

Large-scale Identification of Multiple Digits From Real-World Images with Convolutional Neural Networks

RITCHIE NG

National University of Singapore

ritchieng@u.nus.edu

October 11, 2016

Abstract

Convolutional Neural Networks (ConvNets) have been known to be effective in predicting digits as shown from the typical MNIST dataset. This project aims to predict multiple digits from Google's The Street View House Numbers (SVHN) dataset. A quick search online would show how this has not been thoroughly explored compared to other datasets like MNIST, CIFAR-10, CIFAR-100 and ImageNet. In particular, this project aims to provide the base for others to easily swap out the topology of the ConvNet, except the readout layers, to proven working topologies such as GoogLeNet, VGGNet and ResNet.

I. INTRODUCTION

i. Overview

This project explores how convolutional neural networks (ConvNets) can be used to effectively identify a series of digits from real-world images that are obtained from *The Street View House Numbers (SVHN) Dataset*¹ [4]. ConvNets have evolved dramatically every year since the inception of the ImageNet Challenge² in 2010.

A proverbial ConvNet structure is the *LeNet-5* that has relatively few layers of convolutions, poolings, and full connections. Subsequently, with the advent of the ImageNet Challenge, we are experiencing a gradual trend towards deeper ConvNets with more layers and higher accuracy such as *AlexNet* [3], *ZFNet* [7], *VGGNet* [5], *GoogLeNet* [6], and *ResNet* [2] being the latest state-of-the-art implementation of ConvNets.

To this point, I started off with a simple ConvNet structure as a base where I made refinements to determine my optimal model for identifying multiple digits from real-world images.

ii. Problem Statement

I am attempting to predict a series of numbers given an image of house numbers from the SVHN dataset. An important thing to take note is that instead of the standard identification of numbers, as with the MNIST dataset, I now need to correctly detect the numbers and the sequence of numbers.

¹<http://ufldl.stanford.edu/housenumbers/>

²<http://image-net.org>

iii. Metric

Across the models, I compared the training, validation and test accuracies. And accuracy is measured by Equation 1.

$$\frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

Using Numpy, I created an accuracy function that is catered towards the problem of multi-digit prediction.

```
def accuracy(predictions, labels):  
    return (100.0 * np.sum(np.argmax(predictions, 2).T == labels)  
            / predictions.shape[1] / predictions.shape[0])
```

I compared test accuracies across the models to determine the best model. The higher the test accuracy, the better the model. And the model with the highest test accuracy is deemed as the best model. Validation and training accuracies are not considered for comparison because they are not as representative of the the model's predictive capability on unseen data.

II. ANALYSIS

i. Data Exploration

There are 10 classes in the data, 1 for each digit and 10 denotes '0'. In total, there are 73257 digits for training, 26032 digits for testing, and 531131 extra digits. For my implementation, I used the raw images that came with a variety of dimensions as shown in Figure 1.

Figure 1: Examples of the the database of raw images from SVHN



ii. Exploratory Visualization

When I went into the raw data to further explore the images, the images had a few properties as shown in Table 1.

Image 141 in Figure 2 may be an issue for the ConvNet to predict the correct numbers. Likewise, images 337, 645 and 650 may have issues due to alphabets and confusing lines as shown in Figures 4, 5, and 6. On the other hand, image 180 from Figure 3 would be easy for the ConvNet to predict and fortunately the majority of images are similar to this.

Table 1: *Examining Training Images*

| Image Number | Description |
|--------------|--|
| 141 | The image was barely perceptible even for a person. |
| 180 | This is a typical clear image. |
| 337 | This image has an "N" before the numbers. |
| 645 | This image has multiple alphabets. |
| 650 | This image has a confusing line amongst the numbers. |

Figure 2: *Blur image 141*

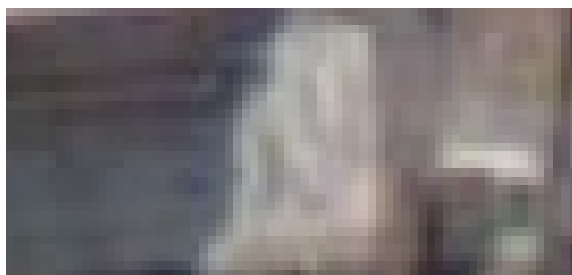


Figure 3: *Clear image 180*



Figure 4: *Image 337 with an alphabet*

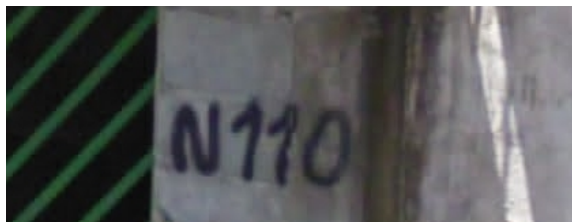


Figure 5: *Image 645 with multiple alphabets***Figure 6:** *Image 650 with confusing line*

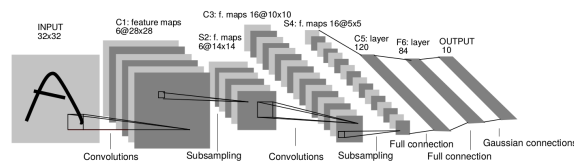
iii. Programming Language and Environment

My programming language of choice is Python 2.7 and I will be using TensorFlow to build our deep ConvNets. Also, this project uses TensorBoard extensively for visualizations. It is a relatively new area for many people who are using Tensorflow and there are detailed instructions on how this has been implemented and how you can view the interactive visualizations in the implementations section.

Other substantial packages that are needed to run the code are shown in the following code snippet. Most of them are used for loading the data and processing the data. With *numpy* being the most used package for data manipulation.

```
import os
import sys
import tarfile
import tarfile
import tensorflow as tf
import numpy as np, h5py
from __future__ import print_function
from six.moves import cPickle as pickle
from six.moves import range
from six.moves.urllib.request import urlretrieve
from scipy import ndimage
from PIL import Image
from numpy import random
```

The models' computations are accelerated with a CUDA-enabled GPU on Amazon Web Service (AWS) with a g2.2xlarge EC2 instance and a 60 GB EBS volume attached to it and each trial went

Figure 7: Typical ConvNet Topology

through 60,000 steps to achieve the results reported.

As the computations are expansive, I recommend using a GPU instance on AWS to run the code I have written. I understand that many struggle to set up the right environment to run Tensorflow, so I created an AMI that is available across all locations in the world when you are launching your GPU instance. Simply search for "TFAMI" under "community AMIs" and you can launch your instance and run the code without doing anything. This is an open-source AMI I made and it is available on Github³ with more information.

With all the languages, packages and environment clarified, I will now proceed to explain ConvNets and why they are used instead of plain-vanilla Multi-Layer Perceptrons (MLPs).

iv. Algorithms and Techniques

Assume we have a dataset of images all with a consistent dimension measuring 128 pixels x 128 pixels, we would require a total of 16,384 input weights plus an additional bias in the case of a traditional feedforward Multi-layer Perceptron (MLP). Importantly, the major flaw is that if we reshape the pixels into a vector, we lose the spatial structure of the images. This is an area that a ConvNet addresses and also the reason why I am using a ConvNet over a traditional MLP in this multi-digit recognition project.

ConvNets preserve the spatial relationship amongst pixels, learning internal feature representations where they are used across the whole image. Hence a digit, for instance, can be blurred, shifted to the left or right, or even distorted and the ConvNet can still detect the digit. Currently, ConvNets are used for object recognition in images and videos with state-of-the-art results. The benefits of a ConvNet are summarized in Table 2. Subsequently, I will explain the components of a ConvNet and how it works.

Table 2: ConvNet Benefits

| Number | Benefit |
|--------|---|
| 1 | Fewer weights |
| 2 | Able to tackle object distortions, changing object positions and orientations |
| 3 | Learn feature representations amongst input pixels |

The typical ConvNet's topology is made up of 3 layers as shown in Figure 7: the convolutional layers, the pooling layers and the fully-connected layers. Now, I will explain the detailed components of a typical ConvNet and how the layers are related to one another.

³<https://github.com/ritchieng/tensorflow-aws-ami>

i Filters

As you can see from Figure 7, filters are the little boxes that map one feature maps to the next layer's feature maps. Think of it as a little camera that goes through the whole image comprising the alphabet "A". You would then choose a filter with a dimension, the box, of say 5×5 pixels and move it across the whole image. How you move the box is called the stride, where if you have a stride of 1, you would move the box 1 pixel at a time in this case. If you have a large stride, you would move more for each step. Once you slide the filter across the whole image once, this would map to a feature map in the following layer. This brings me to my explanation of feature maps.

ii Feature Maps

A feature map is simply the output of one filter that is applied to the previous layer. If you have 10 filters applied to the previous layer, you would have 10 feature maps. So if you have n number of filters, you would have n number of feature maps in the next layer. An important thing to note, is that the process of mapping the inputs to the feature maps in the next layer is called a convolution. Logically, when we map the inputs to the feature maps, we have our convolution layer. But what does the following step called "pooling" does in Figure 7? This is a simpler concept that I am going to tackle now.

iii Pooling Layers

Pooling layers are layers where we have reduced the dimensionality of the previous layer. For example, we have a convolution layer with a dimension of $32 \times 32 \times 10$ (image size \times image size \times number of feature maps). We can apply max pooling or average pooling to the convolution layer to reduce the dimensions to $16 \times 16 \times 10$. We are doing this to reduce the image size for computational efficiency, reduce overfitting or simply to generalize feature representations. Take note that max pooling or average pooling is similar to a filter where we need to specify the dimensions and this would determine the subsequent dimension of the pooling layer. Also, "pooling" has some variants and it is often called "sub-sampling" because we are down-sampling the image to a lower dimensionality. Finally we continue to add a mix of convolutions and pooling until we reach the fully-connected layer.

iv Fully Connected Layers

This is a typical layer in your MLP where we are trying to create non-linear combinations of images to make predictions. Although it is important to note recent structures like *ResNet* [2] do not use fully-connected layers at all and simply use a readout layer like softmax to make predictions.

v Readout Layer

This is often called the softmax layer where we are making the final predictions based on the previous pooling, convolution or fully-connected layer. It is the final layer.

v. Benchmark

My metric of choice is accuracy, as such my benchmark for this problem would be an out-of-sample, test, accuracy of 96%. This was the accuracy [1] claimed to have obtained with their model.

III. METHODOLOGY

i. Data Preprocessing

It is important to note that the images came in many dimensions. As such, I located the bounding boxes and cropped the images using the bounding boxes' location such that all images are of the size 32 x 32 pixels. Moreover, I also normalized the images and converted them from 3 channels, color images, to 1 channel, greyscale images.

It is important to note that there were originally 73257 digits for training, 26032 digits for testing, and 531131 extra digits. I distributed the extra digits and testing digits randomly to increase our training dataset so I have more data to train on. Moreover, I deleted an image that had more than 5 digits as it is an anomaly in the training set. The final shapes of the datasets' tensors and labels' arrays are shown in the following code.

```
Training data shape: (230070, 32, 32, 1)
Training label shape: (230070, 6)
Validation data shape: (5684, 32, 32, 1)
Validation label shape: (5684, 6)
Test data shape: (13068, 32, 32, 1)
Test label shape: (13068, 6)
```

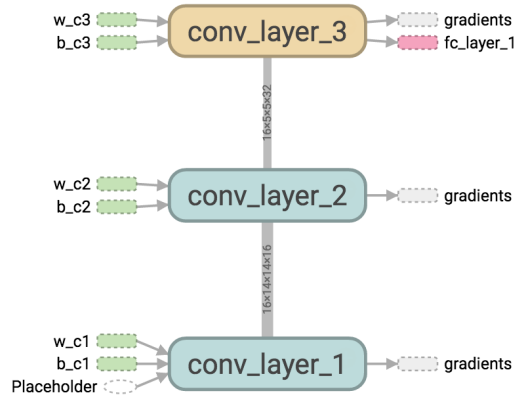
All the processed images are then pickled into a file, "SVHN.pickle", where anyone can easily access the tensors through the following code.

```
print('Loading pickled data...')

pickle_file = 'SVHN.pickle'

with open(pickle_file, 'rb') as f:
    save = pickle.load(f)
    X_train = save['train_dataset']
    y_train = save['train_labels']
    X_val = save['valid_dataset']
    y_val = save['valid_labels']
    X_test = save['test_dataset']
    y_test = save['test_labels']
    del save
    print('Training data shape:', X_train.shape)
    print('Training label shape:', y_train.shape)
    print('Validation data shape:', X_val.shape)
    print('Validation label shape:', y_val.shape)
    print('Test data shape:', X_test.shape)
    print('Test label shape:', y_test.shape)

print('Data successfully loaded!')
```

Figure 8: Trial 1 Convolution Layers

ii. Implementation of Trial 1

In this first trial, I started off with a simple model comprising the following layers shown in Table 3. The idea of using five parallel softmax layers is inspired by the recommendation on tackling multiple digits by [4]. It is important to take note that [4] used 6 parallel softmax layers instead of 5 as they included the length of the digits.

Table 3: ConvNet Topology: Trial 1

| Layer | Description |
|----------------------------|---|
| Input Layer | Numpy array of size (230070, 32, 32, 1) |
| Convolution 1 | Filters: 16 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Max Pooling | Receptive Field: 2 x 2 Stride: 2 Padding: Valid |
| Convolution 2 | Filters: 32 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Max Pooling | Receptive Field: 2 x 2 Stride: 2 Padding: Valid |
| Convolution 3 | Filters: 96 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Dropout | Keep probability of 0.5 |
| Fully Connected Layer (FC) | Nodes: 64 |
| Softmax Layer | 5 softmax (readout) activation layers for 5 digits |

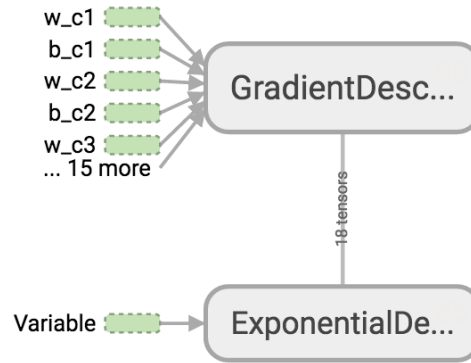
As you can see from Figure 8, we have 3 convolution layers with 3 sets of weights and biases. And the final convolution layer feeds to the fully connected layer as shown in Figure 9.

Moreover, the weights are initialized with special functions I created so it is easier to expand the convolutional networks and have access to TensorBoard's visualizations. The weights are initialized with a random normal distribution with a standard deviation of 0.01. And the functions take in two arguments: the tensor and the names for TensorBoard's visualizations.

Figure 9: Trial 1 FC Layer



Figure 10: SGD TensorBoard graph



```
def init_weights(shape, name):
    return tf.Variable(
        tf.random_normal(shape=shape, stddev=0.01),
        name=name)

def init_biases(shape, name):
    return tf.Variable(
        tf.constant(1.0, shape=shape),
        name=name
    )
```

For my optimizer, I used Stochastic Gradient Descent (SGD) with a step-wise decay with a starting learning rate of 0.05. In the equation as shown in Equation 2, I used step-wise decay, so $\frac{\text{globalstep}}{\text{decaysteps}}$ would return an integer. And in Figure sgdtbtrial1, we can see the SGD's graph with the relevant weights and biases.

$$\text{decayedlearningrate} = \text{learningrate} * \text{decayrate}^{\frac{\text{globalstep}}{\text{decaysteps}}} \quad (2)$$

Importantly, unlike running a ConvNet on the MNIST dataset or majority of the datasets, our loss function is a combination of the parallel softmax (readout) activation layers. The code of the loss function can be seen in the following code and the graphical representation can be seen in Figure 10.

Figure 11: Loss TensorBoard graph

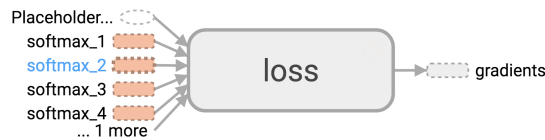
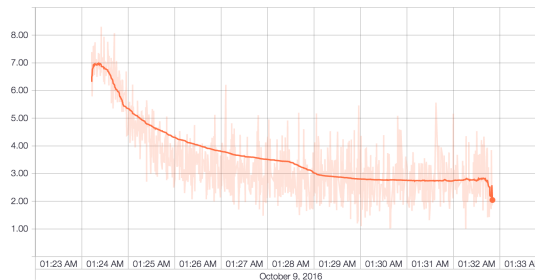


Figure 12: Loss TensorBoard graph



```
with tf.name_scope("loss"):
    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits_1, tf_train_labels[:, 1])) + \
        tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits_2, tf_train_labels[:, 2])) + \
        tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits_3, tf_train_labels[:, 3])) + \
        tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits_4, tf_train_labels[:, 4])) + \
        tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits_5, tf_train_labels[:, 5]))
    # Add scalar summary for cost
    tf.scalar_summary("loss", loss)
```

With this model, I managed to achieve a test accuracy of 87.64% and the training and validation accuracies are shown below. And it is interesting to note how the loss seems to plateau and they are very erratic as seen in Figure 12.

```
Minibatch accuracy: 90.0%
Validation accuracy: 87.2202674173%
Test accuracy: 87.6400367309%
```

I believe this warrants a second trial to refine some hyperparameters like my optimizer, initialization of weights, and the dropout probability.

iii. Refinements

In this section, I implemented our ConvNet with different hyperparameters from my first trial.

First, I changed my dropout probability to 0.8 and all else are similar as shown in Table 4.

Table 4: *ConvNet Topology: Trial 2*

| Layer | Description |
|----------------------------|---|
| Input Layer | Numpy array of size (230070, 32, 32, 1) |
| Convolution 1 | Filters: 16 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Max Pooling | Receptive Field: 2 x 2 Stride: 2 Padding: Valid |
| Convolution 2 | Filters: 32 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Max Pooling | Receptive Field: 2 x 2 Stride: 2 Padding: Valid |
| Convolution 3 | Filters: 96 Receptive Field: 5 x 5 Stride: 2 Padding: Valid |
| ReLU | Rectified Linear Unit Activation |
| Dropout | Keep probability of 0.8 |
| Fully Connected Layer (FC) | Nodes: 64 |
| Softmax Layer | 5 softmax (readout) activation layers for 5 digits |

Second, I changed my optimizer from SGD to AdaGrad. This is because our data is quick sparse for digits with many numbers and AdaGrad is well-suited for such a dataset because it adapts the learning rate to the parameters performing larger updates for infrequent parameters and smaller updates for frequent parameters.

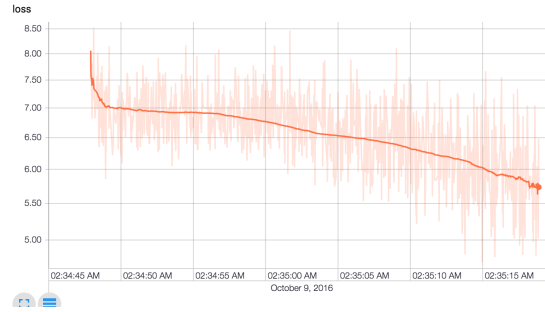
Third, I changed the initialization of the weights using Xavier Initialization. It is a sampling of Gaussian distribution where the variance is a function of the number of neurons as shown in the following custom functions I made where the first function is for initializing weights for the convolution layers and the second function is for the initialization of weights for the FC layer after all the convolutions.

```
def init_weights_conv(shape, name):
    return tf.get_variable(shape=shape, name=name,
                           initializer=tf.contrib.layers.xavier_initializer_conv2d())
def init_weights_fc(shape, name):
    return tf.get_variable(shape=shape, name=name,
                           initializer=tf.contrib.layers.xavier_initializer())
```

With this, I managed to get a test accuracy of 91.62% with the following test and validation accuracies.

```
Minibatch accuracy: 96.25%
Validation accuracy: 90.5594651654%
Test accuracy: 91.6176920722%
```

Interestingly, unlike previously, the loss function is not plateauing and there seem to be more room for learning as shown on Figure 13.

Figure 13: Loss TensorBoard graph for Trial 2

IV. RESULTS

Comparing the two trials, we can see how the second model has a higher test accuracy and has yet to fully converged even with a higher accuracy. Moreover, the second model have higher validation and training accuracies too from Table 5 although we are not using them as a benchmark to decide the best model. Hence, since the second model has the best accuracy, I have decided to settle with this simple ConvNet that produced a respectable 91.62% on unseen data.

Table 5: Comparison of the performance of my models

| | Trial 1 | Trial 2 |
|---------------------|---------|---------|
| Test Accuracy | 87.64% | 91.62% |
| Training Accuracy | 90.00% | 96.25% |
| Validation Accuracy | 87.22% | 90.56% |

i. Model Evaluation and Justification

The huge performance gain from the second trial came from a more suitable optimizer, AdaGrad, compared to SGD due to the nature of our sparse data where we have few images with many digits like 4 or 5. Also, the initialization of the weights using Xavier's initializations contributed to this performance gain too.

However, comparing with my benchmark that is set against [1], our test accuracy of 91.62% failed to meet the benchmark of 96%. Yet, it is important to note how this has room for improvements by simply running with more iterations as the model has not fully converged as shown in 13. Also, the accuracy is sufficiently high compared to the test accuracy of 90% obtained from [1] for another similar dataset using the same implementation.

V. CONCLUSION

i. Visualizations of Predictions

In this section, I will show some predictions that worked well and some that did not. From Figure 14, we can see how it has inaccurately predicted 23 and this is not expected since the image is

Figure 14: *Inaccurate prediction of 28*



Figure 15: *Accurate prediction of 569*



relatively clear with no alphabets or any other obstructions. But it may be how 3 looks similar to 8. However, for an expectedly tough prediction like Figure 15 with many alphabets and a messy background, the model accurately predicts "569". Also, previously I mentioned how lines may be an issue, however the model managed to accurately predict "260" with many lines in the background as shown in Figure 16.

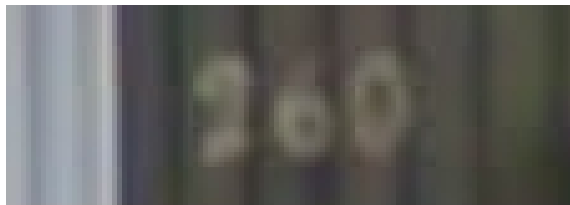
ii. Summary of Tackling the Problem

I will be highlighting which parts are challenging or interesting, and if so, why and how one can go about to tackle that particular part.

i Loading the Data

This is easy and it did not pose much of a challenge.

Figure 16: *Accurate prediction of 260*



ii Exploring the data

This is challenging as I used a lot of Python scripts to explore the data but it is still manageable.

iii Processing the data

This is by far the most challenging task before we actually build our ConvNet. You must be familiar with bounding boxes to know how to process the images. I believe there may be easier ways to process the data by creating a standardized package for future image processing based on certain hyperparameters.

iv Building the model

This is the most challenging part of the model-building process. I struggled in this area as most of the predictions I have done do not use this unique structure where I have multiple parallel softmax layers. But guidance from the paper in [1] helped.

v Visualization of the model and its performance using TensorBoard

This is an underused feature. The majority of code examples online for any dataset do not use TensorBoard. But I decided to put in the effort to implement the solution with TensorBoard so others can explore too.

In sum, the most challenging parts of the project were processing the images and building a ConvNet with parallel softmax layers and using TensorBoard helped to visualize where I went wrong and if the graph I built matched what I had in mind. I highly recommend more people to use TensorBoard.

iii. Area for Improvements

The accuracy of the second model can be further improved with more training as it has yet to converge. Also, more investigation can be done to look at the large variation in our loss function. I did not use any renowned topology highlighted in the introduction such as *VGGNet* or *ResNet* to keep this project focused on how to classify multiple digits. This particular classification of multiple objects in a single image is lacking in examples online hence it provides a base for people to explore how we can use proven topologies such as *ResNet* on this problem while maintaining the parallel readout layers to classify multiple digits. More importantly, there is room for improvement by using existing weights trained on the ImageNet dataset using *ResNet* or other winning topologies and change the readout layer to the parallel readout layers. This may give a substantial boost to our test accuracy and reduce our training time. This method is recently coined as transfer learning and can be explored using this project's base code⁴.

REFERENCES

- [1] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. 12 2013.
- [2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 12 2015.

⁴You can access the open-source source code at <https://github.com/ritchieng/NumNum>.

- [3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [4] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.
- [5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 09 2014.
- [6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. 09 2014.
- [7] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. 11 2013.