Learning to Learn

Swapneel Chalageri
Majors: Computer Science, Biology

Vaidehi Bapat
Major: Computer Science

Nikhilesh Kashyap
Majors: Computer Science, Economics

Zoran Gajic
Major: Molecular Biology & Biochemistry
Minor: Computer Science.

December/16/2017

443/525 Brain Inspired Computing
Professor Konstantinos Michmizos

Contents

Abstract

While learning algorithms for neural networks are important, a commonly overlooked source of error is hyperparameters and model choice. Here we show that both factors lead to considerable changes in classification accuracy and time even when the learning algorithm is kept constant.

1. Introduction

Machine learning holds broad applications to fields such as medicine [1, 2], emotion recognition [3], human-robot interaction [4], and image classification [5]. These tasks utilize learning algorithms that vary in their spatial and time complexity [6]. Similarly, the problem archetypes that each algorithm solves best vary. Thus, one of the initial steps in developing a machine learning approach is algorithm selection. Previous work has shown the unprecedented success of neural networks in these tasks [2, 3, 5]; however, these networks can be slow to train and require extensive compute resources. Many bench-markings of neural networks versus traditional algorithms such as Naïve Bayes classifiers and clustering approaches utilize extensive and complete datasets such as MNIST [7], but lack analysis of these algorithms in stressed datasets. Such data includes sets with high noise, high dimensionality, and a low amount of training data. These data attributes exist in many real-world areas such as medicine and genomics [8].

Another caveat of these algorithms is excessive hyperparameterization. Hyperparameters are parameters that are set before learning begins. These include learning rate in neural networks and perceptrons, k in k-nearest neighbors and timestep in spiking neural networks. While recent gradient based approaches to automating hyperparameter selection have been successful [9, 10], these optimization algorithms require even more computational power and time for training. Thus, in many cases, hyperparameters are still chosen by intuition or heuristics [9].

These observations go beyond simple learning and suggest the question of how do we learn the algorithms to use and the parameters to select for an arbitrary task. This encompasses the idea of learning to learn, a biologically relevant phenomenon that forms the basis of human intelligence. Analysis of learning ability in children versus adults suggests that children are much slower at learning novel tasks [11, 12]. This suggests that either the algorithms/mechanisms used by the child's brain are inherently slower, or that they have not yet optimized learning architecture.

To further our understanding of these networks in data-stressed environments, we utilized a subset of the MNIST dataset to compare a series of selected algorithms in time and accuracy. We then explored the effects of hyperparameterization on the dataset.

2. Learning Algorithms

2.1 K-Nearest Neighbors [13]

K-nearest neighbors utilizes a feature vector f, such that f is a numeric vector of size greater than one. This numeric feature vector can then be represented as a point in an n dimensional space (n is the length of the vector). We then assume that points of similar type (classification) would group together in this space forming clusters or pseudo-clusters. If this is true of the dataset, we can identify the nearest neighbors (up to k neighbors, k being a hyperparameter) and use a weighted majority voting system to select the class of the novel data point based on the weighted mode of the of k-neighbors. The weighting is conducted by class representation in the dataset. i.e. a class that comprises 90% of the data will have its vote scaled down by a factor of 10 (1-p where p is the proportion of representation of the data in the set). Various distance measures are possible, however we opted to utilize simple Euclidian distance.

$$distance = (\sum_{i=1}^{n}(q_i - p_i)^2)^{\frac{1}{2}}$$

2.2 Naïve Bayes [14]

2.2.1 The Naïve Bayes Algorithm

This classifier takes a probabilistic approach to find similarities between test and training samples. The prediction is done based on:

$$arg\max_y \log P(y|f_1, \dots, f_m) = arg\max_y \log P(y, f_1, \dots, f_m)$$

$$= arg\max_y \{\log P(y) + \sum_{i=1}^{m} \log P(f_i|y)\}$$

The marginal probability of each label is calculated as the total frequency of the label in the training data divided by total count of training data.

$$\hat{P}(y) = \frac{c(y)}{n}$$

The conditional probabilities are calculated for each feature in the input sample, for each given value of label. It is calculated as:

$$\hat{P}(F_i = f_i|Y = y) = \frac{c(f_i, y)}{\sum_{f_i' \in \{0,1\}.} c(f_i', y)}$$

2.2.2 Naïve Bayes Smoothing

There is a possibility that the conditional probability is zero owing to no matching training feature for a given label. To solve this, a smoothing factor of 1 is added to the count of f_i given y. The new conditional probability equation after smoothing is as given below:

$$P(F_i = f_i | Y = y) = \frac{c(f_i, y) + 1}{\sum_{f_i' \in \{0,1\}} (c(f_i', y) + 1)}$$

2.2 Perceptron [15]

A Perceptron can be thought of as a single layer ANN where each neuron takes in n inputs and produces one output. These outputs are then mapped to classes. Weights are assigned to each input and a score produced by each neuron as:

$$score(f) = \sum_i f_i \, w_i$$

Where f is in magnitude of a given feature and w is the weight associated with that feature for a given neuron. The class is then assigned as the highest scoring neuron in the layer.

Training the weights was done via the following. For each feature-value pair (f,v), we classify the feature, and if the classified feature matched the value, then we continued. If the classified feature did not match the value, we take all the weights of the perceptrons that fired incorrectly and update them to decrease their chance of firing in the future for this pattern of feature set.

$$w^{y'} = w^{y'} - f$$

Where y' indicates the nodes that did not fire, and f is the feature vector that was misclassified.

We then take the perceptron that should have fired and increase its weights so that it is more likely to fire in the future for this pattern of feature set

$$w^{y} = w^{y} + f$$

Where y indicates the node that did fire and f is the feature vector that was misclassified.

Training consists of several batches where in each batch we iterate through the entire training set and update the weights for each item in the training set.

A learning rate parameter α can be used to increase or decrease the rate at which the weights change. To incorporate this the previous functions become:

$$w^{y} = w^{y} + \alpha f$$
$$w^{y'} = w^{y'} - \alpha f$$

2.2 Artificial Neural Networks and Back-Propagation [16]

A neural network is a series of perceptrons where the output of one perceptron becomes the input for another. This allows for more complicated patterns to be built from simpler features such as intensity, as many initial features can be combined in the hidden layers to produce new abstract and complicated features that may describe the data, but would be nearly impossible for a human to create.

In a neural network we have three types of nodes; input, hidden, and output. There is one "layer" of input nodes and one "layer" of output nodes, but there can be many "layers" of hidden nodes. Information is processed similarly to a perceptron, where a score is calculated as such:

$$score = weight * input + bias$$

Since we need to pass this score to the next layer of nodes we will want to transform it into a set range, as the output will indicate how strongly this node is activating the next node. The function we used for this network was the sigmoid function, given by:

$$\sigma(score) = \frac{1}{1 + e^{-score}}$$

Gradient descent was then used to adjust weights as given an initial error (gradient), we can follow that gradient backwards through the network to identify which portions of the network are most responsible for the error, at which point we can adjust the weights proportional to the amount of error each weight caused. This is known as the back-propagation algorithm and solves the credit assignment problem.

We begin by take a small sample of our training data that we will call a batch. Each item in our batch has a feature and a value (digit) associated with that feature. For every value, we use the forward propagation algorithm to produce the activation of the output nodes. This activation, should, in a perfect world, be a vector of length 10, where every position, except the position corresponding to the value of the feature is 0. The position corresponding to the value should be 1. However, this is never the case, and we will expect to see a value slightly less than 1 for the true class and noise in the other positions. What we can do, is we can subtract the observed values from the expected to give us our error. This error can then be used to find the component-wise error of the last (output) layer with the following expression:

$$\delta^L = (observed - expected) * \sigma'(score^L))$$

Where δ^Lindicates the component-wise error of the output layer, observed is the output from the forward propagation, expected is the given value and $score^L$ is the score for the last layer calculated during forward propagation.

We can then propagate this error by the following equation through the layers of the network using the following:

$$\delta^L = ((weigths^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Where l indicates the lth layer from the last layer (2 indicates 2nd to last), δ^{l+1} indicates the component wise error calculated in the previous step, $\odot$ indicates the Hadamard product and

$\sigma'(z^l)$indicates the derivative of the sigmoid function of the current layer. Another reason we use the sigmoid function is that the derivative can easily be calculated given the value of $\sigma'(z^l)$:

$$\sigma'(z) = \sigma(z) * (1 - \sigma(z))$$

We continue back-propagating the error until we get back to the input nodes, at which point we move onto the next item in the batch.

Once we have all of the component-wise errors for each layer from each batch, we can update the weights and biases according to the following equations:

$$w_{new} = w_{old} - \frac{\eta}{n_b} \sum_{x=0}^{n_b} (\delta_x * activation_x)$$

$$bias_{new} = bias_{old} - \frac{\eta}{n_b} \sum_{x=0}^{n_b} (\delta_x)$$

Where n_b is the number of items in the batch, η is the learning rate which indicates how much the network will change at each step and δ_x indicates the component-wise error of w, and $activation_x$ indicates the sigmoid function applied to the weighted sum on inputs.

The beauty of the network however, is that we can do all of this within matrices, thus making the calculations and code easier and the program easily parallelizable.

Once we have completed the current batch, we continue iterating through the batches until we finish the whole training set, at which point we do the process of splitting the data into batches and training again, until a certain number of iterations.

2.2 Spiking Neural Networks and STDP [17]

Spiking Neural Networks (SNN's) are capable of low latency, high energy efficiency neural networks that can process high volumes of information and generate intelligent outcomes. When it comes to training these networks, many papers have shown that the usage of Spike Timing-Dependent Plasticity is a powerful form of long term synaptic learning.

STDP builds on Hebbian Learning, a form of learning that will strengthen the connection between two neurons I & J if there is a strong connection between the two, strong defined as a presynaptic firing leading to a postsynaptic firing. We can then add features to mitigate Hebbian Learning shortcomings such as lack of competition and chance based strengthening.

This strengthening of a connection coming as a result of a presynaptic spike arrival a few milliseconds before a postsynaptic action potential. This then leads to Long-Term Potentiation. On the other hand, a spike arrival shortly after a postsynaptic action potential, leading to Long-Term

Depression. These fluctuations as a function of time determine learning in our network. We will engineer a feed-forward spiking neural network to behave in this way.

Our network architecture consists of two layers, an input layer containing the 28x28 neurons, and a second layer which will process this information. We trained and tested a network with 100 excitatory neurons by presenting 1,000 examples of the MNIST training set.

The learning method used to calculate weight change is as follows:

$$\Delta w = \eta\mu\left(x_{pre} - x_{tar}\right)(w_{\max} - w)$$

Where η is the learning rate, $w_{\max}$ is a maximum weight boundary, μ is the dependence of the weight on the previous weight and x_{tar} is the target value for the presynaptic trace at the time of spiking.

The input encoding schema followed was rate based.

3. Benchmarking Data

3.1 MNIST

The MNIST dataset [7] contains 70,000 28x28 images of handwritten digits divided into a training set of 60,000 images and a testing set of 10,000 images. We arbitrarily selected 1,000 training and 500 testing images to model data deficiencies observed in real world applications such as genomics where high costs provide barriers to data acquisition. Unless otherwise stated the 1,000-500 training-testing set was used for experiments.

Feature extraction was conducted as follows for each algorithm unless otherwise stated. Initially, images were discretized such that pixels were converted into floats, 0.0 representing white squares, 0.5 representing gray squares and 1.0 representing black squares. The 28x28 images were then converted into a vector such that position 0 of the vector corresponds to the top left pixel and continues such that the bottom right pixel is the last item in the vector. This feature set was then utilized to train each of the various algorithms.

3.2 Naïve Bayes Feature Extraction

We utilized an altered feature set for the Naïve Bayes Classifier:

1. Number of black pixels in rows
2. Number of black pixels in columns
3. Total Number of black pixels in entire image
4. Number of black pixels in grid blocks where the digit images are divided into a grid of size 4×4, with each grid block of size 7×7.

4. Results

4.1 Analysis of K-Nearest Neighbors Hyperparameter.

The K-Nearest neighbor's classification algorithm was utilized to classify the testing set of data for values of K between 0 and 40 (Fig 1). It was observed that low and high values of k resulted in poorer classification of the testing set. However, a bimodal distribution was observed indicating higher order interactions of the algorithm and the dataset.

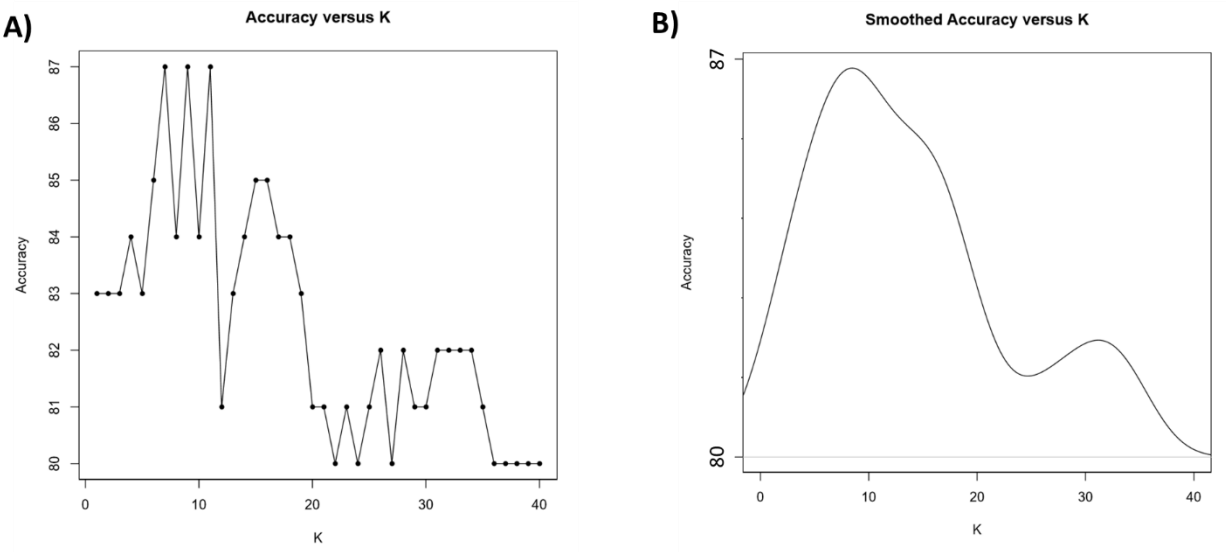


Figure 1: Accuracy of K-Nearest Neighbors Versus the Value of K. Accuracy as a function of K. Values of k from 0 to 40 were tested and their accuracy evaluated on a training-testing set of 1000-500 taken from the MNIST training set. A bimodal distribution was observed. **A)** direct plotting of Accuracy vs K. **B)** kernel density plot of accuracy vs K.

4.2 Analysis of Perceptron Hyperparameter

The perceptron algorithm was used to classify the testing set of data for learning rates between 0.25 and 2 at steps of 0.25 (Fig 2). Each trial was conducted 20 times and the results averaged. We observed that learning rate does not affect overall classification accuracy but does lead to differences in the learning curve. Lower learning rates generally led to better early classification while higher learning rates led to poorer early classifications

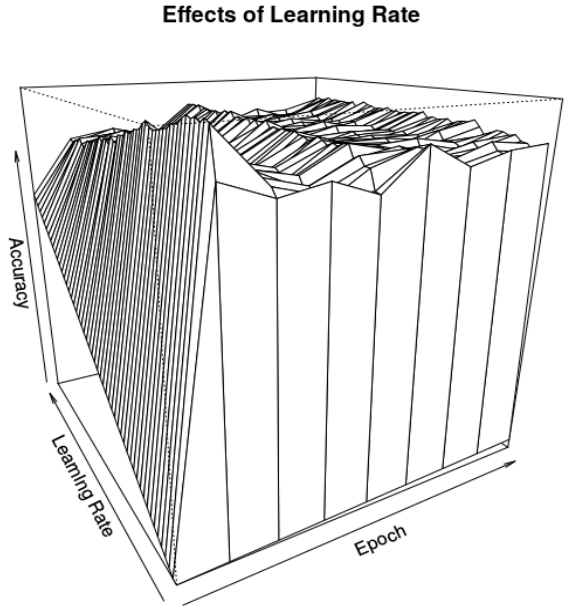


Figure 2: Analysis of the Effect of Learning Rate on Perceptron Classification. Accuracy as a function of learning rate and training epoch. Learning rates from 0.25 to 2.0 were tested with a step of 0.25. Interestingly, higher learning rates lead to a slower learning curve than lower rates.

4.3 Analysis of ANN Hyperparameters

Four hyperparameters were tested. They were initialized as:

Learning rate: 0.1

Number of Hidden Layers: 1

Number of Nodes in Hidden Layer: 30

This initialization was conducted by intuitively estimating which sets of parameters led to the best classification results. A set of trials were conducted such that combinations of hyperparameters were tested 20 times and the results averaged for each training epoch. 50 training epochs were used (Fig 3).

Firstly, the batch size used for training was varied between 1 and 9. Results indicate that increased batch size lead to a slower learning curve as well as a lower overall acquisition of data patterns. Batch size was then fixed at 1 and the learning rate was varied at 0.25 increments between 0.25 and 2.0. As learning rate was increased we saw a faster learning curve. However, as learning rate increased further, we observed an overall decrease in final classification accuracy, peaking at 1.25. The learning rate was then fixed at 1.25 and the size of the hidden layer was varied at steps of 5, ranging from 5 to 45. Interestingly, we observed a damped sinusoidal oscillation of total accuracy as a function of hidden layer size. This oscillation peaked at 20 nodes. The hidden layer size was then set to 20 and the number of hidden layers varies between 0 and 6. Not only did we see a lower learning curve, but we also saw a marked decrease in total accuracy at the end of the training period.

4.4 Observations of the SNN Trained with STDP.

We were only able to successfully run a two-layer network using STDP. This was primarily due to the excessive amount of time required for the simulation. Training and testing a two-layer network on only 1000 data points requires over 90 minutes of running time on a high-end Macintosh laptop. The abysmal 53% success rate of the trained network compounds this. Our implementation utilized BRIAN and thus we can assume that this network was well optimized for speed and memory usage when compared to our self-implemented algorithms.

4.5 Overall Comparison of Algorithms.

We then compared the algorithms in terms of their running time and accuracy achieved on the stressed dataset (Fig 4). We noted that the simpler algorithms generally provided better at classification, and ran faster. With the range: KNN completing in 0.001 seconds and SNN (STDP) completing in 94 minutes. KNN achieved an accuracy of 87 and SNN (STDP) achieved 53% correctness.

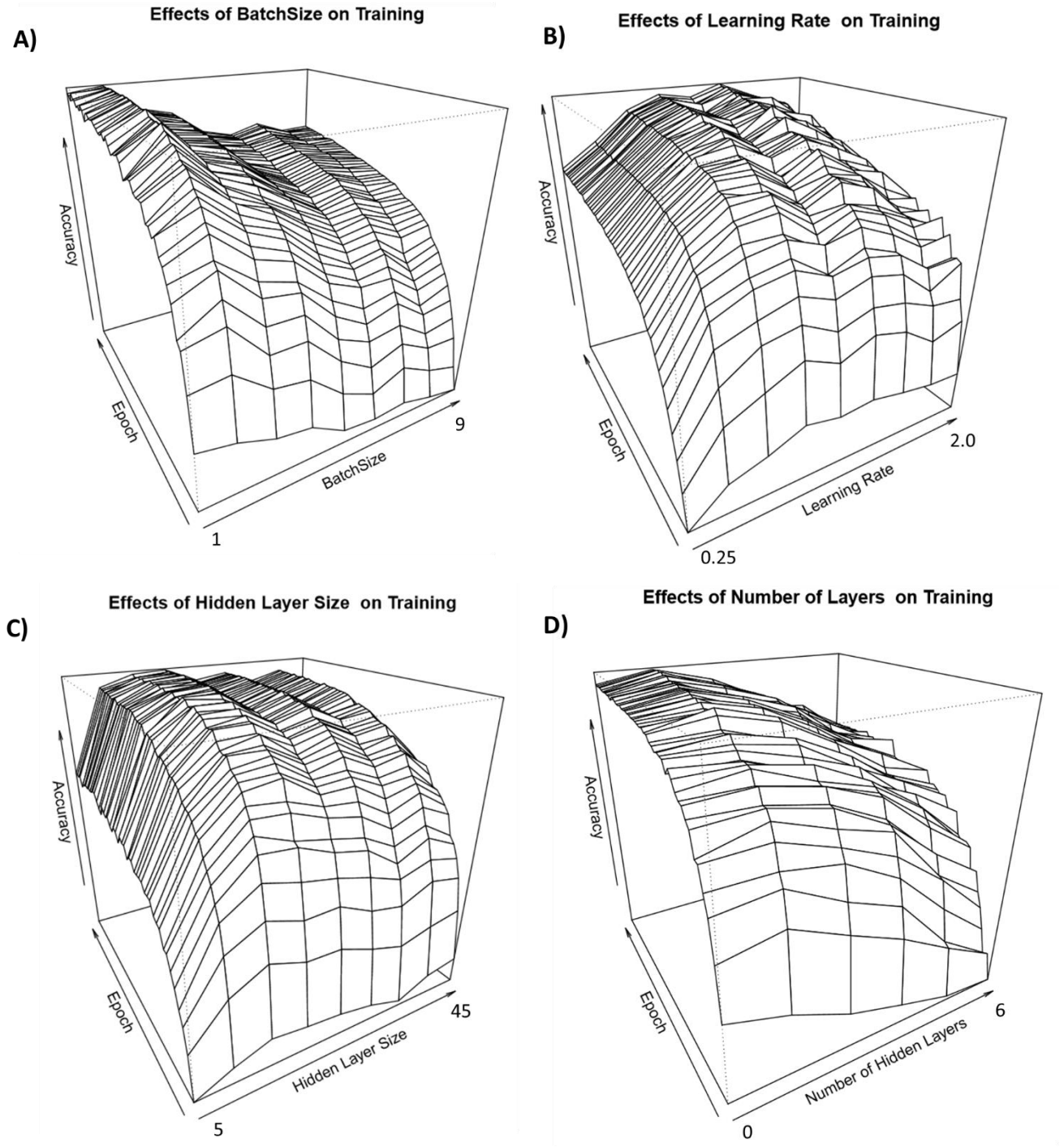
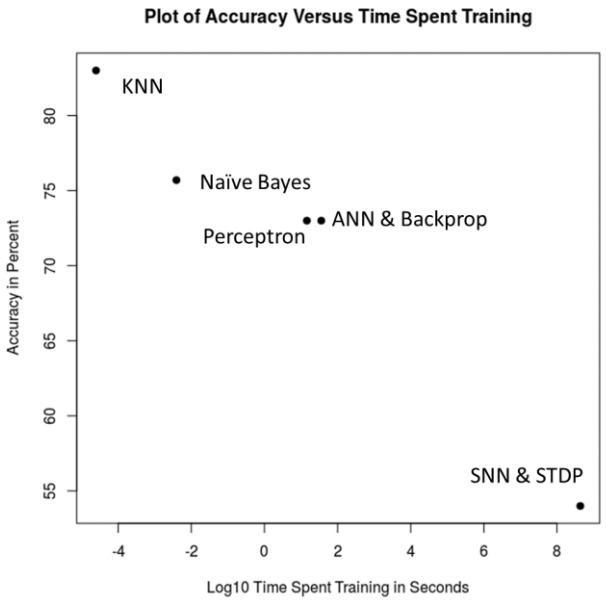


Figure 3: Optimization of Hyperparameters for Artificial Neural Network Back-Propagation. Hyper parameters of the ANN were varied prior to training with backpropagation. **A)** Accuracy as a function of batch size and training epoch. A simple curve showing the optimal batch size as 1. **B)** Accuracy as a function of learning rate and epoch. Note that this curve implies a goldilocks zone for learning rate, not too low and not too high. The optimized learning rate was 1.25. **C)** Accuracy as a function of hidden layer size and training epoch. Note the oscillatory patterns as we vary hidden layer size. The optimal hidden layer size was observed to be 20. **D)** Accuracy as a function of the number of hidden layers and training epoch. Here we see that one hidden layer leads to increased classification accuracy, but excess hidden layers decrease accuracy.

Figure 4: Comparison of Running Time and Classification Accuracy for the Algorithms. Classification accuracy of our algorithms versus the log of their running time. As the complexity of the algorithm increases we notice a decrease in accuracy and an increase in running time for our stressed data set. Note that time is measured in log scale and as such the range between KNN and STDP is 0.001 seconds and 94 minutes. To verify our classifiers, we tested all of the classifiers except for STDP against the entire MNIST dataset. STDP was not tested because BRIAN has been externally validated. When utilizing the whole training set we see; **KNN:95%, NB: 81.6%, Perceptron: 73%. BackProp: 94.1%.**



5. Conclusions

5.1 Neural Networks are not Universally Better at Classification

As we have shown, in data sets where we are lacking comprehensive representation of classes, algorithms such as KNN and Naïve Bayes perform better than neuronal inspired algorithms. This may be due to the propensity for neural networks to exploit arbitrary patterns of the data in order to best classify the training set (overfitting). Likewise, the simplicity of KNN and Naïve Bayes afford easier and faster implementations than neural networks which further promotes their use in experimental settings. We do see however, that when provided the entire dataset, backpropagation (BP) was able to reach similar levels of accuracy as KNN (94.1% BP, 95% KNN). This implies that while KNN and NB are able to effectively classify using small datasets, they may not see drastic improvements with larger sample sizes. Similarly, the simplicity of KNN and NB leads to limits on the improvement of these algorithms, while optimizations such as dropout [18] offer the ability to augment and tune neural networks for high performance. Likewise, when we utilize a spiking network, we see failures in both the accuracy and time specifications. This is arguably a simple problem with a very simple dataset, and to see training times of 94 minutes implies poor scalability. We had initially attempted to implement spatio-temporal backpropagation [19], however we noticed that not only does the time domain lead to excessive running times. The timestep required to validate several approximations made by the designers of spatio-temporal backpropagation made the algorithm almost unusable on personal hardware. Thus, when working with limited data, or computing power, utilization of non-neuronal algorithms may be preferable.

5.2 Gradient Based optimization of Hyperparameters.

We have shown that accuracy as a function of hyperparameters does not result in simple curves. Accuracy as a function of k in KNN exhibits a bimodal structure. This structure is non-conducive to a purely gradient based optimization and requires stochastic restarts or algorithms such as simulated annealing in order to find global maxima. Similarly, we noticed a 7% difference in accuracy between the best and worst KNN classifiers. This stresses the importance of hyperparameters and their optimization.

Similar patterns were observed when we optimized the back propagation hyperparameters. Number of hidden layers, batch size and learning rate would all be conducive to simple hill climbing or gradient descent, however, the sinusoidal patterns observed in the optimization of the hidden layer size offers concerns for purely gradient based optimization approaches. Likewise, these emergent patterns are also a function of the data and might change as we migrate from MNIST to facial recognition to medicine.

Therefore, while back propagation is currently the king of machine learning, novel algorithms and analyses of the supporting parameters are necessary to improve out understanding of the dynamics of neural networks.

5.3 Future Work

While we were unable to test the hyperparameters of STDP, we could expect equal or grater levels of complexity when compared to back propagation due to the added complexity of the temporal domain. To this end, further understanding of the architecture required for learning through a temporal domain could shed light on the technical requirements of endogenous neural systems. Similarly, expanding our analysis to various modifications of classic neuronal algorithm such as recurrent networks could provide interesting comparisons.

While we have uncovered interesting manifestations of hyperparameters in the MNIST dataset, comparison of these results to other real-world datasets provides an interesting mechanism to study the impacts of data and specific problems on neural learning.

6. Acknowledgments

The authors would like to thank Dr. Konstantinos Michmizos, Guangzhi Tang, and Jeff Ames for the knowledge and assistance they provided throughout the semester.

7. References

[1] Deo, R. C., 2015, "Machine Learning in Medicine," Circulation, 132(20), pp. 1920-1930.
[2] Forsting, M., 2017, "Machine Learning Will Change Medicine," J Nucl Med, 58(3), pp. 357-358.
[3] Shojaeilangari, S., Yau, W. Y., Nandakumar, K., Li, J., and Teoh, E. K., 2015, "Robust representation and recognition of facial emotions using extreme sparse learning," IEEE Trans Image Process, 24(7), pp. 2140-2152.
[4] de Greeff, J., and Belpaeme, T., 2015, "Why Robots Should Be Social: Enhancing Machine Learning through Social Human-Robot Interaction," PLoS One, 10(9), p. e0138061.
[5] Yann LeCun, L. B., Yoshua Bengio, Patrick Haffner, 1998, "Gradient-Based Learning Applied to Document Recognition," IEEE.
[6] Lim, T.-S., Loh, W.-Y., and Shih, Y.-S., 2000, "A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms," Machine Learning, 40(3), pp. 203-228.
[7] Yann LeCun, L. B., Corinna Cortes, Christopher J.C. Burges, 2017, "THE MNIST DATABASE of handwritten digits," http://yann.lecun.com/exdb/mnist/.
[8] Yuan, Y., Shi, Y., Li, C., Kim, J., Cai, W., Han, Z., and Feng, D. D., 2016, "DeepGene: an advanced cancer type classifier based on deep learning and somatic point mutations," BMC Bioinformatics, 17(Suppl 17), p. 476.
[9] , and Andrychowicz, M. D., Misha; Gomez, Sergio; Hoffman, Matthew W.; Pfau, David; Schaul, Tom; Shillingford, Brendan; de Freitas, Nando, 2016, "Learning to learn by gradient descent by gradient descent," ARXIV.
[10] Bengio, Y., 2000, "Gradient-based optimization of hyperparameters," Neural Comput, 12(8), pp. 1889-1900.
[11] Harel, B. T., Pietrzak, R. H., Snyder, P. J., Thomas, E., Mayes, L. C., and Maruff, P., 2014, "The development of associate learning in school age children," PLoS One, 9(7), p. e101750.
[12] Rabi, R., and Minda, J. P., 2014, "Rule-based category learning in children: the role of age and executive functioning," PLoS One, 9(1), p. e85316.
[13] 2015, "Tutorial: K-Nearest Neighbor classifier for MNIST," https://lazyprogrammer.me/tutorial-k-nearest-neighbor-classifier-for-mnist/.
[14] 2015, "Bayes classifier and Naive Bayes tutorial (using the MNIST dataset)," https://lazyprogrammer.me/bayes-classifier-and-naive-bayes-tutorial-using/.
[15] 2011, "Project 5: Classification," http://inst.eecs.berkeley.edu/~cs188/sp11/projects/classification/classification.html.
[16] Nielsen, M., 2015, "Neural Networks and Deep Learning."
[17] Peter U. Diehl, M. C., 2015, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," Computational Neuroscience, 9(99).
[18] Salakhutdinov, N. S. a. G. H. a. A. K. a. I. S. a. R., 2014, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," Journal of Machine Learning Research, 15, pp. 1929-1958.
[19] Wu Y., D. L., Li G., Zhu J., 2017, "Spatio-Temporal Backpropagation for Training High-performance Spiking Neural Networks," ArXiv e-prints.