Bridging the Gap: The Interplay of Deep Learning and Optimization Techniques

1st Mahdi Salahshour

2nd Mohammad Navid Fazly

***Abstract*—Deep Learning [1]develops rapidly, which has made many experimental breakthroughs and is widely applied in various fields. Optimization, as an important part of Deep Learning [2], has attracted much attention of researchers. With the exponential growth of data amount and the increase of model complexity, optimization methods in Deep Learning face more and more challenges. A lot of work on solving optimization problems or improving optimization methods in Deep Learning has been proposed successively. The systematic retrospect and summary of the optimization methods from the perspective of Deep Learning are of great significance, which can offer guidance for both developments of optimization and Deep Learning research. In this paper, we first introduce the principles and progresses of commonly used optimization methods in Deep Learning. Next, we compare the results of described methods in two different major Deep Learning problems.**

***Index Terms*—Deep Learning, Optimization**

I. Introduction

Deep Learning is a sub-field of machine learning that uses algorithms to process data and imitate the thinking process. It uses layers of algorithms, where information is passed through each layer, with the output of the previous layer providing input for the next layer.

A neural network is a network of neurons/nodes connected by a set of weights. It is loosely inspired by the way biological neural networks in the human brain process information.

Mathematically, a simple neural network can be represented as follows:

Consider a single-layer neural network with n inputs $(x_1, x_2, ..., x_n)$ and one output y. The output of this network is given by:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where:

- w_i are the weights,
- b is the bias,
- f is the activation function.

The activation function introduces non-linearity into the output of a neuron. This non-linearity helps neural networks learn from the error they make, i.e., update their weights after each iteration during the training phase.

The history of Deep Learning can be traced back to when Walter Pitts and Warren McCulloch created a computer model based on the neural networks of the human brain. [3] The first serious Deep Learning breakthrough came in the mid-1960s, when Soviet mathematician Alexey Ivakhnenko created small but functional neural networks. In the early 1980s, John Hopfield's recurrent neural networks made a splash, followed by Terry Sejnowski's program NetTalk that could pronounce English words. In 1986, Geoffrey Hinton demonstrated that more than just a few neural networks could be trained using back-propagation for improved shape recognition and word prediction. [4]

The intersection of Deep Learning and optimization is a fascinating area. Conventional approaches for designing decision algorithms employ principled and simplified modeling, based on which one can determine decisions via tractable optimization. More recently, Deep Learning approaches that use highly parametric architectures tuned from data without relying on mathematical models, are becoming increasingly popular.

Optimization in Deep learning involves adjusting 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 Gradient Descent (GD) [5], Stochastic Gradient Descent (SGD) [6], Mini Batch Gradient Descent (MGD) [7], Momentum, Nesterov accelerated gradient (NAG) [8], AdaGrad [9], Adam [10], AMSGrad [11] and RMSprop [12]. Each optimizer has specific update rules, learning rates, and momentum to find optimal model parameters for improved performance.

II. Problem Statement

In the field of Deep Learning, there are numerous problems, each with its own unique setup. One of the most critical decisions to make in these setups is choosing the right optimization method. Optimization methods in Deep Learning have different characteristics and their performance can vary depending on the problem at hand.

The choice of optimization method can significantly impact the performance of the Deep Learning model. Therefore, it's crucial to understand the behavior of these algorithms during training and their results on different datasets.

However, it's important to note that the performance of these optimization methods can be sensitive to the hyperparameter tuning protocol. Therefore, careful consideration should be given to the selection and tuning of these parameters.

In this article, we compare two major problems MNIST classification [13] and Link Predition in Graph [14] with different setups and applications in Deep Learning. We also demonstrate the performance of each optimization method in these scenarios.

III. Methods

A. Gradient Descent Algorithm

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent.

Suppose we have a cost function $J(\theta)$, which we want to minimize. Here, θ represents the parameters of the function. The gradient descent algorithm updates the parameters iteratively as follows:

$$\theta = \theta - \alpha \nabla J(\theta) \tag{1}$$

In this equation, α is the learning rate, which determines the size of steps that we take descending on the path of steepest descent, and $\nabla J(\theta)$ is the gradient of the cost function. The minus sign refers to the minimization part of gradient descent. The gradient points in the direction of the greatest rate of increase of the function, and the negative of the gradient points in the direction of steepest descent.

The learning rate α is a hyperparameter which controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we will be taking a long time to converge — especially if we get stuck on a plateau region.

B. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a popular optimization algorithm used in various machine learning models. It is particularly useful when dealing with large-scale data.

The general update equation for SGD in the context of a cost function $J(\theta)$ with parameters θ is given by:

$$\theta = \theta - \eta \nabla J(\theta; x^{(i)}, y^{(i)}) \tag{2}$$

where:

- θ represents the parameters of the model.
- η is the learning rate, a hyperparameter that determines the step size at each iteration while moving toward a minimum of a loss function.
- $\nabla J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the cost function evaluated at the instance $x^{(i)}$ and its target value $y^{(i)}$.

Unlike Batch Gradient Descent that computes the gradient using the whole dataset, SGD randomly picks one instance in the training set at every step and computes the gradients based only on that single instance. This makes the algorithm much faster as it has very little data to manipulate at every iteration. It also enables SGD to train on large datasets, as only one instance needs to reside in memory at each iteration.

SGD's stochastic (random) nature leads to a much more irregular cost function path towards the minimum, but it can also help the algorithm jump out of local minima. Therefore, on balance SGD can reach a good solution faster than Batch Gradient Descent.

One challenge with SGD is choosing a suitable learning rate. If the learning rate is too small, the algorithm will have to go through many iterations to converge, which will take a long time. On the other hand, if the learning rate is too high, SGD might jump across the valley and end up on the other side, possibly even higher up than before. This could make the algorithm diverge, with larger and larger values, failing to find a good solution.

C. Mini-Batch Gradient Descent

Mini-Batch Gradient Descent (MBGD) is a variation of the Gradient Descent algorithm that splits the training dataset into small batches. This algorithm combines the advantages of both Stochastic Gradient Descent and Batch Gradient Descent.

Given a cost function $J(\theta)$, where θ represents the parameters of the model, the gradient of the cost function at a certain point θ is given by:

$$\nabla J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla h_\theta(x^{(i)}) - y^{(i)} \tag{3}$$

In MBGD, instead of using the whole dataset (as in Batch Gradient Descent) or single instances (as in Stochastic Gradient Descent), we use a mini-batch of n instances chosen at random at each step. The update rule for the parameters becomes:

$$\theta = \theta - \eta \nabla J_{i:i+n}(\theta) \tag{4}$$

where η is the learning rate, $J_{i:i+n}$ is the cost function computed over the n instances in the mini-batch, and $\nabla J_{i:i+n}(\theta)$ is the gradient of the cost function computed over the n instances in the mini-batch.

MBGD converges faster than Batch Gradient Descent as it uses more instances at each iteration. It also introduces randomness in the parameter updates, which can help escape local minima.

D. Momentum

The Momentum optimization algorithm is a method that helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and has proven to be effective in deep learning contexts.

The momentum term γ is usually set to 0.9. Unlike Batch Gradient Descent, it tends to roll further, but it can also go past the minimum, hence the name 'momentum'. The update rule is as follows:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \quad (5)$$

$$\theta = \theta - v_t \quad (6)$$

In the equations above, v_t is the velocity at time step t, γ is the momentum term, η is the learning rate, $\nabla_\theta J(\theta)$ is the gradient of the cost function J with respect to the parameters θ, and θ represents the parameters.

The momentum algorithm accumulates the gradient of the past steps to determine the direction to go. This results in faster convergence and reduced oscillation. However, it may overshoot the optimal solution due to the accumulated momentum.

E. Nesterov's Accelerated Gradient

Nesterov's Accelerated Gradient (NAG) is an optimization algorithm that can speed up the convergence of gradient-based methods. It was proposed by Yurii Nesterov in 1983 and has since been widely used in the field of machine learning and deep learning.

The key idea behind NAG is to use a "lookahead" gradient rather than the current gradient at each iteration. This lookahead gradient is computed at the future approximate position of the parameters, which can be estimated using the current momentum.

The update rule for NAG is given by:

$$\begin{aligned} v_{t+1} &= \mu v_t - \eta \nabla f(x_t + \mu v_t) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned}$$

where v_t is the velocity (or momentum) at time t, μ is the momentum factor, η is the learning rate, and $\nabla f(x_t + \mu v_t)$ is the gradient of the function f at the lookahead position $x_t + \mu v_t$.

The lookahead gradient $\nabla f(x_t + \mu v_t)$ allows NAG to make more informed updates, which can lead to faster convergence and less oscillation compared to standard gradient descent.

F. AdaGrad: Adaptive Gradient Algorithm

AdaGrad, short for Adaptive Gradient Algorithm, is an optimization algorithm that was proposed by Duchi et al. in 2011. The key idea behind AdaGrad is to adapt the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

The update rule of AdaGrad is defined as follows:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \quad (7)$$

where:

- $\theta_{t,i}$ is the i-th parameter at time step t,
- η is the global learning rate,
- $G_{t,ii}$ is the sum of the squares of the past gradients w.r.t. $\theta_{t,i}$ up to time step t,
- ϵ is a smoothing term that avoids division by zero (usually on the order of $1e - 8$), and
- $g_{t,i}$ is the gradient at time step t.

AdaGrad has the advantage of eliminating the need to manually tune the learning rate. It is well-suited for dealing with sparse data. However, its main weakness is that its learning rate tends to be monotonically decreasing to the point of becoming infinitesimally small.

G. Adam: Adaptive Moment Estimation

The Adam (Adaptive Moment Estimation) optimization algorithm, proposed by Kingma and Ba in 2015, is an extension to stochastic gradient descent. It computes adaptive learning rates for different parameters.

The algorithm uses running averages of both the gradients and the second moments of the gradients. The name "Adam" is derived from "adaptive moment estimation".

The update rules for Adam are as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{m_t} &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v_t} &= \frac{v_t}{1 - \beta_2^t} \\ \theta_{t+1} &= \theta_t - \frac{\alpha \hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \end{aligned}$$

where:

- m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively.
- $\hat{m_t}$ and $\hat{v_t}$ are bias-corrected estimates of the first and second moment.
- θ_t is the parameter vector at time step t.
- g_t is the gradient at time step t.
- α is the step size.
- β_1 and β_2 are exponential decay rates for the moment estimates. - ϵ is a small scalar to prevent division by zero.

Adam has been shown to work well in practice and compares favorably to other adaptive learning-method algorithms.

H. AMSGrad

AMSGrad is an optimization algorithm used in deep learning, introduced by Reddi et al. in 2018. It is an extension of the Adam optimizer that addresses its convergence issues.

The update rule for AMSGrad is given by:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{v_t} &= \max(\hat{v_{t-1}}, v_t) \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} m_t \end{aligned}$$

where:

- m_t and v_t are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradients respectively.
- $\hat{v_t}$ is the maximum of all v_t up to the current time step.
- θ_t represents the parameters of the model at time step t.
- g_t represents the gradient at time step t.
- β_1, β_2, η, and ϵ are hyperparameters of the algorithm.

AMSGrad resolves the issues of the lack of convergence in Adam optimizer by using a maximum of past squared gradients $\hat{v_t}$ rather than the exponentially decaying average. This ensures that the learning rate does not increase during training.

I. RMSProp: Root Mean Square Propagation

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm that was proposed by Geoff Hinton in his Coursera course. The central idea behind RMSProp is to modulate the learning rate of each weight based on the magnitudes of its gradients.

The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients. Here, we introduce the gradient accumulation variable v_t, which is the exponential moving average of the squared gradient, but we use the gradient on the current mini-batch t.

The RMSProp update rule is as follows:

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2 \tag{8}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} g_t \tag{9}$$

where:

- v_t is the gradient accumulation variable at time step t.
- β is the decay rate. This is a hyperparameter and typical values are [0.9, 0.99, 0.999].
- g_t is the gradient at time step t.
- η is the learning rate.
- ϵ is a smoothing term that avoids division by zero, usually set somewhere in the range from 1e-4 to 1e-8.

As we can see, RMSProp introduces the decay factor to decrease the influence of the historical squared gradient which makes the network learn faster compared to methods that do not normalize the learning rate like SGD.

IV. Experiment

A. MNIST

In this section, we present our experiment on the **MNIST** dataset using a custom-built neural network.

- **Dataset:** The **MNIST** (Modified National Institute of Standards and Technology) dataset is a large database of handwritten digits commonly used for training various image processing systems. The dataset contains 60,000 training images and 10,000 testing images, each of which is a 28x28 grayscale image, representing a digit between 0 and 9.
- **Neural Network Architecture:** Our neural network is designed with an input layer, hidden layers, and an output layer. The input layer has 784 neurons, corresponding to the 28x28 pixels of the input images. The hidden layers are designed with 500 number of neurons and Linear activation function to optimize the learning process. The output layer consists of 10 neurons, each representing a digit from 0 to 9, with the highest activation indicating the predicted digit.
- **Training:** The network was trained using the backpropagation algorithm. We used learning rate of 0.01 , batch size of 64 for Mini Batch GD, and 10 number of epochs to train the model. The performance of the model was evaluated using the accuracy metric on the test set.
- **Results:** Our experiment yielded different results based on different optimization methods. Adagrad, AMSGrad, and Adam are known to perform better than Momentum, Nesterov Accelerated Gradient (NAG), and RMSProp due to several reasons. They adjust the learning rate adaptively for each parameter. This means they can handle sparse data and are less sensitive to the initial learning rate. In contrast, Momentum, NAG, and RMSProp use a fixed learning rate throughout the training process.Adam and AMSGrad, in particular, combine the advantages of Adagrad's adaptive learning rate and RMSProp's noise reduction capability, leading to more stable and efficient convergence. SGD and Mini-Batch Gradient Descent are both variants of the Gradient Descent algorithm, but they handle data differently.SGD computes the gradient using a single training example and updates the parameters for each example. This makes SGD faster as it doesn't need to run through the entire dataset before making an update. However, this can also lead to noisy gradients and oscillations in the loss function.Mini-Batch Gradient Descent strikes a balance between SGD and Batch Gradient Descent. It splits the dataset into small batches and updates the parameters for each batch. This reduces the noise in parameter updates, leading to more stable convergence. Gradient Descent can be slow in training a Neural Network on the MNIST dataset due to several reasons.The MNIST dataset has 60,000 training examples. Gradient Descent considers all these examples for each update, which can be computationally expensive and slow. Neural Networks have a large number of parameters. The optimization of these parameters using Gradient Descent can be slow. For our result check "Fig. 1" and "Fig. 2".

B. Link Prediction - Karate Club Graph [15]

In this experiment, we aim to train a Graph Neural Network (GNN) on the Karate Club dataset. The dataset represents a social network of a university karate club, with nodes representing club members and edges representing friendships between them.

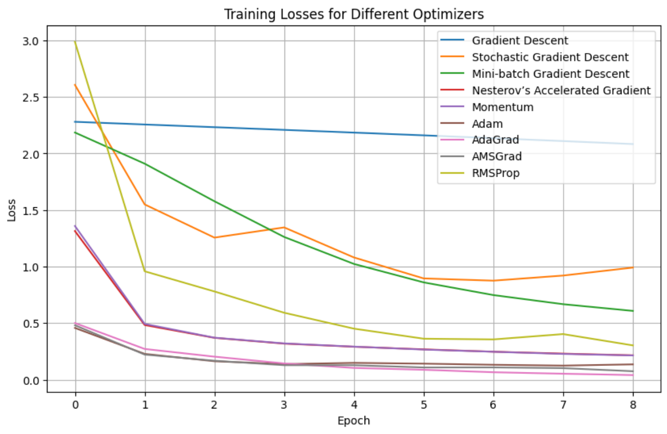

Fig. 1. Training loss in MNIST

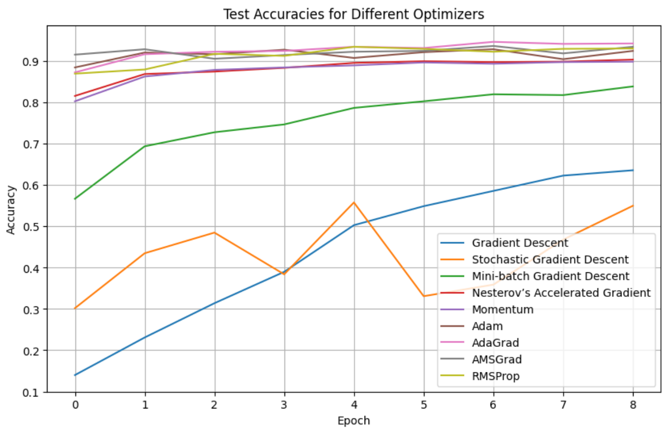

Fig. 2. Test Accuracies in MNIST

- **Dataset: The Karate Club dataset:** The Karate Club dataset is a well-known benchmark in network analysis, consisting of 34 nodes (club members) and 78 edges (friendships). We split the edges into positive examples (existing friendships) and negative examples (non-existent friendships) to create a binary classification task.
- **Neural Network Architecture:** We designed a simple GNN architecture for this task. The model consists of an embedding layer followed by two 1-dimensional convolutional layers and a fully connected layer. The embedding layer maps node indices to a low-dimensional space, while convolutional layers capture local structural information. The global pooling layer aggregates information from all nodes, and the fully connected layer produces the final classification output.
- **Training:** We trained the GNN using various optimization algorithms, including Gradient Descent variants (SGD, Mini-batch SGD, Nesterov's Accelerated Gradient, and Momentum), Adam variants (Adam, AdaGrad, and AMSGrad), and RMSProp. We utilized binary cross-entropy loss and evaluated the model's performance using accuracy metrics on both training and testing sets.
- **Results:** In link prediction tasks, optimization algorithms like Adam, AdaGrad, AMSGrad, RMSProp, SGD with minibatch, and minibatch training are preferred over Nesterov, Momentum, and vanilla Gradient Descent due to their adaptive learning rates, which dynamically adjust based on past gradients, facilitating efficient updates in scenarios of data sparsity common in such tasks. Their faster convergence, enabled by dynamic learning rate adjustments, is crucial for large network datasets, while minibatch training strikes a balance between stochastic and batch updates, enhancing stability and acceleration. Moreover, these algorithms mitigate the risk of overshooting and oscillation, ensuring stable optimization even in complex, nonlinear landscapes. Adam, with its adaptive moment estimation, combines adaptive learning rates with momentum, further enhancing efficiency and stability, making it particularly well-suited for the intricacies of link prediction tasks. "Fig. 3" and "Fig. 4".

V. Conclusion

In image classification tasks like MNIST, optimization algorithms such as Adagrad, AMSGrad, and Adam excel due to their adaptive learning rate mechanisms. Images in MNIST are typically sparse, with many pixels being blank. Algorithms like Adagrad and Adam dynamically adjust learning rates for each parameter, effectively handling this sparsity and reducing sensitivity to initial learning rates. This adaptability enables more stable and efficient convergence, resulting in superior performance compared to algorithms with fixed learning rates like Momentum and RMSProp.

In graph neural network (GNN) tasks like link prediction, the preference for optimization algorithms like Adam, AdaGrad, and AMSGrad stems from similar reasons but within the context of graph data. Graphs are inherently sparse structures, especially in large network datasets where only a fraction of all possible connections exist. Adaptive learning rates offered by these algorithms dynamically adjust based on past gradients, facilitating efficient updates in scenarios of data sparsity common in GNN tasks. Additionally, minibatch training strikes a balance between stochastic and batch updates, enhancing stability and acceleration, which is crucial for large network datasets. These adaptive mechanisms mitigate the risk of overshooting and oscillation, ensuring stable optimization even in complex, nonlinear landscapes inherent to graph data.

Overall, the adaptive nature of these optimization algorithms makes them well-suited for handling the inherent sparsity and complexity present in both image classification and GNN tasks, leading to superior performance and convergence.

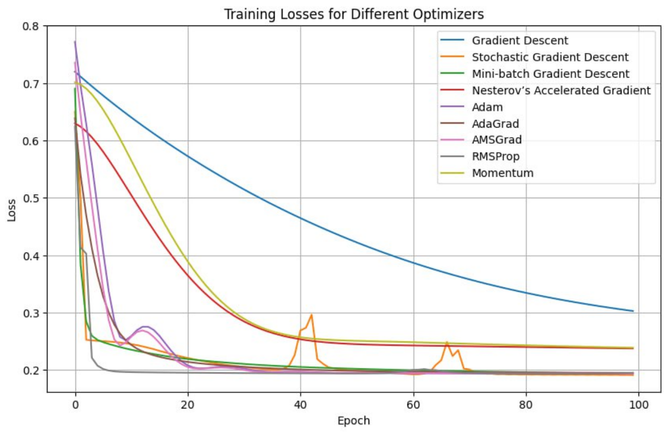

Fig. 3. Training loss in Link Prediction - Karate Club Graph

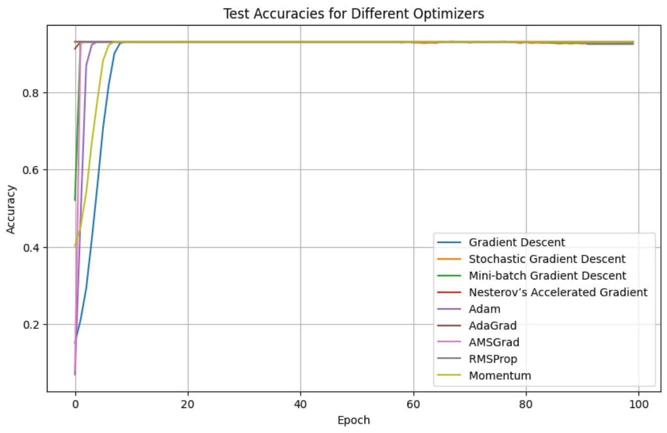

Fig. 4. Test Accuracies in Link Prediction - Karate Club Graph

References

[1] Yann LeCun, Yoshua Bengio ,Geoffrey Hinton , "Deep learning"
[2] Haohan Wang, Bhiksha Raj , "On the Origin of Deep Learning"
[3] Keith D. Foote A Brief History of Deep Learning https://www.dataversity.net/brief-history-deep-learning/, 2022
[4] Mike Thomas The History of Deep Learning: Top Moments That Shaped the Technology https://builtin.com/artificial-intelligence/deep-learning-history, 2022
[5] "An overview of gradient descent optimization algorithms",Sebastian Ruder,ArXiv,2016
[6] Boyd, Stephen; Vandenberghe, Lieven (2004-03-08). "Convex Optimization". Cambridge University Press
[7] Sarit Khirirat , "Mini-batch gradient descent: Faster convergence under data sparsity"
[8] Aleksandar Botev, Guy Lever, David Barber , "Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent"
[9] John Duchi ,Elad Hazan ,Yoram Singer "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"
[10] Diederik P. Kingma, Jimmy Ba ,"Adam: A Method for Stochastic Optimization"
[11] Sashank J. Reddi, Satyen Kale, Sanjiv Kumar , "On the Convergence of Adam and Beyond"
[12] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, Kevin Murphy"Progressive Neural Architecture Search"
[13] "THE MNIST DATABASE of handwritten digits". Yann LeCun, Courant Institute, NYU Corinna Cortes, Google Labs, New York Christopher J.C. Burges, Microsoft Research, Redmond.
[14] Muhan Zhang, Yixin Chen, "Link Prediction Based on Graph Neural Networks"
[15] Wayne W. Zachary, "An Information Flow Model for Conflict and Fission in Small Groups"