PyTorch Introduction

Deep Learning

TensorFlow

- Torch was introduced in 2002 – an open-source machine learning library and scientific computing framework based on Lua.
- Developed at the Idiap Research Institute at EPFL in Lausanne, Switzerland.
- The main developers were Ronan Collobert, Samy Bengio and Johnny Mariétho.
- In 2016 the front-end was converted by Soumith Chintala, Adam Paszke, Sam Gross and Gregory Chanan at Facebook (now Meta) from Lua to Python, and renamed PyTorch.

Model training steps (same as TensorFlow)

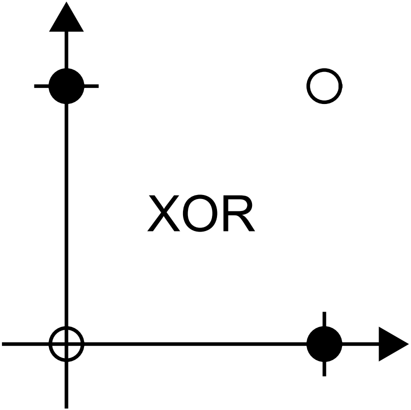
- Load the data.
- Construct the network.
- Train the network.
- Analyze the results.

Data formats

- PyTorch has the same formats for tensors as TensorFlow (features, samples, timesteps and channels).
- PyTorch tensors can be created directly from data, using the `torch.tensor()` command.
- Can be created from a numpy array, using the `torch.from_numpy()`.

Network Inputs

- Each element of a network input is called a feature.
- The data set will consist of Q samples of inputs.
- If each input is a vector (tabular) with R features, then the data set is a (Q, R), or (samples, features), NumPy tensor.
- If the input is a time series, the input is a 3D tensor of the form (samples, timesteps, features).
- For images, the 4D input tensor form is (samples, height, width, channels), where channels are usually colors.
- For videos, the 5D input tensor form is (samples, timesteps, height, width, channels).
- For Keras, the network outputs are also NumPy tensors.

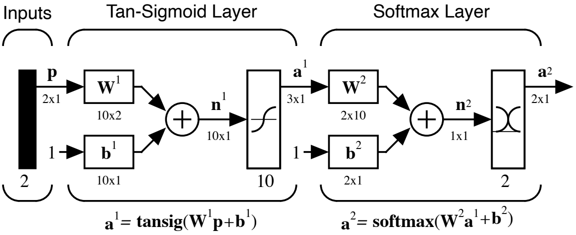

XOR

Generate XOR data

```
import numpy as np
import torch

p = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]],dtype=torch.float32)
t = torch.tensor([0, 1, 1, 0])
print(p)
print(t)
```

```
tensor([[0., 0.],
        [0., 1.],
        [1., 0.],
        [1., 1.]])
tensor([0, 1, 1, 0])
```




```python
model = torch.nn.Sequential(
    torch.nn.Linear(2, 10),
    torch.nn.Tanh(),
    torch.nn.Linear(10, 2))
print(model)
```

```
Sequential(
  (0): Linear(in_features=2, out_features=10, bias=True)
  (1): Tanh()
  (2): Linear(in_features=10, out_features=2, bias=True))
```

Constructing the model using the model subclass method

```python
class TwoLayer(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.dense1 = torch.nn.Linear(2, 10)
        self.tanh = torch.nn.Tanh()
        self.dense2 = torch.nn.Linear(10, 2)
    def forward(self, x):
        x = self.dense1(x)
        x = self.tanh(x)
        x = self.dense2(x)
        return x
model = TwoLayer()
print(model)
```

```
TwoLayer(
  (dense1): Linear(in_features=2, out_features=10, bias=True)
  (tanh): Tanh()
  (dense2): Linear(in_features=10, out_features=2, bias=True))
```

Before training

- We need to select the optimizer.
- First argument of the optimizer contains the variables to be adjusted.
- You can also specify optimizer options, such as learning rate.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

- You also need to define the loss function.

```
loss_fn = torch.nn.CrossEntropyLoss()
```

PyTorch training steps

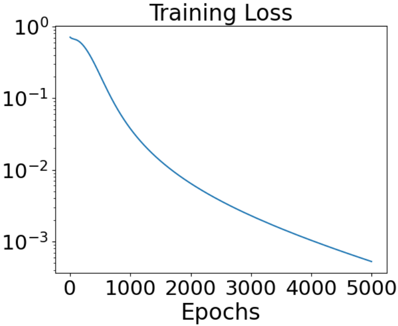
1. Zero the gradient
2. Make a forward pass through the network
3. Calculate the loss
4. Compute the gradient
5. Update the weights

```python
#Check if a GPU is available, and move model and data
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model.to(device)
p = p.to(device)
t = t.to(device)
epochs = 5000
loss_values = []
for epoch in range(epochs):
    # Zero the gradient
    optimizer.zero_grad()
    # Forward pass through the network
    output = model(p)
    # Calculate the loss
    loss = loss_fn(output, t)
    # Compute the gradient
    loss.backward()
    # Save the loss values for plotting
    loss_values.append(loss.cpu().detach().numpy())
    # Update the weights
    optimizer.step()
```

Convergence plot

We saved the loss values, so that we could view the process.

```
import matplotlib.pyplot as plt
plt.semilogy(range(epochs), loss_values)
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.show()
```



```
n = model(p)
softmax = torch.nn.Softmax(dim=1)
a = softmax(n)
print(a)
-----------------------------------------------------
tensor([[9.9967e-01, 3.2964e-04],
        [5.4206e-04, 9.9946e-01],
        [2.5702e-04, 9.9974e-01],
        [9.9960e-01, 4.0006e-04]], device='cuda:0', grad_fn=<SoftmaxBackward0>)
```

We added softmax, because it was not included in the model.

Advanced Data Loading

- Before network training comes Extract, Transform and Load (ETL).
- First, the data are taken from one or multiple files, which may be distributed across multiple machines.
- Next, the data is transformed (normalizing, augmenting by rotating or scaling images, adding noise, etc.)
- Finally, the data is loaded into the training process, often in minibatches.
- `torch.utils.data.Dataset` defines what the data are and how to access them.
- `torch.utils.data.DataLoader` defines how to load data efficiently for training or inference.

PyTorch Dataset

1. Represents the data and enables access to individual samples.
2. Each item of the dataset has an input and a target.
3. Defines how to load and preprocess individual data points.
4. Implements `__getitem__()` to retrieve a single sample and `__len__()` for the total number of samples.
5. Focuses on data representation and access.

PyTorch DataLoader

1. Wraps a `Dataset` and provides utilities for batching, shuffling, and parallel data loading
2. Handles the iteration over the dataset, creating batches, and optionally shuffling
3. Allows easy specification of batch size, number of worker processes, etc.
4. Focuses on efficiently feeding data to the model during training/evaluation

```
import pandas as pd
sample_df = pd.read_csv('SampleDF.csv')
```

Extract two columns that we will use as inputs and targets.

```
P = np.array(sample_df['FVC'])
T = np.array(sample_df['Percent'])
```

Define a Dataset class

```
class SimpleDataset(torch.utils.data.Dataset):
    def __init__(self, P, T):
        # convert into PyTorch tensors
        self.P = torch.tensor(P, dtype=torch.float32)
        self.T = torch.tensor(T, dtype=torch.float32)

    def __len__(self):
        # this should return the size of the dataset
        return len(self.P)

    def __getitem__(self, idx):
        # return one input and one target sample from the dataset
        features = self.P[idx]
        target = self.T[idx]
        return features, target
```

```
dataset = SimpleDataset(P, T)
```

Wrap the data in a `DataLoader`.

```
loader = torch.utils.data.DataLoader(dataset, shuffle=True, batch_size=4)
```

Iterate the `DataLoader`.

```
for feat, targ in loader:
  print('Features: {}, Target: {}'.format(feat, targ))
  break
```

```
Features: tensor([2603., 2762., 1537., 4738.]), Target: tensor([ 71.1591, 72.4174,  63.5281, 113.0464])
```