1 - Introduction to Deep Learning

Giovanni Della Lunga
giovanni.dellalunga@unibo.it

Introduction to Machine Learning for Finance

Bologna - February, 2022

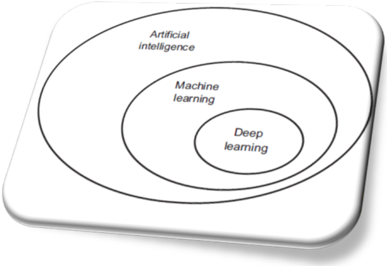
We will talk about...

- What is a Neural Network
- Feedforward Neural Networks
- Keras: the Python Deep Learning API

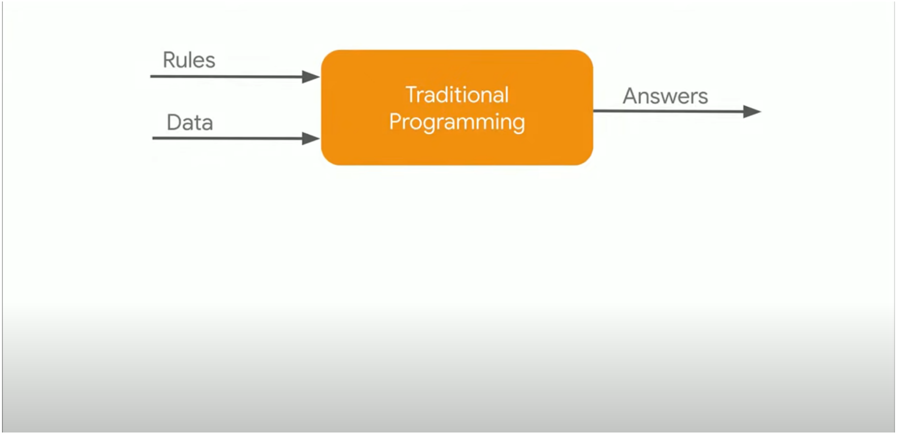
Machine Learning and Deep Learning

Machine Learning and Deep Learning

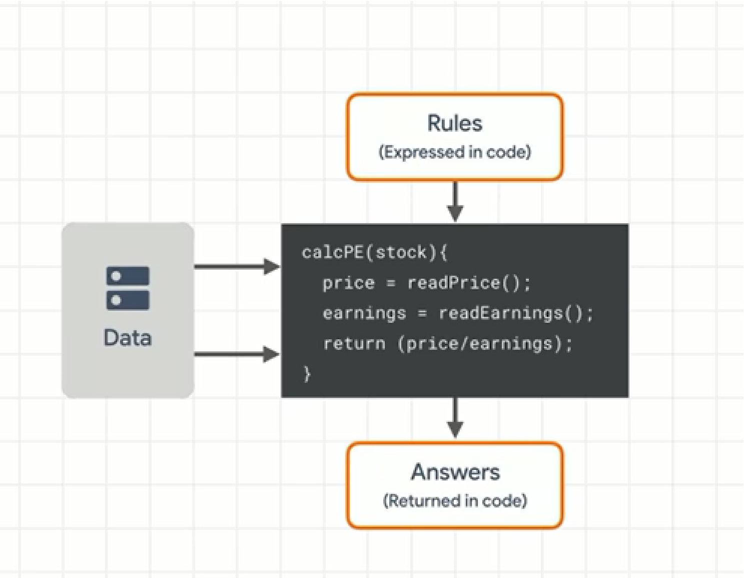
- As we know to do machine learning we need three things:
- Input data points
- Examples of the expected output
- A way to measure whether the algorithm is doing a good job



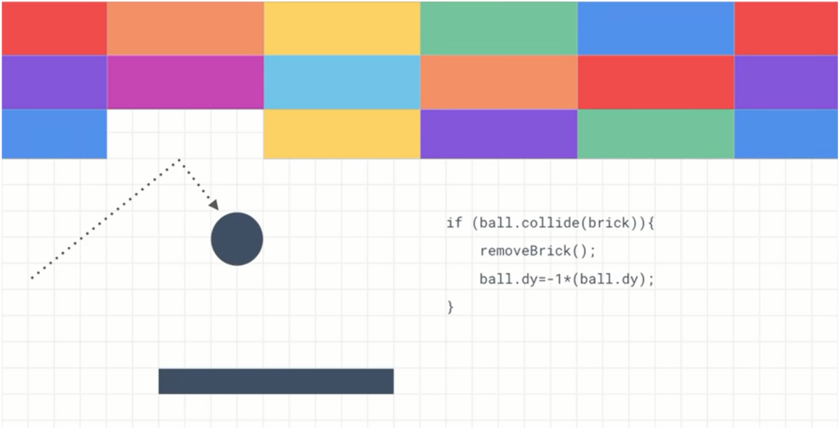
Machine Learning and Deep Learning



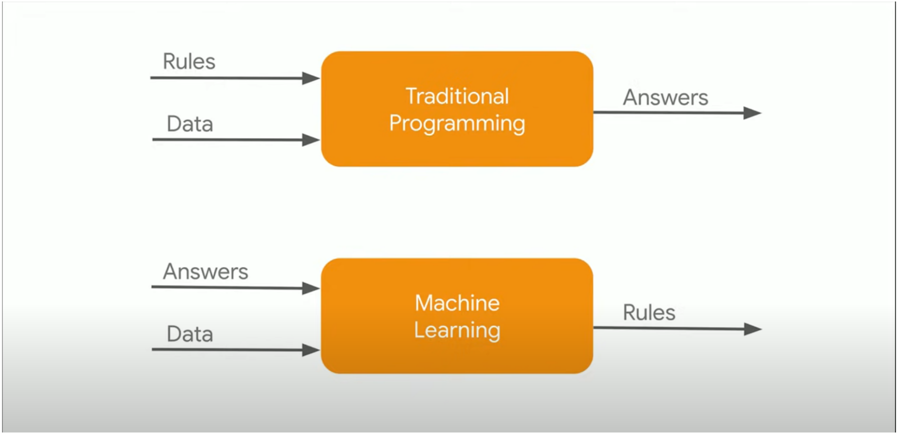
Machine Learning and Deep Learning



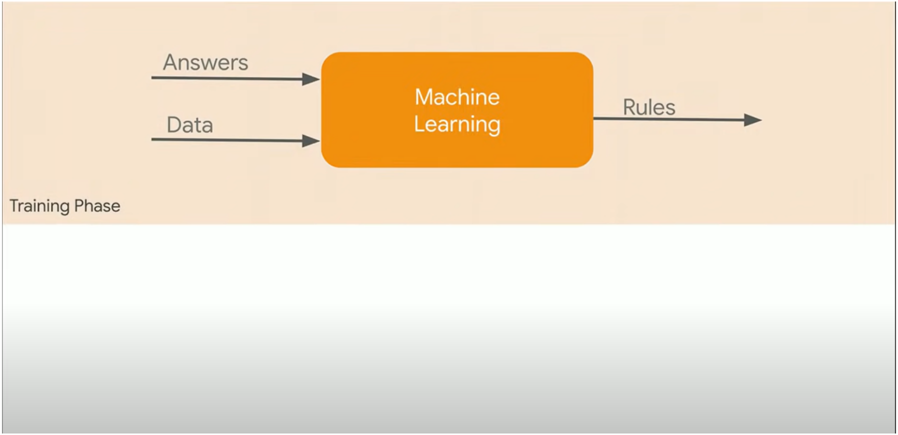
Machine Learning and Deep Learning



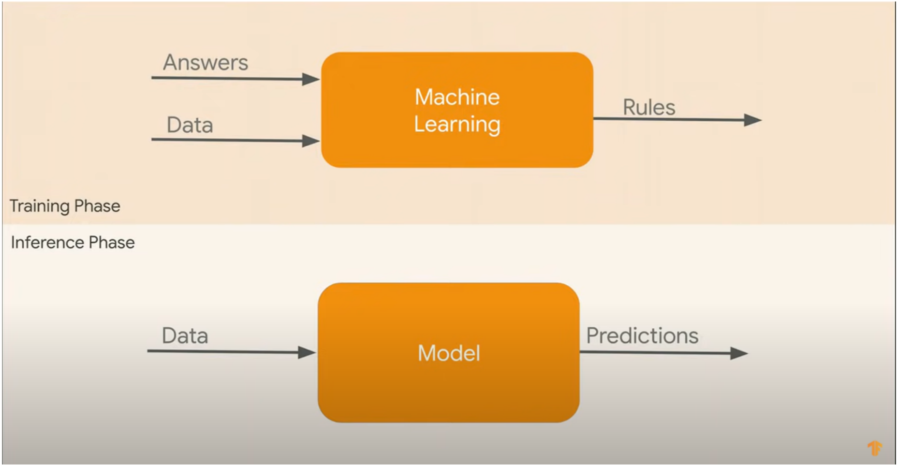
Machine Learning and Deep Learning



Machine Learning and Deep Learning



Machine Learning and Deep Learning



Machine Learning and Deep Learning

- A machine-learning model transforms its input data into meaningful outputs, a process that is **learned** from exposure to known examples of inputs and outputs.
- Therefore, the central problem in machine learning and deep learning is to **meaningfully transform data**: in other words, **to learn useful representations of the input data at hand, representations that get us closer to the expected output**.
- What is a representation? At its core, it is simply a different way to look at data, to represent or encode data.

Machine Learning and Deep Learning

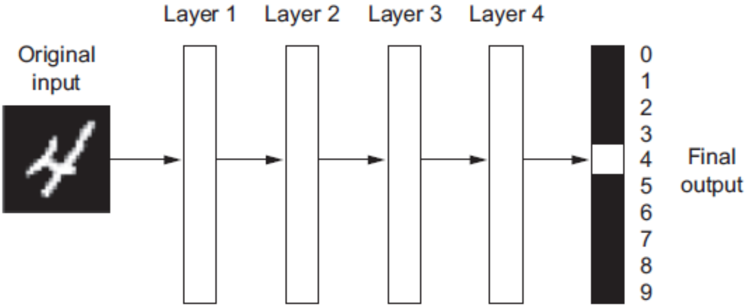
- From a formal poin of view, deep learning learning process involves input variables, which we call X, and output variables, which we call Y.
- We us neural network to learn the mapping function from the input to the output.
- In simple mathematics, the output Y is a dependent variable of input X as illustrated by:

$$Y = f(X)$$

Here, our end goal is to try to **approximate the mapping function** f, so that we can **predict** the output variables Y when we have new input data X.

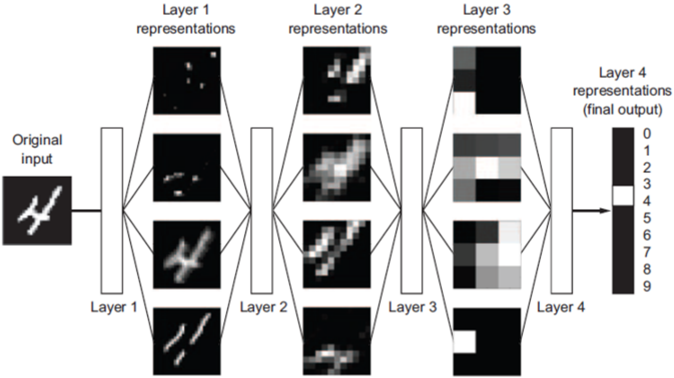
Machine Learning and Deep Learning

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations.



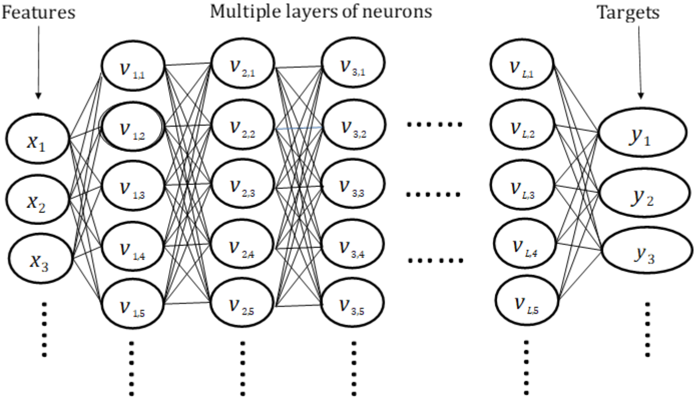
Machine Learning and Deep Learning

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations.



Machine Learning and Deep Learning

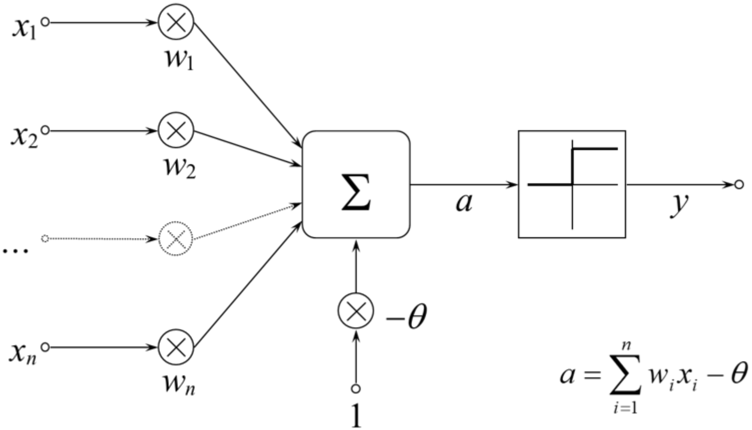
In deep learning, these layered representations are (almost always) learned via models called **neural networks**, structured in literal layers stacked on top of each other.



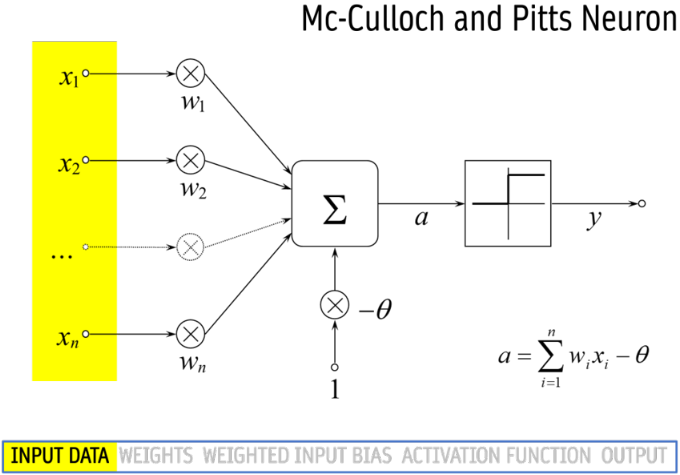
Subsection 1

The McCulloch-Pitts Neuron

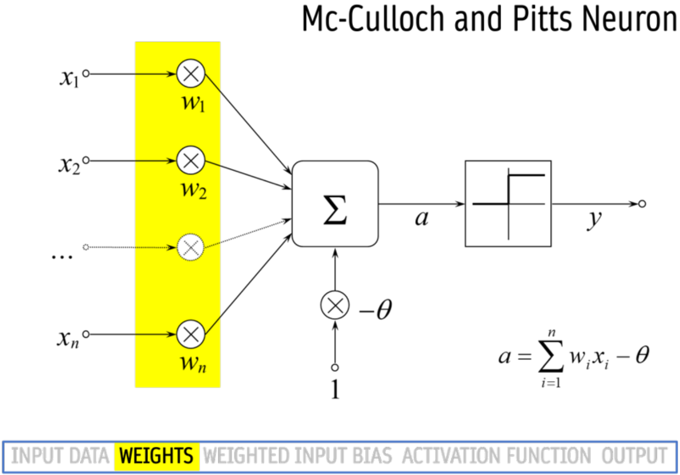
Mc-Culloch and Pitts Neuron



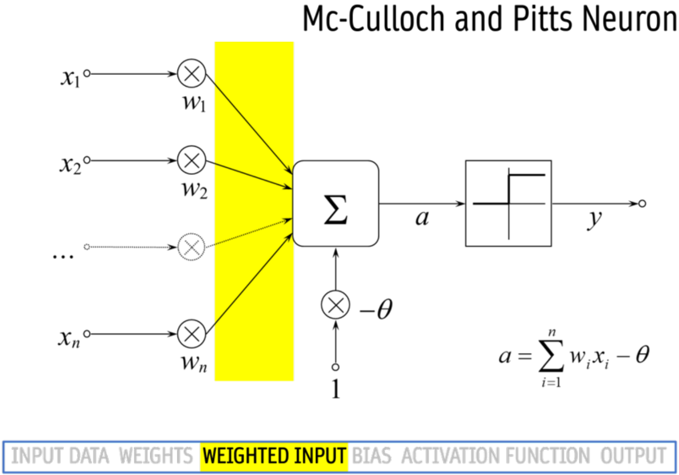
NN Data Flow: Input Data



NN Data Flow: Weights



NN Data Flow: Weighted Input

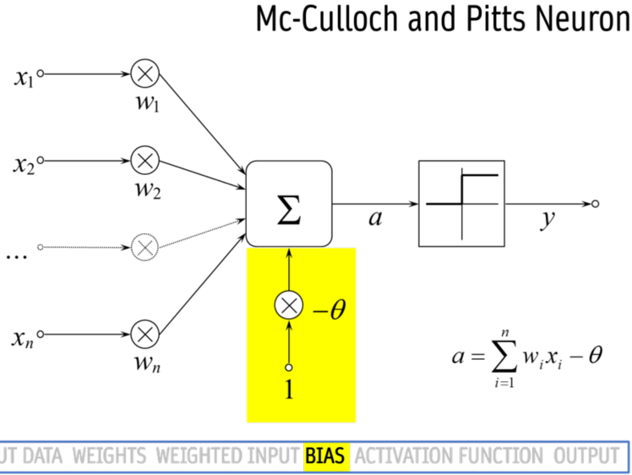


NN Data Flow: Weighted Input

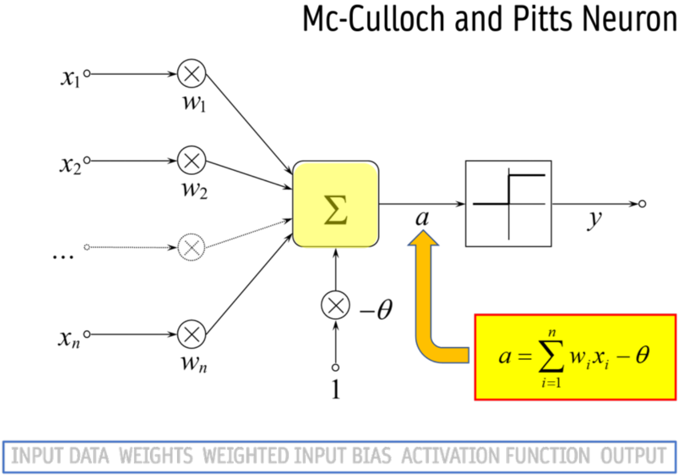
From a functional point of view

- an input signal formally present but associated with a zero weight is equivalent to an absence of signal;
- the threshold can be considered as an additional synapse, connected in input with a fixed weight equal to 1;

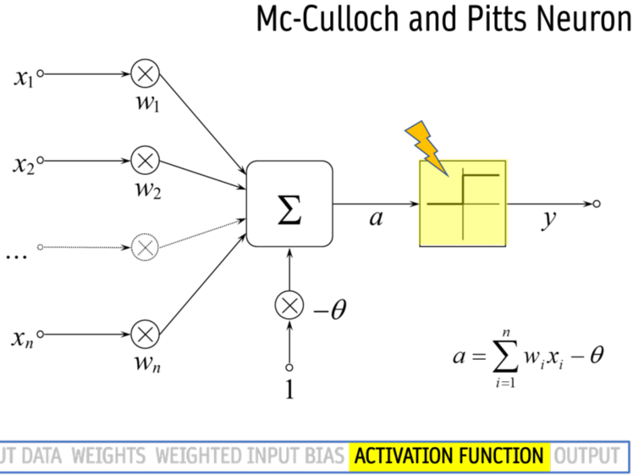
NN Data Flow: Bias



NN Data Flow: Weighted Output



NN Data Flow: Activation Function



NN Data Flow: Activation Function

- The function f is called the response or activation function:
- The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.
- Activation Function decides whether a neuron should be activated or not.
- This means that **it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations**.

NN Data Flow: Activation Function

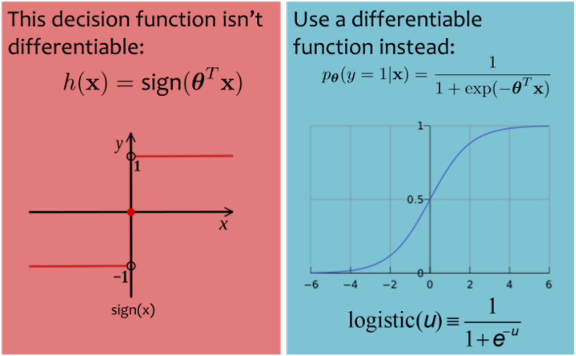
- In the McCulloch and Pitts neuron f is simply the step function, so the answer is binary: it is 1 if the weighted sum of the stimuli exceeds the internal threshold; 0 otherwise.

$$a = \sum_{i=1}^{n} w_i\, x_i - \theta \tag{1}$$

$$y = f(a) = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases} \tag{2}$$

- Other models of artificial neurons predict continuous response functions

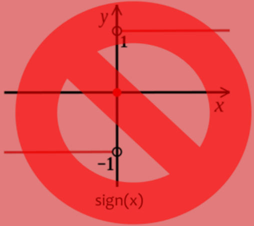
NN Data Flow: Activation Function



NN Data Flow: Activation Function

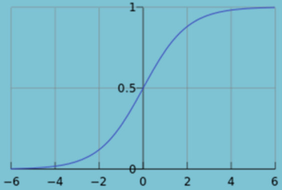
This decision function isn't differentiable:

$$h(\mathbf{x}) = \mathrm{sign}(\boldsymbol{\theta}^T\mathbf{x})$$



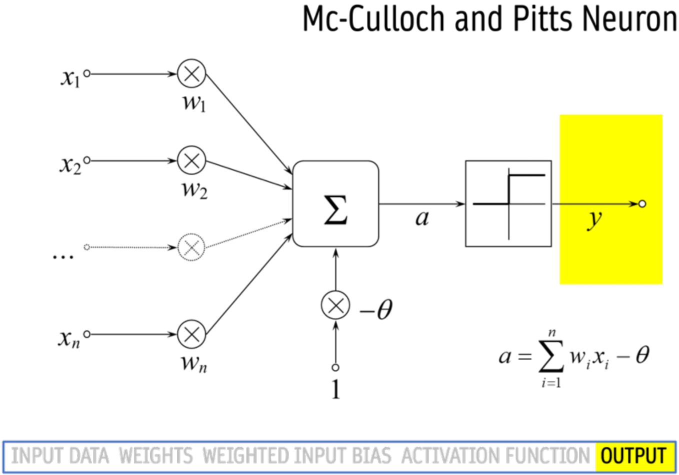
Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y=1|\mathbf{x}) = \frac{1}{1+\exp(-\boldsymbol{\theta}^T\mathbf{x})}$$



$$\mathrm{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

NN Data Flow: Output



Subsection 2

Basic Elements of a Neural Network

Neural Network Basic Constituents

A neural network consists of:

- A set of nodes (neurons), or units connected by links.
- A set of **weights** associated with links.
- A set of thresholds or activation levels.

Neural network design requires:

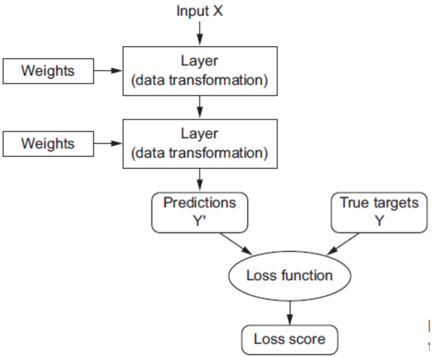
- The choice of the number and type of units.
- The determination of the morphological structure.
- Coding of training examples, in terms of network inputs and outputs.
- Initialization and training of weights on interconnections, through the set of learning examples.

Neural Network Basic Constituents

- The specification of what a layer does to its input data is stored in the layer's weights, which in essence are a bunch of numbers.
- In technical terms, we could say that the transformation implemented by a layer is parameterized by its weights (Weights are also sometimes called the parameters of a layer.)
- In this context, **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets**.

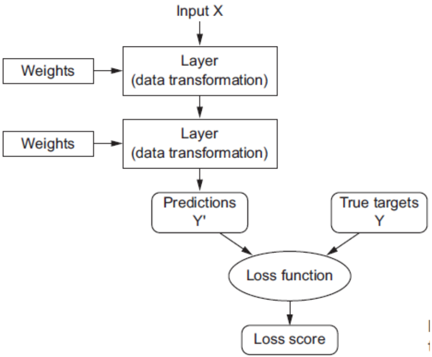
Loss Function

- To control the output of a neural network, you need to be able to measure how far this output is from what you expected.
- This is the job of the **loss function** of the network, also called the objective function.

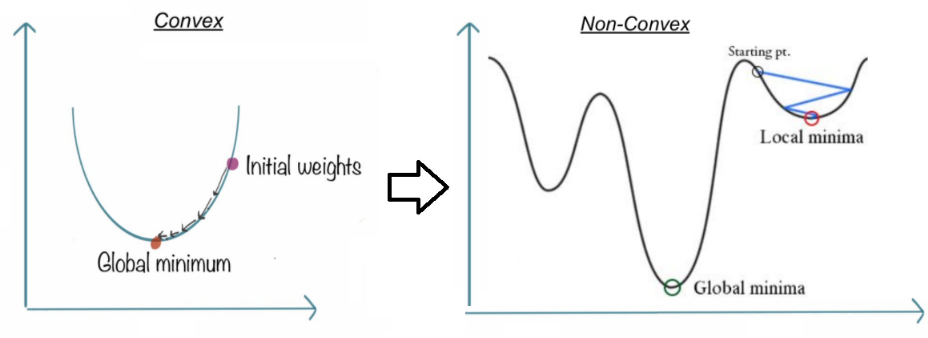


Loss Function

- The loss function takes the predictions of the network and the true target (what you wanted the network to output) and **computes a distance score, capturing how well the network has done on this specific example**

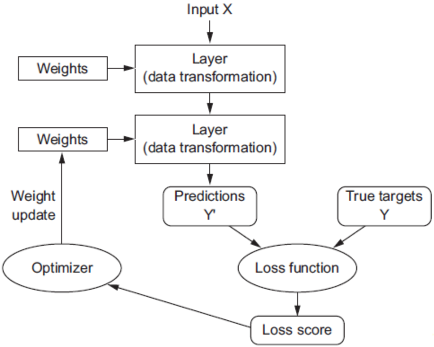


Loss Function



Backpropagation

- The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example.
- This adjustment is the job of the optimizer, which implements what is called the Backpropagation algorithm: the central algorithm in deep learning.

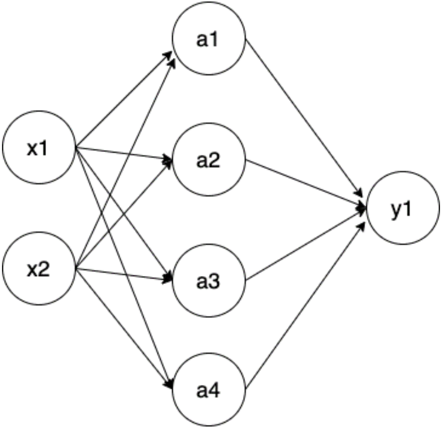


Implementing a Single Layer NN

Implementing a Single Layer NN

- In each hidden unit, take a_1 as example, **a linear operation followed by an activation function, f, is performed**.
- So given input $x = (x_1, x_2)$, **inside node** a_1, we have:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$
$$a_1 = f(w_{11}x_1 + w_{12}x_2 + b_1) = f(z_1)$$

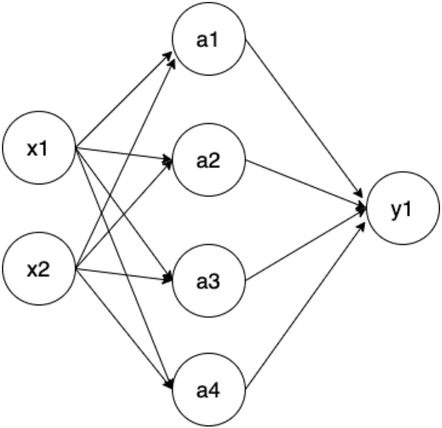


Implementing a Single Layer NN

- Same for node a_2, it would have:

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$
$$a_2 = f(w_{21}x_1 + w_{22}x_2 + b_2) = f(z_2)$$

- And same for a_3 and a_4 and so on



Implementing a single Layer NN

We can also write in a more compact form

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]} \tag{3}$$

The output is one value y_1 in $[0, 1]$, consider this a binary classification task with a prediction of probability

Dataset Generation

Scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity. Here we generate a simple binary classification task with 5000 data points and 20 features for later model validation.

```
from sklearn import datasets
#
X, y = datasets.make_classification(n_samples=5000, random_state=123)
#
X_train, X_test = X[:4000], X[4000:]
y_train, y_test = y[:4000], y[4000:]
#
print('train shape', X_train.shape)
print('test shape', X_test.shape)
```

Implementing a single Layer NN

Let's assume that the first activation function is the tanh and the output activation function is the *sigmoid*. So the result of the hidden layer is:

$$A^{[1]} = \tanh Z^{[1]}$$

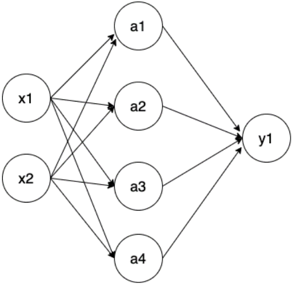
This result is applied to the output node which will perform another linear operation with a different set of weights, $W^{[2]}$:

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

and the final output will be the result of the application of the output node activation function (the sigmoid) to this value:

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

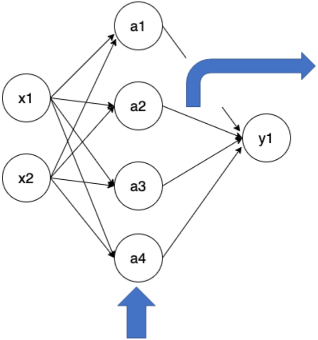
Implementing a single Layer NN



This is the first part of the computation performed by the nodes of the inner layer ...

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN

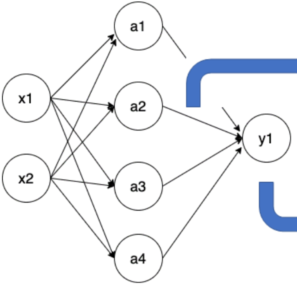


Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN



Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

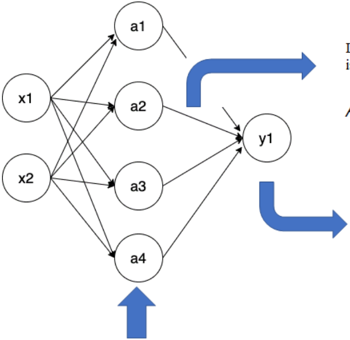
$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN



$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

The Final Output will be the result of the application of the output node activation function (sigmoid):

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ($W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$).
- Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.

```
def init_weights(n_input, n_hidden, n_output):
    params = {}
    params['W1'] = np.random.randn(n_hidden, n_input) * 0.01
    params['b1'] = np.zeros((n_hidden, 1))
    params['W2'] = np.random.randn(n_output, n_hidden) * 0.01
    params['b2'] = np.zeros((n_output, 1))

    return params
```

Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ($W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$).
- Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.

```
params = init_weights(20, 10, 1)

print('W1 shape', params['W1'].shape)
print('b1 shape', params['b1'].shape)
print('W2 shape', params['W2'].shape)
print('b2 shape', params['b2'].shape)
```

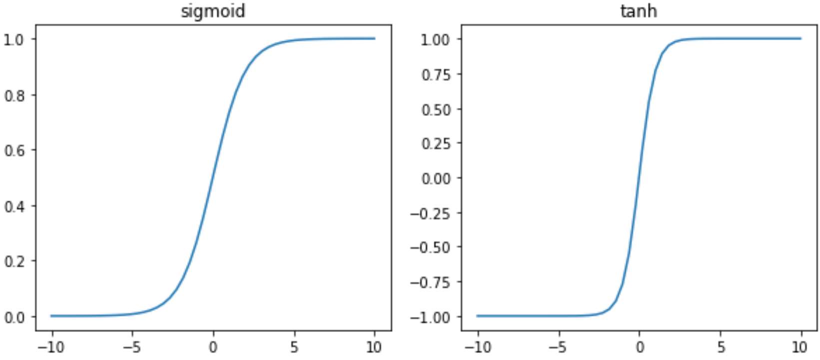
Forward Propagation

$$\begin{aligned}\hat{y} = A^{[2]} = \sigma\left(Z^{[2]}\right) &= \sigma\left(W^{[2]} \cdot A^{[1]} + B^{[2]}\right) \\ &= \sigma\left(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}\right) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

```python
def forward(X, params):
    W1, b1, W2, b2 = params['W1'], params['b1'], params['W2'], params['b2']
    A0 = X
    cache = {}
    Z1 = np.dot(W1, A0) + b1
    A1 = tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache['Z1'] = Z1
    cache['A1'] = A1
    cache['Z2'] = Z2
    cache['A2'] = A2
    return  cache
```

Activation Functions

Function tanh and sigmoid looks as below. Notice that the only difference of these functions is the scale of y.



Logistic Loss Function

Since we have a binary classification problem, we can assume a Logistic Loss Function (see the problem of logistic regression)

$$L(y, \hat{y}) = \begin{cases} -\log \hat{y} & \text{when } y = 1 \\ -\log(1 - \hat{y}) & \text{when } y = 0 \end{cases} \tag{4}$$

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

Where $\hat{y}$ is our prediction ranging in $[0, 1]$ and y is the true value.

Logistic Loss Function

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

```
def loss(Y, Y_hat):
    """
    Y: vector of true value
    Y_hat: vector of predicted value
    """
    assert Y.shape[0] == 1
    assert Y.shape == Y_hat.shape
    m = Y.shape[1]
    s = Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)
    loss = -np.sum(s) / m
    return loss
```

Delta Rule

- Given a generic actual value y, we want to minimize the loss L, and the technic we are going to apply here is gradient descent;
- basically what we need to do is to apply derivative to our variables and move them slightly down to the optimum.
- Here we have 2 variables, W and b, and for this example, the update formula of them would be:

$$W_{new} = W_{old} - \frac{\partial L}{\partial W} \Rightarrow \Delta W = -\frac{\partial L}{\partial W}$$
$$b_{new} = b_{old} - \frac{\partial L}{\partial b} \Rightarrow \Delta b = -\frac{\partial L}{\partial b}$$

Delta Rule

- The delta rule algorithm works by computing the gradient of the loss function with respect to each weight.
- Remember that

$$
\begin{aligned}
\hat{y} = A^{[2]} &= \sigma\left(Z^{[2]}\right) = \sigma\left(W^{[2]} \cdot A^{[1]} + B^{[2]}\right) \\
&= \sigma\left(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}\right) \\
&= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]
\end{aligned}
$$

As you can see $\hat{y}$ depends on both $W^{[1]}$ and $W^{[2]}$. The specification of what a layer does to its input data is stored in the layer's weights. Remember once again that **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets**

Delta Rule

- In order to get the derivative of our targets, chain rules would be applied:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial b}$$

- Let's focus only on the calculation of the derivative with respect to W since the calculation of the other derivative (with respect to b) is completely equivalent...

Delta Rule

$$\frac{\partial L}{\partial W} = \boxed{\frac{\partial L}{\partial \hat{y}}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$

The derivative of the Loss Function with respect to $\hat{y}$ is very easy and can be calculated once for all because it does not depend on the particular layer:

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \Rightarrow$$

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

Gradient Calculation

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \boxed{\frac{\partial \hat{y}}{\partial Z}} \frac{\partial Z}{\partial W}$$

Hidden Layer Activation Function (Hyperbolic Tangent)

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \Rightarrow \frac{d}{dx} \tanh x = 1 - (\tanh x)^2 \tag{5}$$

Output Layer Activation Function (Sigmoid Function)

$$\sigma(x) = \left[\frac{1}{1 + e^{-x}}\right] \Rightarrow \frac{d}{dx}\sigma(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{6}$$

Gradient Calculation

Output Layer

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} = \sigma(Z^{[2]}) \cdot (1 - \sigma(Z^{[2]})) = A^{[2]}(1 - A^{[2]})$$

$$\frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]}$$

So the complete gradient is:

$$\frac{\partial L}{\partial W^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})} \cdot A^{[2]}(1 - A^{[2]}) \cdot {A^{[1]}}^{T}$$
$$= (A^{[2]} - y) \cdot {A^{[1]}}^{T}$$

Gradient Calculation

Hidden Layer

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} = \sigma(Z^{[2]}) \cdot (1 - \sigma(Z^{[2]})) = A^{[2]}(1 - A^{[2]})$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = ?$$

So the complete gradient is:

$$\begin{aligned}\frac{\partial L}{\partial W^{[1]}} &= \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})} \cdot A^{[2]}(1 - A^{[2]}) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}} \\ &= (A^{[2]} - y) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}}\end{aligned}$$

Gradient Calculation

Hidden Layer Now we have to calculate

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}}$$

Remember that

$$Z^{[2]} = W^{[2]} \cdot tanh\left(W^{[1]} \cdot X + b^{[1]}\right) + b^{[2]}$$

and

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot \frac{\partial\, tanh(\dots)}{\partial W^{[1]}} \cdot X = W^{[2]} \cdot (1 - tanh^2(\dots)) \cdot X$$

Gradient Calculation

Hidden Layer

Finally

$$\begin{aligned}\frac{\partial L}{\partial W^{[1]}} &= (A^{[2]} - y) \cdot W^{[2]} \cdot X \cdot \left(1 - tanh^2(\dots)\right) \\ &= (A^{[2]} - y) \cdot W^{[2]} \cdot X \cdot \left(1 - {A^{[1]}}^2\right)\end{aligned}$$

Weights Update

Output Layer

Since

$$\frac{\partial L}{\partial W^{[2]}} = (A^{[2]} - y) \cdot {A^{[1]}}^T \tag{7}$$

We have

$$\Delta W^{[2]} = \frac{1}{m}\left[A^{[2]} - Y\right]{A^{[1]}}^T = \frac{1}{m}\Delta^{[2]}{A^{[1]}}^T \tag{8}$$

$$\Delta b^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \tag{9}$$

Where

$$\Delta^{[2]} = A^{[2]} - Y \tag{10}$$

Weights Update

Hidden Layer

$$
\begin{aligned}
\Delta W^{[1]} &= \frac{1}{m}\left[A^{[2]} - Y\right] \cdot X^T \cdot W^{[2]T} \cdot (1 - A^{[1]^2}) \\
&= \frac{1}{m}\Delta^{[2]} \cdot W^{[2]T} \cdot (1 - A^{[1]^2}) \cdot X^T \\
&= \frac{1}{m}\Delta^{[1]} \cdot X^T
\end{aligned} \tag{11}
$$

$$
\Delta b^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \tag{12}
$$

Where

$$
\Delta^{[1]} = \Delta^{[2]} \cdot W^{[2]T} \cdot (1 - A^{[1]^2})
$$

Weights Update

```
def backward(params, cache, X, Y):
    m = X.shape[1]
    W1 = params['W1']
    W2 = params['W2']
    A1 = cache['A1']
    A2 = cache['A2']
    DL2 = A2 - Y
    dW2 = (1 / m) * np.dot(DL2, A1.T)
    db2 = (1 / m) * np.sum(DL2, axis=1, keepdims=True)
    DL1 = np.multiply(np.dot(W2.T, DL2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(DL1, X.T)
    db1 = (1 / m) * np.sum(DL1, axis=1, keepdims=True)
    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

Introduction to Keras

Introduction to keras

What is Keras?

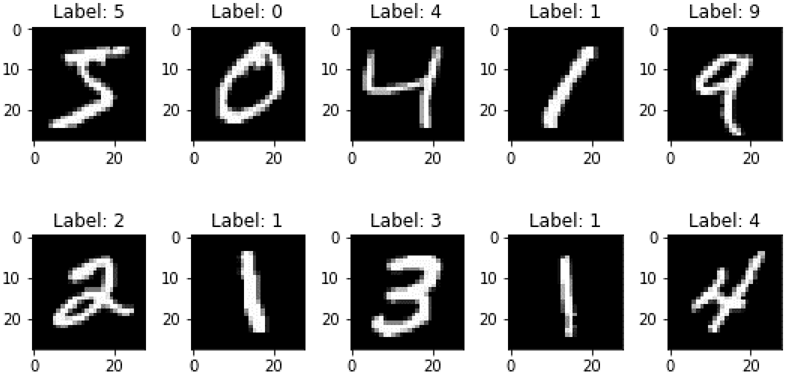
- Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks.
- Its documentation (or specification) is available at https://keras.io.
- It was developed by François Chollet as part of a research project and released as an open source project in March 2015.
- It quickly gained popularity owing to its ease-of-use, flexibility and beautiful design.
- To perform the heavy computations required by neural networks, keras-team relies on a computation backend. At the present, you can choose from three popular open source deep learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK) or Theano.

Introduction to keras

- The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9).
- The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied.
- It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.

Introduction to keras

- As we have said, in the MNIST dataset each digit is stored in a grayscale image with a size of 28x28 pixels.
- In the following you can see the first 10 digits from the training set:



Introduction to keras: Loading the dataset

- Keras provides seven different datasets, which can be loaded in using Keras directly.
- These include image datasets as well as a house price and a movie review datasets.
- The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays

```
import keras

from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Introduction to keras

The typical Keras workflows

- Define your training data: input tensor and target tensor
- Define a network of layers(or model) that maps input to our targets.
- Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
- Iterate your training data by calling the fit() method of your model.

Introduction to keras: Layers

- This is the building block of neural networks which are stacked or combined together to form a neural network model.
- It is a data-preprocessing module that takes one or more input tensors and outputs one or more tensors.
- These layers together contain the network's knowledge.
- Different layers are made for different tensor formats and data processing.

Creating a model with the sequential API

- The easiest way of creating a model in Keras is by using the sequential API, which lets you stack one layer after the other.
- The problem with the sequential API is that it doesn't allow models to have multiple inputs or outputs, which are needed for some problems.
- Nevertheless, the sequential API is a perfect choice for most problems.

```
from keras import models
from keras import layers

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

Introduction to keras

Let's go through this code line by line:

- The first line creates a Sequential model.
- This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially.
- This is called the **sequential** API.

```
from keras import models
from keras import layers

network = models.Sequential()
```

Introduction to keras

Next, we build the first layer and add it to the model.

- It is **Dense** hidden layer with 512 neurons.
- It will use the ReLU activation function.
- Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs.
- It also manages a vector of bias terms (one per neuron).
- When it receives some input data, it computes

$$\phi\left(Z^{[1]} = W^{[1]} \cdot X + B^{[1]}\right), \quad \phi(z) = ReLU(z)$$

```
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

Introduction to keras

Finally, we add a Dense output layer with 10 neurons (one per class).

- Using a 10-way "softmax" layer means that it will return an array of 10 probability scores (summing to 1).
- Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

```
network.add(layers.Dense(10, activation='softmax'))
```

Introduction to keras

- The model's summary() method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters.

```
network.summary()
```

Introduction to keras

- The summary ends with the total number of parameters, including trainable and non-trainable parameters.
- Here we only have trainable parameters.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

Compile a Model

- Before we can start training our model we need to configure the learning process.
- For this, we need to specify an optimizer, a loss function and optionally some metrics like accuracy.
- The **loss function** is a measure on how good our model is at achieving the given objective.
- An **optimizer** is used to minimize the loss(objective) function by updating the weights using the gradients.

Loss Function

- Choosing the right Loss Function for the problem is very important, the neural network can take any shortcut to minimize the loss.
- So, if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

Loss Function

For common problems like Classification, Regression and Sequence prediction, they are simple guidelines to choose a loss function.

For:

- Two- Class classification you can choose binary cross-entropy
- Multi-Class Classification you can choose Categorical Cross-entropy.
- Regression Problem you can choose Mean-Squared Error

Activation Function

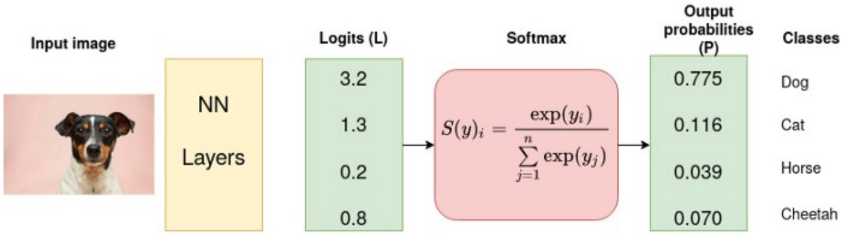
Softmax Activation Function

- Softmax is an activation function that scales numbers/logits into probabilities.
- The output of a Softmax is a vector (say v) with probabilities of each possible outcome.
- The probabilities in vector v sums to one for all possible outcomes or classes
- It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes
- You can think of softmax function as a sort of generalization to multiple dimension of the logistic function

Activation Function

Softmax Activation Function. Mathematically, softmax is defined as

$$S(y)_i = \frac{\exp y_i}{\sum_{j=1}^{n} \exp y_j} \tag{13}$$



Loss Function

Categorical Cross-Entropy

- Remember the logistic Loss Function:

$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) = -\sum_{j=1}^{2} y_j \log \hat{y}_j$$

- Generalizing this result to the n-class classification problem we have

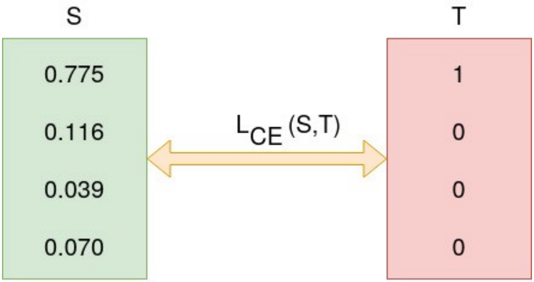
$$L = -\sum_{j=1}^{N_C} y_j \log \hat{y}_j \qquad (14)$$

This last equation define the so called *cross-entropy*

Loss Function

Categorical Cross-Entropy

- In the previous example, Softmax converts logits into probabilities.
- The purpose of the Cross-Entropy is to take the output probabilities (P) and measure the distance from the truth values



Loss Function

Categorical Cross-Entropy

- The categorical cross-entropy is computed as follows

$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

Optimizer

RMSProp

- Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.
- A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable.
- AdaGrad is an extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) over the course of the search.
- A limitation of AdaGrad is that it can result in a very small step size for each parameter by the end of the search that can slow the progress of the search down too much and may mean not locating the optima.
- Root Mean Squared Propagation, or RMSProp, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter.
- The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

Compile the model

So, to make our network ready for training, we need to pick three things, as part of "compilation" step:

- A loss function: the is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An optimizer: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
- Metrics to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).

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

Introduction to keras: Training

- Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the '[0, 1]' interval.
- Previously, our training images for instance were stored in an array of shape '(60000, 28, 28)' of type 'uint8' with values in the '[0, 255]' interval.
- We transform it into a 'float32' array of shape '(60000, 28 * 28)' with values between 0 and 1.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Introduction to keras: Training

We also need to categorically encode the labels:

```
from keras.utils.np_utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Introduction to keras: Training

What is an epoc?

- An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. Datasets are usually grouped into batches (especially when the amount of data is very large). Some people use the term iteration loosely and refer to putting one batch through the model as an iteration.
- If the batch size is the whole training dataset then the number of epochs is the number of iterations. For practical reasons, this is usually not the case. Many models are created with more than one epoch. The general relation where dataset size is d, number of epochs is e, number of iterations is i, and batch size is b would be $d \cdot e = i \cdot b$.
- Determining how many epochs a model should run to train is based on many parameters related to both the data itself and the goal of the model, and while there have been efforts to turn this process into an algorithm, often a deep understanding of the data itself is indispensable.

Introduction to keras: Training

We are now ready to train our network, which in Keras is done via a call to the ‘fit‘ method of the network: we ”fit” the model to its training data.

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

Visualizing the training process

- We can visualize our training and testing accuracy and loss for each epoch so we can get intuition about the performance of our model.
- The accuracy and loss over epochs are saved in the history variable we got whilst training and we will use Matplotlib to visualize this data.

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```