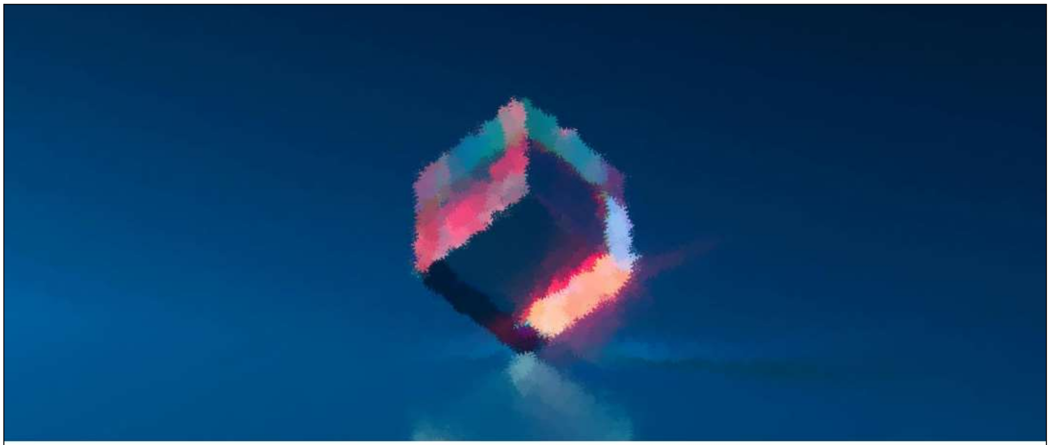


Introduction to Deep Learning

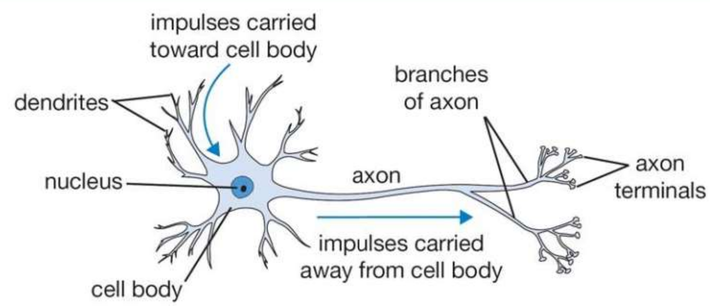
Course Instructor: Dr. M. Umair

Topic: Deep Learning Topics of Interest

Agenda

Artificial Neural Networks

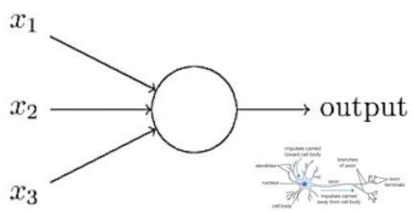
Artificial Neural Networks



An illustration of biological neuron

- The basic computational unit of the brain is a *neuron*.
- Approximately *86 billion* neurons can be found in the human nervous system.
- They are connected with approximately 10ˆ14 - 10ˆ15 *synapses*.

Artificial Neural Networks



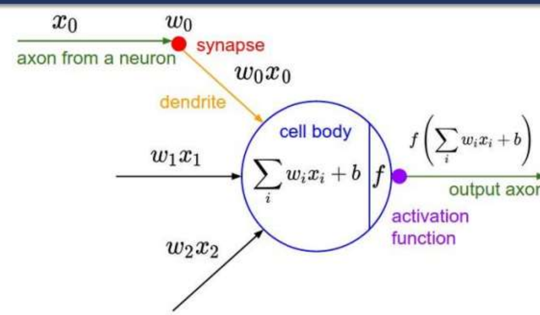
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \le \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

A perceptron

Perceptron's mathematical representation

- A *perceptron* is one of the fundamental building blocks of artificial neural networks.
- It is a type of *artificial neuron* which takes
  - Multiple inputs,
  - Applies weights to those inputs, and
  - Produces an output based on a specified activation function.

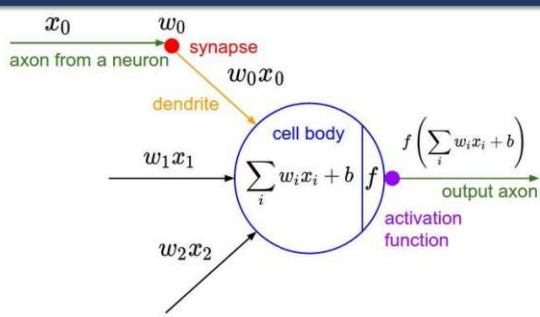
Artificial Neural Networks



A common mathematical model of neuron

- In mathematical model of neuron, the signals that travel along the axons is represented by *x0, x1,* and *x2*.
- A *synaptic strength* at synapse is known as *w0*.
- The incoming signal interact multiplicatively (e.g. *w0.x0*) with the dendrites of the other neuron.
- The idea is that the synaptic strengths (i.e. *w*) are *learnable* & *control* the strength of *influence* of one neuron on another.

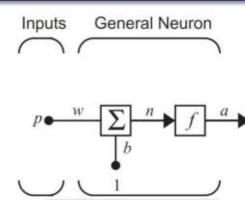
Artificial Neural Networks



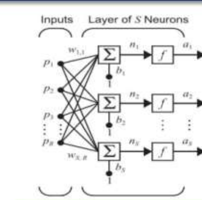
A common mathematical model of neuron

- So, the dendrites carry the signal to the cell body where they all get *summed*.
- If the final sum is above a *certain threshold*, the neuron can *fire*, sending a *spike* along its axon.
- The *firing rate* of the neuron is modeled with an activation function *f*.
- *f* represents the frequency of the spikes along the axon.

Artificial Neural Network



Singal Input Neuron

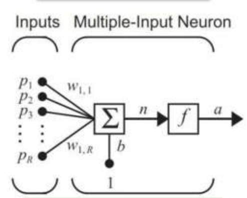


Single Layer of S Neurons

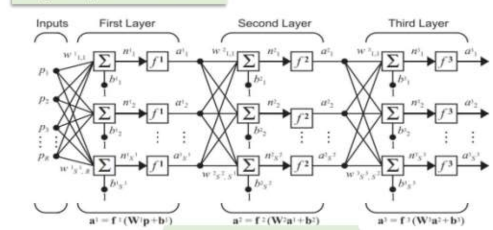
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Weight Matrix **W**

W(2,1) is to 2nd Neuron from 1st Source

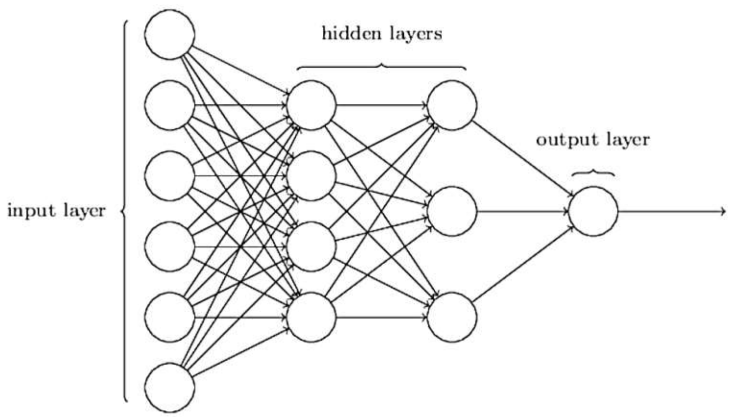


Multiple Inputs Neuron



Three Layers Network

Artificial Neural Network



Weights

Weights

Introduction

- At the *heart of every solution is a model* that describes how features can be transformed into an estimate of the target.
- The **weights** determine the *influence* of each *feature* on our *prediction*.

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b.$$

- Collecting all *features* into a vector $\mathbf{x} \in \mathbb{R}^d$ and all *weights* into a vector $\mathbf{w} \in \mathbb{R}^d$, we can express our model compactly via the dot product between $\mathbf{w}$ and $\mathbf{x}$.

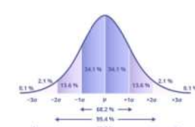
$$\hat{y} = \boldsymbol{w}^{\mathrm{T}}\mathbf{x} + \mathrm{b}$$

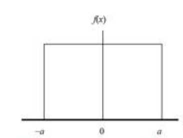
The vector $\mathbf{x}$ corresponds to the features of a single example.

Weights

Some Methods to Initializing Weights

- **Zero Initialization:** Not recommended but common for bias
- **Initialize weights randomly:** Typically, from a normal (Gaussian) or uniform distribution.





- **Glorot Initialization:** Better for *sigmoid* and *tanh* activation functions.
- **He Initialization:** Better for *ReLU* and its variants.

BIAS

BIAS

Introduction

- The BIAS determines the value of the estimate when all features are zero.
- *Given a dataset*, our goal is to *choose* the *weights* w and the *bias* b that, on average, *make our model's predictions fit the true values* observed in the data as closely as possible.
- The *bias* allows the model to *shift the output independently of the input features*, which can be important for fitting data that doesn't pass through the origin.
- Without b, the model's predictions would always be anchored around the origin, reducing flexibility.

BIAS

Introduction

- Imagine a dataset where the *true target value* (output) is *consistently positive*, even when all *features* x_i are *zero*.
- If we didn't have b, the model wouldn't be able to correctly predict a non-zero value for this situation.
- By introducing b, the *model gains the flexibility* to
  - Predict non-zero values even when all feature values are zero or minimal, which broadens the range of possible predictions.

BIAS

Example

- Let's say you're trying to predict house prices, but without b.
- The *model predicts a price of $0 whenever all input features are zero* (e.g., swimming pool, servant quarters, etc.).
- This means that the *model would assume a house has no price unless it has some features*, which *may not make sense*.

Bias

Initialization

- Bias is *initialized* to a *small value*, often *zero* or a *small random number*.
  - *Zero value is much desirable* because it doesn't introduce any initial bias to the model.

During the training process, the ***weights w*** *and* ***bias b*** *are* ***updated*** *along with through an optimization algorithm, usually gradient descent.*

Gradient Descent

Gradient Descent

Introduction

- The *key technique* for optimizing deep learning model consists of *iteratively reducing the error* by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent.*
- The *naivest application of gradient descent* consists of taking the *derivative of the loss function*, which is an average of the losses computed on every single example in the dataset.

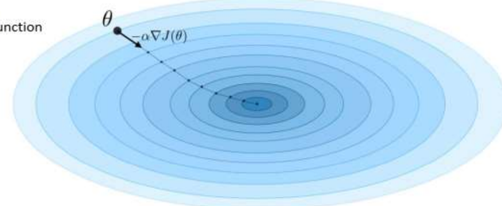
Gradient Descent

Definition

- By noting $\alpha \in \mathbb{R}$ the *learning rate*, the update rule for gradient descent is expressed with the learning rate and the cost function J as follows:

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

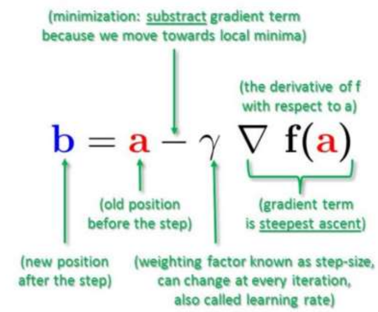
θ - parameters of a model that we want to optimize
α - learning rate
$J(\theta)$ - cost function
$\nabla J(\theta)$ - gradient of the cost function

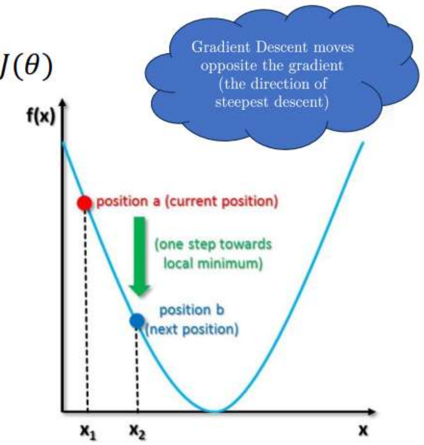


Gradient Descent

Explanation

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$



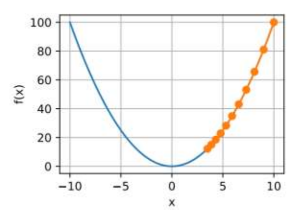


Learning Rate

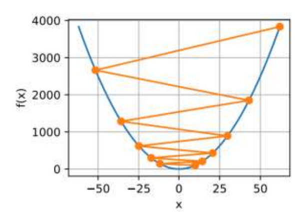
Learning Rate

Examples

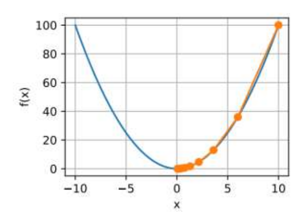
$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$



$\alpha = 0.05$



$\alpha = 1.1$



$\alpha = 0.2$

Loss Function

Loss Function

Introduction

- Loss functions quantify the *distance between* the *real* and *predicted* values of the target.

- The loss will usually be a *nonnegative number* where smaller values are better and perfect predictions incur a loss of 0.

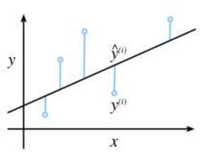
- When training the model, we *seek* parameters $(\mathrm{w}^*\,;\,\mathrm{b}^*)$ that *minimize* the *total loss* across all training examples.

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}}\ L(\mathbf{w}, b)$$

Loss Function

Loss Function - Example

- *For regression problems*, the most *common loss function* is the *squared error*.
- When our *prediction* for an *example i* is $\hat{y}^{(i)}$ and the corresponding *true label* is $y^{(i)}$, the squared error is given by:



$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

Including the $\frac{1}{2}$ factor in the formula simplifies the gradient calculations, as the derivative of $\frac{1}{2}x^2$ with respect to x is just x.

- To *measure* the quality of a model on the *entire dataset* of *n* examples, we simply average the losses on the training set:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}}\ L(\mathbf{w}, b) \Longrightarrow L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

Backpropagation uses this *error measurement* to *update* the *model parameters*, aiming to minimize this error.

Back Propagation
A Gentle Introduction

Back Propagation

How it works?

In each layer, the *neurons apply a weighted sum of their inputs and an activation function* to generate outputs.

For each weight, backpropagation *computes* a *gradient*, which tells us the direction and magnitude of change to reduce the loss.

Forward Pass: Making a Prediction → Calculating the Error (Loss) → Going Backward: Computing the Gradient → Updating the Weights

Loss function quantifies *how far the prediction is from the actual label.*

Using gradients, *update the weights* in the direction that reduces the loss. A *high learning rate* means we make *large adjustments* to weights. A *low learning rate* means smaller, more cautious adjustments.

Activation Function

Activation Function

Introduction

- **Activation Function** (AF) decide whether a *neuron should be activated or not* by calculating the weighted sum and further adding bias to it.
- Activation functions introduce *non-linear properties* into the system allowing the network to learn from complex data.
  - **Note:** *Without non-linear activation functions, your neural network would essentially become a simple linear regression model, incapable of learning complex functions.*
- Activation functions are *mathematical formulas* that dictate the output of a neuron given a certain input.

Activation Function

Introduction

- AF act as the "*gatekeepers*" of each node, deciding *how much signal should pass* through to the next layer.
- When *designing* or *fine-tuning* a neural network, choosing the *right activation function* can significantly impact the model's performance.
- The *choice of activation function can make or break* your network's ability to learn effectively.

Activation Function

Introduction

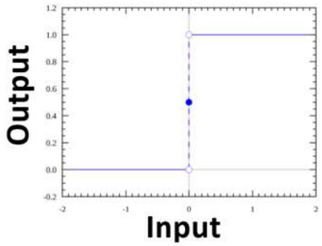
- $\mathbf{Y} = \sum(\boldsymbol{x}.\boldsymbol{w}) + \boldsymbol{b}$
- The resulting value of **Y** can range anywhere from *negative infinity* to *positive infinity.*
- How can we determine if a *neuron* should *fire or not?*

Activation Function

Introduction

- A *naive approach* to activation might be to simply *set a threshold*:

IF Y IS ABOVE A CERTAIN VALUE, WE DECLARE THE NEURON ACTIVATED



Activation Function

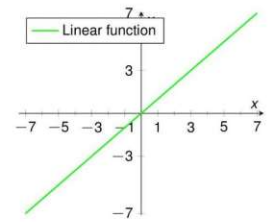
Introduction

- **Issue:** Such a *naive approach* is *not differentiable*, hence *can't use backpropagation* to adjust the weights and biases during the learning phase.
- Therefore, we *need* smooth, *differentiable activation functions*.

Differentiable Activation Functions

Differentiable Activation Functions

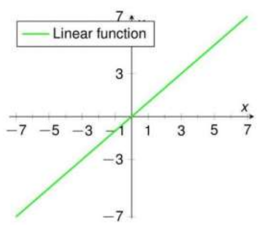
- **Linear Activation Function**



  - $f(x) = \propto x$
  - $f'(x) = \propto$
  - Linear activation is simple.
  - Issue: It does not introduce non-linearity.
  - Issue: It renders the optimization problem convex.

Differentiable Activation Functions

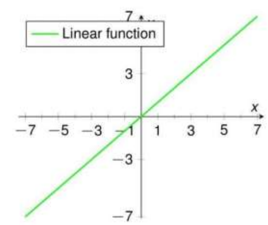
- **Linear Activation Function**



  - Convex optimization problem
    - $y = w.x + b$
    - The objective is to reduce loss.
    - The loss function is $L(\hat{y}, y)$.

Differentiable Activation Functions

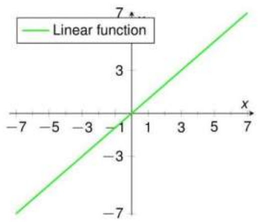
- **Linear Activation Function**



  - Convex optimization problem
  - Consider a *Mean Squared Error* Loss Function
    - $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
    - In the case of a linear activation function, the output y is a linear function of the network parameters w and b.

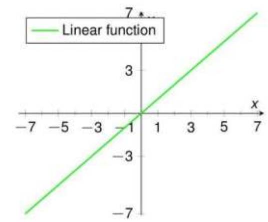
DIFFERENTIABLE ACTIVATION FUNCTIONS

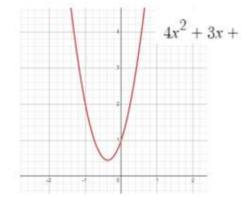
- **Linear Activation Function**
  - CONVEX OPTIMIZATION PROBLEM
    - $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$
    - $L(\hat{y}, y) = \frac{1}{2}(w.x + b - y)^2$
    - When you square a *linear expression*, you get a quadratic expression.
    - This loss function is *quadratic.*



DIFFERENTIABLE ACTIVATION FUNCTIONS

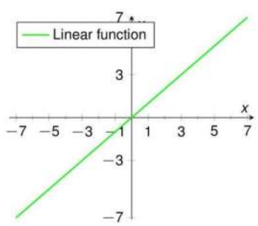
- **Linear Activation Function**
  - CONVEX OPTIMIZATION PROBLEM
    - We know, *quadratic equations are convex.*
    - The *key property* that makes quadratic functions convex is that they have a *single global minimum* or *maximum.*

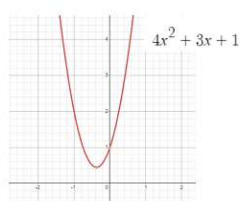




DIFFERENTIABLE ACTIVATION FUNCTIONS

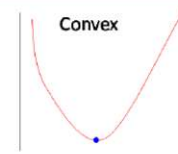
- **Linear Activation Function**
  - CONVEX OPTIMIZATION PROBLEM
    - **ISSUE:** If the loss landscape is *dominated by local minima* and lacks a global minimum, optimization algorithms may *converge to suboptimal solutions.*

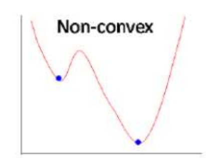


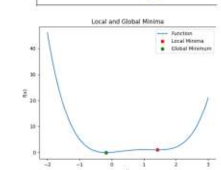


DIFFERENTIABLE ACTIVATION FUNCTIONS

- **Solution: Non-Linear Activation Function**
  - *Non-linear activation functions* help make the optimization landscape more *non-convex.*
  - This can potentially lead to a *greater number of pathways towards the global minimum.*







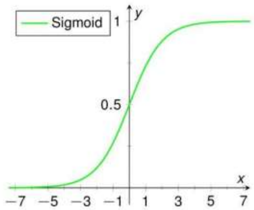
Differentiable Activation Functions

- **Sigmoid Activation Function**

  - $\boldsymbol{f(x) = \frac{1}{1+e^{(-x)}}}$
  - $\boldsymbol{f'(x) = f(x)(1 - f(x))}$

  - Close to biological model and *differentiable.*

  - Probabilistic out (0,1).



Differentiable Activation Functions

```python
import numpy as np
import math

def compute_firing_rate(inputs, weights, bias):
    cell_body_sum = np.sum(inputs * weights) + bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))
    return firing_rate

weights = np.array([0.3, 0.1, -0.1])
bias = 0.2
inputs = np.array([0.7, 0.9, 0.5])

firing_rate = compute_firing_rate(inputs, weights, bias)
print(f"Firing rate: {firing_rate:.4f}")
```

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

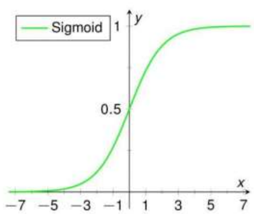
Differentiable Activation Functions

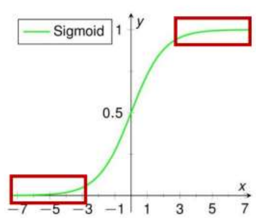
- **Sigmoid Activation Function**

  - **ISSUE:** Saturates for $x \ll 0$ and $x \gg 0$
    - See the *red boxes*
    - *Vanishing gradient problem*

  - **ISSUE:** Not zero-centered

    - The problem: *Always produce positive numbers* independent of what input you get.





Differentiable Activation Functions

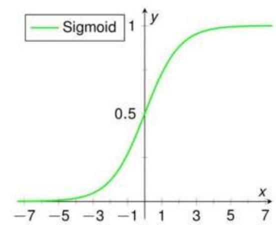
- **Sigmoid Activation Function**

  - **ISSUE:** Not zero-centered

    - This means that if we had a *signal of zero mean as input* into this activation function, it will always be shifted towards a mean that will be greater than zero.

    - This is called the *internal covariate shift* of successive layers.

    - **ISSUE:** The subsequent layers constantly have to adapt to the shifting distribution.

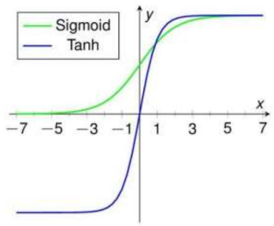


Differentiable Activation Functions

- **Tanh Activation Function**

  - $\boldsymbol{f(x) = \tanh(x)}$
  - $\boldsymbol{f'(x) = 1 - f(x)^2}$

  - Zero-centered
  - A shifted version of the sigmoid function.

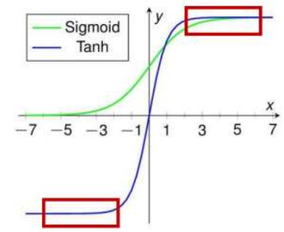


Differentiable Activation Functions

- **Tanh Activation Function**

  - **Issue:** Saturation

    - See the red boxes
    - Vanishing gradient problem



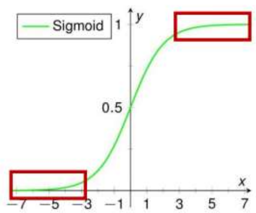
Differentiable Activation Functions

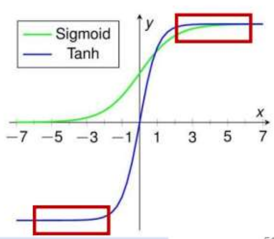
- **Learning from Sigmoid and Tanh Activation Functions**

  - How does x affect y ?
    - Sigmoid and Tanh *map large regions of X* to a *small range in Y.*

    - Large changes in x , minimal changes in y.
    - Gradient vanishes

    - *Problem is amplified* by back-propagation
      - *Multiplication of small gradients*
      - The deeper you build the network, the faster the gradient vanishes.





Differentiable Activation Functions

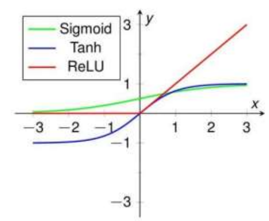
- **ReLU Activation Function**

  - $\boldsymbol{f(x) = \max(0, x)}$
  - $f'(x) = \begin{cases} \mathbf{1} \; \boldsymbol{if\ x > 0} \\ \mathbf{0} \quad \boldsymbol{else} \end{cases}$

  - The idea is simply to set the *negative half-space* to zero and the *positive half-space* to x.

  - Don't have this vanishing gradient problem because we have really *large areas of high values* for the derivative of this function.

  - **Issue:** Not zero-centered.

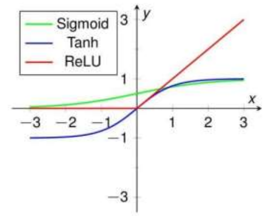


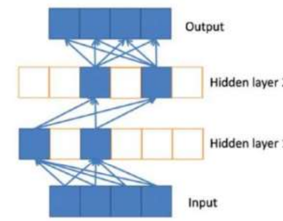
Observation

Typically in classical machine learning, neural networks were limited to approximately three layers because already at this point you get the vanishing gradient problem. The lower layers never seen any of the gradients and therefore never updated their weights.

DIFFERENTIABLE ACTIVATION FUNCTIONS

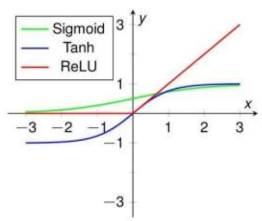
- **ReLU Activation Function**
  - Speed up during learning (i.e., 6x).
  - Good generalization due to *piece-wise linearity.*
  - *Spares-representation* - few elements in a representation vector are non-zero.

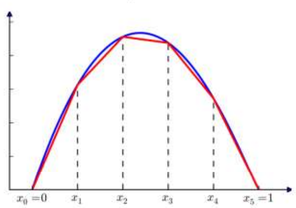




DIFFERENTIABLE ACTIVATION FUNCTIONS

- **ReLU Activation Function**
  - *Constant gradient* for positive inputs
    - Eliminates vanishing gradient problem.
  - Empirical evidence suggests that deep networks with ReLU activation functions can approximate a wide range of non-linear functions.



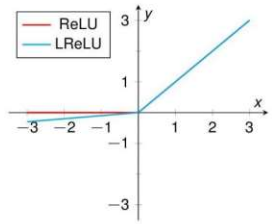


DIFFERENTIABLE ACTIVATION FUNCTIONS

- **ReLU Activation Function**
  - If you have weights and biases trained to yield negative results for x, then you simply always end up with a zero derivative.
  - The ReLU always generates a zero output (for negative results) and this means that they no longer contribute to your training process.

Differentiable Activation Functions

- **Leaky ReLU Activation Function**



  - $f(x) = \begin{cases} x \text{ if } x > 0 \\ \propto x \quad else \end{cases}$
  - $f'(x) = \begin{cases} 1 \text{ if } x > 0 \\ \propto \quad else \end{cases}$
  - Set the negative half-space to a scaled small number.
  - What you get: A very similar effect as the ReLU, but you don't end up with the dying ReLU problem as the derivative is never zero but it's α.

Choosing the Right Activation Function

Choosing the Right Activation Function

- Use the ReLU activation function only in the hidden layers of a neural network.
- Sigmoid/Logistic and Tanh functions should not be used in the hidden layers of a neural network.
  - Vanishing gradient problem
- Some Choices:
  - Regression: Use Linear Activation Function
  - Binary Classification: Use Sigmoid/Logistic Activation Function
  - Multi-class Classification: Use Softmax
  - Multi-label Classification: Use Sigmoid

References

REFERENCES

1. Machine Learning, Deep Learning, Artificial Intelligence. Notes from Stanford University and MIT.
2. Math for Machine Learning, Machine Learning Workshop, Santosh Chapaneri
3. CS231n: Deep Learning for Computer Vision, Stanford University, https://cs231n.github.io/neural-networks-1/
4. Deep Learning – Activation Functions, Bernhard Kainz
5. Dive into Deep Learning, Aston Zhang et al., https://doi.org/10.48550/arXiv.2106.11342
6. Backpropagation, Roger Grosse, University of Toronto
7. Neural Networks and Deep Learning, Michael Nielsen
8. Artificial neural networks, Dr. Najlaa M. Hussein