PyTorch Fundamentals

Thursday, July 9, 2020 11:44 PM

*Review from: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

---

PyTorch Basics

What is PyTorch?

- It's a **Python-based** scientific **computing package** for audiences:
  - A *replacement for NumPy* to *use the power of GPUs*
  - a deep learning research platform that provides maximum flexibility and speed

- ***Tensors*** are similar to NumPy's ndarrays.
  - They can also be used on a GPU to accelerate computing.
  - Tensor operation=> https://pytorch.org/docs/torch
  - **Work like a matrix / multi-dimensional arrays.**

  - A tensor is **easily** converted to Numpy array
    - changing one will change the other because they sahre memory locations (if the Torch Tensor is on CPU)
    - Except a *CharTensor*

- Any operation that mutates a tensor **in-place** is post-fixed with an ``_``.
  - *Example*: ``x.copy_(y)``

Autograd: Automatic Differentiation

The *autograd* package *provides **automatic** differentiation for all* **operations on Tensors**.

- Generally, torch.autograd is an engine for **computing *vector-Jacobian product***.
- The **vector-Jacobian product** makes it very convenient *to feed external gradients into a model that has non-scalar output*.
- Document about autograd.Function is at https://pytorch.org/docs/stable/autograd.html#function

➢ *torch.Tensor* is the central class of the package.

- **Track all operations on it:**
  - Set its attribute .requires_grad as **True**
- **Have all the gradients computed automatically:**
  - When you finish your computation you can call .backward()
    - Called **backprop**
    - Creates the gradient for a tensor by:
      - Combining all equations that were computed to get to tensor and takes the derivative with respect to the initial value as the variable.
- **Gradient for this tensor will be accumulated into this attribute:**
  - .grad
    - *This is also the derivative equation created by **.backward()***
    - Can be used as a function on other tensors.
- **Stop a tensor from tracking history / future tracking:**
  - You can call .detach()
  - Or wrap the code block in with torch.no_grad():
    - *2nd option is helpful if model may have trainable parameters with no gradients.*

- ➢ *Function* another very important class **for autograd implementation**.
    - Tensor and Function are interconnected and *build up an acyclic graph*, that **encodes a complete history of computation**.
    - *Each tensor has a .grad_fn attribute that references a Function that has created the Tensor.*
        - (except for Tensors created by the user - their grad_fn is None).

- If you want **to compute the derivatives**, you can call .backward() on a Tensor.
    - If Tensor is a scalar (i.e. *it holds a one element data*),
        - You **don't need** to specify any arguments to .backward()
    - If Tensor is not a scalar (***has more** elements*):
        - You **need** to specify a gradient argument that is a tensor **of matching shape**.
        - Pass a **vector (tensor)** as argument to .backward to compute the vector-Jacobian product

---

Neural Networks

Neural networks **can be constructed** using the torch.nn package.

- nn depends on autograd to *define models and differentiate them*.
- An nn.Module contains:
    - **layers**,
    - and a **method** forward(input)\ that ***returns the output***.
- Full list of **modules and loss functions** that form the building blocks of deep neural networks:
    - https://pytorch.org/docs/stable/nn.html

➢ Typical **training procedure** for a neural network:

- **Define** the neural network that has some ***learnable parameters*** (or ***weights***).
- **Iterate** over a dataset of inputs.
- **Process** input through the network.
- **Compute** the **loss** (*how far is the output from being correct*).

- **Propagate gradients** back into the network's parameters.

- **Update** the **weights** of the network.
  - *Typically* using a simple **update rule**:

weight = weight - learning_rate * gradient

------ ------ ------ ------ ------ ------ ------ ------ ------ ------

Defining a neural network & Processing inputs and calling backward

- **You** just have to **define the** forward **function.**
  - The ***backward*** function (*where gradients are computed*) is *automatically defined* for you using autograd.

  - You **can use any of the Tensor operations** in the **forward** function.

- Get the ***learnable parameters*** (*weights*) of a model by using net.parameters()

- Place the **input** data to model.

- **Zero out** the **gradients** *before backpropagating* using .zero_grad()
- **Backpropagate** using the .backdraw function

- torch.nn only supports **mini-batches**.
  - The entire ``torch.nn`` package ***only supports inputs*** *that are a* ***mini-batch of samples***, and not a single sample.

  - ***If you have a single sample, just use ``input.unsqueeze(0)`` to add a fake batch dimension.*

- `torch.Tensor` - A *multi-dimensional array* with support for autograd operations like `backward()`. Also *holds the gradient* w.r.t. the tensor.
- `nn.Module` - Neural network module. *Convenient way of encapsulating parameters*, with helpers for moving them to GPU, exporting, loading, etc.
- `nn.Parameter` - A kind of Tensor, that is *automatically registered as a parameter when assigned as an attribute to a* `Module`.
- `autograd.Function` - Implements *forward and backward definitions of an autograd operation*. Every `Tensor` operation creates at least a single `Function` node that connects to functions that created a `Tensor` and *encodes its history*.

------ ------ ------ ------ ------ ------ ------ ------ ------ ------

Loss Function

A loss function takes the (**output**, **target**) pair as inputs, and computes a value that **estimates how far away the output is from the target**.

- There are several different loss functions:
  - <https://pytorch.org/docs/stable/nn.html#loss-functions>

  - *Example:*
    - nn.MSELoss which computes the *mean-squared **error*** **between** the **input** and the **target**.

```
output = net(input)
criterion = nn.MSELoss()
#make target same shape as output
loss = criterion(output, target)
```

- You can follow loss in the backward direction, using its .grad_fn attribute.

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

------ ------ ------ ------ ------ ------ ------ ------ ------ ------

Backprop

To **backpropagate** the error all we have to do is to loss.backward().

- You *need to clear the existing gradients* though, else gradients will be accumulated to existing gradients.

  - *`model.zero_grad()`*

- Then call **function to backpropate** on the ***loss function***:

  - *`loss.backward()`*

------ ------ ------ ------ ------ ------ ------ ------ ------ ------

Update the weights

- The ***weigths* are updated** using an optimizer (*update rule*).

➢ The simplest **update rule** used in practice is the **Stochastic Gradient Descent** (SGD):

weight = weight - learning_rate * gradient

- *Example*: (**Implementation** in Python.)

```
learning_rate = 0.01
for f in net.parameters():
   f.data.sub_(f.grad.data * learning_rate)
```

- There are **various different update rules** such as **SGD**, **Nesterov-SGD**, **Adam**, **RMSProp**, etc.
  - **Enable** these methods with this package: torch.optim

- Incorporate into the training loop.
  - *Example*:

```
import torch.optim as optim
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)
# in your training loop:
optimizer.zero_grad() # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() # Does the update
```