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

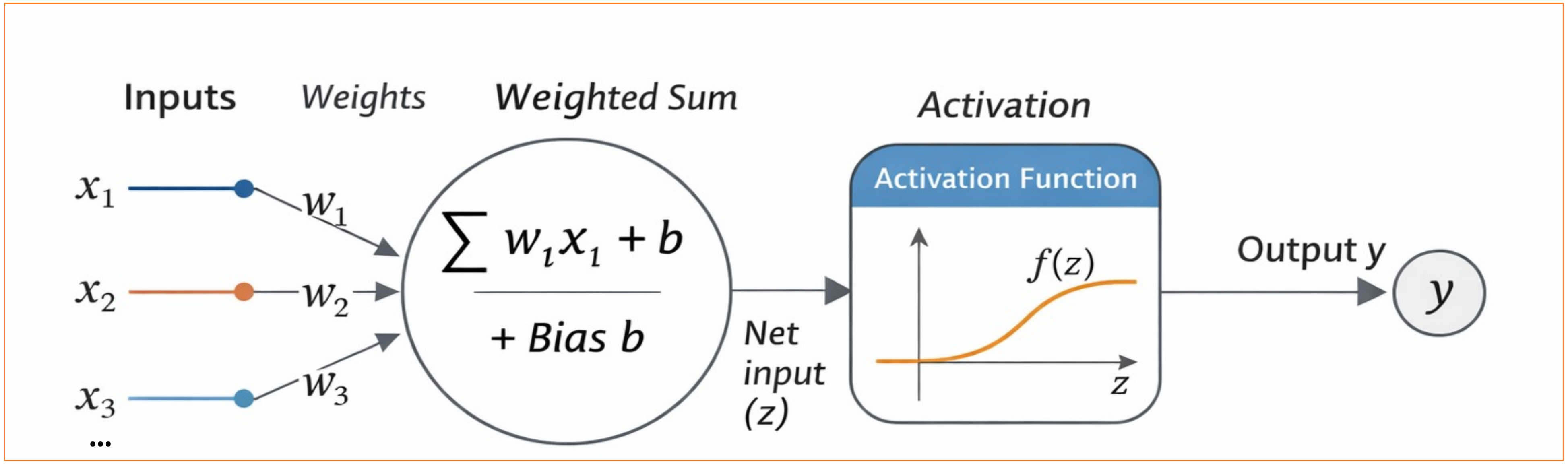
Each neuron calculates its **activation** based on the input values and the weights of the connections

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

Structure of a neuron

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n + b$$

$$y = f(z)$$



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

Activation functions

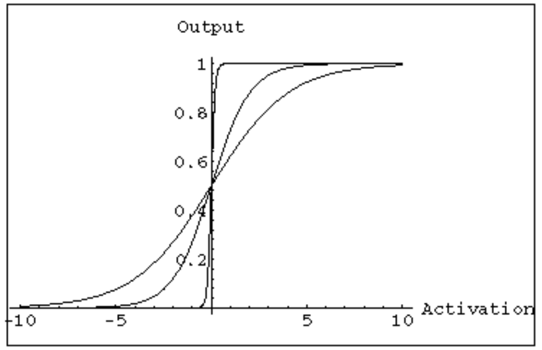
Sigmoid/Logistic

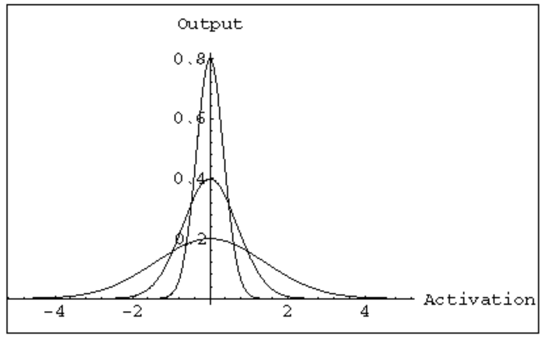
Linear

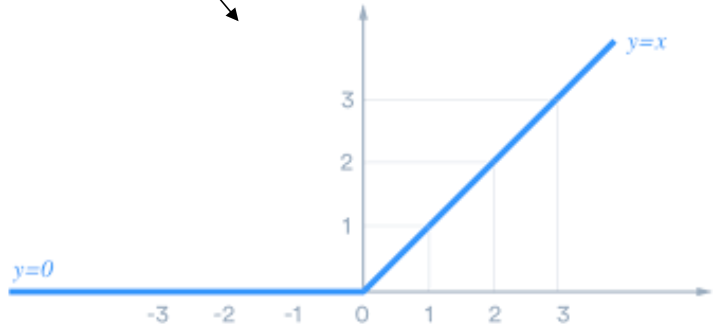
Hyperbolic tangent (Tanh)

Gaussian

RelU (Rectified linear)







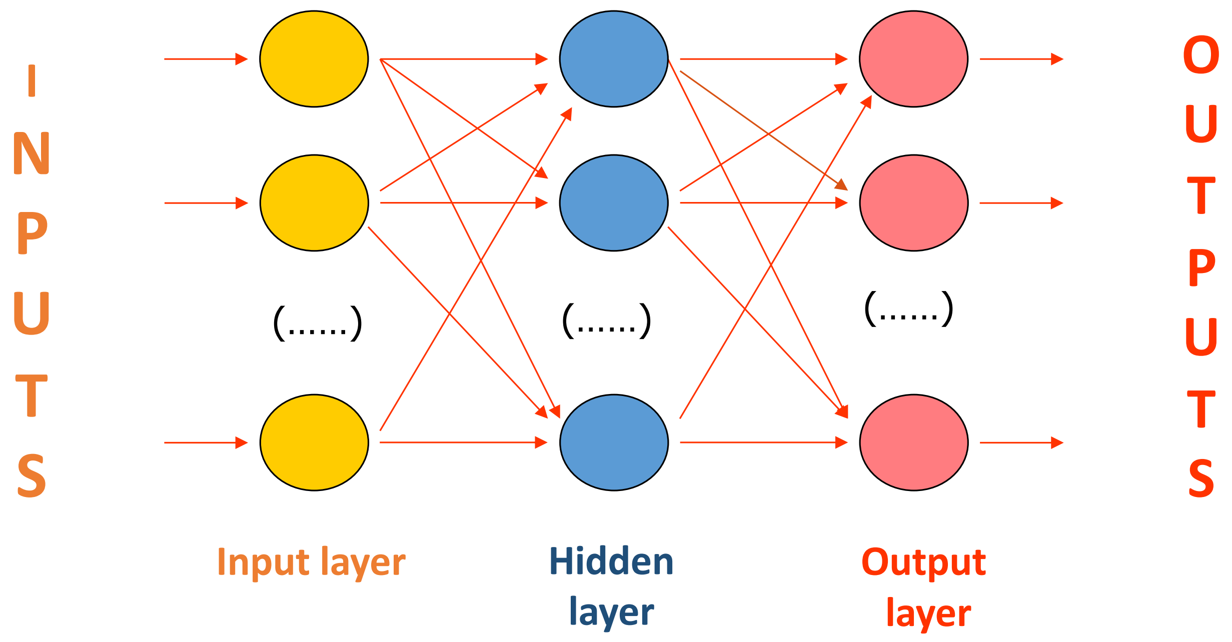
What model do you get if the activation function is the sigmoid function ?

Network topologies

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

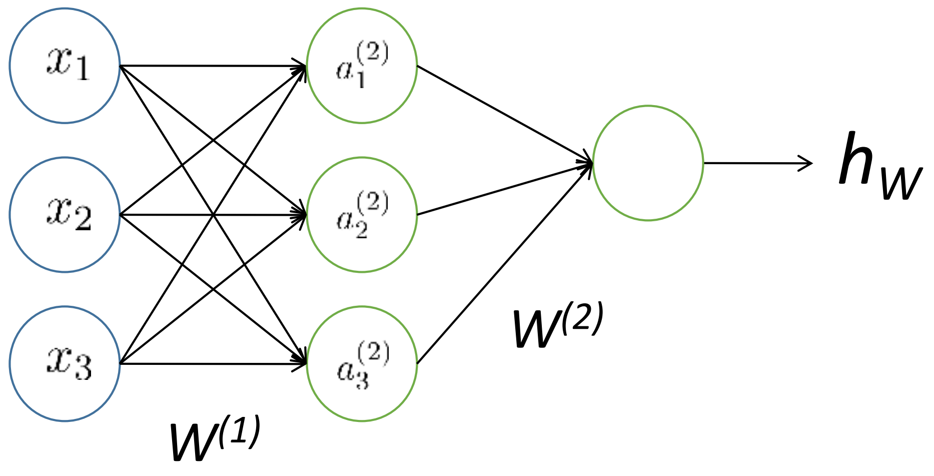
There are countless types of architectures, each with their own potentialities, falling into two categories: supervised and unsupervised, regarding the way they are trained

Feedforward neural network



Multilayer perceptrons (MLPs)

Neural network – computing output



$$a_1^{(2)} = f(b_1^{(1)} + W_{11}^{(1)}x_1 + W_{21}^{(1)}x_2 + W_{31}^{(1)}x_3)$$
$$a_2^{(2)} = f\left(b_2^{(1)} + W_{12}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{32}^{(1)}x_3\right)$$
$$a_3^{(2)} = f\left(b_3^{(1)} + W_{13}^{(1)}x_1 + W_{23}^{(1)}x_2 + W_{33}^{(1)}x_3\right)$$

$$h_W = a_1^{(3)} = f(b_1^{(2)} + W_{11}^{(2)}a_1^{(2)} + W_{21}^{(2)}a_2^{(2)} + W_{31}^{(2)}a_3^{(2)})$$

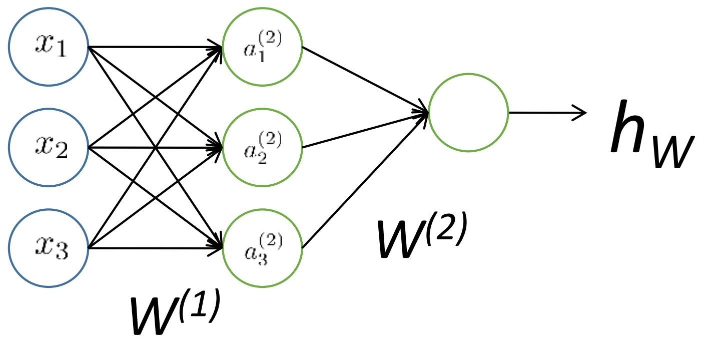
$a_i^{(j)}$ - “activation” of neuron *i* in layer *j*

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

$b^{(j)}$ – biases of the neurons in layer j

The same process would apply to further layers

Neural network – computing output - vectorized



$$z^{(2)} = xW^{(1)} + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)} + b^{(2)}$$
$$h_W = a^{(3)} = f(z^{(3)})$$

Output value

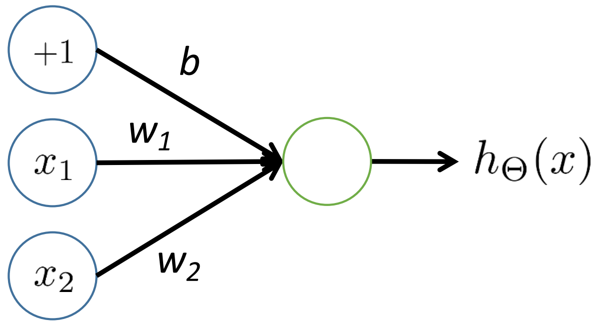
General version for layer i

$$z^{(i+1)} = a^{(i)}W^{(i)} + b^{(i)}$$
$$a^{(i+1)} = f(z^{(i+1)})$$

When processing several examples:
x becomes matrix **X**
a

$\boldsymbol{a}^{(i)}$ becomes a matrix **$\mathbf{A}^{(i)}$**
but the expressions are similar !

Computing the output - exercise



x_1	x_2	*Output*
0	0	
0	1	
1	0	
1	1	

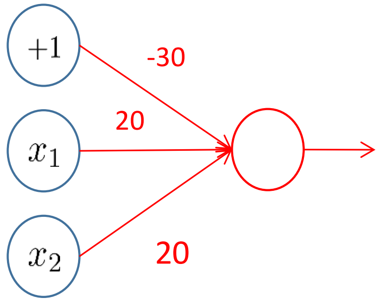
Calculate the output value for the cases where the weights are :

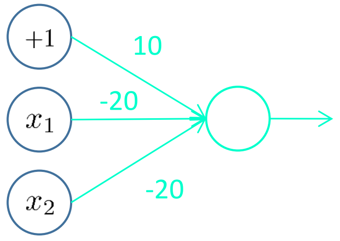
$b = -30$, $w_1 = 20$, $w_2 = 20$

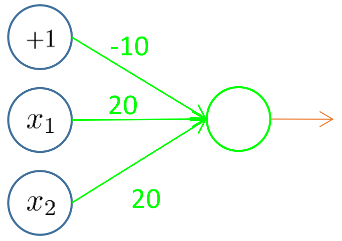
$b = -10$, $w_1 = 20$, $w_2 = 20$

$b = 10$, $w_1 = -20$, $w_2 = -20$

Computing the output - exercise





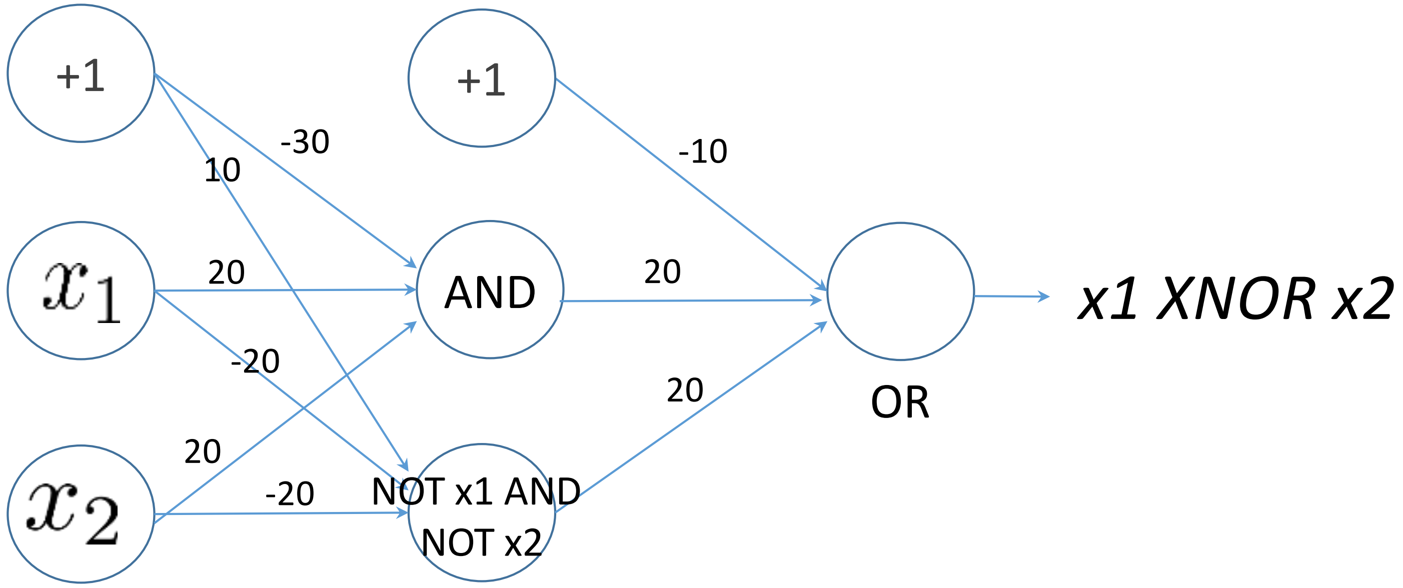


x_1 AND x_2 $\qquad$ (NOT x_1) AND (NOT x_2) $\qquad$ x_1 OR x_2

How to do H (x1, x2) = x1 XNOR x2 with a network with a 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

To implemente feedforward NNs, we will create a module that will include a few files with different componentes that will be complemented in the next session:

- *"neuralnet"* – it will include the main class to create a neural network that will "comunicate" with the other modules;
- "layers" – it will include the layers that will make the NNs (dense, etc);
- "activations" – it will include the activation layers (sigmoid, tahn, relu, etc);
- “data” – dataset (similar to previous ones)

These components will include some core implementations, but also make up a base for further developments

PYTHON IMPLEMENTATION

Python implementation - Layers

- We will first implement a base layer with some shared and **abstract** methods for all other specific layers.
- class Layer:
    - methods:
        - forward_propagation – abstract method that computes the output of a layer given the input;
        - output_shape – abstract method that returns the output shape of the layer;
        - parameters – abstract mehtod that returns the number of parameters of the layer;
        - set_input_shape – sets the input shape of the layer;
        - input_shape – returns the input_shape of the layer;
        - layer_name – returns the name of the layer;

PYTHON IMPLEMENTATION

Python implementation – Dense Layer

- A dense layer, also known as a fully connected layer, connects each neuron to every neuron in the previous and next layers, forming a dense network of connections.
- class DenseLayer(Layer):
  - arguments:
    - n_units – number of neurons in the NN;
    - input_shape – tuple with the input_shape (for tabular data it will be (n_features,))
  - estimated parameters:
    - input – the layer input;
    - output – the layer output;
    - weights – the layer weights;
    - biases – the layer biases;
  - methods:
    - initialize – initializes the layer with random weiights and biases and with the optimizer;
    - parameters – returns the number of parameters of the layer;
    - **forward_propagation - perform forward propagation on a given input;**
    - output_shape - returns the shape of the output of the layer.

Complete this function !

Python implementation – Activation layers

- class ActivationLayer(Layer):
    - methods:
        - forward_propagation - perform forward propagation on a given input by applying the activation function;
        - activation_function – abstract method that computes the activation function on the given input (used on the forward propagation);
        - output_shape - returns the shape of the output of the layer (which is the same as the input_shape);
        - parameters – returns the number of parameters of the layer (returns 0 as activation layers do not have learnable parameters).

PYTHON IMPLEMENTATION

Python implementation – Activation layers

- class ReLUActivation(ActivationLayer):
  - methods:
    - activation_function – applies the ReLU function on the input (argument)

- class SigmoidActivation(ActivationLayer);
  - methods:
    - **activation_function – applies the sigmoid function on the input (argument)**

Complete this function !

PYTHON IMPLEMENTATION

Python implementation – neural network

- Now that we have all the components we can build a NN (forward computation). The NeuralNetwork class will connect all the other classes together.
- class NeuralNetwork:
  - estimated parameters:
    - layers – the layers of the NN.
  - methods:
    - add – add a layer to the NN.
    - **forward_propagation – forward propagation of the full feedforward NN, going through all layers to compute final output for a set of examples**
    - predict – make predictions on new data (calls forward propagation);

Complete this function !

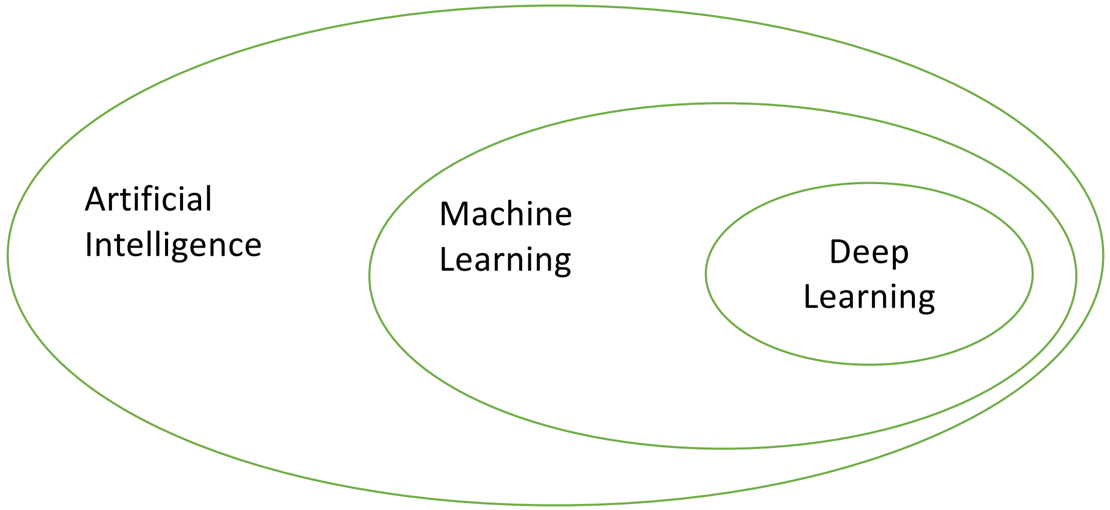
Complete the main to calculate the outputs for the 4 examples in the XNOR example !
Set the NN and its weights as shown in the previous example !!

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.

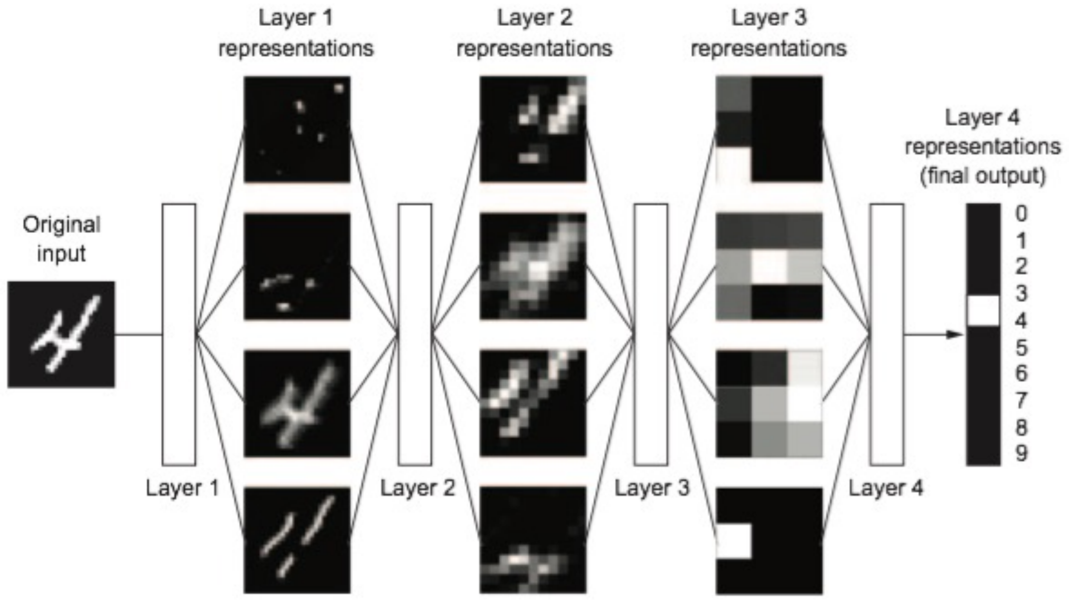
There are deep learning models for supervised and unsupervised learning problems, as well as reinforcement learning

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 (next session)



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"

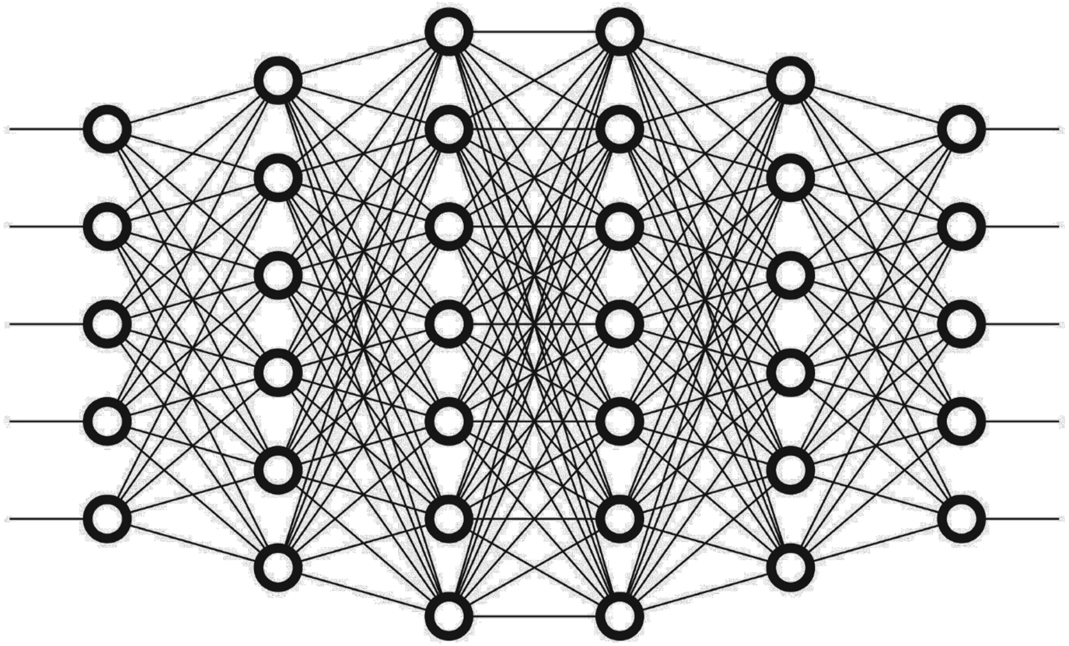
Areas of application

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



Deep Learning frameworks

Tensorflow from **Google**

- Multiple API levels - **Keras** official high-level API, user-friendly and enabling rapid prototyping. Defining a model with Keras often feels like describing a sequence or graph of layers. Common architectures very straightforward to implement
- TensorFlow 2.x adopted "eager execution" by default, making it behave more dynamically like Python code, similar to PyTorch.
- Good support for production deployment (TensorFlow Serving, TensorFlow Lite for mobile/embedded devices, TensorFlow.js for web), scalability across distributed systems
- Powerful visualization tools via TensorBoard
- Some installation problems are common

Deep Learning frameworks

PyTorch by **Meta AI**

- **API:** "Pythonic" feel. It integrates tightly with the Python language and its ecosystem (e.g., NumPy). Defining models and custom operations feels like writing standard object-oriented Python code.
- Primarily uses dynamic computation graphs (define-by-run) - built on-the-fly as the code executes. This offers greater flexibility, especially for models with dynamic structures (common in natural language processing) and makes debugging more straightforward.
- Widely adopted in the research community due to its flexibility and ease of use. Rapidly growing ecosystem and adoption in production environments.
- We will be the main framework for the examples in this course

Implementation in Python: pytorch

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

To install:

```
pip install torch
```

To run the examples in this session

```
pip install torchvision
```

XNOR dataset

Jupyter Notebook:
Xnor-pytorch.ypnnb

XNOR dataset

Dataset that we used before just with 4 exemples
Nonlinear dataset used to illustrate
Just training examples

```
X = torch.tensor([
    [0., 0.],
    [0., 1.],
    [1., 0.],
    [1., 1.]
])
y = torch.tensor([
    [1.],
    [0.],
    [0.],
    [1.]
])
```

Loading the data into Torch tensors

DNNs for XNOR

```
class XNORNet(nn.Module):
   def __init__(self, hidden = 2):
      super().__init__()
      self.net = nn.Sequential(
         nn.Linear(2, hidden),   # hidden layer - 4 nodes
         nn.Sigmoid(),      # nonlinearity (important!)
         nn.Linear(hidden, 1),   # output layer - 1 node
         nn.Sigmoid()       # output as probability
      )

   def forward(self, x):
      return self.net(x)

model = XNORNet(4)
```

Defining the model – Sequential class to create feedforward network

One hidden layer (with number of nodes as parameter)

Sigmoid activation function in hidden and output layers

DNNs for XNOR

Binary cross-entropy loss (Logistic regression)
Adam algorithm for training

```
# Loss and optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.05)
```

```
epochs = 100
for epoch in range(epochs):
    # forward
    y_pred = model(X)
    loss = criterion(y_pred, y)
    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
…
```

Training cycle
In each epoch:
first forward, then backward

```
with torch.no_grad():
    preds = model(X)
            …
```

Predicting for the training cases

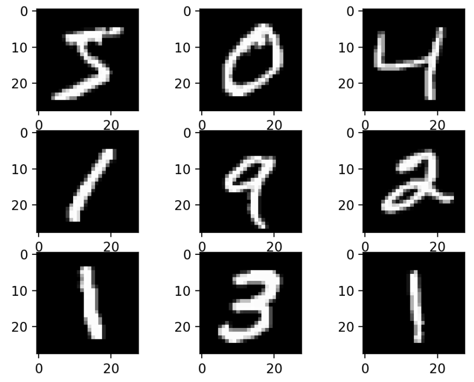
MNIST dataset

Jupyter Notebook:
torch-example-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

Loading the training and test data from a Torch dataset

```
# Transformations to apply to the data
transform = transforms.Compose([
    transforms.ToTensor() # Convert image to PyTorch Tensor
])

# Download and load the training data
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
                                                          transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Download and load the test data
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
                                                          transform=transform)
testloader = DataLoader(testset, batch_size=1000, shuffle=False, num_workers=2)
```

DNNs for MNIST

```
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 128) # Input layer -> Hidden layer 1
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)   # Hidden layer 1 -> Hidden layer 2
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, 10)    # Hidden layer 2 -> Output layer

    def forward(self, x):
        x = self.flatten(x)    # Flatten the image
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)        # Raw scores (logits)
        return x

model = SimpleMLP().to(device)
```

Defining the model and the topology of the NN creating the computation graph functionally

Flattening the data (images -> vector for each example)

2 hidden layers with 128 and 64 neurons

ReLU activation functions

Output layer with 10 neurons (number of classes)

DNNs for MNIST

```
criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Loss function – cross entropy (for multiclass)

```
num_epochs = 10

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
       inputs, labels = data[0].to(device), data[1].to(device)
       optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()
…
```

Training cycle similar to previous example

Boston housing dataset - regression

Jupyter Notebook:
boston-torch.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

Example shows how to load data from a scikit-learn dataset

Example of a regression problem

Check the notebook for details

DNN for Boston housing

```
class HousingNet(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_features, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)   # no sigmoid → regression output
        )

    def forward(self, x):
        return self.net(x)

model = HousingNet(X_train.shape[1])
```

Main difference in the model is the use of a Linear output without an activation function

Loss function: MSE

```
criterion = nn.MSELoss()
optimizer = optim.Adam(…)
```

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 (the data is provided in a CSV file preprocessed and loading is already implemented in the notebook); compare results with best SVM model provided