6️⃣

AutoGrad

강의 소개

PyTorch에서 제공하는 autograd 패키지에 대해서 공부합니다. autograd 패키지는 PyTorch Tensor의 모든 연산에 대해 자동 미분을 제공합니다.

강의에서 소개되었던 개념들을 실제 구현할 때 필요한

(1) gradient들이 computational graph를 통해 input으로 전달하는 방법과

(2) user-defined function을 hook하는 방법을 중점적으로 다뤘습니다.

Autograd의 어떤 기능들을 활용할지 고민하시면서 들으시면 내용을 더 깊게 이해하실 수 있습니다.

Autograd

- **Auto**matic **grad**ient calculating API
- Automatic differentiation is **a building block of every DL library** (forward & backward passes)

Tutorial

- Automatically computing gradients of y w.r.t x

```
x = torch.randn(2, requires_grad = True)
y = x * 3
gradients = torch.tensor([100, 0.1], dtype=torch.float)
y.backward(gradients)
print(x.grad)
```

→ gradients · $\frac{\partial y}{\partial x}$ = [100, 0.1] · 3 = [300, 0.3]

↳ cf. y.backward()는 → 1 · $\frac{\partial y}{\partial x}$

```
tensor([300.0000,    0.3000])
```

▼ x.grad

- Autograd가 매개변수(parameter)의 `.grad` 속성(attribute)에, 모델의 각 매개변수에 대한 변화도(gradient)를 계산하고 저장합니다.
- 참고 : torch.autograd 에 대한 간단한 소개 — PyTorch Tutorials 1.10.2+cu102 documentation

requires_grad

- `requires_grad` indicates autograd to **compute and store gradients**
  - With **requires_grad=False** option, **RuntimeError** occurs when y.backward() is called

```
x = torch.randn(2, requires_grad = True)
y = x * 3
gradients = torch.tensor([100, 0.1], dtype=torch.float)
y.backward(gradients)
print(x.grad)
```

```
tensor([300.0000,    0.3000])
```

backward

- Calling **backward() twice**, you may get **RuntimeError**
  - backward 한 번 호출 → computational graph를 모두 버림 (∵ 연산량 줄이려고)
  - Specify **.backward(retain_graph = True)** to indicate **not to free** intermediate resources
    - computational graph 버리지 않도록 설정 → gradient을 accumulation

```
gradients = torch.tensor([100, 0.1], dtype=torch.float)
y.backward(gradients, retain_graph = True)
print(x.grad)
y.backward(gradients)
print(x.grad)
```

```
tensor([300.0000,    0.3000])
tensor([600.0000,    0.6000])
```

Gradients are accumulated

grad_fn

- A tensor y is a computed result, so it contains the `grad_fn` attribute
    - Referencing **Function** (class) that is called to construct

```
x = torch.randn(2, requires_grad = True)
y = x * 3
z = x / 2
w = x + y
```

```
w, y, z
```

```
(tensor([6.2272, 2.3273], grad_fn=<AddBackward0>),
 tensor([4.6704, 1.7455], grad_fn=<MulBackward0>),
 tensor([0.7784, 0.2909], grad_fn=<DivBackward0>))
```

- grad_fn → computational graph 바로 직전의 operation (backward시 사용될 function class)

▼ 📎 grad_fn

- 해당 Variable 객체를 생성하는 Function 객체를 참조합니다. (예외 상황, Variable 을 사용자가 직접 생성한 경우에는 grad_fn이 값은 `None` 입니다.
- 참고 : [Autograd: 미분 자동화 (taewan.kim)]

hook : register_forward_hook

- Hooking is to alter other software components by **intercepting function calls** (or
  messages, events, etc.) passed between software components

- 1️⃣ Firstly, let's define a simple network composed of **3 layers**

- 2️⃣ First define **a function to be hooked**
    - type(self) should be **Tensor**

```
def hook_func(self, input, output) :
  print('Inside ' + self.__class__.__name__ + ' forward')
  print('')
  print('input: ', type(input))
  print('input[0]: ', type(input[0]))
  print('output: ', type(output))
  print('')
```



```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(10, 20, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(320, 50)
        self.out = nn.Linear(50, 10)

    def forward(self, input):
        x = self.pool1(F.relu(self.conv1(input)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc(x))
        x = F.relu(self.out(x))
        return x
```

- 3️⃣ Then, we **register hook**
    - register_forward_hook → forward 시 hook

```
net = SimpleNet()
```

```
net.conv1.register_forward_hook(hook_func)
```

```
<torch.utils.hooks.RemovableHandle at 0x7f9a6d071898>
```

```
net.conv2.register_forward_hook(hook_func)
```

```
<torch.utils.hooks.RemovableHandle at 0x7f9a6d071710>
```

- 4️⃣ **During a forward pass**, the hooked function gets called automatically

```
input = torch.randn(1, 1, 28, 28)
out = net(input)
```

```
Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.Tensor'>
output:  <class 'torch.Tensor'>

Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.Tensor'>
output:  <class 'torch.Tensor'>
```

hook : register_forward_pre_hook

- With `register_forward_pre_hook`, hook_func gets executed **before the forward pass**
    - ex. forward 전에 실행 → output X (input 내용만 출력)

```
def hook_pre(self, input) :
  print('Inside ' + self.__class__.__name__ + ' forward')
  print('')
  print('input: ', type(input))
  print('input[0]: ', type(input[0]))
```

```
net = SimpleNet()
net.conv1.register_forward_pre_hook(hook_pre)

input = torch.randn(1, 1, 28, 28)
out = net(input)
```

```
Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.Tensor'>
```

hook : register_backward_hook

- hook_func gets executed when **the gradients w.r.t. module inputs** are computed

```
net = SimpleNet()
net.conv1.register_backward_hook(hook_grad)

input = torch.randn(1, 1, 28, 28)
out = net(input)

target = torch.tensor([3], dtype=torch.long)
loss_fn = nn.CrossEntropyLoss()
err = loss_fn(out, target)
err.backward()
```

```
Inside Conv2d backward
Inside class:Conv2d

grad_input:  <class 'tuple'>
grad_input[0]:  <class 'NoneType'>
grad_output:  <class 'tuple'>
grad_output[0]:  <class 'torch.Tensor'>
```

- `grad_input` and `grad_output` mean the gradients w.r.t. input and output, respectively.

```
def hook_grad(self, grad_input, grad_output) :
  print('Inside ' + self.__class__.__name__ + ' backward')
  print('Inside class:' + self.__class__.__name__)
  print('')
  print('grad_input: ', type(grad_input))
  print('grad_input[0]: ', type(grad_input[0]))
  print('grad_output: ', type(grad_output))
  print('grad_output[0]: ', type(grad_output[0]))
  print('')
```

- The hook **should not modify** its arguments
  - But it can optionally **return a new gradient** will be used **in place of grad_input**
  - hook function 내부에서 grad_input, grad_output 자체를 변경하면 X (단, return으로 새로운 grad_output을 반환하는 것은 가능)

▼ 📎 grad_input VS. grad_output

- `grad_input` 은 forward pass로부터 계산된 **현재 layer의 출력에 대한 모델 출력의 기울기**입니다. 따라서 마지막 layer에 대해서는 모델 출력의 자기 자신에 대한 기울기이므로 [1,1]이 되게 됩니다.
- `grad_output` 은 **grad_output과 grad_output에 대한 해당 layer 입력의 기울기를 곱한 값**으로 **chain-rule**에 의한 다음 layer의 grad_output이 됩니다.

```
grad_output = grad_input * (forward_output*(1-forward_output))
# grad of sigmoid(x) w.r.t x is : sigmoid(x) * (1-sigmoid(x))
```

- 참고 : Pytorch - hook (tistory.com)

hook : remove

- `Handle.remove()` will remove the hook

```
net = SimpleNet()
h = net.conv1.register_forward_hook(hook_func)
input = torch.randn(1, 1, 28, 28)
out = net(input)
```

```
Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.Tensor'>
output:  <class 'torch.Tensor'>
```

```
h.remove()
out = net(input)
```

Toy activation example

- 1️⃣ Define **a function to be hooked**

```
save_feat = []
def hook_feat(module, input, output):
  save_feat.append(output)
  return output
```

- 2️⃣ **Register** user-defined function

```
for name, module in model.get_model_shortcuts():
  if(name == 'target_layer_name'):
    module.register_forward_hook(hook_feat)
```

- 3️⃣ Run **forward pass**
- 4️⃣ **Check save_feat**

```python
img = img.unsqueeze(0)
s = model(img)[0]
```

```python
save_feat
```

```
[tensor([[[[ 4.9739e-01,  4.8181e-01, -2.6503e-02,  ..., -9.2160e-02,
             3.6027e-01, -1.2358e-01],
           [-4.5540e-03,  9.0660e-02,  4.7405e-01,  ...,  6.2309e-02,
            -1.6621e-01, -1.1978e-01],
           [ 3.4173e-01,  1.9069e-01, -2.8974e-01,  ...,  5.9405e-02,
             3.0420e-02,  2.9952e-01],
           ...,
```

▼ model.get_model_shortcuts() ?