists("models"):
      os.makedirs("models")
  saver.save(sess,
      "models/{}".format(model_name))


###########################################
##                              ##
##          Parameters          ##
##                              ##
###########################################

# Training Parameters
batch_size = 100
test_batch_size = 1000
test_every = 500
total_batches = int(len(svhn_train_data) /
    batch_size)

learning_rate_init = 0.002
decay_steps = total_batches #2000
decay_rate = 0.5 #0.95
training_epochs = 4

# Regularization
fc_decay = 0.0
train_keep_prob = 0.75

model_name = "baseline_old_data"

# Make it rain!
run()
```

ACKNOWLEDGMENT

We acknowledge Dr. Hamid Tizhoosh and Shivam Kalra for their assistance in teaching us about neural networks and machine learning best practices. We also would like to acknowledge AWS GPU instances for saving us weeks of training and experimentation time.

REFERENCES

[1] Goodfellow, Ian J., et al. "Multi-digit number recognition from street view imagery using deep convolutional neural networks." arXiv preprint arXiv:1312.6082 (2013).
[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. (2012).
[3] Jaderberg, Max, et al. "Reading text in the wild with convolutional neural networks." International Journal of Computer Vision 116.1 (2016): 1-20.
[4] Decoste, Dennis, and Bernhard Schlkopf. "Training invariant support vector machines." Machine learning 46.1 (2002): 161-190.
[5] Lee, Chen-Yu, Patrick W. Gallagher, and Zhuowen Tu. "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree." International conference on artificial intelligence and statistics. 2016.

[6] Sermanet, Pierre, Soumith Chintala, and Yann LeCun. ”Convolutional neural networks applied to house numbers digit classification.” Pattern Recognition (ICPR), 2012 21st International Conference on. IEEE, 2012.

[7] Ioffe, Sergey, and Christian Szegedy. ”Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv preprint arXiv:1502.03167 (2015).

[8] Grundland, Mark, and Neil A. Dodgson. ”Decolorize: Fast, contrast enhancing, color to grayscale conversion.” Pattern Recognition 40.11 (2007): 2891-2896.

[9] Yao, Hang. ”Street View House Numbers.” Online Repository. Available: github.com/hangyao/street_view_house_numbers (2016).

[10] Karpathy, Andrej. ”Cs231n: Convolutional neural networks for visual recognition.” Online Course (2016).

[11] Masci, Jonathan, et al. ”Stacked convolutional auto-encoders for hierarchical feature extraction.” Artificial Neural Networks and Machine LearningICANN 2011 (2011): 52-59.

[12] Zeiler, Matthew D., and Rob Fergus. ”Visualizing and understanding convolutional networks.” European conference on computer vision. Springer International Publishing, 2014.

[13] Jarrett, Kevin, Koray Kavukcuoglu, and Yann LeCun. ”What is the best multi-stage architecture for object recognition?.” Computer Vision, 2009 IEEE 12th International Conference on. IEEE, 2009.

[14] Glorot, Xavier, and Yoshua Bengio. ”Understanding the difficulty of training deep feedforward neural networks.” Aistats. Vol. 9. 2010.

[15] Merity, Stephen. ”Explaining and illustrating orthogonal initialization for recurrent neural networks”. Online Tutorial (2016).

[16] Szegedy, Christian, et al. ”Inception-v4, inception-resnet and the impact of residual connections on learning.” arXiv preprint arXiv:1602.07261 (2016).

[17] Benenson, Rodrigo. ”Classification datasets results”. Available: rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (2017).

[18] Liao, Zhibin, and Gustavo Carneiro. ”Competitive multi-scale convolution.” arXiv preprint arXiv:1511.05635 (2015).

[19] Yan, Ji. ”Digit recognition from Google Street View images”. experimentationground.wordpress.com/2016/09/26/digit-recognition-from-google-street-view-images (2016).

[20] Srivastava, Nitish, et al. ”Dropout: a simple way to prevent neural networks from overfitting.” Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[21] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011. (PDF)

[22] Kingma, Diederik, and Jimmy Ba. ”Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980 (2014).

[23] Abadi, Martn, et al. ”Tensorflow: Large-scale machine learning on heterogeneous distributed systems.” arXiv preprint arXiv:1603.04467 (2016).

[24] TensorFlow. ”TensorBoard: Visualizing Learning”. Online Tutorial: https://www.tensorflow.org/get_started/summaries_and_tensorboard (2017).

[25] TensorFlow. ”Deep MNIST for Experts”. Online Tutorial: https://www.tensorflow.org/get_started/mnist/pros (2017).

[26] Nurlybayev, Baha. ”CNN SVHN.” Online Repository. Available: https://github.com/baha-nur/cnn_svhn (2016).