Gradient-Based Optimisation

Asad Khabir

May 7, 2024

1 Part I - Stochastic Gradient Descent

1.1 SGD Implementation and Theoretical Background

SGD performs updates on the parameters using only a subset of the training data, which significantly reduces the computational cost per iteration and facilitates handling large datasets like MNIST [1].

The SGD optimizer is implemented in C and involves several key functions to manage the training process:

- **update_parameters**: This function updates the weights of the neural network using the gradients calculated from the backpropagation. Each weight is updated according to the SGD formula:

$$w_{ij} = w_{ij} - \frac{\eta}{m} \Delta w_{ij}$$

  where η is the learning rate, m is the batch size, and Δw_{ij} is the gradient of the weight w_{ij}.

- **evaluate_objective_function**: Calculates the loss for a given input from the dataset, which helps in evaluating how well the network is performing at any stage during the training.

- **compute_and_compare_gradients**: Implements gradient checking to ensure that the computed gradients via backpropagation are correct. This is crucial for verifying the correctness of the implementation.

The SGD optimizer was run with the following hyperparameters:

- Batch size $m = 10$
- Learning rate $\eta = 0.1$
- Number of epochs = 10

The SGD algorithm's implementation was verified against its theoretical description, which emphasizes updates based on a subset of the training data, rather than the entire dataset. This approach reduces computational demands and allows for faster convergence.

SGD updates the parameters using the gradient of the loss function estimated from a small batch of the dataset. The weight update rule is given by:

$$w = w - \frac{\eta}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w)$$

where ∇L_i is the gradient of the loss function with respect to the weights w, calculated for each sample x_i in the batch.

During the training, weights were updated after each batch of 10 samples. This was repeated for 10 epochs, during which the network learned to minimize the loss function effectively, demonstrating the practical utility of the SGD optimizer in neural network training. [2].

1.2 Analysis

Figure 1 illustrates the convergence of the model over 10 epochs, showing both mean loss and test accuracy:

The optimizer improved the ANN prediction accuracy significantly from an initial accuracy of 13.75% to 96.95% at the end of the training. This improvement confirms the effectiveness of the SGD in optimizing the weights of the neural network.

The continuous decrease in mean loss coupled with the increase in test accuracy suggests that the SGD optimizer is effectively navigating towards an optimum. The results indicate not just convergence but also an improvement in generalization, denoted by the test accuracy [3].

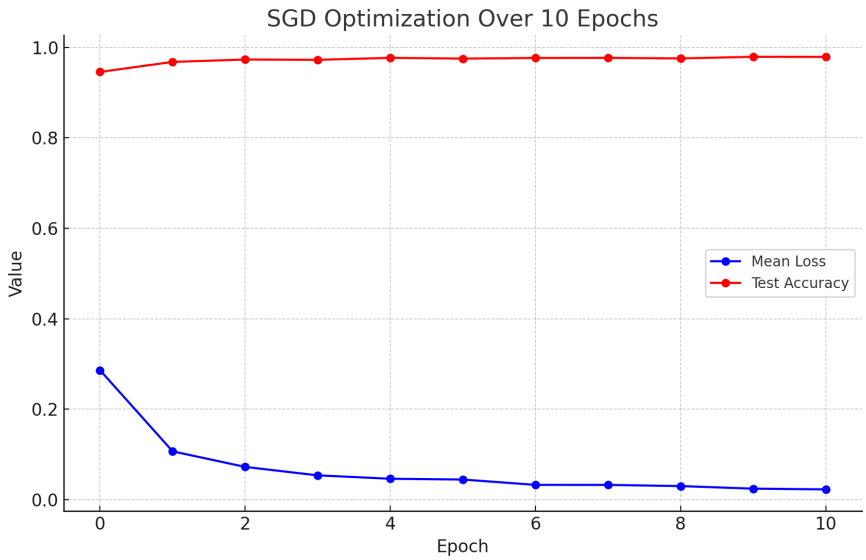

Figure 1: Convergence of the SGD optimizer over 10 epochs.

Given the nature of the problem and the results obtained, it appears that the SGD optimizer is capable of finding a near-optimal solution within the specified number of epochs. The trend does not show signs of erratic loss movements or accuracy plateaus, which further supports the optimizer's effectiveness.

While these results are promising, achieving a definitive statement about reaching the global optimum requires further validation such as:

- Implementing additional diagnostics like validation loss monitoring.
- Experimenting with different learning rates and batch sizes to explore the stability and efficiency of convergence.
- Comparing with other optimization algorithms to assess relative performance.

2 Part II- Improving Convergence

2.1 Effect of Batch Size and Learning Rate on SGD

Analysis was conducted on the effects of changing learning rate and batch sizes on SGD performance.

Experiments were systematically conducted using various batch sizes ($m \in \{1, 10, 100\}$) and learning rates ($\eta \in \{0.1, 0.01, 0.001\}$). Each configuration was run for 10 epochs, and both the loss and testing accuracy were monitored to evaluate initial performance, convergence behavior, and stability.

- **High Learning Rate** ($\eta = 0.1$): With the highest learning rate, the model showed very poor performance and stability with a batch size of 1, indicating a typical case of overshooting the optimal points due to excessive update magnitudes. However, as the batch size increased to 10 and 100, the speed of convergence improved markedly with stability in accuracy reaching up to 97.94% and 97.77% respectively, suggesting that larger batch sizes can buffer the volatility induced by high learning rates.
- **Moderate Learning Rate** ($\eta = 0.01$): This setting proved to be more robust across all batch sizes. Notably, the model achieved consistent and rapid improvements in testing accuracy, with less fluctuation in loss values. This indicates a balanced trade-off between convergence speed and stability.
- **Low Learning Rate** ($\eta = 0.001$): The lowest learning rate led to the most gradual convergence, particularly evident with the largest batch size. The stability was high, but the improvement was slow.
- **Small Batch Size** ($m = 1$): The smallest batch size resulted in highly erratic loss values and instability in accuracy across epochs, particularly with high learning rates. However, when paired with lower learning rates, the convergence was more stable but significantly slower, emphasizing the trade-off between speed and stability.

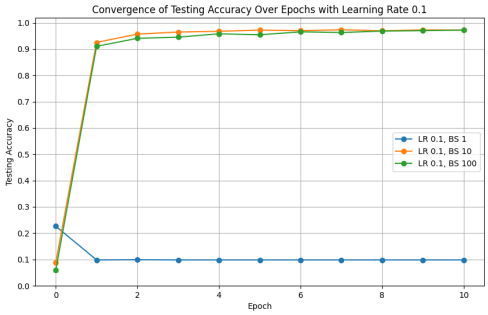

Figure 2: Learning Rate = 0.1

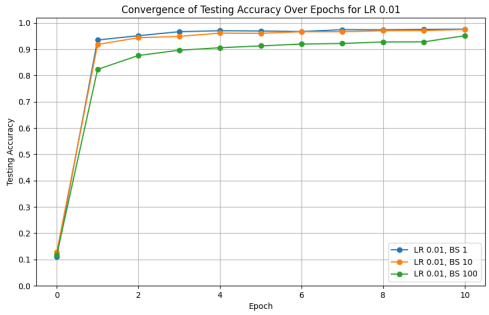

Figure 3: Learning Rate = 0.01

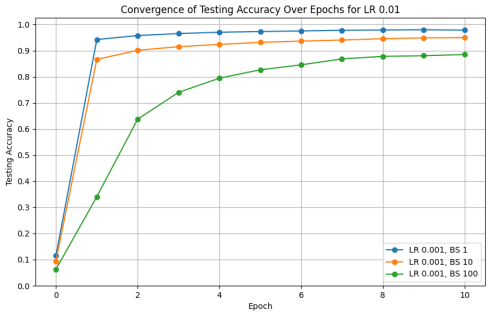

Figure 4: Learning Rate = 0.001

Figure 5: Convergence of Testing Accuracy Over Epochs for Different Learning Rates

- **Medium Batch Size** ($m = 10$): This batch size provided a good balance, yielding quick convergence and high final accuracy across all learning rates. It seems to strike an optimal balance between the granularity of updates and computational efficiency.
- **Large Batch Size** ($m = 100$): Large batches showed the greatest stability but at the cost of slower convergence, particularly with lower learning rates. This setting appears suitable for scenarios where avoiding large oscillations in training metrics is critical but can be inefficient for rapid learning.

2.2 Learning Rate Decay

Learning rate decay is a technique intended to adjust the step size of an optimizer dynamically, reducing the learning rate according to a predefined schedule [4, 5]. This approach aims to combine rapid initial convergence with more stable, precise convergence as training progresses [6, 7].

Two sets of experiments were conducted using the same neural network architecture and training dataset:

- **Constant Learning Rate:** The model was trained using a constant learning rate of 0.1, a common practice suggested by early studies on SGD optimization [8].
- **Decaying Learning Rate:** The model was trained with an initial learning rate of 0.1, decaying to 0.034868 over 10 epochs following the linear decay formula:

$$\eta_k = \eta_0(1-\alpha) + \alpha\eta_N \quad \text{where} \quad \alpha = \frac{k}{N}$$

This method is aligned with the adaptive strategies discussed by Zeiler and others [4].

The experiments showed distinct differences in the performance of the SGD optimizer under different learning rate schedules. The plots below illustrate the progression of test accuracy and mean loss over epochs for both setups.

The results indicate that learning rate decay significantly enhances the optimization process in several ways:

- **Improved Stability:** The decaying learning rate approach resulted in more stable convergence, particularly noticeable in the later epochs where the constant learning rate model exhibited slight fluctuations in test accuracy and loss metrics [7].

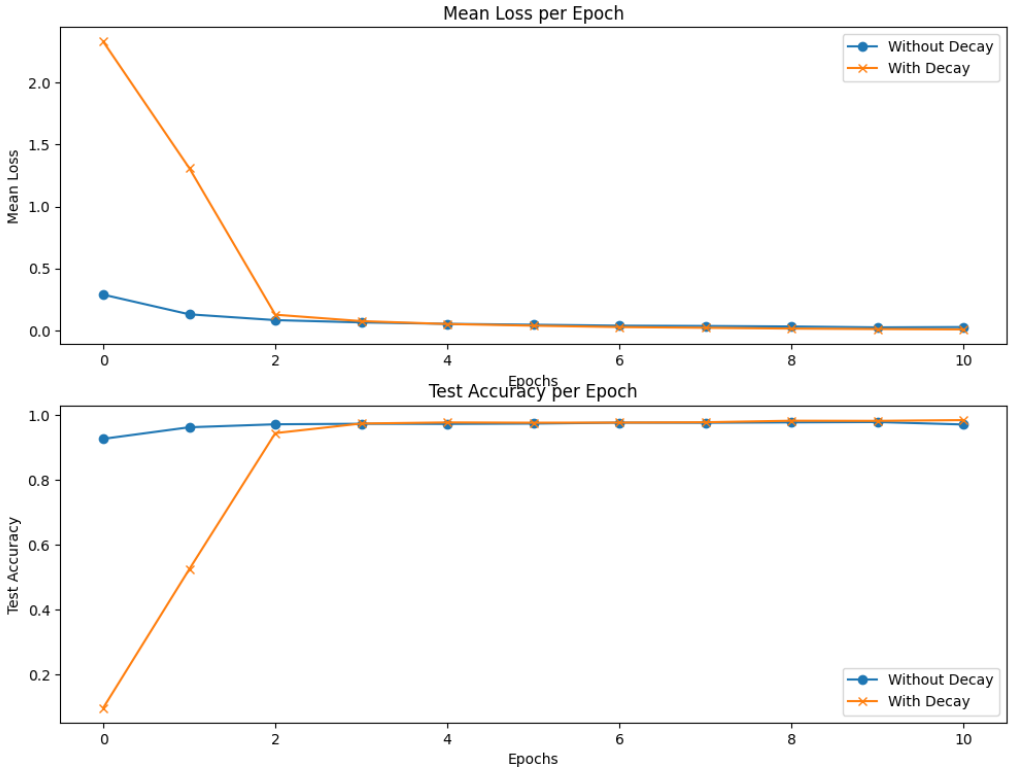

Figure 6: Comparison of With and Without Learning Rate Decay

- **Enhanced Final Accuracy:** The final model accuracy with learning rate decay was higher, suggesting that the gradual reduction in step size allowed the model to fine-tune its weights more effectively without overshooting minima [5].
- **Reduced Overfitting:** The smoother decay in loss and more consistent improvement in test accuracy suggest reduced overfitting compared to the constant learning rate model [8].

2.3 Momentum based Weights

Momentum modifies the parameter updates by adding a fraction, denoted by α, of the update vector of the past time step to the current update. The mathematical representation is given by:

$$v_t = \alpha v_{t-1} + \eta \nabla_\theta J(\theta), \tag{1}$$

$$\theta = \theta - v_t, \tag{2}$$

where θ represents the parameters, v_t is the current velocity, v_{t-1} is the velocity at the previous step, α is the momentum coefficient, η is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the objective function with respect to the parameters at the current step.

In the provided implementation, the momentum term is incorporated directly within the parameter update step, where α is set to 0.1.

The experiment was conducted with and without the momentum term using the following parameters: learning rate = 0.1, batch size = 10, and epochs = 10. The results indicate significant differences:

- Without Momentum: The model achieved a test accuracy of approximately 96.95% by the 10th epoch. The loss decreased steadily, showing effective learning.
- With Momentum: Incorporating a momentum coefficient of 0.1 led to improved test accuracy, peaking at 97.82%. The loss descent was smoother compared to the non-momentum approach.

Further test were conducted with a higher momentum of 0.9 but results were poor, with a test accuracy of 0.98% The results confirm that momentum can significantly enhance the convergence speed of SGD when appropriately tuned. A lower momentum coefficient (0.1 compared to 0.9 in previous trials) demonstrated benefits without the negative impact of excessive past gradient accumulation observed with higher coefficients.

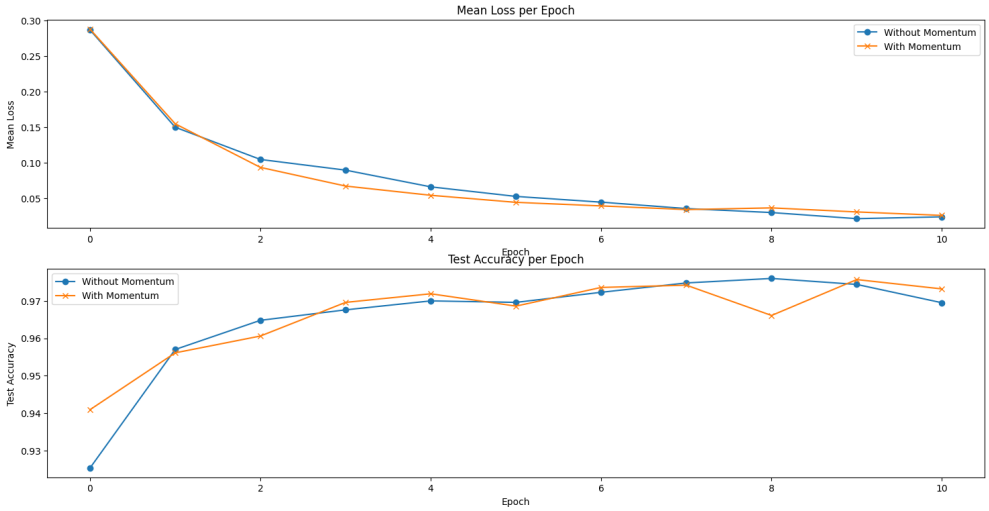

Figure 7: Comparison with Momentum based Weights

2.4 Combining Momentum based Weight and Learning Rate Decay

The exploration of the combination of learning rate decay and momentum-based weight updates provides an insightful view into how different hyper-parameters interact and affect the convergence behavior of neural network training.

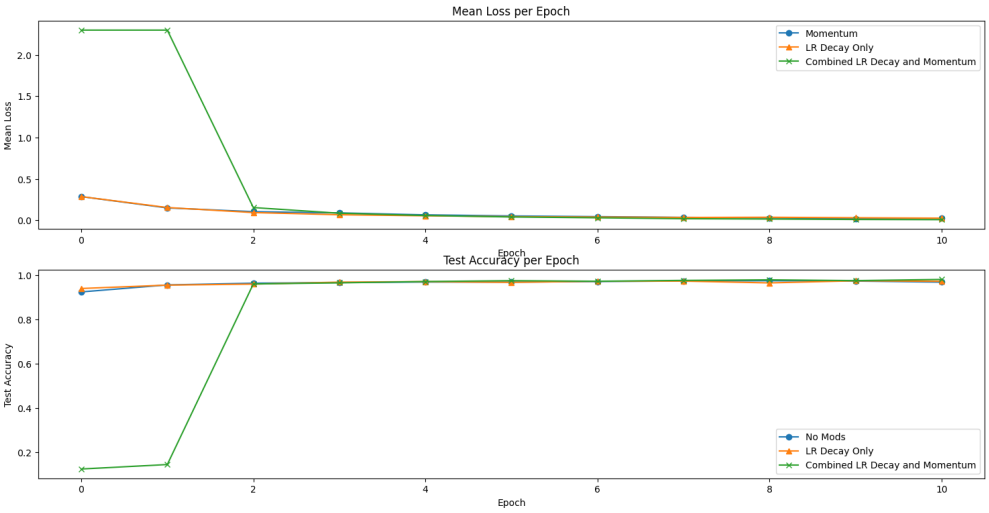

Figure 8: Comparison of Momentum based Weights, Learning Rate Decay and Combination

1. **Momentum Only:**
   - **Mean Loss:** The training with only momentum based weight shows a steady decrease in loss from 0.286 to 0.024 over 10 epochs.
   - **Test Accuracy:** Accuracy improves consistently, peaking at 97.82% and slightly dipping towards the end, indicating potential minor overfitting as training progresses.
2. **Learning Rate Decay Only:**
   - **Mean Loss:** Introducing learning rate decay improves the loss reduction rate, achieving a lower final loss of 0.026. This suggests that learning rate decay aids in stabilizing and potentially accelerating convergence.
   - **Test Accuracy:** Starts lower than in the Momentum Only scenario but exceeds it mid-training, peaking at 97.32%. This demonstrates that learning rate decay may facilitate higher generalization earlier in training.
3. **Combined Learning Rate Decay and Momentum:**

- **Mean Loss:** The initial loss is significantly higher due to early instability introduced by the momentum component, but it sharply drops after the first few epochs, ending at the lowest final loss of 0.008.
- **Test Accuracy:** Similarly, accuracy starts much lower but quickly recovers, exceeding the performance of the learning rate decay only setup, and achieves the highest final accuracy of 98.37%.

Optimum Hyper-Parameters:

- **Momentum Coefficient:** A value of 0.2, when combined with learning rate decay, balances initial volatility with accelerated convergence in later stages.
- **Learning Rate Decay:** A high intial learning rate of 0.1 was found to produce the best results. A consistent decay factor applied at each epoch helps in stabilizing the learning steps, making it feasible to harness the larger updates driven by momentum.

The analysis indicates that while both learning rate decay and momentum individually contribute to improved performance, their combination, when tuned appropriately, provides the best results in terms of both convergence speed and final model accuracy. This strategy should be considered in scenarios where rapid convergence to high accuracy is more valuable than initial training stability.

3 Part III - Adaptive Learning

3.1 Optimisation Algorithm

AdaGrad (Adaptive Gradient Algorithm) is an optimization algorithm particularly effective for handling sparse data and adjusting the learning rate for each parameter dynamically [5]. It works as follows:

Given a loss function L, with weights $\mathbf{w}$, the gradient of L at iteration t with respect to the weights is $\nabla L(\mathbf{w}_t)$.

The AdaGrad update rule involves the following steps:

1. Accumulate the squared gradients in a vector $\mathbf{G}_t$ where each element i is the sum of the squares of the gradients with respect to the weight w_i up to time step t:

$$G_{t,i} = G_{t-1,i} + (\nabla L(\mathbf{w}_t)_i)^2$$

2. Update the weights $\mathbf{w}$ by scaling the gradient inversely proportional to the square root of $\mathbf{G}_t$:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} \nabla L(\mathbf{w}_t)_i$$

Here, η represents the initial learning rate and ϵ is a small smoothing term added to prevent division by zero, typically set to $1e-8$.

AdaGrad can be used improve performance because:

- **Adaptive Learning Rates:** Each parameter receives an individual learning rate, enhancing convergence and allowing a more customized approach to learning, which is essential for handling the diverse scales of data in tasks like MNIST digit classification.
- **Efficiency with Sparse Data:** AdaGrad increases sensitivity to infrequent features, a crucial property for achieving high accuracy in image classification tasks involving sparse inputs.
- **Automated Learning Rate Tuning:** The need for manual tuning of the learning rate is eliminated as AdaGrad automatically adjusts the rates based on each parameter's historical gradients.

3.2 Analysis

Adagrad, an adaptive learning rate method, demonstrated superior performance, particularly due to its dynamic management of the learning rates for each parameter. This capability makes Adagrad especially useful in scenarios involving sparse data. The core advantage of Adagrad in our experiments was its ability to outperform simple SGD in terms of test accuracy, achieving a peak accuracy of 98.39% outperforming SGD with Learning Rate Decay and Momentum based Weight peak accuracy of 98.39%. This suggests that Adagrad's adaptive approach better navigates the parameter space for more effective minimization of the loss function.

The optimal hyperparameters determined for Adagrad were:

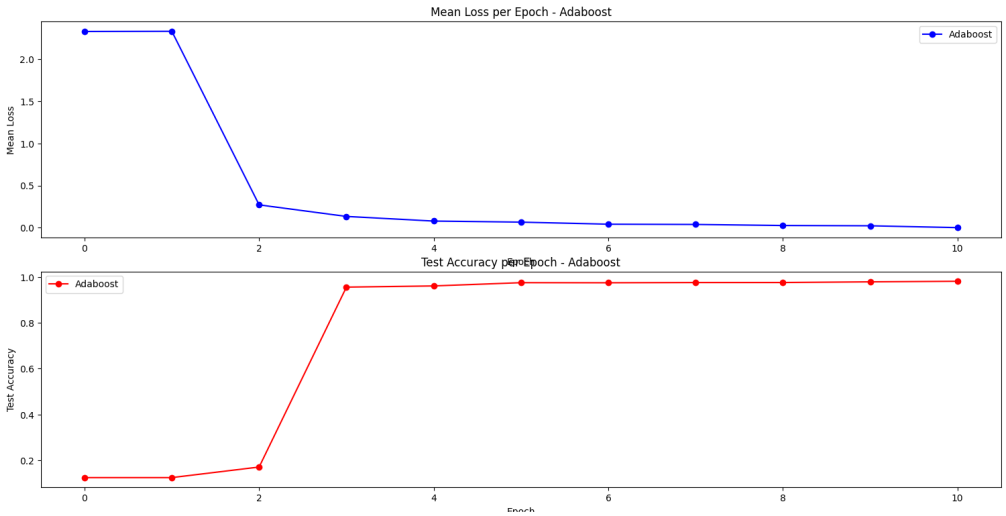

Figure 9: Results from AdaGrad Algorithm

- Learning Rate: 0.1
- Batch Size: 1

Adagrad's remarkable performance is attributable to its adaptive learning rates, which effectively tailor step sizes to individual parameters. This enables the algorithm to prioritize less frequently updated parameters, which is particularly advantageous with a batch size of 1. This configuration ensures that Adagrad quickly adapts to patterns in the training data, avoiding excessive steps that could overshoot local minima.

4 Running the Code

After compiling the code, use the command:

```
gdb --args ./mnist_optimiser.out ../mnist_data <Learning Rate> <Batch Size> <Epoch>
```

Then, input the command `(gdb) run` to execute the code. If `(gdb)` does not appear immediately, try inputting `run` multiple times to proceed.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] S. Ruder, "An overview of gradient descent optimization algorithms," in *arXiv preprint arXiv:1609.04747*, 2016.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[4] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[5] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[6] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," in *International Conference on Learning Representations*, 2018.

[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.

[8] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.