CS2109S PyTorch Command Glossary

Preface

This document serves as a reference point to all things PyTorch. It is designed specifically to avoid unncessary searching online, providing you with the key elements of the `torch` library for Problem Set 5 (and 6, soon!). In fact, you can use it for your projects, other modules, and any where else you take PyTorch with you!

> For more information, visit the official PyTorch documentation: https://pytorch.org/docs/stable/index.html

Table of Contents

Installation

To install PyTorch, use `pip` in your terminal. You can either download it globally across your system or inside a virtual environment (recommended but not necessary).

```
$ pip install torch torchvision
```

> The additional `torchvision` library allows us to make use of popular datasets, hence the additional package installation.

Usage

To use PyTorch, import the library and its sub-modules:

```
import torch
import torch.nn as nn
```

- `torch` is the base library
- `torch.nn` allows you to build neural network layers, create loss functions and optimisers, and more

Tensors

The PyTorch `Tensor` is akin to `NumPy`'s `numpy.ndarray` object – essentially, a n-dimensional matrix. There are a few different ways to create tensors:

Tensor Basics

```
a = torch.tensor(...) # creating a tensor

# data types:
torch.Tensor(...) # any kind of value
torch.FloatTensor(...) # float values only
torch.LongTensor(...) # integer values only
```

You can replace `...` with any value of any numerical data type:

- integer
- float
- n-dim (nested) array of integers/floats

Let's stick to using `torch.tensor(...)` to create tensors in this module. Let's avoid using `Tensor`, `FloatTensor`, and `LongTensor` as they impose restrictions on what values they can hold.

> **Bonus:** if your tensor has a single element (i.e., a `1x1` tensor), you can extract its value using the `.item()` method.
>
> For instance, if `a = torch.tensor(123)`, `a.item()` will return `123`.

Randomness

Most often, you are required to inject randomness to your experiments. Similar to `numpy`, you can generate Tensors of any arbitrary size/dimensionality with random values. Here are some ways to generate random tensors:

- `torch.rand(size)`: draws digits from Uniform distribution `x ~ U(0, 1)`
- `torch.randn(size)`: draws digits from Normal distribution `x ~ N(0, 1)`
- `torch.randint(low, high, size)`: generates tensors with random integers

```
a = torch.rand(10, 10) # a 10x10 matrix
b = torch.rand(10) # vector with 10 elements
c = torch.rand(10, 1) # vector with 10 elements with an extra (insignificant)
dimension
d = torch.rand(28, 28, 28) # a "cube" tensor with 28 elements

e = torch.randn(10, 5) # a 10x5 matrix
```

```
f = torch.randint(0, 100, (5, 5)) # a 5x5 matrix of integers in [0, 100)
```

> All of these random tensors are, by default, `torch.Tensor` object data type. Each element of these tensors are also of the same `torch.Tensor` type.
>
> So, large tensors are made of smaller tensor units. It's the fundamental "building block" of PyTorch (like the "cell" in animal!).

Operations

As with `np.array`, you can perform familiar tensor operations such as addition, subtraction, multiplication, division, and exponentiation.

```
a = torch.tensor(50)
b = torch.tensor(75)
p = torch.tensor(2)

c = a + b
print(c) # torch.Tensor(75)

d = b - a
print(d) # torch.Tensor(25)

e = b * a
print(e) # torch.Tensor(3750)

f = b / a
print(f) # torch.Tensor(1.5000)

g = a ** p
print(g) # torch.Tensor(2500)
```

In fact, when working with PyTorch tensors, you can perform operations with non-tensors as well:

```
a = torch.tensor(50)

b = a + 4 # torch.Tensor(54)
c = a - 4 # torch.Tensor(46)
d = a * 2 # torch.Tensor(100)
d = a / 2 # torch.Tensor(25.0)
e = a ** 2 # torch.Tensor(2500)
```

Working with Gradients

Efficiently computing gradients is what PyTorch is known for. When creating tensors, we use the `requires_grad` parameter to tell PyTorch we hope to perform gradient computation with this variable in the future. This allows PyTorch to store gradient information inside the tensor for later access. By default, this parameter is `False` because it's relatively more space-heavy to store gradients inside the tensor object.

```
a = torch.tensor(10.0, requires_grad=True) # set the param to True, default is
False
```

Partial Differentiation

In Machine Learning, gradient computation involves taking partial derivatives of one variable with respect to another. To achieve this, we use the `backward()` method of tensors (provided that they have `requires_grad=True`).

```
a = torch.tensor(5.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)

c = (2 * a) + b**2 # torch.Tensor(14.0)
c.backward()
```

The variable on which `backward()` is called is the target variable (`c` in this case). All other tensors involved in the computation have their gradient values automatically computed. So, in this case, partial derivatives `dc/da` and `dc/db` are computed automatically and stored within `a` and `b` respectively.

> Fun fact: this is why we call the package `autograd`, which alludes to automatic computation of gradients!!!

Once we call `backward()`, all that's left to do is access the gradient values for each variable of interest. This is done via the `grad` attribute of a tensor:

```
"""
Partial derivatives:

c = 2a + b^2
dc/da = 2
dc/db = 2b
"""

dc_da = a.grad # 2.0
dc_db = b.grad # 4.0
```

Common `torch` Operations

Most, if not all, of these operations are differentiable by nature. This means you can use them within your *computation graph* and compute gradients.

Operation	Remarks
`torch.sum(input)`	Returns the sum of all elements in the input tensor.
`torch.pow(base, exp)`	Returns the exponentiation of the base tensor to the exponent.
`torch.mean(input)`	Returns the mean of all elements in the input tensor.
`torch.square(input)`	Returns the square of elements in the input tensor.
`torch.no_grad()`	Pauses all gradient computation and tracking inside the `with torch.no_grad()` block.
`torch.matmul(input, other)`	Returns the matrix product of tensors `input` and `other`. Same effect as `A @ B`.
`torch.reshape(input, shape)`	Returns the reshaped input matrix if dimensions commute.
`torch.softmax(input, dim)`	Computes the Softmax of an input along a specified dimension/axis.
`torch.max(input, dim)`	Returns the maximum element in the input tensor along a specific dimension/axis.
`torch.manual_seed(seed)`	Sets the random number generator seed to the one specified. Good for reproducibility of runs.
`torch.transpose(input)`	Returns the transposed matrix. Alternatively, can do `input.T` for the same effect.

> For `torch.matmul(...)`, you can also use `@` between the matrices of interest as long as they *commute*. Suppose `A` is a 3x4 tensor and `B` is a 4x5 tensor. `C = A @ B` will be a 3x5 tensor.
>
> PyTorch can also matrix multiply tensors of higher dimensions (3D, 4D, ...) but we will not be getting into that topic just yet.

`torch.nn` Layers API

The speciality of PyTorch lies in its pythonic way of building neural networks. It provides a nice interface to quickly prototype models and train/test them using a compact, low-overhead, neatly-written train-test loop.

`nn.Module`

The `nn.Module` interface provides the necessary methods to facilitate the construction of neural networks, both simple and complex. The `__init__` and `forward` method are the most important: they house the individual layers of the network, and compute the forward pass for a given input tensor respectively.

> **IMPORTANT:** By convention, ALL layers are initialised in the `__init__` method. These same layers are then referenced and used via `self` in the `forward` method.

Here's a snippet of a neural network using PyTorch:

```
class Model(nn.Module):
    def __init__(self):
        super().__init__() # don't forget to inherit from the parent class
        self.layer1 = ...
        self.layer2 = ...
        self.layer3 = ...
        self.layer4 = ...

    def forward(self, x):
        """
        By default, the only input this function can take in is `x`, the input
tensor.
        Don't add any other parameters into this function to keep things simple.
        """
        x = self.l1(x)
        x = self.l2(x)
        x = self.l3(x)
        out = self.l4(x)

        return out
```

Here are two layers you'll use most during your time in CS2109S. It's best to familiarise yourself with them!

Layer	Usage	Remarks
Fully-connected / Dense	`nn.Linear(in_features, out_features, bias=True)`	Inputs are vectors of size `in_features`. Performs `Y=Wx+b` and outputs a vector of size `out_features`.
Convolution	`nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)`	Inputs are images/tensors with `in_channels` number of channels of arbitrary height and width.
ReLU	`nn.ReLU()`	Performs the Rectified Linear Units (ReLU) activation on the input tensor.
Leaky ReLU	`nn.LeakyReLU(negative_slope=0.01)`	Performs the Leaky ReLU activation on the input tensor with the specified negative slope.
Sigmoid	`nn.Sigmoid()`	Performs the Sigmoid activation on the input tensor with the specified negative slope.

> For the sake of keeping PS5 simple, ignore the Convolution layer. Once you cover it in future lectures, give it a spin!

Optimisers and Losses

Most important for any gradient-based computation program is the optimiser and objective (i.e., loss) function. Writing your own optimiser or loss is a tedious process and is error-prone if you are not sure how to write efficient PyTorch code. To alleviate this, PyTorch allows you invoke popular optimisers and losses with a single line of code.

Additionally, here's a list of popular loss functions:

Loss	Usage
Cross Entropy	`nn.CrossEntropyLoss()`
Binary Cross Entropy	`nn.BCELoss()`
Mean Squared Error	`nn.MSELoss()`
Mean Absolute Error	`nn.L1Loss()`
Negative Log Liklihood	`nn.NLLLoss()`

After computing the output of the forward pass using your model, you can

```
loss_fn = nn.XYZLoss() # some arbitary loss from the above table

output = ... # some tensor
target = ... # some tensor
loss = loss_fn(output, target)

"""
As mentioned above, to backpropagate the loss wrt the parameters, you can simply
call
`loss.backward()` and it will compute the partial derivates (i.e., the gradients)
and
store them inside the `.grad` attribute of each and every parameter tensor!

Pretty cool, huh? ;)
"""
```

Here's a list of popular optimisers:

Optimiser	Usage
Stochastic Gradient Descent (SGD)	`torch.optim.SGD(parameters, lr)`
Adaptive Momentum (Adam)	`torch.optim.Adam(parameters, lr=0.001)`
Adaptive Gradient (Adagrad)	`torch.optim.Adagrad(parameters, lr=0.01)`

Here, `parameters` refers to the network parameters and `lr` is the learning rate. Different optimisers have different default learning rates while some require the user to input that in (for example, SGD needs you to specify the `lr` while Adam has a learning rate of `0.001`). In your Problem Sets, if the learning rate is **NOT** specified, it means we expect you to use the default; you don't have to tune these numbers yourself.

Suppose we have a network `net = Net(...)` that's build using the `nn.Module` interface. To access the parameters of the model, we simply call the `net.parameters()` method; it will return a list of all the weights and biases (i.e., parameters) of the model. We pass these parameters into the optimiser, along with any other arguments (like learning rate, for instance).

```
net = Net(...)
optimiser = torch.optim.SGD(net.parameters(), lr=0.001)
```

IMPORTANT NOTES:

- Before you perform a forward pass, we must ensure that the optimiser doesn't have the previous iteration's gradients stored inside it. To flush them, we must reset them to zero. To do so, add the line `optimiser.zero_grad()` before your forward pass through the network using input `x`.
- Additionally, after a backward pass via Backpropagation, we must perform an update step to the parameters within the network. In Gradient Descent, for example, this update step is `w = w - lr * dLdw`. To do so, simply call `optimiser.step()` after the `loss.backward()` line.

Simply put, your forward and backard pass should look like this:

```
for x, y in dataset:
    optimizer.zero_grad() # flush the prev gradients
    output = model(x)
    loss = loss_fn(output, y)
    loss.backward() # perform backpropagation
    optimizer.step() # update parameters
```

There's definitely more than meets the eye when it comes to PyTorch. This library is the primary workhorse around the world, so it's beneficial to learn it from the ground up. Problem Set 5 (and 6!) simply offers a taster on how life with PyTorch is like – it's way better than initialising individual biases and manually writing out the equations for a lot of Machine Learning applications.

Of course, as with any library, if you're interested in diving under its hood, feel free to look at its documentation where you can find fun tutorials and exercises to jog your mind.

> For more information, visit the official PyTorch documentation: https://pytorch.org/docs/stable/index.html

"Happy (Machine) Learning!!!" ~ CS2109S Teaching Team