36212-EX3-S-Optimisation

Bowen Jing

May 12, 2023

I Stochastic Gradient Descent

Optimizing Artificial Neural Networks is crucial in deep learning, as it enables the discovery of accurate and efficient models that can effectively generalize to new data, driving advancements in a wide range of applications and domains. Nevertheless, the prevailing optimization technique for classifying handwritten digits is Stochastic Gradient Descent (SGD)[11], which serves as a stochastic approximation to the well-established optimization algorithm, Gradient Descent. By approximating the gradient using a data subset rather than the entire dataset, SGD reduces the computational complexity typically found in high-dimensional optimization problems, thus trading a more rapid iteration process for a slower convergence rate[1]. The purpose of this coursework is to demonstrate the optimization process involved in training a neural network. A MLP might be considered more suitable than a CNN for this task due to its simpler architecture and more straightforward weight-update mechanisms. With fewer architectural complexities, such as convolutional layers and pooling operations, MLPs make it easier to understand and visualize the gradient-based optimization process.[4] To accomplish the aims mentioned above, I conducted a series of experiments. All the experiments were run on my desktop. The hardware information is as follows: **CPU:** AMD Ryzen 9 5900X 12-Core Processor 3.70 GHz. **GPU:** Nvidia GeForce RTX 3080. **Memory:** Available memory 18.0 GB.

A Optimiser Implementation

The behaviour of stochastic gradient descent(SGD) could be described by the following theorem:

Theorem 1

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

where w: represents the weight between two neurons. η represents the learning rate. m represents the batch size. $\nabla L_i \left(x_i, w \right)$: represents the gradient of the objective function $L_i \left(x_i, w \right)$, where $L = -\sum_{k=0}^{N} \hat{y}_k \log \left(P_k \right)$ with L representing the loss.

All this section's description about the skeleton optimization framework could be found in the *update_parmeters*() in *optimiser.c*. The detailed implementation is as follows:

I first design a method *update_weights*() to take the parameters and update the information based on the theorem 1.

```
void update_weights(weight_struct_t weights[][N_NEURONS], int n_rows, int n_cols, unsigned int
↪  batch_size) {
    for (int i = 0; i < n_rows; i++) {
        for (int j = 0; j < n_cols; j++) {
            weights[i][j].w -= ((learning_rate / batch_size) * weights[i][j].dw);
            weights[i][j].dw = 0.0;
        }
    }
}
```

Then, I just call this funtion four times with the correpond parameters for four times to update all the value in the *update_parmeters*() method

```
update_weights(w_LI_L1, N_NEURONS_LI, N_NEURONS_L1, batch_size);// Input->L1
update_weights(w_L1_L2, N_NEURONS_L1, N_NEURONS_L2, batch_size);// L1 -> L2
update_weights(w_L2_L3, N_NEURONS_L2, N_NEURONS_L3, batch_size);// L2 -> L3
update_weights(w_L3_LO, N_NEURONS_L3, N_NEURONS_LO, batch_size);// L3 -> Output
```

B Validation through Numerical Differentiation

Theorem 2

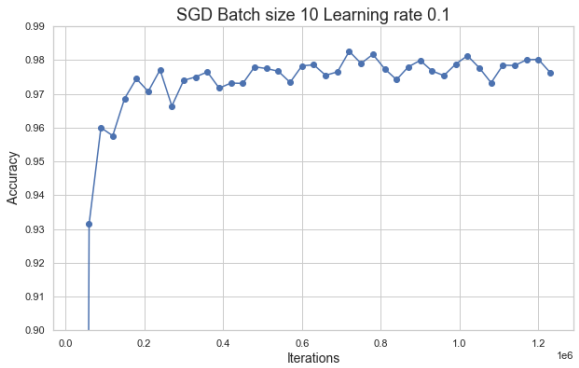
$$\nabla L(x, w) = \frac{L(x, w + \eta) - L(x, w)}{\eta} \tag{2}$$

where η represents the step size.

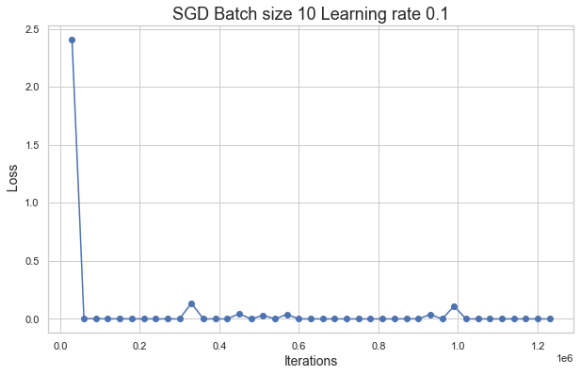
To implement this, I develop a new function within the the optimizer.c file, which I will name numerical_derivative(). Inside this function, I will initially calculate the current loss using the method evaluate_forward_pass() and compute_xent_loss(). Subsequently, I will update the weight of a single connection in the weight matrix W_layer1_layer2(where 'layer1' and 'layer2' denote two interconnected layers) by the step size(0.1) and recalculate the loss using the same method.

Next, I utilize the earlier equation to obtain the numerical approximation of the gradient and compare this with the actual gradient stored in W_layer1_layer2[i][j].gradient. To confirm that the gradient of the analytical solution is stored accurately, we also need to run the methods evaluate_backward_pass_sparse() and store_gradient_contributions(). Finally, I can compare the two values and ascertain the accuracy of the analytical solution. In the end, the testing error was found at around 10%.

C Experiment



(a) Training Accuracy



(b) Training Loss

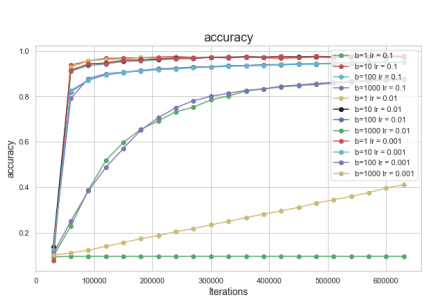
In this experiment, the model is trained for 20 epochs with a learning rate of 0.1 and a batch size of 10. From Fig. (a), the model has achieved peak accuracy of 98.3%. However, the performance shows instability. Specifically, the accuracy has been oscillating within range between 97% and 98%. But the training loss ceases to diminish and becomes stable after 20000 iterations. The stability of the training loss signifies that the model has located an optimum and started to converge. Even if I allocate more epochs for training the model, the performance will not show a significant increase. However, reaching an optimum with a specific model architecture and hyperparameter configuration does not mean the best potential performance. Higher accuracy could be reached through strategies such as dataset preprocessing, optimization of the model's architecture, and hyperparameter selection. As stated by Dan Claudiu Ciresan and colleagues in their research paper, the highest accuracy achieved by a Multilayer Perceptron on the MNIST dataset has reached 99.65%.[3] This serves as a

motivating factor for conducting the following experiments, aimed at exploring the characteristics of each hyperparameter and their impact on the model's performance.

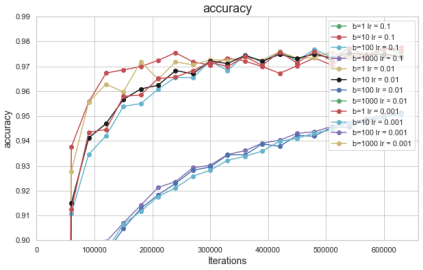
II Improving Convergence

In this section, I will explore the effects of learning rate and batch size on model accuracy. The investigation will commence with a series of experiments aimed at understanding the influence of these hyperparameters on the model's performance. Subsequently, I will delve into optimization techniques, specifically focusing on learning rate decay and momentum-based weight updates. These strategies are designed to enhance convergence in optimization algorithms by adaptively adjusting the learning rate and integrating past gradients into the optimization process. By incorporating these techniques into our training regimen, we will assess their efficacy in improving the model's overall performance and stability.

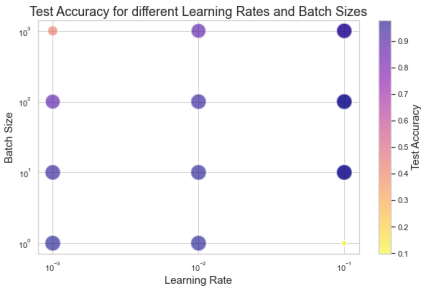
A Tuning batch size and learning rate



(a) Accuracy



(b) Same results but shown more details



(c) Accuracy plot

Learning rate is an important factor which will effect the performance of the SGD. A high learning rate could expedite convergence but may also result in instability and oscillations. On the other hand, a small learning rate may lead to slow convergence or entrapment in suboptimal local minima. Moreover, the learning rate directly impacts training speed. While high rates can foster rapid initial learning, they may also instigate instability. In contrast, extremely low rates, despite eventually reaching a satisfactory solution, might demand many more training epochs.

Interestingly, the ratio of batch size to learning rate emerges as a significant factor. Specifically, when this ratio exceeds 10,000, convergence to the optimum noticeably slows down. Below a ratio of 10, the model fails to converge or generalize effectively.

A few papers proved that sharper minima leads to worse generalisation performance for SGD[9], while Sepp Hochreiter et al hold the different opinions, they think SGD function performs better on a wide minima[6]. Taking into account the results shown above, we can hypothesize that the ratio of learning rate to batch size is directly related to the ability of SGD to converge to a wider minimum[8]. Actually, this has been empirically validated in previous studies, which have demonstrated that this ratio directly determines the ability of SGD to generalize and converge to the optimum[8][2][5]. It has been illustrated in previous studies that this specific ratio directly influences SGD's capacity to generalize and achieve convergence to an optimum.

B Learning rate decay

The learning rate is an important hyperparameter in SGD, governing the size of the steps taken in parameter space towards the direction of the minimum loss. If the learning rate is set too high, SGD can overshoot the minimum and cause the model to fail to converge or even diverge. On the other hand, if it's set too low, the model may take an long time to converge, or it may get stuck in a local minimum. Learning rate decay is one strategy to counteract these issues. It starts with a relatively high learning rate to allow rapid initial progress and prevent getting stuck in poor local minima. Then,

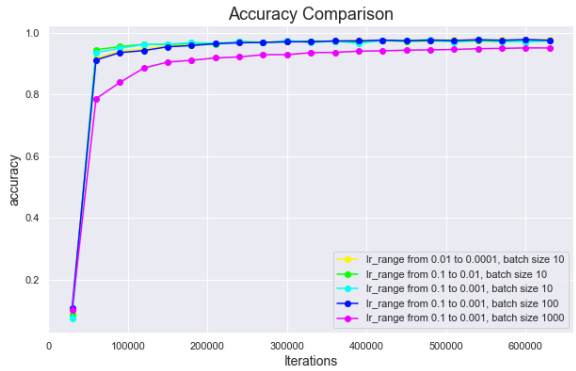
over time, the learning rate gradually reduces, enabling fine-tuning of the model parameters and more precise convergence towards the optimal solution. When compared to normal SGD learning rate decay shows more stability towards the optimum while also providing rapid initial convergence. This makes sense since learning rate governs how malleable the model is to changes in weight, by reducing the learning rate as we go further into the interations the model[17].

Theorem 3

$$\eta_k = \eta_0(1-\alpha) + \alpha\eta_N \;\; where\; \alpha = \frac{k}{N} \tag{3}$$

I set two new global variables in optimiser.c. One is the initial learning rate another is the final learning rate. And then I implement the equation 3 as we can see from the code snip.

```
void learning_rate_decay(unsigned int epoch_counter){
    learning_rate = initial_learning_rate * (1 - (epoch_counter/total_epochs)) +
    ((epoch_counter/total_epochs) * final_learning_rate);
}
```



(a) Learning rate decay accuracy

(b) zoom in

The above discussion highlights the benefits of learning rate decay and increasing batch size in the context of SGD. Learning rate decay, in comparison to regular SGD, exhibits greater stability around the optimum and allows for swift initial convergence. This is attributable to its dynamic adjustment of the learning rate as training progresses, enabling the model to finely tune around the optimum rather than oscillating between local extrema.

On the other hand, it has been demonstrated that increasing the batch size during training can yield similar results, but with added advantages[13]. It allows for better parallelism and can significantly reduce training time, making it a more effective alternative to learning rate decay[7]. Notably, scaling up the batch size increases the step size, reducing the number of parameter updates during model training, which can significantly cut computational time. Moreover, larger batches can be utilized across multiple machines through parallelism, further enhancing efficiency.

Despite the apparent superiority of batch size growth, for the purposes of this assignment, the focus will remain on learning rate decay and momentum-based weight updates. However, the potential benefits of increased batch size present an intriguing avenue for further exploration and optimization of SGD.

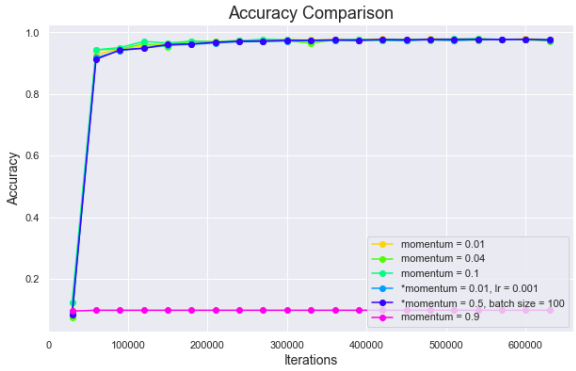
C Momentum based weight update

Momentum in SGD can be conceptualized as a ball rolling down a hill, accelerating learning particularly when the model might find itself trapped in a local minimum. This phenomenon can often occur in complex optimization landscapes where the model may get caught in regions of pathological curvature, resulting in oscillatory behavior. A visual illustration of this is observed in Figure (b), where certain lines depict this back-and-forth oscillation. By introducing momentum, we inject an

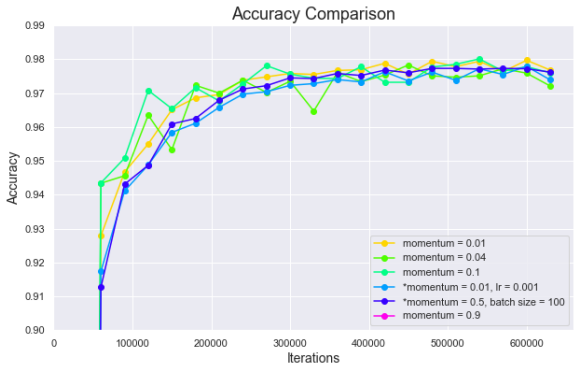
element of 'memory' from previous gradient updates, allowing the model to navigate these difficult regions more effectively and steer towards the global minimum. This analogy of a rolling ball captures the essence of momentum in SGD, providing a more intuitive understanding of its role in enhancing the optimization process.

$$
\begin{aligned}
v &= \alpha v - \frac{\eta}{m} \sum_{i=1}^{m} \nabla L_i \left(x_i, w \right) \\
w &= w + v
\end{aligned}
\tag{4}
$$

```
void update_weights_momentum(weight_struct_t weights[][N_NEURONS], int n_rows, int n_cols, unsigned int batch_size, double alpha, double learning_rate) {
    for (int i = 0; i < n_rows; i++) {
        for (int j = 0; j < n_cols; j++) {
            weights[i][j].v = (alpha * weights[i][j].v) - ((learning_rate / batch_size) * weights[i][j].dw);
            weights[i][j].w += weights[i][j].v;
            weights[i][j].dw = 0.0;
        }
    }
}
//=============================================================================================
update_weights_momentum(w_LI_L1, N_NEURONS_LI, N_NEURONS_L1, batch_size, alpha, learning_rate);
update_weights_momentum(w_L1_L2, N_NEURONS_L1, N_NEURONS_L2, batch_size, alpha, learning_rate);
update_weights_momentum(w_L2_L3, N_NEURONS_L2, N_NEURONS_L3, batch_size, alpha, learning_rate);
update_weights_momentum(w_L3_LO, N_NEURONS_L3, N_NEURONS_LO, batch_size, alpha, learning_rate);
```

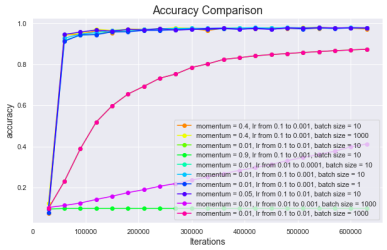


(a) Momentum based weight update accuracy

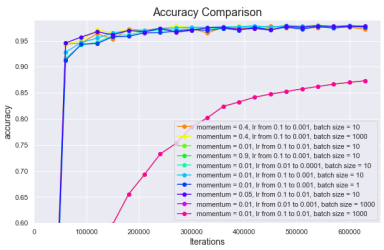


(b) zoom in

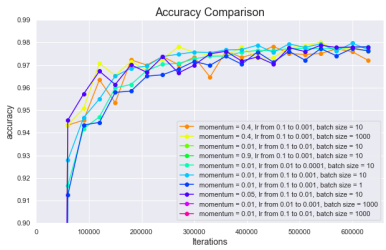
As we can see from the diagram(a), incorporating momentum into SGD is considerated to primarily enhance stability while also improving convergence, However, it's clear that a high momentum value(alpha = 0.9) depicted by the pink line, failed to converge, indicating the critical role of appropriate hyperparameter selection. Despite the zoomed-in graph creating an impression of numerous oscillations, these are minor fluctuations made more noticeable. This stability enhancement, alongside the acceleration in the relevant direction and reduction of noise, provides a compelling argument for the utility of momentum.[15] However, it is also important to consider the increased computational cost and memory requirements, as well as the added complexity of hyperparameter tuning. Maintaining a relatively low decay rate significantly improved stability compared to standard SGD, emphasizing the effectiveness of momentum in optimizing SGD performance.[15].



(a) Overall accuracy



(b) Same results but zoom in



(c) More details

D Learning Rate Decay with Momentum

The fusion of learning rate decay and momentum in our model demonstrates the best of both worlds: the rapid early convergence common with learning rate decay, and the stability often seen with momentum. Such performance aligns with expectations and has been supported by findings from papers[12][14][15]. Evidenced by the light blue line(momentum = 0.01, learning rate from 0.1 to 0.001, batch size = 10), the model exhibits almost no oscillations during the process.

The hyperparameters that emerged as ideal from the tests include a batch size of 10, an initial learning rate of 0.1, a final learning rate of 0.001, and an alpha value of 0.01. Nonetheless, the absence of varied testing on different learning rate combinations could potentially have precluded the emergence of a more efficient set of hyperparameters. The justification for not modifying them lies in their superior performance when tested in isolation with learning rate decay.

III Adaptive Learning

The learning rate is a crucial, yet delicate hyperparameter that can significantly influence the outcomes of machine learning models. Given its importance, various innovative methods have been developed to adaptively select the ideal learning rate during the training process. Among these, Adaptive Moment Estimation, or Adam, has gained significant attention[10]. It has demonstrated superior performance, achieving an impressive 99.12% accuracy on the MNIST dataset with a convolutional neural network, outperforming SGD[16]. Adam is not only computationally efficient and memory-friendly but also well-suited for large datasets. Its hyperparameters carry intuitive interpretations, reducing the need for extensive hyperparameter tuning. Compared to other methods like RMSprop and Adagrad, Adam is more computation-friendly while still delivering high accuracy, making it an excellent choice for our purposes. As such, we chose to delve deeper into the Adam adaptive learning algorithm[10].

A Adam

Theorem 4

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2 \end{aligned} \tag{5}$$

In the equation above, the m_t represents the estimate of the first moment The v_t represents the estimate of the second moment. The β_1 and β_2 represent decay rates. The g_t represents the objective function.

The creators of the Adam optimization algorithm observed that both mt and vt have a tendency to be biased towards zero, given that they are initialized as zero vectors. This effect is more noticeable when the decay rates are small. In order to mitigate these biases, it's crucial to calculate the bias-corrected versions of mt and vt[10].

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{6}$$

These variables are used to update the weight parameters present in the multi-layer perceptron.

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{7}$$

The authors proposed the default values of $0.9, 0.999$, and 10^{-8} for β_1, β_2, and ϵ respectively which will be used for hyperparameter tuning.

B Implementation

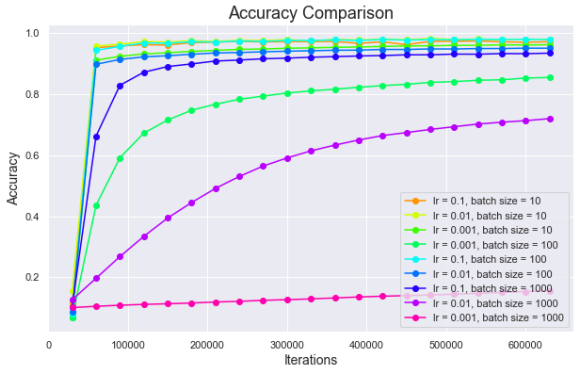
I use update_parameters() as the basic work. I set β_1, β_2, and ϵ are defined as doubles with values of $0.9, 0.999$, and 10^{-8}. And then use for nested loops to calculate the mean(mt) and variance(vt) by employing the equation 5. In each loop, I also calculate the bias corrected versions based on equation 6. Finally, the equation 7 is used to update the weight and set the gradient back to 0.

```
void update_weights_adam(weight_struct_t weights[][N_NEURONS], int n_rows, int n_cols, unsigned int batch_size, double beta_1, double beta_2, double epsilon, double learning_rate) {
    double bias_corrected_mean;
    double bias_corrected_variance;

    for (int i = 0; i < n_rows; i++) {
        for (int j = 0; j < n_cols; j++) {
            weights[i][j].mean = (beta_1 * weights[i][j].mean) + ((1 - beta_1) * weights[i][j].dw);
            weights[i][j].variance = (beta_2 * weights[i][j].variance) + ((1 - beta_2) * (weights[i][j].dw * weights[i][j].dw));

            bias_corrected_mean = (weights[i][j].mean / (1 - beta_1));
            bias_corrected_variance = (weights[i][j].variance / (1 - beta_2));

            weights[i][j].w -= ((learning_rate / ((batch_size * sqrt(bias_corrected_variance)) + epsilon)) * bias_corrected_mean);
            weights[i][j].dw = 0.0;
        }
    }
}
```

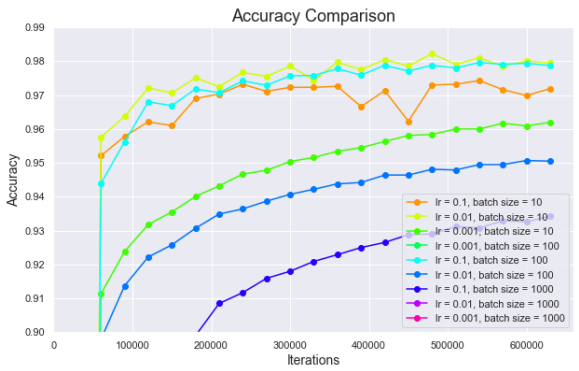
```
update_weights_adam(w_LI_L1, N_NEURONS_LI, N_NEURONS_L1, batch_size, beta_1, beta_2, epsilon, learning_rate);
update_weights_adam(w_L1_L2, N_NEURONS_L1, N_NEURONS_L2, batch_size, beta_1, beta_2, epsilon, learning_rate);
update_weights_adam(w_L2_L3, N_NEURONS_L2, N_NEURONS_L3, batch_size, beta_1, beta_2, epsilon, learning_rate);
update_weights_adam(w_L3_LO, N_NEURONS_L3, N_NEURONS_LO, batch_size, beta_1, beta_2, epsilon, learning_rate);
```

C Discussion

As we can see from the figures above, The overall performance of the Adam optimizer tends to be more stable compared to SGD. These observations underscore the efficacy and robustness of the Adam optimizer in the context of neural network training. Especially, when the learning rate(lr) is 0.01, batch size is 10(yellow line). The yellow line converges rapidly, demonstrating overall stability and commendable accuracy. The observations presented here echo the conclusions of various studies on the Adam optimizer. These studies uniformly highlight the optimizer's attributes of stability, rapid initial convergence, and significant accuracy enhancements. Moreover, the Adam optimizer lessens the



(a) Adam tuning process



(b) Fig.(a) zoom in

reliance on hyperparameters, simplifying the often laborious task of training a neural network, which typically involves extensive trial and error. These findings emphasize the efficiency and resilience of the Adam optimizer within the framework of neural network training.

IV Conclusion

This coursework began with the implementation and exploration of the SGD optimizer. A series of experiments run to investigate the diverse behaviors and impacts of various hyperparameters. Subsequently, the Adam optimizer was implemented, which yielded the highest accuracy among the optimization techniques employed. Although SGD, in conjunction with learning rate decay and momentum, presented a commendable competition, Adam optimizer proved superior in this context.

However, this work is far from finished. Our future endeavors will involve a broader range of experiments and analyses. One key focus will be the evaluation of various training strategies, such as hyperparameter selection and model architecture, in terms of their impact on the risks and generalization abilities of the model. Understanding these facets is essential in order to effectively calibrate the balance between empirical and population risks. This, in turn, will help ensure the model's robustness and reliability when dealing with unseen data.

Moreover, I aim to delve deeper into the intricacies of these strategies. For instance, I plan to investigate how different hyperparameters affect the bias-variance trade-off, which is a fundamental aspect of a model's generalization ability. Similarly, I intend to study the role of model architecture in defining the complexity and capacity of the model, which could significantly influence its performance and adaptability. Thus, the journey continues towards refining and perfecting the model, with the ultimate goal of enhancing not only its predictive accuracy but also its ability to generalize from specific instances to broader patterns.

References

[1] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20, 2007.

[2] Thomas M Breuel. The effects of hyperparameters on sgd training of neural networks. *arXiv preprint arXiv:1508.02788*, 2015.

[3] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[4] Karishma Dasgaonkar and Swati Chopade. Analysis of multi-layered perceptron, radial basis function and convolutional neural networks in recognizing handwritten digits. *International Journal of Advance Research, Ideas and Innovations in Technology*, 4(3):2429–2431, 2018.

[5] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural computation*, 9(1):1–42, 1997.

[7] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in neural information processing systems*, 30, 2017.

[8] Stanisław Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[9] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

[12] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[13] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[14] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, ON, Canada, 2013.

[15] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[16] Ange Tato and Roger Nkambou. Improving adam optimizer. 2018.

[17] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I Jordan. How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878*, 2019.