An overview of Optimization Algorithms

Reya Sadhu, A59026753

Gradient descent optimization algorithms, though having gained popularity, are frequently employed as black-box optimizers due to the scarcity of practical explanations about their strengths and weaknesses. This report aims to offer readers insights into the behavior of various algorithms, enabling them to effectively apply these techniques. Throughout this overview, we explore diverse variants of gradient descent and provide a synopsis of the challenges involved. Different references have been taken from following papers (1), (2),(3),(4).

Goal:

Our final goal in machine learning is to predict something with the help of some data and modify our predictions based on their performance. So, we have these steps,

- Make an initial hypothesis about the parameters
- Predict for each data point using these parameters.
- Quantify the performance by measuring the cost function which represents the discrepancy between the predictions and the original values.
- Find the parameters that minimizes the discrepancy and improve the model performance.

What is Gradient Descent?

Gradient descent is an optimizing algorithm that iteratively adjusts the parameters in the direction of negative gradient, aiming to find the optimal set.

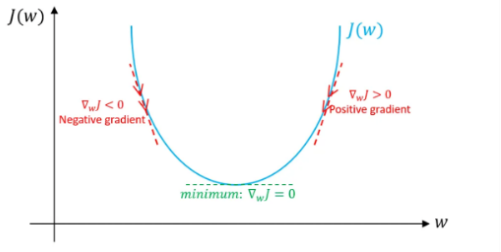


Fig. 1. Gradient Descent on Cost Function

At each step, we compute the gradient, which gives the direction of the steepest ascent. If we move towards the opposite of that direction; we will subsequently find the minima.

$$w_t = w_{t-1} - \eta \frac{\partial J}{\partial w_{t-1}} \tag{1}$$

The learning rate η determines the step size in each iteration, and thus influences the number of iterations to reach the optimal value.

Why Gradient Descent?

Gradient descent relies on knowledge of the gradient to 'slide down the slope', but in school when we want to find the minimum of a function whose gradient can be calculated, we normally don't apply gradient descent. Instead, we just find the minimum analytically by directly solving for when the gradient is 0. So why then, when we move to computers, we suddenly switch tactics and use iterative techniques?
This is because the analytical method consists of a step where we need to calculate the inverse of a matrix which contains all the data points. For a single feature with n data points, we need to inverse a n×n matrix, which runs in $O(n^3)$ runtime complexity. That means this analytical method runs extremely slow when n is large. In that case it's best to use the iterative method, the gradient descent.

Gradient Descent Variants:

We assume that we have J(w) as the cost function, and $h_w(x)$ as the function to be learned, and w is the parameter to be optimized. n is the number of training samples and y is the target output.

$$J(w) = \frac{1}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})^2 \tag{2}$$

Batch Gradient Descent: This computes the gradient of the cost function with respect to the entire dataset. It alternates the following two steps until it converges:

1. Derive the gradient of J(w) for each parameter w

$$\frac{\partial J}{\partial w} = \frac{2}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})x^{(i)} \tag{3}$$

2. Update each w in the negative direction of gradient to minimize the risk function.

$$w_t = w_{t-1} - \eta \frac{\partial J}{\partial w_{t-1}} \tag{4}$$

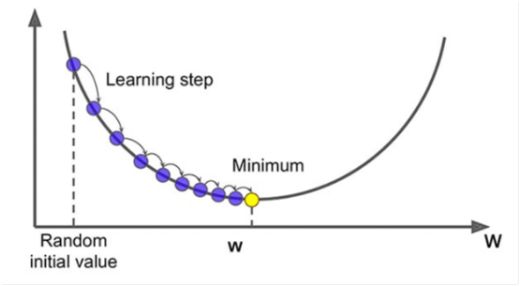


Fig. 2. Converging on minima

Challenges

1. **High computational Complexity:** This algorithm calculates uses the whole training data in each iteration step before updating the parameters. Because we are taking an average of all the data points. So, for large scale data, the cost is too high.

2. **Stuck on local minima:** The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the starting point and the learning rate. Therefore, the optimization may converge to different points with different initialization and learning rate.

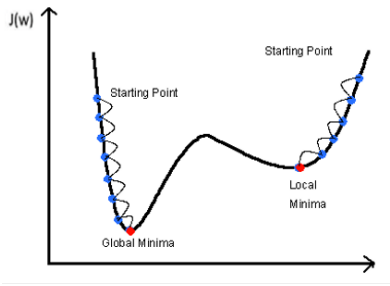


Fig. 3. Local and Global Minima

3. **Saddle Points:** In high-dimensional spaces, there are some points which are local minima for one axis but not for the others. These are called saddle points. The slope of a saddle point is positive in one direction and negative in another direction. These saddle points are surrounded by plateaus, where the gradients are very small or close to zero. So, the algorithm can get stuck in these areas and may never escape due to the zero gradient situation and may never converge.

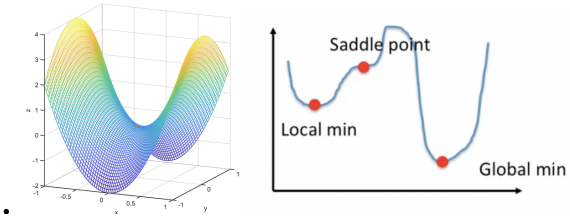


Fig. 4. Saddle points

Stochastic Gradient Descent: It tries to lower the computation per iteration, at the cost of an increased number of iterations necessary for convergence. Instead of computing the sum of all gradients, stochastic gradient descent selects an observation uniformly at random, calculate its gradient and performs a parameter update.

$$J(w) = (h_w(x^{(i)}) - y^{(i)})^2 \tag{5}$$

$$w_t = w_{t-1} - 2\eta(h_w(x^{(i)}) - y^{(i)})x^{(i)} \tag{6}$$

So, for a data with d features and n samples, the computation complexity for each iteration is $O(d)$, whereas for batch GD it was $O(nd)$.
SGD performs frequent updates with just considering one sample. So there's a high variance that cause the objective function to fluctuate heavily. Because of these fluctuations , it is more likely to jump to new and potentially better local minima/global minima. The downside is that, once it reaches close to the minimum value then it doesn't settle down, instead bounces around which gives us a good value for model parameters but not optimal.
So SGD converges much faster compared to Batch, but the error function might not be well minimized. So, a compromise between the two methods, the mini-batch gradient descent method (MSGD), was proposed. The MSGD uses a few(b) independent identically distributed samples as the sample sets to update the parameters in each iteration. It reduces the variance of the gradients and makes the convergence more stable, which helps to improve the optimization speed.

$$J(w) = \frac{1}{b}\sum_{i=1}^{b}(h_w(x^{(i)}) - y^{(i)})^2 \tag{7}$$

$$w_t = w_{t-1} - \eta\frac{2}{b}\sum_{i=1}^{b}(h_w(x^{(i)}) - y^{(i)})x^{(i)} \tag{8}$$

The following image summarizes the performance of difference gradient descent variants clearly. For batch gradients, the gradient will be perpendicular to the contours. For SGD, as we take random sample, the gradient direction won't be exactly perpendicular and takes more number of steps to pass the contours. Where the Mini batch shows less fluctuates and converges faster than SGD. Generally, we refer the mini batch as SGD.

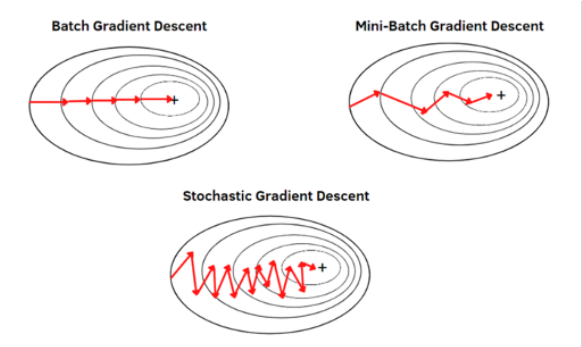


Fig. 5. Fluctuations in GD

Challenges with Gradient Descent:

1. **The Learning Rate Conundrum:** The value of learning rate is very important for this algorithm. If it's too small, it will take a long time to reach the optimal solution and there is also a chance to be trapped into the local minima. But if it's too large it may overshoot the minima and never reach the optimal value. It may even diverge instead of converging. We can use Learning rate schedules like reducing it according to a pre-defined schedule or when the parameter update value is lower than a threshold. However they have to be defined in advance and are thus unable to adapt to a dataset's characteristics.

2. **Sparse Data:** GD uses same learning rate for all the parameters. Now for sparse data where some features occur rarely compared to others, we don't want to update all of them with the same size step. We can use a large update for those features.

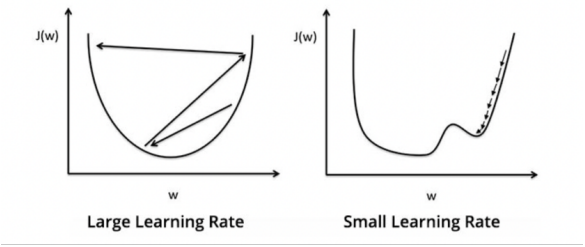


Fig. 6. Overshoot & Slow Convergence

3. **Saddle Points:** It is always possible for these algorithms to stuck in a saddle points or plateau.

4. **Ravines:** : There may be some areas on the cost functions where the surface curves much more steeply in one dimension than on other. The SGD oscillates across the slope of the ravine, and thus very slow progress towards the optimal minima.

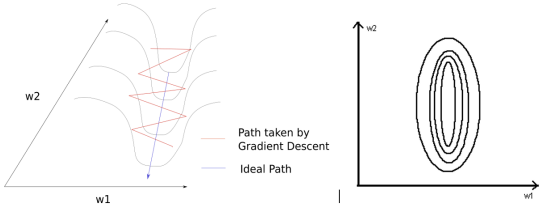


Fig. 7. Ravine

Momentum

Momentum is a method that helps accelerate SGD to the desired direction, skip plateaus and also dampens the oscillation. In physics, momentum is mass multiplied by velocity. So, lets say if a car is coming down a slope, the car is gaining momentum because its velocity increases. The idea of the momentum is, when the function reaches a plateau or saddle point, we get the idea that it is impossible for the car to reach the lowest point so fast since it was having a lot of momentum on the previous step. So, the algorithm keeps track of the past momentum and doesn't stop at the plateau.
Also, when we first start the optimization we want bigger steps, and as we slowly approach the minima, we want small steps. So, based on the previous momentum we can vary how fast or how slow we move across the surface.

$$v_t = \beta v_{t-1} + \eta \frac{\partial J}{\partial w_{t-1}} \tag{9}$$

$$w_t = w_{t-1} - v_t \tag{10}$$

The momentum algorithm introduces the variable v as the speed, which represents the direction and the rate of the parameter's movement in the parameter space. β is the momentum hyperparameter and v_{t-1} keeps track of all the previous slopes. So, by varying β we can change how much weightage we want to give the past gradients and how much for the current gradient. The v_t term increases for dimensions whose gradients point in the same directions and reduces for dimensions whose gradients change directions. So, thus the oscillation get reduced in case of ravines and we get faster convergence.

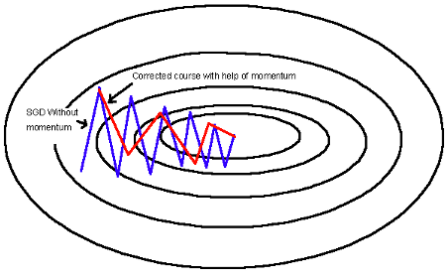


Fig. 8. Less oscillation and fast convergence

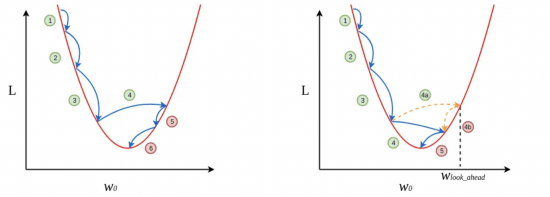
Nesterov Accelerated Gradient

By using momentum, we are considering past gradients. However, if we can also consider future course, we will know when to slow down. In momentum, we use βv_{t-1} to move our parameters w_{t-1}. So computing $w_{t-1} - \beta v_{t-1}$ gives us an approximation of the next position of the parameters.

$$v_t = \beta v_{t-1} + \eta \frac{\partial J}{\partial (w_{t-1} - \beta v_{t-1})} \tag{11}$$

$$w_t = w_{t-1} - v_t \tag{12}$$

The improvement of Nesterov momentum over momentum is reflected in updating the gradient of the future position instead of the current position. This looking ahead help us in avoiding overshoots as we already have some knowledge about the future slope.



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

Fig. 9. No Overshoot in NAG

Now we are able to adapt our update formula to the slope of the contours and thus speed up the SGD. We also want to adapt our update to the individual parameters to perform larger or smaller updates depending on their importance. Also, they need to change as we move closer to the optimal point. If we keep this learning rate fixed, as we reach close to the optimal point, we may see some oscillation. Sometimes we use the the learning rate decay factor d in the SGD's momentum method, which makes the learning rate decrease with the iteration period (1). The formula of the learning rate decay is defined as,

$$\eta_t = \frac{\eta_0}{1 + dt} \tag{13}$$

where η_t is the learning rate at the t-th iteration, η_0 is the original learning rate, and d is a decimal in [0, 1]. But instead of the manually tuned version, we want to use something that can incorporate knowledge of the characteristics of the data and can peform more informative updates.

Adaptive Learning Rate Method

Adagrad Adagrad is a gradient-based optimization algorithm that adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. So, it is well-suited for dealing with sparse data. It also adjusts the learning rate dynamically based on the historical gradient in some previous iterations. Previously, we performed an update for all parameters w at once as every parameter w^i used the same learning rate η. As Adagrad uses a different learning rate for every parameter w^i at every time step t,

$$g_t^i = \frac{\partial J}{\partial w_t^i} \tag{14}$$

$$v_{t-1}^i = \sqrt{\sum_{j=1}^{t-1}(g_j^i)^2 + \varepsilon} \tag{15}$$

$$w_t^i = w_{t-1}^i - \frac{\eta}{v_{t-1}^i} g_{t-1}^i \tag{16}$$

where g_t^i is the gradient of parameter w^i at iteration t, v_{t-1}^i is the accumulate historical gradient of parameter w^i before iteration t, and w_t^i is the value of parameter w^i at iteration t. ε is a small value added to avoid division by zero.
As the training time increases, the accumulated gradient will become larger and larger, making the learning rate tend to zero. So, the parameter update will be too insignificant to acquire any additional knowledge. To handle this fast decreasing learning rate, adadelta and RMSprop were introduced.

AdaDelta Instead of considering all the past gradients, now we restrict the window to a fixed size w. And also the sum of gradients is defined as a recursively decaying average of the past gradients. So the running average $E[(g^i)^2]_t$ depends as a fraction β (like momentum term) of the past running average and the present gradient,

$$E[(g^i)^2]_t = \beta E[(g^i)^2]_{t-1} + (1-\beta)(g_{t-1}^i)^2 \tag{17}$$

$$RMS[g^i]_t = \sqrt{E[(g^i)^2]_t + \varepsilon} \tag{18}$$

$$w_t^i = w_{t-1}^i - \frac{\eta}{RMS[g^i]_t} g_{t-1}^i \tag{19}$$

if we stopped and use equation (18) as the update rule, we have another algorithm called **RMSProp**. However, with this algorithm, we still need to choose the initial learning rate (η) and the unit mismatch still hasn't been solved. Because as we update the parameters, the hypothetical unit of w_t should be equal to Δw_{t-1} as we calculate $w_t = w_{t-1} - \Delta w_{t-1}$

$$\Delta w_{t-1}^i = \frac{\eta}{RMS[g^i]_t} g_{t-1}^i \tag{20}$$

Define,

$$E[(\Delta w^i)^2]_t = \beta E[(\Delta w^i)^2]_{t-1} + (1-\beta)(\Delta w^i)_t^2 \tag{21}$$

$$RMS[\Delta w^i]_t = \sqrt{E[(\Delta w^i)^2]_t + \varepsilon} \tag{22}$$

$RMS[\Delta w^i]_t$ isn't known, we estimate it by $RMS[\Delta w^i]_{t-1}$, and replace η with it, which yields the Adadelta update rule,

$$w_t^i = w_{t-1}^i - \frac{RMS[\Delta w^i]_{t-1}}{RMS[g^i]_t} g_{t-1}^i \tag{23}$$

So, with AdaDelta we remove the need to set for a default learning rate.

Adam Adaptive Moment Estimation (Adam) is another method that combines the adaptive learning rate and momentum methods. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, like momentum:

$$m_t^i = \beta_1 m_{t-1}^i + (1-\beta_1) g_t^i \tag{24}$$

$$v_t^i = \beta_2 v_{t-1}^i + (1-\beta_2)(g_t^i)^2 \tag{25}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients , hence the method is called moment estiamtation. Initially, both mt and vt are set to 0. Both tend to be more biased towards 0 as β_1 and β_2 are equal to 1. This problem is corrected by the Adam optimizer,

$$\hat{m_t^i} = \frac{m_t^i}{1-(\beta_1)^t} \tag{26}$$

$$\hat{v_t^i} = \frac{v_t^i}{1-(\beta_2)^t} \tag{27}$$

So, the update rule becomes,

$$w_t^i = w_{t-1}^i - \frac{\eta}{\sqrt{\hat{v_{t-1}^i}} + \varepsilon} \hat{m_{t-1}^i} \tag{28}$$

As adam combines the benefits of Adaptive Learning rate and momentum, it typically works better than all other optimizers. It's memory efficient and works good for non-convex optimizations.

Which optimizer to use?

SGD is a very basic algorithm and is hardly used in applications now due to its slow computation speed. The problem with that algorithm is its constant learning rate. Moreover, it is not able to handle saddle points very well. Adagrad works better than SGD generally due to frequent updates in the learning rate. It is best when used for dealing with sparse data. RMSProp shows similar results to that of the gradient descent algorithm with momentum, it just differs in the way by which the gradients are calculated. Lastly comes the Adam optimizer, that adds bias-correction and momentum to RMSprop. It inherits the good features of RMSProp and other algorithms. So, the results are generally better than every other optimization algorithm, have faster computation time,less memory requirement and require fewer parameters for tuning. But sometimes, even for a simple convex function, Adam doesn't converge, so does other Adaptive learning methods. So, we need to decide what optimizer to use based on the problem in hand.

Results and Visualization

To compare the performance of different optimizers, A Resnet 50 model has been trained over 21000 images to classify different natural scenes. As it's a pre-trained model, the model has been trained for 7 epochs. I have used early stopping which is another way of optimizing gradient descent where the model is configured to halt training after a specified number of epochs if the loss failed to show improvement over consecutive steps. The best-fitting model from these epochs was then saved for further use. We can see for the same number of epochs, Adam gives the best performance followed by Adadelta and Adagrad. SGD with Nesterov Accelerated Gradient shows a better performance then SGD with momentum and simple SGD. The momentum that has been used is 0.9. Also, if we visualize the time needed for each optimizer to reach these accuracies, We can see though SGD gives the lowest accuracy, it takes the longest time. Adam consistently shows good performance in terms of both time and accuracy. Find the code here:Code

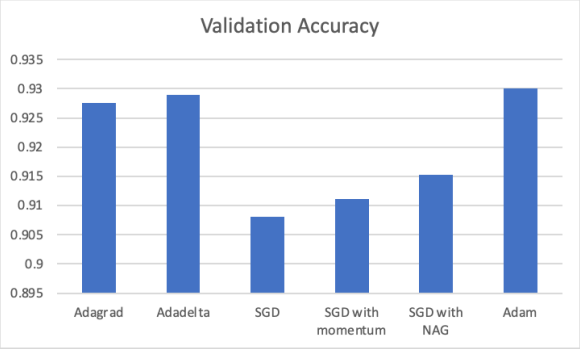


Fig. 10. Accuracy for different optimizers

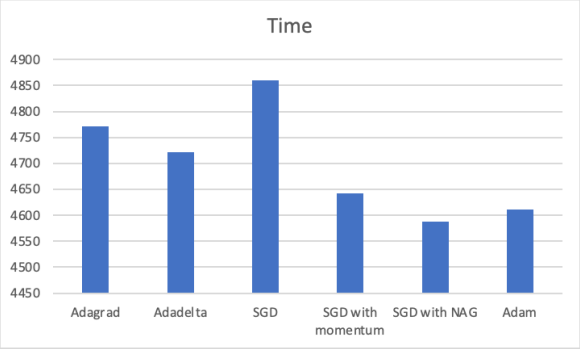


Fig. 11. Time in sec for different optimizers

Conclusion

In this project, we have initially looked at the three variants of gradient descent, among which mini-batch gradient descent is the most popular. We have then investigated algorithms that are most commonly used for optimizing SGD: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, and Adam. Then we apply these optimizers to a practical use case and verify their performance.

Bibliography

1. Leemon Baird and Andrew Moore. Gradient descent for general reinforcement learning. In M. Kearns, S. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11. MIT Press, 1998.
2. Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On empirical comparisons of optimizers for deep learning, 2020.
3. Sebastian Ruder. An overview of gradient descent optimization algorithms. 2017.
4. Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A Survey of Optimization Methods from a Machine Learning Perspective, 2019.