ECE60146: Homework 2

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

Gradient descent is an optimization algorithm commonly used in deep learning to minimize the loss function associated with a model. The goal of training a model is to find the set of parameters (weights and biases) that minimizes the error between the predicted outputs and the actual targets. Gradient descent helps in this process by iteratively updating the parameters based on the negative gradient (slope) of the loss function with respect to those parameters. While gradient descent is the basic optimization algorithm, there are variations to improve its efficiency. In this homework, we will investigate two such variations, namely SGD with Momentum (SGD+) and Adam (Adaptive Moment Estimation).

2 Becoming Familiar with the Primer

Vanilla Stochastic Gradient Descent uses the formula

$$p_{t+1} = p_t - \alpha * \Delta_p L(p_t) \tag{1}$$

Here, at each iteration t, p_t represents the value of the parameter, L denotes the loss function and $\alpha = lr$ is the learning rate. Alternatively, $\Delta_p L(p_t) = g_{t+1}$ can be termed the gradient for the time step $t + 1$. The lr is a hyperparameter that determines how fast the minimum of the loss function is reached by the optimizer. The lr has to be selected carefully, as too big a learning rate can lead to arbitrary oscillations over the loss surface and too small a learning rate can lead to slower attainment of the minima (or in cases non-attainment of the minima!). One way to deal with this problem is to use different lr for different stages of the parameter update, a term called the learning rate scheduling. Predefining the lr may not work for many datasets as the parameters are updated at different pace as per the nature of the dataset. Further, the number of parameters is indicative of the dimension of the loss function. Hence, there could be a requirement of different lr for different dimensions, which can't be satisfied using the vanilla SGD.

Another major issue is the risk of getting stuck at a saddle point (Figure 1. A saddle point refers to a point in the parameter space where the gradient of the loss function is zero but the point is not a minimum or maximum. At a saddle point, the curvature of the loss function is neither strictly convex nor strictly concave. This means that the first derivative is zero, but the point is not an optimal solution. In the vicinity of a saddle point, the vanilla SGD algorithm may make slow progress as it struggles to escape from the flat regions associated with these points. The estimation of p_{t+1} can be improved, in these cases, using $SGD+$ and Adam optimizer.

2.1 SGD with Momentum (SGD+):

In $SGD+$ the learnable parameters are updated using momentum. It involves incorporating historical information to improve the optimization process. Momentum is introduced by retaining the step size from the previous iteration, creating a dynamic interplay between the current gradient and the momentum from the prior step. Specifically, the decision for the current step size is influenced

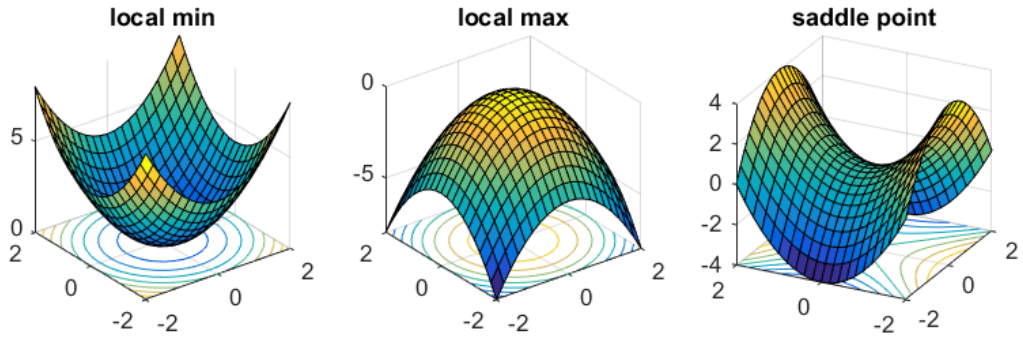

Figure 1: Saddle Point

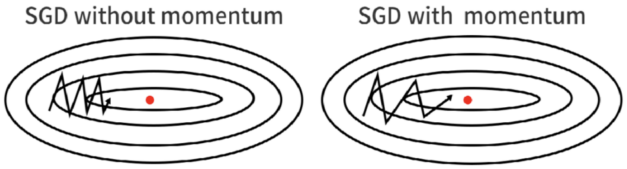

Figure 2: SGD vs SGD+

by both the current gradient value and the preceding step size. The recursive update formula for the step size (v) and the parameter (p_t)is expressed as follows

$$\begin{aligned} v_{t+1} &= \mu v_t + \Delta_p L(p_t) \\ p_{t+1} &= p_t - \alpha * v_{t+1} \end{aligned} \tag{2}$$

In this context, the variable v_t (loosely considered as *velocity*)holds the information pertaining to the step size utilized in the preceding iteration. The momentum coefficient, denoted by μ is constrained within the range $[0, 1]$ dictates the influence of the previous gradient step on the current update. In other words, μ determines the extent of emphasis placed on the historical gradient information. $\mu = 0$ will lead to the vanilla SGD. Similarly, if $\mu = 1$, it results in another undesirable situation where the current update is solely based on the accumulated historical gradients, and there is no decay or forgetting factor applied. As a thumb rule, the exponential average of the past $\frac{1}{1-\mu}$ $v_t's$ are used to calculate v_{t+1}. For example, if $\mu = 0.9$, then the exponential average of the past $\frac{1}{1-0.9}$, i.e. 10 v_ts are used for calculating the current *velocity* [2] Initially, v_0 is initialized to 0, marking the starting point of the momentum accumulation. In this way $SGD+$ aims to mitigate oscillations and improve convergence by considering the momentum accumulated from past steps. In practice, separate step updates are computed for individual learnable parameters, enabling the determination of the current step size by jointly evaluating its prior value and the current gradient value. This approach enhances the adaptability of the optimization process, facilitating more effective convergence in the training of deep neural networks.

The momentum method performs better for non-convex optimisation problems with loss curves exhibiting high curvature, consistent gradient (i.e. a gradually descending slope) and noisy gradient (i.e. a fluctuating gradient). The comparison between the typical loss trajectories on the loss contours during training in vanilla SGD and SGD with momentum is illustrated in Figure 2. Introduction of momentum term serves to diminish oscillations in loss that occur perpendicular to the minima, simultaneously promoting increased progress toward the minima.

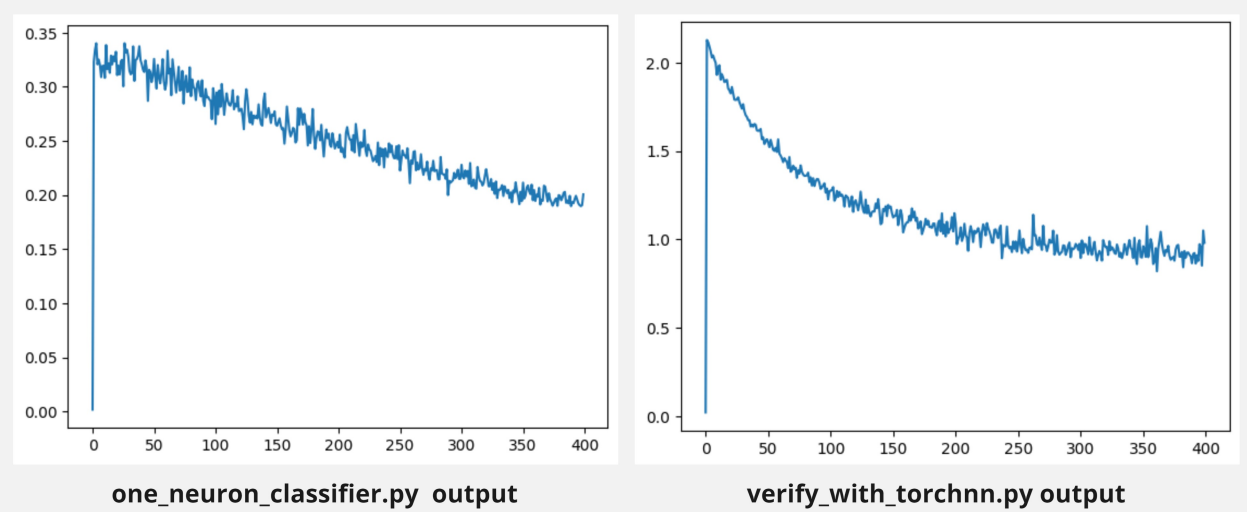

Figure 3: one_neuron_classifier.py vs verify_with_torchnn.py

2.2 Execution of files in "Examples" folder:

The code from the files **one _neuron _classifier.py**, **multi _neuron _classifier.py**, and **verify _with _torchnn.py** was integrated into the Colab notebook and executed. This integration was necessitated by the need to address a bug in the original implementation code of the ComputationalGraphPrimer class, specifically in the backpropagation step for the one-neuron model.

The bug arose from the placement of the statement that averaged the partial loss over the batch samples within the loop that aggregates contributions from individual instances in a batch. To rectify this issue, the averaging statement needed to be relocated outside the loop. As a result of this bug fix, the overridden code was incorporated into the Colab notebook for seamless execution. Although the bug was fixed in Version 1.1.4, I relied upon 1.1.3 as indicated in HW instructions.

Furthermore, modifications to the **verify _with _torchnn.py** code were required for both the one-neuron and multi-neuron cases. Instead of repeatedly adjusting the code within the **verify _with _torchnn.py**, a decision was made to include the relevant code directly in the Colab notebook. This approach was deemed more efficient, allowing for easier adaptation to specific scenarios without the need for frequent manual adjustments in the external script.

The output of (code segement corresponding to) **one _neuron _classifier.py** and **verify _with _torchnn.py** with similar parameters is shown in Figure 3. Here training_iterations=40000, batch_size=8 and learning_rate=10^{-3}

The output of (code segement corresponding to) **multi _neuron _classifier.py** and **verify _with _torchnn.py** with similar parameters is shown in Figure 4. Here training_iterations=40000, batch_size=8 and learning_rate=10^{-6}.

It is crucial to highlight that the **verify _with _torchnn.py** script employs the **torch.nn** version of the handcrafted network, presenting output loss per batch. In contrast, the custom scripts, namely **one _neuron _classifier.py** and **multi _neuron _classifier.py**, exhibit output loss per input training data. This distinction in reporting metrics is essential for a comprehensive understanding of the evaluation results.

Upon comparing the final loss for the one-neuron case, it is observed that the **verify _with _torchnn.py** code yields a loss of $0.9809/8 = 0.1226125$, while the **one _neuron _classifier.py** reports a loss of 0.2005. Similarly, in the multi-neuron scenario, the **verify _with _torchnn.py** code results in a final loss of $8.3505/8 = 0.1043$, whereas the **multi _neuron _classifier.py** indicates a

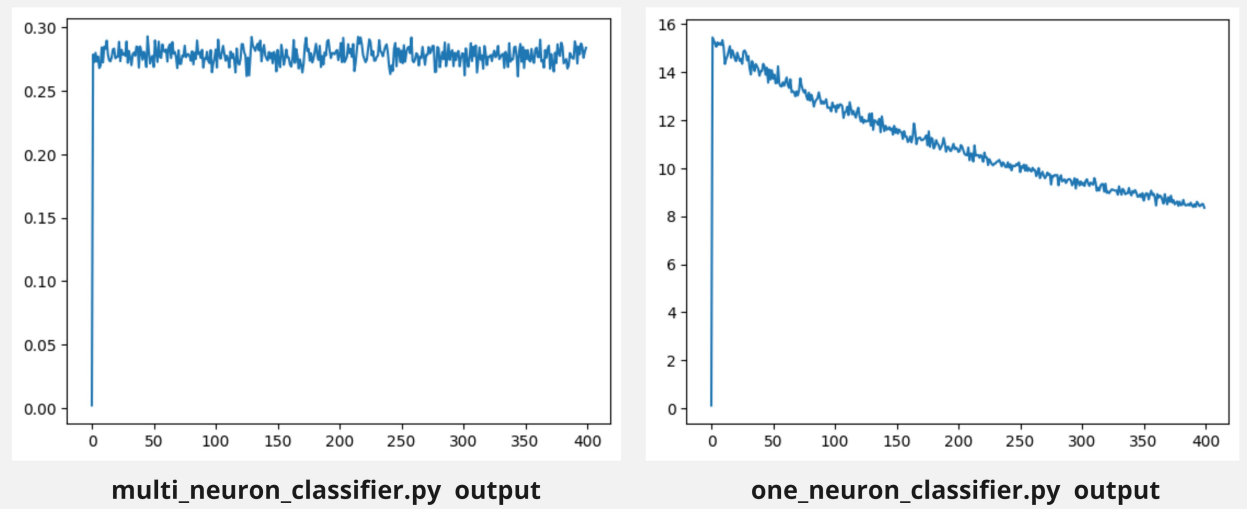

Figure 4: multi_neuron_classifier.py vs verify_with_torchnn.py

loss of 0.2837.

It is noteworthy that the torch.nn-based results are superior. This can be attributed to the fact that the internal torch.nn module leverages optimization techniques during the backpropagation process, leading to more efficient and effective code execution. The observed differences in loss values underscore the advantages of using optimized frameworks like torch.nn for neural network training.

2.3 Adaptive Moment Estimation (Adam):

While momentum gradient descent is a powerful optimization algorithm that helps accelerate convergence and navigate through flat regions of the loss landscape, it is not without its challenges. For example, high momentum values can lead to overshooting, causing the optimizer to move too quickly and potentially oscillate around the minimum. This may hinder convergence and result in a less stable training process. Also, the performance of momentum gradient descent is sensitive to the choice of hyperparameters, especially the momentum coefficient μ. Adam addresses these issues by combining ideas from two popular optimization algorithms namely, the already discussed $SGD+$ and another RMSprop. Adam is designed to provide adaptive learning rates for each parameter by maintaining two moving averages for each parameter: the first moment (mean) and the second moment (uncentered variance). The equations below demonstrate the key logic behind Adam [1]

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_{t+1} \\ v_{t+1} &= \beta_2 v_t + (1-\beta_2)(g_{t+1})^2 \\ p_{t+1} &= p_t - \alpha * \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}} \end{aligned} \tag{3}$$

Following is a brief overview of how the Adam optimizer works:

1. Initialize the first moment (m_0) and second moment (v_0) for each parameter to zero.
2. Compute the gradient of the loss with respect to the parameters ($\Delta_p L(p_t)$)
3. Update the first moment (m_{t+1}) using the formula in Equation 3. Here β_1 is the decay factor (typically close to 1, e.g., 0.9). This update is similar to the $SGD+$ momentum update

4. Update the second moment (v_{t+1}) using the formula in Equation 3. Here β_2 is another decay factor (again typically close to 1, e.g., 0.999). This update is similar to the moving average of squared gradients in RMSprop.

5. After updating the first and second moments a bias correction step is employed to counteract the initialization bias that arises from setting (m_t) and (v_t) to zero at the beginning.The bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$ are calculated as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{4}$$

   These corrected estimates adjust for the underestimation of m_t and v_t in the early iterations, providing more accurate representations of the first and second moments over time. The bias correction contributes to the stability and reliability of the Adam optimizer during the training.

6. Finally, the parameters (p_t) are updated using the calculated moments as shown in 3. The ϵ is a small constant (usually 10^{-8}) to avoid division by zero.

Adam optimizer has been shown to perform well in practice and is widely used for training deep neural networks due to its adaptive learning rates and effective handling of sparse gradients.

3 Implementation and results

3.1 Programming Tasks

As indicated in the HW, the implementation is primarily based on the ComputationalGraphPrimer module [3]. The main structure of the ComputationalGraphPrimer.py class and the extension tasks carried out on the code is pictorially depicted in Figure 5.

For the implementation of both **Stochastic Gradient Descent with momentum (SGD+)** and **Adam** optimization algorithms, I have introduced two subclasses: **ComputationalGraphPrimerSGDplus** and **ComputationalGraphPrimerAdam**. These subclasses inherit from the base class **ComputationalGraphPrimer** provided by the module.

To tailor the behavior of these subclasses to the specific optimization algorithms, we have overridden the **backprop_and _update _params_one_neuron_model()** and **backprop_and_update _params_multi _neuron_model()** functions. This customization ensures that the computational graph primer aligns seamlessly with the intricacies of both **SGD+** and **Adam** optimization methods, providing a foundation for network training described in Section 2. The logic for implementation the code has been directly adopted from the algorithm, with previous year solutions as reference.

Three distinct learning rates, namely 0.01, 0.001, and 0.0001, were systematically explored for both the one-neuron and multi-neuron models. The experimental setup involved the application of three studied learning algorithms: SGD , SGD+ (SGD with momentum), and Adam. The resulting plots provide a comprehensive comparison of the performance across these learning algorithms for the specified learning rates.

The following figures visually depict these curves, illustrating the impact of different learning rates on the convergence behavior of the models under the influence of SGD, SGD+, and Adam

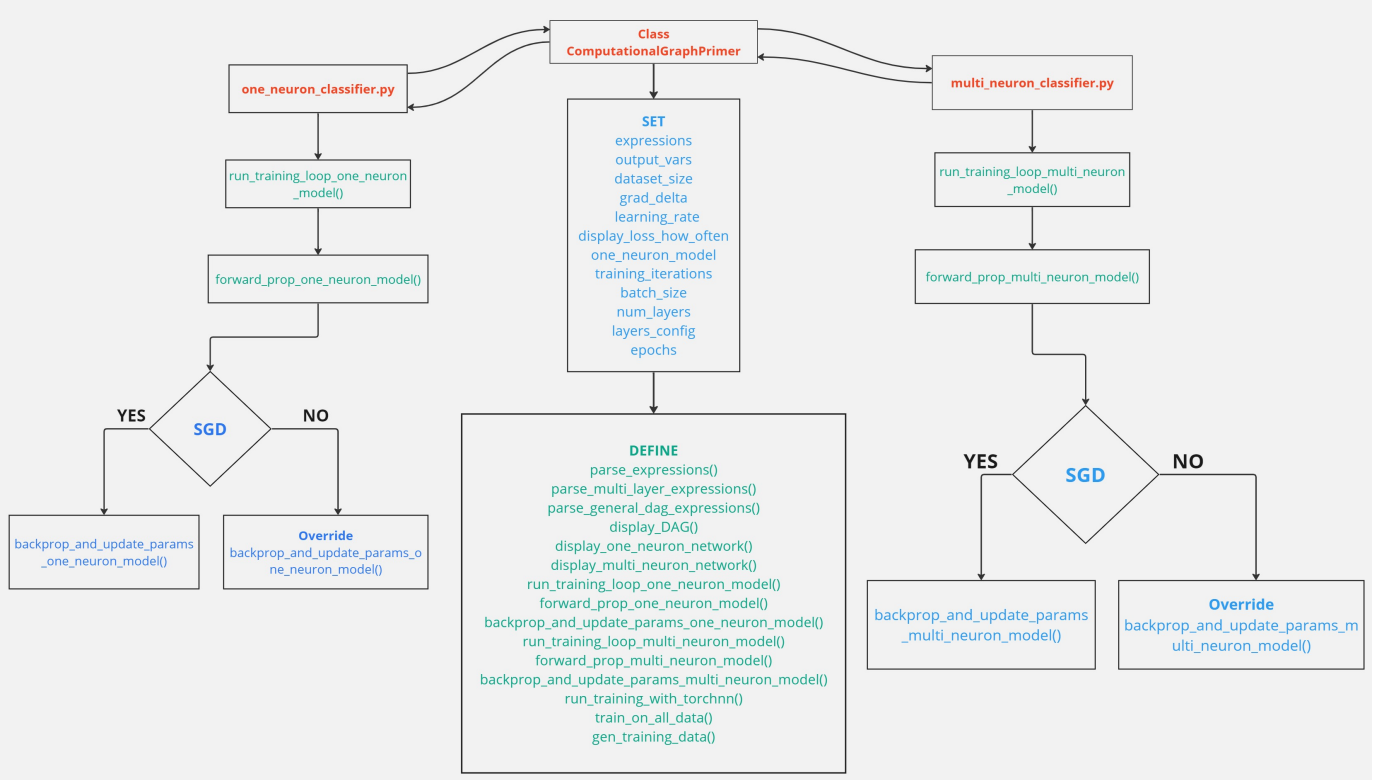

Figure 5: ComputationalGraphPrimer Structure

optimization algorithms. These visualizations serve as valuable insights into the influence of learning rate selection on the training dynamics and convergence efficiency for both one-neuron and multi-neuron models.

Furthermore, noteworthy observations include the noticeable increase in loss for higher learning rates. This phenomenon can be attributed to the larger step sizes associated with higher learning rates. As the algorithm takes larger steps during optimization, it might overshoot the optimal point and escape from local minima. Subsequently, the gradient descent process attempts to search for a better optimum, likely in the direction of steepest ascent compared to the previously encountered local minima. This behavior can result in the loss temporarily increasing before the model (hopefully) converges to a better solution.

This experiment sheds light on the critical role of the learning rate in deep neural network training. The choice of an appropriate learning rate is very important in achieving convergence, preventing overshooting, and navigating the trade-off between exploration and exploitation during the optimization process. As such, it underscores the need for careful tuning of learning rate, to ensure the successful training of deep neural networks.

3.2 Hyperparameter Tuning the the Adam Optimizer:

Two distinct experiments were undertaken to investigate the impact of hyperparameter selection on the performance of the Adam optimizer in the context of a multi-neuron classifier. The experiments involved varying values for β_1 within the range [0.7, 0.9, 0.99] and β_2 within the range [0.8, 0.9, 0.99]. Each experiment consisted of 50,000 training iterations with a batch size of 8, aligning with the task's specifications.

Figure 6: one neuron model for lr=0.0001

As outlined in the task document, the appropriate selection of hyperparameter values is crucial for enhancing a model's accuracy, generalization capabilities, and its ability to extract meaningful patterns from the training data. To further assess the sensitivity of the results to batch size variations, additional experiments were conducted with a batch size of 32.

The tabulated results, presented in the accompanying figures (12 and 13) indicate that the optimal combination of hyperparameters, particularly evident in larger batch sizes, is achieved when β_1 is set to 0.9 and β_2 is set to 0.99. Some combination of the two parameters provide very poor results (e.g. $\beta_1 = 0.6$ and $\beta_2 = 0.6$ in batch size 32 case). While not immediately apparent in the obtained results, these specific hyperparameter values play a potentially pivotal role, especially when dealing with training datasets of larger sizes.

4 Conclusion

This task provided a comprehensive exploration of three key optimization algorithms for training neural networks: Stochastic Gradient Descent (SGD), Stochastic Gradient Descent with momentum (SGD+), and the Adam optimizer. Through experimental analyses, we gained valuable insights into their respective performances under various scenarios and the critical role played by hyperparameter tuning.

The significance of meticulous hyperparameter selection, particularly for the learning rate in SGD and SGD+, and for β_1 and β_2 in the Adam optimizer, was underscored. The experiments highlighted the profound impact of these hyperparameters on the convergence, efficiency, and generalization capabilities of the training process. Specifically, the experiments with Adam optimizer

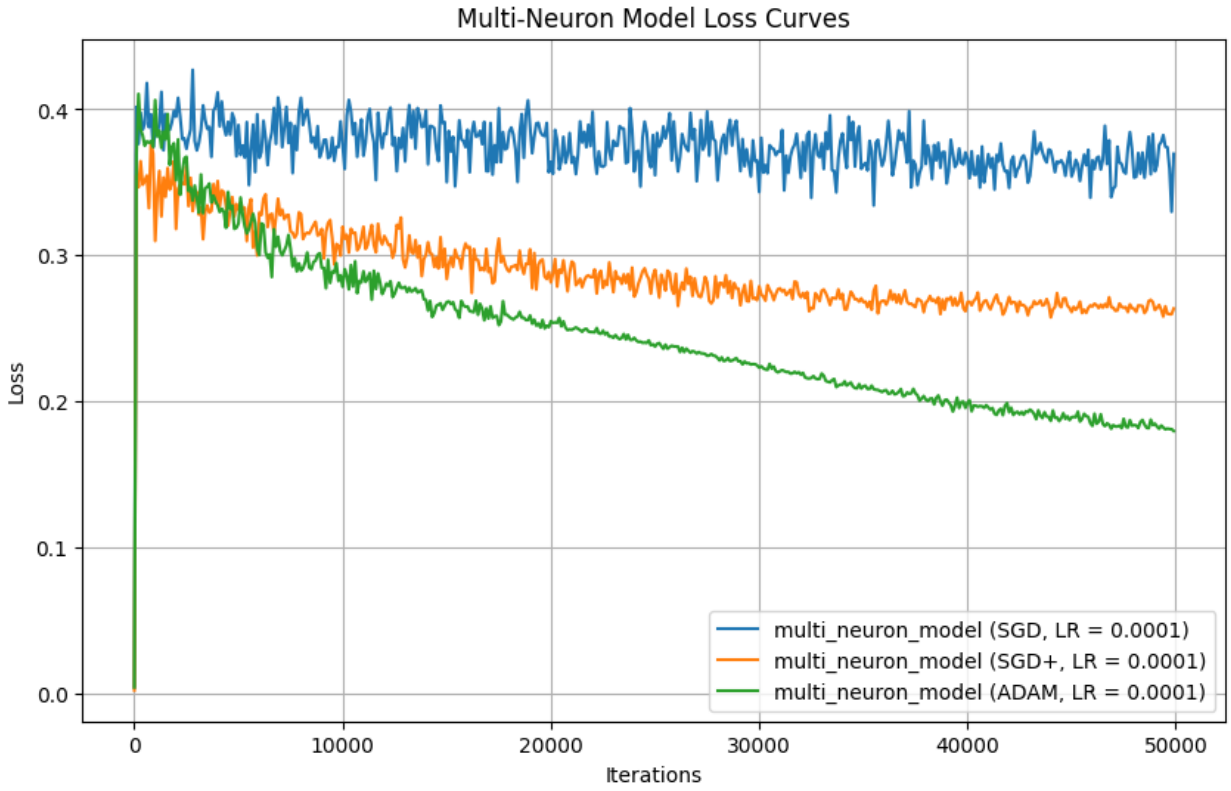

Figure 7: multi neuron model for lr=0.0001

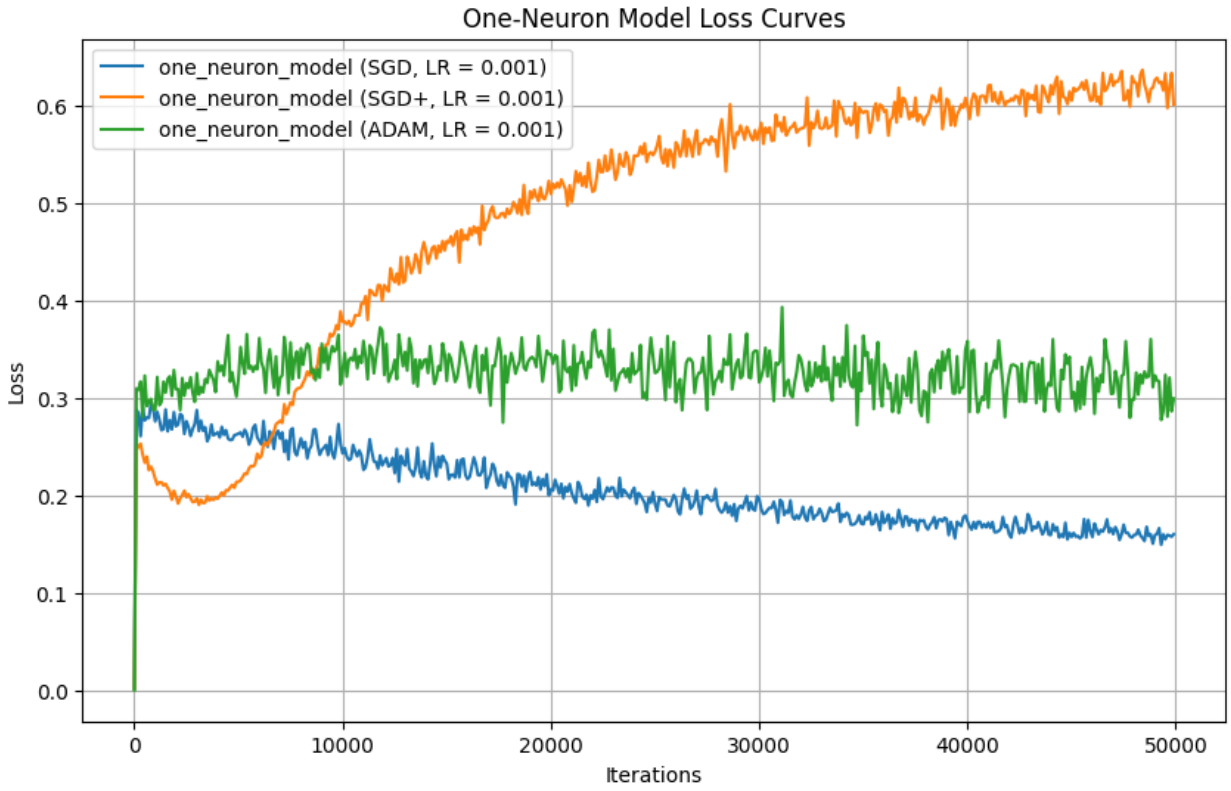

Figure 8: one neuron model for lr=0.001

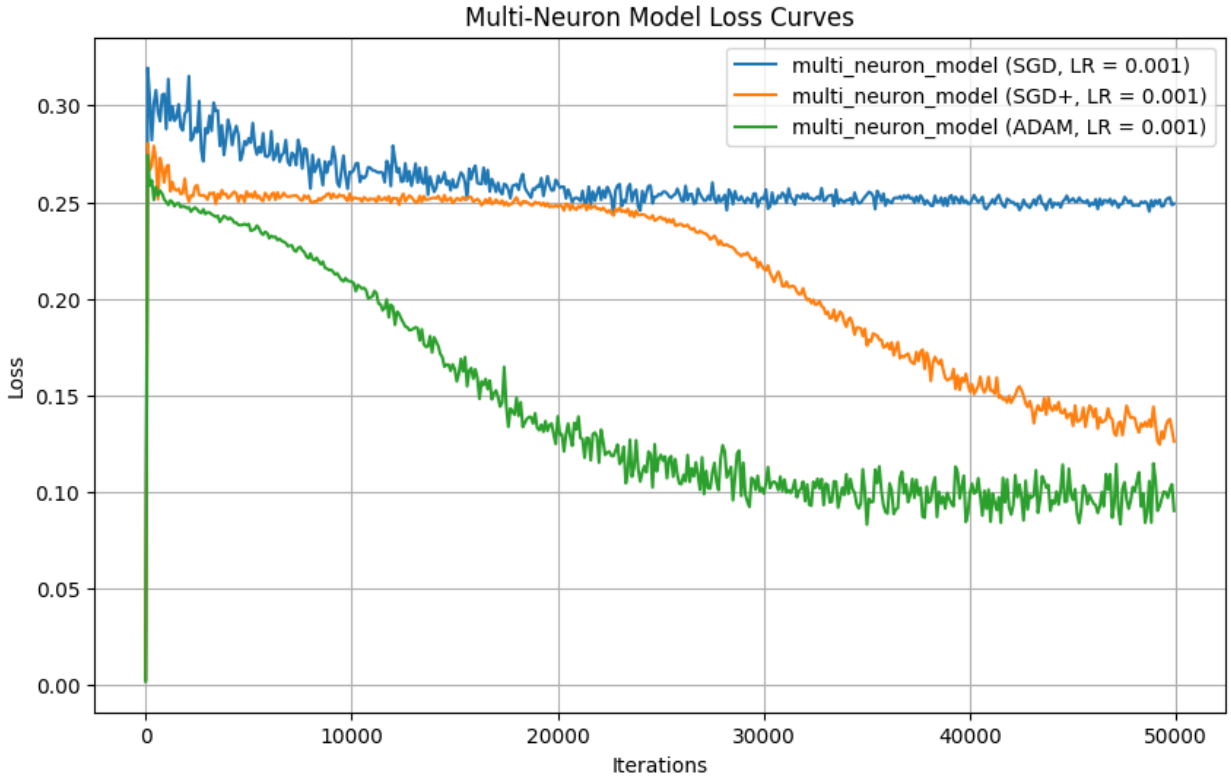

Figure 9: multi neuron model for lr=0.001

demonstrated the sensitivity of results to variations in β_1 and β_2, emphasizing the need for careful tuning to achieve optimal model performance.

This task not only delved into the theoretical aspects of these optimization algorithms but also provided practical insights through experimentation. The findings reiterate the importance of a nuanced approach to hyperparameter tuning. Moving forward, this understanding will serve as a valuable guide for students seeking to optimize the performance of their neural networks in real-world applications.

References

[1] Autograd for Automatic Differentiation and for Auto-Construction of Computational Graphs. URL `https://engineering.purdue.edu/DeepLearn/pdf-kak/AutogradAndCGP.pdf`.

[2] URL `http://tinyurl.com/y3xyabus`

[3] URL `https://engineering.purdue.edu/kak/distCGP/`

Figure 10: one neuron model for lr=0.01

Figure 11: multi neuron model for lr=0.01

	Beta1	Beta2	Time Taken (s)	Final Loss	Minimum Loss
1	0.70	0.70	79.29	0.0862	0.0024
2	0.70	0.90	78.79	0.0935	0.0029
3	0.70	0.99	77.38	0.1007	0.0027
4	0.90	0.70	79.16	0.0936	0.0045
5	0.90	0.90	78.23	0.2030	0.0041
6	0.90	0.99	79.68	0.2027	0.0036
7	0.99	0.70	88.98	0.2020	0.0038
8	0.99	0.90	85.50	0.1902	0.0040
9	0.99	0.99	83.73	0.2147	0.0049

Figure 12: Hyperparameter tuning batch_size=8

	Beta1	Beta2	Time Taken (s)	Final Loss	Minimum Loss
1	0.60	0.60	291.80	0.2090	0.0027
2	0.60	0.90	293.64	0.1967	0.0044
3	0.60	0.99	289.20	0.2083	0.0029
4	0.90	0.60	282.63	0.0823	0.0026
5	0.90	0.90	284.64	0.2043	0.0032
6	0.90	0.99	287.50	0.2242	0.0028
7	0.99	0.60	288.14	0.1968	0.0041
8	0.99	0.90	288.10	0.1983	0.0036
9	0.99	0.99	287.29	0.0743	0.0032

Figure 13: Hyperparameter tuning batch_size=32