1 Learning about PyTorch

PyTorch is a Python library that is widely used for deep learning. PyTorch greatly simplifies the process of building deep learning models, training them via backpropagation and stochastic gradients, loading and processing data, and more. You will need to gain a working understanding of PyTorch in order to successfully complete the homework, and this understanding will quite possibly benefit you later on in your future machine learning endeavors.

For this discussion, you will use https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html as a reference to answer the questions below, rather than lecture materials. If you are not yet familiar with PyTorch, it is strongly encouraged that you review this excellent tutorial in detail before diving into the homework assignment. For the purposes of today’s discussion, do your best to skim through the tutorial and search for the answers to the questions. Some of these answers will not be directly provided in the text and will instead require you to read the code and infer the behavior of certain functions.

Solution: Check out https://tinyurl.com/PyTorch189 for TA Sean Lin’s interactive Colab tutorial.

(a) Compare and contrast NumPy’s ndarray with PyTorch’s tensor. In what ways are these two types similar? In what ways are they different?

Solution: Both ndarray and tensor represent multidimensional arrays. The difference between a NumPy ndarray and a PyTorch tensor is that tensors can be backed by the accelerator memory, such as a GPU, and they store numerical gradient information in a grad attribute when instantiated with requires_grad=True.

On setting requires_grad=True, tensors start forming a backward graph that tracks every operation applied on them to calculate the gradients using something called a dynamic computation graph (DCG). By PyTorch’s design, gradients can only be calculated for floating point tensors.

(b) What is the purpose of the torch.nn.Module class? What functionality does it provide?

Solution: torch.nn.Module is the base class for all neural network modules. Your custom models and layers should also subclass this class.

Modules can contain other Modules, allowing to nest them in a tree structure. Inheriting from torch.nn.Module provides useful functionality to your class. For example, it makes your class keep track of its trainable parameters via the parameters bound method. You can also
easily swap a Module between CPU and GPU with the `to(device)` bound method, where `device` can be a CPU (`torch.device("cpu")`) or GPU (`torch.device("cuda:0")`).

(c) What function is required to make a Module callable? What does calling the Module do for neural networks?

**Solution:** The `forward` function is required. We just have to define the `forward` function, and the backward function (where gradients are computed) is automatically defined for you by PyTorch. We can use any of the Tensor operations in the `forward` function.

Note that, just like in Homework 2, certain intermediate values are stored during the forward pass of a neural network which will be used in backpropagation, and this is automatically handled by the built-in PyTorch layers. Using the `torch.no_grad` context manager disables this functionality and can make evaluation more efficient.

When calling the Module, the `__call__` function is already defined in `torch.nn.Module`. You should call the Module directly (`output = model(data)`) instead of something like `output = model.forward(data)`, because calling the Module performs additional functionality such as registering hooks.

(d) What PyTorch function is used when moving between convolution layers and fully connected layers in a neural network forward pass?

**Solution:** The `flatten` function is used to transform the activations of the convolution layers, which have channel, height, and width dimensions, into flat vectors that are input into the fully connected layers.

(e) How is backpropagation performed in PyTorch?

**Solution:** At an interface level, it is relatively simple. A loss, which is a tensor is computed by comparing the outputs of the model with the desired outputs (labels), using a criterion such as `torch.nn.CrossEntropyLoss`. This loss has a backward bound method which can be invoked that automatically populates the grad variables of all tensors with `requires_grad=True` that were involved in the computation of `loss`.

More details:

PyTorch’s automatic differentiation engine (`torch.autograd`) calculates derivatives. It records a graph of all the operations performed on a gradient enabled tensor and creates an acyclic graph called the dynamic computational graph. The leaves of this graph are input tensors and the roots are output tensors. Gradients are calculated by tracing the graph from the root to the leaf and multiplying every gradient in the way using the chain rule. This is effectively a automatic and efficient implementation of the forward and backward pass you implemented in Homework 2.

PyTorch builds a Dynamic Computational Graph (DCG) from scratch in every iteration providing maximum flexibility to gradient calculation. For example, for a forward operation (function) `Mul` a backward operation (function) called `MulBackward` is dynamically integrated in the backward graph for computing the gradient. Gradient enabled tensors (variables) along with functions (operations) combine to create the dynamic computational graph. The flow of
data and the operations applied to the data are defined at runtime hence constructing the com-
putational graph dynamically. This graph is made dynamically by the autograd class under
the hood. You don’t have to encode all possible paths before you launch the training — what
you run is what you differentiate. This property allows you to add or remove layers, or shuffle
parameters between iterations.

backward is the method which actually calculates the gradient by passing its argument through
the backward graph all the way up to every leaf node traceable from the calling root tensor. The
calculated gradients are then stored in the grad attribute of every leaf node. Remember, the
backward graph is already made dynamically during the forward pass. The backward method
only calculates the gradient using the already made graph and stores them in leaf nodes.

(f) What is the purpose of the zero_grad method? What happens if this method is not called?

Solution: In PyTorch, for every mini-batch during the training phase, we typically want to
explicitly set the gradients to zero before backpropagation, because PyTorch accumulates the
gradients on subsequent backward passes. The default action in PyTorch is to accumulate (i.e.
sum) the gradients on every loss.backward() call.

Because of this, when we start a training loop, we should zero out the gradients so that we
do the parameter update correctly. Otherwise, the gradient would be a combination of the old
gradient, which we have already used to update the model parameters, and the newly-computed
gradient.

(g) How are model updates (e.g., via stochastic gradients) performed in PyTorch?

Solution: Models are updated by passing the model.parameters() into an optimizer, which
typically comes from the torch.optim library (e.g., torch.optim.SGD or torch.optim.Adam).
First, calling loss.backward() calculates and populates gradients with respect to the model
parameters. Then, calling the optimizer’s step bound method updates the parameters. Note
that the optimizer does not update all tensors with a grad attribute, only the ones passed
to it during initialization. An examples is shown in: https://pytorch.org/tutorials/
beginner/blitz/autograd_tutorial.html#usage-in-pytorch

For more information see: https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

(h) What PyTorch class is responsible for processing and providing batches of data for training
and testing? Briefly describe how it is constructed and used.

Solution: The torch.utils.data.DataLoader class.

Code for processing data samples can get messy and hard to maintain; we ideally want our
dataset code to be decoupled from our model training code for better readability and modularity.
PyTorch provides two data primitives: torch.utils.data.DataLoader and
torch.utils.data.Dataset. These classes allow you to use pre-loaded datasets as well as
your own data. Dataset stores the samples and their corresponding labels, and DataLoader
wraps an iterable around the Dataset to enable easy access to the samples.
(i) How does one take advantage of GPUs when training and testing PyTorch models?

**Solution:** To make use of a GPU, we need to transfer the relevant tensors and Modules to the GPU first by calling `.cuda()` or `.to(device)`, where `device` corresponds to a GPU. Once all of the relevant tensors and Modules are on the GPU, the operations will automatically be computed on the GPU. To see an example see: [https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#tensor-operations](https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#tensor-operations)

To additionally make sure of more than one GPU at once, we can make use of data parallelism or model parallelism on GPUs. Data parallelism is when you use the same model for every thread, but feed it with different parts of the data; model parallelism is when you use the same data for every thread, but split the model among threads.