UCLA Samueli School of Engineering

Introduction to PyTorch
2/26/2024

John Zhou

Department of Electrical and Computer Engineering
Neuroscience Interdepartmental Program

Overview

Objective: understand PyTorch under-the-hood and train a simple network

1. What is PyTorch?
2. PyTorch basics
    a. Tensors
    b. Autograd
3. A simple example network
    a. Layers and modules
    b. Losses
    c. Optimizers
    d. Data loading
4. Data handling
5. Implementation tips and common mistakes

What is PyTorch?

PyTorch is a machine learning (ML) framework built by FAIR

What makes frameworks different than regular libraries?

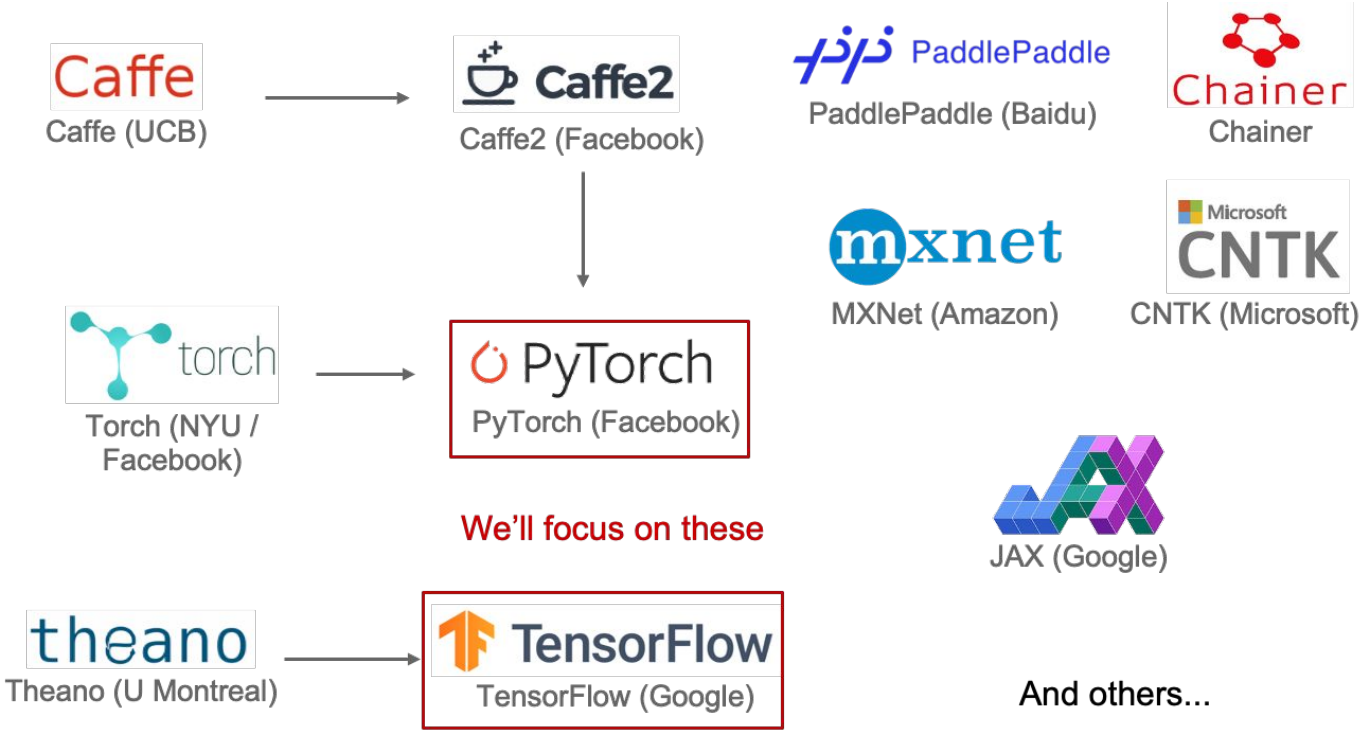
> "A framework embodies some **abstract design**, with more **behavior built in**. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by **plugging in** your own classes. **The framework's code then calls your code** at these points" - Martin Fowler

PyTorch is an object-oriented framework that automates key parts of ML, while handing the following under the hood:

- Automatic gradient computation via torch.autograd
- Allocating memory and performing computations on accelerator hardware

… along with implementations of common layers, losses, optimizers, and others!

A landscape of deep learning frameworks



Graphic from Pan Lu (ECE C147, 2022)

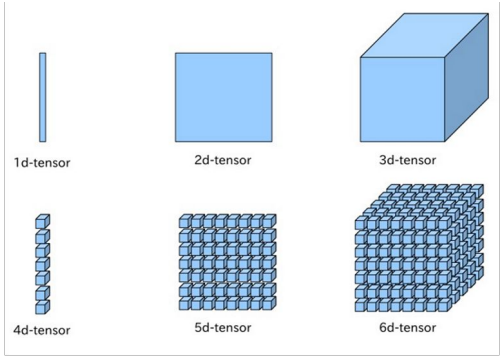
PyTorch basics: tensors

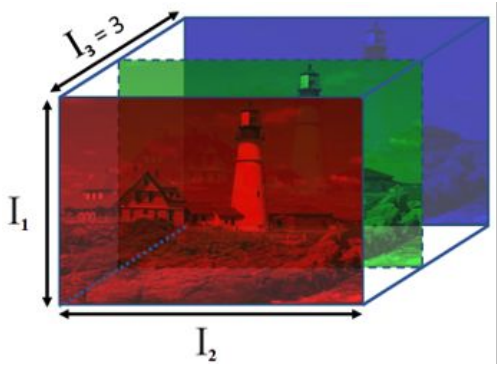
Central data structure in PyTorch

An N-dimensional array with a homogenous data type, basically a beefed-up Numpy array with the advantages:

- Can run on hardware accelerators, e.g., GPU, TPU, MPS
- Can track and store gradients in addition to values

Otherwise similar (but not identical!) syntax, usage, etc.



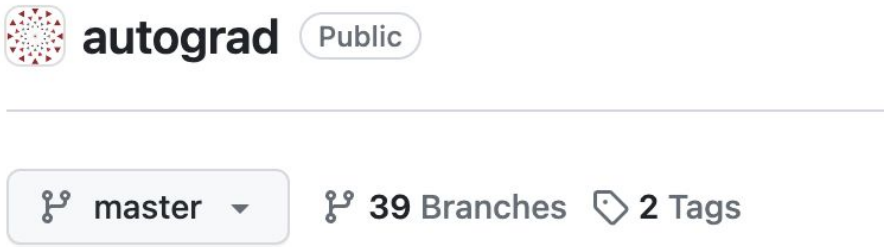


Graphic from Pan Lu (ECE C147, 2022)

PyTorch basics: torch.autograd

Some basic terminology

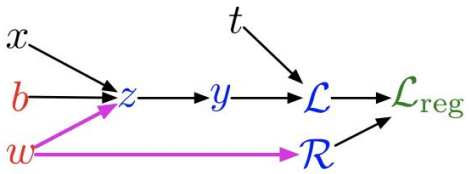
- Automatic differentiation (**autodiff**): a general way of constructing a procedure for computing derivatives of some value output by some function
- Backpropagation (**backprop**): reverse-mode autodiff applied to neural networks
- **Autograd**: one of the first libraries for autodiff, other versions (many with the same name) implemented in other frameworks



Naïve gradient computation

Use the multivariate chain rule to compute the partial derivatives

Example: univariate logistic least squares regression



Forward pass:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Backward pass:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$
$$\overline{\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\,\lambda$$
$$\overline{\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}$$
$$\overline{y} = \overline{\mathcal{L}}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = \overline{\mathcal{L}}\,(y - t)$$

$$\overline{z} = \overline{y}\frac{\mathrm{d}y}{\mathrm{d}z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} = \overline{z}\,x + \overline{\mathcal{R}}\,w$$
$$\overline{b} = \overline{z}\frac{\partial z}{\partial b} = \overline{z}$$

Notation: overbar is the partial derivative of L_reg w.r.t. the variable: in the red box, we see the partial w.r.t. y is the partial of L_reg w.r.t. L multiplied with the partial of L w.r.t. y, i.e. the chain rule!

Example from Richard Zemel (COMS W4995, Fall 2021)

Automating gradient computations

Store pre-specified routines for derivatives of primitive functions - derivatives of complex functions will just be sums and/or products of these!

Original program:

$$z = wx + b$$
$$y = \frac{1}{1 + \exp(-z)}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Sequence of primitive operations:

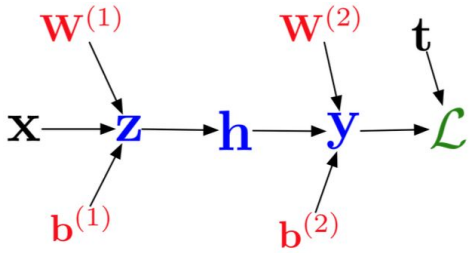
$$t_1 = wx$$
$$z = t_1 + b$$
$$t_3 = -z$$
$$t_4 = \exp(t_3)$$
$$t_5 = 1 + t_4$$
$$y = 1/t_5$$
$$t_6 = y - t$$
$$t_7 = t_6^2$$
$$\mathcal{L} = t_7/2$$

Example from Richard Zemel (COMS W4995, Fall 2021)

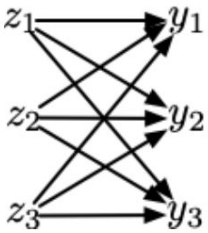
Moving from single variables to vectors

Vectorized variables allow us to use matrix calculus for fast parallel computing

Consider a simple multi-layer perceptron (MLP) with 3 hidden units



In single units, the hidden layer *z* looks like this →



Unit-wise backward pass would be: $\overline{z_j} = \sum_k \overline{y_k} \frac{\partial y_k}{\partial z_j}$

Vectorized as: $\overline{\mathbf{z}} = \overline{\mathbf{y}}^T \frac{\partial \mathbf{y}}{\partial \mathbf{z}}$ where the partial is the Jacobian $\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{pmatrix}$

Also known as the vector-Jacobian product (**VJP**)

Backprop as message passing

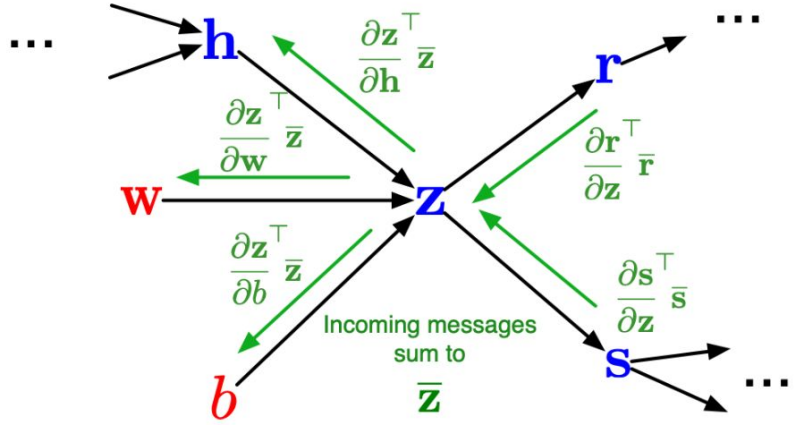
Keeping backpropagation modular

Naive implementation breaks modularity, since a function to compute the gradient of **z** needs to know who its children are and how it is used in the expressions for **r** and **s**

$$\bar{\mathbf{z}} = \frac{\partial \mathbf{r}}{\partial \mathbf{z}}\bar{\mathbf{r}} + \frac{\partial \mathbf{s}}{\partial \mathbf{z}}\bar{\mathbf{s}}$$

In autodiff, each node just sums incoming partials from its **children**, then passes its own messages to its **parents**, where each message contains the arguments needed to compute the VJP

Nodes don't need to know from where the incoming messages came



Reverse-mode auto-differentiation

Putting it all together

1. Create the evaluation graph from primitives of all the modules
2. Forward pass to evaluate the functions at input values
    a. During the forward pass, also save the input values and the functions to compute partial derivatives
3. Start the reverse pass with dL = 1
4. Pass dL backwards to all parent nodes
5. Parent nodes compute the partials using stored inputs and functions, then multiply them with the messages received from their children
    a. If there is a fan-out (a node has more than one child), sum those VJPs together according to the chain rule

Then, the optimizer (e.g., Adam, RMSProp, Adagrad, SGD, etc.) updates all learnable parameters with some function of their computed gradients w.r.t. the loss

Forward vs. reverse-mode autodiff

What's the difference?

Consider the simple 3-layer network below:

- **Forward** mode: we have the inputs and the pre-stored derivatives - can explicitly compute the Jacobians during the forward pass and multiply them as we go, starting with w_0= **x**

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}} \frac{\partial w_{i-1}}{\partial x}$$

- **Reverse** mode: same but do it in reverse, starting with w_3 = **y**:

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} \frac{\partial w_{i+1}}{\partial w_i}$$

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$
$$w_0 = x$$
$$w_1 = h(w_0)$$
$$w_2 = g(w_1)$$
$$w_3 = f(w_2) = y$$

Example from Wikipedia article on automatic differentiation

Why reverse-mode?

Why does PyTorch use reverse-mode autodiff?

According to the chain rule, we need to multiply the Jacobians of all the sub-operations anyway

$$\mathbf{y} = f(\mathbf{x}) = r(q(p(\mathbf{x})))$$

$$\mathbf{a} = p(\mathbf{x}), \quad \mathbf{b} = q(\mathbf{a}), \quad \mathbf{y} = r(\mathbf{b}).$$

- Matrix multiplication is not commutative, but it is **associative**

$$\underbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}_{|\mathbf{y}|\times|\mathbf{x}|} = \underbrace{\frac{\partial r(\mathbf{b})}{\partial \mathbf{b}}}_{|\mathbf{y}|\times|\mathbf{b}|} \underbrace{\frac{\partial q(\mathbf{a})}{\partial \mathbf{a}}}_{|\mathbf{b}|\times|\mathbf{a}|} \underbrace{\frac{\partial p(\mathbf{x})}{\partial \mathbf{x}}}_{|\mathbf{a}|\times|\mathbf{x}|},$$

The direction can save us some compute!

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial r(\mathbf{b})}{\partial \mathbf{b}} \left(\frac{\partial q(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial p(\mathbf{x})}{\partial \mathbf{x}} \right) \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \left(\frac{\partial r(\mathbf{b})}{\partial \mathbf{b}} \frac{\partial q(\mathbf{a})}{\partial \mathbf{a}} \right) \frac{\partial p(\mathbf{x})}{\partial \mathbf{x}}$$

# of operations: $|\mathbf{x}| \cdot |\mathbf{a}| \cdot |\mathbf{b}| + |\mathbf{x}| \cdot |\mathbf{b}| \cdot |\mathbf{y}|$ $\qquad$ $|\mathbf{y}| \cdot |\mathbf{a}| \cdot |\mathbf{b}| + |\mathbf{y}| \cdot |\mathbf{a}| \cdot |\mathbf{x}|$

Forward $\qquad$ Reverse

In NNDL, $\mathbf{y}$ is usually a scalar loss, i.e. $|\mathbf{y}| = 1$

Example from this excellent StackOverflow answer

Takeaways

Just need to understand the basic ideas

You (probably) won't need to worry about these details in day-to-day - key takeaways:

- PyTorch and other ML frameworks provide reverse-mode **autodiff** engines
- Autodiff works **iteratively**, different than naïve gradient computations by hand
- Uses **matrix/vector** calculus instead of single variables to allow massively **parallel** computation
- Incorporates several design choices to try to **minimize memory/computation complexity**
- Follows good OOP design to keep computation graphs **modular**

What does this look like in code?

Best practices for data

Data quality and quantity is arguably the main driver of DL progress

- Test data should **never** be encountered, used, or referenced in any shape or form during training - this includes checking accuracies for early stopping!
  - Instead, use a validation set to do this
- Standardizing across input variables can speed up training and stabilize gradient updates (even though this can be learned by weights and biases)
- Classes should be balanced, otherwise networks can collapse to a trivial solution of just predicting the dominant class in the training data
- Look at your data!
  - Do exploratory analysis, plot and visualize samples, distributions, etc.
  - Print intermediate results and analyze them
  - Even if your code runs smoothly, that doesn't mean that everything is working correctly
  - **Make sure things make sense!**

Tips, tricks, and common mistakes

- PyTorch accumulates gradients over multiple backwards calls, so it is important to call torch.zero_grad() to reset the gradients after each batch (if so desired)
- The computation graph will automatically add any new computations/data
    - If you don't want this (e.g., when computing validation scores), use the context manager with torch.no_grad()
- Try overfitting one batch of data to make sure that your network is learning
- Make sure to check documentation often - e.g., some losses may expect raw logits instead of softmaxed inputs
- Make sure to correctly set train() or eval() when switching between training and inference
- There are many ways to manipulate the shape of a Tensor - be clear about what they do - e.g., moving the underlying data vs. simply providing an alternate view of the same data

Q&A

Thanks for listening! Feel free to email me with any questions:
johnzhou (at) ucla.edu