Multi-Digit Recognition with Street View Housing Numbers

Kevin Van Nguyen

Udacity Machine Learning Engineering Nanodegree Capstone Report

Abstract

This project explores multi-digit recognition using images that come from real world housing numbers. Typical state-of-the-art approaches use computationally demanding convolutional neural networks (CNN) [1]. I instead experiment with a simple design pattern with recently developed methods such as Batch Normalization. My project presents how I overcome two obstacles: 1) A pipeline bottleneck and 2) Overfitting on unseen images. To overcome the bottleneck, I use asynchronous methods to fetch images. For overfitting, I use data augmentation techniques. My best model achieves 95% per digit accuracy on cropped street view housing numbers.

1. Definitions

Recognizing numbers of arbitrary length in natural images is a hard problem [2]. The implications of research in this area have tremendous value for sectors such as urban development. Take for example the Street View feature on Google Maps. Street View provides panoramic views along many streets in the world as you get directions on the phone or laptop. Multi-digit recognition helped Street View transcribe address from those photos. In this project, I experiment the with images used to develop the technology behind that system, the Street View Housing Numbers (SVHN) dataset [3].

1.1. Problem Statement

The multi-digit recognition problem boils down to taking images containing multiple numbers and simultaneously recognizing them [2]. SVHN has various fonts, colors, styles, orientations, and number arrangements. These variations make the SVHN challenging for multi-digit recognition. Environmental factors such as lighting, shadows, specularities further complicate the appearance of these numbers [2].

Research on multi-digit recognition with SVHN gravitates towards object detection that is efficient, scalable, and end-to-end. [2] achieved over 96% accuracy while training a deep CNN for six days. Shortly after, [4] introduced an attention-based model by combining elements of CNN, Recurrent Neral Networks, and reinforcement learning. It took them only a single GPU and three days of training to train a state-of-the-art end to end digit recognition system. Recently, [1] demonstrated they could achieve state-of-the-art results with a unified CNN architecture by placing spatial transformers into CNN layers. The Spatial transformer network overcomes problems related to invariance to translation, scale, and rotation while remaining efficient with parameter use and design complexity [1].

Figure 1: Image in SVHN. This image shows a house number spray painted on a wood board. Our goal for this project is to design software that reads various in image input, like the picture above and generate a label for it (i.e., 10) despite natural variation the comes from taking photos.

1.2. Performance Metric

Human level accuracy on SVHN is 98% [5]. State-of-the-art models typically train for several days on GPUs with architectures as deep as eight to ten convolutional layers. At the onset, I set a computational budget of 10 epochs with five convolutional layers. This budget allows me to do all training on a single laptop without GPUs. Training for several days on GPUs is beyond the scope of this project.

Many open source projects focused on simultaneous SVHN digit reconition [6, 7], use per character accuracy (digit accruacy):

$$Accuracy = \frac{\#ofCorrectDigits}{\#ofDigits} \tag{1}$$

House numbers are usually not longer than three digits [2] and digit accuracy gives a good understanding of how well a model is accurately classifying each digit. My project will focus on digit accuracy based on its interpretable value.

An alternative to digit accuracy is sequence accuracy. Sequence accuracy uses the ratio of the total number of correct sequences and total images (one sequence per image). An adverse property of sequence accuracy is it takes rare cases (such as longer sequences) and holds them on equal ground with common cases (shorter sequences). Class imbalance (not represented equally) inherent in SVHN causes sequence accuracy to disproportionately penalize different sequences.

Figure 2: Example of the different shapes, sizes, and rotation of digits an image have in SVHN.

2. Analysis

SVHN is composed of over 250,000 house number images. Images come in various sizes, angles, and color contrasts. Split into three sets: SVHN consists of 33,402 images for training, 26,032 images for testing, and 202,353 additional images free for experimentation. On average, I find that 1) testing images from the tend to be larger in height and width, 2) extra images tend to be smaller in width, and 3) training images tend to be smaller in height.

Class imbalance is apparent in SVHN. Testing data have a higher proportion of two sequence numbers compared to training and extra. Three or more sequence digits are rare in all cases. To deal with this property, I apply stratified sampling when creating train, test, and validation sets during experimentations.

2.1. Visual Inspection

Visual inspection confirms a housing numbers can vary in location and rotation. For the most part, images have

Characteristics	Train	Validation	Extra
Avg. Height	57.2	71.6	60.8
Avg. Width	128.3	172.6	100.4
Height Range	12 to 501	13 to 516	13 to 668
Width Range	25 to 876	31 to 1083	22 to 668

Table 1: Image characteristics piror to preprocessing show that images vary in size and color. One of the most challenging aspects of this project is preparing the data for training.

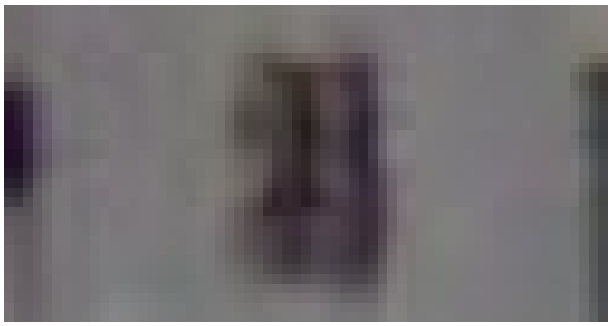

Figure 3: Example of a challenging image to classify, even for the human eye. Can you tell if thats a 23 or 25?

reasonable views of housing numbers. However, some are not identifiable from a first glance. The challenge is in the foreground, background confusion, and very low resolution/blurred as reasons for human level mistakes with classification [5].

2.2. Getting the Images Ready

Concerned about overfitting, I adopt a slight variant of [2]'s approach. They prepared their data with scale variability in mind. To center the digit in each image, I tightly crop the images to their bounding boxes. To do so, I first, find the small rectangular bounding box that will contain the individual character. Second, expand its bounding box by 30% in both the x and the y-direction. Third, crop the image to that bounding box. And finally, I resize the images to 64x64 pixels.

3. Algorithms and Techniques

For my baseline, I train a CNN with RGB images. Once an afterthought in the computer vision community, CNNs have become the dominant approach for almost all recognition and detection tasks (thanks to success at ImageNet 2012) [8]. CNN architectures make the explicit assumption that the inputs are images, which allows me to encode certain properties into the design. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network [9].

Figure 4: Examples of images after I center, expand the bounding box, and resize images.

One fully connected layer. Inspired by GoogleNets [10] crafty utilization of the computing resources, I use one affine layer to connect the feature maps to a softmax operation. Using one saves parameters otherwise spent if I used the two or more fully connected layers.

Simplisitic alternating convolutions. Most modern CNNs used for object recognition are built using alternating convolution and max-pooling layers followed by a small number of fully connected layers [11]. I depart from this by removing max-pooling. For the first two convolutional layers, I have small filters of 3x3 with stride 1 and for the last three 5x5 filters with a stride of 2.

Batch Normalization. During training, convolutional weights are prone to have shifting distributions. These changes slow down the model's ability to learn [12]. I use batch normalization before activation to smooth these "covariate shifts." Another benefit of batch normalization is better resistance to overfitting, that is why I use dropout once after the last convolution with a drop probability of .85.

3.1. Benchmark

Looking towards peers that have worked on similar projects. Open source projects focused on SVHN [6, 7], typically have scores between of 90% and 93%. My benchmark is 92% the average of between 90% and 93%.

Layer	Type	Map	Kernal	Stride
S (7-12)	SOFTMAX	-	-	-
FC6	FC	-	-	-
D5	DROPOUT	256	-	-
C4	CONV	256	5x5	3
C3	CONV	64	5x5	3
C2	CONV	32	3x3	1
C1	CONV	16	3x3	1

Table 2: CNN Architecture. All convolution layers use "SAME" padding, batch normalization, and rectified linear units (Relu). Six independent softmax classifiers are performed after the fully connected layer. Five for predicting a digit at a particular position in the sequence. And one to predict sequence of length in an image.

To the best of my knowledge, there is not an open source implementation project that has a digit accuracy beyond 94%.

4. Methodology

I created a train, test, and validation dataset for the experiment. For validation, I take a 5% stratified random sampling from combined train and extra images for hyperparameter tuning. The remaining combined training and extra images are for training. The test images are left untouched for the final evaluation of the model.

4.1. Pipeline

I use a customized input pipeline to read images with eight threads on a MacBook Pro with 2.2 GHz Intel Core i7, 16 GB 1600 MHz DDR3. Batches are then randomly shuffled during the queue process. Protocol buffers from TensorFlow's standard data format make this process easy. Protocol buffers isolate the model from disk latency and computationally expensive image preprocessing [13]. The training ran for 60,000 steps (10 epochs).

In the pipeline, each image is subject to the following:

- Randomly distorted saturation
- Randomly distorted brightness
- Randomly distorted hue
- Randomly distorted contrast
- Pixel wide mean subtraction
- Resize from 64x64x3 to 32x32x3

4.2. Implementation

For my loss function, I use cross-entropy with Xavier initialized weights. For optimization, I use Adam with a .0001 learning rate. Adam is based on adaptive estimates of lower-order moments which make it straightforward to implement, computationally efficient, and require little memory [9]. Experts often recommend it as their default choice for CNNs [9].

4.3. Refinement

My biggest challenge for the project was overcoming computational bottlenecks in my pipeline. My laptop would not train the model with preloaded variables at batch size 32 or 16. To overcome this, I serialized the data and used prefetch queues to feed training images into our model (mentioned above).

4.4. Initial Solution

As a baseline, I use images with RGB color channels and obtain a best classification accuracy of 93.4%. For training and validation, I get 94.7% and 97.2%. After ten epochs, the model suffers slightly from overfitting. I am encouraged by these results because I score above my original benchmark of 93%. Since my training errors are significantly higher than the baseline of 92%, I focus on data augmentation as opposed to tuning the number of hidden units or add additional layers for model improvement. I can believe I can reduce generalization error by collecting more training data [3] my time is better spent on improving preprocessing.

4.5. Data Augmentation

First, I experiment with converting images from RGB to grayscale. Grayscale transformation decreases the number of parameters for each image from 3,072 (32*32*3) to 1,024 (32*32*1). I get his idea from [5] who found that converting to grayscale did not influence performance on SVHN. Second, I introduce random cropping within the data augmentation pipeline. Random cropping gives me an idea of how well my model will do when small perturbations (changes) are in the training data. This idea comes from [2, 4, 1] who use it in their pipeline to account for image variation.

5. Results

After making refinements to my model, grayscale preprocessing leads to my best performance – 95% accuracy. Whereas my models with, RGB images achieve 93.6

Model	Training	Validation	Test
RGB	0.977	0.947	0.936
Grayscale	0.981	0.958	0.950
RGB + RC	0.956	0.940	0.920
Grayscale + RC	0.948	0.942	0.921

Table 3: Results. In general we see good performance across the board. But Grayscale preprocessing really stands out in terms of performance. I feel the results can improve if given the opportunity to let the model converge.

5.1. Model Evaluation and Validation

The grayscale preprocessing effects exceed my expectation. Experiments with random crops have lower performance 92%. We also notice a slight drop-off will occur when numbers in images deviate in location or rotation. Further discussion is in the Improvement Section. Observing the model in action, some incorrect predictions are within reason. Some of the predictions are hard to even for a human to make. Some images have the number cut off and positioned in a way that makes it seem like it's another number.

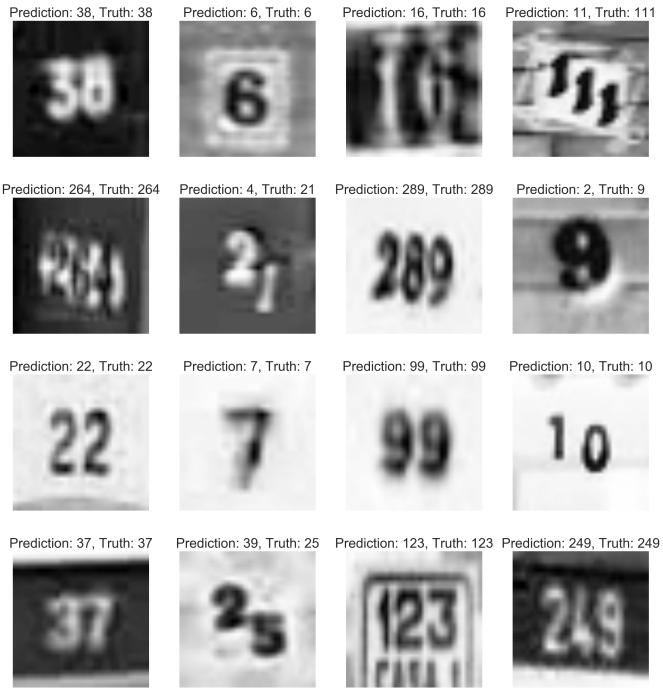

Figure 5: Some predictions made by the model. You can see at image (3, 4 it can make correct predictions in the presence of other text. And in the presence of tricky views, image (2,2) you can understand why a 4 is predicted instead of 21 (true label) as the digits in the picture resembles a 4.

5.2. Justification

Based on initial benchmarks, we accomplish our goal by scoring 95%, well over 92%. In spite the fact we discover challenging images during error analysis. I am confident the model will perform better if given images are reasonably prepared (i.e., not cut off or rotated at a weird angle). Our model architecture demonstrated efficient use of CNNs in a resource-limited situation. Refinement in preprocessing steps improves model performance by reducing overfitting while decreasing the number of parameters introduced to our model. Moreover, our general architecture deals with different adjustments such as random crops when introduced into the pipeline and still obtain results similar to our benchmark.

Figure 6: Example of incorrect predictions. Some of the error shows the model struggles with reasonable challenges such cut off number and deciving rotations.

6. Reflection

This project demonstrates how we took real-world images and addressed problems with efficient multi-digit recognition. The following are key milestones for our project:

- Downloading a large dataset
- Preparing it for our pipeline
- Refining our pipeline to deal with CPU bottlenecks
- Replicating preprocessing steps used by [2]
- Implementing a custom CNN architecture
- Improving upon our initial results

6.1. Improvement

Our project focused on *tightly/reasonably cropped images.* And based on our experiments, we expect a slight decrease in performance on free form unbounded natural images because of limitations related to bounding box cropping. A simple solution is to run our models for longer than ten epochs and allow for our models to continue learning. However, in the spirit of our computational budget, we kept the model running for ten epochs. A natural extension of our project is integrating localization techniques for a drop in replacement for bounding box cropping. Techniques such as Bounding Box regression, Recurrent Convolutional Neural Networks, or Spatial Transformer Networks to localize are tasks we can include into the pipeline before we perform classification.

[1] M. Jaderberg, K. Simonyan, A. Zisserman, et al., Spatial transformer networks, in: Advances in Neural Information Processing Systems, pp. 2017–2025.

[2] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, V. Shet, Multi-digit number recognition from street view imagery using deep convolutional neural networks, arXiv preprint arXiv:1312.6082 (2013).

[3] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016. Urlhttp://www.deeplearningbook.org.

[4] J. Ba, V. Mnih, K. Kavukcuoglu, Multiple object recognition with visual attention, arXiv preprint arXiv:1412.7755 (2014).

[5] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A. Y. Ng, Reading digits in natural images with unsupervised feature learning (2011).

[6] hangyao, street_view_house_numbers, urlhttps://github.com/hangyao/street_view_house_numbers, 2016.

[7] camigord, Ml capstoneproject, urlhttps://github.com/camigord/ML_CapstoneProject, 2013.

[8] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436–444.

[9] A. Karpathy, Cs231n: Convolutional neural networks for visual recognition, urlhttp://cs231n.stanford.edu, 2016.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9.

[11] J. T. Springenberg, A. Dosovitskiy, T. Brox, M. Riedmiller, Striving for simplicity: The all convolutional net, arXiv preprint arXiv:1412.6806 (2014).

[12] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, arXiv preprint arXiv:1502.03167 (2015).

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.