03_how_to_use_pytorch

September 29, 2021

1 How to use PyTorch

Pytorch has been developed at the Facebook AI Research group led by Yann LeCunn and the first alpha version released in September 2016. It provides deep integration with Python libraries like Numpy that can be used to extend its functionality, strong GPU acceleration, and automatic differentiation using its autograd system. It provides more granular control than Keras through a lower-level API and is mainly used as a deep learning research platform but can also replace NumPy while enabling GPU computation.

It employs eager execution, in contrast to the static computation graphs used by, e.g., Theano or TensorFlow. Rather than initially defining and compiling a network for fast but static execution, it relies on its autograd package for automatic differentiation of Tensor operations, i.e., it computes gradients 'on the fly' so that network structures can be partially modified more easily. This is called define-by-run, meaning that backpropagation is defined by how your code runs, which in turn implies that every single iteration can be different. The PyTorch documentation provides a detailed tutorial on this.

- PyTorch Documentation
- PyTorch Tutorials

The resulting flexibility combined with an intuitive Python-first interface and speed of execution have contributed to its rapid rise in popularity and led to the development of numerous supporting libraries that extend its functionality.

1.1 Imports & Settings

[2]:
```
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
%matplotlib inline
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_circles

import torch
import torch.utils.data as utils
import torch.nn as nn
from torch.autograd import Variable
```

```python
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
from livelossplot import PlotLosses
```

[4]:
```python
input_size = 2          # Input data dimensionality
hidden_size = 3         # The number of nodes at the hidden layer
num_classes = 2         # The number of output classes
num_epochs = 20          # The number of times entire dataset is trained
batch_size = 20         # The size of input data for one iteration
learning_rate = 0.01  # The speed of convergence
```

1.2 Create Data

1.2.1 Create Random Data

[5]:
```python
# dataset params
N = 50000
factor = 0.1
noise = 0.1
```

[6]:
```python
# generate data
X, y = make_circles(
    n_samples=N,
    shuffle=False,
    factor=factor,
    noise=noise)
```

1.2.2 Create Torch Tensors

We begin by converting the NumPy or pandas input data to torch Tensors. Conversion from and to Numpy is very straightforward:

[7]:
```python
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y)
```

[8]:
```python
X_tensor.shape, y_tensor.shape
```

[8]:
```
(torch.Size([50000, 2]), torch.Size([50000]))
```

1.2.3 Create Torch Dataset

We can use these PyTorch Tensor to instantiate first a TensorDataset and, in a second step, a DataLoader that includes information about batch_size:

[9]:
```python
dataset = utils.TensorDataset(X_tensor,y_tensor)
```

1.2.4 Define Torch DataLoader

[10]:
```
dataloader = utils.DataLoader(dataset,
                              batch_size=batch_size,
                              shuffle=True)
```

1.3 Build Network

1.3.1 Architecture

PyTorch defines a NN architecture using the Net() class. The central element is the forward function. autograd automatically defines the corresponding backward function that computes the gradients.

Any legal Tensor operation is fair game for the forward function, providing a log of design flexibility. In our simple case, we just link the Tensor through functional input-output relations after initializing their attributes.

[11]:
```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()                    # Inherited from the␣
↪parent class nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.logistic = nn.LogSigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        """Forward pass: stacking each layer together"""
        out = self.fc1(x)
        out = self.logistic(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out
```

[14]:
```
net = Net(input_size, hidden_size, num_classes)
net
```

[14]:
```
Net(
  (fc1): Linear(in_features=2, out_features=3, bias=True)
  (logistic): LogSigmoid()
  (fc2): Linear(in_features=3, out_features=2, bias=True)
  (softmax): Softmax()
)
```

[19]:
```
from pprint import pprint
```

[20]:
```
pprint(list(net.parameters()))
```

```
[Parameter containing:
tensor([[ 0.3008, -0.2117],
        [-0.5846, -0.1690],
        [-0.6639,  0.1887]], requires_grad=True),
 Parameter containing:
tensor([-0.5389,  0.2994,  0.1004], requires_grad=True),
 Parameter containing:
tensor([[-0.5413, -0.4858,  0.5115],
        [-0.4672,  0.0760, -0.2340]], requires_grad=True),
 Parameter containing:
tensor([0.0005, 0.3902], requires_grad=True)]
```

[22]:
```
list(net.parameters())[0]
```

[22]:
```
Parameter containing:
tensor([[ 0.3008, -0.2117],
        [-0.5846, -0.1690],
        [-0.6639,  0.1887]], requires_grad=True)
```

1.3.2 Enable GPU

To enable GPU processing, you can use net.cuda(). See Pytorch docs for placing Tensors on CPU and/or one or more GPU units.

[81]:
```
# net.cuda()
```

1.3.3 Define Loss Function

We also need to define a loss function and the optimizer, using some of the built-in options:

[24]:
```
criterion = nn.CrossEntropyLoss()
```

1.3.4 Select Optimizer

[25]:
```
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

1.4 Train Model

1.4.1 Basic Training

Model training consists in an outer loop for each epoch, i.e., each pass over the training data, and an inner loop over the batches produced by the DataLoader. That executes the forward and backward passes of the learning algorithm. Some care needs to be taken to adjust data types to the requirements of the various objects and functions, e.g. labels need to be integers and the features should be of type floats:

[26]:
```
for epoch in range(num_epochs):
    print(epoch)
```

```
    for i, (features, label) in enumerate(dataloader):

        features = Variable(features.float())
        label = Variable(label.long())

        # Initialize the hidden weights
        optimizer.zero_grad()

        # Forward pass: compute output given features
        outputs = net(features)

        # Compute the loss
        loss = criterion(outputs, label)
        # Backward pass: compute the gradients
        loss.backward()
        # Update the weights
        optimizer.step()
```

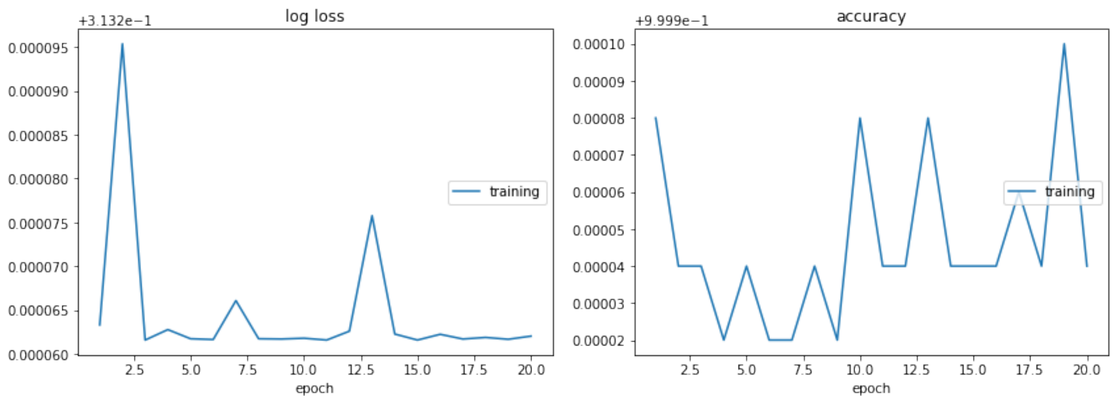
```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

1.4.2 Plotting losses in real time

Below is an example that uses the `livelossplot` package to plot losses throughout the training process as provided by Keras out of the box.

[35]:
```
liveloss = PlotLosses()
for epoch in range(num_epochs):
    print(epoch)
    logs = {}
```

```
    running_loss = 0.0
    running_corrects = 0
    for i, (features, label) in enumerate(dataloader):

        features = Variable(features.float())
        label = Variable(label.long())

        # Intialize the hidden weight to all zeros
        optimizer.zero_grad()

        # Forward pass: compute the output class given a image
        outputs = net(features)

        # Compute the loss: difference between the output class and the␣
↪pre-given label
        loss = criterion(outputs, label)
        # Backward pass: compute the weight
        loss.backward()
        # Optimizer: update the weights of hidden nodes
        optimizer.step()

        _, preds = torch.max(outputs, 1)
        running_loss += loss.detach() * features.size(0)
        running_corrects += torch.sum(preds == label.data)

        epoch_loss = running_loss / len(dataset)
        epoch_acc = running_corrects.float() / len(dataloader.dataset)
        logs['log loss'] = loss.item()
        logs['accuracy'] = epoch_acc.item()

    liveloss.update(logs)
    liveloss.draw()
```



```
log loss:
training   (min:    0.313, max:    0.313, cur:    0.313)

accuracy:
training   (min:    1.000, max:    1.000, cur:    1.000)
```

1.5 Predict

To obtain predictions from our trained model, we pass it feature data and convert the prediction to a Numpy array. We get softmax probabilities for each of the two classes:

[27]:
```python
test_value = Variable(torch.from_numpy(X)).float()
prediction = net(test_value).data.numpy()
```

[36]:
```python
prediction.shape
```

[36]:
```
(50000, 2)
```

From here on, we can proceed as before to compute loss metrics or visualize the result that again reproduces a version of the decision boundary we found above.

1.5.1 Score Prediction

[29]:
```python
accuracy_score(y_true=y, y_pred=np.argmax(prediction, axis=1))
```

[29]:
```
0.99998
```

1.6 Visualize Decision Boundary

1.6.1 Create Feature Space

[30]:
```python
n_vals = 200
x1 = np.linspace(-1.5, 1.5, num=n_vals)
x2 = np.linspace(-1.5, 1.5, num=n_vals)
xx, yy = np.meshgrid(x1, x2)  # create the grid
```

[31]:
```python
X_test = np.array([xx.ravel(), yy.ravel()]).T
X_test = torch.from_numpy(X_test)
X_test.shape
```

[31]:
```
torch.Size([40000, 2])
```

1.6.2 Predict Feature Space

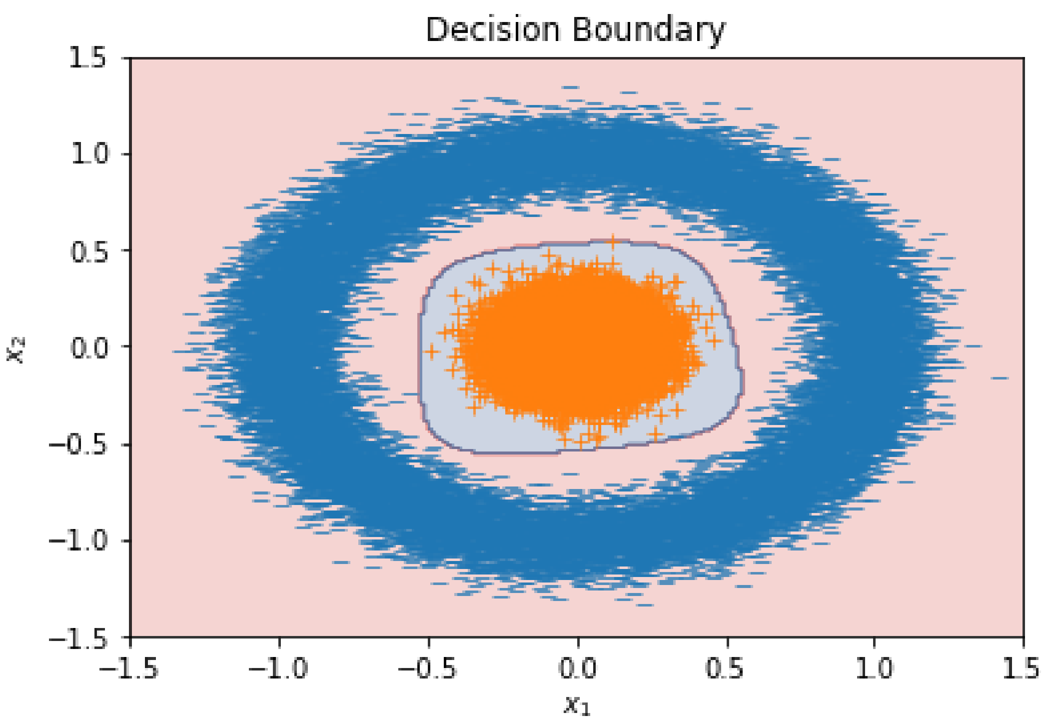
[32]:
```python
zz = net(Variable(X_test).float()).data.numpy()
zz.shape
```

[32]:
```
(40000, 2)
```

[33]:
```
# Create a color map to show the classification colors of each grid point
cmap = ListedColormap([sns.xkcd_rgb["pale red"],
                       sns.xkcd_rgb["denim blue"]])
```

1.6.3 Plot Decision Boundary

[34]:
```
# Plot the classification plane with decision boundary and input samples
plt.contourf(xx, yy, np.argmax(zz, axis=1).reshape(n_vals, -1), cmap=cmap,␣
↪alpha=.25)

# Plot both classes on the x1, x2 plane
data = pd.DataFrame(X, columns=['$x_1$', '$x_2$']).assign(Class=pd.Series(y).
↪map({0:'negative', 1:'positive'}))
sns.scatterplot(x='$x_1$', y='$x_2$', hue='Class', data=data, style=y,␣
↪markers=['_', '+'], legend=False)
plt.title('Decision Boundary')
plt.savefig('boundary', dpi=300);
```



[ ]:
```
```