PYTORCH FOR DEEP LEARNING

By the end you read this article, you will get an intuitive idea on using Pytorch for Deep Learning and by its conclusion you will be comfortable applying your deep learning models. These packages are important prerequisites and you're also familiar with Scikit-learn, Pandas, NumPy, and SciPy.

What is Deep Learning?

Deep learning is a subfield of machine learning with algorithms inspired by the working of the human brain. These algorithms are referred to as artificial neural networks. Examples of these neural networks include Convolutional Neural Networks that are used for image classification, Artificial Neural Networks and Recurrent Neural Networks.

Introduction to PyTorch

PyTorch is a Python machine learning package based on Torch, which is an open-source machine learning package based on the programming language Lua. PyTorch has two main features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Automatic differentiation for building and training neural networks

There are a few reason why one might prefer PyTorch to other deep learning libraries:

1. Unlike other libraries like TensorFlow where you have to first define an entire computational graph before you can run your model, PyTorch allows you to define your graph dynamically.
2. PyTorch is also great for deep learning research and provides maximum flexibility and speed.
3. **Python support** – As mentioned above, PyTorch smoothly integrates with the python data science stack. It is so similar to numpy that you might not even notice the difference.
4. **Dynamic computation graphs** – Instead of predefined graphs with specific functionalities, PyTorch provides a framework for us to build computational graphs as we go, and even change them during runtime. This is valuable for situations where we don't know how much memory is going to be required for creating a neural network.

PyTorch Tensors

PyTorch Tensors are very similar to NumPy arrays with the addition that they can run on the GPU. This is important because it helps accelerate numerical computations, which can increase the speed of neural networks by 50 times or greater.

- In order to use PyTorch, install by running this simple command:

```
conda install Pytorch torch vision -c pytorch
```

- In order to define a pytorch tensor, start by importing torch package:

```
import torch
```

- The default tensor type in PyTorch is a float tensor defined as **torch.FloatTensor**. As an example, you'll create a tensor from a Python list:

```
torch.FloatTensor([[20, 30, 40], [90, 60, 70]])
```

- If you're using a GPU-enabled machine, you'll define the tensor as shown below:

```
torch.cuda.FloatTensor([[20, 30, 40], [90, 60, 70]])
```

- You can also perform mathematical computations such as addition and subtraction using PyTorch tensors:

```
x = torch.FloatTensor([25])
y = torch.FloatTensor([30])
x + y
```

- You can also define matrices and perform matrix operations. Let's see how you'd define a matrix and transpose it:

```
matrix = torch.randn(4, 5)
 matrix
  matrix.t()
```

PyTorch nn Module

This is the module for building neural networks in PyTorch. nn depends on autograd to define models and differentiate them. Let's start by defining the procedure for training a neural network:

1. Define the neural network with some learnable parameters, referred to as weights.
2. Iterate over a dataset of inputs.
3. Process input through the network.
4. Compare predicted results to actual values and measure the error.
5. Propagate gradients back into the network's parameters.
6. Update the weights of the network using a simple update rule:

```
weight = weight – learning_rate * gradient
```

You'll now use the nn package to create a two-layer neural network:

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
model = torch.nn.Sequential(
torch.nn.Linear(D_in, H),
torch.nn.ReLU(),
torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss()
learning_rate = 1e-4
```

some of the parameters used above:

- N is batch size. Batch size is the number of observations after which the weights will be updated.
- D_in is the input dimension
- H is the hidden dimension
- D_out is the output dimension
- torch.randn defines a matrix of the specified dimensions
- torch.nn.Sequential initializes a linear stack of layers
- torch.nn.Linear applies a linear transformation to the incoming data
- torch.nn.ReLU applies the rectified linear unit function element-wise
- torch.nn.MSELoss creates a criterion that measures the mean squared error between n elements in the input x and target y

PyTorch optim Package

Next you'll use the optim package to define an optimizer that will update the weights for you. The optim package abstracts the idea of an optimization algorithm and provides implementations of commonly used optimization algorithms such as AdaGrad, RMSProp and Adam. We'll use the Adam optimizer, which is one of the more popular optimizers.
The first argument this optimizer takes is the tensors, which it should update. In the forward pass you'll compute the predicted y by passing x to the model. After that, compute and print the loss. Before running the backward pass, zero all the gradients for the variables that will be updated using the optimizer. This is done because, by default, gradients are not overwritten when .backward() is called. Thereafter, call the step function on the optimizer, and this updates its parameters. How you'd implement this is shown below,

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
  y_pred = model(x)
  loss = loss_fn(y_pred, y)
  print(t, loss.item())
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

Custom nn Modules in PyTorch

Sometimes you'll need to build your own custom modules. In these cases you'll subclass the nn.Module. You'll then need to define a forward that will receive input tensors and produce output tensors. How to implement a two-layer network using nn.Module is shown below. The model is very similar to the one above, but the difference is you'll use torch.nn.Module to create the neural network. The other difference is the use of stochastic gradient descent optimizer instead of Adam. You can implement a custom nn module as shown below:

```
import torch
class TwoLayerNet(torch.nn.Module):
```

```
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
model = TwoLayerNet(D_in, H, D_out)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)

    loss = criterion(y_pred, y)
    print(t, loss.item())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Building a neural network in Numpy vs. PyTorch

NUMPY

```
## Neural network in numpy

import numpy as np
#Input array
X=np.array([[1,0,1,0],[1,0,1,1],[0,1,0,1]])
#Output
y=np.array([[1],[1],[0]])
#Sigmoid Function
def sigmoid (x):
 return 1/(1 + np.exp(-x))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
 return x * (1 - x)
#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
for i in range(epoch):
  #Forward Propogation
  hidden_layer_input1=np.dot(X,wh)
  hidden_layer_input=hidden_layer_input1 + bh
```

```
  hiddenlayer_activations = sigmoid(hidden_layer_input)
  output_layer_input1=np.dot(hiddenlayer_activations,wout)
  output_layer_input= output_layer_input1+ bout
  output = sigmoid(output_layer_input)
  #Backpropagation
  E = y-output
  slope_output_layer = derivatives_sigmoid(output)
  slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)
  d_output = E * slope_output_layer
  Error_at_hidden_layer = d_output.dot(wout.T)
  d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
  wout += hiddenlayer_activations.T.dot(d_output) *lr
  bout += np.sum(d_output, axis=0,keepdims=True) *lr
  wh += X.T.dot(d_hiddenlayer) *lr
  bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print('actual :\n', y, '\n')
print('predicted :\n', output)
```

Pytorch

```
## neural network in pytorch
import torch
#Input array
X = torch.Tensor([[1,0,1,0],[1,0,1,1],[0,1,0,1]])
#Output
y = torch.Tensor([[1],[1],[0]])
#Sigmoid Function
def sigmoid (x):
  return 1/(1 + torch.exp(-x))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
  return x * (1 - x)
#Variable initialization
epoch=5000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = X.shape[1] #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
#weight and bias initialization
wh=torch.randn(inputlayer_neurons, hiddenlayer_neurons).type(torch.FloatTensor)
bh=torch.randn(1, hiddenlayer_neurons).type(torch.FloatTensor)
wout=torch.randn(hiddenlayer_neurons, output_neurons)
bout=torch.randn(1, output_neurons)
for i in range(epoch):
  #Forward Propogation
  hidden_layer_input1 = torch.mm(X, wh)
  hidden_layer_input = hidden_layer_input1 + bh
  hidden_layer_activations = sigmoid(hidden_layer_input)
   output_layer_input1 = torch.mm(hidden_layer_activations, wout)
  output_layer_input = output_layer_input1 + bout
  output = sigmoid(output_layer_input1)
 #Backpropagation
```

```
  E = y-output
  slope_output_layer = derivatives_sigmoid(output)
  slope_hidden_layer = derivatives_sigmoid(hidden_layer_activations)
  d_output = E * slope_output_layer
  Error_at_hidden_layer = torch.mm(d_output, wout.t())
  d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer
  wout += torch.mm(hidden_layer_activations.t(), d_output) *lr
  bout += d_output.sum() *lr
  wh += torch.mm(X.t(), d_hiddenlayer) *lr
  bh += d_output.sum() *lr
print('actual :\n', y, '\n')
print('predicted :\n', output)
```

Putting it all Together

PyTorch allows you to implement different types of layers such as convolutional layers, recurrent layers, and linear layers, among others.