Introduction to PyTorch

Benjamin Roth

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian-Universität München
beroth@cis.uni-muenchen.de
Why PyTorch?

- Relatively new (Aug. 2016?) Python toolkit based on Torch
- Overwhelmingly positive reception by the deep learning community. See e.g. http://www.fast.ai/2017/09/08/introducing-pytorch-for-fastai/
- *Dynamic* computation graphs:
  - “process complex inputs and outputs, without worrying to convert every batch of input into a big fat tensor”
  - E.g. sequences with different length
  - Control structures, sampling
- Flexibility to implement low-level and high-level functionality.
- Modularization uses object orientation.
Tensors

- Tensors hold data
- Similar to numpy arrays

```python
# 'Uninitialized' Tensor with values from memory:
x = torch.Tensor(5, 3)
# Randomly initialized Tensor (values in [0..1]):
y = torch.rand(5, 3)
print(x + y)
```

Output:

```
0.9404 1.0569 1.1124
0.3283 1.1417 0.6956
0.4977 1.7874 0.2514
0.9630 0.7120 1.0820
1.8417 1.1237 0.1738
[torch.FloatTensor of size 5x3]
```

- In-place operations can increase efficiency: `y.add_(x)`
- 100+ Tensor operations:
  
  [http://pytorch.org/docs/master/torch.html](http://pytorch.org/docs/master/torch.html)
import torch
a = torch.ones(5)
b = a.numpy()
print(b)

Output:

[ 1.  1.  1.  1.  1.]

import numpy as np
a = np.ones(3)
b = torch.from_numpy(a)
print(b)

Output:

1
1
1

[torch.DoubleTensor of size 3]
Automatic differentiation

- Central concept: Tensor class
- A Tensor corresponds to a node in a function graph
- If you set `my_tensor.requires_grad=True`, all operations are tracked, and gradients can be computed automatically
Functional composition

- If a Tensor was created by functional composition \((x = a + b)\), then `my_function = x.grad_fn` references the function (For example, ThAddBackward corresponds to Tensor addition)
- `x.backward()` computes the gradient for the tensor (and, recursively, for all input tensors). The values of the gradient computation are then stored in `a.grad`, `b.grad` and `x.grad`
- `my_function.forward()` method: Computes (Tensor) output value from input Tensors
- `my_function.backward()` method: Provides the gradient for the function. It is used in the recursive gradient computation \((x.backward())\) via the chain rule.
Automatic differentiation: Example

# Set requires_grad=True, if gradient is to be computed
x = Tensor(3 * torch.ones(1), requires_grad=True)
y = x + 2*x**2
y.backward()

Value of x.grad?
Defining a neural network

- A self-defined neural net should inherit from `nn.Module`
- `torch.nn` contains predefined layers:
  - `nn.Linear(input_size, output_size)`
  - `nn.Conv2d(in_channels, out_channels, kernel_size)`, ...
- Set layers as class attributes:
- All parameter Tensors get automatically registered with the neural net (can be accessed by `net.parameters()`)

- Functions without learnable parameters (`torch.nn.functional`) do not have to be registered as class attributes:
  - `relu(...)`, `tanh(...)`, ...

- Prediction needs to be implemented in `net.forward(...)`

```python
class Net(nn.Module):
    def __init__(self, num_features, hidden_size):
        super(Net, self).__init__()
        # self.learnable_layer = ...

    def forward(self, x):
        return # do prediction
```
Linear Regression

- What is layer and learnable parameters?
- How to do prediction?
import torch.nn as nn

class LinearRegression(nn.Module):
    def __init__(self, num_features):
        super(LinearRegression, self).__init__()
        self.linear_layer = nn.Linear(num_features, 1)

    def forward(self, x):
        return self.linear_layer(x)
Linear Regression: prediction for one instance (with untrained model)

- $x_{\text{instance}}$: features, torch.FloatTensor of size 10 (num_features)
- $y_{\text{instance}}$: label, torch.FloatTensor of size 1
- Type of $y_{\text{predicted}}$?

```python
code
num_features = 10
lr_model = LinearRegression(num_features)
y_predicted = lr_model.forward(x_tensor)
```

Linear Regression: training the model

- **Loss function**: Define yourself or pre-defined.
  - \( \text{loss} = (y_{\text{var}} - y_{\text{predicted}})^2 \)
  - \( \text{criterion} = \text{nn.MSELoss()} \)
    - \( \text{loss} = \text{criterion}(y_{\text{var}}, y_{\text{predicted}}) \)

- **Training update**: Define yourself or pre-defined.
  - \( \text{loss}.\text{backward()} \)
    - \( \text{for } w \text{ in lr_model.parameters():} \)
      - \( w.\text{sub_}(w.\text{grad} \ast 0.0001) \)  
        # subtract gradient
  - \( \text{optimizer} = \text{optim.SGD(lr_model.parameters(), lr=0.0001)} \)
    - \( \text{...} \)
    - \( \text{loss}.\text{backward()} \)
    - \( \text{optimizer}.\text{step()} \)

- **Note**:
  - Gradients are accumulated (added) in the Tensors for each call of \( \text{.backward()} \)
  - need to be set to zero for next gradient update
  - \( \text{optimizer}.\text{zero_grad()} \) sets gradients of all network Tensors to zero
Linear Regression: training the model

```python
lr_model = LinearRegression(num_features)
optimizer = optim.SGD(lr_model.parameters(), lr=0.0001)
criterion = nn.MSELoss()
for epoch in range(num_epochs):
    for x_instance, y_instance in data:
        y_pred = lr_model.forward(x_instance)
optimizer.zero_grad()
        loss = criterion(y_pred, y_instance)
        loss.backward()
        optimizer.step()
```

Comments:

- Here, we are using only 1 example at a time for our updates.
- Instead of using plain SGD, there are better learning methods that can adapt their learning rate per parameter (e.g. Adam)
- Question: `step()` does not take any arguments. How does it know which parameters to update?
Materials

- http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- http://pytorch.org/tutorials/beginner/pytorch_with_examples.html
- http://pytorch.org/tutorials/beginner/deep_learning_nlp_tutorial.html
Homework: Boston house prices prediction

Dataset:
- Harrison & Rubinfeld, 1978
- Predict median house price (in 1000USD) per district/town.
- 506 instances, 13 features

Features:
- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per 10,000 USD
- PTRATIO pupil-teacher ratio by town B \(1000(Bk - 0.63)^2\) where Bk is the proportion of blacks by town
- LSTAT % lower status of the population
Homework: Boston houses prediction

- Linear regression:
  \[ \hat{y} = Wx + b \]

- Neural network regression (one hidden layer, ReLu activation):
  \[ \hat{y} = W_B \text{max}(0, W_A x + b_A) + b_B \]
Summary

- PyTorch is one of the most popular deep learning frameworks.
- PyTorch Tensors are similar Numpy Arrays, but they can be combined to build function graphs.
- PyTorch can compute the gradient for you.
- For Training: Gradient of loss w.r.t. parameters. Parameter update with SGD.
- Homework: Neural network regression (contains non-linearity)