Convex and Non-Convex Function
Neural Network Architecture | Activation Functions
Optimization Algorithms in Deep Learning

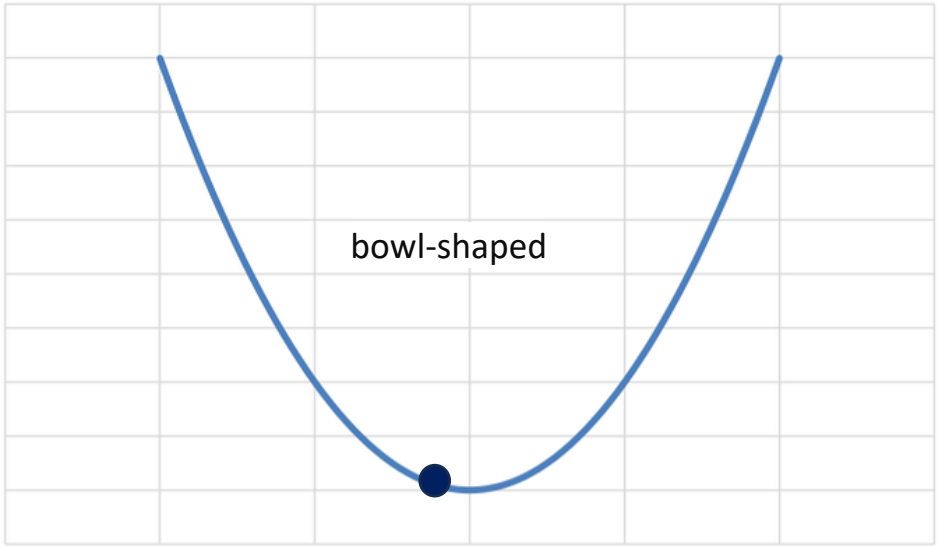


Fig: Convex Function

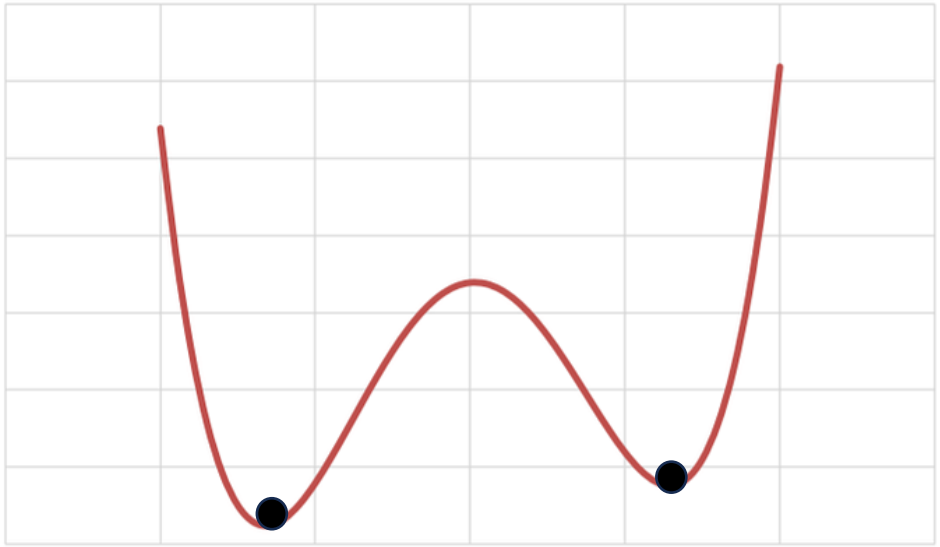


Fig: Non-convex Function

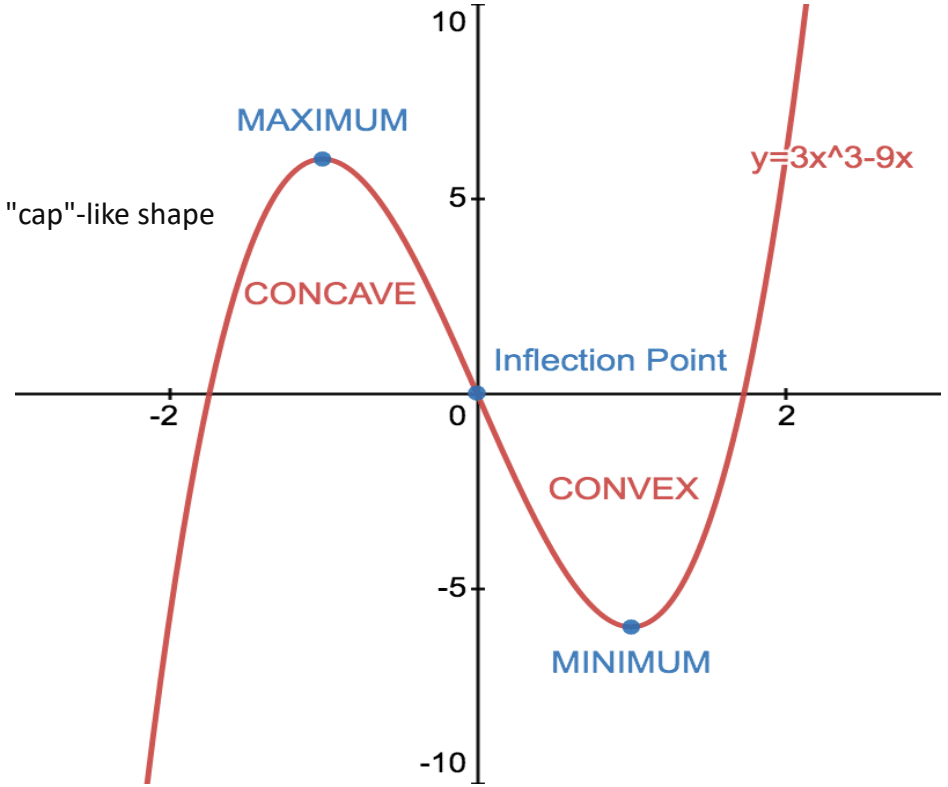


Fig: Convex/concave Function

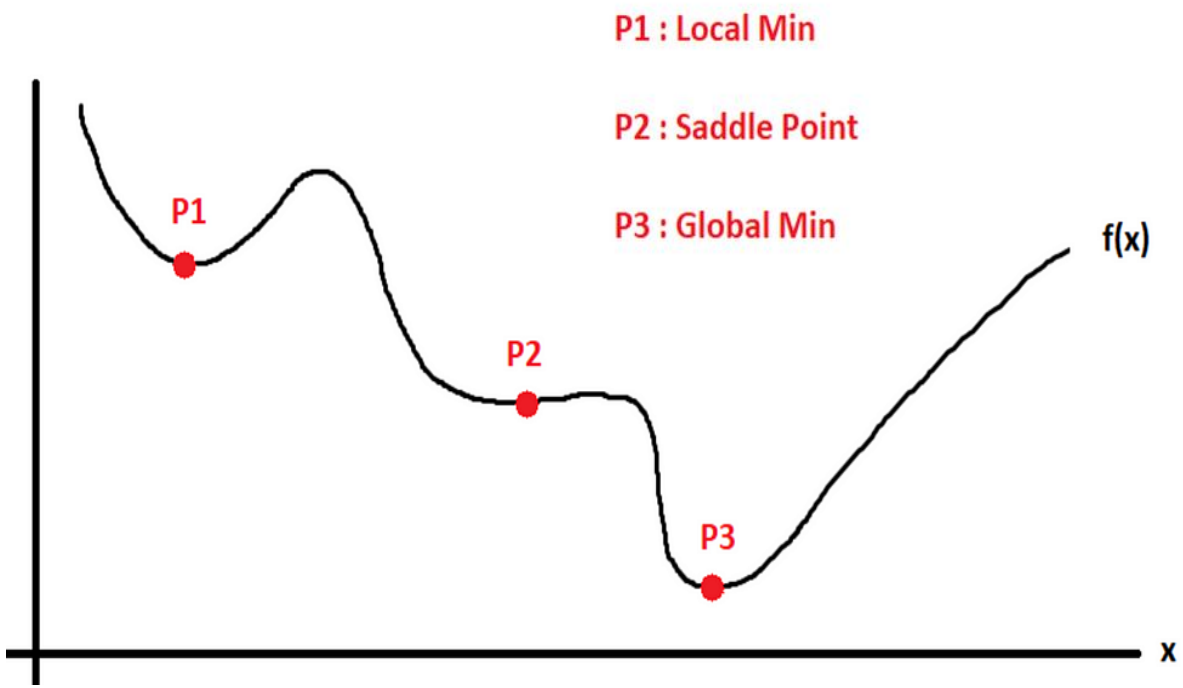


Fig: Non-convex Function

Names and Concepts in Convex Cost Functions:

- **Linear Regression:** Utilizes mean squared error (MSE) as its cost function, which is convex.
- **Logistic Regression:** Employs a logistic loss or cross-entropy loss, which is convex for binary classification tasks.
- **Support Vector Machines (SVM):** For linear SVMs, the hinge loss is convex. The goal is to find a hyperplane that separates the classes with a maximum margin.
- **Lasso (L1 regularization) and Ridge (L2 regularization):** Both regularization techniques add a convex penalty to the loss function to prevent overfitting, with Lasso promoting sparsity in the model coefficients.
- **Gradient Descent:** A popular optimization algorithm often used for models with convex cost functions due to its effectiveness in finding global minima.

Names and Concepts in Non-convex Cost Functions:

- **Neural Networks:** The cost functions in neural networks, including Convolutional Neural Networks (CNNs) for image tasks, Recurrent Neural Networks (RNNs) for sequential data, and Transformers for natural language processing, are inherently non-convex.
- **Deep Reinforcement Learning Models:** Models like Deep Q-Networks (DQN) and policy gradient methods operate with non-convex cost functions.
- **Autoencoders and Generative Adversarial Networks (GANs):** Both types of networks involve non-convex optimization problems, with GANs featuring a particularly challenging landscape due to the adversarial training process.
- **Adam, RMSprop, and SGD with Momentum:** These are examples of optimization algorithms designed to better navigate the complex landscapes of non-convex cost functions. They adjust the learning rate dynamically and use various strategies to avoid local minima.

1. Cross-Entropy Loss (Softmax Loss)

- **Usage:** Primarily used for classification problems.
- **Convexity:** The cross-entropy loss function itself is convex. However, when combined with the non-linearities and the deep architecture of a CNN, the overall optimization problem becomes non-convex.

2. Mean Squared Error (MSE)

- **Usage:** Often used for regression tasks. While less common in pure classification problems, MSE can be used in CNNs for tasks like image reconstruction or any regression-based problem.
- **Convexity:** MSE is a convex function. However, as with cross-entropy loss, when used in the context of CNNs with multiple layers and non-linear activations, the overall optimization landscape becomes non-convex.

3. Binary Cross-Entropy Loss

- **Usage:** Used for binary classification tasks within CNN frameworks. Particularly useful for tasks where the output is a probability that the input belongs to one of two classes.
- **Convexity:** The binary cross-entropy function is convex. However, the composition of this loss function with a CNN's architecture results in a non-convex optimization problem.

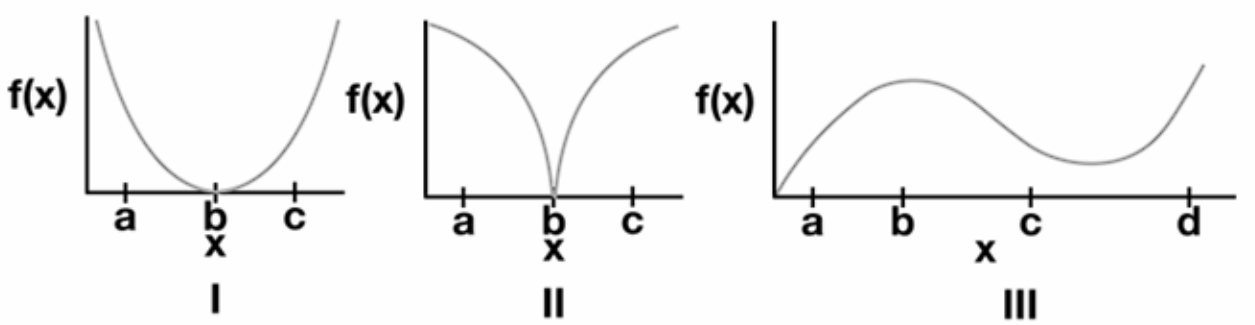
4. Hinge Loss

- **Usage:** Employed in "max-margin" classification, most notably in support vector machines (SVMs), but can also be adapted for use in CNNs, especially for some types of classification tasks.
- **Convexity:** Hinge loss is convex. Nevertheless, similar to other loss functions, the end-to-end training process of a CNN involving hinge loss is non-convex due to the model's complexity.

5. Categorical Cross-Entropy Loss

- **Usage:** An extension of the binary cross-entropy loss, used for multi-class classification problems. It's particularly common in CNNs designed for classifying inputs into more than two categories.
- **Convexity:** Like binary cross-entropy, the categorical cross-entropy loss is convex by itself. The optimization landscape of a CNN using this loss, however, is non-convex.

For each of the functions below (I-III), state whether each one is convex on the given interval or state why not with a counterexample using any of the points a, b, c, d in your answer.



(a) Function in panel I on $[a, c]$

(b) Function in panel II on $[a, c]$

(c) Function in panel III on $[a, d]$

(d) Function in panel III on $[c, d]$

(a) Convex.

(b) Not convex on $[a, c]$: Every function value in (a, b) is greater than the corresponding convex combination of the function values $f(a)$ and $f(b)$. Similar situation happens if we consider the interval (b, c).

(c) Not convex on $[a, c]$: Every function value in (a, c) is greater than the corresponding convex combination of the function values $f(a)$ and $f(c)$.

(d) Convex.

Optimization Algorithms in Deep Learning

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini-Batch Gradient Descent
4. Momentum
5. Nesterov Accelerated Gradient (NAG)
6. Adagrad
7. Adadelta
8. RMSprop
9. Adam
10. Adamax
11. Nadam
12. AMSGrad
13. FTRL (Follow The Regularized Leader)
14. L-BFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno)
15. Proximal Gradient Methods

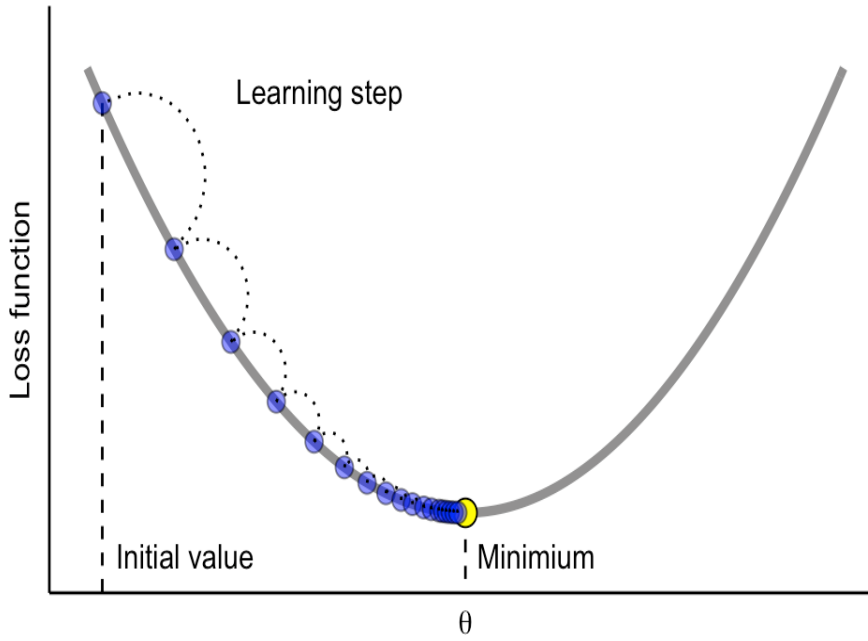
Gradient descent is a first-order **iterative optimization algorithm** for finding the minimum of a function. Gradient descent can be performed on any differentiable loss function. The main goal of gradient descent is to minimize a cost or loss function, optimizing model parameters for better performance.

1. Objective Function: The function you want to minimize (cost or loss function).

2. Gradient Calculation: Compute the gradient of the function, indicating the direction of the steepest ascent.

3. Learning Rate: A hyperparameter that determines the size of the steps taken towards the minimum.

4. Update Rule: A formula to iteratively adjust the parameters in the opposite direction of the gradient to minimize the function.



The main equation for updating parameters in gradient descent is:

$$x_{\text{next}} = x - \alpha \cdot \nabla f(x) \quad \text{Or,} \quad w_{\text{new}} = w_{\text{old}} - \alpha \frac{\partial J}{\partial w}$$

where:

- x_{next} is the updated parameter value,
- x is the current parameter value,
- α is the learning rate,
- $\nabla f(x)$ is the gradient of the function f at x.

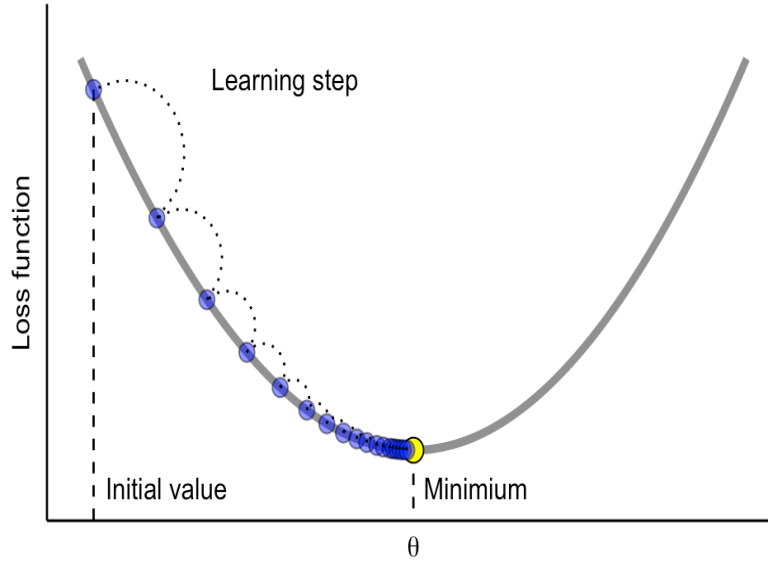


Fig: Gradient Decent

$$w_{\text{new}} = w_{\text{old}} - \alpha \frac{\partial J}{\partial w}$$

where:

- w_{new} is the updated weight.
- w_{old} is the current weight.
- α is the learning rate.
- $\frac{\partial J}{\partial w}$ is the gradient of the loss function with respect to the weight.

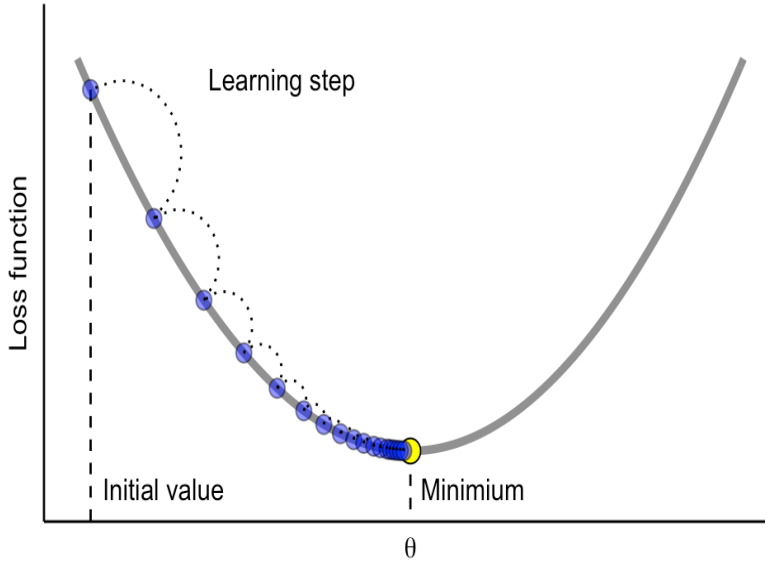


Fig: Gradient Decent

Some loss functions are widely used in machine learning and deep learning models, and they can be minimized effectively using gradient descent to improve model performance.

- **Mean Squared Error (MSE)/MAE:** Used for regression tasks, measuring the average of the squares of the errors between actual and predicted values.
- **Cross-Entropy Loss (or Log Loss)**: Used for binary and multi-class classification tasks, measuring the performance of a classification model whose output is a probability value between 0 and 1.
- **Hinge Loss**: Used for binary classification tasks, especially with Support Vector Machines (SVMs), measuring the error between actual and predicted class labels.
- **Huber Loss**: Used for regression tasks, combining the properties of MSE and Mean Absolute Error (MAE) to be less sensitive to outliers than MSE.
- **Softmax Cross-Entropy Loss:** A generalization of Cross-Entropy Loss for multi-class classification tasks, applied when the model outputs a probability distribution over multiple classes.

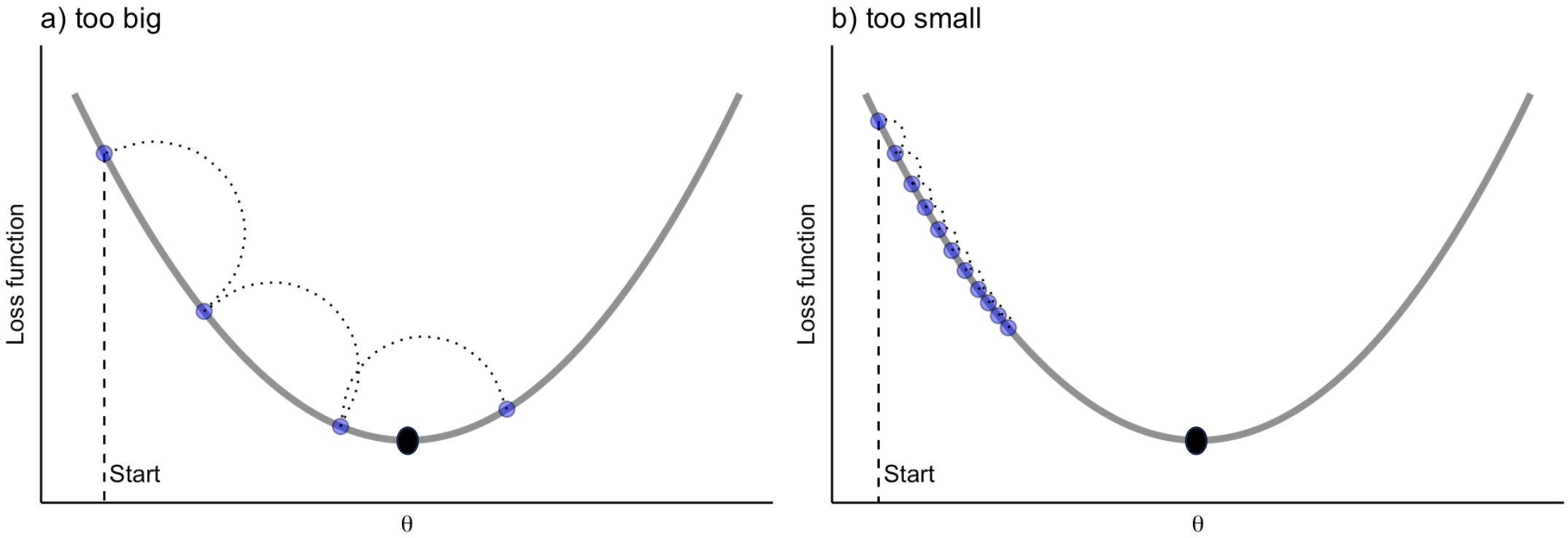
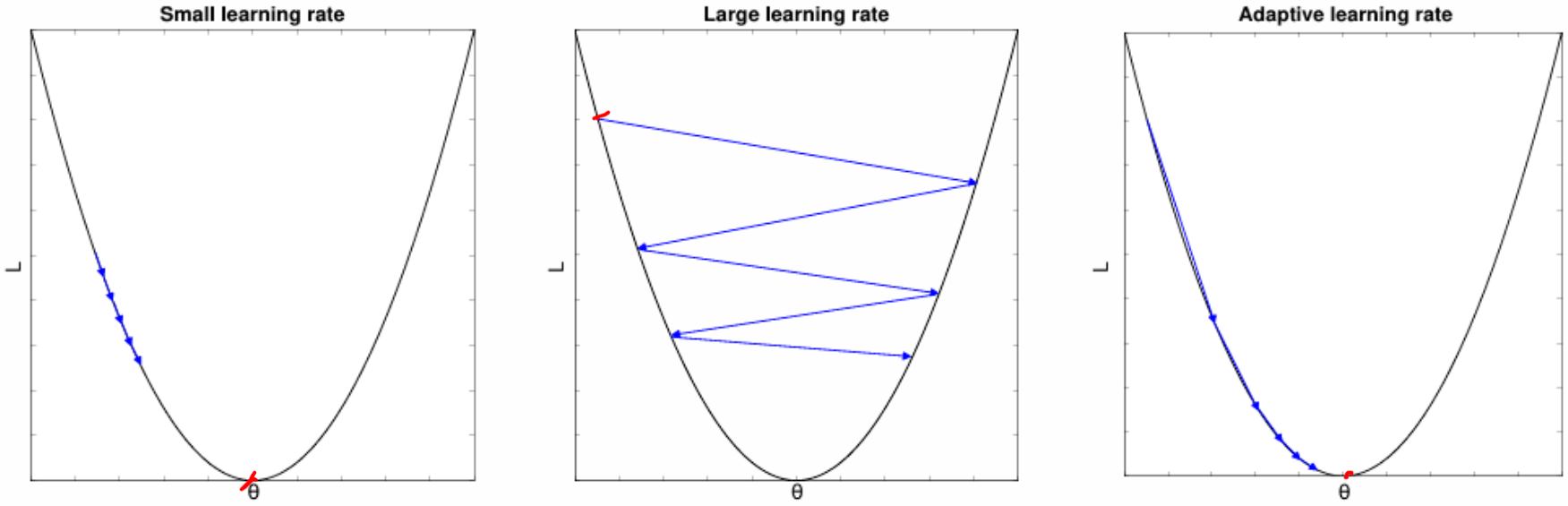


Fig: Importance of Learning Rate



- η too small: long training time
- η too large: miss optima
- Practice: “learning rate decay”: adapt η gradually (e.g.: start with $\eta = 0.01$ and divide every x epoch by 10)

The ideal value of the learning rate (α or η) in gradient descent varies widely depending on the specific problem, the dataset, the model being trained, and the chosen loss function. There isn't a one-size-fits-all value. However, the selection of the learning rate is crucial because:

- **If it's too large**, the algorithm might overshoot the minimum, leading to divergence or oscillation around the minimum without ever settling.
- **If it's too small**, convergence to the minimum can be very slow, significantly increasing the training time and potentially getting stuck in local minima.

The typical range for the learning rate (α or η) in gradient descent is from 0.00001 to 0.1. This range is usually explored to find an optimal value for specific training tasks.

Types of Gradient Descent Optimizer in Machine Learning

Gradient descent is a fundamental **optimization algorithm** used in machine learning and deep learning to **minimize a loss function**. There are several variations of gradient descent, each with its characteristics and use cases. The main difference between Batch Gradient Descent (BGD) and other types of gradient descent algorithms (such as Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent) lies in **how the dataset is used to compute the gradient during each iteration.**

The main types of gradient descent are:

- Batch Gradient Descent (BGD)
- Stochastic Gradient Descent (SGD)
- Mini-Batch Gradient Descent

$$w_{\text{new}} = w_{\text{old}} - \alpha \frac{\partial J}{\partial w}$$

where:

- w_{new} is the updated weight.
- w_{old} is the current weight.
- α is the learning rate.
- $\frac{\partial J}{\partial w}$ is the gradient of the loss function with respect to the weight.

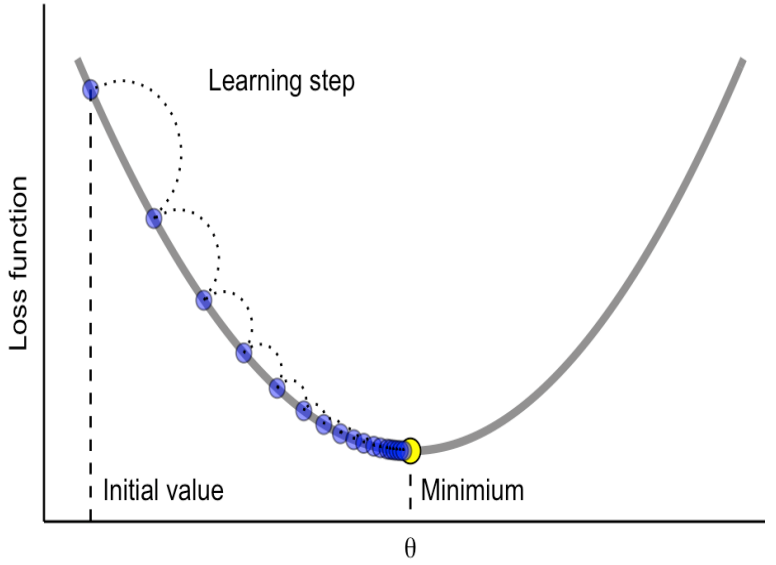


Fig: Gradient Descent

- **Batch Gradient Descent:** Uses the **entire dataset** to compute the gradient, which is more stable but can be slow for large datasets.
- **Stochastic Gradient Descent:** Updates parameters for **each data point**, which can be faster but noisier.
- **Mini-Batch Gradient Descent:** Uses **small batches** of data, balancing the speed and stability of the updates.

Batch Gradient Descent: Uses the **entire dataset** to compute the gradient, which is more stable but can be slow for large datasets.

Given a cost function $J(\theta)$:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(h_\theta(x^{(i)}), y^{(i)}) \quad \Big| \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

MSE

where m is the number of training examples, L is the loss function, h_θ is the hypothesis function, $x^{(i)}$ is the input features, and $y^{(i)}$ is the target output.

The update rule is:

$$\theta = \theta - \alpha \nabla_\theta J(\theta)$$

where α is the learning rate.

Mini-Batch Gradient Descent: Mini-Batch Gradient Descent is a compromise between BGD and SGD. The dataset is divided into small batches, and each batch is used to compute the gradient and update the parameters.

- Total samples: 1000
- Number of mini-batches: 16

To find the number of iterations per epoch, we divide the total number of samples by the number of mini-batches:

$$\text{Mini-Batch Size} = \frac{\text{Total Samples}}{\text{Number of Mini-Batches}} = \frac{1000}{16} = 62.5$$

Iterations Per Epoch

Each epoch consists of one pass through all the mini-batches. So, if we have 16 mini-batches, each epoch will have 16 iterations.

Number of Iterations for Multiple Epochs

If you plan to train for N epochs, the total number of iterations will be:

$$\text{Total Iterations} = \text{Number of Epochs} \times \text{Number of Mini-Batches}$$

For example, if you train for 10 epochs:

$$\text{Total Iterations} = 10 \times 16 = 160$$

Stochastic Gradient Descent: In Stochastic Gradient Descent, the parameters are updated for **each training example**. This means that the gradient is computed using **only one data point at a time**.

Mathematical Formulation:

For each training example $(x^{(i)}, y^{(i)})$, the update rule is:

$$\theta = \theta - \alpha \nabla_\theta L(h_\theta(x^{(i)}), y^{(i)})$$

Example:

Using the same dataset and hypothesis as above, for a single data point $(1, 2)$:

$$\theta_0 = \theta_0 - \alpha(h_\theta(1) - 2)$$

$$\theta_1 = \theta_1 - \alpha(h_\theta(1) - 2) \cdot 1$$

Next Slide >>

The parameters θ_0 and θ_1 are updated after each data point.

Update process for a **single** data point (3,5) using Stochastic Gradient Descent (SGD). So, x=3, and y=5.

Initial Parameters

Assume initial parameters:

- $\theta_0 = 0$
- $\theta_1 = 1$
- Learning rate $\alpha = 0.1$

Hypothesis Function

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Update process for a **single** data point (3,5) using Stochastic Gradient Descent (SGD). So, x=3, and y=5.

Steps to Update Parameters

1. Compute the Hypothesis

For $x = 3$:

$$h_\theta(3) = \theta_0 + \theta_1 \cdot 3$$

Using the initial values:

$$h_\theta(3) = 0 + 1 \cdot 3 = 3$$

2. Calculate the Error

$$\text{Error} = h_\theta(3) - 5$$

$$\text{Error} = 3 - 5 = -2$$

3. Update θ_0

$$\theta_0 = \theta_0 - \alpha(\text{Error})$$

$$\theta_0 = 0 - 0.1 \cdot (-2)$$

$$\theta_0 = 0 + 0.2 = 0.2$$

4. Update θ_1

$$\theta_1 = \theta_1 - \alpha(\text{Error}) \cdot 3$$

$$\theta_1 = 1 - 0.1 \cdot (-2) \cdot 3$$

$$\theta_1 = 1 + 0.6 = 1.6$$

Differences between Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent:

Feature	Batch Gradient Descent (BGD)	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
Gradient Computation	Entire dataset	Single data point	Small subset (mini-batch)
Update Frequency	Once per epoch	Once per training example	Once per mini-batch
Computational Efficiency	Computationally expensive for large datasets	Highly efficient, suitable for large datasets	Moderately efficient
Convergence Stability	Stable and precise updates	Noisy updates, high variance	Balance between BGD and SGD, reduced variance

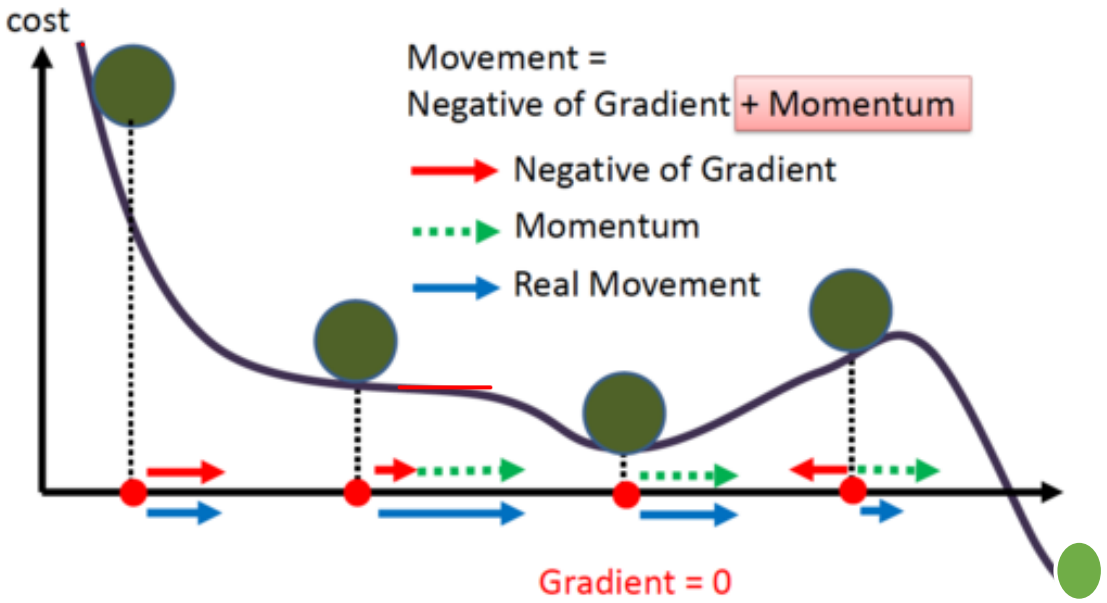
Gradient Descent with Momentum in Machine Learning

Momentum: Momentum is a technique used to accelerate gradient descent algorithms. It improves the stability and speed of convergence in gradient-based optimization by combining the current gradient with the past gradients' influence, leading to more efficient training of machine learning models.

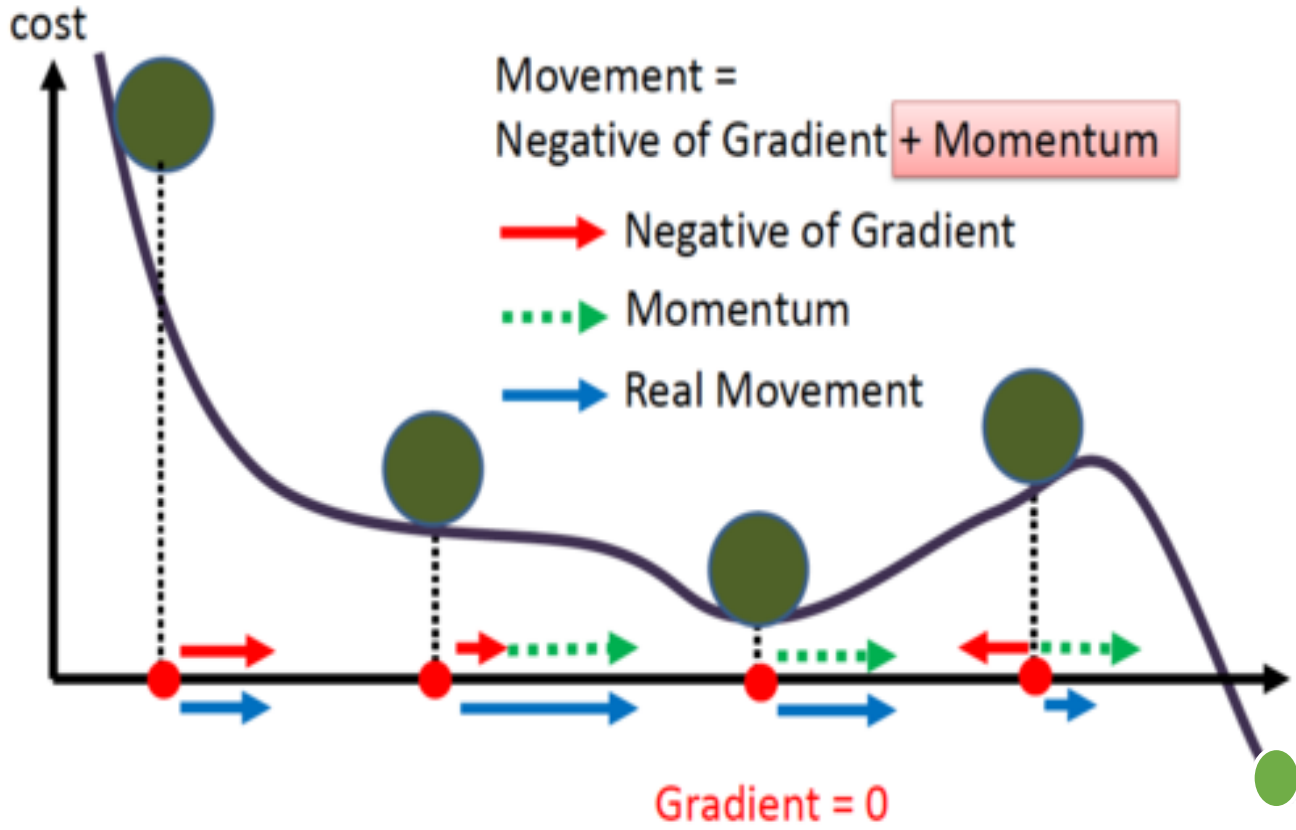
Momentum helps the optimization process by considering both the current gradient and an accumulation of past gradients. There are **two main** steps involved in applying momentum to the gradient descent optimization process:

1. Velocity Update
2. Parameter Update

Note: The main point of the diagram is to illustrate how momentum helps to avoid getting stuck in local minima and can push the optimization process past these points, unlike traditional gradient descent which might oscillate or come back to a local minimum without momentum.



- The **red** arrows indicate the direction of the negative gradient. If the algorithm relied only on these, it might get stuck in local minima where the gradient is zero.
- The **green** dotted arrows represent the momentum. This term accumulates the gradients over time and can help in moving past small local minima.
- The **blue** arrows show the real movement of the parameters, which is the combined effect of the negative gradient and the momentum.



1. **Velocity Update**:

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} - \eta \nabla L(\mathbf{w}^{(k)})$$

- $\mathbf{v}^{(k)}$ represents the velocity or the accumulated gradient at iteration k.
- μ is the momentum coefficient.
- η is the learning rate.
- $\nabla L(\mathbf{w}^{(k)})$ is the gradient of the loss function with respect to the weights $\mathbf{w}$ at iteration k.
- $\mu \mathbf{v}^{(k-1)}$: This term incorporates the influence of past gradients. The coefficient μ (usually between 0.9 and 0.99) determines the contribution of the previous velocity. Higher μ values give more weight to the accumulated past gradients.
- $-\eta \nabla L(\mathbf{w}^{(k)})$: This term represents the current gradient scaled by the learning rate η. This influences the direction and magnitude of the update based on the current gradient.

Parameter Update:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

- $\mathbf{w}^{(k+1)}$: The new parameters for the next iteration.
- $\mathbf{w}^{(k)}$: The current parameters.
- $\mathbf{v}^{(k)}$: The updated velocity that combines the past and current gradient information.

These two steps together ensure that the parameter updates benefit from both the current gradient and an accumulated direction of previous gradients, which helps to smooth out the path towards the minimum of the loss function and accelerates convergence.

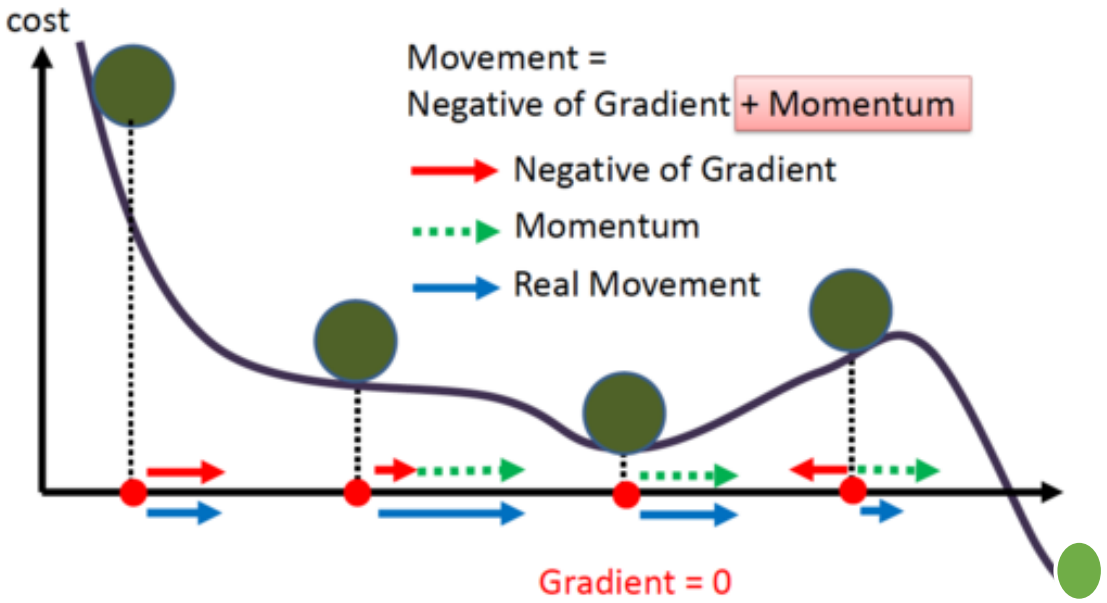
Nesterov Accelerated Gradient (NAG) / Nesterov Momentum in ML

Momentum: Momentum is a technique used to accelerate gradient descent algorithms. It improves the stability and speed of convergence in gradient-based optimization by combining the **current gradient** with the **past gradients**' influence, leading to more efficient training of machine learning models.

Momentum helps the optimization process by considering both the current gradient and an accumulation of past gradients. There are **two main** steps involved in applying momentum to the gradient descent optimization process:

1. Velocity Update
2. Parameter Update

Note: The main point of the diagram is to illustrate how momentum helps to avoid getting stuck in local minima and can push the optimization process past these points, unlike traditional gradient descent which might oscillate or come back to a local minimum without momentum.

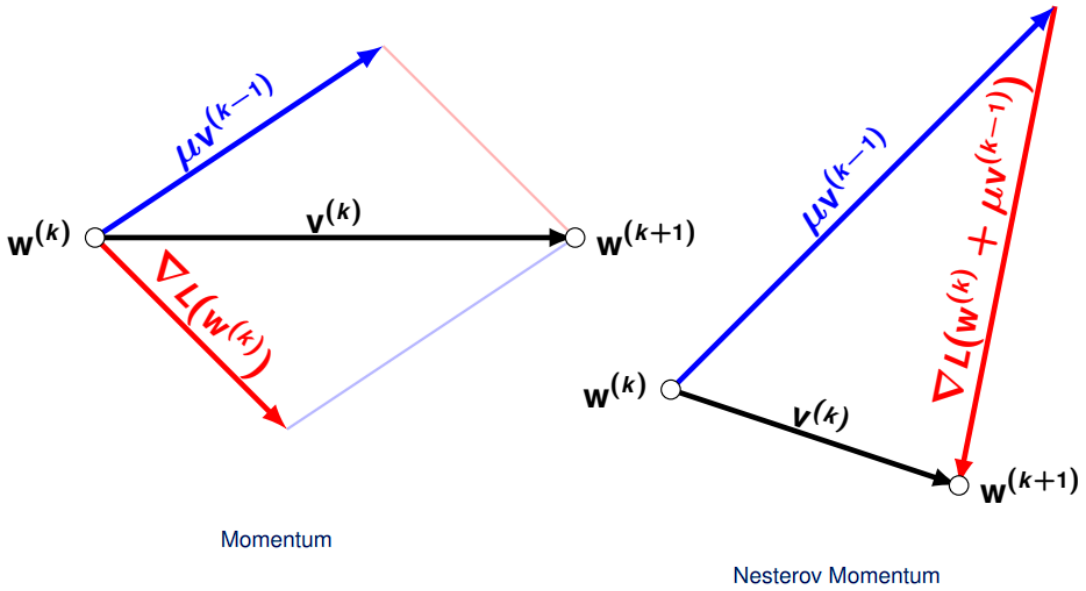


- **Works on the current gradient with the past gradients**:
  - In traditional momentum, the update step involves the current gradient and an accumulated velocity term, which is a combination of past gradients.
  - The update equations are:

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} - \eta \nabla L(\mathbf{w}^{(k)})$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

  - Here, $\mathbf{v}^{(k)}$ is influenced by past gradients via $\mathbf{v}^{(k-1)}$, but the gradient $\nabla L(\mathbf{w}^{(k)})$ is computed at the current position $\mathbf{w}^{(k)}$.



- **Traditional Momentum**: Uses the current gradient at the current position $\mathbf{w}^{(k)}$ along with an accumulated momentum term derived from past gradients.
- **Nesterov Momentum**: Computes the gradient at a future anticipated position $\mathbf{w}^{(k)} + \mu\mathbf{v}^{(k-1)}$, providing a more forward-looking approach that can lead to better convergence properties.

- **Works on future position**:
  - In Nesterov momentum, the gradient is computed at a lookahead position, which takes into account where the parameters will be after the current step.
  - The update equations are:

$$\mathbf{v}^{(k)} = \mu\mathbf{v}^{(k-1)} - \eta\nabla L(\mathbf{w}^{(k)} + \mu\mathbf{v}^{(k-1)})$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

  - The gradient $\nabla L(\mathbf{w}^{(k)} + \mu\mathbf{v}^{(k-1)})$ is computed at the anticipated future position $\mathbf{w}^{(k)} + \mu\mathbf{v}^{(k-1)}$. **Explanation in Next Slide >>**

Two main steps in Nesterov Momentum:

1. **Velocity Update**:

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} - \eta \nabla L(\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)})$$

- **Momentum Component**: $\mu \mathbf{v}^{(k-1)}$ is the contribution from the past velocity, scaled by the momentum coefficient μ.
- **Gradient Component**: $\eta \nabla L(\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)})$ is the learning rate times the gradient of the loss function, evaluated at the "lookahead" position $\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)}$.

2. **Parameter Update**:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

- The current parameters $\mathbf{w}^{(k)}$ are updated by adding the velocity $\mathbf{v}^{(k)}$.

1. **Velocity Update**:

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} - \eta \nabla L(\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)})$$

- $\mathbf{v}^{(k)}$: The velocity (momentum term) at iteration k.
- μ: The momentum coefficient, a value between 0 and 1 that determines the influence of the previous velocity.
- η: The learning rate, a small positive value that controls the step size during gradient descent.
- $\nabla L(\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)})$: The gradient of the loss function L evaluated at the "lookahead" position $\mathbf{w}^{(k)} + \mu \mathbf{v}^{(k-1)}$.
  - $\mathbf{w}^{(k)}$: The current parameters at iteration k.
  - $\mathbf{v}^{(k-1)}$: The velocity (momentum term) from the previous iteration.

2. **Parameter Update**:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

- $\mathbf{w}^{(k+1)}$: The updated parameters for the next iteration.
- $\mathbf{w}^{(k)}$: The current parameters.
- $\mathbf{v}^{(k)}$: The updated velocity from the first equation.

- **Faster Convergence:** By computing the gradient at a future position, Nesterov momentum often leads to faster convergence and more efficient optimization.

- **More Accurate Updates:** The lookahead mechanism provides a more accurate correction to the parameter updates, which can lead to better performance, particularly in cases where the optimization landscape has high curvature or when dealing with noisy gradients.

- **Convex Optimization:** In convex optimization problems, Nesterov Momentum can significantly speed up convergence compared to standard gradient descent and even traditional momentum methods.

- **For Non-Convex Problems:** While lacking the same level of theoretical guarantees, it still provides practical benefits in many real-world applications, especially in deep learning and other complex optimization tasks.

How does AdaGrad Optimizer work?

AdaGrad (Adaptive Gradient Algorithm) is an optimization algorithm designed to adapt the learning rate of each parameter individually during training. It is particularly useful for dealing with sparse data and for improving convergence in gradient descent. It is a **variant of gradient descent** that adapts the learning rate for each parameter individually. **Four main steps:**

1. Weight Initialization
2. **Compute the gradient**
3. **Gather the squared gradients**
4. **Adjust the learning rate**

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$
$$\mathbf{r}^{(k)} = \mathbf{r}^{(k-1)} + \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \frac{\eta}{\sqrt{\mathbf{r}^{(k)}} + \epsilon} \odot \mathbf{g}^{(k)}$$

- **Ada**ptive **Grad**ient
- Adaption based on all past squared gradients
- We use $\odot$ to emphasize the element-wise multiplication

\+ Individual learning rates

\− Learning rate decreases too aggressively

1. Initialize Parameters and Accumulators

- **Initialize parameters**: Start with an initial guess for the parameters θ.
- **Set an initial learning rate**: Choose an initial learning rate η.
- **Initialize gradient accumulators**: Initialize the sum of squared gradients accumulator G to zero for all parameters. This accumulator will store the sum of the squares of the gradients for each parameter.

2. Gradient Computation:

At each iteration t, compute the gradient g_t of the loss function with respect to the parameters θ_t:

$$g_t = \nabla_\theta \mathcal{L}(\theta_t)$$

3. Accumulate Squared Gradients

- **Update the accumulators**: Add the square of the current gradient to the accumulators. This is done element-wise for each parameter:

$$G_t = G_{t-1} + g_t \odot g_t$$

where G_t is the diagonal matrix of accumulated squared gradients and $g_t \odot g_t$ represents the element-wise square of the gradient vector g_t.

Components of the Equation:

- G_t: This is the diagonal matrix of accumulated squared gradients at time step t.
- G_{t-1}: This is the diagonal matrix of accumulated squared gradients at the previous time step $t-1$.
- g_t: This is the gradient of the loss function with respect to the parameters θ at the current time step t.
- $g_t \odot g_t$: This denotes the element-wise (Hadamard) multiplication of the gradient vector g_t with itself, resulting in a vector where each element is the square of the corresponding element in g_t.

4. Update Parameters

- **Adjust the parameters**: Update the parameters using the adapted learning rate for each parameter. The learning rate is scaled by the accumulated gradients to ensure more controlled updates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

where ϵ is a small constant added to prevent division by zero, and $\odot$ denotes element-wise division.

Components of the Equation:

- θ_t: The parameter vector at iteration t.
- θ_{t+1}: The updated parameter vector for the next iteration $t + 1$.
- η: The initial learning rate. This is a predefined constant that controls the step size during the parameter update.
- G_t: The diagonal matrix of accumulated squared gradients up to the current iteration t.
- ϵ: A small constant added to the denominator to prevent division by zero. This is often set to a very small value like 10^{-8}.
- g_t: The gradient of the loss function with respect to the parameters θ at iteration t.
- $\odot$: Denotes element-wise multiplication.

Advantages of AdaGrad

- **Handling Sparse Data:** The adaptive nature of the learning rates makes AdaGrad particularly effective for handling sparse data, where some parameters might update more frequently than others.

- **Adaptive Learning Rates:** The key feature of this update rule is that it adapts the learning rate for each parameter based on the history of squared gradients. Parameters with large accumulated gradients get smaller updates, while parameters with small accumulated gradients get larger updates.

- **Improved Convergence:** By scaling the learning rate inversely with the accumulated gradients, AdaGrad can improve convergence, especially in cases where different parameters have gradients of varying magnitudes.

How does RMSProp Optimizer work?

Non-stationary data refers to time series data whose statistical properties, such as mean, variance, and covariance, change over time. This means that the data does not have a constant mean or variance and can exhibit trends, cycles, or other patterns that evolve over time. Many statistical models assume stationary data. Using non-stationary data in such models can lead to unreliable and misleading results.

Month	Sales
Jan-2018	150
Feb-2018	160
Mar-2018	170
...	...
Jan-2019	155
Feb-2019	165
...	...
Dec-2022	220

Optimization: Deep Learning & AI

RMSProp

RMSProp (Root Mean Square Propagation) is a popular optimization algorithm for training neural networks. Both AdaGrad and RMSProp are based on Stochastic Gradient Descent (SGD) but incorporate additional mechanisms to adapt the learning rate for each parameter, improving the efficiency and effectiveness of the optimization process.

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$

$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1-\rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \frac{\eta}{\sqrt{\mathbf{r}^{(k)}} + \epsilon} \odot \mathbf{g}^{(k)}$$

- Hinton suggests $\rho = 0.9, \eta = 0.001$

\+ The aggressive decrease is fixed

\− We still have to set the learning rate

1. **Gradient Computation**:

$$g^{(k)} = \nabla L(w^{(k)})$$

This equation computes the gradient $g^{(k)}$ of the loss function L with respect to the weights $w^{(k)}$ at iteration k.

2. **Update of the Running Average of Squared Gradients**:

$$r^{(k)} = \rho r^{(k-1)} + (1-\rho) g^{(k)} \odot g^{(k)}$$

Here, $r^{(k)}$ is the running average of the squared gradients, ρ is a decay rate (hyperparameter), and $\odot$ represents the element-wise multiplication. The decay rate ρ helps in exponentially decaying the past squared gradients and combining it with the current squared gradient.

3. **Parameter Update Rule**:

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{r^{(k)}} + \epsilon} \odot g^{(k)}$$

In this equation, $w^{(k+1)}$ represents the updated weights. η is the learning rate, and ϵ is a small value added for numerical stability to prevent division by zero.

Advantages of RMSProp

- **Deep Neural Networks:** RMSProp is highly effective for training deep neural networks where vanishing and exploding gradients are common issues.
- **Recurrent Neural Networks (RNNs):** It works well with RNNs, which are known for their difficulties with gradient propagation.
- **Sparse Gradients:** RMSProp can handle sparse gradients efficiently, making it suitable for applications like natural language processing and recommendation systems.
- **Non-stationary Environments:** In the context of optimization and machine learning, non-stationary refers to a situation where the statistical properties of the data or the objective function change over time. RMSProp can adapt more effectively than traditional gradient descent methods.

How does AdaDelta Optimizer work?

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Adadelta does not have an explicit global learning rate parameter (η) like many other optimization algorithms (e.g., SGD, RMSProp). Instead, it dynamically adjusts the learning rates based on the moving averages of the gradients and updates.

Steps of AdaDelta:

1. Gradient Computation
2. Exponential Moving Average of Squared Gradients
3. Step Size Calculation
4. Exponential Moving Average of Squared Step Sizes
5. Weights Update

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$

$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1-\rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$

$$\boldsymbol{\Delta}_x = -\frac{\sqrt{\mathbf{h}^{(k-1)}}}{\sqrt{\mathbf{r}^{(k)}} + \epsilon} \odot \mathbf{g}^{(k)}$$

$$\mathbf{h}^{(k)} = \rho \mathbf{h}^{(k-1)} + (1-\rho)\boldsymbol{\Delta}_x \odot \boldsymbol{\Delta}_x$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \boldsymbol{\Delta}_x$$

1. **Gradient Computation**:

$$g^{(k)} = \nabla L(w^{(k)})$$

This is the gradient of the loss function L with respect to the weights $w^{(k)}$ at iteration k.

2. **Update of the Running Average of Squared Gradients**:

$$r^{(k)} = \rho r^{(k-1)} + (1-\rho) g^{(k)} \odot g^{(k)}$$

Here, $r^{(k)}$ represents the running average of the squared gradients, where ρ is the decay rate (a hyperparameter), and $\odot$ indicates element-wise multiplication. This equation ensures that the algorithm keeps track of the past squared gradients.

3. **Step Size Calculation:**

$$\Delta_x = -\frac{\sqrt{h^{(k-1)}}}{\sqrt{r^{(k)}} + \epsilon} \odot g^{(k)}$$

This step determines the adaptive step size Δ_x using the accumulated gradients and the moving average of the squared updates.

Here's a brief explanation:

1. $\sqrt{h^{(k-1)}}$: Square root of the historical squared step sizes.
2. $\sqrt{r^{(k)}} + \epsilon$: Square root of the current squared gradients plus a small constant ϵ for numerical stability.
3. $\odot g^{(k)}$: Element-wise multiplication with the current gradient.

4. **Exponential Moving Average of Squared Step Sizes:**

$$h^{(k)} = \rho h^{(k-1)} + (1-\rho)\Delta_x \odot \Delta_x$$

This step updates the moving average of the squared step sizes.

5. **Weights Update:**

$$w^{(k+1)} = w^{(k)} + \Delta_x$$

This step updates the weights by adding the adaptive step size Δ_x.

- **Simplified Hyperparameter Tuning**:
  - Since Adadelta does not require a manual learning rate, it simplifies the hyperparameter tuning process.
- **Stability and Convergence**:
  - The adaptive nature helps in achieving more stable and potentially faster convergence, especially in the presence of noisy gradients.

How does Adam & Nadam (Nesterov-accelerated Adam) Optimizer work?

Optimization: Deep Learning & AI

Adam

Adam optimization algorithm, which is short for "Adaptive Moment Estimation." Adam is an adaptive learning rate optimization algorithm designed for training deep learning models. Adam **combines** the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp.

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} + (1-\mu)\mathbf{g}^{(k)}$$

$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1-\rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$

Bias correction: $\hat{\mathbf{v}}^{(k)} = \dfrac{\mathbf{v}^{(k)}}{1-\mu^k} \quad \hat{\mathbf{r}}^{(k)} = \dfrac{\mathbf{r}^{(k)}}{1-\rho^k}$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\hat{\mathbf{v}}^{(k)}}{\sqrt{\hat{\mathbf{r}}^{(k)}} + \epsilon}$$

1. Gradient Computation:

$$g^{(k)} = \nabla L(w^{(k)})$$

This step computes the gradient of the loss function L with respect to the current parameters $w^{(k)}$.

2. Exponential Moving Average of Gradients:

$$v^{(k)} = \mu v^{(k-1)} + (1 - \mu) g^{(k)}$$

This step calculates the exponential moving average of the gradients. Here, μ is a decay rate for the moving average, typically suggested as 0.9. $v^{(k)}$ represents the first moment estimate (mean).

3. Exponential Moving Average of Squared Gradients:

$$r^{(k)} = \rho r^{(k-1)} + (1-\rho) g^{(k)} \odot g^{(k)}$$

This step calculates the exponential moving average of the squared gradients, where ρ is another decay rate, typically suggested as 0.999. $r^{(k)}$ represents the second moment estimate (uncentered variance). **Next Slide ->>**

4. Bias Correction:

$$\hat{v}^{(k)} = \frac{v^{(k)}}{1-\mu^k}, \quad \hat{r}^{(k)} = \frac{r^{(k)}}{1-\rho^k}$$

To correct the bias introduced during the initialization of the moving averages (when they are closer to zero), bias-corrected estimates $\hat{v}^{(k)}$ and $\hat{r}^{(k)}$ are computed.

1. **First Moment Estimate ($v^{(k)}$):**

   - **Decay Rate (μ)**: This controls the exponential moving average of the gradients. It is typically set to a value like 0.9.
   - **Purpose**: The first moment estimate captures the mean direction of the gradients, helping to smooth out the updates by considering the average gradient direction over time.

2. **Second Moment Estimate ($r^{(k)}$):**

   - **Decay Rate (ρ)**: This controls the exponential moving average of the squared gradients. It is typically set to a value like 0.999.
   - **Purpose**: The second moment estimate captures the variance of the gradients, helping to adjust the learning rate based on the variability of the gradients.

3. Exponential Moving Average of Squared Gradients:

$$r^{(k)} = \rho r^{(k-1)} + (1-\rho) g^{(k)} \odot g^{(k)}$$

This step calculates the exponential moving average of the squared gradients, where ρ is another decay rate, typically suggested as 0.999. $r^{(k)}$ represents the second moment estimate (uncentered variance).

4. Bias Correction:

$$\hat{v}^{(k)} = \frac{v^{(k)}}{1-\mu^k}, \quad \hat{r}^{(k)} = \frac{r^{(k)}}{1-\rho^k}$$

To correct the bias introduced during the initialization of the moving averages (when they are closer to zero), bias-corrected estimates $\hat{v}^{(k)}$ and $\hat{r}^{(k)}$ are computed. **Next Slide ->>**

Why Bias Correction is Needed:

- **Initialization Bias**: When the moving averages are initialized to zero, they start biased towards zero, especially during the initial steps of training. This bias diminishes over time, but it can affect the early iterations of the optimization process.
- **Correcting the Bias**: By dividing by $1 - \mu^k$ and $1 - \rho^k$, the algorithm adjusts for this bias, providing more accurate estimates of the first and second moments. This correction is crucial for ensuring that the adaptive learning rates are appropriately scaled from the beginning of the training process.

5. Parameter Update:

$$w^{(k+1)} = w^{(k)} - \eta \frac{\hat{v}^{(k)}}{\sqrt{\hat{r}^{(k)}} + \epsilon}$$

This step updates the parameters using the bias-corrected estimates. η is the learning rate, and ϵ is a small constant (e.g., 10^{-8}) added for numerical stability to prevent division by zero.

refers to a variant of the Adam optimization algorithm that incorporates Nesterov-accelerated gradient (NAG) to potentially improve the convergence properties of the optimizer. This variant is known as Nadam, which stands for Nesterov-accelerated Adaptive Moment Estimation.

Nadam: refers to a variant of the Adam optimization algorithm that incorporates Nesterov-accelerated gradient (NAG) to potentially improve the convergence properties of the optimizer. This variant is known as Nadam, which stands for Nesterov-accelerated Adaptive Moment Estimation.

The update rule for Nadam incorporates the bias-corrected first and second moment estimates along with the Nesterov gradient:

$$w^{(k+1)} = w^{(k)} - \eta \left(\frac{\mu v^{(k-1)} + (1-\mu) g^{(k)}}{1-\mu^k} \right) \left(\frac{1}{\sqrt{\hat{r}^{(k)}} + \epsilon} \right)$$

Here, $v^{(k-1)}$ is the first moment estimate at the previous step, $g^{(k)}$ is the current gradient, μ is the decay rate for the first moment estimate, $\hat{r}^{(k)}$ is the bias-corrected second moment estimate, and ϵ is a small constant for numerical stability.

- **Robustness:** Nadam tends to be more robust in practice and can provide better convergence in some scenarios compared to Adam.

- **Hyperparameters:** Nadam typically uses the same default hyperparameters as Adam, making it easy to switch between the two.

How does AMSGrad Optimizer work?

The AMSGrad, an improvement over the Adam optimization algorithm, addressing some issues observed with Adam's convergence properties.

Issues with Adam:

- **Empirical Observations:** Adam has been observed to fail to converge to an optimal or good solution in some cases, particularly in convex optimization problems.

- **Theoretical Insight:** Adam and similar methods do not guarantee convergence for convex problems due to an error in the original convergence proof. This means that, under certain conditions, Adam may not find the optimal solution even if one exists.

- **Purpose**: AMSGrad was proposed to address this convergence issue by ensuring that the step size (learning rate) does not increase over time.

- **Modification**: AMSGrad modifies the Adam algorithm by introducing a simple fix to ensure that the adaptive learning rates remain non-increasing.

Bias-Corrected Second Moment Estimate: $\hat{r}^{(k)} = \max(\hat{r}^{(k-1)}, r^{(k)})$

- Instead of directly using the exponentially weighted average of past squared gradients $r^{(k)}$, AMSGrad uses the maximum of the past corrected second moment estimate $\hat{r}^{(k-1)}$ and the current $r^{(k)}$.
- This ensures that $\hat{r}^{(k)}$ is always non-decreasing, preventing the learning rate from becoming too large and destabilizing the training process.

1. **Non-Decreasing Second Moment Estimate**:

   By using the maximum function, AMSGrad ensures that $\hat{r}^{(k)}$ is always greater than or equal to $\hat{r}^{(k-1)}$. This means that $\hat{r}^{(k)}$ will never decrease, and thus the denominator in the parameter update rule ($\sqrt{\hat{r}^{(k)}} + \epsilon$) will not decrease.

2. **Stable Effective Learning Rate**:

   The parameter update rule in AMSGrad is:

$$w^{(k+1)} = w^{(k)} - \eta \frac{\hat{v}^{(k)}}{\sqrt{\hat{r}^{(k)}} + \epsilon}$$

   With $\hat{r}^{(k)}$ being non-decreasing, the fraction $\frac{\hat{v}^{(k)}}{\sqrt{\hat{r}^{(k)}}+\epsilon}$ ensures that the step size (effective learning rate) does not increase unexpectedly. This stability in the denominator prevents large updates that could destabilize the training process.

Thank you!