

# KPR Institute of Engineering and Technology, Coimbatore



## Department of Artificial Intelligence and Machine Learning

**Two Weeks STTP on  
“Machine Learning, Data Science and Gen AI”  
(05-02-2024 to 16-02-2024)**



**Deep Learning Unleashed: PyTorch**  
(a deep learning library)

**Dr.S.Karthikeyan**  
**HoD/AIML**

[aiquantalytics@gmail.com](mailto:aiquantalytics@gmail.com)





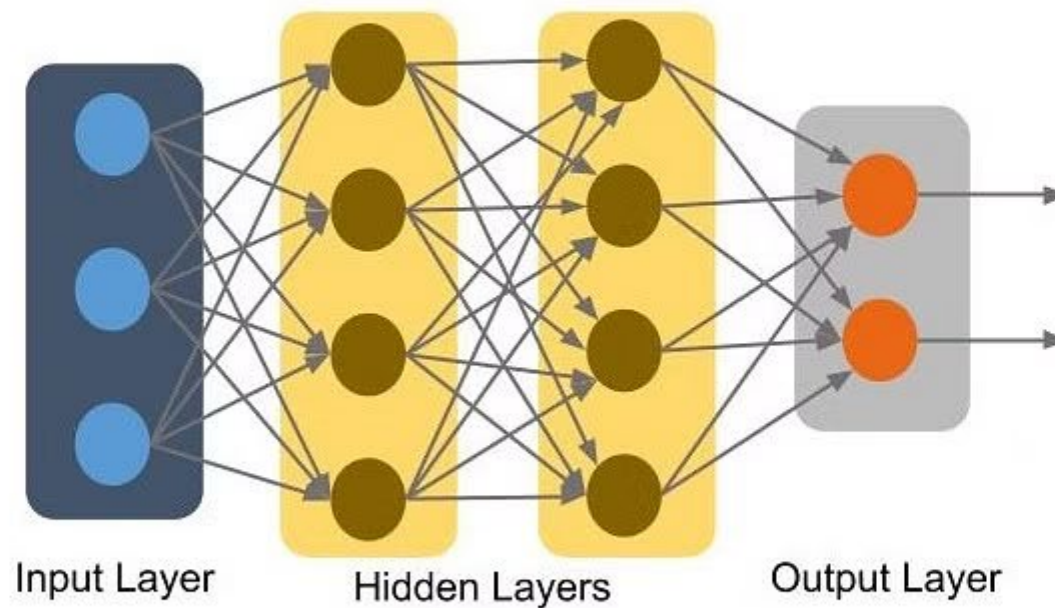
# Outline of the presentation

- **Why Deep Learning Libraries and Why PyTorch ?**
- PyTorch Installation
- Introduction: PyTorch
  - Tensors
  - Variables
  - Models/Optimizers
  - Autograd and the dynamic computational graph
- Examples : Colab
- Summary
- References

# What is Deep Learning ?

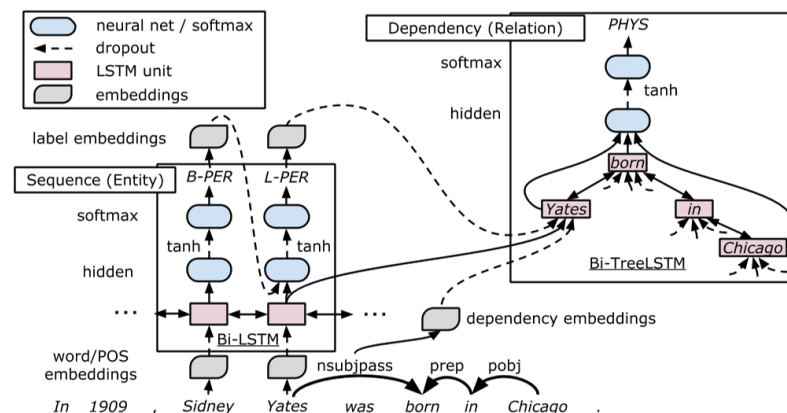
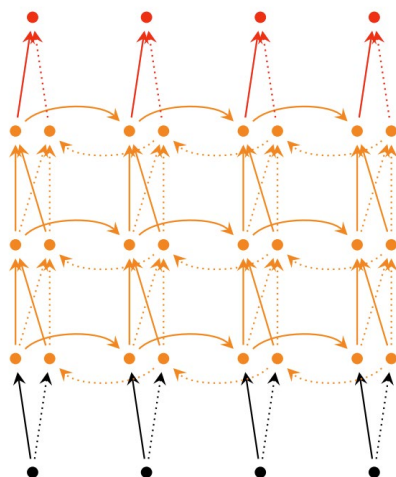


Deep Learning is a subset of Machine Learning and it works on the structure and functions of a human brain. It learns from data that is unstructured and uses complex algorithms to train a neural net.



# Why a DL Library is Necessary?

- Complicated DL architectures
  - Easily build big computational graphs
  - Easily compute gradients

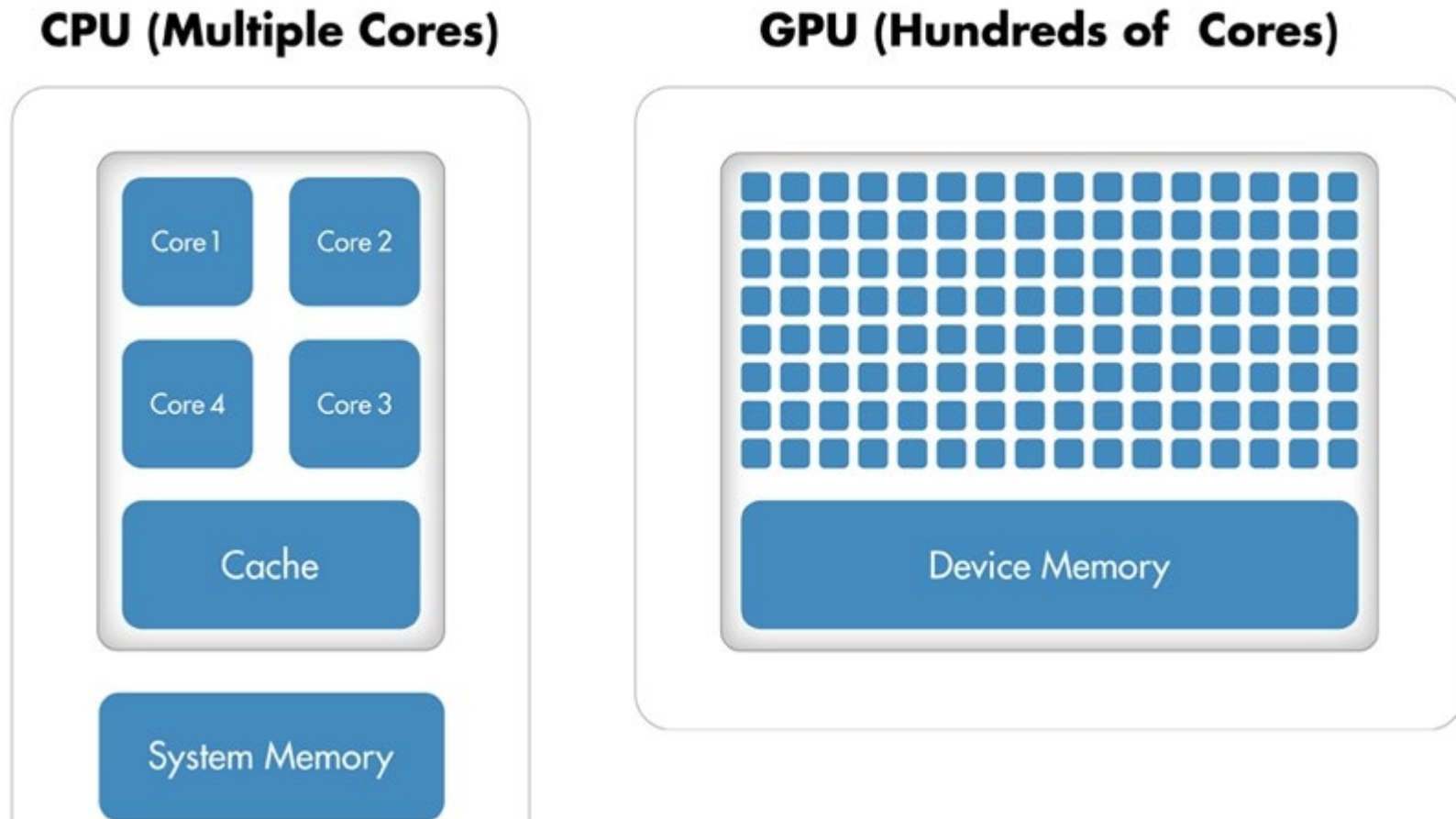


aiquantalytics@gmail.com



# Why a DL Library is Necessary?

- Run it all efficiently on GPU



# What is PyTorch ?

- PyTorch is the premier [open-source deep learning framework](#) developed and maintained by Facebook.
- PyTorch has gained popularity among the research community as it is [easy to develop and debug machine learning models in PyTorch](#).
- PyTorch and easy to integrate with any pipeline.
- With proper seeding, [PyTorch can generate reproducible models](#).
- Like Tensorflow, a machine learning library by Google, [PyTorch works with tensors](#) which can be thought of as matrices with higher dimensions. These are equivalent to ndarrays in NumPy.
- Other deep learning framework available:
  - For example, MATLAB can be used to design and train models.
  - Keras is a high-level API library that uses TensorFlow as backend and suitable for beginners. It is straightforward to build and test models using Keras. However, it can be difficult to debug in Keras.



- At its core, PyTorch is a mathematical library that [allows you to perform efficient computation and automatic differentiation](#) on graph-based models.
- Achieving this directly is challenging, although thankfully, the modern [PyTorch API](#) provides classes and idioms that allow you to easily develop a suite of deep learning models.

Let's get started.



# As a outline

- Open source **Deep learning library**
- Developed by Facebook's AI Research lab
- **It leverages the power of GPUs**
- **Automatic computation of gradients**
- Makes it easier to test and develop new ideas.

Other libraries?





# Automatic differentiation ?

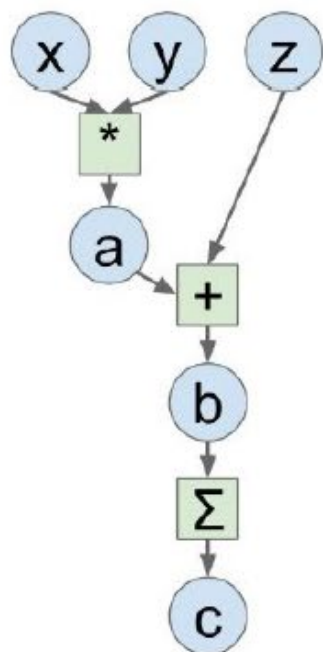
- PyTorch computes the gradient of a function with respect to the inputs by using automatic differentiation.
- Automatic differentiation is a technique that, given a computational graph, calculates the gradients of the inputs.
- Automatic differentiation can be performed in two different ways; forward and reverse mode.

# Automated Differentiation...

- Is not Symbolic Differentiation!
- Is not Numerical Differentiation!
- Instead, it relies on a specific quirk of scientific computing to make gradient computation really easy on computers.

# Why PyTorch?

## Computation Graph



## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

## Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                    feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

## PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```



# Why computational graphs exist

- For a single neuron with  $n$  inputs, we need to keep track of  $O(n)$  gradients.
- For a standard 784x800x10 vanilla **feedforward neural net for MNIST**, we need:
  - $784 + 800 \times 20 + 10 \times 20 + 3 = 16987$  gradients per training example
  - 60,000 training examples \* 2405 gradients = **1 billion gradients per epoch**
- **How do we keep track of tens of billions of gradients?!**

# Computational Graph

**Definition:** a data structure for storing gradients of variables used in computations.

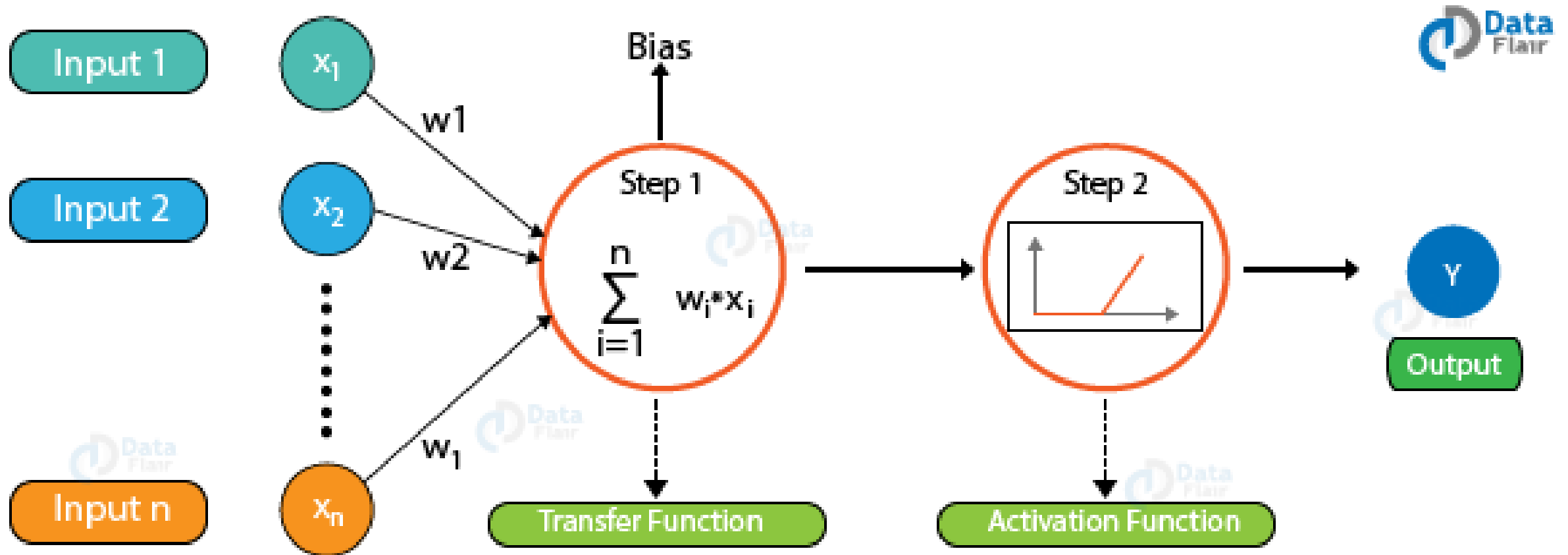
- Node  $v$  represents variable
  - Stores value
  - Gradient
  - The function that created the node
- Directed edge  $(u, v)$  represents the partial derivative of  $u$  w.r.t.  $v$
- To compute the gradient  $dL/dv$ , find the unique path from  $L$  to  $v$  and multiply the edge weights.

To compute the gradients, a tensor must have its parameter `requires_grad = true`. The gradients are same as the partial derivatives.

For example, in the function  $y = 2 * x + 1$ ,  $x$  is a tensor with `requires_grad = True`.

# A neuron in a computational graph

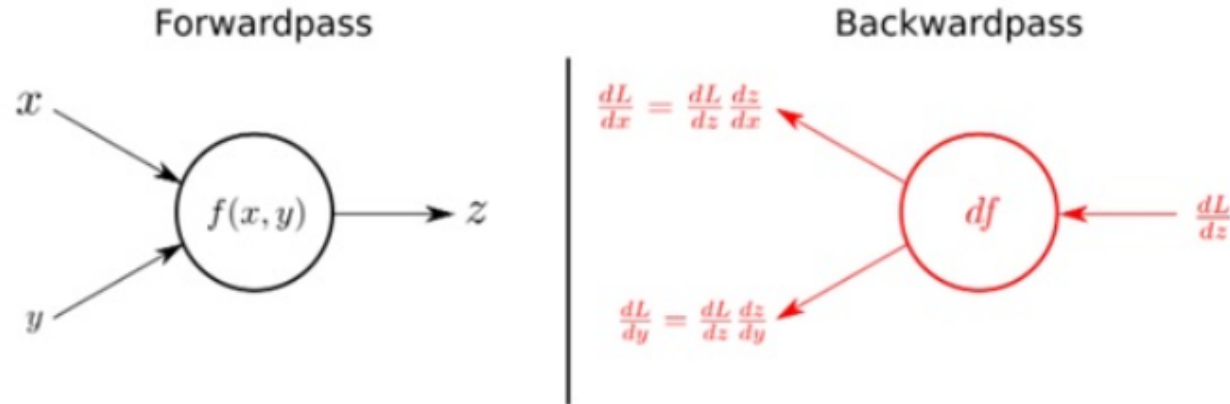
# Backpropagation for neural nets



Some of the popular activation functions used in Artificial Neural Networks are Sigmoid, RELU, Softmax, tanh etc.

# Backpropagation for neural nets: forward & Backward pass

To find the minima of a function, we can use the [gradient descent](#) algorithm. The gradient descent can be mathematically represented as:



Using the input variables  $x$  and  $y$ , the forward pass or propagation calculates output  $z$  as a function of  $x$  and  $y$  i.e.  $f(x, y)$ .

During backward pass or propagation, on receiving  $dL/dz$  (the derivative of the total loss,  $L$  with respect to the output,  $z$ ), we can calculate the individual gradients of  $x$  and  $y$  on the loss function by applying the chain rule, as shown in the figure.

$$^*W_x = W_x - \underset{\substack{\uparrow \\ \text{Learning} \\ \text{rate}}}{a} \left( \underset{\substack{\downarrow \\ \text{Derivative of Error} \\ \text{with respect to weight}}}{\frac{\partial \text{Error}}{\partial W_x}} \right)$$

The equation shows the update rule for a weight  $W_x$ . The new weight  $^*W_x$  is calculated by subtracting the product of the learning rate  $a$  and the derivative of the error with respect to the weight  $\frac{\partial \text{Error}}{\partial W_x}$  from the old weight  $W_x$ .



# Why **PYTORCH**



**Andrej Karpathy** ✓

@karpathy

Following



I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

Life is short, you need **PYTORCH**

# It is divided into three parts; they are:

- **How to Install PyTorch**
  - What Are Torch and PyTorch?
  - How to Install PyTorch
  - How to Confirm PyTorch Is Installed
- **PyTorch Deep Learning Model Life-Cycle**
  - Step 1: Prepare the Data
  - Step 2: Define the Model
  - Step 3: Train the Model
  - Step 4: Evaluate the Model
  - Step 5: Make Predictions
- **How to Develop PyTorch Deep Learning Models**
  - How to Develop an MLP for Binary Classification
  - How to Develop an MLP for Multiclass Classification
  - How to Develop an MLP for Regression
  - How to Develop a CNN for Image Classification

# Getting Started with PyTorch

## Installation

- Via Anaconda/Miniconda:

`conda install pytorch-c pytorch`

- Via pip:

`pip3 install torch`

**NOTE:** Latest PyTorch requires Python 3.8 or later.

# Prerequisites

- Please ensure that you have met the prerequisites below (e.g., numpy), depending on your package manager.
- [Anaconda](#) is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

## START LOCALLY

PyTorch Build	Stable (2.2.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.1	ROCm 5.7	CPU
Run this Command:	<pre>pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118</pre>			

# INSTALLATION

## Anaconda

To install PyTorch with Anaconda, you will need to open an Anaconda prompt via `Start | Anaconda3 | Anaconda Prompt`.

### No CUDA

To install PyTorch via Anaconda, and do not have a **CUDA-capable** system or do not require CUDA, in the above selector, choose OS: Windows, Package: Conda and CUDA: None. Then, run the command that is presented to you.

### With CUDA

To install PyTorch via Anaconda, and you do have a **CUDA-capable** system, in the above selector, choose OS: Windows, Package: Conda and the CUDA version suited to your machine. Often, the latest CUDA version is better. Then, run the command that is presented to you.

## pip

### No CUDA

To install PyTorch via pip, and do not have a **CUDA-capable** system or do not require CUDA, in the above selector, choose OS: Windows, Package: Pip and CUDA: None. Then, run the command that is presented to you.

### With CUDA

To install PyTorch via pip, and do have a **CUDA-capable** system, in the above selector, choose OS: Windows, Package: Pip and the CUDA version suited to your machine. Often, the latest CUDA version is better. Then, run the command that is presented to you.

# VERIFICATION

- To ensure that PyTorch was installed correctly, we can verify the installation by running sample PyTorch code. Here we will construct a [randomly initialized tensor](#).
- From the command line, type:

```
python
```

then enter the following code:

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

The output should be something similar to:

```
tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

- Additionally, to check if your GPU driver and CUDA is enabled and accessible by PyTorch, run the following commands to return whether or not the CUDA driver is enabled:
- `import torch`
- `torch.cuda.is_available()`



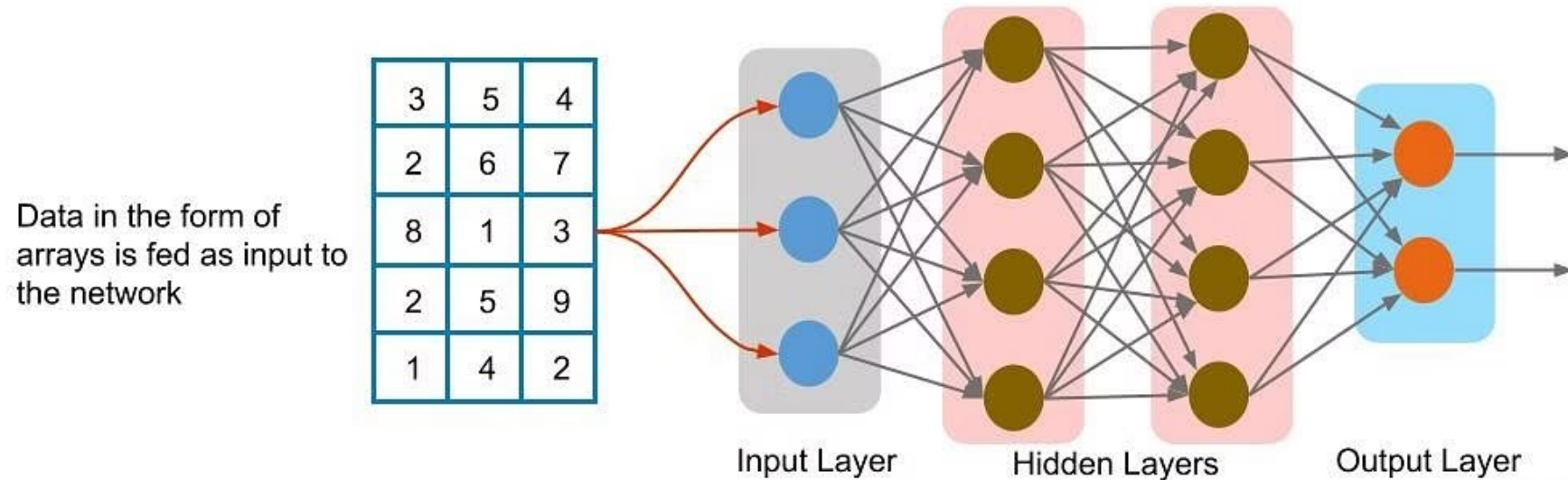
# Tensors

- Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.
- Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.
- Common operations for the creation and manipulation of these Tensors are similar to those for ndarrays in NumPy. (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication)

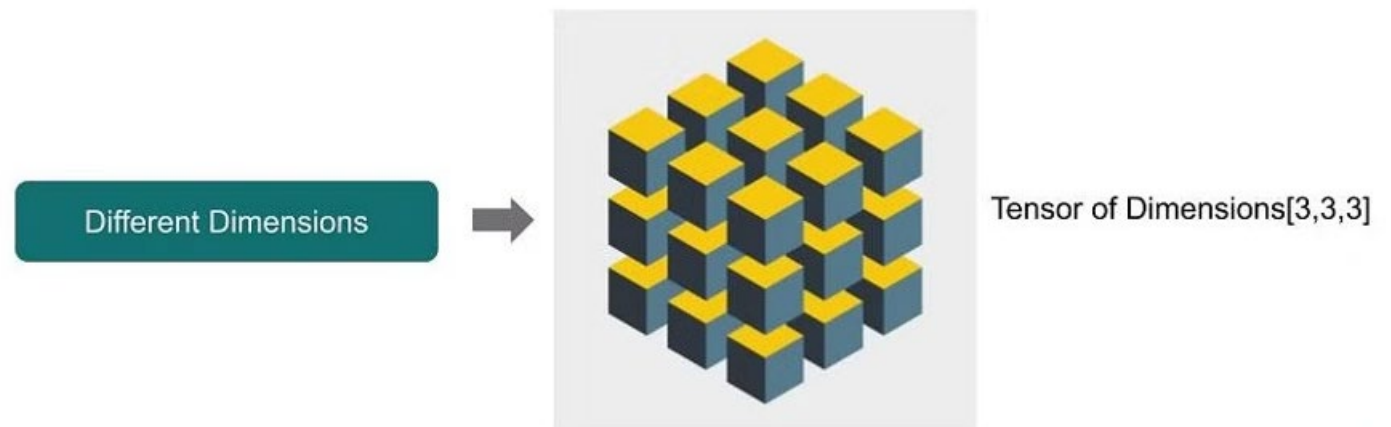
## Tensors



*Tensor is a generalization of vectors and matrices of potentially higher dimensions. Arrays of data with different dimensions and ranks that are fed as input to the neural network are called Tensors.*

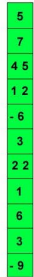


©Simplilearn. All rights reserved.



# Tensor Representation

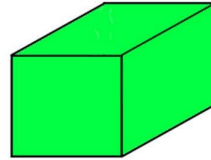
0 TENSOR /  
VECTOR



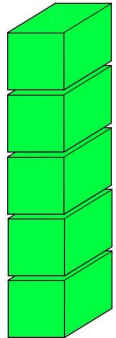
2D TENSOR /  
MATRIX

-9	4	2	5	7
3	0	12	8	61
1	23	-6	45	2
22	3	-1	72	6

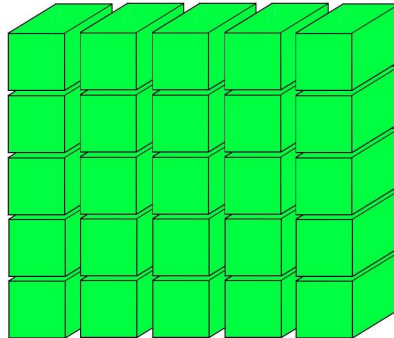
3D TENSOR /  
CUBE



-9	4	2	5	7
3	0	12	8	61
1	23	-6	45	2
22	3	-1	72	6



4D TENSOR  
VECTOR OF CUBES



5D TENSOR  
MATRIX OF CUBES

3

`A = torch.tensor(3)`

0 Dimensions

1.0  
2.0  
3.0

`B =`

`torch.tensor([1.0, 2.0, 3.0])`

1 Dimension

1.0 2.0  
3.0 4.0

`C = torch.tensor([  
[1.0, 2.0],  
[3.0, 4.0]])`

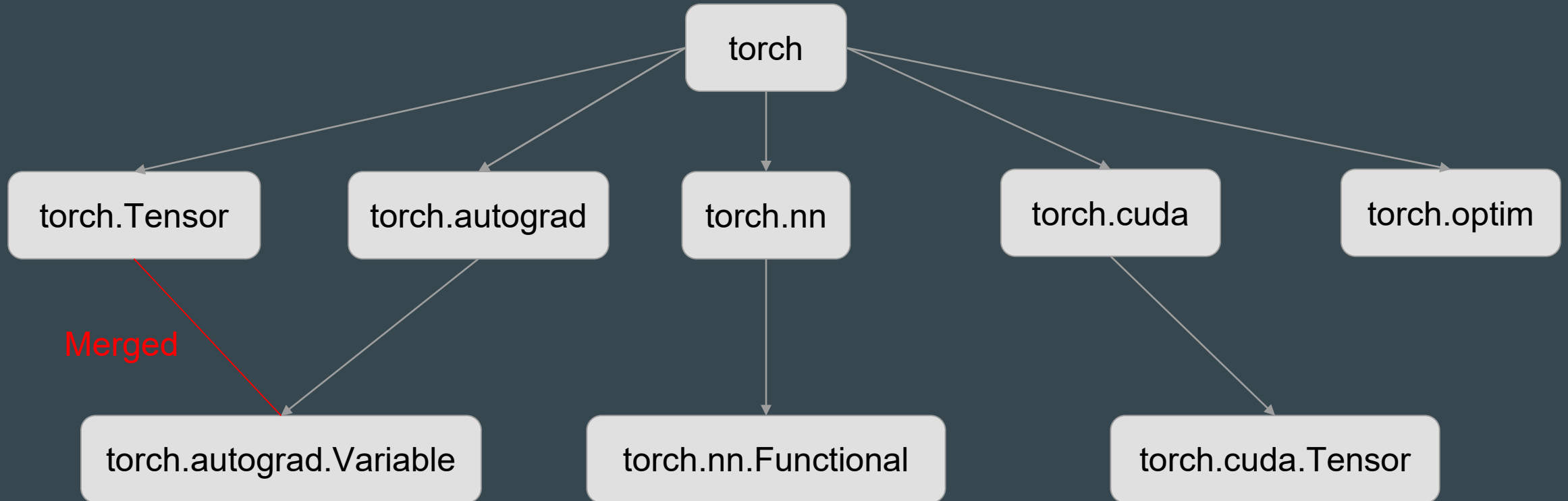
2 Dimensions

5.0	6.0
7.0	8.0
1.0	2.0
3.0	4.0

`D = torch.tensor([  
[[1., 2.], [3., 4.]],  
[[5., 6.], [7., 8.]])`

3 Dimensions

# Structure of PyTorch library



How do we store numbers?

# torch.Tensor

```
In [1]: import torch
```

```
In [2]: import numpy as np
```

```
In [3]: arr = np.random.randn((3,5))
```

```
In [4]: arr
```

```
Out[4]:
```

```
array([[ -1.00034281, -0.07042071,  0.81870386],
       [-0.86401346, -1.4290267 , -1.12398822],
       [-1.14619856,  0.39963316, -1.11038695],
       [ 0.00215314,  0.68790149, -0.55967659]])
```

```
In [5]: tens = torch.from_numpy(arr)
```

```
In [6]: tens
```

```
Out[6]:
```

```
-1.0003 -0.0704  0.8187
-0.8640 -1.4290 -1.1240
-1.1462  0.3996 -1.1104
0.0022  0.6879 -0.5597
[torch.DoubleTensor of size 4x3]
```

```
In [7]: another_tensor = torch.LongTensor([[2,4],[5,6]])
```

```
In [7]: another_tensor
```

```
Out[13]:
```

```
 2  4
 5  6
[torch.LongTensor of size 2x2]
```

```
In [8]: random_tensor = torch.randn((4,3))
```

```
In [9]: random_tensor
```

```
Out[9]:
```

```
1.0070 -0.6404  1.2707
-0.7767  0.1075  0.4539
-0.1782 -0.0091 -1.0463
 0.4164 -1.1172 -0.2888
[torch.FloatTensor of size 4x3]
```

# Creating Tensors from 2d data

[https://colab.research.google.com/drive/1-L2LJmiV\\_rgCtzslJOMfF\\_9fwJuxsj09#scrollTo=xaso\\_dtjsExr](https://colab.research.google.com/drive/1-L2LJmiV_rgCtzslJOMfF_9fwJuxsj09#scrollTo=xaso_dtjsExr)

In [5]:

```
1 import torch
2
3 x= torch.tensor([[1,2,3],[4,5,6]])
4 y= torch.tensor([[7,8,9],[10,11,12]])
5
6 f= 2*x + y
7 print(f)
```

```
tensor([[ 9, 12, 15],
        [18, 21, 24]])
```

Tensors can be created from Python lists with the `torch.Tensor()` function.



```
# Example with 1-D data
data = [1.0, 2.0, 3.0]
tensor = torch.Tensor(data)
print("Example with 1-D data")
print(tensor)

# Example with 2-D data
data = [[1., 2., 3.], [4., 5., 6]]
tensor = torch.Tensor(data)
print("\nExample with 2-D data")
print(tensor)

# Example with 3-D data
data = [[[1.,2.], [3.,4.]],
        [[5.,6.], [7.,8.]]]
tensor = torch.Tensor(data)
print("\nExample with 3-D data")
print(tensor)
```



```
Example with 1-D data
tensor([1., 2., 3.])
```

```
Example with 2-D data
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

```
Example with 3-D data
tensor([[[1., 2.],
         [3., 4.]],
        [[5., 6.],
         [7., 8.]])
```



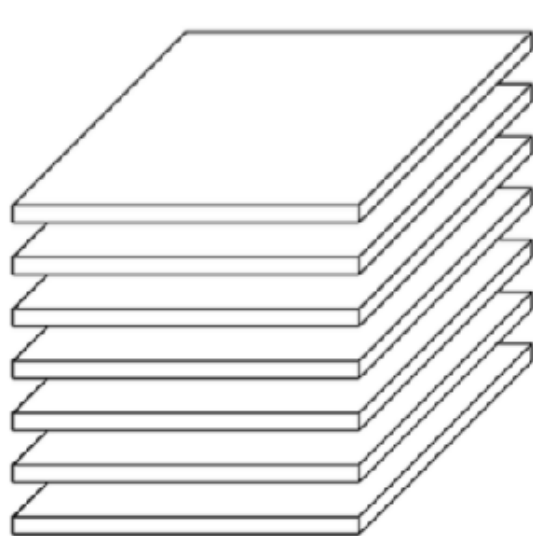


# torch.Tensor

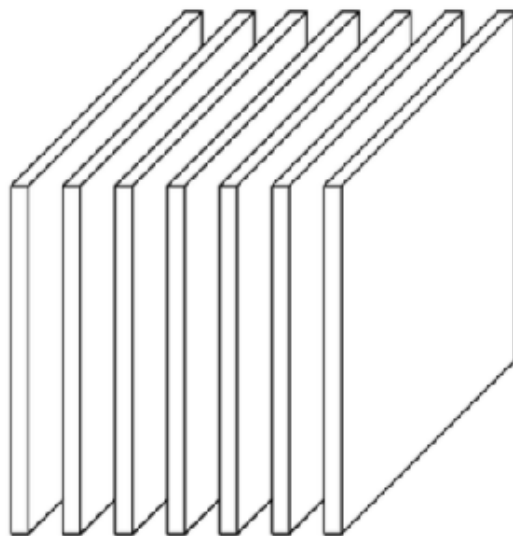
```
a = torch.rand(10, 10, 5)  
print a[0, :, :]
```

```
a=torch.rand(10,10,5)  
# print(a[0,:,:])  
a
```

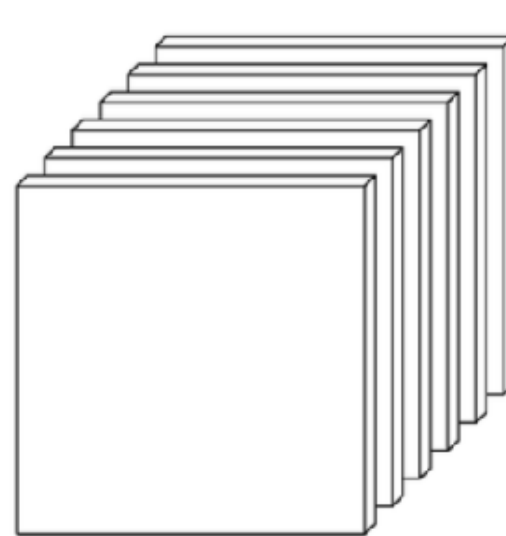
```
tensor([[[0.0906, 0.4504, 0.0248, 0.2231, 0.4575],  
        [0.4016, 0.9206, 0.0685, 0.8278, 0.3382],  
        [0.7372, 0.9916, 0.9010, 0.7928, 0.2139],  
        [0.0805, 0.3459, 0.0934, 0.2869, 0.9638],  
        [0.5314, 0.6018, 0.6045, 0.6114, 0.5295],  
        [0.3878, 0.4669, 0.4391, 0.0751, 0.0704],  
        [0.9769, 0.7784, 0.2606, 0.2783, 0.9258],  
        [0.8198, 0.1502, 0.4674, 0.9762, 0.7562],  
        [0.2367, 0.3164, 0.4686, 0.7513, 0.0115],  
        [0.8363, 0.9509, 0.5663, 0.9417, 0.1170]],  
       [[0.0539, 0.8370, 0.4993, 0.5769, 0.0044],  
        [0.7279, 0.8796, 0.6688, 0.4539, 0.0221],  
        [0.5837, 0.4007, 0.2441, 0.0933, 0.2265],  
        [0.6584, 0.9403, 0.7076, 0.4479, 0.8252],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000],  
        [0.1000, 0.1000, 0.1000, 0.1000, 0.1000]]])
```



(a) Horizontal slices:  $\mathbf{X}_{i,:}$



(b) Lateral slices:  $\mathbf{X}_{:,j}$

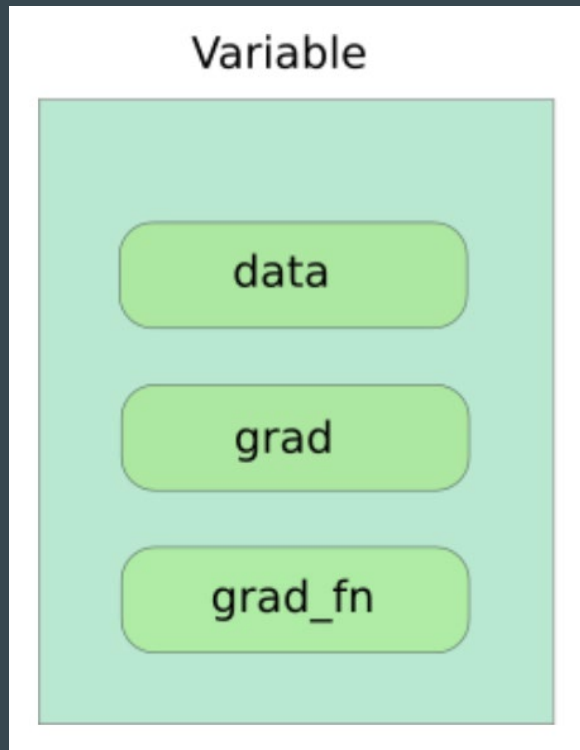


(c) Frontal slices:  $\mathbf{X}_{:,k}$  (or  $\mathbf{X}_k$ )

How do we store numbers? **Tensors.**  
Given tensors, how do we track their gradients?

# Variables

This is the class in PyTorch that corresponds to nodes in the computational graph.

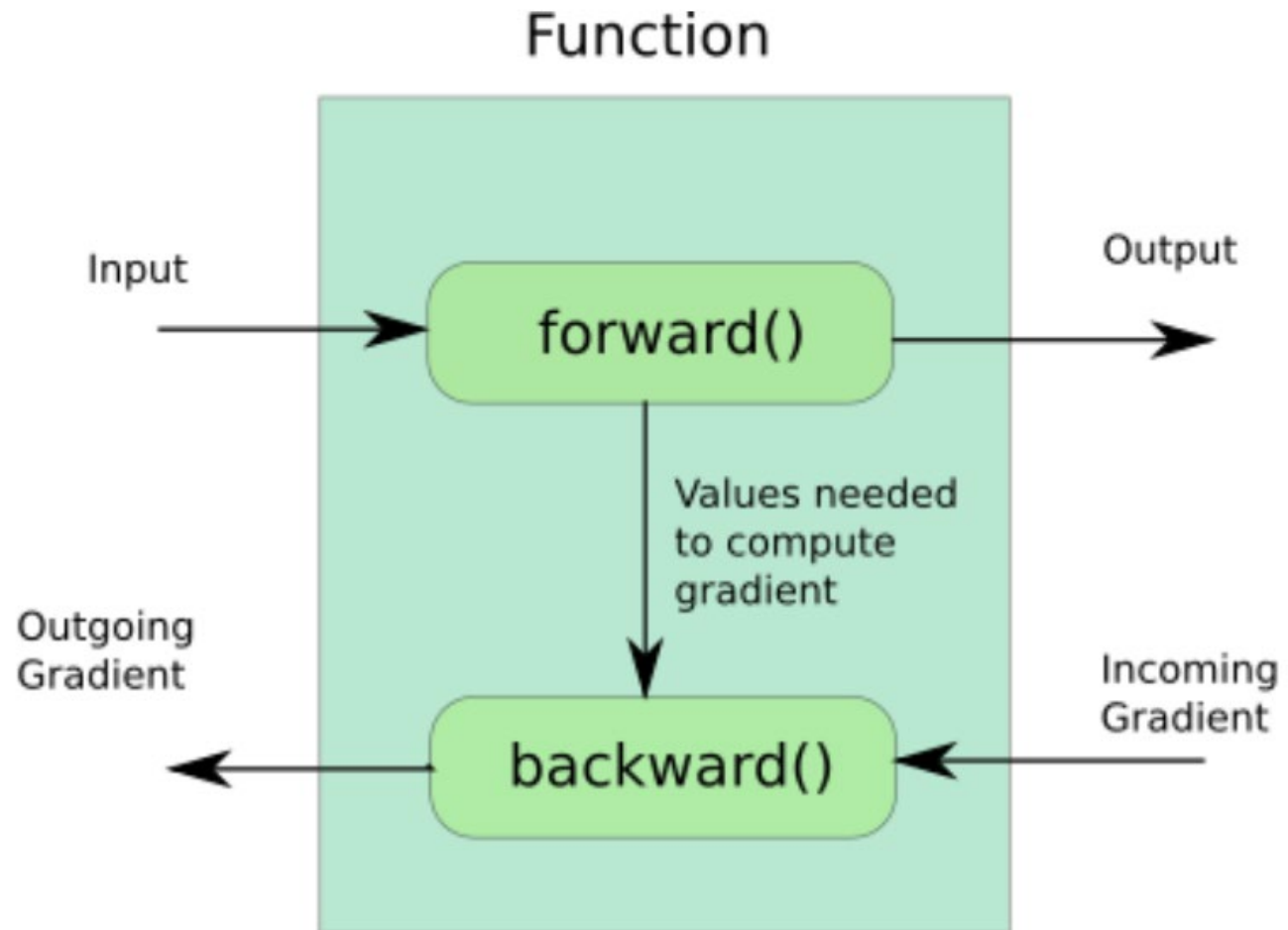


Tensor

Float

Function object

# Functions



How do we store numbers? **Tensors.**

Given tensors, how do we track their gradients? **Variables.**

Given tensors and their gradients, how do we actually  
update the parameter values during training?

How do we store numbers? **Tensors.**

Given tensors, how do we track their gradients? **Variables.**

Given tensors and their gradients, how do we actually  
update the parameter values during training?

# torch.nn.optim

- An optimizer is constructed with a model and hyperparameters.
- For each training example, the computational

E.g. `optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)`

```
for input, target in dataset:  
    optimizer.zero_grad()  
    output = model(input)  
    loss = loss_fn(output, target)  
    loss.backward()  
    optimizer.step()
```

How do we store numbers? **Tensors.**

Given tensors, how do we track their gradients? **Variables.**

Given tensors and their gradients, how do we actually update the parameter values during training? **Optimizers.**

How do we do all this on a GPU?



# How PyTorch hides the computational graph: Aka Pythonic syntactic sugar



Example:

PyTorch masks their special built-in addition function in the `__add__` method of the class `Variable`.

So `a+b` is really:

```
torch.autograd.Variable.__add__(a,b)
```

# CUDA integration

For a variable  $x$ , we can simply write:

```
x = x.cuda() # or
```

```
x = x.to(device) # if we have a previously defined device
```

To accelerate computations on  $x$  via GPU!

This casts  $x.data$  to an object of type `torch.cuda.FloatTensor()` and changes the magic methods associated with  $x$ , which are now written in Nvidia's CUDA API.

How do we store numbers? **Tensors.**

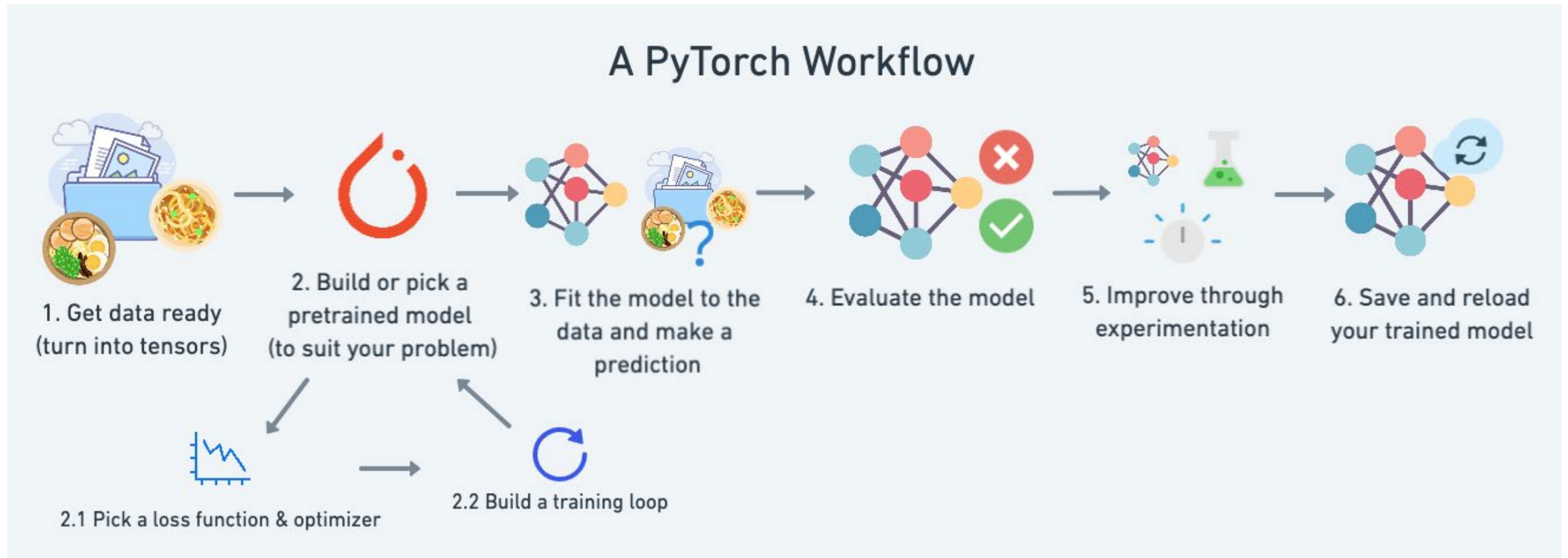
Given tensors, how do we track their gradients? **Variables.**

Given tensors and their gradients, how do we actually update the parameter values during training? **Optimizers.**

How do we do all this on a GPU? **CUDA bindings.**

## 2. PyTorch Deep Learning Model Life-Cycle

1. Prepare the Data.
2. Define the Model.
3. Train the Model.
4. Evaluate the Model.
5. Make Predictions.



# 1. Tensors (bit.ly/pytorchbasics)

## Tensors

Attributes of a tensor 't':

- `t = torch.randn(1)`

`requires_grad`- making a trainable parameter

- By default False
- Turn on:
  - `t.requires_grad_()` or
  - `t = torch.randn(1, requires_grad=True)`
- Accessing tensor value:
  - `t.data`
- Accessing tensor gradient
  - `t.grad`

`grad_fn`- history of operations for autograd

- `t.grad_fn`

```
1 import torch
2
3 N, D = 3, 4
4
5 x = torch.rand((N, D), requires_grad=True)
6 y = torch.rand((N, D), requires_grad=True)
7 z = torch.rand((N, D), requires_grad=True)
8
9 a = x * y
10 b = a + z
11 c = torch.sum(b)
12
13 c.backward()
14
15 print(c.grad_fn)
16 print(x.data)
17 print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
        [0.7797, 0.1519, 0.7513, 0.7269],
        [0.8572, 0.1165, 0.8596, 0.2636]])
tensor([[0.6855, 0.9696, 0.4295, 0.4961],
        [0.3849, 0.0825, 0.7400, 0.0036],
        [0.8104, 0.8741, 0.9729, 0.3821]])
```

## 2. Loading Data, Devices and CUDA

### Numpy arrays to PyTorch tensors

- `torch.from_numpy(x_train)`
- Returns a cpu tensor!

### PyTorch tensor to numpy

- `t.numpy()`

### Using GPU acceleration

- `t.to()`
- Sends to whatever device (cuda or cpu)

### Fallback to cpu if gpu is unavailable:

- `torch.cuda.is_available()`

### Check cpu/gpu tensor OR numpyarray ?

- `type(t)` or `t.type()` returns
  - `numpy.ndarray`
  - `torch.Tensor`
    - CPU - `torch.cpu.FloatTensor`
    - GPU - `torch.cuda.FloatTensor`

# 3. Autograd

- Automatic Differentiation Package
- Don't need to worry about partial differentiation, chain rule etc.
  - `backward()` does that
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - `tensor.grad_zero()`

```
# Create tensors.  
x = torch.tensor(1., requires_grad=True)  
w = torch.tensor(2., requires_grad=True)  
b = torch.tensor(3., requires_grad=True)
```

```
# Build a computational graph.  
y = w * x + b    # y = 2 * x + 3
```

```
# Compute gradients.  
y.backward()
```

```
# Print out the gradients.  
print(x.grad)    # x.grad = 2  
print(w.grad)    # w.grad = 1  
print(b.grad)    # b.grad = 1
```



# 4. Optimizer and Loss

## Optimizer

- Adam, SGD etc.
- An optimizer takes the parameters we want to update, the learning rate we want to use along with other hyper-parameters and performs the updates

## Loss

- Various predefined loss functions to choose from
- L1, MSE, Cross Entropy

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```



# Manual Linear Model

## Model

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- Two components
  - `__init__(self)` : it defines the parts that make up the model- in our case, two parameters, a and b
  - `forward(self, x)` : it performs the actual computation, that is, it outputs a prediction, given the input x

```
class ManualLinearRegression(nn.Module):  
    def __init__(self):  
        super().__init__()  
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter  
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))  
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))  
  
    def forward(self, x):  
        # Computes the outputs / predictions  
        return self.a + self.b * x
```

# PyTorch Example

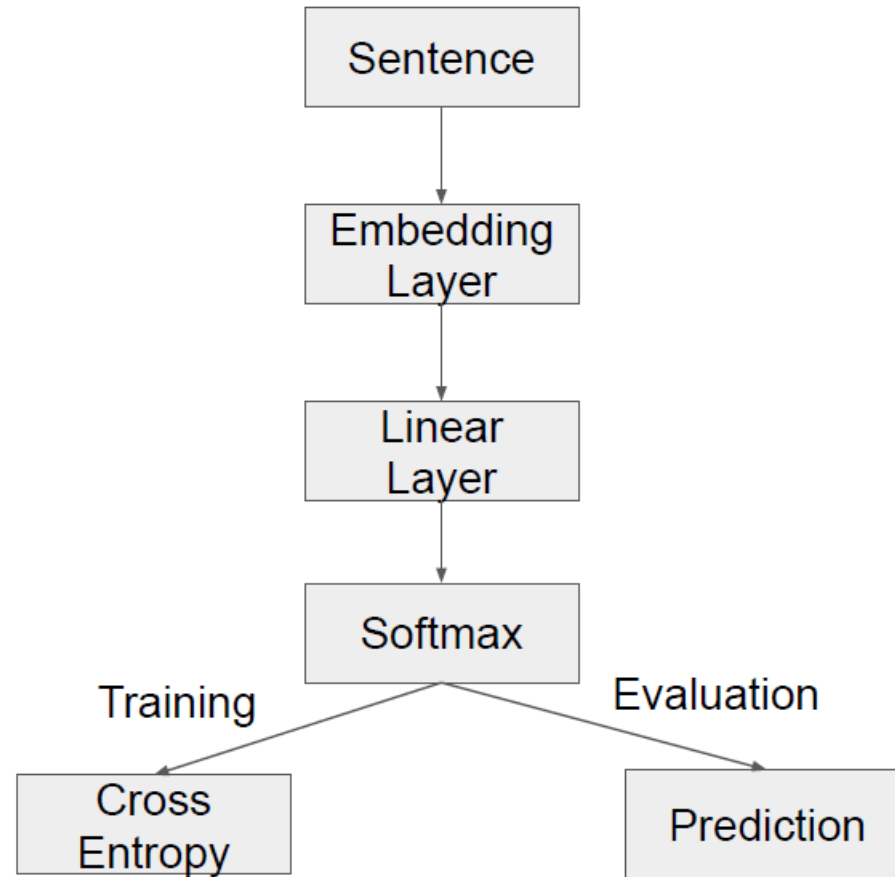
(neural bag-of-words (ngrams) text classification)

[bit.ly/pytorchexample](https://bit.ly/pytorchexample)

# Input String

```
ex_text_str = "MEMPHIS, Tenn. – Four days ago, Jon Rahm was \
    enduring the season's worst weather conditions on Sunday at The \
    Open on his way to a closing 75 at Royal Portrush, which \
    considering the wind and the rain was a respectable showing. \
    Thursday's first round at the WGC-FedEx St. Jude Invitational \
    was another story. With temperatures in the mid-80s and hardly any \
    wind, the Spaniard was 13 strokes better in a flawless round. \
    Thanks to his best putting performance on the PGA Tour, Rahm \
    finished with an 8-under 62 for a three-stroke lead, which \
    was even more impressive considering he'd never played the \
    front nine at TPC Southwind."
```

# Overview



# Design a Model

- Initialize modules.
- Use linear layer here.
- Can change it to RNN, CNN, Transformer etc.
- Randomly initialize parameters
- Forward pass

```
import torch.nn as nn
import torch.nn.functional as F
class TextSentiment(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=True)
        self.fc = nn.Linear(embed_dim, num_class)
        self.init_weights()

    def init_weights(self):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)
```

# Preprocess

- Build and preprocess dataset
- Build vocabulary

```
import torch
import torchtext
from torchtext.datasets import text_classification
NGRAMS = 2
import os
if not os.path.isdir('./.data'):
    os.mkdir('./.data')
train_dataset, test_dataset = text_classification.DATASETS['AG_NEWS'](
    root='./.data', ngrams=NGRAMS, vocab=None)
BATCH_SIZE = 16
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
VOCAB_SIZE = len(train_dataset.get_vocab())
EMBED_DIM = 32
NUN_CLASS = len(train_dataset.get_labels())
model = TextSentiment(VOCAB_SIZE, EMBED_DIM, NUN_CLASS).to(device)
```

# One example of dataset:

```
print(train_dataset[0])
```

```
(2, tensor([ 572, 564, 2, 2326, 49106, 150, 88, 3,
            1143, 14, 32, 15, 32, 16, 443749, 4,
            572, 499, 17, 10, 741769, 7, 468770, 4,
            52, 7019, 1050, 442, 2, 14341, 673, 141447,
            326092, 55044, 7887, 411, 9870, 628642, 43, 44,
            144, 145, 299709, 443750, 51274, 703, 14312, 23,
            1111134, 741770, 411508, 468771, 3779, 86384, 135944, 371666,
            4052]))
```

Create batch ( Used in SGD )

```
def generate_batch(batch):
    label = torch.tensor([entry[0] for entry in batch])
    text = [entry[1] for entry in batch]
    offsets = [0] + [len(entry) for entry in text]
    # torch.Tensor.cumsum returns the cumulative sum
    # of elements in the dimension dim.
    # torch.Tensor([1.0, 2.0, 3.0]).cumsum(dim=0)

    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text = torch.cat(text)
    return text, offsets, label
```



# Training each epoch

Iterable batches

Before each optimization, make previous gradients zeros

Forward pass to compute loss

Backforward propagation to compute gradients and update parameters

After each epoch, do learning rate decay ( optional )

```
from torch.utils.data import DataLoader

def train_func(sub_train_):

    # Train the model
    train_loss = 0
    train_acc = 0


    data = DataLoader(sub_train_, batch_size=BATCH_SIZE, shuffle=True,
                      collate_fn=generate_batch)
    for i, (text, offsets, cls) in enumerate(data):
        optimizer.zero_grad()
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        output = model(text, offsets)
        loss = criterion(output, cls)
        train_loss += loss.item()
        loss.backward()
        optimizer.step()
        train_acc += (output.argmax(1) == cls).sum().item()

    # Adjust the learning rate
    scheduler.step()

    return train_loss / len(sub_train_), train_acc / len(sub_train_)
```



# Test process

```
def test(data_):  
    loss = 0  
    acc = 0  
    data = DataLoader(data_, batch_size=BATCH_SIZE, collate_fn=generate_batch)  
    for text, offsets, cls in data:  
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)  
        with torch.no_grad():  
            output = model(text, offsets)  
            loss = criterion(output, cls)  
            loss += loss.item()  
            acc += (output.argmax(1) == cls).sum().item()  
  
    return loss / len(data_), acc / len(data_)
```

# The whole training process

- Use `CrossEntropyLoss()` as the criterion. The input is the output of the model. First do `logsoftmax`, then compute cross-entropy loss.
- Use SGD as optimizer.
- Use exponential decay to decrease learning rate

Print information to monitor the training process

```
import time
from torch.utils.data.dataset import random_split
N_EPOCHS = 5
min_valid_loss = float('inf')

criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=4.0)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.9)

train_len = int(len(train_dataset) * 0.95)
sub_train_, sub_valid_ = \
    random_split(train_dataset, [train_len, len(train_dataset) - train_len])

for epoch in range(N_EPOCHS):

    start_time = time.time()
    train_loss, train_acc = train_func(sub_train_)
    valid_loss, valid_acc = test(sub_valid_)

    secs = int(time.time() - start_time)
    mins = secs / 60
    secs = secs % 60

    print('Epoch: %d' % (epoch + 1), " | time in %d minutes, %d seconds" % (mins, secs))
    print(f'\tLoss: {train_loss:.4f}(train)\t|\tAcc: {train_acc * 100:.1f}%(train)')
    print(f'\tLoss: {valid_loss:.4f}(valid)\t|\tAcc: {valid_acc * 100:.1f}%(valid)')
```

# Evaluation with testdataset or random news

```
print('Checking the results of test dataset...')
test_loss, test_acc = test(test_dataset)
print(f'\tLoss: {test_loss:.4f}(test)\t|\tAcc: {test_acc * 100:.1f}%(test)')
```

```
import re
from torchtext.data.utils import ngrams_iterator
from torchtext.data.utils import get_tokenizer
```

```
ag_news_label = {1 : "World",
                 2 : "Sports",
                 3 : "Business",
                 4 : "Sci/Tec"}
```

```
def predict(text, model, vocab, ngrams):
    tokenizer = get_tokenizer("basic_english")
    with torch.no_grad():
        text = torch.tensor([vocab[token]
                             for token in ngrams_iterator(tokenizer(text), ngrams)])
        output = model(text, torch.tensor([0]))
    return output.argmax(1).item() + 1
```

```
ex_text_str = "MEMPHIS, Tenn. - Four days ago, Jon Rahm was \
enduring the season's worst weather conditions on Sunday at The \
Open on his way to a closing 75 at Royal Portrush, which \
considering the wind and the rain was a respectable showing. \
Thursday's first round at the WGC-FedEx St. Jude Invitational \
was another story. With temperatures in the mid-80s and hardly any \
wind, the Spaniard was 13 strokes better in a flawless round. \
Thanks to his best putting performance on the PGA Tour, Rahm \
finished with an 8-under 62 for a three-stroke lead, which \
was even more impressive considering he'd never played the \
front nine at TPC Southwind."
```

```
vocab = train_dataset.get_vocab()
model = model.to("cpu")
```

```
print("This is a %s news" %ag_news_label[predict(ex_text_str, model, vocab, 2)])
```

```
vocab = train_dataset.get_vocab()
model = model.to("cpu")
```

```
print("This is a %s news" %ag_news_label[predict(ex_text_str, model, vocab, 1)])
```

This is a Sports news

# PyTorch Neural Network Classification

[https://www.learnpytorch.io/02\\_pytorch\\_classification/](https://www.learnpytorch.io/02_pytorch_classification/)

# References

- <https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/>
- <https://pytorch.org/get-started/locally/>
- Colab :
  - [bit.ly/pytorchbasics](https://bit.ly/pytorchbasics)
  - [bit.ly/pytorchexample](https://bit.ly/pytorchexample)
- <https://github.com/pytorch/examples>
- <https://www.tutorialspoint.com/how-to-create-tensors-with-gradients-in-pytorch>