A Simple functorch Example

11/5/22

In recent years, there has been a small movement of people trying to go from state*ful* (Python OOP, class-based modules) to state*less* (pure functions) neural network code. The standard PyTorch `nn.module` is indeed a OOP-based class. But more recent libraries such as JAX, introduce the ability to feasiby create stateless (just functions!) machine learning models. Now, in PyTorch version 1.13, we have `functorch` in-tree (in the main package). Why stateless? Read this blog post for the differences, but some reasons for why I like stateless code is because:

- Less leaky abstractions (and less unknown abstractions in general!)
- Closer to the mathematical form (after all, a neural network is just a series of functions chained together!)
  - When you learn SGD in class in the mathematical form and then use PyTorch, the disconnect is fairly evident.
- Less compute overhead (less things to keep track of internally –> less memory needed)
- Ability to work a lower level (which, in my opinion, can help facilitate new ideas)
- Ability to work with function transformations such as `vmap`, `pmap`, `jit`, and `grad` (PyTorch has `grad`... yes I know... but applying grad to a stateless function makes much more intuitive sense than applying it to some stateful module!).

This might sound like an advertisement for JAX (which might be coming up in a future blog post!), but it is really to set the stage for functorch. functorch is a library that allows you to accomplish nearly all of the above, but in PyTorch! The basic idea is to purify stateful PyTorch modules into stateless functions like this (source):

```
import torch
import functorch
from functorch import make_functional

model = torch.nn.Linear(3, 3)
func_model, params = make_functional(model)
```

As functorch is relatively new, there aren't many examples out there showing how to use the library. So the goal of the rest of this post is to provide a simple example for creating an image classifier using functorch and PyTorch and updating the weights using SGD (no `torch.optim`!).

Here is the code:

```
import torch
import torchplate
from torchplate import experiment
from torchplate import utils
import functorch
from functorch import grad, grad_and_value
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from rsbox import ml
# import torchopt
import requests
from tqdm.auto import tqdm
import cloudpickle as cp
from urllib.request import urlopen



class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3*32*32, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 16)
        self.fc4 = nn.Linear(16, 3)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

```
class OptExp:
    def __init__(self):
        self.model_module = Net()
        self.criterion = nn.CrossEntropyLoss()
        dataset = cp.load(urlopen("https://stanford.edu/~rsikand/assets/datasets/mini_cifa
        self.trainloader, self.testloader = torchplate.utils.get_xy_loaders(dataset)
        self.model, self.params = functorch.make_functional(self.model_module)  # init net


    def predict(self, x):
        """returns logits"""
        assert self.model is not None
        assert self.params is not None
        logits = self.model(self.params, x)
        return logits


    @staticmethod
    def sgd_step(params, gradients, lr):
        """one gradient step for updating the weights"""
        updated_params = []
        for param, gradient in zip(params, gradients):
            update = param - (lr * gradient)
            updated_params.append(update)

        return tuple(updated_params)


    @staticmethod
    def stateless_loss(params, model, criterion, batch):
        """
        Need to perform forward pass and loss calculation in one function
        since we need gradients w.r.t params (must be args[0]). The first
        value we return also needs to be the scalar loss value.
        """
        x, y = batch
        logits = model(params, x)
        loss_val = criterion(logits, y)
        return loss_val, logits
```

```
    @staticmethod
    def train_step(params, model, criterion, batch, lr):
        """Combine this all into one function for modularity"""
        # has_aux means we can return more than just the scalar loss
        grad_and_loss_fn = grad_and_value(OptExp.stateless_loss, has_aux=True)
        grads, aux_outputs = grad_and_loss_fn(params, model, criterion, batch)  # get the
        loss_val, logits = aux_outputs
        params = OptExp.sgd_step(params, grads, lr)
        return params, loss_val, logits


    def train(self, num_epochs=10, lr=0.01):
        print('Beginning training!')
        epoch_num = 0
        for epoch in range(num_epochs):
            running_loss = 0.0
            epoch_num += 1
            tqdm_loader = tqdm(self.trainloader)
            for batch in tqdm_loader:
                tqdm_loader.set_description(f"Epoch {epoch_num}")

                # update params with one step
                self.params, loss_val, logits = OptExp.train_step(self.params, self.model,

                running_loss += loss_val

            # print loss
            epoch_avg_loss = running_loss/len(self.trainloader)
            print("Training Loss (epoch " + str(epoch_num) + "):", epoch_avg_loss)


        print('Finished training!')


exp = OptExp()
exp.train(num_epochs=50, lr=0.01)
```