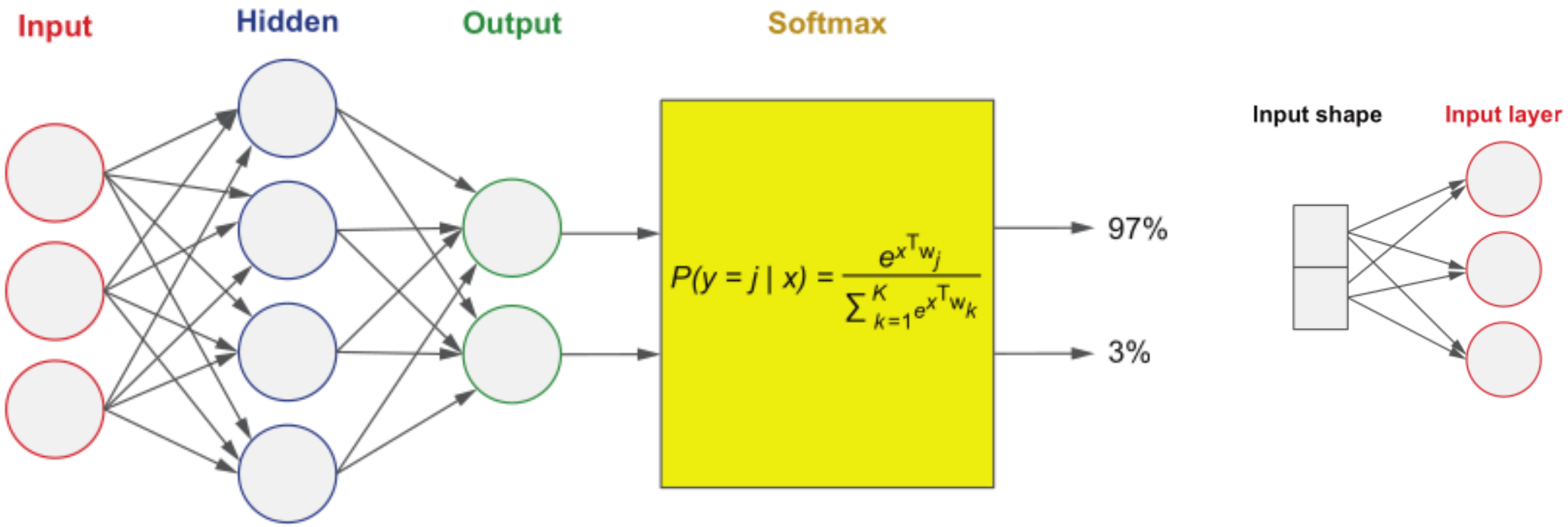
CSE 465
Lecture 2-4

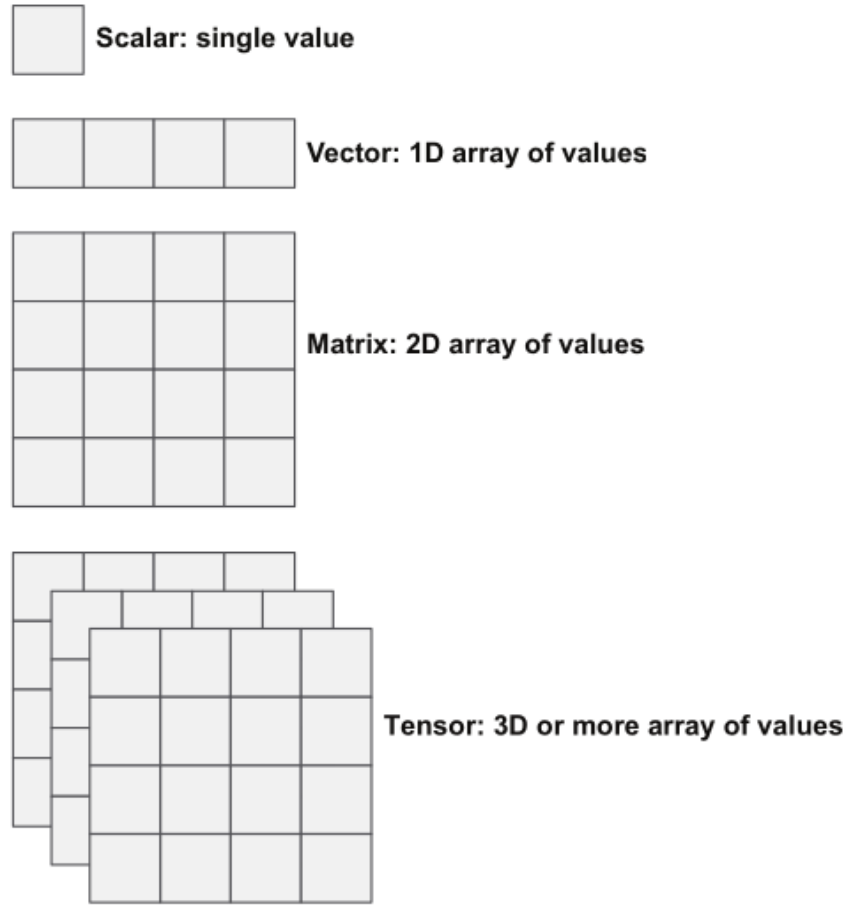
Deep Feed Forward Neural Networks

Artificial Neural Network (ANN)



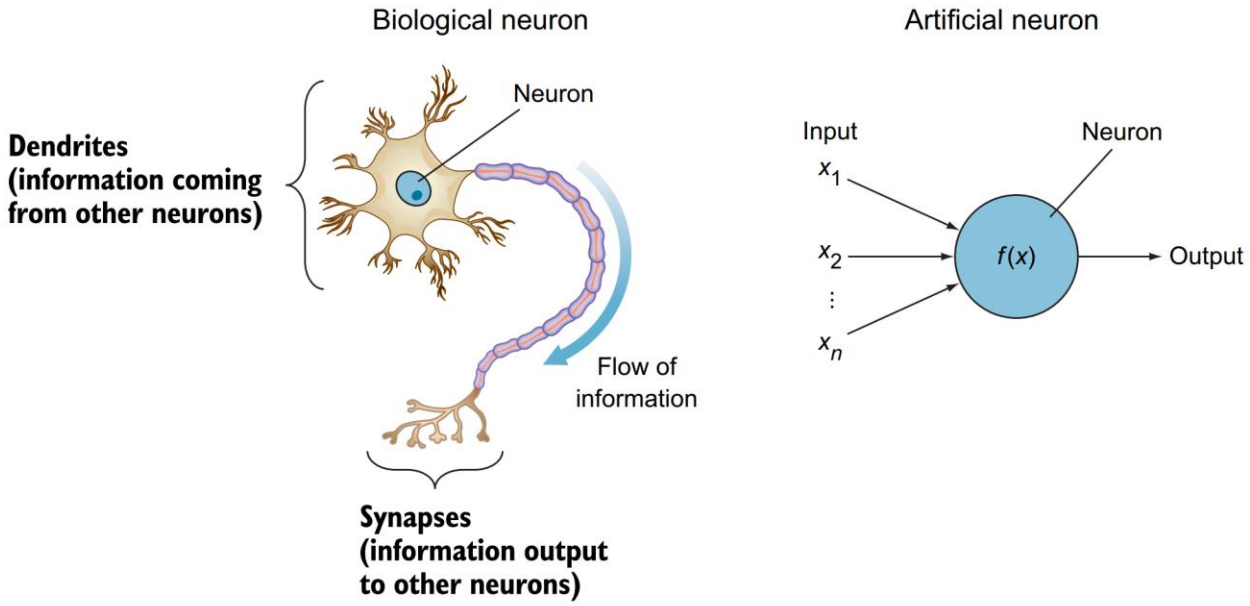
Input to the Network

- The input layer to a neural network takes numbers. All the input data is converted to numbers
  - Everything is a number: The text becomes numbers, speech becomes numbers, pictures become numbers, and things that are already numbers are just numbers.
- Neural networks take numbers as vectors, matrices, or tensors
  - These are simply names for the number of dimensions in an array.
  - A vector is a one-dimensional array, such as a list of numbers.
  - A matrix is a two-dimensional array, like the pixels in a black and-white image.
  - And a tensor is any array of three or more dimensions—for example, a stack of matrices in which each matrix is the same dimension



Perceptron

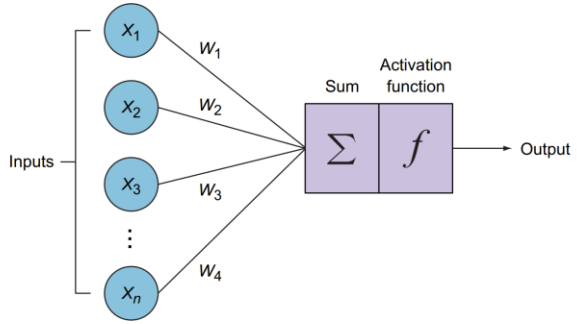
- The simplest neural network is the perceptron, which consists of a single neuron
- Conceptually, the perceptron functions like a biological neuron



Perceptron

- The artificial neuron performs two consecutive functions
  - It calculates the weighted sum of the inputs to represent the total strength of the input signals
  - Then it applies a step function to the result
    - High value if the linear combination exceeds a threshold
    - Low value otherwise
    - Or use something fancier based on the problem at hand
- Not all input features are equally useful or important.
  - To represent that, each input node is assigned a weight value, called its connection weight, to reflect its importance

Perceptron



Input vector: The feature vector that is fed to the neuron

It is usually denoted with an uppercase X to represent a vector of inputs (x1, x2, . . ., xn)

Weights matrix: Each x_i is assigned a weight value w_i

Functions: The calculations performed within the neuron to modulate the input signals: the weighted sum and step activation function

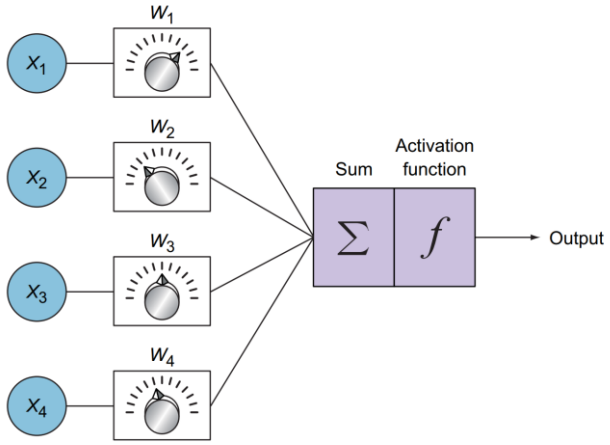
Output: Controlled by the type of activation function

Weights matrix: Feature weights

- Not all input features are equally important (or useful) features
- Each input feature (x1) is assigned its own weight (w1) that reflects its importance in the decision-making process
- Inputs assigned greater weight have a greater effect on the output
- If the weight is high, it amplifies the input signal; and if the weight is low, it diminishes the input signal
- In common representations of neural networks, the weights are represented by lines or edges from the input node to the perceptron

How does a perceptron learn?

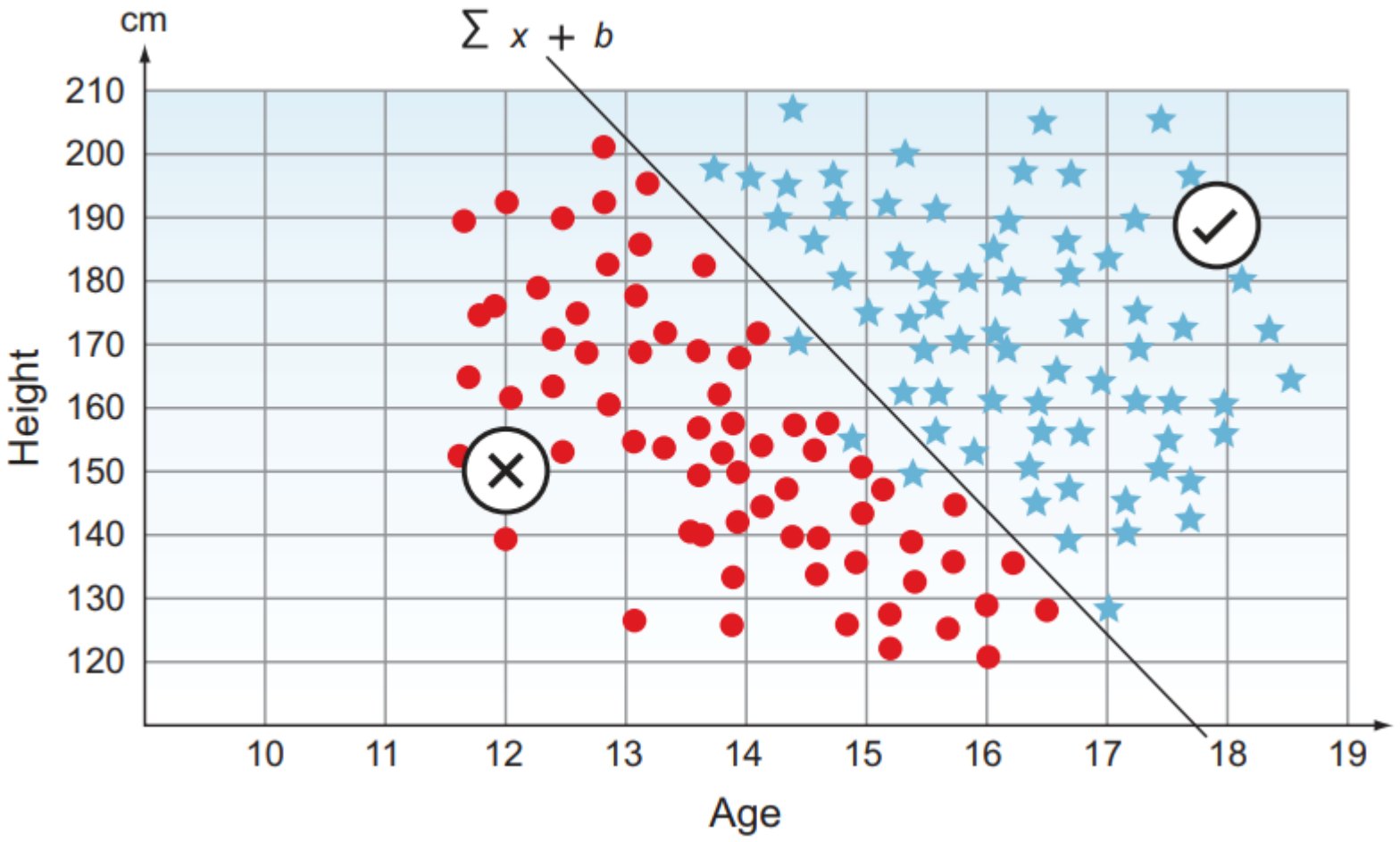
- The perceptron uses trial and error to learn from its mistakes
  - It uses the weights as knobs by tuning their values up and down until the network is trained
- This process is repeated many times, and the neuron continues to update the weights to improve its predictions



Is one neuron enough?

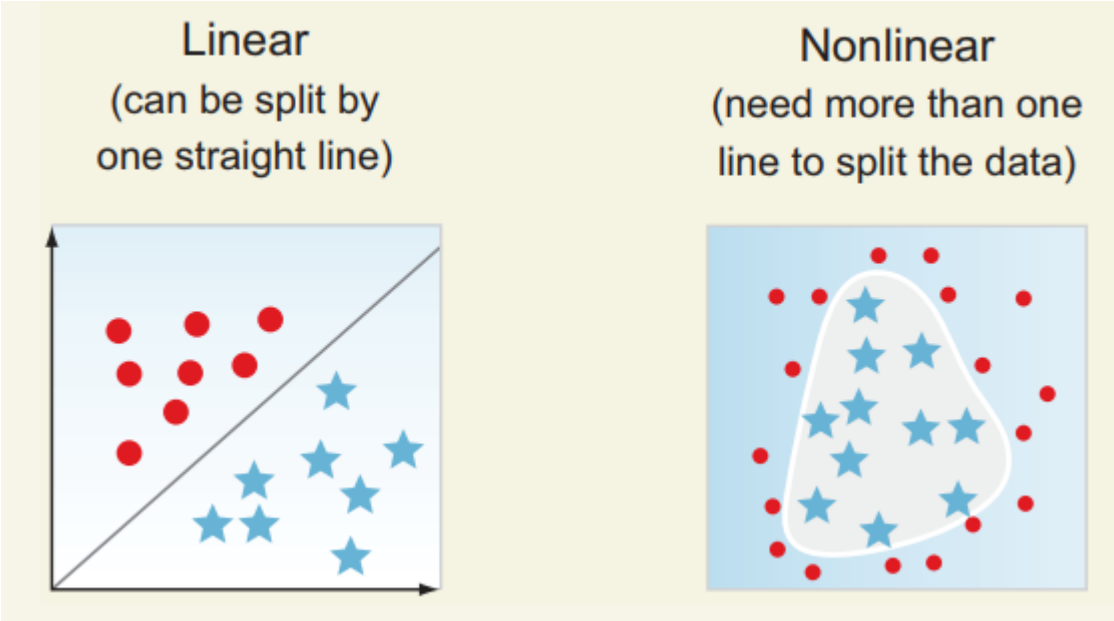
- Suppose we want to train a perceptron to predict whether a player will be accepted into the national Cricket squad
- We collect all the data from previous years and train the perceptron to predict whether players will be accepted based on only two features (height and weight)
  - $z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$
- After the training is complete, we can start using the perceptron to predict new players
- When we get a player having a height of 150 cm and 12 years old, we compute the previous linear expression with these values [(150, 12)]
- In this simple case, the single perceptron may work fine because data was *linearly separable*

Model for accepting/rejecting players



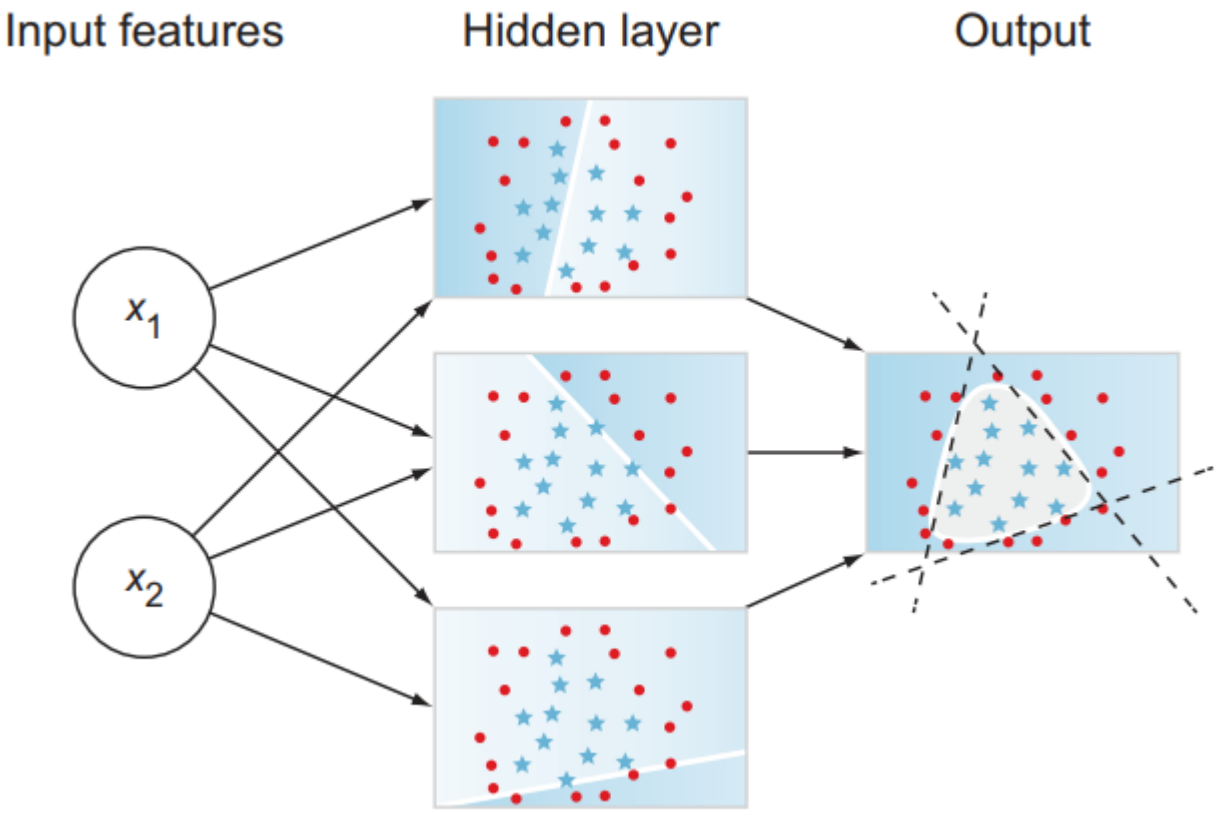
Linear/Nonlinear data

- *Linear datasets*—The data can be split by a single straight line
- *Nonlinear datasets*—The data cannot be split by a single straight line
  - We need more than one line to form a shape that splits the data
- In the linear problem, the stars and dots can be easily classified by drawing a single straight line
- In nonlinear data, a single line will not separate both shapes

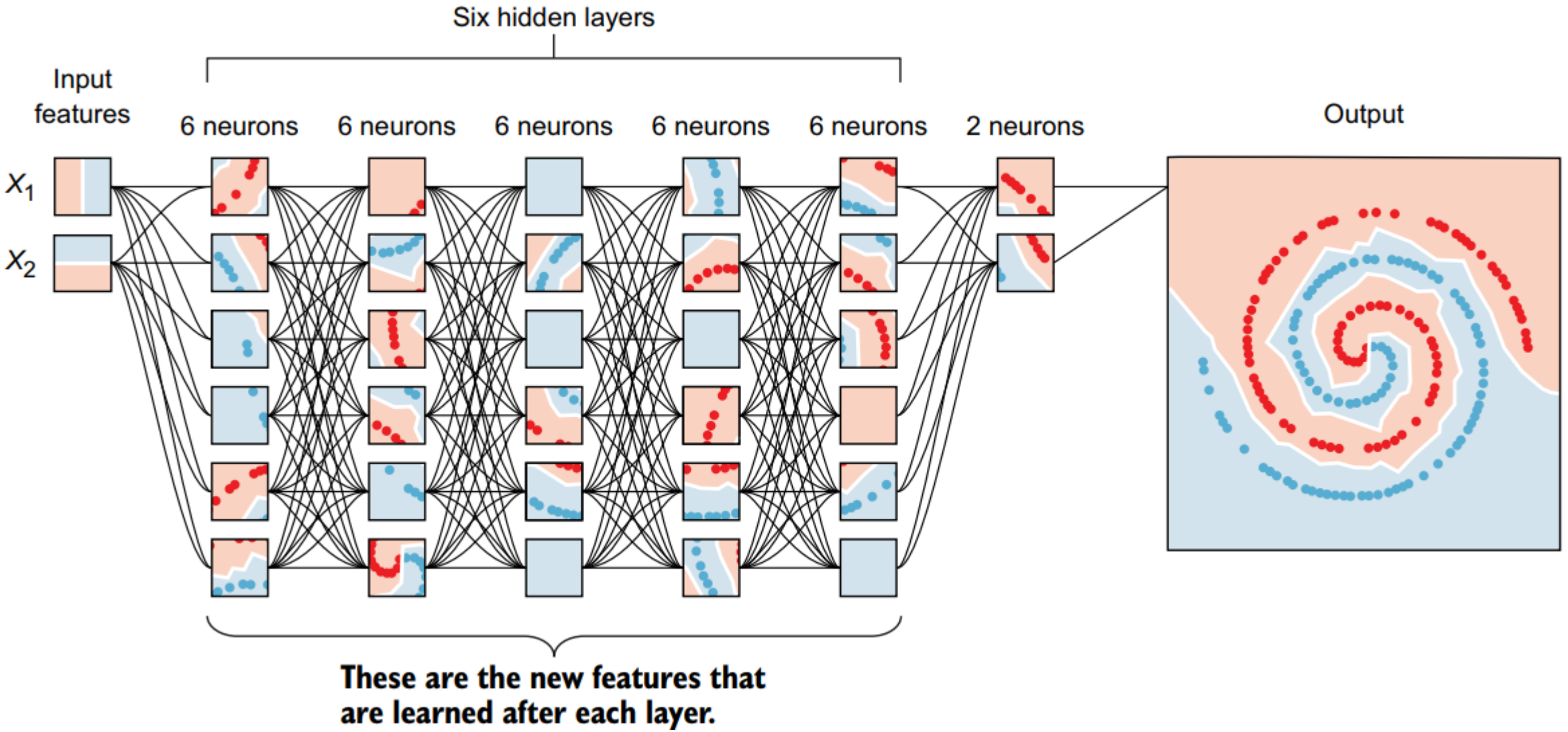


Multiple perceptron

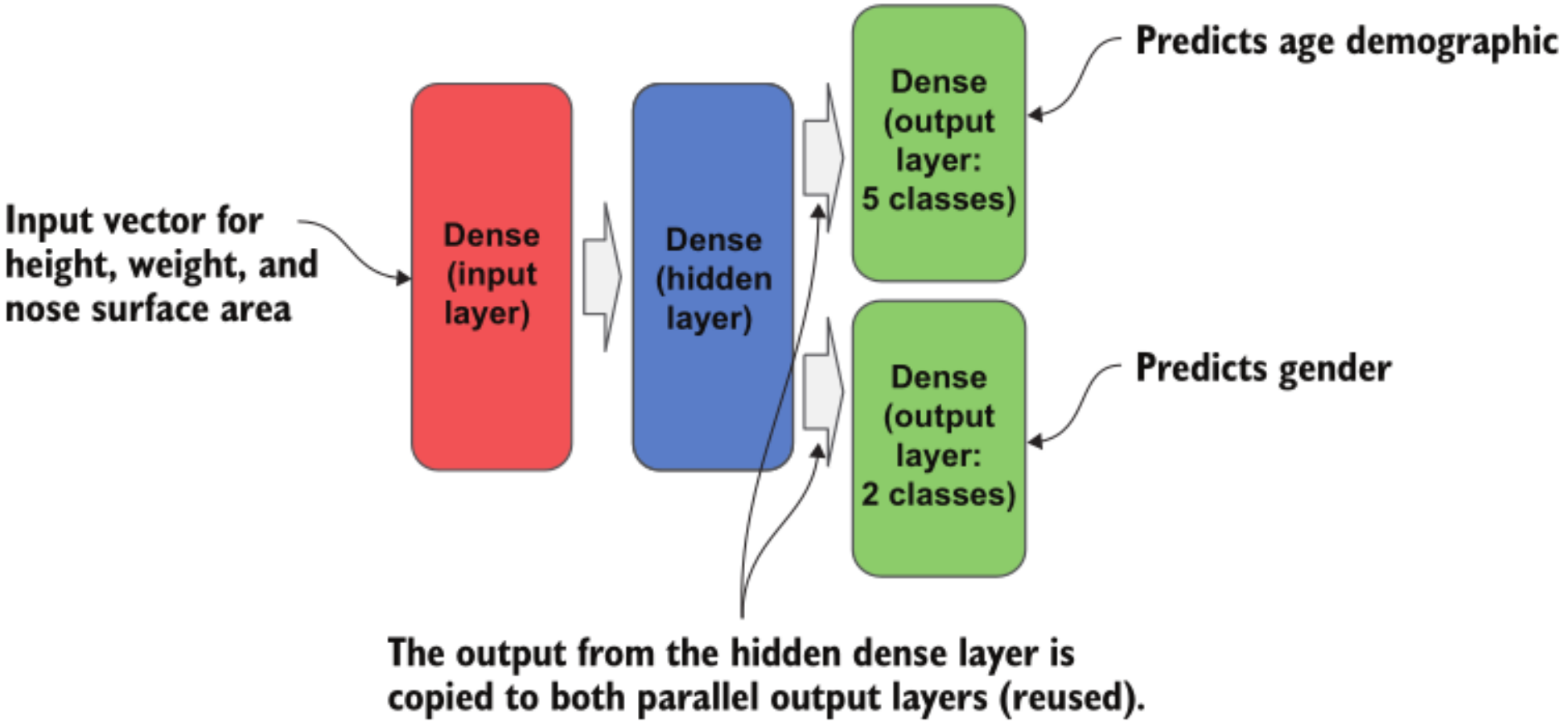
- To split a nonlinear dataset, we need more than one line
- Example of a small neural network that is used to model nonlinear data
- In this network, we use three neurons stacked together in one layer called a *hidden layer*
  - Hidden as we don't see the output of these layers during the training process



Multi-layer perceptron



Multilabel Multiclass Neural Network



Main components of a Neural Network

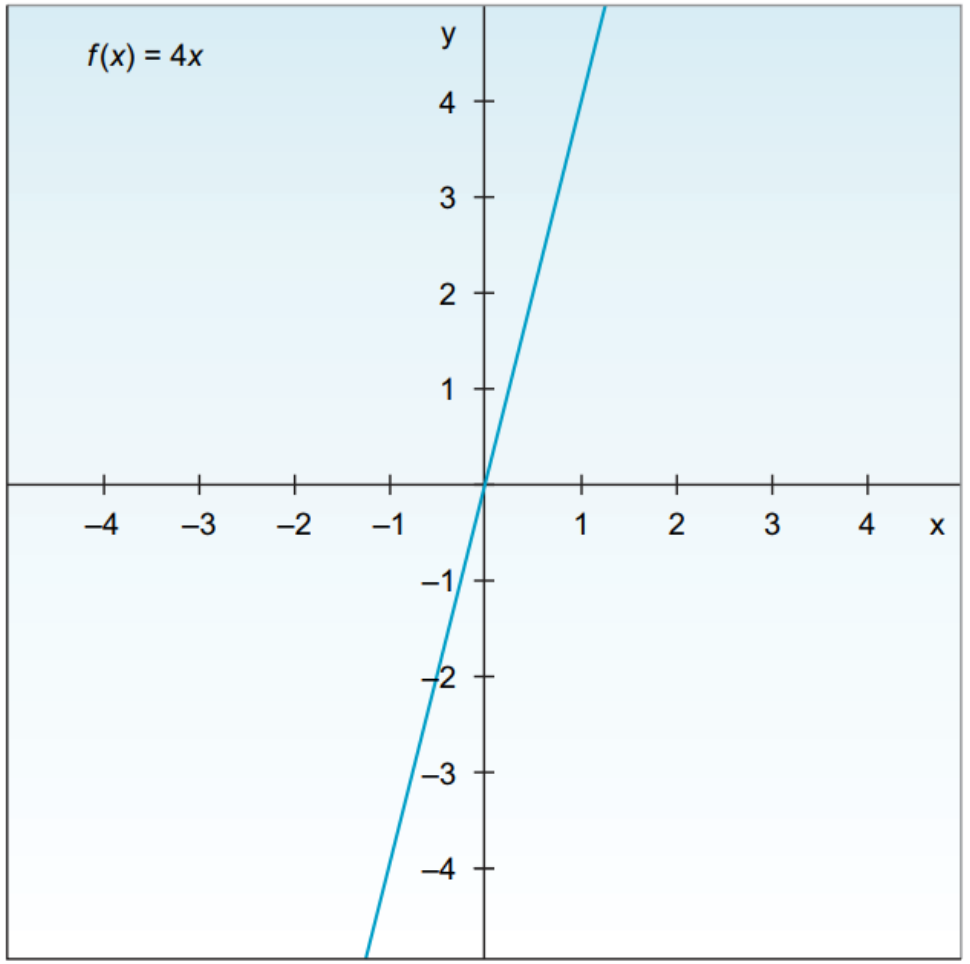
- ***Input layer***— Contains the feature vector
  - It is a layer not the inputs (which come from the dataset)
- ***Hidden layers***—The neurons are stacked on top of each other in hidden layers
  - They are called “hidden” layers because we don’t see or control the input going into these layers or the output
  - All we do is feed the feature vector to the input layer and see the output coming out of the output layer
- ***Weight connections (edges)***—Weights are assigned to each connection between the nodes to reflect the importance of their influence on the final output prediction
  - In graph network terms, these are called *edges* connecting the *nodes*
- ***Output layer*** — We get the answer or prediction from our model from the output layer
  - Depending on the setup of the neural network, the final output
    - Could be a real-valued output (regression problem) or
    - A set of probabilities (classification problem)

Activation functions

- Activation functions are sometimes referred to as *transfer functions* or *nonlinearities* because they transform the linear combination of a weighted sum into a nonlinear model
- The purpose of the activation function is to introduce nonlinearity into the network
- Without it, a multilayer perceptron will perform similarly to a single perceptron no matter how many layers we add
- Activation functions are needed to restrict the output value to a certain finite value

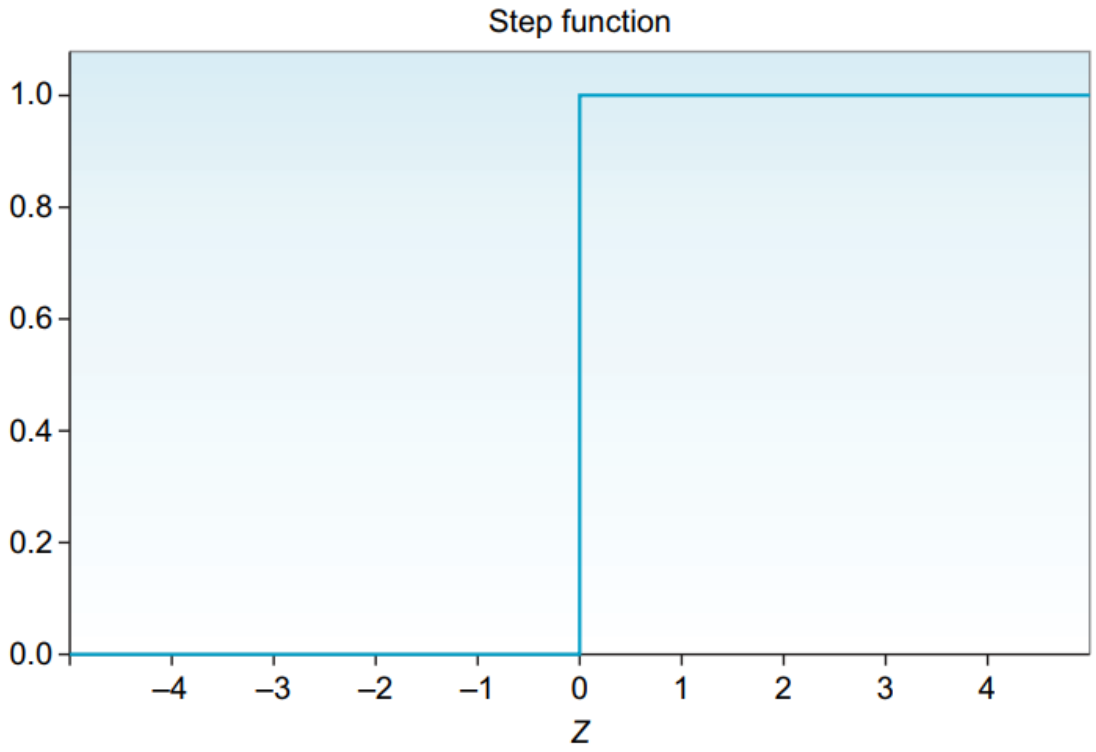
Linear Activation Function

- A *linear transfer function*, also called an *identity function*, indicates that the function passes a signal through unchanged
- In practical terms, the output will be equal to the input, which means we don't have an activation function
- So, no matter how many layers our neural network has, all it is doing is computing a linear activation function or, at most, scaling the weighted average coming in



Step Activation Function

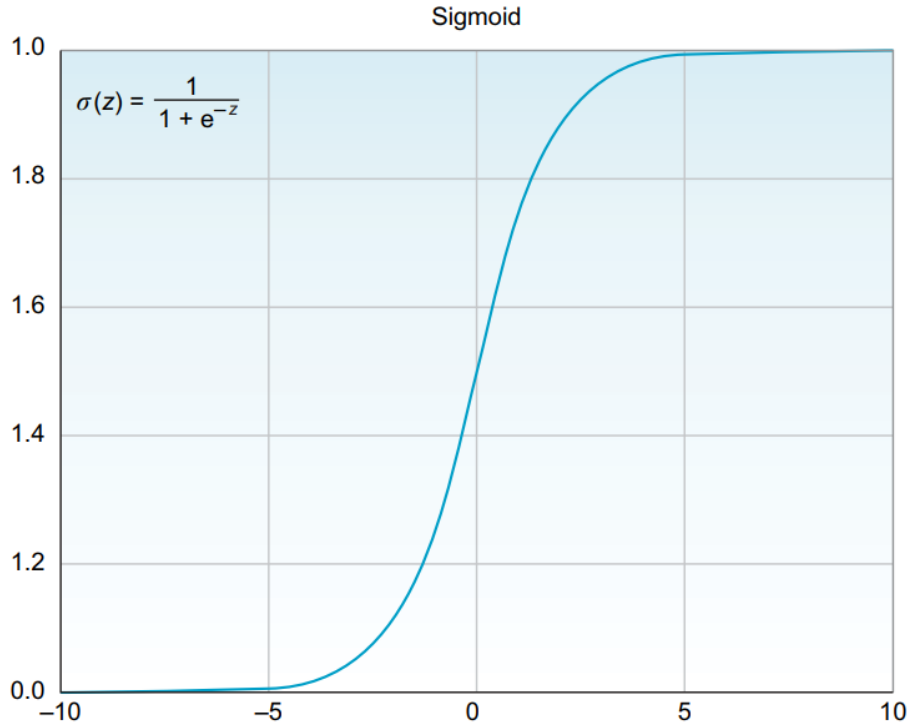
- The *step function* produces a binary output
- It basically says that
  - If the input $x > 0$, it fires (output $y = 1$)
  - Else (input < 0), it doesn't fire (output $y = 0$)
- It is mainly used in binary classification problems like true or false, spam or not spam, and pass or fail



$$\text{Output} = \begin{cases} 0 \text{ If } w \cdot x + b \leq 0 \\ 1 \text{ If } w \cdot x + b > 0 \end{cases}$$

Sigmoid/Logistic Activation Function

- This is one of the most common activation functions
- It is often used in binary classifiers to predict the *probability* of a class when we have two classes
- Sigmoid or logistic functions convert infinite continuous variables (range between $-\infty$ to $+\infty$) into simple probabilities between 0 and 1
- It is also called the *S-shape curve* because when plotted in a graph, it produces an S-shaped curve.



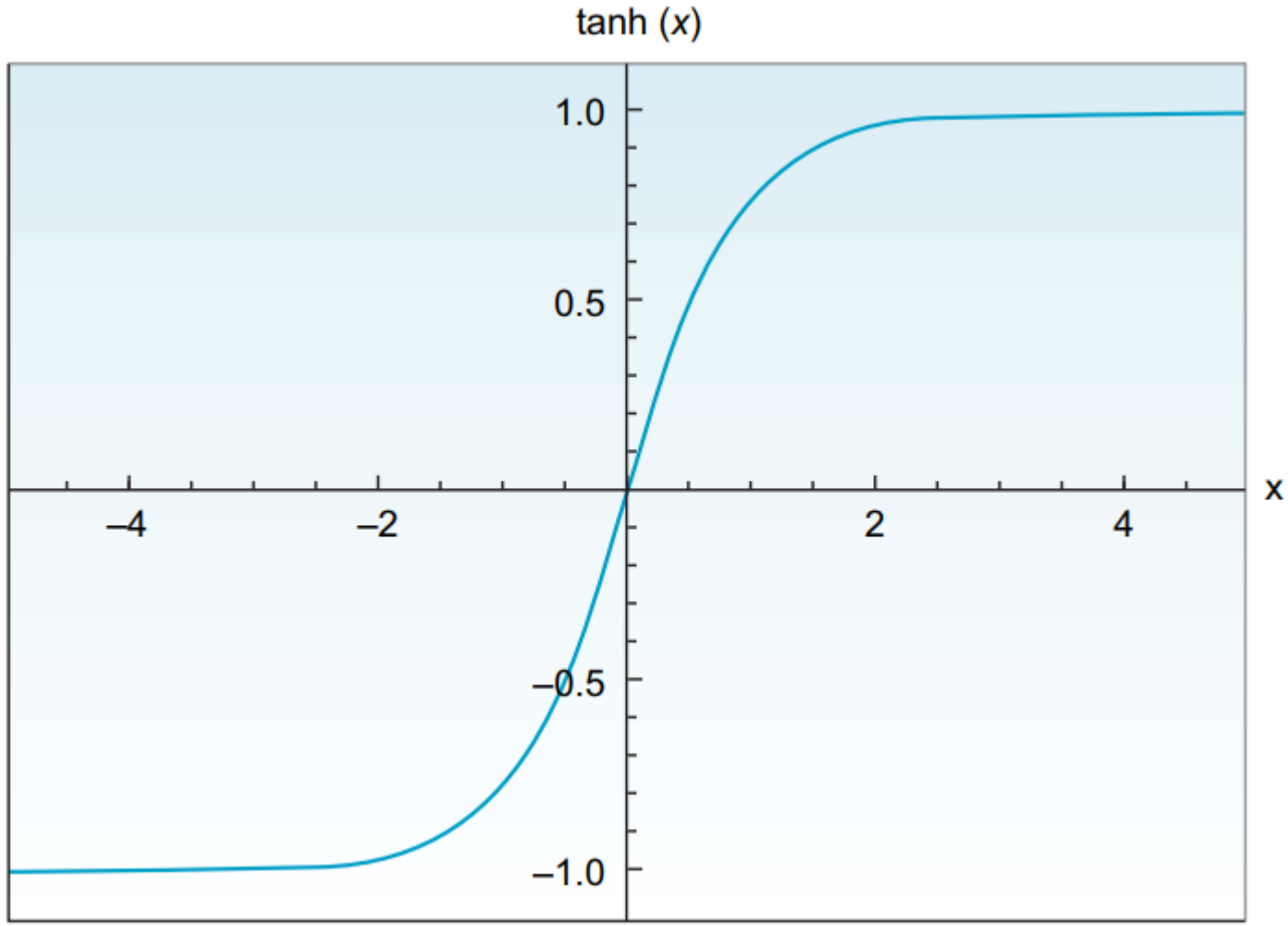
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Hyperbolic Tangent (tanh) Activation Function

- The hyperbolic tangent function is a shifted version of the sigmoid version
- Instead of squeezing the signal values between 0 and 1, tanh squishes all values to the range –1 to 1
- Tanh almost always works better than the sigmoid function in hidden layers because it has the effect of centering data so that the mean of the data is close to zero rather than 0.5, which makes learning for the next layer a little bit easier
- One of the downsides of both sigmoid and tanh functions is that if (z) is very large or very small
  - Then the gradient (or derivative or slope) of this function becomes very small (close to zero), which will slow down gradient descent

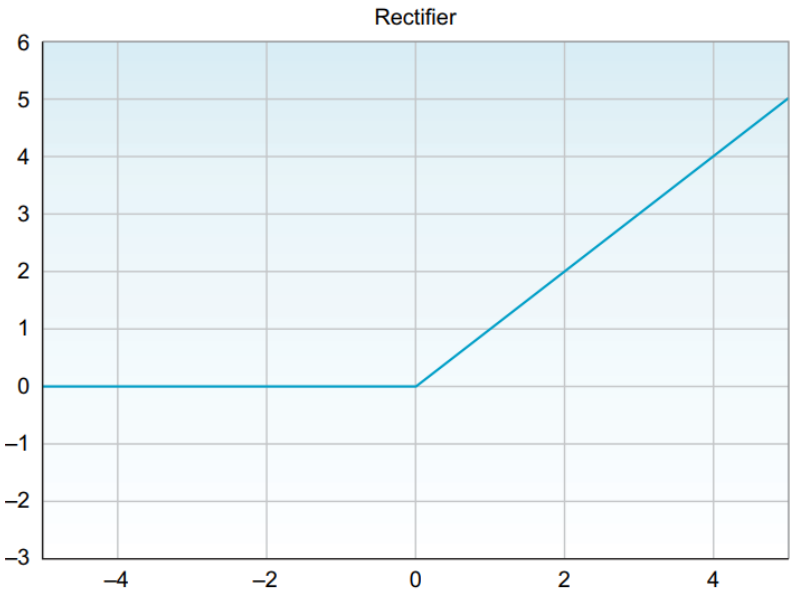
Hyperbolic Tangent (tanh) Activation Function

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



Rectified Linear (ReLU) Activation Function

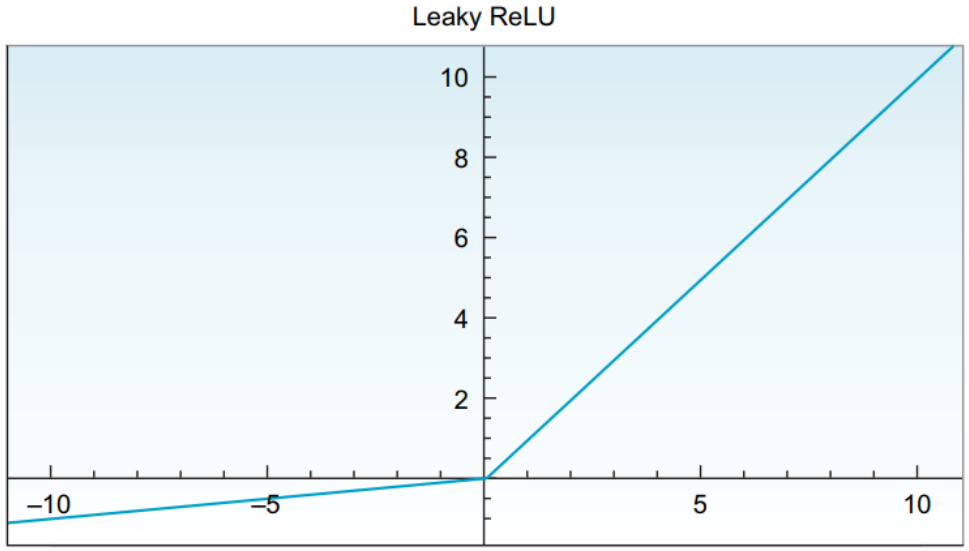
- The rectified linear unit (ReLU) activation function activates a node only if the input is above zero
  - If the input is below zero, the output is always zero
  - But when the input is higher than zero, it has a linear relationship with the output variable
- ReLU is considered the state-of-the-art activation function because it works well in many different situations, and it tends to train better than sigmoid and tanh in hidden layers



$$\text{ReLU}(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$

Leaky ReLU Activation Function

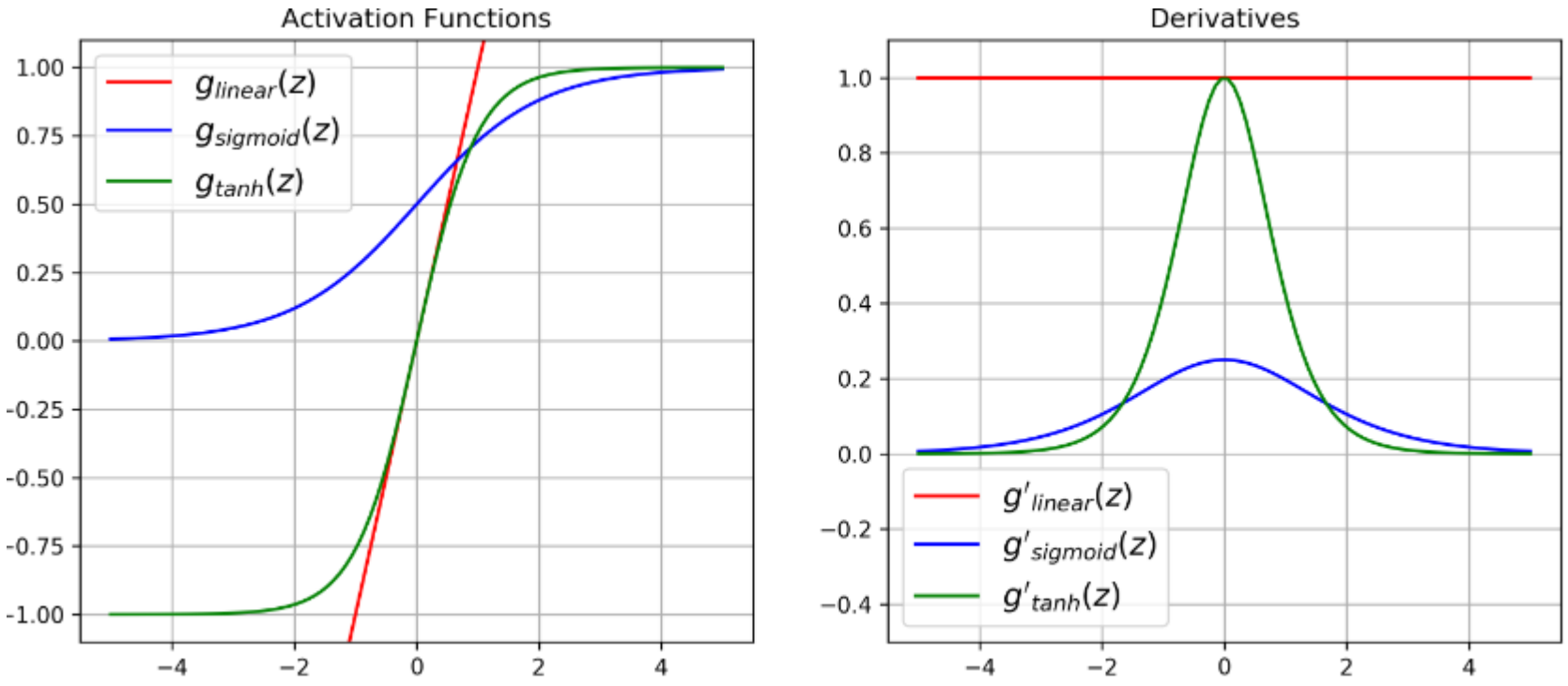
- One disadvantage of ReLU activation is that the derivative is equal to zero when (x) is negative
- Leaky ReLU is a ReLU variation that tries to mitigate this issue
- Instead of having the function be zero when $x < 0$, leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative
- It usually works better than the ReLU function



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

Derivative of Activation Functions

- The learning of the network depends on the activation function
- Some activation functions are better during optimization than the others

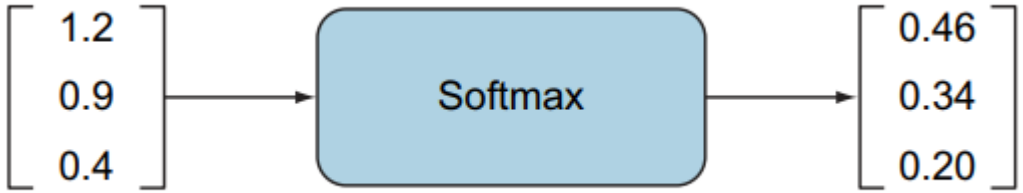


Softmax Function

- The softmax function is a generalization of the sigmoid function
- It is used to obtain classification probabilities when we have more than two classes
- It forces the outputs of a neural network to sum to 1 (for example, 0 < output < 1)
- A very common use case in deep learning problems is to predict a single class out of many options (more than two)
- Softmax is the go-to function that we often use at the output layer of a classifier when we are working on a problem where we need to predict a class between more than two classes
- Softmax works fine classifying two classes, as well – it will basically work like a sigmoid function

Softmax Activation Function (2)

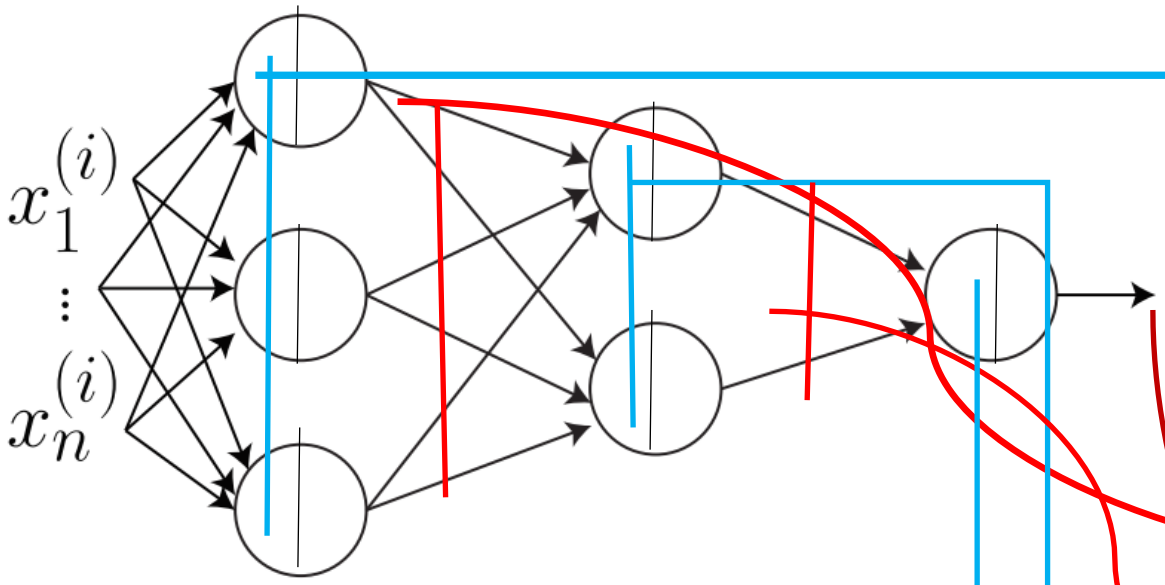
$$\sigma(x_j) = \frac{e^{x_j}}{\Sigma_i e^{x_i}}$$



The feedforward process

- The term *feedforward* is used to imply the forward direction in which the information flows from the input layer through the hidden layers, all the way to the output layer
- This process happens through the implementation of two consecutive functions
  - The weighted sum and
  - The activation function
- In short, the forward pass is the calculations through the layers to make a prediction

A fully connected Neural Network



Forward Calculation

$$z^{[1]} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1]} = g(z^{[1]})$$

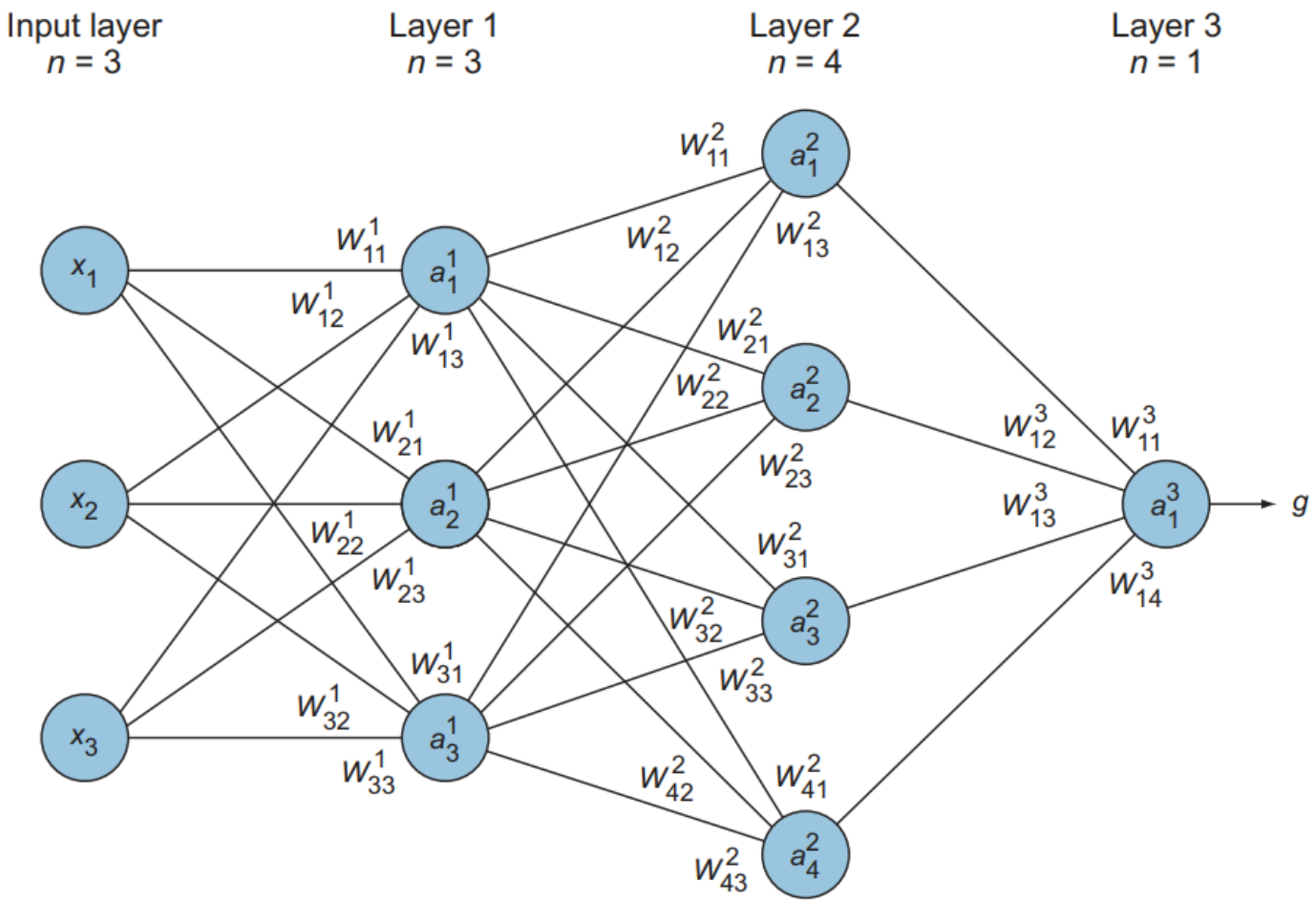
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = g(z^{[2]})$$

$$z^{[3]} = W^{[3]} a^{[2]} + b^{[3]}$$

$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]})$$

The weights break-down



Feedforward calculation

$$a_1^{(1)} = \sigma(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3)$$

$$a_2^{(1)} = \sigma(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)} x_3)$$

$$a_3^{(1)} = \sigma(w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2 + w_{33}^{(1)} x_3)$$

$$a_1^{(2)}, a_2^{(2)}, a_3^{(2)}, \text{ and } a_4^{(2)}$$

$$\hat{y} = a_1^{(2)} = \sigma\,(w_{11}^{(3)} a_1^{(2)} + w_{12}^{(3)} a_2^{(2)} + w_{13}^{(3)} a_3^{(2)} + w_{14}^{(3)} a_4^{(2)})$$

Feedforward process (2)

$$\hat{y} = \sigma\left[\begin{array}{cccc} w_{11}^{3} & w_{12}^{3} & w_{13}^{3} & w_{14}^{3} \end{array}\right] \cdot \sigma\left[\begin{array}{ccc} w_{11}^{2} & w_{12}^{2} & w_{13}^{2} \\ w_{21}^{2} & w_{22}^{2} & w_{23}^{2} \\ w_{31}^{2} & w_{32}^{2} & w_{33}^{2} \\ w_{41}^{2} & w_{42}^{2} & w_{43}^{2} \end{array}\right] \cdot \sigma\left[\begin{array}{ccc} w_{11}^{1} & w_{12}^{1} & w_{13}^{1} \\ w_{21}^{1} & w_{22}^{1} & w_{23}^{1} \\ w_{31}^{1} & w_{32}^{1} & w_{33}^{1} \end{array}\right]\left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array}\right]$$

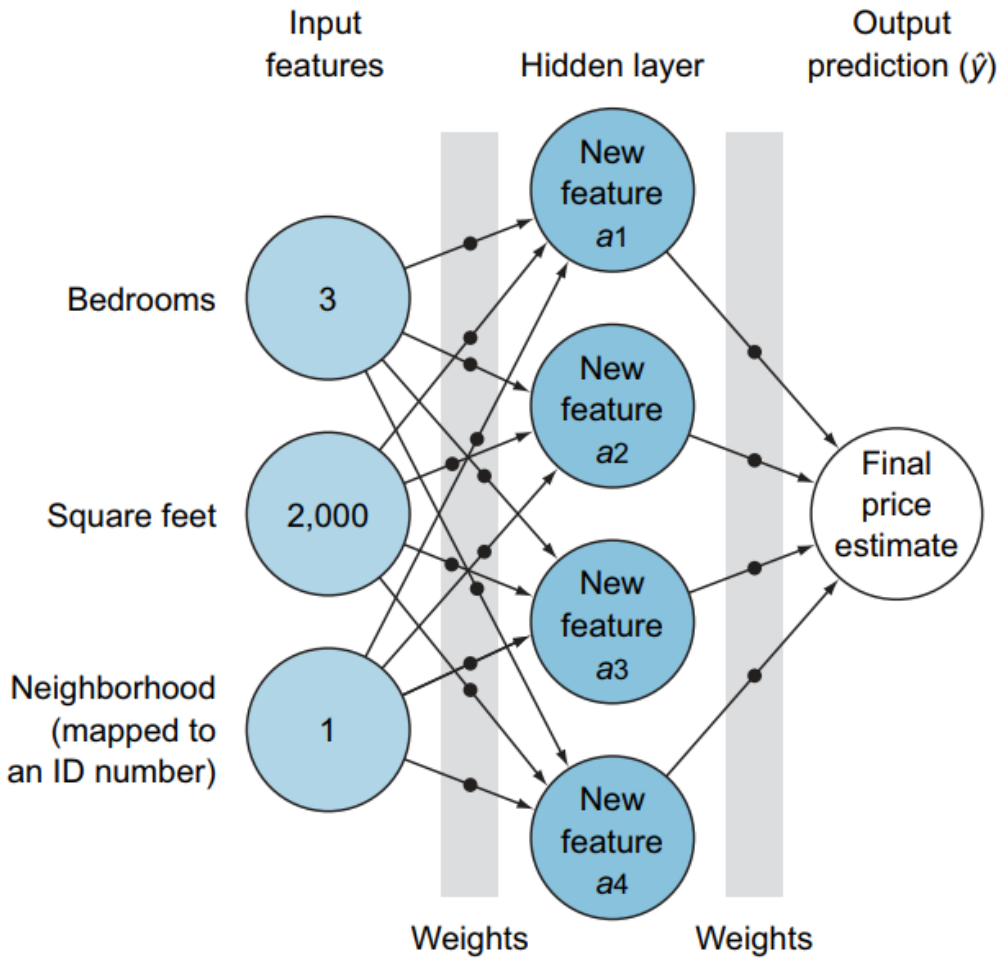
$W^{(3)}$ $W^{(2)}$ $W^{(1)}$

Layer 3 Layer 2 Layer 1 Input vector

Feature learning

- The nodes in the hidden layers (a_i) are the new features that are learned after each layer
- For example, in the network of two slides earlier, we see that we have three feature inputs ($x1$, $x2$, and $x3$)
- After computing the forward pass in the first layer, the network learns patterns, and these features are transformed to three new features with different values (a_1^1, a_2^1, a_3^1)
- Then, in the next layer, the network learns patterns within the patterns and produces new features (a_1^2, a_2^2, a_3^2 , and a_4^2, and so forth)
- The produced features after each layer are not totally understood, and we don't see them, nor do we have much control over them
  - That's why they are called *hidden* layers
- What we do is this: we look at the final output prediction and keep tuning some parameters until we are satisfied by the network's performance

Feature Learning Process



Cost/Loss/Error Function

- The *error function* is a measure of how "wrong" the neural network prediction is with respect to the expected output (the label)
- It quantifies how far we are from the correct solution
- For example, if we have a high loss, then our model is not doing a good job
- The smaller the loss, the better the job the model is doing
- The larger the loss, the more our model needs to be trained to increase its accuracy

Mean Square Error Loss Function

- Mean squared error (MSE) is commonly used in regression problems that require the output to be a real value (like house pricing)

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

- Instead of just comparing the prediction output with the label ($\hat{y}_i - y_i$), the error is squared and averaged over the number of data points
- MSE is a good choice for a few reasons
  - The square ensures the error is always positive, and
  - Larger errors are penalized more than smaller errors
- MSE is quite sensitive to outliers, since it squares the error value
- A variation error function of MSE called *mean absolute error* (MAE) is immune to this issue
  - It averages the absolute error over the entire dataset without taking the square of the error

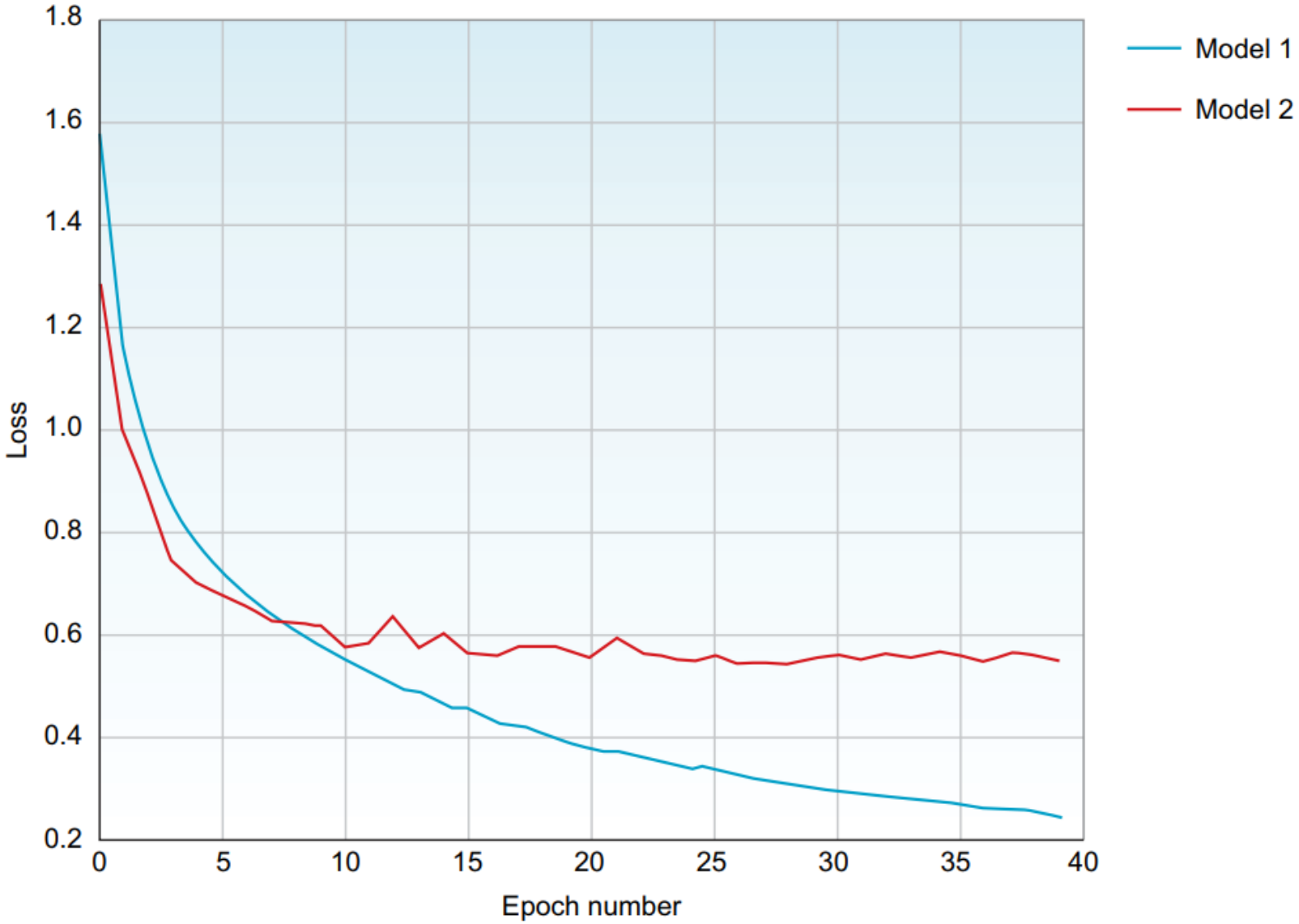
$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i|$$

Cross-entropy loss function

$$\mathcal{L}(\hat{y}, y) = -\Big[(1-y)\log(1-\hat{y}) + y\log\hat{y}\Big]$$

- The loss function $L(\hat{y}; y)$ produces a single scalar value
- Most common for Neural Networks
- Cross-entropy is commonly used in classification problems because it quantifies the difference between two probability distributions
- How close is the predicted distribution to the true distribution?
  - That is what the cross entropy loss function determines

Loss Function to compare models



Training a neural network

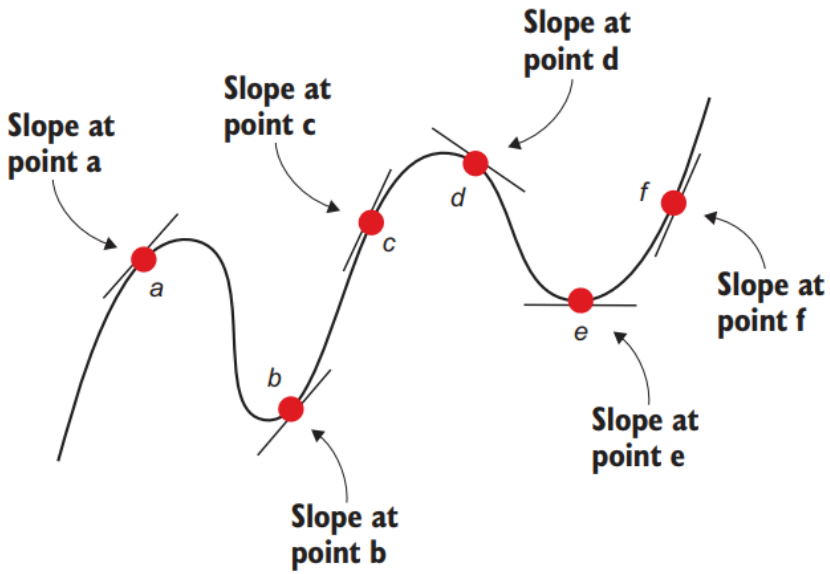
- Training a neural network involves showing the network many examples (a training dataset)
- The network makes predictions through feedforward calculations and compares them with the correct labels to calculate the error
- Finally, the neural network needs to *adjust the weights* (on all edges) until it gets the minimum error value, which means maximum accuracy
- In neural networks, optimizing the error function means updating the weights and biases until we find the *optimal weights*, or the best values for the weights to produce the minimum error
- In mathematical terms:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial W^{[\ell]}}$$

$$b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial b^{[\ell]}}$$

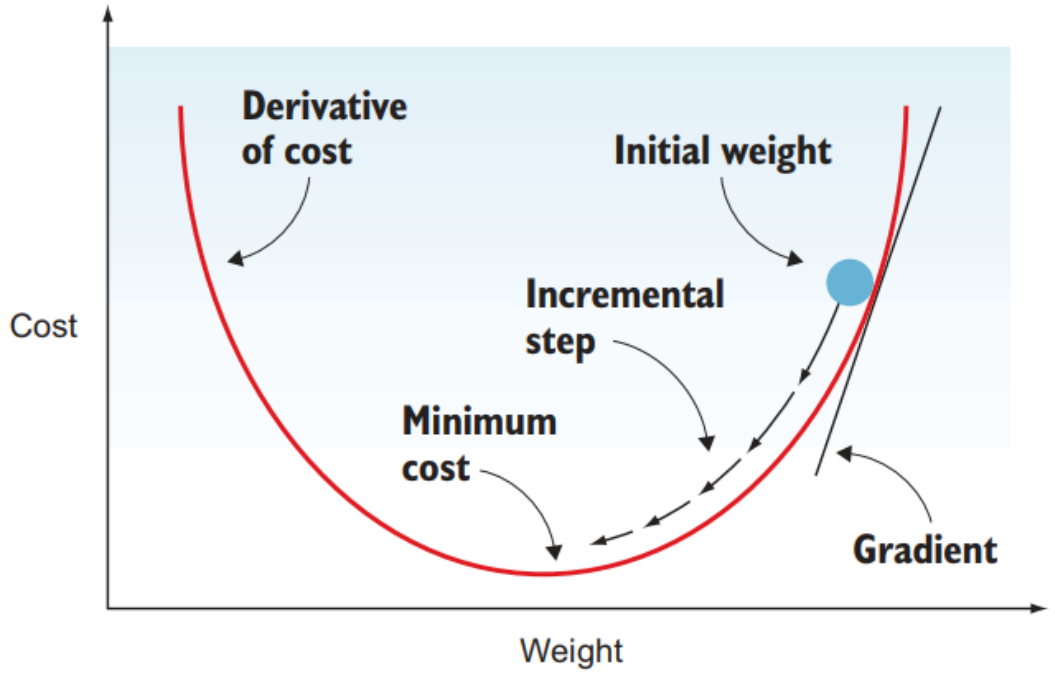
Optimization Algorithm: Gradient Descent

- The general definition of a *gradient* (also known as a *derivative*) is that it is the function that tells the slope or rate of change of the line that is tangent to the curve at any given point
- It is just a fancy term for the slope or steepness of the curve



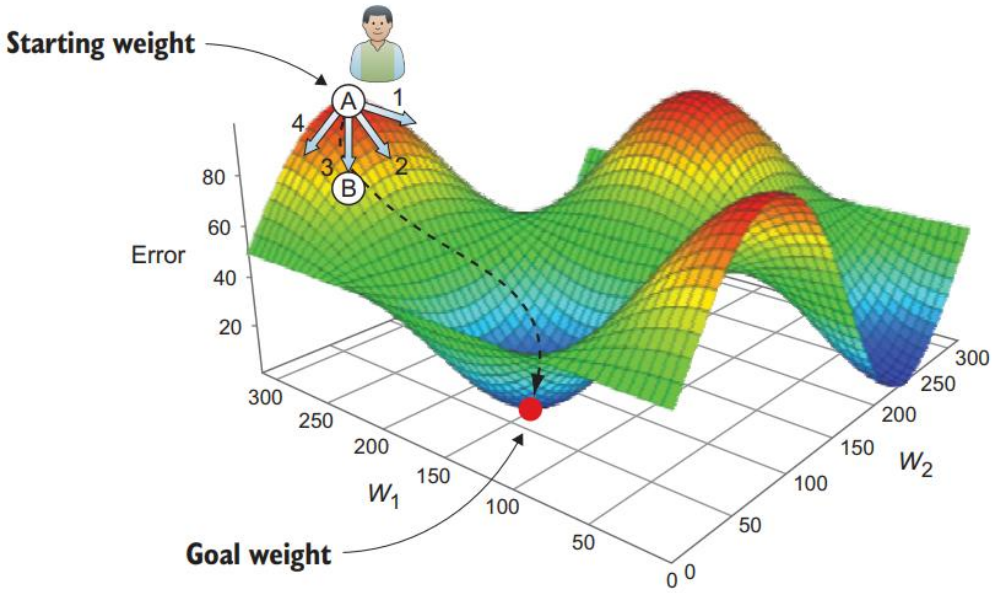
Gradient descent

- *Gradient descent* simply means updating the weights iteratively to descend the slope of the error curve until we get to the point with minimum error
- Gradient descent has several variations: batch gradient descent (BGD), stochastic gradient descent (SGD), and mini-batch GD (MB-GD)

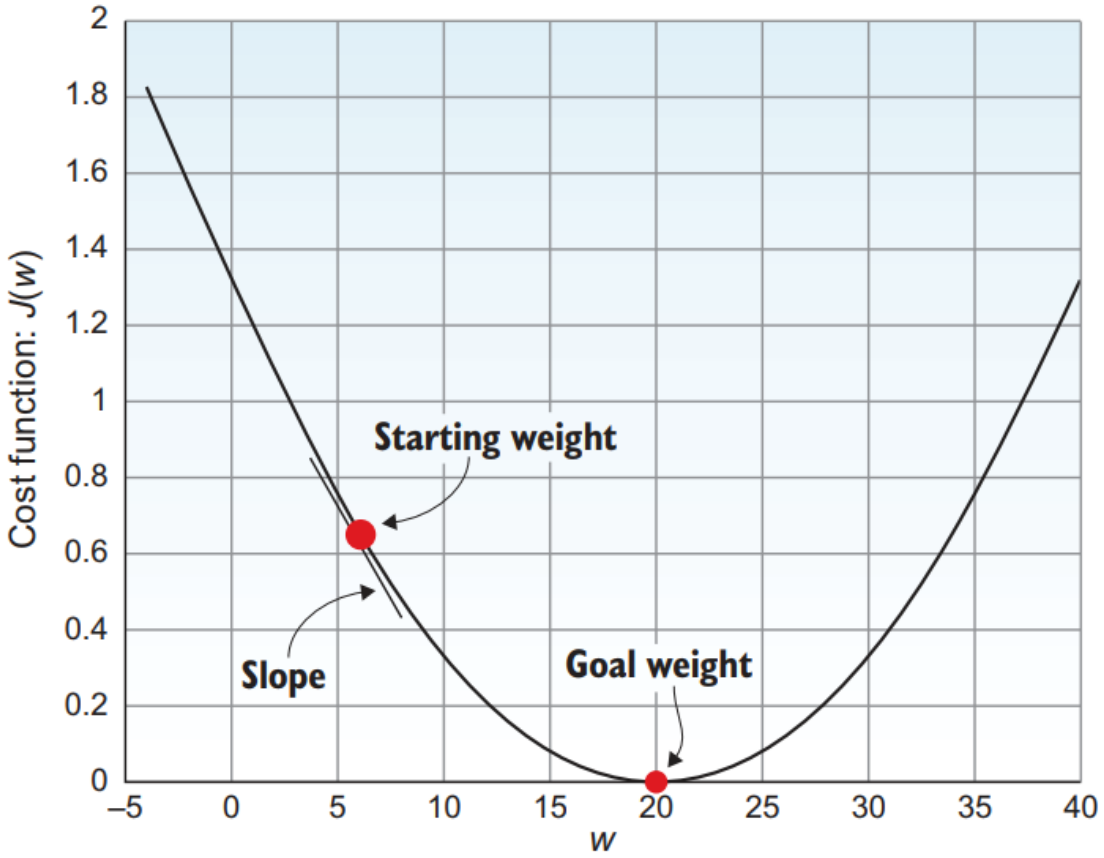


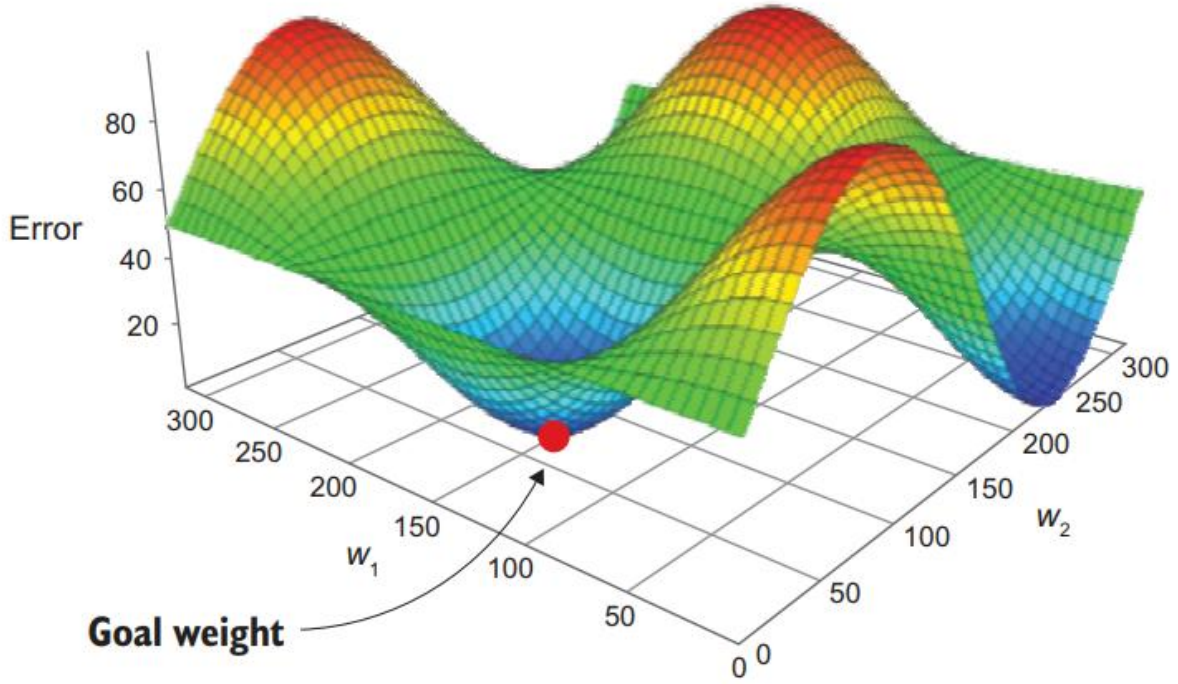
How does Gradient Descent work?

- The random initial weight (starting weight) is at point A, and our goal is to descend this error mountain to the goal *w*1 and *w*2 weight values, which produce the minimum error value
- The way we do that is by taking a series of steps *down* the curve until we get the minimum error
- In order to descend the error mountain, we need to determine two things for each step
  - The step direction (gradient)
  - The step size (learning rate)



Optimizing the network: Cost function visualization





Stochastic/Mini-batch Gradient Descent

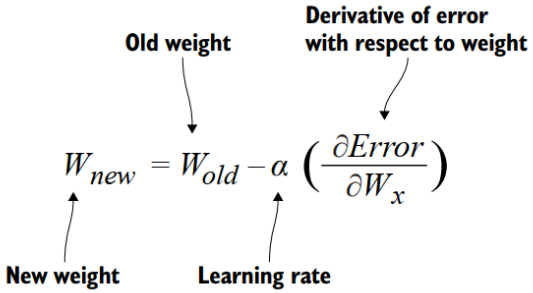
- In stochastic gradient descent, we update the weights based on each data point at a time
  - This make the algorithm jump here and there and kind of unstable sometimes
  - However, end of the day, it provides better optimization
- Mini-batch gradient descent is a compromise between Gradient Descent and stochastic GD
  - Instead of computing the gradient from one sample (SGD) or all samples (BGD), we divide the training sample into *mini-batches* from which to compute the gradient

GD vs SGD vs MBSGD

- In practice, we use mini batch gradient descent (MBSGD) algorithm
- The GD (also called BGD – batch gradient descent) almost always fails to find the optimal point (inherent in GD algorithm) and needs to load all data points to the memory before an update
- The SGD is better in finding the optimal point than the GD, but is unstable
- The MBSGD is the sweet point between GD and SGD: it updates the weight on small batches, so can use the memory optimally
  - Also better at finding the optimum than GD

Backpropagation

- Backpropagation is the core of how neural networks learn
- Up until this point, we have learned that training a neural network typically happens by the repetition of the following three steps
  - Forward pass: get the linear combination (weighted sum), and apply the activation function to get the output prediction ($\hat{y}$)
  - Find the loss: Compare the prediction with the label to calculate the error or loss function
  - Update: Use a gradient descent optimization algorithm to compute the Δw that optimizes the error function



Initialize the W/b matrices

- We cannot initialize all to *0*
  - If we do that then $z^{[3]} = W^{[3]}a^{[2]}+b^{[3]}$ will be 0
  - However, the output of the neural network is defined as $a^{[3]} = g(z^{[3]})$
  - Here $g(\cdot)$ is defined as the sigmoid function. This means $a[3] = g(0) = 0.5$. Thus, no matter what value of $x(i)$ we provide, the network will output $\hat{y} = 0.5$
- What if we had initialized all parameters to be the same non-zero value? In this case, consider the activations of the first layer

$$a^{[1]} = g(z^{[1]}) = g(W^{[1]}x^{(i)} + b^{[1]})$$

  - Each element of the activation vector $a^{[1]}$ will be the same (because $W^{[1]}$ contains all the same values)
  - This behavior will occur at all layers of the neural network