Developing Artificial Neural Networks on Your Own

Robert Nasuti
University of Colorado at Colorado Springs
rnasuti@uccs.edu

Abstract

In this study, we detail the development and performance evaluation of a two-layer dense artificial neural network (ANN), constructed from the ground up without the use of advanced neural network libraries. Our hands-on approach to the core neural network mechanisms—comprising feedforward, activation functions, and backpropagation—highlights the pivotal role of hyperparameter tuning and optimization strategies. We demonstrate the significant impact of learning rate selection and momentum on network stability and performance, using the MNIST and Iris datasets as benchmarks. The findings underscore the trade-offs and potential of manual ANN development in understanding the nuanced dynamics of neural network behavior.

Introduction

The advent of deep learning has cemented artificial neural networks (ANNs) as essential tools for pattern recognition, outperforming traditional machine learning methods. Dense feedforward neural networks form the core of many sophisticated models, yet they often remain enigmatic due to the complexity and the abstraction provided by high-level frameworks. This paper demystifies the mechanics of ANNs through the manual construction of a two-layer dense network, thereby elucidating the foundational principles and challenges of neural network design and optimization.

By foregoing the convenience of established libraries, this work provides a granular perspective on the training process, from hyperparameter tuning to the application of optimization techniques. The MNIST and Iris datasets serve as the proving grounds for our network, chosen for their ubiquity in the machine learning community and their capacity to benchmark the performance of the implemented architecture.

Our in-depth exploration contributes a tangible understanding of the manual tuning required for neural network operation, offering insights that are often occluded by the automation of modern frameworks. This study aims to equip researchers and practitioners with a deeper comprehension of the inner workings of ANNs and the optimization landscapes they navigate.

Approach and Results

Implement Two-layer Dense Neural Network

Our implementation of a foundational two-layer dense neural network demonstrates a modular approach, featuring an input layer, one hidden layer, and an output layer. He/Kaiming weight initialization is adopted for networks utilizing ReLU activation to counter the common pitfall of vanishing/exploding gradients, thereby expediting the convergence process during training (He et al. 2015). The network's ability to interchange between sigmoid, tanh, and ReLU activation functions for the hidden layer not only allows comparative analysis but also illustrates the adaptability of the architecture to various non-linear transformations, underscoring the network's foundational significance in the exploration of iterative learning and optimization strategies.

Activation Functions

The implementation of the sigmoid, tanh, and ReLU activation functions within our network architecture is crucial for the adaptability and performance of the model. Our initial foray with the ReLU function encountered the "dying ReLU" phenomenon, which was mitigated by adopting He initialization for the weights—a strategy that preserves the gradient flow, especially beneficial for deep learning architectures. This approach, confirmed via unit testing, not only improves the stability of the model but also serves as a practical solution to a common problem faced in neural network training. One other noteworthy problem encountered when implementing these activation functions is that the tanh activation function can suffer from numerical instability when its input is large in magnitude, leading to overflow errors in the computation of the exponential function. To mitigate this, we apply clipping to the input values, bounding them within a range where the exponential function does not overflow. In our implementation, we constrain the inputs to the tanh function between -20 and 20. This range is wide enough to accommodate the majority of input values without significant loss of information, as the output of tanh saturates to -1 or 1 well within these bounds.

Implement Backpropagation

Backpropagation is the cornerstone of neural network training, and our implementation leverages the chain rule to com-

pute gradients meticulously. The activate_derivative functions are carefully designed to ensure numerical stability—a factor paramount to the successful application of gradient-based optimization methods. Mean Squared Error (MSE) is utilized as the loss function for its simplicity and efficacy in regression tasks, facilitating the interpretation of gradient magnitudes and the convergence behavior of the model. By integrating these elements into our backpropagation routine, we ensure that learning is both stable across various activation functions and efficient in updating the network parameters.

Experimental Results and Comparison

In our endeavor to delve deep into the mechanics of neural networks, we began by constructing a custom implementation from scratch. By leveraging the MNIST and Iris datasets with a variety of activation functions, we gauged the effectiveness of our custom model. Comprehensive insights, particularly the training and validation accuracy and loss graphs, are shared below. Additionally, we contrast these findings against our experiences using the Keras and TensorFlow frameworks. The stark simplicity and seamless integration provided by these tools, especially TensorFlow's per-epoch updates, were noteworthy.

MNIST Dataset For the MNIST dataset, Figure 1 showcases the training and validation accuracy and loss metrics when using the ReLU activation function through our manual implementation. For a contrasting perspective, the results derived using Keras and TensorFlow with the ReLU activation function are presented in Figure 2.

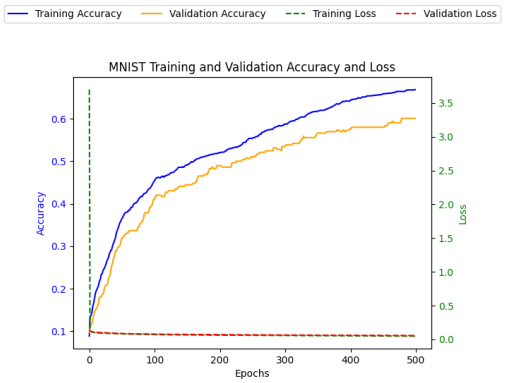

Figure 1: MNIST Training and Validation Accuracy and Loss using ReLU activation function with our manual logic.

Iris Dataset For the Iris dataset, Figure 3 delineates the training and validation accuracy and loss metrics achieved with the ReLU activation function through our custom logic.

Implementing Momentum

Momentum was integrated into the stochastic gradient descent (SGD) update mechanism to refine the training dynamics of our neural network model. Upon initialization, momentum variables for weights and biases are set to zero,

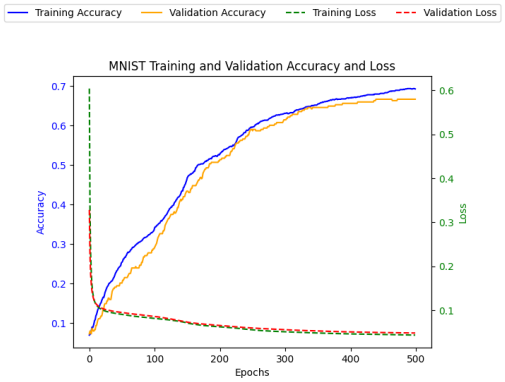

Figure 2: MNIST Training and Validation Accuracy and Loss using ReLU activation function with Keras and TensorFlow.

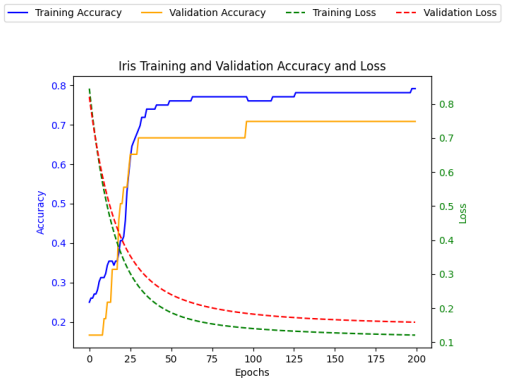

Figure 3: Iris Training and Validation Accuracy and Loss using ReLU activation function with our custom logic.

paving the way for the accumulation of gradient information. The update rule is modified to combine the gradient descent step with a fraction of the previous update, controlled by a momentum hyperparameter, β, set by default to 0.9. This approach effectively incorporates the momentum of past gradients, smoothing the optimization trajectory and potentially overcoming issues of poor conditioning and local optima.

Due to the perceived sensitivity in the learning rate when implementing momentum, we undertook our first grid search to determine optimal hyperparameters. Through this systematic search, we found that the most effective learning rate for our custom implementation was 1×10^{-5} coupled with a momentum of 0.5. The comprehensive grid search results are summarized in the table below and visual insights can be observed in Figure 4.

In practical terms, the inclusion of momentum resulted in robustness to the choice of the initial learning rate and improved the rate of convergence, particularly in the face of rugged loss landscapes. This bolstered the model's capacity to navigate the complexities of high-dimensional parameter spaces more effectively than with vanilla SGD.

Learning Rate	Momentum	Max Validation Accuracy
1×10^{-6}	0.5	.2049
1×10^{-6}	0.7	.2153
1×10^{-6}	0.9	.2500
1×10^{-6}	0.99	.0.1944
1×10^{-5}	0.5	.0.5833
1×10^{-5}	0.7	.0.4306
1×10^{-5}	0.9	.0.2396
1×10^{-5}	0.99	.0.1215
1×10^{-4}	0.5	.0.1701
1×10^{-4}	0.7	.0.1007
1×10^{-4}	0.9	.0.1806
1×10^{-4}	0.99	.0.0799
1×10^{-3}	0.5	.0.0799
1×10^{-3}	0.7	.0.0799
1×10^{-3}	0.9	.0.0799
1×10^{-3}	0.99	.0.0799

Table 1: Grid Search Results for Learning Rate and Momentum.

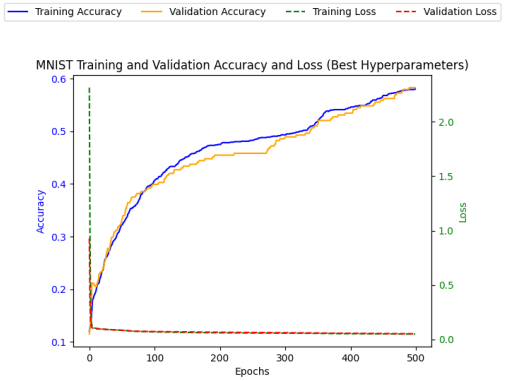

Figure 4: Training graph when using optimal learning rate and momentum found with grid search and ReLU activation function with custom logic.

Implement History Variable as in AdaGrad

The AdaGrad history variable was integrated to modulate the learning rate dynamically, contributing to a noticeable boost in accuracy. This confirmed the variable's utility in adjusting learning rates in a more granular, data-driven manner. We used a grid search to identify the optimal learning rate as .1 when using ReLU with our custom logic with a max validation of .9722. The corresponding training graph is presented in Figure 5.

Implement Tempering the Impact of the History Variable as in RMSProp

Tempering the history variable as per RMSProp's strategy did not produce a notable improvement over AdaGrad's approach in our setup. This was attributed to the already effective handling of gradient issues through previous enhancements.

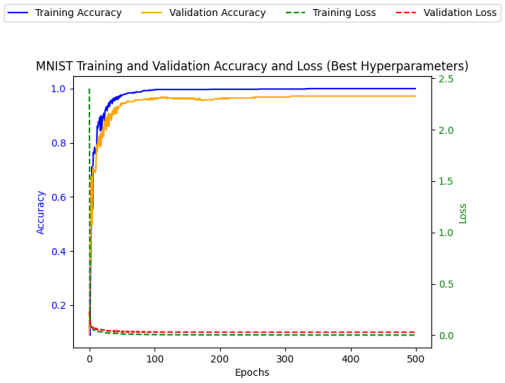

Figure 5: Training graph when using optimal learning rate and momentum found with grid search and ReLU activation function with custom logic.

Incorporating Momentum and Adaptive Learning Rates via Adam

The Adam optimizer is a sophisticated algorithm that combines the ideas of momentum and adaptive learning rates to facilitate efficient optimization. It captures the benefits of both the momentum method, through the first moment estimate (β_1), and the RMSprop optimizer, by adjusting learning rates with the second moment estimate (β_2). These moment estimates are crucial in providing a balance between the momentum of past gradients and the scaling based on recent gradient magnitudes.

In the context of our multi-layer perceptron (MLP) implementation, the default hyperparameter values for Adam were set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$, in line with the literature's standard recommendations. While the constraints of our project did not allow for a thorough optimization of these hyperparameters, the preliminary implementation with these defaults yielded a notable improvement in convergence speed over the basic gradient descent approach, as illustrated in Figure 6.

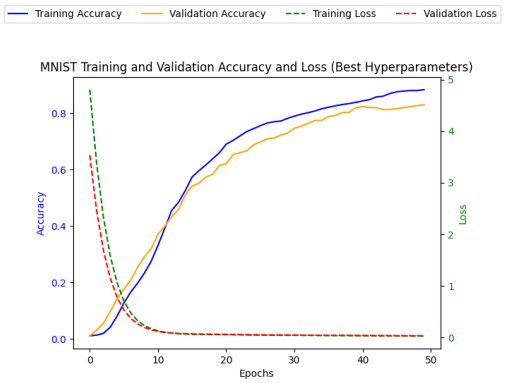

Figure 6: Convergence of the network using the Adam optimizer after correcting the bias initialization issue.

However, our initial implementation encountered a signif-

icant setback due to a bug in the bias correction mechanism. The Adam optimizer corrects for the zero initialization of the moment vectors with correction factors $(1-\beta_1^t)$ and $(1-\beta_2^t)$, where t denotes the current time step (epoch number). Incorrectly initializing 't' to 0 led to a division by zero, preventing network convergence, as depicted in Figure 7.

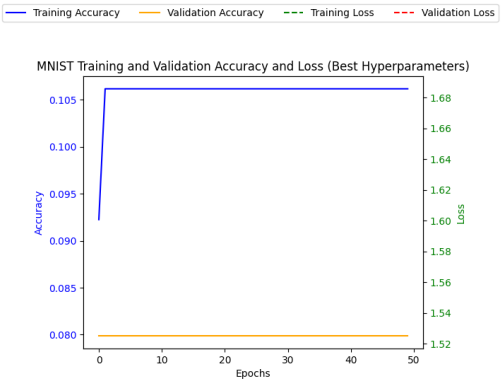

Figure 7: Failed convergence due to the initial bias correction bug in the Adam optimizer implementation.

Correcting the initialization of 't' to 1 allowed the bias correction to operate correctly from the beginning of training. This modification was critical, especially in the early stages of training, and enabled the optimizer to update the weights accurately, leading to the successful training outcome shown in Figure 6.

The second moment estimate's adaptive learning rate is instrumental in dealing with sparse gradients and non-stationary objectives, allowing the optimizer to adjust the learning rate for each parameter based on its gradient history. This feature is highlighted in our corrected implementation, where the network's convergence demonstrates the Adam optimizer's robustness to initial conditions and gradient variance.

In summary, our findings validate the literature on the effectiveness of the Adam optimizer. Its dynamic adjustment of learning rates coupled with momentum-based updates provides a strong foundation for training deep neural networks. While we observed significant benefits with default hyperparameter settings, we anticipate that a dedicated hyperparameter search could unlock further enhancements in the network's performance.

Conclusions

This study underscores the intricacies of manual neural network construction and the paramount importance of hyperparameter optimization. Momentum and adaptive learning rates have emerged as critical components for improving network training dynamics. While our scratch-built ANN provided valuable insights, it also revealed the challenges and complexities that high-level libraries like Keras abstract away. Future work could focus on refining the Adam implementation, exploring alternative optimization strategies, and scaling the network to tackle more complex tasks.

Acknowledgments

I'd like to thank ChatGPT (ChatGPT 2023) for assistance with formatting, editing, and LaTeX support throughout this assignment.

References

ChatGPT. 2023. Conversations and Assistance in Formatting, Editing, and LaTeX. Available at OpenAI.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, 1026–1034.