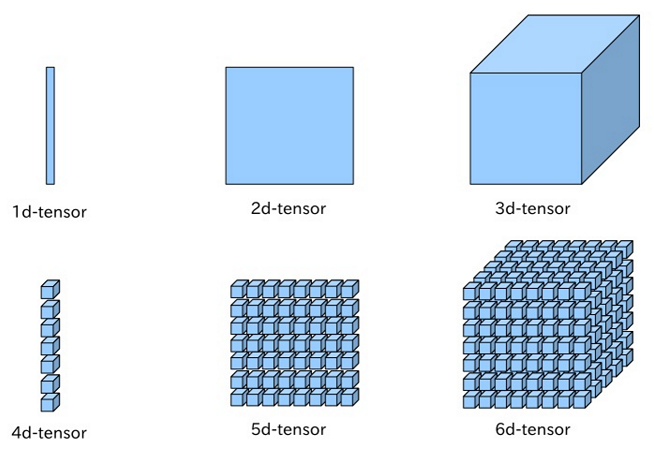
***Question 1 :***

PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook’s AI Research lab (FAIR). It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

A number of pieces of Deep Learning software are built on top of PyTorch, including Uber’s Pyro, HuggingFace’s Transformers, and Catalyst. It is also one of the preferred deep learning research platforms built to provide maximum flexibility and speed. It is known for providing two of the most high-level features; namely, tensor computations with strong GPU acceleration support and building deep neural networks on a tape-based autograd systems.

Python developers may create and train machine learning models using PyTorch, an open-source machine learning framework created by Facebook's AI research team. Here are some PyTorch fundamentals:

**Tensors:** Tensors are PyTorch's fundamental building pieces. Similar to NumPy arrays in terms of having multiple dimensions, but with additional capability for GPU acceleration.



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.

**1. torch.abs()**

*torch.abs(input, out=None) → Tensor*

This function computes the element-wise absolute value of the given input tensor i.e. it returns all positive value of input.

out = | input |

*Parameters*

* input (Tensor) — the input tensor.
* out (Tensor, optional) — the output tensor.

Example :

**In[] : torch.abs(torch.tensor([-10, -22, 3]))**Out[] : tensor([10, 22, 3])

**2. torch.add()**

*torch.add(input, other, out=None)*

This function adds the scalar other to each element of the input and returns a new resulting tensor.

out = input + other

*Parameters*

* input (Tensor) — the input tensor.
* value (Number) — the number to be added to each element of input

Example :

**In[] : a = torch.randn(3)  
 a**  
Out[] : tensor([-2.5680, -0.8406, 0.2862])**In[] : torch.add(a, 20)**Out[] : tensor([17.4320, 19.1594, 20.2862])

**3. torch.sub()**

*torch.sub(input,other, out=None) → Tensor*

This function Subtract the scalar other to each element of the input and returns a new resulting tensor.

out = input — other

*Parameters*

* input (Tensor) — the input tensor.
* other (Number) — the number to be subtracted to each element of input

Example :

**In[] : a = torch.randn(3)  
 a**  
Out[] : tensor([ 1.7225, 0.5430, -1.1199])**In[] : torch.sub(a,4)**Out[] : tensor([-2.2775, -3.4570, -5.1199])

**4. torch.div()**

*torch.div(input, other, out=None) → Tensor*

This function divides each element of the input with the scalar other and returns a new resulting tensor.

out = input / other

*Parameters*

* input (Tensor) — the input tensor.
* other (Number) — the number to be divided to each element of input

Example :

**In[] : a = torch.randn(5)  
 a**  
Out[] : tensor([ 0.1227, -0.0442, 2.6160, 1.6794, 1.5719])**In[] : torch.div(a, 0.2)**Out[] : tensor([ 0.6136, -0.2208, 13.0801, 8.3970, 7.8596])

**5. torch.mul()**

*torch.mul(input, other, out=None)*

This function multiplies each element of the input input with the scalar other and returns a new resulting tensor.

out = input \* other

*Parameters*

* input (Tensor) — the input tensor.
* other (Number) — the number to be multiplied to each element of input

Example :

**In[] : a = torch.randn(3)  
 a**  
Out[] : tensor([ 0.2845, -1.0132, 0.2563])**In[] : torch.mul(a, 5)**Out[] : tensor([ 1.4227, -5.0661, 1.2814])

**6. torch.neg()**

*torch.neg(input, out=None) → Tensor*

This function returns a new tensor with the negative of the elements of input.

out = −1 × input

*Parameters*

* input (Tensor) — the input tensor.
* out (Tensor, optional) — the output tensor.

Example :

**In[] : a = torch.randn(3)  
 a**  
Out[] : tensor([ 1.0753, 0.5619, -2.2713])**In[] : torch.neg(a)**Out[] : tensor([-1.0753, -0.5619, 2.2713])

**7. torch.pow()**

*torch.pow(input, exponent, out=None) → Tensor*

It takes the power of each element in input with exponent and returns a tensor with the result.

exponent can be either a single float number or a Tensor with the same number of elements as input.

out = input^exponent

*Parameters*

* input (Tensor) — the input tensor.
* exponent (float or tensor) — the exponent value
* out (Tensor, optional) — the output tensor.

Example :

**In[] : a = torch.arange(1., 6.)  
 a**  
Out[] : tensor([1., 2., 3., 4., 5.])**In[] : torch.pow(a, 2)**Out[] : tensor([ 1., 4., 9., 16., 25.])

**8. torch.reciprocal()**

*torch.reciprocal(input, out=None) → Tensor*

This function returns a new tensor with the reciprocal of the elements of input.

out = 1 / input

*Parameters*

* input (Tensor) — the input tensor.
* out (Tensor, optional) — the output tensor.

Example :

**In[] : a = torch.randn(5)  
 a**  
Out[] : tensor([ 0.2661, -1.2168, 1.6755, 0.5949, -0.2095])**In[] : torch.reciprocal(a)**Out[] : tensor([ 3.7579, -0.8218, 0.5968, 1.6811, -4.7739])

**9. torch.remainder()**

*torch.remainder(input, other, out=None) → Tensor*

This function computes the element-wise remainder of division. The divisor and dividend may contain both for integer and floating point numbers. The remainder has the same sign as the divisor.

out = input % other

*Parameters*

* input (Tensor) — the dividend
* other (Tensor or float) — the divisor that may be either a number or a Tensor of the same shape as the dividend
* out (Tensor, optional) — the output tensor.

Example :

**In[] : torch.remainder(torch.tensor([-30., -12, -10, 11, 22, 3]), 2)**  
Out[] : tensor([0., 0., 0., 1., 0., 1.])

**10. torch.square()**

*torch.square(input, out=None) → Tensor*

This function returns a new tensor with the square of the elements of input.

out = input \* input

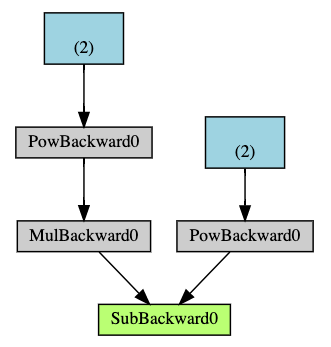
*Parameters*

* input (Tensor) — the input tensor.
* out (Tensor, optional) — the output tensor.

Example :

**In[] : a = torch.randn(4)**  
 **a**  
Out[] : tensor([ 1.1735, -0.9121, 0.0543, -0.5389])**In[] : torch.square(a)**  
Out[] : tensor([1.3772, 0.8320, 0.0029, 0.2905])

**Autograd**: PyTorch employs automatic differentiation to compute tensor gradients with autograd. This makes it simple for developers to calculate the gradients of challenging functions.



import torch

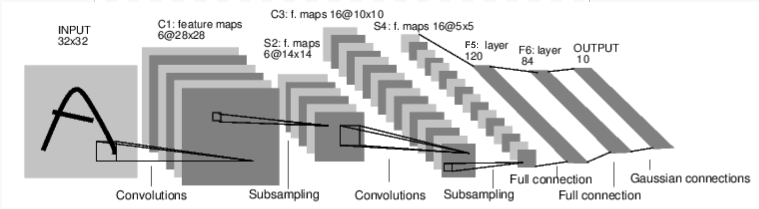
from torchvision.models import resnet18, ResNet18\_Weights

model = resnet18(weights=ResNet18\_Weights.DEFAULT)

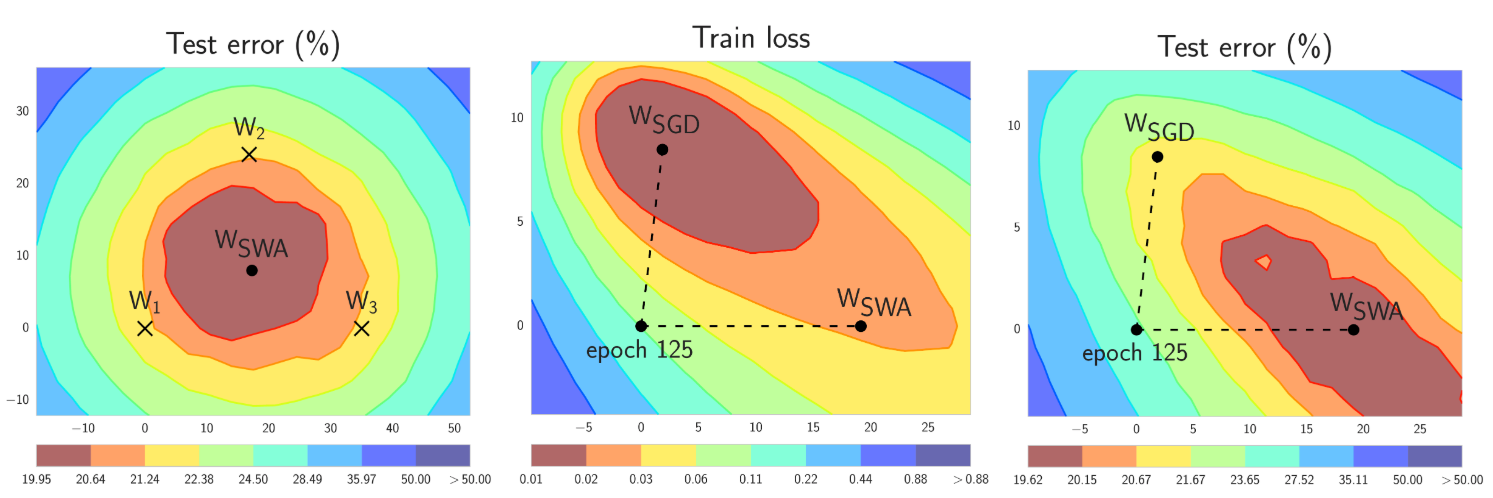
data = torch.rand(1, 3, 64, 64)

labels = torch.rand(1, 1000)

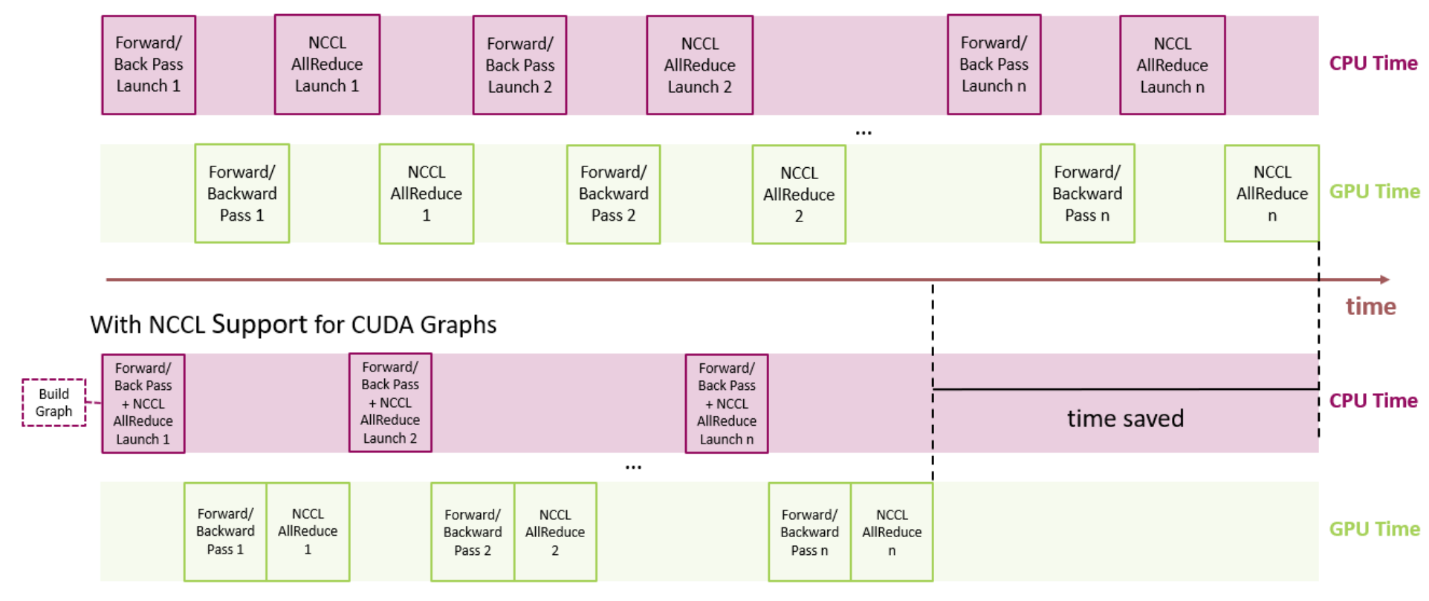
**Neutral Network**: Building neural networks is simple thanks to a collection of modules that PyTorch offers. Layers like linear layers, convolutional layers, and activation functions are among these modules.



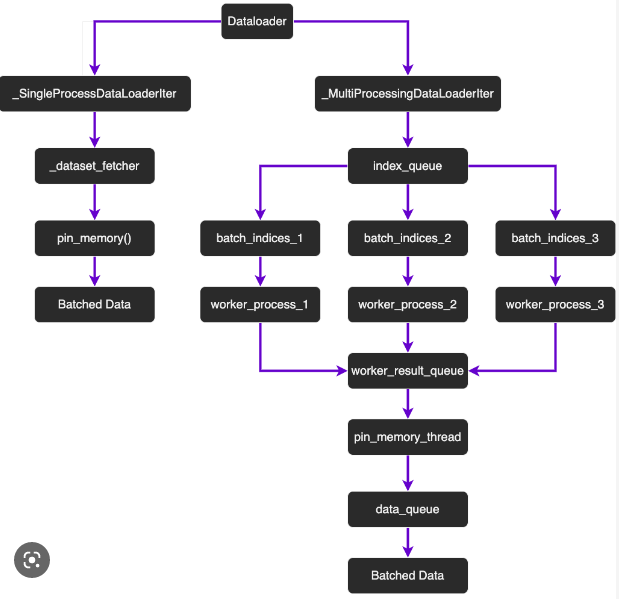
**Optimization**: Stochastic gradient descent (SGD), Adam, and Adagrad are just a few of the optimization algorithms that PyTorch offers to developers to aid in the training of their neural networks.



**CUDA**: PyTorch can run on GPUs thanks to CUDA, which greatly accelerates computing for deep learning tasks. GPU acceleration in PyTorch is made possible using CUDA, a parallel computing framework created by NVIDIA.



**DataLoader**: PyTorch's DataLoader class makes it simple to load and prepare data for neural network training. Furthermore capable of handling data batching and shuffle, the DataLoader class.



***Question 2 :***

A potent platform for creating and refining machine learning models is PyTorch. The fundamental steps for using PyTorch to create a machine learning model are as follows:

**Data Preparation**: Gathering the necessary data is the initial stage in creating a machine learning model. This entails preparing the data, dividing the data into training, validation, and test sets, as well as loading the data into PyTorch tensors

**Model Definition**: The definition of the model architecture is the next stage. This is done in PyTorch by developing a class that derives from the torch. defining the layers and parameters of the model in the nn.Module class.

**Training**: Using an optimizer and a loss function, the model can be trained on the training data once it has been defined. Many optimizers and loss functions, such as cross-entropy loss and stochastic gradient descent (SGD), are available in PyTorch.

**Validation**: The performance of the model can be assessed on the validation set at the conclusion of each training epoch. This aids in tracking the model's development and identifying overfitting.

**Testing**: After the model has been trained, it may be assessed on the test set to gauge how well it performs on data that has not yet been seen.

The trained model can also be saved to a file and loaded later for use in inference. Saving and Loading the Model.

his code defines a neural network with two input features, two hidden layers, and one output neuron. It uses the binary cross-entropy loss and stochastic gradient descent optimizer to train the model on a simple binary classification task. The trained model is saved to a file and loaded later for use in inference.

Here are the steps for building machine learning models using PyTorch:

Step 1: Import the necessary PyTorch modules

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

We import PyTorch and its modules, including “nn” for defining neural networks, “optim” for optimization algorithms, and transforms for data augmentation.

Step 2: Define the CNN architecture

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 6, 5)

        self.pool = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(6, 16, 5)

        self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

        self.fc2 = nn.Linear(120, 84)

        self.fc3 = nn.Linear(84, 10)

        self.relu = nn.ReLU()

    def forward(self, x):

        x = self.pool(self.relu(self.conv1(x)))

        x = self.pool(self.relu(self.conv2(x)))

        x = x.view(-1, 16 \* 5 \* 5)

        x = self.relu(self.fc1(x))

        x = self.relu(self.fc2(x))

        x = self.fc3(x)

        return x

We define our CNN architecture by inheriting from the nn.Module class and defining the layers in the constructor (\_\_init\_\_ method). Our CNN consists of two convolutional layers with ReLU activation functions, two max pooling layers, and three fully connected layers with ReLU activation functions and one final output layer. We use the forward method to define the forward pass of the network.

Step 3: Load and preprocess the data

ransform = transforms.Compose(

    [transforms.RandomHorizontalFlip(),

     transforms.RandomCrop(32, padding=4),

     transforms.ToTensor(),

     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

                                          shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

                                         shuffle=False, num\_workers=2)

We use the “transforms.Compose” function to define a series of transformations to apply to the dataset, including random horizontal flipping and cropping, converting to a PyTorch tensor, and normalization. We load the CIFAR-10 dataset and apply the defined transformations. We use “torch.utils.data.DataLoader” to create batches of the data for training and testing.

Step 4: Create an instance of the CNN

net = Net()

We create an instance of our CNN.

Step 5: Define the loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

We define the cross-entropy loss function and stochastic gradient descent (SGD) optimizer, which updates the weights based on the gradients of the loss function.

Step 6: Train the CNN

for epoch in range(2):  # loop over the dataset multiple times

    running\_loss = 0.0

    for i, data in enumerate(trainloader, 0):

        # get the inputs; data is a list of [inputs, labels]

        inputs, labels = data

        # zero the parameter gradients

        optimizer.zero\_grad()

        # forward + backward + optimize

        outputs = net(inputs)

        loss = criterion(outputs, labels)

        loss.backward()

        optimizer.step()

        # print statistics

        running\_loss += loss.item()

        if i % 2000 == 1999:    # print every 2000 mini-batches

            print('[%d, %5d] loss: %.3f' %

                  (epoch + 1, i + 1, running\_loss / 2000))

            running\_loss = 0.0

Step 7: Test the CNN on the test dataset

correct = 0

total = 0

with torch.no\_grad():

    for data in testloader:

        images, labels = data

        outputs = net(images)

        \_, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)

        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (

    100 \* correct / total))

Full code :

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

# Define the CNN architecture

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 6, 5)

        self.pool = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(6, 16, 5)

        self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

        self.fc2 = nn.Linear(120, 84)

        self.fc3 = nn.Linear(84, 10)

        self.relu = nn.ReLU()

    def forward(self, x):

        x = self.pool(self.relu(self.conv1(x)))

        x = self.pool(self.relu(self.conv2(x)))

        x = x.view(-1, 16 \* 5 \* 5)

        x = self.relu(self.fc1(x))

        x = self.relu(self.fc2(x))

        x = self.fc3(x)

        return x

# Load and preprocess the data

transform = transforms.Compose(

    [transforms.RandomHorizontalFlip(),

     transforms.RandomCrop(32, padding=4),

     transforms.ToTensor(),

     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

                                          shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

                                         shuffle=False, num\_workers=2)

# Create an instance of the CNN

net = Net()

# Define the loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the CNN

for epoch in range(2):  # loop over the dataset multiple times

    running\_loss = 0.0

    for i, data in enumerate(trainloader, 0):

        # get the inputs; data is a list of [inputs, labels]

        inputs, labels = data

        # zero the parameter gradients

        optimizer.zero\_grad()

        # forward + backward + optimize

        outputs = net(inputs)

        loss = criterion(outputs, labels)

        loss.backward()

        optimizer.step()

        # print statistics

        running\_loss += loss.item()

        if i % 2000 == 1999:    # print every 2000 mini-batches

            print('[%d, %5d] loss: %.3f' %

                  (epoch + 1, i + 1, running\_loss / 2000))

            running\_loss = 0.0

print('Finished Training')

# Test the CNN on the test dataset

correct = 0

total = 0

with torch.no\_grad():

    for data in testloader:

        images, labels = data

        outputs = net(images)

        \_, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)

        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (

    100 \* correct / total))