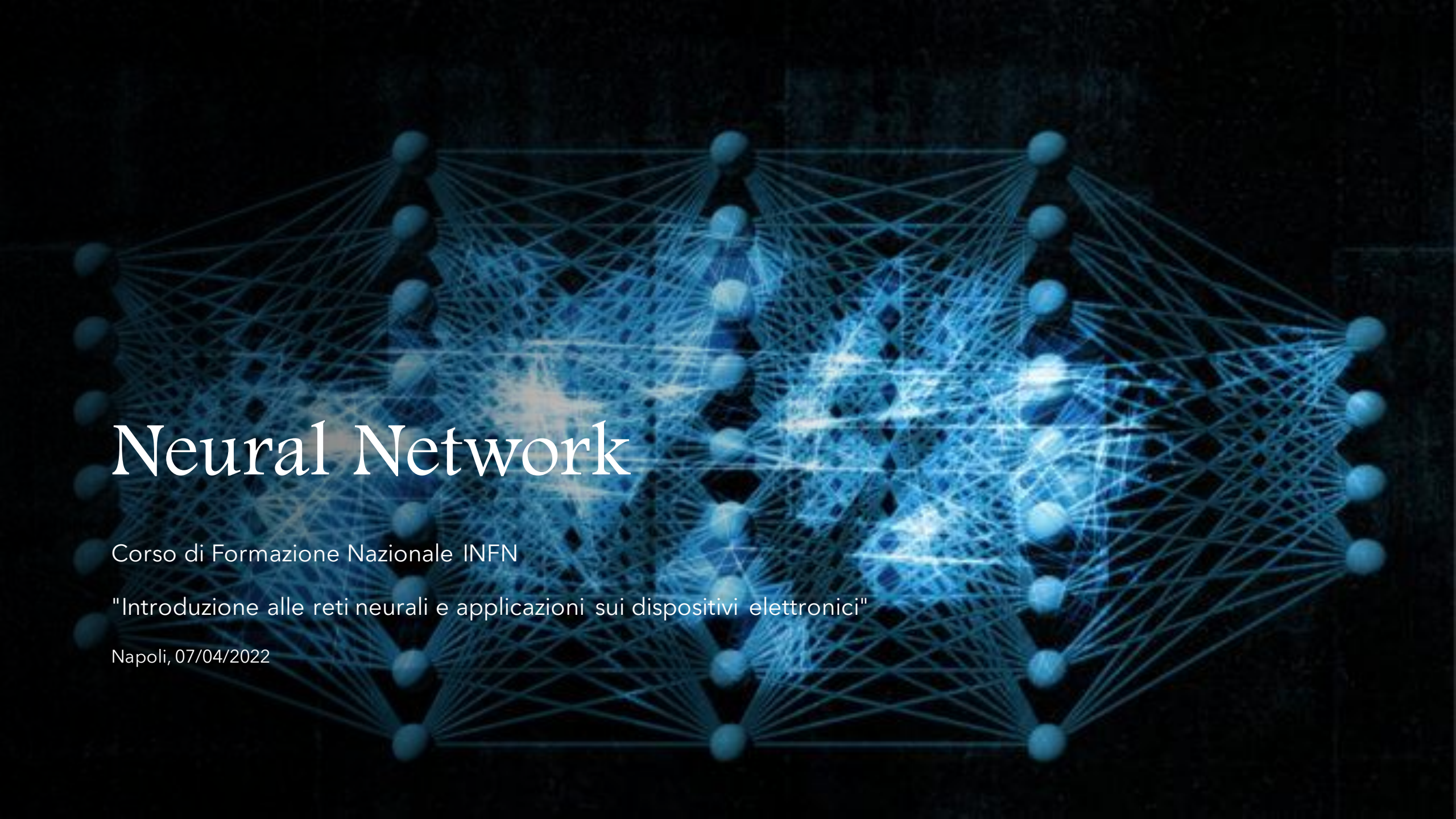


Neural Network

Corso di Formazione Nazionale INFN

"Introduzione alle reti neurali e applicazioni sui dispositivi elettronici"

Napoli, 07/04/2022

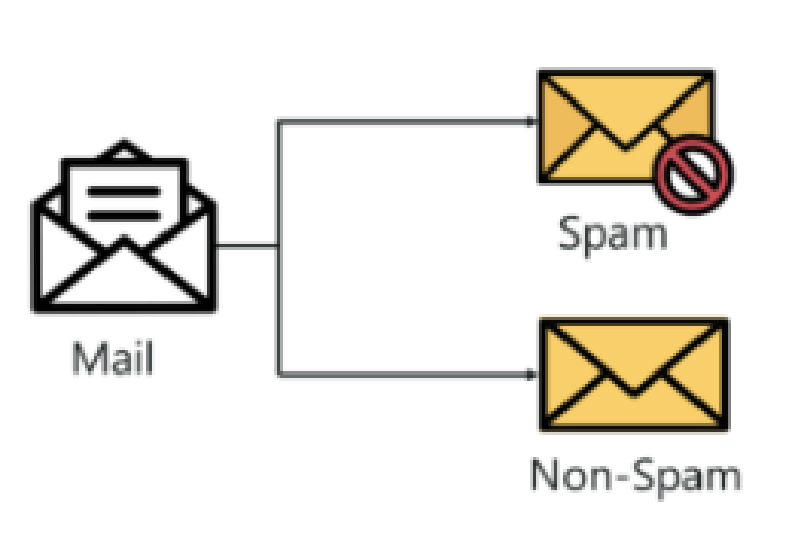
Contents

Machine Learning: classification

- In these slides we want to introduce a powerful classification algorithm: the **Neural Network**
- The simplest version of a Neural Network has already seen in the last lesson: the **Perceptron**
- It represents the building block of a NN in a way that we will explain below in the lesson
- Our goal is always the discrimination of different classes, we want to obtain a model that will be able to discriminate between 2 classes or more through a training done on a specific dataset.
- The algorithm, during the train, adapts several number of parameters (we decide how many params) to discriminate the classes defined by us. It is quite similar to perceptron algorithm but this time we will have a more complicated structure of parameters.

Classification Problem

- There are a lot of examples in the real life in which we want to automatize a classification problem:
  - if an e-mail is spam or not,
  - if a recorded event is a signal or background
  - if we want to develop an archive that is capable of discriminate objects on its own (for example using as input a picture)



Classification problem

- Our first step is the selection of our dataset, we need of a set of features that describes what we want to discriminate (for example if we want to discriminate electronic devices we can use the physical sizes of the object, the weight, the cost, ..)
- Next step, we must define the classes and so the targets (in the previous example: mobile phone, PC, tablet,..)

$$X = \begin{pmatrix} x_{0.0} & \cdots & x_{0,n} \\ \vdots & \ddots & \vdots \\ x_{n,0} & \cdots & x_{n,n} \end{pmatrix}$$

$$Y = \begin{matrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{matrix}$$

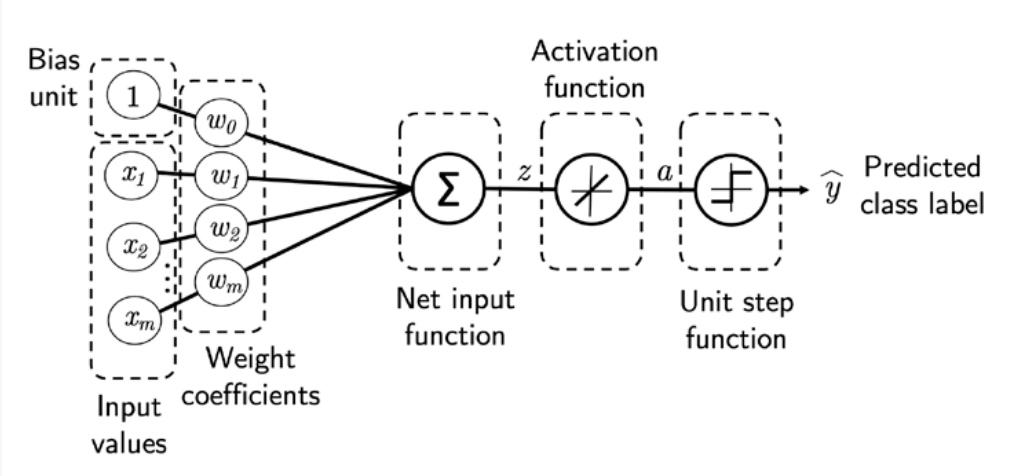
Classification problem

- In the major part of our problems, we start from the X,Y information. We already have at our disposal the features of what we want to discriminate and the classes names
- For our algorithm we need to transform the Y information in a numeric information to insert that in the **loss function**.
- The simplest way to do that is to ordered our classes from 0 to N, and N is the number of classes
- This is a good method but we must avoid to use it, because it add an intrinsic order between our classes: the algorithm could prefer the class with the high number due to how we define the **loss function**.
- The most used method is to create **dummies variables**:

$$
\begin{aligned}
class\ 1 &= \{1,0,0,\dots,0\} \\
class2 &= \{0,1,0,\dots,0\} \\
&\dots \\
classN &= \{0,0,0,\dots,1\}
\end{aligned}
$$

Perceptron

- As seen in the last lesson the simplest algorithm able to discriminate between two classes (binary classificator) is the **perceptron**.
- The perceptron is the starting point to build a Neural Network, but we will need a lot of them!
- As a reminder: in a single perceptron we have m+1 weights (where m is the number of input variables), an activation function, and an output that is a linear function of inputs and weights



From linear to nonlinear

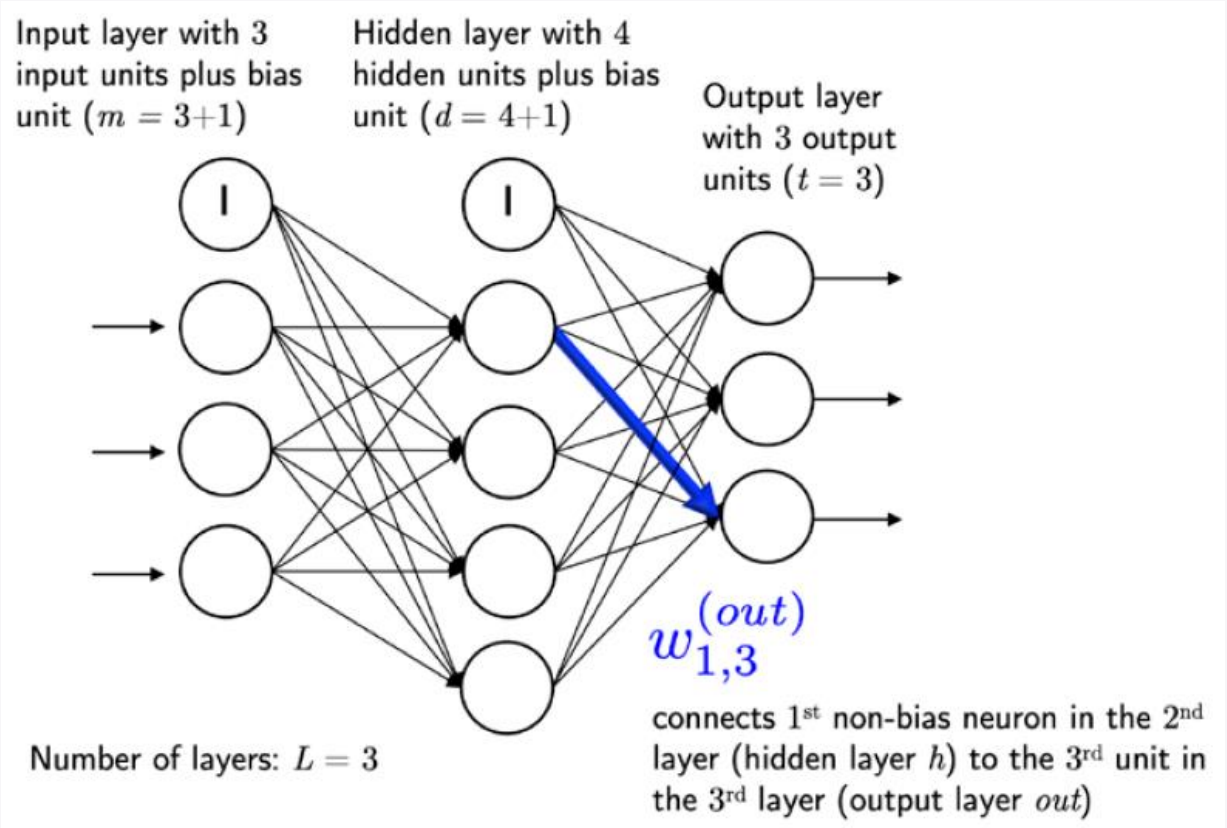
The linearity approch seen in the perceptron implies the wekear assumption of monotonicity: that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative).

This approach has an obviously limit: say for example that we want to predict probability of death based on body temperature. For individuals with a body temperature above 37°C , higher temperatures indicate greater risk. However, for individuals with body temperatures below 37° C, higher temperatures indicate lower risk! In this case, we might resolve the problem with some clever preprocessing. Namely, we might use the distance from 37°C as our feature.

We can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers between input and output. Moreover we can introduce non linearity also in the activation function (as we see soon). In this way the output became a complicated function of the input variables and so we lose the linearity of our model.

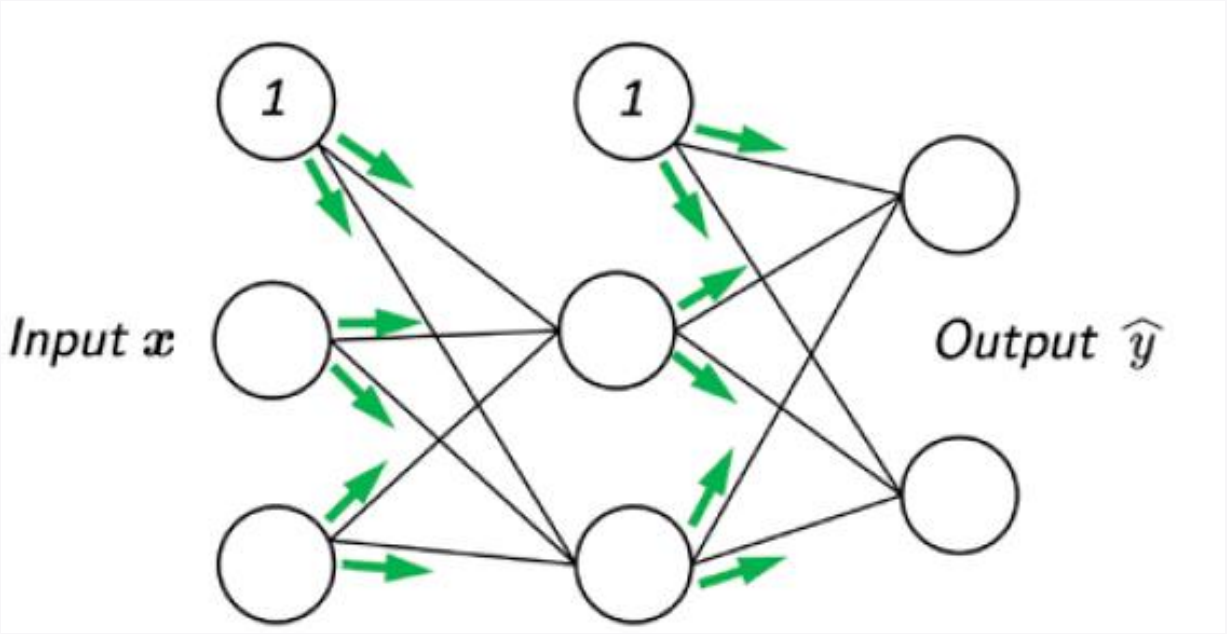
Multilayer perceptron

- The first example of a NN is the Multilayer perceptron, this is a net of fully connected perceptron
- In the schematical view on the right every circle is a perceptron with a fixed number of inputs and outputs
- In the example we have an **input layer,** only one **hidden layer** and an **output layer**



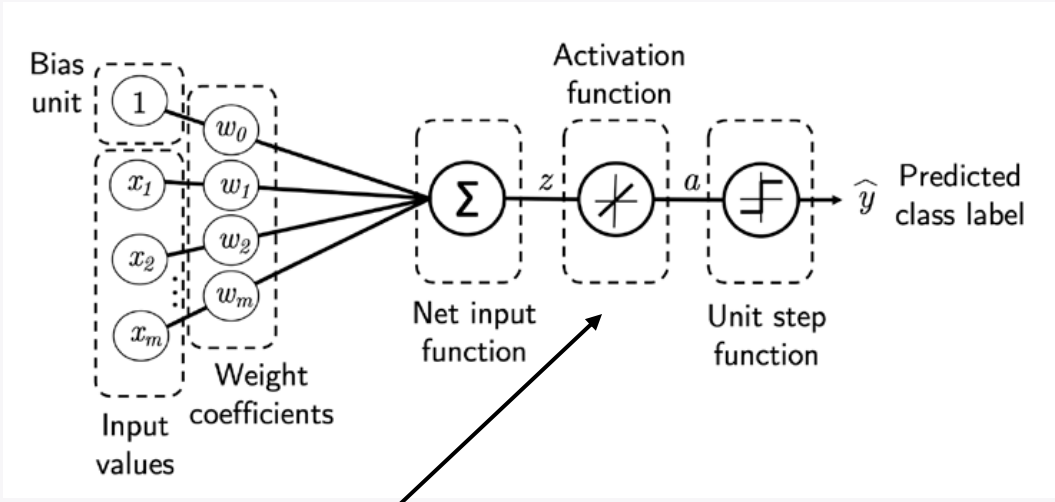
Multilayer perceptron

- For every line we have a weight.
- In this case where we have 2 input variables, an hidden layer with size 2, and 2 outputs, so:
  - (#neurons in the next layer * #variables) + #neurons in the next layer= 6 weights (between input layer and hidden layer)
  - #neurons in the next layer* hidden layer size+ #neurons in the next layer = 6 weights (between hidden layer and output layer)
  - In total 12 weights



Activation Functions

- The activation function provides to the activation or not of a node.
- The functions are in general differentiable operators to transform the inputs to outputs
- Most of them provides to add non-linearity to the model
- The activation function **σ** has as input the weighted sum of the input variables x, added with the bias b

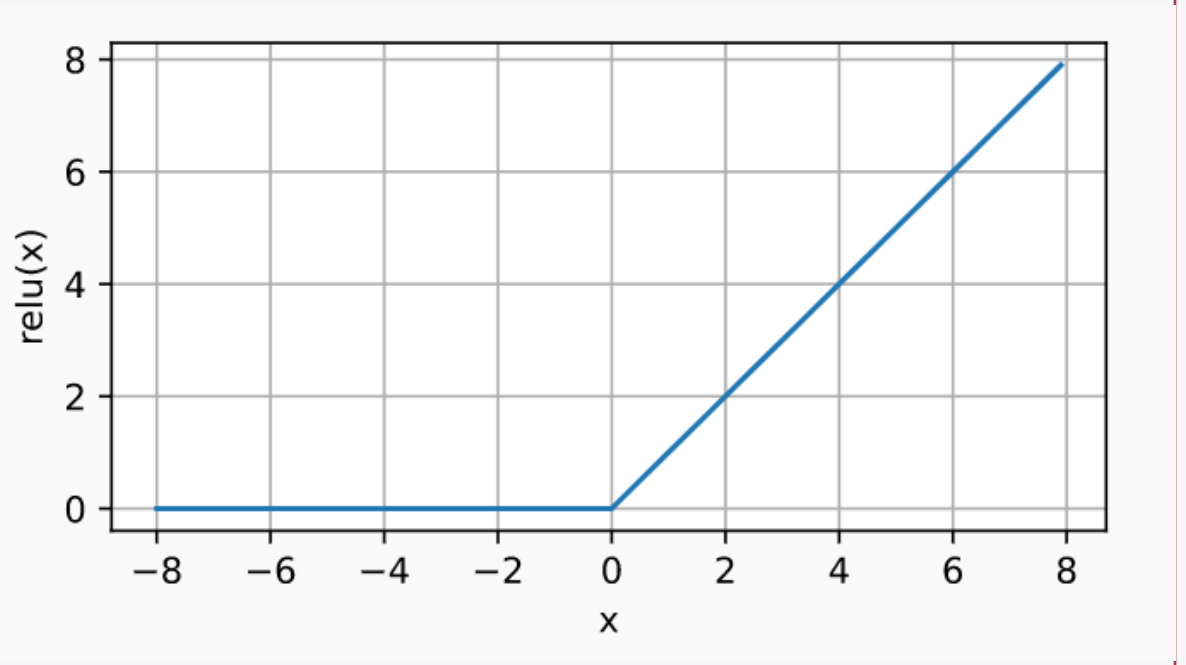


$$\sigma = \sigma\left(\vec{w} \cdot \vec{x} + b\right)$$

Rectified Linear Unit (ReLU)

- One the most popular non linear activation function is the rectified linear unit (ReLU).
- It provides a non linear transformation and returns the max value between the input and 0.

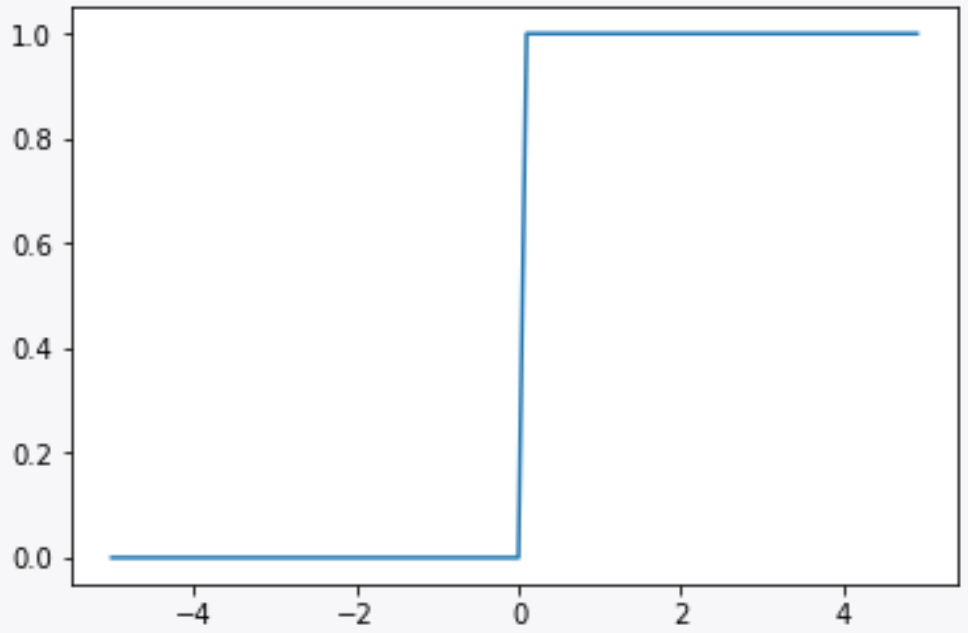
$$ReLU(x) = max(0, x)$$



Rectified Linear Unit (ReLU)

- The ReLU function is also differentiable in R-{0} and its derivative is the Heaviside function.
- In case the input is equal to zero, it is used the left-side derivative.

$$\frac{dReLU(x)}{dx} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

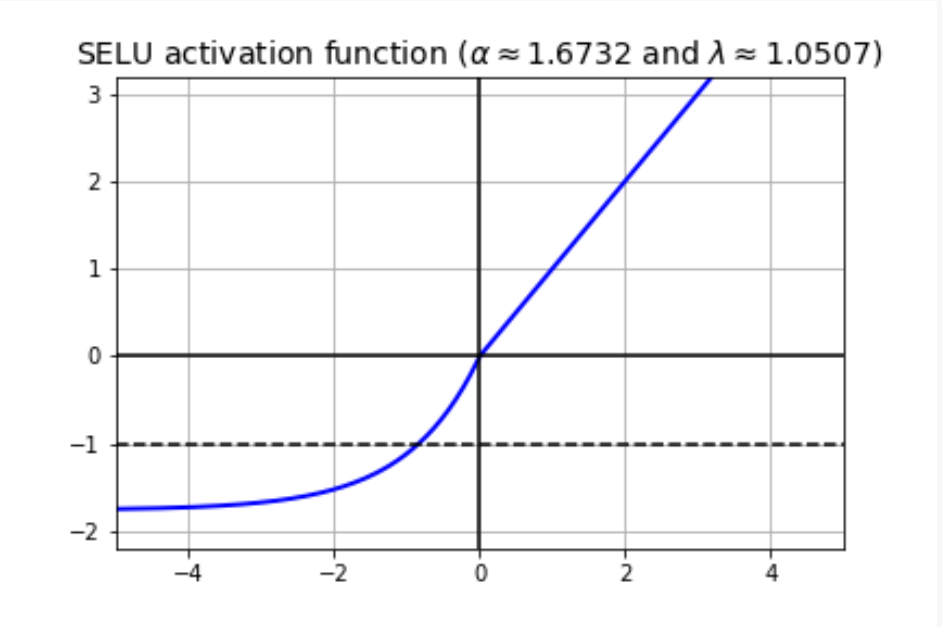


Scaled Exponential Linear Unit (SELU)

- Another possible choice is the Scaled Exponential Linear Unit (SELU).

- The functions depends by two parameters and the equation is the following one:

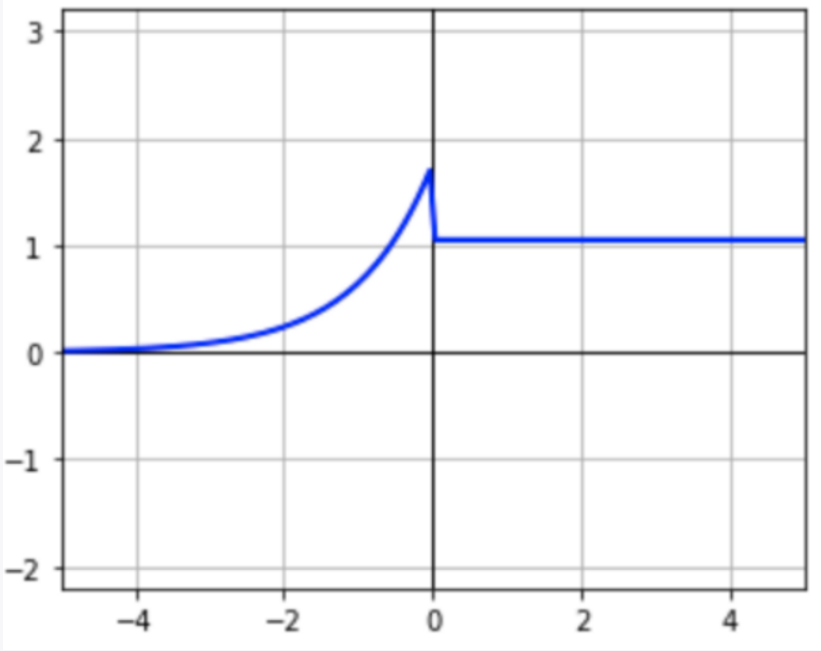
$$SELU(x) = \lambda \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$$



Scaled Exponential Linear Unit (SELU)

- The function is not differentiable in zero.
- Also here is convention to use the left-side value of its derivative.

$$\frac{dSELU(x)}{dx} = \lambda \begin{cases} \alpha e^x & x \leq 0 \\ 1 & x > 0 \end{cases}$$



Sigmoid Function

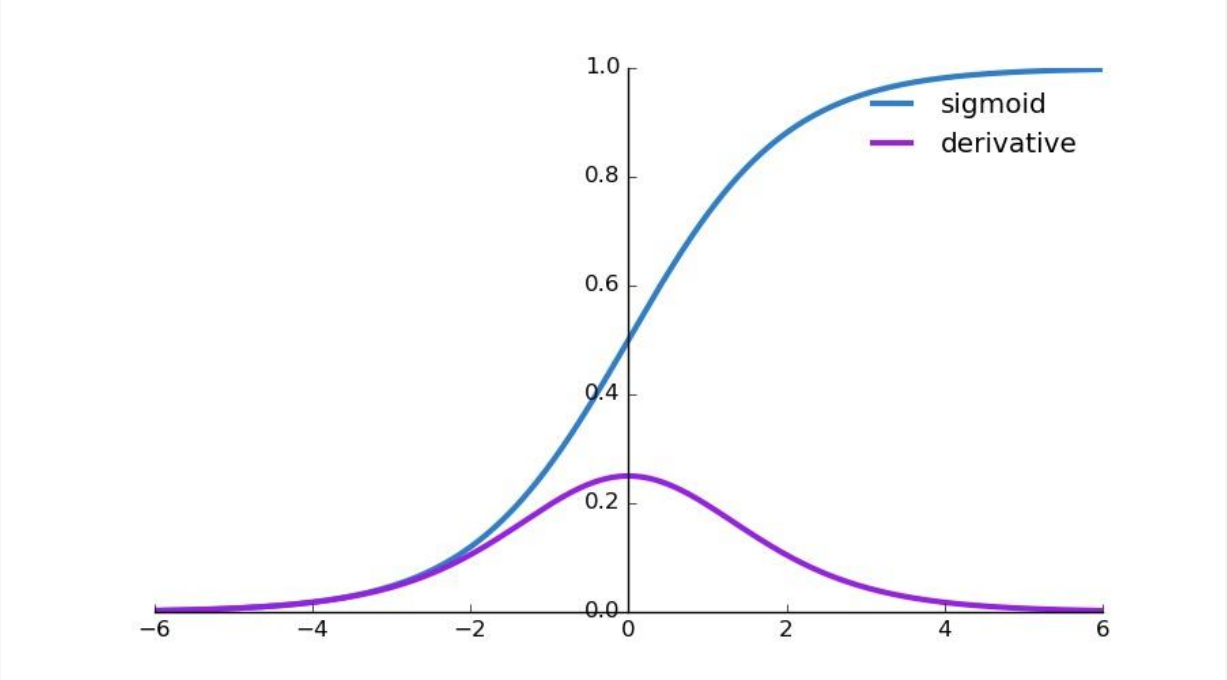
- Sigmoid function:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- Derivative Sigmoid function:

$$\frac{d\ sigmoid(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} =$$

$$= sigmoid(x)(1 - sigmoid(x))$$



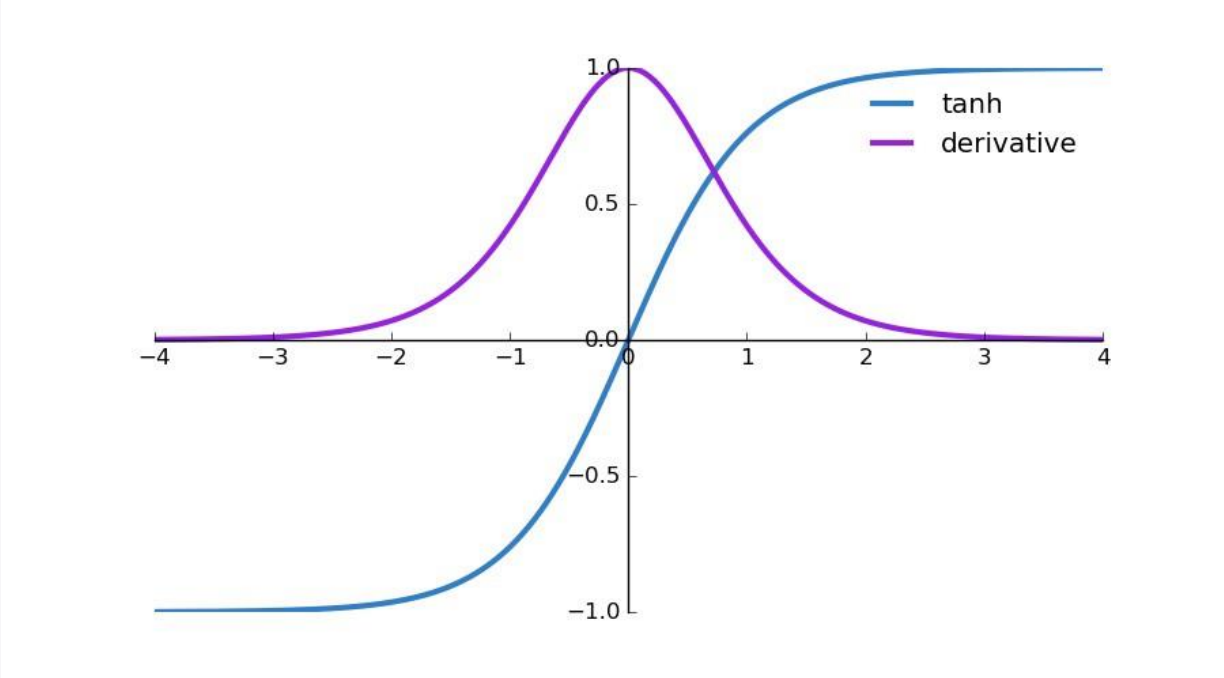
Tanh Function

- Tanh function:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- Derivative Tanh function:

$$\frac{d\,tanh(x)}{dx} = 1 - tanh^2(x)$$



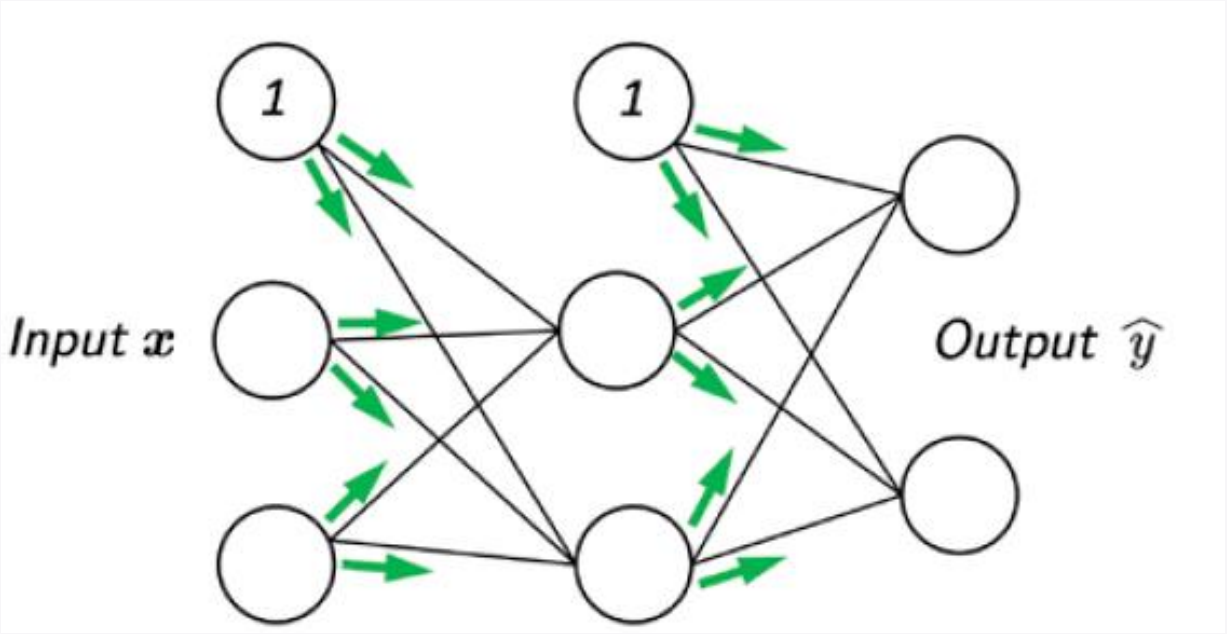
Multilayer perceptron

- In this case there are two outputs.
- The hidden layer output **h** is function of the input **x**:

$$h_1 = \sigma^h(w^i_{11}x_1 + w^i_{12}x_2 + b^i_1)$$
$$h_2 = \sigma^h(w^i_{21}x_1 + w^i_{22}x_2 + b^i_2)$$

- The output **o** is a different function of its input, i.e. **h**:

$$o_1 = \sigma^o(w^h_{11}h_1 + w^h_{12}h_2 + b^h_1)$$
$$o_2 = \sigma^o(w^h_{21}h_1 + w^h_{22}h_2 + b^h_2)$$

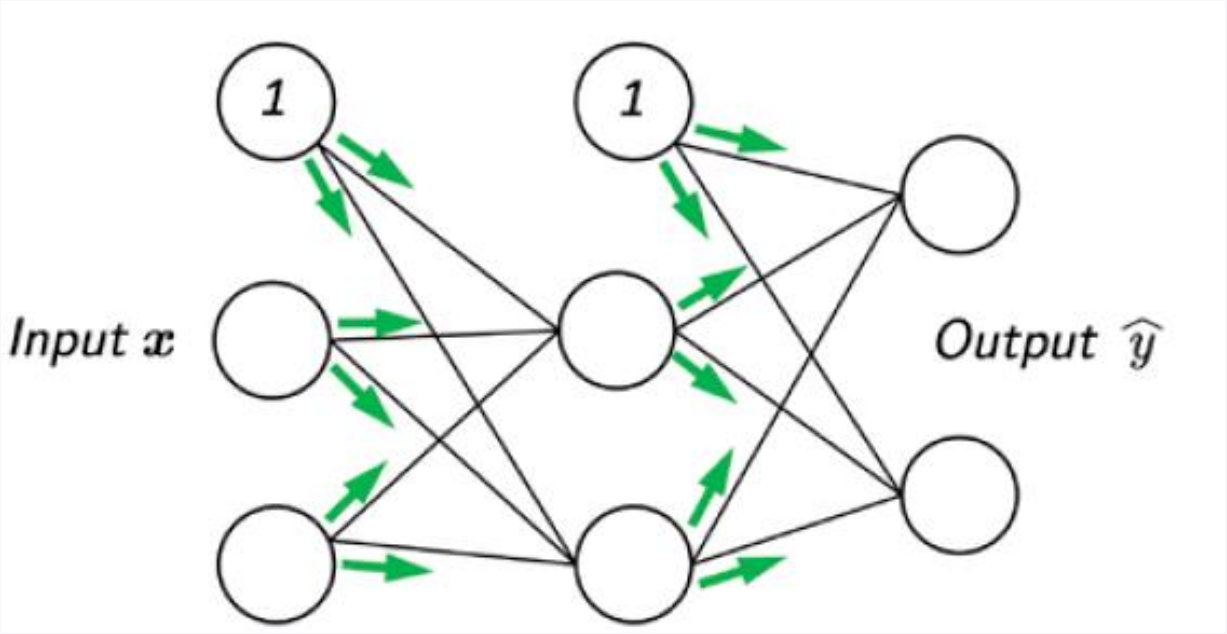


Multilayer perceptron

- How can we interpret the two values?
- In classification problem the goal is to understand how the input **x** is related to the belonging to a certain class.
- The output **o** could be seen as the vector of probabilities of belonging to each class.
- However this is not straightforward:

$$\sum_{i=1}^{\#classes} o_i \neq 1$$

$$o_i \not\geq 0 \;\; \forall \, i \in [1, \#classes]$$



Softmax Regression

- However this is not straightforward:

$$\sum_{i=1}^{\#classes} o_i \neq 1$$

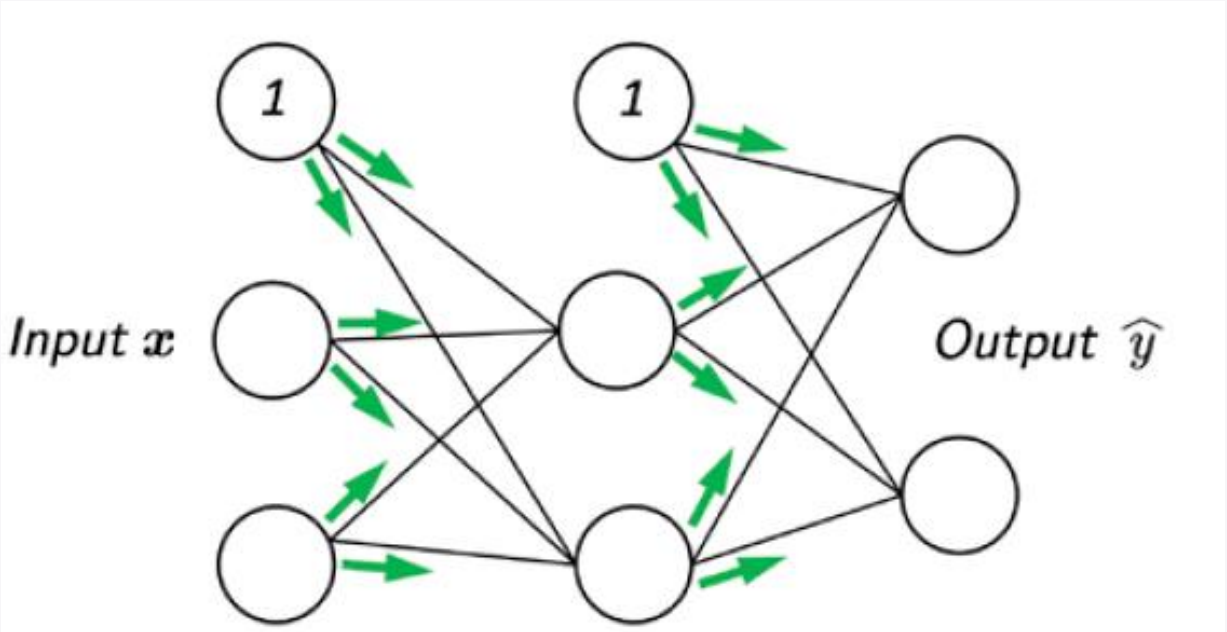
$$o_i \ngeq 0 \ \forall \, i \in [1, \#classes]$$

- Softmax activation function:

$$\mathbf{y} = softmax(\mathbf{o})$$

$$y_i = \frac{e^{o_i}}{\sum_k e^{o_k}}$$

$$y_{pred} = \underset{j}{argmax}\ y_j$$



Parametrization Cost

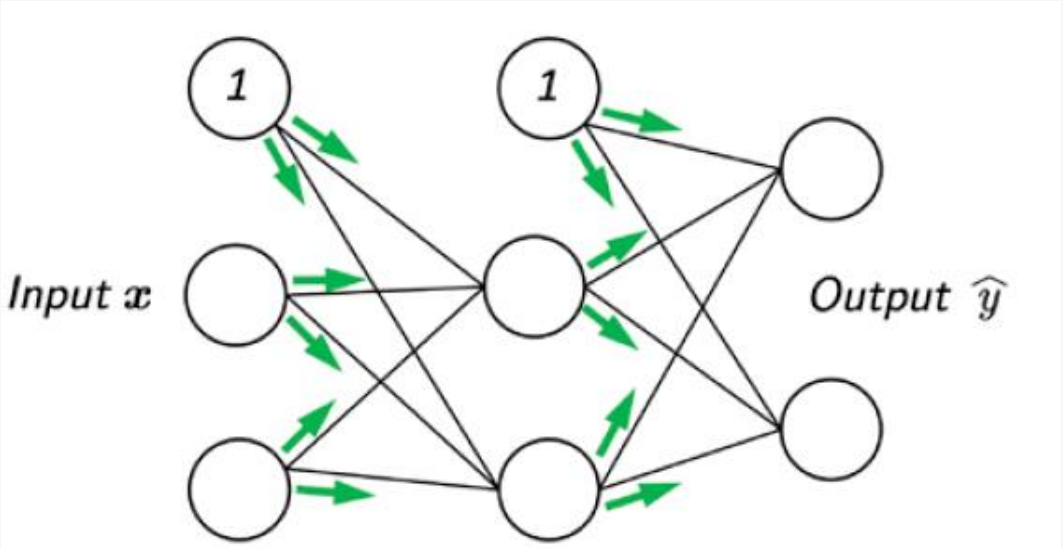
- Fully-connected layers are fundamental in the neural network building up process.
- Adding neurons to a network layer or adding a layer makes our model more complex and capable of facing a wide range of problems.
- The complexity of the model, however, faces directly with computational time, which could be extremely high.

Parametrization Cost

- Suppose to have a hidden layer with d input and q outputs. → **#neurons = q**
- The parametrization cost is → $\mathcal{O}(d \cdot q)$
- It is possible to reduce the parametrization cost introducing an hyperparameter n, in order to have the parametrization cost: → $\mathcal{O}\left(\frac{d \cdot q}{n}\right)$

Vectorization for Minibatches

- Input $X \in \mathcal{R}^{n \times d}$
- Weights $W \in \mathcal{R}^{d \times q}$
- Bias $b \in \mathcal{R}^{1 \times q}$
- Outputs $O = XW + b,\ O \in \mathcal{R}^{n \times q}$



Loss function

- To measure the quality of our predicted probabilities we need a loss function.
- We will suppose that the entire dataset (or the batch we are considering) has n examples {X,Y}.
- The i-th {X,Y} entry is made by the feature vector x-ith and the one-hot label vector y-ith.

Loss function

- It is possible to compare the predicted class with the real class by checking how probable the actual classes are according to out model.

- According to the maximum likelihood estimation, we want to maximize P(Y|X), or minimize the negative log-likelihood.

$$\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^{n} \mathcal{P}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

$$-log\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \sum_{i=1}^{n} -log\mathcal{P}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

Cross-entropy loss function

- The negative log-likelihood is equal to:

$$-log\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \mathbf{y}_{pred}^{(i)})$$

- Where $l(y,y_{pred})$ is the loss function, also called cross-entropy, defined as:

$$l(\mathbf{y}^{(i)}, \mathbf{y}_{pred}^{(i)}) = -\sum_{j=1}^{\#Classes} y_j^{(i)}\ log\ y_j^{(i)\ pred}$$

$$NB: y_j^{(i)\ pred} \leq 1 \ \rightarrow log\ y_j^{(i)\ pred} \leq 0$$

Cross-entropy loss function

- The cross-entropy loss function is a common choice in classification problems.

- Moreover it is generalizable when the vector of label y doesn't contain only binary entries like (1,0,0), but is a generic probability vector, (0.15,0.8,0.05).

$$l(\mathbf{y}^{(i)}, \mathbf{y}_{pred}^{(i)}) = -\sum_{j=1}^{\#Classes} y_j^{(i)} \, log \, y_j^{(i)\,pred}$$

- This is the case where we observe not just a single outcome but an entire distribution over outcomes.

Cross-entropy loss function and softmax

- The softmax and the corresponding loss are very common. What is the corresponding loss function?

$$l(y, y^{pred}) = -\sum_{j=1}^{\#Classes} y_j \, log \frac{e^{o_j}}{\sum_{k=1}^{\#Classes} e^{o_k}}$$

$$= \sum_{j=1}^{\#Classes} y_j \log \sum_{k=1}^{\#Classes} e^{o_k} - \sum_{j=1}^{\#Classes} y_j \, o_j$$

$$= \log \sum_{k=1}^{\#Classes} e^{o_k} - \sum_{j=1}^{\#Classes} y_j \, o_j$$

Cross-entropy loss function and softmax

- The softmax and the corresponding loss are very common. What is the corresponding loss function?

- To understand better let's have a look at the derivative w.r.t. any output o i-th:

$$\partial_{o_i} l(y, y^{pred}) = \frac{e^{o_i}}{\sum_{k=1}^{\#Classes} e^{o_k}} - y_i = softmax(o)_i - y_i$$

- The derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and the y true vector.

Examples of other loss functions

- Mean Squared Error (MSE)/ Quadratic Loss/ L2:

$$MSE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left(y^{(i)} - y^{(i)}_{pred}\right)^2}{n}$$

- Mean Absolute Error (MAE)/ L1 Loss:

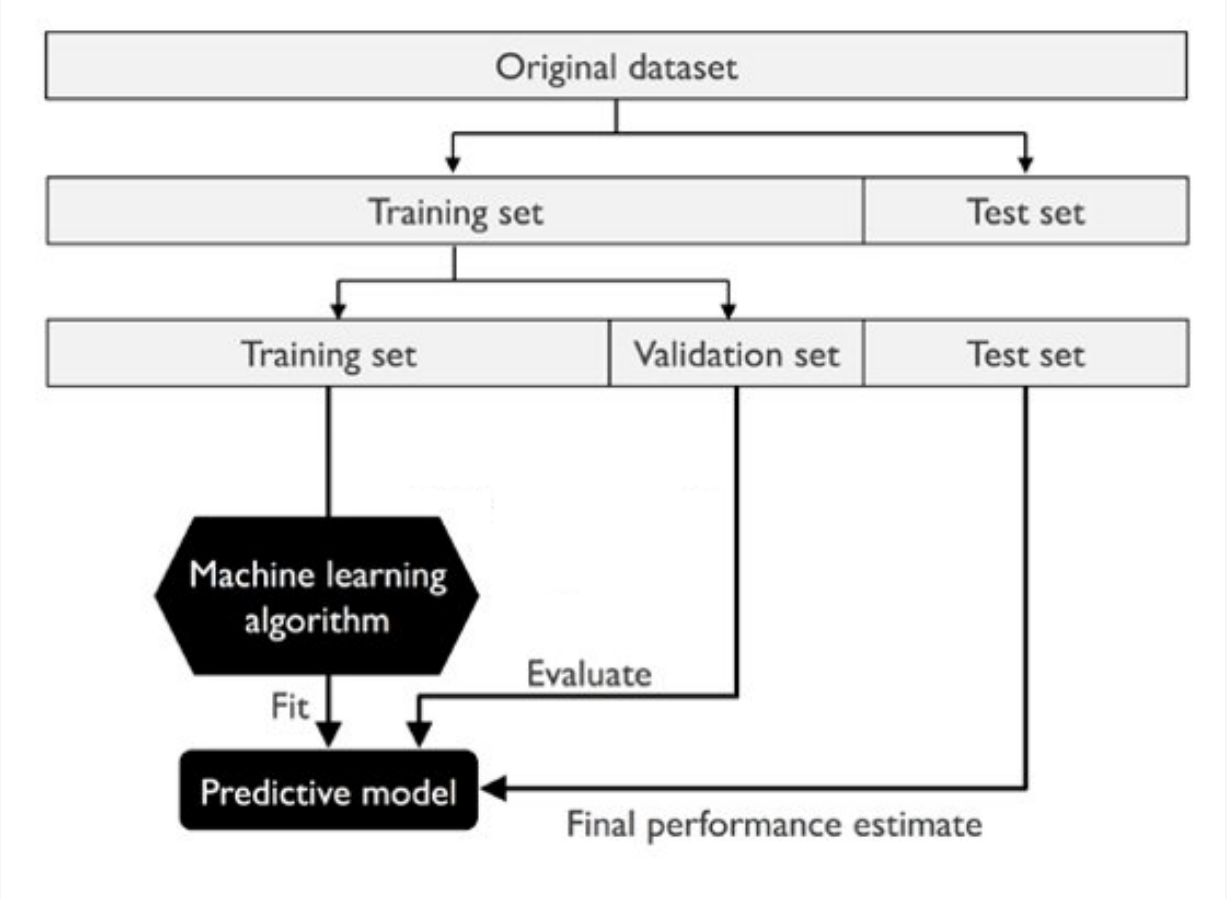
$$MAE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left|y^{(i)} - y^{(i)}_{pred}\right|}{n}$$

- Mean Bias Error (MBE):

$$MBE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left(y^{(i)} - y^{(i)}_{pred}\right)}{n}$$

Validation procedure

- When we want to optimize our model, in principle we should not touch the test data, there is a risk that we might overfit it.
- Ideally we would touch our test data only once to assess the very best model
- The common practice to address this problem is to split our data three ways, incorporating a *validation dataset* (or *validation set*) in addition to the training and test datasets.



Training description

- The main idea of the model training is to iterate over the network different times (number of epochs).
- In each epoch is selected from the test dataset k stochastic minibatches of n (batchsize) entries.
- We then compute the derivative (gradient) of the average loss on the minibatch regarding the model parameters. Finally, we multiply the gradient by a predetermined positive value η (learning rate) and subtract the resulting term from the current parameter values.
- The epoch ends after k iterations, i.e. all over the k batches.

Minibatch stochastic gradient descent

- We iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.

- Backward propagation of the training, parameters updating:

$$\mathbf{w} \rightarrow \mathbf{w} - \frac{\eta}{n} \sum_{i=0}^{n} \partial_{\mathbf{w}} \, l^i(\mathbf{w}, \mathbf{b})$$

$$\mathbf{b} \rightarrow \mathbf{b} - \frac{\eta}{n} \sum_{i=0}^{n} \partial_{\mathbf{b}} \, l^i(\mathbf{w}, \mathbf{b})$$

Hyperparameters

- An hyperparameter is an internal parameter of the model that must be fixed before training, such parameter influences the fit procedure in a way not well known a priori.
- So we cannot know which value is perfect for our model and we need to try different reasonably values to figure out which one is the best.
- **No Free Lunch theorem**: no single classifier works best across all possible scenarios

Hyperparameters

With a DNN we can change a lot of parameters, most of which we just describe below:

- The loss function
- The activation function of every layer
- The learning rate
- The number of epochs
- The number of hidden layers and the number of cells in them
- .. and many others

The hyperparameters can change from an algorithm to another, here we mentioned only the main parameters of a DNN.

Learning rate

- Adjusting the learning rate is often just as important as the actual algorithm
- If it is too large, optimization diverges, if it is too small, it takes too long to train or we end up with a suboptimal result
- If the learning rate remains large we may simply end up bouncing around the minimum and thus not reach optimality
- What we can do: we can decide to start from a reasonable value for the learning rate and then use the method implemented in Keras: "ReduceLROnPlateau". It reduce the learning rate when a monitor, decided by us, has stopped improving. In this way we can obtain a large LR value at the begin of the training with a progressive reduction when we are approaching to the optimized model.

Number of epochs

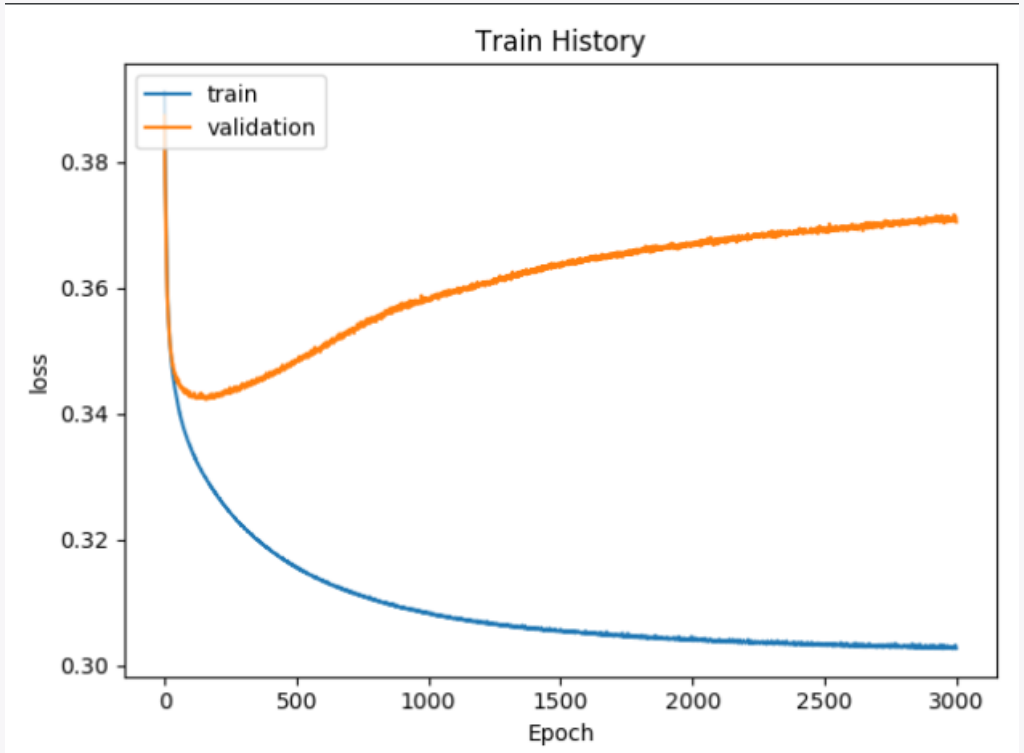
- The epoch is a complete step of the training, it includes the evaluation and the consecutive updating of the weights
- The number of epochs is a delicate choice, because a large number can induce our model to an overfitting problem. Meanwhile a too small value can lead to an under fitting problem
- To avoid a wrong choice we can use the 'EarlyStopping', also implemented by Keras. This method allows to stop the training when a monitor, decided by us (for example: loss function) has stopped improving.

Hidden layers

- As seen before, the number of hidden layers add complexity to our model.
- Adding hidden layers make our algorithm more performing, but at the same time we lead it to an overfitting problem
- Another crucial factor is the number of cells in the hidden layer, also in this case a lot of cells increase the complexity of the model and increase the risk to an overfitting problem
- This choice has to be done carefully, it is the most difficult one and only comparing the evaluation metrics between different approaches we can know which is the best one.

Overfitting problem

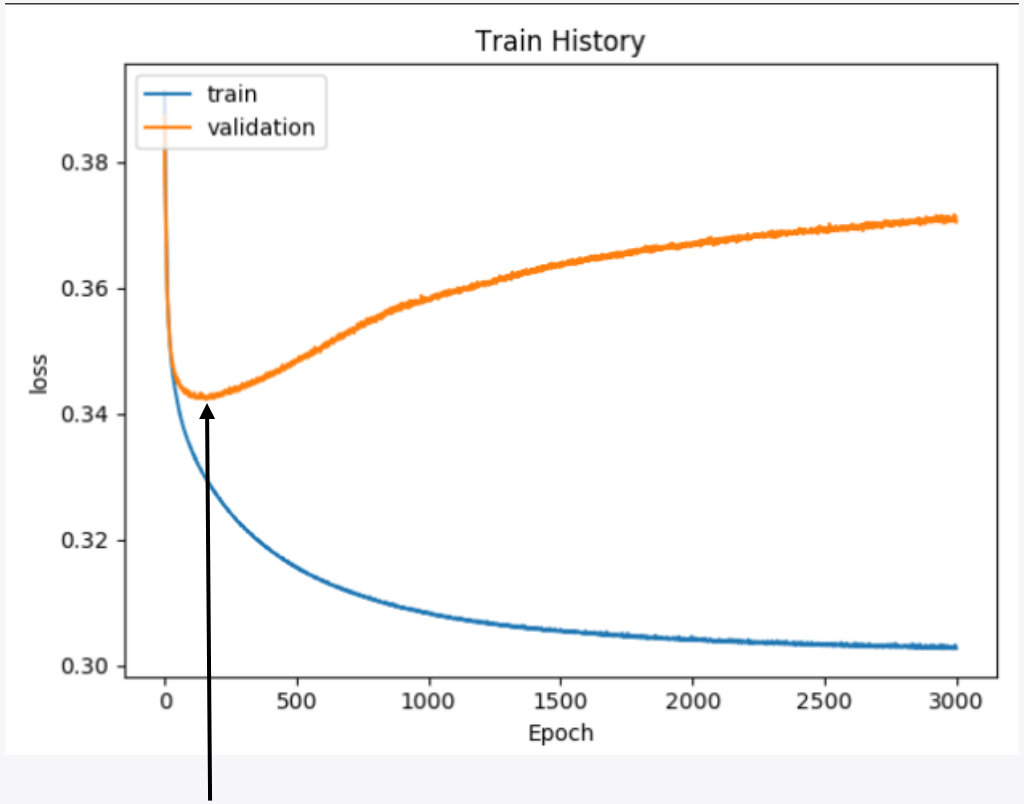
- The more complex the model is, the higher is the risk of overfitting.
- Here a clear example of overfittig, the train loss keeps going down while the validation loss get worse. It is always important to split the training in train and validation set and to have a clear picture of the train history.
- In order to avoid overfitting and make the training stable we have different approach.



Overfitting problem

- In order to avoid overfitting and make the training stable we have different approach:

1. Introduce a callback function that stops the training if the validation loss get worse and restore the best parameters (Early Stop function). Reduce overtraining and time needed for the training.

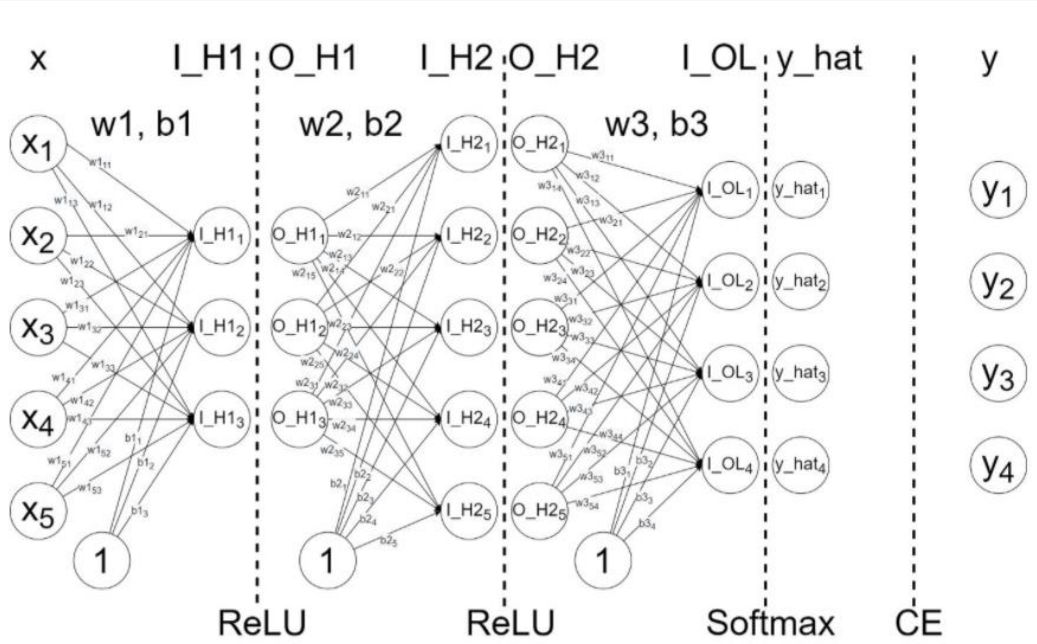


Stop and restore the parameters

Overfitting problem

- In order to avoid overfitting and make the training stable we have different approach:

1. Introduce a callback function that stops the training if the validation loss get worse and restore the best parameters (Early Stop function). Reduce overtraining and time needed for the training.
2. "Weight Decay": introduce penalty terms in the loss.



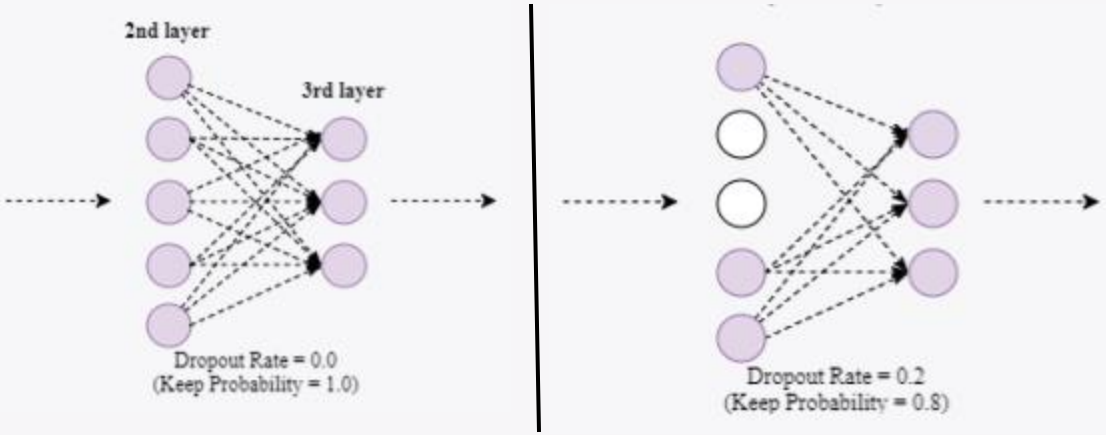
$$loss = CE$$
$$+L1 * (|W3_{11}| + |W3_{12}| + ...) + L1 * (|B3_1| + |B3_2| + ...)$$
$$+L2 * ({W3_{11}}^2 + {W3_{12}}^2 + ...) + L2 * ({B3_1}^2 + {B3_2}^2 + ...)$$
$$+L1 * (|W2_{11}| + |W2_{12}| + ...) + L1 * (|B2_1| + |B2_2| + ...)$$
$$+L2 * ({W1_{11}}^2 + {W1_{12}}^2 + ...) + L2 * ({B1_1}^2 + {B1_2}^2 + ...)$$

Overfitting problem

- In order to avoid overfitting and make the training stable we have different approach:

1. Introduce a callback function that stops the training if the validation loss get worse and restore the best parameters (Early Stop function). Reduce overtraining and time needed for the training.
2. "Weight Decay": introduce penalty terms in the loss.
3. "Dropout": injecting noise while computing each internal layer during forward propagation.

The calculation of the outputs no longer depends on h2 or h3 and their respective gradient also vanishes when performing backpropagation.
In this way, the calculation of the output layer cannot be overly dependent on any one element of (h1,…,h5).

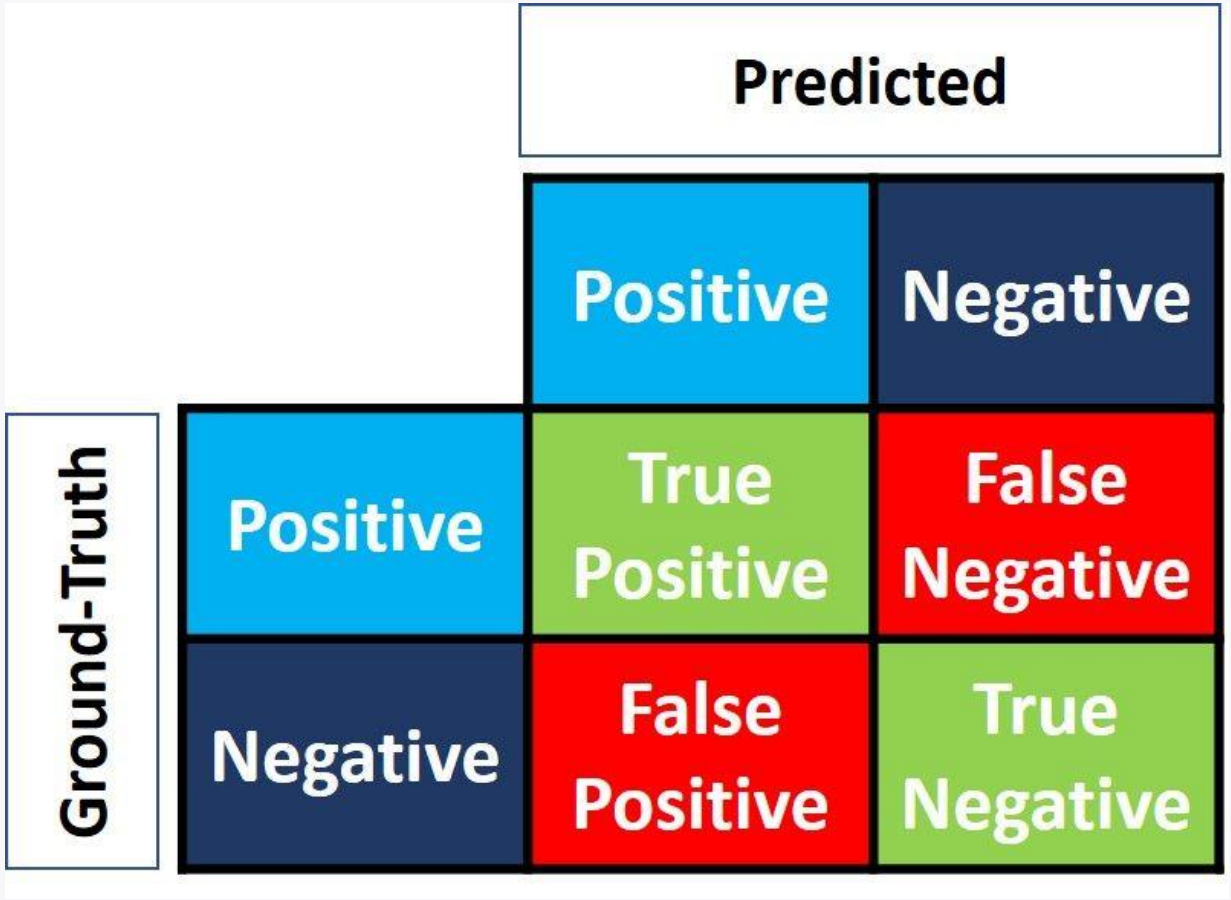


Evaluation metrics

- The idea of building machine learning models works on a constructive feedback principle. You build a model, get feedback from metrics, make improvements and continue until you achieve a desirable accuracy
- Evaluation metrics explain the performance of a model. An important aspect of evaluation metrics is their capability to discriminate among model results
- Simply building a predictive model is not your motive. It's about creating and selecting a model which gives high accuracy on out of sample data. Hence, it is crucial to check the accuracy of your model prior to computing predicted values.

Confusion matrix

- The confusion matrix helps us visualize whether the model is "confused" in discriminating between two or more classes.
- In the figure we have an example of binary model and the corresponding confusion matrix.
- The 4 elements of the matrix represent the 4 metrics that count the number of correct and incorrect predictions the model made.



Accuracy

$$Accuracy = \frac{True_{positive} + True_{negative}}{True_{positive} + True_{negative} + False_{positive} + False_{negative}}$$

- The most famous metrics is the accuracy defined as the ratio between the number of correct predictions to the total number of predictions
- Accuracy values range between 0 and 1. Obviously an accuracy values near to 1 means that our model fits well the datasets
- It is important to stress that a good accuracy value on the training dataset does not imply a good discrimination on the test dataset

Precision

$$Precision = \frac{True_{positive}}{True_{positive} + False_{positive}}$$

- The precision is calculated as the ratio between the number of *Positive* samples correctly classified to the total number of samples classified as *Positive* (either correctly or incorrectly). The precision measures the model's accuracy in classifying a sample as positive.
- When the model makes many incorrect *Positive* classifications, or few correct *Positive* classifications, this increases the denominator and makes the precision small. On the other hand, the precision is high when:
  - The model makes many correct *Positive* classifications (maximize *True Positive*).
  - The model makes fewer incorrect *Positive* classifications (minimize *False Positive*).

Recall

$$Recall = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

- The recall is calculated as the ratio between the number of *Positive* samples correctly classified as *Positive* to the total number of *Positive* samples.
- The recall cares only about how the positive samples are classified. This is independent of how the negative samples are classified, e.g. for the precision.
- The decision of whether to use precision or recall depends on the type of problem being solved. If the goal is to detect all the positive samples (without caring whether negative samples would be misclassified as positive), then use recall. Use precision if the problem is sensitive to classifying a sample as *Positive* in general, i.e. including *Negative* samples that were falsely classified as *Positive*.

F1 score

$$F_1 = \left(\frac{\text{recall}^{-1} + \text{precision}^{-1}}{2} \right)^{-1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

- F1-Score is the harmonic mean of precision and recall values for a classification problem.
- It takes the harmonic mean because punishes extreme values more.
- For example, if we have a model with Precision = 0 and Recall =1, it is clear that this result comes from a dumb classifier which just ignores the input and just predicts one of the classes as output. In this example we will have a F1 score equal to 0

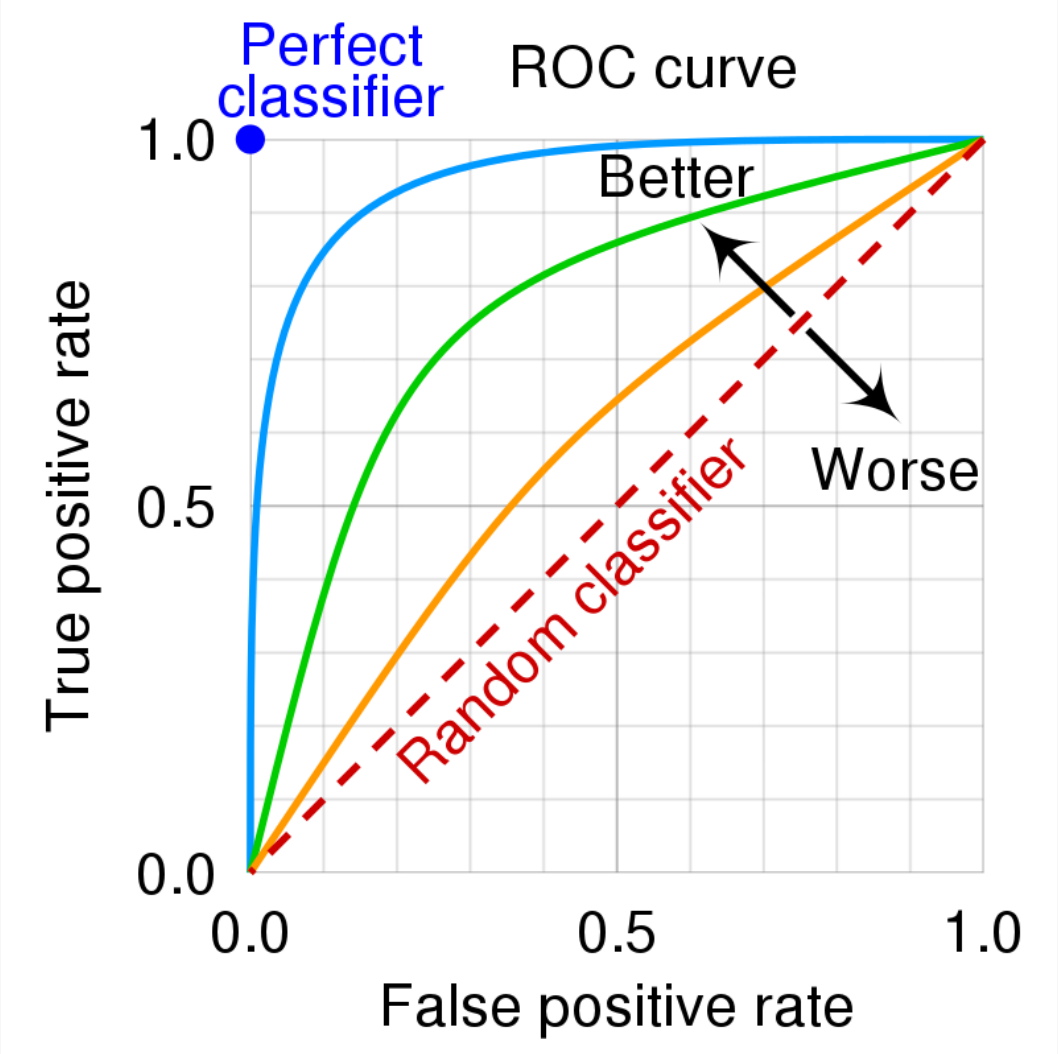
AUC-ROC

- A **Receiver Operating Characteristic curve**, or **ROC curve**, is a plot that illustrates the true positive rate against the false positive rate

$$TPR = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

$$FPR = \frac{False_{positive}}{False_{positive} + True_{negative}}$$

- The metric connected to the ROC curve is th area under the curve **AUC**. An AUC near to 1 indicates a ROC curve near to the best result, an AUC near to 0 indicates a random classifier.



Tutorial NN

- This afternoon we will see how to work with the NNs. We are going to use keras library: keras guide .

```
>>import keras
```

- We will use sequential models:

```
>>model = keras.models.Sequential()
```

- So, we can add how many layers we want using:

```
>>model.add(keras.layers.Dense(...))
```

- Alternatively, we can insert all layers when we initialize the model.
- With `keras.layers` we can access at different types of layers, we will use Dense layer (a fully connected layer) and Dropout layer.
- Into the () we will insert the numbers of cells and the activation function for Dense, while for the Dropout we'll insert the dropout rate.

Tutorial NN

- Once the model is completed, we have to compile and train it:

  ```
  >>model.compile(loss=" , optimizer = .., metrics= …)
  >>history = model.fit(X_train, y_train, validation_split=%, epochs=#, batch_size=#,verbose=(0,1))
  ```

- The fit method returns information on each epoch, activating the verbose variable we also can see the results of each epoch during the training
- Finally, we will use `keras.callbacks` to introduce the early stop and the learning rate reduction in our models
- More on that this afternoon!

References

- Raschka, Sebastian. *Python machine learning*. Packt publishing ltd, 2015.

- Dive Into Deep Learning

http://d2l.ai/chapter_preface/index.html

- Keras Guide

Developer guides (keras.io)