Pytorch basics

Dr. Ahmad Altarawneh

Department of Data Science

Faculty of information technology, Mutah University

Outline

- What is Pytorch?
- Tensors
- Autograd
- MLP using Pytorch
- Datasets and dataloaders
- Optimizers in Pytorch

What is Pytorch

- PyTorch is an open source machine learning (ML) framework based on the Python programming language and the Torch library, and used for creating deep neural networks
- It's one of the preferred platforms for deep learning research.
- The framework is built to speed up the process between research prototyping and deployment.
- Strong competitive to Tensorflow

Pytorch features

- Dynamic Computation Graph (Autograd): PyTorch supports dynamic computational graphs, meaning the graph is created on the fly, making it highly flexible and easy to debug.
- Tensors: PyTorch operates with multi-dimensional arrays (called Tensors), similar to NumPy arrays, but with GPU acceleration.
- Rich Ecosystem: PyTorch has a wide range of libraries for computer vision (TorchVision), natural language processing (TorchText), and reinforcement learning (TorchRL).
- Scalability: PyTorch can scale seamlessly from research experiments to production models

Tensors

- Tensors are multi-dimensional arrays, similar to NumPy arrays, that serve as the fundamental data structure in PyTorch. They allow for efficient computation on CPUs and GPUs.
- Tensors can have any number of dimensions, from 0D (scalar) to ND (multi-dimensional arrays), making them versatile for different types of data (e.g., images, sequences, etc.).
- GPU Acceleration: Unlike NumPy arrays, PyTorch tensors can be processed on both CPUs and GPUs, enabling faster computation for large datasets and deep learning models.
- Automatic Differentiation: PyTorch tensors can track gradients, which is crucial for backpropagation in neural networks.

Creating Tensors

- A tensor can be created simply using the torch library as follows

```
import torch
```

Create a tensor with specified values

```
t1 = torch.tensor([1,2,3])
t1
```

tensor([1, 2, 3])

Create a tensor of zeros

```
t1 = torch.zeros(5)
```

tensor([0., 0., 0., 0., 0.])

Create a tensor of ones

```
t1 = torch.ones(5)
```

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

Create a tensor from numpy array

```
import torch
import numpy as np

arr= np.array([1,2,3])
t1 = torch.from_numpy(arr)
```

tensor([1, 2, 3])

Creating Tensors
Cont.

- WE can create multi dimensional Tensor (e.g., 2d)

Create a 2d tensor	`t1 = torch.tensor([[1,2],[3,4]])`
Create a tensor with a specific datatype	`t1 = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float32)`
Create a normal random-valued tensor	`t1 = torch.randn(3, 3) # create a 3x3 tensor of normally distributed random values`
Create a uniform random-valued tensor	`t1 = torch.rand(3, 3) # create a 3x3 tensor of uniformally distributed random values`
Create a tensor from a Python list	`my_list = [1, 2, 3, 4, 5]` `tensor = tensor = torch.tensor(my_list)`
Create a Create a tensor from a numpy array	`tensor = torch.from_numpy(np_array)    #OR` `tensor = torch.tensor(np_array)`

One can use tensorName.tolist() to convert a tensor to a python list. Likewise, you can use tensorName.numpy() to convert tensor to numpy array

Basic operations on tensors

- One can perform the basic operations on Tensors very easily
  - Add two tensors

```
t1 = torch.tensor([1,2,3])
t2 = torch.tensor([4,5,6])
t3 = t1+t2
t3
```

tensor([5, 7, 9])

  - Subtract two tensors

```
t1 = torch.tensor([1,2,3])
t2 = torch.tensor([4,5,6])
t3 = t1-t2
t3
```

tensor([-3, -3, -3])

  - Multiply and divide two tensors

```
t1 = torch.tensor([1,2,3])
t2 = torch.tensor([4,5,6])
t3 = t1*t2
t3
```

tensor([4, 10, 18])

Basic operations on tensors
Cont.

- Same operation can be done between tensor and scalers

```
t1 = torch.tensor([1,2,3])
t2 = torch.tensor(5)
t3 = t1 + t2
t4 = t1 - t2
t5 = t1 * t2
t6 = t1 / t2
print(t3,t4,t5,t6)
```

```
tensor([6, 7, 8])
tensor([-4, -3, -2])
tensor([ 5, 10, 15])
tensor([0.2000, 0.4000, 0.6000])
```

- Matrix multiplications, can be performed using the @ operator or **matmul** method (this one is used to calculate the dot product as the tensors are 1D)

```
t1 = torch.tensor([1,2,3])
t2 = torch.tensor([4,5,6])
t1@t2
```

tensor(32)

```
t1 = torch.tensor([[1,2],[2,5],[3,1]])
t2 = torch.tensor([[4,5,6]])
t2@t1
```

tensor([[32, 39]])

Auto-gradients calculation

- Tensors have the ability to track the operations done on them and perform auto differentiation when needed w.r.t any variables (tensors) involved in this operation (function)
- To activate this autograd, you have to pass the attribute requires_grad = True
  - For example,

```
w = torch.tensor([0.2, 0.4], requires_grad=True)
```

- Now assume that we want to apply the following equation

$$z = w.x + b$$
$$\hat{y} = Sigmoid(z)$$

$$w = [2, 5], \mathrm{x} = [-10.0, 3.0], \mathrm{b} = 1$$

Cont.

- let us apply a single forward pass and calculate the gradients manually, then using Pytorch and compare

$$z = 2 * -10 + 5 * 3 + 1 = -4$$

$$\hat{y} = Sigmoid(-4) = 0.0179$$

$$loss = (0.0179 - 1)^2 = 0.964$$

$$\frac{\partial loss}{\partial w_1} = 2 * -0.9821 * sig(-4) * \big(1 - sig(-4)\big) * -10 = \mathbf{0.3469}$$

$$\frac{\partial loss}{\partial w_2} = 2 * -0.9821 * sig(-4) * \big(1 - sig(-4)\big) * 3 = \mathbf{-0.1041}$$

$$\frac{\partial loss}{\partial b} = 2 * -0.9821 * sig(-4) * \big(1 - sig(-4)\big) = \mathbf{-0.0347}$$

Using Pytorch

```python
import torch

w = torch.tensor([2.0, 5.0], requires_grad=True)
x = torch.tensor([-10.0, 3.0])
b = torch.tensor(1.0, requires_grad=True)
y = torch.tensor(1.0)

z = torch.dot(w, x) + b
y_hat = torch.sigmoid(z)

mse_loss = (y_hat- y) ** 2

mse_loss.backward()

print(f"Weighted sum (z): {z.item()}")
print(f"Predicted output (y_hat): {y_hat.item()}")
print(f"MSE Loss: {mse_loss.item()}")

print(w.grad)
print(b.grad)
```

```
Weighted sum (z): -4.0
Predicted output (y_hat): 0.01798621006309986
MSE Loss: 0.9643510580062866
tensor([ 0.3469, -0.1041])
tensor(-0.0347)
```

MLP using Pytorch

- Pytorch has vast amount of functionality that simplifies building neural networks

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


class MLP(nn.Module):
    def __init__(self, inputsD, outputD):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(inputsD, 100)
        self.fc2 = nn.Linear(100, 100)
        self.fc3 = nn.Linear(100, outputD)

    def forward(self, X):

        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return X

model = MLP(4, 3)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005)
```

Code components

- Code components:

1. inherits from `nn.Module`, which has the main functionalities, layers that one might need to build a neural network
2. `nn.Linear`: is the dense layer, in Pytorch it is called linear as it performs linear combinations
3. `criterion`: is the loss function we want to use for training the network
4. `optimizer`: is the optimization method we want to use to train the model (network)

5. One might refer to the documentation to find more about layers, loss functions and optimizers
    - https://pytorch.org/docs/stable/nn.html

Dataset(s) and Dataloader(s)

- Before we dive into the training loop, let us discuss about dataloders and datasets
- **Dataset** is a collection of observations on which you want to train the model. It gives you the ability to define how you want to load and process each observation in your dataset
- **DataLoader** is used to load data from a dataset in batches, enabling efficient training of models. It handles the iteration over the dataset and allows for features like shuffling, batching, and parallel data loading

Datasets

- To build a custom dataset in pytorch, you have to inherit the **Dataset** class

```
from torch.utils.data import Dataset
import pandas as pd
import torch

class MyDataset(Dataset):
    def __init__(self, data, labels):
        self.X = data
        self.y = labels

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        sample = torch.tensor(self.X[idx], dtype=torch.float32)
        label = torch.tensor(self.y[idx], dtype=torch.long)
        return sample, label

data = pd.read_csv("Iris.csv")

x = data.iloc[:,:-1].values
y,_ = pd.factorize(data.iloc[:,-1].values)

dataset = MyDataset(x, y)

dataset.__getitem__(0)
```

	A	B	C	D	E
1	SepalLeng	SepalWidtl	PetalLengt	PetalWidtł	Species
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa
11	4.9	3.1	1.5	0.1	Iris-setosa
12	5.4	3.7	1.5	0.2	Iris-setosa
13	4.8	3.4	1.6	0.2	Iris-setosa

The Iris dataset

Ensure that the data returned as a tensor

Datasets: Another method

- One can embed the reading logic inside the dataset class

```
from torch.utils.data import Dataset
import pandas as pd
import torch

class MyDataset(Dataset):
    def __init__(self, path):
      data = pd.read_csv(path)
      self.X = data.iloc[:,:-1].values
      self.y,_ = pd.factorize(data.iloc[:,-1].values)

    def __len__(self):
      return len(self.y)

    def __getitem__(self, idx):
      sample = torch.tensor(self.X[idx], dtype=torch.float32)
      label = torch.tensor(self.y[idx], dtype=torch.long)
      return sample, label

path = '/content/Iris.csv'

dataset = MyDataset(path)

dataset.__getitem__(50)
```

This can be the path to the train subset

Datasets: Another method

- You can create helper functions to do subtasks inside the dataset class

```
from torch.utils.data import Dataset
import pandas as pd
import torch

class MyDataset(Dataset):
    def __init__(self, path):
      data = pd.read_csv(path)
      self.X = data.iloc[:,:-1].values
      self.y = self.factorize(data.iloc[:,-1])

    def __len__(self):
      return len(self.y)

    def __getitem__(self, idx):
      sample = torch.tensor(self.X[idx], dtype=torch.float32)
      label = torch.tensor(self.y[idx], dtype=torch.long)
      return sample, label

    def factorize(self, labels):
      y,_ = pd.factorize(labels.values)
      return y

path = '/content/Iris.csv'
dataset = MyDataset(path)
```

Datasets
Cont.

- The methods __getitem__ and __len__ have to be implemented to fit your custom dataset
- in this case we know that we will pas a pandas dataframe, therefore, we handle it inside the class
- The __getitem__ method defines how to retrieve a single sample from the dataset
- The __len__ method have implementation that return the number of samples in the dataset
  - These implementations may differ from one dataset to another based on the data in hand (its structure and type).
  - For images you might need to implement how the image should be processed before returned, for example.

Dataloader

- Now the way you want to feed the dataset into the model is defined by the dataloader
- For example, do you want to shuffle the data, what is the size of mini-batches, how many CPUs you want to work on your retrieving datafrom the dataset (num_workers)

```
from torch.utils.data import DataLoader

Mydataldr = DataLoader(dataset, batch_size=4, shuffle=True)

for batch in Mydataldr:
    inputs, labels = batch
    print(inputs, labels)
    break
```

Shuffle after each epoch

This gives the first mini-batch of the dataset

```
tensor([[6.5000, 3.0000, 5.5000, 1.8000],
        [6.9000, 3.2000, 5.7000, 2.3000],
        [6.3000, 3.3000, 6.0000, 2.5000],
        [6.7000, 3.1000, 5.6000, 2.4000]]) tensor([1, 1, 1, 1])
```

Why shuffling is important

- Shuffling is important during the training of the neural network for several reasons:

1. Prevent the network from learning the order of the data samples
2. Prevent the model's gradients from giving advantage to early seen examples
   - As the gradient is large at the start and decreasing over time
   - Especially if a **Learning Rate Decay** technique is used
3. Helping in learning from imbalanced datasets
4. Randomness in the order of the data helps in the generalization process

Train and test dataloaders

- for training and testing, we need to create dataset and dataloader for the training and for the testing
  - Split them before creating DS & DL
- You might think this is complicated, but this actually is very convenient.
- Practice and you will find it very easy and intuitive

```
class MyDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.X = data
        self.y = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = torch.tensor(self.X[idx], dtype=torch.float32)
        label = torch.tensor(self.y[idx], dtype=torch.long)
        return sample, label

data = pd.read_csv("Iris.csv")
x = data.iloc[:,:-1].values
y,_ = pd.factorize(data.iloc[:,-1].values)

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.33,
random_state=42)

trainDataset = MyDataset(X_train, y_train)
testDataset = MyDataset(X_test, y_test)

traindataldr = DataLoader(trainDataset, batch_size=8, shuffle=True)
testdataldr = DataLoader(testDataset, batch_size=32, shuffle=True)
```

Is shuffling really needed here?

Steps for a full implementation

- Step 1: import the needed libraries and define the model

```
from torch.utils.data import Dataset, DataLoader
import pandas as pd
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(4, 100)
        self.fc2 = nn.Linear(100, 100)
        self.fc3 = nn.Linear(100, 3)

    def forward(self, X):

        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return X
```

Steps for a full implementation

- Step 2: Define the datasets and dataloaders

```
class MyDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.X = data
        self.y = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = torch.tensor(self.X[idx], dtype=torch.float32)
        label = torch.tensor(self.y[idx], dtype=torch.long)
        return sample, label

data = pd.read_csv("Iris.csv")
x = data.iloc[:,:-1].values
y,_ = pd.factorize(data.iloc[:,-1].values)

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.33, random_state=42)

trainDataset = MyDataset(X_train, y_train)
testDataset = MyDataset(X_test, y_test)

traindataldr = DataLoader(trainDataset, batch_size=8, shuffle=True)
testdataldr = DataLoader(testDataset, batch_size=32, shuffle=True)
```

Steps for a full implementation

- Step 3: Create instance from the model, define the oprimizer and loss function

```
model = MLP(4, 3)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005)
```

Steps for a full implementation

- Step 4: Training loop

```
for epoch in range(1000):
    running_loss = 0.0
    for inputs, targets in traindataldr:
        optimizer.zero_grad()

        outputs = model(inputs)

        loss = criterion(outputs, targets)

        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        test_loss, test_accuracy = evaluate(model, testdataldr, criterion)

    if epoch % 100 == 0:
        print(f"Epoch {epoch+1}, Loss: {running_loss/len(traindataldr)}, and test loss is {test_loss}")
```

Steps for a full implementation

- Step 5: test the model

```
def evaluate(model, dataloader, criterion):
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0

    # Stop calculating the gradients
    with torch.no_grad():
        for inputs, targets in dataloader:
            outputs = model(inputs)

            loss = criterion(outputs, targets)
            test_loss += loss.item()

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

            total += targets.size(0)
            correct += (predicted == targets).sum().item()

    avg_loss = test_loss / len(dataloader)
    accuracy = 100 * correct / total

    return avg_loss, accuracy
```

Important Notes

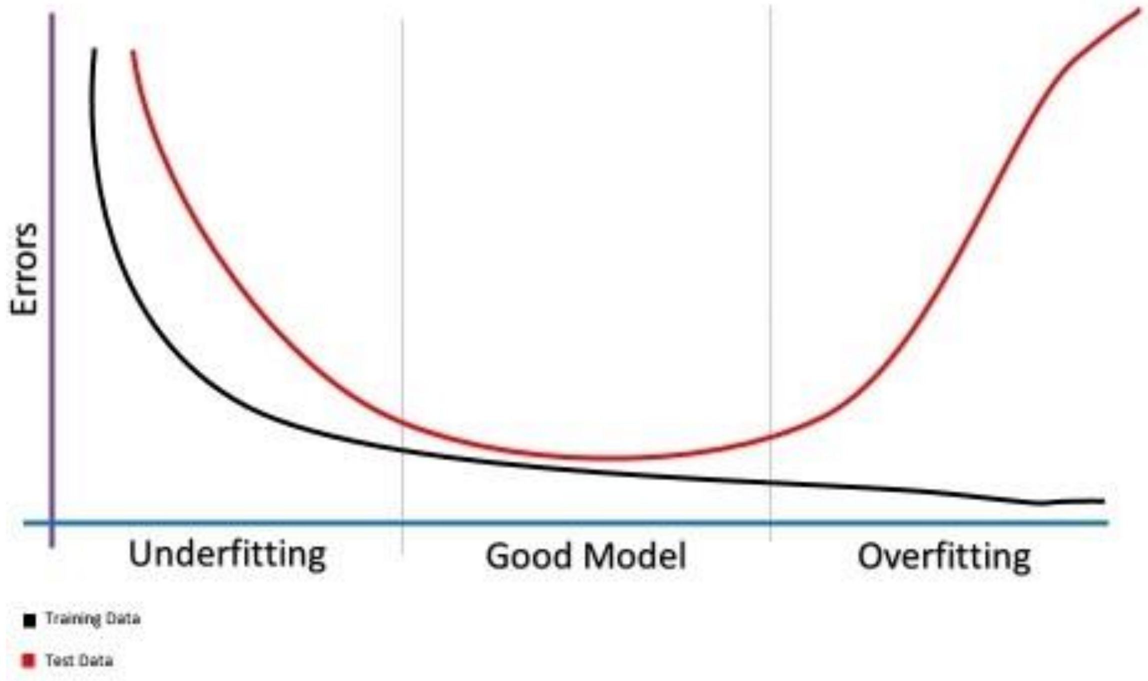
- During the training, in the training loop, you have to set `optimizer.zero_grad()`, so the gradients will not accumulate with batches
- `loss.backward()` calculates the gradients w.r.t loss and all of the model's parameters
- `optimizer.step()` updates the parameters of the model based on the gradients
- `evaluate` function is called inside the training loop to track the test loss along with the training loss, tracking the overfitting
- There is no need for Softmax layer in the model as `CrossEntropyLoss` calculates it internally.
  - if the loss does not calculate the Softmax, then it should be added to the model as a final layer
- `model.eval()` tells Pytorch that the model is in the evaluation mode, so to ignore regularization layers, like dropout (discussed in the next course)
- `with torch.no_grad()` is important to prevent the model from calculating or tracking the gradients during the test process
  - In the test no need to calculate the gradients, calculating gradients is for training

Important Notes
Cont.

- Learning rate, loss function, batch size, number of layers, activation functions , etc. are called hyperparameters, these need to be tuned manually (based on experience)
- If the learning rate is low, you might need more epochs
- small batch sizes are better during the training, 16, 32, and this depends upon the data size (number of observations)

Overfitting and underfitting

- Training and test (validation) losses have to be close to each other, in good model training
- If the training is much lower than the test (validation)Then the model is overfitting (loosing generalization)
- If both (test loss and training loss) are very high and do not decrease over epochs, then the model is underfitting



Pytorch built-in methods

- Pytorch provides various functionalities to use for building your model
- Activation functions:

Activation Function	PyTorch Function
ReLU	torch.nn.ReLU()
LeakyReLU	torch.nn.LeakyReLU()
Sigmoid	torch.nn.Sigmoid()
Tanh	torch.nn.Tanh()
Softmax	torch.nn.Softmax()

Pytorch built-in methods

- Pytorch provides various functionalities to use for building your model
- Optimizers functions:

Optimizer	PyTorch Function
SGD	torch.optim.SGD()
Adam	torch.optim.Adam()
AdamW	torch.optim.AdamW()
RMSprop	torch.optim.RMSprop()

Pytorch built-in methods

- Pytorch provides various functionalities to use for building your model
- Loss functions:

Loss Function	Type	PyTorch Function
CrossEntropyLoss	Classification	torch.nn.CrossEntropyLoss()
BCELoss	Classification	torch.nn.BCELoss()
NLLLoss	Classification	torch.nn.NLLLoss()
HingeEmbeddingLoss	Classification	torch.nn.HingeEmbeddingLoss()
MSELoss	Regression	torch.nn.MSELoss()
L1Loss	Regression	torch.nn.L1Loss()
SmoothL1Loss	Regression	torch.nn.SmoothL1Loss()
HuberLoss	Regression	torch.nn.HuberLoss()

Practice

- Build a neuralnetwork using Pytorch for regression problem
  - use boston dataset: **from sklearn.datasets import load_ boston**
- Define the model, datasets and dataloaders and the training loop
  - use MSE, MAE, RMSE to evaluate the performance of your model
- Use different loss functions and model architecture, and compare the results

- **TRY IT YOURSELF**