CIS 522

Lecture 2: Introduction to PyTorch

1/21/2020

Lecturer: Konrad Kording *Scribe: Brandon, Vatsal*

1 Course Reminders

- Homeworks 0 and 1 have been released on the course website.
- Homework 1 is due on January 30th.
- **Start early!**

1.1 Lecture Overview

Having theoretical models of learning is great for developing intuition and rigorous results about them, but what is also extremely important are the implementation details, and no one wants to rewrite a neural network and backpropagation from scratch every time. In this lecture we will be going over **PyTorch**, a framework developed for making implementing deep learning models super easy and quick, the workflow of designing a deep learning system, and **automatic differentiation** - how PyTorch seamlessly implements backpropagation.

2 The workflow of a deep learning designer

The steps involved in the design of a deep learning system are as follows:

1. **Get a dataset** - Obtain a dataset through online sources/manual data collection/scraping the web etc. You could go for private datasets, Kaggle datasets, Scientific datasets (NSF, NIH etc) and many many more. You may further want to process and understand the data -
   (a) Exploratory data analysis - Explore available data points and features in the data, analyze and clean if necessary. This is very important as data can be inherently biased, incorrect or imbalanced.
   (b) Normalization strategies - feature normalization, feature scaling etc.
   (c) Split into train-validation-test sets
   (d) Class imbalance in the data - popular data classes more in number - for example - in cancer datasets, negative (absence of cancer) data points significantly outnumber the positive ones.

    (e) More data generally leads to better performance

2. **Data Augmentation** - make your model more robust to variations in the data by augmenting your data by data points that are rotated, cropped, skewed, padded , etc.

3. **Write a data loader** - which shuffles and batches the data for mini batch gradient descent (covered in detail later)

4. **Define a neural network** - This is where Pytorch forward/backwards is awesome, you define the network structure with the sequence as per the forward pass, no need to define the backward pass unless you use a function which is non differentiable and you need to write an approximation for the gradient.

    (a) Intuitively choose an architecture for the neural network

    (b) Define the components of the model - Fully Connected Layers, Non Linearities, Convolution layers etc.

    (c) Define the forward pass - sequential data flowing through the model components

    (d) Define the backward pass - oh wait, you don't need to! Pytorch will do that for you using the magical autograd :)

    How to define a network

    (a) Start with a network that is known to solve your problem

    (b) Alternatively with one that is good at solving a similar problem

    (c) Find ways of making it bigger

    (d) Try one of the ideas that others tried on similar problems

    (e) Come up with a new idea

    Is it science/ engineering? Is it evolution? Let's ask at the end of the course.

5. **Define a loss/ objective function** - Write a suitable loss/objective function that correctly represents the problem you are trying to solve. The choice of the loss function can greatly affect convergence. Optimize the function that matters i.e. choose a function that actually measures what matters. You might also want to choose a function whose success you can interpret somehow. Also, consider regularization in your loss to avoid overfitting - consider drop-out, weight decay, early stopping, getting more varied data.

6. **Optimize the neural network** - The optimizers find a good solution on the training data. But optimizers also have all kinds of inductive biases, their choice affects generalization performance Most of the times you would have to babysit through the optimization process, staring at the loss function to ensure your model is free of bugs, your learning rate is correct and everything is working as expected.

7. **Test performance** using a suitable metric and save the model if you are satisfied with the performance Testing the performance of the model is always important - validation set and training set features might be similar and performance on validation set might not represent performance on unseen data. It's important to test on unseen data with varied data features so as to estimate the performance of the model when deployed in the real world

8. **Deploy model** in the production setting. This is where your model will be affected by the scale - latency and compute might be big points of consideration in many critical applications like self driving.

3 Motivating computational graphs

In essence, **computational graphs** help break down large computations into smaller ones by visualizing them as a graph of smaller computations, and help in calculating derivatives of functions.

Before we jump into what these graphs are and how these graphs are useful, let's look at two examples of where this decomposition is useful.

3.1 Logarithm computation

Calculating the value of a logarithm is not a simple computation for a computer to do when it cannot be computed to an exact integer value. However, if we write the logarithm function as

$$\ln(z) = 2 \cdot \text{arctanh}\left(\frac{z-1}{z+1}\right) = 2\left[\frac{z-1}{z+1} + \frac{1}{3}\left(\frac{z-1}{z+1}\right)^3 + \frac{1}{5}\left(\frac{z-1}{z+1}\right)^5 + \cdots\right]$$

we can observe that we can achieve a good approximation for the logarithm using just the elementary operations (addition, subtraction, multiplication, division, exponentiation).

3.2 Inverse square root computation

Another example is the calculation of the inverse square root ($f(x) = \frac{1}{\sqrt{x}}$). This trick[1] is often used in computer graphics programs in order to calculate angles of incidence for projectiles. One can also break this computation down to the five aforementioned operations, allowing computers to easily calculate this function.

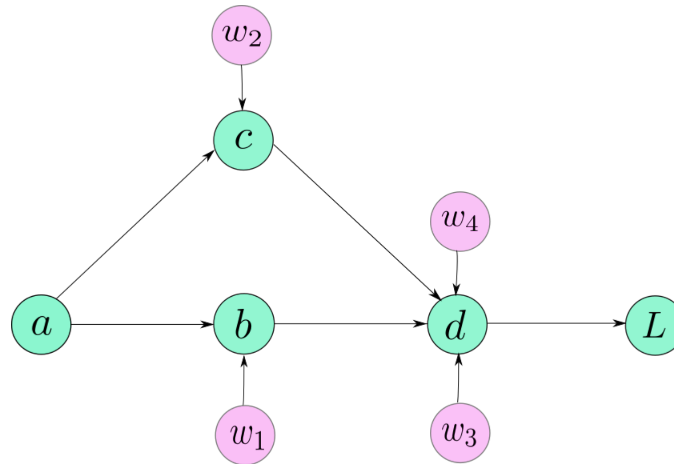
4 Computational graphs for derivatives

The core use of computational graphs is that is helps in the calculation of derivatives of such complex functions. We now fully state their rigorous definition: a computational graph stores gradients

[1] https://en.wikipedia.org/wiki/Fast_inverse_square_root

in intermediate variables used in a computation, and consists of nodes (which each represent the value and function used in the intermediate computation) and edges (which represent a direct computation between two nodes). Derivatives are stored on the **reverse** of each edge; each edge (u, v) stores the partial derivative $\frac{\partial u}{\partial v}$.

To make this more concrete, here is an example of a computational graph[2]:



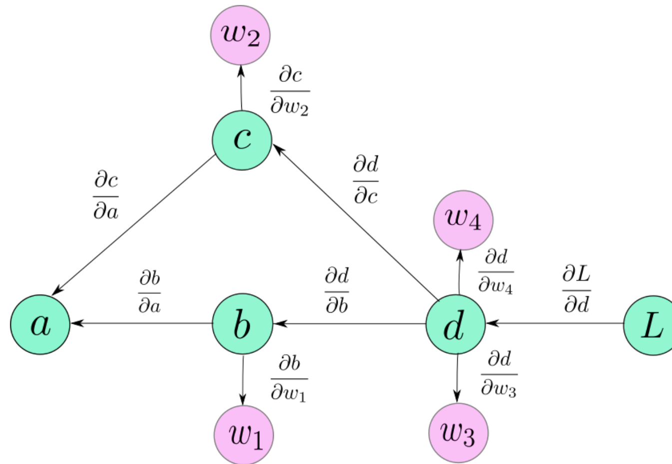
This graph represents the following set of computations:

$$
\begin{aligned}
b &= w_1 a \\
c &= w_2 a \\
d &= w_3 b + w_4 c \\
L &= f(d)
\end{aligned}
$$

Notice that an edge appears for each pair of values in which there is a direct computation between them (for example, (a, c) is an edge since c is computed via $c = w_2 a$).

The core derivative storage happens when we reverse each edge:

[2]https://towardsdatascience.com/getting-started-with-pytorch-part-1-understanding-how-automatic-differentiation-works-5008282073ec



Each edge now stores the partial derivative with its endpoints. For example, the new reversed edge $\frac{\partial d}{\partial c}$ stores the value

$$\frac{\partial d}{\partial c} = \frac{\partial}{\partial c}(w_3 b + w_4 c) = w_4$$

Now, why is this computational graph useful in computing derivatives? This is all due to the chain rule of derivatives: to compute $\frac{\partial L}{\partial w_1}$, we can trace along the path $L \rightsquigarrow w_1$ and compute

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial b} \cdot \frac{\partial b}{\partial w_1}$$

so computing derivatives of a function is reduced to a shortest-path problem!

5 PyTorch

PyTorch was developed to bootstrap all this computational graph formation and backpropagation into a central framework. The framework is based on Torch, a scientific computing Lua library at the time. It was developed by Facebook AI Research, and features this computational graph formation as well as built-in GPU acceleration, which significantly speeds up model training and testing.

5.1 Structure of the PyTorch library

The basic building blocks of PyTorch are **tensors** (defined in `torch.Tensor`), which essentially are multi-dimensional matrices storing numeric values. We can create tensors through a variety of function defined in the standard PyTorch library; one way is to create a random tensor like so:

```
a = torch.rand(10,10,5)
```

This creates a $10 \times 10 \times 5$ tensor. These tensors can be freely manipulated; their values can be changed to yield new tensors, they can be stacked to create new ones, and so on.

To track gradients through tensors, we define torch variables, which are nothing but gradient enabled tensors. Every variable object has several members some of which are:

1. Data: It's the data a variable is holding. x holds a 1x1 tensor with the value equal to 1.0 while y holds 2.0. z holds the product of two i.e. 2.0
2. requires_grad: This member, if true starts tracking all the operation history and forms a backward graph for gradient calculation. For an arbitrary tensor a It can be manipulated in-place as follows: a. requires_grad(True).
3. grad: grad holds the value of gradient. If requires_grad is False it will hold a None value. Even if requires_grad is True, it will hold a None value unless .backward() function is called from some other node. For example, if you call out.backward() for some variable out that involved x in its calculations then x.grad will hold $\frac{\partial \text{out}}{\partial x}$
4. grad_fn: This is the backward function used to calculate the gradient.

Finally, to actually perform backpropagation and update model weights by their gradients, we use **optimizers** to update parameter values appropriately. The module `torch.nn.optim` defines several kinds of optimizers that take the structure of a computational graph as well as the loss value on a forward pass, and updates all parameter values appropriately. We can use an optimizer as follows:

```
for input, target in dataset:
    optimizer.zero_grad()     # Zeroes out all gradients
    output = model(input)     # Forward pass
    loss = loss_fn(output, target) # Loss computation
    loss.backward()           # Computes gradients from loss
    optimizer.step()    # Updates model parameters from gradients
```

(You might notice that we zero out the gradients before each loop. There are scenarios in which we may want to accumulate gradients over different passes, but in most cases, every training loop should begin by zeroing out all gradients in the optimizer.)

5.2 Torchvision

Torchvision library, which is a part of Pytorch, contains all the important datasets as well as models and transformation operations generally used in the field of computer vision. It allows you to import datasets without any hassle.

5.3 Torchtext

Torchtext is a popular package that contains several popular data processing utilities like word embeddings and many popular datasets for natural language.

5.4 GPU acceleration

The way PyTorch makes all this computation fast is by porting computations to the GPU, which enables massive parallelization of computation (for example, matrix multiplication can be massively parallelized since many rows and columns are being multiplied simultaneously). PyTorch does this by employing **CUDA**, a parallel computing framework for interacting with the GPU.