IDC410
A course on Image Processing and Machine Learning
(Lecture 09)

Shashikant Dugad,

IISER Mohali

Reading Material

Suggested Books:

1. **Neural Networks and Deep Learning by Michael Nielsen**
2. **Fundamentals of Deep Learning by Nikhil Buduma**

Source for this presentation:

Neural Networks and Deep Learning by Michael Nielsen

https://www.scaler.com/topics/deep-learning/introduction-to-feed-forward-neural-network/

https://www.turing.com/kb/mathematical-formulation-of-feed-forward-neural-network

http://machine-learning-for-physicists.org. by Florian Marquardt

3Blue1Brown (Youtube Videos)

https://www.datacamp.com/tutorial/introduction-to-activation-functions-in-neural-networks

Activation Functions

Activation Function for a Neuron and Perceptrons

- **Without activation functions, neural networks would just consist of linear operations like matrix multiplication. All layers would perform linear transformations of the input, and no non-linearities would be introduced.**
- **A neural network without an activation function would essentially behave like a simple linear regression model, meaning it can only learn linear relationships between inputs and outputs, unable to capture complex patterns in real-world data due to the lack of handling non-linearity limiting its ability to solve intricate problems like image recognition or natural language processing etc.**
- **Activation functions are the building block of neural networks, enabling to learn complex patterns in data.**
- **Introducing activation functions can mitigate this problem and enable neural networks to learn non-linear behaviours of the input. This greatly increases the flexibility and power of neural networks to model complex data**
- **There are several type of activation functions, each having its own unique properties and is suitable for certain use cases.**

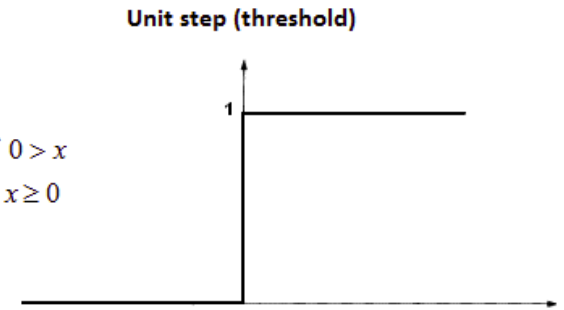
Activation Function for a Neuron and Perceptrons

1. A biological neuron only fires when a certain conditions satisfied
2. Similarly, the artificial neuron will also fire, only when the sum of the inputs (weighted sum) exceeds a certain threshold value, say 0
3. This was the reason for choosing a ***Unit Step (Threshold) function*** as an ***activation function***, originally used by Rosenblatt
4. A smoother version of the above function such as the ***sigmoid*** function, the ***Hyperbolic tangent*** (tan*h*) function are preferred
5. Both ***sigmoid*** and ***tanh*** functions suffer from vanishing gradients problems
6. ***ReLU*** and ***Leaky ReLU*** are the most popularly used activation functions. They are comparatively stable over deep networks.

Unit Step Activation Function

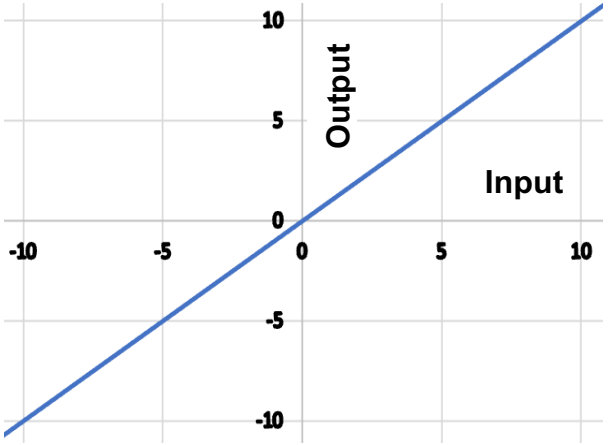
- When the weighted sum of inputs is passed through a step activation function; the perceptron outputs 1 if the sum exceeds a certain threshold,; otherwise, it outputs 0
- The primary advantage of a unit step activation function is its simplicity for binary classification tasks, as it directly outputs a clear **0 or 1** depending on whether the input is below or above a threshold, making it ideal for situations where you need a distinct yes/no decision;
- The primary disadvantage of the unit step activation function is that it has a zero gradient for almost all inputs, which significantly hinders the learning process during backpropagation in a neural network, making it very difficult to train the model effectively
- It can only produce binary outputs (0 or 1), limiting its usability in multi-class classification problems
- The binary nature of the output prevents the network from learning complex non-linear relationships in data

$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$



Linear Activation Function

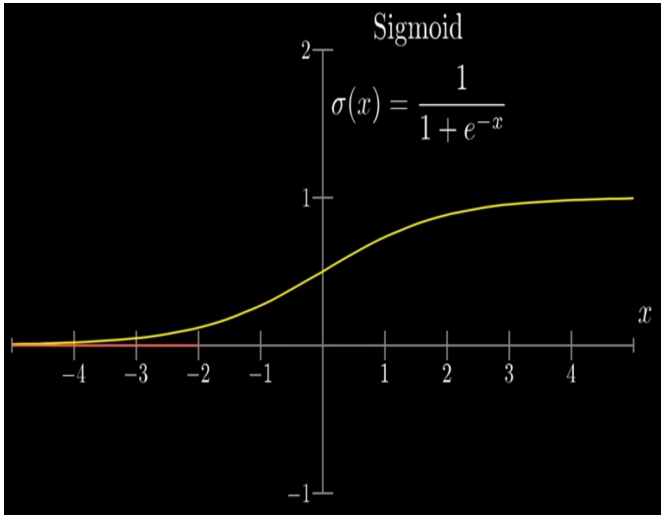
- The **linear activation function** is the simplest activation function, defined as: **y = f(x) = x** and it directly returns the sum of weighted input as the output. Graphically, it's a straight line with a slope 1 and intercept 0
- The main use case of the **linear activation function** is in the output layer of a neural network used for regression where we want to predict a numerical value, using a linear activation function in the output layer ensures the neural network outputs a numerical value. The linear activation function does not squash or transform the output, so the actual predicted value is returned.
- However, the **linear activation function** is rarely used in hidden layers of neural networks. This is because it does not handle any non-linearity. The whole point of hidden layers is to learn non-linear dynamics of the input features. Using a linear activation throughout would restrict the model to just learning linear transformations of the input.



Sigmoid Activation Function

- The sigmoid activation, σ(x), is smooth and continuously differentiable function with following mathematical form:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

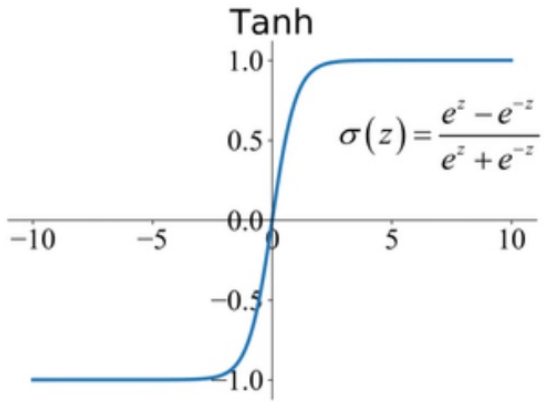


- The sigmoid output varies between 0 to 1 and hence can be easily interpreted as probabilities, making it potential candidate for binary classification problems.
- Sigmoid has strongest gradient near σ~0.5, allowing efficient backpropagation training. However, sigmoid suffer from the ***vanishing gradient*** towards extreme end of σ, slowing down the learning process.
- The main use case of the sigmoid function is the activation for the output layer of binary classification models. It squashes the output to a probability value between 0 and 1, which can be interpreted as the probability of the input belonging to a particular class.

Hyperbolic Tangent (tanh) Activation Function

- The **tanh** (hyperbolic tangent) activation function is also smooth and continuously differentiable function with following mathematical form:

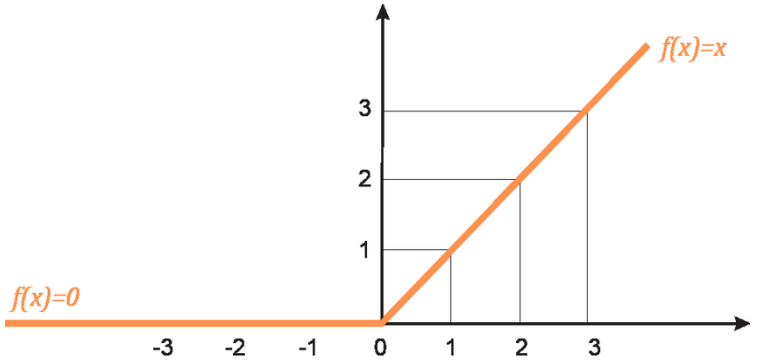
$$\tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



- The **tanh** output varies between -1 to 1 and hence it can deal effectively with negative values and also provides stronger gradients compared to sigmoid resulting in faster learning and convergence and also more resilient to ***vanishing gradient***
- The **tanh** being symmetric around the origin has an advantage for faster convergence of learning algorithm
- The **tanh** function still suffers from the ***vanishing gradient*** problem. This issue is particularly problematic for deep networks with many layers;
- The **tanh** function is frequently used in the hidden layers of a neural network due to its zero-centered nature, when the data is also normalized to have mean zero, it can result in more efficient training.

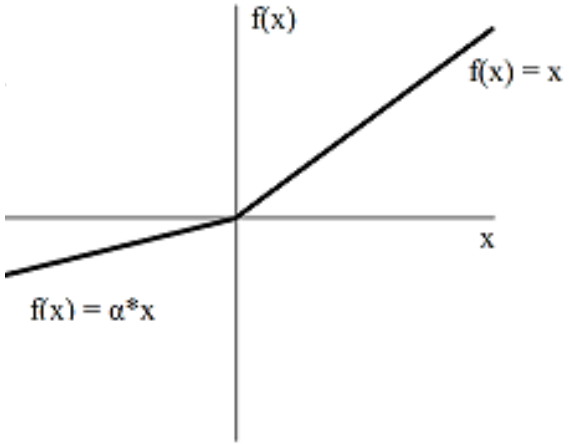
Rectified Linear Unit (ReLU) Activation Function

- The **Rectified Linear Unit (ReLU) activation function** has the form: **f(x) = max(0, x)**. Output is 0 for negative input (f(x)=0 for x<0) and output is same as input for positive input (f(x)=x for x>=0)
- Vinod Nair and Geoffrey Hinton introduced it in their research paper titled "Rectified Linear Units Improve Restricted Boltzmann Machines" published in 2010.
- For inputs>0, **ReLU** acts as a linear function with a constant gradient of 1, thus, it does not alter the scale of positive inputs and allows the gradient to pass through unchanged during backpropagation without having the **vanishing gradient** problem
- The **ReLU** is technically a non-linear function because it has a non-differentiable point at x=0, enabling neural networks to learn complex patterns
- Since **ReLU** outputs zero for x<0, it leads to sparse activations of neurons leading to more efficient computation.
- **ReLU** being computationally inexpensive, mostly used in hidden layers, allows networks to scale to many layers without a significant increase in computational burden, compared to more complex functions like **tanh** or **sigmoid**.



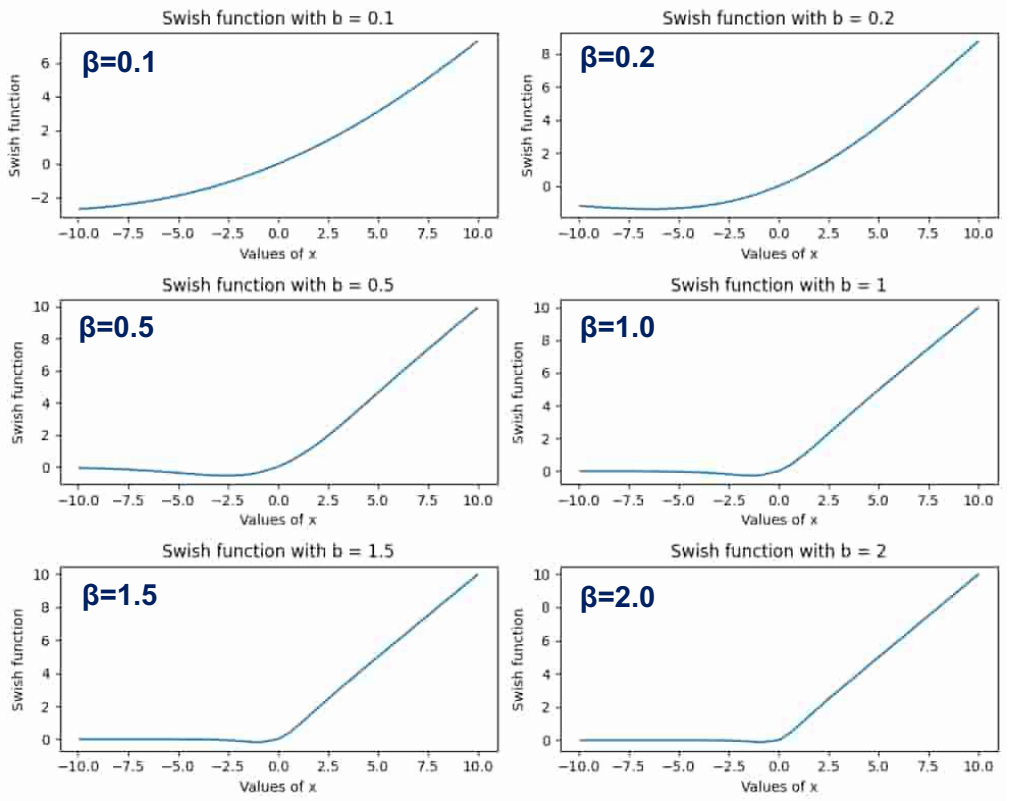
Leaky Rectified Linear Unit Activation Function

- **Leaky ReLU activation function** is just ReLU except that it introduces a small slope for negative values to address the **dying ReLU** problem by allowing a small gradient for negative inputs
- It ensures neurons with negative values can contribute to the output even which can improve the performance of deep neural networks, especially in scenarios with many negative inputs; whereas ReLU can potentially cause neurons to become inactive for negative inputs, hindering learning
- It's slightly more computationally expensive to calculate as compared to standard ReLU
- The functional form of **Leaky ReLU** is:
  f(x) = max(αx, x), where α<<1
- In **Leaky ReLU**, the parameter **α** is constant chosen before training
- **Parametric ReLU (PReLU)** activation function the parameter **α** is kept as a learnable parameter to evolve the slope of the negative values during training



Swish Activation Function

- The **Swish activation function** is a slight modification of the sigmoid function. The mathematical formula for this is: **Swish(x)=x∗sigmoid(βx),** where **β** is a scalable and trainable parameter.
- The **Swish activation function**, unlike the **ReLU** function does not output 0 for all the negative values while it also maintains shape of **ReLU** for the positive values of x
- **Swish function** is a powerful activation function which is quite useful in DNN for classification tasks (Used in YOLOv5)



Softmax Activation Function

- The **Softmax activation function** (normalized exponential function), is useful for multi-class classification problems. It operates on a vector, often referred to as the logits, which represents the raw predictions or scores for each class computed by the previous layers of a neural network.

- For input vector x with elements x_1, x_2, ..., x_C (C is no. of classes), the **softmax** function is defined as:

$$f(x_i) = \frac{e^{xi}}{\sum_{j=1}^{C} e^{x_j}}$$

- **Softmax** function is a probability distribution that amplifies differences in the input vector and each class represents the probability that the input belongs to a particular class

- **Softmax** is used in the output layer of a neural network for classifying an input into one of several (more than two) possible categories (multi-class classification).

- Since **softmax** exponentially amplifies differences, therefor, it can be sensitive to outliers or extreme values.

Network Configuration Terminologies

- **Loss (or Cost Function:** The loss function which is basically an error function is calculated using **Mean Squared Error (MSE)** function for regression tasks or **Cross-Entropy Loss** function for classification tasks.
- **Gradient Descent:** It is exploited in an optimization algorithm for minimizing the loss function by iteratively updating the weights in the direction of the negative gradient.
- **Backpropagation:** The error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated, and the weights are adjusted using gradient descent
- **Stochastic Gradient Descent (SGD):** Updates weights for each training event individually.

Network Configuration Terminologies

- **Batch Gradient Descent:** Updates weights after computing the gradient over the entire dataset.
- **Mini-batch Gradient Descent:** Updates weights after computing the gradient over a small batch of training examples.

Evaluating the Performance of the trained FFN model involves several metrics

- **Accuracy:** The proportion of correctly classified instances out of the total instances.
- **Precision:** The ratio of true positive predictions to the total predicted positives.
- **Recall:** The ratio of true positive predictions to the actual positives.
- **F1 Score:** The harmonic mean of precision and recall, providing a balance between the two.
- **Confusion Matrix:** A table used to describe the performance of a classification model, showing the true positives, true negatives, false positives, and false negatives

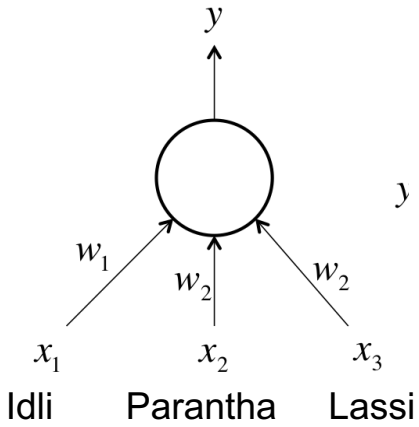
Network Training

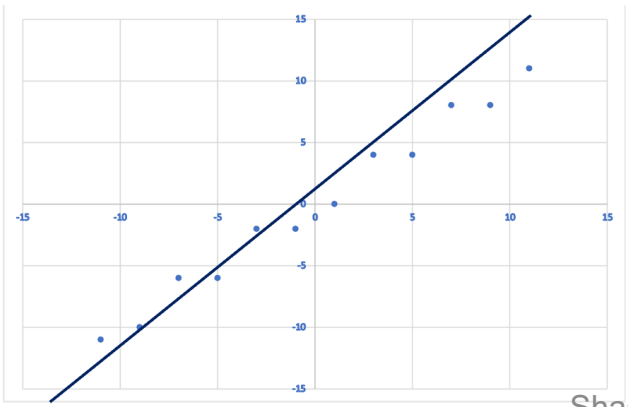
Training Feed-Forward Neural (FFN) Networks

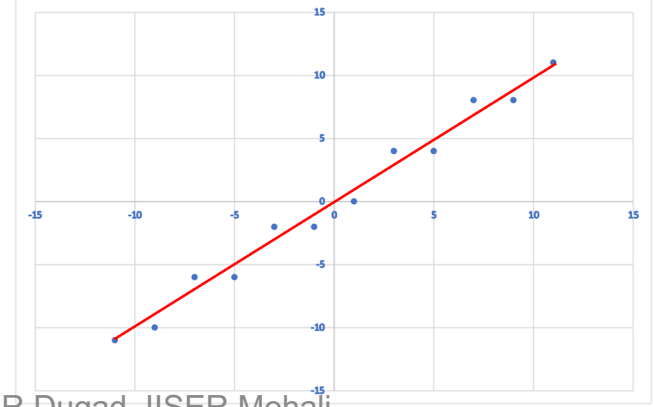
- Let us try to understand the concepts of training using a simple neural network with linear activation function (output == input)

$$y^i = w_1 x_1^i + w_2 x_2^i + w_3 x_3^i$$

- Known parameters during training: # of Idli (x_1), Parantha (x_2), Lassi (x_3) purchased and total cost (y^i) associated with i^{th} purchase
- Let us assume purchases of these items were done large # of times (i >> 1)
- Task: Determine the cost of each item (w_1, w_2 and w_3)







Training Feed-Forward Neural (FFN) Networks

- If t_i is the true total cost and y_i is the total cost predicted by the network for i^{th} purchase for a given value of weights; then, we want to tune each of these weights in such a way that difference between predicted and true cost ($|t_i - y_i|$) is as small as possible!
- We shall choose those weights as our final weights, when the overall difference between predicted and true cost is **smallest** across all the purchases:

The **Loss Function** is defined as: $E = \frac{1}{2n}\sum_{i=1}^{n}(t_i - y_i)^2$

- As a result, our goal will be to select our parameter vector θ (the values for all the weights in our model) such that, the loss function E is as close to 0 as possible
- This problem can be easily solved in exact manner with three purchases of different types
- However, situation changes completely with non-linear activation function