What is Pytorch?

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

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

Why Pytorch?

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

Checking PyTorch version

```
import torch

print(torch.__version__)
```

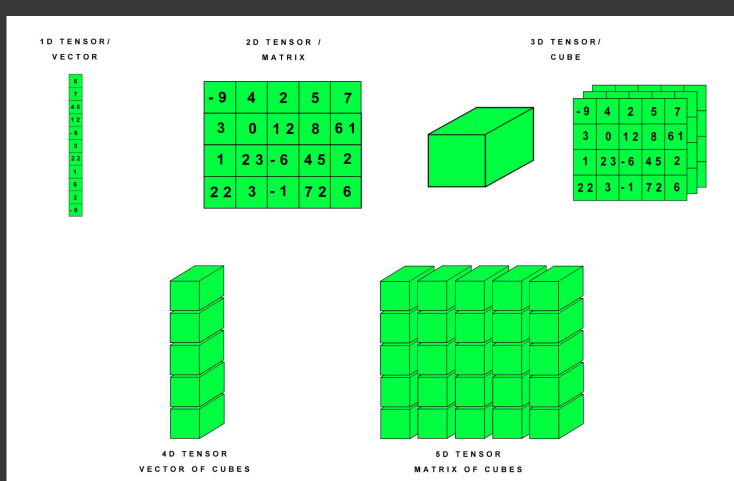
```
1.5.0+cu101
```

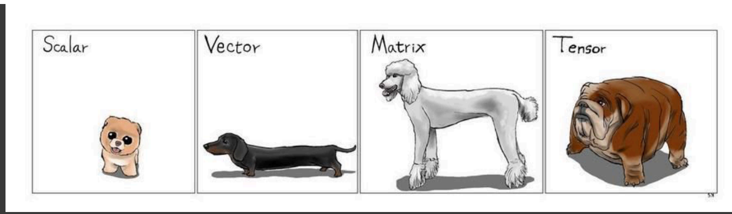
Introduction to Tensors

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

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

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





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

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

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

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

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

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

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

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

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

tensor_values = torch.from_numpy(np_values)

print (tensor_values)
```

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

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

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

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

```
# The contents of a tensor can be accessed and modified using Python’s indexing and slicing 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])
```

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

print(x)
print(y)

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

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

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

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

```
# Tensor Product & Transpose

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

print(mat1)
print(mat2)

print(torch.mm(mat1, mat2))

print(mat1.t())
```

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

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

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

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

x = torch.rand(5, 3)

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

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

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

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

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

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

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

Linear Regression

PyTorch Model Designing Steps

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

Step #1 : Design your model using class with Variables

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

import matplotlib.pyplot as plt

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

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

```
print(torch.__version__)

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

```
1.5.0+cu101
Tesla K80
```

Using GPU for the PyTorch Models

Remember always 2 things must be on GPU

- model
- tensors

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

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


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

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

Explanations:-

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

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

Parameters:

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

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

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

Explanations:-

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

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

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

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

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

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

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

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

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


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

```
Epoch: 0 | Loss: 0.4046969413757324
Epoch: 1 | Loss: 0.3751128613948822
Epoch: 2 | Loss: 0.34870368242263794
Epoch: 3 | Loss: 0.32503488659858704
Epoch: 4 | Loss: 0.30373701453208923
Epoch: 5 | Loss: 0.28449591994285583
Epoch: 6 | Loss: 0.2670438885688782
Epoch: 7 | Loss: 0.2511536478996277
Epoch: 8 | Loss: 0.23663079738616943
Epoch: 9 | Loss: 0.22330956161022186
Epoch: 10 | Loss: 0.2110479176044464
Epoch: 11 | Loss: 0.1997237652540207
Epoch: 12 | Loss: 0.1892329305410385
Epoch: 13 | Loss: 0.17948590219020844
Epoch: 14 | Loss: 0.17040419578552246
Epoch: 15 | Loss: 0.1619214415550232
Epoch: 16 | Loss: 0.15397928655147552
Epoch: 17 | Loss: 0.1465272456407547
Epoch: 18 | Loss: 0.13952045142650604
Epoch: 19 | Loss: 0.1329210102558136
Epoch: 20 | Loss: 0.12669487297534943
Epoch: 21 | Loss: 0.12081220746040344
Epoch: 22 | Loss: 0.11524614691734314
Epoch: 23 | Loss: 0.1099737286567688
Epoch: 24 | Loss: 0.10497380048036575
Epoch: 25 | Loss: 0.10022743046283722
Epoch: 26 | Loss: 0.09571778774261475
Epoch: 27 | Loss: 0.0914301723241806
Epoch: 28 | Loss: 0.08735045045614243
Epoch: 29 | Loss: 0.08346641808748245
Epoch: 30 | Loss: 0.079766184091568
Epoch: 31 | Loss: 0.07623954117298126
Epoch: 32 | Loss: 0.07287701219320297
Epoch: 33 | Loss: 0.06966972351074219
```

```
Epoch: 34 | Loss: 0.06660932302474976
Epoch: 35 | Loss: 0.0636879950761795
Epoch: 36 | Loss: 0.060898974537849426
Epoch: 37 | Loss: 0.05823547765612602
Epoch: 38 | Loss: 0.05569145083427429
Epoch: 39 | Loss: 0.053260963410139084
Epoch: 40 | Loss: 0.05093876272439957
Epoch: 41 | Loss: 0.04871952161192894
Epoch: 42 | Loss: 0.04659825190901756
Epoch: 43 | Loss: 0.044570740312337875
Epoch: 44 | Loss: 0.04263252019882202
Epoch: 45 | Loss: 0.04077941179275513
Epoch: 46 | Loss: 0.039007507264614105
Epoch: 47 | Loss: 0.0373133048415184
Epoch: 48 | Loss: 0.035693153738975525
Epoch: 49 | Loss: 0.034143973141908646
Prediction (after training) 7 13.88995361328125
```

Explanations:-

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

Simplified equation:-

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

Plot of predicted and actual values

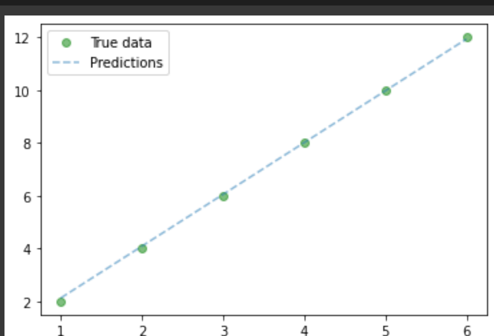
```
# Clear figure
plt.clf()

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

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

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

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



Saving Model to Directory

```
from google.colab import drive

drive.mount('/content/gdrive')

root_path = '/content/gdrive/My Drive/AUST Teaching Docs/AUST Fall 2019/Soft Computing/CSE 4238/Codes/04/'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.goog

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

Save Model

```
save_model = True

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

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

Load Model

```
load_model = True

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

Try Other Optimizers

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