VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

FINAL PROJECT OF INTRODUCTION TO MACHINE LEARNING

Supervisor: **Associate Professor PhD Le Anh Cuong**

Authors: **Nguyen Duy Tuan – 520K0232**

HO CHI MINH CITY, 2023

ACKNOWLEDGEMENTS

I would like to express my gratitude and thanks to my teacher Mr. Le Anh Cuong for his wonderful guidance throughout this project.

I am also grateful to my parents for their continuous support to me throughout this experiment, to my friends for their help, and to all those who contributed directly or indirectly towards the completion of this school project

During this project, I acquired many valuable skills, and I hope that in the years to come, those skills will be put to good use.

Table of Contents

1.Optimizer

What is Optimizer ?

- Optimizers are algorithms that adjust the model’s parameters during training to minimize a loss function. They enable neural networks to learn from data by iteratively updating weights and biases. Common optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop. Each optimizer has specific update rules, learning rates, and momentum to find optimal model parameters for improved performance.
- Optimizer algorithms are optimization method that helps improve a deep learning model’s performance. These optimization algorithms or optimizers widely affect the accuracy and speed training of the deep learning model. But first of all, the question arises of what an optimizer is.

During the training of deep learning models, each iteration tweaks the model's weights and reduces the loss function. An optimizer, either a function or an algorithm, adjusts elements like weights and learning rates in a neural network, aiding in minimizing overall loss and enhancing accuracy. Selecting the right weights for a model can be challenging given the vast number of parameters in a deep learning model. Thus, the choice of an appropriate optimization algorithm becomes crucial for your specific application. Understanding these machine learning algorithms is essential for data scientists stepping into this domain.

Multiple optimizers are available in machine learning to adjust weights and learning rates within a model. However, the selection of the best optimizer hinges on the application at hand. For beginners, the idea of trying out various options and picking the one with the best outcomes might seem plausible. Yet, when dealing with substantial data sizes, even one training cycle can consume significant time. Opting for an algorithm randomly becomes akin to gambling with valuable time, a realization that often dawns on individuals sooner or later in their journey.

This comprehensive guide delves into diverse deep-learning optimizers like Gradient Descent, Stochastic Gradient Descent, Stochastic Gradient Descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta, and Adam. By the end of this article, you'll gain insights into comparing different optimizers and their underlying procedures.

Before proceeding, there are a few terms that you should be familiar with.

- **Epoch:** The number of times the algorithm runs on the whole training dataset. - Sample: A single row of a dataset.
- **Batch:** It denotes the number of samples to be taken to for updating the model parameters.
- **Learning rate:** It is a parameter that provides the model a scale of how much model weights should be updated.
- **Cost Function/Loss Function:** A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.
- **Weights/ Bias:** The learnable parameters in a model that controls the signal between two neurons.

1.1 Gradient Descent (GD)

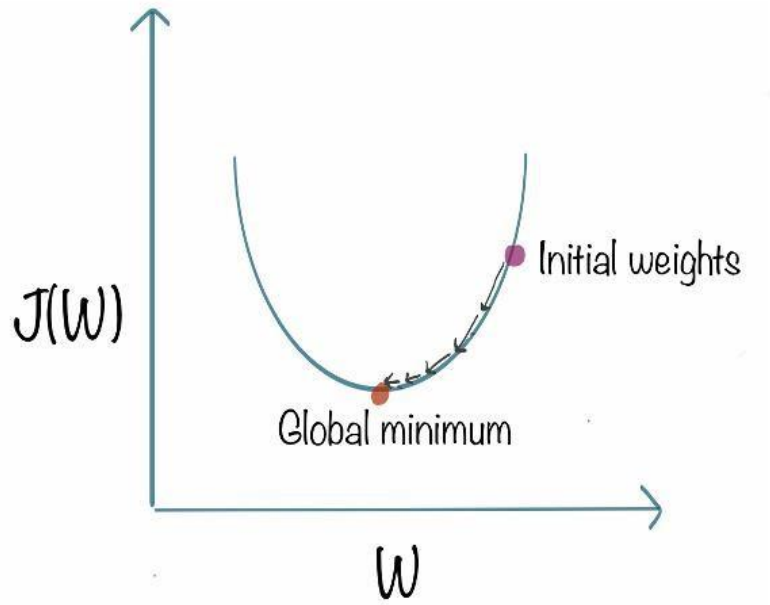
The fundamental optimization algorithm used to minimize loss by iteratively moving in the direction of steepest descent. It can be slow for large datasets due to computing gradients for the entire dataset in each iteration.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t)$$

Where:

- θ_t is the parameter at time t
- η is the learning rate
- $\nabla J(\theta_t)$ is the gradient of the loss function J at θ_t

- Gradient descent works as follows:
    - It starts with some coefficients, sees their cost, and searches for cost value lesser than what it is now.
    - It moves towards the lower weight and updates the value of the coefficients.
    - The process repeats until the local minimum is reached. A local minimum is a point beyond which it can not proceed



- Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions, but it doesn't know how far to travel along the gradient for nonconvex function

1.2 Stochastic Gradient Descent (SGD):

Instead of using the entire dataset for each iteration, SGD computes gradients using a single random data point or a small batch of data. It converges faster but can have noisy updates.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t; x_i, y_i)$$

Where:

- x_i and y_i are a single data point and its corresponding label
- Other variables have the same meaning as in GD

Since we are not using the whole dataset but the batches of it for each iteration, the path taken by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

1.3 Mini-batch Gradient Descent:

In this variation of gradient descent, instead of using the entire training dataset, only a portion of the data is employed to compute the loss function. By utilizing a batch of data rather than the entire dataset, this approach requires fewer iterations. Consequently, the mini-batch gradient descent algorithm outperforms both stochastic gradient descent and batch gradient descent in terms of speed. It stands as a more efficient and resilient alternative to earlier gradient descent variations. As it involves batching, there's no necessity to load all training data into memory, enhancing implementation efficiency. Moreover, while the cost function in mini-batch gradient descent is noisier compared to batch gradient descent, it's smoother than stochastic gradient descent. This characteristic positions mini-batch gradient descent as an ideal choice, striking a balance between speed and accuracy.

However, despite its advantages, the mini-batch gradient descent algorithm has drawbacks. It relies on a hyperparameter called "mini-batch-size," which necessitates fine-tuning to achieve the desired accuracy, although a batch size of 32 is generally considered suitable for most scenarios. Additionally, there are instances where it may lead to subpar final accuracy, prompting the exploration of alternative approaches.

1.4 Momentum:

As discussed in the earlier section, you have learned that stochastic gradient descent takes a much more noisy path than the gradient descent algorithm. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum, and hence computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. However, adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.

$$v_{t+1} = \beta \cdot v_t + (1 - \beta) \cdot \nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t - \eta \cdot v_{t+1}$$

Where:

- v_t is the velocity or momentum at time t
- β is the momentum parameter

1.5 Adagrad

The adaptive gradient descent algorithm diverges from conventional gradient descent methods by employing varying learning rates across iterations. These rate adjustments depend on parameter fluctuations during training: larger parameter changes result in smaller learning rate adjustments. This adaptation proves advantageous as real-world datasets comprise both sparse and dense features, making it unfair to assign identical learning rates to all features. The Adagrad algorithm updates weights using the formula below, where alpha(t) represents distinct learning rates per iteration, n stands as a constant, and E is a small positive value used to prevent division by zero.

$$G_{t+1} = G_t + (\nabla J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_{t+1}+\epsilon}} \cdot \nabla J(\theta_t)$$

Where:

- G_t is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients up to time t for parameter i
- ϵ is a small constant added for numerical stability

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of the AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

1.6 RMS Prop (Root Mean Square):

Resolves Adagrad's diminishing learning rates by using a moving average of squared gradients. It divides the learning rate by an exponentially decaying average of squared gradients.

$$E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1-\beta) \cdot (\nabla J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot \nabla J(\theta_t)$$

Where:

- $E[g^2]_t$ is the exponentially decaying average of squared gradients
- β is the decay rate
- ϵ is a small constant for numerical stability

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn't been published but is still very well-known in the community. RMS prop is ideally an extension of the work RPPROP. It resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the gradient sign, adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction, increasing the step size by a small fraction. If they have opposite signs, we must decrease the step size. Then we limit the step size and can now go for the weight update.

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches simultaneously was the main motivation behind the rise of RMS prop. RMS prop is an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

1.7 AdaDelta:

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches simultaneously was the main motivation behind the rise of RMS prop. RMS prop is an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2.$$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}'_t.$$

$$\mathbf{g}'_t = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

$$\Delta\mathbf{x}_t = \rho\Delta\mathbf{x}_{t-1} + (1-\rho)\mathbf{g}'^2_t,$$

Here St and delta Xt denote the state variables, g't denotes rescaled gradient, delta Xt-1 denotes squares rescaled gradients, and epsilon represents a small positive integer to handle division by 0.

1.8 Adam (Adaptive Moment Estimation):

The Adam optimizer, short for Adaptive Moment Estimation, stands as a widely used optimization technique in deep learning. It builds upon the stochastic gradient descent (SGD) algorithm and is tailored for updating a neural network's weights throughout the training process.

Named after its adaptive moment estimation, Adam excels in dynamically fine-tuning the learning rate for each network weight separately. Unlike SGD, which maintains a uniform learning rate throughout training, Adam computes distinct learning rates by considering past gradients and their second moments.

Adam's creators amalgamated beneficial traits from other optimization methods like AdaGrad and RMSProp. Similar to RMSProp, Adam delves into the second moment of gradients, but unlike RMSProp, it calculates uncentered gradient variance, bypassing the mean subtraction.

By integrating both the first moment (mean) and second moment (uncentered variance) of gradients, Adam optimizer achieves an adaptive learning rate, navigating the optimization landscape effectively during training. This adaptability accelerates convergence and enhances the neural network's overall performance.

In essence, the Adam optimizer extends SGD by dynamically adjusting learning rates based on individual weights. By merging AdaGrad and RMSProp features, it efficiently updates network weights, fostering adaptiveness and efficiency during deep learning training.

$$m_{t+1} = \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla J(\theta_t)$$
$$v_{t+1} = \beta_2 \cdot v_t + (1 - \beta_2) \cdot (\nabla J(\theta_t))^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}+\epsilon}} \cdot \hat{m}_{t+1}$$

Where:

- m_t and v_t are moment estimates of the first moment (mean) and second moment (uncentered variance) of the gradients, respectively
- β_1 and β_2 are decay rates
- $\hat{m}_{t+1}$ and $\hat{v}_{t+1}$ are bias-corrected estimates
- ϵ is a small constant for numerical stability

The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward to implement, has a faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

If the adam optimizer uses the good properties of all the algorithms and is the best available optimizer, then why shouldn't you use Adam in every application? And what was the need to learn about other algorithms in depth? This is because even Adam has some downsides. It tends to focus on faster computation time, whereas algorithms like stochastic gradient descent focus on data points. That's why algorithms like SGD generalize the data in a better manner at the cost of low computation speed. So, the optimization algorithms can be picked accordingly depending on the requirements and the type of data.

2. Exploring "Continual Learning and Test Production" in machine learning.

Continual learning is the ability of a machine learning model to learn from new data and adapt to new tasks without forgetting previous knowledge. This is important because the data and the environment that the model operates in may change over time, and the model needs to keep up with these changes. Continual learning is largely an infrastructural problem, as it requires a robust system that allows frequent model updates and monitoring. There are four main stages to make continual learning a reality:

- Data collection: The system needs to collect and store the new data that the model encounters in production, as well as the feedback from the users or the environment.
- Model training: The system needs to retrain the model using the new data, either incrementally or from scratch, depending on the task and the data availability.
- Model evaluation: The system needs to evaluate the updated model on a validation set or a test set to ensure that it meets the performance and quality standards.
- Model deployment: The system needs to deploy the updated model to production, replacing or complementing the previous model.

Disadvantages of continual learning:

- It can introduce complexity and overhead to the machine learning pipeline and system architecture. For example, it may require more data storage, computation, and network resources, as well as more sophisticated tools and frameworks to manage the model lifecycle.
- It can require more human intervention and supervision. For example, it may need more frequent data validation, model evaluation, and error handling, as well as

more careful design of experiments and metrics to measure the impact of model changes.

- It can pose ethical and legal issues, such as privacy, fairness, accountability, and transparency . For example, it may involve sensitive or personal data that needs to be protected and anonymized, or it may affect the decisions and outcomes of the users or the environment that need to be explained and justified.

Test production is the process of evaluating the performance and reliability of a machine learning model on live data in production. This is important because the model may behave differently in production than in the development or testing environment, due to data distribution shifts, concept drift, or adversarial attacks. Test production is a way to test the model with real-world scenarios and measure the impact of model changes on the business outcomes. There are three main methods to perform test production:

- A/B testing: The system splits the traffic between the old model and the new model, and compares their performance on various metrics, such as accuracy, latency, revenue, etc.
- Canary testing: The system exposes the new model to a small percentage of the traffic, and monitors its performance and feedback, before rolling it out to the rest of the traffic.
- Shadow testing: The system runs the new model in parallel with the old model, but only serves the results from the old model, and logs the results from the new model for analysis.

Disadvantages of test production:

- It can introduce security risks and expose vulnerabilities in the system. For example, hackers can use any bugs or errors they find during testing to access or manipulate sensitive data or cause system failures.
- It can cause data loss or corruption, which can have serious consequences for the organization. For example, testing data can be mixed with production data, or production data can be overwritten or deleted by mistake.
- It can increase the complexity and overhead of the machine learning pipeline and architecture. For example, it may require more data storage, computation, and network resources, as well as more sophisticated tools and frameworks to manage the model lifecycle.
- It can require more human intervention and supervision. For example, it may need more frequent data validation, model evaluation, and error handling, as well as

more careful design of experiments and metrics to measure the impact of model changes.