Custom Artifical Neural Networks for Hands-on Proficiency

Samuel Olowofila
Department of Computer Science
University of Colorado Colorado Springs
CO, 80918 USA

Abstract

This paper explores the construction of a two-layer feedforward neural network from scratch, focusing on manual programming and avoiding specialized ANN libraries. The study uses mathematical tools for data preprocessing and examines the network trained with Stochastic Gradient Descent (SGD), including enhancements like Momentum, RMSProp, AdaGrad, and Adam. Employing the MNIST dataset, it conducts a comparative analysis with a network built using Keras and Tensorflow, aiming to highlight the trade-offs and performance benchmarks of manual neural network construction versus using advanced libraries, as well as compare the performances of prominent activation functions.

Introduction

The realm of Artificial Neural Networks (ANNs) has burgeoned into a pivotal component of modern computational intelligence, offering solutions to complex problems across various domains (Wu and Feng 2018). This paper embarks on an exploratory journey to construct and understand the intricacies of ANNs by delving into the foundational aspects of their creation. The core objective of this study is to design and develop a basic yet robust, two-layer feedforward neural network entirely from scratch. This endeavour necessitates programming each facet of the neural network meticulously, without the aid of any specialized ANN libraries for the network's architecture or core operations.

To achieve this, the study leverages mathematical tools and general-purpose libraries for data pre-processing while scrupulously abstaining from using any libraries that automate the primary operations of the ANN. The focal framework model of this experiment is a two-layer feedforward network equipped with backpropagation and trained using Stochastic Gradient Descent (SGD) (Ruder 2016). Further, this paper investigates enhancements to the SGD method to improve the training process and overall network performance.

The experiment employs a widely recognized dataset-MNIST, providing a common ground for comparison and evaluation. This approach not only fosters a deep understanding of the nuances of neural network construction but also enables a hands-on experience in fine-tuning network parameters and configurations. In the latter part of the study, the manually constructed network is juxtaposed with results obtained using Keras with a Tensorflow backend, to evaluate the efficacy of the custom-built network. This comparative analysis aims to illustrate the performance benchmarks of handcrafted networks against those built using advanced, library-supported environments, thereby offering insights into the trade-offs and benefits of building neural networks from scratch.

Approach

This experiment pivots on a two-layer Dense Neural Network exhibiting random weight initialization, a simple forward propagation or pass, the cross-entropy loss function that is computed from any given labels and the corresponding prediction from the forward pass. The network also features a back propagation phase where the gradients required for updating the network's weights and biases are calculated by computing the error at the output layer and propagating it backwards through the network, applying the derivative of the activation function of choice. All variants of this experiment are performed using only ten epochs to save time and computational resources. Also implemented in the forward pass of this network are activation functions, the importance of which cannot be overemphasized. Firstly, they are introduced to perform non-linearity in the network, ensuring that the model can handle complex patterns that are not distinguishable by a linear boundary. They also decide whether a neuron should be activated or not by calculating its weighted sum and adding its corresponding bias whilst safeguarding the network from vanishing and exploding gradients.

Baseline DNN

As earlier submitted, the primary Dense Neural Network on which this experiment pivots, features two layers poised to accept n inputs and produce m outputs, both of which are adjustable. Here, we perform a number of experiments, primarily juxtaposing the efficiency of three major activation functions: Rectified Linear Unit $(ReLU)$, $sigmoid$, and $tanh$ in an image classification task. The baseline network is preliminarily initialized with random weights and biases for two layers: a hidden layer and an output later. The forward propagation module accepts the user-defined input and

applies a linear transformation followed by the select activation function per experiment and a linear transformation followed by the softmax activation for the output layer, whose presence at the output layer converts the network outputs into probabilities for classification. The output of this function is taken as the probability for each class. The baseline DNN is then experimented on the MNIST dataset to perform a classification task. This process is likewise performed using the Keras library with Tensofrflow backend for comparative analysis as reported in Figure 2.

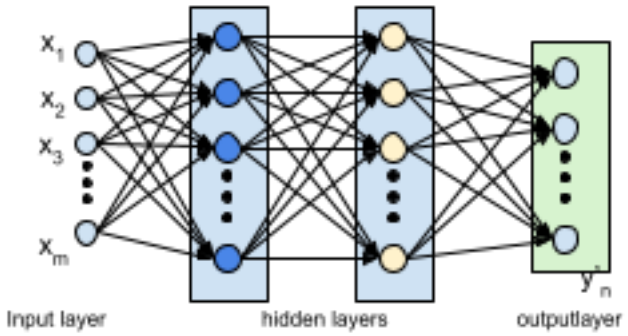

Figure 1: The Baseline DNN Architecture

Momentum

The second variant of this experiment features Momentum, a technique used to accelerate the convergence of gradient descent algorithms in optimization, especially in the context of Stochastic Gradient Descent (Sutskever et al. 2013). It is inspired by the physical concept of momentum in motion, which implies that an object in motion tends to stay in motion. In momentum, a 'velocity' variable is maintained for each parameter (weight/bias) of the model. This velocity is a running average of the gradients, and it accumulates gradients of past steps to determine the direction to move in the parameter space. The momentum coefficient μ determines the contribution of the previous update to the current update.

$$V_{t+1} = \mu V_t + \nabla_\theta J(\Theta) \tag{1}$$

$$\theta = \theta - \eta V_{t+1} \tag{2}$$

2

While V_t represents the velocity at time step t, the gradient of the objective function $J(\theta)$ for each parameter θ, with respect to θ, is denoted by $\nabla_\theta J(\theta)$ in the Velocity Update Equation 1. The parameters are then updated using the velocity vector in Equation 2, where η is the learning rate.

History in AdaGrad

In gradient descent algorithms, especially in their adaptive variants like AdaGrad, history variables are used to adapt the learning rate for each parameter based on the history of its gradients (Duchi, Hazan, and Singer 2011). The primary goal is to achieve more efficient and stable convergence during training, particularly for problems with sparse data or irregular optimization landscapes. AdaGrad (Adaptive Gradient Algorithm) utilizes a history variable to accumulate the squared gradients for each parameter. This accumulated value is then used to adjust the learning rate for each parameter individually.

$$h_i = h_i + (\nabla_{\theta_i} J(\theta))^2 \tag{3}$$

Each parameter θ is assigned a history variable h_i, which is initialized to zero. This variable is then set to accumulate the squares of gradients as depicted in Equation 3.

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{h_i + \varepsilon}} \nabla_{\theta_i} J(\theta) \tag{4}$$

The learning rate η, and ε (usually a small constant like 10^{-8}) are embedded to improve numerical stability when updating the parameters as depicted in Equation 4.

Tempering with RMS-Prop

The history phenomenon, as implemented in AdaGrad, can lead to a significant decline in the efficacy of the learning rate over time, especially in long-running tasks (Hinton, Srivastava, and Swersky 2012). This is because the embedded denominator term $\sqrt{h_i + \varepsilon}$ grows continuously as more gradient squares are summed up, resulting in a progressively lower effective learning rate for each parameter, which in turn slows down learning and increases the difficulty of reaching a global minimum, especially when working with large inputs. RMSProp (Root Mean Square Propagation) addresses that challenge by implementing a modification to the AdaGrad approach by introducing a decay factor β to control the accumulation of the squared gradients. RMSProp uses an exponentially decaying average instead of summing up all past squared gradients. This process tempers the reduction in the learning rate over time.

$$h_i = \beta h_i + (1 - \beta)(\nabla_{\theta_i} J(\theta))^2 \tag{5}$$

The update rule in RMSProp, as represented in Equation 5, expresses the decay factor β, which ranges from 0 to 1, but it is typically set to 0.9. It determines how much of the history to retain versus how much of the recent gradient to consider. The decay factor is there to ensure that history h_i does not grow too large, thereby preventing the effective learning rate from shrinking too rapidly.

Adam Optimization

Adam (Adaptive Moment Estimation) is an optimization technique that fuses the RMSProp and Momentum concepts (Kingma and Ba 2014). It maintains both the first moment (mean) and the second moment (uncentered variance) of the gradients, effectively combining the benefits of adaptive learning rates and momentum (Zhang 2018).

	ReLU		SIGMOID		TANH	
	Scratch	Keras	Scratch	Keras	Scratch	Keras
Loss	0.281	0.2395	0.5777	0.4131	0.2853	0.26
Accuracy	0.9197	0.9329	0.8528	0.8918	0.9183	0.9264

Figure 2: Classification results for the Baseline DNN built from scratch vs built with keras

Algorithm 1 Adam Optimization Algorithm

Input: learning rate α, decay rates β_1, β_2, small constant ϵ, objective function $f(\theta)$, initial parameters θ_0, maximum iterations *max_iterations*
Initialize timestep $t \leftarrow 0$
Initialize first moment vector $m_0 \leftarrow \mathbf{0}$
Initialize second moment vector $v_0 \leftarrow \mathbf{0}$
Initialize parameters $\theta \leftarrow \theta_0$
while $t <$ *max_iterations* **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow$ Compute gradients of $f(\theta)$ at θ
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (Correct bias for first moment)
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (Correct bias for second moment)
    $\theta \leftarrow \theta - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ (Update parameters)
end while
return θ

	Experiments	RELU	SIGMOID	TANH
LOSS	Vanila/Basecase	**0.2801**	0.5777	0.2853
	Momentum	**0.0746**	0.3166	0.1158
	History (AdaGrad)	**0.1617**	0.6767	0.1814
	Tempering (RMSProp)	**0.1207**	0.6009	0.2195
	Adam Opt	**0.0759**	0.725	0.1517
ACCURACY	*Experiments*	RELU	SIGMOID	TANH
	Vanila/Basecase	**0.9197**	0.8528	0.9183
	Momentum	**0.9783**	0.9079	0.9666
	History (AdaGrad)	**0.955**	0.8008	0.9486
	Tempering (RMSProp)	**0.9763**	0.8116	0.9364
	Adam Opt	**0.9795**	0.7707	0.9525

Figure 3: A comparative analysis of results from all instances of the experiment.

Results

In this research, every approach to the optimization of a dense neural network discussed earlier was experimented on the Baseline DNN build.

The uniform classification experiment was performed using the MNIST dataset on variants of the model distinguished by the aforementioned methods. The results show incremental improvement in convergence, as well as improvements in the loss obtained. As depicted in Figure 3, it holds true that the Rectified Linear Unit activation function emerges as the best-performing activation function while the hyperbolic tangent function was the least performing for this classification task. Also, it is observed that the order of introduction of the method dynamics does not result in incremental performance. The execution of the history concept as a stand-alone feat declined the performance of the models. However, tempering reversed the performance decline, and the fusion of both momentum and history resulted in the best-performing instances for the models using ReLU and tanh.

Conclusion

The study's findings show that optimization techniques, particularly Adam combining momentum and history, enhance the performance of a Dense Neural Network (DNN) on the MNIST dataset. ReLU proves to be the most effective activation function, while tanh is the least. Implementing history or momentum alone reduces performance, while the hybrid approach to actuating momentum and history, as in Adam, improves it. These results highlight the significance of choosing the right optimization strategies and activation functions in neural network design. Comparing these results with library-based implementations like Keras and Tensorflow also underscores the educational benefits of manually building neural networks from scratch.

References

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12(7).

Hinton, G.; Srivastava, N.; and Swersky, K. 2012. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on* 14(8):2.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Ruder, S. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G. 2013. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, 1139–1147. PMLR.

Wu, Y.-c., and Feng, J.-w. 2018. Development and application of artificial neural network. *Wireless Personal Communications* 102:1645–1656.

Zhang, Z. 2018. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, 1–2. Ieee.

Supplementary Material

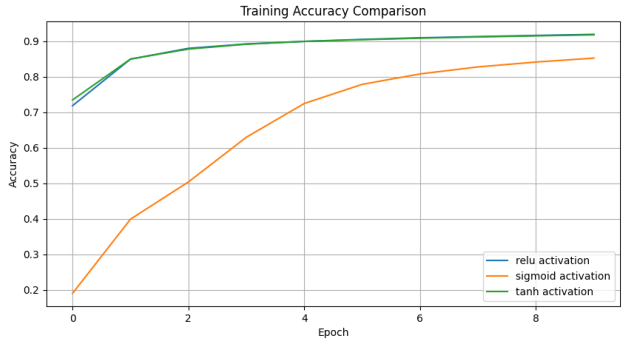

Figure 4: Baseline DNN classification MNIST from stratch

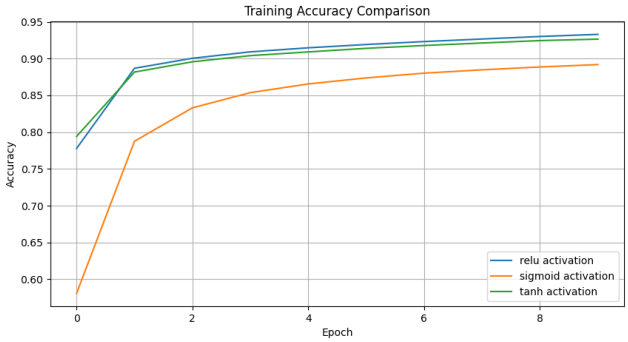

Figure 5: Baseline DNN classification of MNIST with Keras and Tensorflow

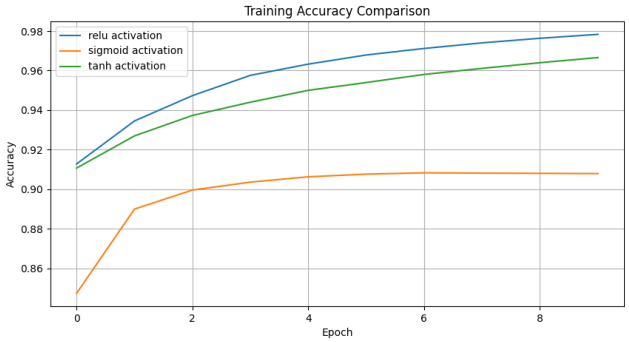

Figure 6: DNN with Momentum

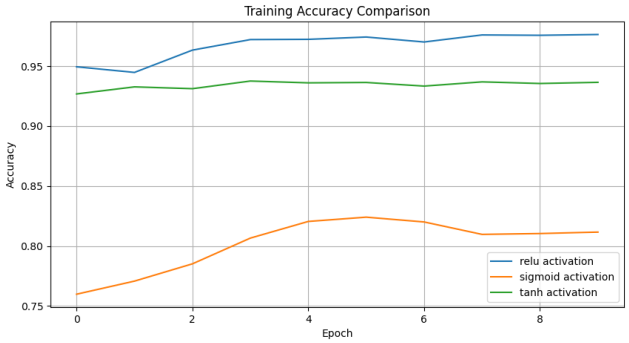

Figure 7: DNN with tempering - RMSProp

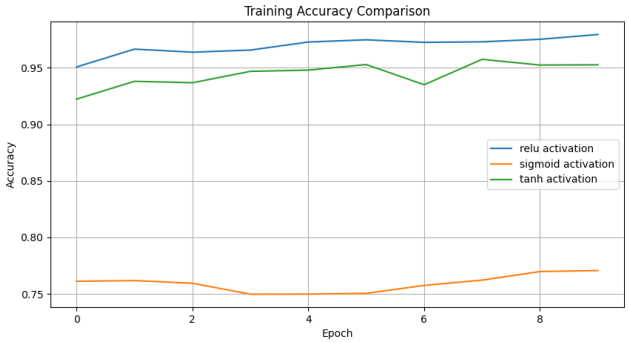

Figure 8: DNN with Momentum and History - Adam