University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 5, 6: September 11, 13

Lecturer: Alexander Schwing Scribe: David Shan

1 Installing PyTorch

`https://pytorch.org/` makes it easy - just select the configuration you want and run the provided command. It may help to have Anaconda/Miniconda installed beforehand.

2 Intro to PyTorch

PyTorch is a python package that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autodiff system

PyTorch makes it easy to run things on the GPU, easy to create complicated machine learning models that use gradient-based learning algorithms, and it already has many popular models, learning algorithms, and loss functions implemented. Usually one uses PyTorch either as:

- A replacement for numpy to use the power of GPUs.
- a deep learning research platform that provides maximum flexibility and speed

PyTorch is the integration of the Torch framework (used for Lua) for the Python language. It is most notable for its use of a Dynamic Computation Graph, which provides a flexible interface for defining and modifying a graph on-the-go. On the other hand, frameworks like TensorFlow define a static graph for computations.

3 Tensors

3.1 Basics

PyTorch uses Tensors to represent all types of data, model parameters, etc. A PyTorch Tensor is fundamentally very similar to **numpy** arrays, providing many of the same functions for operating on Tensors as numpy does. Tensors can live either on the CPU or GPU and accelerate computation by a great amount. Tensors come in several different types that include

- `torch.FloatTensor`
- `torch.LongTensor`
- `torch.ByteTensor`

and more (e.g. ShortTensor, IntTensor, etc).

There are a number of ways to initialize a tensor:

- `torch.FloatTensor(2,3,4)` creates an empty 2x3x4-dimensional tensor
- `torch.FloatTensor([[1, 2], [3,4]])` creates a 2x2 matrix with first row [1, 2] and second row [3,4]
- `torch.zeros(3)` creates a 1-d tensor filled with zeros
- `torch.ones(4,5,6)` creates a 4x5x6-dimensional tensor filled with ones.

Tensors may also be generated from random distributions as well.

- `torch.empty(3,4).uniform_(0, 1)`

Tensors also support many Python/numpy-like operations like slicing and indexing. Just like numpy arrays, Tensors support multidimensional indexing for multidimensional arrays, That means that it is not necessary to separate each dimensions index into its own set of square brackets. Instead, we can use a comma separated values.

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1, 2])
tensor(6)
>>> x[0, 1] = 8
>>> print(x)
tensor([[ 1,  8,  3],
        [ 4,  5,  6]])
```

The syntax for slicing is just like for numpy arrays: `i:j:k` where `i` is the starting index, `j` is the stopping index, and `k` is the step size.

```
>>> x = torch.Tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]

 1
 3
 5
[torch.FloatTensor of size 3]
```

Use `torch.Tensor.item()` to extract the value(s) from a Tensor.

```
>>> x = torch.tensor(1)
>>> x
tensor(1)
>>> x.item()
1
```

3.2 Tensor Computations

Many standard operators are overloaded for tensor computation, e.g.

```
result = tensor_1 + tensor_2
```

However, many standard functions are also implemented for tensors, e.g.

```
result = tensor_1.mean()- tensor_2.sum()
```

3.3 Exercise: Basic Computation

Here is an example for some basic computations that can be performed:

Create a rank-2 tensor whose first row is [1,5,3,7] and whose second row is [2,4,1,10]. Find the sum of each row, and take the mean of these two values. Then, divide by the standard deviation of the maximum value of each column.

```
import torch

inp = torch.FloatTensor([[1,3,5,7],
                         [2,4,1,10]])
sums = inp.sum(dim=1)
mean_sum = sums.mean()
max_vals, _ = inp.max(dim=0)
stddev = max_vals.std()
result = mean_sum/stddev

print("INPUT:␣",inp)
# INPUT:  tensor([[ 1.,  3.,  5.,  7.], [ 2.,  4.,  1., 10.]])
print("SUM␣OF␣ROWS:␣",sums)
# SUM OF ROWS:  tensor([16., 17.])
print("MEAN␣OF␣SUMS:␣",mean_sum)
# MEAN OF SUMS:  tensor(16.5000)
print("MAX␣VALS␣OF␣COLUMNS:␣",max_vals)
# MAX VALS OF COLUMNS:  tensor([ 2.,  4.,  5., 10.])
print("STDDEV␣OF␣MAX␣VALS:␣",stddev)
# STDDEV OF MAX VALS:  tensor(3.4034)
print("FINAL␣RESULT:␣",result)
# FINAL RESULT:  tensor(4.8481)
```

4 Automatic Differentiation

Manually implementing backward pass can become tedious with larger, more complex networks. PyTorch's **autograd** package automates the computation of backward passes in a computation graph. By default, the computation graph is not maintained for a given tensor. To do so, write the following line of code:

```
tensor.requires_grad = True
```

And PyTorch will keep track of all of the operations computed downstream of this tensor.

Calling the backward function (`obj.backward()`) receives the gradient of the output Tensors with respect to some scalar value, and computes the gradient of the input Tensors with respect to that same scalar value.

5 Optimizers

PyTorch has numerous optimizers like AdaGrad, RMSProp, Adam, etc. implemented in its **optim** package. Key to using them:

- `opt = torch.optim.SGD([params], lr = 0.01)`

- `opt.zero_grad()` zeroes out all gradients of your parameters being updated.
- `opt.step()` takes one step of the algorithm.

The model parameters are passed in to be updated at every iteration. More complex methods such as per-layer or even per-parameter learning rates and also be specified.

5.1 Exercise: Minimize Function via Gradient Descent

Here is an example for implementing gradient descent:

Find the minimum of the following function using PyTorch:
$f(x, y) = 3x^2 + 1.5y^2 - 2.3xy + 5.6x + 0.7y + 1.2$

```
import torch

def fn(inp):
    x = inp[0]
    y = inp[1]
    return 3*x**2 + 1.5*y**2 - 2.3*x*y + 5.6*x + 0.7*y + 1.2

val = torch.FloatTensor([1,1])
val.requires_grad = True
lr = 0.1
opt = torch.optim.SGD([val], lr=lr)

for i in range(100):
    opt.zero_grad()
    obj = fn(val)
    print("\t%d: %f"%(i, obj.item()))
    obj.backward()
    opt.step()

print("FINAL PREDICTION: ",val)
# FINAL PREDICTION:  tensor([-1.4485, -1.3438], requires_grad=True)
```

6 Modules

PyTorch's **nn.Module** allows specifying models that are more complex than a sequence of existing Modules. You can define your own class by extending the `torch.nn.Module` class and defining a forward which receives input Tensors and produces output Tensors. To define a model, we first specify the parameters of the model, and then outline how they are applied to the inputs. For operations that do not involve trainable parameters (activation functions such as ReLU, operations like maxpool), we generally use the `torch.nn.functional` module.

- A special tensor subclass - `torch.nn.Parameter` - should be used when creating learnable model parameters
- You can also assign other Modules to be a part of your module
- The forward method defines how you obtain output from your model

6.1 Exercise: Minimize Function via Gradient Descent

Looking at the previous example, we can now implement x and y as parameters of a model.

```
import torch
from torch import nn

class FnModule(nn.Module):
    def __init__(self):
        super(FnModule, self).__init__()
        #self.params = nn.Parameter(torch.ones(2))
        self.x = nn.Parameter(torch.ones(1))
        self.y = nn.Parameter(torch.ones(1))

    def forward(self):
        #x = self.params[0]
        #y = self.params[1]
        return 3*self.x**2 + 1.5*self.y**2 - 2.3*self.x*self.y + 5.6*self.
            x + 0.7*self.y + 1.2

model = FnModule()
lr = 0.1
opt = torch.optim.SGD(model.parameters(), lr=lr)

for i in range(100):
    opt.zero_grad()
    obj = model()
    print("\t%d: %f"%(i, obj.item()))
    obj.backward()
    opt.step()

print("FINAL PREDICTION: ",list(model.parameters()))
# FINAL PREDICTION:  [Parameter containing: tensor([-1.4485],
   requires_grad=True), Parameter containing: tensor([-1.3438],
   requires_grad=True)]
```

7 Loss Functions

In PyTorch, the **nn** package defines a set of useful, commonly used loss functions. However, you can, of course, create losses by manually defining functions. A general rule of thumb is to check to see if something has already been implemented before trying yourself. Some loss functions include:

- L1Loss
- MSELoss
- CrossEntropyLoss

and many more. We can define a loss function as follows:

```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(out, labels)
```

We can also write our own Cross Entropy Loss function as follows:

```
import torch.nn.functional as F

def myCrossEntropyLoss(outputs, labels):
    batch_size = outputs.size()[0]            # batch_size
    outputs = F.log_softmax(outputs, dim=1)   # compute the log of softmax
        values
    outputs = outputs[range(batch_size), labels] # pick the values
       corresponding to the labels
    return -torch.sum(outputs)/num_examples
```

8 Training vs Evaluation

It is imperative that you call `model.train()` before training the model so that dropout, batch normalization, etc. are enabled. Likewise, `model.eval()` must be called before testing or evaluating the model.

9 Datasets

PyTorch makes it easy to manage data with its **Dataset** class. Data management is comprised of two main classes:

- `torch.utils.data.Dataset` represents the actual dataset
- `torch.utils.data.DataLoader` is how you access mini-batches of your data during training

To create a custom dataset, the custom class must inherit `Dataset` and override the following methods:

- `__len__` so that `len(dataset)` returns the size of the dataset.
- `__getitem__` to support the indexing such that `dataset[i]` can be used to get ith sample

`torch.utils.data.DataLoader` is an iterator that provides features such as:

- Batching data
- Shuffling data
- Loading data in parallel using `multiprocessing` workers

We can initialize a Dataloader as following:

```
dataloader = DataLoader(dataset, batch_size=4,
                        shuffle=True, num_workers=4)
```

9.1 Exercise: Create a Dataset!

The dataset consists of the following:

- Inputs: 4-d vectors sampled from the standard gaussian distribution

- Labels: scalar-valued tensors containing the index of the largest value of the input

Implement this so that the number of datapoints is an argument to the constructor for the dataset. Additionally, try using a dataloader to loop through the dataset!

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

class RandomDataset(Dataset):
    def __init__(self, num_data_points):
        self.inputs = torch.FloatTensor(num_data_points, 4).normal_()
        self.outputs = self.inputs.argmax(dim=1)

    def __len__(self):
        return self.inputs.size(0)

    def __getitem__(self, idx):
        return self.inputs[idx, :], self.outputs[idx]


sample_dataset = RandomDataset(10)
dataloader = DataLoader(sample_dataset, batch_size=4)

for inputs, outputs in dataloader:
    print(inputs, outputs)

# tensor([[ 0.9452,  0.6709, -0.9755,  0.7572],
#         [-0.5098, -0.5520,  0.9007, -0.8184],
#         [-0.5248, -0.0029,  0.6773,  1.2228],
#         [-1.3602,  1.7400, -1.3610,  0.9928]]) tensor([0, 2, 3, 1])
# tensor([[-1.7760,  0.1103,  0.9949, -0.4992],
#         [ 0.2422,  0.1627,  0.8432, -0.0862],
#         [ 2.0121,  0.9124, -0.0541,  1.4404],
#         [ 2.0974,  0.2716,  0.4421, -0.5181]]) tensor([2, 2, 0, 0])
# tensor([[ 0.7935,  0.6498, -0.3527, -0.6235],
#         [-1.0466, -2.6111,  0.1390, -0.8698]]) tensor([0, 2])
```

10 Exercise: Lets do Linear Regression!

Download data from
`https://courses.engr.illinois.edu/ece544na/fa2018/secure/linear_regression.data`
`Inputs, outputs = torch.load(linear_regression.data)`

- Inputs are 10-d real-valued features, outputs are 1-d real-valued
- Create a linear model and learn a model that minimizes squared loss between predictions and outputs
- Try to implement this using a Module for your model and a Dataset wrapping the provided dataset

```
import torch
from torch import nn

from torch.utils.data import Dataset, DataLoader

class SampleData(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels
        self.labels.requires_grad = False

    def __len__(self):
        return self.data.size(0)

    def __getitem__(self, idx):
        return self.data[idx, :], self.labels[idx]

class OurModel(nn.Module):
    def __init__(self, feature_size):
        super(OurModel, self).__init__()
        self.model = nn.Linear(feature_size, 1)

    def forward(self, inp):
        return self.model(inp)

data, labels = torch.load('linear_regression.data')
dataset = SampleData(data, labels)

model = OurModel(data.size(1))
lr = 1e-3
opt = torch.optim.SGD(model.parameters(), lr=lr)

dataloader = DataLoader(dataset, batch_size=10)

for i in range(1000):
    for inputs, labels in dataloader:
        opt.zero_grad()
        predictions = model(inputs).squeeze()
        #loss = ((predictions-labels)**2).mean()
        loss = nn.MSELoss()(predictions, labels)
        print("%d: %f"%(i, loss.item()))
        loss.backward()
        opt.step()

print("FINAL MODEL PARAMETERS: ",list(model.parameters()))

# FINAL MODEL PARAMETERS:  [Parameter containing:
# tensor([[ 0.1661, -0.1372, -0.0586,  0.1505, -0.2956,  0.1909, -0.0631,
    0.1642, 0.0469, -0.0360]], requires_grad=True),
# Parameter containing: tensor([-0.0421], requires_grad=True)]
```

References

[1] PyTorch. Welcome to pytorch tutorials. https://pytorch.org/tutorials/.

[2] O. M. Surag Nair, Guillaume Genthial. Introduction to pytorch code examples. https://cs230-stanford.github.io/pytorch-getting-started.html.