

# Vibrastic 101

## Artificial Intelligence Crash Course

Tirtadwipa Manunggal

Machine Learning Enthusiast | Backend Engineer

[tirtadwipa.manunggal@gmail.com](mailto:tirtadwipa.manunggal@gmail.com)

# Overview

- Introduction
- All about Machine Learning
- Let's build our own

# Course Organization

- Theory
- Challenge
- Project
- Question Answer

# Refreshing

Python main characteristics:

- dynamic type system
- interpreted (actually: compiled to bytecode, `*.pyc` files)
- multi-paradigm: imperative, procedural, object-oriented, (functional), *literate*; do whatever you want
- **indentation is important!**
- Python is a high-level, dynamically typed multiparadigm programming language.
- Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable.

## Refreshing (cntd.)

This course **assumes** that you have some programming experience at least:

- Java (static type system, compiled, object-oriented, verbose)
- C/C++ (static type system, compiled, multi-paradigm, low-level)
- Matlab? R?

In this *refreshing*, we're gonna review:

- Basic Python: Basic data types, containers, loops, functions and classes.
- **Pytorch** highlight

# Basic Python : Data types

## Numeric types

- Integers and floats work as you would expect from other languages:

```
x = 3; print(x, type(x))  
y = 2.5; print(type(y))
```

```
print(x, x + 1, x - 1, x * 2, x ** 2)  
print(y, y + 1, y * 2, y ** 2)
```

```
x += 1 # added to 4  
x *= 2 # multiplied to 8
```

## Boolean

```
t, f, aa, bb = True, False, True, False
print(t, f, type(t))
```

```
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
day = "Sunday"
if day == 'Sunday':
    print('Sleep!!!')
else:
    print('Go to work')
```

# String

```
hello = 'hello'  
world = "world"  
print(hello, len(hello))
```

```
hw = hello + ' ' + world  # String concatenation  
print(hw)
```

```
hw12 = '%s %s! your number is: %d' % (hello, world, 12)  # sprintf style string formatting  
print(hw12)
```

```
s = "hello"  
print(s.capitalize())  
print(s.upper())  
print(s.replace('l', '(ell)'))  
print('  world '.strip())
```



# List

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])
print(xs[-1])       # Count from the end of the list
```

```
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)
```

```
xs.append('bar')    # Add a new element to the end of the list
print(xs)
```

```
xs = xs + ['thing1', 'thing2'] # Adding lists (the += op works too)
print(xs)
```

```
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)
```

# Slicing

```
nums = list(range(5)) # range is a built-in function (more on this later)
print(nums)
```

```
print(nums[2:4])      # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
```

```
print(nums[2:])       # Get a slice from index 2 to the end; prints "[2, 3, 4]"
```

```
print(nums[:2])       # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
```

```
print(nums[:])        # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
```

```
print(nums[:-1])      # Slice indices can be negative; prints "[0, 1, 2, 3]"
```

```
nums[2:4] = [8, 9]    # Assign a new sublist to a slice
print(nums)           # Prints "[0, 1, 8, 9, 4]"
```

# Loops

- Basic loop

```
for i in range(10):  
    print(i)
```

- You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']  
for animal in animals:  
    aa = animal + ' :)'  
    print(aa)
```

- If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```
animals = ['cat', 'dog', 'monkey']  
for idx, animal in enumerate(animals):  
    print('Item number %d is a %s' % (idx + 1, animal))
```

# Challenge

## Write loops to draw triangle!

```
h = 3
```

```
*  
**  
***
```

```
h = 5
```

```
*  
**  
***  
****  
*****
```

## List comprehension

- When programming, frequently we want to transform one type of data into another. For example, consider the following code that computes square numbers:

```
nums, squares = [0, 1, 2, 3, 4], []  
for x in nums:  
    squares.append(x ** 2)
```

- You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]  
squares = [x ** 2 for x in nums]
```

- List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]  
even_squares = [x ** 2 for x in nums if x % 2 == 0]
```

# Dictionaries

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some data
```

```
print(d['cat'])          # Get an entry from a dictionary; prints "cute"  
print('cat' in d)
```

```
d['fish'] = 'wet'        # Set an entry in a dictionary  
print(d['fish'])         # Prints "wet"
```

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}  
for animal in d:  
    legs = d[animal]  
    print('A %s has %d legs' % (animal, legs))
```

# Basic Python : Functions

Python functions are defined using the `def` keyword. For example:

```
def sign(x):  
    if x > 0:  
        return 'positive'  
    elif x < 0:  
        return 'negative'  
    else:  
        return 'zero'
```

```
for x in [-1, 0, 1]:  
    print(sign(x))
```

## Function (cntd.)

We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):  
    if loud:  
        print('HELLO, %s' % name.upper())  
    else:  
        print('Hello, %s!' % name)
```

```
hello('Bob')  
hello('Fred', loud=True)
```



# Challenge

Write this as a python function  $\rightarrow \sum_i i^2$

```
def sigma(i):  
    bla bla bla...
```

```
sigma(2) # will print 5
```

# Basic Python : Classes and object oriented programming

- The syntax for defining classes in Python is straightforward.
- Remember to include `self` as the first parameter of the class methods.

```
class Greeter():  
    # Constructor  
    def __init__(self, name):  
        self.name = name # Create an instance variable  
  
    # Instance method  
    def greet(self, loud=False):  
        if loud:  
            print('HELLO, %s!' % self.name.upper())  
        else:  
            print('Hello, %s' % self.name)
```

```
g = Greeter('Fred') # Construct an instance of the Greeter class  
g.greet()           # Call an instance method; prints "Hello, Fred"  
g.greet(loud=True)  # Call an instance method; prints "HELLO, FRED!"
```

- Inheritance

```
class Question(Greeter):  
    def __init__(self, name):  
        super(Question, self).__init__(name)  
  
    def ask(self):  
        print('Are you %s?' % self.name)
```

```
q = Question('Fred')  
q.ask()
```

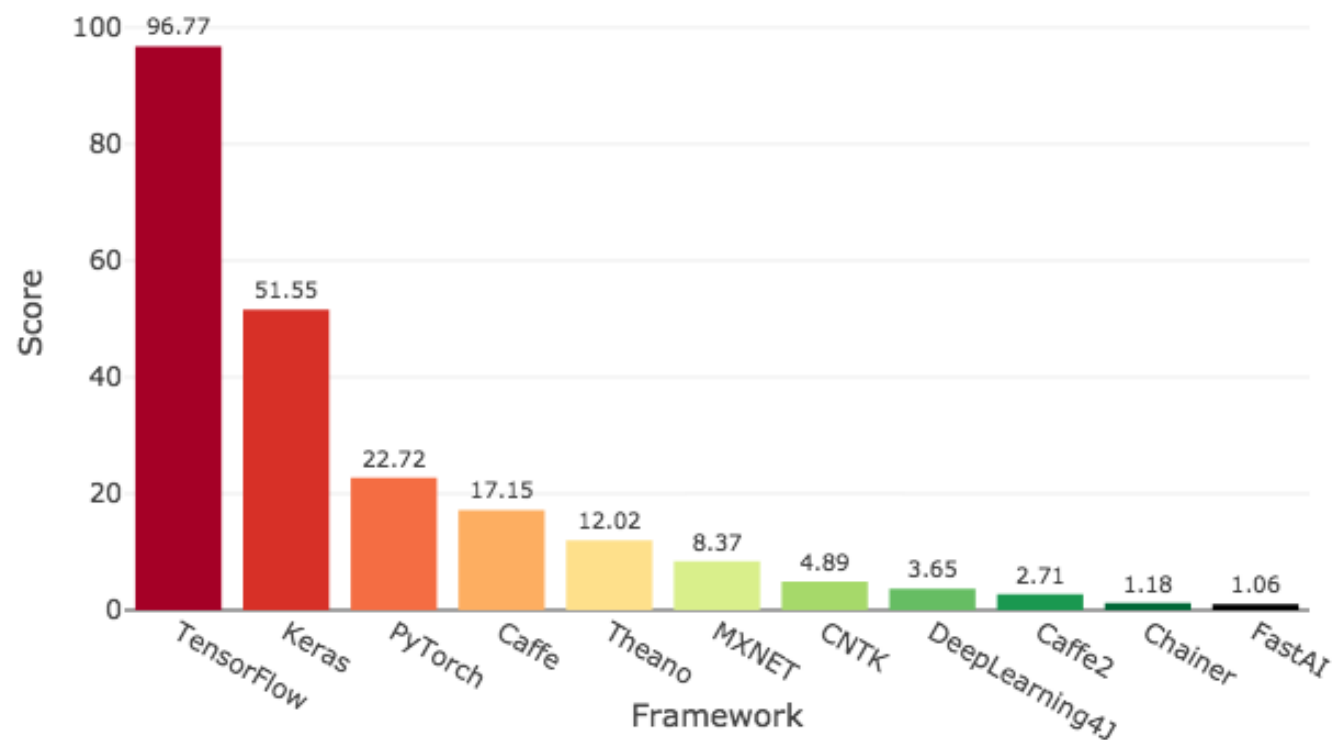
# Basic Python : Import statement

- We have seen already the `import` statement in action.
- Python has a huge number of libraries included with the distribution.
- Most of these variables and functions are not accessible from a normal Python interactive session.
- Instead, you have to import them.
- You can also make your own module
- Browse here : <https://pypi.org/>

# Machine Learning Framework

- **TensorFlow** by Google
- **Keras** by Francois Chollet
- **PyTorch** by Facebook

Deep Learning Framework Power Scores 2018



One of PyTorch feature ... that's loved by researchers

## Autograd

- To help us to praise this feature, let's do some **basic math** beforehand
- Solve these !

$$f(x, y) = xy \quad \frac{\partial}{\partial x} f(3) = ?$$

## Solution

$$\frac{\partial}{\partial x} f(x, y) = y \quad \rightarrow \quad \frac{\partial}{\partial x} f(1, 2) = 2$$

Quite easy, right?

# Challenge

How about...

$$g(x, y) = xy^2 - x^2y \quad \frac{\partial}{\partial x}g(2, 5) = ?$$



# Here is Pytorch come to the play!

- Import the library

```
import torch
```

- Declare our variable and function

```
x = torch.tensor(1.0, requires_grad = True)  
y = torch.tensor(2.0)  
f = x*y
```

- Get our number

```
f.backward()  
print(x.grad.data)
```

# Challenge

**Solve this with Pytorch**

$$g(x, y) = xy^2 - x^2y \quad \frac{\partial}{\partial x}g(2, 5) = ?$$

# Now we're set!

## Preparation

- Open Google Colab
- Put this on the cell in case we need them during the course

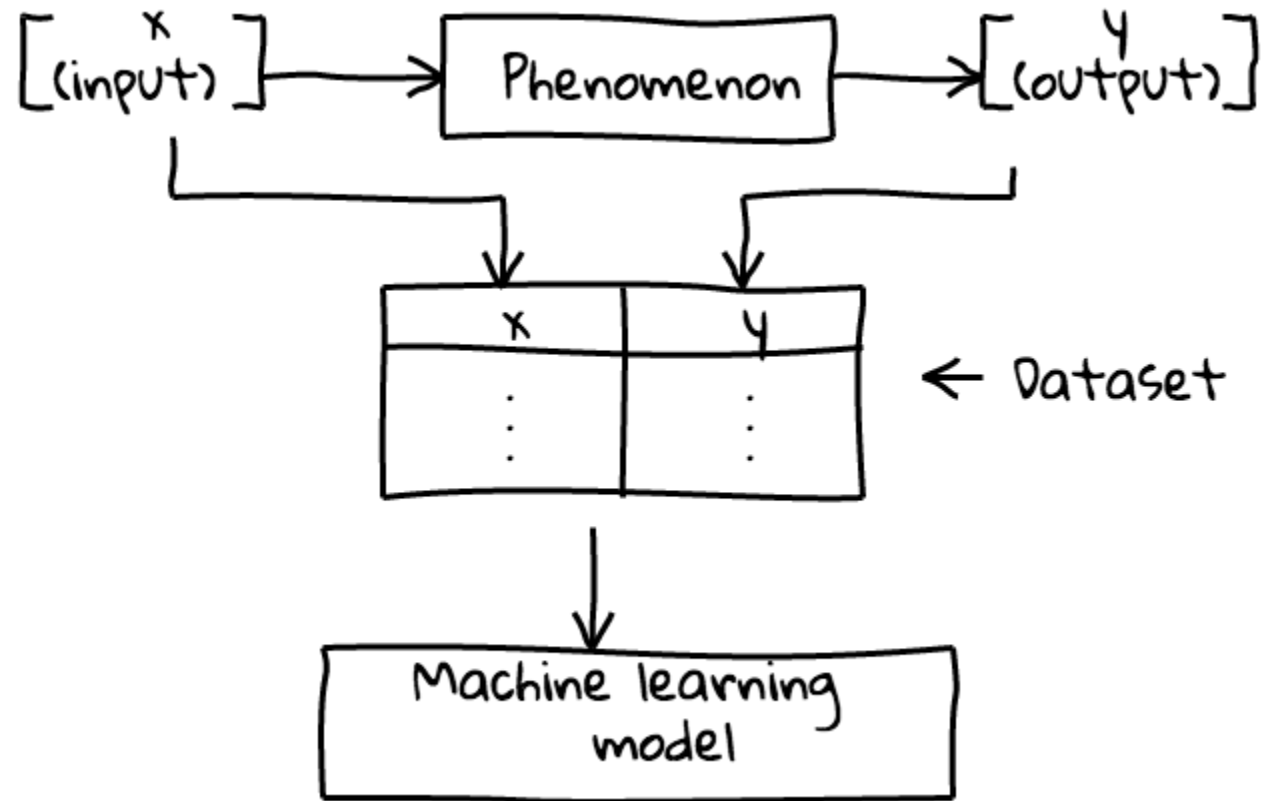
```
import random, math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

# Hold on!

## Definition

- **Machine Learning (ML)** : A subset of artificial intelligence involved with the creation of algorithms which can modify itself without human intervention to produce desired output- by feeding itself through structured data.
- **Deep Learning** : Same, but has numerous layers

## Definition (cntd.)



# Highlight

- **Learning** : Construction and study of systems that can learn from data.
- **Adaptation** : The capacity to adapt implies to be able to modify what has been learn in order to cope with those modifications.
- **Flexibility and robustness** : Self-organization
- **Provide explanations** : Explanations are necessary to validate and find directions for improvement.
- **Discovery/creativity** : Capacity of discovering processes and/or relations previously unknown

## In other words

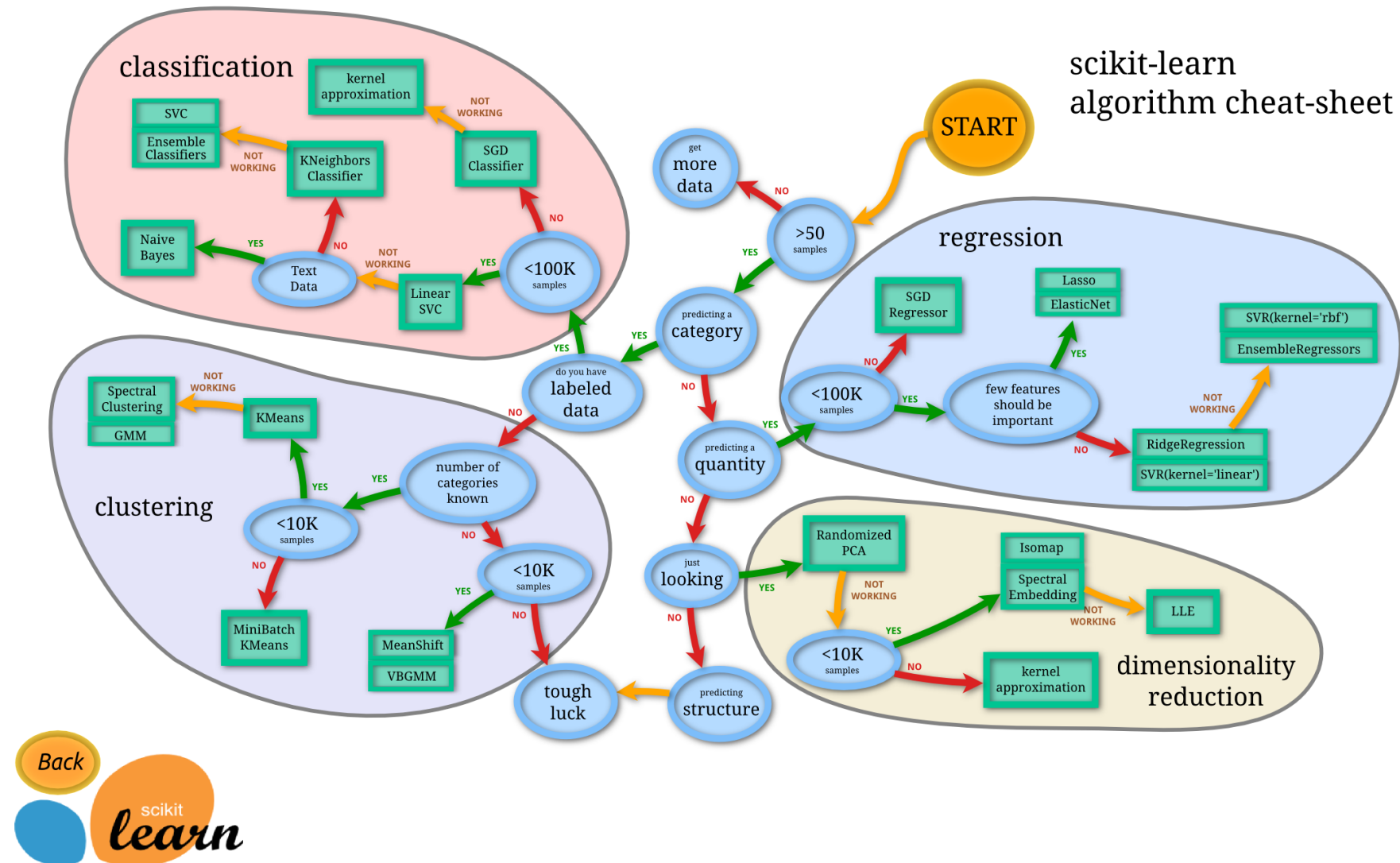
- Having a **process**  $\vec{F} : \mathcal{D} \rightarrow \mathcal{I}$  that **transforms** a given  $\vec{x} \in \mathcal{D}$  in a  $\vec{y}$ .
- Construct on a dataset  $\Psi = \{\langle \vec{x}_i, \vec{y}_i \rangle\}$  with  $i = 1, \dots, N$ .
- Each  $\langle \vec{x}_i, \vec{y}_i \rangle$  represents an **input** and its corresponding **expected output**:  $\vec{y}_i = \vec{F}(\vec{x}_i)$ .
- **Optimize** a **model**  $\mathcal{M}(\vec{x}; \vec{\theta})$  by adjusting its parameters  $\vec{\theta}$ .
  - Make  $\mathcal{M}()$  to be as similar as possible to  $\vec{F}()$  by optimizing one or more error (loss) functions.

# Classification of ML

- **Classification:**  $\vec{F} : \mathcal{D} \rightarrow \{1, \dots, k\}$ ;  $\vec{F}(\cdot)$  defines 'categories' or 'classes' labels.
- **Regression:**  $\vec{F} : \mathbb{R}^n \rightarrow \mathbb{R}$ ; it is necessary to predict a real-valued output instead of categories.
- **Clustering:** group a set of objects in such a way that objects in the same group (*cluster*) are more *similar* to each other than to those in other groups (clusters).
- **Synthesis:** generate new examples that are similar to those in the training data

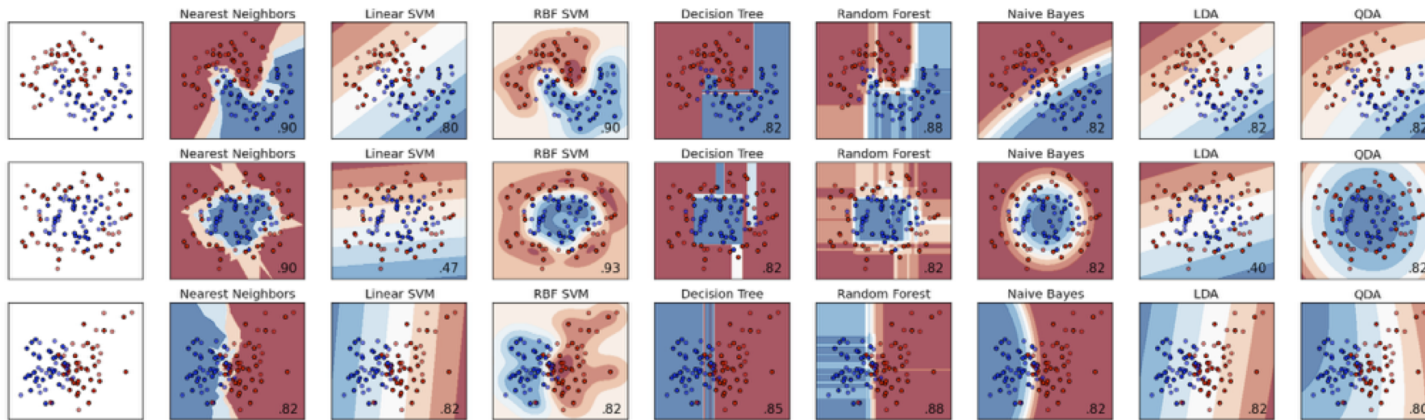


# Classification of ML (cntd.)



# Classification of ML (cntd.)

## Many ML methods



- different assumptions on data
- different scalability profiles at training time
- different latencies at prediction time
- different model sizes (embedability in mobile devices)

## Another Classification of ML

- **Supervised Learning** : Allows you to collect data or produce a data output from the previous experience.
- **Unsupervised Learning** : Finds all kind of unknown patterns in data.
- **Reinforced Learning** : It can be understood using the concepts of agents, environments, states, actions and rewards.

## Supervised Learning hands on

- Sometimes we can observe the pairs  $\langle \vec{x}_i, \vec{y}_i \rangle$ :
- We can use the  $\vec{y}_i$ 's to provide a *scalar feedback* on how good is the model  $\mathcal{M}(\vec{x}; \vec{\theta})$ .
- That feed back is known as the *loss function*.
- Modify parameters  $\vec{\theta}$  as to improve  $\mathcal{M}(\vec{x}; \vec{\theta}) \rightarrow \textit{learning}$ .

# Supervised Learning hands on

- import library

```
import random
import numpy as np
import matplotlib.pyplot as plt
```

- replicable random seed

```
random.seed(42)
```

- create input

```
x = np.arange(100)
```

## Supervised Learning hands on

- let's suppose that we have a phenomenon such that  $y_{\text{real}} = \sin\left(\frac{\pi x}{50}\right)$

```
y_real = np.sin(x*np.pi/50)
```

- introducing some uniform random noise to simulate measurement noise

```
y_measured = y_real + (np.random.rand(100) - 0.5)
```

- plot the real vs measured

```
plt.scatter(x, y_measured, marker='.', color='b', label='measured')  
plt.plot(x, y_real, color='r', label='real')  
plt.xlabel('x'); plt.ylabel('y'); plt.legend(frameon=True);
```

# Supervised Learning hands on

- let's use one of supervised method : Support Vector Machine

```
from sklearn.svm import SVR  
clf = SVR() # using default parameters
```

- training

```
clf.fit(x.reshape(-1, 1), y_measured)
```

- predicting the output

```
y_pred = clf.predict(x.reshape(-1, 1))
```

## Supervised Learning hands on

- plotting the result

```
plt.scatter(x, y_measured, marker='.', color='blue', label='measured')  
plt.plot(x, y_pred, 'g--', label='predicted')  
plt.xlabel('X'); plt.ylabel('y'); plt.legend(frameon=True);
```

- We observe for the first time an important negative phenomenon: overfitting.



# Supervised Learning hands on

- We will be dedicating part of the course to the methods that we have for control overfitting.

```
clf = SVR(C=1e3, gamma=0.0001)
clf.fit(x.reshape(-1, 1), y_measured)
```

- predicting the output

```
y_pred_ok = clf.predict(x.reshape(-1, 1))
```

- plotting the result

```
plt.scatter(x, y_measured, marker='.', color='b', label='measured')
plt.plot(x, y_pred, 'g--', label='overfitted')
plt.plot(x, y_pred_ok, 'm-', label='not overfitted')
plt.xlabel('X'); plt.ylabel('y'); plt.legend(frameon=True);
```

# Unsupervised Learning

In some cases we can just observe a series of items or values, e.g.,  $\Psi = \{\vec{x}_i\}$ :

- It is necessary to find the *hidden structure of unlabeled data*.
- We need a measure of correctness of the model that does not requires an expected outcome.
- Although, at first glance, it may look a bit awkward, this type of problem is very common.
- Related to anomaly detection, clustering, etc.

# Unsupervised Learning hands on

- Let's generate a dataset that is composed by three groups or clusters of elements,  $\vec{x} \in \mathbb{R}^2$ .

```
x_1 = np.random.randn(30, 2) + (5, 5)
x_2 = np.random.randn(30, 2) + (10, 0)
x_3 = np.random.randn(30, 2) + (0, 2)
```

- See the plot

```
plt.scatter(x_1[:, 0], x_1[:, 1], c='red', label='Cluster 1', alpha =0.74)
plt.scatter(x_2[:, 0], x_2[:, 1], c='blue', label='Cluster 2', alpha =0.74)
plt.scatter(x_3[:, 0], x_3[:, 1], c='green', label='Cluster 3', alpha =0.74)
plt.legend(frameon=True); plt.xlabel('$x_1$'); plt.ylabel('$x_2$');
plt.title('Three datasets');
```

## Unsupervised Learning hands on

- Merge all data

```
x = np.concatenate(( x_1, x_2, x_3), axis=0)
```

- See the plot

```
plt.scatter(x[:,0], x[:,1], c='m', alpha =0.74)  
plt.title('Training dataset');
```

# Unsupervised Learning hands on

- let's use one of the unsupervised method : KMeans

```
from sklearn.cluster import KMeans
clus = KMeans(n_clusters=3)
```

- fit and predict the data

```
clus.fit(x)
labels_pred = clus.predict(x)
```

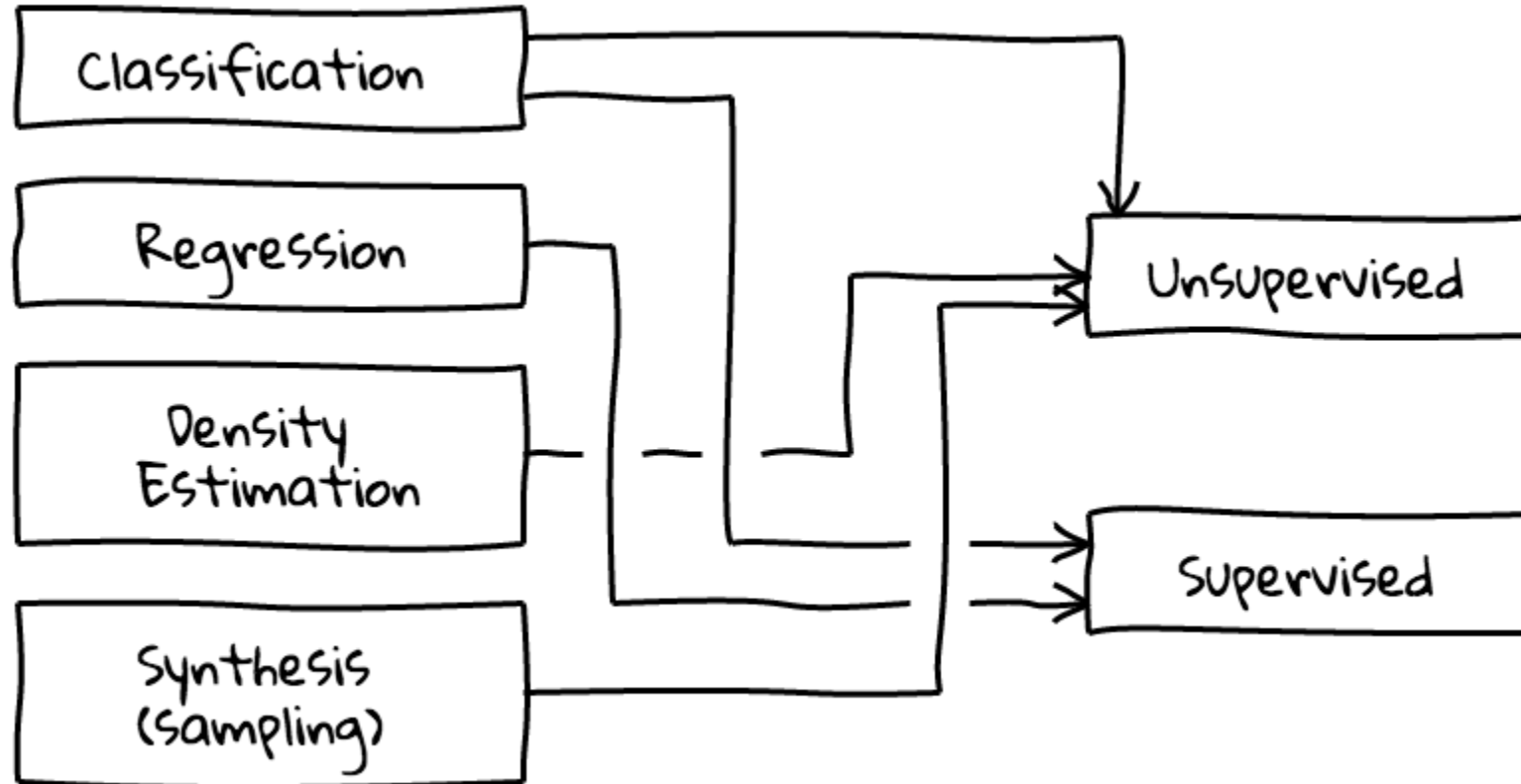
- See the plot

```
cm=iter(plt.cm.Set1(np.linspace(0,1,len(np.unique(labels_pred)))))
for label in np.unique(labels_pred):
    plt.scatter(x[labels_pred==label][:,0], x[labels_pred==label][:,1],
                c=next(cm), alpha =0.74, label='Pred. cluster ' +str(label+1))
plt.legend(loc='upper right', bbox_to_anchor=(1.45,1), frameon=True);
plt.xlabel('$x_1$'); plt.ylabel('$x_2$'); plt.title('Clusters predicted');
```

# Reinforced Learning

- Inspired by behaviorist psychology;
- How to take actions in an environment so as to maximize some notion of cumulative reward?
- Differs from standard supervised learning in that correct input/output pairs are never presented,
- ...nor sub-optimal actions explicitly corrected.
- Involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge)
- see : <https://www.youtube.com/watch?v=yEOEqaEgu94> (4 minutes view)

## Remark



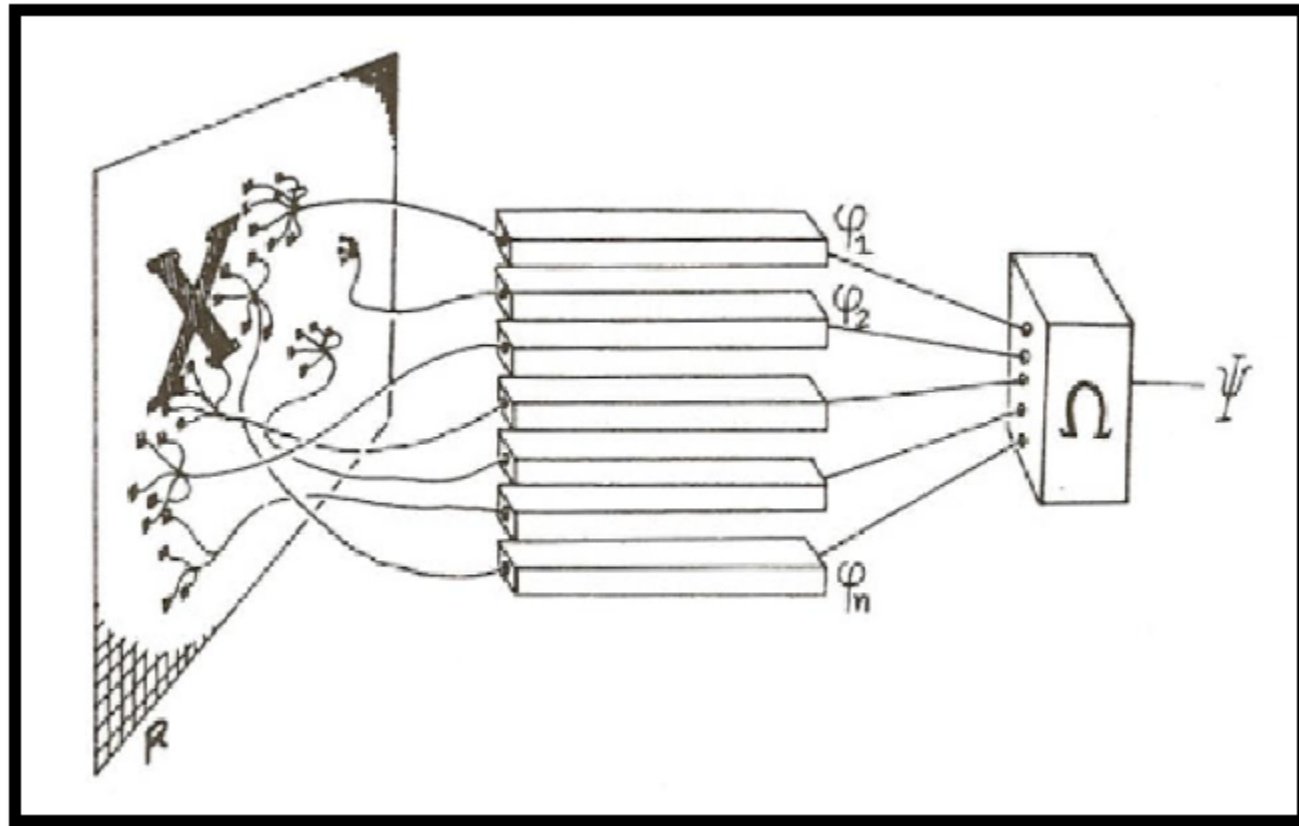
## Let's focus on Artificial Neural Network

- Not all methods are applicable in real life
- Most featured method
- Nature inspired
- Efficient computation
- Evolutionary optimization



# Artificial Neural Network

## Artificial neuron



# Artificial Neural Network

## Artificial neuron as a neuron abstraction

In general terms, an input  $\vec{x} \in \mathbb{R}^n$  is multiplied by a weight vector  $\vec{w}$  and added a bias  $b$  producing the net activation,  $\text{net}$ .  $\text{net}$  is passed to the *activation function*  $f()$  that computed the neuron's output  $\hat{y}$ .

$$\hat{y} = f(\text{net}) = f(\vec{w} \cdot \vec{x} + b) = f\left(\sum_{i=1}^n w_i x_i + b\right).$$

# Artificial Neural Network

## The perceptron

The [Perceptron](#) and its learning algorithm pioneered the research in neurocomputing.

- The perceptron is an algorithm for learning a linear binary classifier.
- That is a function that maps its input  $\vec{x} \in \mathbb{R}^n$  (a real-valued vector) to an output value  $f(\vec{x})$  (a single binary value) as,

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0, \\ 0 & \text{otherwise;} \end{cases}$$

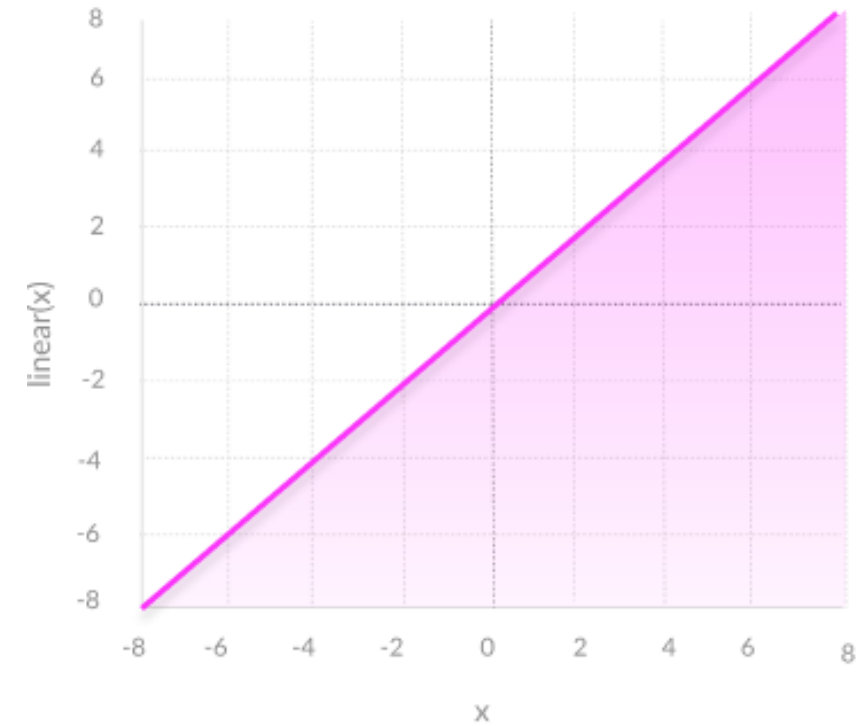
where  $\vec{w}$  is a vector of real-valued *weights*,  $\vec{w} \cdot \vec{x}$  is the *dot product*  $\sum_{i=1}^n w_i x_i$ , and  $b$  is known as the *bias*.

## Activation functions

### Linear Function `nn.Linear`

$$f(x) = x + b$$

- Disadvantages
  - Not possible to use backpropagation
  - All layers of the neural network collapse into one
  - Limited power to handle complexity

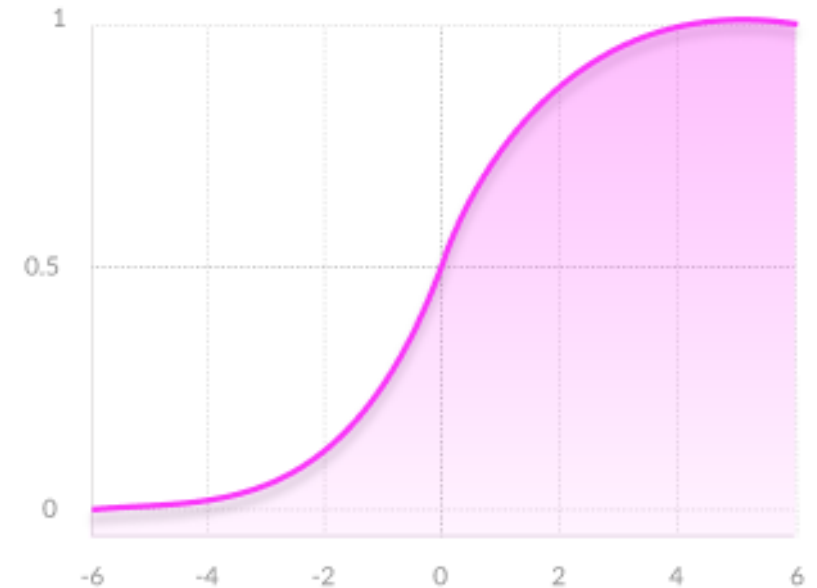


# Activation functions

## Sigmoid Function `nn.Sigmoid`

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

- Advantages
  - Smooth gradient
  - Clear predictions
- Disadvantages
  - Vanishing gradient
  - Outputs not zero centered
  - Computationally expensive

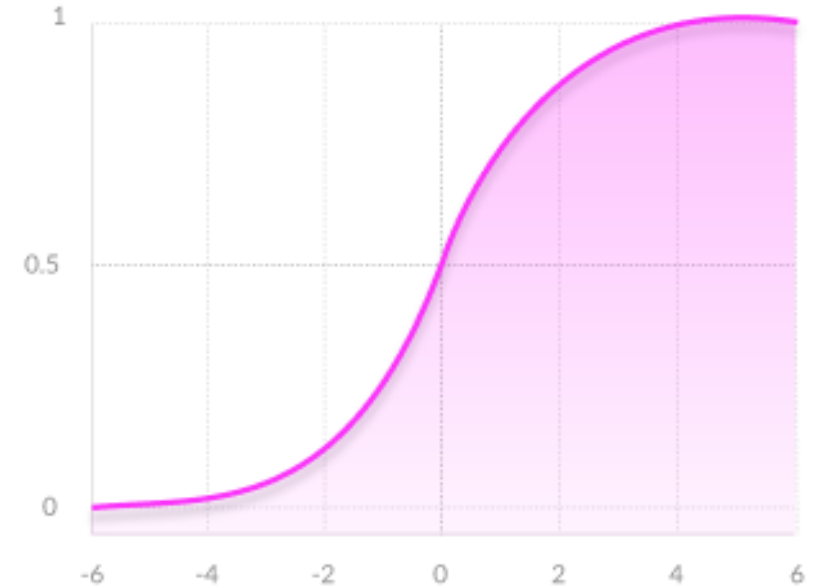


## Activation functions

### Hyperbolic Tangent Function `nn.Tanh`

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

- Advantages
  - Zero centered : strong negative, neutral, and positive values.
  - Otherwise like the Sigmoid function.
- Disadvantages
  - Like the Sigmoid function

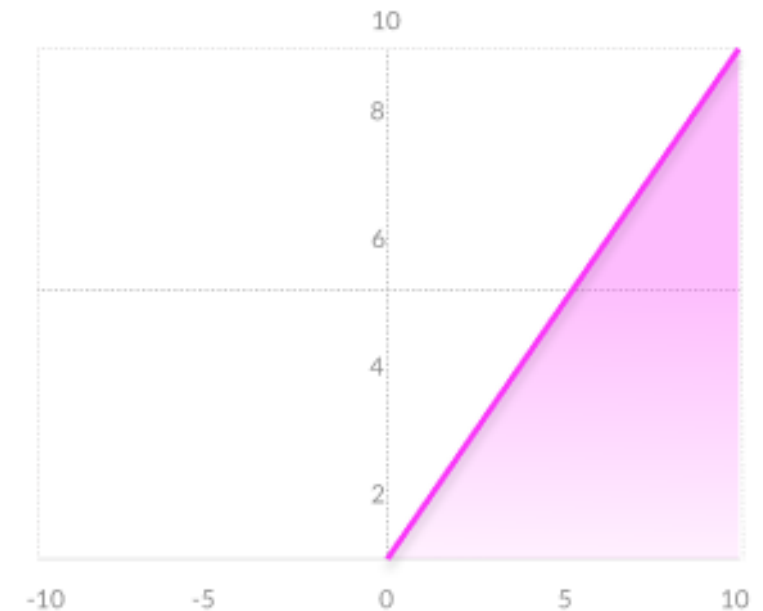


# Activation functions

## ReLU Function `nn.ReLU`

$$f(x) = \max(0, x)$$

- Advantages
  - Computationally efficient : network converge very quickly
  - Non-linear : allows for backpropagation
- Disadvantages
  - The Dying ReLU problem

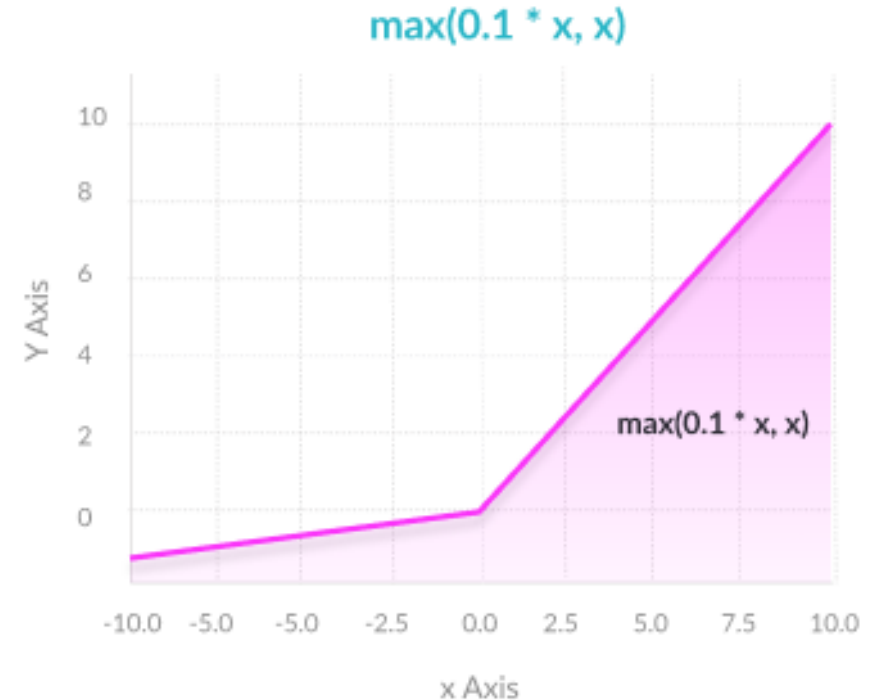


## Activation functions

### Leaky ReLU Function `nn.LeakyReLU`

$$f(x) = \max(0, x)$$

- Advantages
  - Prevents dying ReLU problem
  - Otherwise like ReLU
- Disadvantages
  - Results not consistent



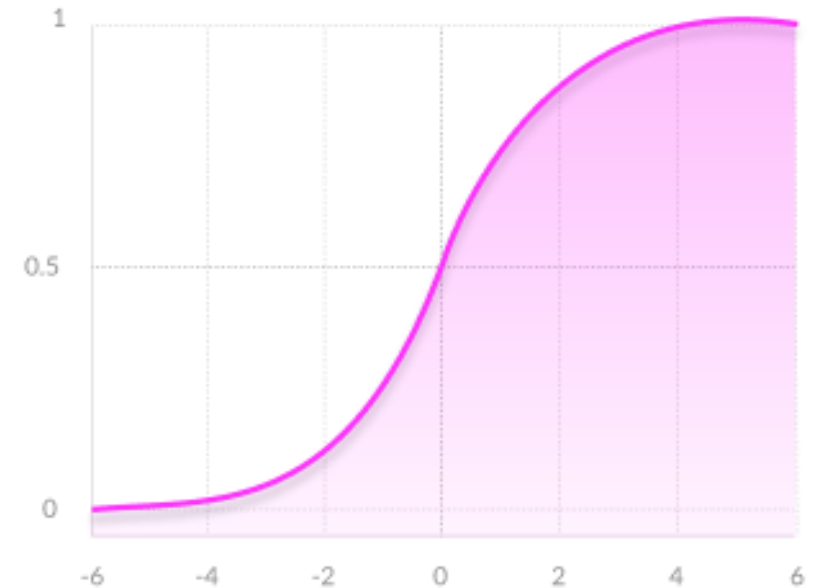


## Activation functions

### Softmax Function `nn.Softmax`

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- Advantages
  - Able to handle multiple classes only one class in other activation functions
  - normalizes the outputs for each class between 0 and 1
  - Useful for output neurons



# Perceptron learning

Learning goes by calculating the prediction of the perceptron,  $\hat{y}$ , as

$$\hat{y} = f(\vec{w} \cdot \vec{x} + b) = f(w_1x_1 + w_2x_2 + \dots + w_nx_n + b) .$$

After that, we update the weights and the bias using the perceptron rule:

$$\begin{aligned} w_i &= w_i + \alpha(y - \hat{y})x_i, \quad i = 1, \dots, n; \\ b &= b + \alpha(y - \hat{y}). \end{aligned}$$

Here  $\alpha \in (0, 1]$  is known as the *learning rate*. Or can be further enhanced using momentum :

$$\vec{w}(t + 1) = \vec{w}(t) + \alpha\Delta\vec{w}(t) + \beta\Delta\vec{w}(t - 1),$$

where  $\beta \in \mathbb{R}^+$  is known as the momentum rate.

## Study of Learning Rate $\alpha$ hands on

- error function  $\rightarrow E(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \frac{1}{2} \|\mathbf{X} \cdot \mathbf{w} - \mathbf{y}\|_2^2$  .

```
def error(X, y, w):  
    return 0.5*np.linalg.norm(X.dot(w) - y)**2
```

- the gradient  $\rightarrow \nabla \mathbf{w} = \nabla_{\mathbf{w}} E(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \mathbf{X}^T \cdot (\mathbf{X} \cdot \mathbf{w} - \mathbf{y})$  .

```
def linear_regression_gradient(X, y, w):  
    return X.T.dot(X.dot(w)-y)
```

- gradient descent loop

```
def gradient_descent(X, y, w_0, alpha, max_iters):  
    'Returns the values of the weights as learning took place.'  
    w = np.array(w_0, dtype=np.float64)  
    w_hist = np.zeros(shape=(max_iters+1, w.shape[0]))  
    w_hist[0] = w  
    for i in range(0, max_iters):  
        delta_weights = -alpha*linear_regression_gradient(X_bias, y, w)  
        w += delta_weights  
        w_hist[i+1] = w  
    return w_hist
```

- plot contour

```
def plot_contour(X_data, y_data, bounds, resolution=50, cmap=cm.viridis,
                 alpha=0.3, linewidth=5, rstride=1, cstride=5, ax=None):
    (minx,miny),(maxx,maxy) = bounds

    x_range = np.linspace(minx, maxx, num=resolution)
    y_range = np.linspace(miny, maxy, num=resolution)
    X, Y = np.meshgrid(x_range, y_range)

    Z = np.zeros((len(x_range), len(y_range)))

    for i, w_i in enumerate(x_range):
        for j, w_j in enumerate(y_range):
            Z[j,i] = error(X_data, y_data, [w_i, w_j])
```

(continuing from previous page)

```
if not ax:
    fig = plt.figure(figsize=(6,6))
    ax = fig.gca()
    ax.set_aspect('equal')
    ax.autoscale(tight=True)
cset = ax.contourf(X, Y, Z, 30, cmap=cmap, rstride=rstride,
                  cstride=cstride, linewidth=linewidth, alpha=alpha)
cset = ax.contour(X, Y, Z, 10, cmap=cmap, rstride=rstride,
                  cstride=cstride, linewidth=linewidth)
plt.clabel(cset, inline=1, fontsize=7)
return Z
```

- try initialize variables

```
X = np.array([[0.0], [1.0], [2.0], [3.0], [4.0]])
X_bias = np.hstack((X, np.ones((N, 1))))
y = np.array([10.5, 5.0, 3.0, 2.5, 1.0])

w_0 = [-3, 2]
alpha = 0.05
max_iters = 25
```

- run learning

```
w_hist = gradient_descent(X_bias, y, w_0, alpha, max_iters)
plot_hist_contour(X_bias, y, w_hist, w_norm, title='end='+str(w_hist[-1]), show_legend=True)
```

- function to run learning on several alpha

```
def alphas_study(alphas):  
    fig = plt.figure(figsize=(11,7))  
    for i,alpha in enumerate(alphas):  
        ax = fig.add_subplot(2,3,i+1)  
        w_hist = gradient_descent(X_bias, y , w_0, alpha, max_iters)  
        plot_hist_contour(X_bias, y, w_hist, w_norm, ax=ax, title='$\\alpha='+str(alpha)+'$')  
    plt.legend(scatterpoints=1, ncol=3, bbox_to_anchor=(-0.2,-0.2), frameon=True);  
    plt.tight_layout()
```

- the alpha

```
alphas = np.linspace(0.02,0.07,6)
```

- study the alpha

```
alphas_study(alphas)
```



## Study of Momentum Rate $\beta$ hands on

- gradient descent with momentum

```
def gradient_descent_with_momentum(X, y, w_0, alpha, beta, max_iters):  
    w = np.array(w_0, dtype=np.float64)  
    w_hist = np.zeros(shape=(max_iters+1, w.shape[0]))  
    w_hist[0] = w  
    omega = np.zeros_like(w)  
    for i in range(max_iters):  
        delta_weights = -alpha*linear_regression_gradient(X, y, w) + beta*omega  
        omega = delta_weights  
        w += delta_weights  
        w_hist[i+1] = w  
    return w_hist
```

- set the variables

```
alpha = 0.05  
beta = 0.5  
max_iters = 25
```

- run the momentum learning

```
w_hist = gradient_descent(X_bias, y, (-3,2), alpha, max_iters)  
w_hist_mom = gradient_descent_with_momentum(X_bias,y, (-3,2), alpha, beta, max_iters)
```

- compare plot

```
def comparison_plot():  
    fig = plt.figure(figsize=(9,4.5))  
    ax = fig.add_subplot(121)  
    plot_hist_contour(X_bias, y, w_hist, \  
        w_norm, ax=ax, title='Gradient descent')  
    ax = fig.add_subplot(122)  
    plot_hist_contour(X_bias, y, w_hist_mom, \  
        w_norm, ax=ax, title='Gradient descent with momentum', show_legend=True)  
    plt.tight_layout()
```

- the plot

```
comparison_plot()
```

- study alpha and momentum

```
def alphas_study_with_momentum(alphas, beta):  
    fig = plt.figure(figsize=(11,7))  
    for i,alpha in enumerate(alphas):  
        ax = fig.add_subplot(2,3,i+1)  
        w_hist = gradient_descent_with_momentum(X_bias, y , w_0, alpha, beta, max_iters)  
        plot_hist_contour(X_bias, y, w_hist, w_norm, ax=ax, title='$\\alpha='+str(alpha)+'$')  
    plt.legend(scatterpoints=1, ncol=3, bbox_to_anchor=(-0.2,-0.2), frameon=True);  
    plt.tight_layout()
```

- run it

```
alphas_study_with_momentum(alphas, 0.5)
```

# Multilayer Perceptron

The composition of layers of perceptrons can capture complex relations between inputs and outputs in a hierarchical way. In order to proceed we need to improve the notation we have been using. That for, for each layer  $1 \leq l \leq L$ , the activations and outputs are calculated as:

$$\text{net}_j^l = \sum_i w_{ji}^l x_i^l \quad | \quad y_j^l = f^l(\text{net}_j^l),$$

where:

- $y_j^l$  is the  $j$ th output of layer  $l$ ,
- $x_i^l$  is the  $i$ th input to layer  $l$ ,
- $w_{ji}^l$  is the weight of the  $j$ -th neuron connected to input  $i$ ,
- $\text{net}_j^l$  is called net activation, and
- $f^l(\cdot)$  is the activation function of layer  $l$ , e.g.  $\tanh()$ , in the hidden layers and the identity in the last layer (for regression)

# Training MLPs with Backpropagation

- Backpropagation of errors is a procedure to compute the **gradient of the error function with respect to the weights** of a neural network.
- We can use the gradient from backpropagation to apply **gradient descent!**

## A math flashback

The **chain rule** can be applied in composite functions as,

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x)) g'(x).$$

or, in Leibniz notation,

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

The **total derivative** of  $f(x_1, x_2, \dots, x_n)$  on  $x_i$  is

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^n \frac{\partial f}{\partial x_j} \cdot \frac{\partial x_j}{\partial x_i}$$

**To apply gradient descent we need... to calculate the gradients**

Applying the chain rule,

$$\frac{\partial \ell}{\partial w_{ji}^l} = \overbrace{\frac{\partial \ell}{\partial \text{net}_j^l}}^{\delta_j^l} \underbrace{\frac{\partial \text{net}_j^l}{\partial w_{ji}^l}}_{\frac{\partial (\sum_i w_{ji}^l x_i^l)}{\partial w_{ji}^l} = x_i^l}$$

hence we can write

$$\frac{\partial \ell}{\partial w_{ji}^l} = \delta_j^l x_i^l$$

## What about the hidden layers ( $1 \leq l < L$ )?

We can express the loss  $\ell$  as a function of the activations of the subsequent layer,

$$\ell = \ell \left( \text{net}_1^{l+1}, \dots, \text{net}_K^{l+1} \right) ,$$

therefore, applying total derivatives,

$$\frac{\partial \ell}{\partial \hat{y}_j^l} = \frac{\partial \ell \left( \text{net}_1^{l+1}, \dots, \text{net}_K^{l+1} \right)}{\partial \hat{y}_j^l} .$$



For the output layer ( $l = L$ )

$$\delta_j^L = \frac{\partial \ell}{\partial \text{net}_j^L} = \frac{\frac{\partial \left( \frac{1}{2} \sum_j (y_j - \hat{y}_j^L)^2 \right)}{\partial \hat{y}_j^L}}{\frac{\partial \ell}{\partial \hat{y}_j^L}} \cdot \underbrace{\frac{\partial \hat{y}_j^L}{\text{net}_j^L}}_{f'(\text{net}_j^L)} = (y_j - \hat{y}_j^L) f'(\text{net}_j^L).$$

therefore

$$\frac{\partial \ell}{\partial w_{ji}^L} = (y_j - \hat{y}_j^L) f'(\text{net}_j^L) x_i^L$$

## Back-propagating the errors to the hidden layer

The  $\delta$ s of the subsequent layers are used to calculate the  $\delta$ s of the more internal ones.

$$\delta_j^l = \frac{\partial \ell}{\partial \text{net}_j^l} = \overbrace{\frac{\partial \ell}{\partial \hat{y}_j^l}}^{\sum_k \delta_k^{l+1} w_{kj}^{l+1}} \underbrace{\frac{\partial \hat{y}_j^l}{\partial \text{net}_j^l}}_{f'(\text{net}_j^l)} = \sum_k \left( \delta_k^{l+1} w_{kj}^{l+1} \right) f'(\text{net}_j^l)$$

Briefly, in each layer (we will omit the sample index  $k$  and layer  $l$ )

$$\delta_j = \begin{cases} \hat{y}_j - y_j & \text{in the output layer,} \\ f'(\text{net}_j) \sum_k \frac{\partial \ell}{\partial \hat{y}_k} & \text{otherwise.} \end{cases}$$

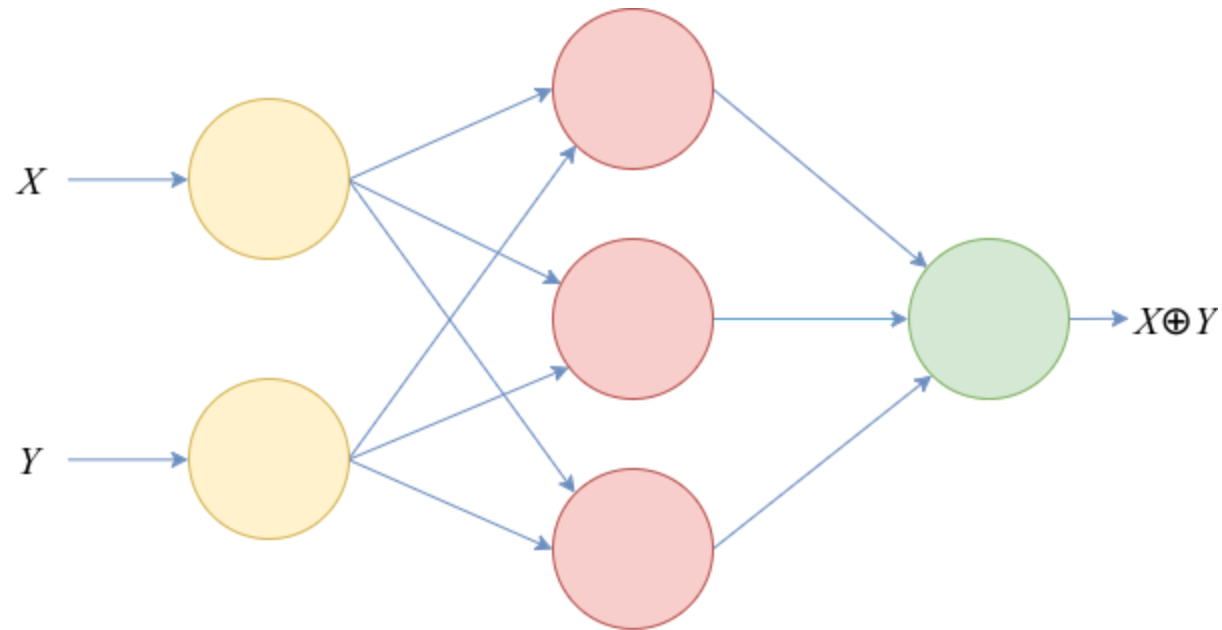
$$\frac{\partial \ell}{\partial w_{ji}} = \delta_j x_i ; \quad \frac{\partial \ell}{\partial x_i} = \delta_j w_{ji} .$$

where

- all nodes  $k$  are in the layer after  $j$ ;
- $\text{net}_j$  is known from propagation:  $\sum_i w_{ji} x_i$ ;
- actually you do not have to save  $a_j$  because  $g'(a_j)$  usually can be computed from  $y_j$ , e.g.
  - identity function:  $f'(\text{net}_j) = 1$ ,
  - tanh:  $f'(\text{net}_j) = 1 - y_j^2$ ;
- $\frac{\partial \ell}{\partial w_{ji}}$  will be used to update the weight  $w_{ji}$  in gradient descent;
- $\frac{\partial \ell}{\partial x_i}$  will be passed to the previous layer to compute the deltas;

# Build MLP in PyTorch

- Let's create network to model XOR gate



- The XOR truth table

<b>x</b>	<b>y</b>	<b>XOR</b>
0	0	0
0	1	1
1	0	1
1	1	0

- Input pair

```
inputs = list(map(lambda s: Variable(torch.Tensor([s])), [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]))
```

- The target

```
targets = list(map(lambda s: Variable(torch.Tensor([s])), [
    [0],
    [1],
    [1],
    [0]
]))
```

- The network

```
class XOR(nn.Module):  
    def __init__(self):  
        super(XOR, self).__init__()  
        self.fc1 = nn.Linear(2, 3, True)  
        self.fc2 = nn.Linear(3, 1, True)  
  
    def forward(self, x):  
        x = F.sigmoid(self.fc1(x))  
        x = self.fc2(x)  
        return x
```

- Initialize the network

```
net = XOR()
```

- Epoch, criterion and optimizer

```
EPOCHS = 50000
criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

- Training loop

```
print("Training loop:")
for idx in range(0, EPOCHS):
    for input, target in zip(inputs, targets):
        optimizer.zero_grad()    # zero the gradient buffers
        output = net(input)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()         # Does the update
    if idx % 5000 == 0:
        print("Epoch {: >8} Loss: {}".format(idx, loss.data.numpy()))
```



- The results:

```
print("Final results:")
for input, target in zip(inputs, targets):
    output = net(input)
    print("Input:[{}{}] Target:[{}] Predicted:[{}] Error:[{}]"
          .format(
            int(input.data.numpy()[0][0]),
            int(input.data.numpy()[0][1]),
            int(target.data.numpy()[0]),
            round(float(output.data.numpy()[0]), 4),
            round(float(abs(target.data.numpy()[0] - output.data.numpy()[0])), 4)
          ))
```

- Inference

```
output = net(Variable(torch.Tensor([1, 0])))
print(output)
```

# Repository

All material in this course can be cloned from <https://github.com/linerocks/vibrastic101>

# References

- <https://github.com/Imarti/machine-learning>
- <https://www.datasetlist.com/>
- <https://pytorch.org/docs/stable/index.html>
- <https://github.com/jcjohnson/pytorch-examples>
- [https://www.deeplearningwizard.com/deep\\_learning/practical\\_pytorch/pytorch\\_feedforward\\_neural\\_network/](https://www.deeplearningwizard.com/deep_learning/practical_pytorch/pytorch_feedforward_neural_network/)
- <https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/>
- <https://www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/>
- <https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a>

**End of slide**

**Thank You !**