Loss Functions

1. Regression Loss Functions:

- Mean Squared Error (L2 (Ridge) Loss): is the mean of squared differences between the actual and predicted value.

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

- Mean Absolute Error (L1 (Lasso) Loss): is the absolute value of the distance between the actual and the predicted values.

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

In general, MSE is more commonly used than MAE because it is differentiable, which makes it easier to optimize using gradient descent. However, MAE can be a useful alternative in cases where the data may contain outliers, or when the goal is to optimize for the median rather than the mean of the error distribution.

- Mean Squared Logarithmic Error Loss: is the relative difference between the log-transformed actual and predicted values.

  Is usually used when you don't want to penalize the large differences in the predicted and the actual values when the predicted and the actual values are big numbers. Example: You want to Predict how many future visitors a restaurant will receive. The future visitors is a continuous value, and therefore, we want to do regression MSLE can here be used as the loss function.

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(\log(Y_i) - \log(\hat{Y}_i))^2$$

2. Binary Classification Loss Functions: are tasks that involve predicting a binary outcome (e.g., 0 or 1, true or false).

- Binary cross-entropy loss: measures the performance of a classification model whose output is a probability value between 0 and 1. It is used when there are only two classes in a classification problem.

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}(Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

- Hinge loss: This loss function is commonly used in support vector machines (SVMs).

$$\ell(y) = \max(0, 1 - t \cdot y)$$

- Squared Hinge Loss: This is a variant of hinge loss that penalizes the error more heavily for larger errors. It is defined as: L = (1 - y * ŷ)^2 . Where y is the true label (-1 or 1), ŷ is the predicted label (-1 or 1), and L is the loss.

3. Multi-Class Classification Loss Functions:

- Categorical cross-entropy (CCE): This loss function is often used in multi-class classification. It measures the difference between the predicted class probabilities and the true label.

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

- Sparse categorical cross-entropy (scce): This loss function is similar to CCE, but it is used when the class labels are integers rather than one-hot encoded vectors.

  In the case of `cce`, the one-hot target might be [0, 0, 1] and the model may predict [.5, .1, .4] (probably inaccurate, given that it gives more probability to the first class).

  In the case of `scce`, the target index might be `[0]`, and the model may predict `[.5]`.

Multilabel Classification vs Multi-Class Classification

- Multi-label classification: We have samples (images), and each sample can belong to multiple classes. We assume that a picture can contain any combination of animals!

• For example, let's say you are training a neural network to predict the ingredients present in a picture of some food. There will be multiple ingredients we need to predict so there will be multiple 1's in Y.
• For this we can't use softmax because softmax will always force only one class to become 1 and other classes to become 0. So instead we can simply keep sigmoid on all the output node values since we are trying to predict each class's individual probability.
• As for the loss we can directly use log loss on each node and sum it, similar to what we did in multiclass classification.

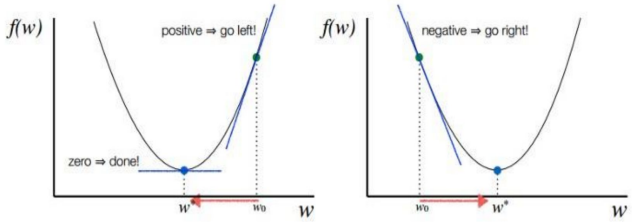
- Multi-class classification: We have samples (images), and each sample belongs to a class. We assume that a picture is either a cat, a dog, or a chick, but not a combination of them!.

Optimizers

- Optimizers: are algorithms or methods used to minimize an error function (loss function) or to maximize the efficiency of production. In other words, the optimizer helps to find the set of weights and biases that result in the lowest possible error for the model.

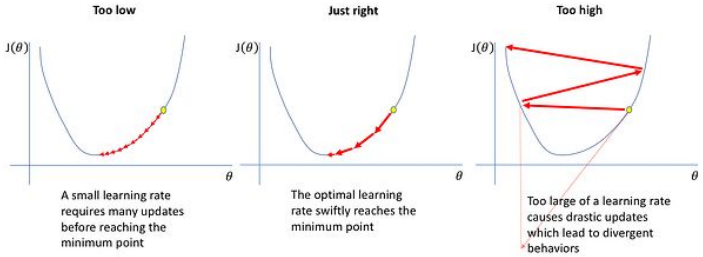
 # Types of optimizers:

- Gradient Descent:
- It is dependent on the derivatives of the loss function for finding minima.
- It will try to find the least cost function value by updating the weights of your learning algorithm and will come up with the best-suited parameter values corresponding to the Global Minima.
- This is done by moving down the hill with a negative slope, increasing the older weight, and positive slope reducing the older weight.
- The size of the update is controlled by a hyperparameter called the learning rate.



• Advantages of Gradient Descent:
– Easy to understand.
– Easy to implement.
• Disadvantages of Gradient Descent :
– May trap at local minima.
– the calculation is very slow.as Weights are changed after calculating gradient on the whole dataset.
– It requires large memory to calculate gradient on the whole dataset

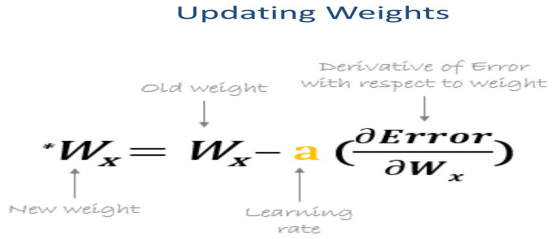
- **Learning Rate:** How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



- Choosing a good learning rate is important for the optimization process to converge to the optimal solution.

- There are techniques for choosing the learning rate:

1. Manual tuning: This involves manually choosing a learning rate and tuning it based on the performance of the model on the validation set.
2. Learning rate schedules: This involves scheduling the learning rate to decrease over time. The learning rate can be scheduled to decay linearly or exponentially, or it can be based on the number of epochs or the number of iterations.
3. Adaptive learning rates: Some optimizers, such as Adagrad, Adadelta, and Adam, adapt the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent parameters. These optimizers do not require a fixed learning rate to be set



- **Stochastic Gradient Descent:**
  - It is a variant of Gradient Descent.
  - If the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.
  - It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example.

+ Advantages:

- Frequent updates of model parameter.
- Requires less memory as no need to store values of loss functions.
- Get new minima (didn't stopped at the 1 st local minima).

+ Disadvantages:

- The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
- High Variance.
- May continue working even after achieving global minima.

- **Mini-Batch Gradient Descent:**
- It is similar to SGD, but instead of updating the parameters after every training example, it updates the parameters after every mini-batch of training examples. A mini-batch is a small set of training examples (e.g., 32 or 64 examples).

+ Advantages:

- Requires medium amount of memory.
- It leads to more stable convergence.

+ Disadvantages:

- Does not guarantee good convergence

- **SGD with momentum:**
  - It is a variant of SGD that introduces a momentum term to the update rule.
  - The momentum term helps the optimizer to "forget" the past gradients and focus on the current gradient, which can help the optimizer to escape from local minima and saddle points and converge to a better solution.
  - It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is known as momentum symbolized by 'γ' and (v is the velocity)

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

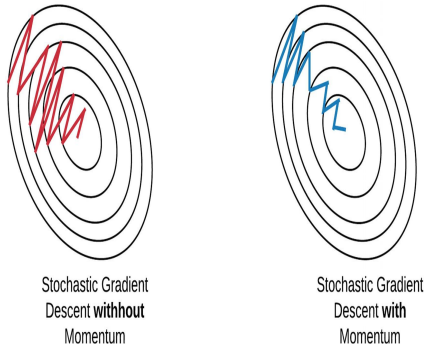
+ Advantages of SGD with momentum

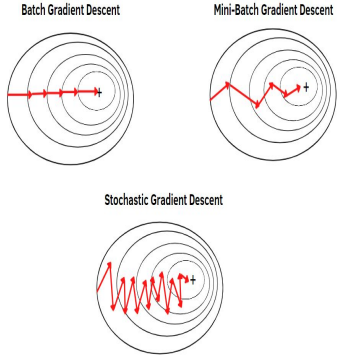
  - Momentum helps to reduce the noise and high variance of the parameters.

  - Converges faster than gradient descent.

+ Disadvantage of SGD with momentum:

- One more hyper-parameter is added which needs to be selected manually and accurately.





- **Adagrad (Adaptive Gradient Algorithm):**
  - All before Optimizers are using constant learning rate for all parameters and for each cycle but this optimizer uses a different learning rate for each iteration(EPOCH) rather than using the same learning rate for determining all the parameters.
  - The epsilon in the denominator is a very small value to ensure division by zero does not occur.

First, each weight has its own **cache** value, which collects the squares of the gradients till the current point.

$$cache_{new} = cache_{old} + (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Cache updation for Adagrad

The cache will continue to increase in value as the training progresses. Now the new update formula is as follows:

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

+ Advantages:
- Learning Rate changes adaptively with iterations.
- It is able to train sparse data as well.

+ Disadvantage:
- If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

- **AdaDelta:**
  - It is a gradient descent optimizer that adapts the learning rate based on the past gradients.
  - It is an extension of AdaGrad which tends to remove the decaying learning Rate problem of it.

+ Advantages:
- Now the learning rate does not decay and the training does not stop.

+ Disadvantage:
- Computationally expensive.

- **RMS-Prop (Root Mean Square Propagation):**

  - It is a gradient descent optimizer that is similar to Adadelta, but it uses a moving average of the squared gradients to scale the learning rate. RMSprop is well-suited for non-stationary objectives and online learning.
  - RMS-Prop is a special version of Adagrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients.
  - RMS-Prop basically combines momentum with AdaGrad.
  - the decay rate * (gamma) value is usually around 0.9 or 0.99.

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

+ Advantages:

• In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
• the learning rate does not decay too quickly, that allowing training to continue for much longer time.

+ Disadvantage:

• Slow Learning.

- **Adam (Adaptive Moment Estimation):**

  - Adam optimizer is one of the most popular and famous gradient descent optimization algorithms.
  - The idea behind Adam optimizer is to utilize the momentum concept from "SGD with momentum" and adaptive learning rate from "Ada delta".
  - It is a method that computes adaptive learning rates for each parameter.

+ Advantages:

• Easy to implement.
• Computationally efficient.
• Little memory requirements.

Adam is a little like combining RMSProp with Momentum. First we calculate our m value, which will represent the momentum at the current point.

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adam Momentum Update Formula

The only difference between this equation and the momentum equation is that instead of the learning rate we keep (1-Beta_1) to be multiplied with the current gradient.

Next we will calculate the accumulated cache, which is exactly the same as it is in RMSProp:

$$cache_{new} = \beta_2 * cache_{old} + (1 - \beta_2) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Now we can get the final update formula:

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}$$

Adam weight updation formula

* The decay rate is used to control the rate at which the learning rate changes over time. In general, a smaller decay rate means that the learning rate will change more quickly, which can help the model converge faster. However, if the decay rate is too small, the learning rate may oscillate or diverge, leading to poor performance. On the other hand, a larger decay rate means that the learning rate will change more slowly, which can help the model avoid oscillations and divergences, but may result in slower convergence.

Metrics

1. Classification Metrics:

- Accuracy: the number of correct predictions divided by the total number of predictions, multiplied by 100.
- Precision: the number of true positive predictions divided by the total number of positive predictions made by the model.
  Precision= True_Positive/ (True_Positive+ False_Positive). It is used to measure the proportion of true positive predictions made by the model.
- Recall: the number of true positive predictions divided by the total number of actual positive cases.It is used to measure the proportion of positive predictions that were actually correct.
  Recall= True_Positive/ (True_Positive+ False_Negative)
- F1 Score: is a balance between precision and recall and it is harmonic mean of precision and recall.
  F1-score= 2*Precision*Recall/(Precision+Recall)
- Confusion Matrix: This is a table that is used to evaluate the performance of a classification model. It shows the number of true positive, true negative, false positive, and false negative predictions made by the model.
- Sensitivity (also known as the true positive rate or recall): It is the number of true positive predictions divided by the total number of actual positive cases. Sensitivity is a measure of the model's ability to correctly identify positive cases.
  Sensitivity= Recall= TP/(TP+FN)
- Specificity (also known as the true negative rate): It is the number of true negative predictions divided by the total number of actual negative cases. Specificity is a measure of the model's ability to correctly identify negative cases.
  Specificity= True Negative Rate= TN/(TN+FP)
- ROC Curve: This is a graphical plot that is used to evaluate the performance of a binary classification model. It shows the relationship between the true positive rate and the false positive rate at different classification thresholds.
- AUC (area under the curve): is the area under the ROC curve and it measures the ability of the model to distinguish between positive and negative cases. A model with a high AUC has a high true positive rate and a low false positive rate.

2. Regression Metrics:

- Mean Absolute Error (MAE): it measures the average difference between the predicted values and the actual values.
  It is defined as the sum of the absolute differences between the predicted and actual values divided by the total number of predictions.

$$MAE = \frac{1}{n}\sum_{j=1}^{n} |y_j - \hat{y}_j|$$

- Mean Squared Error (MSE): finds the average squared error between the predicted and actual values.

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

Callbacks

- ModelCheckpoint: This callback is used to save the model weights at certain intervals during training.
- EarlyStopping: This callback is used to stop training early if the model has not improved after a certain number of epochs. It can be configured to monitor a metric such as accuracy or loss and stop training if the metric has not improved for a specified number of epochs.
- LearningRateScheduler: This callback is used to modify the learning rate during training. It can be configured to reduce the learning rate at certain intervals or when the model has stopped improving.
- TensorBoard: This callback is used to visualize the training process.
- CSVLogger: is used to log metrics during training to a CSV file. It can be used to track metrics such as accuracy, loss, and learning rate during training, and the logged data can be used for later analysis or visualization.

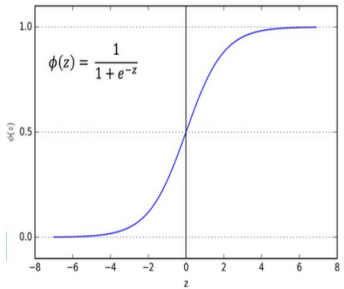
Activation functions

- They are used to introduce non-linearity into the network, allowing it to learn more complex patterns and make more accurate predictions. Without activation function, weight and bias would only have a linear transformation, or neural network is just a linear regression model.

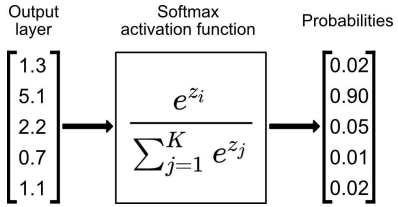
- Types:

1. Linear Activation Function:
   - the function is a line or linear.
   - Equation = f(x) = x.
   - Range : (-infinity to infinity).

2. Non-linear Activation Function:
   - Sigmoid: This function maps any input value to a value between 0 and 1, making it useful for classification tasks. It is used as output functions for binary classification but are generally not used within hidden layers.

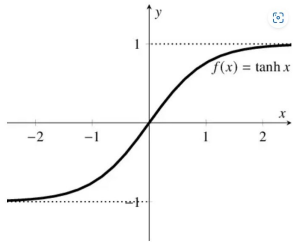


- SOFTMAX: is used to convert a set of inputs into a probability distribution. It is often used in classification tasks with multiple classes. The softmax function takes in a vector of real values and returns a vector of values that sum to 1, representing a probability distribution over the classes.



In summary, The sigmoid function is used to map a single input value to a value between 0 and 1, while the softmax function is used to convert a set of input values into a probability distribution.

- Tanh (hyperbolic tangent): This function maps any input value to a value between -1 and 1. The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.



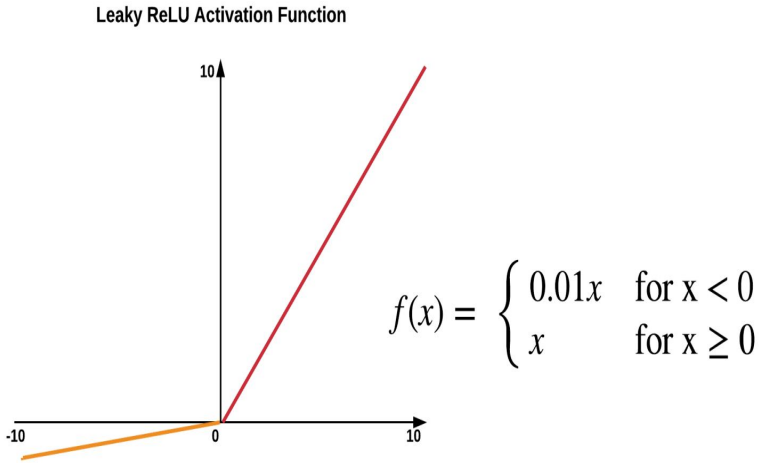
The tanh activation function is also **sort of sigmoidal (S-shaped).**

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

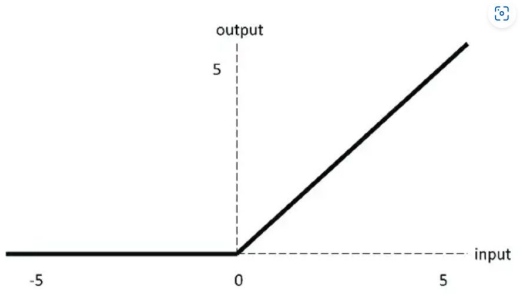
- ReLU (rectified linear unit): This function maps any input value less than 0 to 0, and any input value greater than or equal to 0 to the input value itself. It is fast to compute.

- Leaky ReLU : It allows a small, non-zero gradient when the input is negative, which can help prevent the "dying ReLU" problem.

+ dying ReLU problem: the output of a rectified linear unit (ReLU) activation function becomes stuck at 0, due to the presence of negative input values. When this happens, the gradients of the ReLU units with respect to the input become 0, and the weights of the model are no longer updated. This can lead to suboptimal model performance.



$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



The ReLU is half rectified (from the bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

$$\sigma(x) = \begin{cases} max(0, x) & , x >= 0 \\ 0 & , x < 0 \end{cases}$$

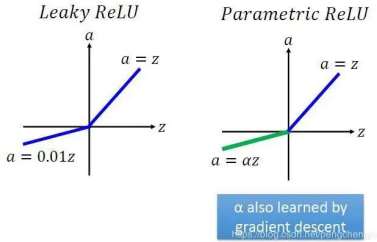
- Parametric ReLU (PReLU): is a variant of the ReLU activation function that introduces a learnable parameter, called the negative slope, that determines the slope of the function for negative input values.

NOTE:

- if $\mathbf{a_i=0}$, f becomes ReLU
- if $\mathbf{a_i>0}$, f becomes leaky ReLU
- if $\mathbf{a_i}$ **is a learnable parameter**, f becomes PReLU

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

ReLU - variant



+ Advantages:

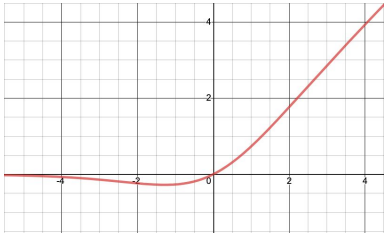
- In the negative region, PReLU has a **small slope**, which can also **avoid** the problem of **ReLU death**.
- Compared to ELU, PReLU is a **linear operation** in the negative region. Although the slope is small, it **does not tend to 0**.

- Swish (A Self-Gated) Function:

- The formula is: **y = x * sigmoid (x)**

+ Advantage:

1. is easy to compute, as it only requires a multiplication and a sigmoid function evaluation.
2. is continuous and smooth, which can make it easier to optimize during training.
3. has a "self-gating" property, which means that it can adjust the strength of the activation depending on the input. For large negative values, the output of the Swish function is close to 0, while for large positive values, the output is close to the input. This can help the network to learn more efficiently.

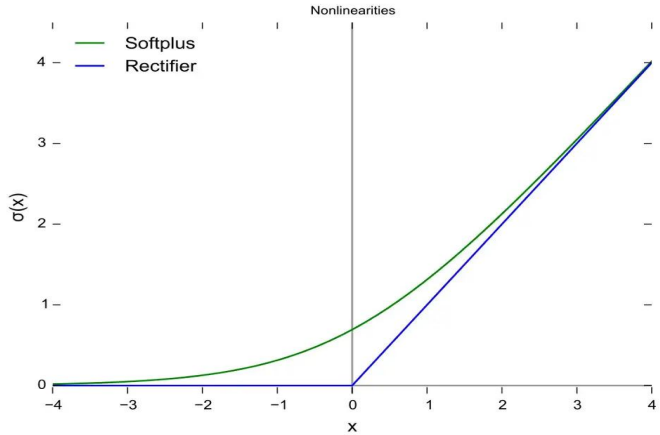


NOTE: Swish's design was inspired by the use of sigmoid functions for gating in LSTMs and highway networks and can only be implemented when your neural network is ≥ 40 layers.

- Softplus: is also called the logistic/sigmoid function and similar to the ReLU function, but it is relatively smooth.

+ properties:

- is smooth and continuous, which can make it easier to optimize during training.
- output of the softplus function is always non-negative, therefore it does not suffer from the vanishing gradient problem, as in RELU.
- has a "self-normalizing" property, which means that it can help to stabilize the activations of the network and prevent overfitting.



- Comparison of softmax , sigmoid and tanh:
  - **Softmax**: used as a categorical activation and the outputs between the range (0,1) so that the **sum of the outputs is always 1.**
  - **Tanh, or hyperbolic tangent:** is a logistic function that maps the outputs to the range of (-1,1). Tanh can be used in binary classification between two classes and it is centered around 0. (mean = 0).
  - **Sigmoid function:** is another logistic function like tanh, but If the sigmoid function inputs are restricted to real and positive values, the output will be in the range of (0,1). This makes sigmoid a great function for predicting a probability for something.

- Why there is a negative values in Tanh ?

1. Improved learning rate.
2. Centered around 0, which can make it easier to initialize the weights of the network.
3. Non-saturating.

- Why is ReLU good for hidden layers?

1. It does not saturate: The output of the ReLU function does not saturate for large input values, which can help the network to learn faster.
2. It helps to alleviate the vanishing gradient problem.

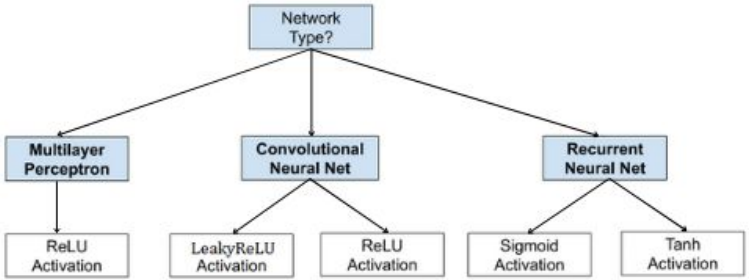
* Saturation: when the output of the activation function becomes close to its maximum or minimum value, and the gradient of the function becomes very small. This can make it difficult for the network to learn, as the gradients of the weights become very small and the network has difficulty updating its weights.Saturation can occur in any activation function, but it is more common in activation functions that have a limited range of output values. For example, the sigmoid and tanh activation functions both have a range of -1 to 1 and can saturate for large input values. The ReLU activation function, on the other hand, has a range of 0 to infinity and does not saturate.

* vanishing gradient problem: It occurs when the gradients of the weights in the network become very small, making it difficult for the network to learn and update its weights. It is often caused by the use of activation functions that saturate for large input values. Activation functions such as the sigmoid and tanh functions have a limited range of output values and can saturate for large input values, resulting in very small gradients. This can make it difficult for the network to learn, as the gradients of the weights become very small and the network has difficulty updating its weights. To alleviate the vanishing gradient problem, it is important to choose activation functions such as the ReLU (Rectified Linear Unit) function and the leaky ReLU function, cause they have a large range of output values and do not saturate.

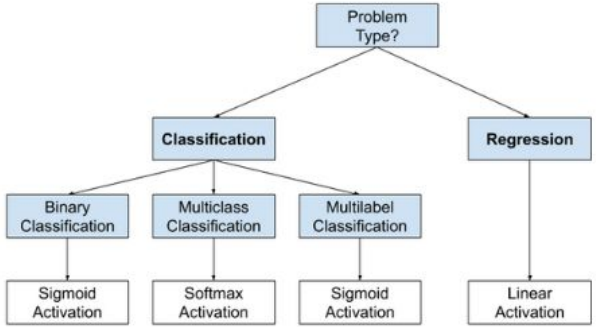
- Other techniques that can be used to alleviate the vanishing gradient problem:
  - Weight initialization.
  - Batch normalization.
  - Skip connections: Skip connections are connections that bypass one or more layers in the network.

- Choose an Activation Functions:

- Activation Functions in Hidden Layers:



- Activation Functions in Output Layers:



- Leaky ReLU: is used to alleviate the "dying ReLU" problem.
- Softplus: used in the same contexts as the ReLU function and it is particularly useful when the input data is positive and the model is prone to vanishing gradients.
- Swish: It is often used as an alternative to the ReLU function in the hidden layers of deep neural networks.

Weight Initialization

- **Zeros initialization and Constant initialization:** this method sets all the weights to zeros or contestant. It is generally not recommended for larger models, as it can lead to poor performance. This is because zero initialization can result in all of the neurons in the network producing the same output, which can prevent the network from learning to distinguish between different features in the input data.
- **Random Initialization:** used to break the symmetry and this process gives much better accuracy than zero initialization. It prevents neuron from learning the same features of its inputs. If the weights are initialized with high value, the learning takes a lot of time and if the weights are initialized with low value, then it slows down the optimization.

- **LeCun Normal Initialization:** initializing the weights of the network using a normal distribution with a mean of 0 and a standard deviation of sqrt(1 / n), where n is the number of input units in the weight tensor.
- **LeCun Uniform Initialization:**The weights are initialized using a uniform distribution within [-limit, limit] where the limit is sqrt(3 / fan_in) where fan_in is the number of input units in the weight tensor.
- **LeCun Initialization:** is often used with ReLU activations and to prevent the vanishing or explosion of the gradients during the backpropagation by solving the growing variance with the number of inputs and by setting constant variance.

- **Xavier Normal Initialization (Glorot):** initializing the weights using a normal distribution with a mean of 0 and a standard deviation of sqrt(2 / (n + m)), where is n the number of input units in the weight tensor and m is the number of output units.
- **Xavier Uniform Initialization:** initializing the weights using a uniform distribution with within [-limit, limit], where limit = sqrt(6 / (fan_in + fan_out)) (fan_in is the number of input units in the weight tensor and fan_out is the number of output units).
- **Xavier initialization:** is often used with sigmoid or tanh activations. It helps to prevent the vanishing or exploding gradient problem.

- **He Normal Initialization :** initializing the weights using a normal distribution with a mean of 0 and a standard deviation of sqrt(2 / (n)), where is n the number of input units in the weight tensor.
- **He Uniform Initialization:** initializing the weights using a uniform distribution with within [-limit, limit], where limit = sqrt(6 / (fan_in)) (fan_in is the number of input units in the weight tensor.
- **He initialization:** is often used with ReLU activations. It helps to prevent the vanishing gradient problem and solves dying neuron problems.

Normal Distribution Vs Uniform Distribution

Normal distribution: is defined by its mean, which is the center of the curve, and its standard deviation, which determines the spread of the curve.

- Often used when the initial values of the weights do not need to be precise.
- Often used when the range of possible values for the weights is not known.
- It is important to break symmetry in the model.

Uniform distribution: is defined by its minimum and maximum values, and the probability of any value within the range is the same.

- Often used when the initial values of the weights do need to be precise.
- Often used when the range of possible values for the weights is known.
- It is important to avoid large initial weights that could slow or destabilize the training process.