2: Linear Regression in Pytorch, the Dumb Way

Aniruddh K Budhgavi
Enigma, IIIT-B

June 2, 2020

1 Introduction

Constructing machine learning models using Numpy is a fantastic way of understanding the nitty-gritties of a particular method, but it is not an ideal way for more complex models or in production environments. Instead, we use libraries like Pytorch, Tensorflow and Keras. These libraries ensure that you spend less time reinventing the wheel, leaving you free to fine-tune the model without getting bogged down by details. They also make your training process faster by utilizing the GPU (if you have one).
My personal preference is towards Pytorch (anyone who has used Tensorflow 1.X will understand why). Pytorch has enough utilities to make the process painless but is flexible when it comes to customizations.

2 Installation

It is worth putting a separate section for this because of a few details. The primary complication is due to **CUDA**, a GPU computing API provided by NVIDIA. The GPU implementation of Pytorch uses CUDA, so **you can only use Pytorch with a CUDA- capable NVIDIA GPU** (or any CPU). You can check if your GPU is supported at `https://developer.nvidia.com/cuda-gpus`.
You can install Pytorch from `https://pytorch.org/get-started/locally/`. There are various package managers and builds which are available.

1. On Windows, I recommend that you use the Anaconda package manager. Use the latest stable version of Pytorch, and be sure to check the CUDA version (later the better).
2. On Ubuntu, you can install using Pip without any troubles. **You may have to install CUDA manually**.To do so, visit `https://developer.nvidia.com/cuda-downloads`.
3. In case your PC is underpowered, you can always run your models in Kaggle. In fact, that is what I do for Deep Learning models, even though I have a GTX 1050. Kaggle provides (as of writing) 30 hrs per week of an NVIDIA Tesla P100 GPU. You can even natively import Kaggle datasets without any trouble.

To check if the installation was successful, open a Jupyter notebook and run:

```
import torch
torch.cuda.is_available()
```

Expected output:

```
True
```

If the output doesn't match, you may have to manually install CUDA. Further, try:

```
torch.cuda.get_device_name()
```

Expected output should be something like:

```
'GeForce GTX 1050'
```

With this, we are ready to build a simple linear regression model using Pytorch.

- **Tip:** You can run Jupyter Notebooks in Visual Studio Code. This gives you all of VS Code's syntax highlighting and autocomplete features. Just be sure to install the Python plugin from the VS Code marketplace.

3 Building the model

1. You can find the code for this model here. It won't make much sense in the browser – download it and open in Jupyter.

2. First, let us import the relevant libraries.

```
import numpy as np
import torch
import matplotlib.pyplot as plt
```

3. Next, let's define a few helper functions.

```
def forward(X, W, b):
    return W*X + b
def mse(Yhat, Y, m):
    return (1/(2*m))*torch.sum((Yhat - Y)**2)
def update(W, b, W_grad, b_grad, alpha):
    W = W -alpha*W_grad
    b = b - alpha*b_grad
    return W, b
```

   We use `forward` to compute $\hat{Y}$. We use `mse` to compute J, the cost function. We use `update` to update the parameters W and b.

4. Next, let's define some hyperparameters.

   - *Hyperparameters* are those variables that you, the creator of the ML model specify in order to tune your model. This is in contrast to the model *parameters*, which are learned through training.
   - W and b are parameters, while α and num_iters are hyperparameters.

```
m = 100 # Number of data points.
noise_qty = 0.1 # How noisy is the data to be generated?
alpha = 0.0001 # The learning rate.
num_iters = 100000 # The number of iterations.
```

5. Next, let's generate the data.

```
X = torch.rand(m)*m
W_optim = torch.rand(1)
b_optim = torch.rand(1)
Y = forward(X, W_optim, b_optim) + torch.rand(m)*(m*noise_qty)
```

6. Our dataset is (X, Y). Let's plot it.

```
plt.scatter(X, Y)
plt.show()
```

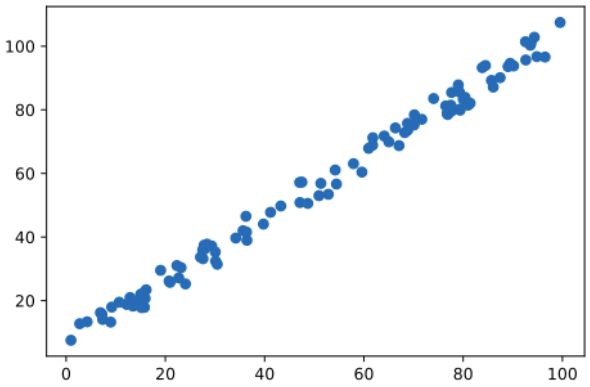


Figure 1: Y as a function of X.

7. Let's initialize W and b.

```
W = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
```

The argument *requires_grad = True* is needed for Pytorch's automatic differentiation package. This tells Pytorch to maintain a record of the computational graph starting from these nodes. If this doesn't make sense, hold on.

8. Let's visualize the current parameters.

```
Yhat = forward(X, W, b).detach()
plt.scatter(X, Y)
plt.plot(X, Yhat, color = "red")
plt.show()
```

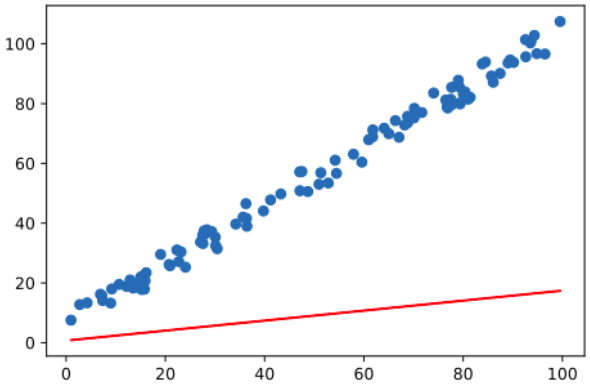


Figure 2: Our line, as expected.

9. Now, we come to the key step of training. There are four steps:
   (a) Compute $\hat{Y}$.
   (b) Compute $J(\hat{(Y)}, Y)$, the cost function.
   (c) Compute $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial b}$.
   (d) Update W and b.

   The code:

```
costs = []
for i in range(num_iters):
    if(i % (num_iters//100) == 0):
        print("\r",i/(num_iters//100), "%", end = "")
    W = W.clone().detach().requires_grad_(True)
    b = b.clone().detach().requires_grad_(True)
    Yhat = forward(X, W, b)
    cost = mse(Yhat, Y, m)
    cost.backward()
    costs.append(cost.item())
    W, b = update(W, b ,W.grad, b.grad, alpha)
print("")
```

Let's go line by line.

(a) The loop is the same as in Numpy.

(b) The `if` block is to make a progress indicator. By using the escape sequence `\r`, we overwrite the previously printed number.

(c) I will explain `W.clone` and `b.clone` momentarily.

(d) The next two lines are to compute $\hat{Y}$ and J.

(e) This line is the deal-maker when it comes to Pytorch.

- **Pytorch includes an automatic differentiation package called Autograd.** Using this, one can automatically compute the derivatives of a tensor with respect to other tensors.
- There are some preconditions as to which tensors can utilize Autograd.
- If you wish to compute the derivative of b with respect to a, then, in the computation graph,
  i. a must be a leaf node. What this means is that a must not itself depend on some other tensors.
  ii. a must be initialized with `requires_grad = True`.
- To compute $\frac{\partial b}{\partial a}$, first compute b as a function of a, then run `b.backward()`.

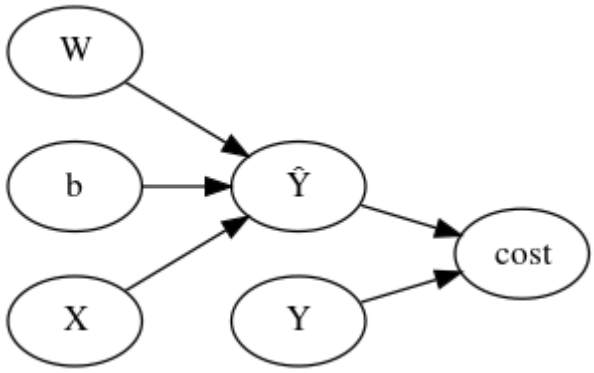
(f) The next line `update` is used to update W and b.

10. Now, we come to why we use *W.clone* and *b.clone*.

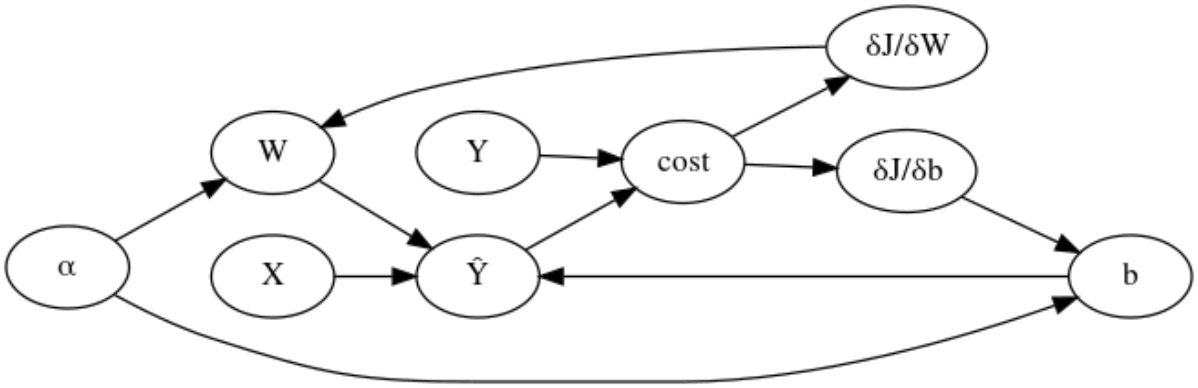
(a) When we use tensors with `requires_grad = True`, Pytorch makes a *computational graph* of the same and stores it.

(b) When we call `tensor.backward()`, this computation graph is used to compute the derivatives.

(c) Here's the computational graph when we run `cost.backward()` for the first time.



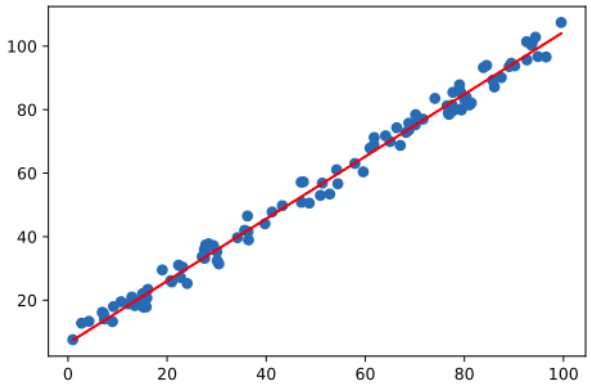
(d) Here's the computational graph when we run `cost.backward()` for the second time.



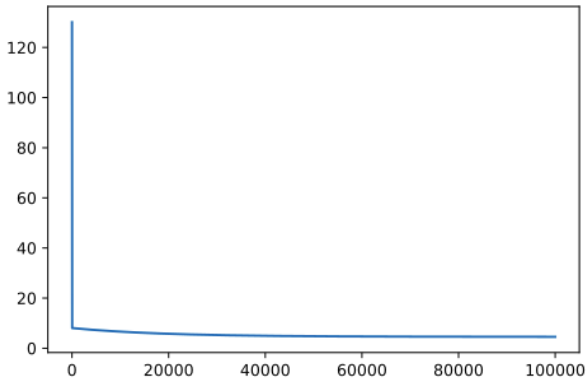
Notice how, in the second graph, W **and** b **are no longer leaf nodes**. What this means is that `W.grad` and `b.grad` will both be `None`. Further, we *definitely* don't want this kind of computational graph for `cost.backward()`.

(e) The solution is to "refresh" W and b at each iteration.

11. With that out of the way, let's see the plot for our new function.



12. And the cost:



4 Why was this the "dumb" way?

- There are too many things that can go wrong here.
- First, there's the business with refreshing W and b at every iteration.
- We could mess up the forward pass, the cost function or the update rule.
- Most importantly, here, we're reinventing the wheel. Pytorch has many utilities which make it a very easy task to define and train models.
- Imagine if we had to define the forward pass for a deep neural network or a convolutional neural network!

Next time, we'll see how we *really* build and train models in Pytorch.