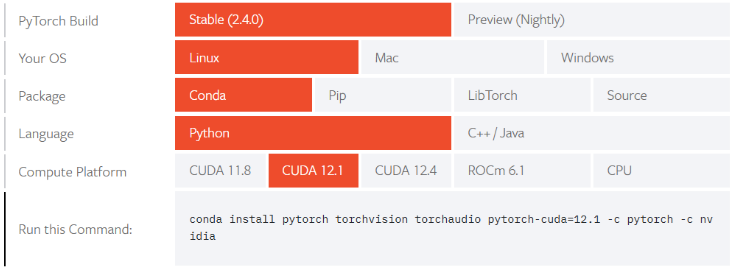
# AIL 722
# PyTorch Tutorial

Vaibhav Bihani

# Contents

# PyTorch : Introduction

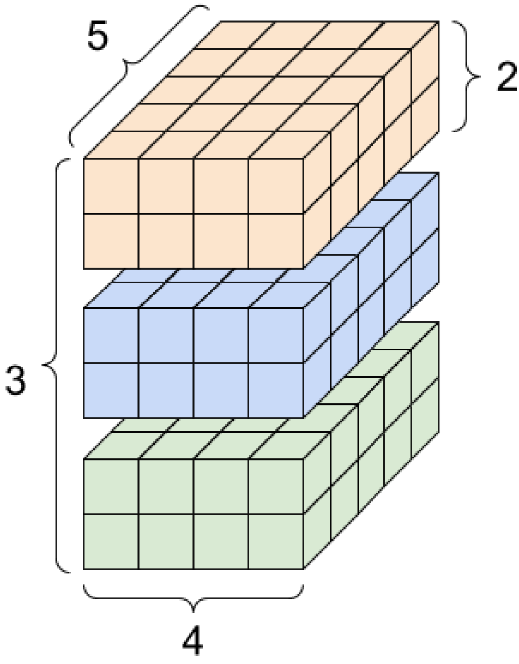
- Website: [https://pytorch.org/](https://pytorch.org/)Python based scientific computing package offering
  - Fast and efficient computation utilising GPUs/TPUs
  - Dynamic computational graph, providing Autograd capabilities
  - Numpy like easy to use API
  - Data Parallel and Model Parallel training
- Installation
  - You’ll need CUDA driver installed (Not required for CPU, Already available with HPC)
  - Installation Link (Select OS,Language(Python) and CUDA version accordingly)




  - Using Anaconda / Miniconda
    - Allows easy setup of environments
    - Automatic dependency resolution
    - Easy sharing of environments

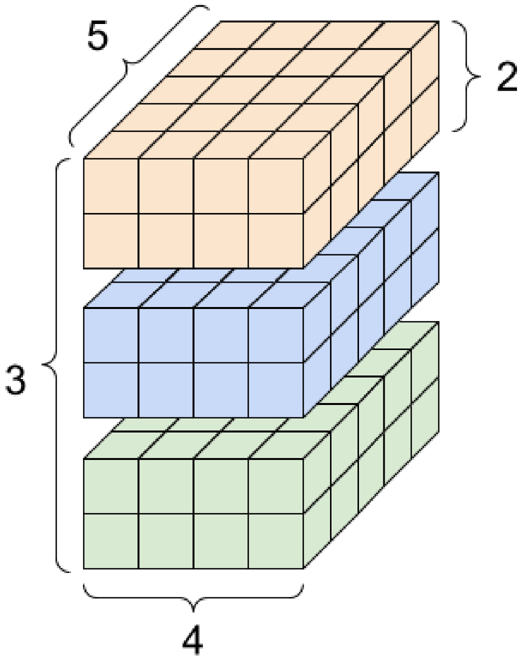
# Tensors : Basics

- N-Dimensional arrays, like Numpy arrays but can run on GPUs
  - t1 = torch.Tensor(3,4,2,5)
  - t1.size() # Returns torch.Size([3,4,2,5])
  - t2 = torch.Tensor([3.2, 4.3, 5.5])
  - t3 = torch.Tensor(np.array([[3.2], [4.3], [5.5]]))
  - t4 = torch.rand(4, 6)
  - t5 = t1 + t2 # addition
  - t6 = t2 * t3 # entry-wise product
  - t7 = t2 @ t3 # matrix multiplication
  - t8 = t1.view(2,12) # reshapes t1 to be 2 by 12
  - t8 = t1.view(2,-1) # same as above
  - t9 = t1[:, -1] # last column from the left
  - t2.add_(t3) #In_place operations have '_' at end



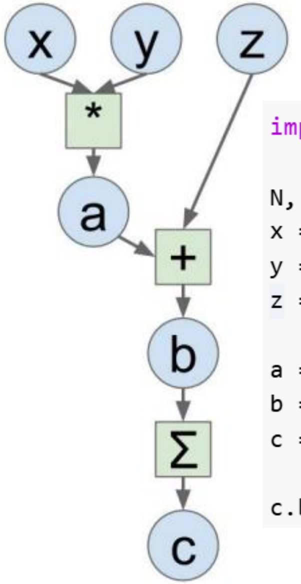
# Tensors : Data Type, Device, and require_grad

- Each Tensor has particular
  - Data Type : torch.Int /torch.Float32 etc.
  - Device : CPU (Default) /Cuda etc.
  - require_grad : True/False (See Autograd)
- Important to keep data type and device consistent in the tensor operations
- Use **.to()** function
  - A.to(torch.Float32)
  - A.to('cuda')
  - A.to(B) (Copies data type and device from B)
- **t.from_numpy()** and **t.numpy()**
- **t.data** : get stored values
- **t.grad** : get computed gradient (None if not computed)



# Autograd : Basics

- Automatic differentiation tool allows you to get gradients without worrying about chain-rule and partial derivatives
- Central to backpropagation-based neural network learning
  - **t1 = torch.randn((3,3), requires_grad = True)**
  - **t2.requires_grad = True**
  - **t2.requires_grad_(True)**
- Creates **Dynamic Computational Graph** of the operations performed on the tensors with requires_grad=True




```python
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()
```

Source: [PyTorch Basics: Understanding Autograd and Computation Graphs (paperspace.com)]

# Autograd : Optimizers

- **loss.backward()** #Performs backprop
- loss must be ‘scalar’
- Gradients get accumulated !
- Use **t.grad.zero_()**
- Tracking all parameters and gradients can be efficiently done using Optimizers

```python
import torch
import torch.optim as optim

# Initialize parameters a and b
a = torch.rand(1, requires_grad=True, dtype=torch.float, device='cuda')
b = torch.rand(1, requires_grad=True, dtype=torch.float, device='cuda')

# 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)
```

# Dataset and Data Loaders

- Decouples model from data
- Generally contains: [Data, Labels]
- Can define custom dataset class inheriting **Dataset** class
- Requires 3 components:
  - __init__(self)
  - __get_item__(self, index)
  - __len__(self)
- Many standard datasets are predefined
- Easy batching and Parallel loading

```python
from torch.utils.data import Dataset, TensorDataset

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

# Data Loaders : Example

- **shuffle and random_split()**

```
from torch.utils.data.dataset import random_split
from torch.utils.data import DataLoader

dataset=CustomDataset(x_train_tensor,y_train_tensor)

train_dataset, val_dataset, test_dataset = random_split(dataset, [60,20,20])

TrainLoader=DataLoader(train_dataset,batch_size=10,shuffle=True)
ValLoader=DataLoader(val_dataset,batch_size=10,shuffle=False)
TestLoader=DataLoader(test_dataset,batch_size=10,shuffle=False)
```

# Models : nn.Module Class

- **Base class** for all neural network modules.
- Requires two components
  - **__init__()**
    - Defines architecture and layers
  - **forward(*inputs)**
    - Defines the computation logic for forward pass
    - Later called using **Model_Instance(input)**
- Can have nested submodules

```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

# Neural Network Example

# Models : Saving and Loading

- **Checkpoint:**
  - Allows resuming training
  - torch.save({'epoch': epoch, 'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(), 'loss': loss,...},
    PATH)
- **Loading:**
  - model = ModelClass( *args)
  - optimizer = OptimizerClass( *args)
  - checkpoint = torch.load( PATH)
  - model.load_state_dict( checkpoint['model_state_dict'])
  - optimizer.load_state_dict( checkpoint['optimizer_state_dict'])
  - epoch = checkpoint['epoch']
  - loss = checkpoint['loss']
  - model.eval() or  model.train()

# PyTorch Lightning

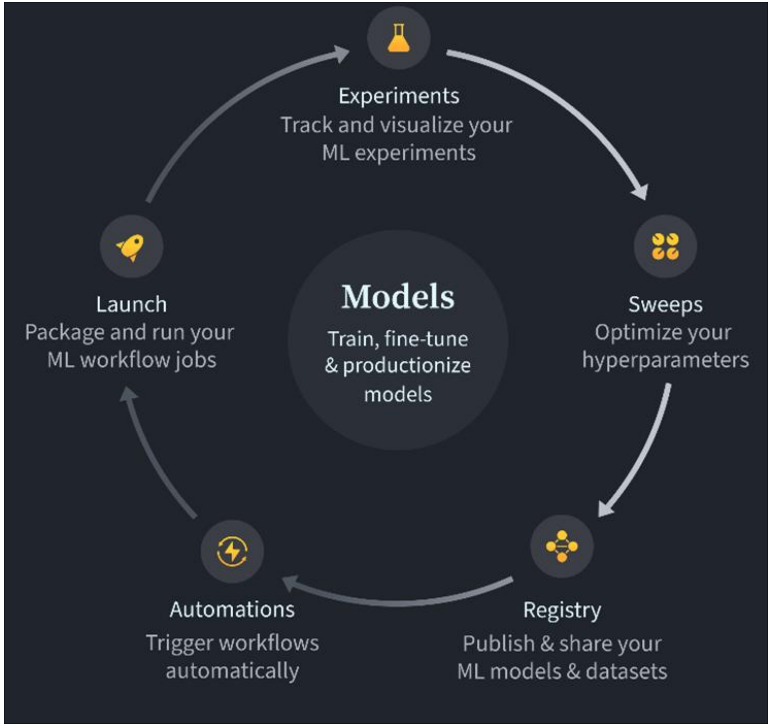
- Wrapper for PyTorch
- Makes the code hardware independent
- Provides highly flexible API for ML development
    - Inbuilt Multi-GPU, Multi-Node parallel training
    - Training Schedulers, Callbacks for Early Stopping etc.
    - 16 bit precision training
    - Metrics and Loggers
    - Profilers for debugging
- 15 Min Tutorial : Lightning in 15 minutes — PyTorch Lightning 2.4.0 documentation
- Can be integrated directly with Tensorboard/WandB etc.

# Weights & Biases

- AI Development Platform

- Online/Offline Tracking and Visualisation of ML models training

- Hyperparameter Tuning [Important in RL Models]



# Thank You