Deep Learning

BCSE-332L

Module 2:

Improving Deep Neural Networks

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

Outline

- ❑Mini-Batch Gradient Descent
- ❑Exponential Weighted Averages
- ❑Gradient Descent with Momentum
- ❑RMSProp and Adam Optimization
- ❑Hyperparameter tuning (Regularization)
- ❑Batch Normalization
- ❑Softmax Regression
- ❑Softmax Classifier
- ❑Deep Learning Frameworks
- ❑Data Augmentation
- ❑Under-fitting Vs Over-fitting

Mini-Batch Gradient Descent

- Mini-Batch Gradient Descent strikes a balance between Batch and Stochastic methods.
- It divides the data into smaller batches, processing each batch separately.
- Basically, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called mini-batches.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations.
- This method offers a compromise between speed and stability, making it a popular choice in deep learning applications.

Mini-Batch Gradient Descent

- Let’s say you have a data set with a million or more training points.
- What’s a reasonable way to implement supervised learning?
- One approach, of course, is to only use a subset of the rows.
- Mini-Batch Gradient Descent is like a skilled juggler, managing the trade-off between computational efficiency and the fidelity of the error gradient.
- It processes data in smaller, manageable chunks, allowing quicker and more frequent updates than batch gradient descent, yet more stable and efficient than the stochastic approach.

Mini-Batch Gradient Descent

❑How Mini-Batch Gradient Descent Works

❑Imagine dividing a large dataset into several mini-batches.

❑Each batch is processed, contributing to the overall learning of the model.

❑This division allows for more frequent updates, speeding up the learning process while ensuring more stability compared to SGD.

❑Configuring Mini-Batch Gradient Descent

❑Configuring the size of each mini-batch is a crucial aspect.

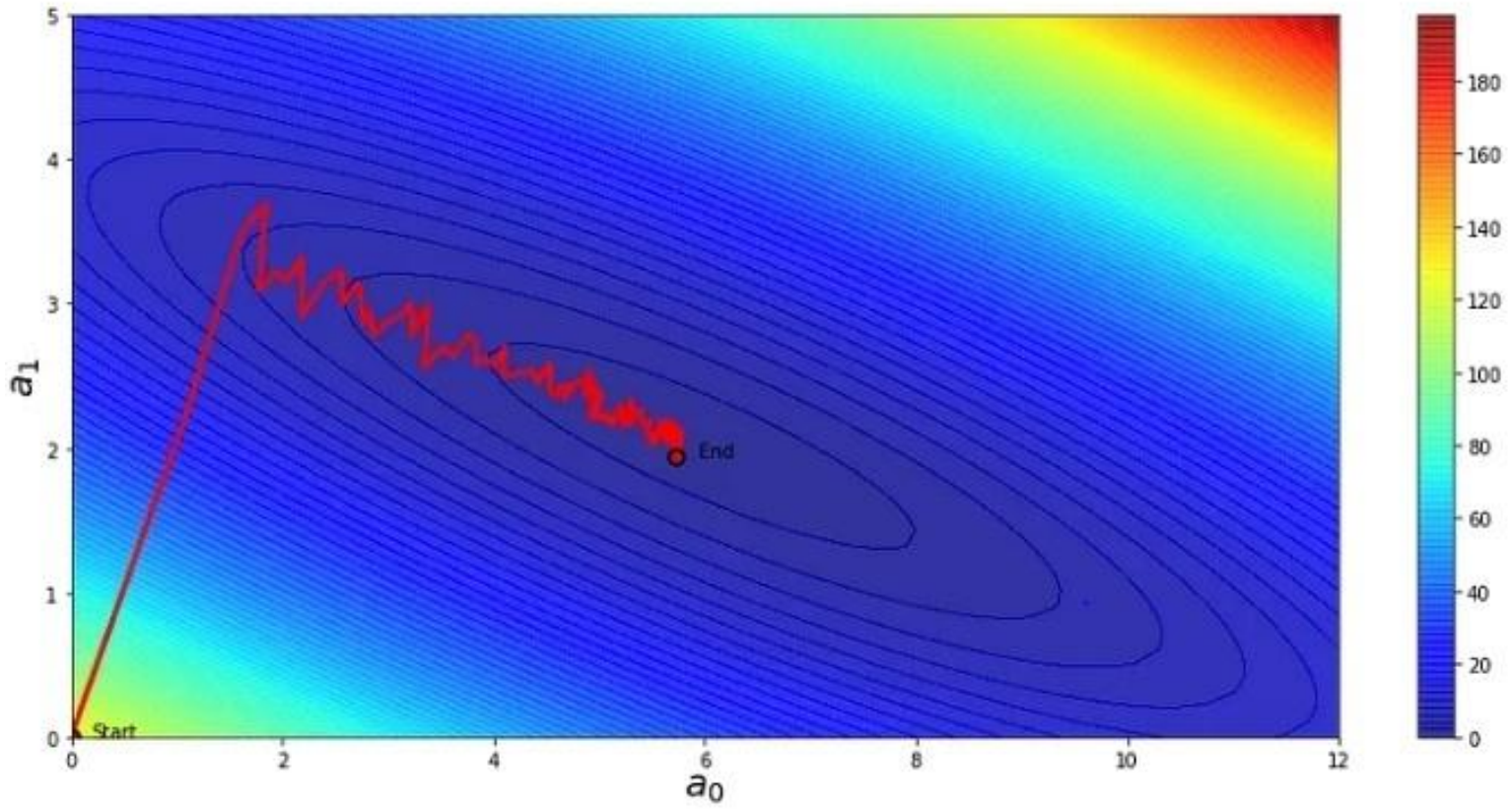
❑It's a balancing act between computational resources and learning efficiency.

❑Common batch sizes include 32, 64, or 128 data points, which are often chosen based on the hardware capabilities, like GPU or CPU memory.

Mini-Batch Gradient Descent

- We are already well-versed with the basics of mini-batch gradient descent now; let’s dive right in and explore in dept about mini-batch gradient descent.
- Mini-batch gradient descent is a gradient descent modification that divides the training dataset into small batches that are used to compute model error and update model coefficients.
- Implementations may opt to sum the gradient over the mini-batch, which minimizes the gradient's variance even further.
- Mini-batch gradient descent attempts to achieve a value between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.
- It is the most frequent gradient descent implementation used in regression techniques, neural networks, and deep learning.

Mini-Batch Gradient Descent



Mini-Batch Gradient Descent

❑Theoretically, in Mini- Batch Gradient descent, we move as follows:

1. We choose random data points and an initial learning rate.
2. In case the error keeps getting higher, reduce the learning rate
3. In case the error is fair but shows slow change, you can increase the learning rate.
4. With the help of minimal codes, we create a program to automate the rate of learning on the given dataset.
5. While reaching towards the last batch, one should decrease the learning rate as it helps reduce the noise and fluctuation in the data generated.
6. We turn down the learning rate once the error stops decreasing.

Mini-Batch Gradient Descent

❑The algorithm used:

1. We take random parameters from the dataset along with a maximum number of epochs the optimization should take place.

   Let theta = model parameters and max_iters = number of epochs.

2. We iterate the dataset to find the bias and the errors in the random parameters take

   for itr = 1, 2, 3, …, max_iters:

3. We divide the dataset into mini-batches and iterate them to update the required values.

   for mini_batch (X_mini, y_mini):

4. The update is made and in the batches, the prediction is made.

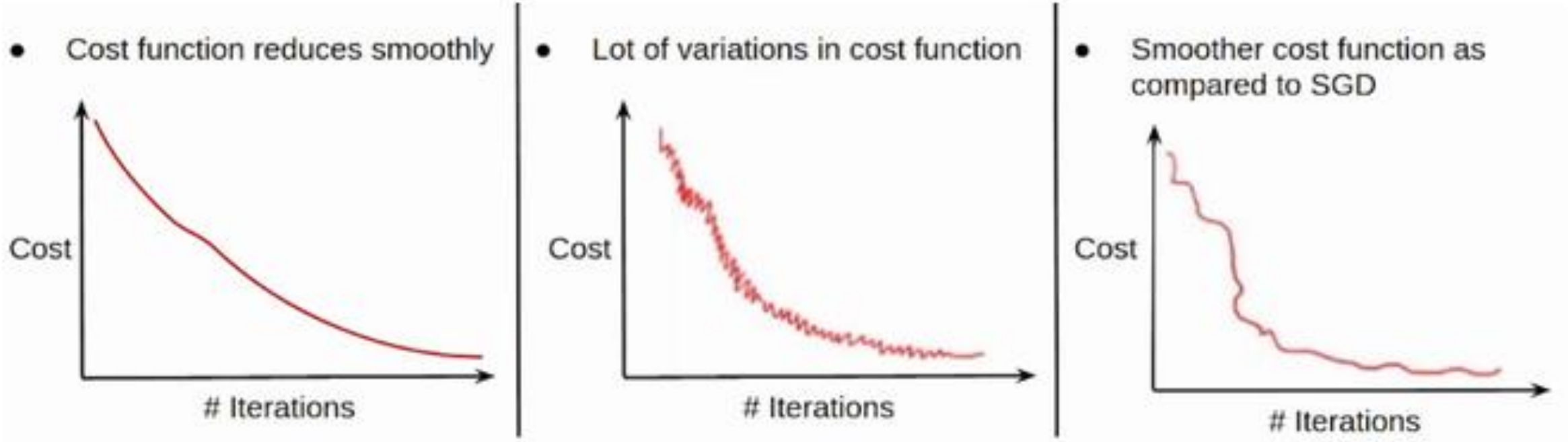
   Predict the batches X_mini:

Mini-Batch Gradient Descent

Batch Descent	Stochastic Descent	Mini-Batch Descent
Uses all training samples while training the dataset and only after the complete cycle makes the required update in the training dataset.	It uses one random training sample and makes the required update in the training dataset.	Divides the dataset into batches and, after completion of every batch makes the required update in the dataset.
Requires too much time and computational space for one epoch.	Faster than Batch descent and requires moderate computational space.	It is the fastest and requires minimal computational space.
Best to use for small training datasets	Best for training big datasets with less computational requirement.	Best for large training datasets.

Mini-Batch Gradient Descent

❑Thus, mini-batch gradient descent makes a compromise between the speedy convergence and the noise associated with gradient update which makes it a more flexible and robust algorithm.



Exponential Weighted Averages

- In time series analysis, there is often a need to understand the trend direction of a sequence by taking into account previous values.
- Approximation of the next values in a sequence can be performed in several ways, including the usage of simple baselines or the construction of advanced machine learning models.
- An exponential (weighted) moving average is a robust trade-off between these two methods.
- Having a simple recursive method under the hood makes it possible to efficiently implement the algorithm.
- At the same time, it is very flexible and can be successfully adapted for most types of sequences.

Exponential Weighted Averages

- Imagine a problem of approximating a given parameter that changes in time.
- On every iteration, we are aware of all of its previous values.
- The objective is to predict the next value which depends on the previous values.
- One of the naive strategies is to simply take the average of the last several values.
- This might work in certain cases but it is not very suitable for scenarios when a parameter is more dependent on the most recent values.
- One of the possible ways to overcome this issue is to distribute higher weights to more recent values and assign fewer weights to prior values.
- The exponential moving average is exactly a strategy that follows this principle.
- It is based on the assumption that more recent values of a variable contribute more to the formation of the next value than precedent values.

Exponential Weighted Averages

- To understand how the exponential moving average works, let us look at its recursive equation:

$$v_t = \underbrace{\beta v_{t-1}}_{\text{trend}} + \underbrace{(1-\beta)\theta}_{\text{current observation}}$$

- v is a time series that approximates a given variable. Its index t corresponds to the timestamp t.
- Since this formula is recursive, the value v_0 for the initial timestamp $t = 0$ is needed.
- In practice, v_0 is usually taken as 0.
- θ is the observation on the current iteration.
- β is a hyperparameter between 0 and 1 which defines how weight importance should be distributed between a previous average value v_{-1} and the current observation θ

Exponential Weighted Averages

- Let us write this formula for first several parameter values:

$$v_0 = 0$$

$$\begin{aligned} v_1 &= \beta v_0 + (1-\beta)\theta_1 \\ &= (1-\beta)\cdot\theta_1 \end{aligned}$$

$$\begin{aligned} v_2 &= \beta v_1 + (1-\beta)\theta_2 \\ &= \beta\cdot((1-\beta)\cdot\theta_1) + (1-\beta)\theta_2 \\ &= (1-\beta)\cdot(\theta_2 + \beta\theta_1) \end{aligned}$$

$$\begin{aligned} v_3 &= \beta v_2 + (1-\beta)\theta_3 \\ &= \beta\cdot((1-\beta)\cdot(\theta_2 + \beta\theta_1)) + (1-\beta)\theta_3 \\ &= (1-\beta)\cdot(\theta_3 + \beta\theta_2 + \beta^2\theta_1) \end{aligned}$$

$$\begin{aligned} v_4 &= \beta v_3 + (1-\beta)\theta_4 \\ &= \beta\cdot((1-\beta)\cdot(\theta_3 + \beta\theta_2 + \beta^2\theta_1)) + (1-\beta)\theta_4 \\ &= (1-\beta)\cdot(\theta_4 + \beta\theta_3 + \beta^2\theta_2 + \beta^3\theta_1) \end{aligned}$$

...

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta)\theta_t \\ &= \beta\cdot((1-\beta)\cdot(\theta_{t-1} + \beta\theta_{t-2} + \ldots + \beta^{t-4}\theta_3 + \beta^{t-3}\theta_2 + \beta^{t-2}\theta_1)) + (1-\beta)\theta_t \\ &= (1-\beta)\cdot(\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \ldots + \beta^{t-3}\theta_3 + \beta^{t-2}\theta_2 + \beta^{t-1}\theta_1) \end{aligned}$$

Exponential Weighted Averages

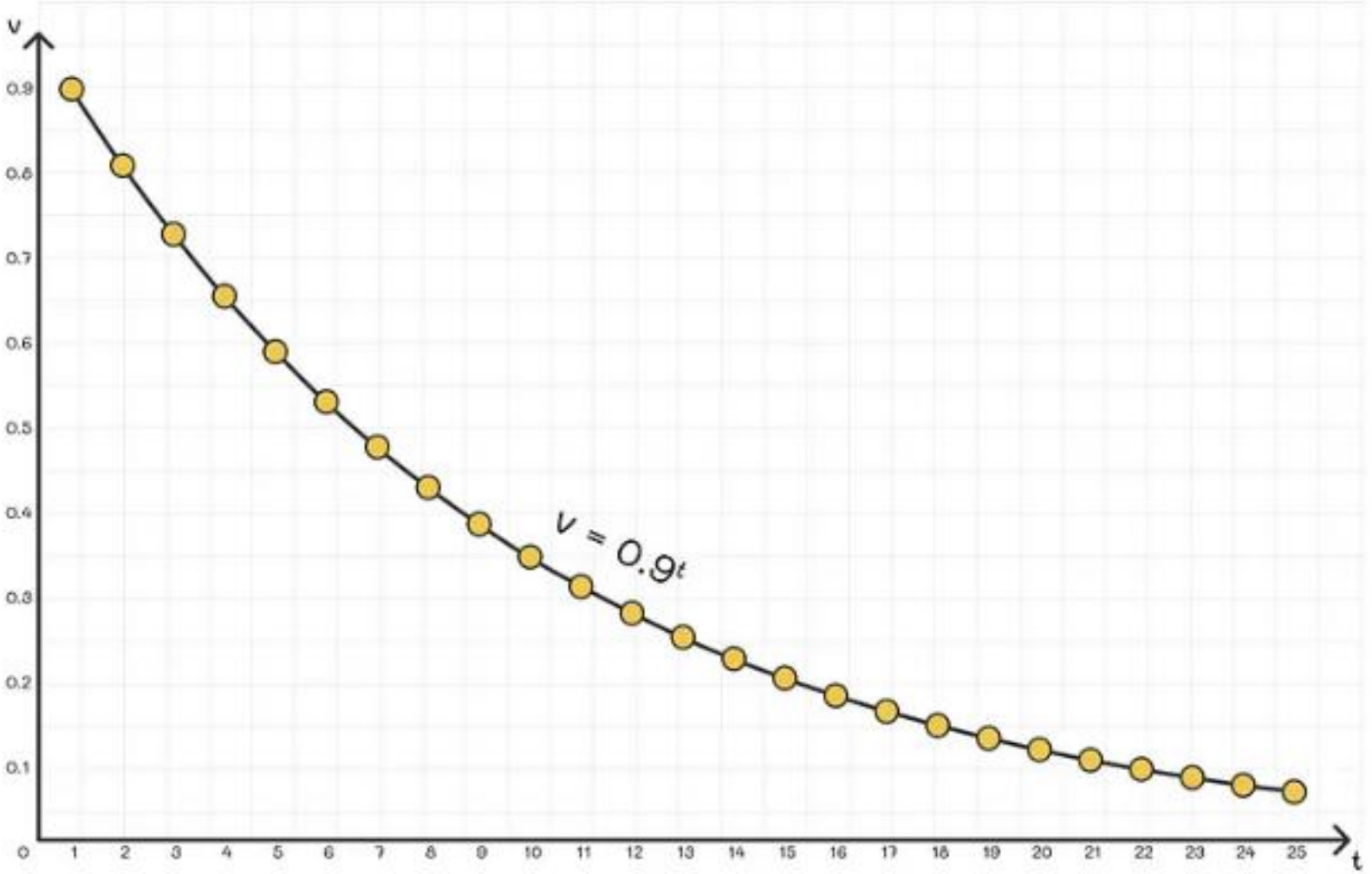
- As a result, the final formula looks like this:

$$v_t = (1 - \beta) \sum_{i=0}^{t} \beta^{t-i} \theta_i$$

- We can see that the most recent observation θ has a weight of 1, the second last observation β, the third last β^2, etc.
- Since $0 < \beta < 1$, the multiplication term β^k goes exponentially down with the increase of k, so the older the observations, the less important they are.
- Finally, every sum term is multiplied by $(1 - \beta)$.

Exponential Weighted Averages

- In practice, the value for β is usually chosen close to 0.9.



Exponential Weighted Averages

- Exponential weighted average has a wide application scope in time series analysis.
- Additionally, it used in variations of gradient descent algorithm for convergence acceleration.
- One of the most popular of them is the Momentum optimizer in deep learning which removes unnecessary oscillations of an optimized function aligning it more precisely towards a local minimum.

Gradient Descent with Momentum (Optimizer)

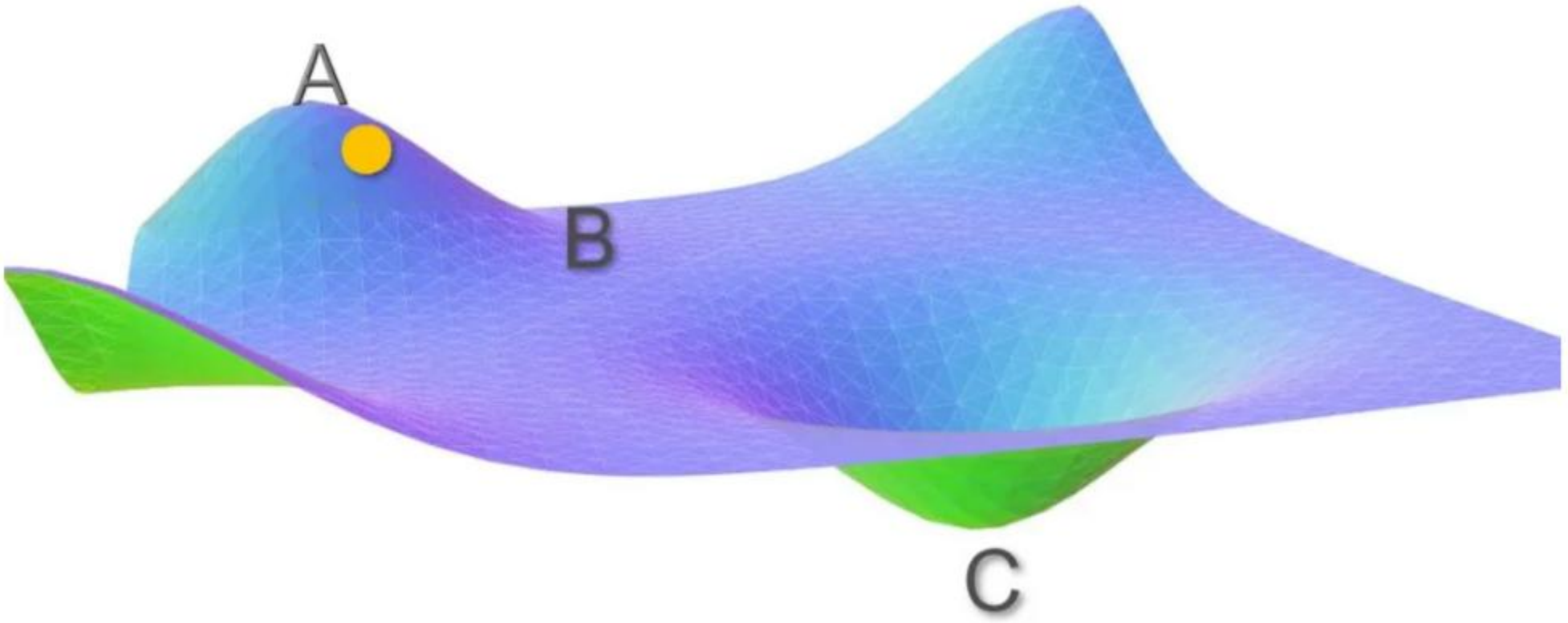
- The problem with gradient descent is that the weight update at a moment (t) is governed by the learning rate and gradient at that moment only.
- It doesn't take into account the past steps taken while traversing the cost space.

$$\Delta w(t) = -\eta \delta(t)$$

- It leads to the following problems.
- The gradient of the cost function at saddle points(plateau) is negligible or zero, which in turn leads to small or no weight updates.
- Hence, the network becomes stagnant, and learning stops
- The path followed by Gradient Descent is very jittery even when operating with mini-batch mode.

Gradient Descent with Momentum (Optimizer)

❑Consider the below cost surface.



Gradient Descent with Momentum

- Let's assume the initial weights of the network under consideration correspond to point A.
- With gradient descent, the Loss function decreases rapidly along the slope AB as the gradient along this slope is high.
- But as soon as it reaches point B the gradient becomes very low.
- The weight updates around B is very small. Even after many iterations, the cost moves very slowly before getting stuck at a point where the gradient eventually becomes zero.
- In this case, ideally, cost should have moved to the global minima point C, but because the gradient disappears at point B, we are stuck with a sub-optimal solution.

Gradient Descent with Momentum

❑**How can momentum fix this?**

❑Now, Imagine you have a ball rolling from point A.

❑The ball starts rolling down slowly and gathers some **momentum** across the slope AB.

❑When the ball reaches point B, it has accumulated enough momentum to push itself across the plateau region B and finally following slope BC to land at the global minima C.

Gradient Descent with Momentum

❑**How can this be used and applied to Gradient Descent?**

❑To account for the momentum, we can use a moving average over the past gradients.

❑In regions where the gradient is high like AB, weight updates will be large.

❑Thus, in a way we are gathering momentum by taking a moving average over these gradients.

❑But there is a problem with this method, it considers all the gradients over iterations with equal weightage.

❑The gradient at t=0 has equal weightage as that of the gradient at current iteration t.

❑We need to use some sort of weighted average of the past gradients such that the recent gradients are given more weightage.

❑This can be done by using an Exponential Moving Average(EMA).

❑An exponential moving average is a moving average that assigns a greater weight on the most recent values.

Gradient Descent with Momentum

- **How can this be used and applied to Gradient Descent?**
- The EMA for a series *Y* may be calculated recursively

$$s(t) = \begin{cases} Y(1) & t = 1 \\ \beta * s(t-1) + (1-\beta) * Y(t) & t > 1 \end{cases}$$

- Where
- The coefficient β represents the degree of weighting increase, a constant smoothing factor between 0 and 1. A lower β discounts older observations faster.
- *Y(t)* is the value at a period *t*.
- *S(t)* is the value of the EMA at any period *t*.

Gradient Descent with Momentum

❑**How can this be used and applied to Gradient Descent?**

❑In our case of a sequence of gradients, the new weight update equation at iteration t becomes

$$v(t) = \beta * v(t-1) + (1-\beta) * \delta(t)$$

❑Let's break it down.

❑$\boldsymbol{v}$**(t)**: is the new weight update done at iteration **t**

❑**β:** Momentum constant

❑δ**(t):** is the gradient at iteration **t**

Gradient Descent with Momentum

❑**How can this be used and applied to Gradient Descent?**

❑Assume the weight update at the zeroth iteration t=0 is zero

❑Think about the constant β and ignore the term (1-β) in the above equation.

$$v(0) = 0$$

$$v(1) = \beta * v(0) + (1-\beta) * \delta(1)$$

$$v(1) = (1-\beta) * \delta(1)$$

$$v(2) = \beta * v(1) + (1-\beta) * \delta(2)$$

$$v(2) = \beta * \{(1-\beta) * \delta(1)\} + (1-\beta) * \delta(2)$$

$$v(2) = (1-\beta)\{\beta * \delta(1) + \delta(2)\}$$

$$v(3) = \beta * v(2) + (1-\beta) * \delta(3)$$

$$v(3) = \beta * \{(1-\beta)\{\beta * \delta(1) + \delta(2)\}\} + (1-\beta) * \delta(3)$$

$$v(3) = (1-\beta)\{\beta^2 * \delta(1) + \beta * \delta(2) + \delta(3)\}$$

$$v(n) = (1-\beta)\sum_{t=1}^{n} \beta^{n-t}\delta(t)$$

Gradient Descent with Momentum

❑**what if β is 0.1?** At n=3; the gradient at t =3 will contribute 100% of its value, the gradient at t=2 will contribute 10% of its value, and gradient at t=1 will only contribute 1% of its value. here contribution from earlier gradients decreases rapidly.

❑**what if β is 0.9?** At n=3; the gradient at t =3 will contribute 100% of its value, t=2 will contribute 90% of its value, and gradient at t=1 will contribute 81% of its value.

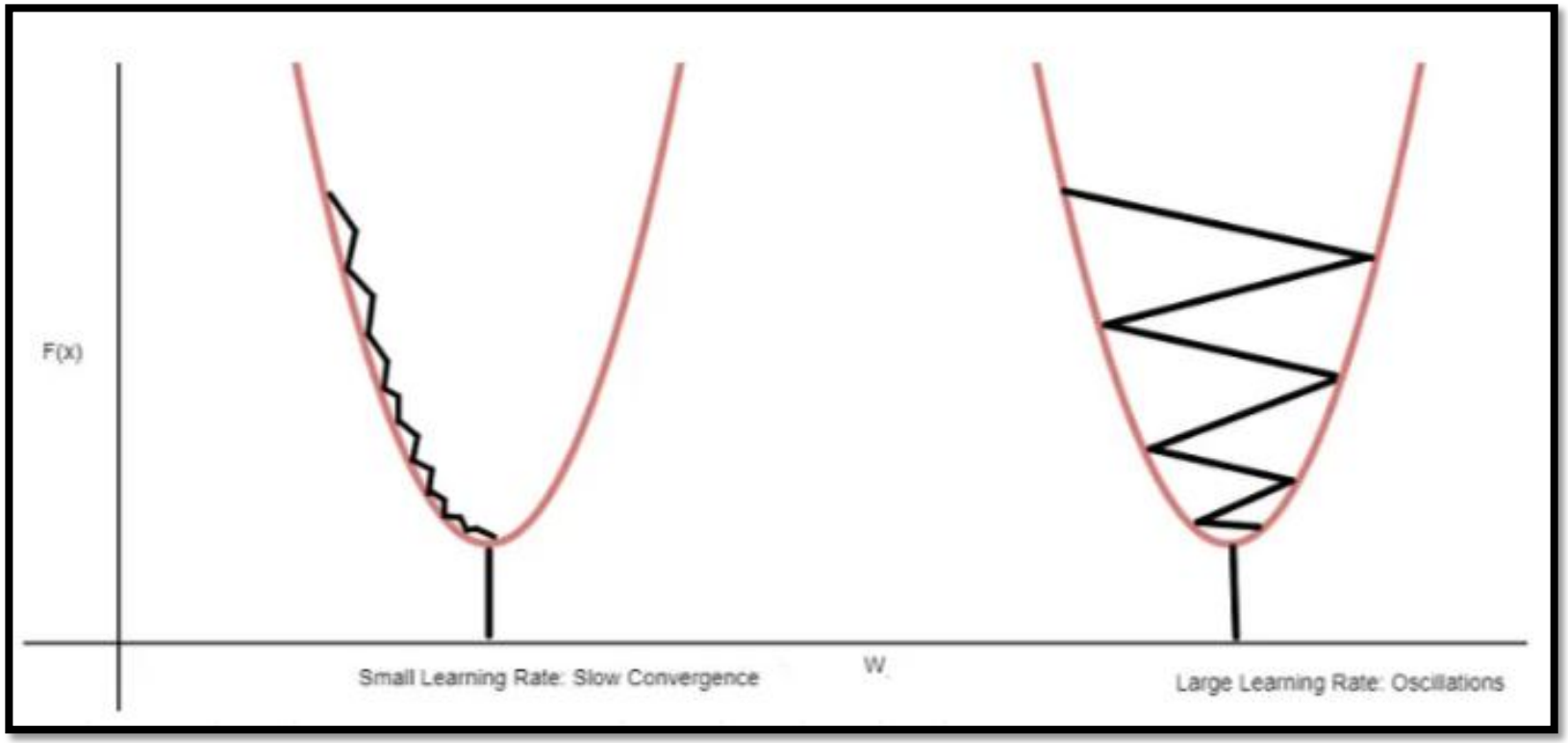
❑From above, we can deduce that higher β will accommodate more gradients from the past. Hence, generally, β is kept around 0.9 in most of the cases.

❑The actual contribution of each gradient in the weight update will be further subjected to the learning rate.

❑This addresses our first point where we said when the gradient at the current moment is negligible or zero the learning becomes zero. Using momentum with gradient descent, gradients from the past will push the cost further to move around a saddle point.

Gradient Descent with Momentum

- In the cost surface shown earlier let's zoom into point C.
- With gradient descent, if the learning rate is too small, the weights will be updated very slowly hence convergence takes a lot of time even when the gradient is high.
- This is shown in the left side image below.
- If the learning rate is too high cost oscillates around the minima as shown in the right side image below.



Gradient Descent with Momentum

❑**How does Momentum fix this?**

❑Let's look at the last summation equation of the **momentum** again.

❑**Case 1**: When all the past gradients have the same sign

The summation term will become large and we will take large steps while updating the weights. Along the curve BC, even if the learning rate is low, all the gradients along the curve will have the same direction(sign) thus increasing the **momentum** and accelerating the descent.

❑**Case 2**: when some of the gradients have +ve sign whereas others have –ve

The summation term will become small and weight updates will be small. If the learning rate is high, the gradient at each iteration around the valley C will alter its sign between +ve and -ve and after few oscillations, the sum of past gradients will become small. Thus, making small updates in the weights from there on and damping the oscillations.

Gradient Descent with Momentum

- By adding a momentum term in the gradient descent, gradients accumulated from past iterations will push the cost further to move around a saddle point even when the current gradient is negligible or zero.
- Even though momentum with gradient descent converges better and faster, it still doesn't resolve all the problems. First, the hyperparameter η (learning rate) has to be tuned manually. Second, in some cases, where, even if the learning rate is low, the momentum term and the current gradient can alone drive and cause oscillations.

RMSProp and Adam Optimization (Optimizer)

❑Gradient descent

❑Gradient descent is the simplest optimization algorithm which computes gradients of loss function with respect to model weights and updates them by using the following formula:

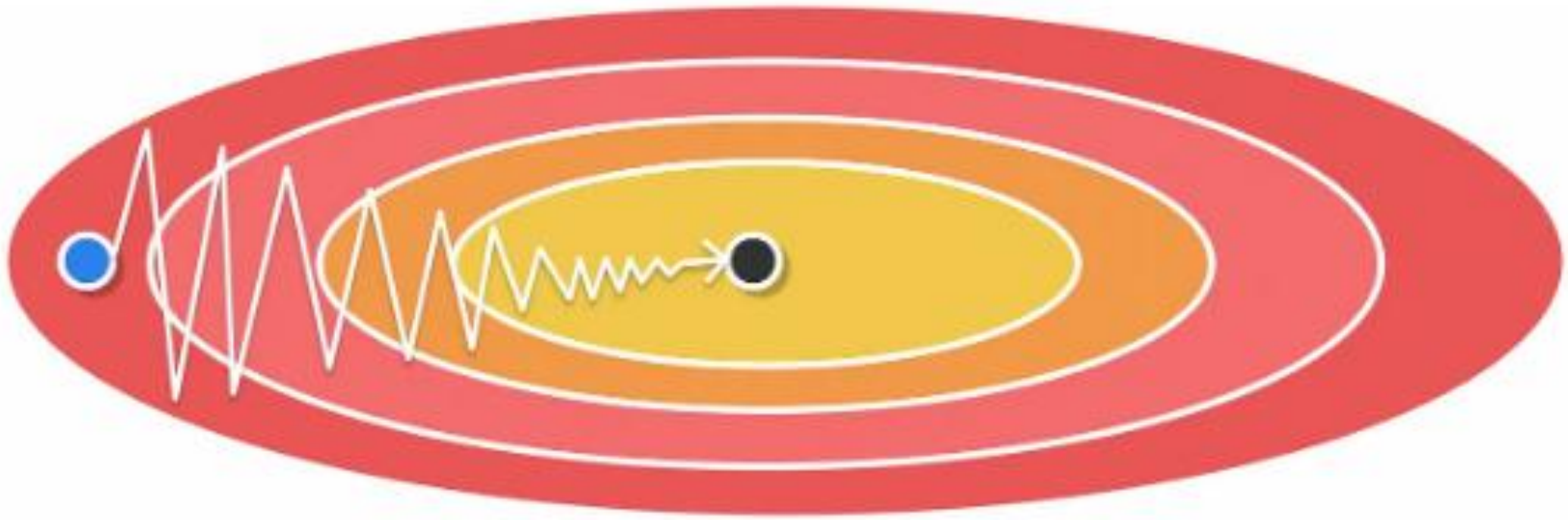
$$w_t = w_{t-1} - \alpha \cdot dw_t$$

❑Where **w** is the weight vector, dw is the gradient of w, α is the learning rate, **t** is the iteration number

RMSProp and Adam Optimization (Optimizer)

❑Gradient descent

❑To understand why gradient descent converges slowly, let us look at the example below of a ravine (A ravine is an area where the surface is much more steep in one dimension than in another) where a given function of two variables should be minimized.



RMSProp and Adam Optimization (Optimizer)

❑Gradient descent

❑From the image, we can see that the starting point and the local minima have different horizontal coordinates and are almost equal vertical coordinates.

❑Using gradient descent to find the local minima will likely make the loss function slowly oscillate towards vertical axes.

❑These bounces occur because gradient descent does not store any history about its previous gradients making gradient steps more undeterministic on each iteration.

❑This example can be generalized to a higher number of dimensions.

❑As a consequence, it would be risky to use a large learning rate as it could lead to disconvergence.

RMSProp and Adam Optimization (Optimizer)

❑Momentum

❑Based on the example above, it would be desirable to make a loss function performing larger steps in the horizontal direction and smaller steps in the vertical.

❑This way, the convergence would be much faster.

❑This effect is exactly achieved by Momentum.

❑Momentum uses a pair of equations at each iteration:

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t$$

$$w_t = w_{t-1} - \alpha v_t$$

❑The first formula uses an exponentially moving average for gradient values ***dw***.

❑Basically, it is done to store trend information about a set of previous gradient values.

❑The second equation performs the normal gradient descent update using the computed moving average value on the current iteration.

❑**α** is the learning rate of the algorithm.

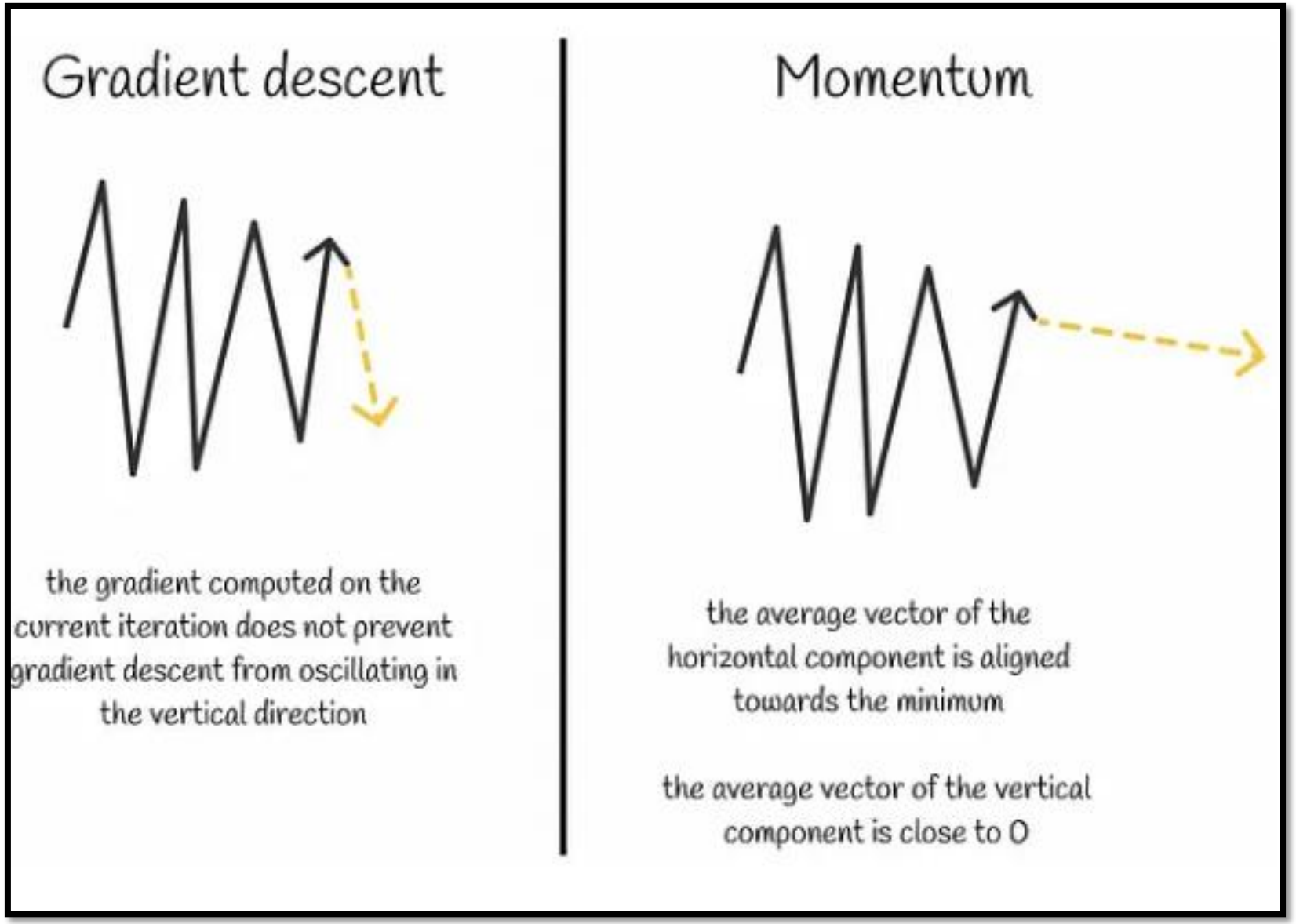
RMSProp and Adam Optimization (Optimizer)

❑Momentum

❑Momentum can be particularly useful for cases like the above.

❑Imagine we have computed gradients on every iteration like in the picture above.

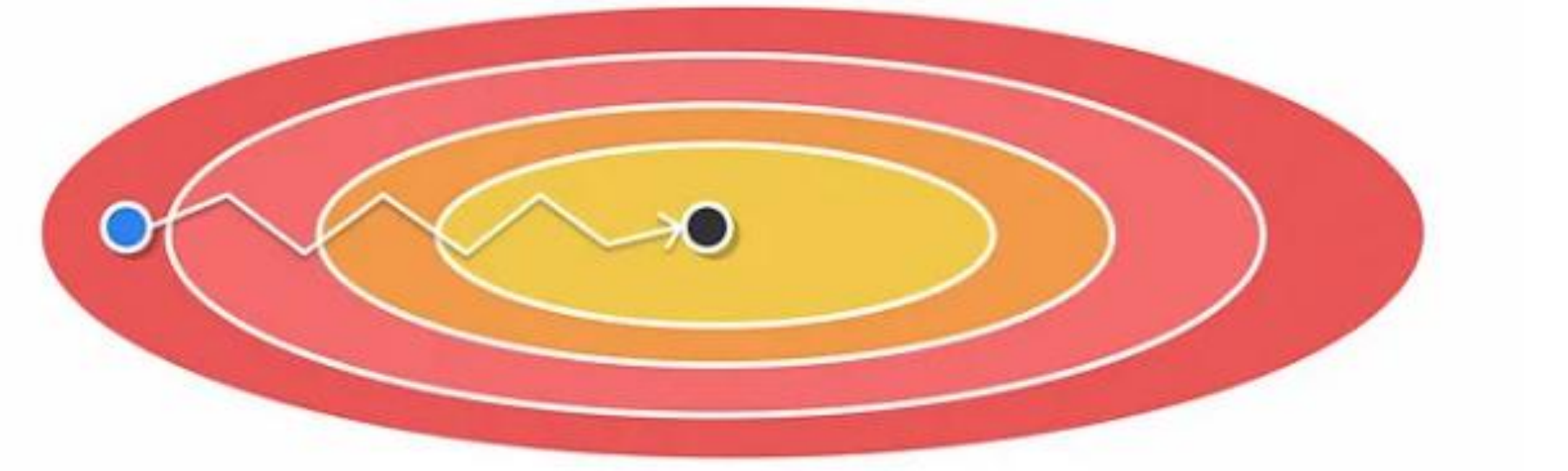
❑Instead of simply using them for updating weights, we take several past values and literaturally perform update in the averaged direction.



RMSProp and Adam Optimization (Optimizer)

❑Momentum

❑As a result, updates performed by Momentum might look like in the figure below.



❑In practice, Momentum usually converges much faster than gradient descent.

❑With Momentum, there are also fewer risks in using larger learning rates, thus accelerating the training process.

RMSProp and Adam Optimization (Optimizer)

❑RMSProp (Root Mean Square Propagation)

❑RMSProp was elaborated as an improvement over AdaGrad which tackles the issue of learning rate decay.

❑Similarly to AdaGrad, RMSProp uses a pair of equations for which the weight update is absolutely the same.

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

$$w_t = w_{t-1} - \frac{a}{\sqrt{v_t + \varepsilon}} dw_t$$

RMSProp and Adam Optimization (Optimizer)

❑RMSProp (Root Mean Square Propagation)

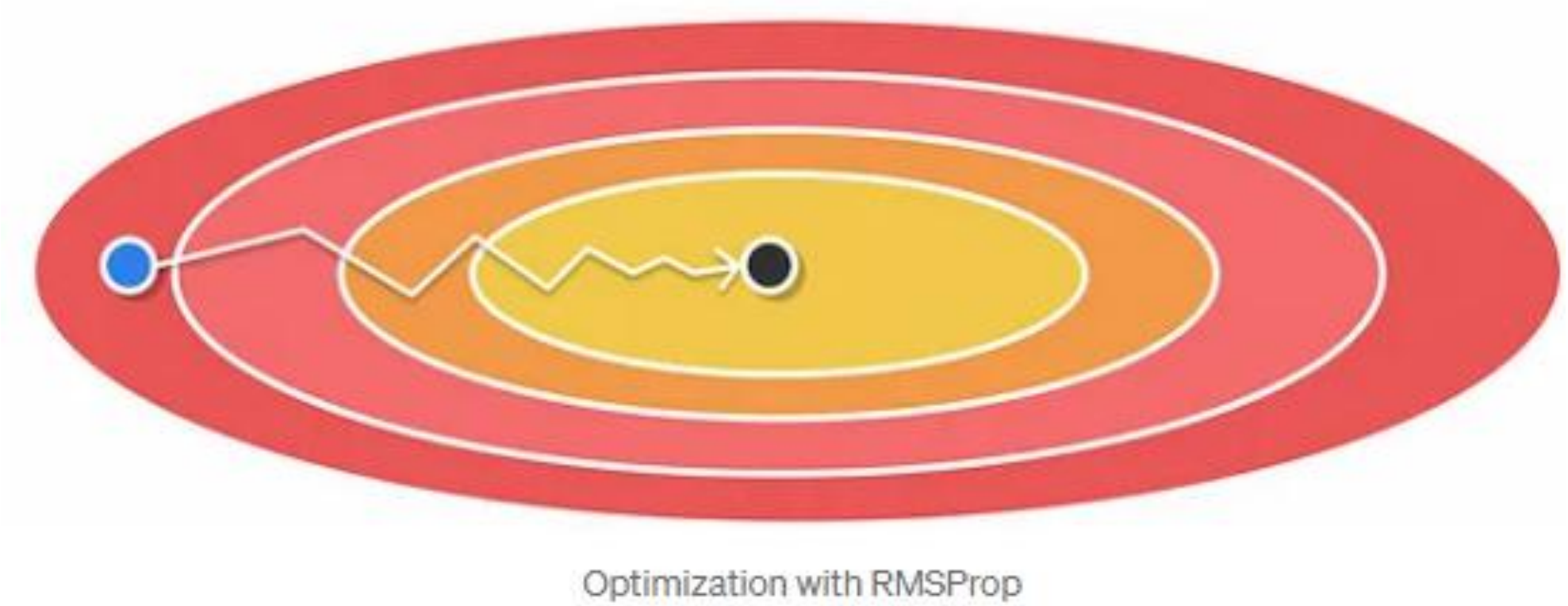
❑However, instead of storing a cumulated sum of squared gradients dw^2 for v, the exponentially moving average is calculated for squared gradients dw^2.

❑Experiments show that RMSProp generally converges faster than AdaGrad because, with the exponentially moving average, it puts more emphasis on recent gradient values rather than equally distributing importance between all gradients by simply accumulating them from the first iteration.

❑Furthermore, compared to AdaGrad, the learning rate in RMSProp does not always decay with the increase of iterations making it possible to adapt better in particular situations.

RMSProp and Adam Optimization (Optimizer)

❑RMSProp (Root Mean Square Propagation)



Optimization with RMSProp

RMSProp and Adam Optimization (Optimizer)

❑RMSProp (Root Mean Square Propagation)

❑**Why not to simply use a squared gradient for v instead of the exponentially moving average?**

❑It is known that the exponentially moving average distributes higher weights to recent gradient values.

❑This is one of the reasons why RMSProp adapts quickly.

❑But would not it be better if instead of the moving average we only took into account the last square gradient at every iteration ($v = dw^2$)?

❑As it turns out, the update equation would transform in the following manner:

$$v_t = dw_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t} + \varepsilon} dw_t \longrightarrow w_t = w_{t-1} - \alpha \cdot \text{sign}(dw_t)$$

Transformation of RMSProp equations when using a squared gradient instead of the exponentially moving average

RMSProp and Adam Optimization (Optimizer)

❑RMSProp (Root Mean Square Propagation)

❑As we can see, the resulting formula looks very similar to the one used in the gradient descent.

❑However, instead of using a normal gradient value for the update, we are now using the sign of the gradient:

- **If $dw > 0$, then a weight w is decreased by α.**
- **If $dw < 0$, then a weight w is increased by α.**

❑To sum it up, if v = dw^2, then model weights can only be changed by ±α.

❑Though this approach works sometimes, it is still not flexible the algorithm becomes extremely sensitive to the choice of α and absolute magnitudes of gradient are ignored which can make the method tremendously slow to converge.

❑A positive aspect about this algorithm is the fact only a single bit is required to store signs of gradietns which can be handy in distributed computations with strict memory requirements.

RMSProp and Adam Optimization (Optimizer)

❑Adam (Adaptive Moment Estimation)

❑For the moment, Adam is the most famous optimization algorithm in deep learning.

❑At a high level, Adam combines Momentum and RMSProp algorithms.

❑To achieve it, it simply keeps track of the exponentially moving averages for computed gradients and squared gradients respectively.

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) dw_t \xrightarrow{\text{bias correction}} \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) dw_t^2 \xrightarrow{\text{bias correction}} \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \frac{\alpha \hat{v}_t}{\sqrt{\hat{s}_t} + \varepsilon} dw_t$$

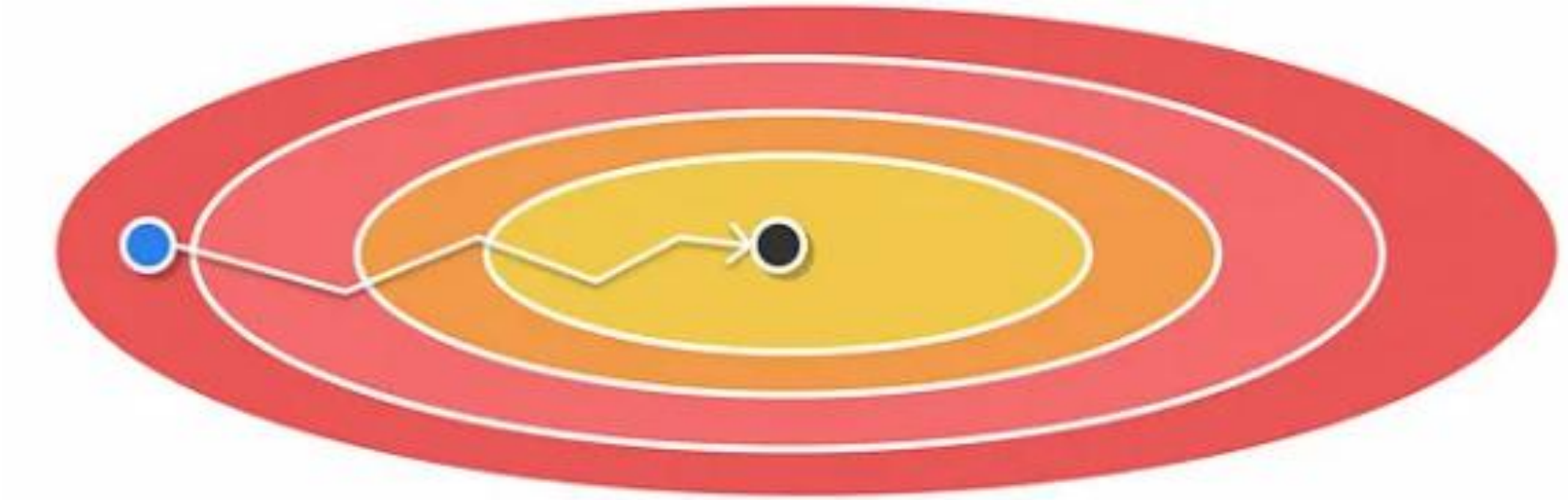
Adam equations

RMSProp and Adam Optimization (Optimizer)

❑Adam (Adaptive Moment Estimation)

❑Furthermore, it is possible to use bias correction for moving averages for a more precise approximation of gradient trend during the first several iterations.

❑The experiments show that Adam adapts well to almost any type of neural network architecture taking the advantages of both Momentum and RMSProp.



Optimization with Adam

RMSProp and Adam Optimization (Optimizer)

- We have looked at different optimization algorithms in neural networks.
- Considered as a combination of Momentum and RMSProp, Adam is the most superior of them which robustly adapts to large datasets and deep networks.
- Moreover, it has a straightforward implementation and little memory requirements making it a preferable choice in the majority of situations.

Hyperparameter Tuning (Regularization)

- While developing deep learning models you must have encountered a situation in which the training accuracy of the model is high but the validation accuracy or the testing accuracy is low.
- This is the case which is popularly known as overfitting in the domain of deep learning.
- Also, this is the last thing a deep learning practitioner would like to have in his model.
- Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function, discouraging the model from assigning too much importance to individual features or coefficients.

Hyperparameter Tuning (Regularization)

❑Let's explore some more detailed explanations about the role of Regularization:

1. **Complexity Control**: Regularization helps control model complexity by preventing overfitting to training data, resulting in better generalization to new data.
2. **Preventing Overfitting**: One way to prevent overfitting is to use regularization, which penalizes large coefficients and constrains their magnitudes, thereby preventing a model from becoming overly complex and memorizing the training data instead of learning its underlying patterns.
3. **Balancing Bias and Variance**: Regularization can help balance the trade-off between model bias (underfitting) and model variance (overfitting) in machine learning, which leads to improved performance.
4. **Feature Selection**: Some regularization methods, such as L1 regularization (Lasso), promote sparse solutions that drive some feature coefficients to zero. This automatically selects important features while excluding less important ones.
5. **Handling Multicollinearity**: When features are highly correlated (multicollinearity), regularization can stabilize the model by reducing coefficient sensitivity to small data changes.
6. **Generalization**: Regularized models learn underlying patterns of data for better generalization to new data, instead of memorizing specific examples.

Hyperparameter Tuning (Regularization)

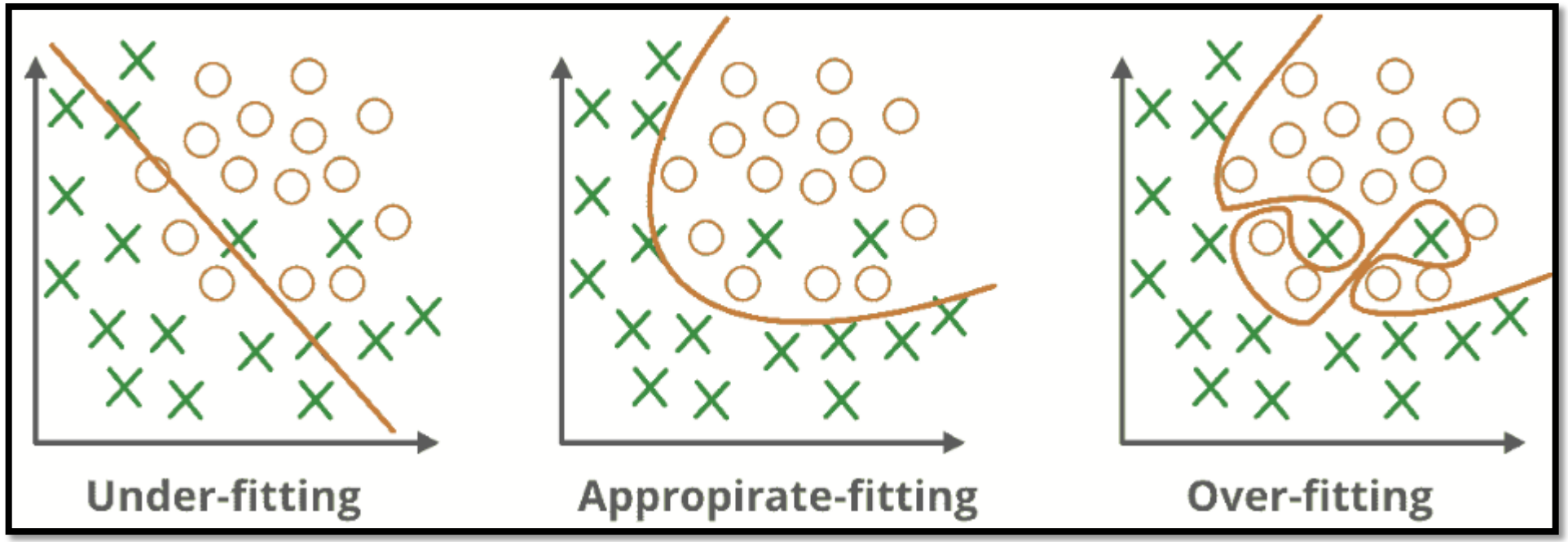
❑What are Overfitting and Underfitting?

❑**Overfitting** is a phenomenon that occurs when a Machine Learning model is constrained to the training set and not able to perform well on unseen data. That is when our model learns the noise in the training data as well. This is the case when our model memorizes the training data instead of learning the patterns in it.

❑**Underfitting** on the other hand is the case when our model is not able to learn even the basic patterns available in the dataset. In the case of the underfitting model is unable to perform well even on the training data hence we cannot expect it to perform well on the validation data. This is the case when we are supposed to increase the complexity of the model or add more features to the feature set.

Hyperparameter Tuning (Regularization)

❑What are Overfitting and Underfitting?



Hyperparameter Tuning (Regularization)

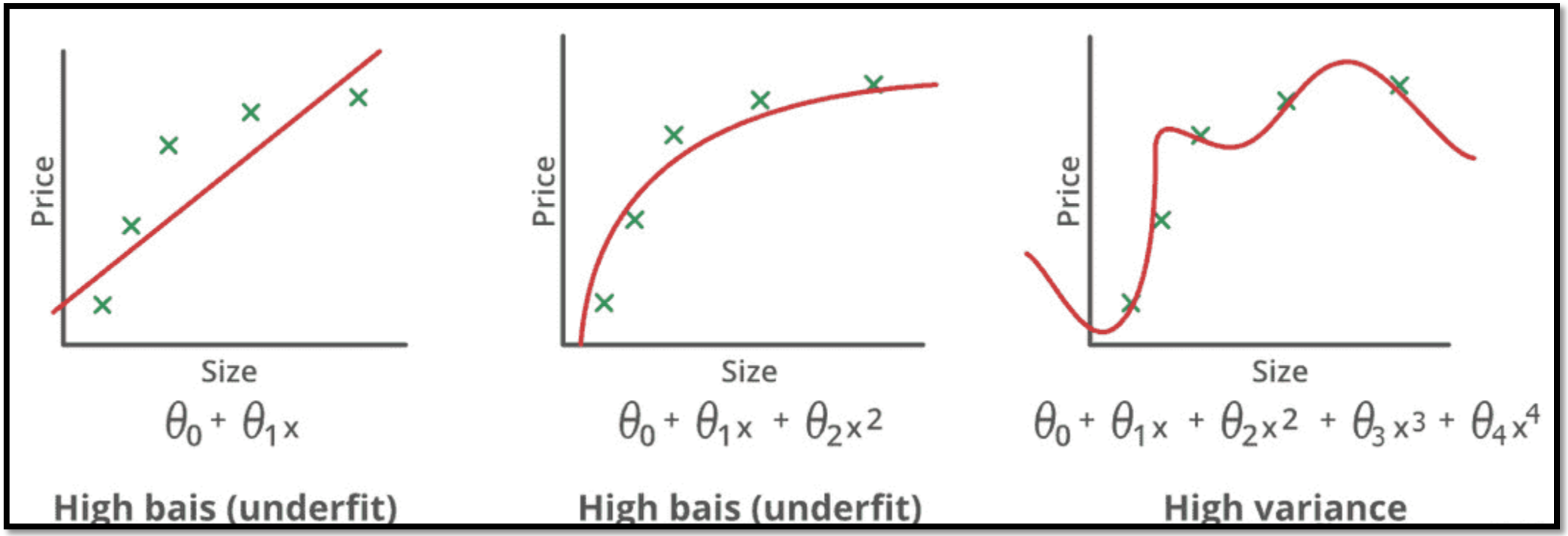
❑What are Bias and Variance?

❑**Bias** refers to the errors which occur when we try to fit a statistical model on real-world data which does not fit perfectly well on some mathematical model. If we use a way too simplistic a model to fit the data then we are more probably face the situation of **High Bias** which refers to the case when the model is unable to learn the patterns in the data at hand and hence performs poorly.

❑**Variance** implies the error value that occurs when we try to make predictions by using data that is not previously seen by the model. There is a situation known as **high variance** that occurs when the model learns noise that is present in the data.

Hyperparameter Tuning (Regularization)

❑What are Bias and Variance?



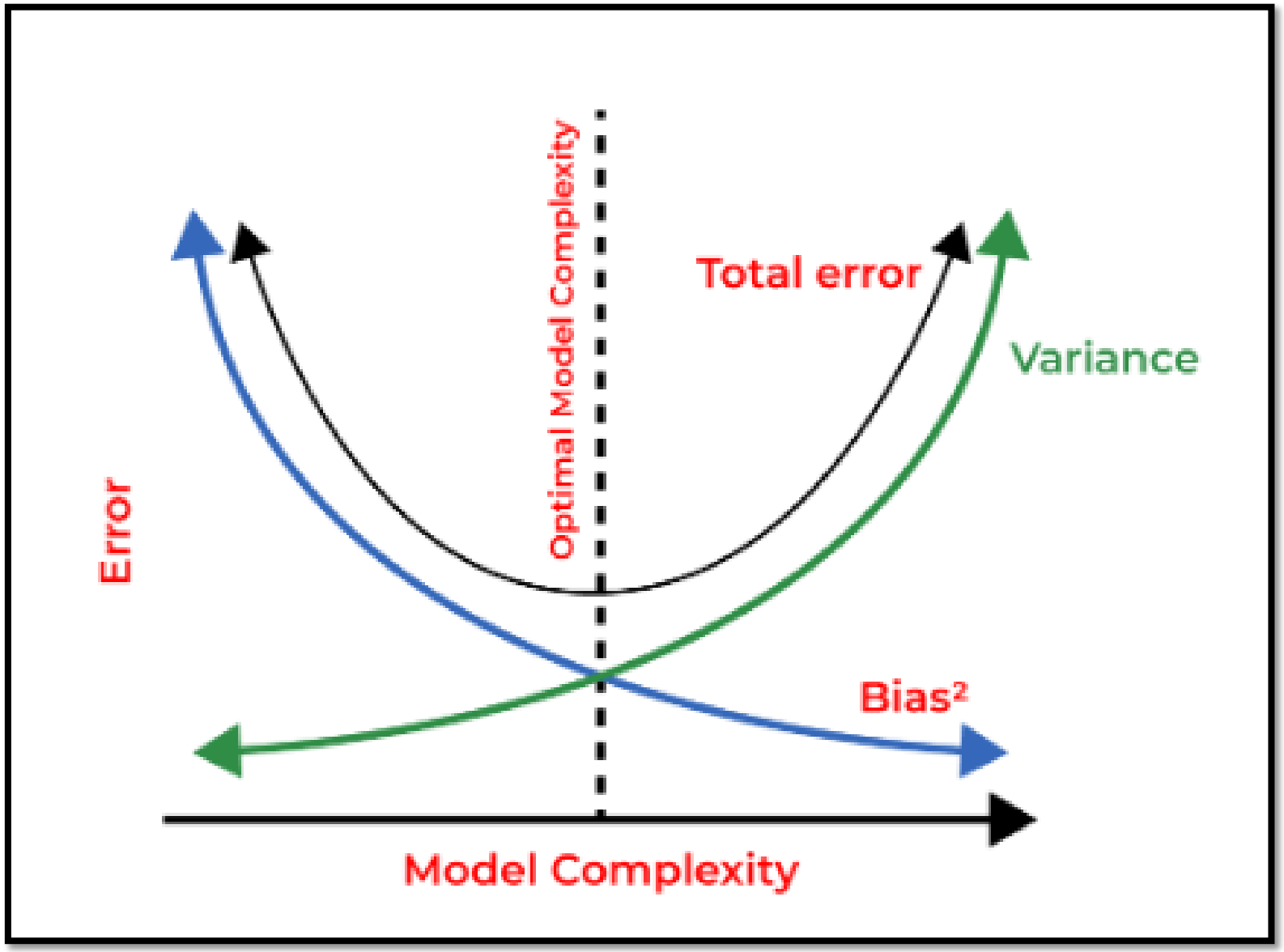
Hyperparameter Tuning (Regularization)

❑Bias Variance tradeoff

- ❑The bias-variance tradeoff is a fundamental concept in machine learning.
- ❑It refers to the balance between bias and variance, which affect predictive model performance.
- ❑Finding the right tradeoff is crucial for creating models that generalize well to new data.
- ❑The bias-variance tradeoff demonstrates the inverse relationship between bias and variance.
- ❑When one decreases, the other tends to increase, and vice versa.
- ❑Finding the right balance is crucial.
- ❑An overly simple model with high bias won't capture the underlying patterns, while an overly complex model with high variance will fit the noise in the data.

Hyperparameter Tuning (Regularization)

❑Bias Variance tradeoff



Hyperparameter Tuning (Regularization)

❑Regularization in Machine Learning

❑Regularization is a technique used to reduce errors by fitting the function appropriately on the given training set and avoiding overfitting.

❑The commonly used regularization techniques are :

❑Lasso Regularization – L1 Regularization

❑Ridge Regularization – L2 Regularization

❑Elastic Net Regularization – L1 and L2 Regularization

Hyperparameter Tuning (Regularization)

❑Regularization in Machine Learning

1. **L1 Regularization (Lasso)**:

   - Objective Function: The objective function in L1 regularization is augmented with a penalty term that is the sum of the absolute values of the coefficients multiplied by a regularization parameter (lambda).
   - Mathematical Expression: $\min_{D,X} \frac{1}{2}\|Y - DX\|_F^2 + \lambda\|X\|_1$
   - Usage: L1 regularization encourages sparsity by penalizing large coefficients, effectively promoting solutions where many coefficients are zero, resulting in a sparse representation.

Hyperparameter Tuning (Regularization)

❑Regularization in Machine Learning

2. **L2 Regularization (Ridge)**:

- Objective Function: The objective function in L2 regularization is augmented with a penalty term that is the sum of the squares of the coefficients multiplied by a regularization parameter (lambda).
- Mathematical Expression: $\min_{D,X} \frac{1}{2}\|Y - DX\|_F^2 + \frac{\lambda}{2}\|X\|_F^2$
- Usage: L2 regularization penalizes large coefficients, but unlike L1 regularization, it doesn't encourage sparsity as strongly. It tends to distribute the weight more evenly across all coefficients.

Hyperparameter Tuning (Regularization)

❑Regularization in Machine Learning

3. **Elastic Net Regularization**:

- Objective Function: Elastic Net combines both L1 and L2 penalties in the objective function.
- Mathematical Expression: $\min_{D,X} \frac{1}{2}\|Y - DX\|_F^2 + \lambda_1\|X\|_1 + \frac{\lambda_2}{2}\|X\|_F^2$
- Usage: Elastic Net provides a balance between L1 and L2 regularization, allowing for both feature selection and coefficient shrinkage. It can handle correlated features better than Lasso alone.

Hyperparameter Tuning (Regularization) Experiential Learning

- Students need to study and understand the “Dropout Regularization Function”

Batch Normalization

- Training Deep Neural Networks is a difficult task that involves several problems to tackle.
- Despite their huge potential, they can be slow and be prone to overfitting.
- Thus, studies on methods to solve these problems are constant in Deep Learning research.
- **Batch Normalization** – commonly abbreviated as Batch Norm – is one of these methods.
- Currently, it is a widely used technique in the field of Deep Learning.
- It improves the learning speed of Neural Networks and provides regularization, avoiding overfitting.
- But why is it so important? How does it work? Furthermore, how can it be applied to non-regular networks such as Convolutional Neural Networks?

Batch Normalization

- **Normalization**
- To fully understand how Batch Norm works and why it is important, let's start by talking about normalization.
- **Normalization is a pre-processing technique used to standardize data.**
- In other words, having different sources of data inside the same range.
- Not normalizing the data before training can cause problems in our network, making it drastically harder to train and decrease its learning speed.
- For example, imagine we have a car rental service.
- Firstly, we want to predict a fair price for each car based on competitors' data.
- We have two features per car: the age in years and the total amount of kilometers it has been driven for.
- These can have very different ranges, ranging from 0 to 30 years, while distance could go from 0 up to hundreds of thousands of kilometers.
- We don't want features to have these differences in ranges, as the value with the higher range might bias our models into giving them inflated importance.

Batch Normalization

❑Normalization

There are two main methods to normalize our data. The most straightforward method is to scale it to a range from 0 to 1:

$$x_{normalized} = \frac{x - m}{x_{max} - x_{min}}$$

x the data point to normalize, m the mean of the data set, x_{max} the highest value, and x_{min} the lowest value. This technique is generally used in the inputs of the data. The non-normalized data points with wide ranges can cause instability in Neural Networks. The relatively large inputs can cascade down to the layers, causing problems such as exploding gradients.

The other technique used to normalize data is forcing the data points to have a mean of 0 and a standard deviation of 1, using the following formula:

$$x_{normalized} = \frac{x - m}{s}$$

being x the data point to normalize, m the mean of the data set, and s the standard deviation of the data set. Now, each data point mimics a standard normal distribution. Having all the features on this scale, none of them will have a bias, and therefore, our models will learn better.

In Batch Norm, we use this last technique to normalize batches of data inside the network itself.

Batch Normalization

❑Batch Normalization

❑**Batch Norm is a normalization technique done between the layers of a Neural Network instead of in the raw data.**

❑It is done along mini-batches instead of the full data set.

❑It serves to speed up training and use higher learning rates, making learning easier.

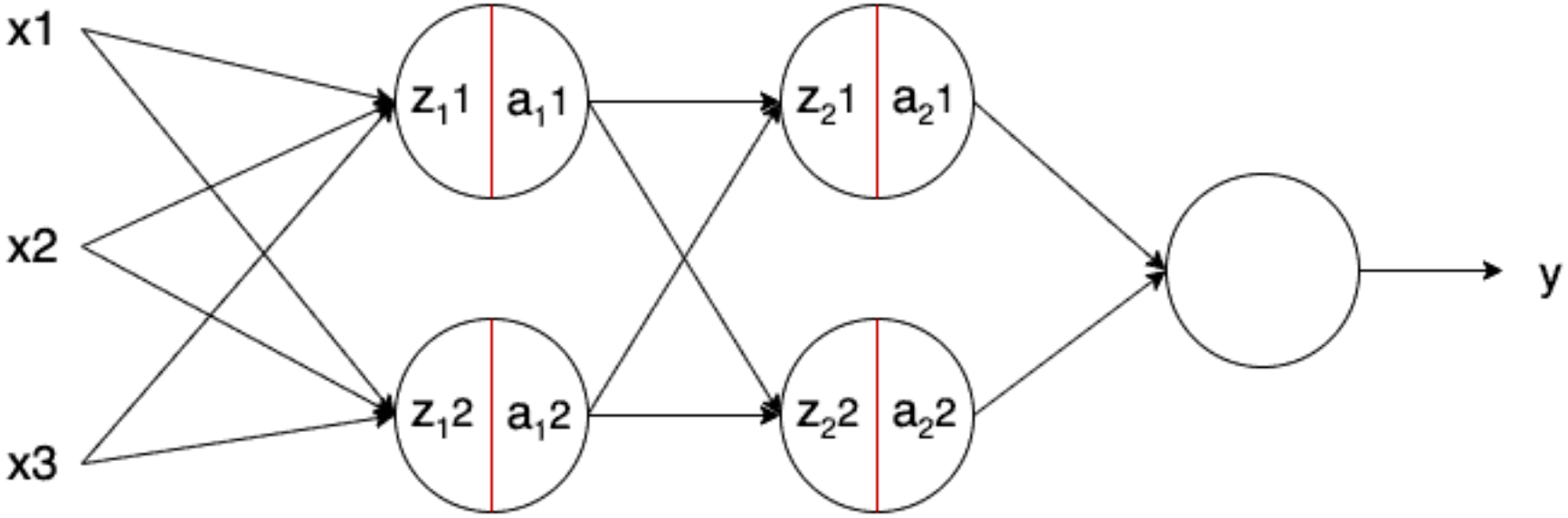
❑Following the technique explained in the previous section, we can define the normalization formula of Batch Norm as:

$$z^N = \left(\frac{z - m_z}{s_z} \right)$$

being m_z the mean of the neurons' output and s_z the standard deviation of the neurons' output.

❑Batch Normalization

In the following image, we can see a regular feed-forward Neural Network: x_i are the inputs, z the output of the neurons, a the output of the activation functions, and y the output of the network:



Batch Norm – in the image represented with a red line – is applied to the neurons' output just before applying the activation function. Usually, a neuron without Batch Norm would be computed as follows:

$$z = g(w, x) + b; \qquad a = f(z)$$

❑Batch Normalization

being $g()$ the linear transformation of the neuron, w the weights of the neuron, b the bias of the neurons, and $f()$ the activation function. The model learns the parameters w and b. Adding Batch Norm, it looks as:

$$z = g(w, x); \qquad z^N = \left(\frac{z - m_z}{s_z}\right) \cdot \gamma + \beta; \qquad a = f(z^N)$$

being z^N the output of Batch Norm, m_z the mean of the neurons' output, s_z the standard deviation of the output of the neurons, and γ and β learning parameters of Batch Norm. Note that the bias of the neurons (b) is removed. This is because as we subtract the mean m_z, any constant over the values of z – such as b – can be ignored as it will be subtracted by itself.

The parameters β and γ shift the mean and standard deviation, respectively. Thus, the outputs of Batch Norm over a layer results in a distribution with a mean β and a standard deviation of γ. These values are learned over epochs and the other learning parameters, such as the weights of the neurons, aiming to decrease the loss of the model.

Batch Normalization

- **Normalization**
- To fully understand how Batch Norm works and why it is important, let's start by talking about normalization.
- **Normalization is a pre-processing technique used to standardize data.**
- In other words, having different sources of data inside the same range.
- Not normalizing the data before training can cause problems in our network, making it drastically harder to train and decrease its learning speed.
- For example, imagine we have a car rental service.
- Firstly, we want to predict a fair price for each car based on competitors' data.
- We have two features per car: the age in years and the total amount of kilometers it has been driven for.
- These can have very different ranges, ranging from 0 to 30 years, while distance could go from 0 up to hundreds of thousands of kilometers.
- We don't want features to have these differences in ranges, as the value with the higher range might bias our models into giving them inflated importance.

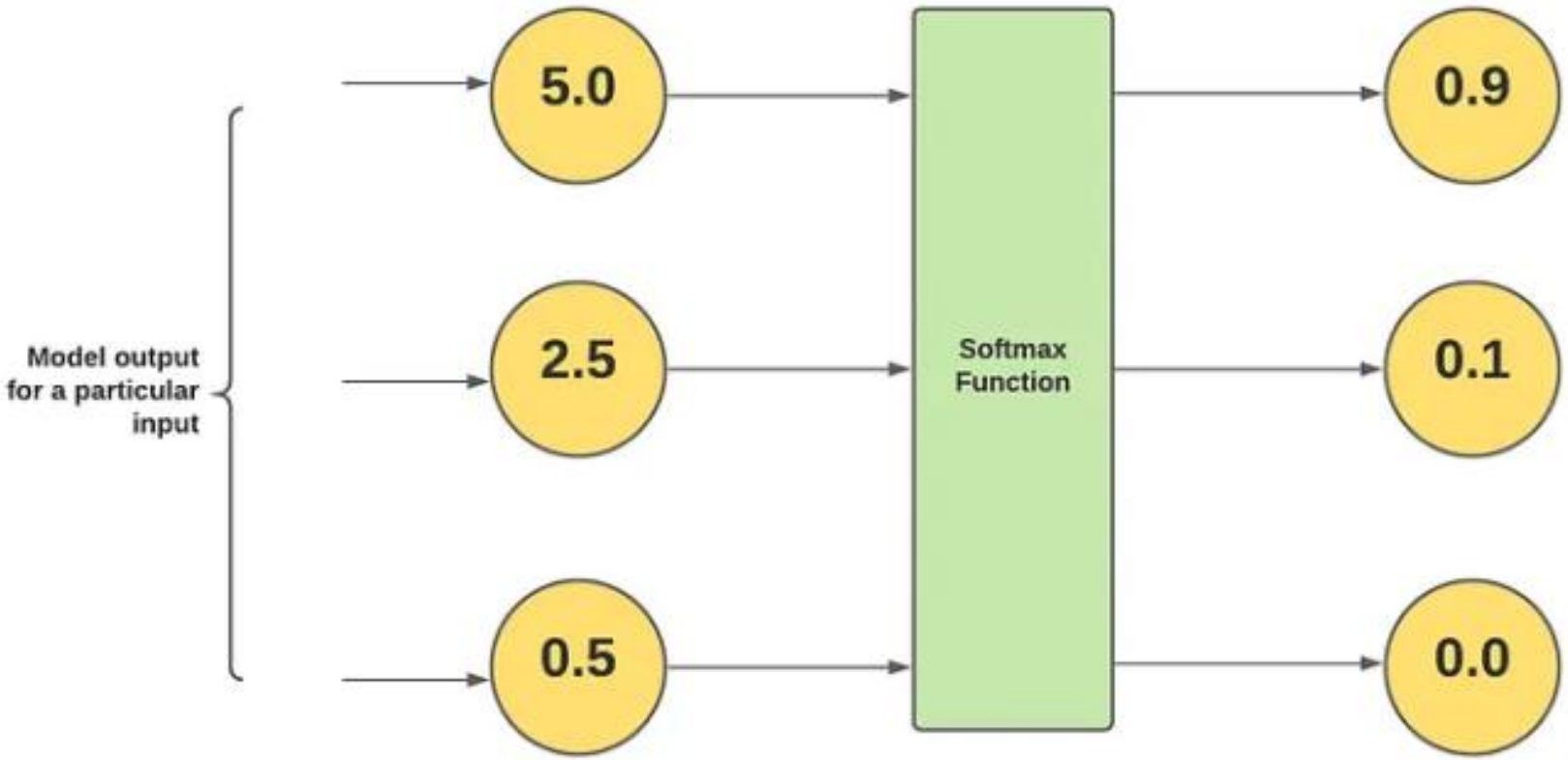
Softmax Regression

- Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes.
- In logistic regression we usually assume that the labels are binary i.e., $y(i) \in \{0,1\}$.
- The classifier distingushes between instance belongs to which class.
- While Softmax regression allows us to handle $y(i) \in \{1,\ldots,K\}$ where K is the number of classes.
- When we give an instance x, to the Softmax Regression classifier, it first computes a score $sk(x)$ for each class k, then estimates the probability of each class by applying the softmax function to the scores.
- The equation to compute softmax score $sk(x)$ for class k is as follows:

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Softmax score for class k

Softmax Regression



Softmax Regression

Softmax Regression

- After computing all the score for all classes for the instance *x*, we estimate the probability, *pk*, that the instance belongs to the class *k*, by applying softmax function on the scores.

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Softmax function

- Here *k* is the number of classes.
- *s(x)* is a vector containing the scores of each class for the instance *x.*
- *σ(s(x))k* is the estimated probability that the instance x belongs to class *k* for the scores of each class for that instance.

Softmax Regression

- From above equation we notice, the softmax function computes the exponential of every score, then normalizes them by dividing with the sum of all the exponentials.
- Just like the Logistic Regression classifier, the Softmax Regression also predicts the class with the highest estimated probability i.e., the class with the highest score.
- In softmax regression sum of all probabilities is equal to 1.

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\left(\boldsymbol{\theta}^{(k)}\right)^T \mathbf{x}\right)$$

Softmax regression classifier prediction

- The argmax operator returns the value of a variable that maximizes a function. In this equation, it returns the value of *k* that maximizes the estimated probability *σ(s(x))k*.

Softmax Regression

❑Cost Function

❑In training the objective is to have a model that estimates high probability for the target class and low probability for the other classes.

❑Keeping this objective in mind we are going to minimize the cost function, also called as c**ross entropy.**

❑Cross entropy cost function penalizes the model when it estimates a low probability for a target class.

❑Cross entropy is a measure of how well a set of estimated class probabilities match the target classes. So, we can say that it measures the performance of the model.

❑The softmax cost function is similar to logistic regression, except that we now sum over the K different possible values of the class label.

❑The equation of Cost function is as follows:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

Cross Entropy Cost Function

Softmax Regression

❑Cost Function

❑yk(i) is the target probability that says whether the instance i belongs to class k or not.

❑The probability will be equal to 1 or 0 depending on which class the instance belongs.

❑When there are just two classes (K = 2), this cost function is same as logistic regression cost function.

❑We cannot solve for the minimum of J(θ) analytically, and thus as usual we'll resort to an iterative optimization algorithm, Gradient Descent.

❑We take derivatives on softmax regression equation that we got after applying softmax function to the scores, to calculate probabilities Pk.

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

Softmax Regression

Softmax Regression

❑Cost Function

❑After taking derivatives on above equation, the gradient vector of the cost function with regards to θ(k) is given by,

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

Cross Entropy Gradient Vector for class k

❑∇θ(k)J(θ) is a vector, so that its jth element is the partial derivative of J(θ) with respect to the jth element of θ(k).

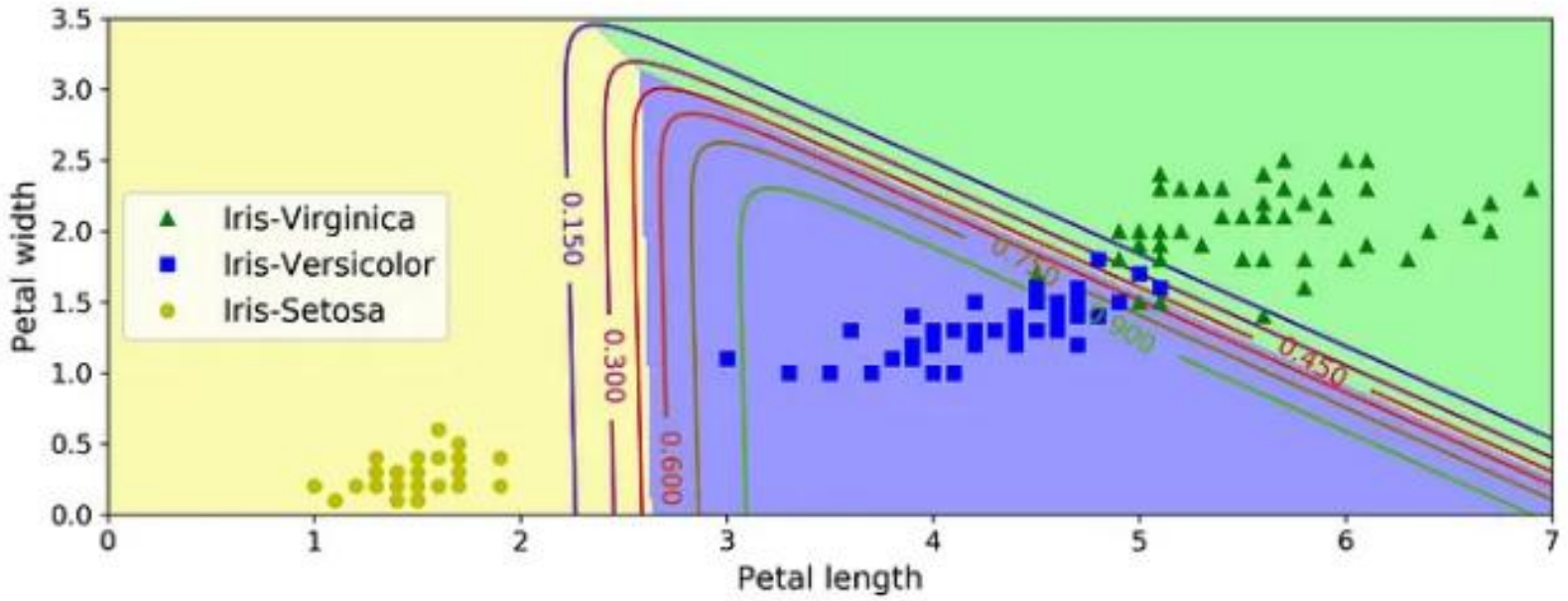
❑This way we can compute the gradient vector for every class, then use Gradient Descent to find the parameter matrix Θ that minimizes the cost function J(θ).

Softmax Regression

❑Decision Boundary

❑Let's consider the IRIS dataset.

❑When we use Softmax Regression to classify the iris flowers into all three classes, we get the following classification.



Softmax Regression decision boundaries

Softmax Regression

❑Decision Boundary

❑We can observe the resulting decision boundaries for three classes, represented by the background colors.

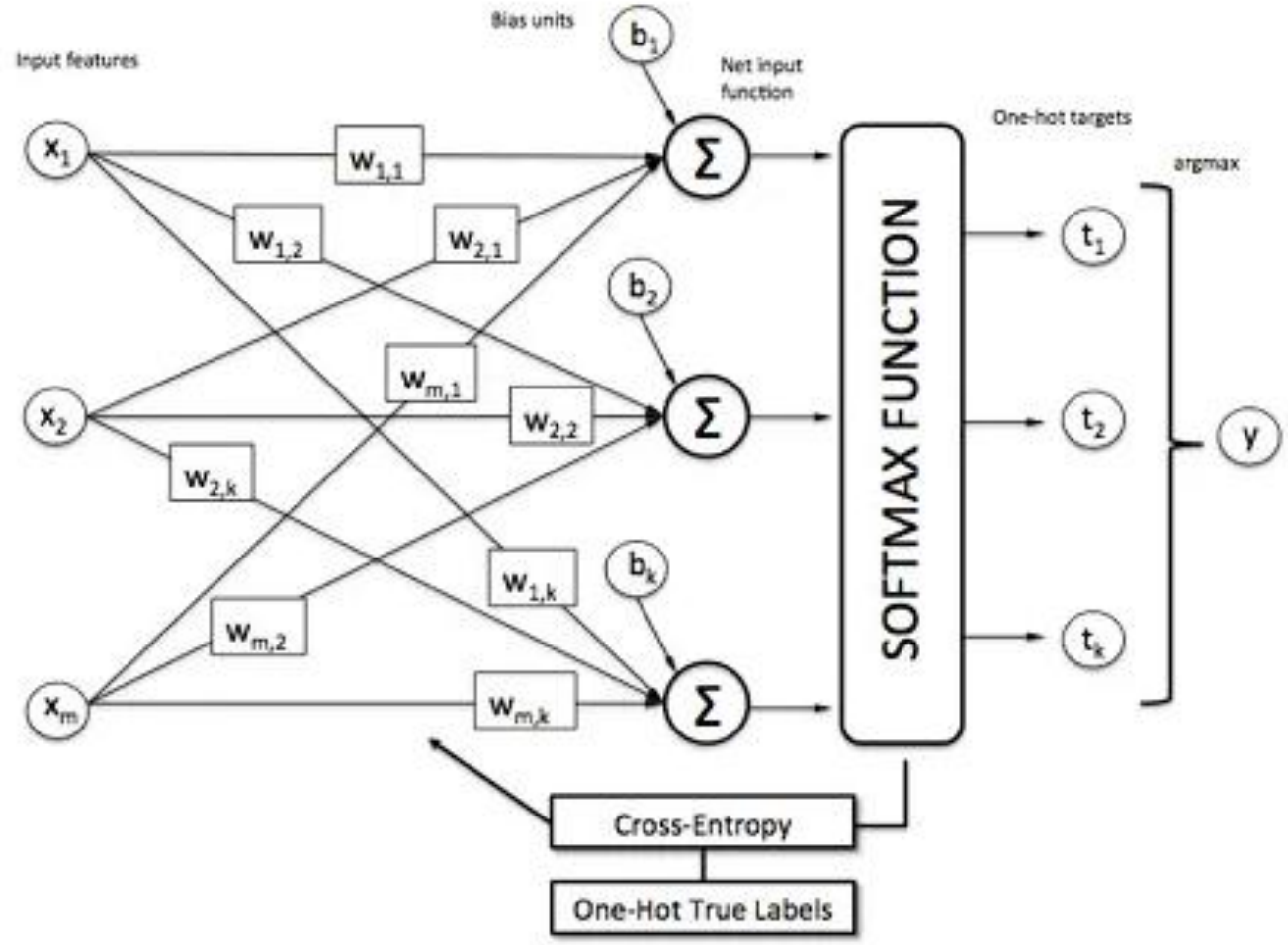
❑The decision boundaries between any two classes are linear. The probabilities for the Iris-Versicolor class, represented by the curved lines (e.g., the line labeled with 0.450 represents the 45% probability boundary).

❑Notice that the model can predict a class that has an estimated probability below 50%.

❑For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

Softmax Classifier

- Softmax classifier which allows us to find the predictions for multi-label outputs.



Softmax classifier

Softmax Classifier

- Generally, as seen in the above picture softmax function is added at the end of the output since it is the place where are the nodes meet finally and thus they can be classified.
- Here, X is the input of all the models and the layers between X and Y are the hidden layers and the data is passed from X to all the layers and Received by Y.
- Suppose, we have 10 classes and we predict for which class the given input belongs to.
- So for this what we do is allot each class with a particular predicted output.
- Which means that we have 10 outputs corresponding to 10 different class and predict the class by the highest probability it has.

Note for Students

- This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.