Deep Learning with Pytorch, core materials

written by User 1006 on Functor Network
original link: https://functor.network/user/1006/entry/559

My notes for a few chapters (Ch5-Ch7) of the book "Deep Learning with PyTorch". Mostly to familiarize myself with the fundamentals.

Ch5 The Mechanics of Learning

The temperature unit conversion example

The data tensors are:

```
t_c = torch.tensor([0.5, 14.0, 15.0, ..., 21.0])
t_u = torch.tensor([36.7, 55.9, 58.2, ..., 68.4])
```

we conjecture a linear relationship: t_c = w * t_u + b, and proceed to estimate w and b.

To do so, first define our loss:

```
def loss_fn(t_p, t_c):
    squred_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Note that this loss function returns a scalar via averaging. This is MS loss.

Now initialize parameters:

```
w = torch.ones(())
b = torch.zeros(())
```

note that w is right now written as "tensor(1.)", and it can take any dimension that is fit. For example, if z is torch.zeros(5), then w + z gives tensor([1., 1., 1., 1., 1.]).

Now we **invoke the model**:

```
t_p = model(t_u, w, b)
```

Since t_u.shape is torch.Size([11]), here the output t_p is also of this shape.

Finally, we evaluate the loss given this initial w & b:

```
loss = loss_fn(t_p, t_c)
```

Now we explain the basic idea of gradient descent. We aim to do the following:

```
w = w - lr * loss_rate_of_change_wrt_w
b = b - lr * loss_rate_of_change_wrt_b
```

This "loss_rate_of_change_wrt_w" can be written via Chain rule:

```
d loss_fn / dw = (d loss_fn / d t_p) * (d t_p / d w)
```

After some steps, we define our gradient function:

```
def grad_fn(t_u, t_c, t_p, w, b):
    ....

    return torck.stack([dloss_dw.sum(), dloss_db.sum()])
```

Mathematically, we are doing the following calculation:

$\nabla_{w,b} L \equiv (\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}) = (\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b})$

where m stands for model/prediction. **note that m is a vector output**.

Then, we iterate to fit the model:

```
def training_loop(params, t_u, t_c):
    w, b = params
    t_p = model(t_u, w, b) #forward pass
    loss = loss_fn(t_p, t_c)
    grad = grad_fn(t_u, t_c, t_p, w, b) #backward pass

    params = params - lr * grad

    return params
```

Often needs to normalize inputs. First we do it in a ad hoc way:

```
t_un = 0.1 * t_u
```

and then run training loop on normalized input. In the end, given data "t_un", we can call the trained model to give us predictions:

```
t_p = model(t_un, *params) #note we use argument unpacking here
```

recall *params means passing elements of params as individual arguments.

Now we use "autograd" to compute gradients automatically.

```
def model(t_u, w, b):
    return w * t_u + b

def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()

params = torch.tensor([1.0, 1.0], requires_grad = True)

loss = loss_fn(model(t_u, *params), t_c)

loss.backward()
```

at this point, if you run “params.grad”, the output is “tensor([xxx, xxx]). This gives $(\frac{\partial \text{loss}}{\partial w}, \frac{\partial \text{loss}}{\partial b})$

note that calling backward lead derivatives to accumulate at leaf nodes, not store. Thus, we need to “zero the gradient explicitly” after using it for parameter updates.

Thus the correct code should look like:

```
def training_loop(n_epochs, lr, params, t_u, t_c):
    for epoch in range(1, n_epochs+1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= lr * params.grad

    return params
```

note we are **encapsulating the update in a no_grad context**. Within this block, autograd looks away.

Also, we are updating params “in place”. This will be modified later when we use the optimizer packages to automate the process, immediately below.

```
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs+1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad() # zeros the grad
        loss.backward()
        optimizer.step() # updates value

    return params
```

Note that right now we make an assumption that we have a linear relationship, thus the params is 2-dim. Later in ch6 we will train on the same data and loss function, but we remove this assumption and just feed the algorithm a general nn.

Now, we split datasets to training and validation to avoid overfitting.

```
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples) # 20% as validation
```

```
shuffled_indices = torch.randperm(n_samples)
train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u

def training_loop(...):
    # the only thing different is we evaluate validation loss in addition, in every step
```

The important thing to remember is that we don't need to keep track of gradients/computational graph when we evaluate params with the validation set, therefore, we could do something like:

```
...
with torch.no_grad():
    val_t_p = ...
    val_loss = ...

optimizer.zero_grad()
train_loss.backward()
optimizer.step()
```

Sometimes we use the "set_grad_enabled" context:

```
def calc_forward(t_u, t_c, is_train):
    with torch.set_grad_enabled(is_train):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

    return loss
```

Ch6 Using a Neural Network to Fit the Data

solve the same problem via a full NN.

Multilayer NN is made up of compositions of functions:

```
x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
```

```
y = f(w_n * x_n + b_n)
```

we need to replace our linear model with a NN unit. torch.nn is the submodule dedicated to nn: it contains the building blocks needed to create all sorts of NN architectures.

The building blocks are called "modules" in Pytorch parlance, or layers in other frameworks.

A Pytorch module is a python class deriving from the nn.Module base class. A module contains Parameter instance as attributes, which are tensors whose values are optimized during training.

A module can also have submodules as attributes, and it will be able to track their parameters as well. Check via "nn.ModuleList" or "nn.ModuleDict".

All PyTorch-provided subclasses of nn.Module have a "**call**" method. This allows instantiating an nn.Linear, and call it as if it is a function:

```
import torch.nn as nn

linear_model = nn.Linear(1,1)
linear_model(t_un_val) # recall t_un_val contains 2 data points
```

the output is a tensor of size 2 by 1.

calling an instance of nn.Module with a set of arguments ends up calling a method "forward" with the same arguments.

the "forward" method is what executes the computation, but **call** does other things besides calling "forward". So try to use call, do not use forward directly:

```
y = model(x) # good
y = model.forward(x) # bad
```

The constructor to " "nn.Linear" accepts 3 arguments:

1. the number of input features
2. the number of output features
3. does it include bias (default true)

```
linear_model = nn.Linear(1,1) # 1 input-dim, 1 output-dim
linear_model(t_un_val)

linear_model.weight # this gives w
linear_model.bias # this gives b
```

Often we need to batch the inputs and feed it to the model in one go. The input tensor is of size $B \times Nin$:

```
x = torch.ones(10,1)
linear_model(x) # gives a tensor of shape (10,1)
```

So if we are given data as: t_c = [1,2,3,4,5] t_u = [2,3,4,5,6]

then we need to reshape inputs to $B \times Nin$, where $Nin = 1$:

```
t_c = torch.tensor(t_c).unsqueeze(1)
t_u = torch.tensor(t_u).unsqueeze(1)
```

The resulting t_c is a tensor of shape (5,1). It looks like a 5 by 1 column vector.

The whole implementation looks like this:

```
def training_loop(n_pochs, optimizer, model, loss_fn, t_u_train, t_u_val, t_c_train, t_c_val

    for epoch in range(1, n_epochs+1):
        t_p_train = model(t_u_train)
        loss_train = loss_fn(t_p_train, t_c_train)

        t_p_val = model(t_u_val)
        loss_val = loss_fn(t_p_val, t_c_val)

        optimizer.zero_grad()
        loss_train.backward()
        optimizer.step()
```

Note that we could modify the code to use nn.MSELoss() as argument for loss_fn, instead of our handwritten code.

Final step, replace this linear model with a multi-layerd NN. The simple way to concatenate modules is through the "nn.Sequential" container:

```
seq_model = nn.Sequential(nn.Linear(1, 13), nn.Tanh(), nn.Linear(13,1))
```

We could name the parameters in a certain way to make inspecting parameters easier.

Ch7 Telling Birds from Airplanes: Learning from Images

First we download data CIFAR10, and this is downloaded as a torchvision dataset. Some common methods for datasets are:

```
len(cifar10)
cifar[4] # calling the __getitem__ method
```

For image data, we also need the "transforms.ToTensor()" method to turn images into tensors. In the end, the resulting "img_t" has torch.Size([3, 32, 32]), with type torch.float32. Other useful methods include "transforms.Normalize".