Classifying Street View House Numbers with Deep Learning

Eric Conlon

2016-11-9

Contents

1 Introduction

Character recognition is a classic computer vision problem with many interesting variants. Even restricting ourselves to 10-digit classification still leaves much room for uncertainty and creativity. The Street View House Numbers dataset (SVHN) [Net+11] poses a such a classification problem that is generally considered to be tougher than the common MNIST dataset [LCB98].

In this report I explain some simple approaches to classification of both datasets with both conventional and deep learning methods. The conventional baseline here is a Support Vector Classifier using Histogram of Oriented Gradients features, and the deep learning challenger is a Convolutional Neural Network. These solutions may not rank among the state of the art, but it is my hope that the details of end-to-end architecture and practical engineering considerations will prove to be interesting. I especially want to focus on how much you can learn in a single sitting at your computer, rather than what you can learn over several days.

2 Datasets

MNIST is a classic labeled dataset consisting of single handwritten digits (0-9) that have been extensively processed. Each digit is centered alone within a 28x28 grid, and all pixel intensities are between to 0 (background) and 255 (foreground). These images do not require common preprocessing techniques such as rescaling, contrast enhancement, or whitening to be immediately useful. There are around 60,000 training examples and 10,000 test examples.

SVHN is a labeled dataset that comes from images of house numbers collected by Google's Street View project. The digits have a wide variety of foreground and background colors, textures, noise, distractions, orientations, and scales. There are two formats provided: the first includes raw images with sets of bounding rectangles, and the second includes a more MNIST-like centered digits. The models here consume only the second format, but I will say more about this classifiers applicability to the first. Overall, this dataset contains almost 8 times more examples than MNIST.

I went with an approximate 60%-20%-20% training-validation-testing split for both datasets (of various total size). For SVHN, I included the "easier" and much larger `extra` dataset in the training and validation sets.

3 Design

The application follows a standard machine learning pipeline model. See Figure 1 for a visualization of the major stages of this this pipeline. I tried to obey a few guiding principles in its implementation:

It must possible train, test, explore, and evaluate independently. Too often processing-intensive applications are tightly coupled and require recomputation of intermediate stages. Here, results from most steps are serialized to disk between stages and tagged with their provenance in order to speed up subsequent re-executions of the pipeline from any given stage. Most stages are also executable independently directly from the command line with the desired arguments, and higher-level variants can also be executed. For example, one can train a model, then later deserialize the trained model and run predictions on another dataset; or one can drive parameter selection over fixed training and validation datasets.

Model selection should be an automated part of the training process. The Tensorflow [Mar+15] compatibility layer with SciKit Learn [Ped+11] presented all sorts of inconsistencies in implementation and documentation, so I had to put together basic parameter search agnostic of either. It's worth noting that many ML engineers serialize a lot of their model parameters and configuration into a rich textual config format (e.g. in the artifacts for [KSH12]). For simplicity's sake I did not, but I will the next time around. This might have completed the loop where I had to manually save the best parameters.

Common preprocessing, metrics, and visualizations should apply to any model and dataset. If I were to throw CIFAR-10 [KH09] at this project, not much would have to change! I would still be able to preprocess, train, test, evaluate, and explore models and predictions the same as MNIST and SVHN. Though the bulk of the application is in a large Python module with unit tests, I also incorporated Jupyter [PG07] notebooks into the workflow to explore results visually.

4 Preprocessing and Augmentation

Minimal preprocessing was required for MNIST. Digits came centered and isolated in images, and all pixel values were quantized to 0 or 1. SVHN, on the other hand, required extensive work. The processing steps for each cropped digit image were

1. Convert to grayscale (to reduce dimensionality)

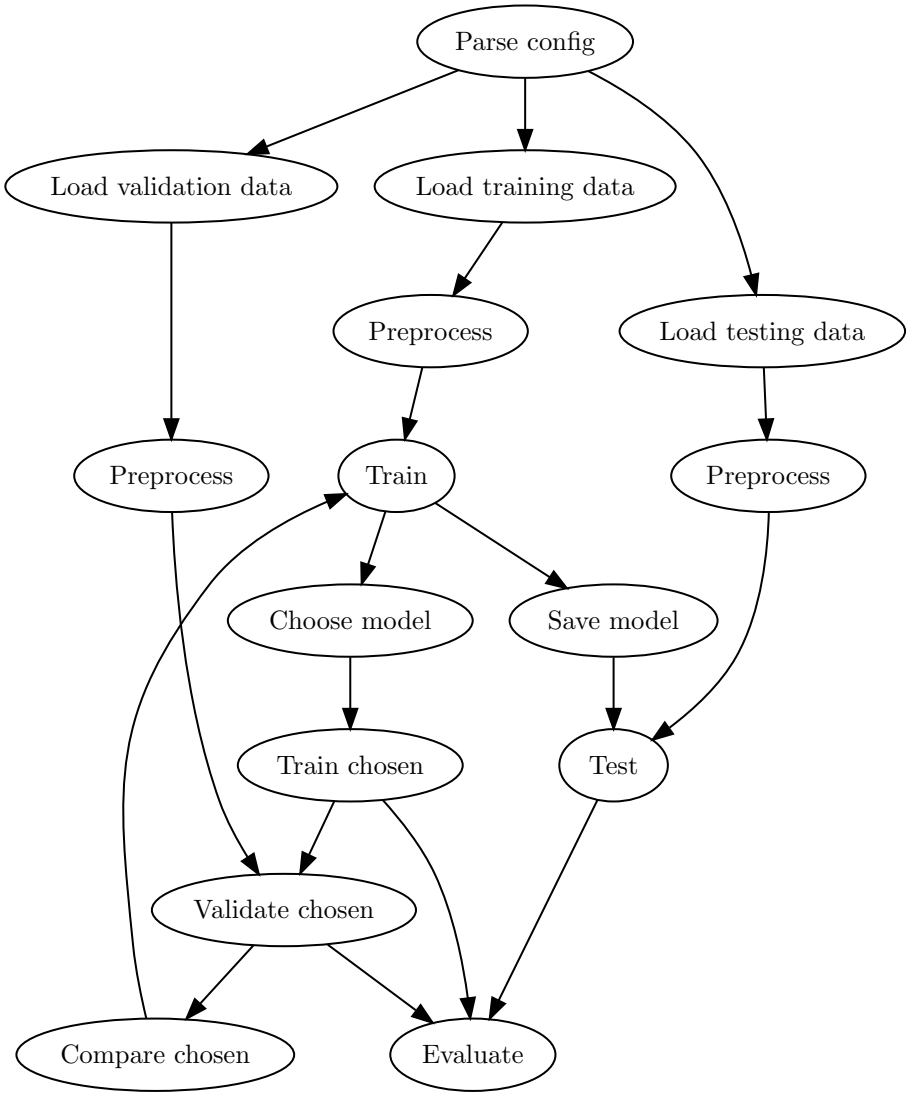

Figure 1: Pipeline stages

2. Crop on the left and right (4 pixels) to reduce distraction
3. Center on the mean
4. Perform Global Contrast Normalization (variance scaling over whole image)
5. Perform Local Contrast Normalization (variance scaling over patches)

Jarrett et al [J+09] describe the Local Contrast Normalization step, which is more or less a thresholded, Gaussian-weighted variance scaling around each pixel. The results of this preprocessing can be seen in figures to follow. Numpy [VCV11], SciPy [J+01], and SciKit Image [Van+14] were quite helpful in these processing operations.

For smaller datasets, augmentation with small coordinate transformations (scaling, translation, and rotation) of data in the preprocessed training set helped to reduce overfitting and improve accuracy. This was useful for MNIST in particular since the digits were so well cropped and centered that these perturbations looked quite reasonable. The SVHN dataset was not as well cropped, and moreover there was plenty of training data, so it did not benefit from this augmentation.

Figure 2: MNIST sample

Figure 3: MNIST augmented

Figure 4: SVHN sample

Figure 5: SVHN processed

5 Conventional Learning

Conventional learning requires that we extract relevant features from our dataset before learning. Sometimes this extraction can be minimal preprocessing (one-hotting, projecting, or rescaling) and sometimes it can be more extensive, especially in higher dimensional datasets. In the case of MNIST and SVHN, we are looking at 784- and 3072-dimensional input spaces, respectively. Even if we were to amass enough data to make learning that size space tractable, we might still suspect that there were more relevant lower-dimensional representations.

One such representation uses Histogram of Oriented Gradients (HOG) as the feature space [Net+11]. HOG first calculates approximate gradients in both directions for each pixel in the image, then yields a gradient that is the weighted mode of those gradients in a neighborhood around each pixel. The intuition is that changes in value between pixels are more relevant than the pixel values themselves, and that the direction of these change is more useful still. (Note that aside from RGB to grayscale conversion, the preprocessing mentioned above was not performed here – local and global mean and variance scaling are essentially irrelevant when taking local derivatives.)

Using a relatively untuned Support Vector Classifier with RBF kernel over HOG features from MNIST images yielded 73.4% accuracy on the test set. The same method over a smaller subset of SVHN yielded only 35.3% accuracy! With the SciKit Learn implementation I couldn't finish a run over the full dataset in any reasonable amount of time, so don't take this result too seriously.

Figure 6: MNIST HOG

Figure 7: SVHN HOG

6 Deep Learning

Deep learning with neural networks, in contrast to conventional learning, requires no feature extraction step. One might think this would make things easy, but there are quite a few knobs to turn! I used a variant called a Convolutional Neural Network which uses multiple layers and channels of local convolutions, non-linear activations, and pooling to approximate patch-based features, followed by multiple fully-connected layers to classify based on those features.

With extensive parameter selection, a setup with 2 convolution layers (each with width 5, depth 64) and 2 fully-connected layers (size 1024; size 10) seemed to work best. I can't stress enough how fragile and difficult this process was – it took a few hundred runs on big and small datasets to come up with something that works just ok. This particular network configuration seemed to be able to memorize a small training set just fine, so it seemed appropriate to try to scale up with these parameters. However, there are a lot of blanks you have to fill in even after you've gotten that far.

It only complicates matters that training a neural network is not a speedy process. On real-world datasets it can take days or weeks to train with specialized frameworks and appropriate hardware. Many competitive solutions to common classifications problems process their entire dataset more than 10 or 20 times. I did all this work on a 5 year old laptop, so I had to be smart about trimming work where I could!

In no particular order, here are some relevant pieces of advice I wish I could have given my past self to maximize accuracy while minimize training time:

Use something better than Gradient Descent. Tensorflow comes with a few sophisticated optimizers that generally work better than simple Gradient Descent. Just switching to the Adam algorithm made a big difference in speed and accuracy.

Decrease your learning rate over time. Tensorflow comes with primitives for exponential learning rate decrease – use them! Decay rate and factor are two additional parameters to tune but learning is generally pretty robust within an order of magnitude of optimal values.

Preprocess carefully. Mean and variance scaling make a big difference

(on the order of tens of percent of accuracy).

Initialize carefully. Xavier initialization is terrible with ReLU, and should not be recommended any more. Simple random normal initialization is better.

Try inverting your images. I didn't find any references for this, but including inverted copies of images in the same batch increased accuracy quite a bit. The intuition here is that we are trying to learn how to see dark digits on light backgrounds and vice versa, so we want to train our network to classify based on contrast changes of both kinds (increasing or decreasing intensity). Feeding an image and its inverse in the same batch attempts to train these features equally each optimization step.

Regularize to learn. Regularization is a simple and effective way to prevent overfitting. Do it.

Train with equal representation. Choosing the same number of examples for each class in each batch reduced bias a lot. It's also relevant to remember the exploration/exploitation tradeoff: Alternating sequential and random batches is a good way to ensure you see the whole dataset, reinforce things you've seen, and insulate you from bad dataset orderings.

Use sampling for prediction. If you have a large validation set, it's going to be prohibitively slow to calculate validation accuracy each time after some number of optimization steps. Predicting on a random sample from the validation set (especially one with equal representation) is good enough to estimate accuracy.

Know when to quit. It's tough to know a priori how much data you need to see before you can pass judgement on a set of parameters. Instead I set a hard cap on the number of examples I was willing to train on, and also set limit on the number of prediction steps for which the accuracy was not improving. (That is to say I stopped training when it looked like it stopped learning.) For unit testing I trained on smaller datasets and stopped when an artificial accuracy limit was hit.

7 Voting

There is quite a bit of randomness in both the theory and implementation of the deep learning solution. Practically, I would see results differing by a few percentage points over the same data with different initialization conditions and seeds. A simple technique to reduce this variance was to introduce voting between multiple models: I trained a straight soft-voting ensemble of 10 classifiers (simply averaging the distributions of all ten, then argmaxing to get the

predicted class) and saw a final test accuracy several percent higher than the average individual validation accuracy.

8 Results

For every model and dataset, the following are calculated and written to disk:

- Accuracy
- Per-class precision, recall, and F1
- Gold and predicted class distributions
- Confusion matrix
- Most and least certain examples (by entropy)
- Convolution weights
- Convolution activations for selected examples
- Learning curve (loss and accuracy over time)

The most relevant topline metric is probably accuracy on test datasets (shown in Figure 8). Take this data with a grain of salt – as mentioned above, I've traded some accuracy for speed! For an idea of the relative complexity of some of these, it takes about 5 minutes to train a single CNN on MNIST to the given accuracy, and it takes around 10 times that amount of time to train a single CNN on SVHN.

Model	MNIST	SVHN
SVC	73.4%	35.3%
CNN	90.7%	66.7%
Vote	93.5%	79.3%

Figure 8: Accuracy by model and dataset

9 Conclusion

In short: Deep learning is not a panacea!

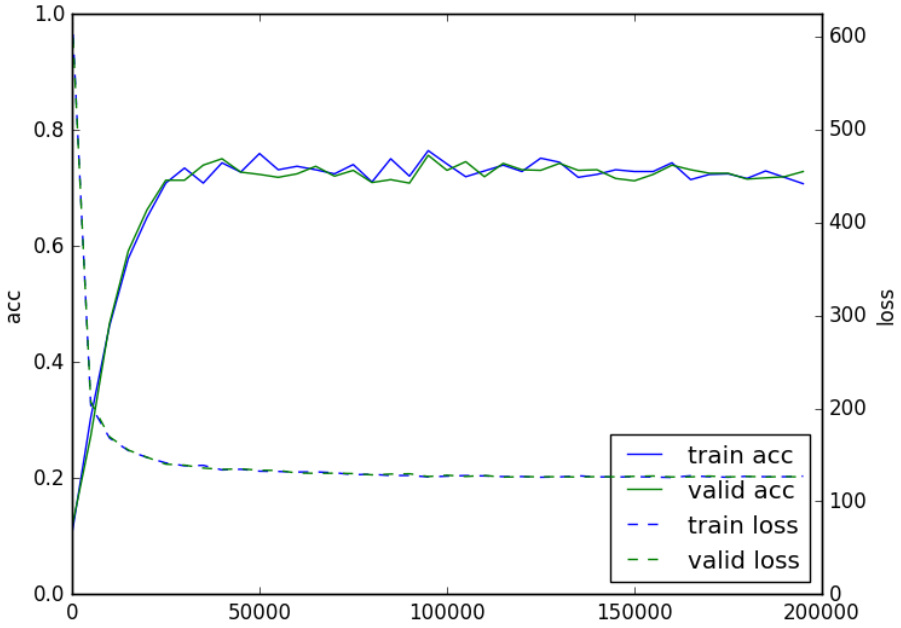

Figure 9: SVHN CNN learning curve

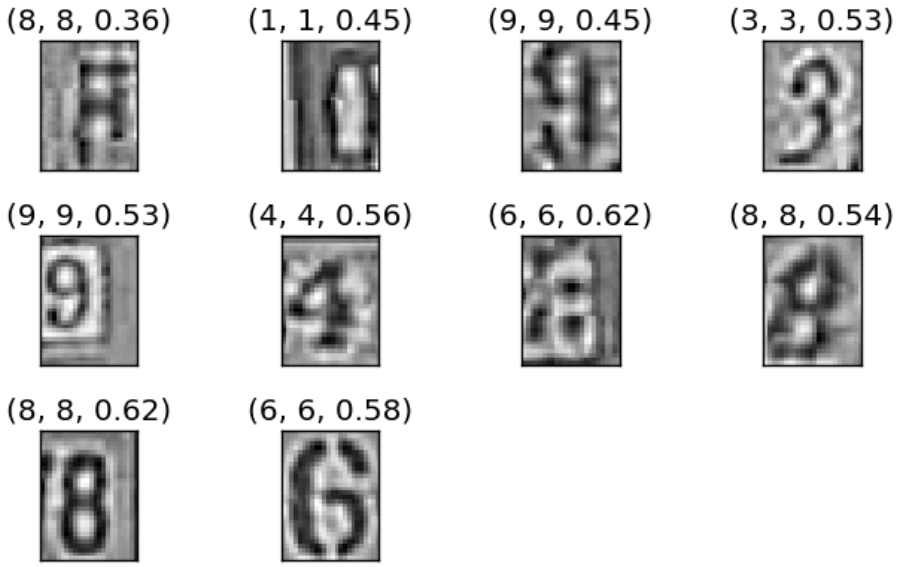

Figure 10: SVHN CNN least certain correct preditions

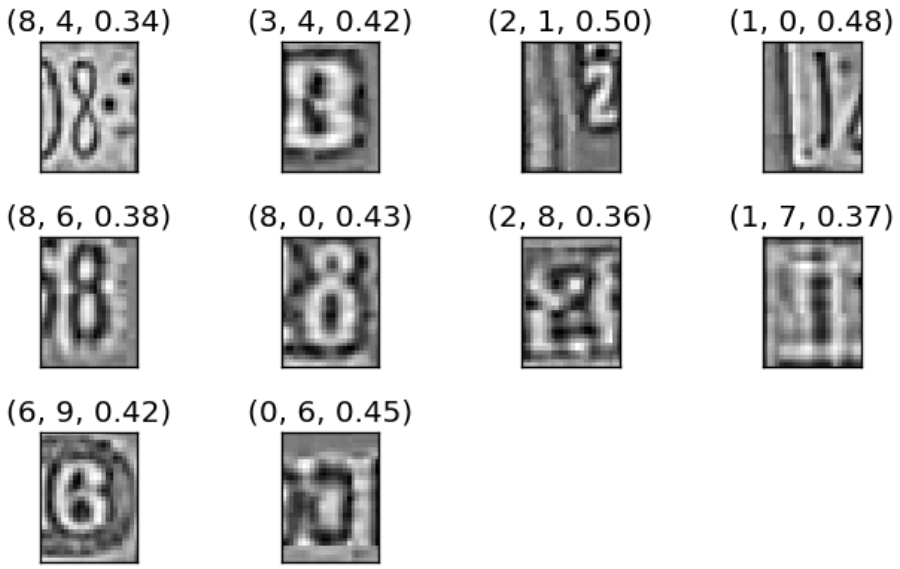

Figure 11: SVHN CNN least certain incorrect preditions

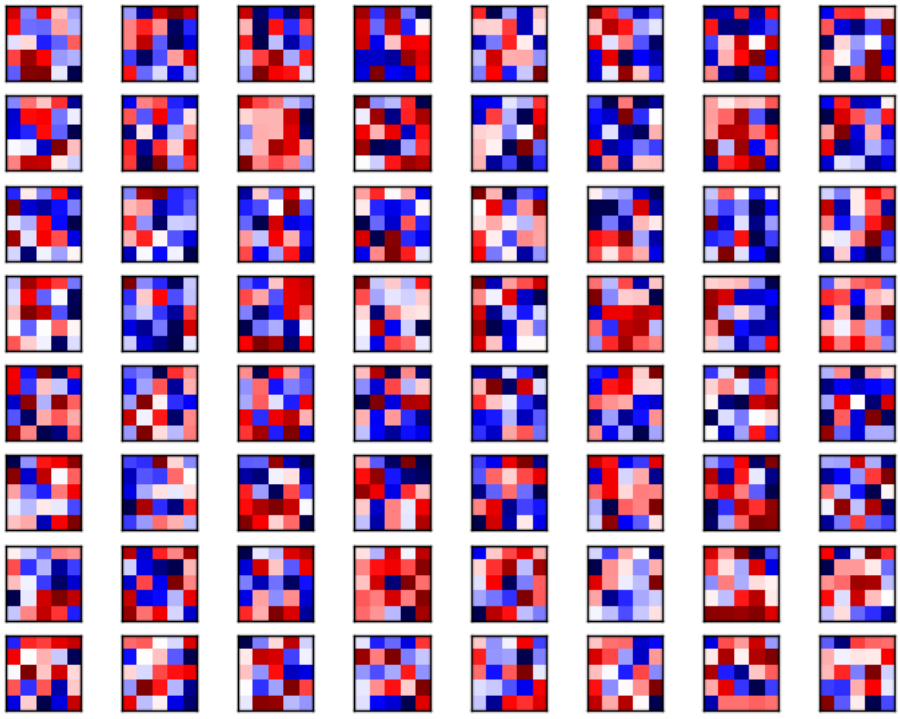

Figure 12: SVHN CNN first convolution layer weights

References

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

[J+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2016-11-11]. 2001–. URL: `http://www.scipy.org/`.

[PG07] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: `10.1109/MCSE.2007.53`. URL: `http://ipython.org`.

[J+09] Kevin Jarrett, Koray Kavukcuoglu, Yann Lecun, et al. "What is the best multi-stage architecture for object recognition?" In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE. 2009, pp. 2146–2153.

[KH09] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images". In: (2009).

[Net+11] Yuval Netzer et al. "Reading digits in natural images with unsupervised feature learning". In: (2011).

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[VCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[Van+14] Stefan Van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (2014), e453.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.