PyTorch Tutorial

EECS 445 Staff

Fall 2023

In recent years, a variety of neural network libraries have been developed to abstract away mathematical and implementation-level details of modern deep learning. This includes details like automatic gradient calculations, distributed training, and support for dedicated hardware (e.g. GPUs, TPUs).

In EECS 445, we will use PyTorch, an industry-standard framework open-sourced by Meta AI, and adopted by many machine learning researchers. This framework will be used in **Project 2** to create and train neural networks. This tutorial will cover the fundamentals of PyTorch by creating a very simple feedforward neural network for binary classification and training it.

For this tutorial, suppose that we are performing a binary classification task on a dataset $\{\bar{x}^{(i)}, y^{(i)}\}_{i=1}^{n}$ with $\bar{x}^{(i)} \in \mathbb{R}^2$ and $y^{(i)} \in \{0, 1\}$. Also assume that the training and testing datasets are of the form `X_train, y_train, X_test, y_test` (similar to Project 1) and that our neural network uses the *sigmoid* activation function.

Note: This tutorial is adapted from tutorials in the PyTorch library documentation.

1 PyTorch Tensors

Before jumping into neural networks, it's important to discuss the concept of **tensors**. Tensors work similarly to NumPy arrays that we've used before, but are more optimized for use with neural networks in PyTorch.

NumPy arrays and PyTorch tensors have similar interfaces, and so the NumPy skills learned earlier in the course transfer over to using PyTorch tensors. For example, the code snippet below creates two matrices $A, B \in \mathbb{R}^{3\times5}$ filled with random values, then (1) adds them together, and (2) computes the sum of each row of A.

```
import torch

A = torch.rand((3,5))
B = torch.rand((3,5))

print(A+B)          # prints element-wise sum of matrices A and B
print(A.sum(dim=1)) # prints sum of each row of A. dim specifies dimension to sum across
```

> Refer to the Tensor documentation for the full list of supported operations and construction methods.

Many objects in PyTorch are represented as a tensor, including both feature vectors and neural network weights. Understanding their usage is critical for implementing neural networks in PyTorch.

2 Defining a Neural Net

In PyTorch, all neural networks are defined as subclasses of the `torch.nn.Module` base class.

The code snippet below shows an example of a basic feedforward neural network implemented by inheriting from this base class.

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__() # call superclass constructor

        # Step 1. Define fully-connected layers for the neural network
        self.hidden1 = nn.Linear(2, 3)
        self.hidden2 = nn.Linear(3, 2)
        self.output = nn.Linear(2, 2)

    def forward(self, x):
        # Step 2. Override the forward function to define the forward pass
        z2 = self.hidden1(x)
        h2 = F.sigmoid(z2)
        z3 = self.hidden2(h2)
        h3 = F.sigmoid(z3)
        z4 = self.output(h3)
        return z4
```

There's a bit to unpack in the above code, so let's break it down by understanding the two steps it takes to define a neural network in PyTorch:

1. Define the layers and components of the neural network in the constructor (`__init__()`)
2. Override `forward()` to implement the forward pass of the algorithm.

2.1 Defining Model Architectures

Regardless of the type of neural network being created (e.g., feedforward or convolutional), all neural networks consist of layers transforming inputs into outputs. In PyTorch, the `nn` submodule provides definitions for layers that you can directly use to perform the transformation in question.

To define the layers that are part of your architecture, PyTorch requires that you instantiate each layer object as a member variable of your neural network class. This is done in the constructor of the new class.

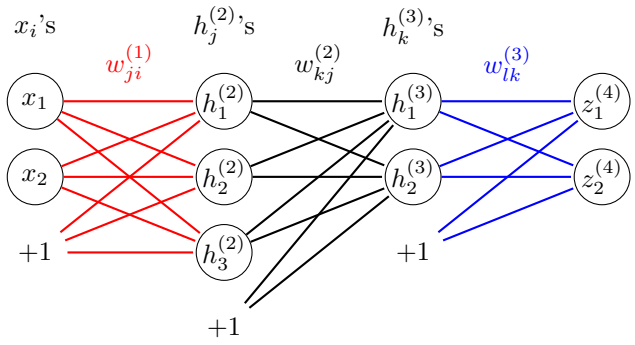
```
def __init__(self):
    super(Net, self).__init__() # call superclass constructor

    # Step 1. Define fully-connected layers for the neural network
    self.hidden1 = nn.Linear(2, 3)
    self.hidden2 = nn.Linear(3, 2)
    self.output = nn.Linear(2, 2)
```

In the case of our simple feedforward neural network, we create 3 fully-connected layers using the `nn.Linear` object:

1. Hidden Layer 1 - takes as input a feature vector $\bar{x} \in \mathbb{R}^2$ and produces an output vector $\bar{z}^{(2)} \in \mathbb{R}^3$. Each component $z_i^{(2)}$ corresponds to the pre-activation of a single neuron in the first hidden layer. We'll look at applying activation functions later.
2. Hidden Layer 2 - takes as input the post-activations from Hidden Layer 1 and produces $\bar{z}^{(3)} \in \mathbb{R}^2$, where each component $z_i^{(3)}$ corresponds to the pre-activation of a single neuron in the second hidden layer.
3. Output Layer - takes as input the post-activations from Hidden Layer 2 and produces the outputs $\bar{z}^{(4)} \in \mathbb{R}^2$. This is an example of a neural network with a **multi-node** output, where each node corresponds to a class we'd like to predict. This style of neural network will be used in Project 2.

Notice then that the layers defined in `__init__()` match the following architecture!



> Refer to the documentation for `torch.nn` to read more about other potential layers that could be used in a PyTorch neural network architecture, including those for convolutional neural networks.

2.2 Implementing Forward Propagation

Now that we've defined the layers of our neural network, we have to tell PyTorch how to propagate input feature vectors forward through the neural network.

This implementation resides in the `forward()` function below:

```
def forward(self, x):
    # Step 2. Override the forward function to define the forward pass
    z2 = self.hidden1(x)
    h2 = F.sigmoid(z2)
    z3 = self.hidden2(h2)
    h3 = F.sigmoid(z3)
    z4 = self.output(h3)
    return z4
```

The input to this function is a *batch* of feature vectors, with each vector $\bar{x} \in \mathbb{R}^2$. Mathematically, this batch is represented as a matrix $X \in \mathbb{R}^{B \times 2}$, where B is the batch size, and each row of X is a feature vector.

The return value is the output of the neural network for each feature vector in the batch. Mathematically, this is represented as a matrix $Z^{(4)} \in \mathbb{R}^{B \times 2}$ (as we have 2 output nodes).

For simplicity, we omit the explicit batch notation in the explanation below. Going step by step through the above snippet, we can describe what each line of code does (starting with the line `z2 = self.hidden1(x)`):

1. Take the batch of feature vectors $\bar{x} \in \mathbb{R}^2$ and pass it through the first hidden layer `self.hidden1` to get the batch of pre-activations $\bar{z}^{(2)} \in \mathbb{R}^3$
2. Apply the *sigmoid* activation function to the pre-activations $\bar{z}^{(2)} \in \mathbb{R}^3$, producing the post-activations $\bar{h}^{(2)} = \sigma(\bar{z}^{(2)}) \in \mathbb{R}^3$ of the first hidden layer. `F.sigmoid` is PyTorch's implementation of the sigmoid activation function.
3. Take the outputs $\bar{h}^{(2)} \in \mathbb{R}^3$ from the first hidden layer and pass it through the second hidden layer `self.hidden2` to get the batch of pre-activations $\bar{z}^{(3)} \in \mathbb{R}^2$.
4. Apply the sigmoid activation function to the pre-activations $\bar{z}^{(3)} \in \mathbb{R}^2$, producing the post-activations $\bar{h}^{(3)} = \sigma(\bar{z}^{(3)}) \in \mathbb{R}^2$ of the second hidden layer.
5. Take the outputs $\bar{h}^{(3)} \in \mathbb{R}^3$ from the second hidden layer and pass it through the output layer `self.output` to get $\bar{z}^{(4)}$
6. Return $\bar{z}^{(4)}$

With that exercise, you can see how each line of `forward()` corresponds to a specific step of forward propagation in a feedforward neural network.

3 Loading Data

PyTorch additionally provides tools to load training and testing data via the `Dataset` and `DataLoader` classes. We will go over how to use them here.

3.1 The `Dataset` Class

The `Dataset` class is the base class for any PyTorch dataset. To create a custom dataset that inherits from `Dataset`, you must implement the following functions:

- `__init__()` - constructor for the dataset, initialize all relevant member variables for data loading
- `__len__()` - returns the number of examples in the dataset
- `__getitem__(index)` - return the $(\bar{x}, y)$ feature-label pair for the example at the given index

> All methods that start and end with __ in Python are called dunder methods, and are primarily used for operator overloading.

The example below offers a simple implementation for a custom `Dataset` object that extends the PyTorch implementation:

```
from torch.utils.data import Dataset

class ExampleDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, index):
        return self.features[index], self.labels[index]
```

We can then create a training and testing dataset as follows:

```
train_dataset = ExampleDataset(X_train, y_train)
test_dataset = ExampleDataset(X_test, y_test)
```

> Refer to the documentation for the Dataset API for more detail on different types of datasets and ways to customize them!

3.2 The `DataLoader` Class

Recall from earlier discussions that SGD can be implemented using **mini-batches**. In mini-batch SGD, the estimate for $\nabla_{\bar{\theta}} R_n(\bar{\theta})$ is made using a small random subset of examples (called a **mini-batch**) rather than a single example. PyTorch automatically handles batching of examples via the `DataLoader` class.

Constructing a `DataLoader` requires passing in a subclass of `Dataset`, along with the batch size B required. Assuming that we use the `train_dataset` and `test_dataset` variables defined in the last code snippet, we can define data loaders as follows (where we use a mini-batch size of $B = 32$ examples for both):

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)
```

This means as we iterate through `train_loader` and `test_loader`, we will receive *batches* of feature vectors and labels - multiple feature vectors and labels to train on at the same time. Mathematically,

- a batch of feature vectors is represented as a matrix $X \in \mathbb{R}^{B \times d}$, where B is the batch size and d is the feature vector size. For this example, $B = 32$ and $d = 2$.
- a batch of labels is represented as a vector $\bar{y} \in \mathbb{R}^B$, where B is the batch size. For this example, $B = 32$.

As mini-batches provide more stable gradient estimates when running SGD on a neural network, we will generally use `train_loader` and `test_loader` directly when training neural networks in PyTorch.

> Refer to the Data Loader API to read about more details and customizations for use with data loaders.

4 Training Neural Networks

Once we've defined our neural network and created a `Dataset` subclass to load our training/testing data, the training process in PyTorch is comparatively simple. For this tutorial, assume we are using the following hyperparameters:

- Learning rate: $\eta = 0.01$
- Number of training epochs: 10 (loops over training data)

The training process can then be implemented via the following steps:

1. Create an instance of our neural network using the definition from Section 2.

   ```
   net  = Net()
   ```

2. Create training and testing datasets and data loaders using the definitions from Section 3.

   ```
   train_dataset = ExampleDataset(X_train, y_train)
   test_dataset = ExampleDataset(X_test, y_test)

   train_loader = DataLoader(train_dataset, batch_size=32)
   test_loader = DataLoader(test_dataset, batch_size=32)
   ```

3. Define your loss function (`criterion`) and your optimization algorithm (`optimizer`). Here we're using cross-entropy loss and SGD with the given learning rate of $\eta = 0.01$.

   ```
   import torch.optim as optim

   criterion = nn.CrossEntropyLoss()
   optimizer = optim.SGD(net.parameters(), lr=0.01)
   ```

   Note specifically that the `optim.SGD` object takes `net.parameters()` as input. This function returns a reference to the parameters of the neural network. Using this reference, the optimizer is able to access and update the parameters using SGD!

4. Define a training loop to train and evaluate the neural network:

```
for epoch in range(10):  # loop over the dataset multiple times
    total_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + update step
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        total_train_loss += loss.item()

    # the following line disables gradient calculations
    with torch.no_grad():
        # print the average test loss for the model after training on this epoch
        total_test_loss = 0
        for inputs, labels in test_loader:
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            total_test_loss += loss
    print(
        "Epoch %d : Train Loss: %.3f, Test Loss: %.3f"
        % (
            epoch + 1,
            total_train_loss / len(train_loader),
            total_test_loss / len(test_loader),
        )
    )
    running_loss = 0.0

print("Finished Training")
```

The most critical part of the training loop are the following lines:

```
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

Let's go line by line to understand what the code does (starting with the line `outputs = net(inputs)`):

1. Pass the mini-batch of input feature vectors, `inputs`, into the model `net` to perform forward propagation and get the outputs
2. Calculate the loss associated with this mini-batch of examples using the loss function `criterion`
3. Call `loss.backward()` to perform backpropagation and compute the derivatives of the loss with respect to each model parameter
4. Update model parameters using SGD by calling `optimizer.step()`

> You can find the complete tutorial here, which includes helpful tips for the project: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

4.1 Fixed epochs vs Early Stopping

Given the size of our training data, we're unlikely to have a learned a good model after a single epoch. Thus, we will have to loop over the data multiple times and train for *multiple* epochs.

In that case, how will we know on which epoch to stop training? We will use a method called **early stopping** and leverage the performance on a validation dataset.

When using early stopping, we will stop training after the validation loss does not decrease for some number of epochs. That number is defined by the hyperparameter called **patience**. For example, if our patience is 5, and validation loss does not decrease for *5* epochs, we stop training.

5 Other Helpful Tutorials

PyTorch Neural Networks:
https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

Training a Classifier (used above):
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Transfer Learning (for the challenge):
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html