Note: Some of the contents were collected from Andrew Ng's Deep Learning course on Coursera.

Python Basics with Numpy

This section will help familiarize you with the functions we'll need for this course.

Instructions:

- You will be using **Python 3**.
- After coding your function, run the cell right below it to check if your result is correct.

After this session you will:

- Be able to use iPython Notebooks
- Be able to use numpy functions and numpy matrix/vector operations
- Be able to vectorize code

iPython Notebooks

iPython Notebooks are interactive coding environments embedded in a webpage. After writing your code, you can run the cell by either pressing **"SHIFT"+"ENTER"**. Write code using Python 3 only.

- Click **Edit + Notebook Settings** for more options such as GPU/TPU

```
test = None
test = "Hello World"


print ("test: " + test)
```

```
    test: Hello World
```

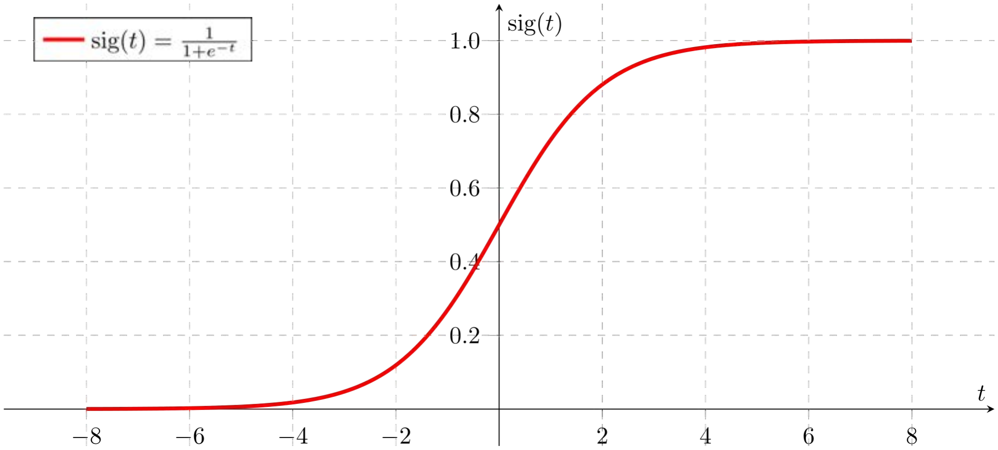
1 - Building basic functions with numpy

Numpy is the main package for scientific computing in Python. It is maintained by a large community (www.numpy.org). In this lab session, you will learn several key numpy functions such as `np.exp`, `np.log`, and `np.reshape`.

1.1 - sigmoid function, np.exp()

Before using `np.exp()`, you will use `math.exp()` to implement the sigmoid function.

$sigmoid(x) = \frac{1}{1+e^{-x}}$ is sometimes also known as the **logistic function**. It is a **non-linear function** used not only in Machine Learning (Logistic Regression), but also in Deep Learning.



Note: To refer to a function belonging to a specific package you could call it using **package_name.function()**. Run the code below to see an example with **math.exp()**.

```
import math

def basic_sigmoid(x):
    """
    Compute sigmoid of x.

    Arguments:
    x -- A scalar

    Return:
    s -- sigmoid(x)
    """
    ### Code ###
    s = 1 / (1 + math.exp(-x))

    return s
```

```
basic_sigmoid(-100)
```

```
3.7200759760208356e-44
```

- The use of **sigmoid nonlinear functions** was inspired by the ouputs of biological neurons. In the brain, the output of a neuron is typically all or nothing **(on or off)**, and hence, can mathematically be modeled as a function with only two outputs.
- Since neurons begin to fire **(turn on)** after a **certain input threshold has been surpassed**, the simplest mathematical function to model this behavior is the **(Heaviside)** step function, which

outputs zero below a threshold input value and outputs one above the threshold input value.

- However, this function is not smooth **(it fails to be differential at the threshold value)**. Therefore, the sigmoid class of functions is a differentiable alternative that still captures much of the behavior of biological neurons.
- Sigmoidal functions are known generally as **activation functions**, and more specifically as a **squashing functions**. The "**squashing**" refers to the fact that the output of the function exists between a **finite limit**, usually 0 and 1. These functions are incredibly useful in **determining probability**.
- Sigmoidal functions are frequently used in machine learning, specifically to **model the output of a node or "neuron."** These functions are inherently non-linear and thus allow neural networks to **find non-linear relationships between data features**. This greatly expands the utility of neural networks and allows them (in principle) to learn any function.
- **Without these activation functions, your neural network will be very similar to that of a linear model (which will be a bad predictor for the data that carries lot of non linearity).**

Note: Actually, **we rarely use the "math" library in deep learning** because the inputs of the functions are real numbers. In deep learning we **mostly use matrices and vectors**. This is why numpy is more useful.

```
### One reason why we use "numpy" instead of "math" in Deep Learning ###
x = [1, 2, 3]
basic_sigmoid(x) # you will see this give an error when you run it, because x is a vector.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-0468e34d7ed7> in <module>()
      1 x = [1, 2, 3]
----> 2 basic_sigmoid(x) # you will see this give an error when you run it, because x
is a vector.

<ipython-input-3-eb5af23c1745> in basic_sigmoid(x)
     12     """
     13     ### Code ###
---> 14     s = 1 / (1 + math.exp(-x))
     15 
     16     return s

TypeError: bad operand type for unary -: 'list'
```

SEARCH STACK OVERFLOW

In fact, if $x = (x_1, x_2, \ldots, x_n)$ is a row vector then $np.exp(x)$ will apply the exponential function to every element of x. The output will thus be: $np.exp(x) = (e^{x_1}, e^{x_2}, \ldots, e^{x_n})$

```
import numpy as np

# example of np.exp
x = np.array([1, 2, 3])
print(np.exp(x)) # result is (exp(1), exp(2), exp(3))
```

```
     [ 2.71828183  7.3890561  20.08553692]
```

Furthermore, if x is a vector, then a Python operation such as $s = x + 3$ or $s = \frac{1}{x}$ will output s as a vector of the same size as x.

```
# example of vector operation
x = np.array([1, 2, 3])
print (x + 3)
```

```
     [4 5 6]
```

Any time you need more info on a numpy function, we encourage you to look at the official documentation.

You can also create a new cell in the notebook and write `np.exp?` (for example) to get quick access to the documentation.

Example: Implementing the sigmoid function using numpy. x could now be either a **real number, a vector, or a matrix**. The data structures we use in numpy to represent these shapes (vectors, matrices...) are called numpy arrays.

$$\text{For } x \in \mathbb{R}^n \text{, } sigmoid(x) = sigmoid\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ \dots \\ \frac{1}{1+e^{-x_n}} \end{pmatrix} \tag{1}$$

```
import numpy as np # this means you can access numpy functions by writing np.function() inste

def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size

    Return:
    s -- sigmoid(x)
    """

    s = 1 / (1 + np.exp(-x))
    s = 1 / (1 + np.exp(-x))

    return s


x = np.array([1, 2, 3])
sigmoid(x)
```

```
array([0.73105858, 0.88079708, 0.95257413])
```

1.2 - Sigmoid gradient

As you will see in the theory class lecture, **you will need to compute gradients to optimize loss functions using backpropagation**. Let's code your first gradient function.

Implementing the function `sigmoid_grad()` to compute the gradient of the sigmoid function with respect to its input x. The formula is:

$$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2}$$

You often code this function in **two steps**:

1. Set s to be the sigmoid of x. You might find your sigmoid(x) function useful.
2. Compute $\sigma'(x) = s(1 - s)$

Worry about how I found out the derivative?

Let's denote the sigmoid function as: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

Here's a detailed derivation:

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{\left(1+e^{-x}\right)^{2}} \\
&= \frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\cdot\frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\cdot\left(\frac{1+e^{-x}}{1+e^{-x}}-\frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}}\cdot\left(1-\frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)\cdot(1-\sigma(x))
\end{aligned}
$$

So the derivative of the sigmoid is: $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$

```
def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the sigmoid function with r
    You can store the output of the sigmoid function into variables and then use it to calcul

    Arguments:
    x -- A scalar or numpy array

    Return:
    ds -- Your computed gradient.
    """

    ### CODE ###
    s = sigmoid(x)
    ds = s-s*s

    return ds


x = np.array([1, 2, 3])
print ("sigmoid_derivative(x) = " + str(sigmoid_derivative(x)))
```

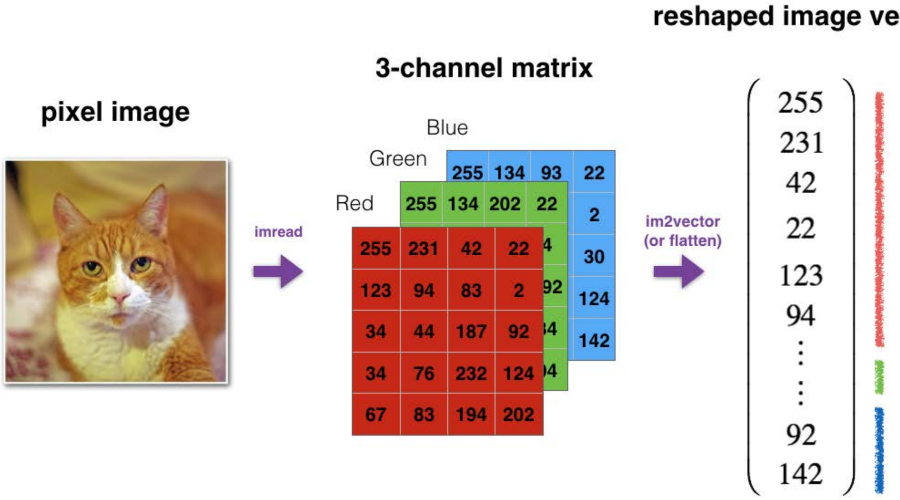
```
     sigmoid_derivative(x) = [0.19661193 0.10499359 0.04517666]
```

1.3 - Reshaping arrays

Two common numpy functions used in deep learning are np.shape and np.reshape().

- **X.shape is used to get the shape (dimension) of a matrix/vector X.**
- **X.reshape(...) is used to reshape X into some other dimension.**

For example, in computer science, an image is represented by a 3D array of shape $(length, height, depth = 3)$. However, when you read an image as the input of an algorithm you convert it to a vector of shape $(length * height * 3, 1)$. **In other words, you "unroll", or reshape, the 3D array into a 1D vector.**



Example: Implementing `image2vector()` that takes an input of shape (length, height, 3) and returns a vector of shape (length*height*3, 1). For example, if you would like to reshape an array v of shape (a, b, c) into a vector of shape (a*b,c) you would do:

```
v = v.reshape((v.shape[0]*v.shape[1], v.shape[2])) # v.shape[0] = a ; v.shape[1] = b ; v.shape[2] = c
```

- Please don't hardcode the dimensions of image as a constant. Instead look up the quantities you need with `image.shape[0]`, etc.

```
def image2vector(image):
    """
    Argument:
    image -- a numpy array of shape (length, height, depth)

    Returns:
    v -- a vector of shape (length*height*depth, 1)
    """
    length = image.shape[0]
    height = image.shape[1]
    depth = image.shape[2]
    depth = image.shape[2]
    v = image.reshape(length*height*depth, 1)

    return v


# This is a 3 by 3 by 2 array, typically images will be (num_px_x, num_px_y,3) where 3 repres
image = np.array([[[ 0.67826139,  0.29380381],
                   [ 0.90714982,  0.52835647],
                   [ 0.4215251 ,  0.45017551]],

                  [[ 0.92814219,  0.96677647],
                   [ 0.85304703,  0.52351845],
                   [ 0.19981397,  0.27417313]],

                  [[ 0.60659855,  0.00533165],
                   [ 0.10820313,  0.49978937],
                   [ 0.34144279,  0.94630077]]])

print ("image2vector(image) = " + str(image2vector(image)))
```

```
image2vector(image) = [[0.67826139]
 [0.29380381]
 [0.90714982]
 [0.52835647]
 [0.4215251 ]
 [0.45017551]
 [0.92814219]
 [0.96677647]
 [0.85304703]
 [0.52351845]
 [0.19981397]
 [0.27417313]
 [0.60659855]
 [0.00533165]
 [0.10820313]
 [0.49978937]
 [0.34144279]
 [0.94630077]]
```

```
import numpy as np;
image = np.random.rand(3,3,3);
image2vector(image)
```

```
array([[0.37070266],
       [0.2245434 ],
       [0.08617355],
       [0.6705214 ],
       [0.64904656],
       [0.01902036],
       [0.05953873],
       [0.35932862],
       [0.95741091],
       [0.49962773],
       [0.560809  ],
       [0.92919232],
       [0.46775356],
       [0.45532101],
       [0.54416541],
       [0.90361086],
       [0.41796505],
       [0.75217415],
       [0.97927841],
       [0.56095234],
       [0.63326321],
       [0.40145603],
       [0.85507152],
       [0.26455511],
       [0.2077953 ],
       [0.32712933],
       [0.48968725]])
```

What you need to remember:

- `np.exp(x)` works for any `np.array` x and applies the exponential function to every coordinate
- the sigmoid function and its gradient
- `image2vector` is commonly used in deep learning
- `np.reshape` is widely used. In the future, you'll see that keeping your matrix/vector dimensions straight will go toward eliminating a lot of bugs.
- `numpy` has efficient built-in functions

2 - A note on Python Numpy Vectors

```
import numpy as np

a = np.random.randn(5)

b = np.random.rand(5)

print (a)

print (b)
```

```
    [-0.3952258   1.41211196 -0.9766941   0.9207573  -0.14710787]
    [0.3320662  0.68606533 0.59471945 0.97293171 0.80447056]
```

```
# (5,) is a Rank 1 array, it is neither a Col vector nor a Row vector. (Don't use it)
print (a.shape)
```

```
    (5,)
```

```
print (a.T)
```

```
     [-0.3952258   1.41211196 -0.9766941   0.9207573  -0.14710787]
```

```
print (np.dot(a, a.T))
```

```
     3.973629712464255
```

```
a = np.random.randn(5,1)

print (a)
```

```
     [[ 0.56409505]
      [-1.73038046]
      [ 0.36109628]
      [ 0.42446548]
      [-0.05806326]]
```

```
print (a.T)
```

```
     [[ 0.56409505 -1.73038046  0.36109628  0.42446548 -0.05806326]]
```

```
print (np.dot(a, a.T)) # Outer product of 2 vectors will give you a matrix
```

```
     [[ 0.31820322 -0.97609905  0.20369262  0.23943887 -0.0327532 ]
      [-0.97609905  2.99421655 -0.62483395 -0.73448677  0.10047153]
      [ 0.20369262 -0.62483395  0.13039052  0.1532729  -0.02096643]
      [ 0.23943887 -0.73448677  0.1532729   0.18017094 -0.02464585]
      [-0.0327532   0.10047153 -0.02096643 -0.02464585  0.00337134]]
```

```
a = np.random.randn(5,1)

print (a) # Column Vector
```

```
     [[-2.08863811]
      [-1.41235975]
      [ 0.42640614]
      [-2.21727178]
      [-2.33846804]]
```

```
print (a.shape) # 5x1 Martix or a Column Vector
```

```
     (5, 1)
```

```
a = np.random.randn(1, 5)

print (a) # Row Vector
```

```
     [[-0.08247535 -2.08232818  0.72972682  0.70277153  0.20583417]]
```

```
print (a.shape) # 1x5 Martix or a Row Vector
```

```
(1, 5)
```

2.1 - Vectorization

Vectorization is the art of getting rid of excessive for loops in your code. In deep learning, you deal with very large datasets. Hence, a non-computationally-optimal function can become a huge bottleneck in your algorithm and can result in a model that takes ages to run. To make sure that your code is computationally efficient, you will use vectorization. For example, try to tell the difference between the following implementations of the dot/elementwise product.

```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

x1 = np.random.rand(1000000)
x2 = np.random.rand(1000000)

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = 0
for i in range(len(x1)):
    dot+= x1[i]*x2[i]
toc = time.process_time()
print ("Dot Product of Vectors = " + str(dot) + "\n ----- Computation time = " + str(1000*(to


### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i]*x2[i]
toc = time.process_time()
print ("Elementwise Multiplication = " + str(mul) + "\n ----- Computation time = " + str(1000

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j]*x1[j]
toc = time.process_time()
print ("General Dot Product = " + str(gdot) + "\n ----- Computation time = " + str(1000*(toc
     Dot Product of Vectors = 249827.02160934
      ----- Computation time = 572.2960600000002ms
     Elementwise Multiplication = [0.04938253 0.08781817 0.24899559 ... 0.61758554 0.0345799
      ----- Computation time = 525.3037479999998ms
     General Dot Product = [250128.69795932 250332.07022047 250298.14500865]
      ----- Computation time = 2368.9006ms
```

```
### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("Dot Product of Vectors = " + str(dot) + "\n ----- Computation time = " + str(1000*(to

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("Elementwise Multiplication = " + str(mul) + "\n ----- Computation time = " + str(1000

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("General Dot Product = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -
```

```
Dot Product of Vectors = 249827.02160934033
 ----- Computation time = 5.7933969999997ms
Elementwise Multiplication = [0.04938253 0.08781817 0.24899559 ... 0.61758554 0.0345799
 ----- Computation time = 12.078004000000142ms
General Dot Product = [250128.69795931 250332.07022047 250298.14500865]
 ----- Computation time = 6.919284000000303ms
```

```
print ("test: " + test)
```

```
test: Hello World
```

As you may have noticed, the **vectorized implementation is much cleaner and more efficient**. *For bigger vectors/matrices, the differences in running time become even bigger.*

Note that `np.dot()` performs a matrix-matrix or matrix-vector multiplication. This is different from `np.multiply()`, which performs an element-wise multiplication.

2.2 Implement the L1 and L2 loss functions

Exercise: Implement the numpy vectorized version of the L1 loss. You may find the function abs(x) (absolute value of x) useful.

Reminder:

- The loss is used to evaluate the performance of your model. The bigger your loss is, the more different your predictions ($\hat{y}$) are from the true values (y). In deep learning, you use optimization algorithms like Gradient Descent to train your model and to minimize the cost.
- L1 loss is defined as:

$$L_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}| \tag{6}$$

```
# GRADED FUNCTION: L1

def L1(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L1 loss function defined above
    """

    ### START CODE HERE ### (≈ 1 line of code)


    ### END CODE HERE ###

    return loss


yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L1 = " + str(L1(yhat,y)))
```

Expected Output

L1
1.1

Exercise: Implement the numpy vectorized version of the L2 loss. There are several way of implementing the L2 loss but you may find the function np.dot() useful. As a reminder, if $x = [x_1, x_2, \ldots, x_n]$, then `np.dot(x,x)` = $\sum_{j=0}^{n} x_j^2$.

- L2 loss is defined as

$$L_2(\hat{y}, y) = \sum_{i=0}^{m} (y^{(i)} - \hat{y}^{(i)})^2 \tag{7}$$

```
# GRADED FUNCTION: L2

def L2(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L2 loss function defined above
    """

    ### START CODE HERE ### (≈ 1 line of code)


    ### END CODE HERE ###

    return loss


yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L2 = " + str(L2(yhat,y)))
```

Expected Output

L2
0.43

What to remember:

- Vectorization is very important in deep learning. It provides computational efficiency and clarity.
- You have reviewed the L1 and L2 loss.
- You are familiar with many numpy functions such as np.sum, np.dot, np.multiply, etc...

```
##------------------------------------------------------------------------
---------------
# IMPLEMENT HMM USING VITERBI ALGORITHM
#
##------------------------------------------------------------------------
---------------
import numpy as np
import pandas as pd
from collections import Counter

# LOAD INPUT
# HERE SAMPLE_DATASET TAKES TRAINING DATA AS AN INPUT


def input_data(sample_dataset):
    data = pd.read_csv(sample_dataset, sep="\t", header = None)
    #data = pd.read_csv('training_Dragon_1000.data.txt', sep="\t", header
= None)
    data.columns = ["state", "outcome"]
    state = len(data.columns)
    trainData = data.values
    return trainData


##------------------------------------------------------------------------
------------------
# FOR EMISSION MATRIX
# CALCULATE THE PROBABILITY OF DIFFERENT TEST SAMPLE AND STORED IN
EMMISION MATRIX
##------------------------------------------------------------------------
------------------

def emission_matrix_calculate(trainData):
    mask1 = (trainData == "fair")
    number_of_fair = len(trainData[mask1])
    mask2 = (trainData == "cheat")
    number_of_cheat = len(trainData[mask2])
    count_Fair_Win = 0
    count_cheat_Win = 0
    for i in range(len(trainData)):
        if(trainData[i][0] == "fair" and trainData[i][1] == "win"):
            count_Fair_Win += 1;
        if(trainData[i][0] == "cheat" and trainData[i][1] == "win"):
            count_cheat_Win += 1;

    p_fair_win = count_Fair_Win/number_of_fair
    p_cheat_win = count_cheat_Win/number_of_cheat
    p_fair_lose = 1 - p_fair_win
    p_cheat_lose = 1 - p_cheat_win

    emission_matrix = np.array([[p_cheat_lose, p_cheat_win], [p_fair_lose,
p_fair_win]])
    return emission_matrix



##------------------------------------------------------------------------
------------------
# FOR TRANSITION MATRIX
# CALCULATE THE PROBABILITY OF DIFFERENT STATE TRANSITION FOR TRAINING
DATA
# AND VALUE STORED IN TRANSITION MATRIX FOR FURTHER CALCULATION
##------------------------------------------------------------------------
------------------


def transition_matrix_count(trainData):
    count_tran_Fair = 0
    count_tran_Cheat = 0
    cheat_to_fair = 0
    fair_to_fair = 0
    fair_to_cheat = 0
    cheat_to_cheat = 0
    for j in range(len(trainData)):
        if(j!=0):
            if(trainData[j][0] == "fair"):
                count_tran_Fair += 1
                if(trainData[j-1][0] == "cheat"):
                    cheat_to_fair += 1
                else:
                    fair_to_fair += 1
            else:
                count_tran_Cheat += 1
                if(trainData[j-1][0] == "fair"):
                    fair_to_cheat += 1
                else:
                    cheat_to_cheat += 1

    c2c = cheat_to_cheat/count_tran_Cheat
    f2c = fair_to_cheat/count_tran_Cheat
    c2f = cheat_to_fair/count_tran_Fair
    f2f = fair_to_fair/count_tran_Fair
    transition_matrix = np.array([[c2c, c2f], [f2c, f2f]])
    return transition_matrix


##-------------------------------------------------------------------------
---------------
# INPUT_OBSERVATION FUNCTION TAKES TESTING DATA SET
#
##-------------------------------------------------------------------------
---------------


def input_observation(testingData):
    observation = []
    with open(testingData) as f:
        rawdata = f.readlines()
        observation = [1*(s.strip() == 'win') for s in rawdata]
    observation = np.array(observation).reshape(len(observation), 1)
    return observation


##-------------------------------------------------------------------------
---------------
# MAX_SCORE FUNCTION USE FOR VITERBI ALGORITHM
# IT RETURN THE MAXIMUM SCORE FOR A PARTICULAR STATE AT GIVEN POSITION
##-------------------------------------------------------------------------
---------------
def max_score(outcome, score, transition_matrix):
    max_score = []
    for x, y in zip(score, transition_matrix):
        max_score.append(np.log2(outcome)+ x + np.log2(y))
    return np.max(max_score)
    # return np.max([np.log2(outcome) + x + np.log2(y) for x, y in
zip(score,transition_matrix)])

##-------------------------------------------------------------------------
---------------
# IMPLEMENT HMM USING VITERBI ALGORITHM
# HERE SCORE STORE THE ALL COMBINATION OF VALUES FOR DIFFERENT STATE IN
DIFFERENT POSITION
#
##-------------------------------------------------------------------------
---------------

def viterbi(observations, emission_matrix, initial_pro,
transition_matrix):
    score = np.zeros(shape=(2, len(observations)), dtype=float)
    final_output = np.zeros([len(observations)])
    count_cheat_step = 0
    count_fair_step = 0
    for index, outcome in enumerate(observations):
        # print(index)
        if index == 0:
            score[0][index] =
np.log2(emission_matrix[0][outcome[0]])+(initial_pro[0])
            score[1][index] =
np.log2(emission_matrix[1][outcome[0]])+(initial_pro[1])
        else:
            score[0][index] = max_score(emission_matrix[0][outcome[0]],
score[:, index - 1], transition_matrix[0])
            score[1][index] = max_score(emission_matrix[1][outcome[0]],
score[:, index - 1], transition_matrix[1])
        if (score[0][index] > score[1][index]):
            final_output[index] = 0
            count_cheat_step += 1
        else:
            final_output[index] = 1
            count_fair_step += 1

    print("fair", count_fair_step)
    print("cheat", count_cheat_step)
    return score, final_output


# MAIN FUNCTION FUNCTION
# IMPLEMENT VITERBI ALGORITHM
def main():

    sample_dataset = "training_Pyramid_1000.data.txt"  #"sData.txt"
    testing_data = "testing_Pyramid_1000.data.txt"     #"s_ob.txt"

    observations = input_observation(testing_data)
    initial_pro = np.array([0.5, 0.5])

    trainData = input_data(sample_dataset)

    emission_matrix = emission_matrix_calculate(trainData)
    transition_matrix = transition_matrix_count(trainData)

    print("Transition Matrix\n", transition_matrix)
    print("Emission Matrix\n", emission_matrix)

    score, final_output = viterbi(observations, emission_matrix,
initial_pro, transition_matrix)
```

```
    #print(final_output)

main()

#
#
#
#
# training_Pyramid_1000.data.txt
# testing_Pyramid_1000.data.txt
#
# training_Dragon_1000.data.txt
# testing_Dragon_1000.data.txt
#
# training_Lion_1000.data.txt
# testing_Lion_1000.data.txt
#
#
```

Logistic Regression (One hidden unit neural network)

We will build a logistic regression classifier to recognize cats. This session will step you through how to do this with a Neural Network mindset.

Instructions:

- Do not use loops (for/while) in your code, unless the instructions explicitly ask you to do so.

You will learn to:

- Build the general architecture of a learning algorithm, including:
  - Initializing parameters
  - Calculating the cost function and its gradient
  - Using an optimization algorithm (gradient descent)
- Gather all three functions above into a main model function, in the right order.

1 - Packages

First, let's run the cell below to import all the packages that you will need during this session.

- numpy is the fundamental package for scientific computing with Python.
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- matplotlib is a famous library to plot graphs in Python.

2 - Overview of the Problem set

Problem Statement: You are given a dataset in ".h5" format containing:

- a training set of **m_train images** labeled as **cat (y=1)** or **non-cat (y=0)**
- a test set of m_test images labeled as cat or non-cat
- each image is of shape **(num_px, num_px, 3)** where 3 is for the **3 channels (RGB)**. Thus, each image is square **(height = num_px)** and **(width = num_px)**.

You will build a **simple image-recognition algorithm** that can correctly classify pictures as cat or non-cat.

Let's get more familiar with the dataset. Load the data by running the following code.

```
from google.colab import drive

drive.mount('/content/gdrive')
root_path = '/content/gdrive/My Drive/AUST Teaching Docs/AUST Fall 2019/Soft Computing/CSE 42
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mou
```

```
import numpy as np
import h5py

def load_dataset():
    train_dataset = h5py.File(root_path + 'train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

    test_dataset = h5py.File(root_path + 'test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig


# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y = load_dataset()
```

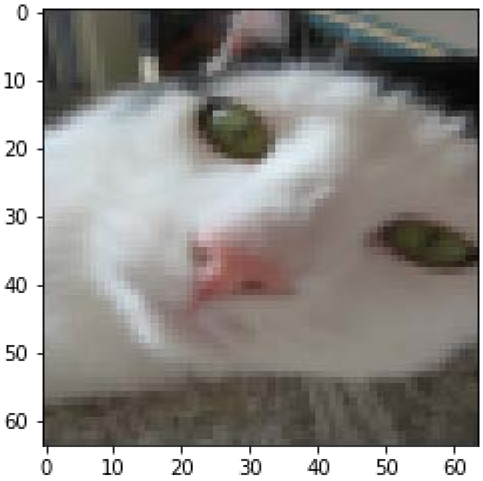
```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-37-a20b7e07fcf1> in <module>()
----> 1 train_set_x_orig, train_set_y, test_set_x_orig, test_set_y = load_dataset()

                              2 frames
/usr/local/lib/python3.6/dist-packages/h5py/_hl/files.py in make_fid(name, mode,
userblock_size, fapl, fcpl, swmr)
    140         if swmr and swmr_support:
    141             flags |= h5f.ACC_SWMR_READ
--> 142         fid = h5f.open(name, flags, fapl=fapl)
    143     elif mode == 'r+':
    144         fid = h5f.open(name, h5f.ACC_RDWR, fapl=fapl)

h5py/_objects.pyx in h5py._objects.with_phil.wrapper()

h5py/_objects.pyx in h5py._objects.with_phil.wrapper()

h5py/h5f.pyx in h5py.h5f.open()

OSError: Unable to open file (unable to open file: name = '/content/gdrive/My
Drive/AUST Teaching Docs/AUST Fall 2019/Soft Computing/CSE 4238/Codes/Lab
02/dataset/train_catvnoncat.h5', errno = 2, error message = 'No such file or
directory', flags = 0, o_flags = 0)
```

- We added "_orig" at the end of image datasets **(train and test)** because we are going to preprocess them. After preprocessing, we will end up with `train_set_x` and `test_set_x` (the labels `train_set_y` and `test_set_y` don't need any preprocessing).

- Each line of your `train_set_x_orig` and `test_set_x_orig` is an array representing an image. You can visualize an example by running the following code. **Feel free also to change the `index` value and re-run to see other images.**

```
import matplotlib.pyplot as plt

# Example of a picture
index = 14
plt.imshow(train_set_x_orig[index])
```

```
<matplotlib.image.AxesImage at 0x7f2ac6abe0f0>
```



- Many software bugs in deep learning come from having **matrix/vector** dimensions that don't fit. **If you can keep your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.**

Exercise: Find the values for:

- `m_train` (number of training examples)
- `m_test` (number of test examples)
- `num_px` (= height = width of a training image) Remember that `train_set_x_orig` is a numpy-array of shape `(m_train, num_px, num_px, 3)`. For instance, you can access `m_train` by writing `train_set_x_orig.shape[0]`.

```
### START CODE HERE ### (≈ 3 lines of code)
m_train = train_set_y.shape[1]
m_test = test_set_y.shape[1]
num_px = train_set_x_orig.shape[1]
### END CODE HERE ###

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Expected Output for m_train, m_test and num_px:

m_train	209
m_test	50
num_px	64

For convenience, you should now reshape images of shape `(num_px, num_px, 3)` in a numpy-array of shape `(num_px * num_px * 3, 1)`. After this, our training (and test) dataset is a numpy-array where each column represents a flattened image. There should be `m_train (respectively m_test)` columns.

Exercise: Reshape the training and test data sets so that images of size `(num_px, num_px, 3)` are flattened into single vectors of shape `(num\_px * num\_px * 3, 1)`.

A trick when you want to flatten a matrix X of shape `(a,b,c,d)` to a matrix X_flatten of shape `(b*c*d, a)` is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T is the transpose of X
```

```
# Reshape the training and test examples

### START CODE HERE ### (≈ 2 lines of code)
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T
### END CODE HERE ###

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
train_set_x_flatten shape: (12288, 209)
```

```
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
```

Expected Output:

train_set_x_flatten shape	(12288, 209)
train_set_y shape	(1, 209)
test_set_x_flatten shape	(12288, 50)
test_set_y shape	(1, 50)

- To represent color images, the **red, green and blue channels (RGB)** must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from **0 to 255**.
- One common preprocessing step in machine learning is to **center and standardize your dataset**, **meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array**. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).
- During the training of your model, **you're going to multiply weights and add biases to some initial inputs in order to observe neuron activations**. Then you backpropogate with the gradients to train the model. But, it is extremely important for each feature to have a similar range such that our gradients don't explode. You will see that more in detail later in the lectures.

Let's standardize our dataset.

```
train_set_x = train_set_x_flatten / 255
test_set_x = test_set_x_flatten / 255
```

What you need to remember:

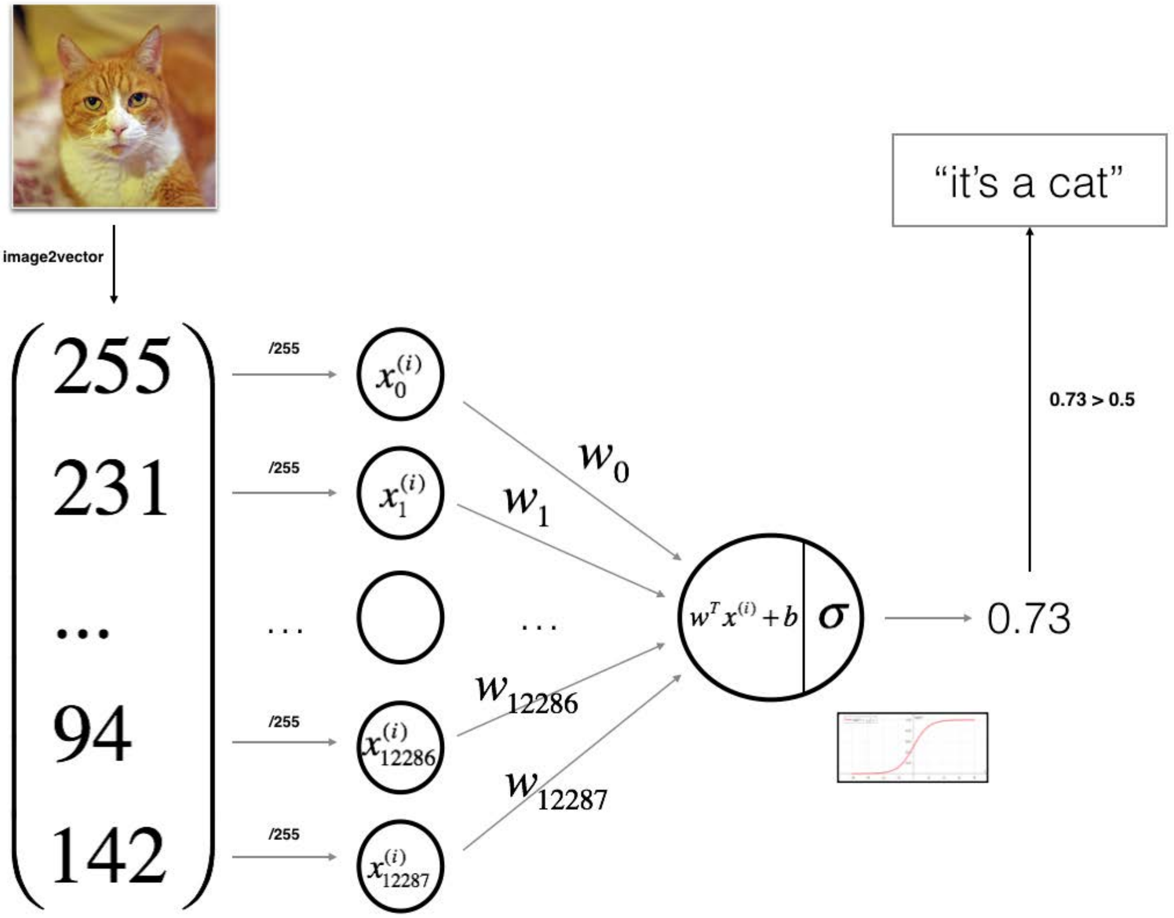
Common steps for pre-processing a new dataset are:

- Figure out the dimensions and shapes of the problem (m_train, m_test, num_px, ...)
- Reshape the datasets such that each example is now a vector of size (num_px * num_px * 3, 1)
- "Standardize" the data

3 - General Architecture of the learning algorithm

It's time to design a simple algorithm **to distinguish cat images from non-cat images.**

You will build a Logistic Regression. The following Figure explains why **Logistic Regression is actually a very simple Neural Network!**



Mathematical expression of the algorithm:

For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)}) \tag{3}$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

Key steps:

In this exercise, you will carry out the following steps:

```
- Initialize the parameters of the model
- Learn the parameters for the model by minimizing the cost
- Use the learned parameters to make predictions (on the test set)
- Analyse the results and conclude
```

4 - Building the parts of our algorithm

The main steps for building a Neural Network are:

1. Define the model structure (*such as number of input features*)
2. Initialize the model's parameters
3. Loop:
   - Calculate current loss (*forward propagation*)
   - Calculate current gradient (*backward propagation*)
   - Update parameters (*gradient descent*)

You often build 1-3 separately and integrate them into one function we call `model()`.

4.1 - Helper functions

Exercise: Using your code from "Python Basics", implement `sigmoid()`. As you've seen in the figure above, you need to compute $sigmoid(w^T x + b)$ to make predictions.

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    x -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """

    ### START CODE HERE ### (≈ 1 line of code)
    s = 1 / (1 + np.exp(-z))
    ### END CODE HERE ###

    return s


print ("sigmoid(0) = " + str(sigmoid(0)))
print ("sigmoid(9.2) = " + str(sigmoid(9.2)))
```

```
sigmoid(0) = 0.5
sigmoid(9.2) = 0.9998989708060922
```

Expected Output:

sigmoid(0)	0.5
sigmoid(9.2)	0.999898970806

4.2 - Initializing parameters

Exercise: Implement parameter initialization in the cell below. You have to initialize `w` as a vector of zeros. If you don't know what numpy function to use, look up `np.zeros()` in the Numpy library's documentation.

```
def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """

    ### START CODE HERE ### (≈ 1 line of code)
    w = np.zeros(shape=(dim, 1))
    b = 0
    ### END CODE HERE ###

    return w, b
```

```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0.]
 [0.]]
b = 0
```

Expected Output:

w	[[0.] [0.]]
b	0

For image inputs, w will be of shape **($num\_px \times num\_px \times 3, 1$)**.

4.3 - Forward and Backward propagation

Now that your parameters are initialized, you can do the "**forward**" and "**backward**" propagation steps for learning the parameters.

Exercise: Implement a function `propagate()` that computes the cost function and its gradient.

Hints:

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, \ldots, a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})$

Here are the two formulas you will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m}X(A-Y)^T \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)}-y^{(i)}) \tag{8}$$

```
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained above

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, number of exampl

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b

    Tips:
    - Write your code step by step for the propagation
    """

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)

    ### START CODE HERE ### (≈ 2 lines of code)
    A = sigmoid(np.dot(w.T, X) + b)  # compute activation
    cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A)))  # compute cost
    ### END CODE HERE ###

    # BACKWARD PROPAGATION (TO FIND GRAD)

    ### START CODE HERE ### (≈ 2 lines of code)
    dw = (1 / m) * np.dot(X, (A - Y).T)
    db = (1 / m) * np.sum(A - Y)
    db = (1 / m) * np.sum(A - Y)
    ### END CODE HERE ###


    grads = {"dw": dw,
             "db": db}

    return grads, cost


w, b, X, Y = np.array([[1], [2]]), 2, np.array([[1,2], [3,4]]), np.array([[1, 0]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

```
dw = [[0.99993216]
 [1.99980262]]
db = 0.49993523062470574
cost = 6.000064773192205
```

Expected Output:

dw	[[0.99993216] [1.99980262]]
db	0.499935230625
cost	6.000064773192205

d) Optimization

- You have initialized your parameters.
- You are also able to compute a cost function and its gradient.
- Now, you want to update the parameters using gradient descent.

Exercise: Write down the optimization function. The goal is to learn w and b by minimizing the cost function J. For a parameter θ, the update rule is $\theta = \theta - \alpha \text{ } d\theta$, where α is the learning rate.

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of exam
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
```

```
    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respect to the
    costs -- list of all the costs computed during the optimization, this will be used to plo

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """

    costs = []

    for i in range(num_iterations):

        # Cost and gradient calculation (≈ 1-4 lines of code)
        ### START CODE HERE ###
        grads, cost = propagate(w, b, X, Y)
        ### END CODE HERE ###

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule (≈ 2 lines of code)
        ### START CODE HERE ###
        w = w - learning_rate * dw  # need to broadcast
        b = b - learning_rate * db
        ### END CODE HERE ###

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

        # Print the cost every 100 training examples
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" % (i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

```
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100, learning_rate = 0.009, print

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("db = " + str(grads["db"]))
```

```
     w = [[0.1124579 ]
      [0.23106775]]
     b = 1.5593049248448891
     dw = [[0.90158428]
      [1.76250842]]
     db = 0.4304620716786828
```

Expected Output:

w	[[0.1124579] [0.23106775]]
b	1.55930492484
dw	[[0.90158428] [1.76250842]]
db	0.430462071679

Exercise: The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. Implement the `predict()` function. There is two steps to computing predictions:

1. Calculate $\hat{Y} = A = \sigma(w^T X + b)$
2. Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector `Y_prediction`. If you wish, you can use an `if`/`else` statement in a `for` loop (though there is also a way to vectorize this).

```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for the examples
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a cat being present in the picture
    ### START CODE HERE ### (≈ 1 line of code)
    A = sigmoid(np.dot(w.T, X) + b)
    ### END CODE HERE ###

    for i in range(A.shape[1]):
        # Convert probabilities a[0,i] to actual predictions p[0,i]
        ### START CODE HERE ### (≈ 4 lines of code)
        Y_prediction[0, i] = 1 if A[0, i] > 0.5 else 0
        ### END CODE HERE ###

    return Y_prediction


print("predictions = " + str(predict(w, b, X)))
```

```
predictions = [[1. 1.]]
```

Expected Output:

predictions	[[1. 1.]]

5 - Merge all functions into a model

You will now see how the overall model is structured by putting together all the building blocks (functions implemented in the previous parts) together, in the right order.

Exercise: Implement the model function. Use the following notation: - Y_prediction for your predictions on the test set - Y_prediction_train for your predictions on the train set - w, costs, grads for the outputs of optimize()

```
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning_rate=0.5, print_cos
    """
    Builds the logistic regression model by calling the function you've implemented previousl

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px * num_px * 3, m_tra
    Y_train -- training labels represented by a numpy array (vector) of shape (1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px * num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of shape (1, m_test)
    num_iterations -- hyperparameter representing the number of iterations to optimize the pa
    learning_rate -- hyperparameter representing the learning rate used in the update rule of
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    ### START CODE HERE ###
    # initialize parameters with zeros (≈ 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent (≈ 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples (≈ 2 lines of code)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    ### END CODE HERE ###

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) *
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 10


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d


d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_r
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

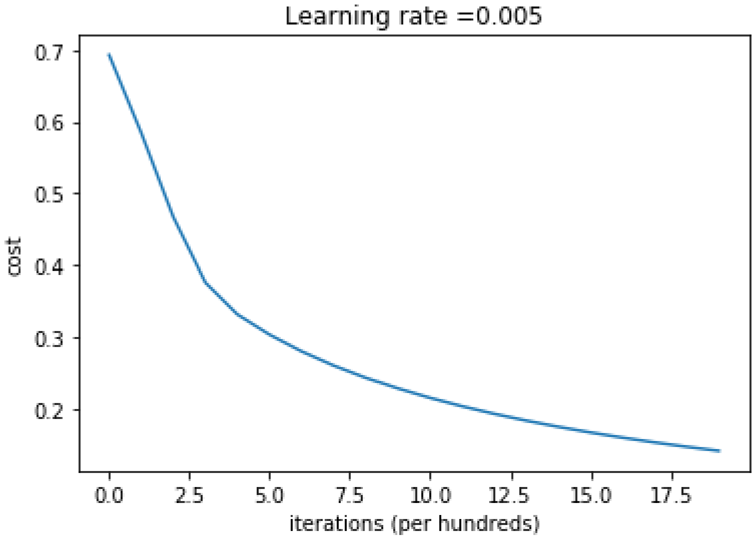
Expected Output:

```
Train Accuracy    99.04306220095694 %
Test Accuracy     70.0 %
```

Comment: Training accuracy is close to 100%. This is a good sanity check: your model is working and has high enough capacity to fit the training data. Test error is 70%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier. But no worries, you'll build an even better classifier next week!

Also, you see that the model is clearly overfitting the training data. Later you will learn how to reduce overfitting, for example by using regularization. Using the code below (and changing the `index` variable) you can look at predictions on pictures of the test set.

```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



Interpretation: **You can see the cost decreasing.** It shows that the parameters are being learned. However, you see that you could train the model even more on the training set. Try to increase the number of iterations in the cell above and rerun the cells. You might see that the training set accuracy goes up, but the test set accuracy goes down. **This is called overfitting.**

6 - Further analysis

Let's analyze it further, and examine possible choices for the learning rate α.

Choice of learning rate

Reminder: In order for Gradient Descent to work you must choose the learning rate wisely. The learning rate α determines how rapidly we update the parameters. If the learning rate is too large we may "overshoot" the optimal value. Similarly, if it is too small we will need too many iterations to converge to the best values. That's why it is crucial to use a well-tuned learning rate.

Let's compare the learning curve of our model with several choices of learning rates. Run the cell below. This should take about 1 minute. Feel free also to try different values than the three we have initialized the `learning_rates` variable to contain, and see what happens.

```
learning_rates = [0.01, 0.001, 0.0001]
models = {}
for i in learning_rates:
    print ("learning rate is: " + str(i))
    models[str(i)] = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations =
    print ('\n' + "-------------------------------------------------------" + '\n')

for i in learning_rates:
    plt.plot(np.squeeze(models[str(i)]["costs"]), label= str(models[str(i)]["learning_rate"])

plt.ylabel('cost')
plt.xlabel('iterations')

legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```

```
learning rate is: 0.01
train accuracy: 99.52153110047847 %
test accuracy: 68.0 %

-------------------------------------------------------

learning rate is: 0.001
train accuracy: 88.99521531100478 %
test accuracy: 64.0 %

-------------------------------------------------------

learning rate is: 0.0001
```

Interpretation:

- Different learning rates give different costs and thus different predictions results.
- If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (though in this example, using 0.01 still eventually ends up at a good value for the cost).
- A lower cost doesn't mean a better model. You have to check if there is possibly overfitting. It happens when the training accuracy is a lot higher than the test accuracy.
- In deep learning, we usually recommend that you:
    - Choose the learning rate that better minimizes the cost function.
    - If your model overfits, use other techniques to reduce overfitting.

What to remember from this assignment:

1. Preprocessing the dataset is important.
2. You implemented each function separately: initialize(), propagate(), optimize(). Then you built a model().
3. Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm.

Finally, if you'd like, we invite you to try different things on this Notebook.

```
- Play with the learning rate and the number of iterations
- Try different initialization methods and compare the results
- Test other preprocessings (center the data, or divide each row by its standard deviation)
```

Note: Some of the contents were collected from Andrew Ng's course on Coursera.

PYTORCH

What is Pytorch?

PyTorch is a python package built by **Facebook AI Research (FAIR)** that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autograd (*Automatic Gradient Calculation*) system

Why Pytorch?

- **More Pythonic**
  - Flexible
  - Intuitive and cleaner code
  - Easy to learn & debug
  - Dynamic Computation Graph (*network behavior can be changed programmatically at runtime*)
- **More Neural Networkic**
  - Write code as the network works
  - forward/backward

Checking PyTorch version

```
import torch

print(torch.__version__)
```

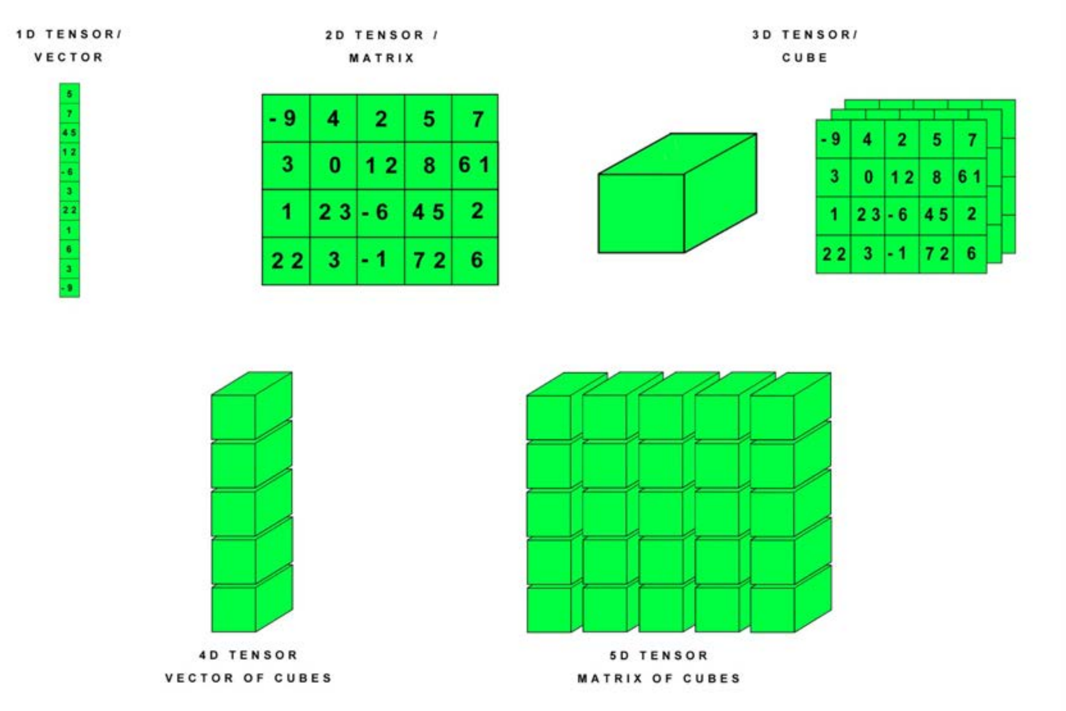
```
1.5.0+cu101
```

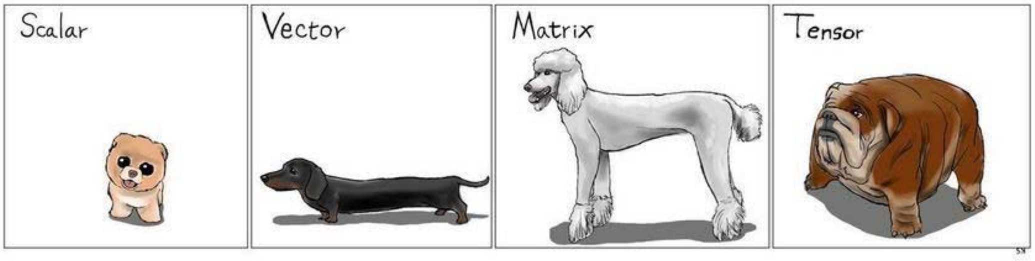
Introduction to Tensors

A **PyTorch Tensor** is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic **n-dimensional array** to be used for arbitrary **numeric computation**.

The biggest difference between a numpy array and a PyTorch Tensor is that a **PyTorch Tensor can run on either CPU or GPU**. To run operations on the GPU, **just cast the Tensor to a cuda datatype**.

A scalar is **zero-order tensor** or rank zero tensor. A vector is a **one-dimensional** or first order tensor, and a matrix is a **two-dimensional** or second order tensor.





A torch.Tensor is a **multi-dimensional matrix** containing elements of a **single data type**.

`torch.Tensor` is an alias for the default tensor type (`torch.FloatTensor`).

```
torch.tensor([[1., -1.], [1., -1.]])

     tensor([[ 1., -1.],
             [ 1., -1.]])
```

```
x = torch.rand(5, 3)
print(x)

     tensor([[0.9827, 0.6272, 0.9839],
             [0.3059, 0.6539, 0.4298],
             [0.1513, 0.3223, 0.4269],
             [0.8147, 0.7093, 0.9322],
             [0.6179, 0.9020, 0.0030]])
```

```
# Converting numpy arrays to tensors
import numpy as np
torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
# Converting numpy arrays to tensors
np_values = np.array([[1, 2, 3], [4, 5, 6]])

tensor_values = torch.from_numpy(np_values)

print (tensor_values)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
# A tensor of specific data type can be constructed by passing a torch.dtype

torch.zeros([2, 4], dtype=torch.int32)
```

```
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=torch.int32)
```

```
# The contents of a tensor can be accessed and modified using Python’s indexing and slicing n
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x[1][2])

# Modify a certain element
x[0][1] = 8
print(x)
```

```
tensor(6)
tensor([[1, 8, 3],
        [4, 5, 6]])
```

```
# Use torch.Tensor.item() to get a Python number from a tensor containing a single value

x = torch.tensor([[1]])
print (x)

print(x.item())

x = torch.tensor(2.5)

print(x.item())
print(x.item())

     tensor([[1]])
     1
     2.5
```

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x.size())

     torch.Size([2, 3])
```

```
# Tensor addition & subtraction
x = torch.rand(5, 3)
y = torch.rand(5, 3)

print(x)
print(y)

print(x + y)
print(x - y)

     tensor([[0.3006, 0.7646, 0.5699],
             [0.8115, 0.7876, 0.3602],
             [0.1724, 0.3275, 0.0543],
             [0.1840, 0.0529, 0.7849],
             [0.8471, 0.1306, 0.6419]])
     tensor([[0.0678, 0.3605, 0.4748],
             [0.4979, 0.9121, 0.1280],
             [0.6105, 0.7482, 0.8507],
             [0.0198, 0.8611, 0.7846],
             [0.3896, 0.1397, 0.8953]])
     tensor([[0.3684, 1.1252, 1.0447],
             [1.3094, 1.6997, 0.4882],
             [0.7830, 1.0757, 0.9049],
             [0.2038, 0.9140, 1.5694],
             [1.2367, 0.2703, 1.5372]])
     tensor([[ 2.3281e-01,  4.0406e-01,  9.5038e-02],
             [ 3.1355e-01, -1.2456e-01,  2.3215e-01],
             [-4.3807e-01, -4.2075e-01, -7.9637e-01],
             [ 1.6420e-01, -8.0820e-01,  2.6971e-04],
             [ 4.5753e-01, -9.0982e-03, -2.5342e-01]])
```

```
# Syntax 2 for Tensor addition & subtraction in PyTorch
print(torch.add(x, y))
print(torch.sub(x, y))

     tensor([[0.3684, 1.1252, 1.0447],
             [1.3094, 1.6997, 0.4882],
             [0.7830, 1.0757, 0.9049],
             [0.2038, 0.9140, 1.5694],
             [1.2367, 0.2703, 1.5372]])
     tensor([[ 2.3281e-01,  4.0406e-01,  9.5038e-02],
             [ 3.1355e-01, -1.2456e-01,  2.3215e-01],
             [-4.3807e-01, -4.2075e-01, -7.9637e-01],
             [ 1.6420e-01, -8.0820e-01,  2.6971e-04],
             [ 4.5753e-01, -9.0982e-03, -2.5342e-01]])
```

```
# Tensor Product & Transpose

mat1 = torch.randn(2, 3)
mat2 = torch.randn(3, 3)

print(mat1)
print(mat2)

print(torch.mm(mat1, mat2))

print(mat1.t())
```

```
tensor([[ 0.5743, -1.4231,  2.0308],
        [-0.8048,  0.6091,  0.6772]])
tensor([[-0.3789,  1.0735, -0.1960],
        [-0.4697, -0.3032,  0.1264],
        [ 0.5271,  0.5391,  1.0751]])
tensor([[ 1.5213,  2.1428,  1.8909],
        [ 0.3759, -0.6835,  0.9628]])
tensor([[ 0.5743, -0.8048],
        [-1.4231,  0.6091],
        [ 2.0308,  0.6772]])
```

```
# Elementwise multiplication
t = torch.Tensor([[1, 2], [3, 4]])
t.mul(t)
```

```
tensor([[ 1.,  4.],
        [ 9., 16.]])
```

```
# Shape, dimensions, and datatype of a tensor object

x = torch.rand(5, 3)

print('Tensor shape:', x.shape)   # t.size() gives the same
print('Number of dimensions:', x.dim())
print('Tensor type:', x.type())   # there are other types
```

```
Tensor shape: torch.Size([5, 3])
Number of dimensions: 2
Tensor type: torch.FloatTensor
```

```
# Slicing
t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Every row, only the last column
print(t[:, -1])
```

```
# First 2 rows, all columns
print(t[:2, :])

# Lower right most corner
print(t[-1:, -1:])
```

```
     tensor([3., 6., 9.])
     tensor([[1., 2., 3.],
             [4., 5., 6.]])
     tensor([[9.]])
```

Linear Regression

PyTorch Model Designing Steps

1. **Design your model using class with Variables**
2. **Construct loss and optimizer (select from PyTorch API)**
3. **Training cycle (forward, backward, update)**

Step #1 : Design your model using class with Variables

```
from torch import nn
import torch
from torch import tensor

import matplotlib.pyplot as plt

x_data = tensor([[1.0], [2.0], [3.0], [4.0], [5.0], [6.0]])
y_data = tensor([[2.0], [4.0], [6.0], [8.0], [10.0], [12.0]])

# Hyper-parameters
input_size = 1
output_size = 1
num_epochs = 50
learning_rate = 0.01
```

```
print(torch.__version__)

print(torch.cuda.get_device_name())
```

```
     1.5.0+cu101
     Tesla K80
```

Using GPU for the PyTorch Models

Remember always 2 things must be on GPU

- model
- tensors

```
class Model(nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate nn.Linear module
        """
        super().__init__()
        self.linear = torch.nn.Linear(input_size, output_size)  # One in and one out

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.linear(x)
        return y_pred


# our model
model = Model()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
     Model(
       (linear): Linear(in_features=1, out_features=1, bias=True)
     )
```

Explanations:-

`torch.nn.Linear(in_features, out_features, bias=True)`

Applies a linear transformation to the incoming data: $y = W^T * x + b$

Parameters:

- `in_features` – size of each input sample (i.e. size of x)
- `out_features` – size of each output sample (i.e. size of y)
- `bias` – If set to False, the layer will not learn an additive bias. **Default: True**

Step #2 : Construct loss and optimizer (select from PyTorch API)

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Explanations:-

MSE Loss: Mean Squared Error (Default: 'mean')

- $\hat{y}$: prediction
- y : true value

$$MSE\ (sum) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

$$MSE\ (mean) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Step #3 : Training: forward, loss, backward, step

```
# Credit: https://github.com/jcjohnson/pytorch-examples

# Training loop
for epoch in range(num_epochs):
    # 1) Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data.to(device))

    # 2) Compute and print loss
    loss = criterion(y_pred, y_data.to(device))
    print(f'Epoch: {epoch} | Loss: {loss.item()} ')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    # Getting gradients w.r.t. parameters
    loss.backward()
    # Updating parameters
    optimizer.step()


# After training
hour_var = tensor([[7.0]]).to(device)
y_pred = model(hour_var)
print("Prediction (after training)",  7, model(hour_var).data[0][0].item())
```

```
Epoch: 0 | Loss: 0.4046969413757324
Epoch: 1 | Loss: 0.3751128613948822
Epoch: 2 | Loss: 0.34870368242263794
Epoch: 3 | Loss: 0.32503488659858704
Epoch: 4 | Loss: 0.30373701453208923
Epoch: 5 | Loss: 0.28449591994285583
Epoch: 6 | Loss: 0.2670438885688782
Epoch: 7 | Loss: 0.2511536478996277
```

```
Epoch: 8 | Loss: 0.23663079738616943
Epoch: 9 | Loss: 0.22330956161022186
Epoch: 10 | Loss: 0.2110479176044464
Epoch: 11 | Loss: 0.1997237652540207
Epoch: 12 | Loss: 0.1892329305410385
Epoch: 13 | Loss: 0.17948590219020844
Epoch: 14 | Loss: 0.17040419578552246
Epoch: 15 | Loss: 0.1619214415550232
Epoch: 16 | Loss: 0.15397928655147552
Epoch: 17 | Loss: 0.1465272456407547
Epoch: 18 | Loss: 0.13952045142650604
Epoch: 19 | Loss: 0.1329210102558136
Epoch: 20 | Loss: 0.12669487297534943
Epoch: 21 | Loss: 0.12081220746040344
Epoch: 22 | Loss: 0.11524614691734314
Epoch: 23 | Loss: 0.1099737286567688
Epoch: 24 | Loss: 0.10497380048036575
Epoch: 25 | Loss: 0.10022743046283722
Epoch: 26 | Loss: 0.09571778774261475
Epoch: 27 | Loss: 0.0914301723241806
Epoch: 28 | Loss: 0.08735045045614243
Epoch: 29 | Loss: 0.08346641808748245
Epoch: 30 | Loss: 0.079766184091568
Epoch: 31 | Loss: 0.07623954117298126
Epoch: 32 | Loss: 0.07287701219320297
Epoch: 33 | Loss: 0.06966972351074219
Epoch: 34 | Loss: 0.06660932302474976
Epoch: 35 | Loss: 0.0636879950761795
Epoch: 36 | Loss: 0.060898974537849426
Epoch: 37 | Loss: 0.05823547765612602
Epoch: 38 | Loss: 0.05569145083427429
Epoch: 39 | Loss: 0.053260963410139084
Epoch: 40 | Loss: 0.05093876272439957
Epoch: 41 | Loss: 0.04871952161192894
Epoch: 42 | Loss: 0.04659825190901756
Epoch: 43 | Loss: 0.044570740312337875
Epoch: 44 | Loss: 0.04263252019882202
Epoch: 45 | Loss: 0.04077941179275513
Epoch: 46 | Loss: 0.039007507264614105
Epoch: 47 | Loss: 0.0373133048415184
Epoch: 48 | Loss: 0.035693153738975525
Epoch: 49 | Loss: 0.034143973141908646
Prediction (after training) 7 13.88995361328125
```

Explanations:-

- Calling `.backward()` mutiple times accumulates the gradient (**by addition**) for each parameter.
- This is why you should call `optimizer.zero_grad()` after each .step() call.
- Note that following the first `.backward` call, a second call is only possible after you have performed another **forward pass**.

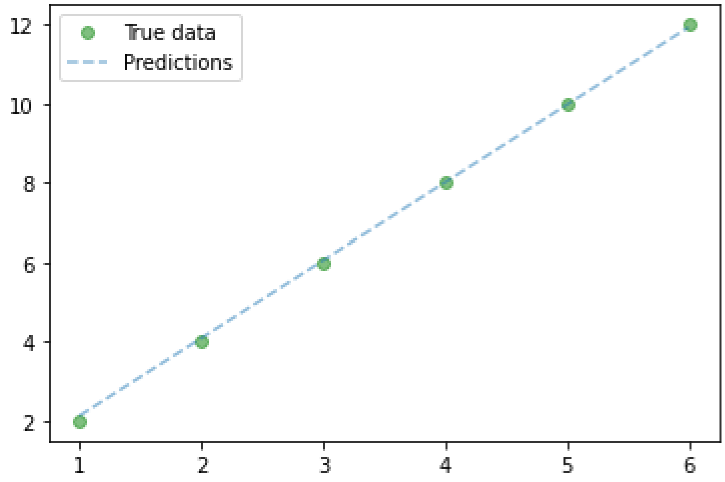
- `optimizer.step` performs a parameter update based on the current gradient (**stored in .grad attribute of a parameter**)

Simplified equation:-

- `parameters = parameters - learning_rate * parameters_gradients`
- parameters W and b in ($y = W^T * x + b$)
- $\theta = \theta - \eta \cdot \nabla_\theta$ [General parameter θ]
    - θ : parameters (our variables)
    - η : learning rate (how fast we want to learn)
    - ∇_θ : parameters' gradients

Plot of predicted and actual values

```
# Clear figure
plt.clf()

# Get predictions
predictions = model(x_data.to(device)).cpu().detach().numpy()

# Plot true data
plt.plot(x_data, y_data, 'go', label='True data', alpha=0.5)

# Plot predictions
plt.plot(x_data, predictions, '--', label='Predictions', alpha=0.5)

# Legend and plot
plt.legend(loc='best')
plt.show()
```



Saving Model to Directory

```
from google.colab import drive

drive.mount('/content/gdrive')

root_path = '/content/gdrive/My Drive/AUST Teaching Docs/AUST Fall 2019/Soft Computing/CSE 42
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318

Enter your authorization code:
··········
Mounted at /content/gdrive
```

Save Model

```
save_model = True

if save_model is True:
    # Saves only parameters
    # wights & biases
    torch.save(model.state_dict(), root_path + 'linear_regression.pkl')

# Save the model checkpoint
# torch.save(model.state_dict(), root_path + 'linear_regression.ckpt')
```

Load Model

```
load_model = True

if load_model is True:
    model.load_state_dict(torch.load(root_path + 'linear_regression.pkl'))
```

Try Other Optimizers

- torch.optim.Adagrad
- torch.optim.Adam
- torch.optim.Adamax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop

- torch.optim.SGD

* Official PyTorch Tutorials *

https://pytorch.org/tutorials/

Basic Comparison

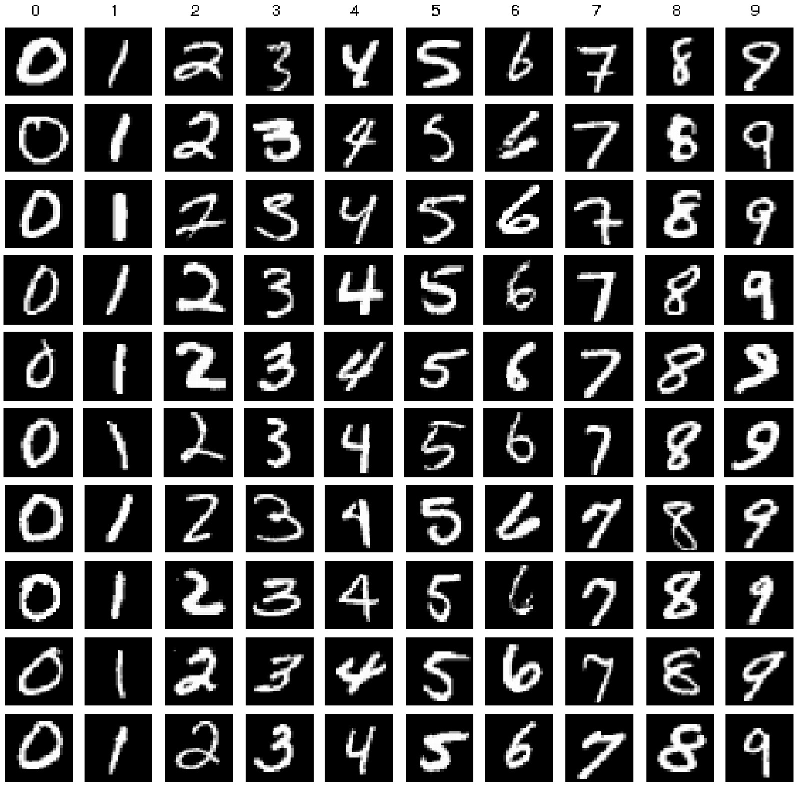
- **Linear Regression**
    - Output: numeric value given inputs
- **Logistic Regression**
    - Output: probability [0, 1] given input belonging to a class

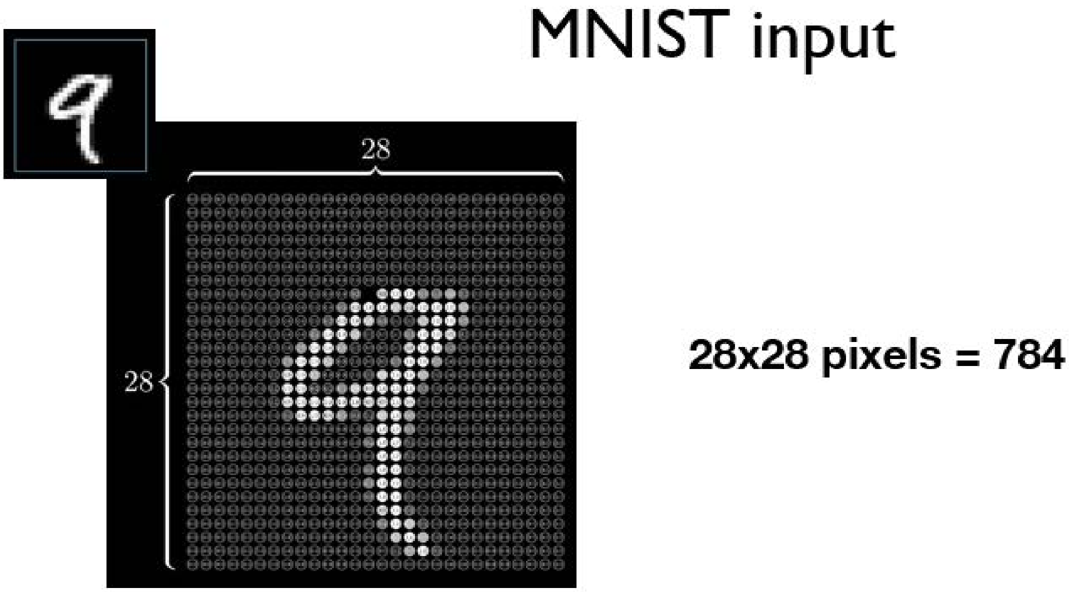
Logistic Regression Example: Positive vs Negative

Input: Sequence of Words

Output: Probability of positive

- Input: "Delivery speed was good"
- Output: **p = 0.8**
- Input: "Terrible Customer Service"
- Output: **p = 0.2**

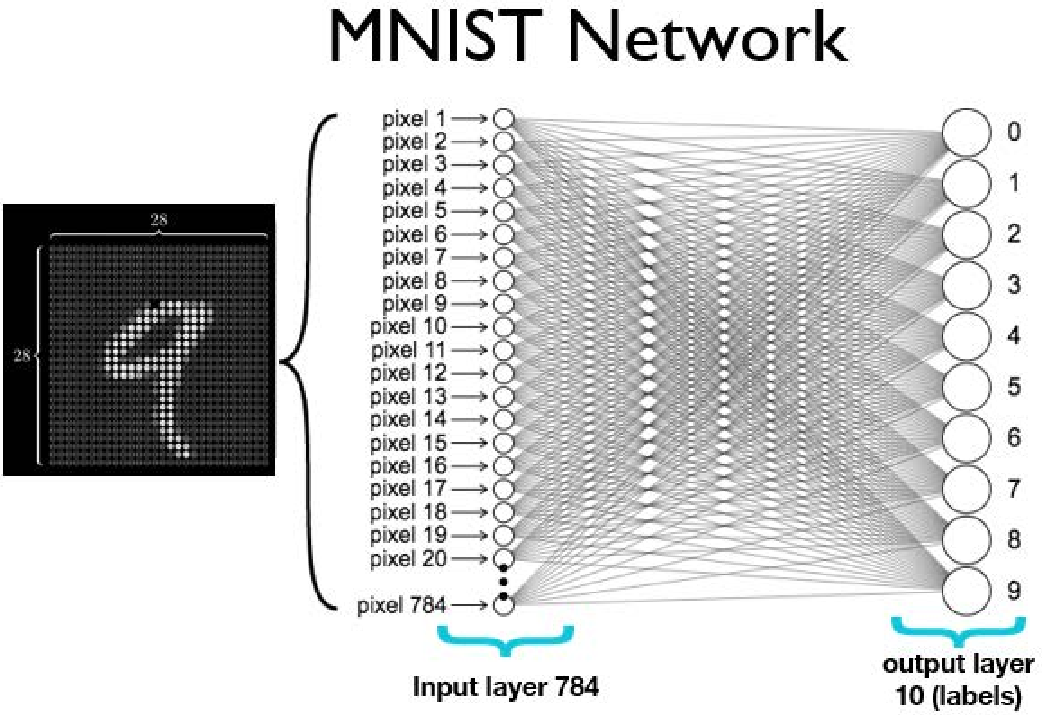




MNIST input

28x28 pixels = 784

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```



MNIST Network

- **Input dimension:**
  - Size of image: $28 \times 28 = 784$

- **Output dimension: 10**
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
# Hyperparameters

batch_size = 100
num_iters = 3000
input_dim = 28*28 # num_features = 784
output_dim = 10

learning_rate = 0.001

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Loading MNIST Dataset

- **totaldata:** 60,000
- **minibatch:** 100
    - Number of examples in **1** iteration
- **iterations:** 3,000
    - *1 iteration: one mini-batch forward & backward pass. That means a parameter (wights and biases) update.*
- **epochs**
    - 1 epoch: running through the whole dataset once
    - $epochs = iterations \div \frac{totaldata}{minibatch} = 3000 \div \frac{60000}{100} = 5$

```
'''
LOADING DATASET
'''
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

'''
MAKING DATASET ITERABLE
'''
```

```
num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

```
print(len(train_dataset))
print(len(test_dataset))
```

```
60000
10000
```

```
# Inspecting a single image (28 pixel x 28 pixel) -->  28x28 matrix of numbers

train_dataset[0]
```

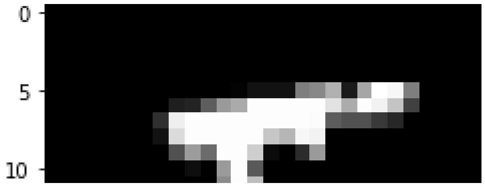
```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0706, 0.0706, 0.0706,
           0.4941, 0.5333, 0.6863, 0.1020, 0.6510, 1.0000, 0.9686, 0.4980,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.1176, 0.1412, 0.3686, 0.6039, 0.6667, 0.9922, 0.9922, 0.9922,
           0.9922, 0.9922, 0.8824, 0.6745, 0.9922, 0.9490, 0.7647, 0.2510,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1922,
           0.9333, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922,
           0.9922, 0.9843, 0.3647, 0.3216, 0.3216, 0.2196, 0.1529, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0706,
           0.8588, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922, 0.7765, 0.7137,
           0.9686, 0.9451, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.3137, 0.6118, 0.4196, 0.9922, 0.9922, 0.8039, 0.0431, 0.0000,
           0.1686, 0.6039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0549, 0.0039, 0.6039, 0.9922, 0.3529, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.5451, 0.9922, 0.7451, 0.0078, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0431, 0.7451, 0.9922, 0.2745, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.1373, 0.9451, 0.8824, 0.6275,
           0.4235, 0.0039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3176, 0.9412, 0.9922,
           0.9922, 0.4667, 0.0980, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

```
# One Image Size
print(train_dataset[0][0].size())
print(train_dataset[0][0].numpy().shape)
# First Image Label
print(train_dataset[0][1])
```

```
torch.Size([1, 28, 28])
(1, 28, 28)
5
```

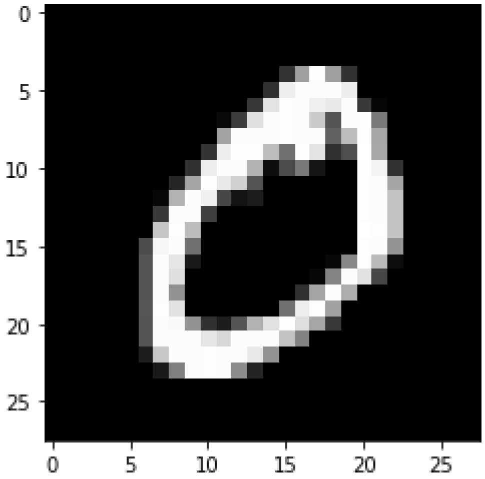
```
## Displaying a MNIST Image

import matplotlib.pyplot as plt
import numpy as np

show_img = train_dataset[0][0].numpy().reshape(28, 28)
plt.imshow(show_img, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f5f6d2845c0>
```



```
## Displaying another MNIST Image
# Label
print("Label:")
print(train_dataset[1][1])

show_img = train_dataset[1][0].numpy().reshape(28, 28)
plt.imshow(show_img, cmap='gray')
```

```
Label:
0
<matplotlib.image.AxesImage at 0x7f5f6d21deb8>
```



Step #1 : Design your model using class

```
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_size, num_classes):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        logits  = self.linear(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas


'''
INSTANTIATE MODEL CLASS
'''
model = LogisticRegressionModel(input_size=input_dim,
                                num_classes=output_dim)
```

```
# To enable GPU
model.to(device)
```

```
LogisticRegressionModel(
  (linear): Linear(in_features=784, out_features=10, bias=True)
)
```

Step #2 : Construct loss and optimizer (select from PyTorch API)

Unlike linear regression, we do not use MSE here, we need Cross Entropy Loss to calculate our loss before we backpropagate and update our parameters.

```
criterion = nn.CrossEntropyLoss()
```

It does 2 things at the same time.

1. Computes softmax ([Logistic or Sigmoid]/softmax function)
2. Computes Cross Entropy Loss

```
# INSTANTIATE OPTIMIZER CLASS
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

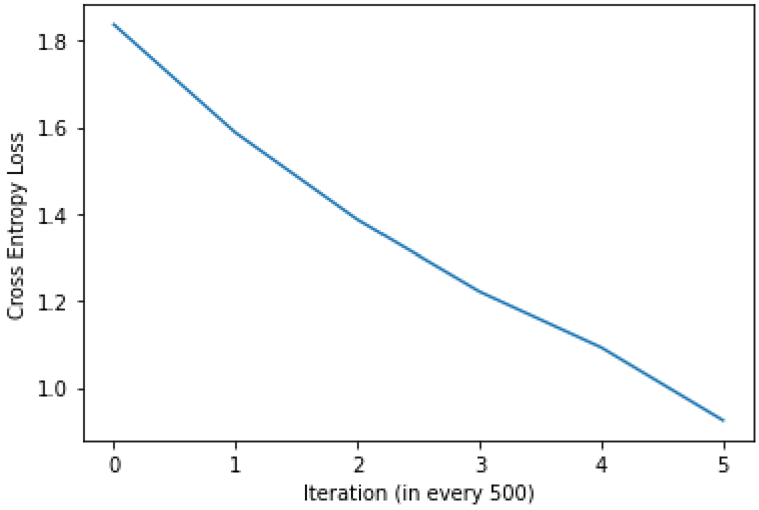
Step #3 : Training: forward, loss, backward, step

```
'''
TRAIN THE MODEL
'''
iteration_loss = []
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        logits, probas = model(images)

        # Calculate Loss: PyTorch implementation of CrossEntropyLoss works with logits, not p
        loss = F.cross_entropy(logits, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.view(-1, 28*28).to(device)

                # Forward pass only to get logits/output
                logits, probas = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(probas, 1)

                # Total number of labels
                total += labels.size(0)


                # Total correct predictions
                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            iteration_loss.append(loss.item())
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 1.8367596864700317. Accuracy: 68.15
Iteration: 1000. Loss: 1.5870451927185059. Accuracy: 76.65
Iteration: 1500. Loss: 1.3868762254714966. Accuracy: 79.71
Iteration: 2000. Loss: 1.2214820384979248. Accuracy: 81.31
Iteration: 2500. Loss: 1.092697024345398. Accuracy: 82.53
Iteration: 3000. Loss: 0.9246507287025452. Accuracy: 83.12
```

```
import matplotlib
import matplotlib.pyplot as plt

print (iteration_loss)
plt.plot(iteration_loss)
plt.ylabel('Cross Entropy Loss')
plt.xlabel('Iteration (in every 500)')
plt.show()
```

```
[1.8367596864700317, 1.5870451927185059, 1.3868762254714966, 1.2214820384979248, 1.0926
```



```
from google.colab import drive

drive.mount('/content/gdrive')

root_path = '/content/gdrive/My Drive/AUST Teaching Docs/AUST Fall 2019/Soft Computing/CSE 42
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318

Enter your authorization code:
··········
Mounted at /content/gdrive
```

Save Model

```
save_model = True

if save_model is True:
    # Saves only parameters
    # wights & biases
    torch.save(model.state_dict(), root_path + 'MNIST_logistic.pkl')
```

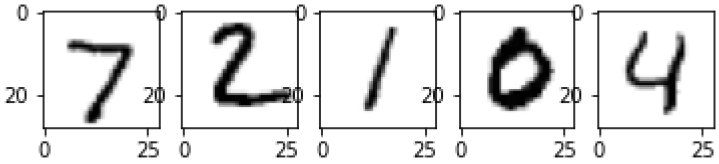
Load Model

```
load_model = True

if load_model is True:
    model.load_state_dict(torch.load(root_path + 'MNIST_logistic.pkl'))
    print('Trained Model Loaded')
```

```
Trained Model Loaded
```

Testing Loaded Model with Digits

```
for images, labels in test_loader:
    break

fig, ax = plt.subplots(1, 5)
for i in range(5):
    ax[i].imshow(images[i].view(28, 28), cmap=matplotlib.cm.binary)

plt.show()
```

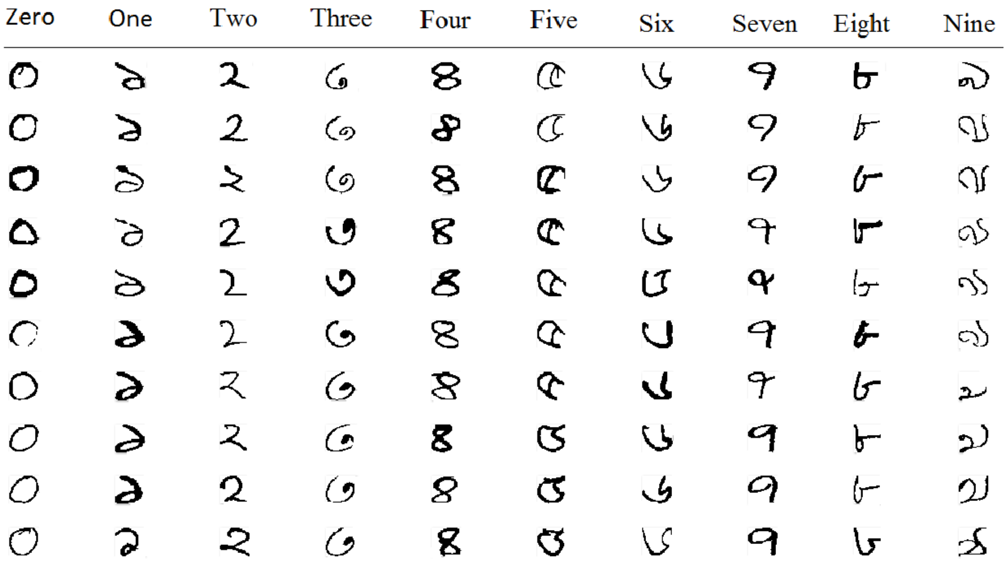


```
_, predictions = model.forward(images[:5].view(-1, 28*28).to(device))
predictions = torch.argmax(predictions, dim=1)
print('Predicted labels', predictions.cpu().numpy())

     Predicted labels [7 2 1 0 4]
```

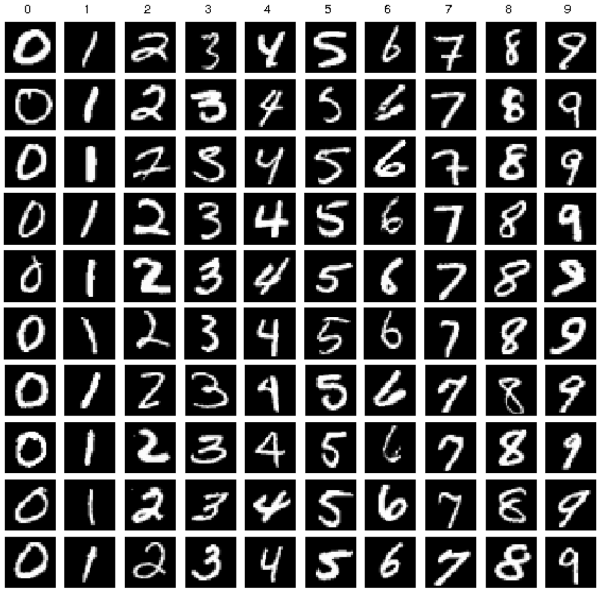
NumtaDB: Bengali Handwritten Digits

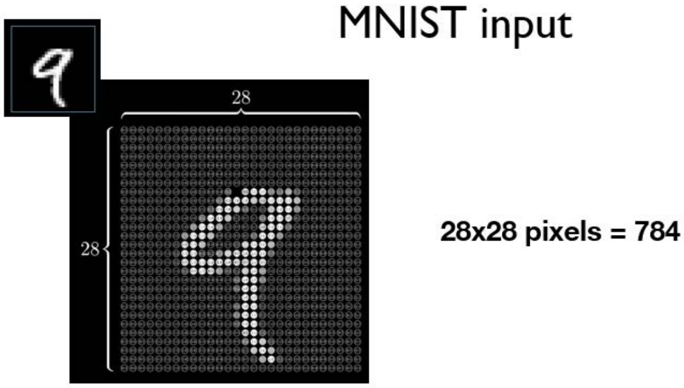
Dataset Link: https://www.kaggle.com/BengaliAI/numta/

Snapshot from NumtaDB



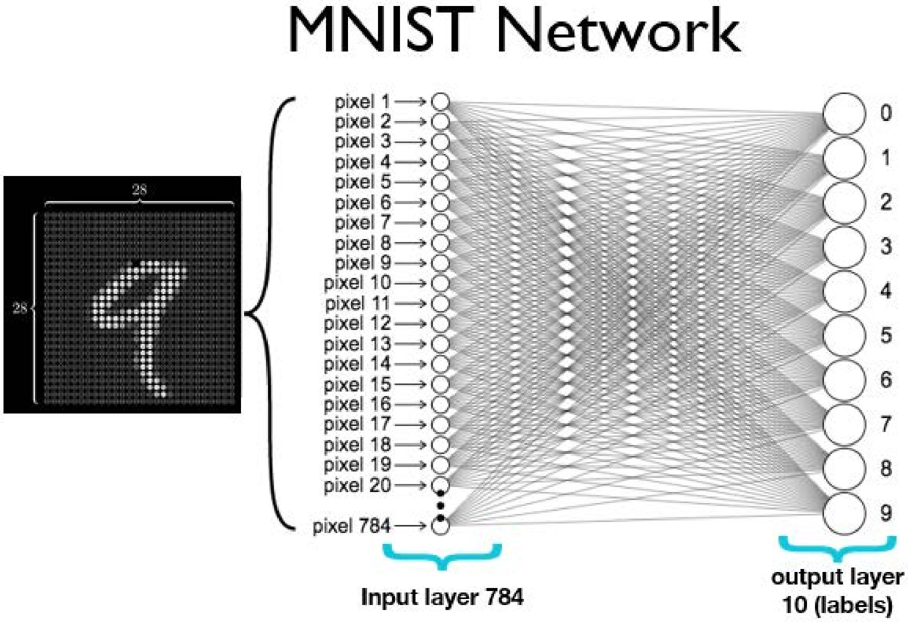
MNIST Digit Recognizer (Neural Network)





One Layer FNN with Sigmoid Activation

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```



- Our input size is determined by the size of the image **(height x width) = (28X28)**. Hence the size of our input is **784 (28 x 28)**.
- When we pass an image to our model, it will try to predict if it's **0, 1, 2, 3, 4, 5, 6, 7, 8, or 9**. That is a total of 10 classes, hence we have an output size of 10.
- Determining the **hidden layer size** is one of the crutial part. The first layer prior to the non-linear layer. This can be any **real number**. A large number of hidden nodes denotes a **bigger model with more parameters**.
- The bigger model isn't **always the better model**. On the otner hand, bigger model requires **more training samples** to learn and converge to a good model.
- Actually a bigger model **requires more training samples** to learn and converge to a good model. Hence, it is wise to pick the model size for the problem at hand. Because it is a simple problem of recognizing digits, we typically would not need a big model to achieve good results.
- Moreover, too small of a hidden size would mean there would be **insufficient model capacity to predict competently**. Too small of a capacity denotes a **smaller brain capacity** so no matter how many training samples you provide, it has a maximum capacity boundary in terms of its **predictive power**.

- **Input dimension:**
  - Size of image: $28 \times 28 = 784$
- **Output dimension: 10**

    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
# Hyperparameters

batch_size = 100
num_iters = 3000
input_dim = 28*28 # num_features = 784
num_hidden = 100 # num of hidden nodes
output_dim = 10

learning_rate = 0.1  # More power so we can learn faster! previously it was 0.001

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Loading MNIST Dataset

```
'''
LOADING DATASET
'''
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

'''
MAKING DATASET ITERABLE
'''
num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

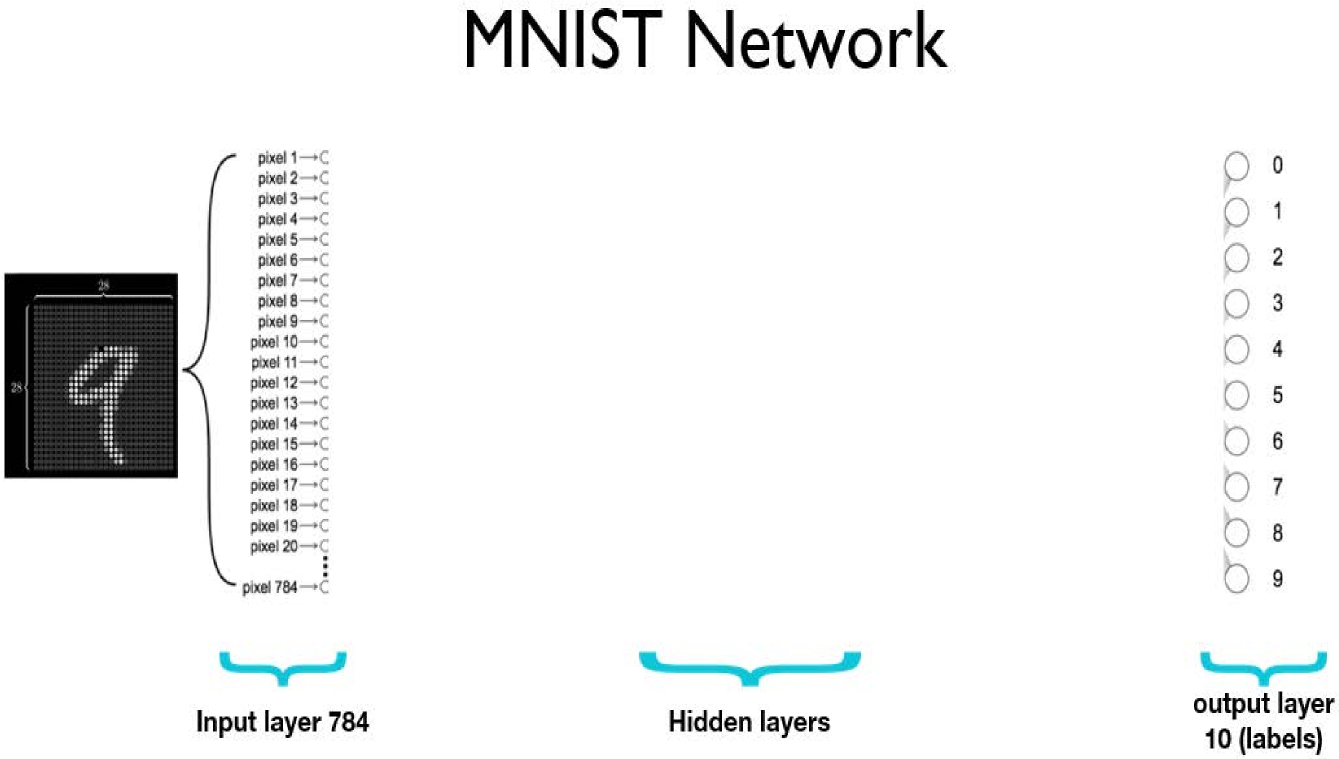
```
    Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/
                                            9920512/? [00:20<00:00, 1332586.75it/s]

    Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
    Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/
    0%                                      0/28881 [00:00<?, ?it/s]

    Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
    Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/r
                                            1654784/? [00:18<00:00, 1026194.01it/s]

    Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
    Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/r
    0%                                      0/4542 [00:00<?, ?it/s]
```

```
print(len(train_dataset))
print(len(test_dataset))
```

```
    60000
    10000
```

```
# One Image Size
print(train_dataset[0][0].size())
print(train_dataset[0][0].numpy().shape)
# First Image Label
print(train_dataset[0][1])
```

```
    torch.Size([1, 28, 28])
    (1, 28, 28)
    5
```



Step #1 : Design your model using class

```
class NeuralNetworkModel(nn.Module):
    def __init__(self, input_size, num_classes, num_hidden):
        super().__init__()
        ### 1st hidden layer
        self.linear_1 = nn.Linear(input_size, num_hidden)

        ### Non-linearity
        self.sigmoid = nn.Sigmoid()

        ### Output layer
        self.linear_out = nn.Linear(num_hidden, num_classes)

    def forward(self, x):
        # Linear layer
        out  = self.linear_1(x)
        # Non-linearity
        out = self.sigmoid(out)
        # Linear layer (output)
        probas  = self.linear_out(out)
        return probas


'''
INSTANTIATE MODEL CLASS
'''
model = NeuralNetworkModel(input_size = input_dim,
                           num_classes = output_dim,
                           num_hidden = num_hidden)
# To enable GPU
model.to(device)
```

```
NeuralNetworkModel(
  (linear_1): Linear(in_features=784, out_features=100, bias=True)
  (sigmoid): Sigmoid()
  (linear_out): Linear(in_features=100, out_features=10, bias=True)
)
```

Step #2 : Construct loss and optimizer

Unlike linear regression, we do not use MSE here, we need Cross Entropy Loss to calculate our loss before we backpropagate and update our parameters.

```
criterion = nn.CrossEntropyLoss()
```

It does 2 things at the same time.

1. Computes softmax ([Logistic or Sigmoid]/softmax function)

2. Computes Cross Entropy Loss

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Step #3 : Training: forward, loss, backward, step

```
'''
TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.view(-1, 28*28).to(device)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs, 1)

                # Total number of labels
                total += labels.size(0)


                # Total correct predictions
                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 0.6597447991371155. Accuracy: 86.5
Iteration: 1000. Loss: 0.41724541783332825. Accuracy: 89.48
Iteration: 1500. Loss: 0.4041314721107483. Accuracy: 90.35
Iteration: 2000. Loss: 0.3359662592411041. Accuracy: 90.97
Iteration: 2500. Loss: 0.22867584228515625. Accuracy: 91.64
Iteration: 3000. Loss: 0.24442128837108612. Accuracy: 91.95
```

Expanding Neural Network variants

2 ways to expand a neural network

- Different non-linear activation
- More hidden layers

One Layer FNN with Tanh Activation

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Hyperparameters
batch_size = 100
num_iters = 3000
input_dim = 28*28 # num_features = 784
num_hidden = 100
output_dim = 10

learning_rate = 0.1

# Device
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())


num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

class NeuralNetworkModel(nn.Module):
    def __init__(self, input_size, num_classes, num_hidden):
        super().__init__()
        ### 1st hidden layer
        self.linear_1 = nn.Linear(input_size, num_hidden)

        ### Non-linearity
        self.tanh = nn.Tanh()

        ### Output layer
        self.linear_out = nn.Linear(num_hidden, num_classes)

    def forward(self, x):
        # Linear layer
        out  = self.linear_1(x)
        # Non-linearity
        out = self.tanh(out)
        # Linear layer (output)
        probas  = self.linear_out(out)
        return probas

model = NeuralNetworkModel(input_size = input_dim,
                           num_classes = output_dim,
                           num_hidden = num_hidden)
# To enable GPU
model.to(device)

criterion = nn.CrossEntropyLoss()
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

    iter = 0
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):

            images = images.view(-1, 28*28).to(device)
            labels = labels.to(device)

            # Clear gradients w.r.t. parameters
            optimizer.zero_grad()

            # Forward pass to get output/logits
            outputs = model(images)

            # Calculate Loss: softmax --> cross entropy loss
            loss = criterion(outputs, labels)

            # Getting gradients w.r.t. parameters
            loss.backward()

            # Updating parameters
            optimizer.step()

            iter += 1

            if iter % 500 == 0:
                # Calculate Accuracy
                correct = 0
                total = 0
                # Iterate through test dataset
                for images, labels in test_loader:

                    images = images.view(-1, 28*28).to(device)

                    # Forward pass only to get logits/output
                    outputs = model(images)

                    # Get predictions from the maximum value
                    _, predicted = torch.max(outputs, 1)

                    # Total number of labels
                    total += labels.size(0)


                    # Total correct predictions
                    if torch.cuda.is_available():
                        correct += (predicted.cpu() == labels.cpu()).sum()
                    else:
                        correct += (predicted == labels).sum()
                accuracy = 100 * correct.item() / total

                # Print Loss
                print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 0.21413597464561462. Accuracy: 90.9
Iteration: 1000. Loss: 0.3538866341114044. Accuracy: 92.31
Iteration: 1500. Loss: 0.15589021146297455. Accuracy: 93.24
Iteration: 2000. Loss: 0.3556366264820099. Accuracy: 93.98
Iteration: 2500. Loss: 0.2028314620256424. Accuracy: 94.64
Iteration: 3000. Loss: 0.333248496055603. Accuracy: 95.05
```

One Layer FNN with ReLU Activation

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Hyperparameters
batch_size = 100
num_iters = 3000
input_dim = 28*28 # num_features = 784
num_hidden = 100
output_dim = 10

learning_rate = 0.1

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())


num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

class NeuralNetworkModel(nn.Module):
    def __init__(self, input_size, num_classes, num_hidden):
        super().__init__()
        ### 1st hidden layer
        self.linear_1 = nn.Linear(input_size, num_hidden)

        ### Non-linearity
        self.relu = nn.ReLU()

        ### Output layer
        self.linear_out = nn.Linear(num_hidden, num_classes)

    def forward(self, x):
        # Linear layer
        out  = self.linear_1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear layer (output)
        probas  = self.linear_out(out)
        return probas

model = NeuralNetworkModel(input_size = input_dim,
                           num_classes = output_dim,
                           num_hidden = num_hidden)
# To enable GPU
model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()
        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.view(-1, 28*28).to(device)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs, 1)

                # Total number of labels
                total += labels.size(0)


                # Total correct predictions
                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 0.18737341463565826. Accuracy: 91.48
Iteration: 1000. Loss: 0.3523785471916199. Accuracy: 93.14
Iteration: 1500. Loss: 0.22952955961227417. Accuracy: 93.83
Iteration: 2000. Loss: 0.09236818552017212. Accuracy: 94.86
Iteration: 2500. Loss: 0.262081503868103. Accuracy: 95.21
Iteration: 3000. Loss: 0.14769437909126282. Accuracy: 95.89
```

Two Layer FNN with ReLU Activation

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

```
# Hyperparameters
batch_size = 100
num_iters = 3000
input_dim = 28*28 # num_features = 784
num_hidden = 100
output_dim = 10

learning_rate = 0.1

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())


num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

class DeepNeuralNetworkModel(nn.Module):
    def __init__(self, input_size, num_classes, num_hidden):
        super().__init__()
        ### 1st hidden layer: 784 --> 100
        self.linear_1 = nn.Linear(input_size, num_hidden)
        ### Non-linearity in 1st hidden layer
        self.relu_1 = nn.ReLU()

        ### 2nd hidden layer: 100 --> 100
        self.linear_2 = nn.Linear(num_hidden, num_hidden)
        ### Non-linearity in 2nd hidden layer
        self.relu_2 = nn.ReLU()

        ### Output layer: 100 --> 10
        self.linear_out = nn.Linear(num_hidden, num_classes)

    def forward(self, x):
        ### 1st hidden layer
        ### 1st hidden layer
        out  = self.linear_1(x)
        ### Non-linearity in 1st hidden layer
        out = self.relu_1(out)

        ### 2nd hidden layer
        out  = self.linear_2(out)
        ### Non-linearity in 2nd hidden layer
        out = self.relu_2(out)

        # Linear layer (output)
        probas  = self.linear_out(out)
        return probas

# INSTANTIATE MODEL CLASS

model = DeepNeuralNetworkModel(input_size = input_dim,
                               num_classes = output_dim,
                               num_hidden = num_hidden)
# To enable GPU
model.to(device)

# INSTANTIATE LOSS & OPTIMIZER CLASS

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.view(-1, 28*28).to(device)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs, 1)

                # Total number of labels
                total += labels.size(0)


                # Total correct predictions
                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 0.38067626953125. Accuracy: 91.27
Iteration: 1000. Loss: 0.1768297553062439. Accuracy: 93.35
Iteration: 1500. Loss: 0.10338889807462692. Accuracy: 95.04
Iteration: 2000. Loss: 0.1981402188539505. Accuracy: 95.89
Iteration: 2500. Loss: 0.05458816513419151. Accuracy: 96.15
Iteration: 3000. Loss: 0.14130154252052307. Accuracy: 96.5
```

Three Layer FNN with ReLU Activation

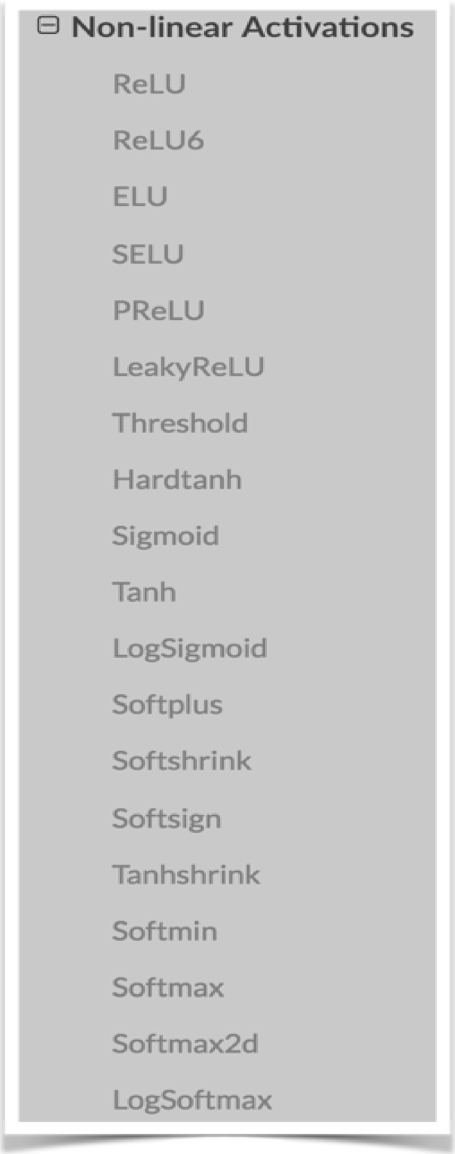
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Hyperparameters
batch_size = 100
num_iters = 3000
input_dim = 28*28 #num_features = 784
num_hidden = 100
output_dim = 10
```

```
learning_rate = 0.1

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1]
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())


num_epochs = num_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)   # It's better to shuffle the whole

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

class DeepNeuralNetworkModel(nn.Module):
    def __init__(self, input_size, num_classes, num_hidden):
        super().__init__()
        ### 1st hidden layer: 784 --> 100
        self.linear_1 = nn.Linear(input_size, num_hidden)
        ### Non-linearity in 1st hidden layer
        self.relu_1 = nn.ReLU()

        ### 2nd hidden layer: 100 --> 100
        self.linear_2 = nn.Linear(num_hidden, num_hidden)
        ### Non-linearity in 2nd hidden layer
        self.relu_2 = nn.ReLU()

        ### 3rd hidden layer: 100 --> 100
        self.linear_3 = nn.Linear(num_hidden, num_hidden)
        ### Non-linearity in 3rd hidden layer
        self.relu_3 = nn.ReLU()

        ### Output layer: 100 --> 10
        self.linear_out = nn.Linear(num_hidden, num_classes)

    def forward(self, x):
        ### 1st hidden layer
        out  = self.linear_1(x)
        ### Non-linearity in 1st hidden layer
        out = self.relu_1(out)

        ### 2nd hidden layer
        out  = self.linear_2(out)
        ### Non-linearity in 2nd hidden layer
        out = self.relu_2(out)

        ### 3rd hidden layer
        out  = self.linear_3(out)
        ### Non-linearity in 3rd hidden layer
        out = self.relu_3(out)

        # Linear layer (output)
        probas  = self.linear_out(out)
        return probas

# INSTANTIATE MODEL CLASS

model = DeepNeuralNetworkModel(input_size = input_dim,
                               num_classes = output_dim,
                               num_hidden = num_hidden)
# To enable GPU
model.to(device)

# INSTANTIATE LOSS & OPTIMIZER CLASS
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1
        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.view(-1, 28*28).to(device)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs, 1)

                # Total number of labels
                total += labels.size(0)


                # Total correct predictions
                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy)
```

```
Iteration: 500. Loss: 0.377877801656723. Accuracy: 90.61
Iteration: 1000. Loss: 0.2982105016708374. Accuracy: 94.45
Iteration: 1500. Loss: 0.2584376633167267. Accuracy: 94.73
Iteration: 2000. Loss: 0.08460734039545059. Accuracy: 95.71
Iteration: 2500. Loss: 0.10547266900539398. Accuracy: 95.72
Iteration: 3000. Loss: 0.04601271450519562. Accuracy: 96.85
```

What's Next?

- Try with other activations from Pytorch.
- Try different activations for different layers (We used ReLU Only)
- Try adding more hidden layers
- Try increasing the hidden layer neurons (We used 100 here in this example)
- Try experimenting with different neurons for different hidden layers (We here in this examples used a fixed sixe: 100)



⊟ Non-linear Activations

ReLU

ReLU6

ELU

SELU

PReLU

LeakyReLU

Threshold

Hardtanh

Sigmoid

Tanh

LogSigmoid

Softplus

Softshrink

Softsign

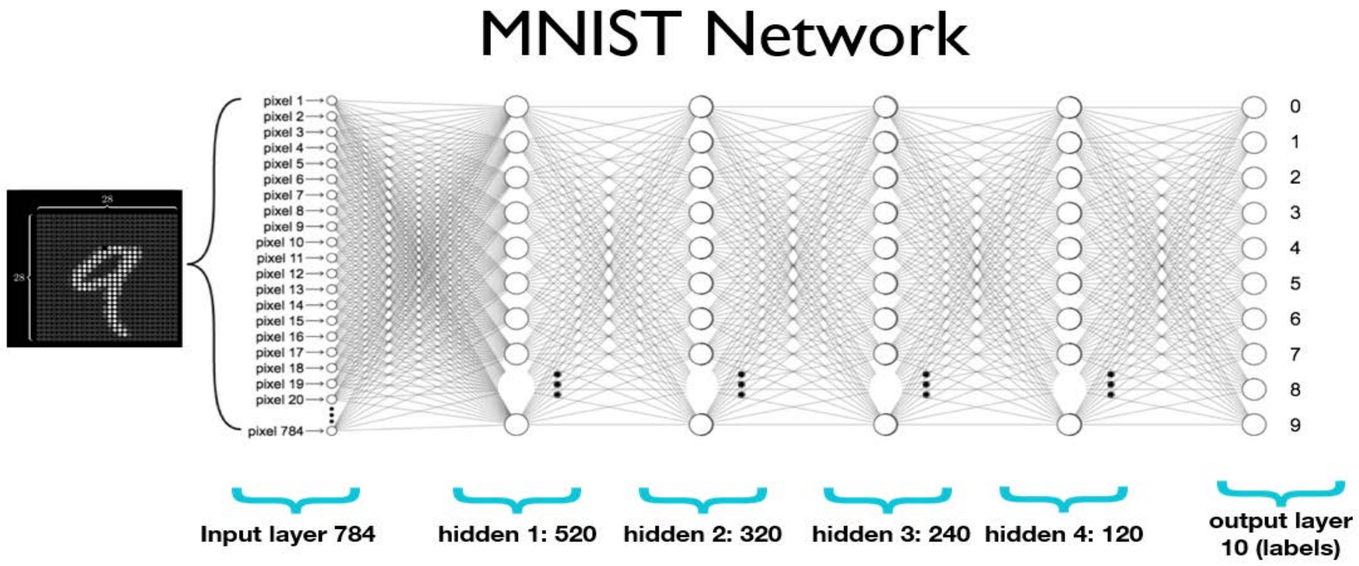
Tanhshrink

Softmin

Softmax

Softmax2d

LogSoftmax



MNIST Network

Natural Language Processing (NLP) - A hands-on introduction

Popular Libraries

- NLTK
- spaCy

NLTK & spaCy is a free open-source library for Natural Language Processing (NLP) in Python to support teaching, research, and development. Which are:-

- Free and Open source
- Easy to use
- Modular
- Well documented
- Simple and extensible

In this notebook, I will provide basic NLP tasks that we need in order to process raw text to find useful informations. For each tasks, we will be using NLTK as well as spaCy. Good news is that both are installed in Google Colab by default.

Some definitions

- **Corpus** - Corpora is the plural of Corpus. **"Corpus"** mainly appears in NLP area or application domain related to texts/documents, because of its meaning "a collection of written texts"
  - **Example:** A collection of news documents.
- **Dataset** - dataset appears in every application domain (in can be **image/video/text/numerical/mixed**) --- a collection of any kind of data is a dataset.
- **Lexicon** - vocabulary or list of Words and their meanings.
  - **Example:** English dictionary.
- **Token** - Each "entity" that is a part of whatever was split up based on rules.
  - For examples, each word is a token when a sentence is "tokenized" into words. Each sentence can also be a token, if you tokenized the sentences out of a paragraph.

Tokenization

Tokenization is the process of breaking a stream of text up into sentences, words, phrases, symbols, or other meaningful elements called tokens.

```
import nltk
nltk.download('punkt')

# For tokenizing words and sentences
from nltk.tokenize import word_tokenize, sent_tokenize

s = "Good muffins cost $3.88\nin New York. Please buy me two of them.\n\nThanks."

print (sent_tokenize(s))
print (word_tokenize(s))
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
['Good muffins cost $3.88\nin New York.', 'Please buy me two of them.', 'Thanks.']
['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York', '.', 'Please', 'buy', 'me
```

```
import spacy

# Small spaCy model
nlp = spacy.load("en_core_web_sm")

doc = nlp("Good muffins cost $3.88\nin New York. Please buy me two of them.\n\nThanks.")

print("\n\nTokenized Sentences")

for i, sent in enumerate(doc.sents):
        print('-->Sentence %d: %s' % (i, sent.text))

print("\n\nTokenized Words")

tokens = [token.text for token in doc]
print(tokens)
```

```


Tokenized Sentences
-->Sentence 0: Good muffins cost $3.88
in New York.
-->Sentence 1: Please buy me two of them.


-->Sentence 2: Thanks.


Tokenized Words
['Good', 'muffins', 'cost', '$', '3.88', '\n', 'in', 'New', 'York', '.', 'Please', 'buy
```

Downloading Large spaCy model

```
!python -m spacy download en_core_web_lg

import en_core_web_lg

nlp = en_core_web_lg.load()
```

```
Collecting en_core_web_lg==2.2.5
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_lg
     |████████████████████████████████| 827.9MB 1.2MB/s
Requirement already satisfied: spacy>=2.2.2 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: blis<0.5.0,>=0.4.0 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: srsly<1.1.0,>=1.0.2 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: numpy>=1.15.0 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: thinc==7.4.0 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /usr/local/lib/python3.6/dis
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: importlib-metadata>=0.20; python_version < "3.8" in /usr/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.6/dist-packages (from
Building wheels for collected packages: en-core-web-lg
  Building wheel for en-core-web-lg (setup.py) ... done
  Created wheel for en-core-web-lg: filename=en_core_web_lg-2.2.5-cp36-none-any.whl size
  Stored in directory: /tmp/pip-ephem-wheel-cache-inhjlz6a/wheels/2a/c1/a6/fc7a877b1efca
Successfully built en-core-web-lg
Installing collected packages: en-core-web-lg
Successfully installed en-core-web-lg-2.2.5
✓ Download and installation successful
You can now load the model via spacy.load('en_core_web_lg')
```

Filtering stopwords

- **Stopwords** are common words that **generally** do not contribute to the meaning of a sentence.
- Most search engines will filter stopwords out of search queries and documents in order to **save space and time** in their index.
    - Removing stopwords is not a hard and fast rule in NLP. It depends upon the task that we are working on.
    - For tasks like text classification, where the text is to be classified into different categories, stopwords are removed or excluded from the given text so that more focus

can be given to those words which define the meaning of the text.

- All Stopwords collection including Bengali.

```
nltk.download('stopwords')
from nltk.corpus import stopwords

# All english stopwords list
english_stops = set(stopwords.words('english'))

print (english_stops)

words = ['The', 'natural', 'language', 'processing', 'is', 'very', 'interesting']
filtered_words = [word for word in words if word.lower() not in english_stops]    # word.lowe

print(filtered_words)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
{'hers', 'wouldn', 'who', 'after', 'until', 'd', 'there', 'in', 'don', 'about', 'now',
['natural', 'language', 'processing', 'interesting']
```

```
spacy_stopwords = spacy.lang.en.stop_words.STOP_WORDS

print('Number of stop words: %d' % len(spacy_stopwords))
print('First ten stop words: %s' % list(spacy_stopwords)[:10])
```

```
Number of stop words: 326
First ten stop words: ['hers', 'various', 'nobody', 'who', 'after', 'until', 'per', 'lea
```

```
doc = nlp("Good muffins cost $3.88\nin New York. Please buy me two of them.\n\nThanks.")

tokens = [token.text for token in doc if not token.is_stop]

print(tokens)
```

```
['Good', 'muffins', 'cost', '$', '3.88', '\n', 'New', 'York', '.', 'buy', '.', '\n\n',
```

Adding Custom Stopwords

```
english_stops = set(stopwords.words('english'))

print (english_stops)

english_stops.remove('is')
english_stops.add('natural')


words = ['The', 'natural', 'language', 'processing', 'is', 'very', 'interesting']
filtered_words = [word for word in words if word.lower() not in english_stops]

print(filtered_words)
```

```
{'hers', 'wouldn', 'who', 'after', 'until', 'd', 'there', 'in', 'don', 'about', 'now',
['language', 'processing', 'is', 'interesting']
```

Edit Distance

The edit distance is the number of character changes necessary to transform the given word into the suggested word.

```
from nltk.metrics import edit_distance

print(edit_distance("Birthday","Bday"))

print(edit_distance("university", "varsity"))
```

```
4
4
```

Removing Punctuation

```
import string
import nltk

nltk.download('punkt')

puncset = list(string.punctuation)

sentence = "Hun Sen's Cambodian can't People's Party won 64 of the 122 parliamentary seats in

sentence = sentence.lower()
print(sentence)
sentence = nltk.word_tokenize(sentence)
print(sentence)
sentence = [i for i in sentence if i not in puncset] # Removing punctuation
print(sentence)
sentence = [w for w in sentence if w.isalpha()] # Removing numbers and punctuation
print(sentence)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
hun sen's cambodian can't people's party won 64 of the 122 parliamentary seats in party
['hun', 'sen', "'s", 'cambodian', 'ca', "n't", 'people', "'s", 'party', 'won', '64', 'o
['hun', 'sen', "'s", 'cambodian', 'ca', "n't", 'people', "'s", 'party', 'won', '64', 'o
['hun', 'sen', 'cambodian', 'ca', 'people', 'party', 'won', 'of', 'the', 'parliamentary
```

Normalizing Text

The goal of both stemming and lemmatization is to "**normalize**" words to their **common base form**, which is useful for many text-processing applications.

- **Stemming** = heuristically removing the affixes of a word, to get its **stem (root)**.
    - It is a rule-based process of stripping the suffixes **("ing", "ly", "es", "s" etc)** from a word
- **Lemmatization** = Lemmatization process involves first determining the part of speech of a word, and applying different normalization rules for each part of speech.

Consider:

- I was taking a **ride** in the car.
- I was **riding** in the car.

Imagine every word in the English language, every possible tense and affix you can put on a word. **Having individual dictionary entries per version would be highly redundant and inefficient.**

- Lisa **ate** the food and washed the dishes.
- They were **eating** noodles at a cafe.
- Don't you want to **eat** before we leave?
- We have just **eaten** our breakfast.
- It also **eats** fruit and vegetables.

Unfortunately, that is not the case with machines. **They treat these words differently**. Therefore, we need to normalize them to their root word, which is **"eat"** in our example.

Stemming

- One of the **most popular** stemming algorithms is the Porter stemmer, which has been around since 1979.
- Several other stemming algorithms provided by NLTK are Lancaster Stemmer and Snowball Stemmer.

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
```

```
example_words = ["python","pythoner","pythoning","pythoned","pythonly"]

for w in example_words:
  print(stemmer.stem(w))
```

```
python
python
python
python
pythonli
```

Lemmatization

Lemmatize takes a part of speech parameter, "pos." **If not supplied, the default is "noun"**.

```
## Lemmatization using NLTK

from nltk.stem import WordNetLemmatizer

nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize('cooking'))
print(lemmatizer.lemmatize('cooking', pos='v'))  # noun = n, verb = v, ajdective = a
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
cooking
cook
```

```
## Lemmatization using spaCy

doc = nlp('Jim bought 300 shares of Acme Corp. in 2006.')

lemma_words = []

for token in doc:
    lemma_words.append(token.lemma_)

print(lemma_words)
```

```
['Jim', 'buy', '300', 'share', 'of', 'Acme', 'Corp.', 'in', '2006', '.']
```

Comparison between stemming and lemmatizing

The major difference between these is, as you saw earlier, **stemming can often create non-existent words**, whereas **lemmas are actual words**, you can just look up in an English dictionary.

```
print(stemmer.stem('believes'))
print(lemmatizer.lemmatize('believes'))
```

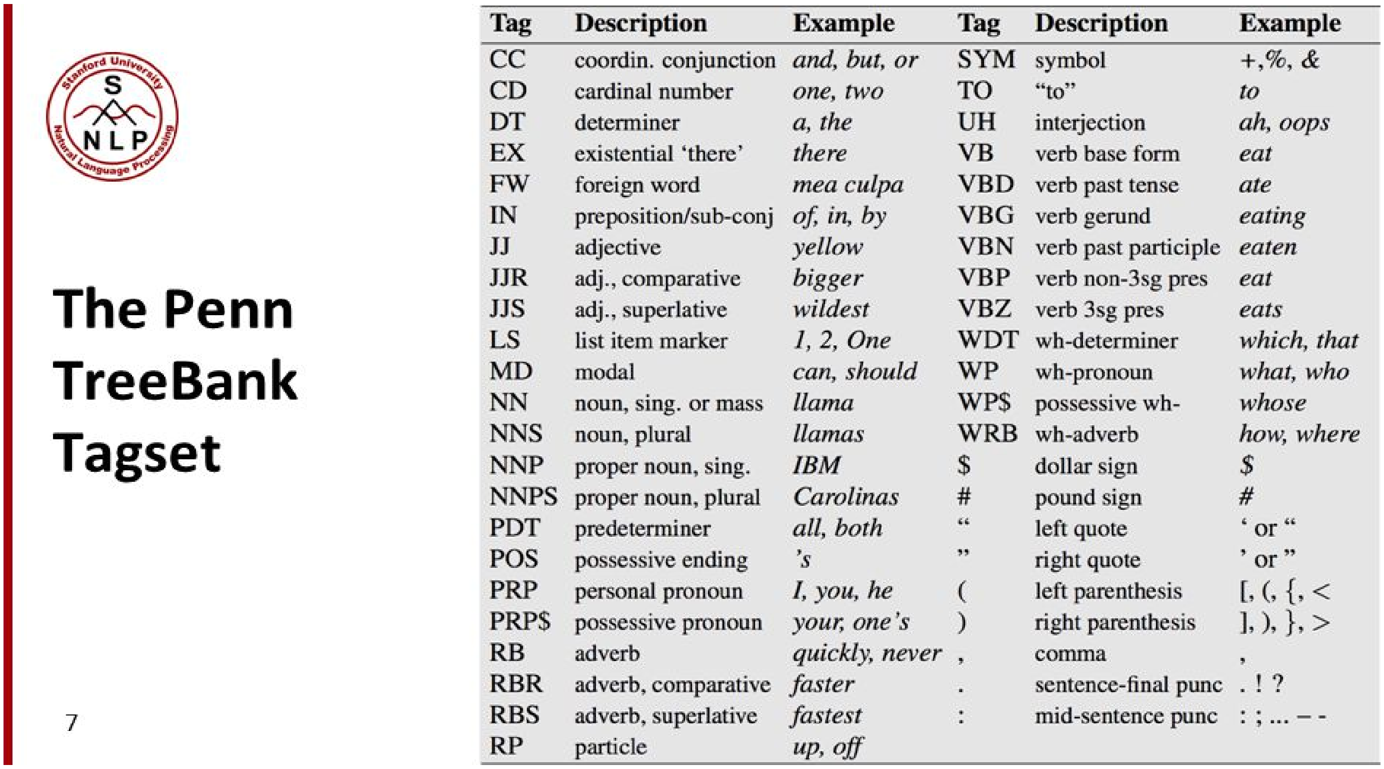
```
    believ
    belief
```

Part-of-speech Tagging

The English language is formed of different parts of speech (POS) like nouns, verbs, pronouns, adjectives, etc. POS tagging analyzes the words in a sentences and associates it with a POS tag depending on the way it is used.

Full tag list.

Penn Bank Part-of-Speech Tags



The Penn TreeBank Tagset

7

Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two*	TO	"to"	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential 'there'	*there*	VB	verb base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb gerund	*eating*
JJ	adjective	*yellow*	VBN	verb past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, sing.	*IBM*	$	dollar sign	*$*
NNPS	proper noun, plural	*Carolinas*	#	pound sign	*#*
PDT	predeterminer	*all, both*	"	left quote	' or "
POS	possessive ending	*'s*	"	right quote	' or "
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one's*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

```
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag

nltk.download('averaged_perceptron_tagger')

words = word_tokenize('Jim bought 300 shares of Acme Corp. in 2006.')
```

```
tagged_words = pos_tag(words)

print(tagged_words)
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
[('Jim', 'NNP'), ('bought', 'VBD'), ('300', 'CD'), ('shares', 'NNS'), ('of', 'IN'), ('Ac
```

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp('Jim bought 300 shares of Acme Corp. in 2006.')

for token in doc:
    print(token.text, token.pos_, token.tag_)
```

```
Jim PROPN NNP
bought VERB VBD
300 NUM CD
shares NOUN NNS
of ADP IN
Acme PROPN NNP
Corp. PROPN NNP
in ADP IN
2006 NUM CD
. PUNCT .
```

Named-entity Recognition

Named-entity recognition is a subtask of information extraction that seeks to locate and classify elements in text into pre-defined categories such as the names of **persons**, **organizations**, **locations**, **expressions of times**, **quantities**, **monetary values**, **percentages**, etc.

NE Type and Examples:-

- **ORGANIZATION** - Georgia-Pacific Corp., WHO
- **PERSON** - Eddy Bonte, President Obama
- **LOCATION** - Murray River, Mount Everest
- **DATE**- June, 2008-06-29
- **TIME** - two fifty a m, 1:30 p.m.
- **MONEY** - 175 million Canadian Dollars, GBP 10.40
- **PERCENT** - twenty pct, 18.75 %
- **FACILITY** - Washington Monument, Stonehenge
- **GPE** - South East Asia, Midlothian

```
from nltk import pos_tag, ne_chunk
from nltk.tokenize import wordpunct_tokenize

nltk.download('maxent_ne_chunker')
nltk.download('words')

sent = 'Jim bought 300 shares of Acme Corp. in 2006.'

print(ne_chunk(pos_tag(wordpunct_tokenize(sent))))
```

```
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping chunkers/maxent_ne_chunker.zip.
[nltk_data] Downloading package words to /root/nltk_data...
[nltk_data]   Unzipping corpora/words.zip.
(S
  (PERSON Jim/NNP)
  bought/VBD
  300/CD
  shares/NNS
  of/IN
  (ORGANIZATION Acme/NNP Corp/NNP)
  ./.
  in/IN
  2006/CD
  ./.)
```

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Jim bought 300 shares of Acme Corp. in 2006.")

for ent in doc.ents:
    print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

```
Jim 0 3 PERSON
300 11 14 CARDINAL
Acme Corp. 25 35 ORG
2006 39 43 DATE
```

Word Embedding

All texts need to be converted to numbers before starts processing by the machine. Specifically, vectors of numbers.

Text is messy in nature and machine learning algorithms prefer well defined fixed-length inputs and outputs.

Word Embedding is one such technique where we can represent the text using vectors. Before deep learning era, the popular forms of word embeddings were:

- **BoW**, which stands for Bag of Words
- **TF-IDF**, which stands for Term Frequency-Inverse Document Frequency

Bag-of-Words (BoW)

The **Bag-of-Words (BoW)** model is a way of representing text data when modeling text with machine learning algorithms. The **Bag-of-Words (BoW)** model is popular, simple to understand, and has seen great success in **language modeling** and **document classification**.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Example (BoW)

Consider the following 4 sentences:-

- It was the best of times.
- it was the worst of Times.
- it is the time of stupidity.
- it is the age of foolishness.

Form this above example, let’s consider each line as a separate **“document”** and the 4 lines as our entire corpus of documents.

Vocabulary

What would be the total vocabulary???

Bag of Words (BoW) Model

1. Design the Vocabulary

The unique words by ignoring case, punctuations, and making them into root words are:

1. it
2. was
3. the
4. best
5. of
6. time
7. worst
8. stupidity
9. is
10. age
11. foolishness

Vocabulary contains 11 words while the full corpus contains 24 words.

2. Create Document Vectors

The objective is to turn each document of text into a vector so that we can use as input or output for a machine learning model.

Because we know the vocabulary has 11 words, we can use a fixed-length document representation of 11, with one position in the vector to score each word. The simplest scoring method is to mark the presence of words as a boolean value, 0 for absent, non-zero (positive value) for present. There can be other methods such as count based methods of the terms if more than one occurance of a trem.

In this example the binary vector of four documents would look as follows:

	it	was	the	best	of	time	worst	stupidity	is	age	foolishness
Document #1 [It was the best of times.]	1	1	1	1	1	1	0	0	0	0	0
Document #2 [it was the worst of Times.]	1	1	1	0	1	1	1	0	0	0	0
Document #3 [it is the time of stupidity.]	1	0	1	0	1	1	0	1	1	0	0
Document #4 [it is the age of foolishness.]	1	0	1	0	1	0	0	0	1	1	1

Problems

- Ordering of words have been discarded which **ignores the context**. These unordered words **can't preserve document semantics** For instance, **“this is interesting”** vs **“is this interesting”**. Moreover, **"stupidity"** and **"foolishness"** are considered two different words in the dictionary.
- We are retaining no information on the **grammar of the sentences**.
- New documents that overlap with the vocabulary of known words, but may contain **words outside of the vocabulary**.
- If the vocabulary size increases the **document representation dimension** also increases.

Managing Vocabulary

In the previous example, the **length of the document vector** is equal to the number of known words which is 11 words.

For a very large corpus, such as thousands of books, the length of the vector **might be thousands or millions of positions**. Further, each document may contain **very few of the known words in the vocabulary**. This results in a vector with lots of zero scores, called a sparse vector or sparse representation. Sparse vectors require more memory and computational resources **(space and time complexity)**

It's very important to decrease the size of the vocabulary when using a bag-of-words model.

Solution #1

There are simple text cleaning techniques that can be used as a first step, such as:

- Ignoring case
- Ignoring punctuation
- Ignoring frequent words that don’t contain much information, called stop words, like “a,” “of,” etc.
- Fixing misspelled words.
- Reducing words to their stem (e.g. “play” from “playing”) using stemming algorithms.

Solution #2

Each word or token is called a “gram”. Creating a vocabulary of two-word pairs is, in turn, called a **bigram model**.

An **N-gram** is an N-token sequence of words: a 2-gram (more commonly called a bigram) is a two-word sequence of words like “please turn”, “turn your”, or “your homework”, and a **3-gram (more commonly called a trigram)** is a three-word sequence of words like “please turn your”, or “turn your homework”.

For example, the bigrams in the first line of text in the previous section: **“It was the best of times”** are as follows:

- “it was”
- “was the”
- “the best”
- “best of”
- “of times”

A vocabulary then tracks triplets of words is called a trigram model and the general approach is called the **n-gram model**, where n refers to the number of grouped words.

Note: Often a simple bigram approach is better than a 1-gram bag-of-words model.

One-Hot Representation

The one hot representation, as the name suggests, starts with a zero vector, and sets as 1 the corresponding entry in the vector if the word is present in the sentence or document.

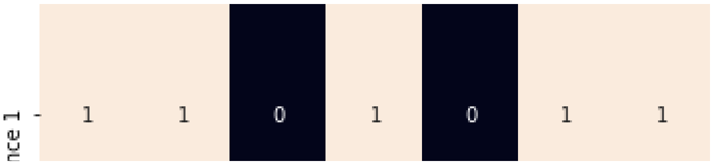
Tokenizing the sentences, ignoring punctuation, and treating everything as lowercase, will yield a vocabulary of size 8: `{time, fruit, flies, like, a, an, arrow, banana}`.

The binary encoding for **“like a banana”** would then be:

```
[0, 0, 0, 1, 1, 0, 0, 1]
```

```
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns

corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']

one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()

print (one_hot)

print (one_hot_vectorizer.vocabulary_)

dictionary = sorted(one_hot_vectorizer.vocabulary_)

print(dictionary)

sns.heatmap(one_hot, annot=True, cbar=False, xticklabels=dictionary,
                                             yticklabels=['Sentence 1','Sentence 2'])
```

```
[[1 1 0 1 0 1 1]
 [0 0 1 1 1 1 0]]
{'time': 6, 'flies': 3, 'like': 5, 'an': 0, 'arrow': 1, 'fruit': 4, 'banana': 2}
['an', 'arrow', 'banana', 'flies', 'fruit', 'like', 'time']
<matplotlib.axes._subplots.AxesSubplot at 0x7f4d71f2b5c0>
```



Term Frequency (TF)

Term Frequent (**TF**) is a measure of how frequently a term, t, appears in a document, d:

$$TF_{t,d} = \frac{n_{t,d}}{\text{Total number of terms in document } d}$$

$n_{t,d}$ = Number of times term t appears in a document d. Thus, each document and term would have its own **TF** value.

Consider these 3 documents like **BoW** model:-

- It was the best of the time.
- it was the worst of Times.
- it is the time of stupidity.

The vocabulary or dictionary of the entire corpus would be:-

1. it
2. was
3. the
4. best
5. of
6. time
7. worst
8. is
9. stupidity

Now we will calculate the **TF** values for the **Document 3**.

Document 3 :- **it is the time of stupidity.**

- Number of words in Document 3 = **6**

- TF for the word **‘the’** = (number of times **‘the’** appears in Document 3) / (number of terms in Document 3) = **1/6**

Likewise:-

- TF(**'it'**) = 1/6
- TF(**'was'**) = 0/6 = 0
- TF(**'the'**) = 1/6
- TF(**'best'**) = 0/6 = 0
- TF(**'of'**) = 1/6
- TF(**'time'**) = 1/6
- TF(**'worst'**) = 0/6 = 0
- TF(**'is'**) = 1/6
- TF(**'stupidity'**) = 1/6

We can calculate all the term frequencies for all the terms of all the documents in this manner:-

Term	Document#1	Document#2	Document#3	TF (Document#1)	TF (Document#2)	TF (Document#3)
it	1	1	1	1/7	1/6	1/6
was	1	1	0	1/7	1/6	0
the	2	1	1	2/7	1/6	1/6
best	1	0	0	1/7	0	0
of	1	1	1	1/7	1/6	1/6
time	1	1	1	1/7	1/6	1/6
worst	0	1	0	0	1/6	0
is	0	0	1	0	0	1/6

```
import math

print(math.log((3),10))

print(math.log((330),10))

print(math.log((3/3),10))

print(math.log((4/3),10))

print(math.log((4/5),10))
```

```
    0.47712125471966244
    2.518513939877887
    0.0
    0.1249387366082999
    -0.09691001300805638
```

Inverse Document Frequency (IDF)

IDF is a measure of how important a term is. We need the IDF value because computing just the **TF alone is not sufficient** to understand the importance of words:

$$IDF_t = log\,(\frac{\text{Total Number of Documents}}{\text{The Number of Documents with Term }t})$$

A problem with scoring word frequency is that highly frequent words **(‘is’, ‘the’, ‘a’ etc)** start to dominate in the document (e.g. larger score), but may not contain as much **“useful information”** to the model comapre to the rarer but **domain specific words**.

One approach is to rescale the frequency of words by **how often they appear in all documents**, so that the scores for frequent words like “the” that are also frequent **across all documents are penalized**.

This approach to scoring is called Term Frequency – Inverse Document Frequency, or TF-IDF for short, where:

- **Term Frequency:** is a scoring of the frequency of the word in the current document.
- **Inverse Document Frequency:** is a scoring of how rare the word is across documents.

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

We can calculate the IDF values for **Document 3**:

Document 3 :- **it is the time of stupidity.**

IDF(‘it’) = log(total number of documents/number of documents containing the word ‘it’) = log(3/3) = log(1) = 0

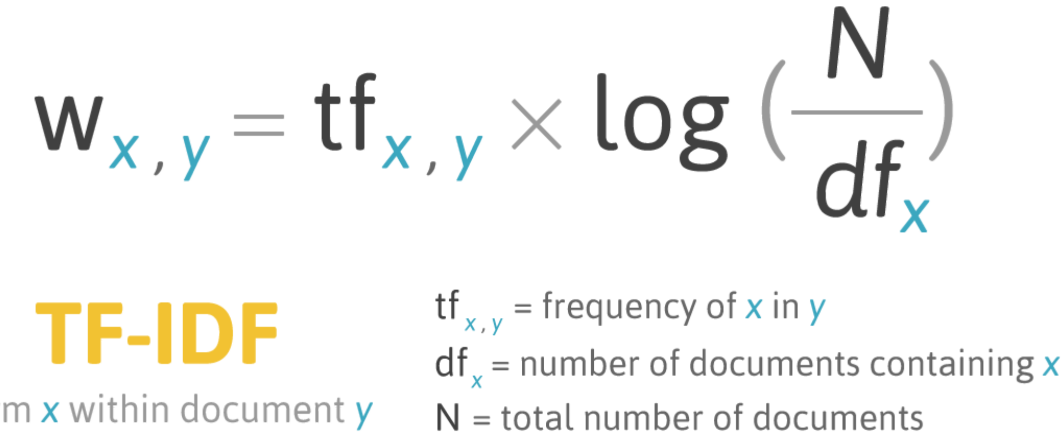
We can calculate the IDF values for each word like this. Thus, the IDF values for the entire vocabulary would be:

Term	Document#1	Document#2	Document#3	IDF
it	1	1	1	0.00
was	1	1	0	0.18
the	2	1	1	0.00
best	1	0	0	0.48
of	1	1	1	0.00
time	1	1	1	0.00
worst	0	1	0	0.48
is	0	0	1	0.48
stupidity	0	0	1	0.48

We can now compute the TF-IDF score for each word in the corpus. Words with a higher score are more important, and those with a lower score are less important:

$$(TF-IDF)_{t,d} = TF_{t,d} * IDF_t$$

You can find the overall summary in the following figure.


$w_{x,y} = tf_{x,y} \times \log(\frac{N}{df_x})$
TF-IDF
Term x within document y
$tf_{x,y}$ = frequency of x in y
df_x = number of documents containing x
N = total number of documents

We can now calculate the TF-IDF score for every word in **Document 3**:

Document 3 :- **it is the time of stupidity.**

TF-IDF('it', Document 3) = TF('it', Document 3) * IDF('it') = 1/6 * 0 = 0

Likewise:-

- TF(**'it'**) = (1/6) * 0 = 0
- TF(**'is'**) = (1/6) * 0.48
- TF(**'the'**) = (1/6) * 0 = 0
- TF(**'best'**) = (0/6) * 0.48 = 0
- TF(**'time'**) = (1/6) * 0 = 0
- TF(**'of'**) = (1/6) * 0 = 0
- TF(**'stupidity'**) = (1/6) * 0.48

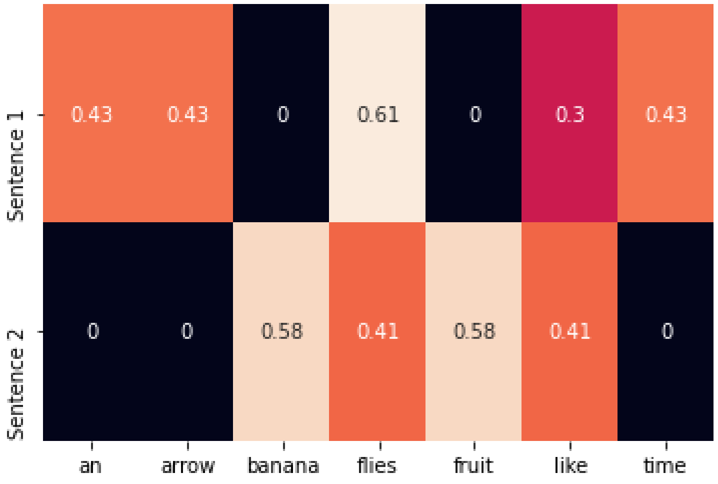
Similarly, we can calculate the TF-IDF scores for all the words with respect to all the documents.

- First, notice how if there is a very common word that occurs in all documents (i.e., n = N), IDF(w) is 0 and the TFIDF score is 0, thereby completely penalizing that term.
- Second, if a term occurs very rarely, perhaps in only one document, the IDF will be the maximum possible value, log N

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns

corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']
```

```
tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()

print (tfidf)

print (tfidf_vectorizer.vocabulary_)

dictionary = sorted(tfidf_vectorizer.vocabulary_)

print(dictionary)

sns.heatmap(tfidf, annot=True, cbar=False, xticklabels=dictionary,
                                           yticklabels=['Sentence 1','Sentence 2'])
```

```
[[0.42519636 0.42519636 0.         0.60506143 0.         0.30253071
  0.42519636]
 [0.         0.         0.57615236 0.40993715 0.57615236 0.40993715
  0.        ]]
{'time': 6, 'flies': 3, 'like': 5, 'an': 0, 'arrow': 1, 'fruit': 4, 'banana': 2}
['an', 'arrow', 'banana', 'flies', 'fruit', 'like', 'time']
<matplotlib.axes._subplots.AxesSubplot at 0x7f4d71e8c198>
```



Summary

Bag of Words just creates a set of vectors containing the count of word occurrences in the document, while the TF-IDF model contains information on the more important words and the less important ones as well.

Bag of Words vectors are easy to interpret. However, TF-IDF usually performs better in machine learning models.

Understanding the context of words is important. Detecting the similarity between the words ‘time’ and ‘age’, or 'stupidity' and 'foolishness'.

This is where Word Embedding techniques such as **Word2Vec, Continuous Bag of Words (CBOW), Skipgram**, etc come into play.

Bag-of-Words Text Classification

We will show how to build a simple Bag of Words (BoW) text classifier using PyTorch. The classifier is trained on IMDB movie reviews dataset.

```
from pathlib import Path

import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from google_drive_downloader import GoogleDriveDownloader as gdd
from torch.utils.data import DataLoader, Dataset
from sklearn.feature_extraction.text import CountVectorizer
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
     device(type='cpu')
```

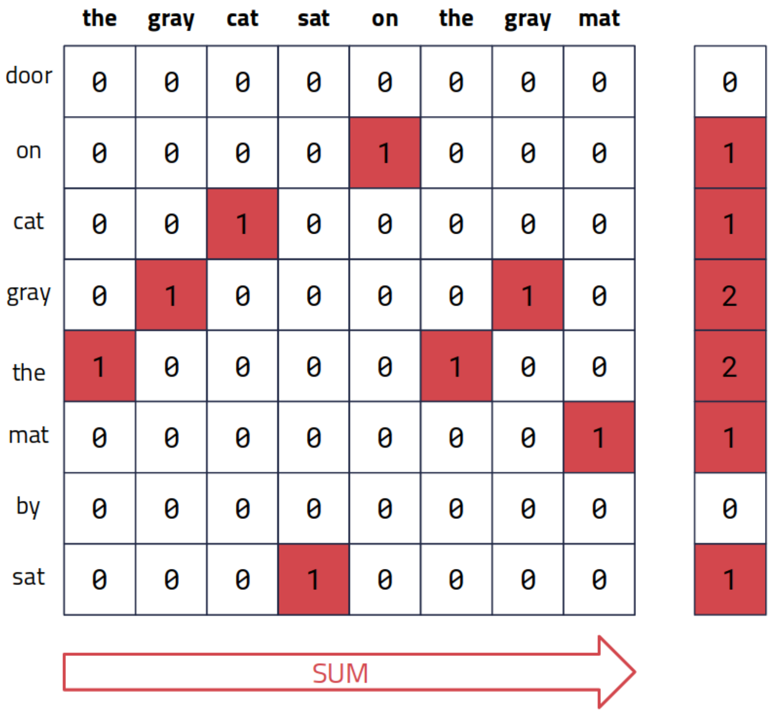
```
DATA_PATH = 'datasets/imdb.csv'
if not Path(DATA_PATH).is_file():
    gdd.download_file_from_google_drive(
        file_id='1nEqdpGkMhAPDWZn4oij5o1Irf4T8PxvF',
        dest_path=DATA_PATH,
    )
```

```
     Downloading 1nEqdpGkMhAPDWZn4oij5o1Irf4T8PxvF into datasets/imdb.csv... Done.
```

```
# View some example records
pd.read_csv(DATA_PATH).sample(5)
```

	A very, very, very slow-moving, aimless movie about a distressed, drifting young man.	0
542	However, after seeing the short again after ab...	1
80	This if the first movie I've given a 10 to in ...	1
535	the movie is littered with overt racial slurs ...	0
937	This movie is great--especially if you enjoy v...	1
503	Cinematography noteworthy including fine views...	1

Bag-of-Words Sentiment Classification



	the	gray	cat	sat	on	the	gray	mat	
door	0	0	0	0	0	0	0	0	0
on	0	0	0	0	1	0	0	0	1
cat	0	0	1	0	0	0	0	0	1
gray	0	1	0	0	0	0	1	0	2
the	1	0	0	0	0	1	0	0	2
mat	0	0	0	0	0	0	0	1	1
by	0	0	0	0	0	0	0	0	0
sat	0	0	0	1	0	0	0	0	1

So the final bag-of-words vector for `['the', 'gray', 'cat', 'sat', 'on', 'the', 'gray', 'mat']` is `[0, 1, 1, 2, 2, 1, 0, 1]`

```
class Sequences(Dataset):
    def __init__(self, data):
        self.vectorizer = CountVectorizer(stop_words='english')
        self.sequences = self.vectorizer.fit_transform(data.review.tolist())
        self.labels = data.label.tolist()
        self.token2idx = self.vectorizer.vocabulary_
        self.idx2token = {idx: token for token, idx in self.token2idx.items()}

    def __getitem__(self, i):
        return self.sequences[i, :].toarray(), self.labels[i]

    def __len__(self):
        return self.sequences.shape[0]


df = pd.read_csv(DATA_PATH, header=None)

print(len(df))

df.columns = ["review", "label"]

df_train = df.head(900)
df_test = df.tail(100)

dataset = Sequences(df_train)

train_loader = DataLoader(dataset, batch_size=900)
```

```
1000
```

```
class BagOfWordsClassifier(nn.Module):
    def __init__(self, vocab_size, hidden1, hidden2):
        super().__init__()
        ### 1st hidden layer: vocab_size --> 128
        self.linear_1 = nn.Linear(vocab_size, hidden1)
        ### Non-linearity in 1st hidden layer
        self.relu_1 = nn.ReLU()

        ### 2nd hidden layer: 128 --> 64
        self.linear_2 = nn.Linear(hidden1, hidden2)
        ### Non-linearity in 2nd hidden layer
        self.relu_2 = nn.ReLU()

        ### Output layer: 64 --> 1
        self.linear_out = nn.Linear(hidden2, 1)

    def forward(self, inputs):
        ### 1st hidden layer
        out = self.linear_1(inputs.squeeze(1).float())
        ### Non-linearity in 1st hidden layer
        out = self.relu_1(out)

        ### 2nd hidden layer
        out = self.linear_2(out)
        ### Non-linearity in 2nd hidden layer
        out = self.relu_2(out)

        # Linear layer (output)
        logits  = self.linear_out(out)

        return logits
```

```
model = BagOfWordsClassifier(len(dataset.token2idx), 128, 64)
model
```

```
BagOfWordsClassifier(
  (linear_1): Linear(in_features=2624, out_features=128, bias=True)
  (relu_1): ReLU()
  (linear_2): Linear(in_features=128, out_features=64, bias=True)
  (relu_2): ReLU()
  (linear_out): Linear(in_features=64, out_features=1, bias=True)
)
```

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

```
train_losses = []

for epoch in range(150):
    losses = []
    total = 0
    for inputs, target in train_loader:
        model.zero_grad()

        output = model(inputs)
        loss = criterion(output.squeeze(), target.float())

        loss.backward()

        optimizer.step()

        losses.append(loss.item())
        total += 1

    epoch_loss = sum(losses) / total
    train_losses.append(epoch_loss)

    print(f'Epoch #{epoch + 1}\tTrain Loss: {epoch_loss:.3f}')
```

```
Epoch #1        Train Loss: 0.693
Epoch #2        Train Loss: 0.690
Epoch #3        Train Loss: 0.687
Epoch #4        Train Loss: 0.684
Epoch #5        Train Loss: 0.680
Epoch #6        Train Loss: 0.676
Epoch #7        Train Loss: 0.671
Epoch #8        Train Loss: 0.666
Epoch #9        Train Loss: 0.660
Epoch #10       Train Loss: 0.653
Epoch #11       Train Loss: 0.645
Epoch #12       Train Loss: 0.637
Epoch #13       Train Loss: 0.627
Epoch #14       Train Loss: 0.617
Epoch #15       Train Loss: 0.606
Epoch #16       Train Loss: 0.594
Epoch #17       Train Loss: 0.580
Epoch #18       Train Loss: 0.566
Epoch #19       Train Loss: 0.551
Epoch #20       Train Loss: 0.535
Epoch #21       Train Loss: 0.518
Epoch #22       Train Loss: 0.500
Epoch #23       Train Loss: 0.481
Epoch #24       Train Loss: 0.462
Epoch #25       Train Loss: 0.442
Epoch #26       Train Loss: 0.422
Epoch #27       Train Loss: 0.402
Epoch #28       Train Loss: 0.381
Epoch #29       Train Loss: 0.361
Epoch #30       Train Loss: 0.340
Epoch #31       Train Loss: 0.320
Epoch #32       Train Loss: 0.300
Epoch #33       Train Loss: 0.281
Epoch #34       Train Loss: 0.262
Epoch #35       Train Loss: 0.245
Epoch #36       Train Loss: 0.228
Epoch #37       Train Loss: 0.211
Epoch #38       Train Loss: 0.196
Epoch #39       Train Loss: 0.182
Epoch #40       Train Loss: 0.169
Epoch #41       Train Loss: 0.156
Epoch #42       Train Loss: 0.145
Epoch #43       Train Loss: 0.134
Epoch #44       Train Loss: 0.125
Epoch #45       Train Loss: 0.116
Epoch #46       Train Loss: 0.107
Epoch #47       Train Loss: 0.100
Epoch #48       Train Loss: 0.093
Epoch #49       Train Loss: 0.087
Epoch #50       Train Loss: 0.081
Epoch #51       Train Loss: 0.076
Epoch #52       Train Loss: 0.071
Epoch #53       Train Loss: 0.067
Epoch #54       Train Loss: 0.063
Epoch #55       Train Loss: 0.059
Epoch #56       Train Loss: 0.056
Epoch #57       Train Loss: 0.053
Epoch #58       Train Loss: 0.050
Epoch #59       Train Loss: 0.047
```

```
def predict_sentiment(text):
  test_vector = torch.LongTensor(dataset.vectorizer.transform([text]).toarray())

  output = model(test_vector)

  prediction = torch.sigmoid(output).item()

  if prediction > 0.5:
    print(f'{prediction:0.3}: Positive sentiment')
    return 1
  else:
    print(f'{prediction:0.3}: Negative sentiment')
    return 0
```

```
test_text = "The story itself is just predictable and lazy."
predict_sentiment(test_text)
```

```
0.00139: Negative sentiment
0
```

```
test_text = "Excellent cast, story line, performances."
predict_sentiment(test_text)
```

```
1.0: Positive sentiment
```

```
1
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix


pred_labels = []

sentences = list(df_test['review'])
labels = df_test['label']

print(sentences)

for sentence in sentences:
  pred_labels.append(predict_sentiment(sentence))

# accuracy: (tp + tn) / (p + n)
accuracy = accuracy_score(labels, pred_labels)
print('Accuracy: %f' % accuracy)

# precision tp / (tp + fp)
precision = precision_score(labels, pred_labels)
print('Precision: %f' % precision)

# recall: tp / (tp + fn)
recall = recall_score(labels, pred_labels)
print('Recall: %f' % recall)

# f1: 2 tp / (2 tp + fp + fn)
f1 = f1_score(labels, pred_labels)
print('F1 score: %f' % f1)

# confusion matrix
matrix = confusion_matrix(labels, pred_labels)
print(matrix)
```

```
["Otherwise, don't even waste your time on this.  ", 'This one just fails to create a
0.00244: Negative sentiment
0.0071: Negative sentiment
0.000537: Negative sentiment
0.67: Positive sentiment
0.18: Negative sentiment
3.82e-05: Negative sentiment
0.000176: Negative sentiment
0.18: Negative sentiment
1.9e-05: Negative sentiment
0.293: Negative sentiment
0.941: Positive sentiment
```

```
0.817: Positive sentiment
0.134: Negative sentiment
0.00282: Negative sentiment
1.0: Positive sentiment
0.238: Negative sentiment
0.943: Positive sentiment
0.102: Negative sentiment
0.0429: Negative sentiment
0.0232: Negative sentiment
0.996: Positive sentiment
0.637: Positive sentiment
0.101: Negative sentiment
1.96e-05: Negative sentiment
0.00742: Negative sentiment
0.99: Positive sentiment
0.00192: Negative sentiment
0.142: Negative sentiment
0.972: Positive sentiment
0.748: Positive sentiment
0.964: Positive sentiment
0.999: Positive sentiment
0.999: Positive sentiment
1.0: Positive sentiment
0.624: Positive sentiment
0.0691: Negative sentiment
1.0: Positive sentiment
0.996: Positive sentiment
0.884: Positive sentiment
0.993: Positive sentiment
0.0985: Negative sentiment
1.0: Positive sentiment
1.0: Positive sentiment
0.273: Negative sentiment
0.982: Positive sentiment
0.977: Positive sentiment
0.972: Positive sentiment
1.0: Positive sentiment
0.995: Positive sentiment
0.99: Positive sentiment
0.999: Positive sentiment
0.998: Positive sentiment
0.992: Positive sentiment
0.000587: Negative sentiment
4.56e-07: Negative sentiment
0.000502: Negative sentiment
0.994: Positive sentiment
```

Job Related Topics - Part I [Optional]

- Create a professional email address
  - First name + last name = firstlast@domain.com
  - First name . last name = first.last@domain.com
  - First name - last name = first-last@domain.com

  - First name . middle name . last name = first.middle.last@domain.com
  - First name - middle name - last name = first-middle-last@domain.com
  - First initial + last name = flast@domain.com
  - First initial + middle name + last name = fmiddlelast@domain.com
  - First initial + middle initial + last name = fmlast@domain.com

- The shorter your email the better
- Complete your Linkedin profile
- Prepare a CV in Latex
- Seperate your contact number [personal vs professional]
- Create GitHub profile [Username may only contain alphanumeric characters or single hyphens, and cannot begin or end with a hyphen.]
- Build your website using GitHub pages [would see a demo in the next class]

Cosine Similarity & Euclidean Distance

Consider the following 4 sentences:-

- It was the best of times.
- it was the worst of Times.
- it is the time of stupidity.
- it is the age of foolishness.

```
# Define the documents
doc1 = "It was the best of times."

doc2 = "it was the worst of Times."

doc3  = "it is the time of stupidity."

doc4  = "it is the age of foolishness."

documents = [doc1, doc2, doc3, doc4]
```

```
# Scikit Learn using Bag of Words
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

# Create the Document Term Matrix
bag_of_words_vectorizer = CountVectorizer()
bag_of_words = bag_of_words_vectorizer.fit_transform(documents).toarray()

# Convert Sparse Matrix to Pandas Dataframe if you want to see the word frequencies.
df = pd.DataFrame(bag_of_words,
                  columns=bag_of_words_vectorizer.get_feature_names(),
                  index=['doc1', 'doc2', 'doc3', 'doc4'])

print(bag_of_words)
display(df)
```

```
[[0 1 0 0 1 1 0 1 0 1 1 0]
 [0 0 0 0 1 1 0 1 0 1 1 1]
```

```
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances

# Compute Cosine Similarity
print("Cosine Similarity")
print(cosine_similarity(df, df))

# Compute Euclidean Distance
print("Euclidean Distance")
print(euclidean_distances(df, df))
```

```
Cosine Similarity
[[1.         0.83333333 0.5        0.5       ]
 [0.83333333 1.         0.5        0.5       ]
 [0.5        0.5        1.         0.66666667]
 [0.5        0.5        0.66666667 1.        ]]
Euclidean Distance
[[0.         1.41421356 2.44948974 2.44948974]
 [1.41421356 0.         2.44948974 2.44948974]
 [2.44948974 2.44948974 0.         2.        ]
 [2.44948974 2.44948974 2.         0.        ]]
```

Improvement using Stopword Filtering and Lemmatization

```
import spacy

# Small spaCy model
nlp = spacy.load("en_core_web_sm")

lemma_documents = []

# Lemmatize docs (individual)
def lemmatize_docs(x):
  x = x.lower()
  doc = nlp(x)
  lemma_words = [w.lemma_ if w.lemma_ !='-PRON-' else w.text for w in doc]
  return " ".join(lemma_words)

for doc in documents:
  lemma_documents.append(lemmatize_docs(doc))

print(lemma_documents)
```

```
['it be the good of time .', 'it be the bad of time .', 'it be the time of stupidity .',
```

```
# Scikit Learn using Bag of Words
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
import pandas as pd

# Create the Document Term Matrix
bag_of_words_vectorizer = CountVectorizer(stop_words='english')
bag_of_words = bag_of_words_vectorizer.fit_transform(lemma_documents).toarray()

# Convert Sparse Matrix to Pandas Dataframe if you want to see the word frequencies.
df = pd.DataFrame(bag_of_words,
                  columns=bag_of_words_vectorizer.get_feature_names(),
                  index=['doc1', 'doc2', 'doc3', 'doc4'])

print(bag_of_words)
display(df)
```

```
[[0 0 0 1 0 1]
 [0 1 0 0 0 1]
 [0 0 0 0 1 1]
 [1 0 1 0 0 0]]
```

	age	bad	foolishness	good	stupidity	time
doc1	0	0	0	1	0	1
doc2	0	1	0	0	0	1
doc3	0	0	0	0	1	1
doc4	1	0	1	0	0	0

```
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances

# Compute Cosine Similarity
print("Cosine Similarity")
print(cosine_similarity(df, df))

# Compute Euclidean Distance
print("Euclidean Distance")
print(euclidean_distances(df, df))
```

```
Cosine Similarity
[[1.  0.5 0.5 0. ]
 [0.5 1.  0.5 0. ]
 [0.5 0.5 1.  0. ]
 [0.  0.  0.  1. ]]
Euclidean Distance
[[0.         1.41421356 1.41421356 2.        ]
 [1.41421356 0.         1.41421356 2.        ]
 [1.41421356 1.41421356 0.         2.        ]
 [2.         2.         2.         0.        ]]
```