INFR10086 Machine Learning (MLG) Semester 2, 2024/5

Tutorial 3: PyTorch

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

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.

$$\begin{tikzcd} & + & \\ a \arrow[ur] & & b \arrow[ul] \end{tikzcd}$$

2 Maximum likelihood estimation of a Gaussian mean

Recall that given a data set $\{x_1, x_2, \dots, x_n\}$ of n i.i.d. Gaussian samples, the log likelihood of the Gaussian mean is

$$\log p(x_1, \dots, x_n) = \log \prod_{i=1}^{n} p(x_i) = \sum_{i=1}^{n} \left[-\frac{d}{2} \log 2\pi - \frac{1}{2} \log |\Sigma| - \frac{1}{2}(x_i - \mu)^\top \Sigma^{-1} (x_i - \mu) \right]. \tag{1}$$

To simplify this tutorial, let's focus on the case where $\Sigma = \sigma^2 I$. We end up with

$$\log p(x_1, \dots, x_n) = -\frac{nd}{2} \log 2\pi - \frac{nd}{2} \log \sigma^2 - \frac{1}{2\sigma^2} \sum_{i=1}^{n} \|x_i - \mu\|^2. \tag{2}$$

Since we are going to minimize a function, we implement the negative log likeliohod as follows.

```
def neg_log_likelihood(samples, mu, var_):
    n, d = samples.shape

    return (d / 2.0 * math.log(2 * math.pi) + d / 2.0 * math.log(var_)
        + 0.5 / var_ * torch.linalg.matrix_norm(samples - mu) ** 2 / n)
```

Note that we have also divided the negative log likelihood by n, to make the objective an average. Instead of computing the gradient of the log likelihood by hand, we can instead call the `backward` function and ask PyTorch to compute it automatically for us.

```
loss = neg_log_likelihood(samples, mu, var_)
loss.backward()
```

Discussion. If we compare the code and the math, explain how

$$\frac{1}{n} \frac{1}{2\sigma^2} \sum_{i=1}^{n} \|x_i - \mu\|^2 \tag{3}$$

is implemented as `0.5 / var_ * torch.linalg.matrix_norm(samples - mu) ** 2 / n`.

Discussion. What does the computation graph look like for `neg_log_likelihood`?

In class, we have derived the optimal solution for the likelihood, and we know the maximum likelihood estimator is

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{4}$$

In this tutorial, we are going to pretend that we do not know this, and will use gradient descent to find the maximum likelihood estimator.

To pretend that we have a data set, we generate 500 random Gaussian samples in 100-dimensional space with a mean of 0 and a variance of 1.0. We know that the maximum likelihood estimate is exactly 0, but let's see whether gradient descent is able to find it.

Run the following code and observe its output.

```
import torch
import numpy
import math

def neg_log_likelihood(samples, mu, var_):
    n, d = samples.shape

    return (d / 2.0 * math.log(2 * math.pi) + d / 2.0 * math.log(var_)
        + 0.5 / var_ * torch.linalg.matrix_norm(samples - mu) ** 2 / n)

var_ = 1.0
d = 100
samples = torch.tensor(numpy.random.standard_normal((500, d)) * math.sqrt(var_))

mu = torch.nn.Parameter(torch.zeros(d))
torch.nn.init.uniform_(mu)
opt = torch.optim.SGD([mu], lr=0.2)

for i in range(30):
    opt.zero_grad()
    loss = neg_log_likelihood(samples, mu, var_)
    print('loss={:.8}, norm(mu)={:.4}'.format(loss.item(),
        torch.linalg.vector_norm(mu).item()))
    loss.backward()
    opt.step()
```

Discussion. We are using the class `torch.optim.SGD`, but is this really stochastic gradient descent?

Discussion. Where is the step size in the above code?

Discussion. A common mistake is to forget `zero_grad()` when writing a training script. What do you think `zero_grad()` does to the computation graph? Why is it a problem if we forget to write that line?

Discussion. Does gradient descent actually find the optimal solution?

3 Logistic regression

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

$$L = \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^\top x_i)\right). \tag{5}$$

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

$$p(y_i|x_i) = \frac{1}{1 + \exp(-y_i w^\top x_i)}. \tag{6}$$

Discussion. Show that

$$\mathbb{1}_{y_i=1}[-\log p(y_i = 1|x_i)] + (1 - \mathbb{1}_{y_i=1})[-\log p(y_i = -1|x_i)] = -\log p(y_i|x_i), \tag{7}$$

where

$$\mathbb{1}_c = \begin{cases} 1 & \text{if } c \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

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).

```python
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. Explain why the above cross entropy loss is 0, when `y_i = torch.tensor(1)`. 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

$$p(y = +1|x) = \frac{\exp(w_{+1}^\top x)}{\exp(w_{+1}^\top x) + \exp(w_{-1}^\top x)} = \frac{1}{1 + \exp(w_{-1}^\top x - w_{+1}^\top x)} \tag{9}$$

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?