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 photographs of 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. This 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]. 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. 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) image 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, and specularities further complicate the appearance of these numbers [2].

1.2. Research and Innovation

Research on multi-digit recognition with SVHN gravitates towards deep learning that is efficient, scalable, and end-to-end. In 2013, it took [2] six days to train a CNN to achieve over 96% accuracy on SVHN. A year later, it took [4] three days to train an attention-based model to superpass [2]. And in 2015, [1] demonstrated they could achieve state-of-the-art results with a unified CNN architecture. Placing a Spatial Transformer (ST) into CNN layers, the model learned how to do preprocessing to invariances in translation, scale, and rotation by itself!

Figure 1: Example image from SVHN. A house number spray painted on a what appears to be a wood board. Our goal is to design software that detects numbers from images like above.

1.3. Performance Metric

Human level accuracy on SVHN is 98% [5]. State-of-the-art models typically use several GPUs trained for several days on architectures as deep as eight to ten layers. At the onset, I set a computational budget of 15 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.

A commonly used metric for simultaneous SVHN digit recognition [6, 7] is per character accuracy (digit accuracy):

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

Digit accuracy measures how well a model is accurately classifying each digit. This project focuses on digit accuracy because of its interpretative 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 unfavorable property of sequence accuracy is its susceptibility to hold rare cases (such as longer sequences) on equal ground with common cases (shorter sequences). Class

imbalance 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 can 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.

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 prior to preprocessing show that images vary in size and color. A challenging aspect of this project is preparing the data for training.

2.1. Class Imbalance

Class imbalance is apparent in SVHN. A higher proportion of two sequence numbers exists in test compared to train and extra sets. Three or more sequence digits are rare in all cases. To handle these imbalances, I apply stratified sampling when creating train, test, and validation for experimentation.

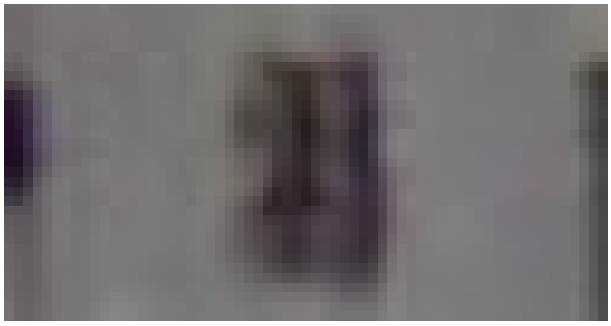

Figure 3: Example of a challenging image to classify, even for the human eye. Can you tell if what that number is? I would not be surprised if you guessed 23 or 25.

2.2. Visual Inspection

Visual inspection confirms a housing numbers can vary in location and rotation. For the most part, housing numbers are recognizable with the human eye. However, occasionally some images are hard to detect from a typical glance. [5] explains human eyesight has trouble with foreground, background confusion, and very low resolution/blurred when determining numbers in SVHN.

3. Algorithms and Techniques

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 them to encode certain properties into their design. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network [9].

3.1. Convolutional Architecture

My classification experiments use one primary architecture. Inspired by guidelines from [10, 11, 12], the structure is a straightforward blend of various techniques. The first two the convolutional layers contain filters size 3x3 and the last two contain filters size 5x5. *"SAME"* mode convolution is used for all the layers. Each convolution has Batch Normalization applied before a non-linear, rectified linear unit (Relu) activation function. Increases in the number of convolutional maps occur in sequence: 16, 32, 64, and 256. Dropout with probability 0.85 happens after the last convolutional layer. Next, a fully connected layer (a matrix multiplication followed by a bias offset) connects the previous layer with 512 hidden units. Afterward, six

independent softmax classifiers used after the fully connected layer. Five for predicting a digit at a particular position in the sequence. And one to predict the length of sequence for an image.

3.2. Loss and optimization

With Xavier initialized weights, I use cross-entropy for my loss function. For optimization, I use Adam – a state of the art algorithm developed for improving the performance of stochastic gradient descent. Adam is straightforward to implement, computationally efficient, and require little memory [9]. Experts often recommend it as their default choice for CNNs [9].

3.3. Benchmark

The model benchmark is 92% accuracy. Based on peers that have worked on similar projects [6, 7], typically have scores between of 90% and 93%. 92% is the average of between 90% and 93%. To the best of my knowledge, there is not an open source implementation project that has a digit accuracy beyond 94%.

4. Methodology

4.1. Getting the Images Ready

Concerned about overfitting, I adopt a slight variant of [2]'s approach who prepare their data with scale variability in mind. The goal is to images centered on digits by tightly cropping them to their bounding boxes. To do so; first, I find the small rectangular bounding box containing 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 re-size the images to 64x64 pixels.

4.2. Sampling

I use a validation set for hyper-parameter tuning. A 5% stratified random sample is taken from combined train and extra images for hyper-parameter tuning. The remaining combined training and extra images are for training. Test images are left untouched for the final evaluation of the model.

4.3. Preprocessing Pipeline

Confined to a MacBook Pro with 2.2 GHz Intel Core i7, 16 GB 1600 MHz DDR3, I use a customized input pipeline to read images with eight threads. Randomly shuffled batches are feed to the queue. Protocol buffers from TensorFlow's standard data format make this process easy. Protocol buffers isolate the model from disk latency and computationally expensive pre-processing [13].

Figure 4: Examples of images after bounding box expanision.

4.4. Adding Noise

The easiest and most common method to reduce overfitting on image data is to perform transformations on image pixels without changing their label [9]. In this stochastic pipeline, each image has a random chance of getting exposed to distorted saturation, brightness, hue, and contrast. Random mix/combinations distortions, add random noise and helps my model remedy overfitting during training. I take advantage of this later at test time by performing several forward passes with different random decisions and then averaging over them [9]. Last steps include pixel wide means subtraction to normalize the images before resizing them from 64x64x3 to 32x32x3.

4.5. Implementation

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 16 or 32. To overcome this, I serialized the data and used pre-fetch queues to feed training images into the model. After refining the pipeline, I trained my model with RGB colored images for ten epochs and obtained a solution of 93.4% on the test set. For training and validation, I get 94.7% and 97.2%.

4.6. Baseline

Moving forward 94.7% / 97.2% for my train/validation is my baseline. With scores significantly higher than my

original benchmark of 92%, I decide to focus my experiments on reducing overfitting. The process of improving the algorithm now boils down to narrowing the gap between my training and validation score (94.7% and 97.2%).

4.7. Dealing with Overfitting

The plan to improve the algorithm at this point is to add additional noise to the training data. Data augmentation is the most common method to reduce overfitting [9]. I do this because my initial steps with data augmentation were too conservative. Also, since I know I can reduce generalization error by collecting more training data [3] I stray away from engaging in tuning the number of hidden units or adding additional layers for model improvement.

4.8. Grayscale Images

I include in the pipeline a transformation step that converts images from RBG to grayscale. Grayscale transformation decreases the number of parameters for each image from 3,072 (32*32*3) to 1,024 (32*32*1). This idea for from [5] who found that converting to grayscale did not influence performance on SVHN.

4.9. Random Cropping

I also experiment with random cropping during data augmentation. This gives an indication how well my model can learn 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 pipeline, grayscale preprocessing turns out to be my best performer with – 95% accuracy. My model with RGB images achieves 93.6% accuracy. And when combined with random cropping (RC) both RGB and grayscale models have 92% accuracy. These results tell me that small perturbation (changes) during training decrease test performance to an extent, however not below my original benchmark.

5.1. Model Evaluation and Validation

The effects of grayscale preprocessing exceed my expectation. Experiments with random crops have lower performance at 92%. I also notice a slight drop-off will occur when numbers in images deviate in location or rotation. For a better understanding of how my model is making predictions, I use two plots: 1) random predictions and 2) errors made during predictions. These scenarios allow me to compare predictions to ground truth through visual inspection.

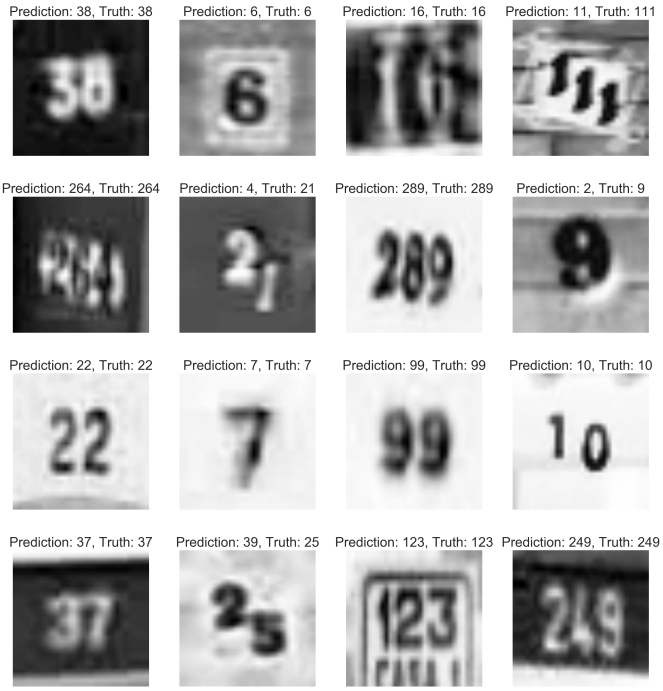

Figure 5: Predictions made by the model. You can see from the image (3 right, 1 up), the model can make correct predictions in the presence of other text. And in the presence of tricky views, such as image (2 right, 2 up) you can understand why a 4 is predicted instead of 21 (true label) as the digits in the picture resembles a 4.

Model	Training	Validation	Test
RGB (initial solution)	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 2: Results. Grayscale preprocessing really stands out in terms of performance beating out my baseline and benchmark.

5.2. Learning Rate

To evaluate the trade off between my model's loss function, I plot the training loss over 15 epochs at learning rates: 1e-3, 1e-4, and 1e-5 to compare loss over time.

The graph above shows that 1e-5 does not converge as fast as the other learning rates while plateau very early. 1e-3 (a higher learning rate) shows good improvement over time, but plateaus after 28k steps. Of the three, 1e-4 has the most promise with an exponential shape and low bias.

5.3. Train/Val accuracy and Overfitting

To get insights into the amount of overfitting in my model, I plot training and validation accuracy over time. Since the gap between the training and validation accuracy is

Figure 6: Error analysis - Errors made by the model. The model struggles with numbers that are cut off, rotation variances, and light colors that match the background.

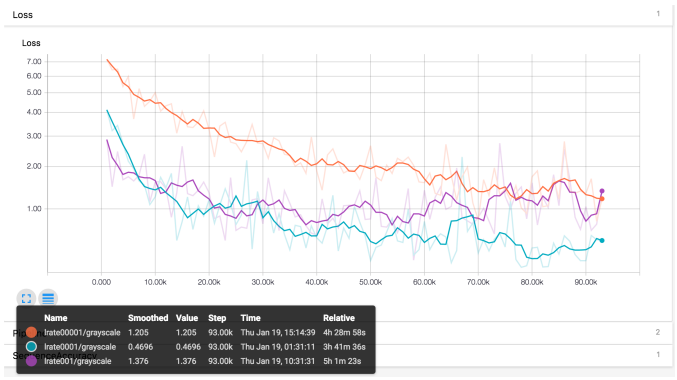

Figure 7: Analyzing Loss with Tensorboard. An initial learning rate of 1e-4 has a nice exponential shape and performs best out of the three.

minuscule, I am confident my data augmentation steps earlier helped me control for overfitting.

5.4. Observing the model in action (Justification)

Based on up close visual inspects, incorrect predictions from my model are within reason. They 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. Based on initial benchmarks, I accomplish our goal by scoring 95%, well over my 92% benchmark. 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). My model architecture demonstrated

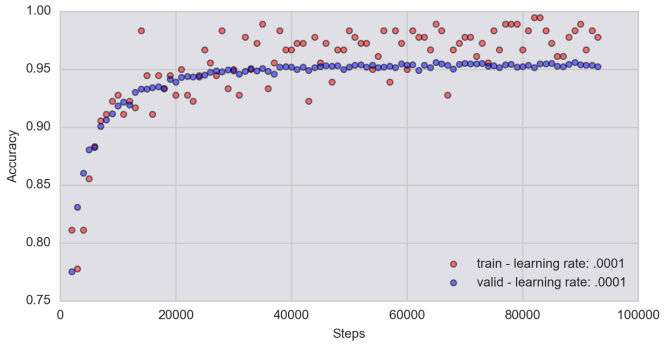

Figure 8: Analyzing overfitting with train/val accuracy. Overfitting is not apperant in the plot above, which indicates model refinements have remdied the problem of overfitting.

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.

6. Reflection

Recognizing numbers of arbitrary length in natural images is a hard problem. However, I believe it's very solvable. This project demonstrates how one can go from raw images to simultaneously digit recognition. A robust pipeline can make training a deep CNN possible on a single laptop; data augmentation can limit overfitting, and a good learning rate can make a big difference in overall performance.

The following is key milestones from this project:

- Downloading a large real world dataset of images
- Preparing it for end-to-end deep learning
- Refining the pipeline to deal with CPU bottlenecks
- Replicating preprocessing steps used by [2]
- Implementing a custom CNN architecture
- Improving upon my initial results
- Evaluating the importance of the learning rate

A difficult challenging aspect of my project was getting the images into the pipeline. A second problem was training the model without a GPU, which meant training took a long time (3-5 hours). I believe both of these problems are solvable with improvements in software improvements and hardware upgrades.

6.1. Improvement

The following would be key milestones for this project if I were to do it again:

- Downloading a large real world dataset of images
- **Preprocessing as a learnable part of the network**
- Implementing a custom CNN architecture
- Improving upon my initial results
- Evaluating the importance of the learning rate

My project focused on *tightly/reasonably cropped images.* And based on experiments, I expect a slight decrease in performance on free form unbounded natural images because of limitations related to bounding box cropping.

6.2. Next Steps

A natural extension of my project is making preprocessing a learnable part of the network by integrating localization techniques for a drop in replacement for bounding box cropping. In other words, I would replace elements of data preprocessing, and argumentation for localization techniques such as Bounding Box regression, Recurrent Convolutional Neural Networks, or Spatial Transformer Networks.

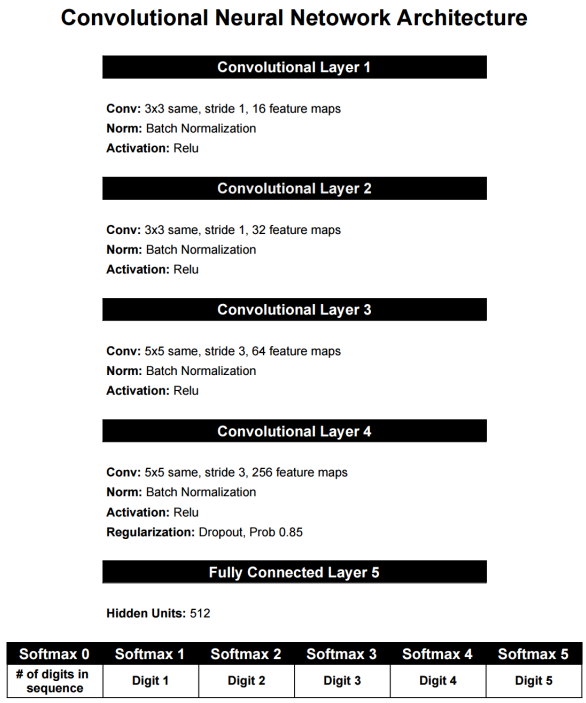

Figure 9: Detailed breakdown of CNN architecture.

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

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

[12] 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.

[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.

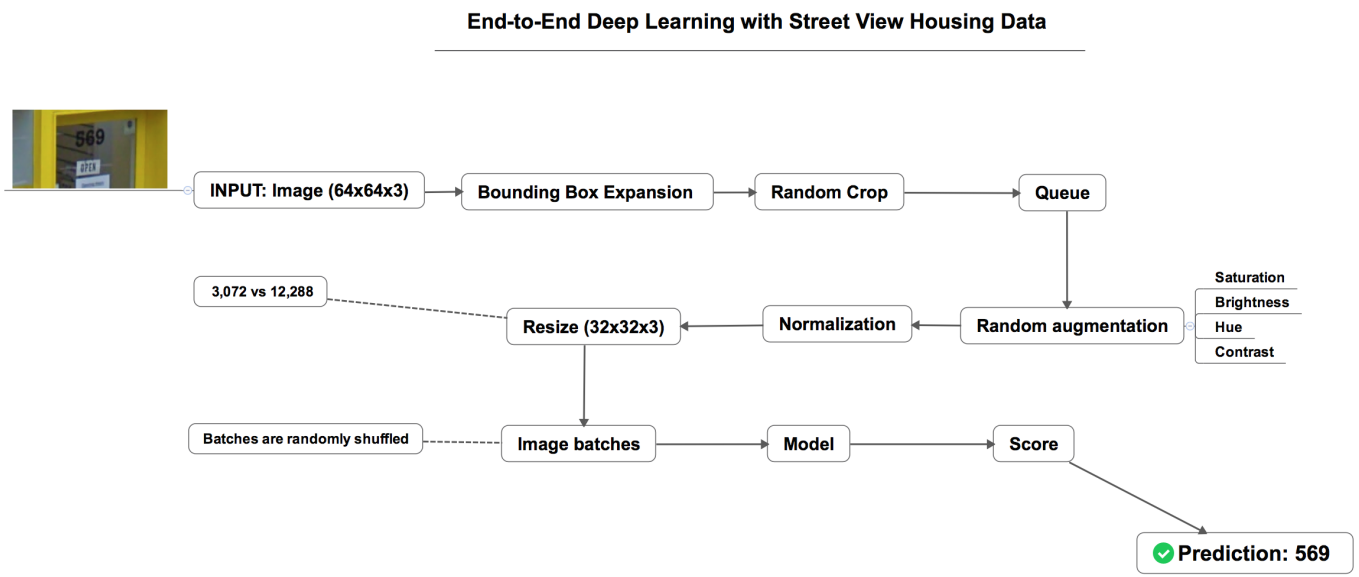

Figure 10: High level overview of my end to end project.