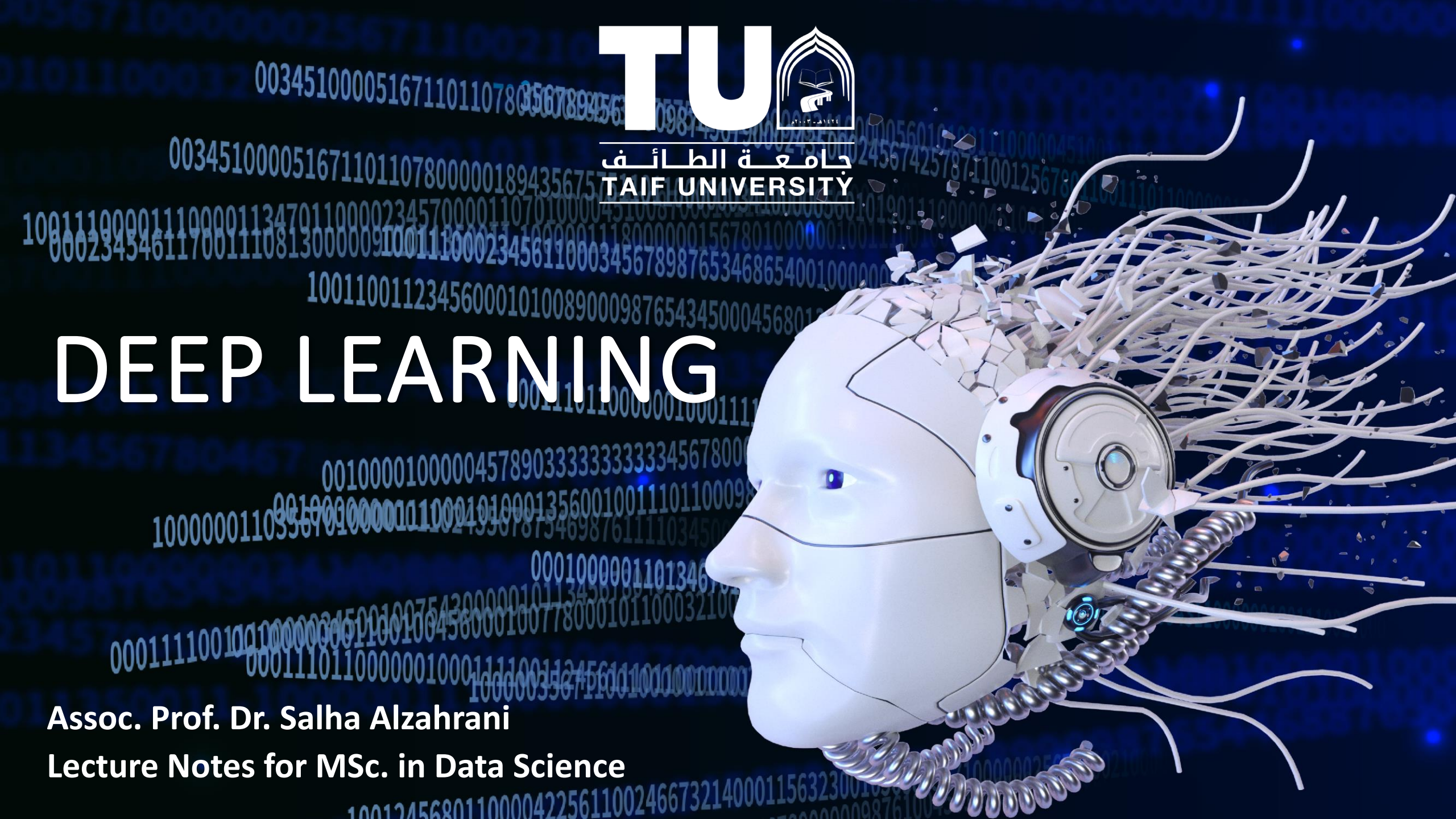
جامعة الطائف
TAIF UNIVERSITY

DEEP LEARNING



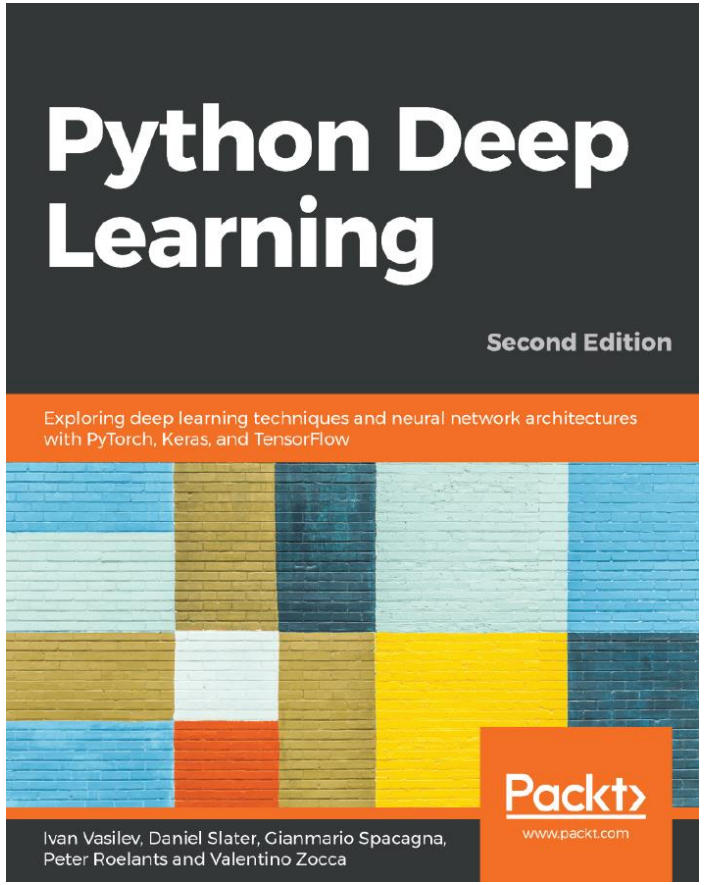
Assoc. Prof. Dr. Salha Alzahrani

Lecture Notes for MSc. in Data Science

My Notes

In this chapter, we will cover the following topics:

- The need for neural networks
- An introduction to neural networks
- Training neural networks



The need for neural networks

- Neural networks have been around for many years, and they've gone through several periods during which they've fallen in and out of favor.
- But recently, they have steadily gained ground over many other competing machine learning algorithms.
- This resurgence is due to having computers that are fast, the use of graphical processing units (GPUs) versus the most traditional use of computing processing units (CPUs), better algorithms and neural net design, and increasingly larger datasets.
- Deep learning algorithms have excelled in many real-world applications; for example, both Google Now and Apple's Siri assistants rely on deep networks for speech recognition.
- Deep learning algorithms have excelled in image classification.
    - Let's take the **ImageNet Large-Scale Visual Recognition Challenge** (http://image-net.org/challenges/LSVRC/). The participants train their algorithms using the ImageNet database. It contains more than one million high-resolution color images in over a thousand categories (cars, people, trees, and so on).
    - One of the tasks in the challenge is to **classify unknown images in these categories**.
    - In 2011, the winner achieved a top-five accuracy of 74.2%.
    - In 2012, another team used a convolutional network (a special type of deep network) and won a top-five accuracy of 84.7%.
    - Since then, the winners have always been convolutional networks and the current top-five accuracy is 97.7%.

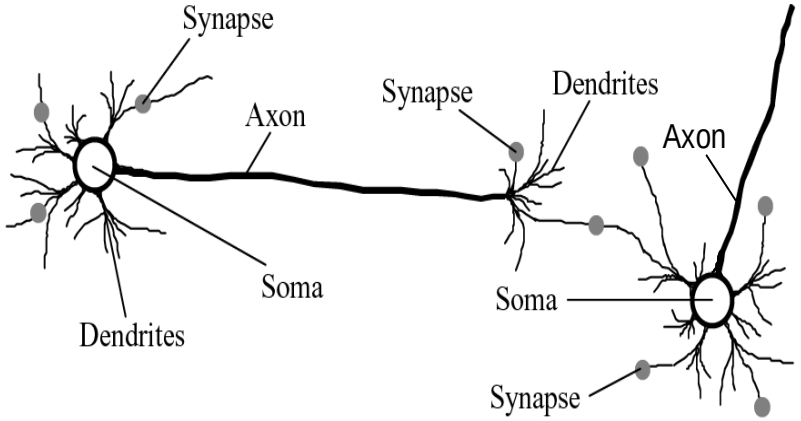
An introduction to neural networks

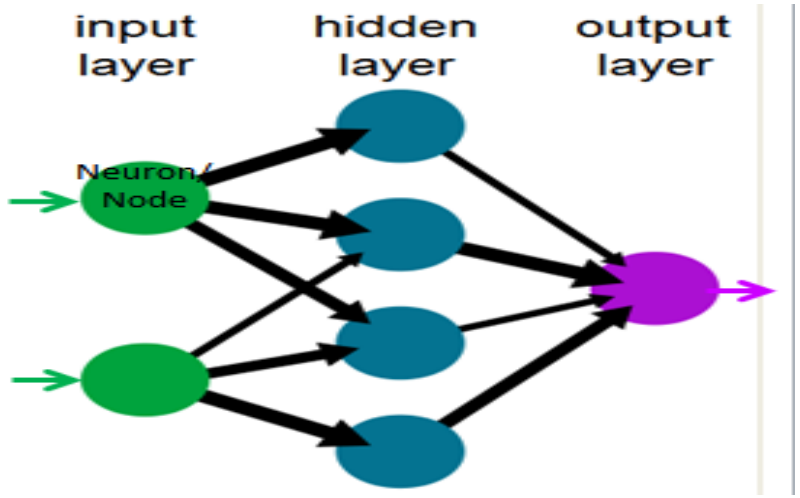
A **neural net** is not a fixed program, but rather a **model**, a system that processes information, or inputs. The characteristics of a neural network are as follows:

- Information processing occurs in its simplest form, over simple elements called **neurons**.
- Neurons are connected and they exchange signals between them through **connection links**.
- Connection links between neurons can be stronger or weaker (via **weights**), and this determines how information is processed.
- Each neuron has an internal **state** that is determined by all the incoming connections from other neurons.
- Each neuron has a different **activation function** that is calculated on its state, and determines its output signal.

Analogy between biological and artificial neural networks

(BNN)	(ANN)
Soma	Neuron (or Node)
Dendrite	Input
Axon	Output
Synapse	Weight





An introduction to neural networks

Pro Tip

Initially, neural networks were inspired by the biological brain (hence the name). Over time, however, we've stopped trying to emulate how the brain works and instead we focused on finding the correct configurations for specific tasks including computer vision, natural language processing, and speech recognition. You can think of it in this way: for a long time, we were inspired by the flight of birds, but, in the end, we created airplanes, which are quite different. We are still far from matching the potential of the brain. Perhaps the machine learning algorithms in the future will resemble the brain more, but that's not the case now. Hence, for the rest of this book, we won't try to create analogies between the brain and neural networks.

An introduction to neural networks

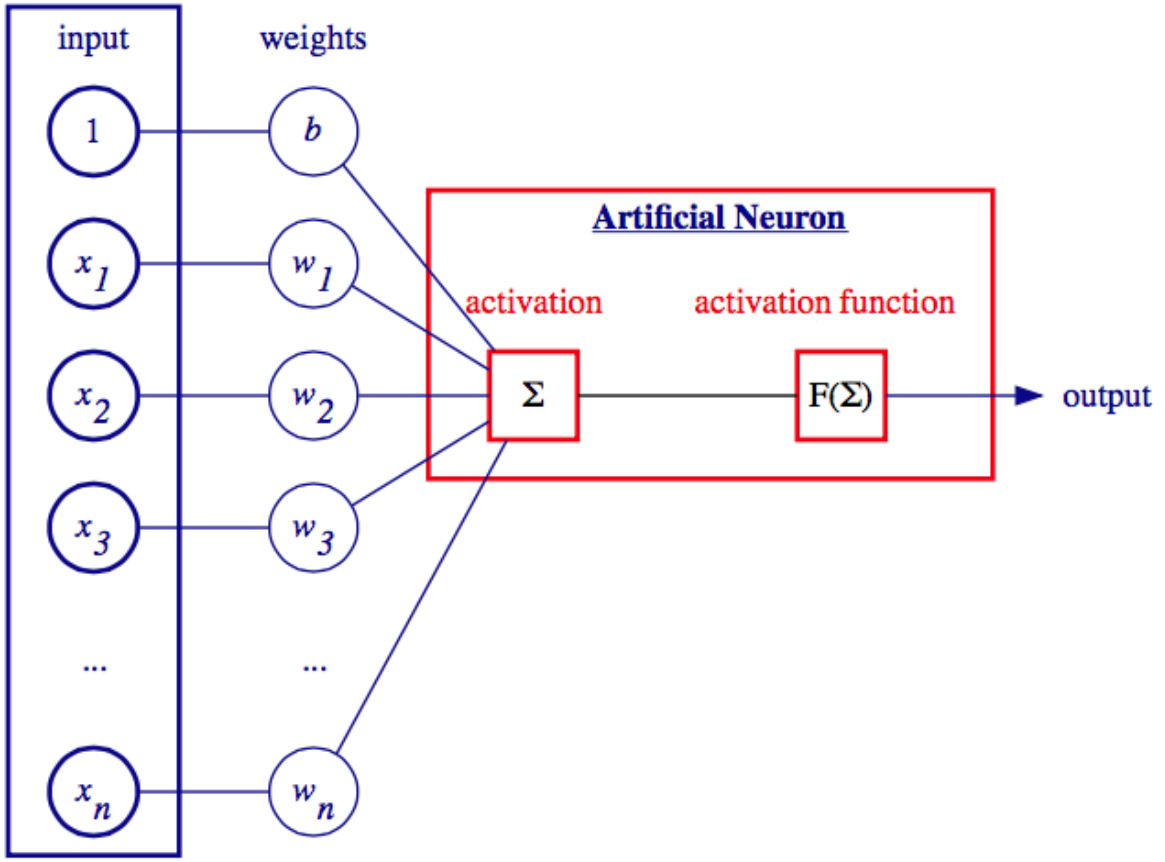
A more general description of a neural network would be as a **computational graph of mathematical operations**.

We can identify two main characteristics for a neural net:

1) **The neural net architecture:** This describes the set of connections between the neurons namely:
   - feedforward,
   - recurrent,
   - multi or single-layered,
   - the number of layers,
   - the number of neurons in each layer.
2) **The learning:** This describes what is commonly defined as the **training**. The most common but not exclusive way to train a neural network is with:
   - the gradient descent,
   - backpropagation.

Neurons

- A neuron is a mathematical function that takes one or more input values, and outputs a single numerical value.



- In this diagram, we can see the different elements of the neuron. The neuron is defined as follows:

$$y = f(\sum_i x_i w_i + b)$$

Neurons

1. First, we compute the weighted sum $\sum x_i w_i$ of the inputs x_i and the weights w_i (also known as an **activation value**).

   Here, x_i is either numerical values that represent the input data, or the outputs of other neurons (that is, if the neuron is part of a neural network):

   - The weights w_i are numerical values that represent either the strength of the inputs or, alternatively, the strength of the connections between the neurons.
   - The weight b is a special value called bias whose input is always 1.

2. Then, we use the result of the weighted sum as an input to the **activation function** f, which is also known as **transfer function**. There are many types of activation functions, but they all have to satisfy the requirement to be **non-linear**, which we'll explain later in the chapter.

Layers

- A neural network can have an indefinite number of neurons, which are organized in **interconnected layers** :
  - **The input layer** represents the dataset and the initial conditions. For example, if the input is a grayscale image, the input layer is the intensity of one pixel of the image. For this very reason, we don't generally count the input layer as a part of the other layers. When we say 1-layer net, we actually mean that it is a simple network with just a single layer, the output, in addition to the input layer.

  - **The output layer** can have more than one neuron. This is especially useful in classification, where each output neuron represents one class. For example, in the case of the Modified National Institute of Standards and Technology (MNIST) dataset, we'll have 10 output neurons, where each neuron corresponds to a digit from 0-9. In this way, we can use the 1-layer net to classify the digit on each image. We'll determine the digit by taking the output neuron **with the highest activation function value**.

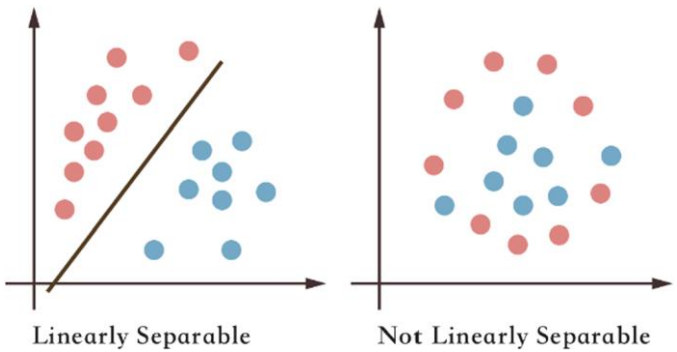
Layers

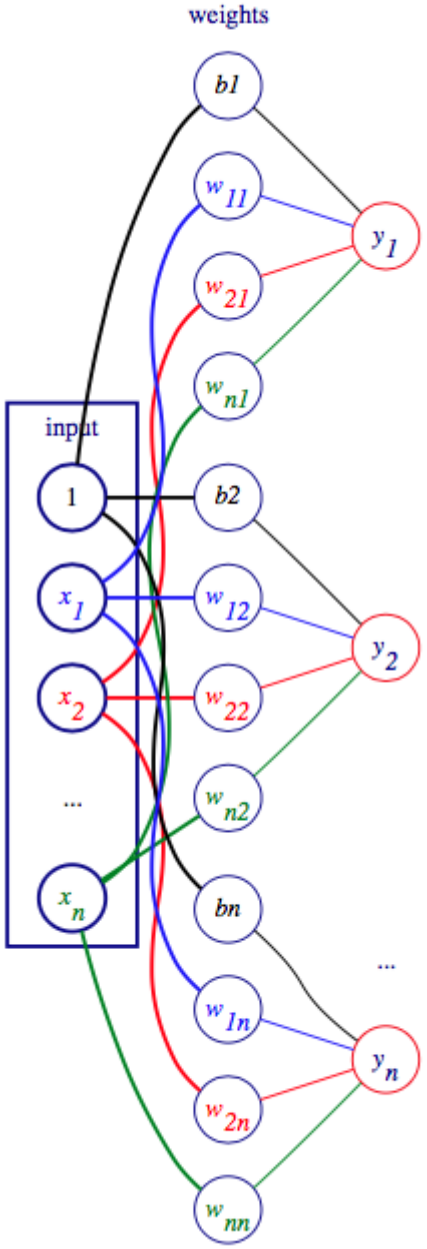
How do we choose the architecture of a neural network?

- The number of neurons in the **input layer** is decided according to the number of **features** in your dataset (i.e. one input neuron for each feature)
- The number of neurons in the **output layer** is decided as follows:
  - Classification tasks ➔ number of neurons in the output layer = number of classes (i.e. one neuron for each class)
  - Regression tasks ➔ number of neurons in the output layer = one neuron to estimate or predict the value

1-layer feedforward network

- This diagram shows the 1-layer feedforward network. Weight w_{ij} connects the i-th input neuron with the j-th output neuron. The first input, 1, is the bias unit, and the weight, b1, is the bias weight:
- The neurons on the left represent the input with bias b, the middle column represents the weights for each connection, and the neurons on the right represent the output given the weights w.
- The neurons of one-layer can be connected to the neurons of other layers, but not to other neurons of the same layer. In this case, the input neurons are connected only to the output neurons.
- **Limitation:** 1-layer neural nets can only classify linearly separable classes.

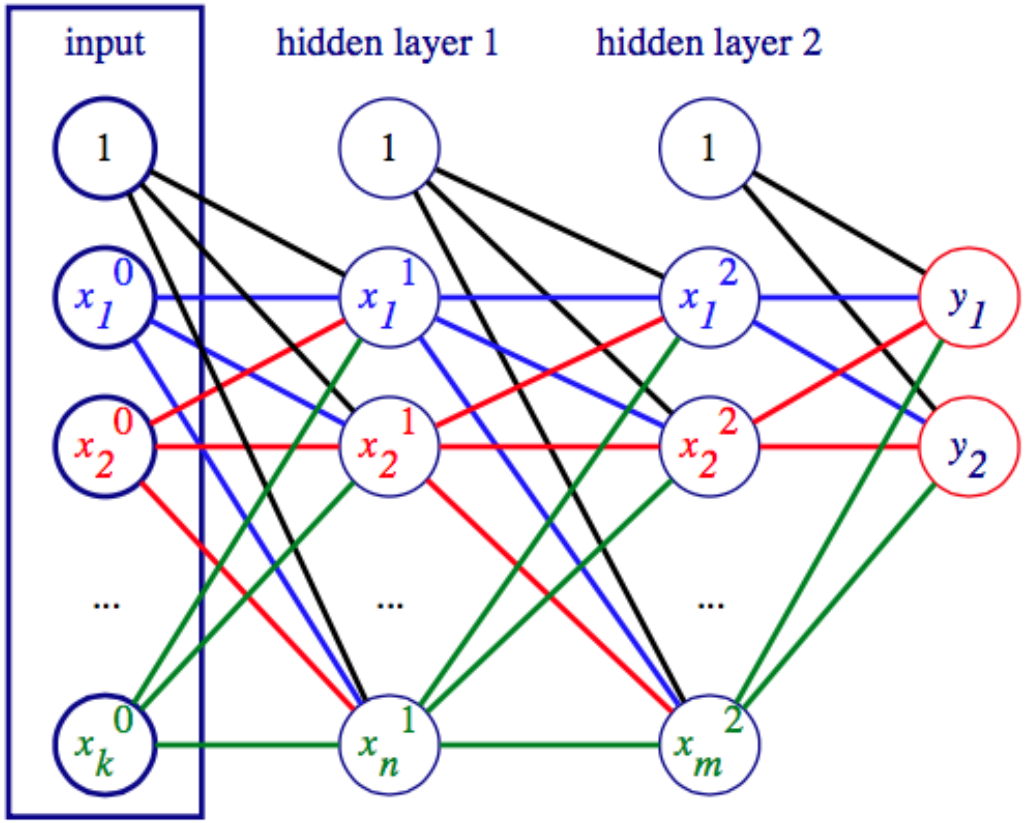




Multi-layer neural networks

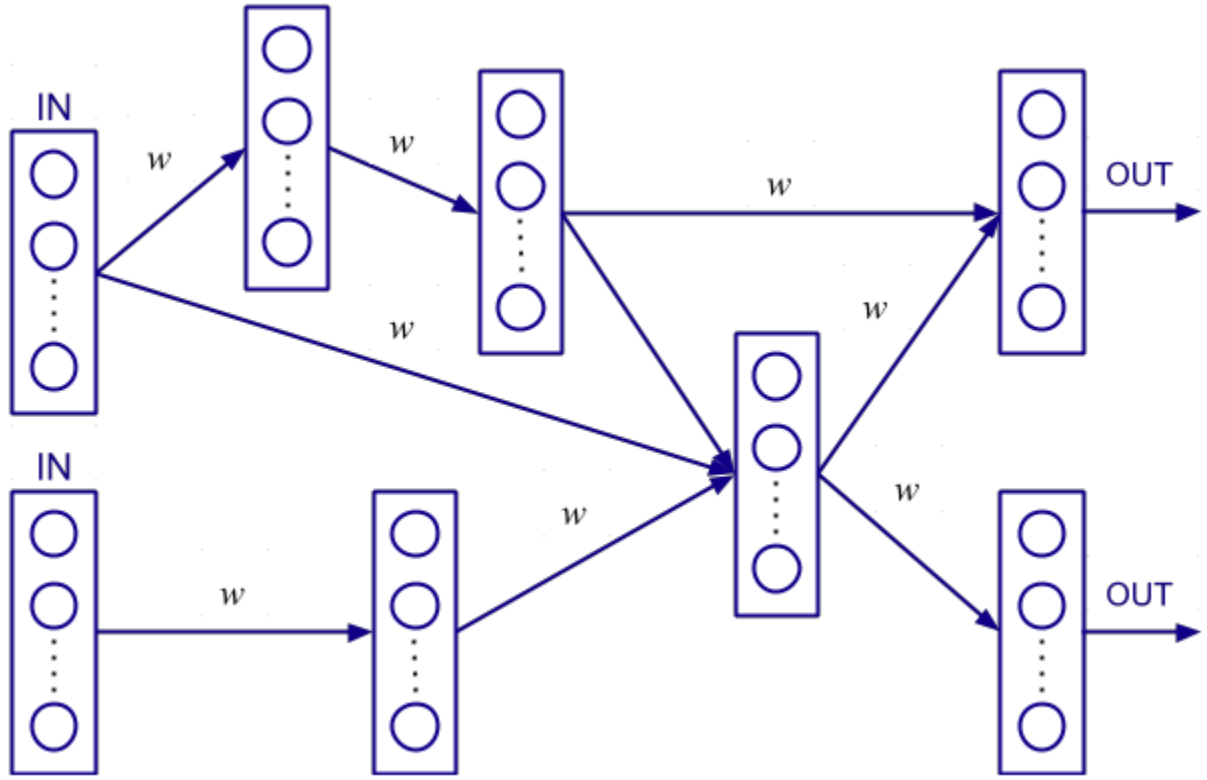
- We can add more layers between the input and the output to solve more problems. These extra layers are called **hidden layers**.

- This diagram demonstrates a **3-layer fully connected neural network with two hidden layers**.
  - The input layer has *k* input neurons,
  - the first hidden layer has *n* hidden neurons,
  - the second hidden layer has *m* hidden neurons.
  - The output, in this example, is the two classes *y1* and *y2*.
  - On top is the always-on bias neuron.
  - A unit from one-layer is connected to all units from the previous and following layers (hence fully connected).
  - Each connection has its own weight, *w*, not shown for simplicity.



Multi-layer neural networks

- But we are not limited to networks with sequential layers, as shown in the preceding diagram.
- The neurons and their connections form **directed cyclic graphs**. We also chose to organize them in layers; therefore, the layers are also organized in the directed cyclic graph.
- The diagram also depicts a valid neural network with two input layers, two output layers, and randomly interconnected hidden layers.
- For the sake of simplicity, we've depicted the multiple weights, *w*, connecting the layers as a single line.



Multi-layer neural networks

Pro Tip

There is a special class of neural networks called recurrent networks, which represent a directed cyclic graph (they can have loops). We'll discuss them in detail in chapter 8, *Reinforcement Learning Theory*.

Different types of activation function

We now know that multi-layer networks can classify linearly inseparable classes. But to do this, they need to satisfy one more condition. If the neurons don't have activation functions, their output would be the weighted sum of the inputs, $\sum_i w_i x_i$, which is a **linear function**. Then the entire neural network, that is, a composition of neurons, becomes a composition of linear functions, which is also a linear function. This means that even if we add hidden layers, the network will still be equivalent to a simple linear regression model, with all its limitations. To turn the network into a **non-linear** function, we'll use non-linear activation functions for the neurons. Usually, all neurons in the same layer have the same activation function, but different layers may have different activation functions. The most common activation functions are as follows:

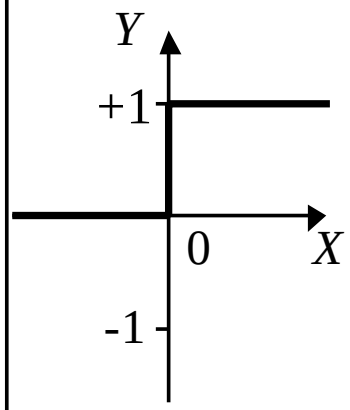
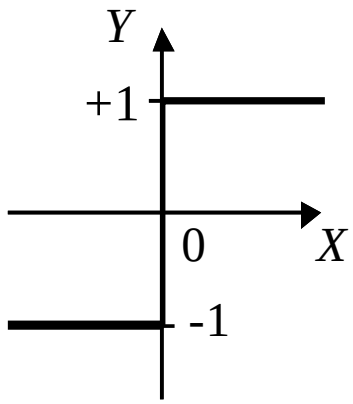
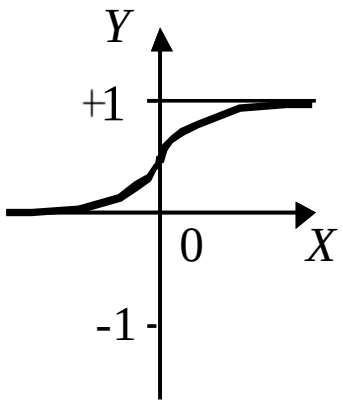
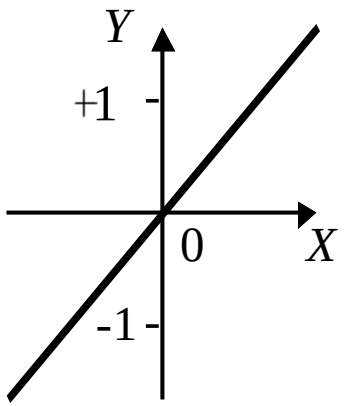
Different types of activation function

- $f(a) = a$: This function lets the activation value go through and is called the **identity function.**
- $f(a) = \begin{cases} 1 \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$: This function activates the neuron; if the activation is above a certain value, it's called the **threshold activity function.**
- $f(a) = \frac{1}{1+exp(-a)}$: This function is one of the most commonly used, as its output is bounded between 0 and 1, and it can be interpreted stochastically as the probability of the neuron activating. It's commonly called the **logistic function**, or the **logistic sigmoid.**

Different types of activation function

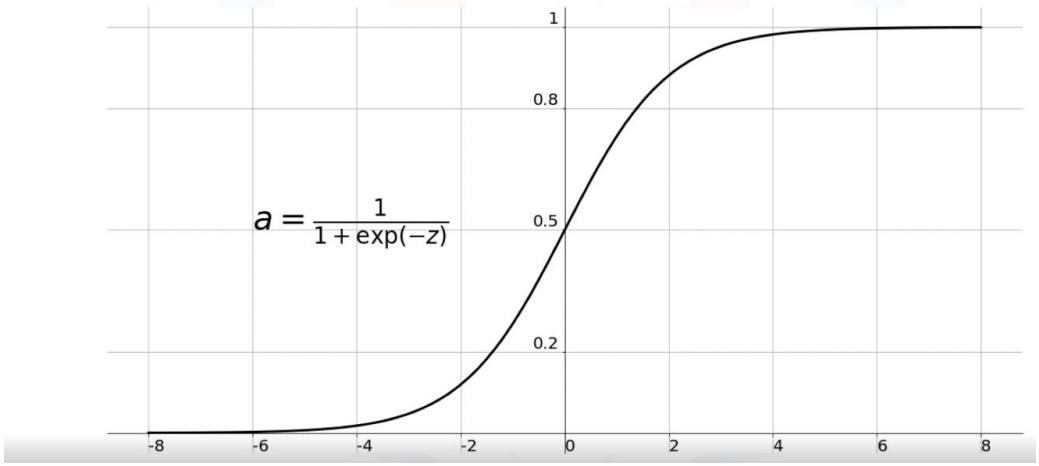
- $f(a) = \frac{2}{1+exp(-a)} - 1 = \frac{1-exp(-a)}{1+exp(-a)}$: This activation function is called **bipolar sigmoid**, and it's simply a logistic sigmoid rescaled and translated to have a range in (-1, 1).
- $f(a) = \frac{exp(a)-exp(-a)}{exp(a)+exp(-a)} = \frac{1-exp(-2a)}{1+exp(-2a)}$: This activation function is called the **hyperbolic tangent** (or **tanh**).
- $f(a) = \begin{cases} a \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$: This activation function is probably the closest to its biological counterpart. It's a mix of the identity and the threshold function, and it's called the **rectifier**, or **ReLU**, as in **Rectified Linear Unit**. There are variations on the ReLU, such as Noisy ReLU, Leaky ReLU, and ELU (Exponential Linear Unit).

Different types of activation function

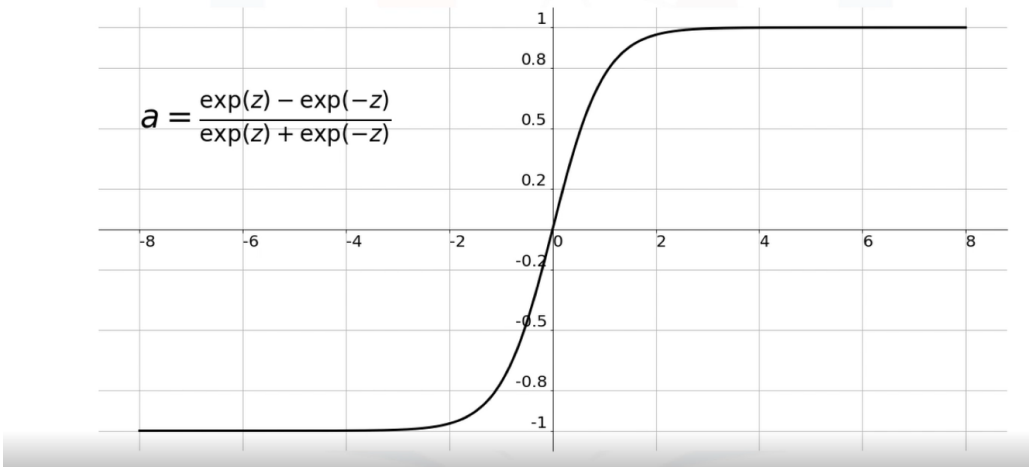
Step function	*Sign function*	*Sigmoid function*	*Linear function*
			
$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{if } X < 0 \end{cases}$	$Y^{sign} = \begin{cases} +1, \text{if } X \geq 0 \\ -1, \text{if } X < 0 \end{cases}$	$Y^{sigmoid} = \frac{1}{1+e^{-X}}$	$Y^{linear} = X$

Different types of activation function

Sigmoid Function

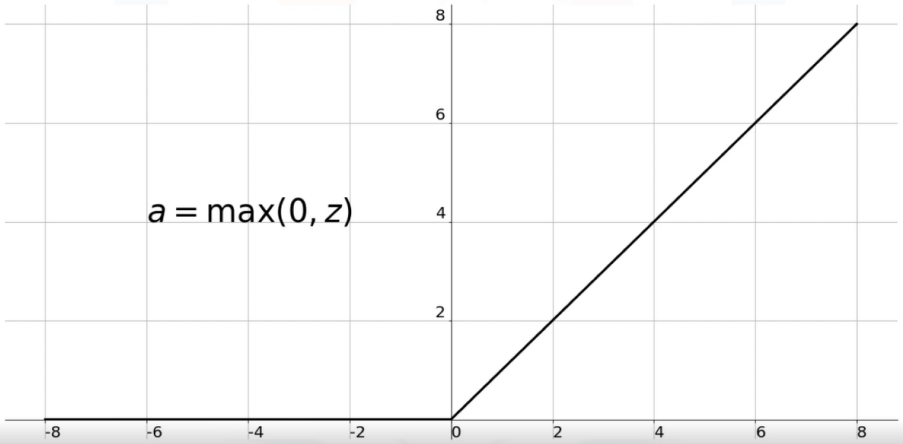


Hyperbolic Tangent Function



Different types of activation function

ReLU Function



Softmax Function

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1.6 \\ 0.55 \\ 0.98 \end{bmatrix}$$

$$a_i = \frac{e^{z_i}}{\sum_{k=1}^{m} e^{z_k}}$$

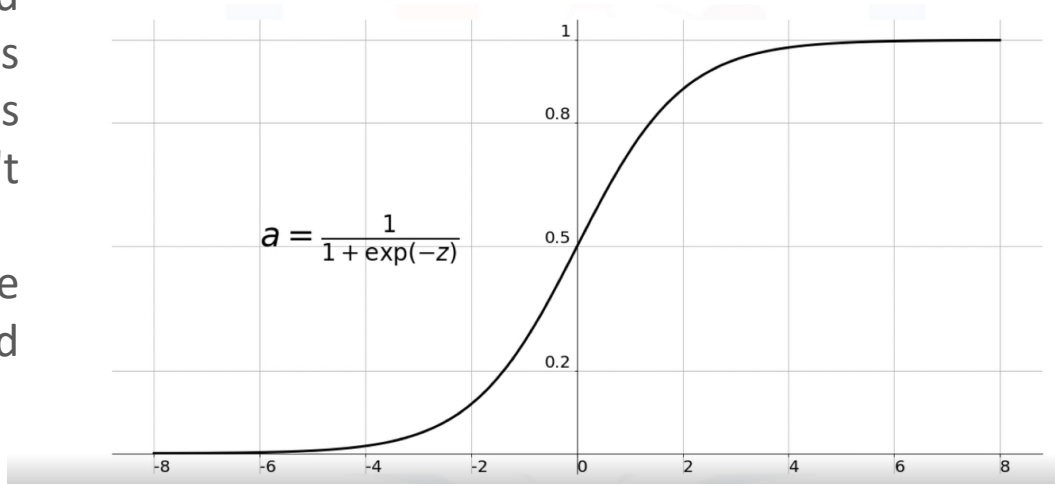
Softmax

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0.51 \\ 0.18 \\ 0.31 \end{bmatrix}$$

Sigmoid function

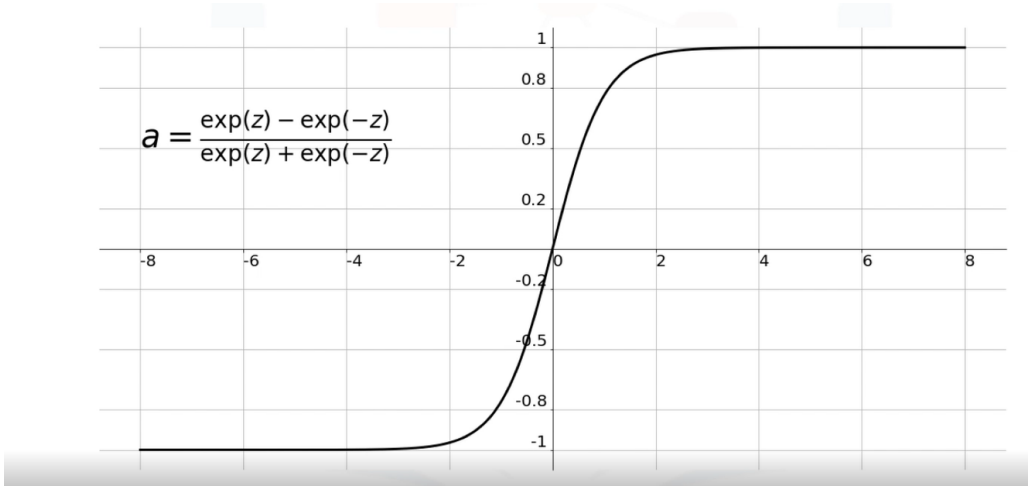
- At z = 0, a is equal to 0.5 and when z is a very large positive number, a is close to 1, and when z is a very large negative number, a is close to zero.
- Sigmoid functions used to be widely used as activation functions in the hidden layers of a neural network.

- However, as you can see, the function is pretty flat beyond the +4 and -4 region. This means that once the function falls in that region, the gradients become very small. This results in the **vanishing gradient problem**, as the network doesn't really learn.
- Another problem with the sigmoid function is that the values only range from 0 to 1. This means that the sigmoid function is **not symmetric around the origin**.



Hyperbolic tangent, or tanh, function

- It is very similar to the sigmoid function.
- It is actually just a scaled version of the sigmoid function, but unlike the sigmoid function, it's **symmetric over the origin**. It ranges from -1 to +1.
- However, although it overcomes the lack of symmetry of the sigmoid function, it also leads to the **vanishing gradient problem** in very deep neural networks.

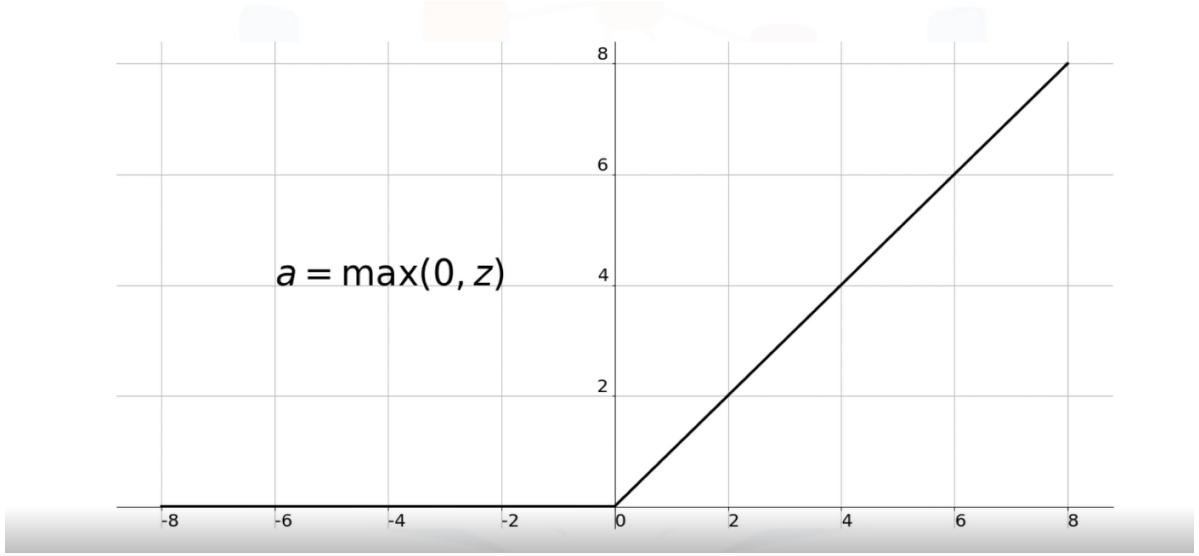


The rectified linear unit, or ReLU, function

- The most widely used activation function when designing networks today.
- In addition to it being nonlinear, the main advantage of using the ReLU function over the other activation functions is that **it does not activate all the neurons at the same time**.

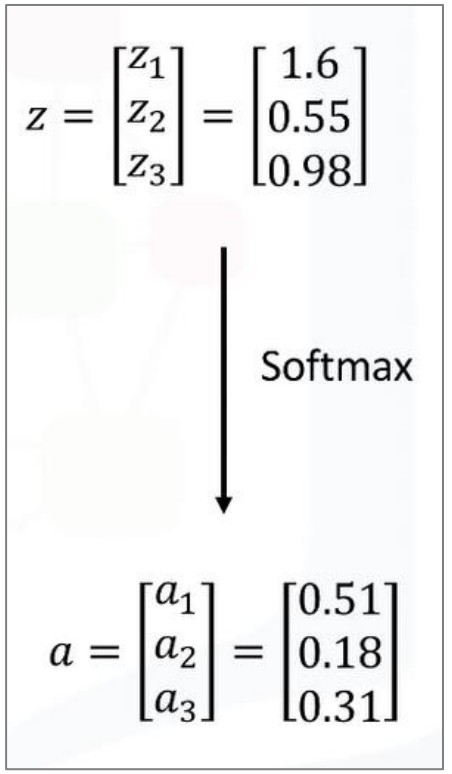
- According to the plot, if the input is negative, it will be converted to 0, and the neuron does not get activated. This means that at a time, only a few neurons are activated, making the network sparse and very efficient.

- Also, the ReLU function was one of the main advancements in the field of deep learning that **led to overcoming the vanishing gradient problem**.



Softmax function

- It is also a type of a sigmoid function, but it is handy when we are trying to handle **classification** problems.
- It is ideally used in the output layer of the classifier where we are actually trying to get the probabilities to define the class of each input.
- Example, if a network with 3 neurons in the output layer
- This way, it is easier for us to classify a given data point and determine to which category it belongs.



Training neural networks

- We have seen how neural networks can map inputs onto determined outputs, depending on fixed weights.
- Once the architecture of the neural network has been defined and includes the feed forward network, the number of hidden layers, the number of neurons per layer, and the activation function, we'll need to set the weights, which, in turn, will define the internal states for each neuron in the network.
- First, we'll see how to do that for a 1-layer network using an optimization algorithm called **gradient descent**.
- Second, we'll extend it to a deep feed forward network with the help of **backpropagation**.

Training neural networks

- The general concept we need to understand is the following:

  - Every neural network is an approximation of a function, so each neural network will not be equal to the desired output, but instead will differ by some value called **error**.
  - During training, the aim is **to minimize this error**.
  - Since the error is a function of the weights of the network, we want to minimize the error with respect to the weights.
  - The error function is a function of many weights and, therefore, a function of many variables. Mathematically, the set of points where this function is zero represents a hypersurface, and to find a minimum on this surface, we want to pick a point and then follow a curve in the direction of the minimum.

Training neural networks

- How to train linear regression with gradient descent ?
  - The output of a linear regression algorithm is a single value, *y*, and is equal to the dot product of the input values x and the weights *w*:

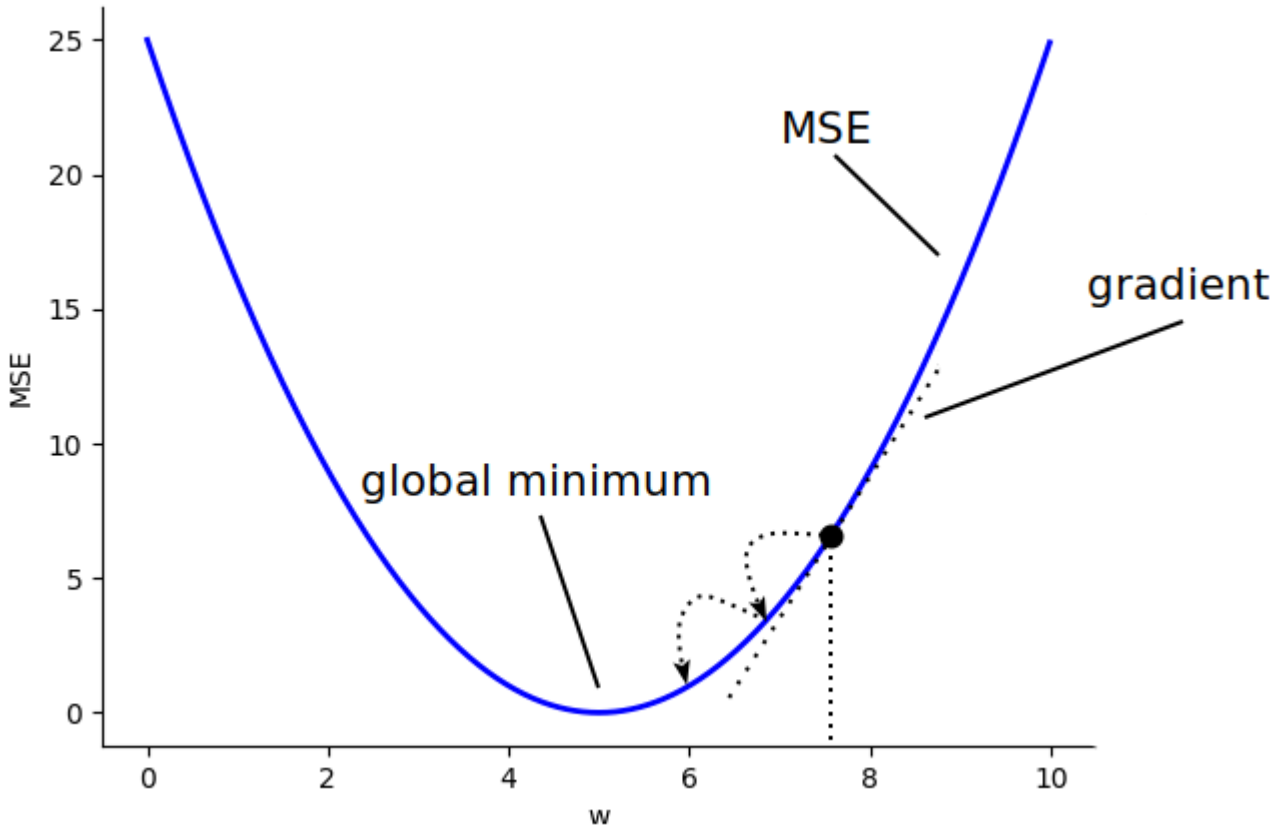
$$y = \vec{x} \cdot \vec{w}$$

  - Linear regression is a special case of a neural network; that is, it's **a single neuron with the identity activation function**. We'll learn how to train linear regression with gradient descent and, then, we'll extend it to training more complex models. You can see how the gradient descent works in the following code block:

```
Initialize the weights w with some random values
repeat:
    # compute the mean squared error (MSE) loss function for all samples of the training set
    # we'll denote MSE with J
    J = MSE = 1/n * sum_{i=0}^{n} (y^i - t^i)^2 = 1/n * sum_{i=0}^{n} (x^i • w - t^i)^2
    # update the weights w based on the derivative of J with respect to each weight
    w → w - λ∇(J(w))
until MSE falls below threshold
```

Training neural networks

- Our goal is to adjust the weights, w, in a way that will help the algorithm to predict the target values.
- To do this, first we need to know how the output y^i differs from the target value t^i for each sample of the training dataset (we use superscript notation to mark the i-th sample).
- We'll use the mean-squared error loss function (MSE), which is equal to the mean value of the squared differences $y^i - t^i$ for all samples (the total number of samples in the training set is n). Each y^i is a function of w, and therefore, J is also a function of w.
- As we mentioned previously, the loss function J represents a hypersurface of dimension equal to the dimension of w (we are implicitly also considering the bias).
- To illustrate this, imagine that we have only one input value, x, and a single weight, w. We can see how the MSE changes with respect to w in the diagram:



Training neural networks

- How to train in a multi-layer network with **backpropagation**?
  - Calculate the error in the final hidden layer.
  - Propagate that error back from the last layer to the first layer; hence, we get the name backpropagation.

The update rule for the weights of each layer is given by the following equation:

$$w_{i,j} \rightarrow w_{i,j} - \eta \delta_j y_i$$

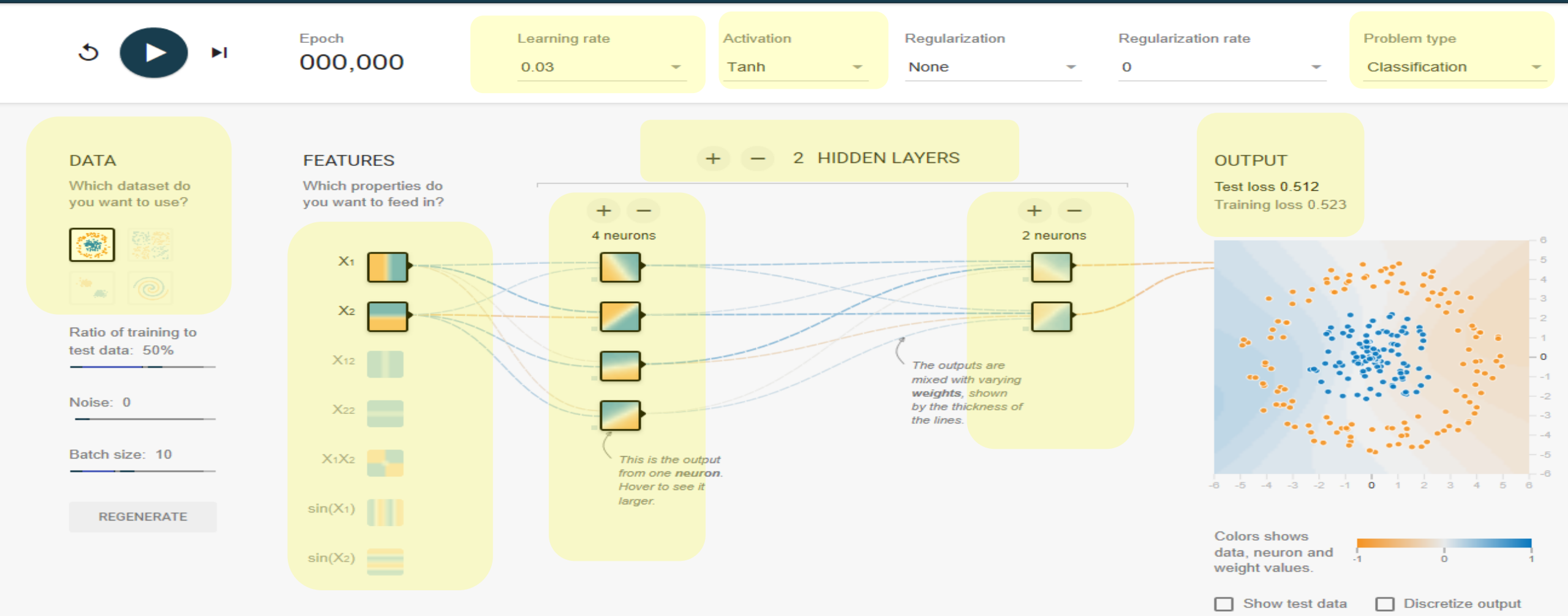
$$\delta_j = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial a_j}$$

$$\frac{\partial a_j}{\partial w_{i,j}} = y_i$$

Watch this video: What is Neural Networks in 5 Minutes!

Let's have fun ☺

Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.



Framework to build a neural network using Keras

Keras is the high-level API of TensorFlow 2: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

https://keras.io/

Framework to build a neural network using Keras

1. Prepare & Explore Dataset
- Clean data
- Split into train and test
- Split into predictors and target
- Normalize data

2. Define the neural network architecture

3. Compile the neural net

4. Fit / train the neural net

5. Evaluate the neural net

6. Make predictions/ classifications for unseen data

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Regression Example

In this example you learn how to build a **regressor** with neural networks using Keras

Dataset Example

concrete_data.csv

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

Regression Example

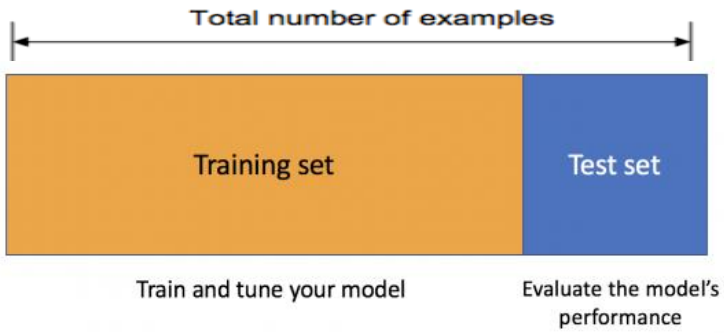
1. Prepare & Explore Dataset

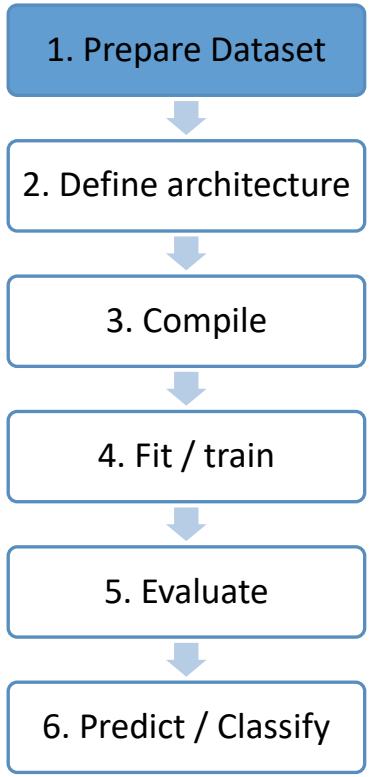
In [1]:
```
import numpy as np
import pandas as pd
```

In [3]:
```
#read the dataset
concrete_data = pd.read_csv('concrete_data.csv')
```

In [4]:
```
concrete_data
```

In [5]:
```
#split the dataset into train and test
train_data = concrete_data.sample(frac=0.8, random_state=0)
test_data = concrete_data.drop(train_data.index)
```





Regression Example

```
In [4]: concrete_data
```

Out[4]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30
...	...	...	...	...	...	...	...	...	...
1025	276.4	116.0	90.3	179.6	8.9	870.1	768.3	28	44.28
1026	322.2	0.0	115.6	196.0	10.4	817.9	813.4	28	31.18
1027	148.5	139.4	108.6	192.7	6.1	892.4	780.0	28	23.70
1028	159.1	186.7	0.0	175.6	11.3	989.6	788.9	28	32.77
1029	260.9	100.5	78.3	200.6	8.6	864.5	761.5	28	32.40

1030 rows × 9 columns

Regression Example

```
In [6]: train_data
```

Out[6]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
747	500.0	0.0	0.0	200.0	0.0	1125.0	613.0	3	26.06
718	122.6	183.9	0.0	203.5	0.0	958.2	800.1	7	10.35
175	362.6	189.0	0.0	164.9	11.6	944.7	755.8	91	79.30
828	522.0	0.0	0.0	146.0	0.0	896.0	896.0	28	74.99
713	157.0	236.0	0.0	192.0	0.0	935.4	781.2	3	9.69
...	...	...	...	...	...	...	...	...	...
673	212.0	141.3	0.0	203.5	0.0	973.4	750.0	7	15.03
595	186.2	124.1	0.0	185.7	0.0	1083.4	764.3	28	17.60
445	165.0	0.0	143.6	163.8	0.0	1005.6	900.9	56	36.56
117	313.3	262.2	0.0	175.5	8.6	1046.9	611.8	28	59.80
464	167.0	75.4	167.0	164.0	7.9	1007.3	770.1	100	56.81

824 rows × 9 columns

Regression Example

```
In [7]: test_data
```

Out[7]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
11	198.6	132.4	0.0	192.0	0.0	978.4	825.5	28	28.02
19	475.0	0.0	0.0	228.0	0.0	932.0	594.0	180	42.62
23	139.6	209.4	0.0	192.0	0.0	1047.0	806.9	180	44.21
25	380.0	0.0	0.0	228.0	0.0	932.0	670.0	270	53.30
28	427.5	47.5	0.0	228.0	0.0	932.0	594.0	28	37.43
...	...	...	...	...	...	...	...	...	...
999	150.9	0.0	183.9	166.6	11.6	991.2	772.2	28	15.57
1003	366.0	187.0	0.0	191.3	6.6	824.3	756.9	28	65.91
1014	132.0	206.5	160.9	178.9	5.5	866.9	735.6	28	33.31
1018	321.4	0.0	127.9	182.5	11.5	870.1	779.7	28	37.27
1021	298.2	0.0	107.0	209.7	11.1	879.6	744.2	28	31.88

206 rows × 9 columns

Regression Example

Predictors and Target

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

predictors

target

Regression Example

In [8]:
```
#split train and test data into predictors and target
train_predictors = train_data[train_data.columns [train_data.columns  != 'Strength']]
train_target = train_data['Strength']

test_predictors = test_data[test_data.columns [test_data.columns  != 'Strength']]
test_target = test_data['Strength']
```

In [18]:
```
train_predictors
```

Out[18]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
747	500.0	0.0	0.0	200.0	0.0	1125.0	613.0	3
718	122.6	183.9	0.0	203.5	0.0	958.2	800.1	7
175	362.6	189.0	0.0	164.9	11.6	944.7	755.8	91
828	522.0	0.0	0.0	146.0	0.0	896.0	896.0	28
713	157.0	236.0	0.0	192.0	0.0	935.4	781.2	3
...	...	...	...	...	...	...	...	...
673	212.0	141.3	0.0	203.5	0.0	973.4	750.0	7
595	186.2	124.1	0.0	185.7	0.0	1083.4	764.3	28
445	165.0	0.0	143.6	163.8	0.0	1005.6	900.9	56
117	313.3	262.2	0.0	175.5	8.6	1046.9	611.8	28
464	167.0	75.4	167.0	164.0	7.9	1007.3	770.1	100

824 rows × 8 columns

In [19]:
```
train_target
```

Out[19]:
```
747    26.06
718    10.35
175    79.30
828    74.99
713     9.69
       ...
673    15.03
595    17.60
445    36.56
117    59.80
464    56.81
Name: Strength, Length: 824, dtype: float64
```

Regression Example

predictors - DataFrame

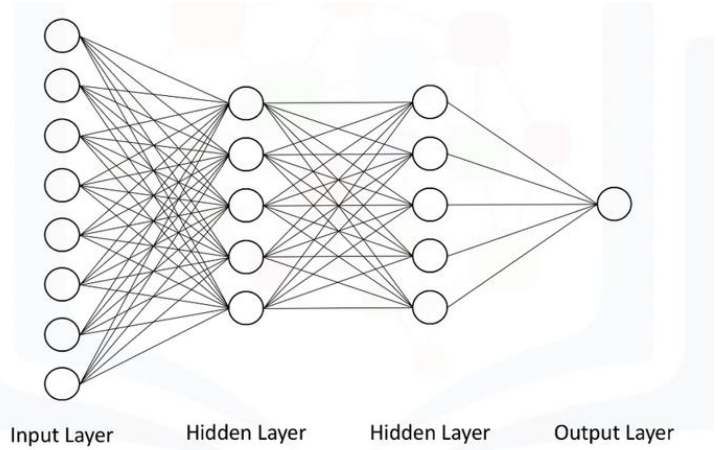
Index	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
0	540	0	0	162	2.5	1040	676	28
1	540	0	0	162	2.5	1055	676	28
2	332.5	142.5	0	228	0	932	594	270
3	332.5	142.5	0	228	0	932	594	365
4	198.6	132.4	0	192	0	978.4	825.5	360
5	266	114	0	228	0	932	670	90

In [9]:
```
#data normalization
train_predictors = (train_predictors - train_predictors.mean()) / train_predictors.std()
test_predictors = (test_predictors - test_predictors.mean()) / test_predictors.std()
```

predictors_norm - DataFrame

Index	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
0	2.47671	-0.856472	-0.846733	-0.916319	-0.620147	0.862735	-1.21708	-0.279597
1	2.47671	-0.856472	-0.846733	-0.916319	-0.620147	1.05565	-1.21708	-0.279597
2	0.491187	0.79514	-0.846733	2.1744	-1.03864	-0.526262	-2.23983	3.55134
3	0.491187	0.79514	-0.846733	2.1744	-1.03864	-0.526262	-2.23983	5.05522
4	-0.790075	0.678079	-0.846733	0.488555	-1.03864	0.0704925	0.647569	4.97607
5	-0.145138	0.464818	-0.846733	2.1744	-1.03864	-0.526262	-1.29191	0.701883

Regression Example



1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

2. Define the neural network architecture

In [10]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

In [11]:
```
model = keras.Sequential()
n = train_predictors.shape[1]
model.add(layers.Dense(5, activation='sigmoid', input_shape = (n,)  ))
model.add(layers.Dense(5, activation='sigmoid'))
model.add(layers.Dense(1) )
```

Regression Example

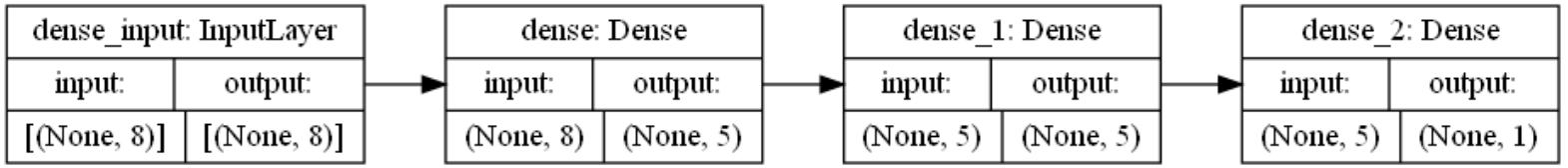
3. Compile the neural net

In [22]:
```
model.compile(optimizer='adam', loss='mse')
```

In [26]:
```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['accuracy', 'mse']
)
```

In [27]:
```
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```

Out[13]:



1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Regression Example

4. Fit / train the neural net

```
In [14]: model.fit(train_predictors, train_target, epochs = 100)
26/26 [==============================] - 0s 399us/step - loss: 708.2107
Epoch 93/100
26/26 [==============================] - 0s 519us/step - loss: 744.7943
Epoch 94/100
26/26 [==============================] - 0s 479us/step - loss: 670.8252
Epoch 95/100
26/26 [==============================] - 0s 439us/step - loss: 681.2140
Epoch 96/100
26/26 [==============================] - 0s 439us/step - loss: 658.1641
Epoch 97/100
26/26 [==============================] - 0s 439us/step - loss: 693.0096
Epoch 98/100
26/26 [==============================] - 0s 399us/step - loss: 734.2052
Epoch 99/100
26/26 [==============================] - 0s 399us/step - loss: 696.3537
Epoch 100/100
26/26 [==============================] - 0s 439us/step - loss: 679.9786
Out[14]: <tensorflow.python.keras.callbacks.History at 0x210fc4a4070>
```

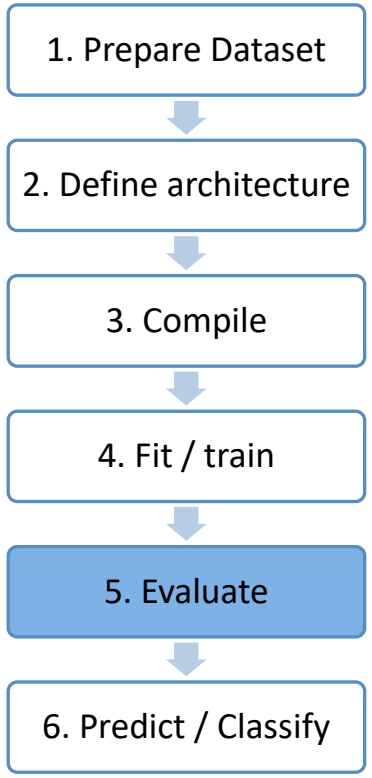
1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Regression Example

5. Evaluate the neural net

In [32]:
```
result = model.evaluate(test_predictors, test_target, verbose=0)
dict(zip(model.metrics_names, result))
```

Out[32]:
```
{'loss': 133.75021362304688, 'accuracy': 0.0, 'mse': 133.75021362304688}
```



Regression Example

Using tanh activation function

In [33]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

In [35]:
```
model = keras.Sequential()
n = train_predictors.shape[1]
model.add(layers.Dense(5, activation='tanh', input_shape = (n,)  ))
model.add(layers.Dense(5, activation='tanh'))
model.add(layers.Dense(1) )

model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['accuracy', 'mse']
)

model.fit(train_predictors, train_target, epochs = 100)
```

In [36]:
```
result = model.evaluate(test_predictors, test_target, verbose=0)
dict(zip(model.metrics_names, result))
```

Out[36]:
```
{'loss': 658.7315063476562, 'accuracy': 0.0, 'mse': 658.7315063476562}
```

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

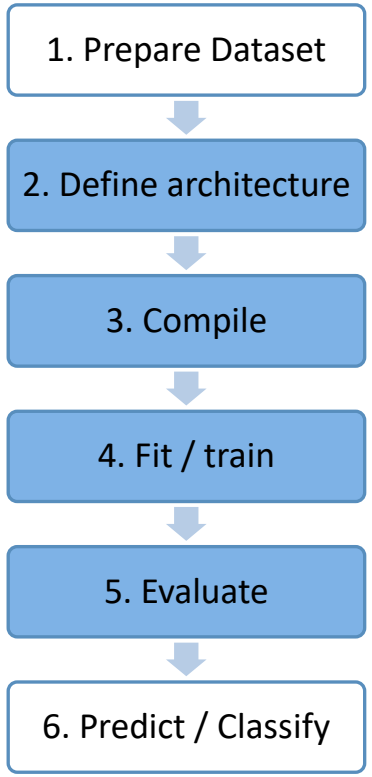
Regression Example

Using tanh activation function and more neurons

In [38]:
```
model = keras.Sequential()
n = train_predictors.shape[1]
model.add(layers.Dense(50, activation='tanh', input_shape = (n,)  ))
model.add(layers.Dense(50, activation='tanh'))
model.add(layers.Dense(1) )

model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['accuracy', 'mse']
)

model.fit(train_predictors, train_target, epochs = 100)
```

In [39]:
```
result = model.evaluate(test_predictors, test_target, verbose=0)
dict(zip(model.metrics_names, result))
```

Out[39]:
```
{'loss': 46.75392150878906, 'accuracy': 0.0, 'mse': 46.75392150878906}
```



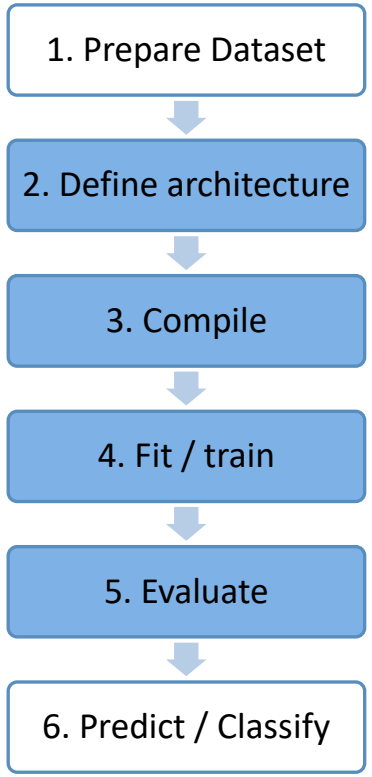
Regression Example

Using relu activation function and more neurons

In [40]:
```
model = keras.Sequential()
n = train_predictors.shape[1]
model.add(layers.Dense(50, activation='relu', input_shape = (n,)  ))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dense(1) )

model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['accuracy', 'mse']
)

model.fit(train_predictors, train_target, epochs = 100)
```

In [41]:
```
result = model.evaluate(test_predictors, test_target, verbose=0)
dict(zip(model.metrics_names, result))
```

Out[41]:
```
{'loss': 40.989158630371094, 'accuracy': 0.0, 'mse': 40.989158630371094}
```



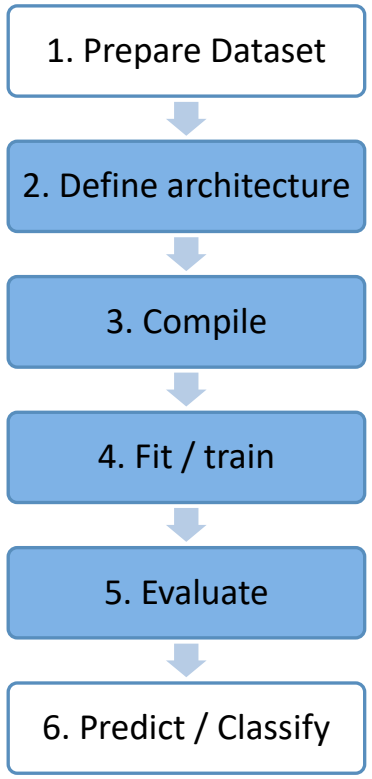
Regression Example

Using relu activation function and more epochs

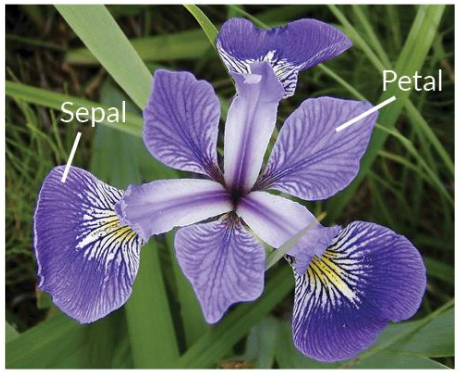
In [42]:
```
model = keras.Sequential()
n = train_predictors.shape[1]
model.add(layers.Dense(50, activation='relu', input_shape = (n,)  ))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dense(1) )

model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['accuracy', 'mse']
)

model.fit(train_predictors, train_target, epochs = 500)
```

In [43]:
```
result = model.evaluate(test_predictors, test_target, verbose=0)
dict(zip(model.metrics_names, result))
```

Out[43]:
```
{'loss': 25.419971466064453, 'accuracy': 0.0, 'mse': 25.419971466064453}
```



Classification Example



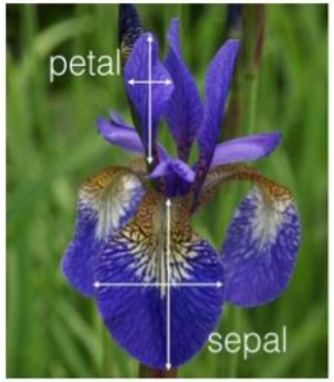
Iris Versicolor

Iris Setosa

Iris Virginica

- Attribute Information (All in centimeters)
  - Sepal length
  - Sepal width
  - Petal length
  - Petal width
  - Flower class

Ex: 5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica



$$\text{decision} = \begin{cases} \text{setosa} = 0 \\ \text{versicolor} = 1 \\ \text{virginica} = 2 \end{cases}$$

Classification Example

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
7	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor
6.9	3.1	4.9	1.5	versicolor
5.5	2.3	4	1.3	versicolor
6.5	2.8	4.6	1.5	versicolor
6.3	3.3	6	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3	5.9	2.1	virginica
6.3	2.9	5.6	1.8	virginica
6.5	3	5.8	2.2	virginica

iris_data_original.csv

decision = { setosa = 0, versicolor = 1, virginica = 2

Classification Example

1. Prepare & Explore Dataset

In [5]:
```
import numpy as np
import pandas as pd
```

In [7]:
```
#read the dataset
iris_data = pd.read_csv('iris_data.csv')
```

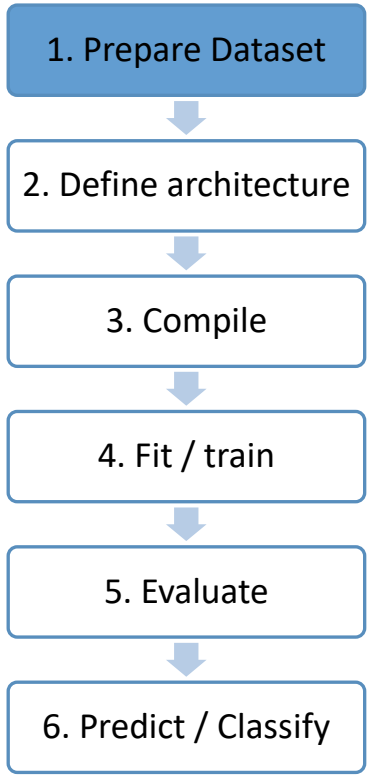
In [11]:
```
iris_data.head()
```

Out[11]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

In [12]:
```
iris_data.shape
```

Out[12]: (150, 5)



Classification Example

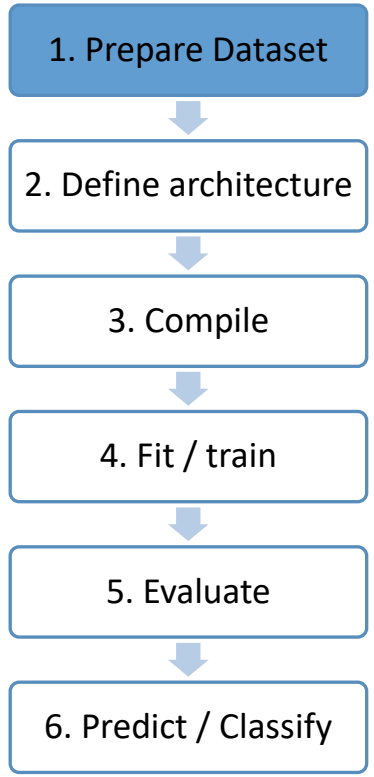
In [13]:
```
#split the dataset into train and test
train_data = iris_data.sample(frac=0.8, random_state=0)
test_data = iris_data.drop(train_data.index)
```

In [14]:
```
train_data
```

Out[14]:

	sepal_length	sepal_width	petal_length	petal_width	species
114	5.8	2.8	5.1	2.4	2
62	6.0	2.2	4.0	1.0	1
33	5.5	4.2	1.4	0.2	0
107	7.3	2.9	6.3	1.8	2
7	5.0	3.4	1.5	0.2	0
...	...	...	...	...	...
57	4.9	2.4	3.3	1.0	1
131	7.9	3.8	6.4	2.0	2
65	6.7	3.1	4.4	1.4	1
32	5.2	4.1	1.5	0.1	0
138	6.0	3.0	4.8	1.8	2

120 rows × 5 columns



Classification Example

Predictors and Target

sepal_length	sepal_width	petal_length	petal_width	species
5.8	2.8	5.1	2.4	2
6	2.2	4	1	1
5.5	4.2	1.4	0.2	0
7.3	2.9	6.3	1.8	2
5	3.4	1.5	0.2	0
6.3	3.3	6	2.5	2
5	3.5	1.3	0.3	0
6.7	3.1	4.7	1.5	1
6.8	2.8	4.8	1.4	1
6.1	2.8	4	1.3	1
6.1	2.6	5.6	1.4	2
6.4	3.2	4.5	1.5	1

predictors

target

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example

```
In [17]:
#split train and test data into predictors and target
train_predictors = train_data[train_data.columns [train_data.columns  != 'species']]
train_target = train_data['species']

test_predictors = test_data[test_data.columns [test_data.columns  != 'species']]
test_target = test_data['species']
```

```
In [18]:
train_predictors
```

Out[18]:

	sepal_length	sepal_width	petal_length	petal_width
114	5.8	2.8	5.1	2.4
62	6.0	2.2	4.0	1.0
33	5.5	4.2	1.4	0.2
107	7.3	2.9	6.3	1.8
7	5.0	3.4	1.5	0.2
...	...	...	...	..
57	4.9	2.4	3.3	1.0
131	7.9	3.8	6.4	2.0
65	6.7	3.1	4.4	1.4
32	5.2	4.1	1.5	0.1
138	6.0	3.0	4.8	1.8

120 rows × 4 columns

```
In [19]:
train_target
```

```
Out[19]:
114    2
62     1
33     0
107    2
7      0
      ..
57     1
131    2
65     1
32     0
138    2
Name: species, Length: 120, dtype: int64
```

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example

Target Variable

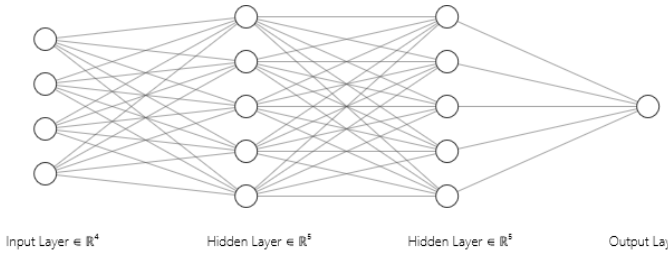
species
2
1
0
2
0
2
0
1
1
1
2
1

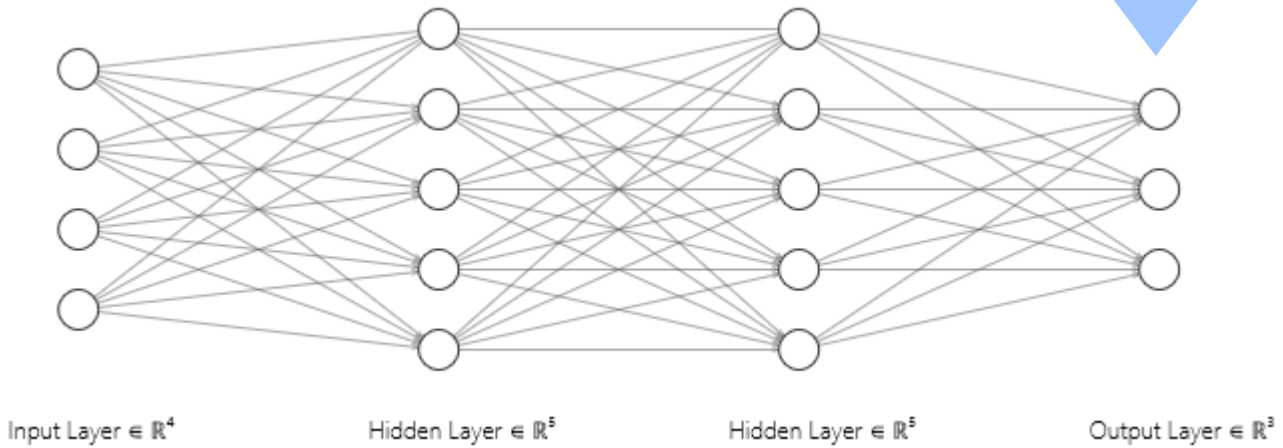
to_categorical

0	1	2
0	0	1
0	1	0
1	0	0
0	0	1
1	0	0
0	0	1
1	0	0
0	1	0
0	1	0
0	1	0
0	0	1
0	1	0

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example





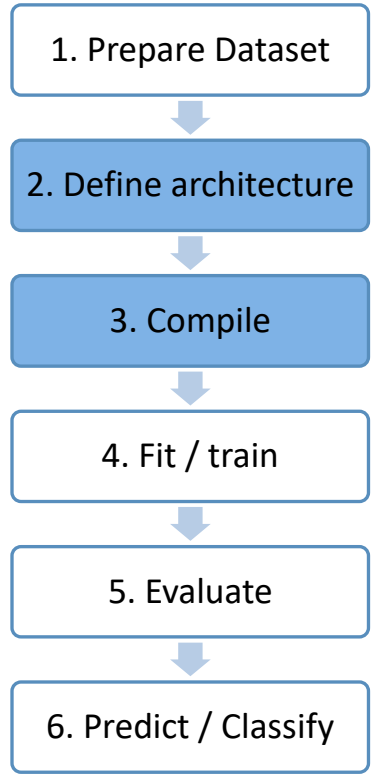
1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example

2. Define the neural network architecture

In [23]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import to_categorical
```

In [24]:
```
train_target = to_categorical(train_target)
test_target = to_categorical(test_target)

n = train_predictors.shape[1]

model = keras.Sequential()
model.add(layers.Dense(5, activation='relu', input_shape = (n,)  ))
model.add(layers.Dense(5, activation='relu'))
model.add(layers.Dense(3, activation='softmax'))
```

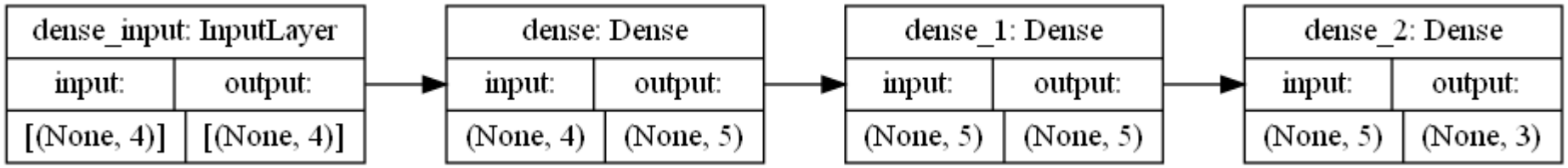


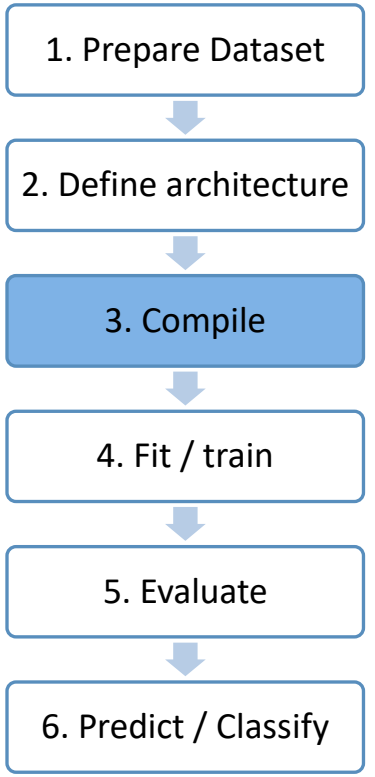
Classification Example

3. Compile the neural net

In [27]:
```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy', 'mse']
)
```

In [28]:
```
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```





Classification Example

4. Fit / train the neural net

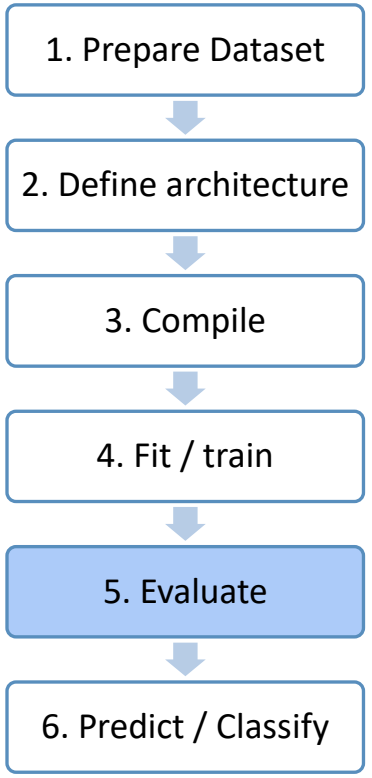
```
In [33]: model.fit(train_predictors, train_target, epochs = 500)
4/4 [==============================] - 0s 665us/step - loss: 0.3144 - accuracy: 0.9833 - mse: 0.0526
Epoch 493/500
4/4 [==============================] - 0s 998us/step - loss: 0.3133 - accuracy: 0.9833 - mse: 0.0523
Epoch 494/500
4/4 [==============================] - 0s 665us/step - loss: 0.3124 - accuracy: 0.9833 - mse: 0.0522
Epoch 495/500
4/4 [==============================] - 0s 665us/step - loss: 0.3115 - accuracy: 0.9833 - mse: 0.0520
Epoch 496/500
4/4 [==============================] - 0s 665us/step - loss: 0.3105 - accuracy: 0.9833 - mse: 0.0518
Epoch 497/500
4/4 [==============================] - 0s 665us/step - loss: 0.3097 - accuracy: 0.9833 - mse: 0.0516
Epoch 498/500
4/4 [==============================] - 0s 665us/step - loss: 0.3089 - accuracy: 0.9833 - mse: 0.0513
Epoch 499/500
4/4 [==============================] - 0s 665us/step - loss: 0.3081 - accuracy: 0.9833 - mse: 0.0511
Epoch 500/500
4/4 [==============================] - 0s 665us/step - loss: 0.3072 - accuracy: 0.9833 - mse: 0.0509
Out[33]: <tensorflow.python.keras.callbacks.History at 0x1bad8878850>
```

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example

5. Evaluate the neural net

```
In [34]: result = model.evaluate(test_predictors, test_target, verbose=0)
         dict(zip(model.metrics_names, result))

Out[34]: {'loss': 0.32942456007003784,
          'accuracy': 0.9333333373069763,
          'mse': 0.05889347568154335}
```



Classification Example

6. Make predictions / classifications for unseen data

In [35]:
```
#not yet until we enhanced the results
predictions = model.predict(test_predictors)
predictions
```

Out[35]:
```
array([[9.55777764e-01, 4.42221612e-02, 1.50082755e-07],
       [9.92598891e-01, 7.40105845e-03, 2.36261066e-10],
       [9.81547713e-01, 1.84522867e-02, 6.97093538e-09],
       [9.76115048e-01, 2.38848701e-02, 1.71564274e-08],
       [9.39242959e-01, 6.07565977e-02, 4.66694530e-07],
       [9.49310422e-01, 5.06893098e-02, 2.43942111e-07],
       [9.81958807e-01, 1.80411879e-02, 6.44480558e-09],
       [9.70251381e-01, 2.97486130e-02, 3.70304392e-08],
       [9.57280278e-01, 4.27195579e-02, 1.32757407e-07],
       [9.67882514e-01, 3.21174525e-02, 4.84762452e-08],
       [1.90118507e-01, 7.05148757e-01, 1.04732804e-01],
```

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

Classification Example

target_test - NumPy array

	0	1	2
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0
7	1	0	0
8	1	0	0
9	1	0	0
10	0	1	0
11	0	1	0
12	0	1	0
13	0	1	0
14	0	1	0
15	0	1	0
16	0	1	0

Format Resize Background color

predictions - NumPy array

	0	1	2
0	0.998895	0.0011048	1.33821e-08
1	0.999859	0.000140683	1.72893e-10
2	0.998583	0.0014172	4.93178e-09
3	0.997961	0.00203906	7.99126e-09
4	0.997987	0.00201265	2.53669e-08
5	0.997227	0.00277296	3.57516e-08
6	0.999677	0.000323148	1.1178e-09
7	0.998945	0.00105514	6.54986e-09
8	0.997907	0.00209259	3.06606e-08
9	0.998985	0.00101535	8.18075e-09
10	0.00165205	0.990878	0.00746996
11	0.00280989	0.994361	0.00282896
12	0.00283221	0.342945	0.654223
13	0.00196613	0.509868	0.488166
14	0.00209021	0.754434	0.243476
15	0.0133355	0.985931	0.000733132
16	0.00441847	0.992631	0.00295094

Format Resize Background color

1. Prepare Dataset
2. Define architecture
3. Compile
4. Fit / train
5. Evaluate
6. Predict / Classify

End Notes

- The sigmoid and hyperbolic tangent functions are avoided in many applications nowadays due to the vanishing gradient problem.
- The ReLU function is a general activation function and is used in most cases these days.
- Note that ReLU function should only be used in hidden layers.
- Generally, you can begin with using the ReLU activation function and then switch to other activation functions in case ReLU doesn't yield optimum results.

My Notes

Recap!

- ❑ The need for neural networks
- ❑ Neural nets architecture
- ❑ Neurons
- ❑ Layers
- ❑ Activation functions.
- ❑ Learning / Training / Fit
- ❑ Regression : Concrete dataset
- ❑ Classification : Irish dataset

@SalhaAlzahrani