MODULE 3: BASICS OF LOGISTIC REGRESSION & NEURAL NETWORK

BA713 - Machine Learning & AI

CONTENTS

LOGISTIC REGRESSION OVERVIEW

NEURAL NETWORK OVERVIEW

ACTIVATION FUNCTIONS

LOGISTIC REGRESSION OVERVIEW

- models the probabilities for classification problems with two possible outcomes
- Extension→ Linear Regression model for classification problems
- Linear model does not output probabilities→ treats classes as numbers (0 and 1)
- Linear model→ no meaningful threshold at which you can distinguish one class from the other
- Logistic Regression uses→ logistic function (SIGMOID) → output of a linear equation between 0 and 1

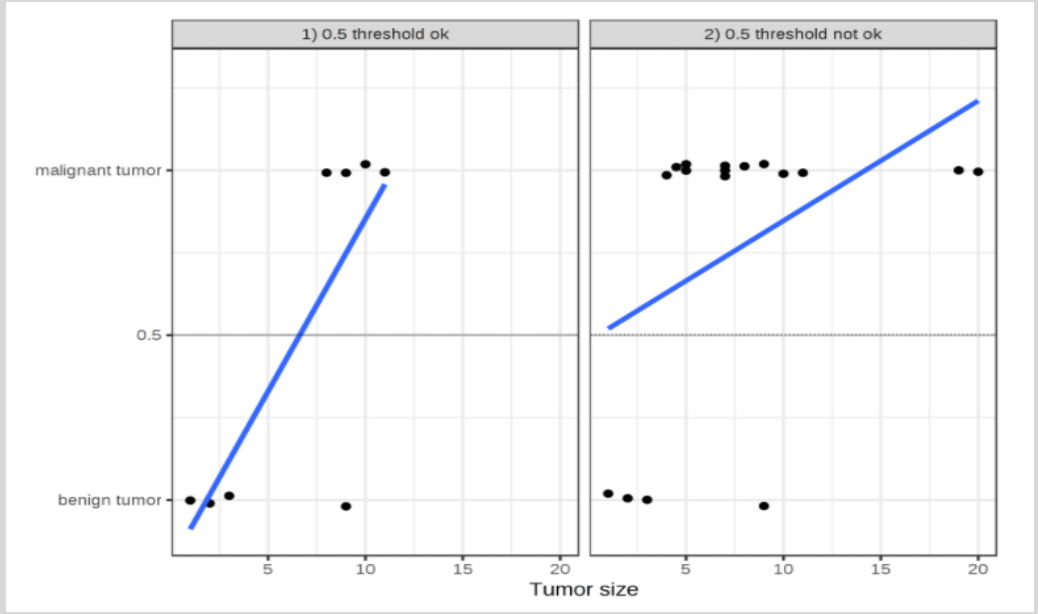


Figure a: Classification using Linear Regression

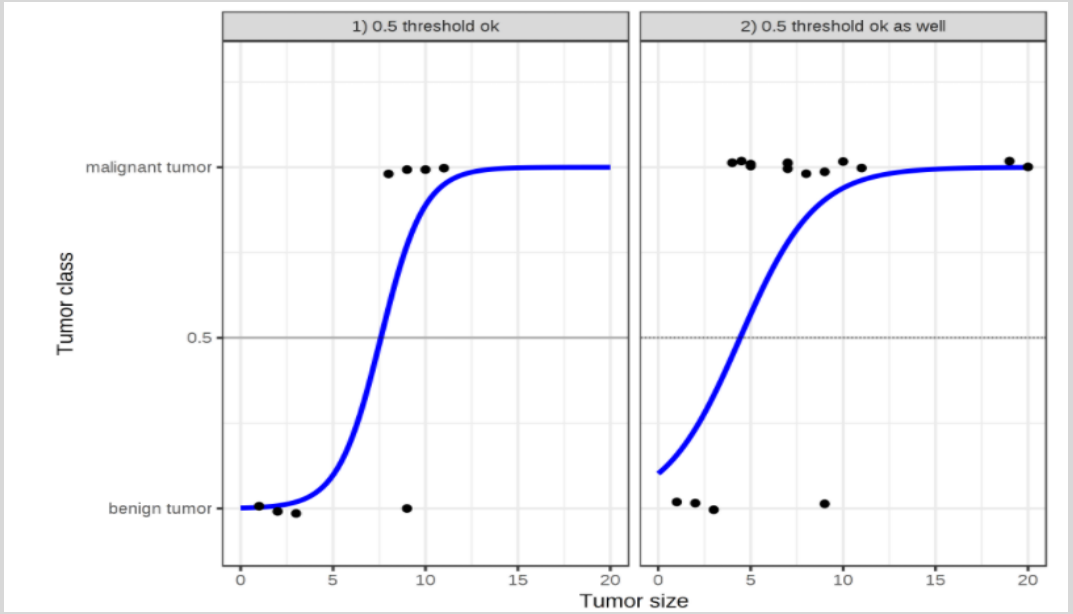


Figure b: Classification between malignant & benign depending on Tumor size based on threshold = 0.5 using Logistic Regression

Source: https://christophm.github.io/interpretable-ml-book/logistic.html

SIGMOID/LOGISTIC FUNCTION

- ensure output that always falls between 0 and 1
- **For example**: predict if a dog will bark during the middle of the night using LR model

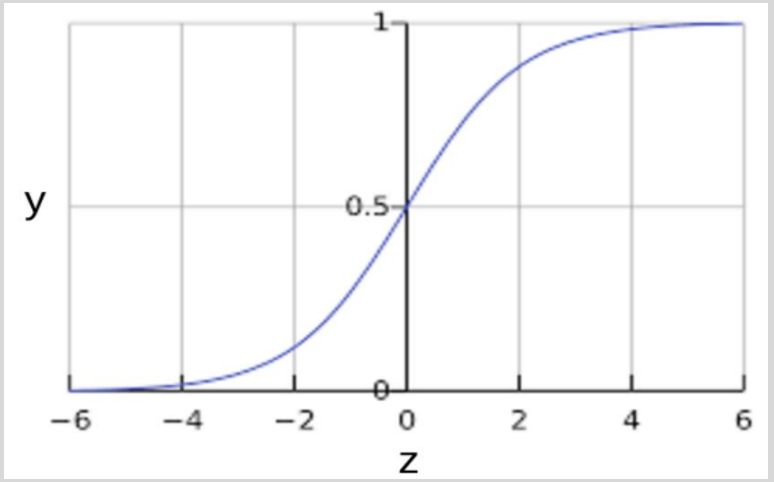


Figure : Sigmoid or Logistic Function

$$y = \frac{1}{1 + e^{-z}}$$

Sigmoid Function Equation

y'→ output of LR model

$$y' = \frac{1}{1 + e^{-z}}$$

z/ logit → output of linear layer of model trained using Logistic Regression

y' or sigmoid(z) yields→ probability between 0 and 1

$$z = b + w_1 x_1 + w_2 x_2 + \ldots + w_N x_N$$

b-bias

w-model's learned weights

x- feature values

log-odds ratio

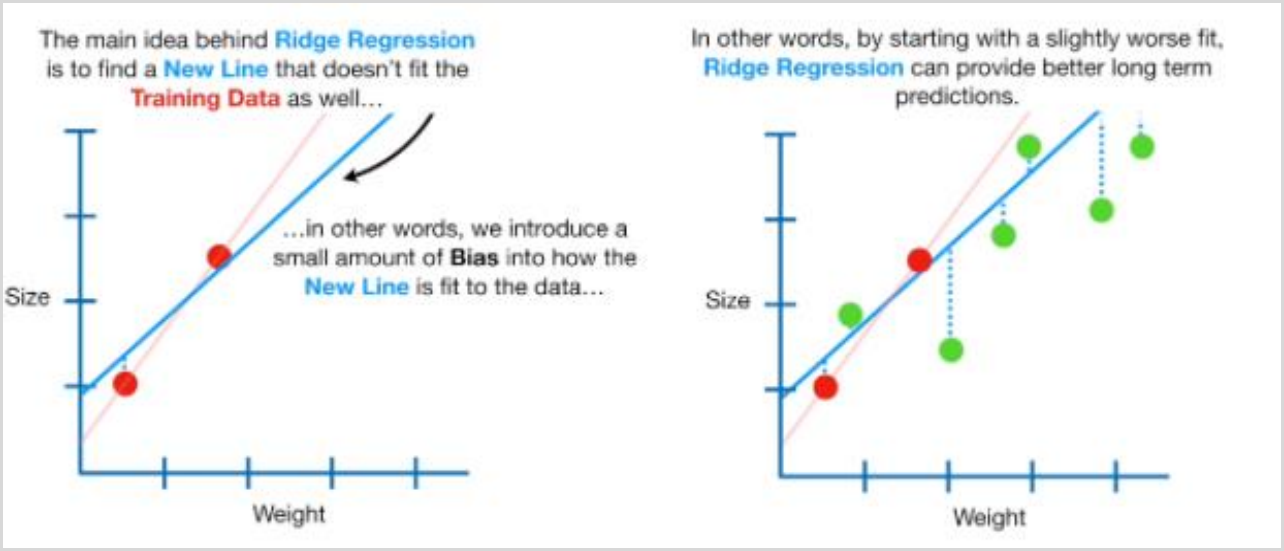
$$z = \log\left(\frac{y}{1 - y}\right)$$

Inverse of sigmoid, z can be defined as log of probability of label 1 (dog barks) divided by probability of label 0 (dog doesn't bark)

Source: https://developers.google.com

LOGISTIC REGRESSION & REGULARIZATION

- Important for Logistic Regression→ too many dimensions overfitting
- **L2 regularization (L2 weight decay)** → penalizes huge weights→ **Ridge regression**
- L2 penalizes→ $(weight)^2$
- **Early stopping** → limiting training steps or learning rate



$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^{\mathsf{T}}\theta)$$

Estimated probability for LR model

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha\frac{1}{2}\Sigma_{i=1}^{n}\theta_i^2$$

Ridge regression cost function with regularization term

σ→ Sigmoid function
θ→ Bias term
x→ feature value
α → hyperparameter controls regularization
MSE→ Mean Squared Error

Source: https://www.andreaperlato.com/theorypost/ridge-and-lasso-regression/

SOFTMAX REGRESSION/ MULTINOMIAL LOGISTIC REGRESSION

- Logistic Regression model can be generalized to support multiple classes directly
- when given an instance x, the Softmax Regression model first computes a score s_k(x) for each class k, then estimates the probability of each class by applying the softmax function (normalized exponential) to the scores
- estimate the probability p_k that the instance belongs to class k by running the scores through the softmax function
- function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials)

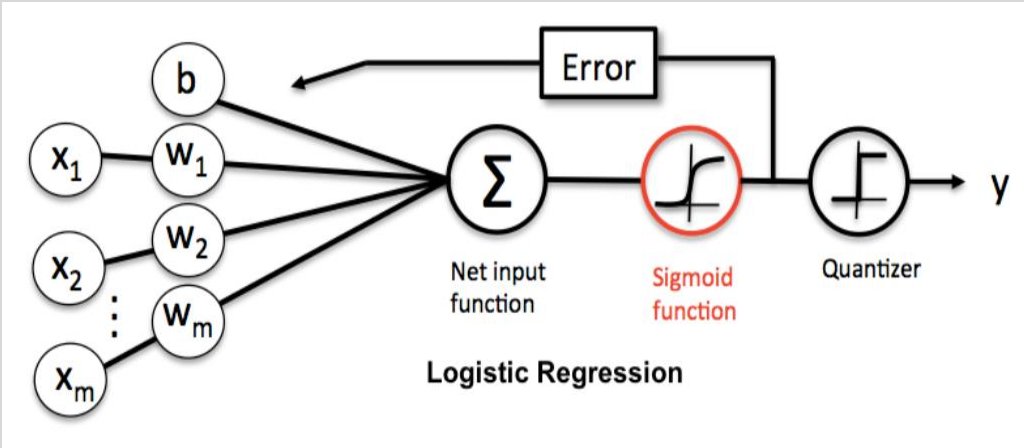
$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

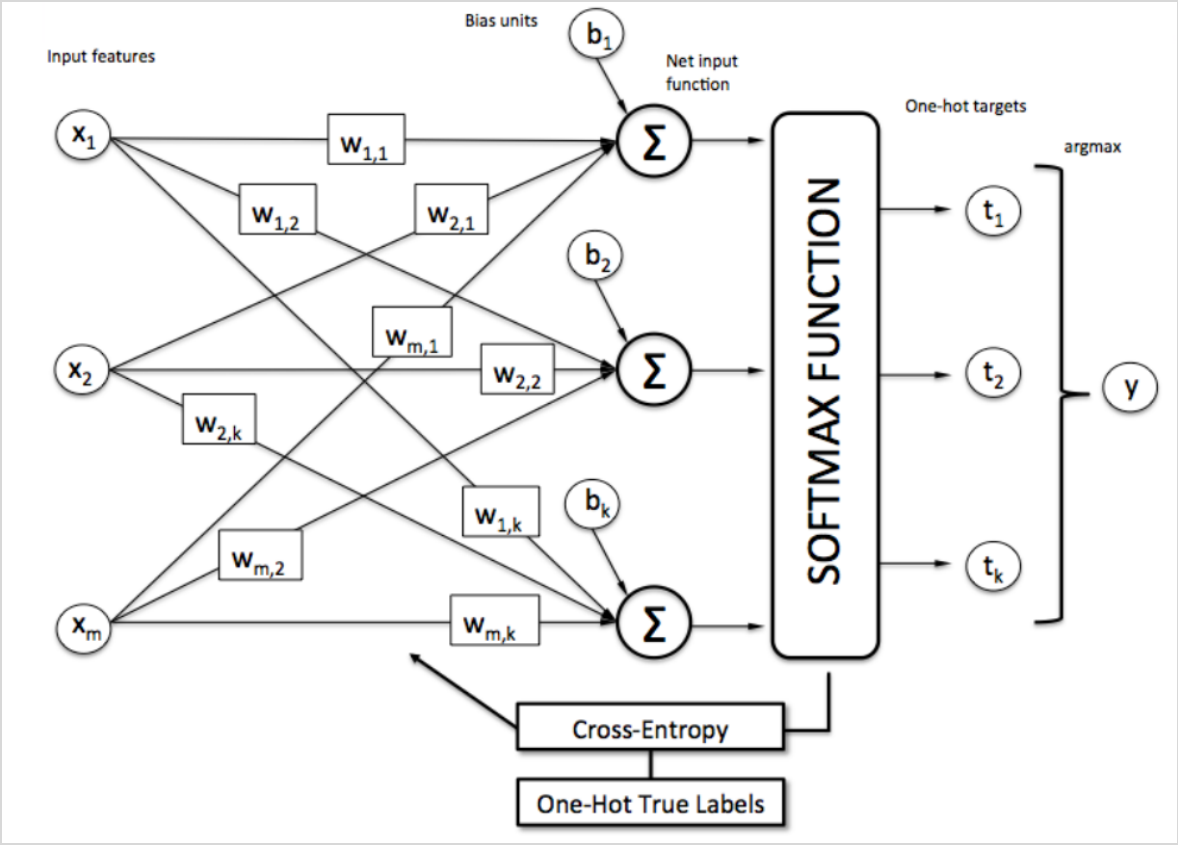
In this equation:

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k, given the scores of each class for that instance.

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

LOGISTIC REGRESSION VS. MULTINOMIAL LOGISTIC REGRESSION



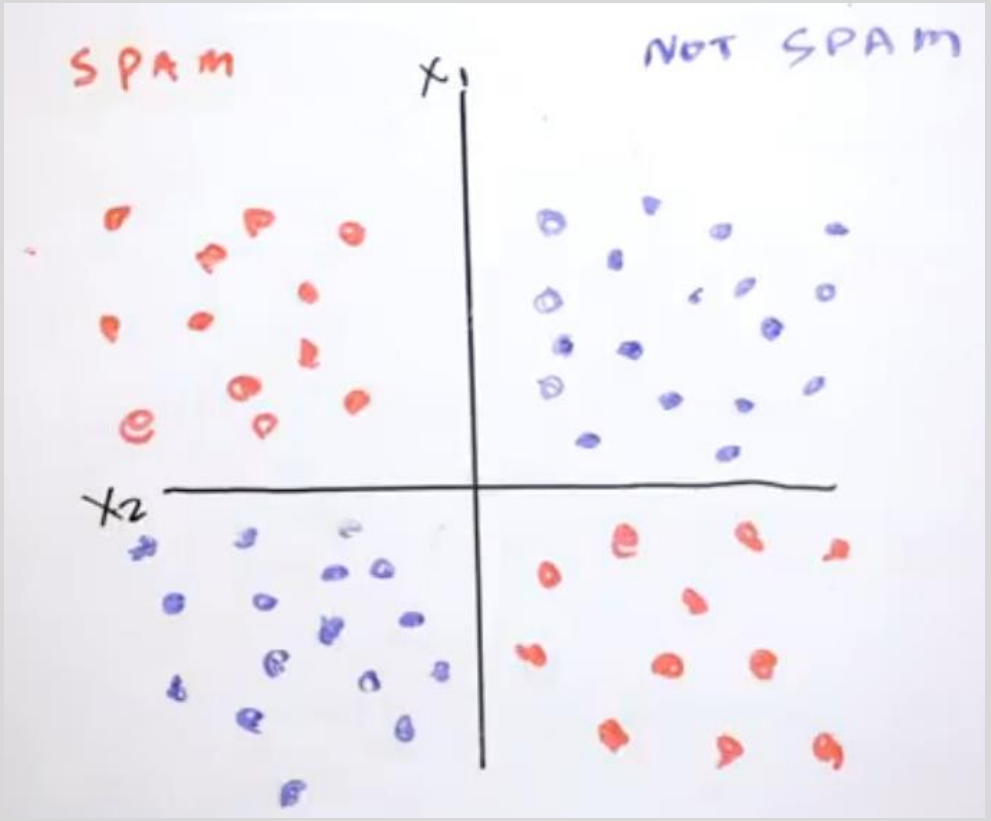


Multi-class Logistic Regression/ SoftMax Regression

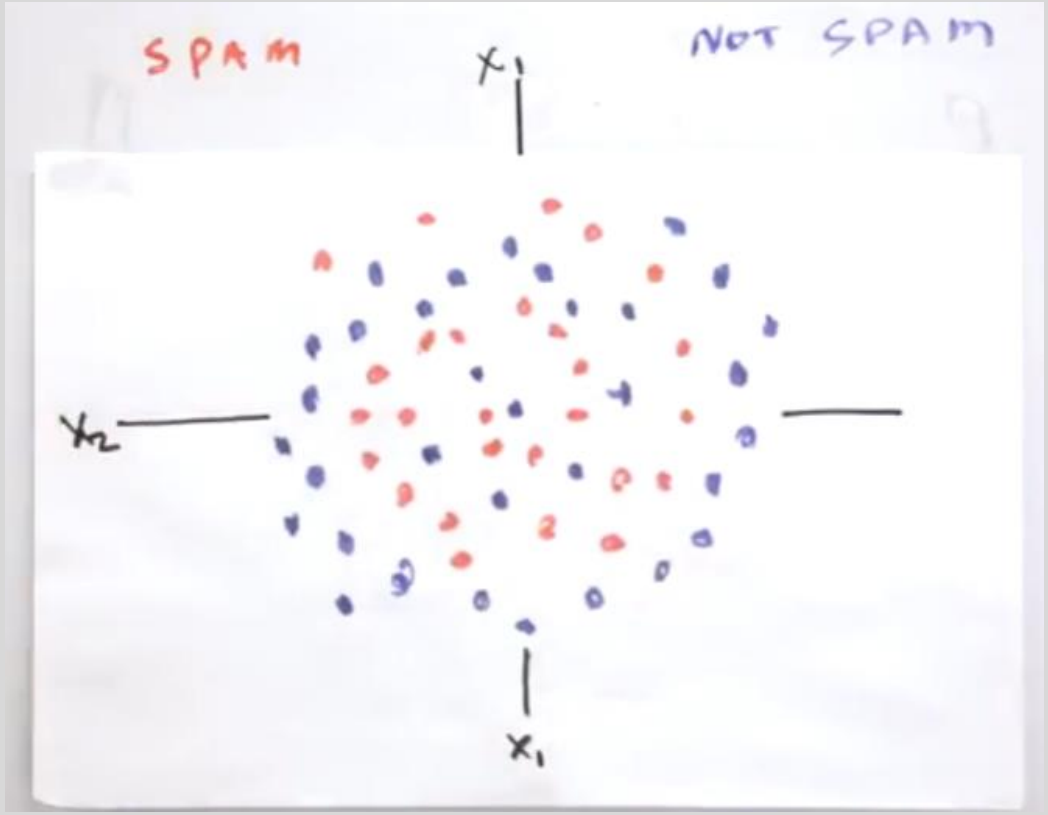
Source: https://www.kdnuggets.com/2016/07/softmax-regression-related-logistic-regression.html

LOGISTIC REGRESSION HANDS-ON

WHY DO WE NEED NEURAL NETWORKS?



Simple non-linear problem



Complex non-linear problem

Source: https://developers.google.com

SHALLOW NEURAL NETWORKS OVERVIEW

Perceptron is one of the simplest Artificial Neural Network (ANN) architectures, invented in 1957 by Frank Rosenblatt.

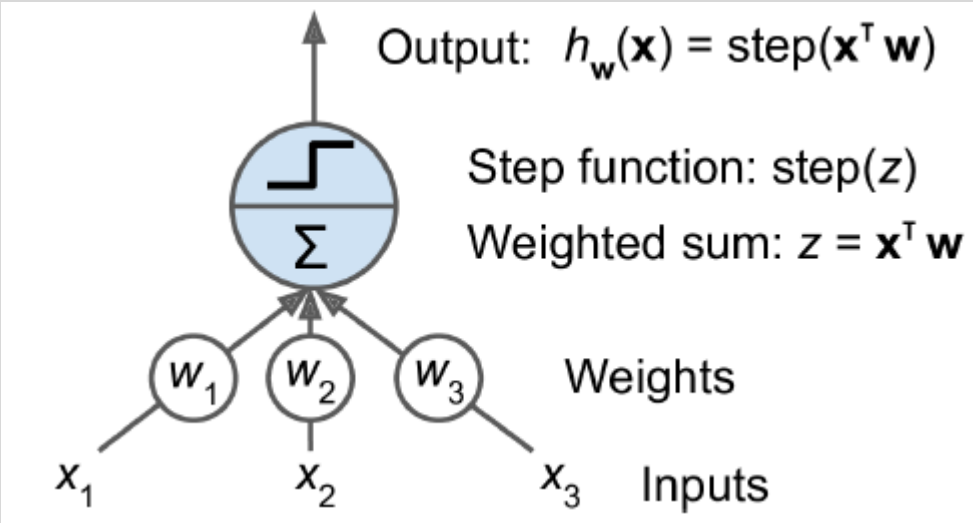


Figure a: Simple Perceptron/Threshold Logic Unit (TLU)/Linear Threshold Unit (LTU)

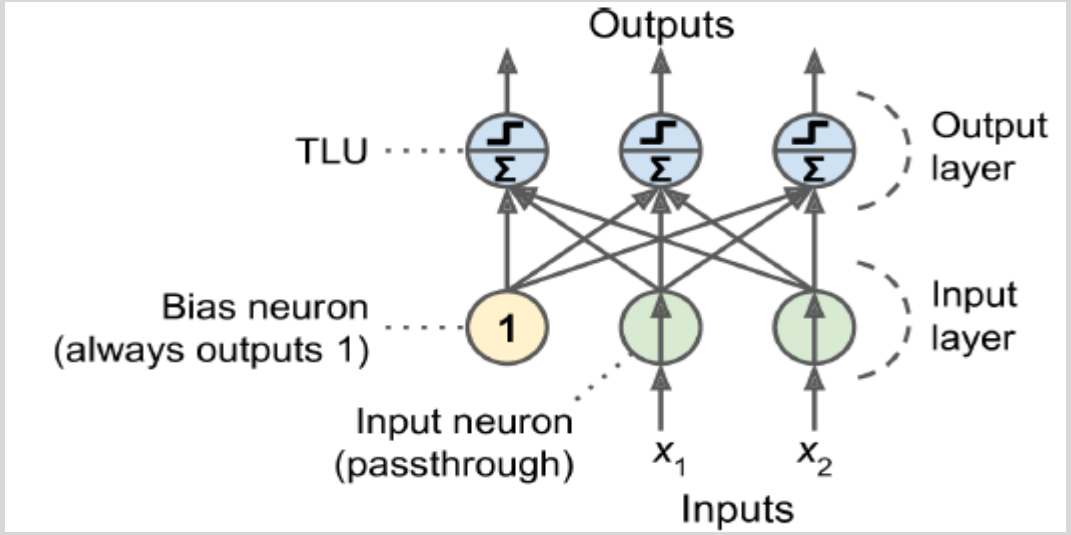


Figure b: Simple Perceptron with 2 input neurons, one bias neuron & 3 output neurons

Weighted sum of inputs: $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \boldsymbol{x}^T \boldsymbol{w}$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

W→ Weight matrix
b→ Bias vector
X→ feature matrix
Ø → activation function

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

PERCEPTRON LEARNING RULE (WEIGHT UPDATE)

- Inspired by the biological neuron
- **Hebbian Learning**
- Takes into account the error made by the network when it makes a prediction
- reinforces connections that help reduce the error
- Perceptron is fed one training instance at a time, and for each instance it makes its predictions
- every output neuron that produced a wrong prediction→ update the weights

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

In this equation:

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

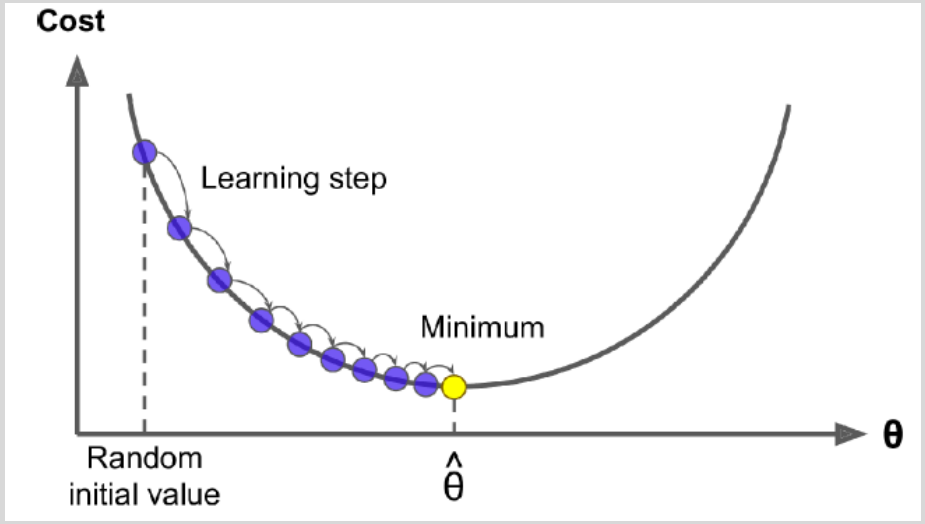
Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

GRADIENT DESCENT

- generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- Tweak parameters iteratively in order to minimize a cost function (loss or error of a MLP)
- measures the local gradient of the error function w.r.t the **parameter vector θ**, and it goes in the direction of descending gradient
- Once the gradient is zero, you have reached a minimum
- start by filling θ with random values→ **random initialization**
- improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function, until the algorithm converges to a minimum.

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Partial derivative of cost function



Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

EFFECT OF LEARNING RATE ON GRADIENT DESCENT

- important parameter in Gradient Descent is the size of the steps
- Step_size determined by learning rate (lr) of the algorithm
- **lr stable:** converge smoothly, avoid local minima
- **lr too small:** many iterations, long time to converge
- **lr too high:**
- less iterations, short time, overshoot
- jump across the valley and end up on the other side, possibly even higher up than you were before
- algorithm diverge, with larger and larger values→ fail to find a good solution

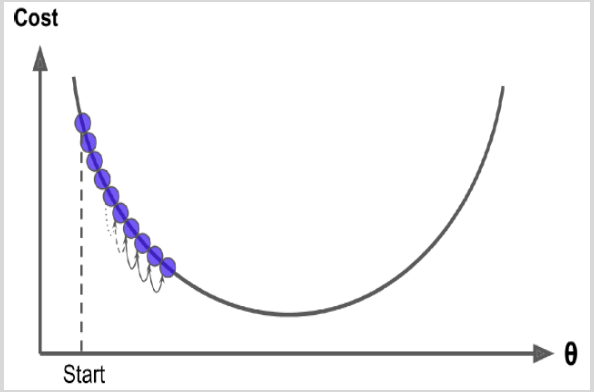


Figure a : Learning Rate too low

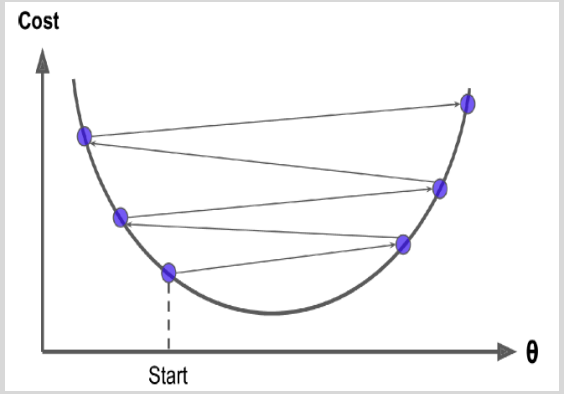


Figure b: Learning Rate too high

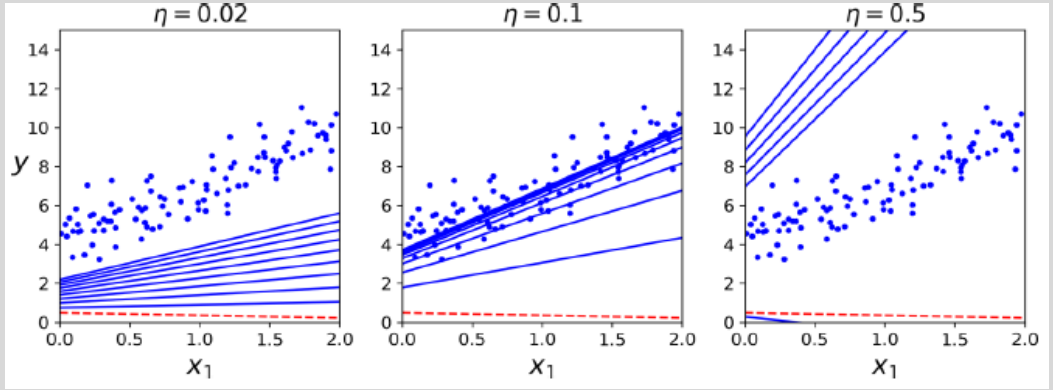


Figure c: Gradient Descent with various learning rates

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

TWO MAIN CHALLENGES OF GRADIENT DESCENT

- **Problems: [1] Local minimum & [2] Plateau**
- Random initialization of algorithm→ local minimum
- Small learning rate converges slowly→ **false local minimum**
- If it starts on the right, then it will take a very long time to cross the **plateau**
- if you stop too early, you will never reach the **global minimum**
- Gradient descent→ ensure all features have a similar scale→ else longer time to converge

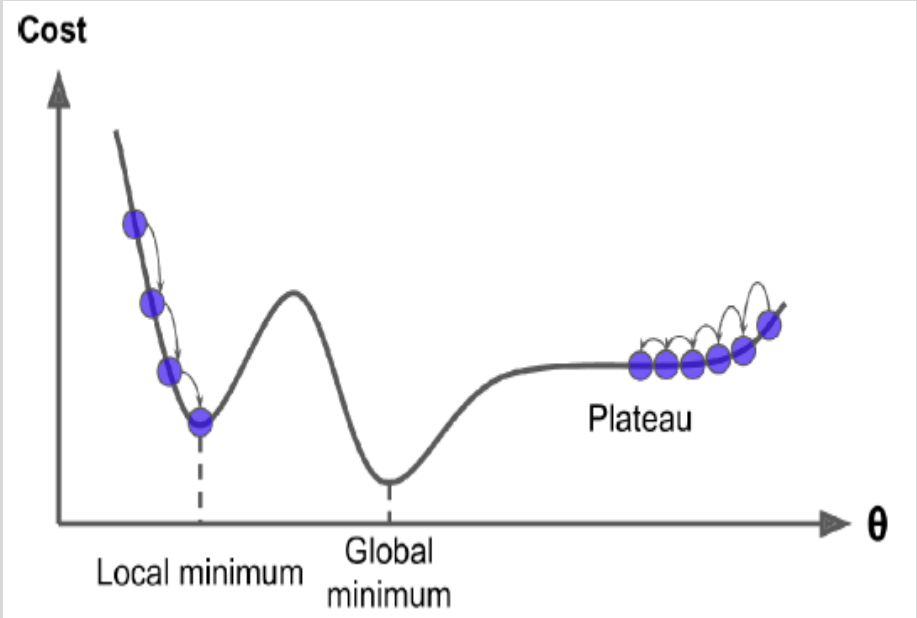


Figure a: Gradient Descent pitfalls

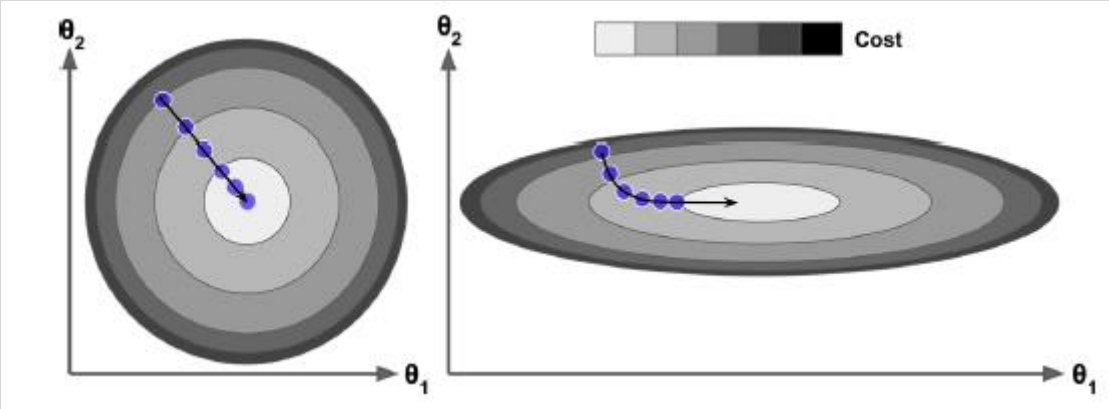
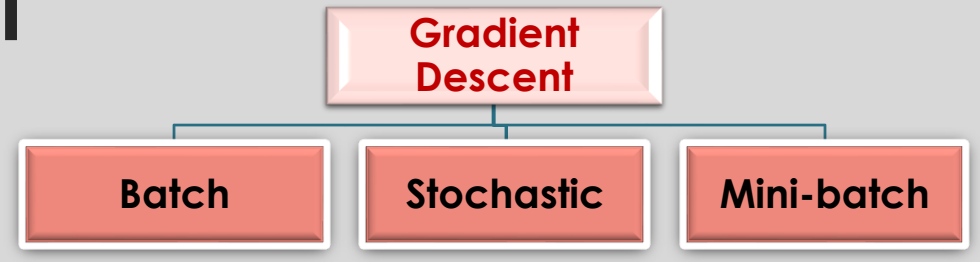


Figure b: Gradient Descent with (left) and without (right) feature scaling. Feature 1 has smaller values than feature 2.

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

FLAVORS OF GRADIENT DESCENT



Batch/ Full Gradient Descent:

- uses the whole batch of training data at every step→ slow for large datasets

Stochastic Gradient Descent:

- picks a random instance in the training set at every step and computes the gradients based only on that single instance→ faster for large datasets
- Random nature→ the cost function will bounce up and down, decreasing only on average
- once it gets to the minimum, there it will continue to bounce around, never settling down→ good but not optimal
- ensure to shuffle the instances during training such that they are not sorted by label

Mini-batch Gradient Descent:

- computes the gradients on small random sets of instances called mini-batches
- Performance boost, less erratic with larger mini-batches

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

MULTI-LAYER PERCEPTRON & BACKPROPAGATION

- In **1986**, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper→ backpropagation (Gradient Descent)
- Gradient descent→ efficient technique for computing the gradients automatically
- Forward and backward passes
- backpropagation algorithm→ compute the gradient of the network's error with regard to every single model parameter
- how each connection weight and each bias term should be tweaked in order to reduce the error
- performs a regular Gradient Descent step
- process is repeated until the network converges to the solution

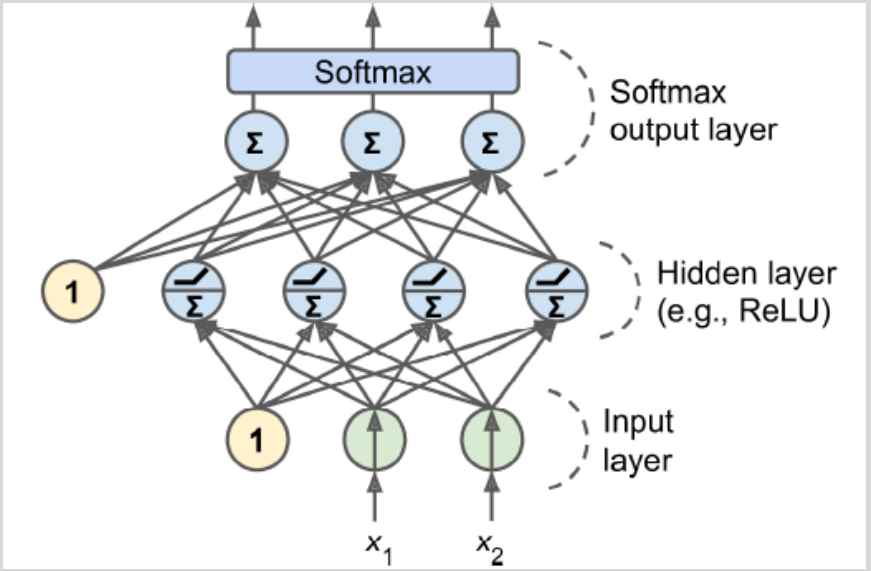
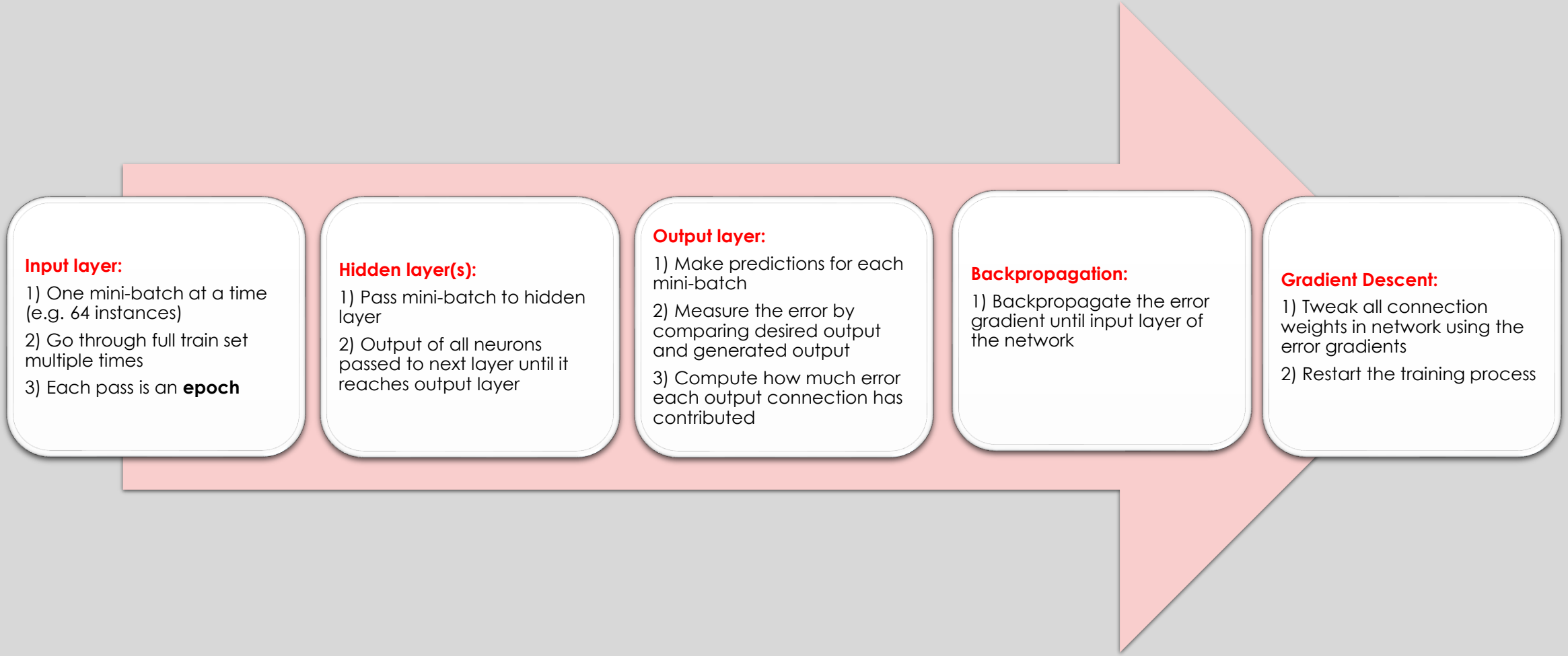


Figure : Multi-Layer Perceptron (MLP) with backpropagation & bias neurons

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

BACKPROPAGATION TRAINING ALGORITHM FOR MLP

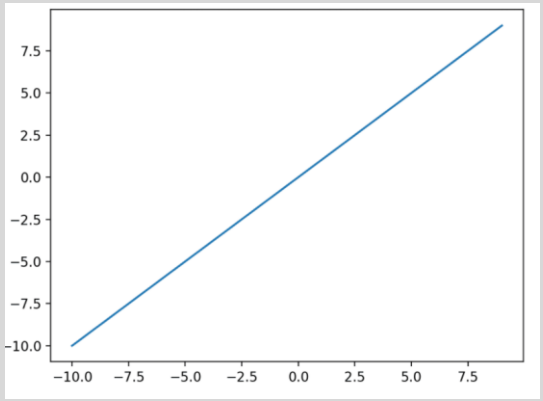


Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

ACTIVATION/TRANSFER/SQUASHING FUNCTION

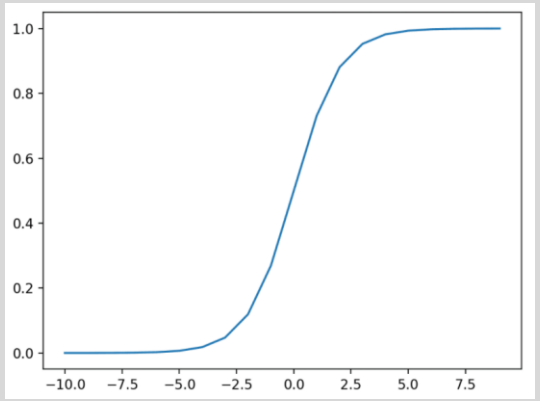
- defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network

$$f(x) = w^T x + b$$



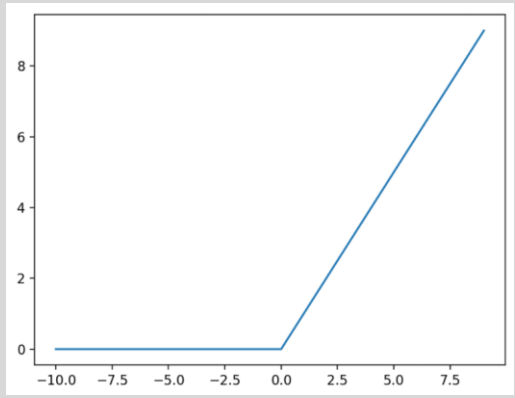
Linear

$$f(x) = \left(\frac{1}{(1 + exp^{-x})} \right)$$



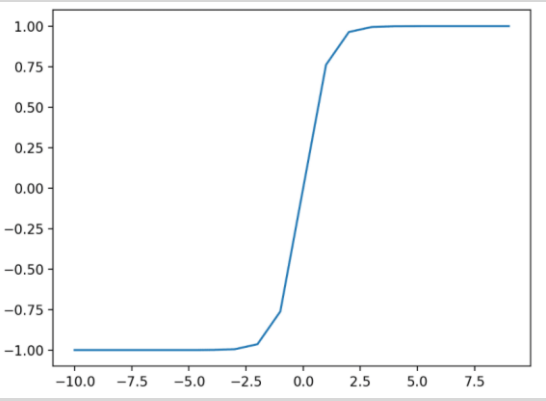
Sigmoid

$$f(x) = max(0, x) = \begin{cases} x_i, & if\ x_i \geq 0 \\ 0, & if\ x_i < 0 \end{cases}$$



Rectified Linear Unit(ReLU)

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



Hyperbolic Tangent (Tanh)

Source: https://machinelearningmastery.com/

COMMON ACTIVATION FUNCTIONS

Linear

- "identity" (multiplied by 1.0) or "no activation"
- does not change the weighted sum of the input in any way and instead returns the value directly

$$f(x) = w^T x + b$$

Sigmoid/ Logistic

- logistic function→ S-Shaped
- takes any real value as input and outputs values in the range 0 to 1
- Larger input→ positive→ close to 1 and vice versa

$$f(x) = \left(\frac{1}{(1 + exp^{-x})} \right)$$

Softmax

- outputs a vector of values that sum to 1.0 that can be interpreted as probabilities of class membership
- "softer" version of argmax
- argmax function that outputs a 0 for all options and 1 for the chosen option.
- allows a probability-like output of a winner-take-all function

$$f(x_i) = \frac{exp\ (x_i)}{\sum_j exp\ (x_j)}$$

Rectified Linear Unit (ReLU)

- Common for Hidden layers
- Max(0,x)→input value (x) is negative, then a value 0 is returned, otherwise, the value is returned

$$f(x) = max\ (0, x) = \begin{cases} x_i, & if\ x_i \geq 0 \\ 0, & if\ x_i < 0 \end{cases}$$

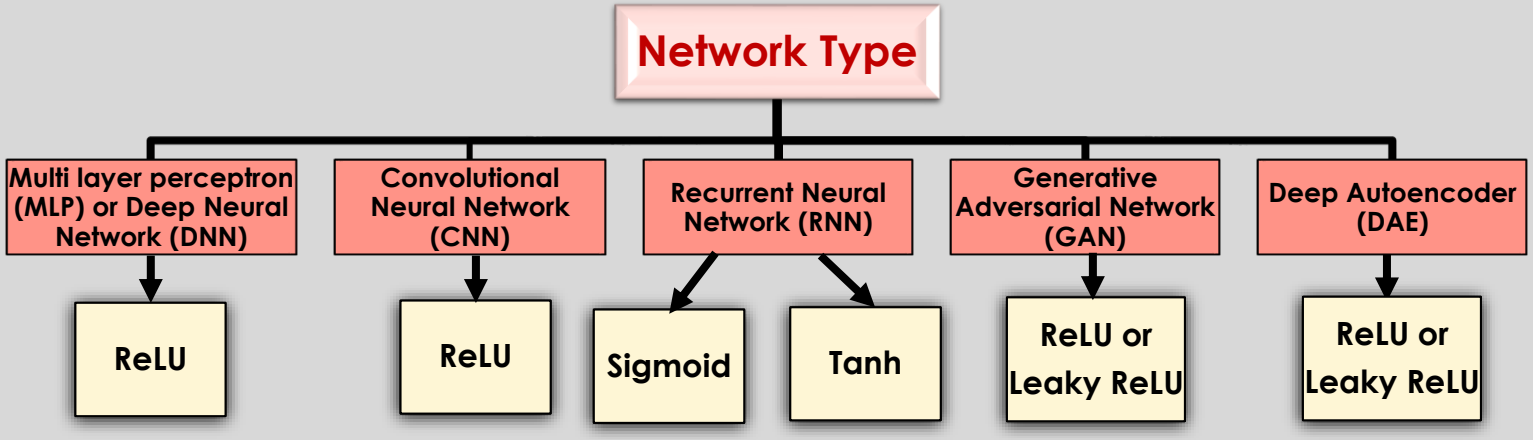
Hyperbolic Tangent (Tanh)

- S-shape, range between -1 to +1
- Larger input→ positive→ close to 1 and vice versa

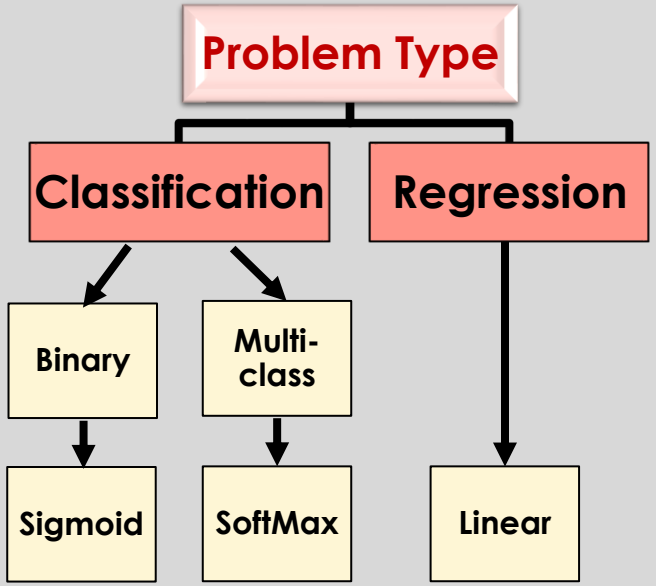
$$f(x) = \left(\frac{e^x - e^{-x}}{e^x + e^{-x}} \right)$$

Source: https://machinelearningmastery.com/

HOW TO CHOOSE AN ACTIVATION FUNCTION



Choosing for a Hidden Layer

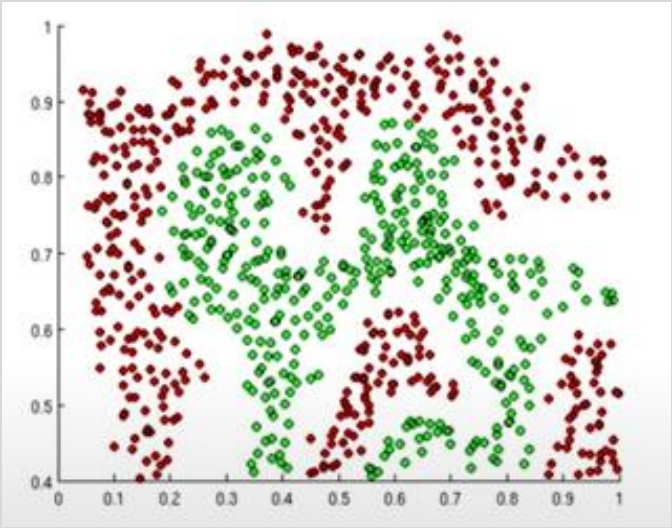


Choosing for an Output Layer

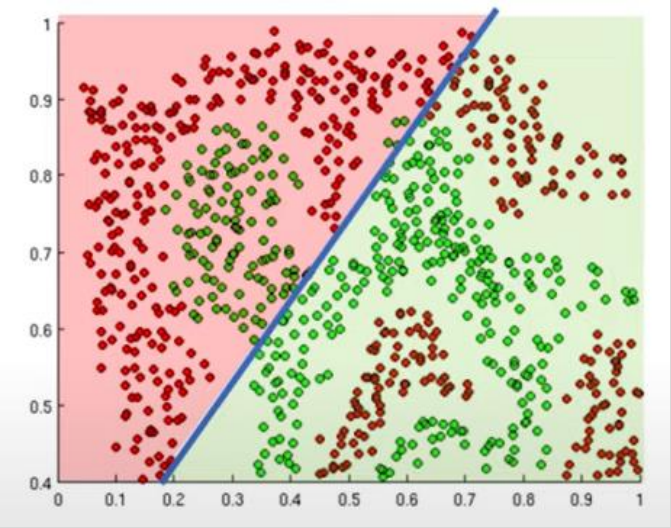
Source: https://machinelearningmastery.com/

IMPORTANCE OF ACTIVATION FUNCTIONS

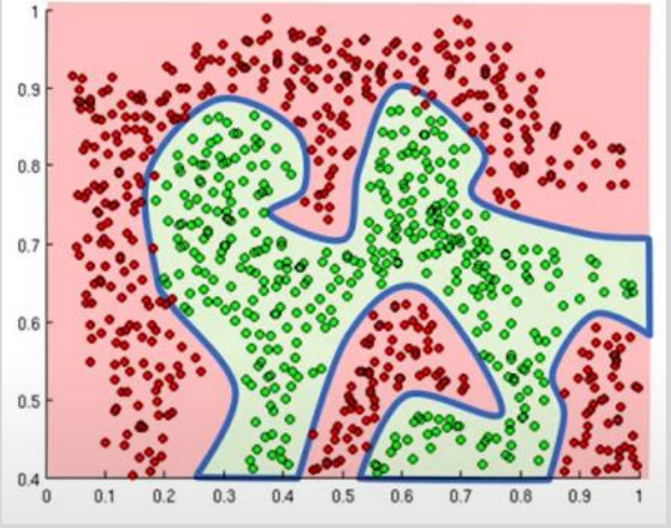
- Non-linear function
- perform complex computations in the hidden layers and then transfer the result to the output layer
- Introduce non-linearities into the network



Identify spam (red) and non-spam (green)



Linear activation functions produce linear decisions no matter the size of the network



Non-linearities allow us to approximate complex functions

Source: introtodeeplearning.com

MLP HANDS-ON

THANKS!

Do you have any questions?