Agenda

- PyTorch Components
  - Tensors
  - Autograd
  - Optimizers & Loss functions
  - Model
- PyTorch model training
- Saving model and Deployment
- Dataset and Dataloader
- Hands-on example
- Q & A

Tensors

- Tensors are similar to NumPy’s ndarrays
- Can run on GPUs or other hardware accelerators
- Optimized for automatic differentiation

Autograd

- Automatic differentiation package : No need to worry about back propagation partial derivatives and chain rule

- Tensors track their computational history and support gradient computation

- **`requires_grad=True`** : Tells PyTorch that we want to compute gradients for the specific tensor

Autograd : `backward()`

- The **`backward()`** function is responsible for calculation of gradients and **accumulate** (not apply) them in respective tensors

- The tensor with **`requires_grad=True`**: has attribute to check the gradients values : '**`grad`**'

- Because of the **accumulate** it is important to zero the accumulated values before any calculations '`zero_grad()`'

Optimizers

- Optimizers facilitate the update of tensors values with the gradients
- In PyTorch the reset of accumulated gradients is facilitated by the optimizer **'`zero_grad()`'**
- To facilitate the updates of tensors values with the gradient values and learning rate **'`step()`'** method should be evoked

Loss Functions

Recap on Linear Regression Loss function

Mean Squared Error (MSE)

$$f(x_i) = w_0 + w_1 x_i$$

$$e_i = y_i - f(x_i)$$

$$\mathcal{L}(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (w_0 + w_1 x_i)\right)^2$$

We need to find the value of parameters that minimize this cost or loss function.

torch.nn.MSELoss

Recap on Logistic Regression Loss Function

Loss Function of Logistic Regression (4)

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n} y^i \log\left(p\left(x^i\right)\right) + \left(1 - y^i\right)\log(1 - p\left(x^i\right))$$

- Given this loss function, what do you think we are gonna do next?

- We will define our objective function

$$\boxed{\underset{w_o, w_1}{\operatorname{argmin}}\ \mathcal{L}(\boldsymbol{w})}$$

torch.nn.BCELoss

Other Loss functions

- Kullback-Leibler divergence : **`torch.nn.KLDivLoss`**
- Cosine Embedding : **`torch.nn.CosineEmbeddingLoss`**
- Negative log likelihood loss : **`torch.nn.NLLLoss`**
- Cross entropy loss : **`nn.CrossEntropyLoss`**

In PyTorch a loss function is called ***criterion***

[More Loss functions and description]()

Creating a simple ANN & DNN

Model

- A **model** is represented by a regular **Python class** that inherits from the **Module** class
- **`__init__(self):`** it defines the parts that make up the model
- **`forward(self,x):`** performs a forward pass

```
import torch

class myModel(torch.nn.Module):
  def __init__(self) :
    super(...).__init__()
            ...
  def forward(self,x):
    ...
    return x
```

Model important attributes

- **`model.train()`** : sets the model to training mode. Keep track of the gradients and computations in the gragh

- **`model.eval()`** : sets the model to evaluation mode (no need to accumulate gradients and ignore dropout)

- **`model.parameters()`** : retrieves an iterator over all model's parameters

- **`model.state_dict()`** : retrieves model current values for all parameters

Sequential Models

```
import torch

class myModel(torch.nn.Module):
   def __init__(self) :
     super(...).__init__()
              ...
   def forward(self,x):
     ...
```

```
import torch

model = torch.nn.Sequential(...) model.train()
...
model.eval()
```

Training the model

Typical procedure in training a neural network using PyTorch:

1. Define the model
2. Define loss function
3. Define optimizer
4. Define training loop

```
import torch
import torch.optim as optim

# 1. define model
model = torch.nn.Sequential(...)

# 2. define loss function (i.e regression)
criterion = torch.nn.MSELoss()

# 3. Define Optimizer
optimizer = optim.SGD(model.parameters(),
lr=0.1)

# define training loop

Next slide ...
```

Training the model

Typical procedure in training a neural network using PyTorch:

1. Define the model
2. Define loss function
3. Define optimizer
4. **Define training loop**

```
import torch

# 4. Define training loop
for epoch in range(n_epochs):
 for batch in batches:
   inputs = batch[0].to(device)
   labels = batch[1].to(device)

   # zero the parameter gradients
   optimizer.zero_grad()

   # forward + backward + optimize
   outputs = net(inputs)
   loss = loss_function(outputs, labels)

   # accumulate gradients and update params
   loss.backward()
   optimizer.step()
```

Saving and loading Model

- PyTorch models store the learned parameters in an internal state dictionary `model.state_dict()`

- Can be persisted via the PyTorch save method `torch.save(...)`

```
import torch
model = torch.nn.Sequential(...)
torch.save(model.state_dict(), 'model_weights.pt')
.....
model.load_state_dict(torch.load('model_weights.pt'))
```

Loading a Dataset

- PyTorch provides a number of pre-loaded datasets, such as MNIST, CIFAR10, CIFAR100
- They inherit from `torch.utils.data.Dataset` and implement functions specific to the particular data.
- Prepare data for training with `DataLoaders` (mini-batches, shuffle, multi-processing)

```
import torch
from torchvision import datasets

training_data =
datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor())
```

```
train_dataloader = DataLoader(training_data,
                              batch_size=64,
                              shuffle=True,
                              num_workers=2)
```

Custom DataLoader and Dataset

- The custom loader must extend torch.utils.data.DataLoader

```
import torch
from torch.utils.data import Dataset

class MyDatasetLoader(Dataset):
    """ Create data iterator “""
    def __init__(self, X, y):
      …
    def __len__(self):
      …

    def __getitem__(self, idx):
      return self.X[idx, :], self.y[idx]
```

```
training_data = MyDatasetLoader(X, y)
train_loader = DataLoader(training_data, batch_size=64,
```

Resources

PyTorch

https://pytorch.org