a_gental_introduction_to_torch_autograd

February 27, 2021

1 A Gental Introduction To torch.autograd

`torch.autograd` is PyTorch's automatic differentiation engine which powers neural network training.

1.1 Background

Training a neural network happens in two steps:
* Forward propogation: run the input data and obtain output through the nested functions of this NN. * Backward propogation: NN adjusts its parameters proportionate to the error in its guess through gradient descent methods.

1.2 Usage in PyTorch

For instance, we load a pretrained *resnet18* model from `torchvision`.
We create a random data tensor to represent a single image with 3 channels, and hight & weight of 64.
The corresponding label is initialized to some random values.

[10]:
```
import torch, torchvision
model = torchvision.models.resnet18(pretrained=True)
data = torch.rand(1, 3, 64, 64)
labels = torch.rand(1, 1000)
```

Forward pass: run the input data through the model through each of its layers to make a prediction.

[11]:
```
prediction = model(data)
```

Calculate the error (loss) between the model's prediction and the corresponding label.

Backward pass: we call `.backward()` on the error tensor. Autograd then calculates and stores the gradients for each model parameter in the parameter's `.grad` attribute.

[12]:
```
loss = (prediction - labels).sum()
loss.backward()
```

Next, we load an optimizer, in this case SGD with a learning rate of 0.01 and momentum of 0.9. We register all the parameters of the model in the optimizer.

[13]:
```
optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
```

Finally, we call `.step()` to initiate gradient descent. The optimizer adjusts each parameter by its gradients stored in `.grad`.

[14]:
```
optim.step()
```

1.3 Differentiation in Autograd

How `autograd` collects gradients? Let's create two tensors with `requires_grad=True`, which tells `autograd` that every operation on these two tensors should be tracked.

[15]:
```
import torch
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
print(a, b)
```

```
tensor([2., 3.], requires_grad=True) tensor([6., 4.], requires_grad=True)
```

Create another tensor from `a` and `b`:

$$Q = 3a^3 - b^2.$$

[16]:
```
Q = 3 * a ** 3 - b ** 2
Q
```

[16]:
```
tensor([-12.,  65.], grad_fn=<SubBackward0>)
```

[17]:
```
Q.backward()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-17-42395d6027c1> in <module>
----> 1 Q.backward()

C:\Softwares\Anaconda3\lib\site-packages\torch\tensor.py in backward(self,␣
 ↪gradient, retain_graph, create_graph)
    219                 retain_graph=retain_graph,
    220                 create_graph=create_graph)
--> 221         torch.autograd.backward(self, gradient, retain_graph,␣
 ↪create_graph)
    222
    223     def register_hook(self, hook):

C:\Softwares\Anaconda3\lib\site-packages\torch\autograd\__init__.py in␣
 ↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables)
    124
    125     grad_tensors_ = _tensor_or_tensors_to_tuple(grad_tensors,␣
 ↪len(tensors))
--> 126     grad_tensors_ = _make_grads(tensors, grad_tensors_)
```

```
    127     if retain_graph is None:
    128         retain_graph = create_graph

C:\Softwares\Anaconda3\lib\site-packages\torch\autograd\__init__.py in␣
↪_make_grads(outputs, grads)
     48             if out.requires_grad:
     49                 if out.numel() != 1:
---> 50                     raise RuntimeError("grad can be implicitly created␣
↪only for scalar outputs")
     51                 new_grads.append(torch.ones_like(out,␣
↪memory_format=torch.preserve_format))
     52             else:

RuntimeError: grad can be implicitly created only for scalar outputs
```

Why there is an error? Because we can only call `.backward()` only when the output (`Q` here) is a scalar. Now, since both `a` and `b` are two-dim tensors, thus `Q` is a vector, then we need to pass a `gradient` argument in `Q.backward()`.

'gradient' is a tensor of the same shape as `Q`, here we choose $[1, 1]$. Before we give the reason, let's introduce the **vector calculus using `autograd`** first.

1.4 Vector Calculus using `autograd`

Mathematically, if we have a vector valued function $\vec{y} = f(\vec{x})$, then the gradient matrix is called Jacobian matrix

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}.$$

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector $\vec{v}$, it returns $J^T\vec{v}$.

If l is a scalar function w.r.t. $\vec{y}$, namely $l = g(\vec{y})$ and $\vec{v}$ is the gradient of this scalar function, that is,

$$\vec{v} = \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \cdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix},$$

then

$$J^T\vec{v} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \cdots & \cdots & \cdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \cdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \cdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}.$$

Let's go back to the previous example $Q = 3a^3 - b^2$.

If we let $l = Q_1 + Q_2$, then $\vec{v} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. We pass this $\vec{v}$ as a `gradient` argument in `Q.backward()`,

then its execution, we obtain the vector-Jacobian product

$$\begin{pmatrix} \frac{\partial(Q_1+Q_2)}{\partial a_1} \\ \frac{\partial(Q_1+Q_2)}{\partial a_2} \\ \frac{\partial(Q_1+Q_2)}{\partial b_1} \\ \frac{\partial(Q_1+Q_2)}{\partial b_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial Q_1}{\partial a_1} \\ \frac{\partial Q_2}{\partial a_2} \\ \frac{\partial Q_1}{\partial b_1} \\ \frac{\partial Q_2}{\partial b_2} \end{pmatrix},$$

which comes from the fact that $Q_1 = 3a_1^3 - b_1^2$ and $Q_2 = 3a_2^3 - b_2^2$. Up to now, we obtain the gradients, which are stored in the corresponding tensor's `.grad` attribute, namely `a.grad` and `b.grad`.

[9]:
```
external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)
```

Check if collected gradients are correct

[10]:
```
print(9 * a ** 2 == a.grad)
print(-2 * b == b.grad)
```

```
tensor([True, True])
tensor([True, True])
```

Equivalently, we can find from the choice of $\vec{v}$ that, we can directly call `Q.sum().backward()`.

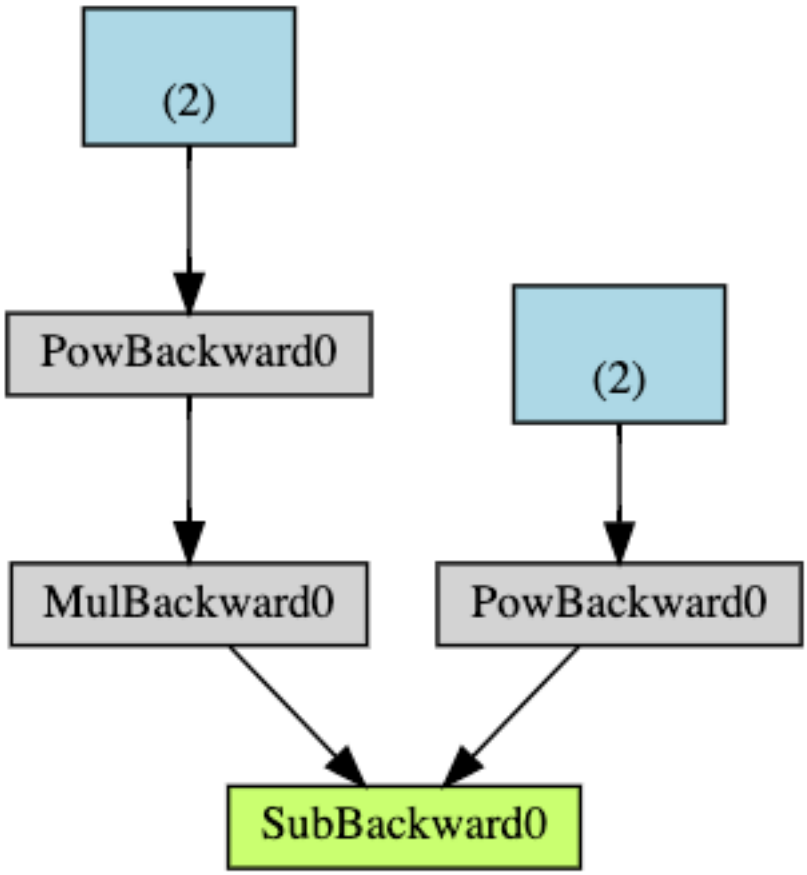
[17]:
```
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
Q = 3 * a ** 3 - b ** 2
Q.sum().backward()
print(9 * a ** 2 == a.grad)
print(-2 * b == b.grad)
```

```
tensor([True, True])
tensor([True, True])
```

1.5 Computation Graph

Below is the visual representation of the DAG in the above example. DAG consists of *Function* objects. Leaves are the input tensors, roots are the output tensors.

The arrows are in the direction of the forward pass. The nodes represent the backward functions of each operation in the forward pass. The leaf nodes in blue represent our leaf tensors `a` and `b`.



We can see `a` takes power first, then times 3, while `b` takes power and times 1. At last, there is a subtraction between $3a^3$ and b^2.

DAGs are dynamic in PyTorch. After each `.backward()` call, autograd starts populating a new graph. We can change the shape, size and operations at every iteration if needed.

1.6 Exclusion from the DAG

`torch.autograd` tracks operations on all tensors which have their `requires_grad` flag set to `True`. Setting this attribute to `False` excludes the tensor from the gradient computaion DAG.

We show that even if only a single input tensor has `requires_grad=True`, the output tensor still will require gradient.

[18]:
```
import torch
x = torch.rand(5, 5)
y = torch.rand(5, 5)
z = torch.rand(5, 5, requires_grad=True)

a = x + y
b = x + z
print(f"Does 'a' require gradients? {a.requires_grad}")
```

```
print(f"Does 'b' require gradients? {b.requires_grad}")
```

```
Does 'a' require gradients? False
Does 'b' require gradients? True
```

Frozen parameters: parameters that don't compute gradients. Useful if you know in advance that you won't need the gradients of those parameters, and reduce autograd computaions.

Finetuning a pretrained network: freeze most of the model and typically only modify the classifier layers to make predictions on new labels. As before, we load a pretrained resnet18 model, and freeze all the parameters.

[19]:
```
from torch import nn, optim
model = torchvision.models.resnet18(pretrained=True)

for param in model.parameters():
    param.requires_grad = False
```

We want to finetune the model on a new dataset with 10 labels. In resnet, the classifier is the last linear layer `model.fc`. We can simply replace it with a new linear layer (unfrozen by default) that acts as our classifier.

[20]:
```
model.fc = nn.Linear(512, 10)
```

Now the only parameters that compute gradients are the weights and bias of `model.fc`.

[21]:
```
# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

Although we register all the parameters in the optimizer, the only parameters that are computing gradients and hence updated in gradient descent are the weights and bias of the classifier.

The same exclusionary functionality is available as a context manager in `torch.no_grad()`.