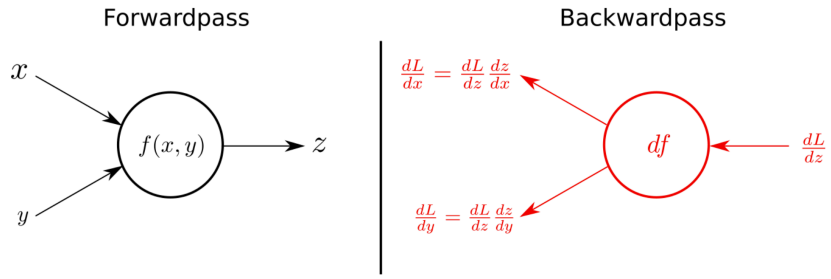
AutoGrad & Optimizer

- **torch.nn.Module**

  딥러닝 구성하는 layer의 base class

  Input, Output, Forward, Backward 정의

  학습 대상이 되는 parameter(tensor) 정의



- **nn.Parameter**

  Tensor 객체의 상속 객체

  nn.Module 내에 atrribute가 될 때, required_grad=True 지정되어 학습 대상이 되는 Tensor

  우리가 직접 지정할 일은 잘 없음

  => 대부분은 layer, weights 값들이 지정되어 있음

```
class MyLiner(nn.Module):
  def __init__(self, in_features, out_features, bias=True):
    super().__init__()
    self.in_features = in_features
    self.out_features = out_features
    self.weights = nn.Parameter( torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self, x : Tensor):
    return x @ self.weights + self.bias
```

- **Backward**

  layer에 있는 parameter들의 미분 수행

  forward의 결과값 (model의 output=예측치) & 실제값 간의 차이(loss)에 대해 미분 수행

  해당 값으로 parameter 업데이트

```
for epoch in range(epochs):
  ......
  optimizer.zero_grad() # 이전 grad 영향 안 주게
  outputs = model(inputs) # y^
  loss = criterion(outputs, labels) print(loss) # get loss for the predicted output
  loss.backward() # get gradients w.r.t to parameters
  optimizer.step() # update parameters
  .........
```

- **Backward form the scratch**

  실제 backward, module 단계에서 직접 지정 가능

  Module에서 backward, optimize 오버라이딩

  직접 미분 수식 작성 필요 => 쓸 일 없음, 순서 이해 필요

```
class LR(nn.Module):
  def __init__(self, dim, lr=torch.scalar_tensor(0.01)):
    super(LR, self).__init__()
    # intialize parameters
    self.w = torch.zeros(dim, 1, dtype=torch.float).to(device)
    self.b = torch.scalar_tensor(0).to(device)
    self.grads = {"dw": torch.zeros(dim, 1, dtype=torch.float).to(device),
            "db": torch.scalar_tensor(0).to(device)}
    self.lr = lr.to(device)

  def forward(self, x):
    ## compute forward
    z = torch.mm(self.w.T, x)
    a = self.sigmoid(z)
    return a
```

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

```
  def sigmoid(self, z):
    return 1/(1 + torch.exp(-z))

  def backward(self, x, yhat, y):
    ## compute backward
    self.grads["dw"] = (1/x.shape[1]) * torch.mm(x, (yhat - y).T)
    self.grads["db"] = (1/x.shape[1]) * torch.sum(yhat - y)

  def optimize(self):
    ## optimization step
    self.w = self.w - self.lr * self.grads["dw"]
    self.b = self.b - self.lr * self.grads["db"]
```

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$:= \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$