Ahsanullah University of Science and Technology (AUST)

Department of Computer Science and Engineering

LABORATORY MANUAL

Course No.: CSE4238

Course Title: Soft Computing Lab

For the students of 4th Year, 2nd Semester of
B.Sc. in Computer Science and Engineering program

TABLE OF CONTENTS

COURSE OUTCOMES

1. Demonstrate a comprehensive understanding of the fundamental concepts and properties of soft computing methodologies such as fuzzy sets and logic, artificial neural networks, probabilistic reasoning, and genetic algorithms.
2. Generate fundamental concepts used in Soft computing. The concepts of Soft Computing, major technology trends driving Deep Learning and optimization techniques using Genetic Algorithm (GA)
3. Develop fully connected deep neural networks, efficient (vectorized) neural networks with key parameters of a neural networks architecture
4. Implement a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence. Be able to implement a neural network in PyTorch
5. Present the applications of neural networks and fuzzy sets in information processing, decision making, and control systems in a clear and concise manner.

PREFFERED TOOL(S)

Jupyter Nootbook / Google Colaboratory

REFERENCES

BOOKS

1. Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence (1st Edition)
   Authored by: Jyh-Shing Roger Jang, Chuen-Tsai Sun and Eiji Mizutani Publisher: Pearson

2. Dive into Deep Learning
   Authored by: Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

3. Fuzzy Logic with Engineering Applications, Third Edition
   Authored by: Timothy J. Ross
   Publisher: Wiley

ONLINE RESOURCES

1. https://www.deeplearningbook.org/

ADMINISTRATIVE POLICY OF THE LABORATORY

- Students must perform class assessment tasks individually without help of others.
- Viva for each program will be taken and considered as a performance.
- Plagiarism is strictly forbidden and will be dealt with punishment

Session 01

Goals:

1. To know about NumPy Library
2. To Know about Pytorch Framework

Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. To use Numpy, we first need to import the numpy package as `import numpy as np`

```
import numpy as np
```

```
a=np.array([[1,3],[2,4]])
print(np.min(a,1))
```

```
    [1 2]
```

Arrays

A numpy array is a grid of values, **all of the same type**, and is indexed by a tuple of nonnegative integers. **The number of dimensions is the rank of the array**; the shape of an array is a tuple of integers giving the size of the array along each dimension. We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
    <class 'numpy.ndarray'> (3,) 1 2 3
    [5 2 3]
```

```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b)
```

```
    [[1 2 3]
     [4 5 6]]
```

```
b[0][0]
```

```
    1
```

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
    (2, 3)
    1 2 4
```

Numpy also provides many functions to create arrays:

```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)

     [[0. 0.]
      [0. 0.]]
```

```
b = np.ones((1,2))   # Create an array of all ones
print(b)

     [[1. 1.]]
```

```
c = np.full((2,2), 7) # Create a constant array
print(c)

     [[7 7]
      [7 7]]
```

```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)

     [[1. 0.]
      [0. 1.]]
```

```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)

     [[0.71565102 0.14143954]
      [0.68749501 0.07470517]]
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. **Numpy tries to guess a datatype when you create an array**, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

You can read all about numpy datatypes in the documentation.

```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)

     int64 float64 int64
```

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Important Note

`*` is elementwise multiplication, not matrix multiplication. **We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.** dot is available both as a function in the numpy module and as an instance method of array objects:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
     219
     219
```

You can also use the @ operator which is equivalent to numpy's dot operator.

```
print(v @ w)
```

```
     219
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
     [29 67]
     [29 67]
     [29 67]
```

```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

```
     [[19 22]
      [43 50]]
     [[19 22]
      [43 50]]
     [[19 22]
      [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
     10
     [4 6]
     [3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
print(x)
print("transpose\n", x.T)
```

```
    [[1 2]
     [3 4]]
    transpose
     [[1 3]
     [2 4]]
```

```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
    [[1 2 3]]
    transpose
     [[1]
     [2]
     [3]]
```

```
import numpy as np

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

```
    [1 2 3]
```

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]
```

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]
```

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

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__)
```

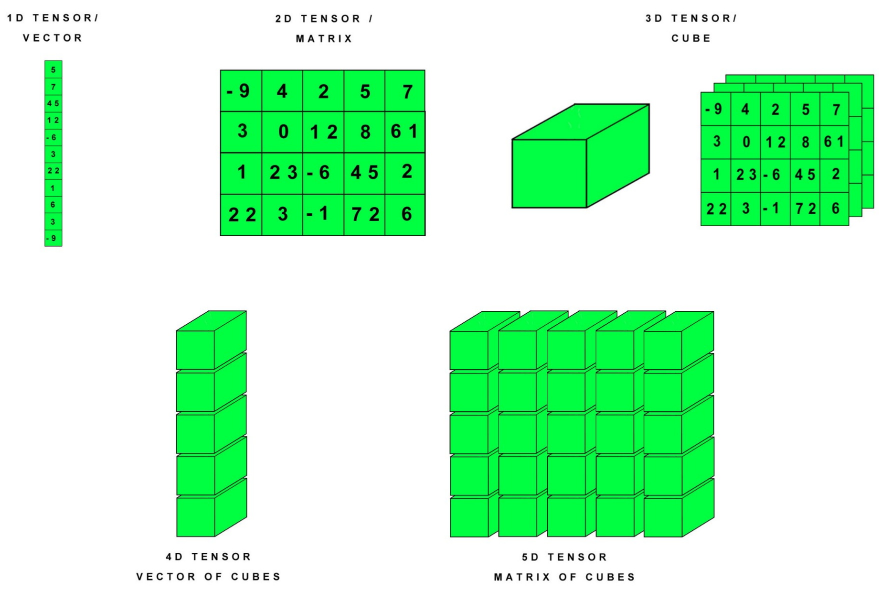
```
2.0.0+cu118
```

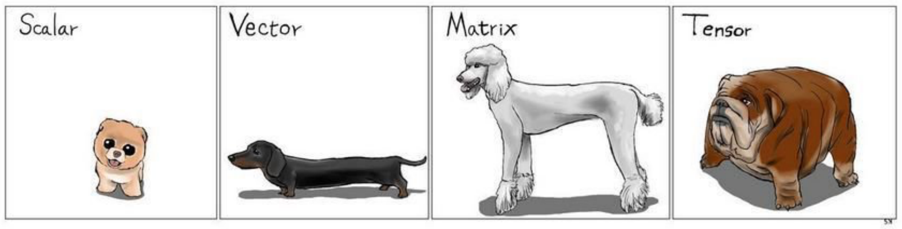
Introduction to Tensors

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([[0.7878, 0.7632, 0.5334],
             [0.3148, 0.8141, 0.5708],
             [0.8645, 0.3849, 0.7457],
             [0.5847, 0.7187, 0.6906],
             [0.1597, 0.6442, 0.0510]])


# 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]])
```

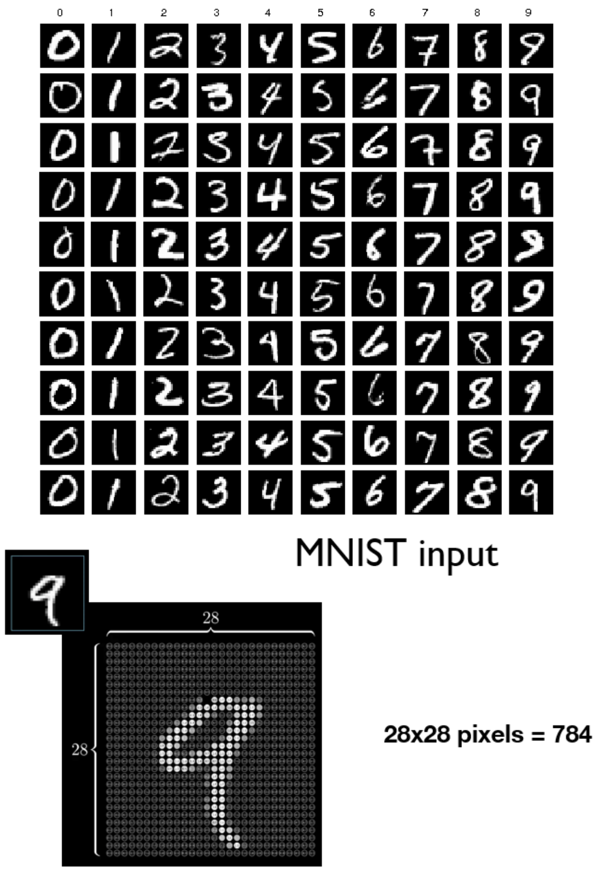
```
tensor_values = torch.from_numpy(np_values)

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 notation:
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())

     tensor([[1]])
     1
     2.5


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

     torch.Size([2, 3])


x.shape

     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([[0.9885, 0.6527, 0.4791],
             [0.5478, 0.9898, 0.3387],
             [0.7152, 0.6006, 0.5892],
             [0.2309, 0.0228, 0.5204],
             [0.7644, 0.6659, 0.4507]])
     tensor([[0.6583, 0.9113, 0.5257],
             [0.1855, 0.2396, 0.9312],
             [0.4075, 0.2966, 0.8326],
             [0.7614, 0.1222, 0.7190],
             [0.2609, 0.6813, 0.6104]])
     tensor([[1.6467, 1.5640, 1.0048],
             [0.7332, 1.2294, 1.2699],
             [1.1228, 0.8972, 1.4218],
             [0.9923, 0.1450, 1.2395],
             [1.0253, 1.3472, 1.0611]])
     tensor([[ 0.3302, -0.2585, -0.0466],
             [ 0.3623,  0.7502, -0.5925],
             [ 0.3077,  0.3040, -0.2434],
             [-0.5304, -0.0994, -0.1986],
             [ 0.5035, -0.0153, -0.1597]])


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

     tensor([[1.6467, 1.5640, 1.0048],
             [0.7332, 1.2294, 1.2699],
             [1.1228, 0.8972, 1.4218],
             [0.9923, 0.1450, 1.2395],
             [1.0253, 1.3472, 1.0611]])
     tensor([[ 0.3302, -0.2585, -0.0466],
             [ 0.3623,  0.7502, -0.5925],
             [ 0.3077,  0.3040, -0.2434],
             [-0.5304, -0.0994, -0.1986],
             [ 0.5035, -0.0153, -0.1597]])


# 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([[-0.8031,  0.2446,  0.7940],
             [-0.3707,  0.0465,  1.4219]])
     tensor([[ 0.3405, -0.5077,  0.0098],
             [ 2.4161,  0.2791, -1.2381],
             [ 0.3947,  0.1022, -0.7730]])
     tensor([[ 0.6309,  0.5572, -0.9245],
             [ 0.5475,  0.3465, -1.1604]])
     tensor([[-0.8031, -0.3707],
             [ 0.2446,  0.0465],
             [ 0.7940,  1.4219]])


# 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]],[[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([[7., 8., 9.],
             [7., 8., 9.]])
     tensor([[[1., 2., 3.],
              [4., 5., 6.],
              [7., 8., 9.]],

             [[1., 2., 3.],
              [4., 5., 6.],
              [7., 8., 9.]]])
     tensor([[[7., 8., 9.]]])


print(t[0,-2:-1, :1])

     tensor([[4.]])
```

Session 02

Goals:

1. To know about Neural Network
2. To Know about Different Variations of Neural Network

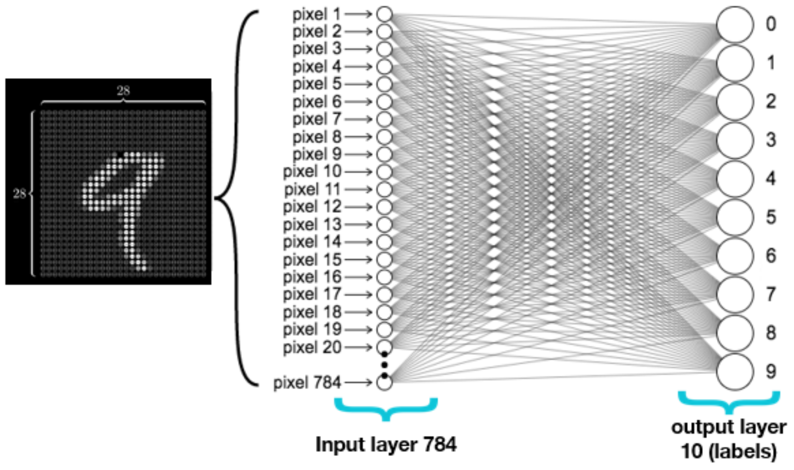
MNIST Digit Recognizer (Neural Network)



One Layer FNN with Sigmoid Activation

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

MNIST Network



- 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] from [0-255]
                            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 training datas

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

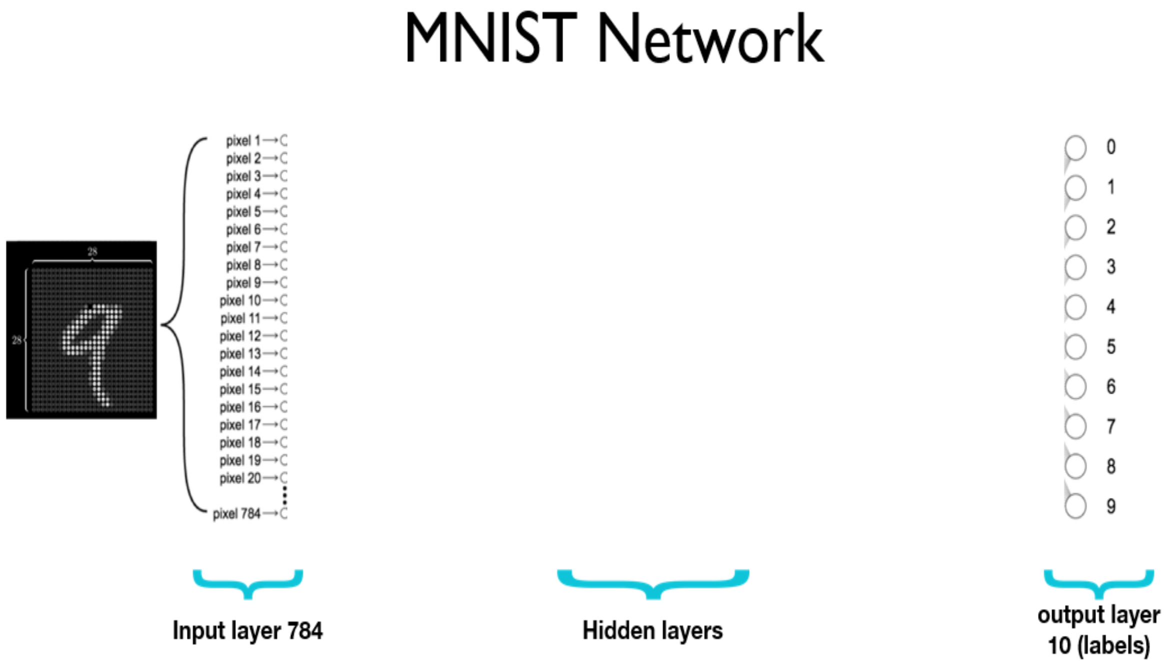
```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-imag
                                        9920512/? [00:20<00:00, 1332586.75it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labe
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/raw/t10k-images
                                        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/raw/t10k-labels
0%                                      0/4542 [00:00<?, ?it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
/pytorch/torch/csrc/utils/tensor_numpy.cpp:141: UserWarning: The given NumPy array is not writeable, a
```

```
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] from [0-255]
                            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 training datas

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()
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] from [0-255]
                            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 training datas

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] from [0-255]
                            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 training datas
```

```
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
        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
```

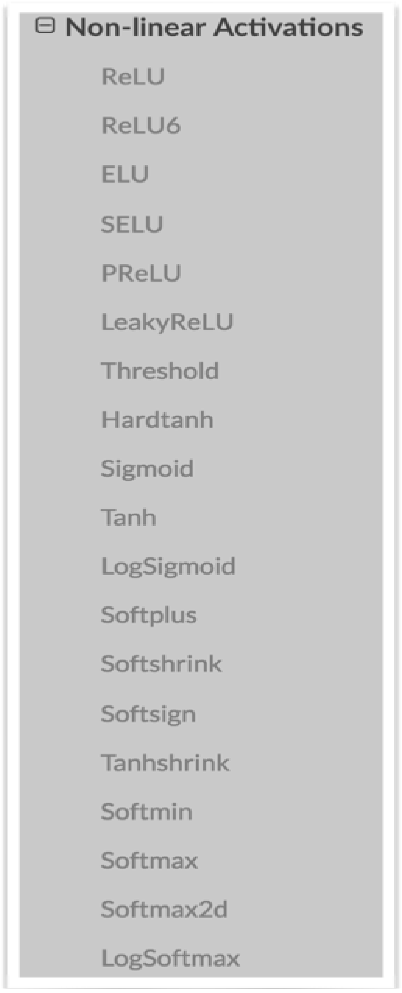
```
# Hyperparameters
batch_size = 100
num_iters = 3000
input_dim = 28*28 #num_features = 784
num_hidden = 100
output_dim = 10

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] from [0-255]
                            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 training datas

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

Session 03

Goals:

1. To know about Convolutional Neural Network

Convolutional Neural Network

Convolutional Neural Networks (CNNs) are widely used in computer vision tasks such as image classification, object detection, and image segmentation. Here are some ideas for CNN applications:

a. Image Classification:
   Create a CNN for classifying common objects in everyday life. Develop a CNN for medical image classification, such as detecting diseases from X-rays or MRI scans.

b. Object Detection:
   Build an object detection system for autonomous vehicles to identify pedestrians, vehicles, and road signs. Create a system that detects and tracks specific objects in a video stream, like tracking a soccer ball during a match.

c. Image Segmentation:
   Develop a CNN for semantic segmentation in satellite images to classify different land-use categories. Build a real-time video segmentation model that can separate objects from their background.

d. Face Recognition:
   Create a CNN-based face recognition system for security applications. Build a system that can estimate the age, gender, and emotions of individuals from their facial expressions.

e. Style Transfer:
   Implement neural style transfer using CNNs to transform ordinary photos into artistic styles of famous painters. Create a mobile app that allows users to apply various artistic styles to their photos in real-time.

f. Anomaly Detection:
   Develop a CNN to identify anomalies in manufacturing processes by analyzing sensor data. Create a system that can detect anomalous activities in video surveillance, such as break-ins or accidents.

g. Medical Imaging:
   Build a CNN for detecting and localizing tumors in medical images like mammograms or CT scans. Create a system for diagnosing skin conditions based on dermatological images.

h. Emotion Recognition:
   Develop a CNN-based system that can analyze facial expressions in real-time to recognize emotions. Create a sentiment analysis tool that uses CNNs to analyze emotions in text and multimedia content.

i. Video Analysis:
Build a CNN-based action recognition system that can identify human actions in videos, such as dancing or playing sports. Develop a video summarization tool that uses CNNs to extract key frames and scenes from long video footage.

j. Generative Models:
Implement a CNN-based generative adversarial network (GAN) for generating realistic images, like faces or landscapes. Create a CNN-based text-to-image generator that can turn textual descriptions into visual representations.

k. Autonomous Robots:
Integrate CNNs into a robot's vision system to enable it to navigate and interact with its environment. Build a robot that can sort and categorize objects based on visual cues using a CNN.

When working on these ideas, be sure to gather and preprocess the relevant datasets, fine-tune your network architecture, and carefully train and evaluate your models to achieve the best results. Also, consider the ethical implications and potential biases in your data and models, especially in applications like face recognition and sentiment analysis.

```
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


'''
LOADING DATASET
'''
train = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1] from [0-255]
                            download=True)

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

```
print(len(train))
```

```
     60000
```

```
from torch import tensor
traindata = [train[i] for i in range(len(train))]
train = torch.stack([d[0] for d in traindata], dim=0)
train=train[0:59999]
ys = [d[1] for d in traindata]
train_y = tensor(ys)

testdata = [test[i] for i in range(len(test))]
test = torch.stack([d[0] for d in testdata], dim=0)
test=test[0:9999]
ys = [d[1] for d in testdata]
test_y = tensor(ys)
```

Seeding

```
torch.manual_seed(120)
```

```
     <torch._C.Generator at 0x7f8f3c9c4650>
```

Dataset Loading

```
train = torch.rand(6000, 28, 28, 1)
test = torch.rand(1000, 28, 28, 1)
train_y = torch.randint(0,9, (6000,))
test_y = torch.randint(0,9, (1000,))
```

```
print(train_y)
```

```
tensor([1, 4, 5,  ..., 2, 8, 0])
```

Model Architeture

```
class NeuralNetwork(nn.Module):
    def __init__(self, input_dim):
        super(NeuralNetwork, self).__init__()
        self.cnn_layer_1 = nn.Conv2d(in_channels=1, out_channels=16,kernel_size=5, stride=1, padding=2)
        self.cnn_layer_2 = nn.Conv2d(in_channels=16, out_channels=32,kernel_size=5, stride=1, padding=2)

        self.flatten = nn.Flatten()
        self.maxpool = nn.MaxPool2d(2,2)

        self.linear_layer_1 = nn.Linear(32*7*7, 512)
        self.linear_layer_2 = nn.Linear(512, 128)
        self.linear_layer_3 = nn.Linear(128, 10)

        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
        self.dropout = nn.Dropout(.2)

        # self.flatten = nn.Flatten()

    def forward(self, x):

        x = self.cnn_layer_1(x)
        x = self.dropout(x)
        x = self.relu(x)
        x = self.maxpool(x)

        #print(x.shape)

        x = self.cnn_layer_2(x)
        x = self.dropout(x)
        x = self.relu(x)
        x = self.maxpool(x)

        #print(x.shape)

        x = self.flatten(x)
        #print(x.shape)

        x = self.linear_layer_1(x)
        x = self.dropout(x)
        x = self.relu(x)

        x = self.linear_layer_2(x)
        x = self.dropout(x)
        x = self.relu(x)

        x = self.linear_layer_3(x)
        #logits = self.sigmoid(x)
        return x
```

Model Creation

```
model = NeuralNetwork(784)
print(model)
```

```
NeuralNetwork(
  (cnn_layer_1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (cnn_layer_2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (linear_layer_1): Linear(in_features=1568, out_features=512, bias=True)
  (linear_layer_2): Linear(in_features=512, out_features=128, bias=True)
  (linear_layer_3): Linear(in_features=128, out_features=10, bias=True)
  (relu): ReLU()
  (sigmoid): Sigmoid()
  (dropout): Dropout(p=0.2, inplace=False)
)
```

```
from torchsummary import summary
summary(model,(1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 28, 28]             416
           Dropout-2           [-1, 16, 28, 28]               0
              ReLU-3           [-1, 16, 28, 28]               0
         MaxPool2d-4           [-1, 16, 14, 14]               0
            Conv2d-5           [-1, 32, 14, 14]          12,832
           Dropout-6           [-1, 32, 14, 14]               0
              ReLU-7           [-1, 32, 14, 14]               0
         MaxPool2d-8             [-1, 32, 7, 7]               0
           Flatten-9                 [-1, 1568]               0
           Linear-10                  [-1, 512]         803,328
          Dropout-11                  [-1, 512]               0
             ReLU-12                  [-1, 512]               0
           Linear-13                  [-1, 128]          65,664
          Dropout-14                  [-1, 128]               0
             ReLU-15                  [-1, 128]               0
           Linear-16                   [-1, 10]           1,290
================================================================
Total params: 883,530
Trainable params: 883,530
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.49
Params size (MB): 3.37
Estimated Total Size (MB): 3.87
----------------------------------------------------------------
```

Optimizer

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=.001)


def trainModel(model, loss_fn, optimizer):
    model.train()

    batch = 100

    loss = 0
    for i in range(train.shape[0]):
      x, y = torch.reshape(train[i],(1,1,28,28)), torch.tensor([train_y[i]], dtype=torch.float)
      label=torch.zeros([1,10,], dtype=torch.float32)
      label[0,int(y.item())]=1
      # Compute prediction error
      pred = model(x)
      #print(pred)
      #print(label)
      loss += loss_fn(pred, label)

      if i>0 and (i+1)%batch == 0:
          # Backpropagation
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()
          print(f'Training Loss: {loss.item():.4f}', end="\r")
          loss = 0
    print()
```

Model testing

```
def testModel(model, loss_fn):
    model.eval()

    size = test.shape[0]
    correct=0
    loss = 0
    total =10000
    with torch.no_grad():
      for i in range(test.shape[0]):
        x, y = torch.reshape(test[i],(1,1,28,28)), torch.tensor([test_y[i]], dtype=torch.float)
        label=torch.zeros([1,10,], dtype=torch.float32)
        label[0,int(y.item())]=1
        pred = model(x)
        #print(pred)
        predicted = torch.argmax(pred)
        #print(predicted)
        #print(y)


        # Total correct predictions
        correct += (predicted == int(y)).sum()

        loss += loss_fn(pred, label)

    loss /= size
    accuracy = 100 * correct.item() / total

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



epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    trainModel(model, loss_fn, optimizer)
    testModel(model, loss_fn)
print("Done!")
```

```
     Epoch 1
     -------------------------------
     Training Loss: 0.8743
     Loss: {tensor(0.0794)}. Accuracy: 97.75
     Epoch 2
     -------------------------------
     Training Loss: 0.6520
     Loss: {tensor(0.0633)}. Accuracy: 98.16
     Epoch 3
     -------------------------------
     Training Loss: 0.6977
     Loss: {tensor(0.0399)}. Accuracy: 98.8
     Epoch 4
     -------------------------------
     Training Loss: 0.1341
     Loss: {tensor(0.0383)}. Accuracy: 98.8
     Epoch 5
     -------------------------------
     Training Loss: 1.1260
     Loss: {tensor(0.0332)}. Accuracy: 99.01
     Done!
```

Session 04

Goals:

1. To know about Recurrant Neural Network (RNN)
2. To Know about Long Short Term Memory (LSTM)

RNN vs LSTM

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are both types of neural networks used for processing sequential data. However, there are significant differences between the two, primarily in their ability to handle long-range dependencies and mitigate the vanishing gradient problem. Here are some key differences:

Architecture:

RNN: RNNs have a simple architecture with a single hidden layer. They process input sequences one step at a time, and each step's output is used as input for the next step.
LSTM: LSTMs, on the other hand, have a more complex architecture with multiple interacting layers (gates) within the recurrent unit. These gates control the flow of information and allow LSTMs to capture long-term dependencies.

Handling Long-Term Dependencies:

RNN: RNNs struggle to capture long-range dependencies in sequential data because of the vanishing gradient problem. This means that as sequences get longer, RNNs may have difficulty retaining and propagating information over many time steps.
LSTM: LSTMs were specifically designed to address the vanishing gradient problem. They use specialized memory cells and gating mechanisms that allow them to capture and propagate information over long sequences, making them better suited for tasks that require modeling long-term dependencies.

Gating Mechanisms:

RNN: RNNs do not have explicit gating mechanisms. They simply apply a weighted sum of the current input and the previous hidden state at each time step.
LSTM: LSTMs have three gating mechanisms: the input gate, the forget gate, and the output gate. These gates regulate the flow of information, allowing LSTMs to add or remove information from the cell state, which helps in managing long-term dependencies.

Gradient Flow:

RNN: RNNs often suffer from the vanishing gradient problem, which can make training deep networks challenging. As gradients backpropagate through many time steps, they tend to become very small or very large, affecting the learning process.
LSTM: LSTMs are better at mitigating the vanishing gradient problem due to their gating mechanisms. The gates allow gradients to flow more easily through the network, enabling the training of deep LSTM architectures.

Computational Complexity:

RNN: RNNs are computationally less complex compared to LSTMs, which makes them faster to train and deploy.
LSTM: LSTMs are more computationally intensive due to their additional gating mechanisms and multiple internal operations. This increased complexity can result in longer training times and higher resource requirements.

In summary, while RNNs are simple and computationally efficient, they struggle with capturing long-term dependencies in sequential data. LSTMs, with their complex architecture and gating mechanisms, are designed to address these issues and are better suited for tasks that require modeling relationships over longer sequences. When working with sequential data, LSTMs are often the preferred choice when available resources and computational time allow for their use.

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


# Hyperparameters
sequence_length = 28
input_size =28
hidden_size = 28
num_layers = 2
num_classes= 10
batch_size = 100
num_iters = 1200
learning_rate = 0.01  # 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 DATASET
'''
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1] from [0-255]
                            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,drop_last=True)   # It's better to shuffle the whole t

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

RNN: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

LSTM: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size= hidden_size
        self.num_layers = num_layers

        #self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True) # For uni Directional LSTM
        #self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True) # For uni Directional RNN
        #self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True,bidirectional=True) # For Bi
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True,bidirectional=True) # For Bi
        #self.fc = nn.Linear(hidden_size, num_classes) #For uni Directional
        self.fc = nn.Linear(hidden_size*2, num_classes) #For Bidirectional

    def forward(self, x):
        # set initial hidden and cell states
        #h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device) #For uni Directional
        #c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device) #For uni Directional
        h0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(device) #For Bidirectional
        c0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(device) #For Bidirectional

        #Forward Propagation
        #out, _  = self.rnn(x,h0)
        out, _  = self.lstm(x,(h0,c0)) #out: tensor of shape (batch size, seq_length, hidden_size)
        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

input_size – The number of expected features in the input x

hidden_size – The number of features in the hidden state h

num_layers – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1

bias – If False, then the layer does not use bias weights b_ih and b_hh. Default: True

batch_first – If True, then the input and output tensors are provided as (batch, seq, feature) instead of (seq, batch, feature). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: False

dropout – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0

bidirectional – If True, becomes a bidirectional LSTM. Default: False

proj_size – If > 0, will use LSTM with projections of corresponding size. Default: 0

```
'''
INSTANTIATE MODEL CLASS
'''
model = RNN( input_size, hidden_size, num_layers, num_classes)
# To enable GPU
model.to(device)
```

```
RNN(
  (lstm): LSTM(28, 28, num_layers=2, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=56, out_features=10, bias=True)
)
```

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

```
'''
TRAIN THE MODEL
```

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

        images = images.reshape(-1, sequence_length, input_size).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 % 300 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.reshape(-1, sequence_length, input_size).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: 300. Loss: 0.23381160199642181. Accuracy: 94.48
Iteration: 600. Loss: 0.1339578628540039. Accuracy: 95.12
Iteration: 900. Loss: 0.1674966961145401. Accuracy: 97.25
Iteration: 1200. Loss: 0.08532698452472687. Accuracy: 97.11
```

Session 05

Goals:

1. To know about Natural Language Processing
2. To Know about Basic Terminologies of NLP

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}`.

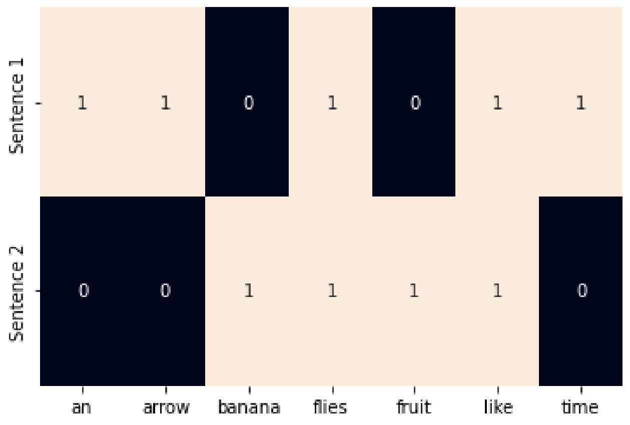
The binary encoding for **"like a banana"** would then be:

```
[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 0x7fdb7ea99fa0>
```



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
stupidity	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

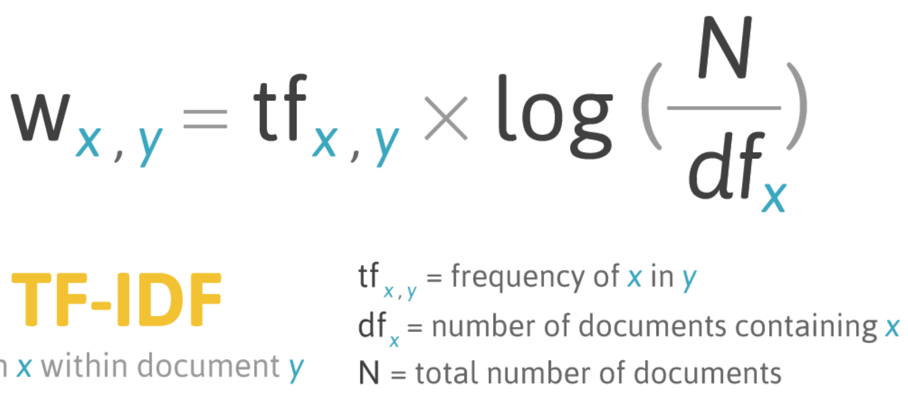
We can calculate the IDF values for each word like this. Thus, the IDF values for the entire vocabulary would be:

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.


$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$
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
import matplotlib as plt

corpus = ['Neural networks are a fundamental component of artificial intelligence, playing a pivotal role in
          'Their ability to mimic the human brain interconnected structure and learning capabilities enables
          'Neural networks have revolutionized various industries, such as healthcare, finance, and autonomo
          'They have significantly enhanced natural language processing, making virtual assistants and langu
          'Furthermore, neural networks have propelled computer vision to new heights, enabling machines to

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,linewidths=.5, xticklabels=dictionary,
                                           yticklabels=['Sentence 1','Sentence 2','Sentence 3'])
```

```
[[0.         0.         0.26824958 0.         0.         0.26824958
  0.26824958 0.         0.         0.         0.         0.
  0.         0.         0.26824958 0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.26824958 0.         0.         0.         0.         0.
  0.26824958 0.         0.26824958 0.         0.         0.
  0.         0.         0.         0.         0.26824958 0.
  0.         0.17964987 0.17964987 0.         0.         0.26824958
  0.         0.26824958 0.26824958 0.         0.         0.
  0.         0.         0.26824958 0.         0.         0.
  0.         0.26824958 0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.        ]
 [0.21862917 0.21862917 0.         0.         0.12317186 0.
  0.         0.         0.         0.         0.21862917 0.
  0.21862917 0.21862917 0.         0.         0.         0.
  0.21862917 0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.21862917
  0.         0.         0.         0.21862917 0.         0.21862917
  0.         0.         0.         0.21862917 0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.21862917 0.         0.
  0.         0.         0.         0.         0.21862917 0.21862917
  0.         0.         0.21862917 0.21862917 0.21862917 0.
  0.35277728 0.         0.21862917 0.         0.         0.
  0.         0.         0.21862917]
 [0.         0.         0.         0.23177513 0.26115613 0.
  0.         0.23177513 0.         0.23177513 0.         0.23177513
  0.         0.         0.         0.         0.23177513 0.
  0.         0.         0.         0.23177513 0.         0.23177513
  0.         0.         0.15522251 0.23177513 0.         0.
  0.         0.23177513 0.         0.         0.         0.
  0.         0.         0.23177513 0.         0.         0.
  0.         0.15522251 0.15522251 0.         0.         0.
  0.23177513 0.         0.         0.         0.         0.
  0.         0.23177513 0.         0.         0.         0.
  0.23177513 0.         0.         0.         0.         0.
  0.         0.         0.         0.23177513 0.23177513 0.23177513
  0.         0.         0.        ]
 [0.         0.         0.         0.         0.1418874  0.
  0.         0.         0.25184911 0.         0.         0.
  0.         0.         0.         0.         0.         0.25184911
  0.         0.         0.25184911 0.         0.         0.
```

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.

```
  0.25593446 0.         0.         0.         0.         0.
```

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 = '/content/imdb.csv'
# if not Path(DATA_PATH).is_file():
#     gdd.download_file_from_google_drive(
#         file_id='1EWReHFoPXK2Z-zdELR_LC6J6VRgp-QgN',
#         dest_path=DATA_PATH,
#     )
```

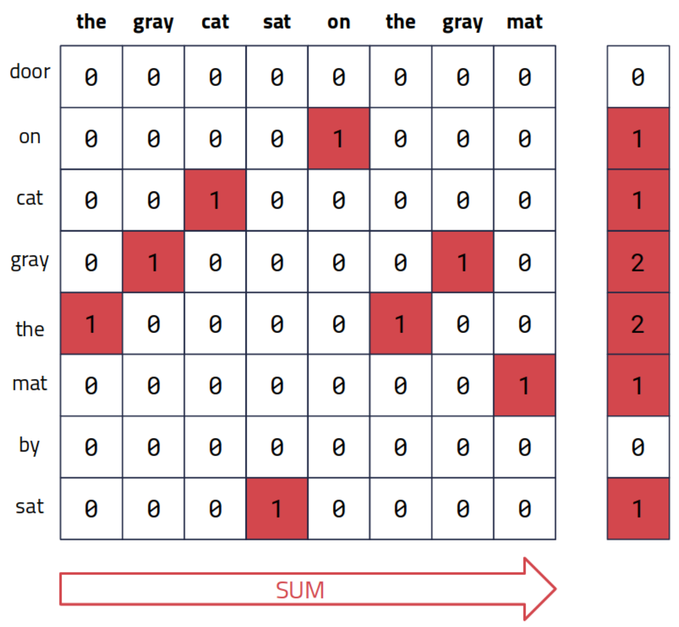
```
# Upload imdb.csv file in colab
DATA_PATH = '/content/imdb.csv'
import pandas as pd
df=pd.read_csv(DATA_PATH)
print(df.head())
```

```
                                                review sentiment
     0  One of the other reviewers has mentioned that ...  positive
     1  A wonderful little production. <br /><br />The...  positive
     2  I thought this was a wonderful way to spend ti...  positive
     3  Basically there's a family where a little boy ...  negative
     4  Petter Mattei's "Love in the Time of Money" is...  positive
```

```
import numpy as np
x=np.array(pd)
print(x)
```

```
     <module 'pandas' from '/usr/local/lib/python3.8/dist-packages/pandas/__init__.py'>
```

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.sentiment.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)

print((df))
codes=[0,1]
df.columns = ["review", "sentiment"]
df["sentiment"] = df["sentiment"].astype('category')
df["sentiment"] = df["sentiment"].cat.codes

df_train = df.head(900)
df_test = df.tail(100)
print(df_train)
dataset = Sequences(df_train)

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

```
                                                review sentiment
     0     One of the other reviewers has mentioned that ...  positive
     1     A wonderful little production. <br /><br />The...  positive
     2     I thought this was a wonderful way to spend ti...  positive
     3     Basically there's a family where a little boy ...  negative
     4     Petter Mattei's "Love in the Time of Money" is...  positive
```

```
...                                                  ...       ...
1494  Zoey 101 is basically about a girl named Zoey ...  negative
1495  This movie is terrible, it was so difficult to...  negative
1496  The only thing serious about this movie is the...  positive
1497  2005 was one of the best year for movies. We h...  positive
1498  According to John Ford's lyrically shot, ficti...  positive

[1499 rows x 2 columns]
                                                review  sentiment
0    One of the other reviewers has mentioned that ...          1
1    A wonderful little production. <br /><br />The...          1
2    I thought this was a wonderful way to spend ti...          1
3    Basically there's a family where a little boy ...          0
4    Petter Mattei's "Love in the Time of Money" is...          1
..                                                 ...        ...
895  But it is kinda hilarious, at least if you gre...          1
896  One of the two Best Films of the year. A well ...          1
897  I managed to see this at the New York Internat...          1
898  Why else would he do this to me?<br /><br />No...          0
899  Minimal script, minimal character development,...          0

[900 rows x 2 columns]
```

```
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=16683, 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()
        #print(target)
        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: 0.676
Epoch #3        Train Loss: 0.654
Epoch #4        Train Loss: 0.626
Epoch #5        Train Loss: 0.592
Epoch #6        Train Loss: 0.555
Epoch #7        Train Loss: 0.514
Epoch #8        Train Loss: 0.472
Epoch #9        Train Loss: 0.428
Epoch #10       Train Loss: 0.384
Epoch #11       Train Loss: 0.341
Epoch #12       Train Loss: 0.300
Epoch #13       Train Loss: 0.261
Epoch #14       Train Loss: 0.225
Epoch #15       Train Loss: 0.192
Epoch #16       Train Loss: 0.162
Epoch #17       Train Loss: 0.136
Epoch #18       Train Loss: 0.113
Epoch #19       Train Loss: 0.094
Epoch #20       Train Loss: 0.077
Epoch #21       Train Loss: 0.063
Epoch #22       Train Loss: 0.052
Epoch #23       Train Loss: 0.043
Epoch #24       Train Loss: 0.035
Epoch #25       Train Loss: 0.029
Epoch #26       Train Loss: 0.024
Epoch #27       Train Loss: 0.020
```

```
    Epoch #28       Train Loss: 0.016
    Epoch #29       Train Loss: 0.014
    Epoch #30       Train Loss: 0.012
    Epoch #31       Train Loss: 0.010
    Epoch #32       Train Loss: 0.008
    Epoch #33       Train Loss: 0.007
    Epoch #34       Train Loss: 0.006
    Epoch #35       Train Loss: 0.005
    Epoch #36       Train Loss: 0.005
    Epoch #37       Train Loss: 0.004
    Epoch #38       Train Loss: 0.004
    Epoch #39       Train Loss: 0.003
    Epoch #40       Train Loss: 0.003
    Epoch #41       Train Loss: 0.003
    Epoch #42       Train Loss: 0.002
    Epoch #43       Train Loss: 0.002
    Epoch #44       Train Loss: 0.002
    Epoch #45       Train Loss: 0.002
    Epoch #46       Train Loss: 0.002
    Epoch #47       Train Loss: 0.002
    Epoch #48       Train Loss: 0.001
    Epoch #49       Train Loss: 0.001
    Epoch #50       Train Loss: 0.001
    Epoch #51       Train Loss: 0.001
    Epoch #52       Train Loss: 0.001
    Epoch #53       Train Loss: 0.001
    Epoch #54       Train Loss: 0.001
    Epoch #55       Train Loss: 0.001
    Epoch #56       Train Loss: 0.001
    Epoch #57       Train Loss: 0.001
```

```
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)
```

```
    0.332: Negative sentiment
    0
```

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

```
    0.78: 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


from sklearn.metrics import classification_report
```

```
pred_labels = []

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

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)
print(classification_report(labels, pred_labels,digits=4))
```

```
0.0724: Negative sentiment
3.02e-09: Negative sentiment
0.994: Positive sentiment
0.864: Positive sentiment
0.998: Positive sentiment
0.505: Positive sentiment
0.998: Positive sentiment
0.274: Negative sentiment
0.935: Positive sentiment
0.972: Positive sentiment
0.999: Positive sentiment
0.000407: Negative sentiment
0.982: Positive sentiment
1.76e-09: Negative sentiment
0.992: Positive sentiment
0.999: Positive sentiment
0.00205: Negative sentiment
7.81e-06: Negative sentiment
0.000193: Negative sentiment
0.461: Negative sentiment
0.984: Positive sentiment
0.999: Positive sentiment
Accuracy: 0.800000
Precision: 0.862745
Recall: 0.771930
F1 score: 0.814815
[[36  7]
 [13 44]]
              precision    recall  f1-score   support

           0     0.7347    0.8372    0.7826        43
           1     0.8627    0.7719    0.8148        57

    accuracy                         0.8000       100
   macro avg     0.7987    0.8046    0.7987       100
weighted avg     0.8077    0.8000    0.8010       100
```

Session 06

Goals:

1. To know about the Sentence Representation Techniques of NLP
2. To know about Techniques for Sentence Preprocessing

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

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 $3.88\nin New York.', 'Please buy me two of t
['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York', '.'
```

```
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 t

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)
```

```
/usr/local/lib/python3.8/dist-packages/torch/cuda/__init__.py:497
  warnings.warn("Can't initialize NVML")


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
```

Downloading Large spaCy model

```
!python -m spacy download en_core_web_lg

import en_core_web_lg

nlp = en_core_web_lg.load()
```

```
    /usr/local/lib/python3.8/dist-packages/torch/cuda/__init__.py:497
      warnings.warn("Can't initialize NVML")
    2023-01-30 07:43:41.145527: E tensorflow/stream_executor/cuda/cud
    Looking in indexes: https://pypi.org/simple, https://us-python.pk
    Collecting en-core-web-lg==3.4.1
      Downloading https://github.com/explosion/spacy-models/releases/
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 587.7/587.7 MB 2.3 M
    Requirement already satisfied: spacy<3.5.0,>=3.4.0 in /usr/local/
    Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /usr/loca
    Requirement already satisfied: pathy>=0.3.5 in /usr/local/lib/pyt
    Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /usr/
    Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /usr/local/
    Requirement already satisfied: spacy-loggers<2.0.0,>=1.0.0 in /us
    Requirement already satisfied: thinc<8.2.0,>=8.1.0 in /usr/local/
    Requirement already satisfied: numpy>=1.15.0 in /usr/local/lib/py
    Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /usr/local/
    Requirement already satisfied: typer<0.8.0,>=0.3.0 in /usr/local/
    Requirement already satisfied: srsly<3.0.0,>=2.4.3 in /usr/local/
    Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/lo
    Requirement already satisfied: packaging>=20.0 in /usr/local/lib/
    Requirement already satisfied: jinja2 in /usr/local/lib/python3.8
    Requirement already satisfied: wasabi<1.1.0,>=0.9.1 in /usr/local
    Requirement already satisfied: setuptools in /usr/local/lib/pytho
    Requirement already satisfied: langcodes<4.0.0,>=3.2.0 in /usr/lo
    Requirement already satisfied: smart-open<7.0.0,>=5.2.1 in /usr/l
    Requirement already satisfied: pydantic!=1.8,!=1.8.1,<1.11.0,>=1.
    Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in /usr/lo
    Requirement already satisfied: spacy-legacy<3.1.0,>=3.0.10 in /us
    Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/l
    Requirement already satisfied: typing-extensions>=4.2.0 in /usr/l
    Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/pyt
    Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/loca
    Requirement already satisfied: certifi>=2017.4.17 in /usr/local/l
    Requirement already satisfied: chardet<5,>=3.0.2 in /usr/local/li
    Requirement already satisfied: confection<1.0.0,>=0.0.1 in /usr/l
    Requirement already satisfied: blis<0.8.0,>=0.7.8 in /usr/local/l
```

```
Requirement already satisfied: click<9.0.0,>=7.1.1 in /usr/local/
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib
Installing collected packages: en-core-web-lg
Successfully installed en-core-web-lg-3.4.1
✔ Download and installation successful
You can now load the package 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', 'int
filtered_words = [word for word in words if word.lower() not in english
```

```
print(filtered_words)
```

```
{"haven't", 'where', 'out', 'the', 'mightn', 'down', 'shouldn', '
['natural', 'language', 'processing', 'interesting']
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
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
```

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

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

print(tokens)
```

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

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', 'int
filtered_words = [word for word in words if word.lower() not in english

print(filtered_words)
```

```
    {'hers', 'wouldn', 'who', 'after', 'until', 'd', 'there', 'in', '
    ['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

sentence = sentence.lower()
print(sentence)
sentence = nltk.word_tokenize(sentence)
print(sentence)
sentence = [i for i in sentence if i not in puncset] # Removing punctua
print(sentence)
sentence = [w for w in sentence if w.isalpha()] # Removing numbers and
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 parliai
['hun', 'sen', "'s", 'cambodian', 'ca', "n't", 'people', "'s", 'p
['hun', 'sen', "'s", 'cambodian', 'ca', "n't", 'people', "'s", 'p
['hun', 'sen', 'cambodian', 'ca', 'people', 'party', 'won', 'of',
```

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

import nltk
from nltk.stem import WordNetLemmatizer
nltk.download("omw-1.4")

nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()

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

```
     [nltk_data] Downloading package omw-1.4 to /root/nltk_data...
     [nltk_data] Downloading package wordnet to /root/nltk_data...
     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'))
```

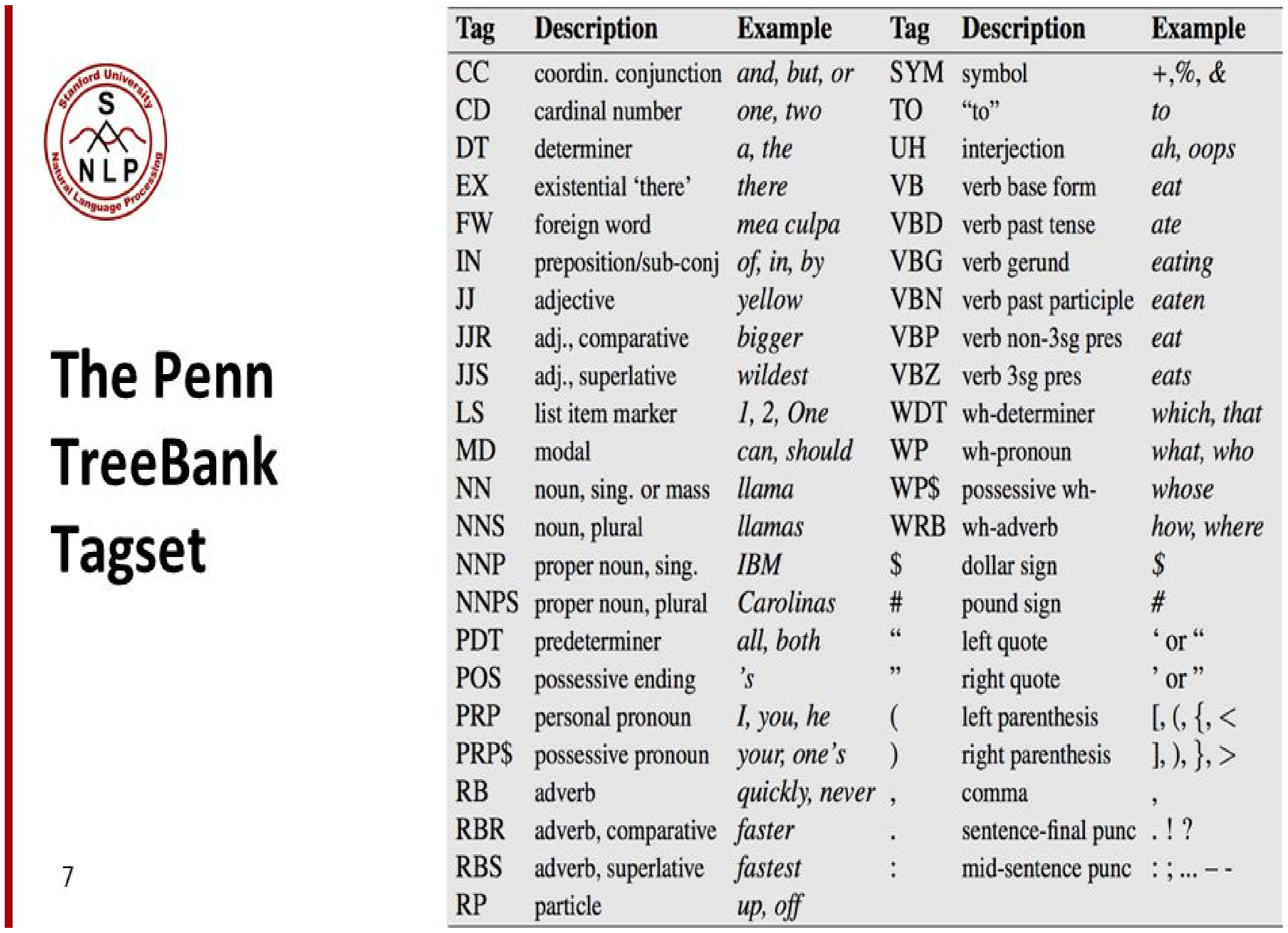
```
believ
belief
```

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', 'NN
```

```
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
```

Session 07

Goals:

1. Evaluation of the project.

Evaluation Criteria:

1.	Project Design	- 10 Marks
2.	Project Report	- 10 Marks
3.	Project Presentation	- 10 Marks