Gradient calculations with PyTorch

October 28, 2019

1   Introduction

The upcoming Reinforcement learning assignment will involve some gradient computations. While all of this can be done using only NumPy, the calculations can be made easier by using automatic differentiation packages, such as PyTorch. The PyTorch interface resembles NumPy, but PyTorch enhances it with a bunch of new features and improvements, such as calculating derivatives, GPU computing, etc. The PyTorch documentation is available at `https://pytorch.org/docs/stable/index.html`.

2   Getting started

Let's revisit the least squares example from the Python introduction. Recall that in order to perform the optimization with gradient descent, we had to manually calculate the derivatives of the objective w.r.t. the parameters:

```
1  # Gradients of loss w.r.t. a and b
2  dlda = np.mean(2 * xs**2 * self.a + 2*self.b * xs - 2*xs*ys)
3  dldb = np.mean(2*xs  * self.a + 2*self.b - 2*ys)
```

While calculating derivatives is generally easy, it may quickly become a tedious task if the model is complex. However, since the calculations are usually straight forward (and narrow down to applying the chain, product and quotient rules a bunch of times), this process can be automated — and here's where automatic differentiation libraries, like PyTorch, come in handy.

Let's see how that program could be rewritten using PyTorch.

2.1   Converting least squares

The basic class in PyTorch is `torch.Tensor`, which can be thought of as the PyTorch equivalent of a NumPy array (`numpy.ndarray`).

The first step in rewriting the least squares code is to replace our NumPy arrays with Torch tensors. The most general way of doing that is to directly convert an array to a tensor with the `torch.from_numpy` function:

```
1   tensor = torch.from_numpy(some_numpy_array)
```

However, usually a cleaner code can be produced by directly replacing NumPy calls with corresponding Torch functions. In many cases, the names and syntax remain the same — for example, `np.ones(shape)` becomes `torch.ones(shape)`, and `np.linspace(start, stop, n)` can be replaced with `torch.linspace(start, stop, n)`. There are, however, cases where the names are slightly different — for example, pseudoinverse is calculated in NumPy with `np.linalg.pinv(matrix)`, while in PyTorch it's `torch.pinverse(matrix)`. Similarly, the `randn` function is located within the core module in PyTorch — so, `np.random.randn(*shape)` becomes `torch.randn(*shape)`.

Therefore, if we look at the `GradientRegressor.fit` function, the following snippet

```
1   # Randomly initialize a and b (a,b ~ N(0, 1))
2   self.a = np.random.randn()
3   self.b = np.random.randn()
```

would change to

```
1   # Randomly initialize a and b (a,b ~ N(0, 1))
2   self.a = torch.randn(1)  # PyTorch requires the shape to be explicitly set to 1
3   self.b = torch.randn(1)
```

In the same way, `np.mean` in lines 39, 47 and 48, `np.ones` in line 72 of the original NumPy tutorial change to `torch.mean` and `torch.ones`, respectively. Similar changes have to be made throughout the code to calls to other NumPy functions in the code.

There's another change that has to be made — Matplotlib does not support plotting Torch tensors, and they need to be converted back to NumPy before passing them to `plt.plot`. This is done simply by the `.numpy()` function:

```
1   numpy_array = some_torch_tensor.numpy()
```

The code after these changes can be found in `least_squares_torch_0.py`

2.2 Automatic differentiation

So far, so good — the code still works (hopefully) but so far, except for changing the library, we haven't really seen any benefits from using Torch. Let's look again at the derivative calculations:

```
1   # Gradients of loss w.r.t. a and b
2   dlda = np.mean(2 * xs**2 * self.a + 2*self.b * xs - 2*xs*ys)
3   dldb = np.mean(2*xs  * self.a + 2*self.b - 2*ys)
```

We'd like to have those derivatives calculated *by itself*, instead of specifying the formulas manually. First,

we need to tell PyTorch that we're interested in gradients w.r.t. `a` and `b`. This can be done by passing `requires_grad=True` to the function creating the tensor:

```
1  self.a = torch.randn(1, requires_grad=True)
2  self.b = torch.randn(1, requires_grad=True)
```

Now every term calculated based on `a` and `b` will allow us to calculate the gradient using the `backward` function:

```
1  loss = torch.mean((self.a * xs + self.b - ys)**2)
2  loss.backward()
```

The gradient is calculated w.r.t. every term with `requires_grad=True` that the `loss` variable depends on; so, in our case, it calculates $\nabla_{a,b}\mathcal{L}$. The resulting gradient for each parameter is stored in `param.grad` — in this case, `self.a.grad` and `self.b.grad`.

The parameter update thus becomes:

```
1  # Update values
2  self.a.data = self.a - self.lr * self.a.grad
3  self.b.data = self.b - self.lr * self.b.grad
```

The `.data` field has to be used, such that our parameter updates are not tracked by the gradient computation mechanism (which would cause a huge mess). Don't worry about this; this procedure will be simplified in the next section.

The `backward` function, by default, accumulates gradients in the `.grad` buffer. This means that every call to `backward` **adds the gradient** to what is currently stored in the buffer, instead of overwriting it. This is useful when dealing with large models and dataset/batch sizes, where the whole data doesn't fit into memory and the gradients have to me calculated for a larger number of samples and averaged. In our case, we want to calculate the current gradient value at each iteration, so we have to manually zero out the gradients:

```
1  self.a.grad[:] = 0
2  self.b.grad[:] = 0
```

And that's it! Now we're able to perform our optimizations without calculating any derivatives manually. The code can be found in `least_squares_torch_1.py`

2.3 Using optimizers

Even though the code is now simpler, it still include a bunch of manual steps, such as updates and gradient resets performed for each parameter. These can also be simplified by using optimizers.

To get started, we need to instiantiate an optimizer inside our GradientRegressor class. We can do it inside the `fit` method (or inside `__init__`, with a `self` prefix):

```
1  optimizer = torch.optim.SGD([self.a, self.b], lr=self.lr)
```

This call instantiates the basic Stochastic Gradient Descent (SGD) optimizer. The first argument is an iterable (tuple, list, generator...) with parameters that the optimizer will update, and `lr` represents the learning rate.

Now, the parameter update can be replaced with a call to `optimizer.step()`, and gradients can be zeroed with `optimizer.zero_grad()`:

```
1  # Update values
2  optimizer.step()
3  optimizer.zero_grad()
```

In addition to the basic SGD optimizer, more advanced algorithms are also available (RMSprop, Adagrad, Adam...). The final code can be found in `least_squares_torch_2.py`

3   More complex operations

In this section, we'll talk about scaling gradients using the *stop-gradient operator*.

3.1   Scaling gradients

Imagine a scenario where we have a parametrized function (for example, a neural network) $f_\theta(x)$ and we're interested in scaling the calculated gradients by the values of some other function, $g(x)$, before performing the parameter update:

$$\theta \leftarrow \theta + \alpha \sum_i^N g(x_i) \nabla_\theta f_\theta(x_i) \tag{1}$$

We've seen this scenario in the policy gradient case, where the gradients were modulated with the discounted return G. The gradients can easily be calculated with

```
1  f_values = f(x)  # Parametrized by theta
2  g_values = g(x)  # Not parametrized by theta
3
4  objective = torch.sum(g_values * f_values)
5  objective.backward()
```

Keep in mind, however, that the `backward` call actually calculates the gradient of the sum $\nabla_\theta \sum_i^N g(x_i) f_\theta(x_i)$, and not $\sum_i^N g(x_i) \nabla_\theta f_\theta(x_i)$. These terms are, however, the same — the sum of gradients is gradient of the sum, and $g(x)$ does not depend on θ, and can therefore be treated as a constant for the sake of differentiation:

$$\nabla_\theta \sum_i^N g(x_i) f_\theta(x_i) \;=\; \sum_i^N \nabla_\theta g(x_i) f_\theta(x_i) \;=\; \sum_i^N g(x_i) \nabla_\theta f_\theta(x_i) \tag{2}$$

But how about a scenario where g is also parametrized by the same parameter vector θ (for example, both are the output of the same neural network)? In that case, the second part of Equation (2) does not hold anymore, as we cannot simply pull $g_\theta(x_i)$ outside of the differentiation operator ∇_θ. If we follow the approach from the listing above, we end up calculating $\sum_i^N \nabla_\theta g(x_i) f_\theta(x_i)$, and end up with a different result.

This can be changed by treating the values of $g_\theta(x)$ as numbers, and forgetting about the parametrization w.r.t. θ. In math, this is sometimes denoted by the *stop-gradient* operator $\perp$. The resulting function is identical in terms of values:

$$\perp(g_\theta(x)) \rightarrow g_\theta(x) \tag{3}$$

The derivatives w.r.t. parameters, however, are set to zero:

$$\nabla_\theta \perp (g_\theta(x)) \rightarrow 0 \tag{4}$$

Therefore, if we take the gradient of the product (summation omitted for notation clarity):

$$\nabla_\theta \perp (g_\theta(x_i)) \, f_\theta(x_i) \;=\; \nabla_\theta \perp (g_\theta(x_i)) \, f_\theta(x_i) \;=\; \perp(g_\theta(x_i)) \nabla_\theta f_\theta(x_i) \;+\; f_\theta(x_i) \nabla_\theta \perp (g_\theta(x_i)) \tag{5}$$

However, since by definition of $\perp$ we get $\nabla_\theta \perp (g_\theta(x)) \rightarrow 0$, the last term of (5) disappears:

$$\perp(g_\theta(x_i)) \nabla_\theta f_\theta(x_i) \;+\; f_\theta(x_i) \nabla_\theta \perp (g_\theta(x_i)) = g_\theta(x_i) \nabla_\theta f_\theta(x_i) \tag{6}$$

In PyTorch, the *stop-gradient* operation can be done by detaching the tensor from its computation graph with the detach function:

```
1  f_values = f(x)  # Parametrized by theta
2  g_values = g(x)  # Also parametrized by theta
3
4  g_stopgrad = g_values.detach()  # Same as g_values, but not parametrized by theta anymore
5
6  objective = torch.sum(g_stopgrad * f_values)
7  objective.backward()
```