Neural Networks

Analogy: natural learning

The brain is a highly complex, **non-linear, parallel** structure

It has an ability to organize its neurons to perform complex tasks

A neuron is 5/6 times slower than a logic gate

The brain overcomes slowness through a parallel structure

The human cortex has 10 billion neurons and 60 trillion synapses

What are neural networks?

(Artificial) Neural networks are models of machine learning that follow an **analogy** **with the functioning of the human brain**

A neural network is a **parallel** processor, consisting of simple processing units (neurons)

Knowledge is stored in the **connections** between the neurons

Knowledge is acquired from the environment (data) through a **learning process** (training algorithm) that **adjusts the weights** of the connections

Basic unit - Artificial neurons

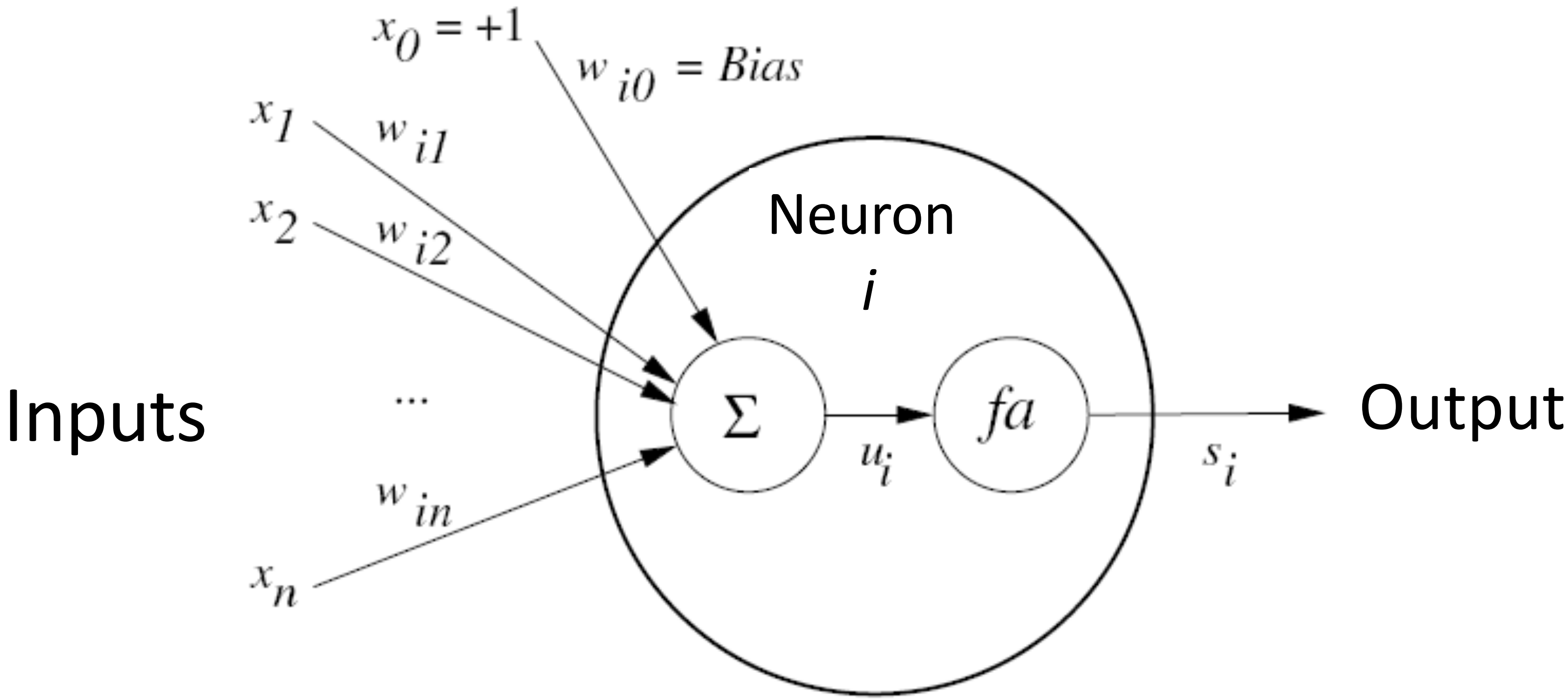
Receive a set of inputs (data or connections)

A **weight** (numerical value) is associated with each connection

Each neuron calculates its **activation** based on the input values and the weights of the connections

The calculated signal is passed on to the output after being filtered by an **activation function**

Structure of a neuron



What model do you get if the activation function *fa* is the identity function ?

Activation functions

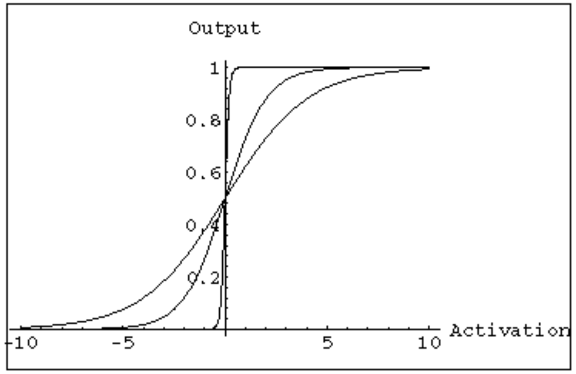
Sigmoid/Logistic

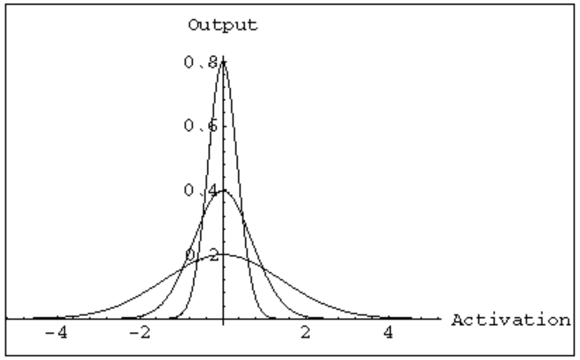
Linear

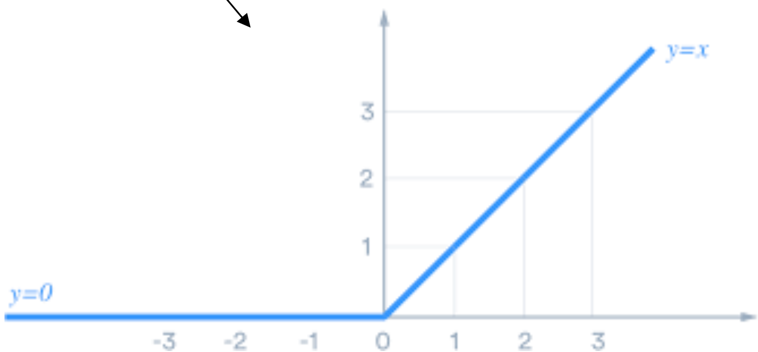
Hyperbolic tangent (Tanh)

Gaussian

RelU (Rectified linear)







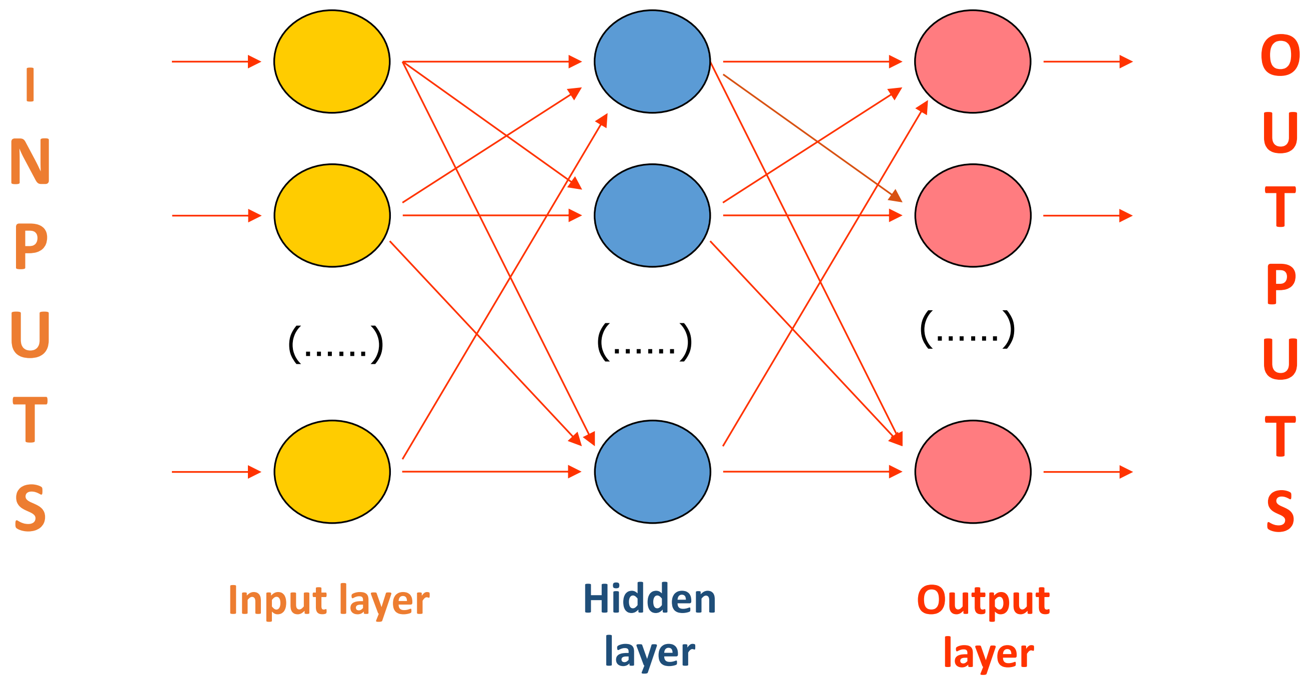
What model do you get if the activation function *fa* is the sigmoid function ?

Network topologies

Architecture (or **topology**) - the way nodes interconnect in a network structure (graph)

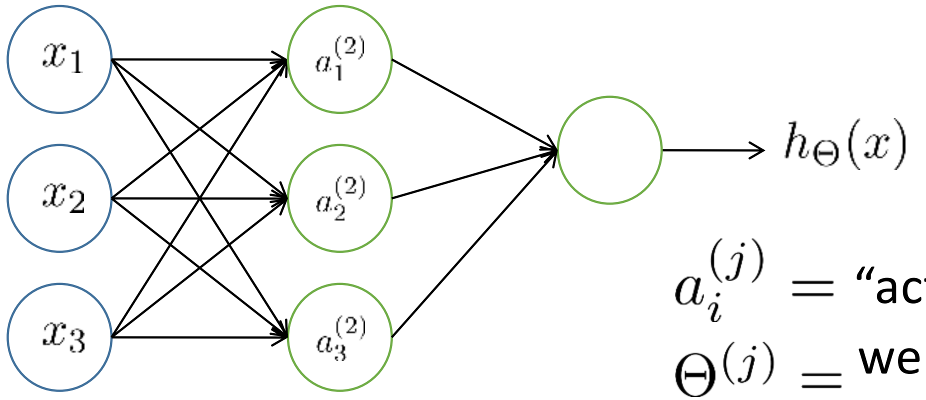
There are countless types of architectures, each with their own potentialities, falling into two categories: supervised and unsupervised, regarding the way they are trained

Feedforward neural network



Multilayer perceptrons (MLPs)

Neural network – computing output



$a_i^{(j)}$ = "activation" of neuron *i* in layer *j*

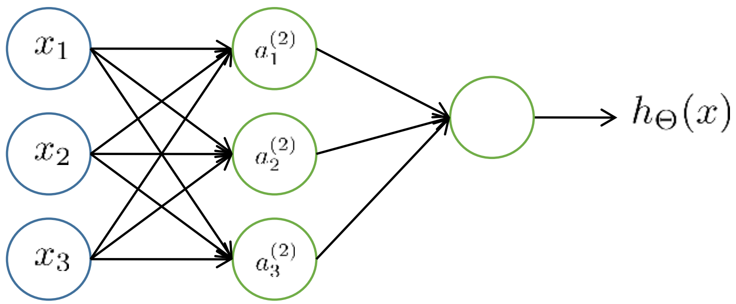
$\Theta^{(j)}$ = weight matrix for connections between neurons of layers *j* and *j+1* (rows – destination, columns - origin)

sigmoid =1

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

=1

Neural network – computing output - vectorized



$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)} x$$

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

Hidden layer activation values

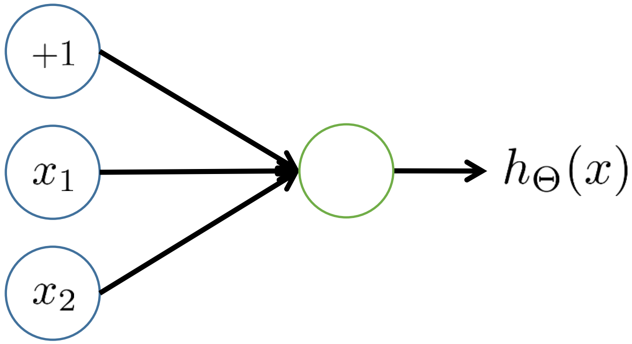
Add $a_0^{(2)} = 1$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$h_\Theta(x) = a^{(3)} = g(z^{(3)})$$

Output value

Computing the output - exercise



x_1	x_2	$h_\Theta(x)$
0	0	
0	1	
1	0	
1	1	

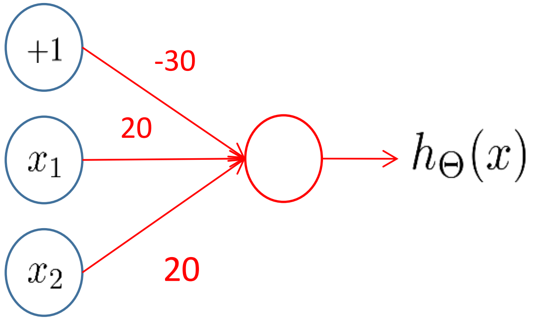
Calculate the output value for the cases where the weights are :

- 30, 20, 20

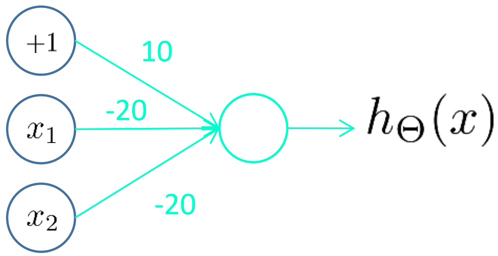
-10, 20, 20

10, -20, -20

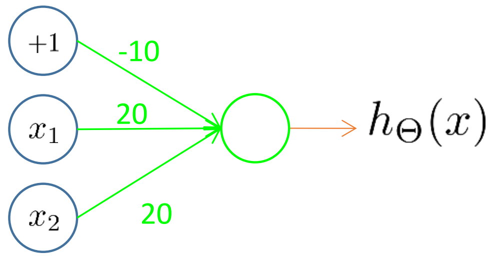
Computing the output - exercise



x_1 AND x_2



(NOT x_1) AND (NOT x_2)

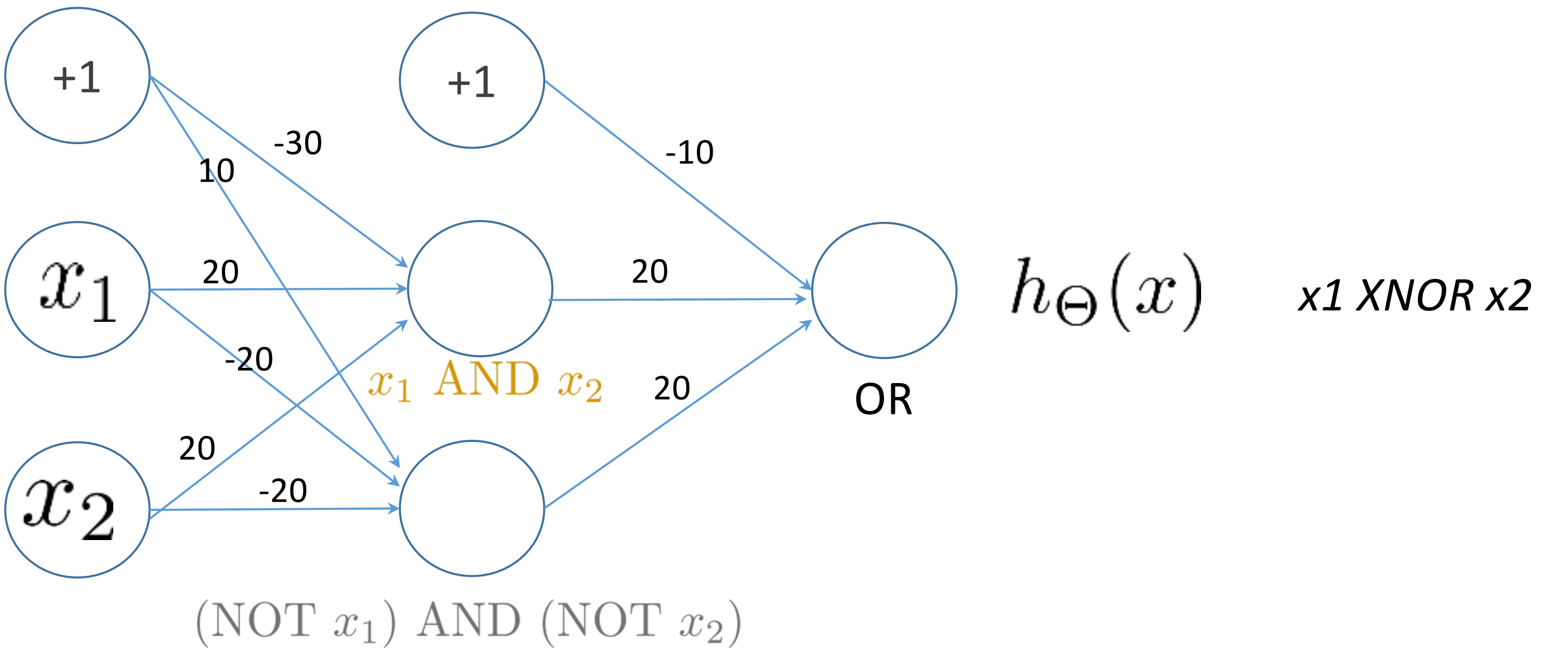


x_1 OR x_2

How to do H (x1, x2) = x1 XNOR x2 with a network with an hidden layer?

Note that: x1 XNOR x2 = (x1 AND x2) OR (NOT x1 AND NOT x2)

Computing the output - exercise



x1 XNOR x2 = (x1 AND x2) OR (NOT x1 AND NOT x2)

Pre-processing the data

Data standardization in neural networks is common given the used activation functions used; features with very different distributions of values are not convenient

Missing values in input features may be represented as zeros, which do not influence the neural net training process

Interpreting the outputs in classification

When using neural nets (and other functional models) to address multiclass classification datasets, we need to convert the output of the model (numerical value) into the discrete value (predicted class) that is desired.

As mentioned before, we will typically use one neuron per class, which implies to apply **one-hot encoding** to the output variable

In one-hot encoding, there are M output neurons (1 per class), being chosen for any case the class with the highest value.

Using the **softmax** activation function, we can get probabilities for all classes

NNs for the different supervised tasks

When using neural nets to address supervised learning tasks, three main types of cases arise, which demand different configurations of the network and the training process:

Problem type	Output layer	Activation function (output)	Loss function
Binary classification	One single node	Sigmoid	Binary Cross entropy
Multiclass classification	One node per class	Softmax (sigmoid in each node, normalized to sum 1)	Generalized cross entropy
Regression	One node	Linear (or RELU if only positive outputs are possible)	Mean Squared Error

Training: supervised learning

Data: Training examples consisting of **inputs** and their desired **outputs**

Objective: To set the values of the **connection weights** that minimize a cost function

Multiple **gradient-descent** algorithms exis:

- The most used historically is ***Backpropagation*** and derivatives
- Other algorithms: Marquardt-Levenberg, Rprop, Quickprop
- Most recent: RMSprop, Adam, SGD, Adagrad

NN training will be detailed in the next session !!

Python Implementation - numpy - Exercise

Define a class MLP that implements a neural network with an intermediate layer, with the following fields:

- ***X, y*** – taken from the dataset (as in the case of linear/ logistic regression)
- ***h*** – number of neurons in the hidden layer
- ***W1***– matrix of weights connecting the input layer to the intermediate layer (dimension h x n+1 – where n is the number of inputs)
- ***W2*** – Weight matrix connecting the intermediate layer to the single output (matrix of dimensions 1 by h+1

Methods:

- ***predict*** – predict output for an exemple
- ***predictMany*** – predict outputs for a set of examples (e.g. training or test set)
- ***costFunction*** – calculate error for the training set (error function equal to that defined in linear or logistic regression)
- ***buildModel*** – neural network training

Python implementation - numpy

Constructor

```
class MLP:

   def __init__(self, dataset, hidden_nodes = 2):
        self.X, self.y = dataset.getXy()
        self.X = np.hstack ( (np.ones([self.X.shape[0],1]), self.X ) )
        self.h = hidden_nodes
        self.W1 = np.zeros([hidden_nodes, self.X.shape[1]])
        self.W2 = np.zeros([1, hidden_nodes+1])

        if normalize:
            self.normalize()
        else:
            self.normalized = False
```

Python implementation - exercise

Predicting the output for an example or a set of examples


```
def  predict( self,  instance):
          x = np.empty([self.X.shape[1]])
          x[0] = 1
          x[1:] = np.array(instance[:self.X.shape[1]-


     if self.normalized: …


          …
```

```
def predictMany(self, Xpred = None):

      if Xpred is None: ## use training set
         Xp = self.X
      else:
         Xp = Xpred

      ...
```

PYTHON IMPLEMENTATION

Python implementation - solution

Predicting output for an example - solution

```
def predict( self, instance):
        x = np.empty([self.X.shape[1]])
        x[0] = 1
        x[1:] = np.array(instance[:self.X.shape[1]-

        if self.normalized: …

        z2 = np.dot(self.W1, x)
        a2 = np.empty([z2.shape[0]+1])
        a2[0] = 1
        a2[1:] = sigmoid(z2)
        z3 = np.dot(self.W2, a2)

        return sigmoid ( z3 )
```

Python implementation - solution

PYTHON IMPLEMENTATION

Predicting output for a set of examples

```
def predictMany(self, Xpred = None):

    if Xpred is None: ## use training set
        Xp = self.X
    else:
        Xp = Xpred

    Z2 = np.dot(Xp, self.W1.T)
    A2 = np.hstack ( (np.ones([Z2.shape[0],1]), sigmoid(Z2) ))

    Z3 = np.dot(A2, self.W2.T)
    predictions = sigmoid(Z3)

    return predictions
```

Python implementation - exercise

Cost function – implement for mean of squared errors


```
def costFunction(self, weights = None, loss = "mse"):
    if weights is not None:
        self.W1 = weights[:self.h * self.X.shape[1]].reshape([self.h, self.X.shape[1]])
        self.W2 = weights[self.h * self.X.shape[1]:].reshape([1, self.h+1])

    predictions = self.predictMany()
    m = self.X.shape[0]

        …
```

PYTHON IMPLEMENTATION

Python implementation - solution

Cost function – sum of squared errors or cross entropy

```
def costFunction(self, weights = None, loss = "mse"):
      …

if loss == "mse":
         sqe = (predictions- self.y.reshape(m,1)) ** 2
         res = np.sum(sqe) / (2*m)
      elif loss == "entropy":
         p = np.clip(predictions, 1e-15, 1 - 1e-15)    # avoid log zero
         cost  = (-self.y.dot(np.log(p)) - (1-self.y).dot(np.log(1-p)) )
         res = np.sum(cost) / m
      else:
         print("Non existing loss")
         return None
return res
```

Need to first do prediction for a set of examples

PYTHON IMPLEMENTATION

Python implementation - numpy

Example - XNOR

```
def  setWeights(self, w1, w2):
        self.W1 = w1
        self.W2 = w2
```

```
def test():
    ds= Dataset("xnor.data")
    nn = MLP(ds, 2)
    w1 = np.array([[-30,20,20],[10,-20,-20]])
    w2 = np.array([[-10,20,20]])
    nn.setWeights(w1, w2)

    print( nn.predict(np.array([0,0]) ) )
    print( nn.predict(np.array([0,1]) ) )
    print( nn.predict(np.array([1,0]) ) )
    print( nn.predict(np.array([1,1]) ) )
    print(nn.costFunction())

test()
```

PYTHON IMPLEMENTATION

Python implementation - numpy

Model training – uses scipy optimization methods

```
def costFunction(self, weights = None):
    if weights is not None:
        self.W1 = weights[:self.h * self.X.shape[1]].reshape([self.h, self.X.shape[1]])
        self.W2 = weights[self.h * self.X.shape[1]:].reshape([1, self.h+1])

(…)
```

```
def build_model(self):
        from scipy import optimize
        size = self.h * self.X.shape[1] + self.h+1
        initial_w = np.random.rand(size)
        result = optimize.minimize(lambda w: self.costFunction(w), initial_w, method='BFGS',
                            options={"maxiter":100, "disp":False} )
        weights = result.x
        self.W1 = weights[:self.h * self.X.shape[1]].reshape([self.h, self.X.shape[1]])
        self.W2 = weights[self.h * self.X.shape[1]:].reshape([1, self.h+1])
```

Python implementation - numpy

Example - XNOR

```
def  test2():
      ds=  Dataset("xnor.data")
      nn = MLP(ds,  5)
      nn.build_model()
      print(  nn.predict(np.array([0,0])  )  )
      print(  nn.predict(np.array([0,1])  )  )
      print(  nn.predict(np.array([1,0])  )  )
      print(  nn.predict(np.array([1,1])  )  )
      print(nn.costFunction())

test()
```

Try with different numbers of neurons in the hidden layer

Implementing in Python: scikit_learn

The **MLPClassifier** and **MLPRegressor** classes implement NNs in scikit-learn

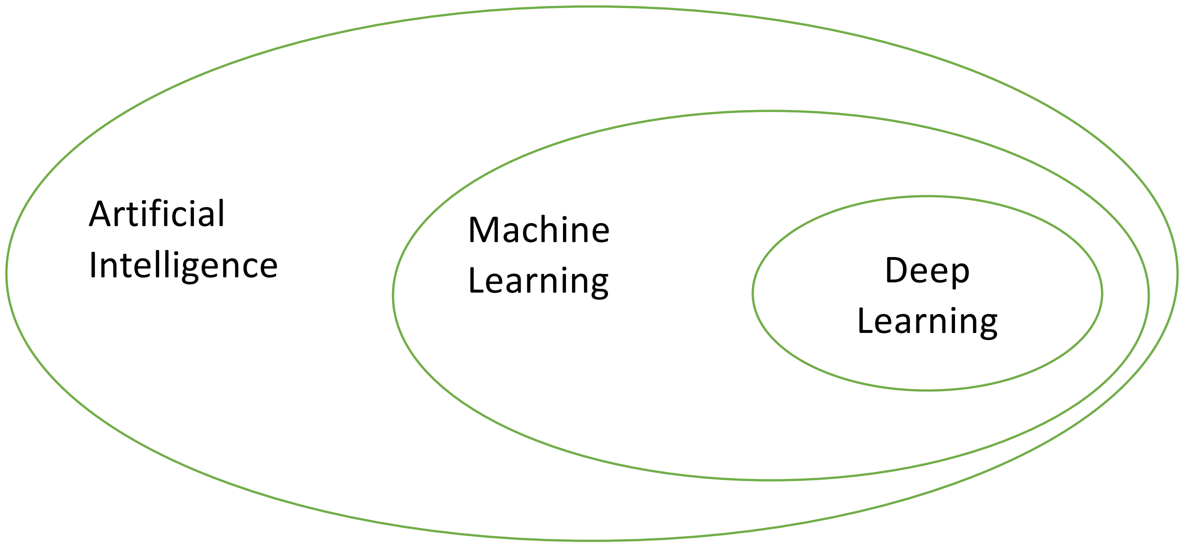
These classes follow the same interface all other supervised models

Deep neural networks

Deep learning: what is it?

Machine Learning field characterized by a greater complexity of models and the ability to learn representations from input data

Deep learning models consist of successive layers of representations, in a number that is typically high (deep models)



Deep learning: what is it?

Used models are typically based in neural networks structured in several processing layers

Different types of neurons and architectures used for different types of problems: feedforward neural networks, recurrent networks, convolutional networks, etc.

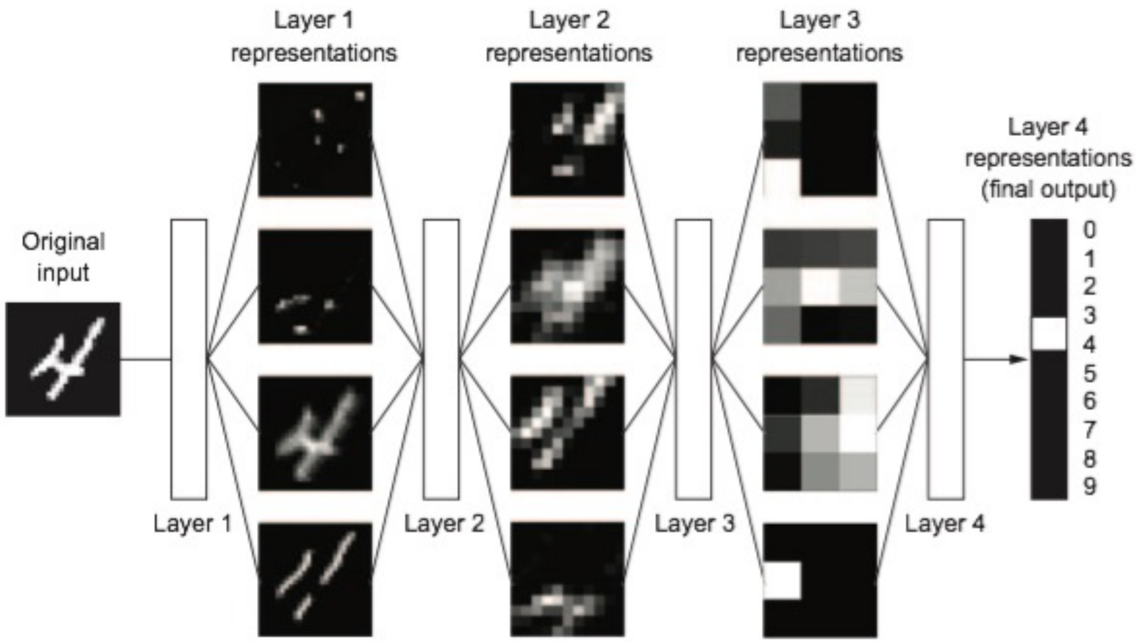
There are deep learning models for supervised and unsupervised learning problems, as well as reinforcement learning

Deep learning: what is it?

Several layers process information by creating distinct and typically more abstract representations of the inputs

In the case of supervised learning, the last layer represents the output of the neural net

Learning by **gradient descent** methods



Deep learning: major factors

Like any technology, DL does not solve all problems and will not always be the best option for any learning task

A determining factor for DL success is the availability of large-scale datasets; for problems with little data, other models can give more consistent results with less computational effort

In terms of hardware, the use of graphics processors (GPU) brings advantages in training DL models by accelerating the process by factors of more than 10x

Improvements in relation to "shallow" neural networks: activation functions (RelU), weight initialization, optimization algorithms (RMSprop, Adam), methods for addressing overfitting, pre-training, and training "by layers"

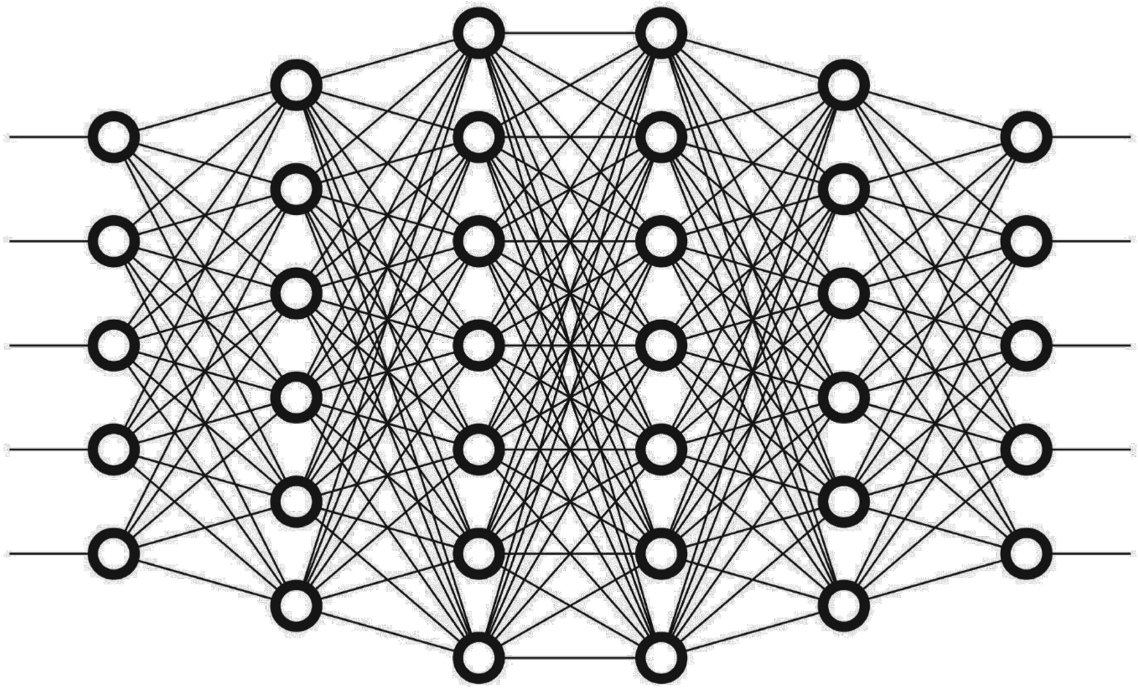
Areas of application

DL/DNNs have been applied in several fields with high quality results, including:

- Image classification (e.g. ImageNet)
- Spoken Text Recognition
- Handwritten text transcription
- Automatic translation of texts
- Natural Language Responses / Digital Assistants
- Gaming (e.g. Go)
- Chemical retrosynthesis
- Classification of protein and DNA sequences

Deep neural networks (DNNs)

DNNs are supervised DL models, being **feedforward** NNs with typically several hidden layers



Implementation in Python: keras and tensorflow

To implement python DL models we will use the ***tensorflow*** package: this allows you to create, train, and apply several distinct DL models

Keras (tensorflow.keras) contains interfaces that allow you to use the tensorflow backend; allows you to run on CPU or GPU depending on the machine's h/w. Allows to create the computation graph providing a set of high-level classes to define model structure.

To install tensorflow typically running ***pip install tensorflow*** will be enough

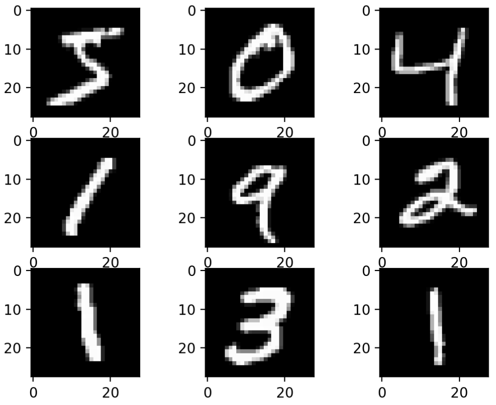
An alternative is to run on google collab: https://colab.research.google.com/

MNIST dataset

Jupyter Notebook:
ex_dnn_mnist.ipynb

MNIST dataset	Dataset with images of digits in 28x28 pixels grid with 784 features representing pixels in grey scale 0 to 255. Output: class representing the digit (10 classes, digits 0-9). Available as a keras dataset. 60K samples – training set; 10K images – test

This dataset will be used here in the following example, vectorizing images to a 1D vector with 784 values as inputs for each image, and defining a multi-class classification problem with 10 possible outputs



DNNs for MNIST

Defining network topology; feed forward

```
network = models.Sequential()
network.add(Input( (28*28,) ) )
network.add(layers.Dense(512, activation='relu'))
network.add(layers.Dense(256, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))
```

Two hidden layers with 512 and 256 neurons (RelU) and one output layer with 10 neurons (*softmax* - 1 neuron w/ sigmoid function for each class; one-hot encoding)

Exercise

- Play with the number of layers and with the number of neurons in each layer

DNNs for MNIST

`network.summary()`

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 10)                2570

=================================================================
Total params: 535818 (2.04 MB)
Trainable params: 535818 (2.04 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

DNNs for MNIST

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Define training algorithm (RMSprop), *loss function* (*cross entropy*) and error metric (*accuracy*)

Train the model (*fit*):
Define size of each batch and number of iterations (epochs)

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

DNNs for MNIST

Predicting probabilities ...

```
test_preds = network.predict(test_images)
```

... and class labels

```
test_classes = np.argmax(network.predict(test_images), axis=-1)
```

Evaluate the model in the test set:
calculate loss and accuracy

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

Boston housing dataset - regression

Jupyter Notebook:
ex_dnn_housing.ipynb

Boston dataset

Predicting the average price of houses in Boston (in the 1970s)
Inputs: several features from the house
404 training examples
102 test examples

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) =
        boston_housing.load_data()

print(train_data.shape)
print(test_data.shape)
print(max(train_targets), min(train_targets))
```

Boston housing: preparing the data

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Standardizing inputs

Boston housing: DNN

```
def build_model(hidden = 64):
    model = models.Sequential()
    model.add(Input((train_data.shape[1],)))
    model.add(layers.Dense(hidden, activation='relu') )
    model.add(layers.Dense(hidden, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Function to create the model
Two hidden layers with 64 neurons (default); RelU
Output layer with one output; linear activation
Loss function: MSE
Metric: Mean Absolute Error

Boston housing: DNN

Build the model with training data

```
model = build_model()
model.fit(train_data, train_targets, epochs=80, batch_size=16, verbose=1)


test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
print(test_mse_score, test_mae_score)
```

Evaluate in test set

Exercises

- try with different number of hidden nodes
- adapt the function to be able to receive other inputs (number layers, metric, dropout, …)

Exercise

- Load the dataset HAR - *Human Activity Recognition using Smartphones* dataset
- Dataset description:

The experiments have been carried out with a group of 30 volunteers. Each person performed six activities (WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING) wearing a smartphone. Using its embedded accelerometer and gyroscope, we captured 3-axial linear acceleration and 3-axial angular velocity. The experiments have been video-recorded to label the data manually.

- **Variables:** For each record in the dataset it is provided:
  - A 561-feature vector with time and frequency domain variables.
  - Its activity label.
  - An identifier of the subject who carried out the experiment.
- More details:

https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

The exercise aims to create Deep Neural Networks for this dataset and compare the performance with “shallow” ML models