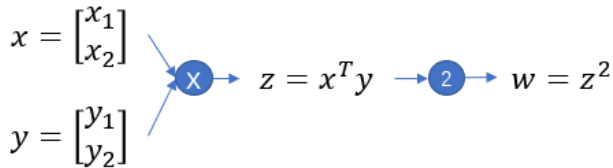
PyTorch: Use **`create_graph`** to Compute Second-order Derivative

Written by Tongyu Lu

This article assumes that readers have been familiar with the concept of computation graph.

Suppose we want to back-propagate through such a computation graph:



We shall do this in PyTorch:

```
In [1]: import torch
```

```
In [2]:
1 x = torch.tensor([3.0,1.0], requires_grad=True)
2 y = torch.tensor([1.0,2.0], requires_grad=True)
3 z = torch.sum(x*y)
4 w = z**2
5 w.backward() #get dw/dx, dw/dy
6 print(x.grad)
7 print(y.grad)
8 print(z.grad)
9
```

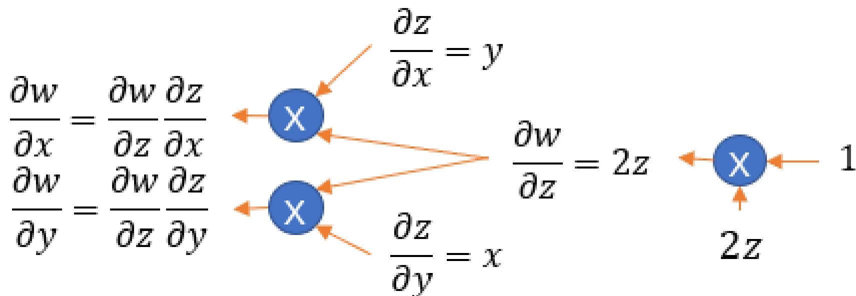
```
tensor([10., 20.])
tensor([30., 10.])
None
```

$x, y \in \mathrm{R}^2, z \in \mathrm{R}$

$w = z^2, \quad z = x^T y, \quad w(x, y) = (x^T y)^2$

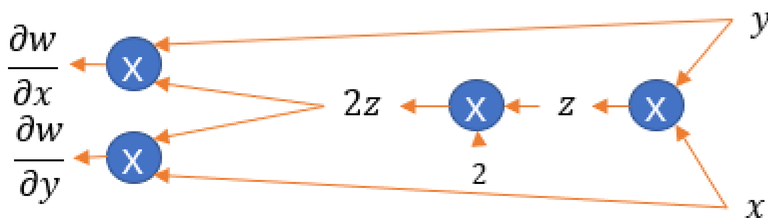
$\nabla_x w = 2y^T xy, \quad \nabla_y w = 2x^T yx$

The corresponding backward graph can be shown as follows:

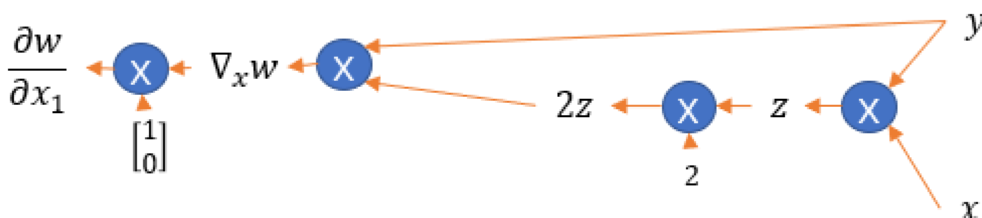


Now, how can we compute $\partial^2 w / \partial x_1^2$ using computation graph without doing calculus?

First, we construct the computation graph for $\partial w / \partial x$ (just based on the back-propagation computation graph)

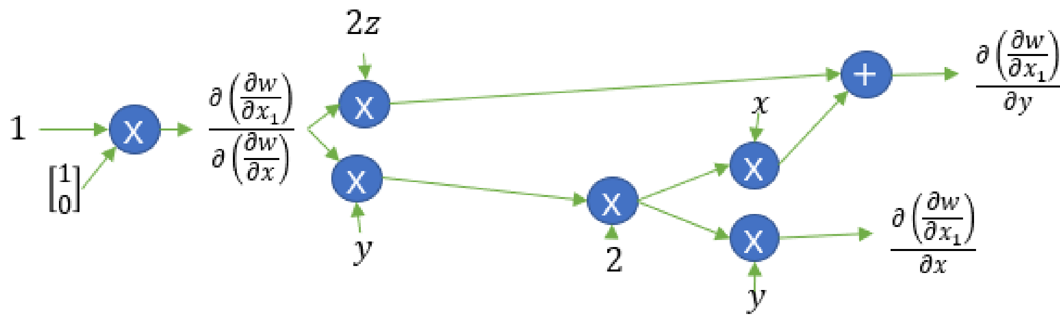


Then, we construct the computational graph for $\partial w / \partial x_1$



Now, treat this computational graph shown above as a feed-forward network from right to left.

Then, how to do back-propagation for this "1-order derivative computation graph"? That is in the same fashion as the back-propagation in general case:



If you check this result with analytical solutions acquired by calculus, you will find that they are same.
Now, we flow data through this computation graph and we get the final 2-order derivatives.
Therefore, if we want to compute a term which contains gradients (e.g. regularization for gradients), we shall plug in the "1-order derivative computation graph", and finally we can treat that term as a feed-forward computation graph.

But how to realize this idea in PyTorch? We shall do this:

```
In [3]:
x = torch.tensor([3.0,1.0], requires_grad=True)
y = torch.tensor([1.0,2.0], requires_grad=True)
z = torch.sum(x*y)
w = z**2
'''get dw/dx, dw/dy, and get the formula (computation graph) for dw/dx1, dw/dx2, dw/dy1, dw/dy2'''
w.backward(create_graph=True)
print(x.grad)
print(y.grad)

partial_x_1 = x.grad[0]
print(partial_x_1)
x.grad.data.zero_() #release grad logging
y.grad.data.zero_() #release grad logging
print(x.grad)
print(y.grad)


partial_x_1.backward() #get d(dw/dx1)/dx, d(dw/dx1)/dy
print(x.grad) #[d2w/dx1dx1, d2w/dx1dx2]
print(y.grad) #[d2w/dx1dy1, d2w/dx1dy2]
```

```
tensor([10., 20.], grad_fn=<CloneBackward>)
tensor([30., 10.], grad_fn=<CloneBackward>)
tensor(10., grad_fn=<SelectBackward>)
tensor([0., 0.], grad_fn=<CloneBackward>)
tensor([0., 0.], grad_fn=<CloneBackward>)
tensor([2., 4.], grad_fn=<CloneBackward>)
tensor([16., 2.], grad_fn=<CloneBackward>)
```

We shall see that: once we do `.backward(create_graoh=True)`, `x.grad`, `y.grad`, will have additional attribute `grad_fn=<CloneBackward>` which indicate that the gradients are differentiable.
This means that, we can treat the grads just as the middle variables such as `z`.
What will happen if we throw away `.grad.data.zero_()`?

```
In [4]:
x = torch.tensor([3.0,1.0], requires_grad=True)
y = torch.tensor([1.0,2.0], requires_grad=True)
z = torch.sum(x*y)
w = z**2
w.backward(create_graph=True) #get dw/dx, dw/dy, and get the formula (computation graph) for dw/dx1, dw/dx2, dw/dy1, dw/dy2
print(x.grad)
print(y.grad)

partial_x_1 = x.grad[0]
print(partial_x_1)
#without releasing grad logging

partial_x_1.backward() #get d(dw/dx1)/dx, d(dw/dx1)/dy
print(x.grad) #[d2w/dx1dx1, d2w/dx1dx2]+[dw/dx1, dw/dx2]
print(y.grad) #[d2w/dx1dy1, d2w/dx1dy2]+[dw/dy1, dw/dy2]
```

```
tensor([10., 20.], grad_fn=<CloneBackward>)
tensor([30., 10.], grad_fn=<CloneBackward>)
tensor(10., grad_fn=<SelectBackward>)
tensor([12., 24.], grad_fn=<CloneBackward>)
tensor([46., 12.], grad_fn=<CloneBackward>)
```

We shall see that the result is the addition between the 1-order derivative and the 2-oder derivative. This is because we did not release the 1-order gradient before calculating the 2-oder derivative. The default consideration of PyTorch is to add up the two gradients obtained by the two `backward()` operations.

An Application Using **create_graph**: MAML

This article assumes that readers have known something about meta learning and MAML.

Refer to the slide created by Hungyi Lee (meta learning and MAML):
http://speech.ee.ntu.edu.tw/~tlkagk/courses/ML_2019/Lecture/Meta1%20(v6).pdf
And the source code of homework 13 (for MAML toy example):
https://colab.research.google.com/drive/1MFJwRdOTefd6UOYRsNjdc7BWuB7Qe3lY
All homework are available at:
http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

Recall what MAML wants to do is to compute a single update step on each training task to update the model parameter in each training task.
Formally, our model is defined as $\hat{Y} = f(X|\theta)$ where θ is the model parameter. for each task i, the initial model parameter ϕ is updated into $\theta_i^* = \phi - \epsilon\nabla_\phi l_i\big(f\big(X_i^{train}|\phi\big), Y_i^{train}\big)$, and meta loss $L = \sum_i l_i(f(X_i^{test}|\theta_i^*), Y_i^{test})$
From the formulation we can see that our meta loss L is a function of θ_i^*, and θ_i^* is a function of ϕ and ∇_ϕ.
Therefore, if we want to update ϕ minimizing L, we must be able to obtain the computation graph for ∇_ϕ.
Now, you may realize that if you want to implement MAML using PyTorch, you may consider `create_graph` operation!

Here is how this operation is used in meta-training:

```
epoch = 1
for e in range(epoch):
    meta_model.model.train()
    for x, y in tqdm(train_loader):
        x = x.to(device)
        y = y.to(device)
        sub_models = meta_model.gen_models(bsz)

        meta_l = 0
        for model_num in range(len(sub_models)):

            sample = list(range(10))
            np.random.shuffle(sample)

            #pretraining for only 1 step
            pretrain_optim.zero_grad()
            y_tilde = pretrain(x[model_num][sample[:5],:])
            little_l = F.mse_loss(y_tilde, y[model_num][sample[:5],:])
            little_l.backward()
            pretrain_optim.step()

            # meta learning

            y_tilde = sub_models[model_num](x[model_num][sample[:5],:])
            little_l = F.mse_loss(y_tilde, y[model_num][sample[:5],:])
            #compute gradient ∇ϕ, obtain its computation graph for high-order gradient
            little_l.backward(create_graph = True)
```

```
            sub_models[model_num].update(lr = 1e-2, parent = meta_model.model)
            #clear gradient in optimizer (avoid from gradient cumulation)
            meta_optimizer.zero_grad()

            #compute 2nd-order gradient
            #in detail: the update() method in sub_model is defined as such:
            #layers[par].weight = layers[par].weight-lr*parent_layers[par].weight.grad
            #parent_layers[par].weight.grad has computation graph because of
            #create_graph=True
            #therefore, when again using sub_models for forwarding, we actually
applying computation graph of grad. Therefore, the meta-update will consider the
computation graph of grad.
            y_tilde = sub_models[model_num](x[model_num][sample[5:],:])
            meta_l = meta_l + F.mse_loss(y_tilde, y[model_num][sample[5:],:])

        meta_l = meta_l / bsz
        meta_l.backward()
        meta_optimizer.step()
        meta_optimizer.zero_grad()
```

Hope this example could help you!

And of course, MAML does not necessarily use 2nd derivative, just for simplification. You may think about how to do that with PyTorch!