Artificial Neural Networks
and
Deep Learning

Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

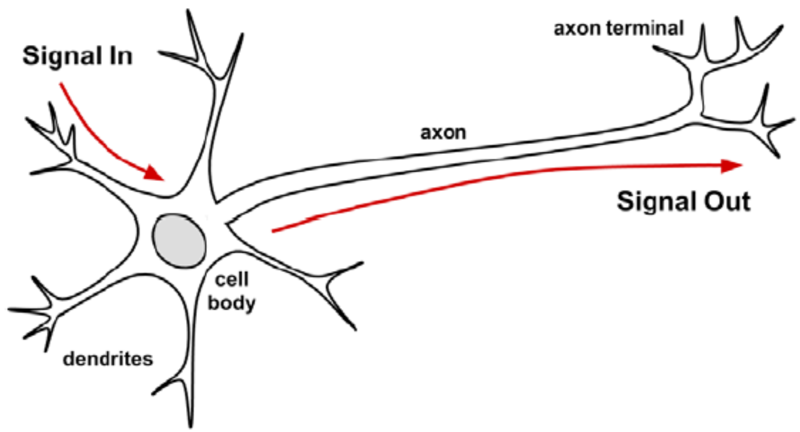
Symbolic Learning vs. Neuronal Learning

- Symbolic Learning
  - induction of Rules and Decision Trees
  - works with discrete combinations of attribute values
  - uses logical / relational operators (=,>, <)

- Neuronal Learning
  - works by adjusting non-linear and continuous weights of its inputs
  - uses numeric operators (×, +)
  - makes a search in a finer space of granularity than the algorithms of induction of rules

Artificial Neural Network

- Inspired in the human brain consist of a huge number of neurons with extremely high inter-connectivity

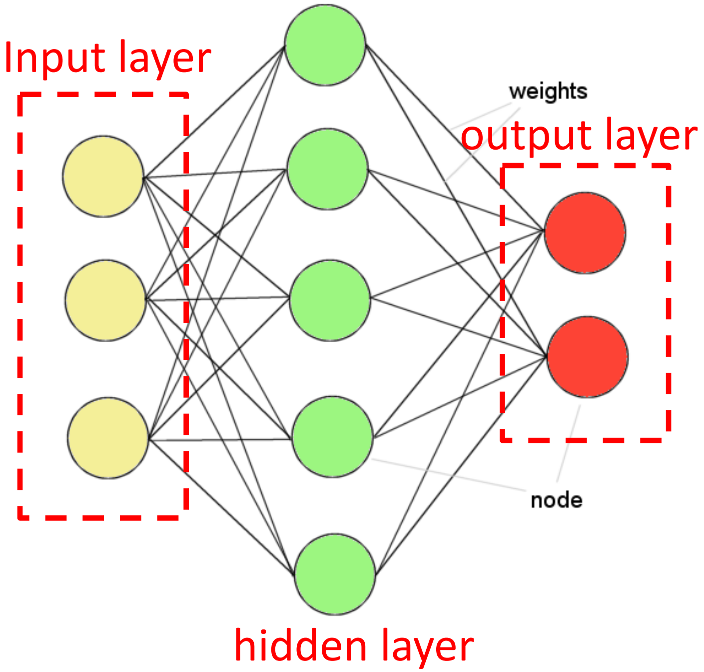


- ANNs incorporate the two fundamental components of biological neural nets:
  1. Neurones (nodes)
  2. Synapses (weights)

Neural Network – The basics

A Neural Network consists of:

- Neurons (processing elements)
- Inter-connections between neurons with numerical weights

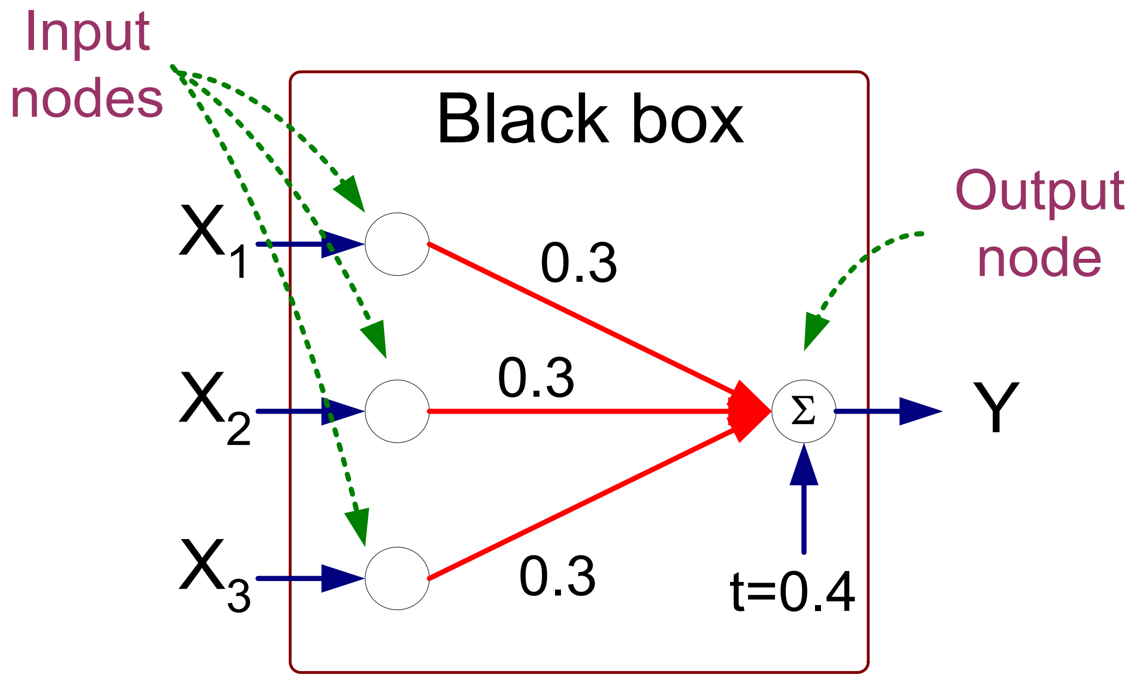


Learning process

- Consists in adjusting the intensity of the connections between the neurons (synapses) represented by weights, during the training process of the network

Neural Networks – The basics

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0

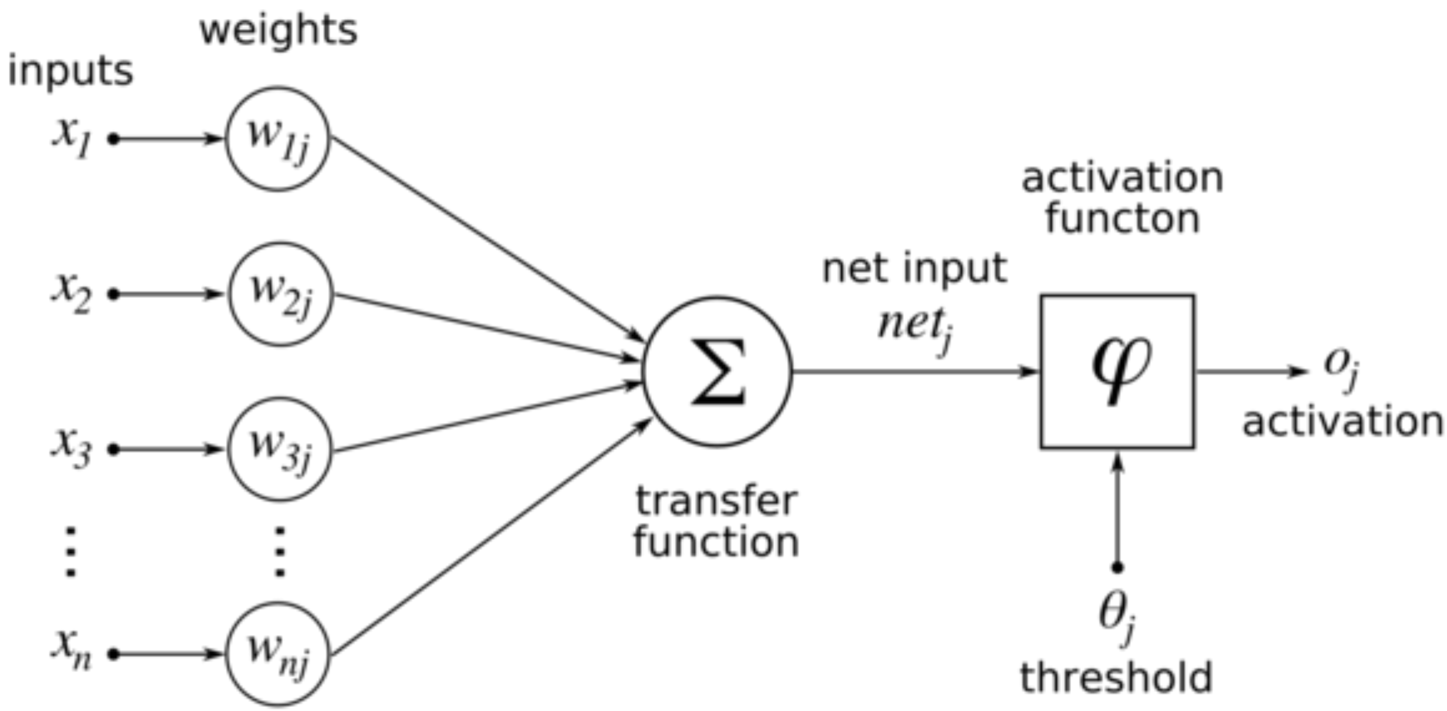


A neural network maps the input variables to an output variable y

- The input variables corresponds to the attributes or characteristics of our problem
- The output variable can be discrete (classification) or continuous (regression)

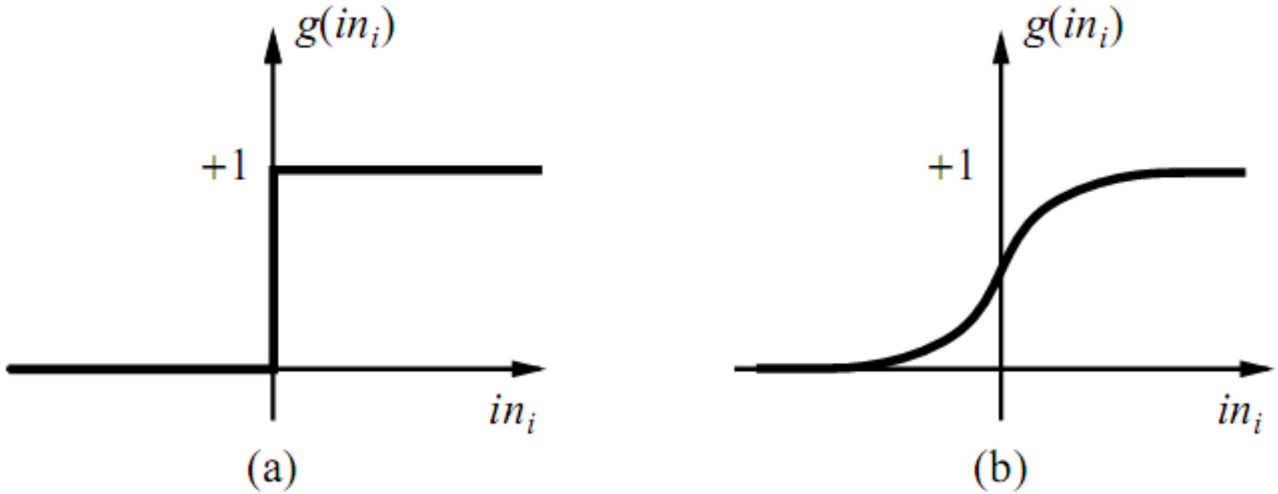
Mathematical Model of Artificial Neuron

Each node in ANN do the sum of products of its inputs(**X**) and their corresponding weights(**W**), adds a bias and apply an Activation function g(x) to get its output and feed it as an input to the next neuron



Activation function main purpose is to convert an input signal of a node to an output signal

Most popular types of Activation Functions



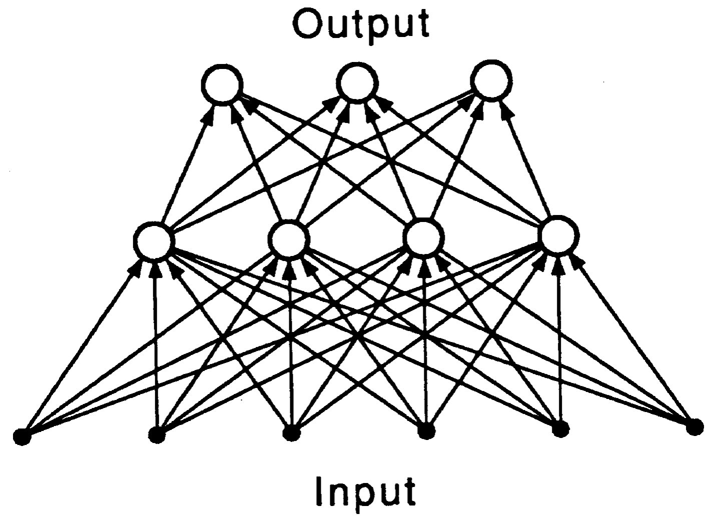
- (a) is a step function or threshold function, sign function
  (b) is a sigmoid function $1/(1+e^{-x})$

- Changing the bias weight $W_{0,i}$ moves the threshold location
- Different functions give different models
- Using a nonlinear function which approximates a linear threshold allows a network to approximate nonlinear functions

Topologies of Neural Networks

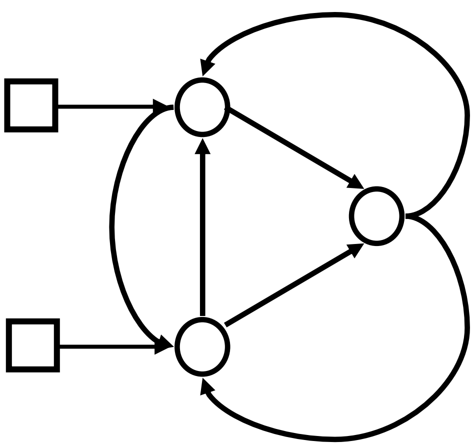
Feed-forward neural networks are the most common models

- These are directed acyclic graphs
  - Single-Layer Feed-Forward (Perceptron)
  - Multi-Layer Feed-Forward



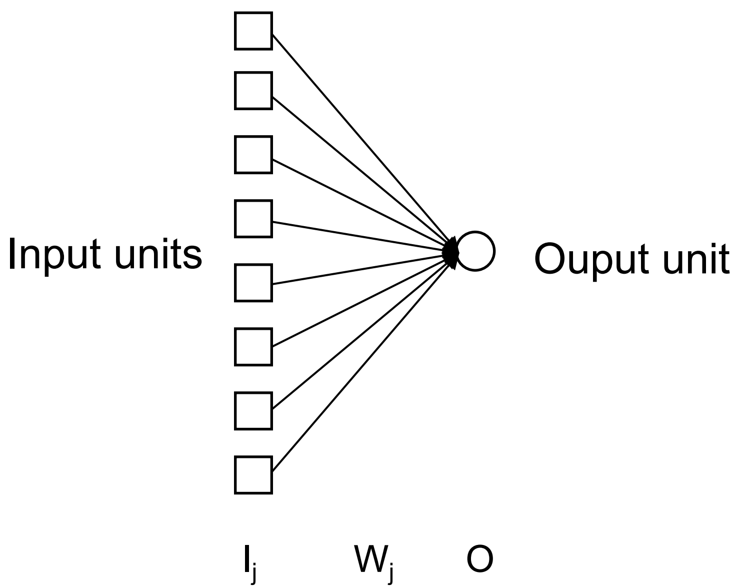
Recurrent networks have at least one feedback connection

- They have directed cycles
- The response to an input depends on the initial state which may depend on previous inputs

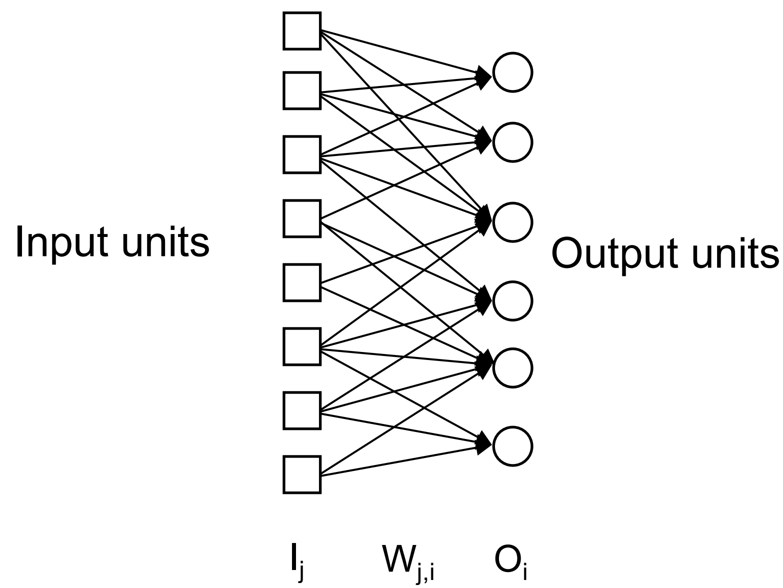


Perceptron

- Perceptron is a network of one input layer of neurons that feed forward to one output layer of neurons - there is no intermediate level, only the level of entry and exit



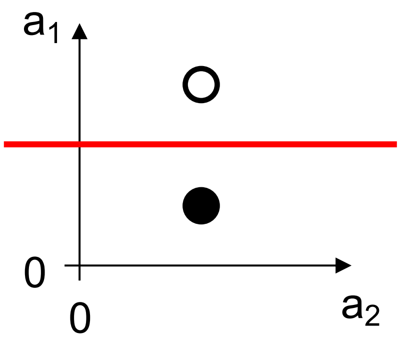
Single Perceptron



Perceptron Network

Expressiveness of Perceptron

- Perceptron can represent only linearly separable functions (i.e. functions for which such a separation hyperplane exists)

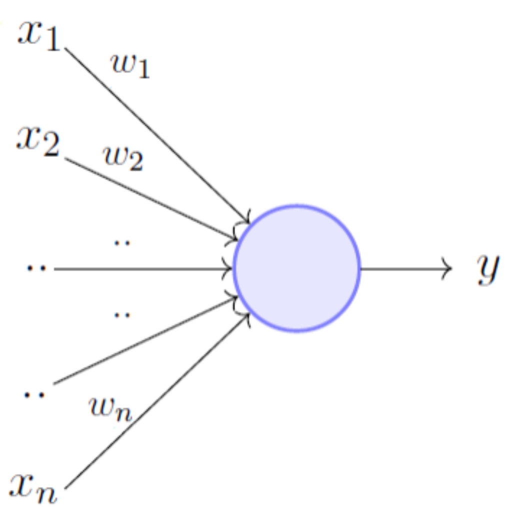


- Such perceptron have limited expressivity
- There exists an algorithm that can fit a threshold perceptron to any linearly separable training set

Perceptron Learning Algorithm

– Initialize the weights and threshold to small random numbers
– Present a vector **x** to the neuron inputs and calculate the output
– Update the weights according to the error

- Applied learning function: $W_j(t+1) = W_j(t) + \alpha \times (y - g_W(x)) \times x_j$



$$
\begin{aligned}
y &= 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0 \\
&= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0
\end{aligned}
$$

Perceptron Learning

- The squared error for an example with input x and desired output y is:

$$E = \frac{1}{2} Err^2 = \frac{1}{2}(y - g_W(x))^2$$

- Perform optimization search by **gradient descent**:

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}(y - g(\Sigma_{j=0}^{n} W_j x_j)) = -Err \times g'(in) \times x_j$$

- Simple weight update rule: $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$

- Positive error $\Rightarrow$ increase network output
  - increase weights on positive inputs,
  - decrease on negative inputs

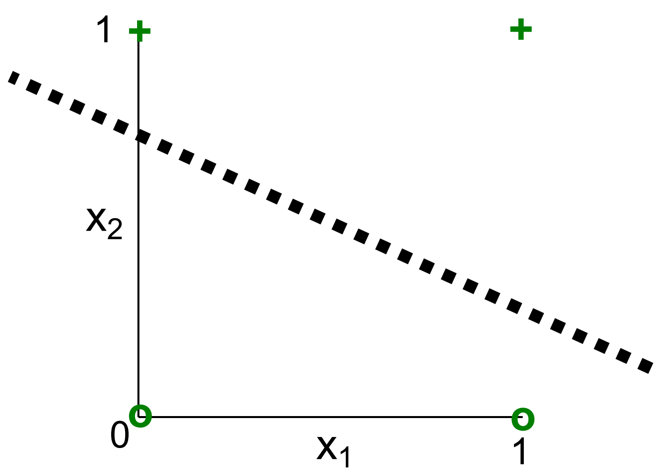
Perceptron Learning

- The weight updates need to be applied repeatedly for each weight W_j in the network, and for each training suite in the training set
- One such cycle through all weighty is called an **epoch** of training
- Eventually, mostly after **many epochs**, the weight changes converge towards zero and the training process terminates

- The **perceptron learning process always finds** a set of weights for a perceptron that solves a problem correctly with **a finite number of epochs, if such a set of weights exists**

Perceptron Learning Example

- Data: (0,0)→0, (1,0)→0, (0,1)→1, (1,1)→1



- Initialization:
  - $W_1(0) = 0.92$,
  - $W_2(0) = 0.62$,
  - $W_0(0) = 0.22$,
  - $\alpha = 0.1$

- Training – epoch 1:

out1 = sign(0.92*0 + 0.62*0 – 0.22) = sign(-0.22) = 0 ✓
out2 = sign(0.92*1 + 0.62*0 – 0.22) = sign(0.7) = 1 X

$W_1(1) = 0.92 + 0.1 * (0 - 1) * 1 = 0.82$
$W_2(1) = 0.62 + 0.1 * (0 - 1) * 0 = 0.62$
$W_0(1) = 0.22 + 0.1 * (0 - 1) * (-1) = 0.32$

out3 = sign(0.82*0 + 0.62*1 – 0.32) = sign(0.5) = 1 ✓
out4 = sign(0.82*1 + 0.62*1 – 0.32) = 1 ✓

Perceptron Learning Example

- Training – epoch 2:

out1 = sign(0.82*0 + 0.62*0 – 0.32) = sign(0.32) = 0 ✓
out2 = sign(0.82*1 + 0.62*0 – 0.32) = sign(0.5) = 1 X

$W_1(2) = 0.82 + 0.1 * (0 – 1) * 1 = 0.72$
$W_2(2) = 0.62 + 0.1 * (0 – 1) * 0 = 0.62$
$W_0(2) = 0.32 + 0.1 * (0 – 1) * (-1)= 0.42$

out3 = sign(0.72*0 + 0.62*1 – 0.42) = sign(0.3) = 1 ✓
out4 = sign(0.72*1 + 0.62*1 – 0.42) = 1 ✓

- Training – epoch 3:

out1 = sign(0.72*0 + 0.62*0 – 0.42) = 0 ✓
out2 = sign(0.72*1 + 0.62*0 – 0.42) = 1 X

$W_1(3) = 0.72 + 0.1 * (0 – 1) * 1 = 0.62$
$W_2(3) = 0.62 + 0.1 * (0 – 1) * 0 = 0.62$
$W_0(3) = 0.42 + 0.1 * (0 – 1)*(-1) = 0.52$

out3 = sign(0.62*0 + 0.62*1 – 0.52) = 1 ✓
out4 = sign(0.62*1 + 0.62*1 – 0.52) = 1 ✓

Perceptron Learning Example

- Training – epoch 4:

  out1 = sign(0.62*0 + 0.62*0 – 0.52) = 0 ✓

  out2 = sign(0.62*1 + 0.62*0 – 0.52) = 1 X

  $W_1(4) = 0.62 + 0.1 * (0 - 1) * 1 = 0.52$
  $W_2(4) = 0.62 + 0.1 * (0 - 1) * 0 = 0.62$
  $W_0(4) = 0.52 + 0.1 * (0 - 1) * (-1) = 0.62$

  out3 = sign(0.52*0 + 0.62*1 – 0.62) = 0 X

  $W_1(4) = 0.52 + 0.1 * (1 - 0) * 0 = 0.52$
  $W_2(4) = 0.62 + 0.1 * (1 - 0) * 1 = 0.72$
  $W_0(4) = 0.62 + 0.1 * (1 - 0) * (-1) = 0.52$

  out4 = sign(0.52*1 + 0.72*1 – 0.52) = 1 ✓

  .....

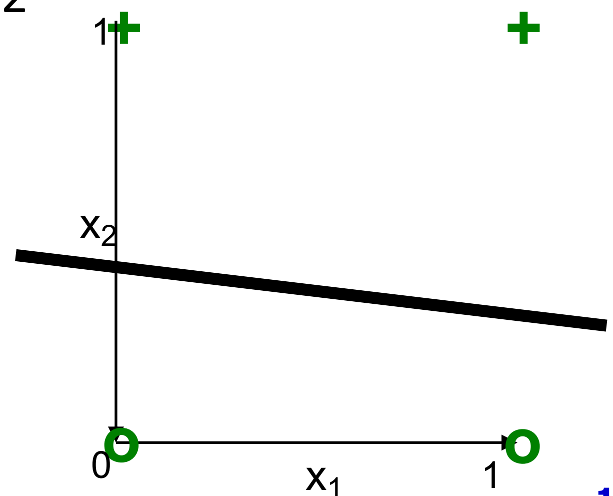
- Finally:

  out1 = sign(0.12*0 + 0.82*0 – 0.42) = 0 ✓

  out2 = sign(0.12*1 + 0.82*0 – 0.42) = 0 ✓

  out3 = sign(0.12*0 + 0.82*1 – 0.42) = 1 ✓

  out4 = sign(0.12*1 + 0.82*1 – 0.42) = 1 ✓



Example: Finding Weights for AND Operation

There are two input weights W_1 and W_2 and a threshold W_0. For each training pattern the perceptron needs to satisfy the following equation:

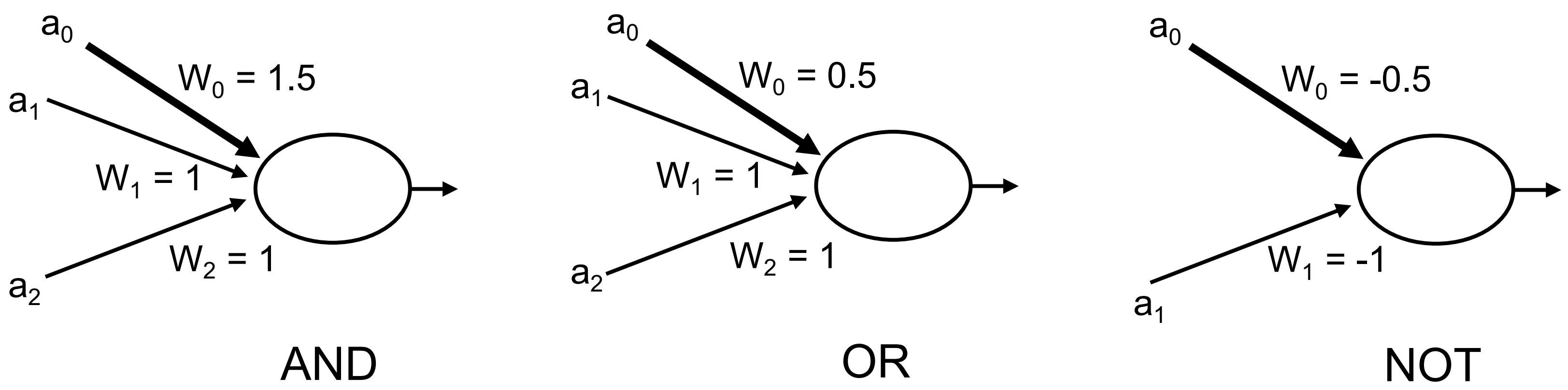
$$out = g(W_1*a_1 + W_2*a_2 - W_0) = sign(W_1*a_1 + W_2*a_2 - W_0)$$

For a binary AND there are four training data items available that lead to four inequalities:

- $W_1*0 + W_2*0 - W_0 < 0 \quad \Rightarrow W_0 > 0$
- $W_1*0 + W_2*1 - W_0 < 0 \quad \Rightarrow W_2 < W_0$
- $W_1*1 + W_2*0 - W_0 < 0 \quad \Rightarrow W_1 < W_0$
- $W_1*1 + W_2*1 - W_0 \geq 0 \quad \Rightarrow W_1 + W_2 \geq W_0$

There is obvious an **infinite number of solutions** that realize a **logical AND**; e.g. $W_1 = 1$, $W_2 = 1$ and $W_0 = 1.5$

Logical Functions



- These Boolean functions that can be implemented with an artificial neuron

Limitations of Simple Perceptrons

XOR

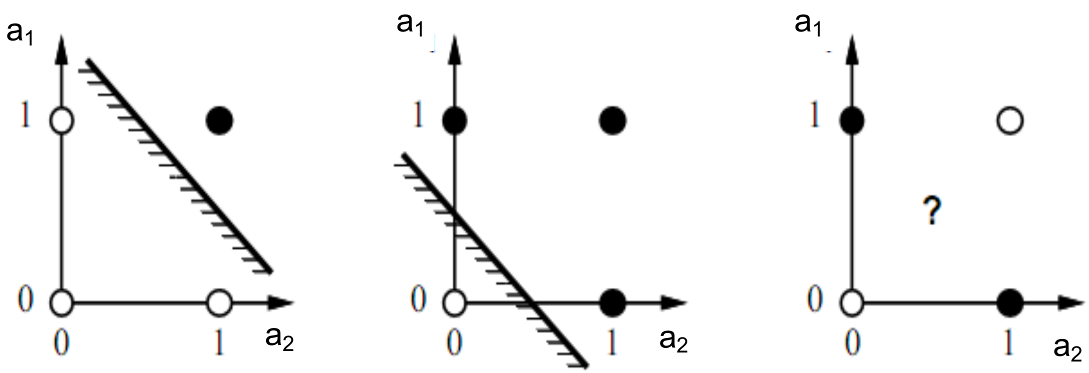
- $W_1{*}0 + W_2{*}0 - W_0 < 0 \quad \Rightarrow W_0 > 0$
- $W_1{*}0 + W_2{*}1 - W_0 \geq 0 \quad \Rightarrow W_2 \geq W_0$
- $W_1{*}1 + W_2{*}0 - W_0 \geq 0 \quad \Rightarrow W_1 \geq W_0$
- $W_1{*}1 + W_2{*}1 - W_0 < 0 \quad \Rightarrow W_1 + W_2 < W_0$

- The 2nd and 3rd inequalities are not compatible with inequality 4, and there is no solution to the XOR problem
- XOR requires two separation hyperplanes!
- There is thus a need for more complex networks that combine simple perceptrons to address more sophisticated classification tasks

Expressiveness of Perceptrons

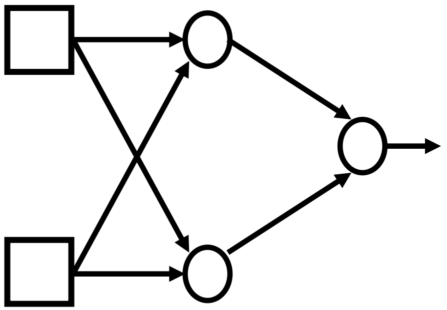
- A perceptron with g = step function can model Boolean functions and linear classification:
  - a perceptron can represent AND, OR, NOT, but **not XOR**
- A perceptron represents a linear separator for the input space

$\sum_j W_j a_j > 0$



(a) a_1 **and** a_2 (b) a_1 **or** a_2 (c) a_1 **xor** a_2

Multi-Layer Feed-Forward

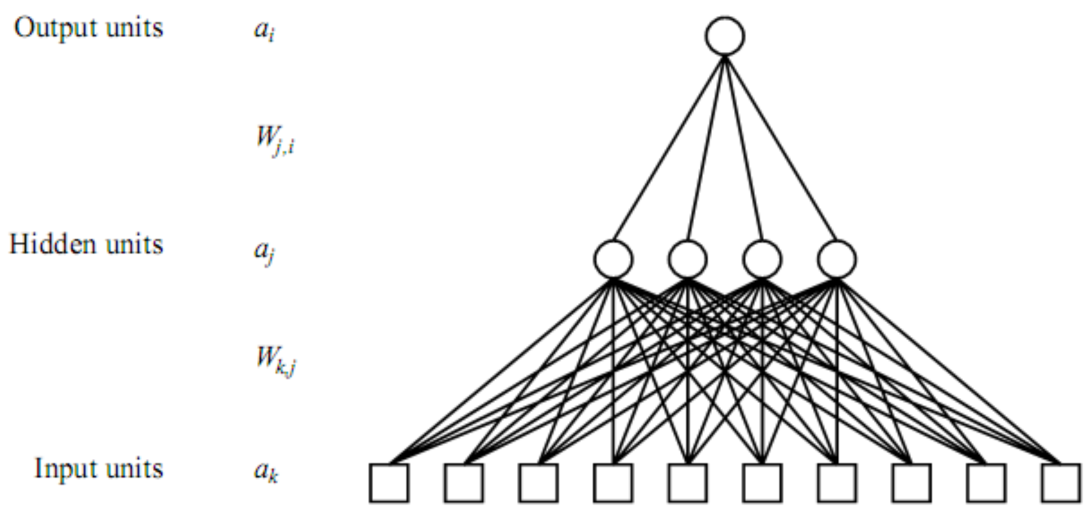


- Multi-Layer Feed-Forward Structures have:
  - one input layer
  - one output layer
  - one or many hidden layers of processing units

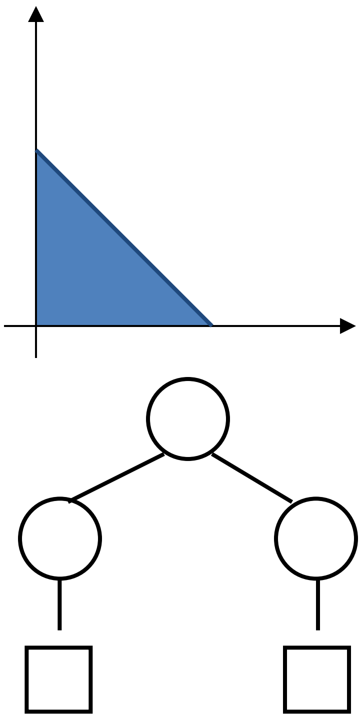
- The hidden layers are between the input and the output layer

Multi-Layer Feed-Forward

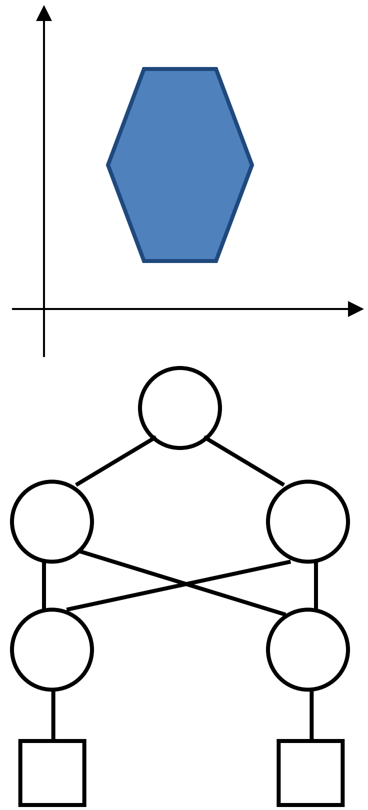
- Multi-Layer Perceptrons (MLP) have fully connected layers
- Hidden layers enlarge the space of hypotheses that the network can represent
- Learning is done by **back-propagation algorithm**
  - ➥ errors are back-propagated from the output layer to the hidden layers



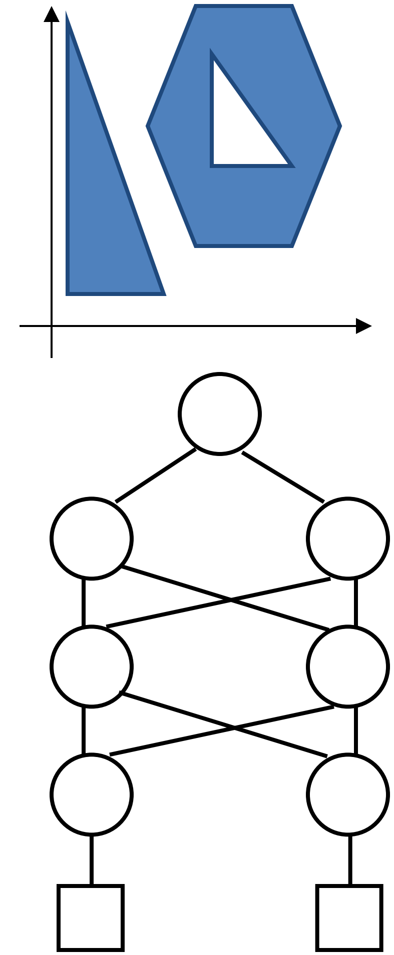
Number of Hidden Layers vs. Expressiveness



One layer draws linear boundaries



Two layer combines the boundaries.



Three or more layers can generate arbitrarily boundaries

R: Neural Networks

```
library(neuralnet)  #The neuralnet algorithm only works for numeric datasets
```

```
m <- neuralnet(AtribObj ~ Predictors, data = mydata, hidden = 5)
```

- AtribObj: attribute to predict
- Predictors: attributes to use in the model
- date: dataframe with predictive attributes and attribute to predict
- hidden: is the number of neurons in the inner layer of the network (default = 1)

  Empirical Rule: N. neurons of the inner layer = 2/3*(#inputs + #outputs)

The f. returns a neural net object that can be used to make predictions

Predictions:

```
p <- compute (m, test)
```

- m: is the model generated by the neural net
- test: test set with the same characteristics as the training set

The function returns a list with two components:

– $neurons: contains the neurons at each level of the network

– $net.result: stores the values provided by the template

R: Neural Networks

```
library(nnet)     # The nnet works for numeric/categorical variables
                  #only capable of modeling a single layer network
```

`nnmodel <- nnet(formula, data, ... ,size, rang=0.5, decay=0, maxit=100)`

- `formula: class ~ x1 + x2 + ...`
- `data: train dataset`
- `size: number of neurons in the inner layer (default = 1)`
- `rang: initial random weights on [-rang, rang]. Value about 0.5 unless the inputs are large, in which case it should be chosen so that rang * max(|x|) is about 1.`
- `decay: parameter for weight decay. Default 0.`
- `maxit: maximum number of iterations. Default 100.`

`The f. returns an object nnet to make predictions`

`Predictions:`

`nnpredict <- predict(nnmodel, tst.set, type = "class")`

- `Nnmodel: neural net model generate`
- `tst.set: test set with same features as train set`

Neural Networks

Advantages

- Accuracy of classification is usually high, even for complex problems
- Distributed processing, the knowledge is distributed through the weights of the links
- Robust in handling examples even if they contain errors
- They handle well redundant attributes, since the weights associated with them are usually very small
- Results can be discrete, real values, or a vector of values (discrete or real)

Disadvantages

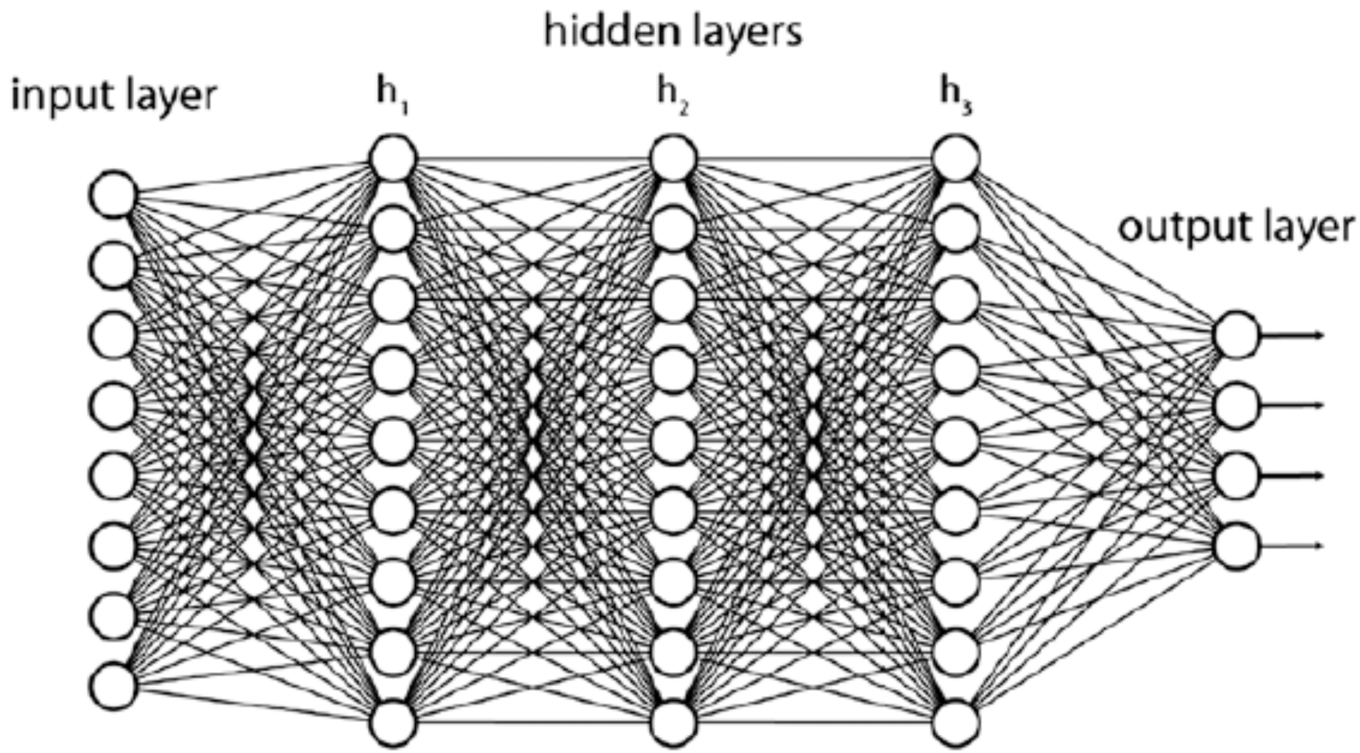
- Difficult to determine optimal network topology for a problem
- Difficult to use - have many parameters to define, require long training time
- Requires specific pre-processing of data
- Difficult to understand the learning function (weights)
- Do not provide a model or explanations of results
- It is not easy to incorporate domain knowledge

Deep Learning

Deep Learning

Deep neural networks are distinguished from ANNs by having many hidden layers

Deep means **many layers**



Deep Learning – APIs

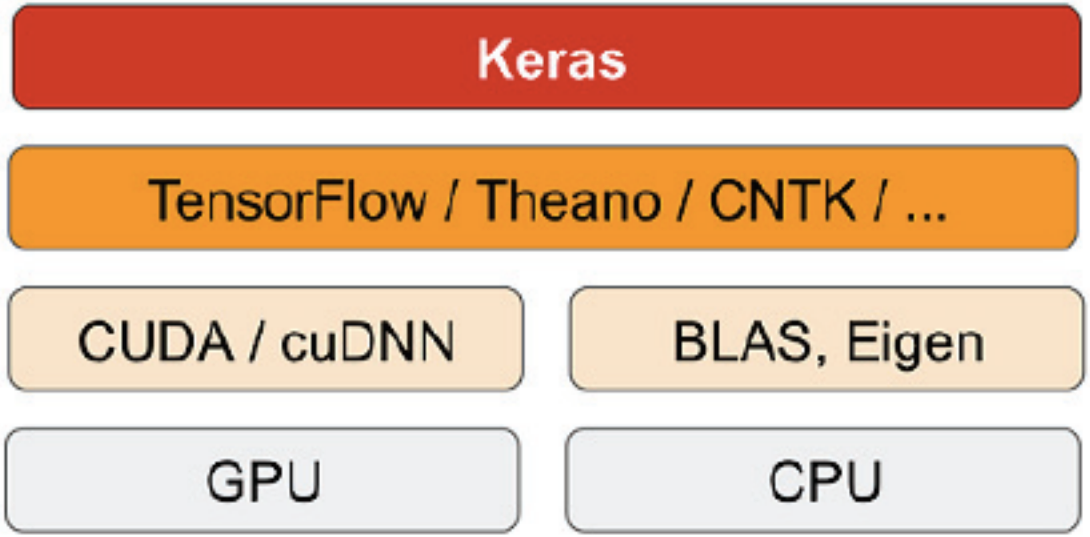
Keras is a high-level **deep learning API** written in Python that can run on top of any of these **three deep learning frameworks**:

- TensorFlow (from Google)
- CNTK (from Microsoft)
- Theano (from the Montreal Institute for Learning Algorithms, Université Montréal, Canada)

Keras facilitates the following key aspects:

- Built-in CNN, RNN, and autoencoder models as well as support classes and methods (metrics, optimizers, regularizers, visualization, and so on) - enables **easy and fast prototyping**
- Excellent modularity and extensibility
- Allow the same code to run seamlessly on CPU and GPU

Keras model-level library



- Any code written with Keras can be run with any of these backends without having to change anything in the code
- it is possible to switch between any backend engine during development — if one of these backends proves to be faster for a specific task
- It is **recommended to use the TensorFlow** backend as the default, because it's the most widely adopted, most scalable, and most, production ready

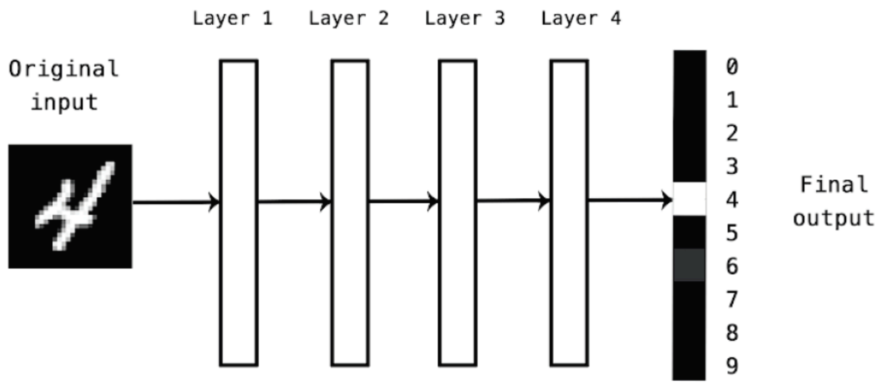
Setting up a deep-learning workstation

- Deep-learning code must run on a modern **NVIDIA GPU**
- Some applications—in particular, image processing with convolutional networks and sequence processing with recurrent neural networks—will be excruciatingly slow on a CPU

- Other alternative is running on **Google Cloud Platform**

Deep Learning – a multistage way to learn data

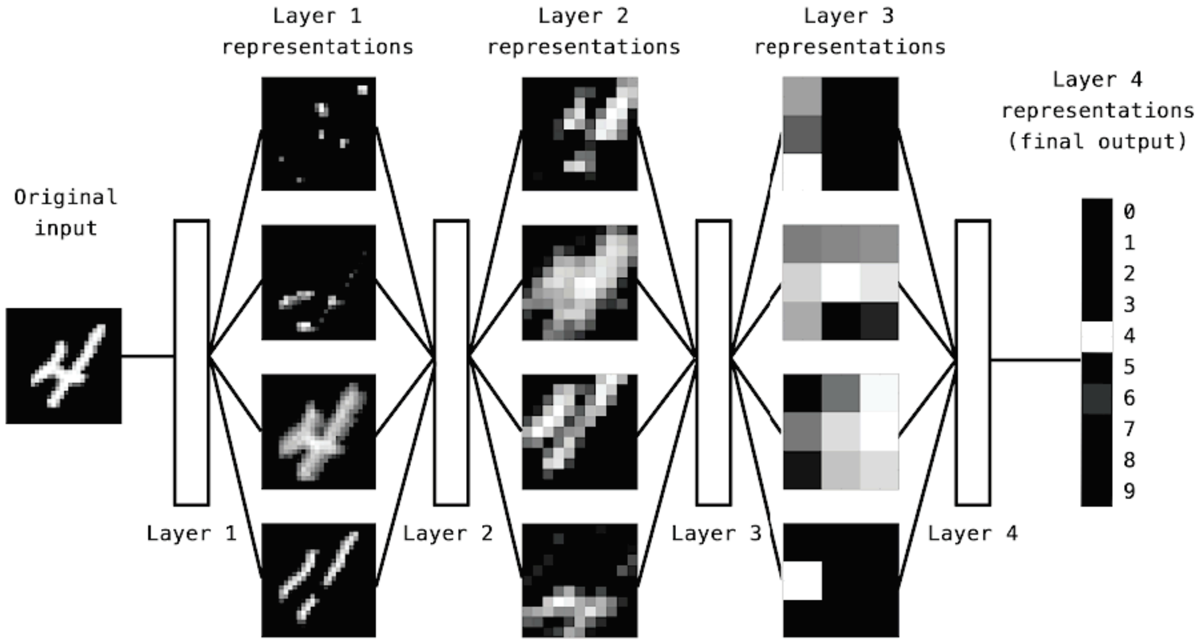
- Deep learning employs a stack of multiple hidden layers of non-linear processing units



- The input of a hidden layer is the output of its previous layer
- Features are extracted from each hidden layer
- Features from different layers represent abstracts or patterns of different levels. Hence, **higher-level features are derived from lower-level features**, which are extracted from previous layers
- All these together form **a hierarchical representation** learned from data

Deep Learning – a multistage way to learn data

A deep network works as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly useful, with regard to some task



Deep learning is, technically: a multistage way to learn data

Tensors

- Tensors are the most common data representation for deep neural networks
- Tensors are a generalization of vectors and matrices to an arbitrary number of dimensions (in the context of tensors, "dimension" is called "axis")
  - Scalars (0D tensors)
  - Vectors (1D tensors)
  - Matrices (2D tensors)
  - 3D tensors and higher-dimensional tensors - are arrays of matrices that can visually be interpret as a cube of numbers
  - An array of 3D tensors is a 4D tensor, and so on

- In deep learning, generally we manipulate tensors that are 0D to 4D, although we may go up to 5D to process video data

Tensor Key attributes

- **Number of axes (rank)** — a 3D tensor has three axes, and a matrix has two axes
- **Shape**— is an integer vector that describes how many dimensions the tensor has along each axis
- **Data type**—This is the type of the data contained in the tensor; A tensor's type could be **integer** or **double**. On rare occasions, can be a character tensor

Data batches

- In general, the **first axis** in a data tensor is the **sample axis** (also called the sample dimension)
- Deep-learning models don't process an entire dataset at once; rather, they break the data into **small batches**

Data pre-processing

VECTORIZATION

All inputs and targets in a deep network must be **tensors of floating-point data** (or, in specific cases, tensors of integers)

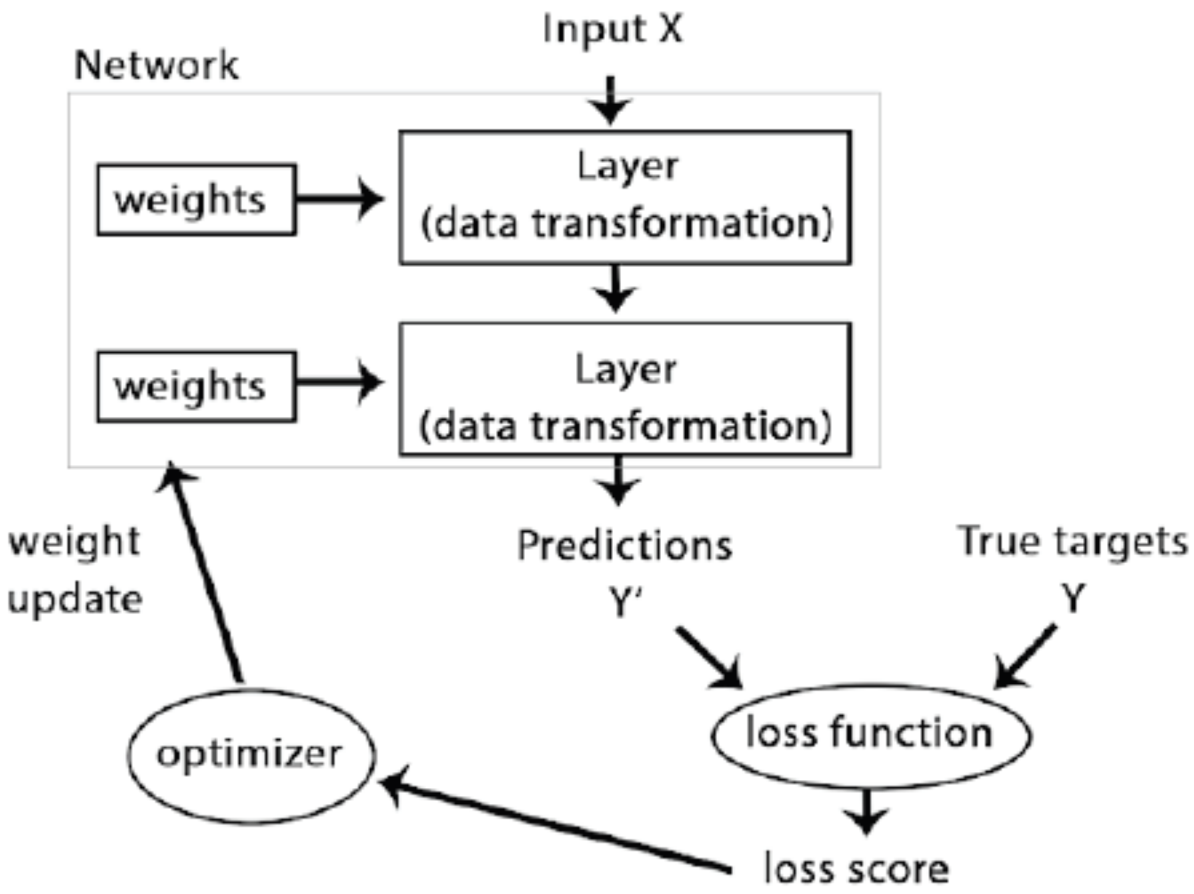
All data need to process: sound, images, text must first turn into tensors — a step called **data vectorization**

NORMALIZATION

- It isn't safe to feed into a neural network data that takes relatively large values
- Or data that is heterogeneous, where the ranges vary greatly — Doing so can trigger large gradient updates that will prevent the network from converging

  The data should have the following characteristics:
  - **Take small values**—Typically, most values should be in the 0–1 range
  - **Be homogenous**—, all features should take values in roughly the same range

Training a Neural Network



Example IMDB dataset

Classifying movie reviews

IMDB dataset: a binary classification example

50,000 highly polarized reviews from the Internet Movie Database

- 25,000 reviews for training
- 25,000 reviews for testing

Each set consisting of 50% negative and 50% positive reviews

The IMDB dataset has already been pre-processed:

- the reviews (sequences of words) are sequences of integers, where each integer stands for a specific word in a dictionary
- It only be considered the top 10,000 most frequently occurring words

IMDB dataset: Vectorization

- The sequences of integers can’t be feed into a neural network
- It is necessary to turn the lists into **tensors**
- **One-hot-encode** the lists to turn them into vectors of 0s and 1s

- The input data is vectors, and the labels are scalars (1s and 0s)

Layers: the building blocks of Deep Learning

Different layers are appropriate for different tensor formats and different types of data processing:

- **2D tensors** that store samples, features is often processed by **densely connected layers**, also called **fully connected** or **dense layers**
- **3D tensors** that store sequence data is typically processed by **recurrent layers**
- **4D tensors** that store image data are usually processed by **2D convolution layers**

IMDB - Network Architecture

```
> network <- keras_model_sequential() %>%
             layer_dense(units = 16, activation = "relu",
                         input_shape = c(10000)) %>%
             layer_dense(units = 16, activation = ”relu") %>%
             layer_dense(units = 1, activation = "sigmoid")
```

The network consists of a sequence of three layers, **densely connected**

- The first layer accept as input 1D tensor, or a vector the dimension 10000. This layer will return a tensor where the first dimension has been transformed to be 16.
- The second layer didn’t receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the previous layer
- The third (and last) layer is a **1-way sigmoid layer**, because we are facing a binary classification problem

Pipe (%>%) operator

- Is from *magrittr package*
- Is used for adding layers to the network
- Is a shorthand for passing the value on it's left hand side as the first argument to the function on the right hand side

```
network <- keras_model_sequential() %>%
           layer_dense(units=512, activation="relu",
                       input_shape = c(28*28)) %>%
           layer_dense(units = 10, activation = "softmax")
```

Is equivalent to

```
network <- keras_model_sequential()
layer_dense(network, units=512, activation="relu",
            input_shape=c(28*28))
layer_dense(network, units=10, activation="softmax")
```

Configuring the Learning Process

After the network architecture definition it is necessary to define:

1. **Activation function**
2. **Loss function (objective function)**—The quantity that will be minimized during training. It represents a measure of success for the task at hand
3. **Optimizer**—Determines how the network will be updated based on the loss function. It implements a **specific variant of stochastic gradient descent**

Activation Function

Without an activation function the layer could only learn linear transformations of the input data —a dot product and an addition:

output = dot(W, input) + b

A hidden layer with a **relu (rectified linear unit)** activation function implements the following chain of tensor operations:

output = relu(dot(W, input) + b)

relu is the most popular activation function in deep learning, because is easier to train and converges faster but, the only point of using it, is to introduce non-linearity

Loss Function

A **Loss function** is the goal of the whole network, it encapsulates what the model is trying to achieve

The most appropriate **loss functions** for a:

- two-class classification problem — **binary crossentropy**
- many-class classification problem — **categorical crossentropy**
- regression problem — **mean-squared error**
- sequence learning problem — **Connectionist Temporal Classification (CTC)**
- truly new research problem — it will be necessary to develop an objective function

IMDB - Compilation Step

In the compilation step it is necessary to define:

- **A loss function** — measures how good the network is performing on its training data
- **An optimizer** —the mechanism through which the network will update itself based on the data it sees and its loss function - **gradient descent** are defined by the *rmsprop* optimizer
- **Metrics to monitor during training and testing** — the most used is accuracy (fraction of the images correctly classified)

```
> model %>% compile( optimizer = "rmsprop",
                     loss = "binary_crossentropy",
                     metrics = c("accuracy") )
```

Training, Validation, Test Sets

The model is developed on the **training set**

Developing a model always involves tuning its configuration:
- choosing the number of layers or the size of the layers - **hyperparameters**
- choosing the network's weights - **parameters**

Tuning is repeat many times:
- running one experiment, evaluating on the **validation set**
- and modifying the model as a result - a significant amount of information about the validation set is put into the model

The model performance must be evaluate on a completely new dataset - **test dataset**

IMDB - Train, Validation and Test Sets

```
> train_data <- imdb$train$x
> train_labels <- imdb$train$y

> test_data <- imdb$test$x
> test_labels <- imdb$test$y


> val_indices <- 1:10000
> x_val <- x_train[val_indices,]
> y_val <- y_train[val_indices]

> partial_x_train <- x_train[-val_indices,]
> partial_y_train <- y_train[-val_indices]
```

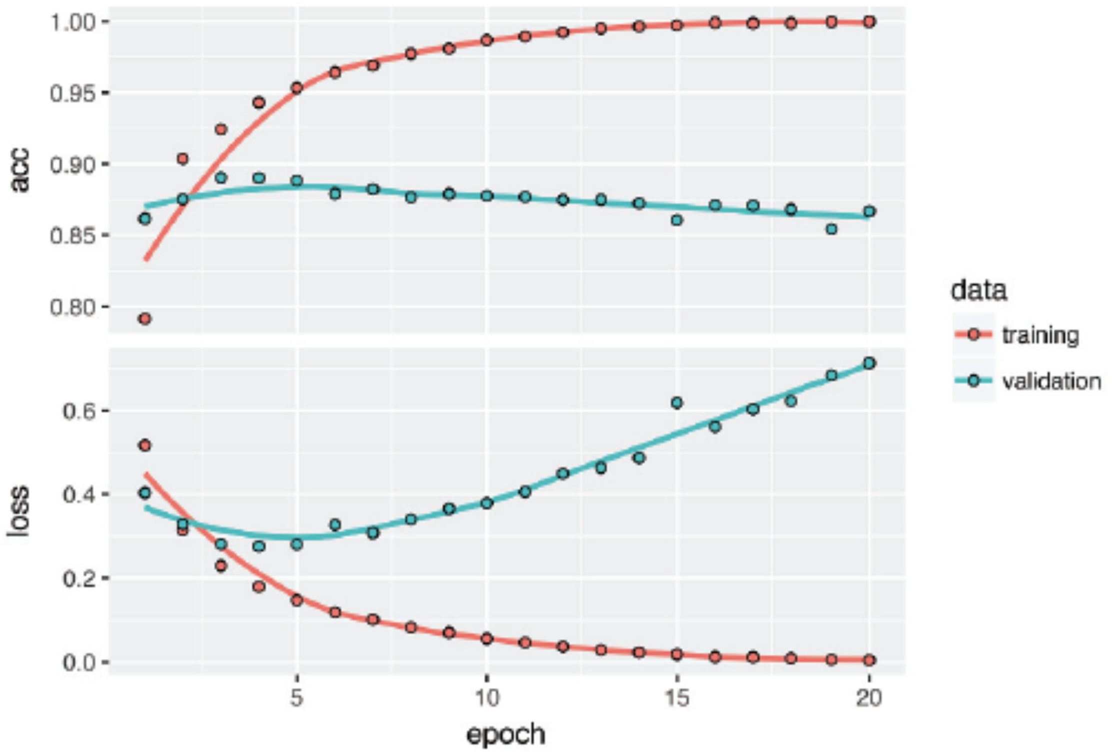
IMDB - Train Step

```
> history <- model %>% fit(partial_x_train,
                           partial_y_train,
                           epochs=20,
                           batch_size=512,
                           validation_data=list(x_val,y_val) )
```

- the network fit method iterates on the training data in mini-batches of 512 samples, 20 times over (each iteration over all the training data is called an **epoch**)
- At each iteration, the **network will compute the gradients of the weights with regard to the loss on the batch**, and **update the weights accordingly**

Plot the training and validation metrics

> plot(history)



- The **training loss decreases** and the **training accuracy increases** with every epoch — which is expected when running a **gradient-descent optimization**
- But, for the **validation loss and accuracy** peak at the **fourth epoch** — **overfitting**
- To prevent overfitting, stop training after three epochs, or use a range of techniques to mitigate overfitting

Prevent overfitting in neural networks

The most common ways to prevent overfitting in neural networks:

- **Get more training data**

- **Reduce the capacity of the network**: the number of layers and the number of units per layer
  - Start with relatively few layers and parameters, and increase the size of the layers or add new layers, until diminishing returns with regard to validation loss

- **Add weight regularization**
  - Is done by adding to the loss function of the network a cost associated with having large weights

- **Add dropout**
  - randomly dropping out (setting to zero) a number of output features of the internal layers during training

Prevent overfitting: Weight regularization

Overfitting can be reduced by putting constraints on the complexity of a network - forcing its weights to take only small values - makes the distribution of weight values more regular — **weight regularization**

Weight regularization is done by adding to the loss function of the network a cost associated with having large weights.

There are two kind of regularization:

- **L1 regularization** —The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights)
- **L2 regularization** —The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights)

IMDB - Train step with Weight regularization

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments

```
> model <- keras_model_sequential() %>%
          layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
                      activation = "relu", input_shape = c(10000)) %>%
          layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
                      activation = "relu") %>%
          layer_dense(units = 1, activation = "sigmoid")
```

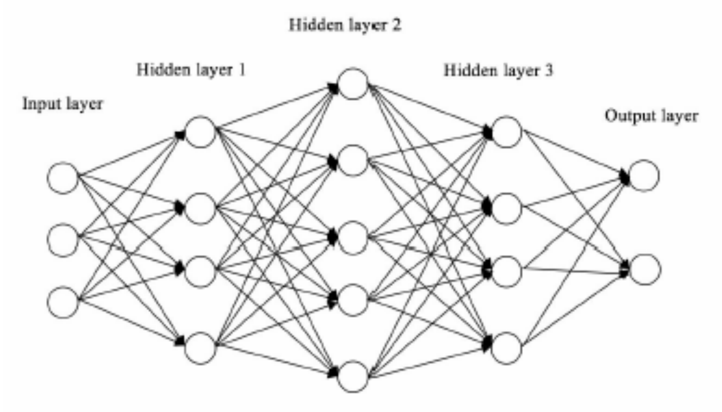
Every coefficient in the weight matrix of the layer will add to the total loss of the network 0.001*weight_coefficient_value

Different weight regularizers available in Keras

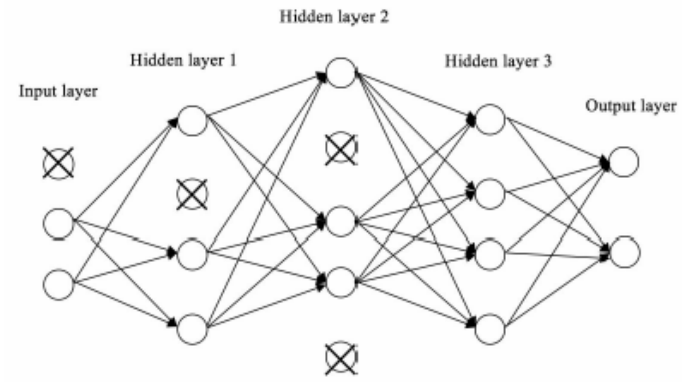
- regularizer_l1(0.001)
- regularizer_l2(0.001)
- regularizer_l1_l2(l1 = 0.001, l2 = 0.001)

Prevent overfitting: Dropout

Dropout consists of **randomly** dropping out (**setting to zero**) a number of output features of the internal layers **during training**



Standard NN



NN with dropout

- **Dropout rate:** is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5
- At **test time**, no units are dropped out

IMDB - Train step with dropout

In Keras **dropout** is inserted in a network via *layer_dropout* which is applied to the output of the layer right before it

```
> model <- keras_model_sequential() %>%
        layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
        layer_dropout(rate = 0.5) %>%
        layer_dense(units = 16, activation = "relu") %>%
        layer_dropout(rate = 0.5) %>%
        layer_dense(units = 1, activation = "sigmoid")
```

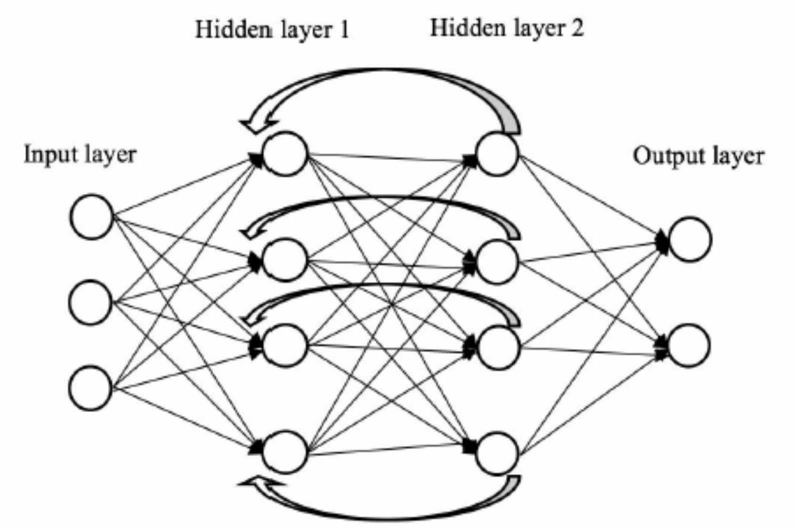
Keras workflow

1. Define the training data: input tensors and target tensors
2. Define a network of layers (or model) that maps the inputs to the targets
3. Configure the learning process by choosing:
   - a loss function,
   - an optimizer,
   - some metrics to monitor
4. Iterate on the training data by calling the fit() method with the model developed

Deep Learning Types

Recurrent Neural Networks (RNNs)

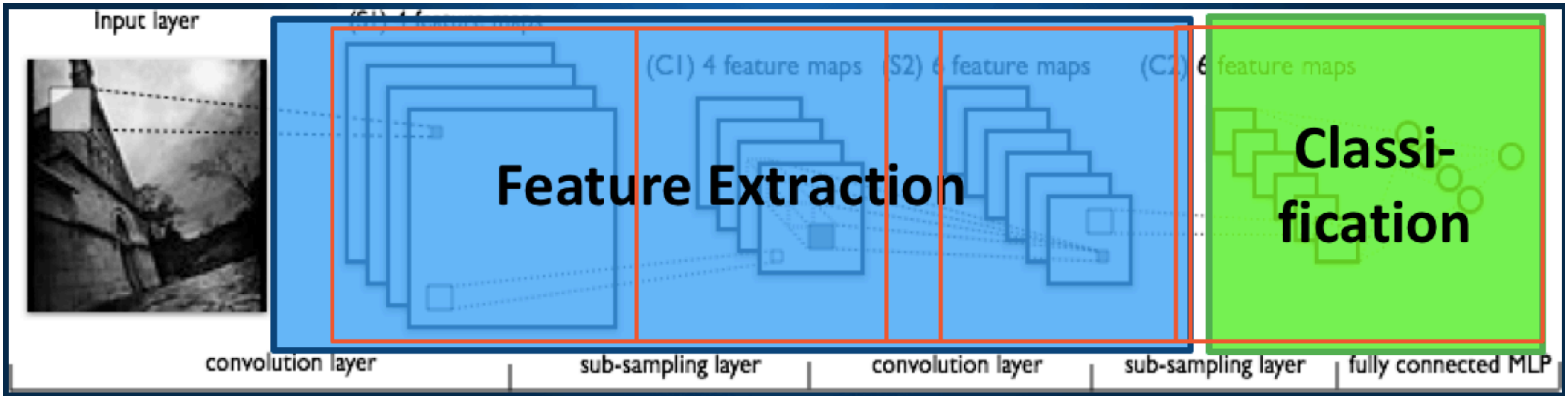
- processes information incrementally while maintaining an internal model of what it's processing, built from past information and constantly updated as new information comes in — **keeps memories of what came before**
- are designed to work with sequence prediction problems and sequence data in general such as, **speech recognition, natural language processing, timeseries.**



Deep Learning Types

Convolutional Neural Networks (CNNs / ConvNets)

- Is a type of artificial neural network used in **image processing and computer vision** that is specifically designed to process **pixel data**



Convolutional Neural Networks (CNNs / ConvNets)

CNNs specifically are inspired by the biological visual cortex. The cortex has small regions of cells that are sensitive to the specific areas of the visual field

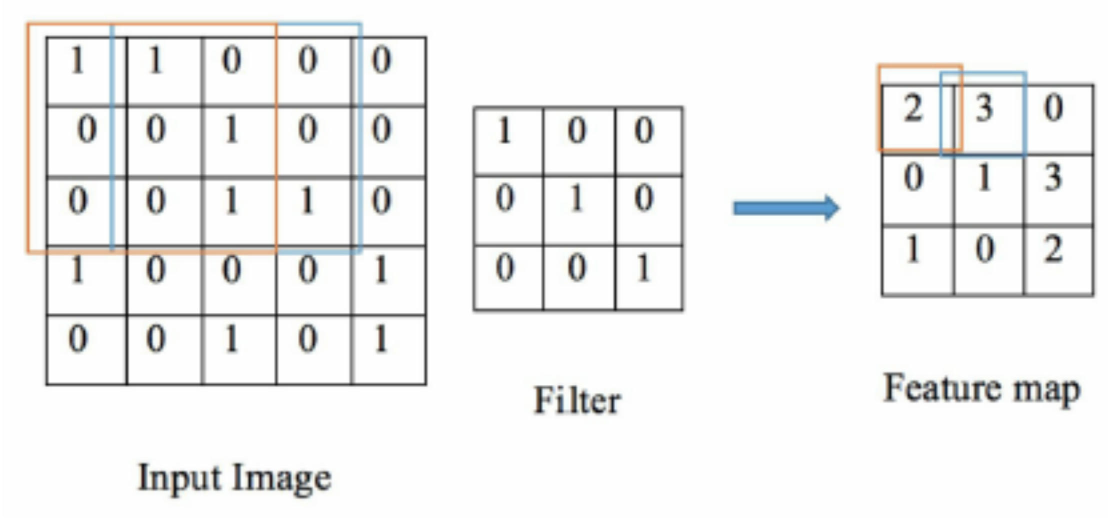
A ConvNet is a sequence of layers that transforms one volume of activations to another through a differentiable function

The ConvNet architectures are built with four main types of layers:

- Convolutional layer
- Non-linear layer (or activation layer)
- Pooling layer (or downsampling layer)
- Full Connection layer

The convolutional layer

- Computes the **dot product** between the weights of the convolutional layer and a small region connected to in the input layer
- The **small region is the receptive field**, and the **weights** are the values on a filter. As the filter slides from the beginning to the end of the input layer, the dot product between the weights and current receptive field is computed
- A **new layer called feature map** is obtained after convolving over all sub-regions

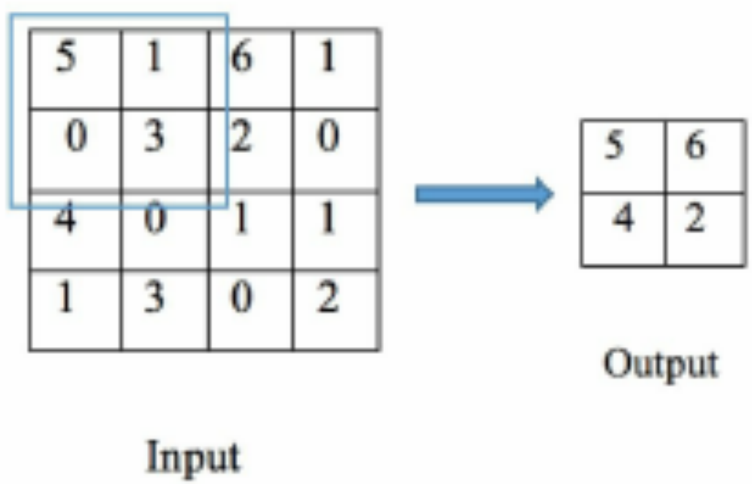


The non-linear layer

- After each convolutional layer, there is a **non-linear layer** - called **activation layer**, in order to introduce non-linearity
- **ReLu** is the most popular function for the non-linear layer in deep neural networks
- Normally, after obtaining features via one or more pairs of convolutional layers and non-linear layers, the dimension increases dramatically, which can easily lead to **overfitting**

The pooling layer

- The **pooling layer** (also called **downsampling layer**) aggregates statistics of features by sub-regions to generate much lower dimensional features
- Typical pooling methods include **max pooling** and **mean pooling**, which take the max values and mean values over all non-overlapping sub-regions
- Example, of a 2*2 max pooling filter on a 4*4 feature map:



Full Connected Layers

- CNN takes in an image, pass it through a sequence of convolutional layers, nonlinear layers, pooling layers
- After image processing it uses **fully connected layers** to output the probabilities of each possible class