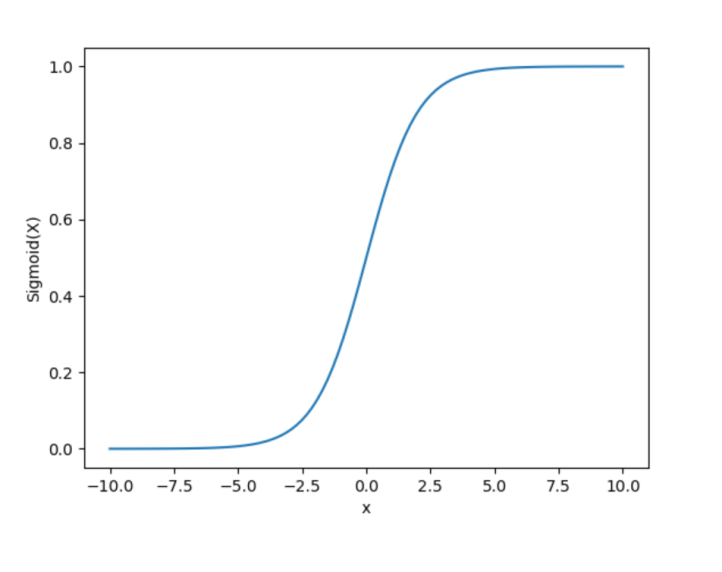
**1. Explain the Activation Functions in your own language**

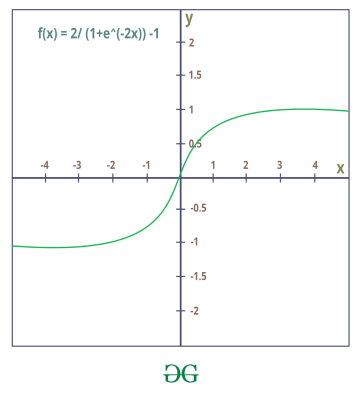
1. **sigmoid**
2. **tanh**
3. **ReLU**
4. **ELU**
5. **LeakyReLU**
6. **swish**
7. **Sigmoid Function**

****

### It is a function which is plotted as ‘S’ shaped graph.

* **Equation : A = 1/(1 + e-x)**
* **Nature : Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.**
* **Value Range : 0 to 1**
* **Uses : Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be *1* if value is greater than 0.5 and *0* otherwise.**

### Tanh Function

****

* **The activation that works almost always better than sigmoid function is Tanh function also known as Tangent Hyperbolic function. It’s actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.**
* **Equation :-  
  f(x) = tanh(x) = 2/(1 + e-2x) – 1  
  OR  
  tanh(x) = 2 \* sigmoid(2x) – 1**
* **Value Range :- -1 to +1**
* **Nature :- non-linear**
* **Uses :- Usually used in hidden layers of a neural network as it’s values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.**

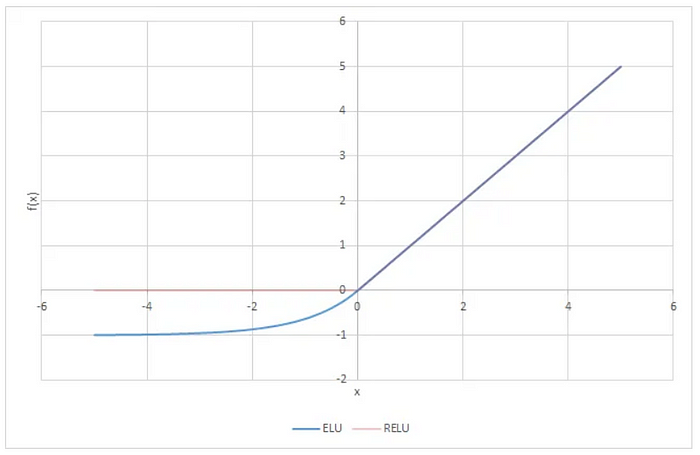
**c)RELU Function**

* **It Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network.**
* **Equation :- *A(x) = max(0,x)*. It gives an output x if x is positive and 0 otherwise.**
* **Value Range :- [0, inf)**
* **Nature :- non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.**
* **Uses :- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.**

**In simple words, RELU learns *much faster* than sigmoid and Tanh function**.

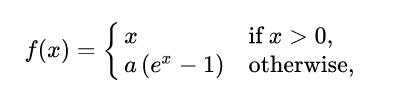
# d)ELU (Exponential LU) Function >

**Exponential Linear Units (ELUs) are utilized to expedite the deep learning process by shifting the mean activations closer to zero. This is achieved by introducing an alpha constant, which must be a positive number.**

****

**ELU blue ReLU brown**

**ELU has been demonstrated to yield more accurate results and faster convergence compared to ReLU. While both ELU and ReLU produce similar outputs for positive inputs, ELU smoothly smooths negative inputs (to -alpha) at a slower rate, whereas ReLU sharpens this smoothing abruptly.**

****

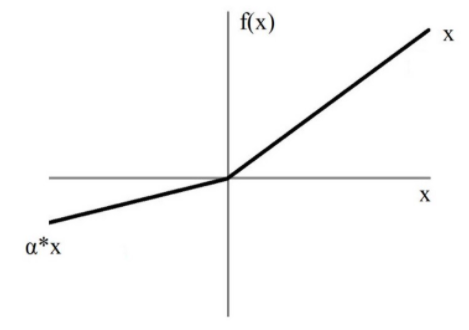
**alpha is a hyper parameter, with positive value constraint**

**e)Leaky ReLu is nothing more than an improved version of the ReLU activation function. As mentioned in the previous section, using ReLU may “kill” some neurons in our neural network and these neurons may never become active again.**

**Leaky ReLU was defined to solve this problem. Unlike “vanilla” ReLU, where all values  of the neurons are zero for input values , in the case of Leaky ReLU we add a small linear component to the function:**

**Math. definition of Leaky-ReLU
Fig. 5 Math. definition of Leaky-ReLU**

**Leaky-ReLU activation looks as follows:**

**Fig. 6 Leaky ReLU.**

**Basically, we have replaced the horizontal line for values below zero with a non-horizontal linear line. The slope of this linear line can be set by the parameter , which is multiplied by the input .**

**The advantage of using Leaky ReLU and replacing the horizontal line is that we avoid zero gradients. This is because in this case we no longer have “dead” neurons that are always zero and thus no longer contribute to the training**.

**2. What happens when you increase or decrease the optimizer learning rate?**

**When you increase or decrease the optimizer learning rate, it affects how quickly or slowly the model's parameters are updated during training.**

* **Increasing Learning Rate:**
  + **Definition: A higher learning rate means larger steps are taken in the direction of the gradient.**
  + **Criteria:**
    - **Pros:**
      * **Faster convergence.**
      * **Can escape local minima.**
    - **Cons:**
      * **Risk of overshooting the optimal solution.**
      * **May cause instability in training.**
* **Decreasing Learning Rate:**
  + **Definition: A lower learning rate means smaller steps are taken in the direction of the gradient.**
  + **Criteria:**
    - **Pros:**
      * **More precise convergence.**
      * **Stable training process.**
    - **Cons:**
      * **Slower convergence.**
      * **Risk of getting stuck in local minima.**

**Example:**

**If the learning rate is too high, the loss function might oscillate or even diverge:**

**L(t+1)=L(t)−𝛼×∇ient(L(t))**

**where L(t) is the loss at iteration t and 𝛼 is the learning rate.**

**If the learning rate is too low, the model might take too long to converge:**

**L(t+1)=L(t)−𝛼×∇ient(L(t))**

**Python Example:**

**Python**

**import tensorflow as tf**

**# High learning rate**

**optimizer\_high = tf.keras.optimizers.SGD(learning\_rate=0.1)**

**# Low learning rate**

**optimizer\_low = tf.keras.optimizers.SGD(learning\_rate=0.001)**

**In summary, the learning rate is a crucial hyperparameter that needs to be carefully tuned to balance the speed and stability of the training process.**

**3. What happens when you increase the number of internal hidden neurons?**

**More hidden layers shouldn't prevent convergence, although it becomes more challenging to get a learning rate that updates all layer weights efficiently. However, if you are using full-batch update, you should be able to determine a learning rate low enough to make your neural network progress or always decrease the objective function.**

**Assuming that you are using a full-batch update, at a given iteration, in order to guarantee sufficient objective function decrease without manually specifying a learning rate, you can**[**perform line search**](https://deepchecks.com/)**to find a learning rate that satisfies the two Wolfe conditions.**

**You can also use L-BFGS with line search to optimise your neural network efficiently. L-BFGS uses an approximation of the Hessian (second order gradient) which in a way sets a learning rate for every parameter. minFunc for Matlab and scipy optimise for python have L-BFGS.**

**However, before going further I would do the following two things:**

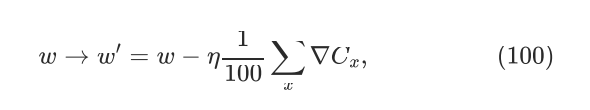
**1) I would check that the implementation is correct by checking that the gradient function is correct. First, numerically approximate the gradient for a parameter using the function value, then compare it with the value returned by the gradient function. The two values should be approximately the same.**

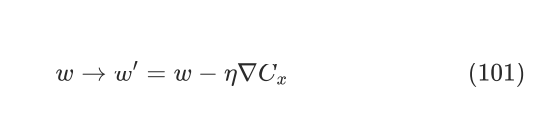
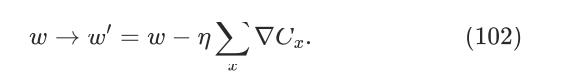
**2) I would also compare the neural network (NN) results with easy-to-use NN code available online. The autograd python library shows a quick implementation of multi-layer perceptron and its performance on a toy example. You can set the learning rate and the number of hidden layers fairly easy.**

**4. What happens when you increase the size of batch computation?**

**Let's first suppose that we're doing online learning, i.e. that we're using a mini­batch size of 1. The obvious worry about online learning is that using mini­batches which contain just a single training example will cause significant errors in our estimate of the gradient. In fact, though, the errors turn out to not be such a problem. The reason is that the individual gradient estimates don't need to be super­accurate. All we need is an estimate accurate enough that our cost function tends to keep decreasing. It's as though you are trying to get to the North Magnetic Pole, but have a wonky compass that's 10­-20 degrees off each time you look at it. Provided you stop to check the compass frequently, and the compass gets the direction right on average, you'll end up at the North Magnetic Pole just fine.**

**Based on this argument, it sounds as though we should use online learning. In fact, the situation turns out to be more complicated than that.As we know we can use matrix techniques to compute the gradient update for all examples in a mini­batch simultaneously, rather than looping over them. Depending on the details of our hardware and linear algebra library this can make it quite a bit faster to compute the gradient estimate for a mini­batch of (for example) size 100 , rather than computing the mini­batch gradient estimate by looping over the 100 training examples separately. It might take (say) only 50 times as long, rather than 100 times as long.Now, at first it seems as though this doesn't help us that much.**

**With our mini­batch of size 100 the learning rule for the weights looks like:[](https://i.sstatic.net/QDhgB.png)**

**where the sum is over training examples in the mini­batch. This is versus[](https://i.sstatic.net/X4CkK.png)  
for online learning. Even if it only takes 50 times as long to do the mini­batch update, it still seems likely to be better to do online learning, because we'd be updating so much more frequently. Suppose, however, that in the mini­batch case we increase the learning rate by a factor 100, so the update rule becomes  
[](https://i.sstatic.net/KMnF1.png)  
That's a lot like doing separate instances of online learning with a learning rate of η. But it only takes 50 times as long as doing a single instance of online learning. Still, it seems distinctly possible that using the larger mini­batch would speed things up**.

**5. Why we adopt regularization to avoid overfitting?**

**Regularization is adopted to avoid overfitting by adding a penalty to the model's complexity, which discourages the model from fitting the noise in the training data. Overfitting occurs when a model learns the training data too well, including its noise and outliers, leading to poor generalization to new, unseen data.**

**Types of Regularization:**

1. **L1 Regularization (Lasso)**
   * **Definition: Adds the absolute value of the magnitude of coefficients as a penalty term to the loss function.**
   * **Criteria:**
     + **Encourages sparsity (many coefficients become zero).**
     + **Useful for feature selection.**
2. **L2 Regularization (Ridge)**
   * **Definition: Adds the squared magnitude of coefficients as a penalty term to the loss function.**
   * **Criteria:**
     + **Encourages small coefficients.**
     + **Does not necessarily lead to sparsity.**

**Mathematical Expressions:**

**For a linear regression model, the regularized loss functions are:**

**L1 Regularization:**

**L=∑((yi−(w×xi+b))2)+𝜆×∑(|wj|)**

**L2 Regularization:**

**L=∑((yi−(w×xi+b))2)+𝜆×∑(wj2)**

**Example in Python:**

**Python**

**from sklearn.linear\_model import Lasso, Ridge**

**# Lasso (L1 Regularization)**

**lasso = Lasso(alpha=0.1)**

**lasso.fit(X\_train, y\_train)**

**# Ridge (L2 Regularization)**

**ridge = Ridge(alpha=0.1)**

**ridge.fit(X\_train, y\_train)**

**By applying regularization, we aim to find a balance between fitting the training data well and maintaining the model's ability to generalize to new data.**

**6. What are loss and cost functions in deep learning?**

**Loss and cost functions are essential components in deep learning, used to measure how well a model's predictions match the actual data.**

**Loss Function:**

* **Definition: A loss function quantifies the difference between the predicted output and the actual output for a single data point.**
* **Attributes:**
  + **Required:**
    - **Predicted value (y^)**
    - **Actual value (y)**
  + **Variable:**
    - **Type of loss function (e.g., Mean Squared Error, Cross-Entropy)**

**Cost Function:**

* **Definition: A cost function aggregates the loss over the entire dataset, providing a single scalar value that represents the model's performance.**
* **Attributes:**
  + **Required:**
    - **Loss function**
    - **Dataset**
  + **Variable:**
    - **Type of cost function (e.g., Sum of Losses, Mean of Losses)**

**Examples:**

1. **Mean Squared Error (MSE):**
   * **Loss Function: Measures the average squared difference between predicted and actual values.**
   * **Formula:**

**L(y,y^)=(y−y^)2**

* + **Cost Function: Average of the MSE over all data points.**
  + **Formula:**

**J=1N∑i=1N(yi−y^i)2**

1. **Cross-Entropy Loss:**
   * **Loss Function: Measures the difference between two probability distributions for a classification task.**
   * **Formula:**

**L(y,y^)=−∑i=1Cyilog⁡(y^i)**

* + **Cost Function: Average of the cross-entropy loss over all data points.**
  + **Formula:**

**J=−1N∑i=1N∑j=1Cyijlog⁡(y^ij)**

**Python Example:**

**Python**

**import numpy as np**

**# Mean Squared Error Loss Function**

**def mse\_loss(y\_true, y\_pred):**

**return np.mean((y\_true - y\_pred) \*\* 2)**

**# Cross-Entropy Loss Function**

**def cross\_entropy\_loss(y\_true, y\_pred):**

**return -np.sum(y\_true \* np.log(y\_pred)) / y\_true.shape[0]**

**In summary, loss functions evaluate individual predictions, while cost functions aggregate these evaluations over the entire dataset to guide the optimization process.**

**7. What do ou mean by underfitting in neural networks?**

**Underfitting in neural networks occurs when a model is too simple to capture the underlying patterns in the data. This results in poor performance on both the training and test datasets.**

**Required attributes for identifying underfitting:**

* **High training error**
* **High validation/test error**
* **Model complexity is too low (e.g., too few layers or neurons)**

**Variable attributes that may indicate underfitting:**

* **Insufficient training time**
* **Inadequate feature selection**
* **Over-regularization (e.g., too high dropout rate or weight decay)**

**Distinction between underfitting and overfitting:**

* **Underfitting: Model is too simple, fails to capture data patterns.**
* **Overfitting: Model is too complex, captures noise along with data patterns.**

**Criteria for underfitting:**

* **Training error is high.**
* **Validation/test error is high.**
* **Simplistic model architecture.**

**Criteria for overfitting:**

* **Training error is low.**
* **Validation/test error is high.**
* **Complex model architecture.**

**Example of a simple neural network in Python that might underfit:**

**Python**

**import tensorflow as tf**

**from tensorflow.keras.models import Sequential**

**from tensorflow.keras.layers import Dense**

**# Simple neural network with potential to underfit**

**model = Sequential([**

**Dense(10, activation='relu', input\_shape=(input\_dim,)),**

**Dense(1, activation='linear')**

**])**

**model.compile(optimizer='adam', loss='mean\_squared\_error')**

**model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_val, y\_val))**

**In this example, the model has only one hidden layer with 10 neurons, which might be too simple for complex datasets, leading to underfitting.**

**8. Why we use Dropout in Neural Networks?**

**Dropout is used in neural networks to prevent overfitting. Overfitting occurs when a model learns the training data too well, including its noise and outliers, which negatively impacts its performance on new, unseen data. Dropout addresses this by randomly "dropping out" (i.e., setting to zero) a fraction of the neurons during training, which forces the network to learn more robust features.**

**Key attributes of Dropout:**

* **Rate (p): The probability of dropping out a neuron. Common values are between 0.2 and 0.5.**
* **Training Phase: Dropout is only applied during training, not during evaluation or testing.**

**Criteria for using Dropout:**

1. **Prevent Overfitting: Helps in generalizing the model by preventing it from becoming too reliant on specific neurons.**
2. **Improves Robustness: Forces the network to learn redundant representations, making it more robust to changes in the input data.**

**Example in Python using Keras:**

**Python**

**from keras.layers import Dropout**

**# Adding a Dropout layer with a dropout rate of 0.5**

**model.add(Dropout(0.5))**

**Mathematically, if h is the output of a neuron before dropout and p is the dropout rate, the output after dropout h′ is:**

**h′=h×r**

**where r is a random variable that is 0 with probability p and 1 with probability 1−p.**