INFR10086 Machine Learning (MLG) Semester 2, 2023/4

Tutorial 3: PyTorch

In this tutorial, we will introduce PyTorch, a package for building and running deep neural networks.

Implementation. Before we begin, you will need to install PyTorch by doing the following in your terminal.

```
$ pip install torch
```

To check whether you have the package properly installed, you can run the following in your python REPL.

```
>>> import torch
>>>
```

If you see nothing, that's good news and it means the package has been insatlled properly.

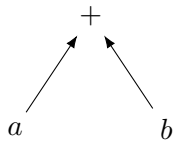
The main use of PyTorch is to automate the computation of a gradient. Up until now, we have been deriving gradients by hand. Once the parameters are nested inside several function applications, deriving gradients can be tedious and error prone. To allow PyTorch to help us derive the gradient of a function, we need to specify the function. The way we specify a function is by constructing a **computation graph**.

1 Basic elements

PyTorch operates on tensors. For the purpose of this tutorial (and this course), a tensor is simply a multidimensional array, a generalization of matrices. A vector is an order-1 tensor; a matrix is an order-2 tensor. Tensors in PyTorch are intentionally designed to be like numpy arrays.

```
>>> import torch
>>> a = torch.tensor([1, 2, 3])
>>> b = torch.tensor([4, 5, 6])
>>> a + b
[5, 7, 9]
```

To specify a function as a computation graph, we need to create vertices and edges. Every tensor is a vertex, and every operation creates a new vertex. The edges are managed internally by PyTorch, and are something we don't need to worry in most cases. The above code creates the following graph.



2 Linear regression

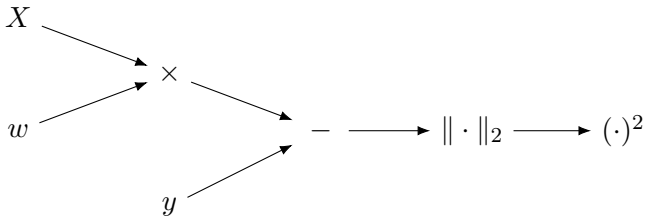
Let's start with computing the mean-squared error, something we are faimilar with. Given a data set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, the mean-squared error can be written as

$$L = \|Xw - y\|_2^2 \tag{1}$$

where

$$X = \begin{bmatrix} — & x_1 & — \\ — & x_2 & — \\ & \vdots & \\ — & x_n & — \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{2}$$

We can decompose the computation of L into the following graph.



The same math equation can be expressed in python code as `loss = torch.norm(X @ w - y) ** 2`. The python code is again intentionally similar to the math equation. Every vertex in PyTorch has a `backward()` function. We can just do `loss.backward()` to compute the gradient. Every vertex also a `grad` field, and we can use `w.grad` to get the resulting gradient. The code for computing a single gradient would become the following.

```
import torch
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
w = torch.tensor([3.0, 2.0, 1.0], requires_grad=True)
y = torch.tensor([2.0, 4.0, 8.0])
loss = torch.norm(X @ w - y) ** 2
loss.backward()
print(w.grad)
```

PyTorch has its own mean-squared error called `torch.nn.MSELoss`. Below is another version of computing a gradient with `MSELoss`.

```
import torch
from torch import nn
mse = nn.MSELoss(reduction='sum')
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
w = torch.tensor([3.0, 2.0, 1.0], requires_grad=True)
y = torch.tensor([2.0, 4.0, 8.0])
loss = mse(X @ w, y)
loss.backward()
print(w.grad)
```

Implementation. Run the above PyTorch code.

Implementation. We know that the gradient of L is $2(X^\top Xw - X^\top y)$. Run the code below. Is the result the same as the one computed with PyTorch?

```
import numpy
X = numpy.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
w = numpy.array([3.0, 2.0, 1.0])
y = numpy.array([2.0, 4.0, 8.0])
print(2 * (X.T @ (X @ w) - X.T @ y))
```

3 Logistic regression

For this example, we are going to compute the gradient of log loss. Recall that given a data set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, the log loss for binary classification is

$$L = \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^\top x_i)\right). \tag{3}$$

Note that $y_i \in \{+1, -1\}$. Also recall that the probability of y_i given x_i is defined as

$$p(y_i|x_i) = \frac{1}{1 + \exp(-y_i w^\top x_i)}. \tag{4}$$

Discussion. Show that

$$\mathbb{1}_{y_i=+1}[-\log p(y_i = 1|x_i)] + (1 - \mathbb{1}_{y_i=1})[-\log p(y_i = -1|x_i)] = -\log p(y_i|x_i), \tag{5}$$

where

$$\mathbb{1}_c = \begin{cases} 1 & \text{if } c \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

In general $\mathbb{E}_{z\sim q(z)}[-\log p(z)] = \sum_z q(z)[-\log p(z)]$ is called the cross entropy of p relative to q. Can you see that the left hand side is cross entropy?

In PyTorch, the labels are integers counting from 0, so for binary classification, the classes are 0 and 1 (as opposed to +1 and −1). In principle, one weight vector is enough to do binary classification. If you can compute $p(y_i = +1|x_i)$, then $p(y_i = -1|x_i)$ has to be $1 - p(y_i = +1|x_i)$. PyTorch uses the log of softmax to compute cross entropy, and only takes as input the results of the dot product. In this case, it is typical to use two weight vectors (one weight vector per class).

```
import torch
from torch import nn
ce = nn.CrossEntropyLoss(reduction='sum')
x_i = torch.tensor([1.0, 2.0, 3.0])
w = torch.tensor([[4.0, 5.0, 6.0], [7.0, 8.0, 9.0]], requires_grad=True)
y_i = torch.tensor(1)
loss = ce(w @ x_i, y_i)
print(loss)
```

Discussion. Why is the cross entropy loss above 0, when `y_i = torch.tensor(1)`?

Implementation. Change the code above and let `y_i = torch.tensor(0)`. What is the loss now?

Discussion. Recall that if we express the probability with softmax

$$p(y = +1|x) = \frac{\exp(w_{+1}^\top x)}{\exp(w_{+1}^\top x) + \exp(w_{-1}^\top x)} = \frac{1}{1 + \exp(w_{-1}^\top x - w_{+1}^\top x)} \tag{7}$$

Run the following code.

```
import numpy
import math
x_i = numpy.array([1.0, 2.0, 3.0])
w = numpy.array([[4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
s = w @ x_i
print(s)
loss = math.log(1 + math.exp(s[1] - s[0]))
print(loss)
```

Is this the same loss value that you get when you let `y_i = torch.tensor(0)`?

Instead of writing our own matrix multiplication, it is more common to use the `torch.nn.Linear` in PyTorch. The matrix `w` in our code above is 2×3, but in `torch.nn.Linear` the input dimension goes before the output dimension. Instead of using `@` for matrix multiplication as in `w @ x_i`, `torch.nn.Linear` actually needs to be called like a function, as in `w(x_i)`.

```
import torch
from torch import nn
ce = nn.CrossEntropyLoss(reduction='sum')
```

```
w = nn.Linear(3, 2)

x_i = torch.tensor([1.0, 2.0, 3.0])
y_i = torch.tensor(0)
loss = ce(w(x_i), y_i)
print(loss)
y_i = torch.tensor(1)
loss = ce(w(x_i), y_i)
print(loss)
```

Discussion. Run the above python code a few times. You will observe that it gives a different result each time. What is going on?

Discussion. We can print the parameters in `torch.nn.Linear` by calling `parameters()`. Run the following code to inspect the parameters. What are the parameters?

```
import torch
from torch import nn
w = nn.Linear(3, 2)
print(list(w.parameters()))
```

Discussion. Do you think it is correct to say that `torch.nn.Linear` is linear? Or is it actually affine?

4 Stochastic Gradient Descent

We will train classifiers on real data sets in another tutorial. For now, we will focus on getting familiar with the basic usage of PyTorch. Below is a minimal example of stochastic gradient descent.

```
import torch
from torch import nn
from torch import optim

ce = nn.CrossEntropyLoss(reduction='sum')
w = nn.Linear(3, 2)
opt = optim.SGD(w.parameters(), lr=0.1)

for i in range(20):
    opt.zero_grad()

    # fake data
    x_i = torch.rand(3)
    y_i = torch.tensor(i % 2)

    loss = ce(w(x_i), y_i)
```

```
print(loss)
loss.backward()

opt.step()
```

Discussion. What is the step size of stochastic gradient descent in the above code?

Discussion. A common mistake is to forget `zero_grad()` when writing a training script. In fact, if you comment out that line, the above code runs just fine. What do you think `zero_grad()` does to the computation graph? Why is it a problem if we forget to write that line?