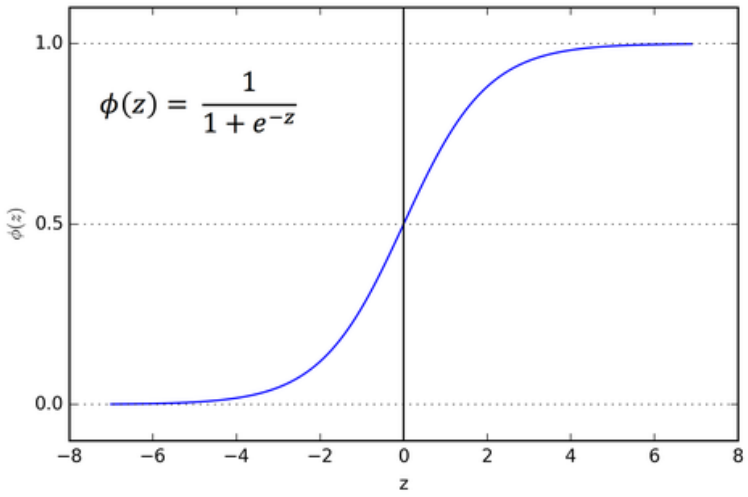
ReLu

상태	DNN
담당자	

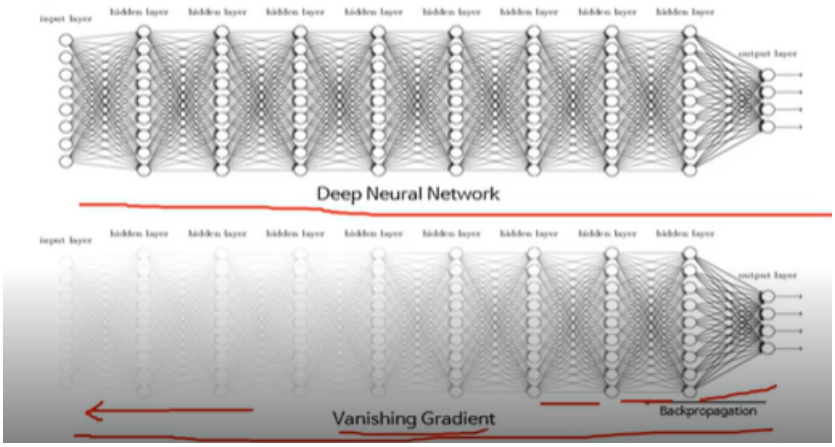
Activation function으로서의 Sigmoid의 문제점

input → network → activation function → output 의 순으로 순전파가 이루어지며 backpropagation을 통해 Loss의 gradient을 구해 weight을 업데이트한다.

Sigmoid의 문제는 gradient을 구할 때 발생하는데, sigmoid 함수의 특성 상 어느 지점에서는 gradient가 거의 0에 가깝다.



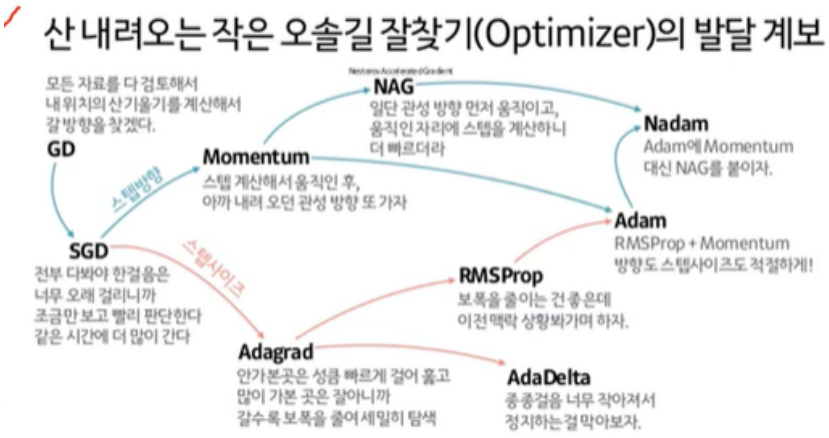
Backpropgation을 통해 gradient을 앞단으로 전파할 때 activation function을 곱하게 되는데, 아주 작은 activation function 값이 곱해지면서 앞단으로 갈수록 gradient가 소멸해버리는 문제가 발생한다. 이런 현상을 Vanishing Gradient라고 한다.



ReLU란?

> $f(x) = max(0, x)$으로 gradient은 무조건 0 혹은 1이다. 매우 작은 gradient가 곱해지지 않기 때문에 vanishing gradient 문제가 발생하지 않는다는 장점이 있다. (음수 영역에서 0이 되기는 함)

Optimizer in PyTorch



MNIST

```
# Lab 10 MNIST and softmax
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import random
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# for reproducibility
random_seed = 777

torch.manual_seed(random_seed)
torch.cuda.manual_seed(random_seed)
torch.cuda.manual_seed_all(random_seed) # if use multi-GPU
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
np.random.seed(random_seed) # if use any numpy computation
random.seed(random_seed)


# parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

```
mnist_train = dsets.MNIST(root="MNIST_data/", train=True, transform=transforms.ToTensor(), download=True)
mnist_test = dsets.MNIST(root="MNIST_data/", train=False, transform=transforms.ToTensor(), download=True)

# dataset loader
data_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True)
```

- `transform=transforms.ToTensor()` 데이터타입을 Tensor 형태로 변경
- `torch.utils.data.DataLoader` : 학습에 필요한 데이터를 로딩해주는 인스턴스
  - `drop_last` : 배치 크기를 채우지 못한 마지막 불완전 배치를 사용하지 않음

```
linear = torch.nn.Linear(784, 10, bias=True).to(device)
```

MNIST 데이터는 28 * 28 = 784 사이즈이며, 숫자는 0부터 9까지이므로 output은 10으로 설정한다.

```
torch.nn.init.normal_(linear.weight)

>>
Parameter containing:
tensor([[-0.1953, -0.4404,  0.0139,  ...,  0.7510,  1.5190,  0.9637],
        [ 0.4146, -1.0660, -0.9969,  ..., -0.3259, -0.2695, -1.3316],
        [ 0.2600, -1.1821,  0.0065,  ...,  1.0648, -1.2251,  0.9841],
        ...,
        [ 0.0279, -1.7242, -1.1723,  ...,  1.3651, -0.0689, -0.7931],
        [ 0.0952,  0.1939,  0.1636,  ..., -1.0489,  0.6224, -1.7181],
        [ 2.6504,  0.6297, -1.6571,  ..., -0.0781, -1.1398, -0.8711]],
       requires_grad=True)
```

모델의 파라미터를 초기화한다.

```
# define cost/loss & optimizer
criterion = torch.nn.CrossEntropyLoss().to(device)    # Softmax is internally computed.
optimizer = torch.optim.Adam(linear.parameters(), lr=learning_rate)
```

```
total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        # reshape input image into [batch_size by 784]
        # label is not one-hot encoded
        X = X.view(-1, 28 * 28).to(device)
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = linear(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')


>>

Epoch: 0001 cost = 4.848181248
Epoch: 0002 cost = 1.464641452
Epoch: 0003 cost = 0.977406502
Epoch: 0004 cost = 0.790303528
Epoch: 0005 cost = 0.686833322
Epoch: 0006 cost = 0.618483305
Epoch: 0007 cost = 0.568978667
Epoch: 0008 cost = 0.531290889
Epoch: 0009 cost = 0.501056492
Epoch: 0010 cost = 0.476258427
Epoch: 0011 cost = 0.455025405
Epoch: 0012 cost = 0.437031567
Epoch: 0013 cost = 0.421489984
Epoch: 0014 cost = 0.408599794
Epoch: 0015 cost = 0.396514893
Learning finished
```

▼ 🤔 클래스로 짜면 loss가 다르게 떨어지는 이유에 대해서는 고민

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# for reproducibility
random_seed = 777

torch.manual_seed(random_seed)
torch.cuda.manual_seed(random_seed)
torch.cuda.manual_seed_all(random_seed) # if use multi-GPU
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
random.seed(random_seed)

# parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

```
class SoftmaxMNIST(nn.Module): # 앞으로 모델은 다른 데다가 써놓고 임포트하자
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(784, 10)

    def forward(self, x):
        return self.linear(x)


model = SoftmaxMNIST()
criterion = torch.nn.CrossEntropyLoss().to(device) # Softmax가 internally computed
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        # reshape input image into [batch_size by 784]
        # label is not one-hot encoded
        X = X.view(-1, 28 * 28).to(device)
        Y = Y.to(device)

        hypothesis = model(X)
        cost = criterion(hypothesis, Y)

        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')

>>
Epoch: 0001 cost = 0.612622023
Epoch: 0002 cost = 0.343994766
Epoch: 0003 cost = 0.307820767
Epoch: 0004 cost = 0.291483581
Epoch: 0005 cost = 0.281717360
Epoch: 0006 cost = 0.274238199
Epoch: 0007 cost = 0.269270748
Epoch: 0008 cost = 0.265026242
Epoch: 0009 cost = 0.261815339
Epoch: 0010 cost = 0.258980662
Epoch: 0011 cost = 0.256636649
Epoch: 0012 cost = 0.254542708
Epoch: 0013 cost = 0.252477646
Epoch: 0014 cost = 0.251285672
Epoch: 0015 cost = 0.249446645
Learning finished
```

MNIST with ReLU

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```
class DNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = torch.nn.Linear(784, 256, bias=True)
        self.linear2 = torch.nn.Linear(256, 256, bias=True)
        self.linear3 = torch.nn.Linear(256, 10, bias=True)
        self.relu = torch.nn.ReLU()
        self.model = torch.nn.Sequential(self.linear1, self.relu, self.linear2, self.relu, self.linear3).to(device)

    def forward(self, x):
        return self.model(x)
```

여기서 `torch.nn.Sequential` 객체는 그 안에 포함된 객체를 순차적으로 실행해 주는 역할을 한다. 즉 `self.linear1` → `self.relu` → `self.linear2` → `self.relu` → `self.linear3` 순으로 통과하는 셈이다.

마지막에 `self.linear3`은 `softmax`을 통과해 cross-entropy loss를 구할 계획이므로 ReLU activation function을 통과하지 않도록 설계한다.

모델을 클래스 단위로 작성할 때에는 `__init__` 단에서 클래스 단위에서 필요한 모든 정의를 다 해두는 것이 좋다. 예를 들어 `self.model` 같은 경우도 처음 한 번만 그 구조를 정의하면 되므로 `__init__`단에서 정의하고 `forward` 에서는 불러와서 쓰기만 할 수 있도록 작성하는 것이 좋다. 이렇게 하면 불필요하게 함수를 호출할 때마다 모델을 새로 정의할 필요가 없으므로 computation cost을 줄일 수 있다는 이점이 있다.

```
model = DNN()
criterion = torch.nn.CrossEntropyLoss().to(device) # softmax가 internally compute
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        # reshape input image into [batch_size by 784]
        # label is not one-hot encoded
        X = X.view(-1, 28 * 28).to(device)
        Y = Y.to(device)

        hypothesis = model(X)
        cost = criterion(hypothesis, Y)

        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')
```