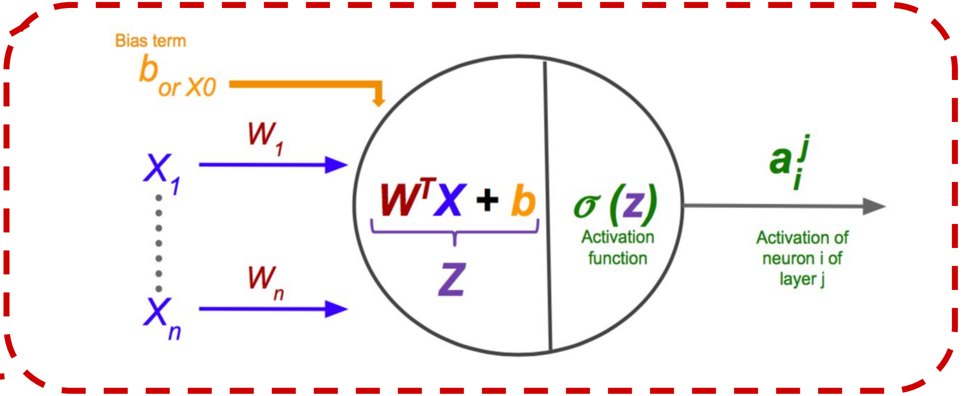
UNIVERSITÀ
DI PARMA
DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

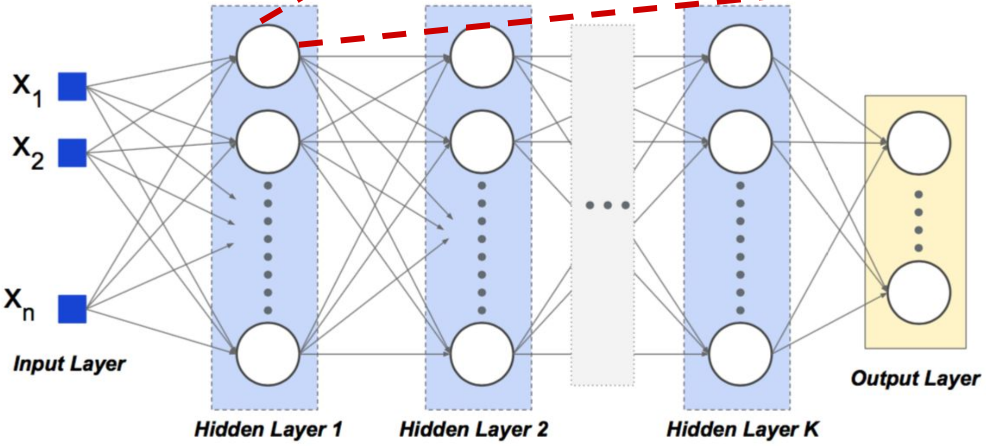
Neural Networks

Gianfranco Lombardo, Ph.D
gianfranco.lombardo@unipr.it

Artificial Neural Network (ANN)

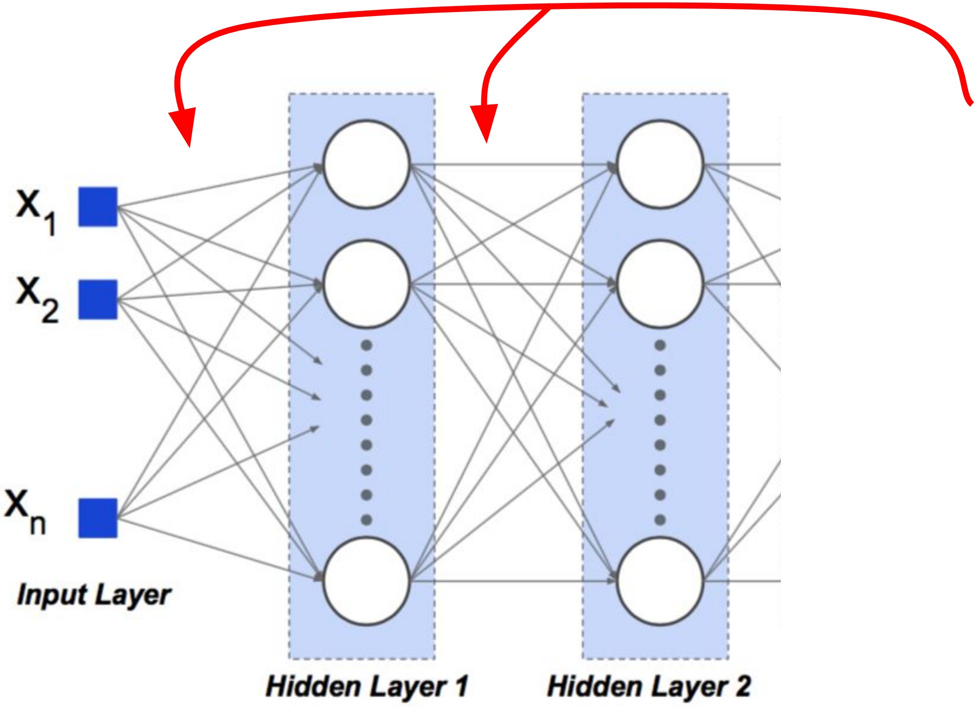
- Neurons in the same layer don’t have to communicate with each other





- Each neuron at layer j -1 is connected with all the neurons in the next layer j

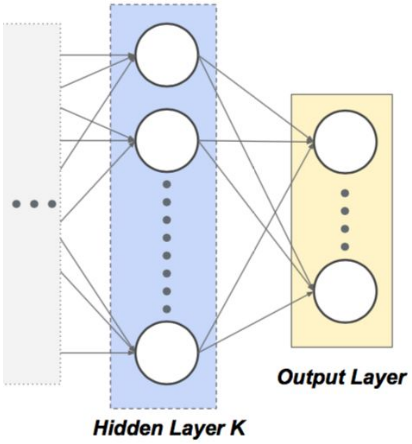
Artificial Neural Network (ANN)



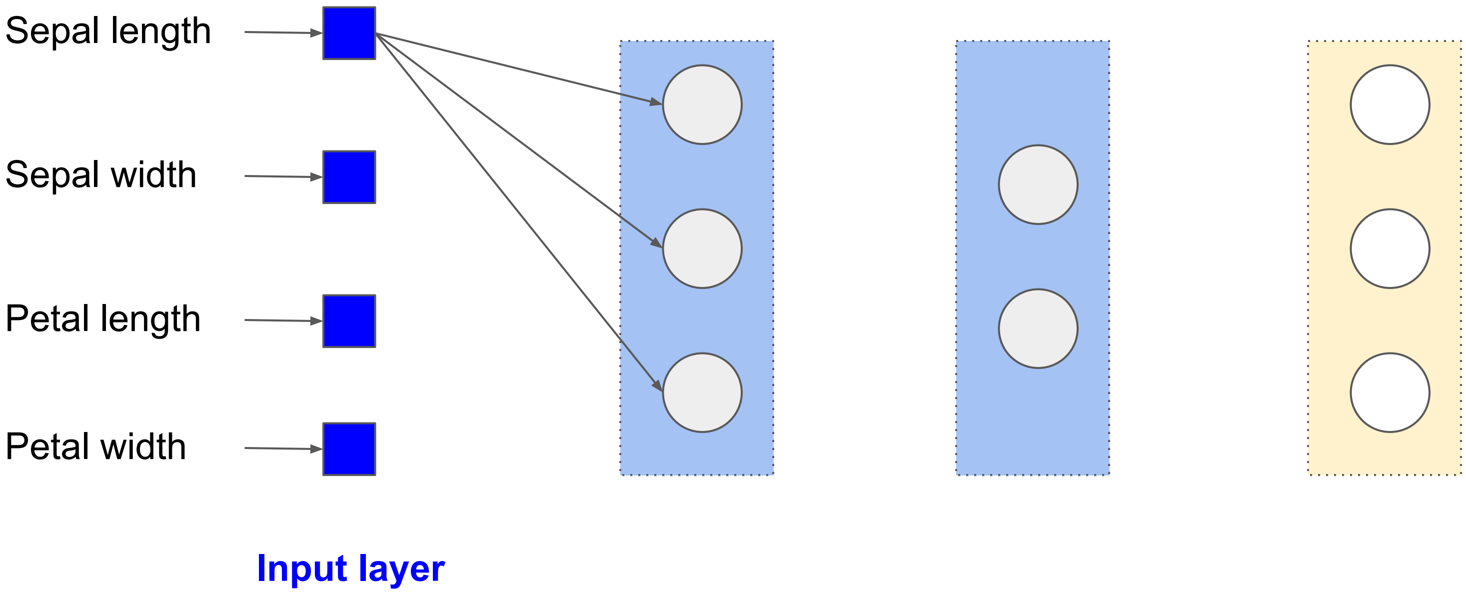
- Each connection between input and neurons, and between neurons and neurons has an associated weight **w**
- Weights are randomly initialized
- ***Our goal during the training step is to learn these weights***
- All weights between two layers are organized in a matrix ***W***

The output layer

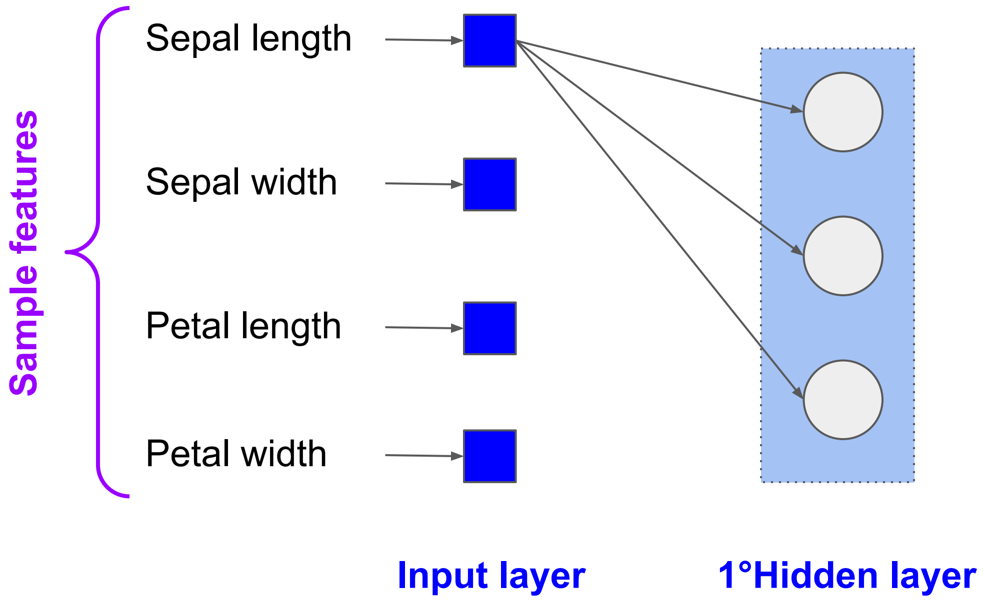
- We should say something more about the output layer.
- The structure of this last layer depends on the task you want to perform.
- y in our dataset has to be formatted depending on this layer
- Usually the number of neurons is equal to the number of expected possible outputs, but we should pay attention:
  - **Regression**: For example price prediction of a house, we need only one neuron with an activation function that is able to produce the value we need (for example a linear function)
  - **Binary classification**: We can have one neuron with a sigmoid activation function. <u>But it is not the only possibility!</u>
  - **Multi-class or multi-label classification :** We will have a number of neurons equal to the number of the possible classes. Each neuron is like a binary classifier.



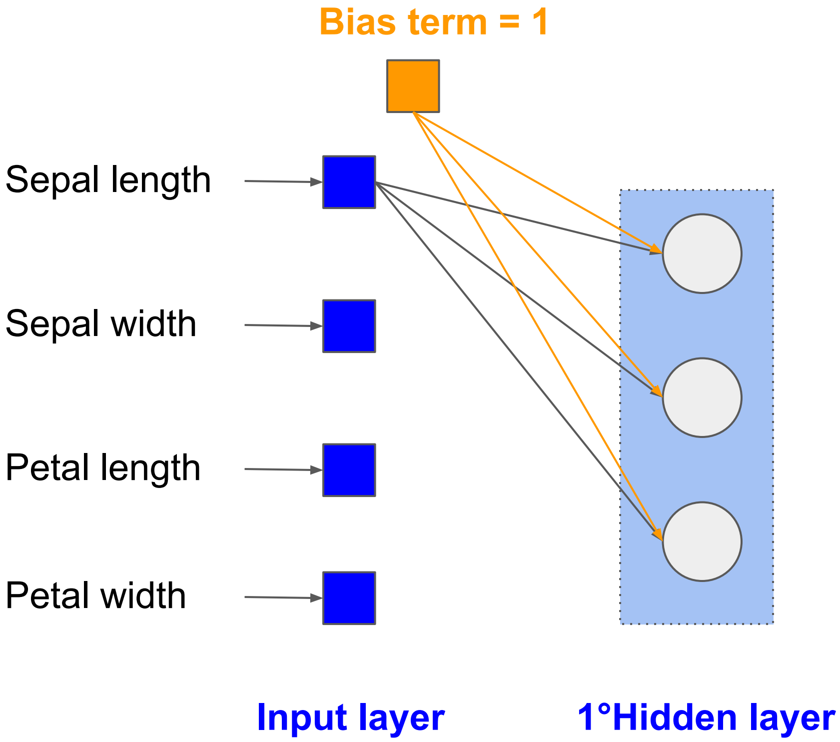
Some examples: Iris classification



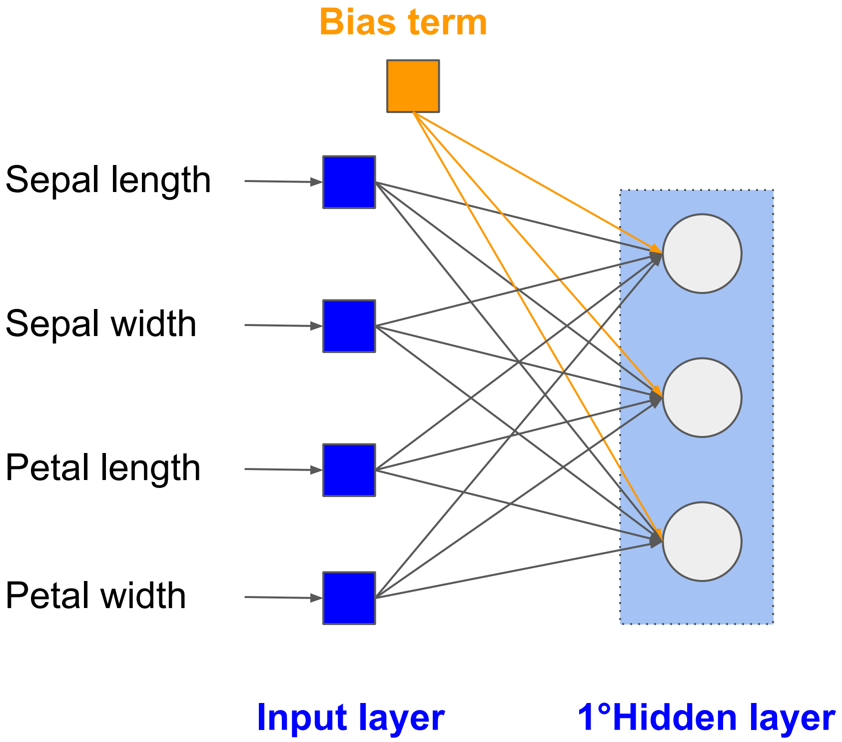
Some examples: Iris classification



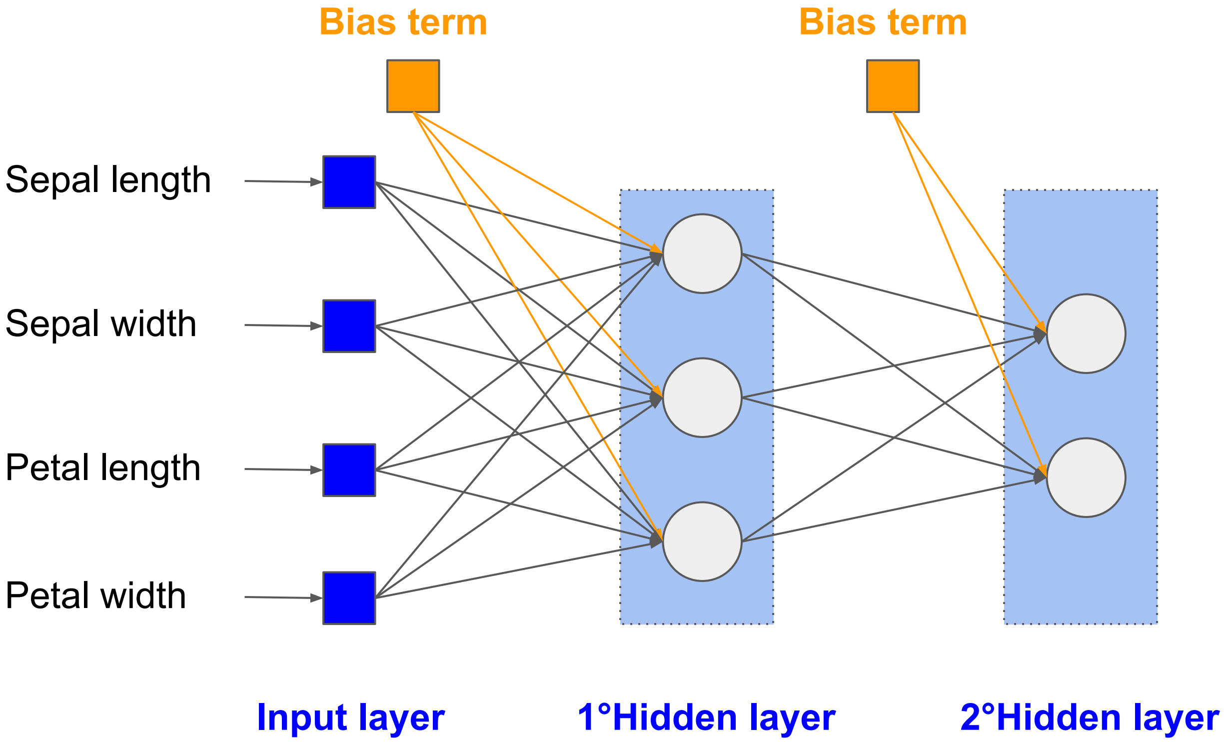
Some examples: Iris classification



Some examples: Iris classification

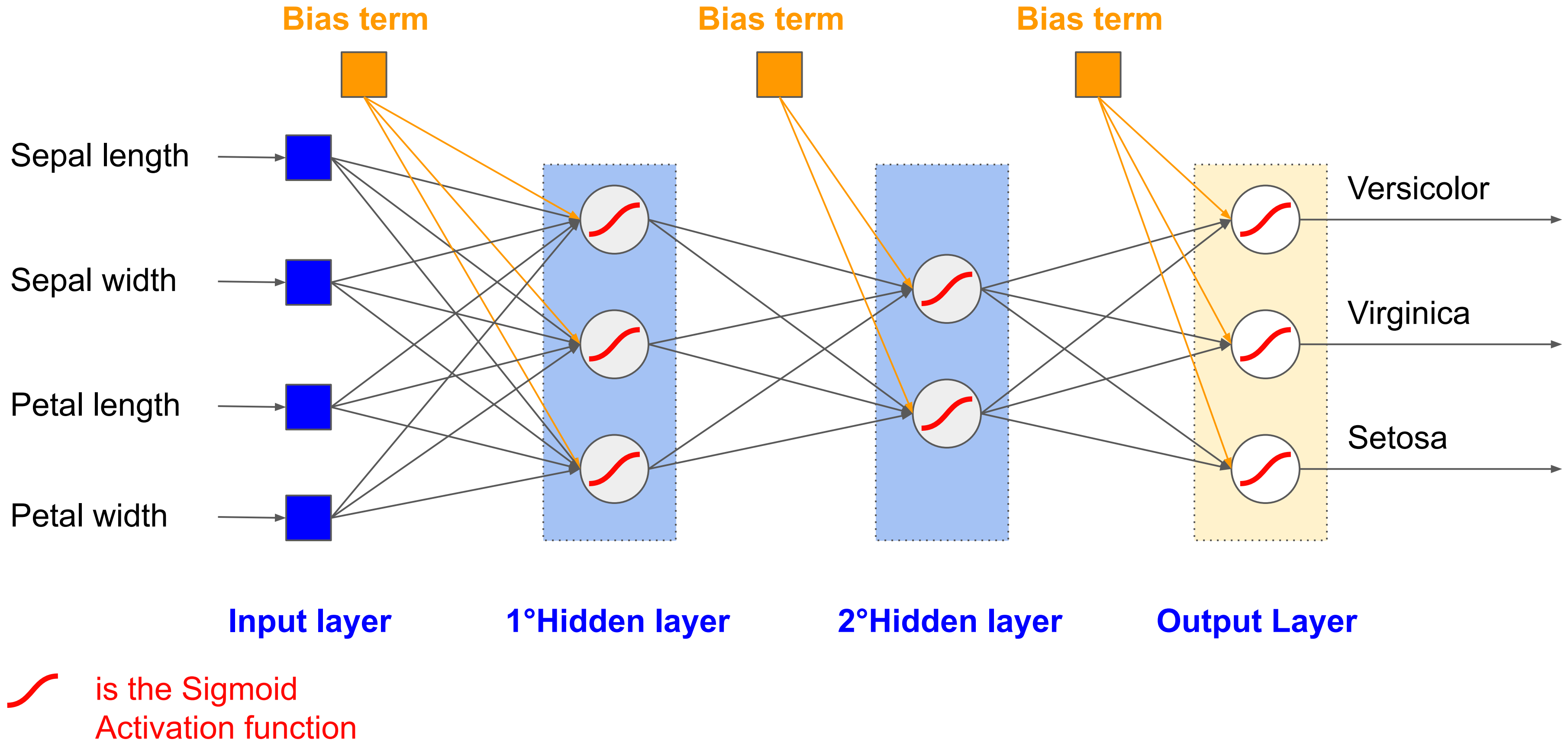


Some examples: Iris classification

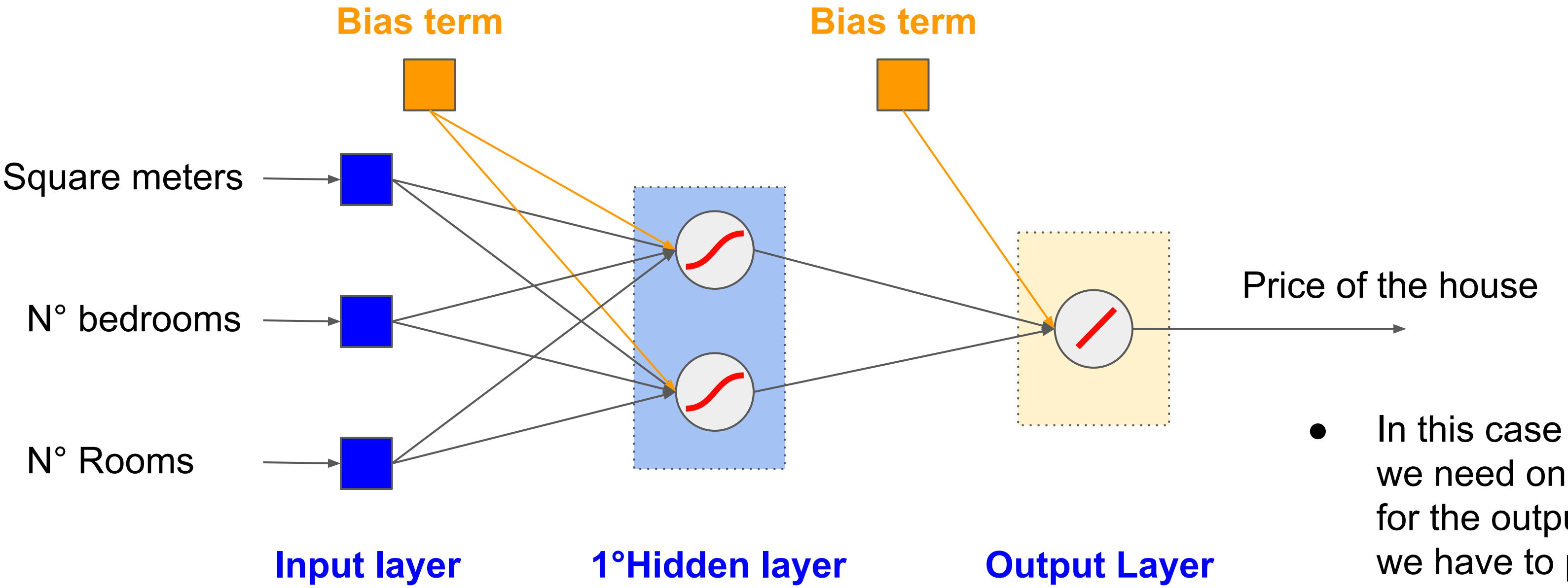


- Now we want to add the output layer to make predictions
- How many neurons we need ?

Some examples: Iris classification



Another example: House price prediction



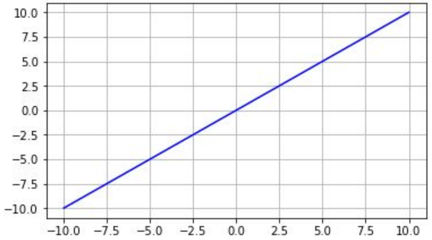
- In this case of regression we need only one neuron for the output layer since we have to predict only one value
- We cannot use the Sigmoid activation function since the value we are predicting is not between 0-1 but can be any value


is the Sigmoid Activation function


is the Linear Activation function

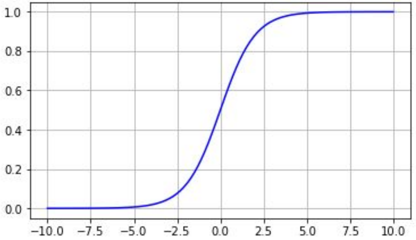
Activation functions in Neural networks

Linear Activation



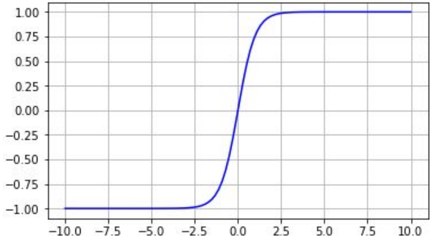
f(u) = u

Sigmoid Activation



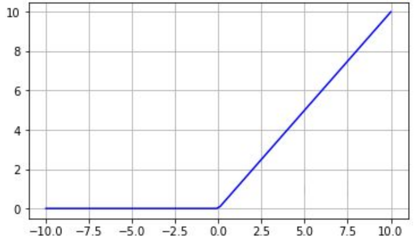
$$f(u) = \frac{1}{1+e^{-u}}$$

Tanh Activation



$$f(u) = \frac{e^{u}-e^{-u}}{e^{u}+e^{-u}}$$

ReLU Activation



f(u) = max(0,u)

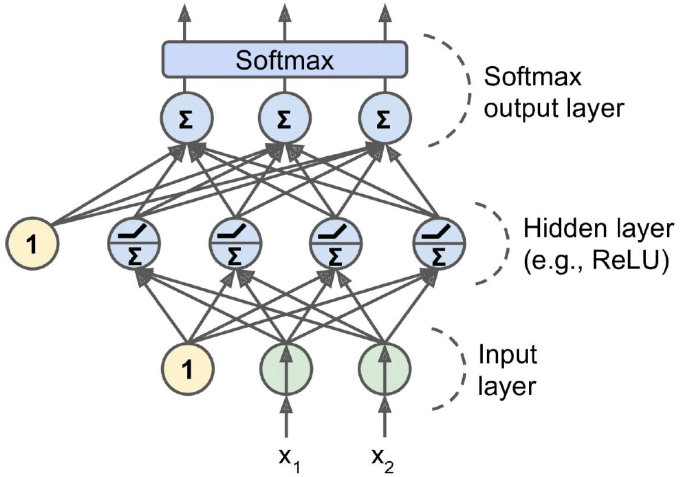
The Softmax output layer: Only for Classification!

- When the classes are exclusive (class 0 is a dog, class 1 is a cat, class 2 is a mouse) and the problem is not multi-label (to each sample we assign one and only one label, another type of neuron is used

- We can replace individual activation functions with a shared Softmax function

- The output of each neuron corresponds to the estimated probability of the corresponding class

- The Softmax function is a generalization of Logistic Regression in order to support multiple classes directly without having to train and combine multiple binary classifiers



The Softmax output layer

- The idea is the following:
  - Given an instance x or its representation in the penultimate layer of a neural network
  - The Softmax model compute a score $\mathbf{s_k(x)}$ for each class **k**
  - Then it estimates the probability of each class by applying the softmax function to these scores

- $\mathbf{s_k(x)} = W^{(k)\,T}\,X$ (Remember logistic for binary classification?)
  - Each class has its own dedicated parameter vector $W^{(k)}$

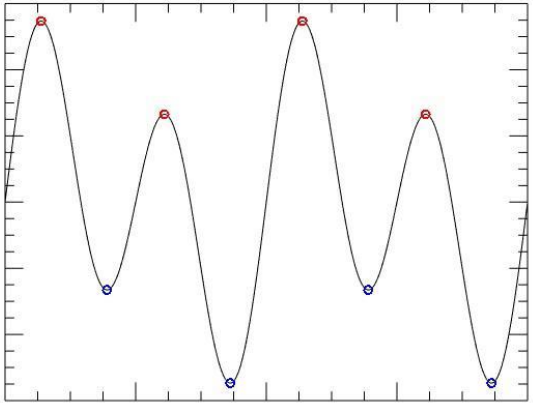
- Now we can compute the probability p_k that the instance belongs to class k

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- It predicts the class with the highest probability (class with the highest score)

Neural networks: Optimization problem

- Our goal is to **minimize** an **objective function**, which measures the difference between the actual output t and the predicted output y.
  - In this case we will consider as the objective function the **squared loss function**.



Squared loss function

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - f(\mathbf{w}^{\mathbf{T}}\mathbf{x}))^2$$

Loss functions

Squared loss function

$$E = \frac{1}{2}(t - y)^2$$

Mean squared error

$$E = \frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)^2$$

Mean absolute error

$$E = \frac{\sum_{i=1}^{n} | t_i - y_i |}{n}$$

Cross entropy

$$E = -\sum_{i=1}^{n} t_i \log(y_i)$$

Kullback Leibler divergence

$$E = \sum_{i=1}^{n} t_i \log\left(\frac{t_i}{y_i}\right)$$

Cosine proximity

$$E = \frac{t \ y}{||t|| \ ||y||}$$

Loss functions

- **For Regression task**
  - Mean Squared Error Loss
  - Mean Absolute Error Loss

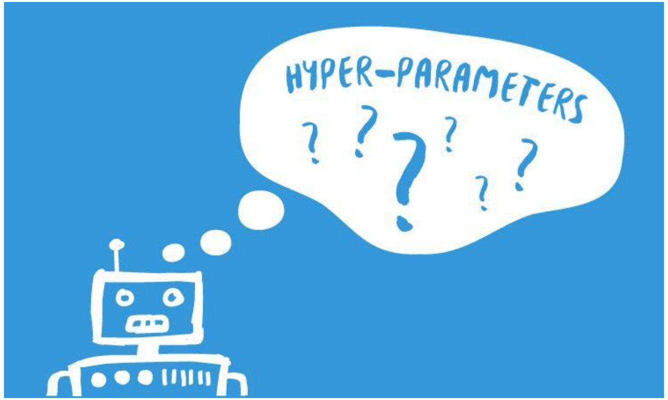
- **For Binary Classification**
  - Binary Cross-Entropy

- **For Multi-Class Classification**
  - Multi-Class Cross-Entropy Loss
  - Kullback Leibler Divergence Loss

- **<u>Note: This list is not exhaustive</u>**

Hyper-parameters

- Hyperparameters are the parameters which determine the **network structure** (e.g. Number of Hidden Units) and the parameters which determine **how** the **network** is **trained** (e.g. Learning Rate)
  - Number of neurons
  - Number of layers
  - Learning rate
  - Batch size
  - Number of epochs
  - others

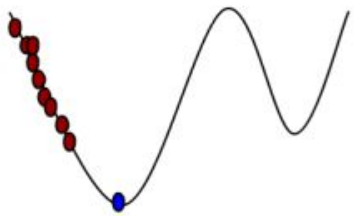


Learning rate

- Training parameter that controls the size of weight changes in the learning phase of the training algorithm.
- The learning rate determines how much an updating step influences the current value of the weights.

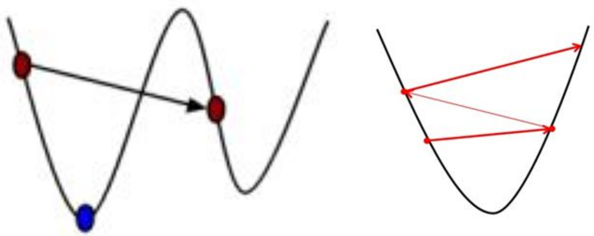
$$w_i^{new} = w_i^{old} - \eta \frac{\partial E}{\partial w_i}$$

Very small learning rate



Many updates required before reaching the minimum.

Too big learning rate



Drastic updates can lead to divergent behaviors, missing the minimum.

Hyper-parameters

Number of epochs

- The number of epochs is the number of times the whole training data is shown to the network while training.

- Remember that at the beginning weights are randomly initialized. Our training is sensitive to this initialization

Batch size

- The number of samples shown to the network before the gradient computation and the parameter update.

Stochastic Gradient Descent

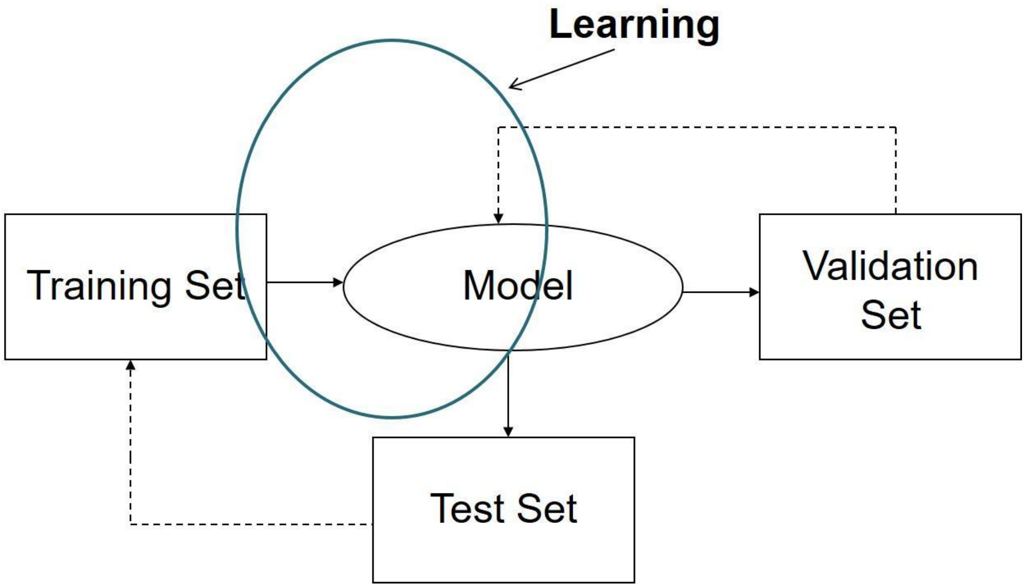
Gradient Descent

Batch Gradient Descent

Validation set

- Data set with the 'same' goal of the test set (verifying the quality of the model which has been learnt), but not as a final evaluation, but as a way to fine-tune the model.

- Its aim is to provide a feedback which allows one to find the best settings for the learning algorithm (parameter tuning).

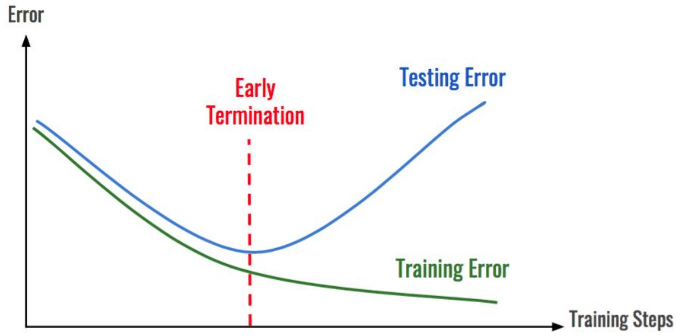


Early stopping

- Early stopping is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent

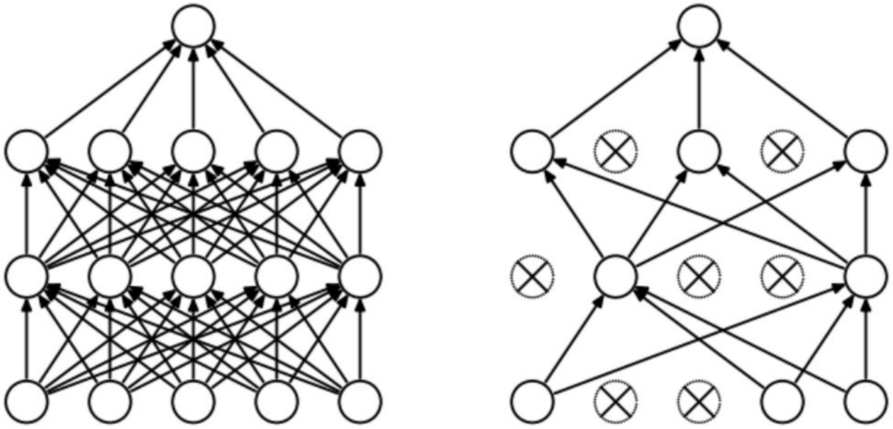
- Stop training as soon as the error on the validation set is higher than it was the last time it was checked
    - We can define a patient parameters: We accept that a patient number of times the validation error can be higher than the previous iteration. After this number is reached, training will be stopped.

- Use the weights the network had in that previous step as the result of the training run



Dropout

- It is another form of regularization for Neural Networks
- At each update during training time, randomly setting a fraction rate of input units to 0.
- It helps to prevent overfitting.



(a) Standard Neural Net (b) After applying dropout.

Choosing the Loss function and the best optimizer

- To minimize the loss we don't have only Gradient Descent or Stochastic Gradient Descent (SGD).

- Other gradient-based optimizers are available, in particular in Keras such as:

    - RMSprop

    - Adam

- It is important to deeply understand the problem we are dealing with when we have to choose the loss function and the best optimizer for our task

Keras

- Keras is an open-source library that provides tools to develop artificial neural networks

- Keras acts as an interface for the TensorFlow library

- First install TensorFlow: pip install tensorflow

- Then pip install Keras (Optional with the latest version of Tensorflow)

Example: breast cancer classification

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
```

Example: breast cancer classification

```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.20)

print("Training set dimensions (train_data):")
print(X_train.shape)
```

Example: breast cancer classification

```
model = models.Sequential()
#The first layer that you define is the input layer. This layer needs to know the input dimensions of your data.
# Dense = fully connected layer (each neuron is fully connected to all neurons in the previous layer)
model.add(layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
# Add one hidden layer (after the first layer, you don't need to specify the size of the input anymore)
model.add(layers.Dense(64, activation='relu'))
# If you don't specify anything, no activation is applied (ie. "linear" activation: a(x) = x)
model.add(layers.Dense(1,activation='sigmoid'))
```

Example: breast cancer classification

```
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=[tf.keras.metrics.Precision()])

# Fit the model to the training data and record events into a
History object.
history = model.fit(X_train, y_train, epochs=10, batch_size=1,
validation_split=0.2, verbose=1)

# Model evaluation
test_loss,test_pr = model.evaluate(X_test,y_test)
print(test_pr)
```

Breast cancer: Plot Loss VS Epochs

```
# Plot loss (y axis) and epochs (x axis) for training set and
validation set
plt.figure()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.plot(history.epoch,
np.array(history.history['loss']),label='Train loss')
plt.plot(history.epoch,
np.array(history.history['val_loss']),label = 'Val loss')
plt.legend()
plt.show()
```

Setting learning rate and optimizer

```
opt = keras.optimizers.Adam(lr=0.01)
model.compile(loss='binary_crossentropy',optimizer=opt, metrics=[...])
```

https://keras.io/api/optimizers/adam/

Available optimizers:
https://keras.io/api/optimizers/

Modifications to use Softmax (Suggested for Multi-class problems)

```
from tensorflow.keras.utils import to_categorical
y = to_categorical(y)
```

```
…
…
…
```

```
model.add(layers.Dense(2,activation='softmax'))
```

- Now our y is a matrix with a number of columns equal to the number of possible classes
- The column value is equal to 0 or 1 depending on the class associated to that example (row)

Example: Boston regression

```
from sklearn.datasets import load_boston
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.keras import models
from tensorflow.keras import layers
```

Example: Boston regression

```
X, y = load_boston(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.20)

print("Training set dimensions (train_data):")
print(X_train.shape)
```

Example: Boston regression

```
model = models.Sequential()
model.add(layers.Dense(64,
activation='relu',input_shape=(X_train.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1,activation='relu'))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mse'])

history = model.fit(X_train, y_train, epochs=10, batch_size=1,
validation_split=0.2, verbose=1)
test_loss_score, test_mse_score = model.evaluate(test_data,
test_targets)

# MSE
print(test_mse_score)
```

Possible metrics

https://keras.io/api/metrics/

Dropout, early stopping and validation data

- Dropout is a simple layer you should add
  - model.add(layers.Dropout(VALUE between 0 and 1))

- Early stopping is a callback

```
from keras.callbacks import EarlyStopping (for new version try tf.keras)
es = EarlyStopping(monitor='val_loss',mode='min', verbose=1, patience= 10)
….
model.fit(X_train, Y_train,epochs=300,validation_data=(X_val, Y_val),callbacks=[es])
```

UNIVERSITÀ DI PARMA

DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Lab

Gianfranco Lombardo, Ph.D
gianfranco.lombardo@unipr.it

Exercise

- Starting from the breast cancer example try to report how precision and recall changes:
  - Increasing number of epochs
  - Increasing the batch size
  - Modify the network architecture: try to add more hidden layers and to build a funnel structure
    - First hidden layer should have 75% of input layer
    - Second: 50%
    - Third: 25%
    - Fourth: 12.5%
    - Five: 6% and so on

Cars sale prediction

- **Cars dataset**: the task is to predict the value of a potential car sale (i.e. how much a particular person will spend on buying a car) for a customer on the basis of the following attributes: age, gender, average miles driven per day, personal debt, monthly income
- Create a sequential model (loss='mse', optimizer='rmsprop', metrics=['mse'], epochs=150, batch_size=50) adding:
  - A (“Dense”) layer with some nodes (input_dim=5, activation=‘relu’)
  - A hidden (“Dense”) layer with other nodes (activation=‘relu’)
  - An output (“Dense”) layer composed of 1 node
- Plot the mean squared error over the epochs for the training set and for the validation set (validation_spilt=0.2)
- Compute the root mean squared error on the test set

Sonar classification

- **Sonar dataset**: the task is to train a network to discriminate between sonar signals (60 features) bounced off a metal cylinder (class M) and those bounced off a roughly cylindrical rock (class R).
- Create a sequential model (loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'], epochs=100, batch_size=5) adding:
  - An input (“Dense”) layer composed of 60 nodes (input_dim=60, activation=‘relu’)
  - An output (“Dense”) layer composed of 1 node (activation='sigmoid‘)
- Make predictions for the labels of the test set and evaluate the model