An Exercise in Hyperparameter Tuning and Domain-Specific Noise Reduction

Trevor McInroe

Northwestern University

Abstract

The success of the training process of neural networks is strongly guided by hyperparameters, which are parameters chosen by the researcher and not explicitly learned by the model itself. As such, it is important that researchers understand the implications of hyperparameter choices. In addition, nerual networks are notorious for long training times relative to models that can be estimated analytically. In this exercise, we explore the epoch-by-epoch progress of a dense neural network trained on the MNIST dataset. We record high-level metrics such as classification accuracy and elapsed training time as well as several more granular classification metrics across various combinations of hyperparameter choices. Specifically, we test varying levels of learning rate, mini-batch size, and number of nodes in the hidden layer. Once "best performing" hyperparameters have been chosen, we compare the performance of a network trained on the base dataset against a network trained on a modified version of the dataset that employs a domain-specific heuristic to eliminate "noise". From these experiments, we find that hyperparameter choices prove critical in network performance and that heuristics on data cleaning can be useful in terms of reducing training times.

An Exercise in Hyperparameter Tuning and Domain-Specific Noise Reduction

Introduction

The process of training a neural network requires the researcher to set parameters that guide the learning process that are not explicitly learned themselves, called *hyperparameters*. Some examples of hyperparameters are the learning rate, the number of weights in the hidden layers of the network, and the mini-batch size. Selecting intelligent values for these hyperparameters is of great consequence, as they directly affect the learning process of the network. There are two main ways to intelligently select hyperparameters: (a) manually sampling from a distribution of valid values or (b) using a separate learning algorithm on the hyperparameters themselves. While work in (b) by researchers such as Safarik et al. (2018) and Dubrawski (1997) have showed significant success, we will only explore (a) in this exercise.

In addition to the tuning of hyperparameters, researchers should also be aware of the overall compute time required to train their networks to completion. Due to their large number of learnable parameters and the iterative nature of their learning, the training process of neural networks is notorious for taking a significant amount of time, especially when compared to models that can be estimated analytically.

Through this study, we hope to gain some intuition around the effects of various architecture and hyperparameter settings as well as the efficacy of domain-specific data-reduction techniques to speed up training time.

Methods and Literature Review

In this study, we examine the process of hyperparameter tuning as well as a domain-specific data-denoising process. Specifically, we will use a fully-connected neural network with a single hidden layer trained on the famous MNIST dataset. In their study of gradient-based learning, Lecun et al. (1998) create the MNIST dataset as an extension of the original "National Institute of Standards and Technology" dataset. MNIST contains grayscale 28x28 images that depict handwritten digits of numbers ranging from 0 to 9. The dataset contains 60,000 training images and 10,000 test

images. For a distribution of classes for both datasets, see table A1.

To make our data "fit" into our models, we need to "flatten" our 3D image tensors, $x_t \in \mathbb{R}^{28x28x1}$, into a 1D tensor, $x_t \in \mathbb{R}^{784}$. Our network's weights will be initalized as an orthogonal projection via QR decompostion of a matrix with weights drawn from $\sim N(0, 1)$, which has been shown to have benefits over other random initalization regimes[1] (Saxe et al., 2013). This results in $\boldsymbol{\theta}^{(\ell)}_{*,i} \perp \boldsymbol{\theta}^{(\ell)}_{*,j}$ for all column pairs, (i, j), within a given layer, ℓ, in our network's weight matrices, $\boldsymbol{\theta}$. For the act of classification, we use the *softmax* function. This will result in the network's output being the probability of each example in the mini-batch belonging to each class. As Goodfellow et al. (2016) point out, the base version of the softmax function is prone to both numerical **underflow** and **overflow**. In order to make the function more stable, we simply subtract the maximum value of the i^{th} row in the input matrix from the i^{th} row of the input matrix. Simply adding or subtracting a scalar from the input into the softmax function does not change the output of the softmax function (Goodfellow et al., 2016).

$$\boldsymbol{z} = \boldsymbol{x} - max_i(\boldsymbol{x}_i)$$
$$softmax(\boldsymbol{z})_i = \frac{e^{\boldsymbol{z}_i}}{\sum_{j=1}^{C} e^{z_j}}$$

where $\boldsymbol{x}$ is the output matrix from the layer directly preceeding the softmax function, C is the number of classes, and e is the constant Euler's e. As is a common complement, we use *cross entropy* as our loss function:

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} log \hat{y}_{ij}$$

where y_{ij} is 1 if ith ground-truth label is of class j (otherwise 0), $\hat{y}_{ij}$ is the predicted probability that the ith example is from class j, N is the number of examples, and log is the natural logarithm. To optimize our networks, we will use the popular adaptive moment estimation (Adam) algorithm from Kingma and Ba (2014). For a mathematical description of the gradient calculation and weight updating process, see equation 1 and equation 2, respectively, in the Appendix.

[1] All code can be found here: https://github.com/trevormcinroe/msds458/tree/master/week2. The code for this initalization schema can be found within the NN._init_saxe() method in the nn.py file

The first three experiments will be exercises in exploring hyperparameter choices. Specifically we will study the effects of various learning rates, mini-batch sizes, and the number of nodes in the hidden layer in experiments 1, 2, and 3, respectively. To test the varying hyperparameter levels, we will train the model on our training data for 100 epochs five separate times, using randomly assigned seeds. After each epoch, a run over the entirety of the testing data will be made. Over the 100 epochs, we will record several metrics on the model. First, we will look at the overall classification accuracy, as it is the main metric of concern. We will show the average accuracy of the five passes over each epoch as well as the minimum and maximum to observe the consistency of the hyperparameters. Second, we will observe the average amount of time it took to train a single model over 100 epochs. In addition, for the experiment on the number of hidden nodes, we will record the number of learnable parameters in the models. While they are not being explicitly observed, the hyperparameters will be frozen as learning rate (0.01), batch size (64), and hidden units (100).

Using the results of the first three experiments, a single model will be trained using the best-performing hyperparameters. We define "best performing" as the balance between network classification performance (higher is better), variance in performance across runs (lower is better), and training time (lower is better). This model will be examined thoroughly in experiments 4 and 5. Specifically, we will look into several classification metrics such as Precision, Recall, and F1-Score.

The final experiment will be similar to experiment 4. However, experiment 5 will attempt to filter some of the "noise" of the input data using manually-discovered heuristics. Observing the images that make up the MNIST dataset (example in Figure A2) reveals that most samples have a significant area of "white-space". We postulate that this consistent white-space is mostly noise and, thus, adds no informational value from which the network could learn. We will then train a model on the "denoised" data and compare its performance to the model from experiment 4 across a variety of metrics. We note that this identification of "noise" is *domain-specific*. That is, applies specifically and only to this dataset and may not transfer well to other

datasets.

Results

Experiment 1

Observing table A3 and figure A4 , we note four main results. First, we note a strong divergence and high variance in performance in the networks that were trained with the highest learning rate (0.1), indicated by the slight downward slope in the accuracy curve for both the training and testing datasets. We note that this behavior coincides with the behavior examined in depth by by Philipp et al. (2017) and is deemed the "exploding gradient problem". Second, the lowest learning rate takes significantly longer to converge than the other learning rates. Thirdly, we observe strong diminishing marginal returns in convergence speeds as learning rate increases. Fourth, the amount of time to train 100 epochs did not vary significantly as the learning rate changed.

Experiment 2

Observing table A5 and figure A6, we note two main results. First, a larger mini-batch size leads to significant decreases in the amount of time it takes to train each epoch. This, of course, is assuming that memory overhead is of no consequence, which is true for our experiment as our data is quite small. Second, larger batch sizes tend to lead to faster convergence and better overall performance on both the training and testing datasets. However, the jumps in improvement shrink, suggesting that, perhaps, too large of a batch size may be detrimental.

Experiment 3

Based on the results in table A7 and figure A8, we note several findings. First, based on the observations in both the training and testing runs, a larger number of hidden units results in greater representational capacity, as denoted by the notable increases in accuracy. However, we also note that there are diminishing marginal returns to doing so, as the level shift in accuracy grows smaller at each jump in the

number of hidden units. Second, we note the significantly smaller variability in runs with a higher number of hidden units. Thirdly, we note that the time to train each epoch increases with the number of hidden nodes, which is logical due to the increased amount of computations that comes with a larger network.

Experiment 4

The best performing hyperparameters from the previous three experiments are as follows: learning rate: 0.01, batch size: 256, and number of hidden units: 100. Using this architecture created a neural network with 79,510 learnable parameters. The results of training this network can be seen in figure A9 (blue line, "raw data") and, in more detail, table A10 ("raw data").

Experiment 5

Taking an average of the i^{th} pixel in each image in our training set reveals that there are 67 pixels that are always 0 (pure white). With this list of indexes, we drop each of those pixels from both the training and testing set. This results in ∼8.4% reduction in the size of the input vector for each training example which results in the network having 72,810 learnable parameters or an ∼8.5% reduction in model size. As the results from experiment 3 suggest, this smaller number of learnable parameters should result in faster training time.

Observing the accuracy curves in figure A9, we see that the smaller network trained on the "denoised" data performed nearly identically across both the training and testing sets. In addition, observing the more detailed performance metrics in table A10 confirms these results. These results suggest that employing domain-specific heuristics for data cleaning can be well-worth the effort.

Conclusion

In this study we examined the process of hyperparameter tuning for a neural network on an image-data classification problem. We proved that choices around hyperparameters affect both the performance of the model as well as the compute time

required to train them. In addition, we explored a dataset-specific heuristic for removing noise that resulted in a smaller, therefore faster to train, model that performed just as well as a model on the raw data. Both of these insights suggest that effort spent both on finding heuristics to clean a given dataset and tuning network hyperparameters is effort well invested.

Future Work

In this study we did not perform an exhaustive search of the hyperparameter space. That is, testing all possible combinations in our search space. Given the rate at which the conbinatorics grow, this would have significantly increased the compute required for the experiments. Due to this, there must be a certain point at which applying an independent learning algorithm to the hyperparameters themselves would be more efficient. At what point does this occur and which learning algorithms are the most efficient? Finally, the results from experiment 2 suggested that choosing too large of a batch size could be detrimental to the performance of the network. Is this true and, if so, is this phenomenon dataset- or model-dependent?

References

Dubrawski, A. (1997). Stochastic validation for automated tuning of neural network's hyper-parameters. *Robotics and Autonomous Systems*, *21*(1), 83–93. https://doi.org/10.1016/S0921-8890(97)00008-0

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep learning*. MIT Press.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization, arXiv:1412.6980.

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Philipp, G., Song, D. & Carbonell, J. G. (2017). The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions, arXiv:1712.05577.

Safarik, J., Jalowiczor, J., Gresak, E. & Rozhon, J. (2018). Genetic algorithm for automatic tuning of neural network hyperparameters, SPIE. https://doi.org/10.1117/12.2304955

Saxe, A. M., McClelland, J. L. & Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, arXiv:1312.6120.

Appendix

Backpropagation and Weight Updating

For our network architecture that uses cross-entropy loss, we first calculate the gradient of the loss with respect to the network's weights as:

$$\begin{aligned}
\hat{y} &= softmax(ReLU(XW^{(0)} + b^{(0)})W^{(1)} + b^{(1)}) \\
z^{(\ell)} &= a^{(\ell-1)}W^{(\ell)} + b^{(\ell)} \\
a^{(\ell)} &= g^{(\ell)}(z^{(\ell)}) \\
\delta^{(output)} &= \nabla_{output}\mathcal{L}(\hat{y}, y) = \hat{y} - y \\
\delta^{(\ell)} &= (\delta^{(\ell+1)}W^{(\ell+1)\intercal}) \odot g'^{(\ell)}(z^{(\ell)}) \\
\nabla_{b^{(\ell)}} &= \delta^{(\ell)} \\
\nabla_{W^{(\ell)}} &= a^{(\ell-1)\intercal}\delta^{(\ell)}
\end{aligned} \tag{1}$$

where $g^{(\ell)}$ is the activation function of layer ℓ, $a^{(\ell-1)}$ for the input layer is our mini-batch of data X, and $\odot$ denotes the elementwise product. This gradient is packed into a collection $\mathcal{G}$ which are then passed to Adam as:

$$\begin{aligned}
m_t^{(p,\ell)} &\leftarrow \beta_m m_{t-1}^{(p,\ell)} + (1-\beta_m)\mathcal{G}_t^{(p,\ell)} \\
v_t^{(p,\ell)} &\leftarrow \beta_v v_{t-1}^{(p,\ell)} + (1-\beta_v)\mathcal{G}_t^{2,(p,\ell)} \\
\hat{m}_t^{(p,\ell)} &\leftarrow m_t^{(p,\ell)}/(1-\beta_m^t) \\
\hat{v}_t^{(p,\ell)} &\leftarrow v_t^{(p,\ell)}/(1-\beta_v^t) \\
\boldsymbol{\theta}_t^{(p,\ell)} &\leftarrow \boldsymbol{\theta}_{t-1}^{(p,\ell)} - \alpha\frac{\hat{m}_t^{(p,\ell)}}{\sqrt{\hat{v}_t^{(p,\ell)}} + \epsilon}
\end{aligned} \tag{2}$$

where α is our learning rate, ϵ is a small stability value, β_v is our velocity coefficient, β_m is our momentum coefficient, $m_t^{(p,\ell)}$ is our momentum matrix, $v_t^{(p,\ell)}$ is our velocity matrix, $\boldsymbol{\theta}_t^{(p,\ell)}$ are the network's learnable parameters, and $\mathcal{G}_t^{(p,\ell)}$ is the gradient matrix for parameter p (weights or biases) for layer ℓ, at step t.

Tables and Figures

	0	1	2	3	4	5	6	7	8	9
Train	9.8%	11.2%	9.9%	10.2%	9.7%	9.0%	9.9%	10.4%	9.8%	9.9%
Test	9.8%	11.4%	10.3%	10.1%	9.8%	8.9%	9.6%	10.3%	9.7%	10.1%

Figure A1. Distribution of classes rounded to one decimal place

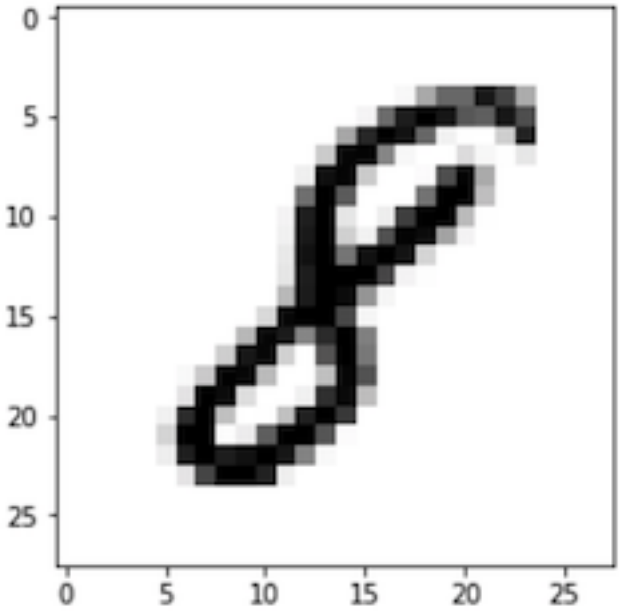

Figure A2. Example of an "8" from the MNIST dataset

Learning rate	Seeds	Training time (s)
0.00001	706, 9767, 8192, 5374, 6530	394
0.0001	3423, 6488, 5346, 1379, 1216	391
0.001	5893, 7380, 512, 8782, 3610	389
0.01	3694, 1406, 5642, 5188, 9519	335
0.1	400, 8778, 7814, 7766, 515	367

Figure A3. Learning rates, seeds, and training time per 100 epochs for Experiment 1

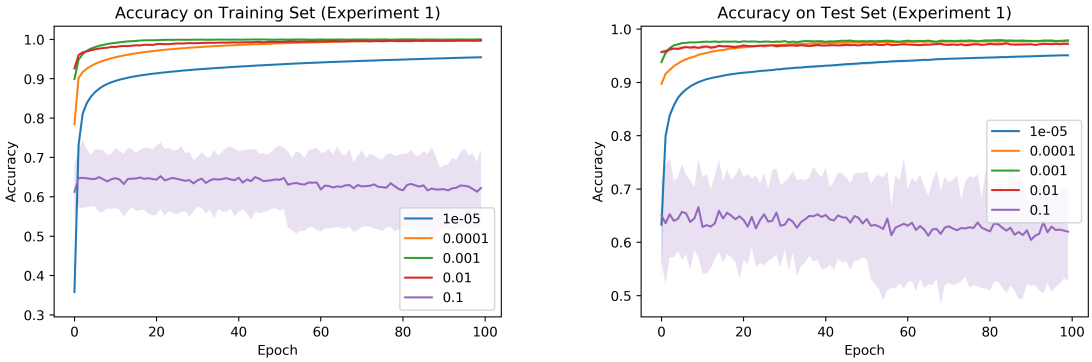

Figure A4. Accuracy of training (*left*) and testing (*right*) for Experiment 1

Batch size	Seeds	Training time (s)
16	7073, 2702, 3795, 9957, 3139	876
32	3199, 822, 6097, 2124, 8736	563
64	2555, 2001, 22, 308, 7580	346
128	31, 4422, 7676, 3023, 724	250
256	8209, 9818, 4435, 2809, 8937	218

Figure A5. Batch-size, seeds, and training time per 100 epochs for Experiment 2

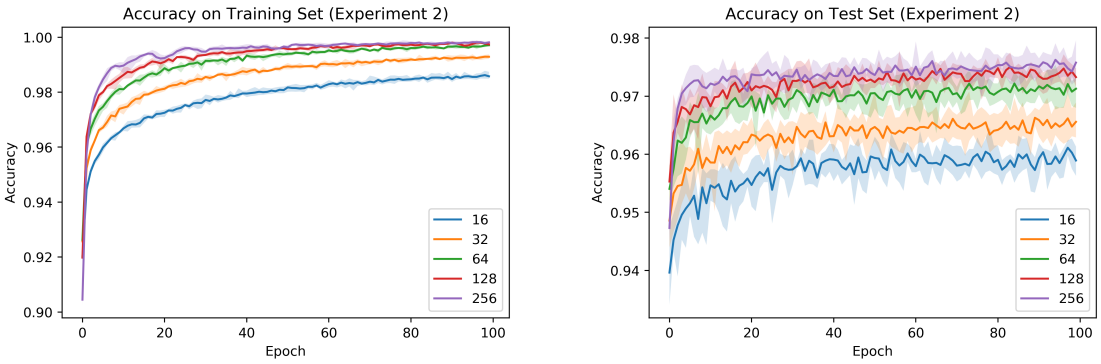

Figure A6. Accuracy of training (*left*) and testing (*right*) for Experiment 2

Hidden Units	Seeds	Parameters	Training time (s)
10	7858, 8839, 6658, 5206, 4797	7,690	142
25	6767, 768, 9758, 7486, 1627	19,885	178
50	1886, 7769, 7327, 3323, 4468	39,760	242
100	931, 5823, 1803, 5598, 794	79,510	383
200	6654, 8857, 2601, 5244, 3079	159,010	496

Figure A7. Number of hidden units, seeds, trainable parameters, and training time per 100 epochs for Experiment 3

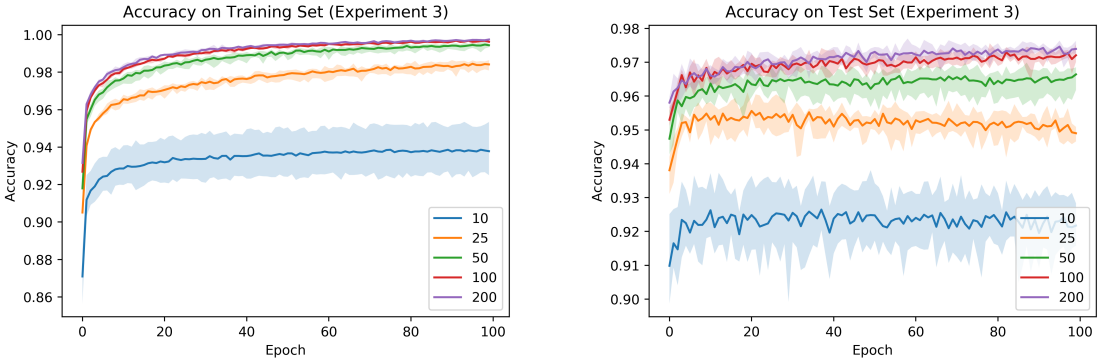

Figure A8. Accuracy of training (*left*) and testing (*right*) for Experiment 3

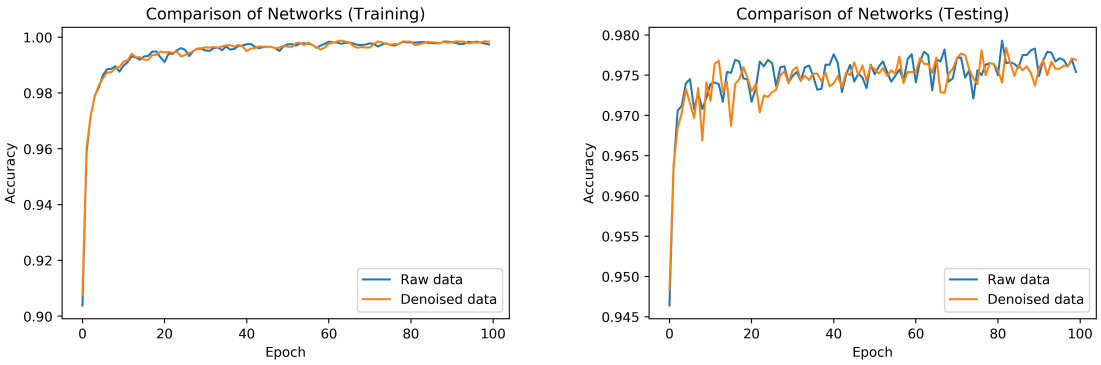

Figure A9. Accuracy of training (*left*) and testing (*right*) for Experiment 4 & 5

	0	1	2	3	4	5	6	7	8	9
					Precision					
Raw data	0.98	0.99	0.97	0.97	0.98	0.96	0.99	0.98	0.98	0.95
Denoised	0.99	0.99	0.97	0.97	0.98	0.98	0.98	0.98	0.96	0.96
					Recall					
Raw data	0.99	0.99	0.98	0.98	0.96	0.98	0.97	0.98	0.95	0.97
Denoised	0.99	0.99	0.98	0.97	0.97	0.97	0.98	0.97	0.98	0.97
					F1-Score					
Raw data	0.99	0.99	0.98	0.97	0.97	0.97	0.98	0.98	0.97	0.96
Denoised	0.99	0.99	0.98	0.97	0.98	0.97	0.98	0.98	0.97	0.97

Figure A10. Comparing performance of the networks from Experiment 4 & 5 across a variety of classification metrics on the testing dataset