ANNs Training: Loss Functions & Optimizers

تدريب وتعليم الشبكات العصبية الاصطناعية

Eng. Mustafa Othman
Data Scientist & Analyst

Today's Outline:

- **Artificial Neural Networks (ANNs) Overview**
  - Building, Training & Optimizing ANNs
- **Training & Optimizing an ANN:**
  - **Loss Functions**
    - Regression
    - Classification
  - **Optimizers**
    - Gradient Descent
    - Backpropagation
  - **Demo: Training & Optimizing an ANN using Python**

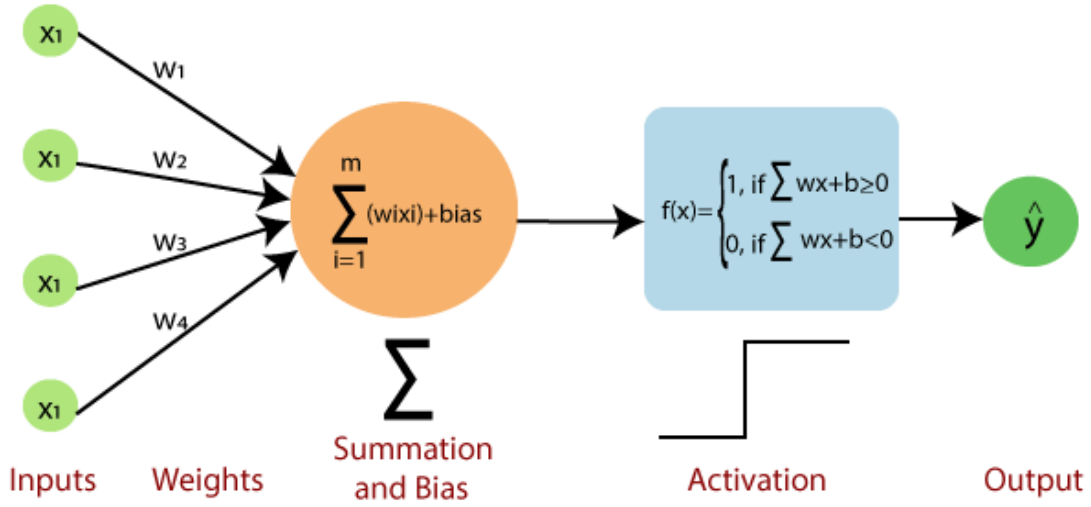
Artificial Neural Networks (ANNs) Overview

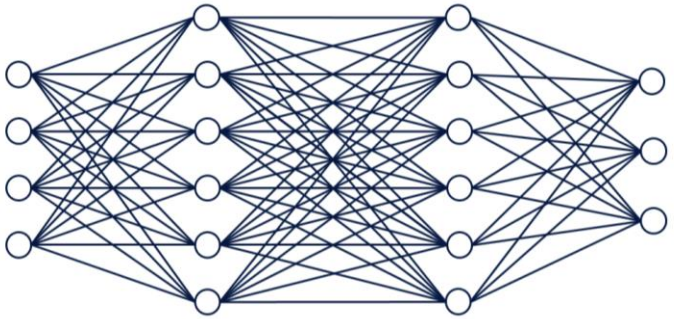
"We are all now connected by the Internet, like neurons in a giant brain." ~ Stephen Hawking

Eng. Mustafa Othman
Data Scientist & Analyst

Artificial Neural Networks Overview (1)
(Building ANNs)

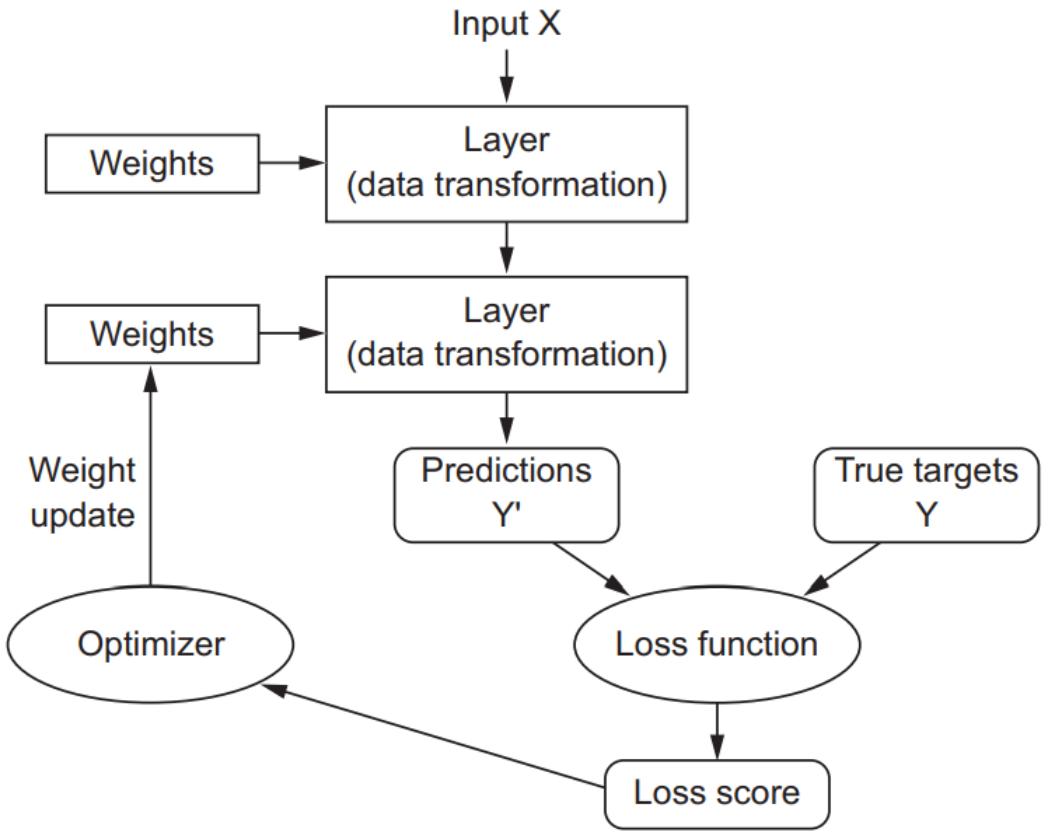
- **Parameters:**
  - Weights (**w**)
  - Bias (**b**)
- **Activation Functions:**
  - Sigmoid
  - ReLU
  - Linear
- **Layers:**
  - Input Layer
  - Hidden Layer(s)
  - Output Layer





Artificial Neural Networks Overview (2)
(Training & Optimizing ANNs)

- **Forward Propagation**
- **Backward Propagation**
  - Updating Weights
- **Loss Functions:**
  - Regression (Mean Squared Error (**MSE**))
  - Classification (**Cross-Entropy**)
- **Optimizers:**
  - SGD
  - Adam
- **Optimizer Hyperparameters:**
  - Epochs & Batch Size
  - Learning Rate (**η**)



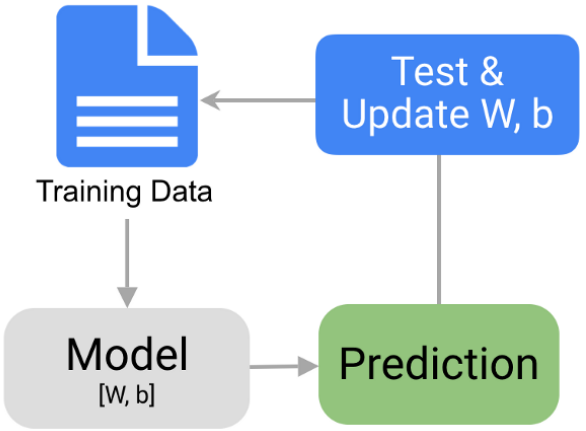
ANNs Training & Optimization

"Predict, Compare, & Learn" ~ Unknown

Eng. Mustafa Othman
Data Scientist & Analyst

ANNs Training & Optimization (Predict, Compare, & Learn)

- **Supervised learning** process can be broken down into **3** main steps: **predict**, **compare**, & **learn**.
- **Predict**: Perhaps this process begged the question, "How do we set **weight** values, so the network predicts accurately?".
- **Compare**: Comparing gives a measurement of how much a prediction "**missed**" by.
- **Learn**: Learning tells each **weight** how it can change to **reduce** the error.

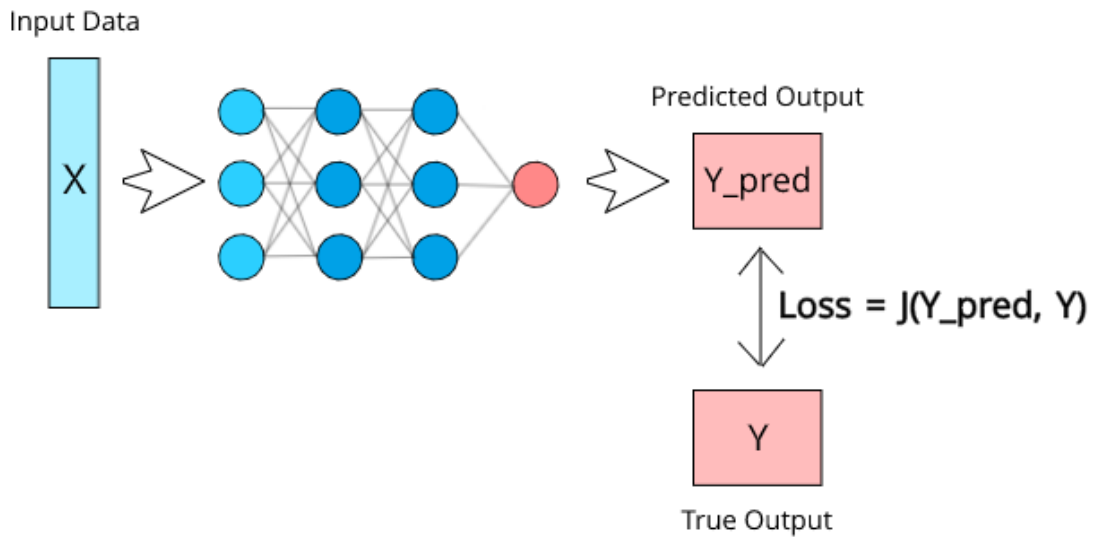


Loss Functions

“Don’t grieve. Anything you lose comes round in another form.” ~ Rumi

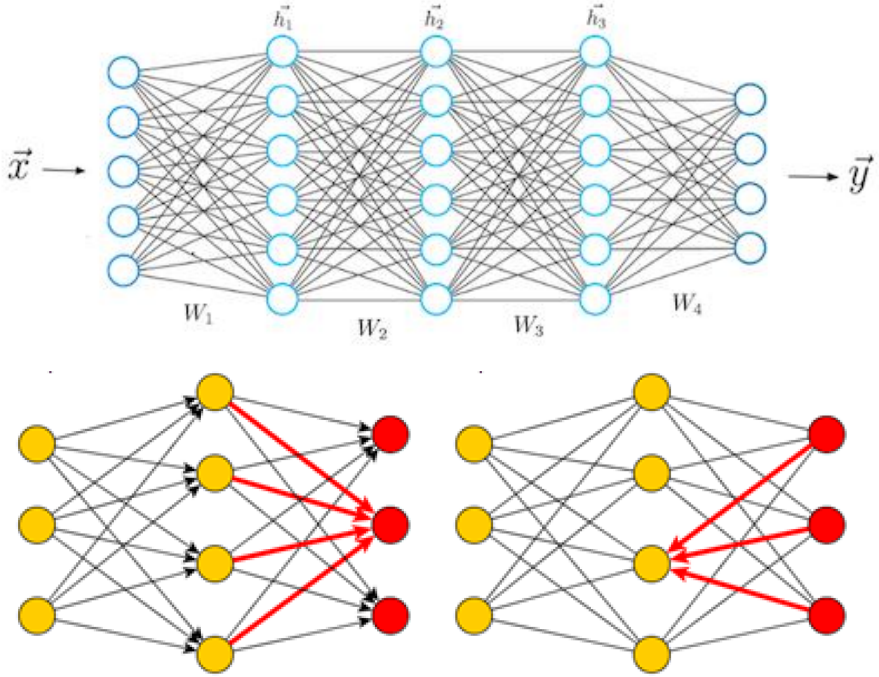
Loss Functions (00) (Overview)

- The **loss function** in a neural network quantifies the **difference** between the **expected** outcome and the **actual** outcome produced by the machine learning model.
- This loss essentially tells you something about the **performance** of the network: the **higher** it is, the **worse** your network performs overall.
- From the loss function, we can derive the **gradients** which are used to **update the weights**.
- Neural Network uses optimizing strategies like **stochastic gradient descent** to **minimize the error** in the algorithm. The way we compute this error is by using a **Loss Function**.



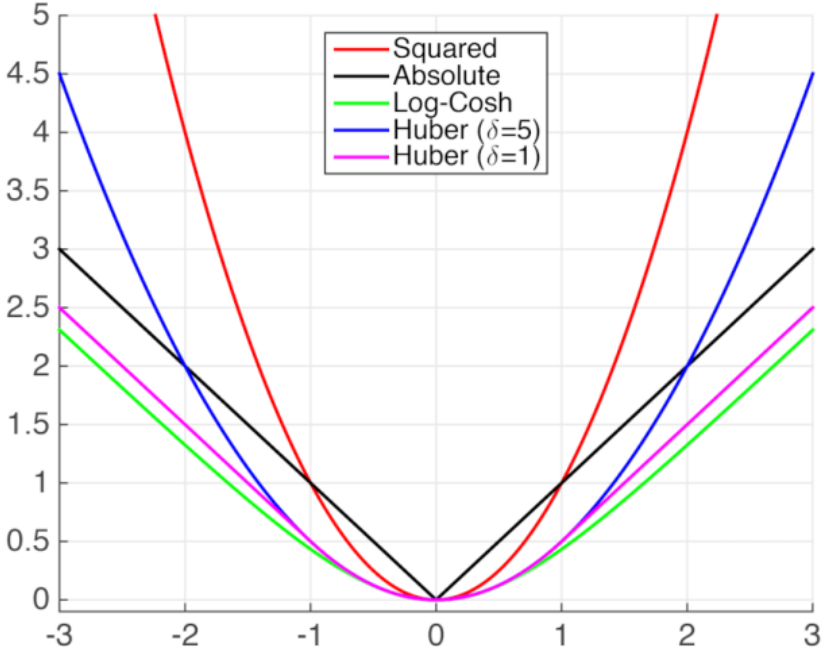
Loss Functions (01)
(Why?)

- Each training input is loaded into the neural network in a process called **forward propagation**.
- Once the model has produced an output, this predicted output is compared against the given target output in a process called **backpropagation** — the hyperparameters of the model are then adjusted so that it now outputs a result **closer** to the target output. This is where loss functions come in.
- While **forward propagation** refers to the **computational** process of predicting an output for a given input vector x, **backpropagation** and gradient descent describe the process of improving the **weights** and **biases** of the network in order to make better predictions.
- So, the loss function is used to **quantify** how **good** or **bad** the model is performing. These are divided into two categories i.e., Regression loss and Classification Loss.



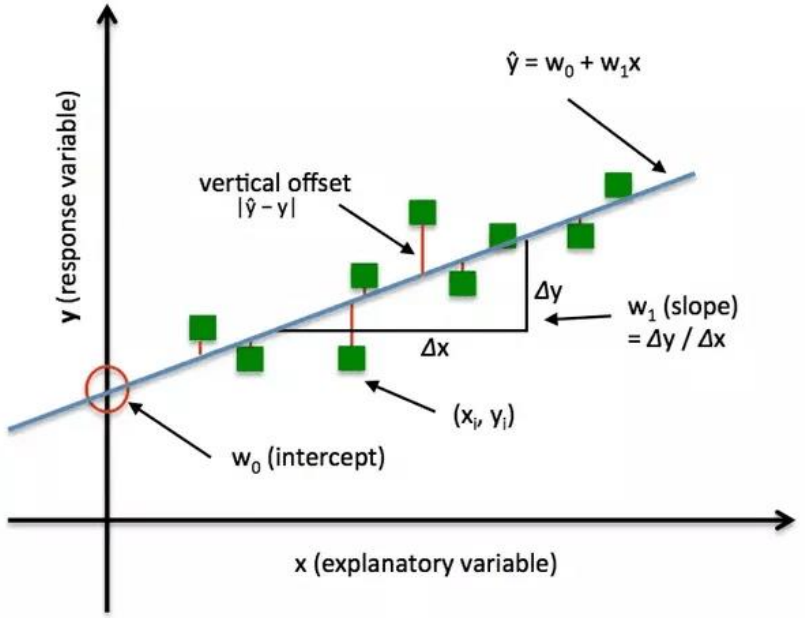
Loss Functions (02)
(Types)

- In **supervised learning**, there are two main types of loss functions — these correlate to the 2 major types of neural networks: **regression** and **classification** loss functions.
- **Regression** Loss Functions — used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels); Ex. **Mean Squared Error, Mean Absolute Error**
- **Classification** Loss Functions — used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging; Ex. **Binary Cross-Entropy, Categorical Cross-Entropy.**



Loss Functions (03)
(Mean Absolute Error "MAE")

- **Mean Absolute Error (MAE)** is the mean of the absolute value of the errors.
- **MAE** is the easiest to understand, because it's **the average error**.
- If **MAE** is **zero**, this indicates that the model predictions are perfect.
- **MAE** is used in cases when the training data has a large number of **outliers** to mitigate this.

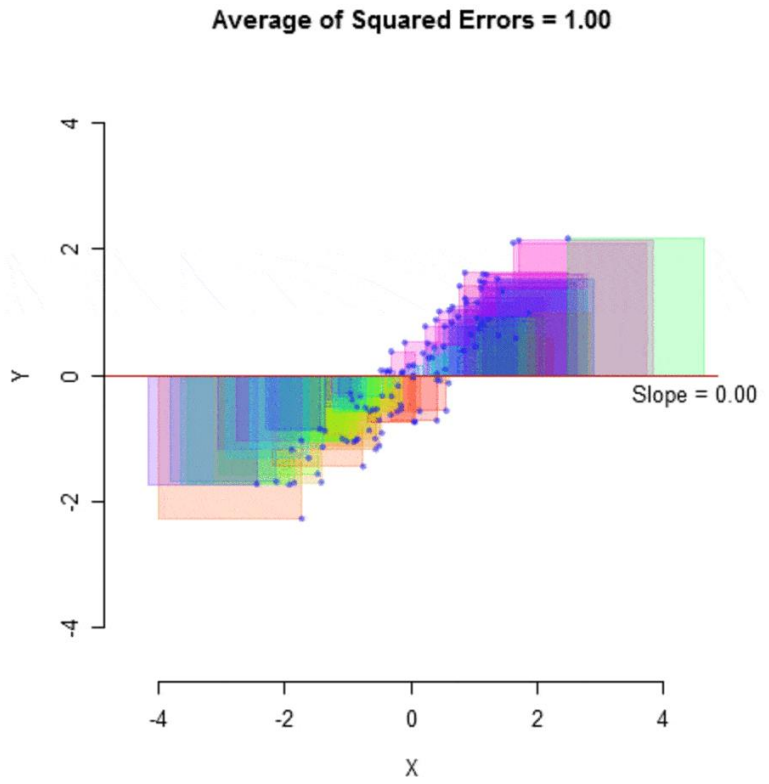


$$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}|$$

Loss Functions (04)
(Mean Squared Error "MSE")

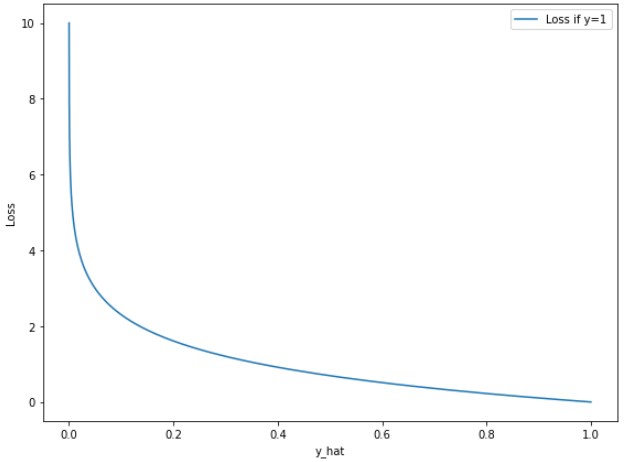
- **Mean Squared Error** (MSE) is the mean of the squared errors.
- **MSE** values are generally **larger** compared to the MAE since the residuals are being squared.
- **MSE** is highly sensitive to **outliers**, which can dramatically affect the loss because the distance is squared.
- **MSE** is more popular than MAE, because MSE "**punishes**" larger errors, which tends to be useful in the real world.

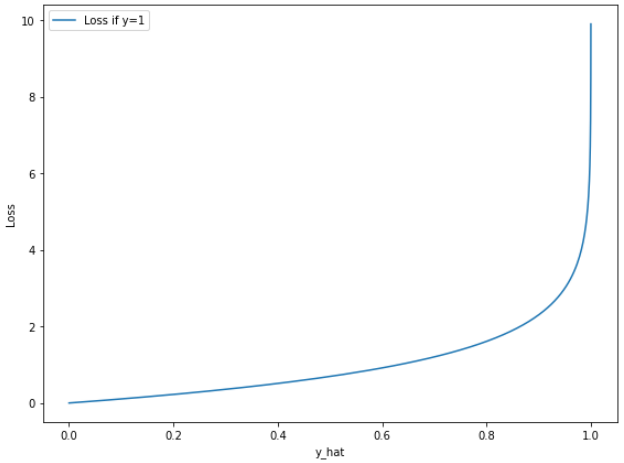
$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$



Loss Functions (05)
(Cross-Entropy)

- **Classification** neural networks work by outputting a vector of **probabilities** — the probability that the given input fits into each of the pre-set categories; then selecting the category with the **highest probability** as the final output.
- **Cross-Entropy** loss is also called **logarithmic** loss, **log** loss, or **logistic** loss. Each **predicted** class probability is compared to the **actual** class desired output **0** or **1** and a score/loss is calculated that **penalizes** the probability based on how far it is from the actual expected value. The **penalty** is logarithmic in nature yielding a **large score** for large differences close to **1** and **small score** for small differences tending to **0**.
- Cross-Entropy calculates the average **difference** between the **predicted** and **actual** probabilities.





Loss Functions (06) (Cross-Entropy)

Binary Cross-Entropy

- **Binary cross-entropy** is a loss function that is used in **binary classification** tasks. These are tasks that answer a question with only two choices (yes or no, A or B, 0 or 1, left or right).

$$C = -\frac{1}{N}\sum_{i=1}^{N}(y_i\ log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i))$$

- **Sigmoid** is the only activation function compatible with the **binary cross-entropy** loss function. You must use it on the last block before the target block.

Categorical Cross-Entropy

- **Categorical cross-entropy** is a loss function that is used in **multi-class classification** tasks.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(\hat{y}_{ij})$$

- **Softmax** is the only activation function recommended to use with the **categorical cross-entropy** loss function.

Loss Functions (07)
(Summary)

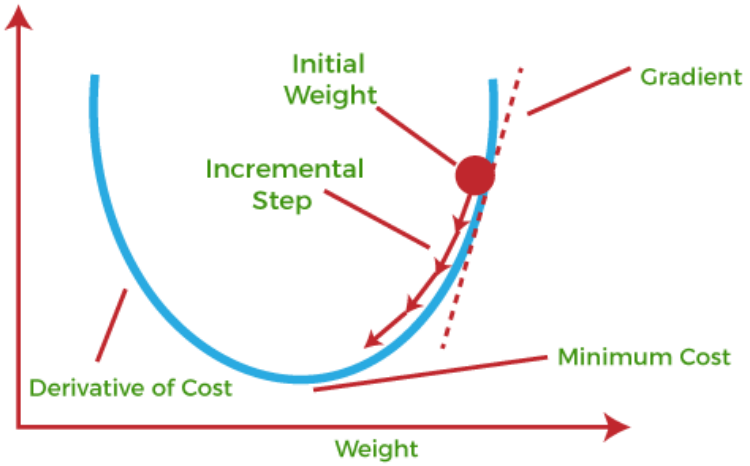
- A **loss function** measures how **good** a neural network model is in **performing** a certain task, which in most cases is regression or classification.
- We must **minimize** the value of the loss function during the **backpropagation** step in order to make the neural network better.
- We only use the **cross-entropy** loss function in **classification** tasks when we want the neural network to predict **probabilities**.
- For **regression** tasks, when we want the network to predict **continuous** numbers, we must use the **mean squared error** loss function.
- We use **mean absolute percentage** error loss function during demand forecasting to keep an eye on the performance of the network during training time.

Optimization: Minimize Cost

"Optimizing your strength is not optional, it's an obligation." ~ J.R. Rim

Optimization (Overview)

- In **mathematical** terminology, **Optimization** algorithm refers to the task of **minimizing/maximizing** an objective function f(x) parameterized by x.
- Similarly, in **machine learning**, **optimization** is the task of **minimizing the cost function** parameterized by the model's parameters.
- In this lecture we will study:
  - Gradient Descent
  - Optimizers



Gradient Descent (GD)

"I don't look at a problem and put variables in there that don't affect it." ~ Bill Parcells

Gradient Descent (GD) (00)
(Basics)

- **Gradient Descent** is an **optimization algorithm** used to train machine learning models by **minimizing errors** between predicted and actual results.
- The **Cost/Loss/Error** function measures the **distance/residual ($y`_i - y_i$)** between the **predicated $y`_i$** and the **actual y_i** for each data point (x_i, y_i)
- The goal is to **optimize** the function **(J) as low as** possible.
- How to find the best parameters **θ_0** and **θ_1** that optimize **J**, this optimization is done using **Gradient Descent (GD)**.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

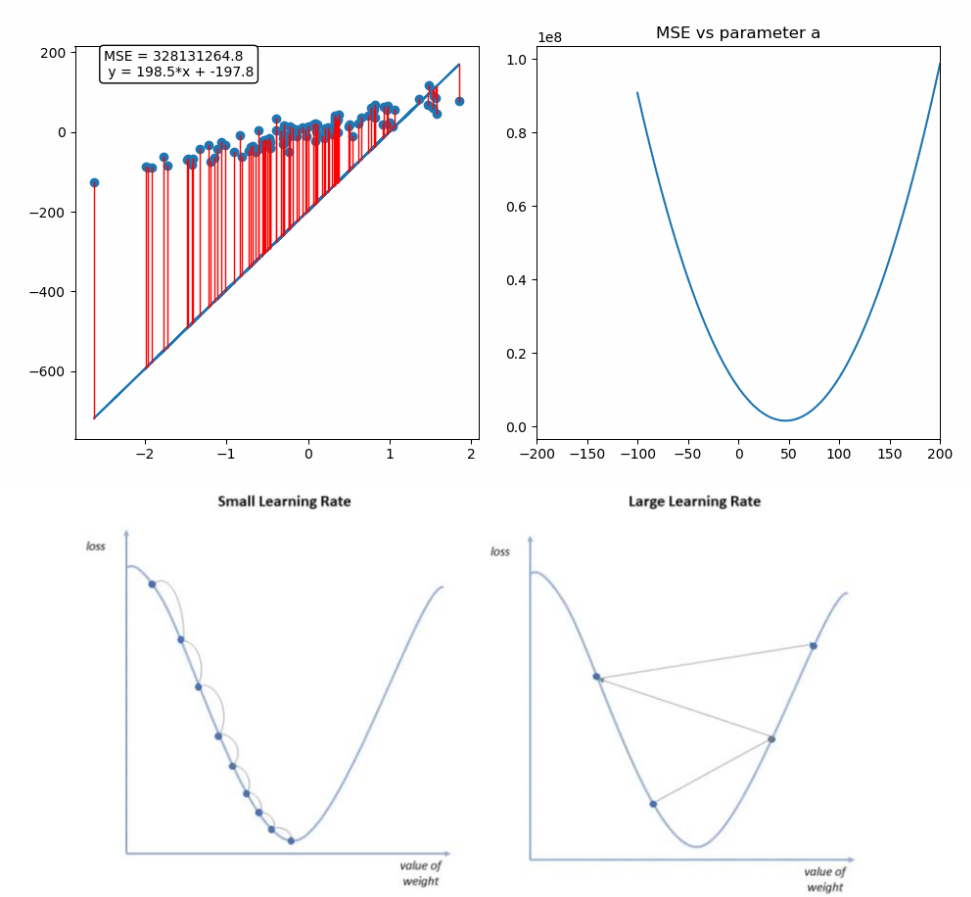
$$j(\theta_0, \theta_1) = \frac{1}{2n}\sum_{i=1}^{n}\left(y_i' - y_i\right)^2$$

$$= \frac{1}{2n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x_i - y_i\right)^2$$

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Gradient Descent (GD) (01) (Illustration)

- Gradient descent is a **first-order iterative optimization algorithm** for finding a **local minimum** of a **differentiable** function.
- Derivative gives us the **direction** of the guess; we know if we are getting **closer** or **further** away from the minimum.
- Gradient Descent algorithm is to keep **updating θ** until **convergence** to the **minimum**. By convergence we mean repeating until the function has no longer getting smaller.
- **Learning rate** (also referred to as **step size** or the **alpha**) is the size of the steps that are taken to reach the minimum.



Gradient Descent (GD) (02) (Differentiation)

- Choose initial points for θ_0 and θ_1.
- Calculate the partial derivative with respect to θ_0 and θ_1.

$$\frac{\partial j}{\partial \theta_0} = \frac{\partial j}{\partial \theta_0}\frac{1}{2n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)^2$$

$$= \frac{1}{2n}\frac{\partial}{\partial \theta_0}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)^2$$

$$= \frac{2}{2n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)$$

$$\frac{\partial j}{\partial \theta_1} = \frac{\partial j}{\partial \theta_1}\frac{1}{2n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)^2$$

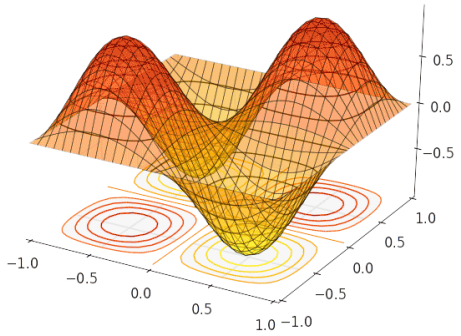
$$= \frac{1}{2n}\frac{\partial}{\partial \theta_1}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)^2$$

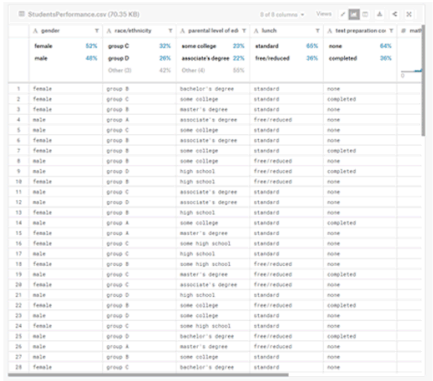
$$= \frac{2}{2n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)x_i$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(\theta_0+\theta_1 x_i-y_i\right)x_i$$

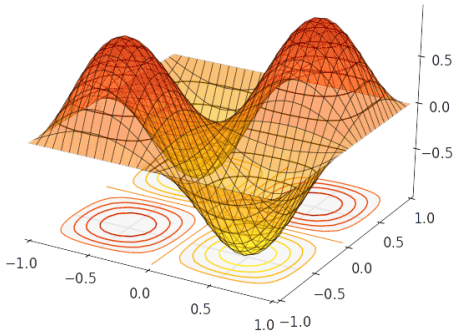
Gradient Descent (GD) (03)
(Types)

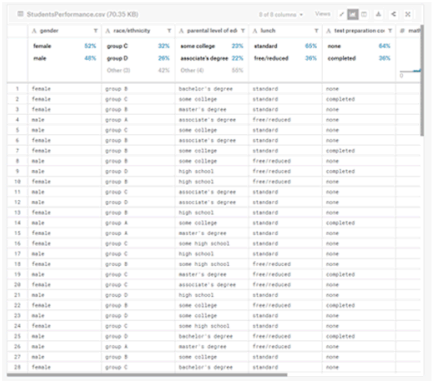
BATCH GRADIENT DESCENT



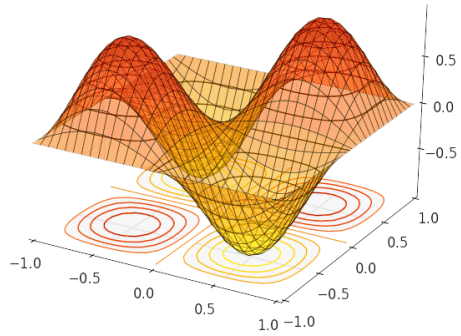


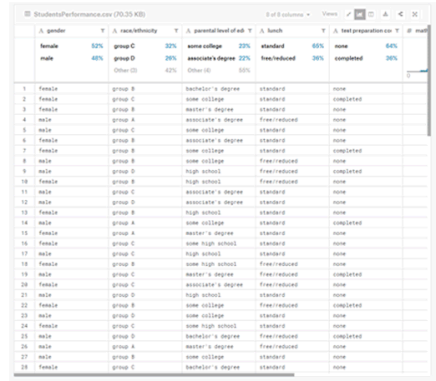
MINI-BATCH GRADIENT DESCENT





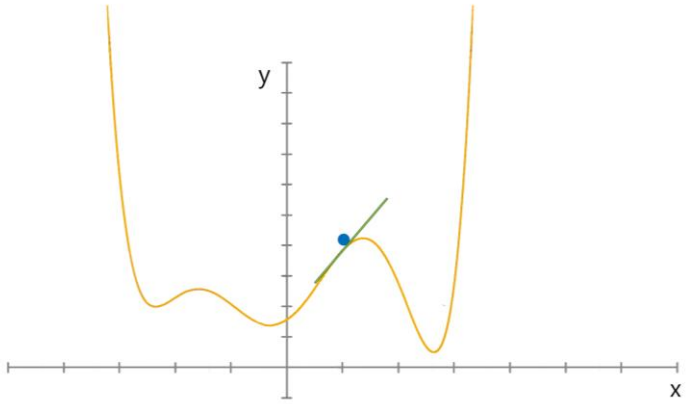
STOCHASTIC GRADIENT DESCENT

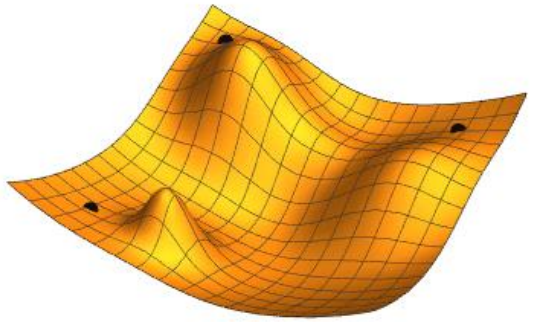


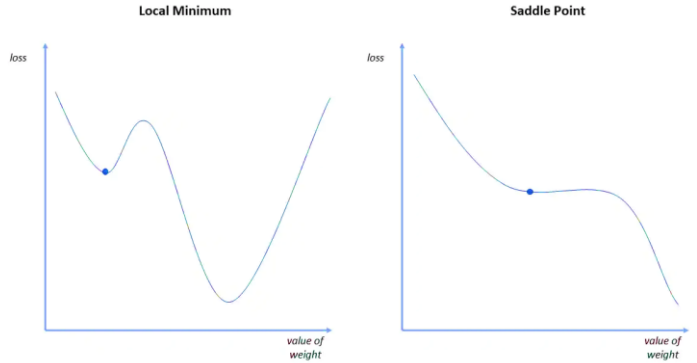


Gradient Descent (GD) (04)
(Local Minima & Saddle Points)

- Choosing a **random** point may lead to the **local minimum**.
- Local minima mimic the shape of a global minimum, where the slope of the cost function increases on either side of the current point.
- However, with **saddle points**, the **negative** gradient only exists on one side of the point, reaching a local maximum on one side and a local minimum on the other.
- **Noisy gradients** can help the gradient escape local minimums and saddle points.

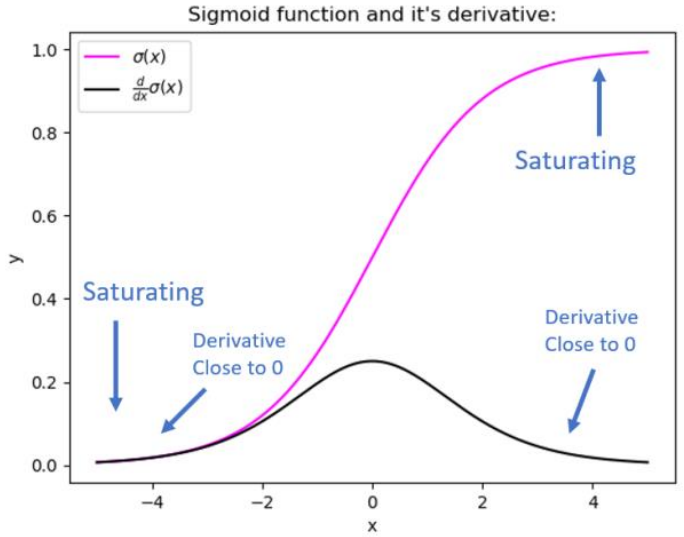






Gradient Descent (GD) (05)
(Vanishing & Exploding Gradients)

- **Vanishing gradients:** This occurs when the gradient is **too small**. As we move backwards during **backpropagation**, the gradient continues to become **smaller**, causing the earlier layers in the network to **learn more slowly** than later layers. When this happens, the weight parameters update until they become insignificant—i.e., 0—resulting in an algorithm that is **no longer learning**. Using **ReLU** activation function and reducing the **model complexity** would be helpful in handling this situation.
- **Exploding gradients:** This happens when the gradient is **too large**, creating an unstable model. In this case, the model weights will **grow too large**, and they will eventually be represented as NaN. One solution to this issue is to leverage a **dimensionality reduction** technique, which can help to minimize complexity within the model.

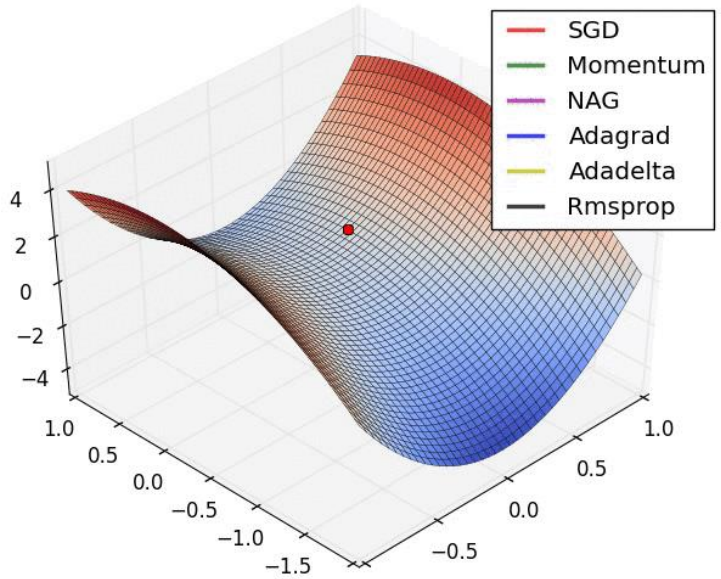


Optimizers

"To optimize the whole, we must sub-optimize the parts." ~ W. Edwards Deming

Optimizers (00) (Overview)

- An **optimizer** is a function or an algorithm that modifies the attributes of the neural network, such as **weights** and **learning rate**.
- **ANNs Optimizers Types:**
  - SGD / RMSprop / Adam / Adadelta / Adagrad / Adamax
- Key Terms:
  - **Epoch** – The number of times the algorithm runs on the whole training dataset.
  - **Sample** – A single row of a dataset.
  - **Batch** – It denotes the number of samples to be taken to for updating the model parameters.
  - **Learning rate** – It is a parameter that provides the model a scale of how much model weights should be updated.
  - **Cost Function/Loss Function** – A cost function is used to calculate the cost that is the difference between the predicted value and the actual value.
  - **Weights/ Bias** – The learnable parameters in a model that controls the signal between two neurons.

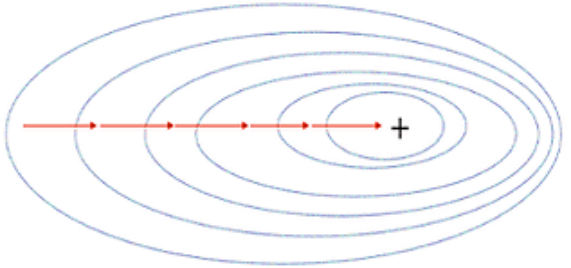


Optimizers (01)
(Stochastic Gradient Descent "SGD")

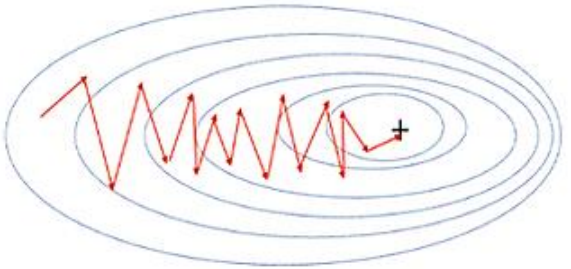
- In **Stochastic Gradient Descent (SGD)**, instead of taking the whole dataset for each iteration, we **randomly** select the batches of data. That means we only take **few samples** from the dataset.
- **Advantages:**
  - Frequent updates of model parameters, hence, **converges in less time**.
  - Requires **less memory** as no need to store values of loss functions.
- **Disadvantages:**
  - **High variance** in model parameters.
  - May **shoot** even after achieving global minima.
  - To get the same convergence as gradient descent needs to slowly **reduce** the value of learning rate.

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

Gradient Descent

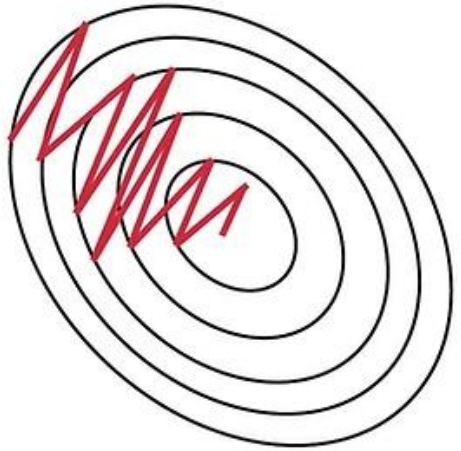


Stochastic Gradient Descent

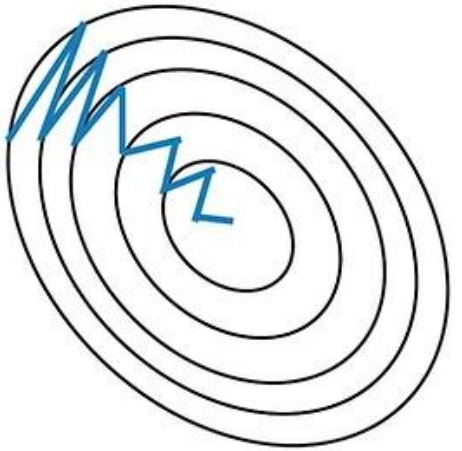


Optimizers (02)
(SGD with Momentum)

- A major disadvantage of the **MB-SGD** algorithm is that updates of weight are very noisy. **SGD with momentum** overcomes this disadvantage by **denoising** the gradients.
- **Advantages:**
  - Has all **advantages** of the SGD algorithm.
  - Converges **faster** than the GD algorithm.
- **Disadvantages:**
  - We need to **compute** one more variable for each update.



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

Optimizers (03)
(Adaptive Gradient Descent "Adagrad")

- **Adagrad** optimizer tries to offer this adaptiveness by **decaying** the **learning rate** in proportion to the updated history of the gradients. It uses different learning rates for each iteration.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

- **Advantages:**
  - **Learning rate** changes for each training parameter. Don't need to manually tune the learning rate.
  - Able to train on **sparse data**.
- **Disadvantages:**
  - **Computationally expensive** as a need to calculate the second order derivative.
  - The learning rate is always **decreasing** results in slow training.

Optimizers (04)
(Root Mean Square "RMSProp")

- **RMSProp** is an improvement to the **Adagrad** optimizer. This aims to reduce the **aggressiveness** of the learning rate by taking an **exponential average** of the gradients instead of the cumulative sum of squared gradients.
- **Advantages:**
  - The algorithm **converges quickly** and requires **lesser** tuning than gradient descent algorithms and their variants.
- **Disadvantages:**
  - The problem with RMSProp is that the **learning rate** must be defined **manually**, and the suggested value doesn't work for every application.

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

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t+1}} + \epsilon} \cdot g_t$$

Optimizers (05)
(Adaptive Momentum Estimation "Adam")

- **Adam (Adaptive Moment Estimation)** works with momentums of first and second order.
- **Adam** combines **AdaGrad**, **RMSprop** and **momentum** methods into one. Adam also utilizes the concept of **momentum** by adding fractions of previous gradients to the current one.
- **Advantages:**
  - The method is **too fast** and converges rapidly.
  - Rectifies **vanishing learning rate**, **high variance**.
- **Disadvantages:**
  - Computationally **costly**.

$$m_{t+1} = \beta_1 \cdot m_t + (1-\beta_1) \cdot g_t$$

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

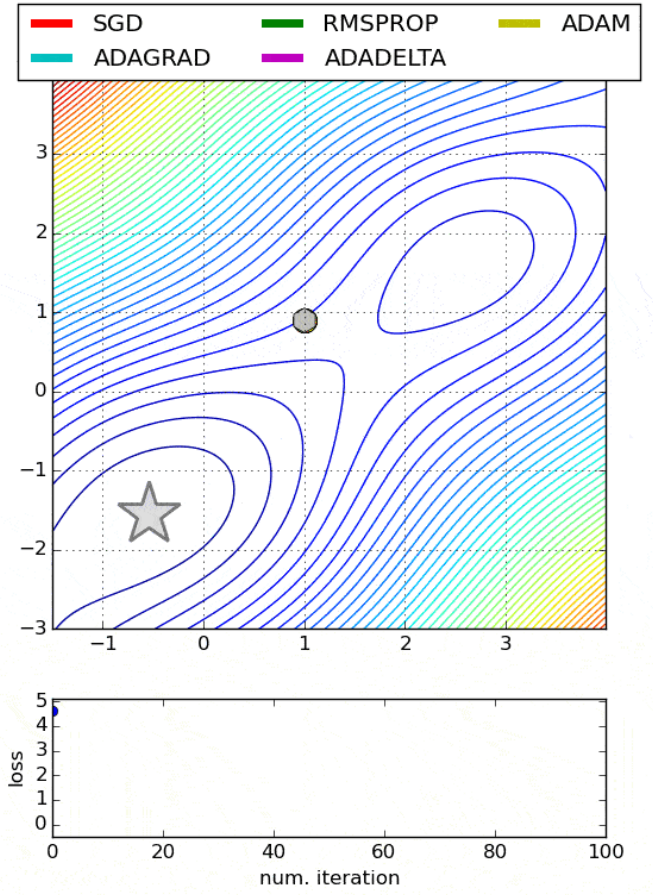
$$m_{t+1} = \frac{m_{t+1}}{(1-\beta_1^t)}$$

$$v_{t+1} = \frac{v_{t+1}}{(1-\beta_2^t)}$$

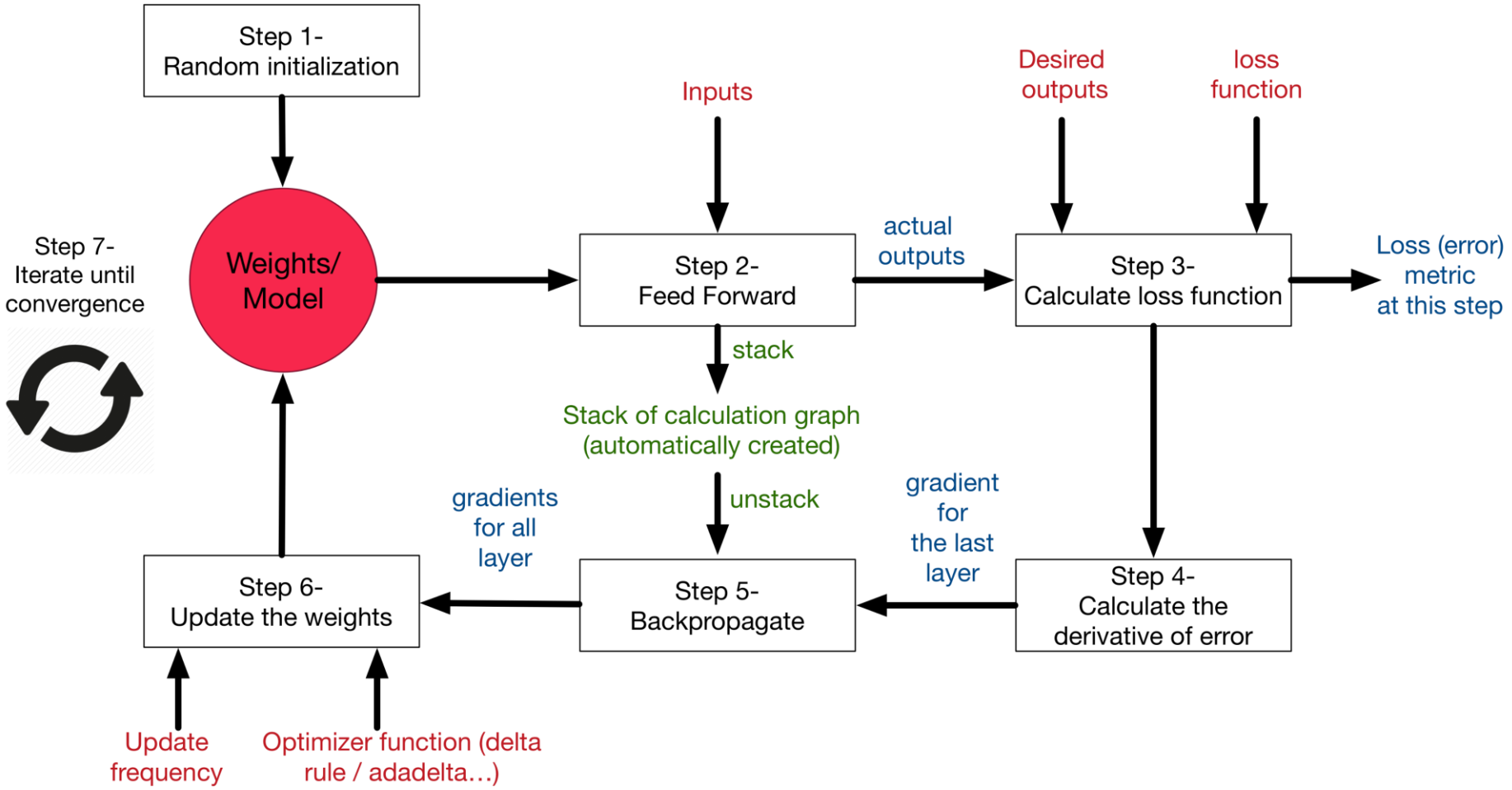
$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t+1}} + \epsilon} \cdot m_{t+1}$$

Optimizers (06) (Summary)

- It is observed that the **SGD** algorithm (red) is stuck at a **saddle point**. So, **SGD** algorithm can only be used for **shallow networks**.
- All the other algorithms except SGD finally converges one after the other, **AdaDelta** being the **fastest** followed by momentum algorithms.
- **AdaGrad** and **AdaDelta** algorithm can be used for **sparse data**.
- **Momentum** and **NAG** work well for most cases but is **slower**.
- **Adam** is the **fastest** algorithm to converge to minima.
- So, **Adam** is considered **the best algorithm** amongst all the algorithms discussed above.



Putting it All Together



Further Readings

- Deep Learning Illustrated, Jon Krohn
  - Chapters 8, 9
- Deep Learning: A Visual Approach, Andrew Glassner
  - Chapters 13, 14, 15

THANKS

Keep Moving Forward! ☺

Eng. Mustafa Othman
Data Scientist & Analyst