Is Amplitude All You Need?

Michael Yantosca

Abstract

Training neural networks for machine learning tasks is expensive in time and energy. One of the current trends for combatting this is to minimize the amount of intermediate state required, especially if said intermediate state bears the imprint of expert systems from previous decades of research. The supposition is that with enough data all concepts are shallow. To investigate the feasibility of this approach, the work introduces PINNIPED, a Parameterized Interactive Neural Network Plotting Iteratively Experimental Decisions, which can minutely specify the parameters of a neural network and produce plots and intermediate model states over the course of training. An attempt is made to train a feedforward neural network using only backpropagation and stochastic gradient descent with amplitude data for input and English phoneme labels as classifier output. While the parameters and techniques employed were by no means exhaustive, the results hint that for some tasks a certain amount of expertise is in fact required, usually with respect to the preprocessing phase of ingesting the data.

Contents

1 Introduction

The past decade in natural language processing (NLP) research has seen a trend of eliminating complex intermediate stages in models in favor of deep-learning approaches that translate input data directly to the desired output. Salient examples in this vein include "Deep Speech"[12];

"Listen, Attend and Spell"[2]; and most recently, "Attention Is All You Need"[21]. While the former two rely on recurrent neural networks to avoid phonemic representations or costly hidden Markov models (HMMs), the last dispenses with recurrence and convolutions entirely, asserting that a simplified attention mechanism is all that is required to achieve competitive results.

These intermediate state reductions have been directed primarily at the models which have been heretofore the province of linguistic theory. If the issue is mostly a matter of linear transformation, the question then arises as to how minimally one can preprocess the acoustic data and still make reasonable predictions.

In the research mentioned above, acoustic input data typically takes the form of spectrograms[12, p. 2] and filter-bank spectra features[2, p. 2]. The Transformer architecture in the last paper deals strictly with textual sequence-to-sequence translation as opposed to speech recognition, but the minimalist principles cohere with the other work, and the removal of recurrence and the published results are worth mentioning.

Since spectrograms and similar data representations are themselves simply convolutions of the raw amplitude data, one could theoretically apply the same minimalist principles in the opposite direction and use windowed batches of raw amplitudes as direct input to a neural network. This work measures the feasibility of such an approach applied to English phoneme classification. In keeping with the theme of simplicity, the neural network constructed solely uses backpropagation and stochastic gradient descent for training a feedforward architecture.

2 Design and Implementation

The complexities of splitting continuous audio data into phonemic frames is glossed over in the interest of time. The training and test sets come from the University of East Anglia time series classification archive[1][10], which provided in the Phoneme challenge a small subset of the 370,000 labeled phoneme frames originally published by Hamooni and Mueen[11]. The frames are normalized as a series of 1,024 amplitude values followed by a numeric phoneme label.

Notably, the Phoneme challenge currently reports a maximum achieved accuracy of 30.28%. This may be due to the fact that the training set (214 examples) is nearly an order of magnitude smaller than the test set (1,896 examples), and data-hungry techniques such as neural networks are likely to fare poorly in the context of the challenge. However, the uniformity of the data layout provided in easily digestible ARFFs[3] present a manageable environment for an initial foray in evaluating the usefulness of training a neural network solely on amplitude values with only minor, inexpensive preprocessing.

The Python program written to execute the experiments is dubbed the Parameterized Interactive Neural Network Iteratively Plotting Experimental Decisions, or PINNIPED for short. PINNIPED is built on PyTorch[5][4], leveraging the framework's autograd feature, and can operate in either training mode or test mode. The user may specify a number of parameters to minutely specify the neural network to be trained, including non-linear activation, learning rate, learning momentum, batch size, training epochs, number of hidden layers and nodes within each layer individually, and input test or training set ARFFs.

Users are encouraged to review the included `README.md` or the program's help message for details on usage.

2.1 Neural Network Layout

The neural network composed by user-specification at the command line is a simple feedforward network with backpropagation[8, pp. 284–296]. The counts of nodes per layer is specified through the `--layer-dims` command-line argument as a comma-separated list of integer values.

As such, the number of hidden layers permitted is arbitrary, but this work focuses on the single hidden-layer case. At least 2 layers, i.e., input and output, must be specified. The layer itself is implemented as a PyTorch Linear module.

The same non-linear activation function is applied to the output of all hidden layers. The user may choose between the sigmoid, hyperbolic tangent, or rectified linear unit (ReLU). The non-linear transformation is implemented through the corresponding Pytorch module.

A normalization preprocessing step is also performed during data loading to compress the dynamic range of the samples into the range $[0, 1]$, which was done under the assumption that it might provide at least some guard against gain variations between training, validation, and test utterances. The network pipeline itself has another normalization layer at the head, namely the LayerNorm1d module provided by PyTorch, to take advantage of the best-practice per-sample normalization techniques developed by the PyTorch contributors for the platform.

2.2 Training

Training may be parameterized on the command line according to a few different typical regimens. For the purpose of validation, the user may reserve either the last fraction of the training samples (e.g., the last quarter) or samples taken at regular intervals (e.g., every fourth sample). PINNIPED supports both batch backpropagation and stochastic gradient descent (SGD), but it does not support online training.

Under SGD, the order of the training samples is permuted before training starts[19], and reservation is done *after* this initial permutation. No permutation of sample order is done if not specified by the user. The division betwen training and validation sets holds for the duration of training, but SGD also permutes the training set sample order at the start of each epoch so that each training sample is seen once per epoch but in a potentially different order each time. The validation set order is not permuted after reservation since no learning is taking place during validation, i.e., the PyTorch autograd mechanism is turned off for testing.

Batching can be done in batches of 1 up to the total training set size. Specifying a number greater than the training set size will default to the training set size. If the specified batch size does not divide the training set evenly, the last batch will be smaller than this specification. Users who require exact equality in size for all batches must either truncate the training set or augment it with additional samples if the set is not evenly divisible by the batch size.

In order to make it easier to plot training, validation, and test accuracy in one graph, the test set may be passed through as a sort of second validation set. With intermediate plotting turned on, this will generate the performance of the model at every training stage against all the sets. This is not recommended for development testing since it can overfit to the test set and furthermore violates the best practices of test-set blinding that have been established in the field of machine learning so as to allow each model to be judged on its own merits and not the furtive hand of its inventor. However, the option is left as a convenience for illustrative and didactic purposes.

2.3 Artifacts

A number of artifacts can be generated with user-specified regularity, namely plots of performance and evolution along with intermediate model states.

Using matplotlib[17] library for generation, the plots include the following:

- a plot of training, validation, and test accuracy over time (epochs)
- per-layer heatmaps of weight vector changes by angle and norm per node over time (epochs)

- per-layer heatmaps of non-linear activation counts of nodes against uniform bins
- heatmaps of training, validation, and test confusion matrices

All plots are rendered in PNG format. Heatmaps are generated as `pcolormesh`[16] rather than `imshow`[15] plots because the latter do not visually scale as well. The color values used by the heatmaps belong to the 'hot' scale[14], which seemed to be the most visually intuitive representation in all the use cases above.

The weight angle change graph uses the standard formula for angle difference between two vectors[18, p. 335],

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta$$
$$\theta = \arccos\left[\frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}\right]$$

where θ is the angle between the two vectors $\mathbf{u}$ and $\mathbf{v}$. Because of documented concerns over precision in PyTorch[7], additional precautions were taken in the boundary cases close to the extrema of the arccos range. Additionally, colinear weight vectors, i.e., vectors with an angle of 0 radians between them, may not have the same magnitude, hence, the change in norm plot. If either vector had zero length, the change in angle was clamped to zero on the premise that zero vectors are colinear with all vectors, so to speak.

The intermediate model states are saved as Pickle serializations of the PyTorch model state dictionaries. This binary format for Python object serialization can be imported for test-only runs of the program. The intermediate activation values are captured by a forward hook[20] which increments a tensor[6] of bins. Because none of the modules themselves have unique name properties, currying[9] was employed to associate a common hook wrapped in a partial lambda to avoid code duplication.

If the user turns on debug output, most of the actual values used to generate the plots will be printed to `stderr`. The confusion matrices are printed in this mode with a sparse text layout blanking out cells holding zeros for legibility as well as prefixing hits with a plus (+) and misses with a minus (-).

2.4 Testing

PINNIPED can run in a test-only mode, which is useful for evaluating a model against multiple test sets and avoiding the time penalty of repeating the same training multiple times. Of course, it should be noted that SGD training runs are not guaranteed to be perfectly reproducible.

The test-only mode does not yield any artifacts other than the output to stderr of the accuracy. No additional models are saved nor any plots made. The latter would be a worthwhile enhancement, however.

3 Experiments

3.1 Parameters

The experiments were varied along the following parameters:

- nodes per hidden layer, $N_H \in \{1, 4, 16, 64\}$
- learning rate, $\lambda \in \{0.1, 0.01, 0.001\}$
- batch size, $b \in \{1, 40, 80, 160\}$
- momentum, $\alpha \in \{0.0, 0.1, 0.5, 0.9, 1.0\}$

- activation unit, $u \in \{\text{sigmoid}, \tanh, \text{ReLU}\}$

All experiments ran training with SGD for 256 epochs, reserving every fourth sample for validation. Thus, the training set size was 160, the validation set size 54, and the test set size 1,896. The experiments did not follow a complete test matrix but rather varied along one parameter axis with respect to the following baseline: $N_H = 64$, $\lambda = 0.001$, $b = 160$, $\alpha = 0.0$. Each set of parameters was tested 3 times to provide some coverage of random variation without making the experimentation run too long since PINNIPED has not been optimized for performance. The plots shown in the following sections are not averaged or otherwise aggregated but merely representative samples of the experiments.

3.2 Classification Error Over Time

The classification errors show the typical learning curve with a plateau as the model stabilizes and reaches its minimum error state. Frequently, equilibrium is achieved well before the defined 256 epoch limit.

Variation in b Fig 3.2.1 illustrates the variation in error resulting from a change in batch size. The largest batch size exhibits the greatest distance between training and validation error at equilibrium, as well as the slowest convergence. Decreasing batch size appears to accelerate convergence as well as improve accuracy across all sets. This alacrity toward convergence correlates to earlier overfitting as one observes the validation error start to increase in the $b = 1$ case while the error in the $b = 160$ case is still descending for both training and validation after 256 epochs. The confusion matrices in Fig 3.2.2 correlate with the plots of error over time.

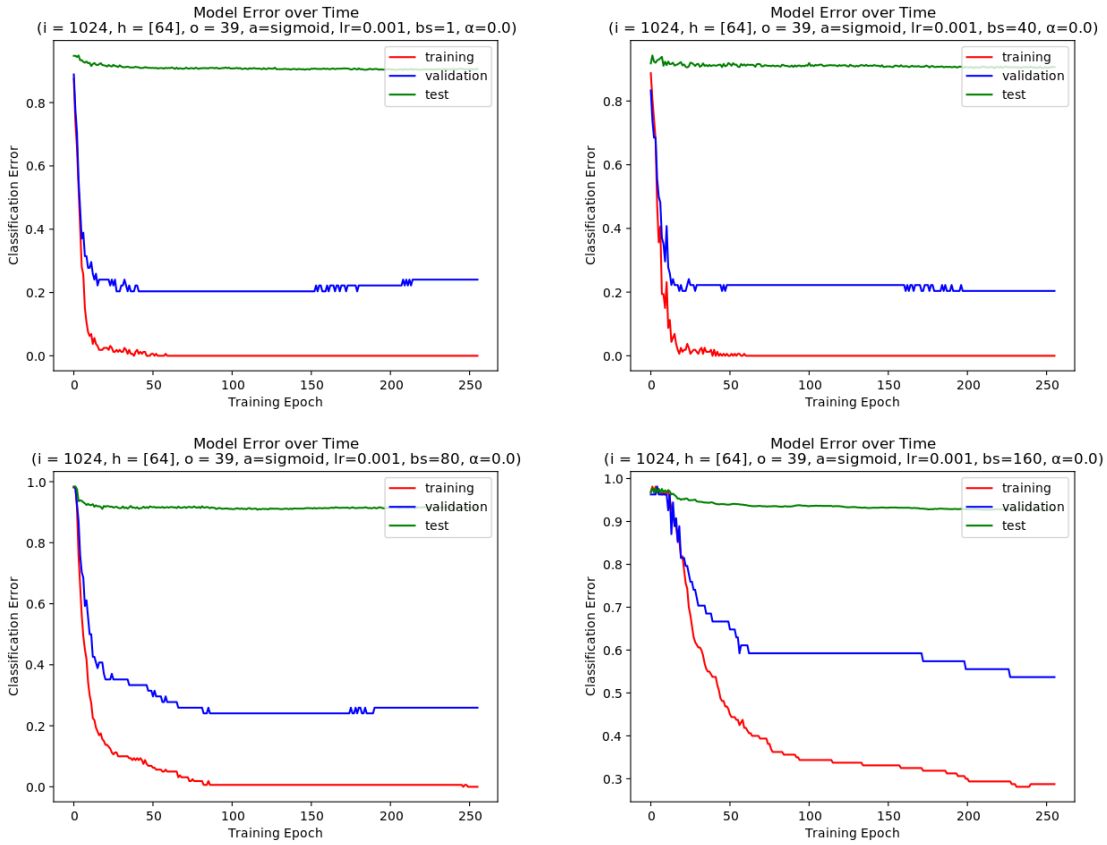

Figure 3.2.1: Classification Errors Varied by Batch Size

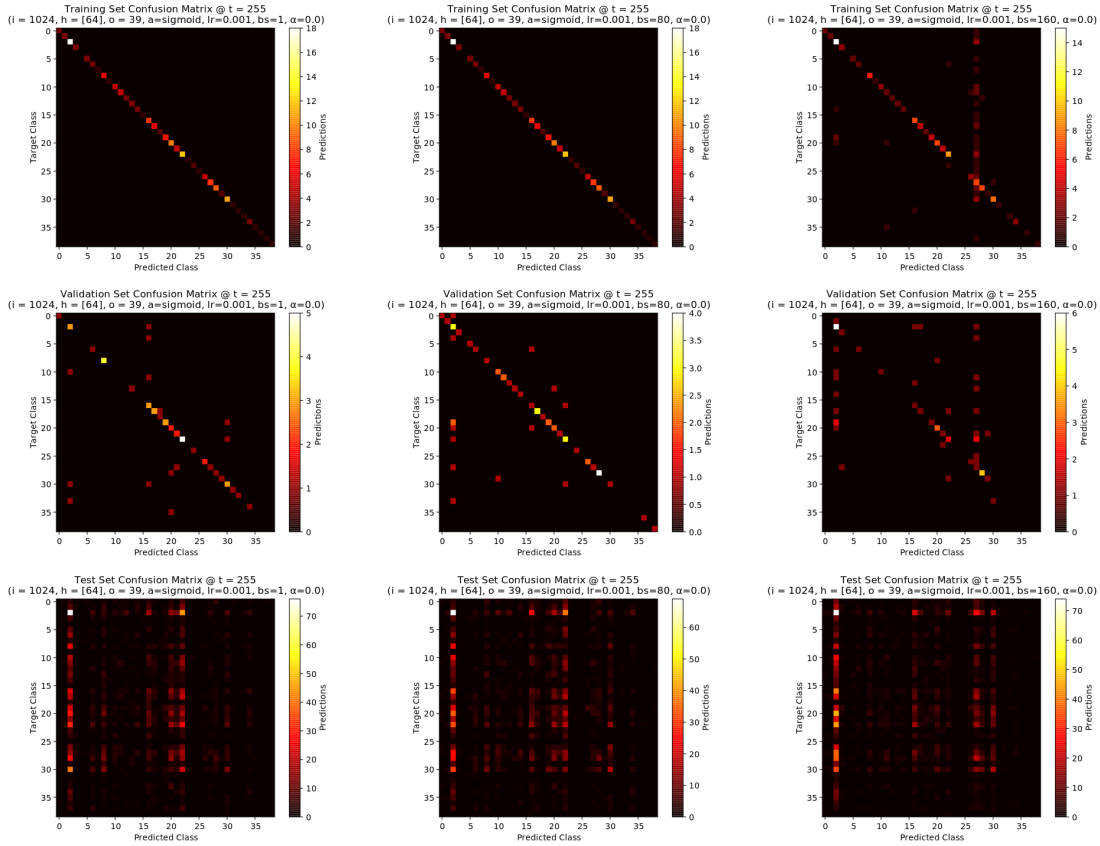

Figure 3.2.2: Confusion Matrices Varied by Batch Size

Variation in λ Varying the learning rate led to numerical instability for values greater than 0.001. Attempts were made to made to mitigate this, but there is a systemic issue in play here that requires more in-depth investigation. Consequently, the plots for weight changes and activations observed by varying the learning rate were not captured and are omitted from later sections.

As expected, the intermediate stages witnessed a slower decay in error than that observed in the baseline learning rate $\lambda = 0.001$ The confusion matrices from the last commonly plotted iteration in Fig 3.2.3 suffice for illustration.

The training runs with faster learning rates get quickly wedged in a local minimum that reflects a major statistical frequency in the training data. The case $\lambda = 0.01$ still has echoes of the more common class priors inducing errors by their prevalence, but a diagonal is beginning to at least emerge on the training and validation confusion matrices. The other cases are unlikely to escape the majority class, which may go toward explaining the numerical errors observed that prevented completion of those training runs.

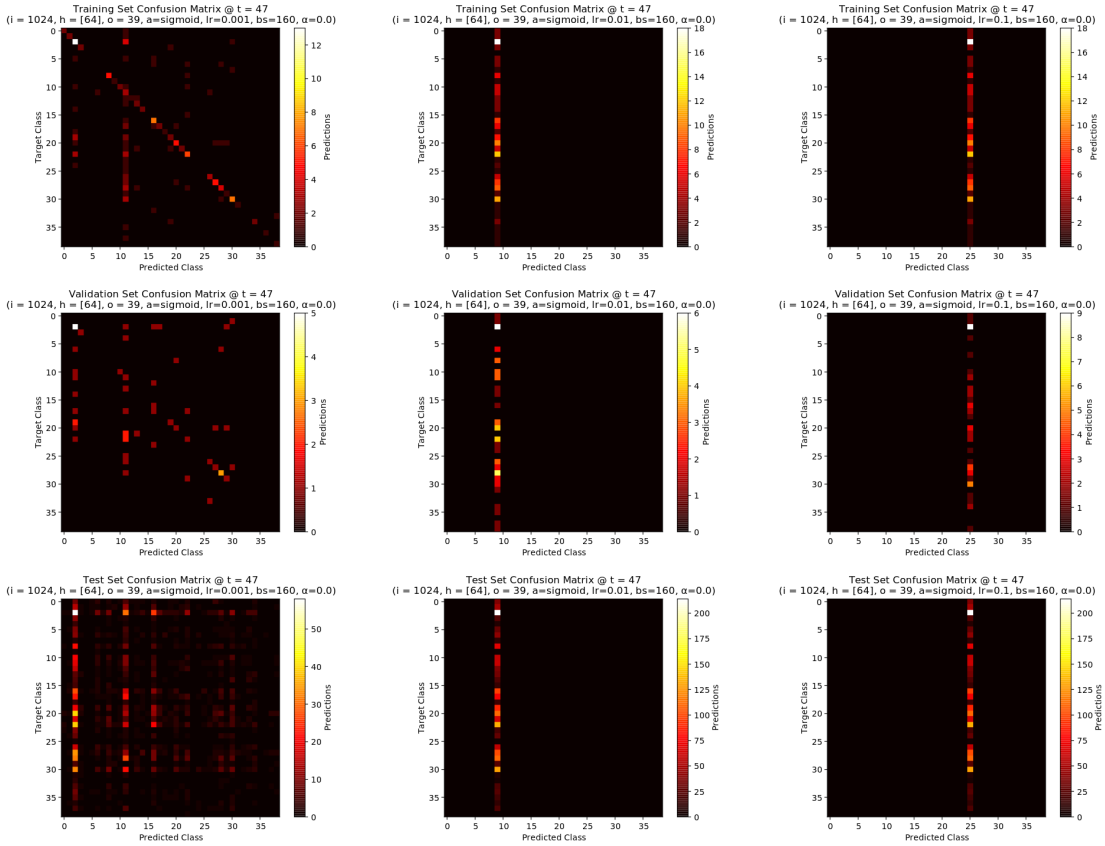

Figure 3.2.3: Confusion Matrices Varied by Learning Rate

Variation in N_H Fig 3.2.4 illustrates the variation in error resulting from a change in node count for the hidden layer. The lower, less expressive node counts appear to introduce significant jitter in the evolution of the error metric. This makes intuitive sense since a weight change in a single node has a greater potential impact on the hidden layer output when the nodes are few. There are fewer counterweights to counteract the mistakes of one hidden node. The lower count cases also reach the overfitting point much more quickly. There appears to be a linear relationship between the inflection point and the number of nodes. Assume a function f that maps node count to the inflection epoch, i.e., $f(N_H) = t_{overfit}$. In these examples, $f(1) \approx 75, f(4) \approx 100, f(16) \approx 175, f(64) > 256$.

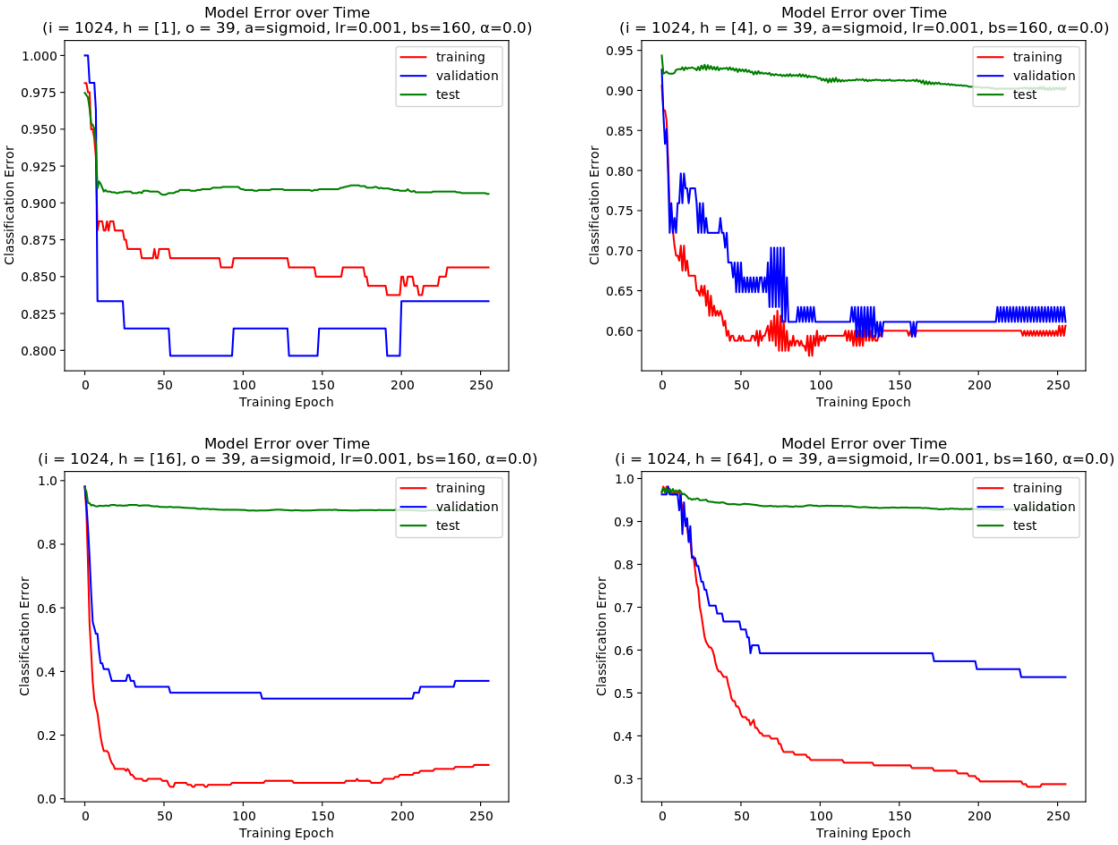

Figure 3.2.4: Classification Errors Varied by Hidden Layer Node Count

Variation in α Fig 3.2.5 illustrates the variation in error resulting from a change in momentum. From the juxtaposition of images, the momentum appears to have an inversely proportional relationship with stability, in keeping with its ostensible use as a means of overcoming plateaus in learning.

The *effect* could be compared to that observed from the constant reintroduction of a catalyst into a system so that the reagents never reach equilibrium, but ironically the *cause* is the diametric opposite. By dampening the weight changes, the system fails to converge quickly and follows a harmonic attenuation towards what is presumably an equilibrium state. The most textbook case of this is observed when $\alpha = 0.9$, which is unsurprisingly the value asserted as typical by Duda, Hart, and Stork in their text[8, p. 314].

Disappointingly, the final accuracy, even in the test set, is rather lower than what was observed for training with no momentum. Whatever local minima it may have avoided, the training with momentum seems to have settled on a suboptimal local minimum of its own.

The confusion matrices depicted in Fig 3.2.6 underscore the poor performance caused by adding momentum, particularly when $\alpha = 1.0$. The majority classes dominate more as momentum is increased, at least for the 256 epochs observed.

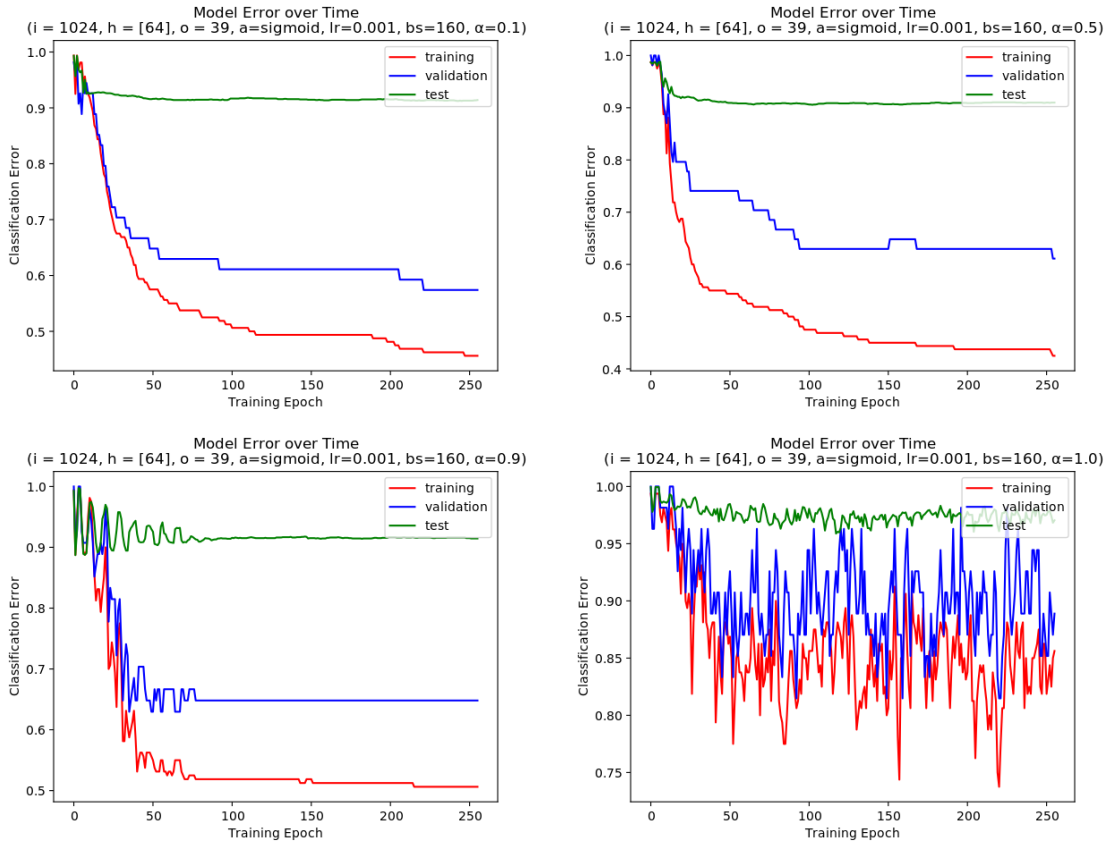

Figure 3.2.5: Classification Errors Varied by Momentum

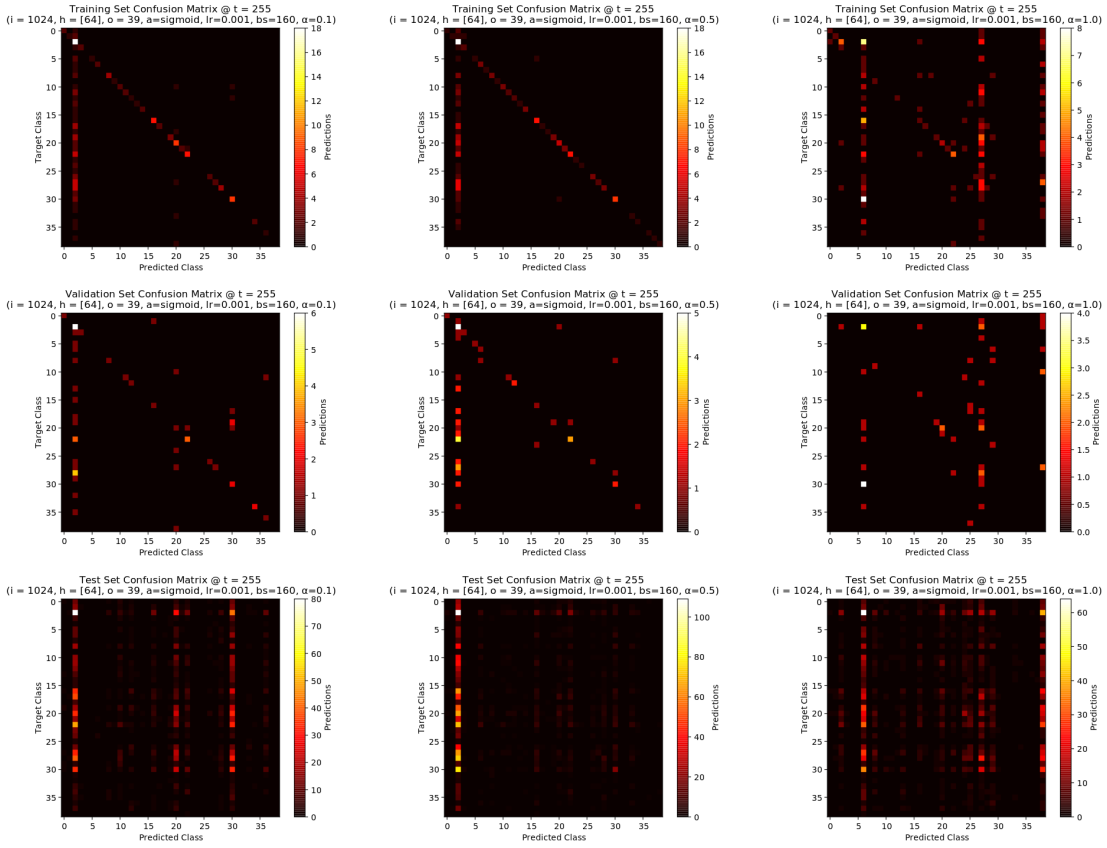

Figure 3.2.6: Confusion Matrices Varied by Momentum

Variation in u Fig 3.2.7 illustrates the variation in error resulting from a change in non-linear activation unit. The hyperbolic tangent easily outstrips the sigmoid in overfitting the training set in what appears to be approximately only 20 epochs. Of the regimens, tanh appears to be the most stable and consequently the most impervious to improvement. The validation and test accuracy is significantly better compared with the sigmoid in the same cohort, but better results were achieved in Fig 3.2.1 by smaller batch sizes.

The ReLU activation function appears to provide the best absolute accuracy on the validation set, but it clearly begins to overfit around epoch 150. It also appears to undergo periods of jitter where the weights are perhaps vacillating in response to the training sample order permutations.

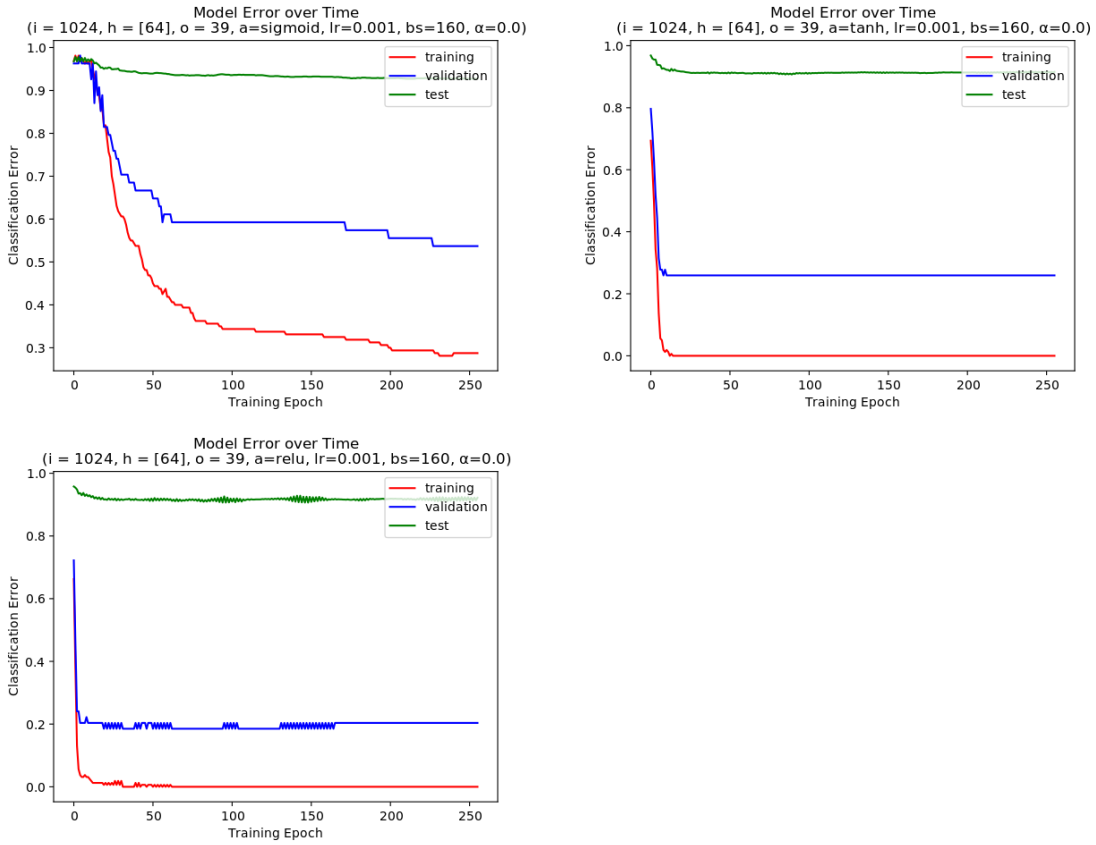

Figure 3.2.7: Classification Errors Varied by Activation Unit

3.3 Changes in Weight Over Time

To plot changes in layer weights over the training cycle, a heatmap was adopted to simultaneously illustrate the changes for all layers in a single concise graphic. A frequent observation was that the weight changes quickly rose in a crescendo in the first few epochs of training and then quickly dropped off and stabilized. The classification error follows a similar enough pattern that the error graph might appear proportional to a cross-section of the weight change heat map if it were extruded into three-dimensional space. Plots for weight vector norm difference per epoch were created, but they are omitted here as the information is redundant with the angle delta plots.

Variation in b Fig 3.3.1 illustrates the weight change history for variations in batch size. As batch size increases, the weight changes become more gradual and less pronounced. This follows intuitively since smaller batch sizes incur opportunities for more immediate backpropagation multiple times within an epoch. As such, the model changes to fit each sample, becoming quicker to overfit as the batch size decreases.

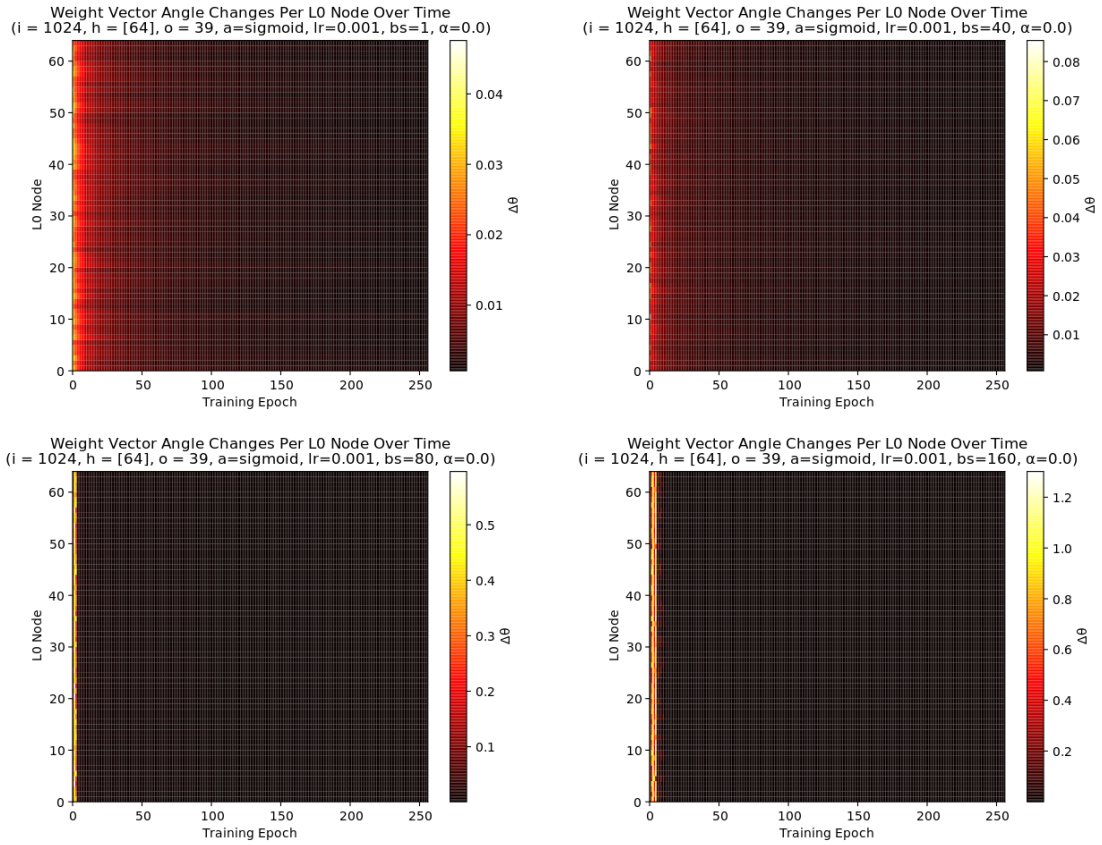

Figure 3.3.1: Weight Vector Changes Varied by Batch Size

Variation in N_H Fig 3.3.2 illustrates the weight change history for variations in hidden node count. With a single hidden node, the weight changes appear relatively small for the most part, although some of that may be an issue with artifacting in matplotlib that is masking the ostensible maxima. With four hidden nodes, it appears that nodes 0 and 2 incur the most sustained change over time, while nodes 1 and 3 remain relatively quiescent. As node count increases, the majority of the time is spent making minor adjustments with a burst of change in the early epochs. The burst is more pronounced in the $N_H = 64$ both in relative and absolute magnitude.

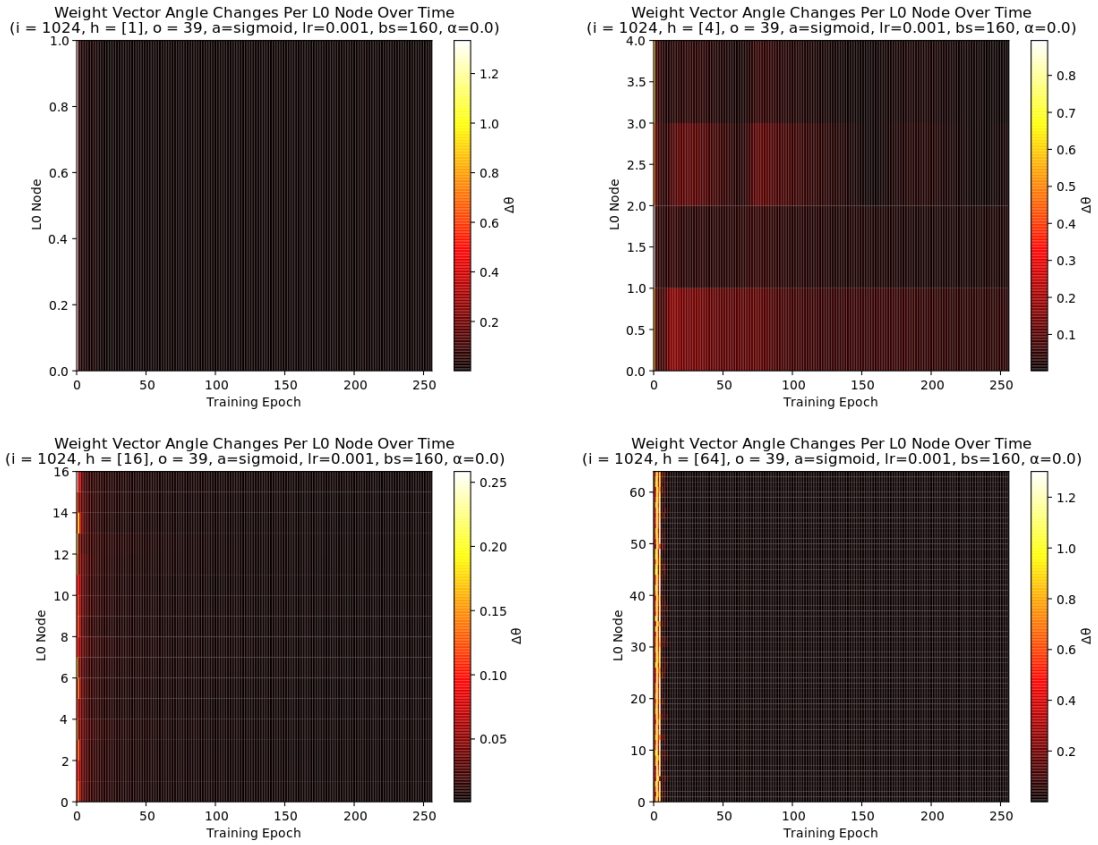

Figure 3.3.2: Weight Vector Changes Varied by Hidden Layer Node Count

Variation in α Fig 3.3.3 illustrates the weight change history for variations in momentum. There is little than can be remarked on with respect to the changes due to momentum given the controls for the other model parameters. The graph is essentially indistinguishable from the same graph in Fig 3.3.1 where the batch size, hidden node count, learning rate, and activation unit match. It may be that there are observable differences under different parameter control cohorts, but these were not explored on account of time and available computational resources.

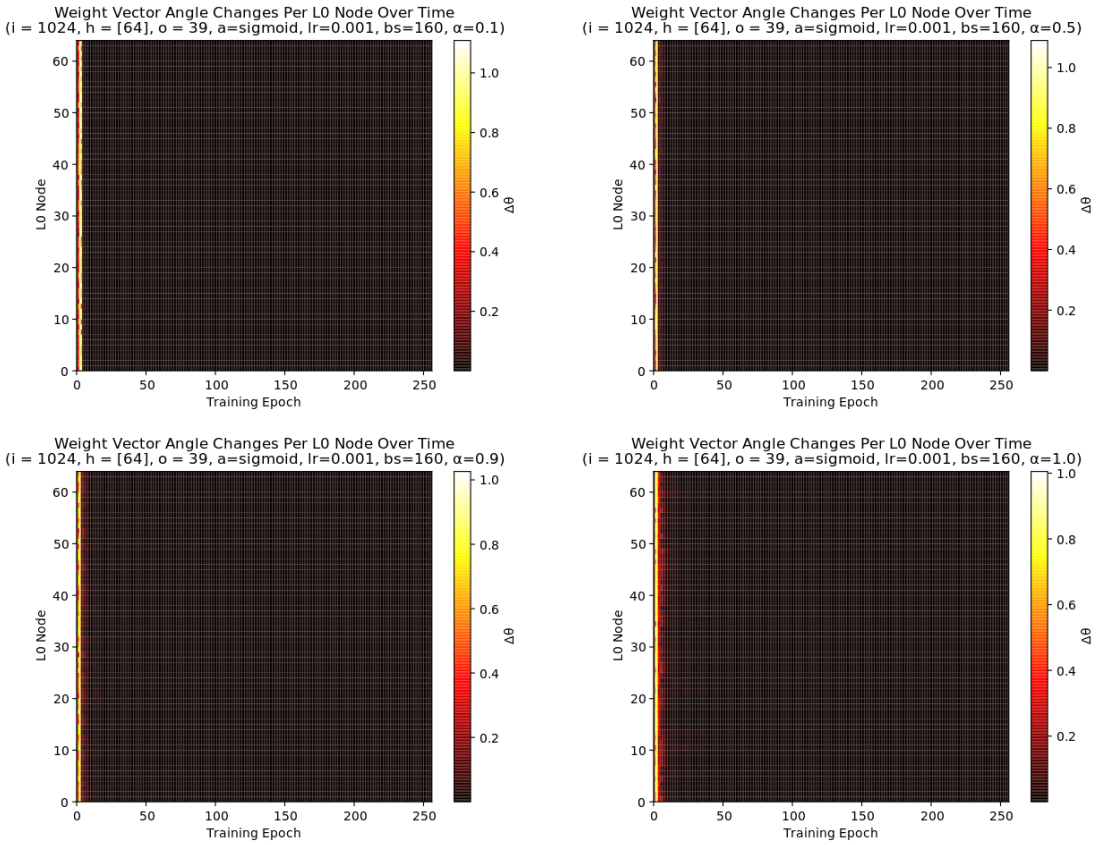

Figure 3.3.3: Weight Vector Changes Varied by Momentum

Variation in u Fig 3.3.4 illustrates the weight change history for variations in activation unit. The weight changes under the hyperbolic tangent are more diffused than under the sigmoid, though the scale is appreciably smaller – by more than half. The reduction in scale does permit a glimpse at distinguishing the weight changes over time across different nodes. What appears in the sigmoid to be uniform across the whole layer seems in the sigmoid to vary in decay with a few choice nodes undergoing an extended trail of revisions while most nodes go quiescent around the time of the inflection toward stabilization in the training set.

The ReLU activation function presents the most consistent fluctuation over time in the weights and resembles a seismograph or a spectrogram when plotting the respective nodes against time. The dynamic range is also smaller than for the sigmoid or tanh activation functions, and seems to support a notion of behavior that makes many small tweaks over time versus the bursty nature of the sigmoid with respect to weight change.

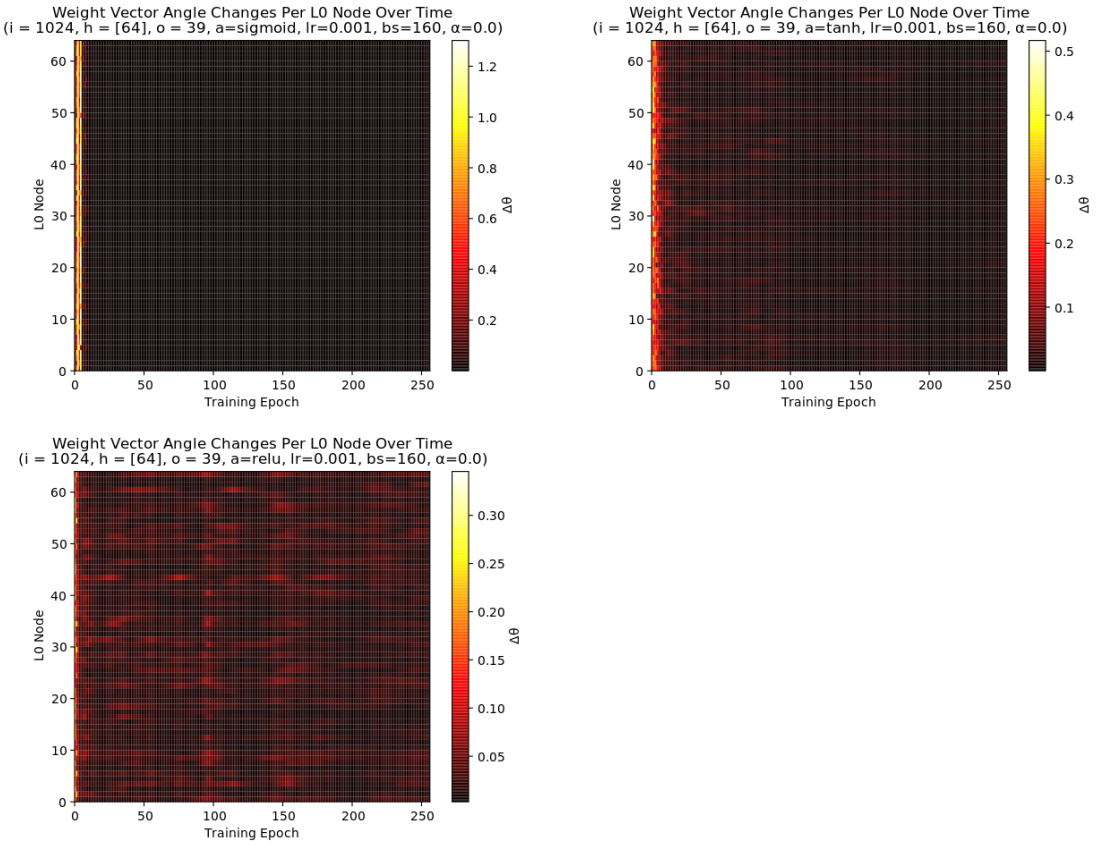

Figure 3.3.4: Weight Vector Changes Varied by Activation

3.4 Non-Linear Activations

Variation in b Fig 3.4.1 illustrates the output of the activation unit for each hidden input node over variations in batch size. When batch size is small, the distribution of activations appears to follow a normal distribution with mean at 0.5. As batch size increases, this gradually morphs into a bimodal distribution with peaks at 0 and 1. The distribution appear to be consistent across multiple nodes. The larger peak occurs around zero (0), which makes sense since the model is likely concentrating its "voting power" in the learned class outputs and zeroing out most others.

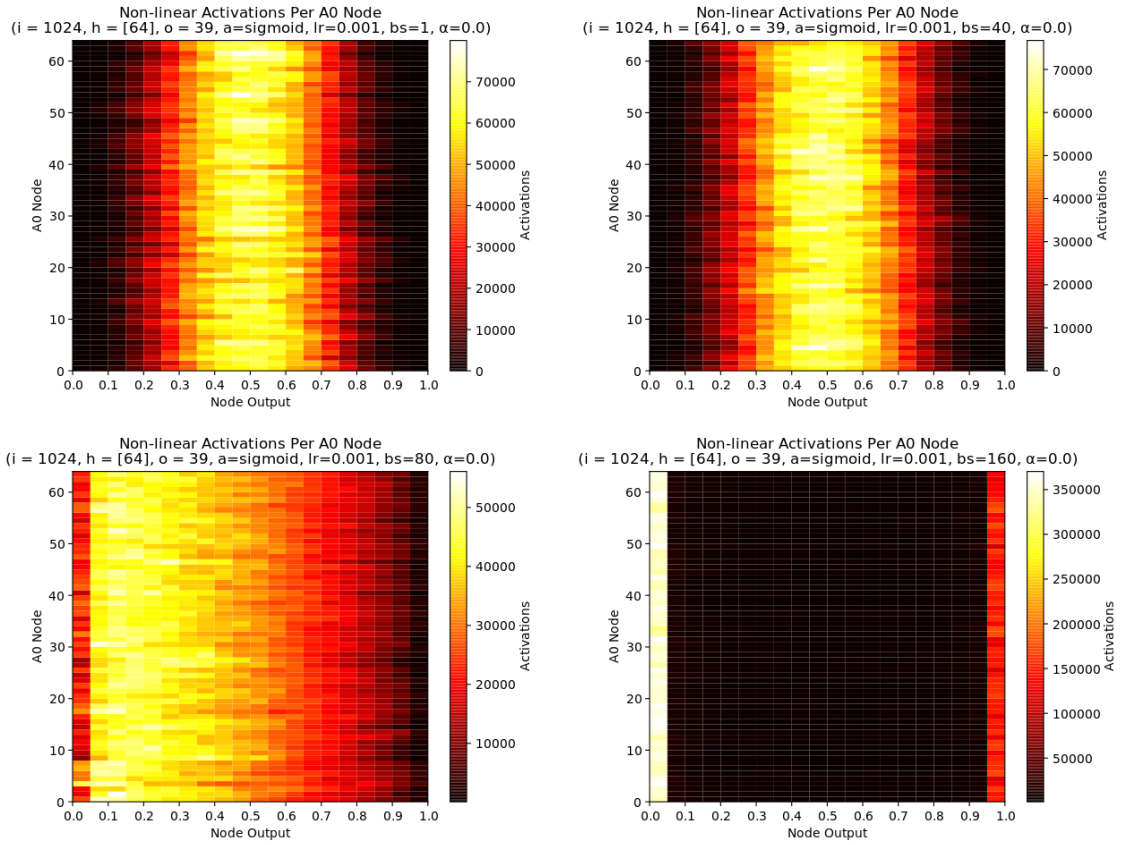

Figure 3.4.1: Activations Varied by Batch Size

Variation in N_H Fig 3.4.2 illustrates the output of the activation unit for each hidden input node over variations in hidden node count. The extrema in node count exhibit the bimodal distribution seen previously in the large batch size while the middle counts are closer to the normal distribution. The case where $N_H = 4$ appears to be more disorganized in its distribution than where $N_H = 16$, though this appears to correlate to the weight vector changes observed.

This suggests that a node subject to more emphatic swings in its weights over longer periods of time will reflect this in the form of a normal random activation pattern, while nodes which remain relatively cool will settle into a stable bimodal pattern. This may also correlate with the overfitting inflection point, which does not appear to be reached in the case where $N_H = 64$.

It should be noted that the $N_H = 1$, while bimodal, has a more gradual slope than $N_H = 64$. This supports the prior supposition that extended jitter diffuses the activations over a wider area. The single node case may also be exhibiting some degeneracy by virtue of it having to compress a 39-class decision into a single scalar that is then weighted accordingly by the output layer.

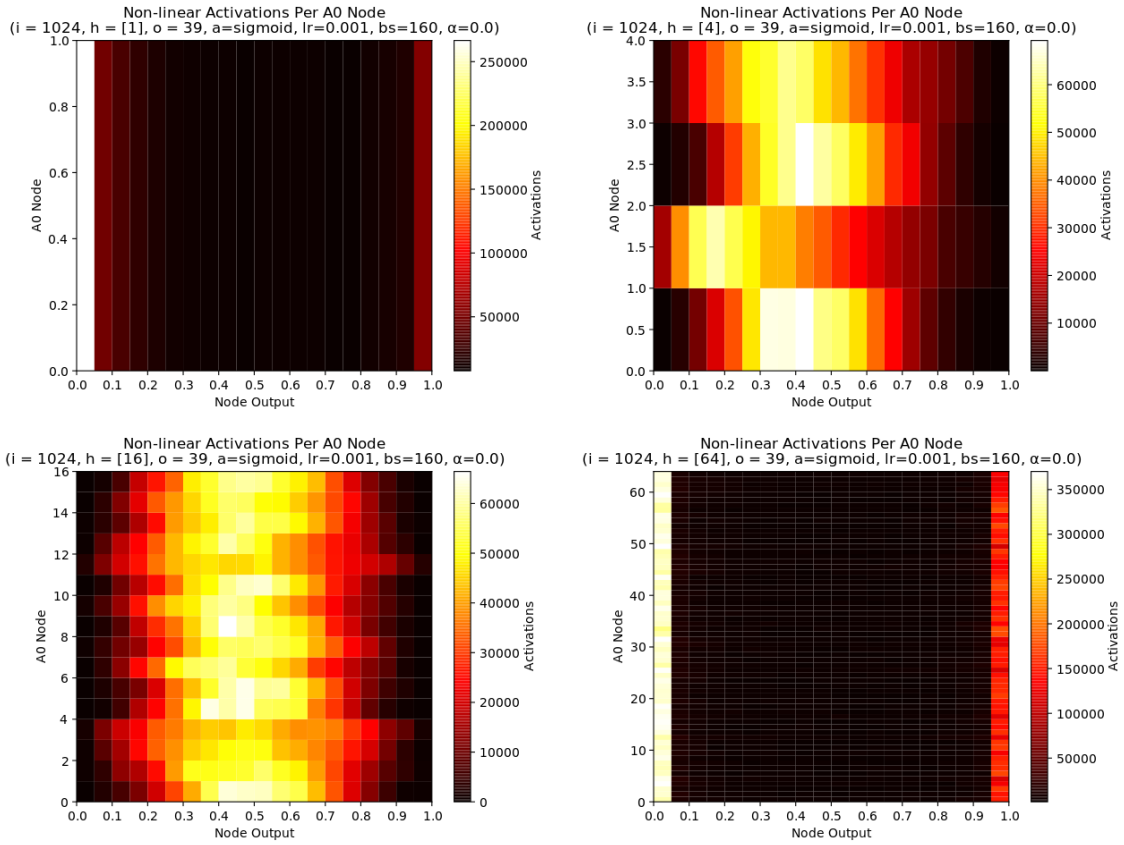

Figure 3.4.2: Activations Varied by Hidden Layer Node Count

Variation in α Fig 3.4.3 illustrates the output of the activation unit for each hidden input node over variations in momentum. The bimodal distribution of the comparable zero-momentum case is here preserved. What is interesting is the inversion of major and minor peaks when $\alpha = 1.0$. The discrepancy is not as large compared to the other cases, but the strong activations take precedence over the weak activations. The noisiness of the corresponding error graph in Fig 3.2.5 provides a clue: with $\alpha = 1.0$, the model never settles enough to confidently activate one node or a small handful of nodes in the output layer and leave the rest quiet as a mature model might. With the best accuracy *in the training set* briefly courting 25 percent, the model has simply not progressed enough.

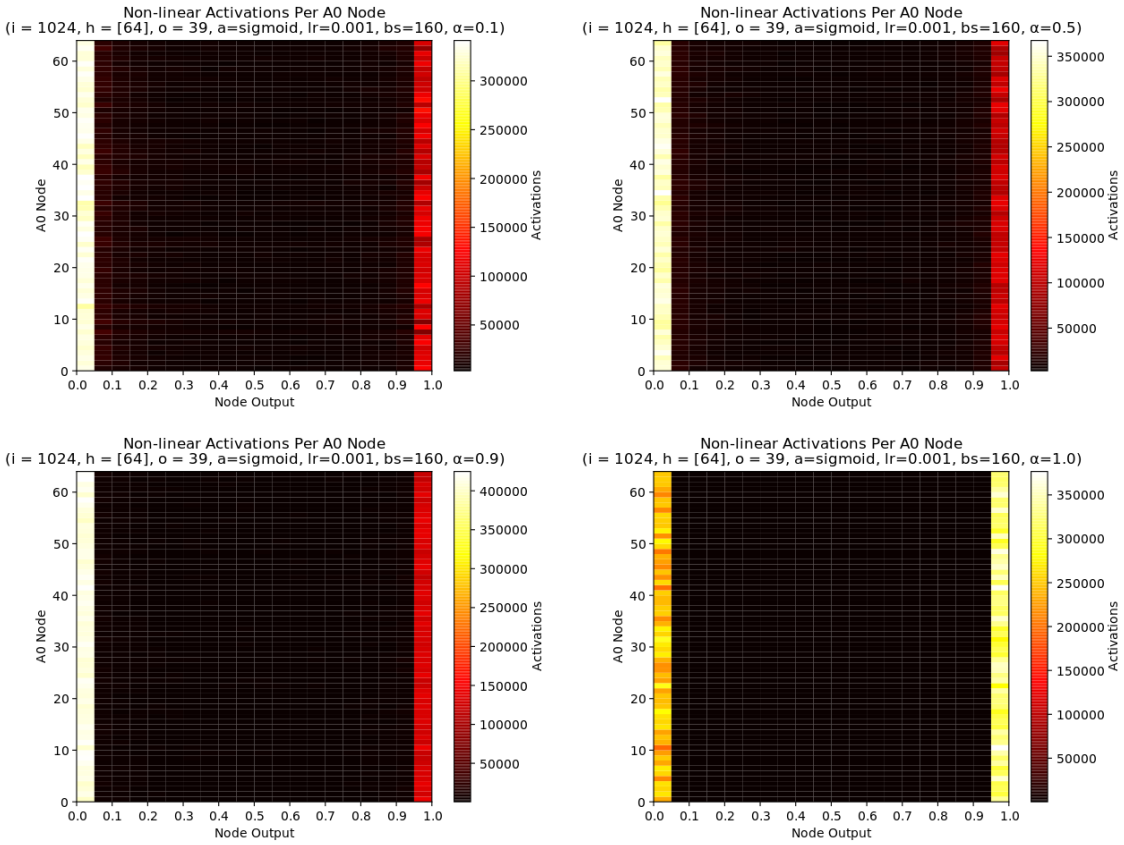

Figure 3.4.3: Activations Varied by Momentum

Variation in u Fig 3.4.4 illustrates the output of the activation unit for each hidden input node over variations in activation unit. The difference between the sigmoid and hyperbolic tangent is striking. While the sigmoid shows the bimodal distribution that suggests a confidence in output, tanh exhibits a nearly uniform distribution in activations, or at least a normal distribution with a high variance and not much representation at the extrema. One is left to wonder whether the performance of the hyperbolic tangent is truly good even at overfitting or if random chance played a significant role in selecting the right argmax values. A more rigorous study might help suss out the answer to these questions.

The ReLU plot shows an overwhelming preference for activations in the highest bin. This is an artifact of the implementation that includes some information loss. The ReLU function simply clamps negative values to zero but provides no upper bound. The plotting mechanism assumed a range of $[0, 1]$, and so several test runs crashed *in medias res*. The quickest fix was to clamp copies of the activation tensors used for plotting to an upper bound of 1, so the last bin technically represents any activation greater than 0.95. The ReLU function also distributes more heat to the lower bins than the sigmoid, though not as evenly as the hyperbolic tangent.

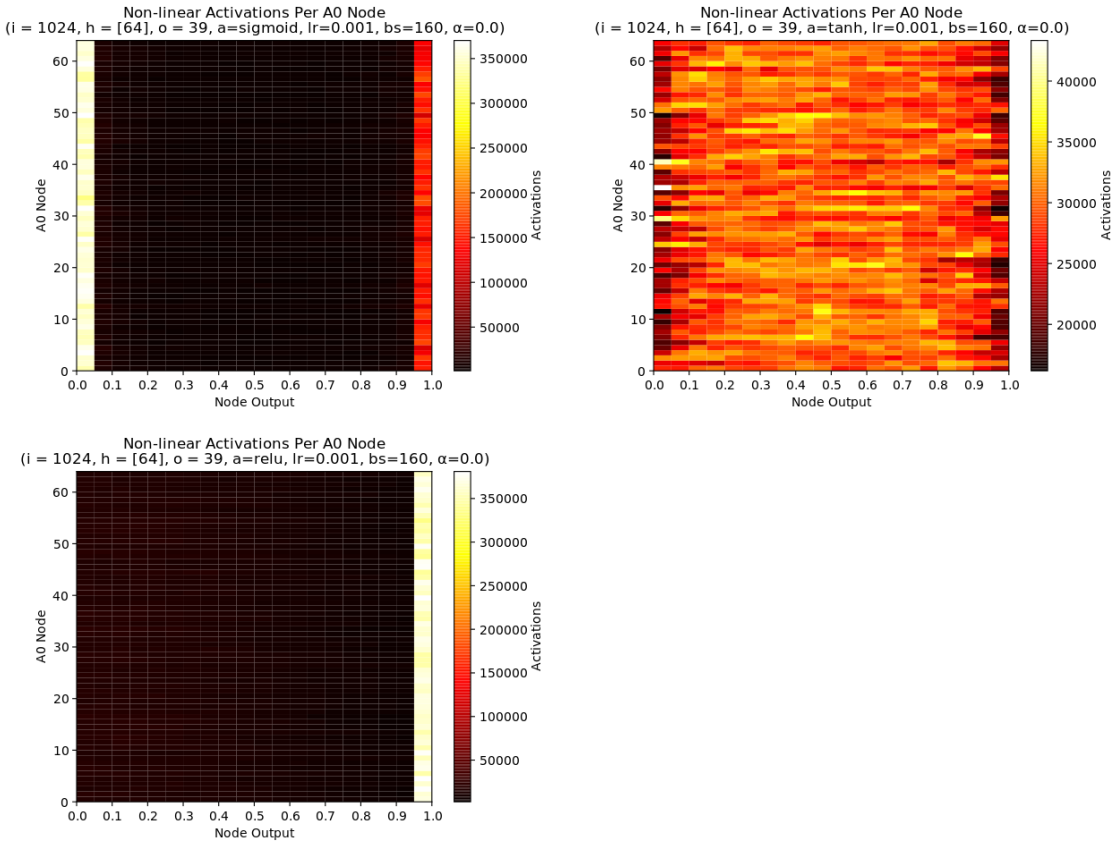

Figure 3.4.4: Activations Varied by Activation

4 Conclusions

Although the experiments were undertaken in the spirit of the Phoneme challenge presented by the University of East Anglia, the methodology of learning against a training set so small, particularly one dwarfed by the corresponding test set, was never likely to yield any notable accuracy. Most of the phoneme classes only included 4 samples while a few predominant phonemes had significantly higher populations, e.g., 18.

Further, a data-naive application of neural networks is only valuable insofar as the training set precisely captures the statistical relationship between the input features in reality and that these selfsame statistical relationships provide a sufficient test statistic for classification. The samples used for training came from a sanitized and somewhat ideal corpus, namely the straight reading of dictionary words. This could hardly be considered on its face to be strictly representative of utterances in the wild. Brief smoke tests of inverting the relationship between test set and training set were attempted but did not significantly improve the model's performance compared to a case similarly parameterized, e.g.,

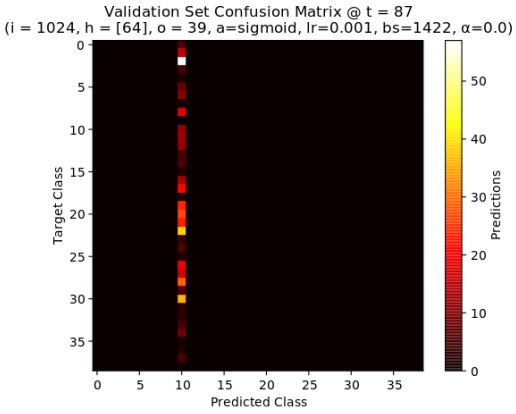

Figure 4.0.1: Confusion Matrix from Inverting the Training and Test Sets

There is an additional tacit but nonetheless potentially significant factor contributing to overfitting in this and similar experiments. The entire corpus from which the phoneme slices are derived is in English. Entire classes of phonemes exist which English speakers do not utter in their native tongue as lexical constituents, e.g., trills, clicks, implosives, and pharyngeal stops. Even if the experiments were moderately successful, any conclusions drawn would be constrained to Anglophonic applications.

In terms of performance, profiling with the standard Python profiling libraries revealed the the data-loading and the forward hook for activation histograms as the most time-intensive hotspots. Part of the difficulty in expediting the loading of the data is that the ARFF format is not conducive to direct ingestion into PyTorch tensors. The naive conversion from a heterogeneous numpy array through unzipping and tuple-slicing simply takes an inordinate amount of time relative to the training time. Each nominal class label, though technically the string representation of a number, had to be decoded from UTF-8 byte encodings.

The activation heatmap maintained separate bins for each hidden node and had to assign each hidden output to its respective bin in the respective node, an operation with cubic asymptotic complexity dimensioned by layers, layer nodes, and sample iterations. It might be worth investigating whether a row- or column-wise version of the Pytorch `tensor.histc` operation exists that could mitigate some of the looping cost, to say nothing of the forward hook overhead. A better design might have been to extend the Sigmoid, ReLU, and Tanh modules with innate state for tracking histograms inside the forward override itself.

Let us assume for the sake of argument that having considered the issues mentioned above that the goal is simply to construct a decently performing English phoneme classifier for preconstrained, normalized sample windows. Whereas amplitude values provide the basis for the spectral criteria used in most acoustic language processing frameworks, the translation from the time domain into the frequency domain requires a convolution of the input. This would be better represented through a complete graph of the input nodes as opposed to a mere feedforward network. At that point, one may as well compute the Fourier transform, a thoroughly analyzed and widely adopted computation specially suited to acoustic modeling, and be done with it rather than spend epochs accumulating vast corpora and fiddling uninterpretable parameters.

Though the answer to the question posited by the title may appear self-evident, the experiments undertaken provided an opportunity for a data-driven discourse in the applicability of neural networks and the dangers of using them indiscriminately. Douglas Hofstadter touches

on these and similar issues succinctly yet comprehensively in an article on his experiences with Google Translate[13]. Is amplitude (or more generally, data) all you need? In a word, no.

References

[1] Anthony Bagnall et al. *The UEA multivariate time series classification archive, 2018*. Oct. 31, 2018. URL: https://arxiv.org/pdf/1811.00075.pdf (visited on 09/25/2019).

[2] William Chan et al. *Listen, Attend and Spell*. Aug. 20, 2015. URL: https://arxiv.org/abs/1508.01211 (visited on 10/06/2019).

[3] The SciPy Community. *scipy.io.arff.loadarff*. Sept. 27, 2019. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.arff.loadarff.html (visited on 09/27/2019).

[4] Torch Contributors. *PyTorch: NN*. Sept. 27, 2019. URL: https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_nn.html (visited on 09/27/2019).

[5] Torch Contributors. *torch.nn*. Sept. 27, 2019. URL: https://pytorch.org/docs/stable/nn.html (visited on 09/27/2019).

[6] Torch Contributors. *torch.tensor*. Sept. 28, 2019. URL: https://pytorch.org/docs/stable/tensors.html (visited on 09/28/2019).

[7] *Document torch.acos() behavior near -1 and 1*. Aug. 22, 2019. URL: https://github.com/pytorch/pytorch/issues/8069 (visited on 09/29/2019).

[8] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. 2nd ed. New York, NY: Wiley-Interscience (John Wiley & Sons, Inc.), 2001.

[9] Python Software Foundation. *functools - Higher-order functions and operations on callable objects*. Oct. 2, 2019. URL: https://docs.python.org/3.6/library/functools.html (visited on 10/06/2019).

[10] Hossein Hamooni and Abdullah Mueen. *Dataset: Phoneme*. Ed. by William Vickers. Aug. 9, 2019. URL: http://www.timeseriesclassification.com/description.php?Dataset=Phoneme (visited on 09/25/2019).

[11] Hossein Hamooni and Abdullah Mueen. *Dual-domain Hierarchical Classification of Phonetic Time Series*. University of New Mexico. URL: https://www.cs.unm.edu/~hamooni/papers/Dual_2014/Phoneme.pdf (visited on 09/25/2019).

[12] Awni Hannun et al. *Deep Speech: Scaling up end-to-end speech recognition*. Dec. 19, 2014. URL: https://arxiv.org/abs/1412.5567 (visited on 10/06/2019).

[13] Douglas Hofstadter. *The Shallowness of Google Translate*. The Atlantic. Jan. 30, 2018. URL: https://www.theatlantic.com/technology/archive/2018/01/the-shallowness-of-google-translate/551570/ (visited on 10/09/2019).

[14] John Hunter et al. *matplotlib.colorbar*. May 18, 2019. URL: https://matplotlib.org/api/_as_gen/matplotlib.colorbar.html (visited on 10/05/2019).

[15] John Hunter et al. *matplotlib.pyplot.imshow*. Aug. 26, 2019. URL: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html (visited on 10/05/2019).

[16] John Hunter et al. *matplotlib.pyplot.pcolormesh*. Aug. 26, 2019. URL: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.pcolormesh.html (visited on 10/06/2019).

[17] John Hunter et al. *matplotlib.pyplot.plot*. Aug. 26, 2019. URL: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html (visited on 10/05/2019).

[18] David C. Lay. *Linear Algebra and its applications*. 4th ed. Boston, MA: Addison-Wesley, 2012.

[19] D. Oliver. *Torch: how to shuffle a tensor by its rows?* URL: https://stackoverflow.com/questions/44738273/torch-how-to-shuffle-a-tensor-by-its-rows (visited on 10/02/2019).

[20] *PyTorch - Best way to get at intermediate layers in VGG and ResNet*. URL: https://forums.fast.ai/t/pytorch-best-way-to-get-at-intermediate-layers-in-vgg-and-resnet/5707/6 (visited on 10/06/2019).

[21] Ashish Vaswani et al. *Attention Is All You Need*. URL: https://arxiv.org/abs/1706.03762 (visited on 10/06/2019).

5 Appendix A - PINNIPED Source Code

The complete source for PINNIPED is given as a single Python source file and included here for convenience.

```
#!/usr/bin/python3

import sys
import argparse
import scipy.io.arff
import torch
import math
from collections import OrderedDict
import pickle
import matplotlib
matplotlib.use('cairo')
import matplotlib.pyplot as mplp
import matplotlib.colors as mplc
import functools
import os.path

"""
Command line arguments
"""
parser = argparse.ArgumentParser(description='Parameterized Interactive Neural Network Iteratively Plotting Experimental Decisions')
parser.add_argument('--train-arff', type=str)
parser.add_argument('--test-arff', type=str)

parser.add_argument('--model-file', type=str, default='model.nn')
parser.add_argument('--epochs', type=int, default=1)
parser.add_argument('--batch-size', type=int, default=1)
parser.add_argument('--learning-rate', type=float, default=0.5)
parser.add_argument('--momentum', type=float, default = 0.0)
parser.add_argument('--activation-unit', type=str.lower, choices=['sigmoid', 'tanh', 'relu'], default='sigmoid')
parser.add_argument('--layer-dims', type=str, required=True)

reserved = parser.add_mutually_exclusive_group()
reserved.add_argument('--reserve-frac', type=float, default=None)
reserved.add_argument('--reserve-every', type=int, default=None)

parser.add_argument('--autograd-backprop', action='store_true')
parser.add_argument('--sgd', action='store_true')
```

```
parser.add_argument('--interactive', action='store_true')
parser.add_argument('--debug', action='store_true')
parser.add_argument('--activation-bins', type=int, default=20)
parser.add_argument('--plot-every', type=int, default=1)
parser.add_argument('--plot-confusion-every', type=int, default=1)
parser.add_argument('--workspace-dir', type=str, default='.')
parser.add_argument('--save-every', type=int, default=1)
parser.add_argument('--profile', action='store_true')

activation_units = { 'sigmoid' : torch.nn.Sigmoid, 'tanh' : torch.nn.Tanh, 'relu' : torch.nn.ReLU }

"""
Encode a series of amplitudes in the range [0,1] according to the series dynamic range.
"""
def compress(x):
    dmax = max(x)
    dmin = min(x)
    return torch.tensor([(d - dmin)/(dmax - dmin) for d in x]).type(torch.double)

"""
Load training data from ARFF input file.
"""
def load_arff(arff_fname):
    arff_data, arff_meta = scipy.io.arff.loadarff(arff_fname)
    classes = len(arff_meta['target'][1])

    # Convert ARFF to torch tensors.
    X, Y = zip(*[ (XY[0:-1], XY[-1].decode('UTF-8')) for XY in [ tuple(nd) for nd in arff_data ] ])
    L = OrderedDict()
    # This mapping is a precaution in case the class labels are not a monotonic sequence corresponding to the indices.
    for c in range(classes):
        y = arff_meta['target'][1][c]
        L[y] = one_hots[c]
    X = torch.stack([compress(x) for x in X ])
    Y = torch.stack([ L[y] for y in Y ])
    return X, Y, L

def weight_change(W_i, W_j, dWNorm, dTheta):
    for layer in range(len(W_j)):
        dwnorm = torch.zeros(W_j[layer].size()[0], 1).to(torch.double)
        dtheta = torch.zeros(W_j[layer].size()[0], 1).to(torch.double)
        for node in range(W_j[layer].size()[0]):
            dwnorm[node] = torch.norm(W_j[layer][node].sub(W_i[layer][node])).item()
            dtheta[node] = d_theta(W_i[layer][node],W_j[layer][node])
        dWNorm[layer] = torch.cat((dWNorm[layer], dwnorm), 1)
        dTheta[layer] = torch.cat((dTheta[layer], dtheta), 1)


def d_theta(w_i, w_j):
    norm_w_ij = torch.norm(w_i) * torch.norm(w_j)
    if norm_w_ij.item() == 0.0:
        # Any zero-length vector can be considered to be simultaneously colinear and not with any other.
        # Erring on the side of colinearity and avoiding div by zero.
        return 0.0
    else:
        # Clamping per https://github.com/pytorch/pytorch/issues/8069
        return torch.acos(torch.clamp(torch.dot(w_i, w_j) / norm_w_ij, min=-1.0, max=1.0)).item()

"""
Train NN.
"""
def train_nn(model, X, Y, test_X, test_Y):
    classes = model[-1].out_features
    sample_count = X.size()[0]
    trained_confusion = torch.zeros(classes, classes).type(torch.int)
    validated_confusion = torch.zeros(classes, classes).type(torch.int)
    tested_confusion = torch.zeros(classes, classes).type(torch.int)
    sample_indices = torch.arange(0,sample_count)
    grad_bias = [p.data.new_zeros(p.size()).requires_grad_(False).type(torch.double) for p in model.parameters()]
```

```
if args.autograd_backprop:
    for layer in [layer for layer in model.children() if type (layer) if type(layer) is torch.nn.Linear]:
        if args.debug:
            print(layer.weight.data, file=sys.stderr)
else:
    for layer in [layer for layer in model.children() if type (layer) if type(layer) is torch.nn.Linear]:
        layer.weight.data.copy_(torch.rand_like(layer.weight.data))
        layer.weight.data -= 0.5
        layer.weight.data *= 0.2
        if args.debug:
            print(layer.weight.data, file=sys.stderr)

if args.sgd:
    # Stochastic gradient descent: shuffle
    sample_indices = torch.randperm(sample_count)

if args.reserve_frac is not None:
    # Reserve the last fraction
    reserved = math.ceil(sample_count * args.reserve_frac)
    training_indices = sample_indices[0:-reserved]
    reserved_indices = sample_indices[-reserved:]
elif args.reserve_every is not None:
    # Reserve at regular intervals
    training_indices = torch.stack([i for i in sample_indices if i % args.reserve_every != 0])
    reserved_indices = sample_indices[0::args.reserve_every]

optimizer = torch.optim.SGD(model.parameters(), lr=args.learning_rate, momentum=args.momentum)


trained_N = len(training_indices)
reserved_N = len(reserved_indices)
tested_N = len(test_Y)
trained_error = []
validated_error = []
tested_error = []
indices = {}

hits = {}
accuracy = {}

dWNorm = [ torch.empty(layer.weight.data.size()[0], 0).to(torch.double) for layer in
           [ layer for layer in model.children() if type(layer) is torch.nn.Linear ] ]
dTheta = [ torch.empty(layer.weight.data.size()[0], 0).to(torch.double) for layer in
           [ layer for layer in model.children() if type(layer) is torch.nn.Linear ] ]

for epoch in range(args.epochs):
    if args.sgd:
        # Stochastic gradient descent: shuffle
        training_indices = torch.stack([training_indices[i] for i in torch.randperm(trained_N)])
    passed = 0
    failed = 0
    LW_i = [ layer.weight.data.clone().detach().requires_grad_(False).to(torch.double) for layer in
             [ layer for layer in model.children() if type(layer) is torch.nn.Linear ] ]
    # Permute the order of the data for stochastic batch descent.
    reserved_X = X.index_select(0, reserved_indices)
    reserved_Y = Y.index_select(0, reserved_indices)

    batch = 0
    offset = 0
    training_X = X.index_select(0, training_indices)
    training_Y = Y.index_select(0, training_indices)

    while offset < trained_N:
        # Zero out gradients at the start of each batch.
        if args.autograd_backprop:
            optimizer.zero_grad()
        else:
            with torch.no_grad():
```

```
                model.zero_grad()

        # Map training set input and target to the current batch based on pre-permuted index order.
        batch_indices = training_indices[offset:offset+args.batch_size]
        batch_X = X.index_select(0, batch_indices)
        batch_Y = Y.index_select(0, batch_indices)

        # Train the model on the input X
        trained_Y = model(batch_X)

        # Calculate losses for back-propagation.
        loss = loss_fn(trained_Y, batch_Y)
        loss.backward()

        # Backpropagation of loss gradients.
        if args.autograd_backprop:
            optimizer.step()
        else:
            # Turn off autograd so as to not pollute the gradients during back-prop.
            with torch.no_grad():
                b_p = 0
                for p in model.parameters():
                    grad_bias[b_p].copy_(args.learning_rate * (1 - args.momentum) * p.grad + args.momentum * grad_bias[b_p])
                    p -= grad_bias[b_p]
                    b_p += 1
        offset += args.batch_size

    # Test predictions on training set at the end of the epoch. Gains or losses between batches within an epoch are not captured.
    trained_hits, trained_accuracy = test_batch(model, training_X, training_Y, trained_confusion)
    # Test predictions on validation set.
    validated_hits, validated_accuracy = test_batch(model, reserved_X, reserved_Y, validated_confusion)
    # Test predictions on test set if one has been supplied.
    if test_X is not None and test_Y is not None:
        tested_hits, tested_accuracy = test_batch(model, test_X, test_Y, tested_confusion)

    # Report epoch results.
    trained_error.append(1.0 - trained_accuracy)
    print("TRAIN[{}]: {}/{} ({})".format(epoch, trained_hits, trained_N, trained_accuracy), file=sys.stderr)
    if args.debug:
        print_confusion_matrix(trained_confusion)

    validated_error.append(1.0 - validated_accuracy)
    print("VALID[{}]: {}/{} ({})".format(epoch, validated_hits, reserved_N, validated_accuracy), file=sys.stderr)
    if args.debug:
        print_confusion_matrix(validated_confusion)

    if test_X is not None and test_Y is not None:
        tested_accuracy = float(tested_hits) / float(tested_N)
        tested_error.append(1.0 - tested_accuracy)
        print("TEST[{}]: {}/{} ({})".format(epoch, tested_hits, tested_N, tested_accuracy), file=sys.stderr)
        if args.debug:
            print_confusion_matrix(validated_confusion)


    # Calculate weight changes over the epoch.
    LW_j = [ layer.weight.data.clone().detach().requires_grad_(False).to(torch.double) for layer in
             [ layer for layer in model.children() if type(layer) is torch.nn.Linear ] ]
    weight_change(LW_i, LW_j, dWNorm, dTheta)
    if ((epoch + 1) % args.plot_every == 0) or ((epoch + 1) % args.plot_confusion_every == 0) or (epoch + 1 == args.epochs):
        plot_confusion_matrix(model, epoch, 'training', trained_confusion)
        plot_confusion_matrix(model, epoch, 'validation', validated_confusion)
        if args.test_arff is not None:
            plot_confusion_matrix(model, epoch, 'test', tested_confusion)

    if ((epoch + 1) % args.plot_every == 0) or (epoch + 1 == args.epochs):
        plot_training_validation_accuracy(model, epoch, trained_error, validated_error, tested_error)
        plot_weight_angle_changes(model, epoch, dTheta)
        plot_weight_magnitude_changes(model, epoch, dWNorm)
        plot_activation_heatmap(model, epoch, activations)
```

```
            if (args.interactive):
                show_combined_plots(epoch)

        # Reset confusion matrices for next epoch.
        trained_confusion.fill_(0)
        validated_confusion.fill_(0)
        tested_confusion.fill_(0)

        if ((epoch + 1) % args.save_every == 0):
            with open(os.path.join(args.workspace_dir, '{}.{}'.format(args.model_file, epoch)), 'wb') as fout:
                pickle.dump(model.state_dict(), fout)


def show_combined_plots(epoch):
    return

def plot_training_validation_accuracy(model, epoch, trained_error, validated_error, tested_error):
    mplp.plot(list(range(len(trained_error))), trained_error, 'r-')
    mplp.plot(list(range(len(validated_error))), validated_error, 'b-')
    legend_labels=('training', 'validation')
    if len(tested_error) != 0:
        legend_labels += ('test',)
        mplp.plot(list(range(len(tested_error))), tested_error, 'g-')
    mplp.legend(labels=legend_labels, loc='upper right')
    mplp.xlabel('Training Epoch')
    mplp.ylabel('Classification Error')
    mplp.title('Model Error over Time\n{}'.format(model_params_shorthand))
    mplp.savefig(os.path.join(args.workspace_dir, 'error-{}.png'.format(epoch)))
    mplp.close()

def plot_weight_angle_changes(model, epoch, dTheta):
    layers = [n for n, c in model.named_children() if type(c) is torch.nn.Linear]
    i = 0
    for dtheta in dTheta:
        mplp.pcolormesh(dtheta.numpy(), cmap='hot')
        mplp.colorbar(label='Δ ')
        mplp.xlabel('Training Epoch')
        mplp.ylabel('{} Node'.format(layers[i]))
        mplp.title('Weight Vector Angle Changes Per {} Node Over Time\n{}'.format(layers[i], model_params_shorthand))
        mplp.savefig(os.path.join(args.workspace_dir, 'weight-angle-changes-{}-{}.png'.format(layers[i], epoch)))
        mplp.close()
        i+=1

def plot_weight_magnitude_changes(model, epoch, dWNorm):
    layers = [n for n, c in model.named_children() if type(c) is torch.nn.Linear]
    i = 0
    for dwnorm in dWNorm:
        mplp.pcolormesh(dwnorm.numpy(), cmap='hot')
        mplp.colorbar(label='Δ||w||')
        mplp.xlabel('Training Epoch')
        mplp.ylabel('{} Node'.format(layers[i]))
        mplp.title('Weight Vector Norm Changes Per {} Node Over Time\n{}'.format(layers[i], model_params_shorthand))
        mplp.savefig(os.path.join(args.workspace_dir, 'weight-magnitude-changes-{}-{}.png'.format(layers[i], epoch)))
        mplp.close()
        i += 1

def plot_confusion_matrix(model, epoch, which, confusion_matrix):
    mplp.imshow(confusion_matrix.numpy(), cmap='hot')
    mplp.colorbar(label='Predictions')
    mplp.xlabel('Predicted Class')
    mplp.ylabel('Target Class')
    mplp.title('{} Set Confusion Matrix @ t = {}\n{}'.format(which.capitalize(), epoch, model_params_shorthand))
    mplp.savefig(os.path.join(args.workspace_dir, 'confusion-matrix-{}-{}.png'.format(which, epoch)))
    mplp.close()

def print_confusion_matrix(M):
    s = ""
    for r in range(M.size()[0]):
        s += "|"
```

```
        for c in range(M.size()[1]):
            if M[r][c] == 0:
                mark = ' '
            else:
                mark = '+' if r == c else '-'
            s += "{}{}|".format(mark, M[r][c] if M[r][c] > 0 else ' ')
        s += "\n"
    print(s)

def plot_activation_heatmap(model, epoch, activations):
    for layer in activations:
        mplp.pcolormesh(activations[layer].numpy(), cmap='hot')
        mplp.colorbar(label='Activations')
        mplp.xlabel('Node Output')
        mplp.ylabel('{} Node'.format(layer))
        bin_ticks = [x for x in range(args.activation_bins + 1) if (x % (args.activation_bins/10)) == 0]
        mplp.xticks(bin_ticks, [ b / args.activation_bins for b in bin_ticks])
        mplp.title('Non-linear Activations Per {} Node\n{}'.format(layer, model_params_shorthand))
        mplp.savefig(os.path.join(args.workspace_dir, 'activations-{}-{}.png'.format(layer, epoch)))
        mplp.close()

def test_batch(model, X_in, Y_tru, M_confusion):
    with torch.no_grad():
        Y_out = model(X_in)
        Y_lbl = torch.stack([one_hots[y.argmax().item()] for y in Y_out])
        hits = Y_tru.eq(Y_lbl).all(1).to(torch.int).sum()
        for i in range(Y_tru.size()[0]):
            M_confusion[Y_tru[i].argmax().item()][Y_out[i].argmax().item()] += 1
    return (hits, float(hits) / float(Y_tru.size()[0]))

"""
Test NN.
"""
def test_nn(model, X, Y):
    M_confusion = torch.zeros(model[-1].out_features, model[-1].out_features)
    hits, accuracy = test_batch(model, X, Y, M_confusion)
    print("TEST: {}/{} ({})".format(hits, Y.size()[0], accuracy ), file=sys.stderr)
    if args.debug:
        print_confusion_matrix(M_confusion)

def capture_hidden_outputs_hook(module, features_in, features_out, **kwargs):
    for y in features_out:
        d = 0
        # ReLU may provide values greater than 1. This clamps them for the purposes of graphing.
        y_clamped = y.clone().detach().requires_grad_(False).clamp(min=0.0, max=1.0)
        for f in y_clamped:
            f_bin = math.ceil(args.activation_bins * f) - 1
            activations[kwargs['name']][d][f_bin] += 1
            d += 1
    return None

"""
Main
"""

args = parser.parse_args()

if args.profile:
    import cProfile, pstats, io, pstats
    pr = cProfile.Profile()
    pr.enable()


os.makedirs(args.workspace_dir, mode=0o770, exist_ok=True)

# Get layer dims.
layer_D = [ int(D) for D in args.layer_dims.split(",") ]

if len(layer_D) < 2:
```

```
    raise argparse.ArgumentError("Must define at least two layer dimensions (input + output)")

# Determine activation unit.
activation_unit = activation_units[args.activation_unit]

# Create shorthand string to describe model params (useful in plots).
model_params_shorthand = '(i = {}, h = {}, o = {}, a={}, lr={}, bs={},  ={})'.format(layer_D[0],
                                                                                    layer_D[1:-1],
                                                                                    layer_D[-1],
                                                                                    args.activation_unit,
                                                                                    args.learning_rate,
                                                                                    args.batch_size,
                                                                                    args.momentum)

# Set loss function to be mean squared error with summation over each training batch.
loss_fn = torch.nn.MSELoss(reduction='sum')

# Build set of layers in order.
layers = OrderedDict()
activations = OrderedDict()
layers['P'] = torch.nn.LayerNorm(layer_D[0], elementwise_affine=False)
for i in range(len(layer_D)-1):
    linear_layer_name = 'L{}'.format(i)
    layers[linear_layer_name] = torch.nn.Linear(layer_D[i], layer_D[i+1]).to(torch.double)
    if i < len(layer_D) - 2:
        activation_layer_name = 'A{}'.format(i)
        layers[activation_layer_name] = activation_unit().to(torch.double)
        if args.train_arff is not None:
            activations[activation_layer_name] = torch.zeros(
                layer_D[i+1], args.activation_bins).to(torch.double).requires_grad_(False)
            layers[activation_layer_name].register_forward_hook(functools.partial(capture_hidden_outputs_hook,
                                                                                  name=activation_layer_name))

model = torch.nn.Sequential(layers)
one_hots = torch.nn.functional.one_hot(torch.arange(0,model[-1].out_features)).type(torch.double)

# Load training input (in ARFF format).
if args.train_arff is not None:
    X_train, Y_train, L_train = load_arff(args.train_arff)

X_test = None
Y_test = None
L_test = None

# Load testing input (in ARFF format).
if args.test_arff is not None:
    X_test, Y_test, L_test = load_arff(args.test_arff)

# Create NN model.
print(model, file=sys.stderr)

# Train or test, depending on user spec.
# @TODO: Save trained model for future loading. Otherwise, have to train before testing.
if args.train_arff is not None:
    train_nn(model, X_train, Y_train, X_test, Y_test)
    with open(os.path.join(args.workspace_dir, args.model_file), 'wb') as fout:
        pickle.dump(model.state_dict(), fout)
elif args.test_arff is not None:
    with open(os.path.join(args.workspace_dir, args.model_file), 'rb') as fin:
        model.load_state_dict(pickle.load(fin))
    test_nn(model, X_test, Y_test)

if args.profile:
    pr.disable()
    s = io.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats('tottime')
    ps.print_stats()
    print(s.getvalue())
```