PyTorch - Model Building

PyTorch

- PyTorch is the most popular research deep learning framework. It allows us to write fast deep learning code in Python.

- It allows us to write state of the art deep learning code accelerated by GPUs with Python.

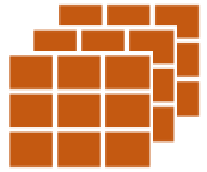
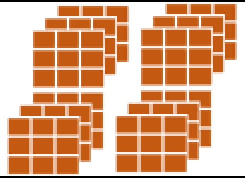
- It enables us access to many prebuilt deep learning models from Torch Hub.

- Stacks for everything : Pre-process data, model data, deploy model.

- Initially designed and used in-house by Facebook/Meta, now open source used by OpenAI, Microsoft, Tesla and many other companies.

Tensor

- Defining a Tensor
- Manipulating

Mathematical Name	Example	Tensor Dimension/Order	Description
Scalar	X	0	Simply a magnitude/Number
Vector	$[X_1, X_2, X_3]$	1	1-D Array/Matrix Having both Magnitude and direction. Three entries are magnitudes in three coordinate axes.
Matrix	$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}$	2	2-D Array/Matrix Table like Structure
3D Matrix/3-Tensor		3	3-D Array/ Matrix Cube or Parallelepiped liked structure Multiple 2D arrays stacked one after another.
Higher Oder Tensor		n	Multiple 3D or higher order matrix stacked one after another. Difficult to visualize.

Autograd stands for **automatic differentiation**.

1. It's PyTorch's **automatic differentiation engine** that powers neural network training.
2. **Automatic differentiation** is the process of computing **gradients** (partial derivatives) of a function with respect to its input variables.
3. These gradients are essential for optimizing model parameters during training.
4. Autograd automatically computes and stores gradients for each model parameter in the **.grad** attribute

How Autograd Works in NN

1. **Neural networks (NNs)** consist of nested functions executed on input data.
2. These functions are defined by **parameters** (weights and biases), stored in **tensors**.
3. Training an NN involves two steps:
    1. **Forward Propagation**: The NN makes its best guess about the output by running input data through its functions.
    2. **Backward Propagation (Backprop)**: The NN adjusts its parameters based on the error in its guess.
        1. It traverses backward from the output, collecting derivatives (gradients) of the error with respect to the parameters.
        2. These gradients guide parameter updates using techniques like **gradient descent**.

Low Level Implementation

N=5000, D_in=100, D_out =1 | H=10

Visualizing tabular format:

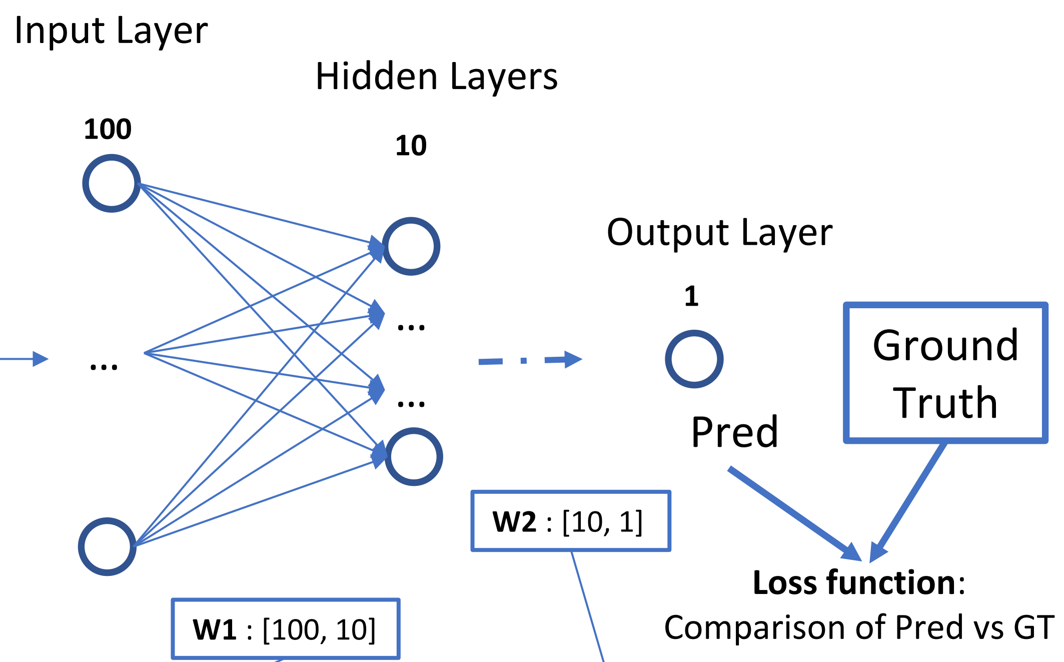
Number of sample N = 5000

X			Y
f1	.....	f100	Target/Output
3	.....	90	23
2	.....	143	8
...	.....	....	....
---	.....	....	....
---	.....	...	...

100 features

$$Z^{[1]} = W^{[1]} \cdot A^{[0]} + b^{[1]}$$

$$A^{[1]} = ReLu(Z^{[1]})$$



Shape tracking

A0: [N , 100] @ **W1** : [100, 10] → **A1**: [N, 10]

A1: [N , 10] @ **W2** : [10,1] → **A2:** [N,1]

Note: Bias is ignored for sake of simplicity.

Linear Regression model class

Low Level Implementation

```
from torch import nn

class LinearRegressionModel(nn.Module):

  def __init__(self):
    super().__init__()
    self.weights = nn.Parameter(torch.randn(1, dtype=torch.float), requires_grad=True)
    self.bias = nn.Parameter(torch.randn(1, dtype=torch.float), requires_grad=True)

   # Forward defines the computation in the model
  def forward(self,  x: torch.Tensor) -> torch.Tensor:
    return self.weights * x + self.bias
```

Linear Layer

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)
```

NN for Regression

$$Z^{[1]} = W^{[1]} \cdot A^{[0]} + b^{[1]}$$
$$A^{[1]} = ReLu(Z^{[1]})$$

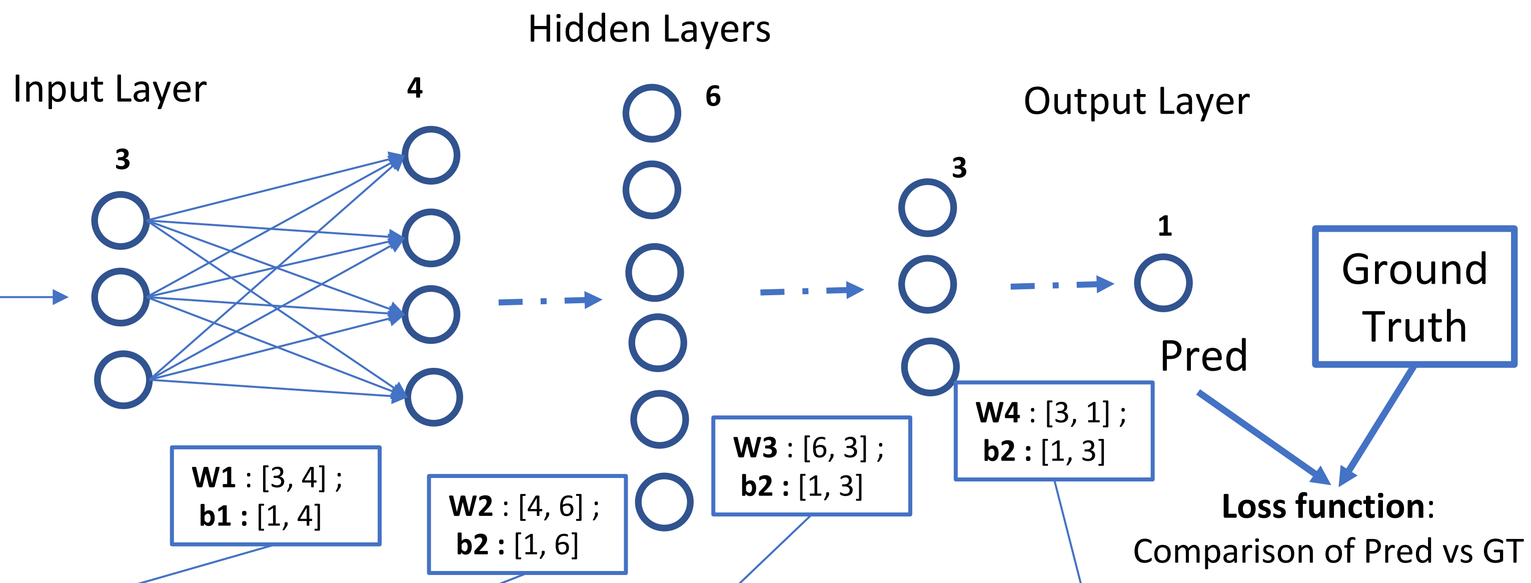
Visualizing data in tabular format:

Number of sample Ns = 5

X			Y
f1	f2	f3	Target/Output
3	2000	90	23
2	800	143	8
2	850	167	9
1	550	267	9
4	2000	396	25

Three features

NN- Architecture



Loss function: Comparison of Pred vs GT

Shape tracking

A0: [Ns , 3] @ **W1** : [3, 4] + **b1 :** [1,4] □ **A1**: [Ns, 4]

A1: [Ns , 4] @ **W2** : [4,6] + **b2 :** [1, 6] □ **A2:** [Ns,6]

A2: [Ns , 6] @ **W3** : [6,3] + **b3 :** [1, 3] □ **A3:** [Ns,3]

A3: [Ns , 3] @ **W4** : [3,1] + **b4 :** [1, 1] □ **A4:** [Ns,1]

- The **'torch. nn'** module is essential for designing any neural network in PyTorch. This class can be used to implement a layer like a fully connected layer, a convolutional layer, a pooling layer, an activation function, and also an entire neural network by instantiating a torch. **nn.Module** object.

- All models in PyTorch inherit from the subclass **nn.Module**, which has useful methods like parameters(), __call__() and others.

The nn.Module class has two methods that you have to override.

1. **__init__** function. This function is invoked when you create an instance of the nn.Module. Here you will define the various parameters of a layer such as filters, kernel size for a convolutional layer, dropout probability for the dropout layer.

2. **forward function**. This is where you define how your output is computed. This function doesn't need to be explicitly called, and can be run by just calling the nn.Module instance like a function with the input as it's argument.

- What exactly is a "layer"?

It is essentially a step in the neural network computation. We can also think of the ReLU activation as a "layer". However, there are no tunable parameters associated with the ReLU activation function. We don't need to keep track of "states" associated with the ReLU acitvation, so it is not initalized as a "layer" in the __init__ function.)

- The **__init__** method is where we typically define the attributes of a class. In our case, all the "sub-components" of our model should be defined here, along with any other setting that we wish to save.

- The **forward** method is called when we use the neural network to make a prediction. Another term for "making a prediction" is **running the forward pass**, because information flows *forward* from the input through the hidden layers to the output. When we compute parameter updates, we run the **backward pass** by calling the function loss.backward(). During the backward pass, information about parameter changes flows *backwards*, from the output through the hidden layers to the input. The forward method is called from the __call__ function of nn.Module, so that when we run model(input), the forward method is called.

PyTorch model building essentials

PyTorch Module	What does it do?
torch.nn	Contains all of the building blocks for computational graphs (essentially a series of computations executed in a particular way).
torch.nn.Parameter	Stores tensors that can be used with nn.Module. If requires_grad=True gradients (used for updating model parameters via **gradient descent**) are calculated automatically, this is often referred to as "autograd".
torch.nn.Module	The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass nn.Module. Requires a forward() method be implemented.
torch.optim	Contains various optimization algorithms (these tell the model parameters stored in nn.Parameter how to best change to improve gradient descent and in turn reduce the loss).
def forward()	All nn.Module subclasses require a forward() method, this defines the computation that will take place on the data passed to the particular nn.Module (e.g. the linear regression formula above).
torch.utils.data.Dataset	Represents a map between key(label) and sample(features pairs of the data)
torch.utils.data.DataLoader	Creates a Python iterable over a torch Dataset, allows to iterate over the data

Key Components:

- torch.nn provides several essential classes and functions:
  - nn.Module: The base class for all neural network modules.
  - nn.Linear: A linear transformation (fully connected layer) that computes output from input using weights and biases.
  - nn.ReLU: The rectified linear unit activation function.
  - nn.CrossEntropyLoss: A loss function commonly used for classification tasks.
  - nn.Sequential: A container for organizing layers sequentially.
  - nn.Parameter: Explicitly specifies tensors as learnable parameters for the model.

Example Usage:

- Suppose you want to create a simple linear regression model using PyTorch.
- In the equation **W * X + b**, both **W** and **b** need to be treated as learnable parameters.
- You can achieve this by defining them as nn.Parameter.
- nn.Parameter explicitly designates tensors as learnable parameters.
- During training, these tensors are updated to minimize the loss function.
- It ensures that the model learns from the data and adapts its parameters.

In PyTorch, **nn.functional** is a module that provides a collection of **stateless** functions for applying various operations on tensors. Unlike the nn.Module class, which represents neural network layers and maintains internal state (such as weights), the functions in nn.functional do not have any internal state. Here are some key points about nn.functional:

Statelessness:

- The functions in nn.functional operate directly on input tensors without maintaining any internal parameters or state.
- They are useful for operations that don't require learning or updating weights, such as activation functions, pooling, and distance calculations.

1.Common Use Cases:

- **Activation Functions**: Functions like ReLU (F.relu), sigmoid (F.sigmoid), and tanh (F.tanh) are commonly used for introducing non-linearity in neural networks.
- **Pooling**: nn.functional provides functions for average pooling (F.avg_pool2d, F.avg_pool3d) and max pooling (F.max_pool2d, F.max_pool3d).
- **Attention Mechanisms**: For tasks like scaled dot product attention, you can use F.scaled_dot_product_attention.

2.Comparison with nn.Module:

- While nn.Module provides an object-oriented interface for neural network layers, nn.functional offers a functional (non-object-oriented) approach.
- Use nn.Module when you need to learn and update weights (e.g., in training), and use nn.functional for stateless operations (e.g., during inference).

Neural Networks as Sequences

nn.Sequential

```
import torch
import torch.nn as nn

# Define the neural network architecture
model = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 10),
    nn.LogSoftmax(dim=1)
)
```

Neural Networks as Classes

nn.Module

```
import torch
import torch.nn as nn

class CustomNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, 10)
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.log_softmax(x)
        return x

# Create an instance of the CustomNetwork
model = CustomNetwork()
```

In **PyTorch**, the nn.Module is a fundamental building block for creating neural networks.

- The nn.Module class serves as the base class for all neural network modules in PyTorch.
- While creating a NN model class, it should be a subclass of nn.Module.
- Neural networks are composed of layers or modules that perform operations on data. Each of these layers or modules is an instance of nn.Module.
- By inheriting from nn.Module, we gain access to useful methods and attributes for managing the neural network.

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

class MyModel(nn.Module):
  def __init__(self):
    super().__init__() # Define layers (submodules)
    self.conv1 = nn.Conv2d(1, 20, 5)
    self.conv2 = nn.Conv2d(20, 20, 5)

  def forward(self, x): # Define the forward pass
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))

# Usage:
model = MyModel()
```

In this example:

- MyModel inherits from nn.Module.
- We define two convolutional layers (conv1 and conv2) as submodules of our model.
- The forward method specifies how data flows through the layers during inference.
- Remember that submodules assigned in this way are registered and their parameters are converted when you call methods like .parameters(), .to(), etc. The super().__init__() call in the constructor ensures proper initialization.

STEPS : Image data

1. Read/Load the data
2. Visualization
3. Transformation : Reshape and augmentations
4. Loading the data in PyTorch dataset format to make it compatible with PyTorch model

```
# Option A.  ImageFolder to create dataset
from torchvision import datasets
train_data = datasets.ImageFolder(root=train_dir,
                                  transform=data_transform, # a transform for the data
                                  target_transform=None) # a transform for the label/target

test_data = datasets.ImageFolder(root=test_dir,
                                 transform=data_transform)

train_data, test_data
```

```
# Option B.  Build our own custom data class equivalent to built in ImageFolder
```

5. Convert data into data loaders for batch processing
6. Define CNN Architecture, Loss criteria, Optimizer
7. Train –Train Loop
8. Evaluation

CNN Architecture

FashionMNIST_CNN_.ipynb

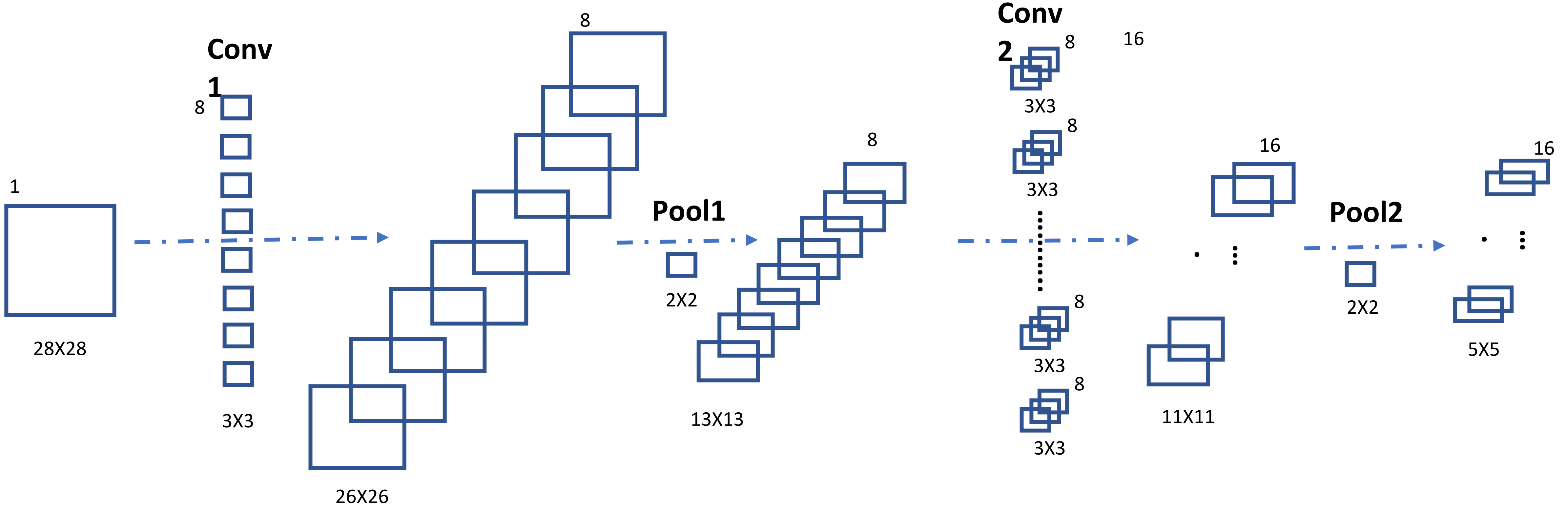
$$\begin{tikzcd}
\text{Input: Image} \arrow[r] & \text{Conv\_L1 : Output of Conv\_L1} \arrow[r] & \text{Pool\_L1 : Output of Pool\_L1} \arrow[r] & \text{Conv\_L2 : Output of Conv\_L2} \arrow[r] & \text{Pool\_L2 : Output of Pool\_L2} \arrow[r] \arrow[d, to=FC, bend left] & {} \\
& & & \text{FC1 FC2 FC3 FC4 Softmax} & &
\end{tikzcd}$$

FC1 FC2 FC3 FC4 Softmax

CNN Example 1:

P1: $(3 \times 3 \times 1 + 1) \times 8 = 80$

P2: $(3 \times 3 \times 8 + 1) \times 16 = 1168$



Input: Image	Conv1	Output of Conv1	Pool1	Output of Pool1	Conv2	Output of Conv2	Pool2	Output of Pool2

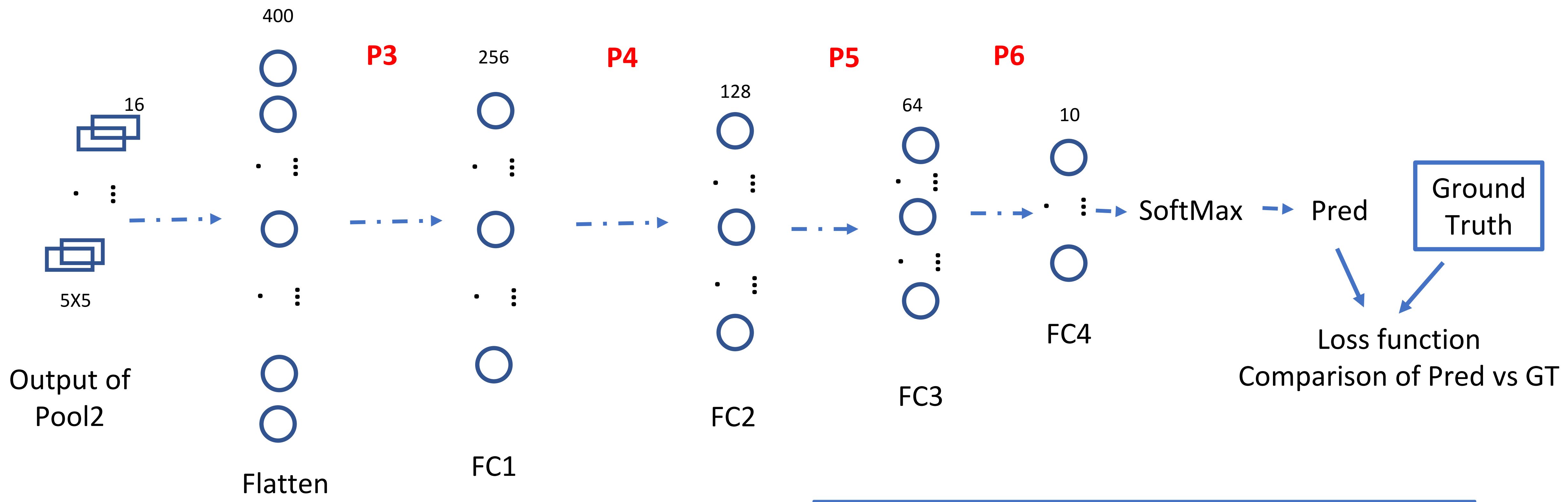
$$\text{Output shape after Conv} = \frac{n+2p-f}{s} + 1$$

$$\text{Output shape after Pool} = \frac{n-f}{s} + 1$$

$$\text{Calculating Parameters} = \left(f^l \times f^l \times n_c^{l-1} + 1\right) \times n_f^l$$

P3 : [256, 400] + b(256) =$((256 \times 400) + 256)$ =102,656

P4 : [128, 256] + b(128) =$((128 \times 256) + 128)$ =32,896



P5 : [64, 128] + b(64) =$((64 \times 128) + 64) = 8{,}256$

P6 : [10, 64] + b(10) =$((10 \times 64) + 10) = 650$

CUDA /GPU

A GPU is a graphics processing unit which is essentially very fast at crunching numbers. Originally designed for video games.

The beautiful thing about PyTorch is that it enables you to leverage a GPU through an interface called CUDA. The CUDA is a parallel computing platform and application programming interface, which is an API that allows software to use certain types of graphics processing units for general purpose computing.

So PyTorch leverages CUDA to enable you to run your machine learning code on Nvidia GPUs. Now there is also an ability to run your PyTorch code on TPU, which is a tensor processing unit. However, GPUs are far more popular when running various types of PyTorch code.

Thanks!