(07-1) Tips

- MLE

'likelihood(가능도)가 최대가 되는 θ' = '관측치를 가장 잘 설명하는 θ'를 찾아내는 과정
→ Optimization

How? Gradient Descent 사용해서 θ 업데이트.

$$\theta \leftarrow \theta - \alpha \times \frac{\partial L(x;\theta)}{\partial \theta}$$

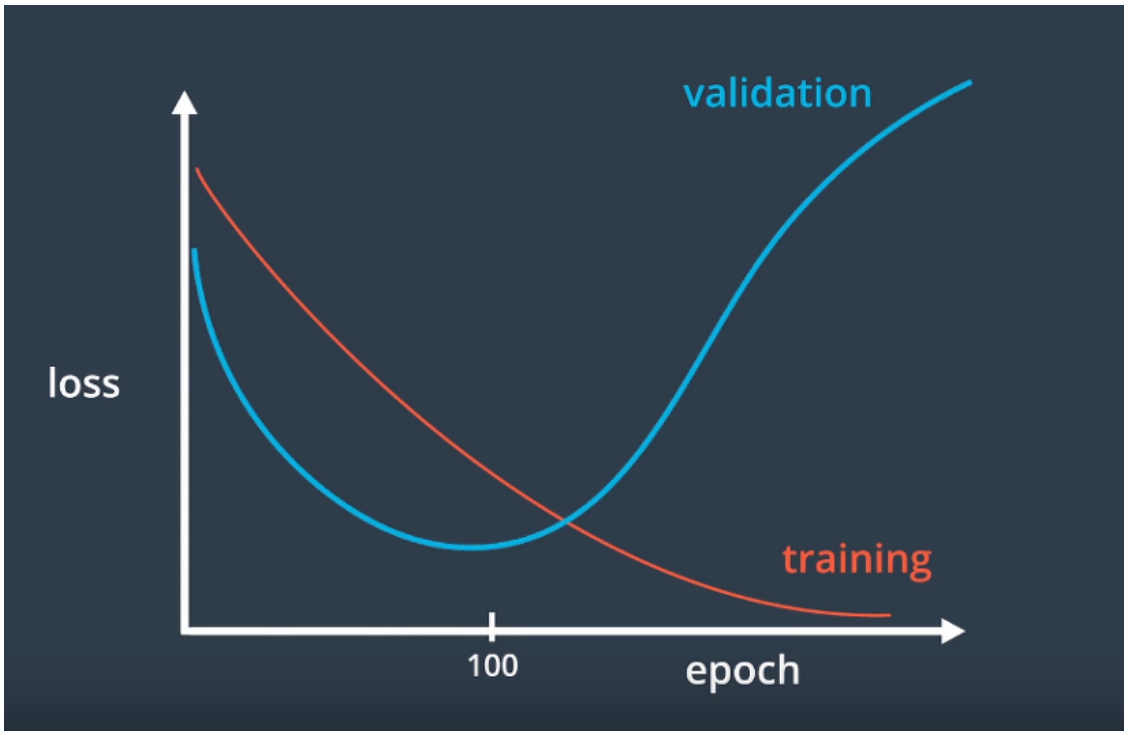
- Overfitting

MLE의 숙명적인 문제점 $\rightarrow$ 과적합 (주어진 데이터에 과도하게 fitting된 경우)

해결방법?

1. 데이터 셋 나누기

   주어진 데이터를 Training set, Test set, (Development/Validation set)으로 나눈 후 Training set으로 모델을 형성하고 Test set으로 과적합 방지. 이때 Test set에 대해서도 과적합이 발생할 수 있는데, 이를 위해 Validation set을 추가적으로 구성하기도 함.



※ **epoch**: 전체 데이터 셋에 대해 한 번 학습을 완료한 상태, 예를 들어 epoch값이 20이면 전체 데이터 셋에 대해 20번 학습이 이루어졌다는 의미.

당연히 training set으로 학습이 여러번 이루어질수록 과적합이 발생할 확률이 높고 이에 따라 Loss는 감소한다. 반면 형성된 모델을 검정할때 사용되는 validation set에 대한 loss는, 과적합으로 인해 epoch값이 커짐에 따라, 증가하는 양상을 보인다. 따라서 validation loss가 가장 작은 지점의 epoch를 선택하는 것이 바람직하다.

2. 단순히 데이터가 많을수록 편향되지 않을 가능성이 높기 때문에, 많은 데이터를 확보하는 것도 도움이 된다.

3. **Regularization**

   - Early Stopping (validation loss가 더이상 낮아지지 않을때)
   - Reducing Network Size (neural network가 학습할 양을 줄이는 것)
   - Weight Decay (neural network 의 weight parameter의 크기 제한)
   - Dropout
   - Batch normalization $\rightarrow$ 이 두 가지는 딥러닝에서 가장 흔히 사용됨, 뒤에서 다룰 예정

- Basic Approach to Train DNN(Deep Neural Network)

1) 신경망 구조 모델 형성 (input, output size 지정)

2) 모델 학습 및 과적합 확인 - input과 output사이에 layer들이 얼마나 깊고 넓은지에 따라 과적합 판단하고 과적합일 경우 위 해결방법 중 하나를 시행한 후 과정을 반복한다.

- 개념 정리 with Code

```
x_train = torch.FloatTensor([[1, 2, 1],
                             [1, 3, 2],
                             [1, 3, 4],
                             [1, 5, 5],
                             [1, 7, 5],
                             [1, 2, 5],
                             [1, 6, 6],
                             [1, 7, 7]
                            ])
y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])


x_test = torch.FloatTensor([[2, 1, 1], [3, 1, 2], [3, 3, 4]])
y_test = torch.LongTensor([2, 2, 2])
```

- *Tensor vs. Variable ?*

  2018년에 합쳐진 class로 이제는 Tensor로 통합됨. 기존에는 Variable에서 gradient를 자동으로 계산해주는 역할이었는데 이제는 Tensor가 그 기능 담당해서 굳이 Variable을 사용할 이유가 없어짐.

  *Tensor*는 *Numpy*와 유사하지만, GPU를 활용해서 속도가 더 빠르고 PyTorch에 *Tensor* 연산을 위한 다양한 함수가 존재한다.

- *torch.Tensor ?*

  데이터를 Tensor 객체로 만들어주는 함수. 이때 데이터는 list나 array류. *requires_grad* 매개변수로 gradient 값을 저장 유무를 결정함.

  위에서 Float, Long(정수)은 dtype에 따라 Tensor를 설정해준 것.

```
# 모델 정의
class SoftmaxClassifierModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 3)
    def forward(self, x):
        return self.linear(x)

model = SoftmaxClassifierModel()
```

※ 대부분의 PyTorch 구현체에서 Class를 사용하는 이유를 살펴보기에 앞서, 우선 **Class**의 개념을 설명해보겠다.

class는 붕어빵 틀이라고 생각하면 쉽다. class라는 붕어빵 틀을 형성한 후에는 서로 다른 여러개의 붕어빵을 만들 수 있듯, 객체를 만들 수 있다. 이는 특정 method를 포함하고 있는 class에 대해서 다른 데이터(재료)로, 서로에게 어떠한 영향도 주지않고 서로 다른 객체(붕 어빵)를 만들 수 있다는 것을 의미한다. method를 독립적으로 여러개 만들어서 구현해낸 객체보다 하나의 class만으로도 여러개의 객체를 구현해낼 수 있기 때문에, class가 좀 더 효율적이다.

- *SoftmaxClassifierModel (SCM)*은 sub class, *nn.Module*은 super class. 이는 *torch.nn.Module*의 속성을 *SCM*이 상속받았다고 볼 수 있다.
  - *_ init_* (생성자)

객체 생성 시 호출될때 실행되는 초기화 함수

*super()*는 sub class내에서 super class의 method를 호출하기 위한 함수로, *super()._ init _ ()* 의 의미는 *nn.Module*의 *_ init _* 을 호출한다고 볼 수 있다.

*nn.Linear(3,3)*는 3개의 input이 3개의 output으로 나오는 것을 의미한다. 다중 선형 회귀

- *forward*

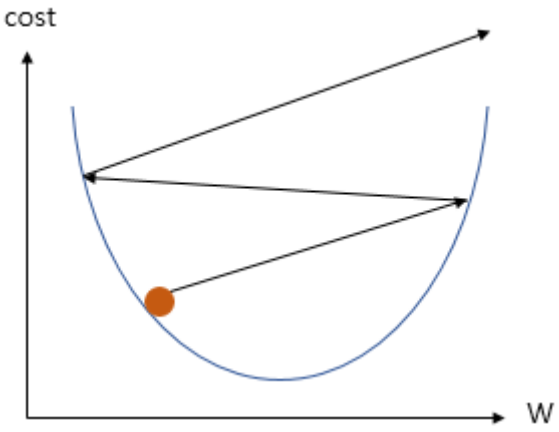
  *self.linear(x)*의 결과는 y_{pred}값에 해당된다. (아래 모델 훈련 코드에서 'prediction'으로 지정함)

- *nn.Module* ?

  PyTorch 내 *nn*패키지는 신경망 layer들과 거의 동일한 *Module*의 집합 정의.

```
# 모델 최적화
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

- *optim* ? 최적화 알고리즘에 정보 제공
- SGD ? Stochastic Gradient Descent (확률적 경사 하강법)
- *model.parameters()* ? 우리가 update해야하는 모든 변수 지정. 위 model에서 저장된 3개 (input 3)의 가중치와 편향 불러오는 함수
- *lr* ? **learning rate**, 모델의 필요한 크기보다 높으면 기울기 발산하고 너무 작을 경우 loss/cost 가 거의 줄어들지 않음 $\rightarrow$ 보통은 적절한 숫자를 잡아서 조정 과정 필요



Optimizer에 대한 좀 더 구체적인 내용은 (09-1)에서 다루기로.

```
# 전체 training set에 대해 경사하강법 20회 반복
def train(model, optimizer, x_train, y_train):
    nb_epochs = 20
    for epoch in range(nb_epochs):

        # H(x) 계산
        prediction = model(x_train)

        # cost 계산
        cost = F.cross_entropy(prediction, y_train)

        # cost로 H(x) 개선
        optimizer.zero_grad()
        cost.backward()
        optimizer.step()
```

```
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

- *F.cross_entropy* ?

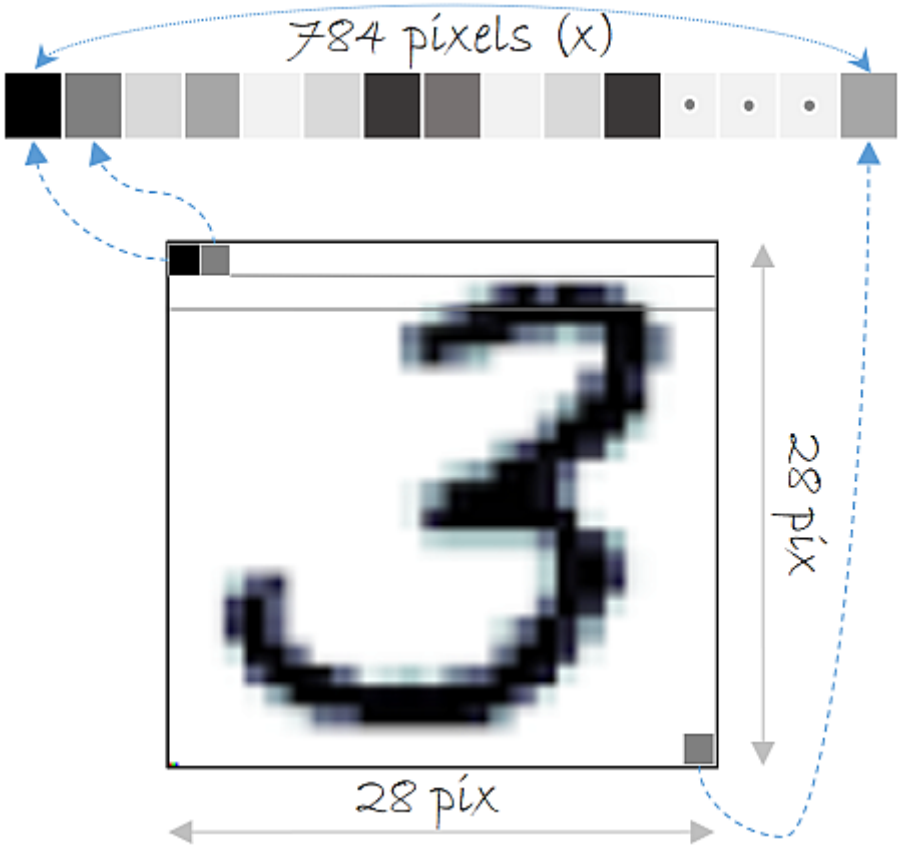
$$F.cross\_entropy = F.log\_softmax() + F.nll\_loss()$$

(F는 *torch.nn.functional*, nll은 negative log likelihood의 약자)

- line 13 - back propagarion 하기 이전에 모든 gradient를 0으로/ 왜? *.backward()*를 호출할 때마다 gradient가 buffer에 누적되기 때문에
- line 14 - *backward()*는 모든 loss에 대한 gradient 계산하는 과정으로, 이를 통해 model의 매개변수인 가중치를 업데이트하고자 함 / 참고로 loss 계산하는(여기서는 cost) line10까지는 forward()에 해당됨
- line 15 - 가중치 업데이트

(07-2) MNIST

※ Torchvision 패키지 내 데이터셋




```
# MNIST dataset
mnist_train = dsets.MNIST(root='MNIST_data/', # 경로
                          train=True,
                          transform=transforms.ToTensor(),
                          download=True)
```

- *transforms.ToTensor()* ?

Image (0-255/ height, width, channel 순)를 PyTorch (0-1, channel, height, width 순)으로 변환해주는 함수

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

- *DataLoader* ? 직관적으로 데이터를 불러올때 사용되는 함수.
  - Batch ?

    데이터의 크기가 클수록 전체 데이터에 대해 경사 하강법을 수행하는 것은 비효율적이며 불가능할 수 있다. 이때 전체 데이터를 더 작은 단위로 나눠서 해당 단위로 학습하는 개념이 mini Batch, 이때 mini Batch의 크기를 **batch size**라 부름.

    전체 데이터에 한번에 경사 하강법을 수행하는 것과 달리, 위와 같은 방법은 일부 데이터만을 보고 수행하기 때문에 최적값으로 수렴하는 과정에서 헤매기도 함. but fast
  - shuffle ?$\quad$ 전체 데이터에서 batch size만큼의 일부분을 불러올때, 순서를 섞어서 무작위로 불러올지
  - drop_last ? $\quad$

    batch size만큼 불러올때, 그 크기에 맞지 않아서 마지막에 남는 데이터들을 drop할지 안할지 결정. 이때 True하면 마지막 배치가 상대적으로 과대 평가되는 현상을 막을 수 있음.

※ Epoch / Batch size / **Iteration**

Iteratation은 한 번의 epoch 내에서 이루어지는 매개변수 parameters의 업데이트 횟수를 의미함

예를 들어 우리가 1000개의 훈련 데이터셋이 있고 batch size가 500이라 한다면, 1 epoch를 완성하기 위해서는 2번의 iteration이 필요하다.

```
# MNIST data image of shape 28 * 28 = 784
linear = torch.nn.Linear(784, 10, bias=True).to(device)
```

10은 0~9의 output을 의미함

- *to()* ? $\quad$ 모델의 parameter에 대한 연산을 어디서 수행할지 결정해서 지정한 장치의 메모리로 보냄.

```
# define cost/loss & optimizer
criterion = torch.nn.CrossEntropyLoss().to(device)    # Softmax is internally computed.
optimizer = torch.optim.SGD(linear.parameters(), lr=0.1)
```

Softmax 회귀에서는 *F.cross_entropy()*를 사용했지만 여기서는 *CrossEntropyLoss()*. 둘 모두 PyTorch에서 제공하는 cross_entropy 함수며 모두 Softmax를 포함하고 있다!

```
for epoch in range(training_epochs): #epoch =15
    avg_cost = 0
    total_batch = len(data_loader)

    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) # 'linear'로 분류(classify)한 결과

        cost = criterion(hypothesis, Y)
        cost.backward() # Gradient 계산
        optimizer.step() # Gradient로 업데이트

        avg_cost += cost / total_batch

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

print('Learning finished')
# 결과 생략
```

- X은 이미지, Y는 label (여기서는 0부터 9까지)
- *Y=Y.to(device)* ? 이미지 데이터에 대한 label (0~9), one-hot encoding이 아니라 / CUDA로 GPU를 사용하지 않는 경우 *to(device)* 부분은 모두 지움
- One-hot encoding ?

  선택지의 개수만큼 차원을 가지면서, 각 선택지의 index에 해당되면 1 아니면 0을 갖도록 one-hot vector로 표현하는 방법
- *view* ?$\quad$

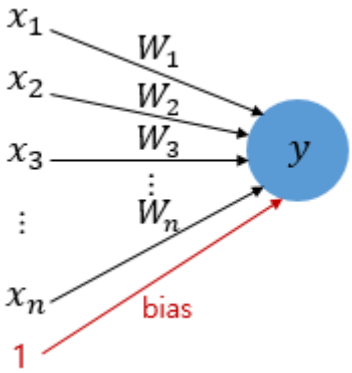
  데이터가 처음 불러올때는 '(batch size) $\times$ 1 (channel) $\times$ 28 (height) $\times$ 28 (width) '이지만, *view*를 통해 '(batch size) $\times$ 784'의 크기로 변환
- 추가적으로! Test를 진행할때는 gradient를 계산하지 않기 때문에 '*torch.no_grad()* '함수를 사용해야한다. (Test 코드는 생략, 09-1에 나와있음)

(08-1) Perceptron

- Perceptron

다수의 input에서 하나의 output을 생성하는 알고리즘으로 실제 뇌의 신경 세포 뉴런의 동작과 유사해서 인공신경망이라 부름. 아래 그림에 나타나있듯, input은 $x_1,..., x_n$이고 가중치 $W_1, ...,W_n$와 bias b를 적용시킨 output은 $\sum^{n}_{i}W_ix_i + b$이다. 이때의 output은 sigmoid와 같은 *activation function*을 거쳐서 최종적인 output이 된다.

※ activation function (비선형 활성화 함수)는 input에 수학적 변환을 수행해 output을 내는 함수이며, 그 예로는 sigmoid, ReLU, Leaky ReLU, Softmax, Hyperbolic tangent 등이 있다.



Perceptron은 초기 형태의 인공신경망으로, Linear classifier를 위해 만들어진 모형. (*b*=bias)

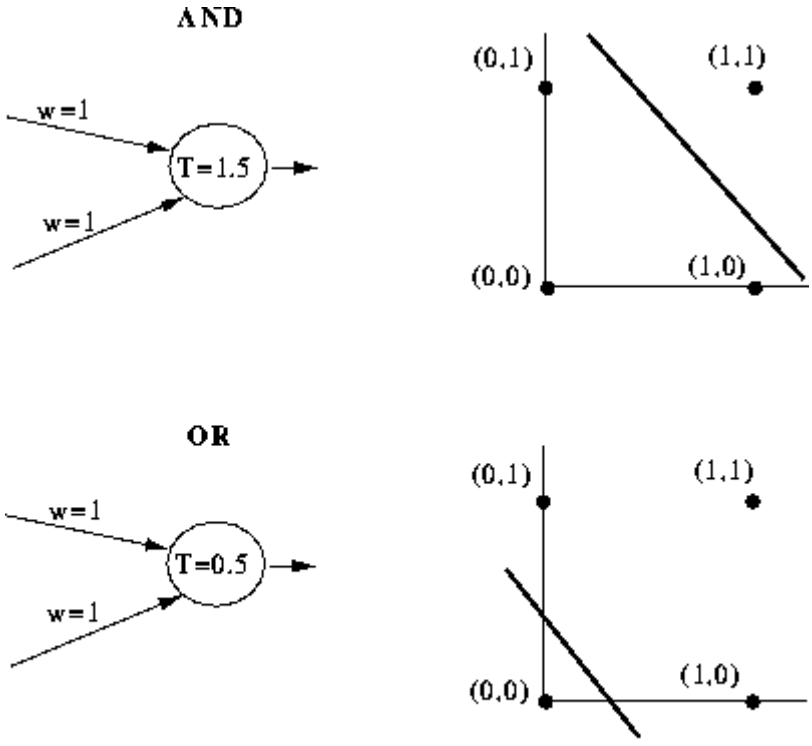
$$if \quad \sum_{i}^{n} W_i x_i + b \geq 0 \rightarrow \quad y = 1$$

$$if \quad \sum_{i}^{n} W_i x_i + b < 0 \rightarrow \quad y = 0$$

그 종류는 단층과 다층으로 나뉘는데, 위 그림은 input layer와 output layer 두 단계로만 이루어졌기 때문에 단층에 해당된다. (08-1) 파트에서는 우선 단층 perceptron에 대해 다루도록 하겠다.

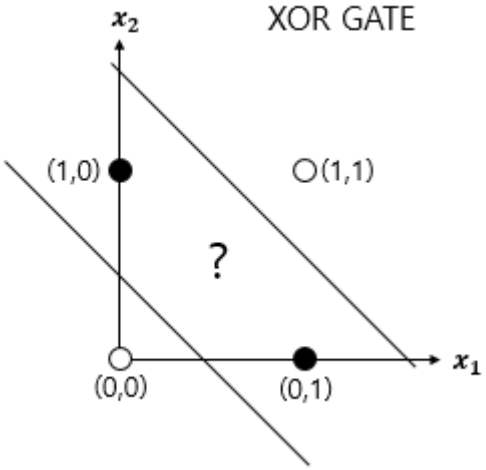
- AND, OR

Perceptron은 초기에 AND, OR문제를 해결하기 위해 만들어졌음. 아래와 같이

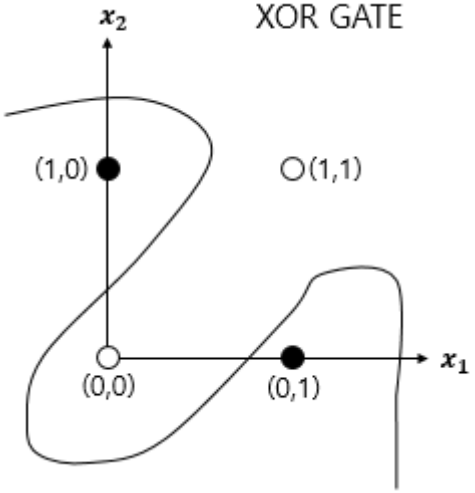


- XOR

AND, OR의 경우 단층 perceptron으로도 문제를 해결할 수 있지만 XOR는 단층의 한계로 문제 해결이 어렵다. perceptron 자체가 linear classifier를 위한 모형이기 때문이다.



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0




```
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)
```

```
# nn layers
linear = torch.nn.Linear(2, 1, bias=True)
sigmoid = torch.nn.Sigmoid()

# model
model = torch.nn.Sequential(linear, sigmoid).to(device)

# define cost/loss & optimizer
criterion = torch.nn.BCELoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1)
```

- line2에서 Linear 함수가 한번만 적용된 것으로 단층 perceptron임을 알 수 있으며, line3에서는 *activation function*으로 sigmoid를 사용했다.
- line9 - 위에서는 Softmax가 포함된 *torch.nn.CrossEntropyLoss()*를 사용했었는데, 여기서는 output이 0과 1인 Logistic reg를 이용한 classification이기 때문에 *BCELoss()*를 사용했다. (BCE = Binary Cross Entropy)

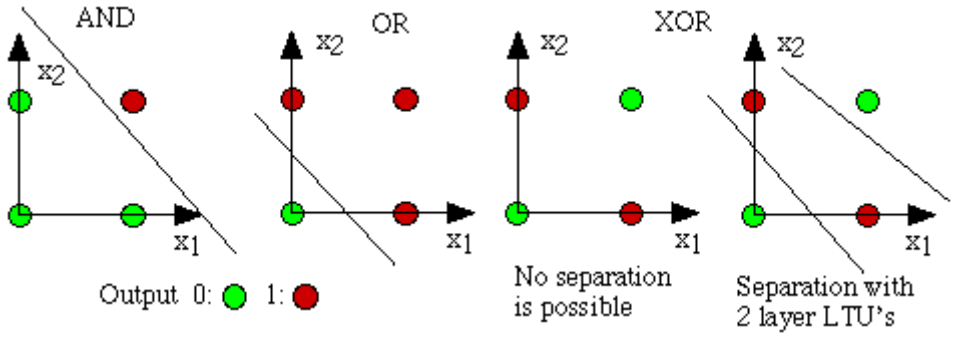
```
for step in range(10001): # 10,000번의 학습
    optimizer.zero_grad()
    hypothesis = model(X)

    # cost/loss function
    cost = criterion(hypothesis, Y)
    cost.backward()
    optimizer.step()

    if step % 100 == 0:
        print(step, cost.item())
# 결과생략
```

- 생략된 결과를 보면, 200번째부터는 loss 값의 변화가 매우 적으며 이는 학습이 제대로 이루어지지 않음을 나타낸다. 다시 말해, 단층 perceptron으로는 XOR문제를 해결할 수 없다.

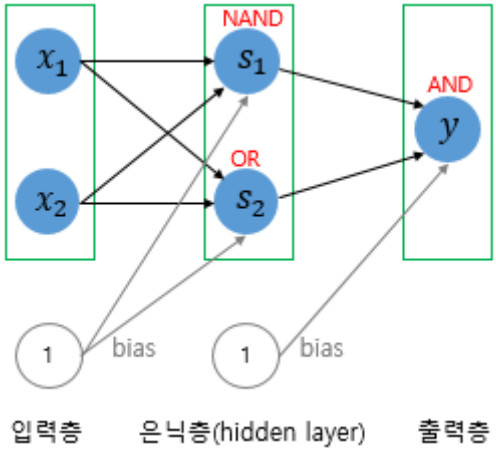
(08-2) Multi Layer Perceptron

- Multi Layer Perceptron (MLP)

XOR 문제를 해결하기 위한 여러개의 층 구조를 갖는 perceptron으로, 그림에 나와있듯 두 직선으로는 성공적으로 분류할 수 있다. 이때 여러 층을 쌓는 다는 것은 기존의 AND, OR 를 조합한다고도 볼 수 있다.



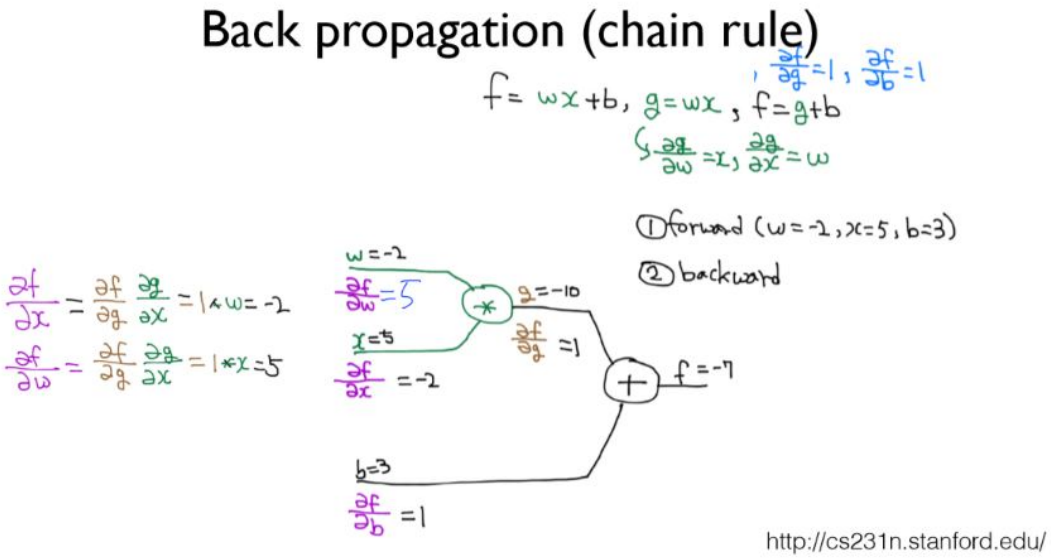
여러 층들 중에서 input layer와 output layer 사이에 존재하는 층들은 hidden layer(은닉층)이라 한다.

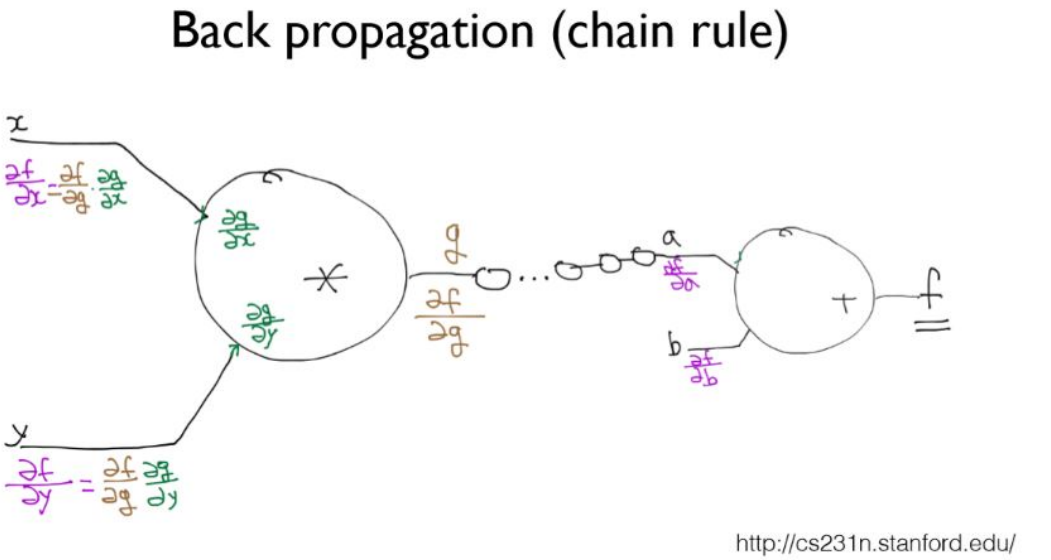


결국, MLP는 hidden layer가 1개 이상인 perceptron을 일컫는다. 2개 이상일 경우는 '심층 신경망(Deep Neural Network)'라 한다.

- Backpropagation

- 인공 신경망의 학습은 오차를 최소화하는 가중치를 찾는 목적으로 순전파와 역전파를 반복하는 것을 말한다. 인공 신경망이 순전파 과정을 진행하여 예측값과 실제값의 오차를 계산하였을 때 어떻게 역전파 과정에서 경사 하강법을 사용하여 가중치를 업데이트하는지 이해해보자.
- Gradient Descent(경사하강법)는 최적화 알고리즘으로, 오차 곡선을 따라 일정한 크기로 내려가며 global minimum을 찾는다. 이때 일정한 크기는 해당 지점에서 접선의 기울기와 learning rate를 통해 결정한다.
- 결국 최적화를 위해서는 미분값, 즉 input이 output에 미치는 영향을 계산해야 하는데, MLP의 경우 노드가 많아지고 미분값 계산이 복잡해진다. 미분값을 알아야 weight를 조정하는데, layer가 많아 계산이 힘들게 되는 것이다. 이에 우리는 미분값을 알기 위해 error를 뒤에서부터 앞으로 계산하는 방법을 사용한다.
- 이해를 돕기 위한 아주 간단한 예시를 살펴보자.





1) forward propagation: (-2*5)+3=-7=f(g(x)) , 즉 input layer부터 순서대로 계산을 진행, 출력값을 얻어낸다.

2) backward propagation: 함성함수의 미분법인 연쇄법칙을 이용해서, 각 노드에서의 미분값을 이용하여 input이 출력에 미치는 미분값을 알아낼 수 있다. 이 경우 layer가 여러개로 늘어나더라도, 간단한 term들의 계산을 통하여 알아낼 수 있다. 더 자세한 예시 및 수식은 https://wikidocs.net/60682 를 참고!

- 위 코드(8-1)에 적용할 경우 다음과 같다.

1. Forward Propagation

   우선 input layer에서 hidden layer 방향으로 진행되며, sigmoid 함수의 입력값인 $z_i=\sum^{n}_{i}W_ix_i + b$ 를 계산한다. 그 다음 hidden layer에서 sigmoid 함수를 지나고 hidden layer의 출력값은 $h_i=sigmoid(z_i)$가 된다. hidden layer의 개수만큼 이런 과정이 반복된다.

   마지막 hidden layer를 지난 결과는 인공 신경망이 최종적으로 출력하는 값이 되며, 실제값을 예측하기 위한 값으로서 예측값이라고도 한다. (위 코드들에서는 *model()*을 적용한 결과로 변수명 hypothesis로 지정됨)

   그 다음 과정은, Loss/Cost를 계산하는 것이며 위 코드들에서는 *criterion()* 부분에 해당된다.

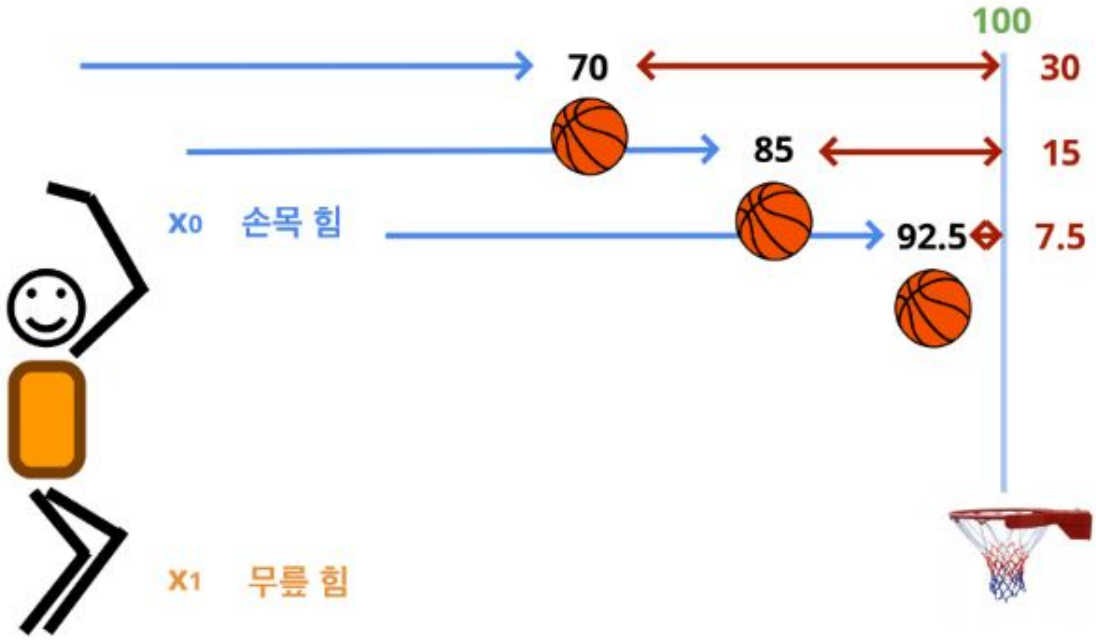
2. Back Propagation, step 1

   Forward와는 반대로 output layer에서 input layer 방향으로 진행되며, 가중치(model의 매개변수)를 업데이트 한다. 1단계에서는 output layer와 hidden layer들간의 가중치를 업데이트한다.

3. Back Propagation, step 2

   2단계에서는 hidden layer들 사이에서 가중치를 업데이트한다.

   요약하자면 가중치 업데이트를 위하여 필요한 것이 total error에 대한 가중치별 미분값이고, 이를 역전파를 통해서 알아낼 수 있는 것이다.



- 개념에 대한 쉬운 예시로, 자유투 연습은 MLP 학습과 비슷하다. 자유투를 던지는 과정을 순전파 과정이라고 하면, 던진 공이 어느 지점에 도착했는지 확인하고 던질 위치를 수정하는 과정이 역전파 과정이라고 비유할 수 있는 것이다. (이미지 출처: 손의성 교수님 인공지능 수업)

Code1 : XOR -nn

```
X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)
```

```
# nn layers
linear1 = torch.nn.Linear(2, 2, bias=True)
linear2 = torch.nn.Linear(2, 1, bias=True)
sigmoid = torch.nn.Sigmoid()
```

- 2개의 hidden layer가 존재하는 MLP

```
# model
model = torch.nn.Sequential(linear1, sigmoid, linear2, sigmoid).to(device)

# define cost/loss & optimizer
criterion = torch.nn.BCELoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1)  # modified learning rate
from 0.1 to 1
```

```
for step in range(10001):
    optimizer.zero_grad()
    hypothesis = model(X)

    # cost/loss function
    cost = criterion(hypothesis, Y)
    cost.backward()
    optimizer.step()

    if step % 100 == 0:
        print(step, cost.item())
```

결과를 보면, 단층 propagation과 달리 학습이 반복될수록 loss/cost의 값이 계속해서 감소하는 형태가 보이며 이는 MLP로는 XOR 문제를 해결할 수 있음을 의미한다.

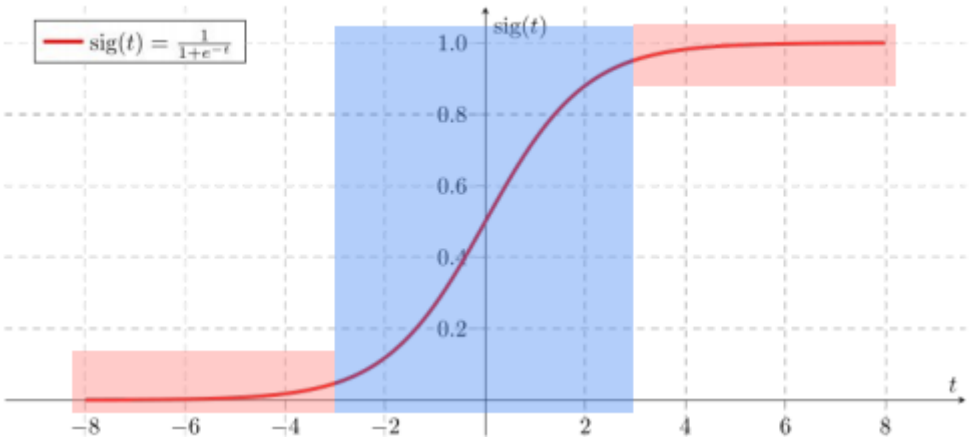
Code2 : XOR -nn-wide-deep

```
# nn layers
linear1 = torch.nn.Linear(2, 10, bias=True)
linear2 = torch.nn.Linear(10, 10, bias=True)
linear3 = torch.nn.Linear(10, 10, bias=True)
linear4 = torch.nn.Linear(10, 1, bias=True)
sigmoid = torch.nn.Sigmoid()
```

hidden layer를 4개로 늘리면, 학습이 반복될수록 loss/cost가 감소하는 폭이 더 커진다.

(09-1) ReLU

Problem of Sigmoid

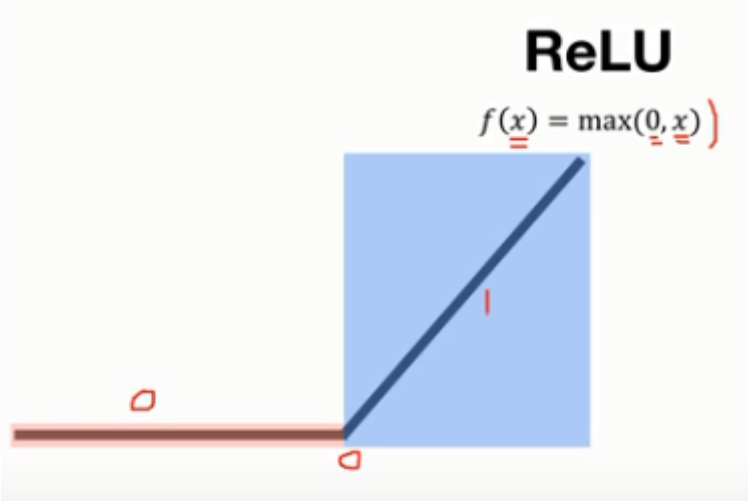


파란색 영역에서는 gradient가 잘 계산되지만, 빨간색 영역의 경우 gradient가 0에 가까운 작은 값을 가지게 된다.

Backpropagation 과정에서 loss의 gradient가 가중치를 업데이트한다. 그 과정에서 gradient가 너무 작으면 소멸될 수 있으며 결국 input layer에 가까울수록, output layer에 가까운 gradient의 영향력이 사라지게 된다. 이를 **Vanishing Gradient**라 함.

ReLU

Vanishing Gradient 문제 해결



ReLU는 빨간색 영역의 gradient는 0, 파란색 영역의 gradient는 1이 되는 activation function이다. (음수로 activation이 되는 경우에는 gradient가 아예 사라지는 문제점이 발생하긴 함)

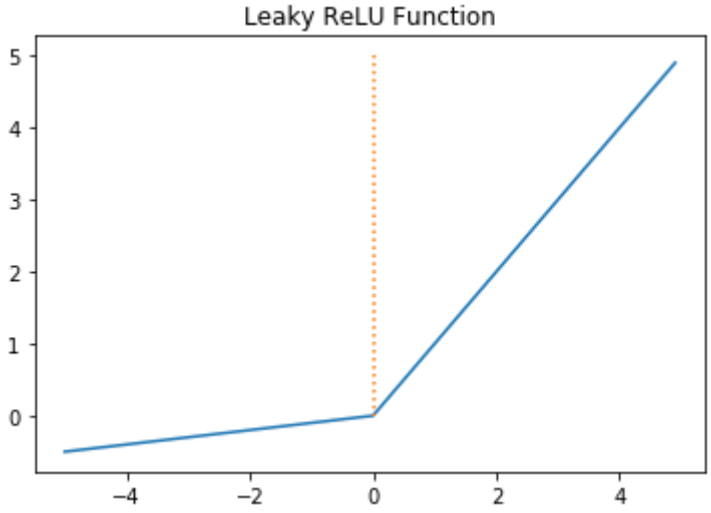
torch.nn.relu() 함수 활용

Leaky ReLU

하지만 ReLU 함수에서도 여전히 문제점이 존재하는데, 입력값이 음수면 기울기도 0이 된다. 그리고 이 뉴런은 다시 회생하는 것이 매우 어렵다. 이 문제를 죽은 렐루(dying ReLU)라고 한다.

죽은 렐루를 보완하기 위해 ReLU의 변형 함수 중 Leaky ReLU를 소개한다. Leaky ReLU는 입력값이 음수일 경우에 0이 아니라 0.001과 같은 매우 작은 수를 반환하도록 되어있다.

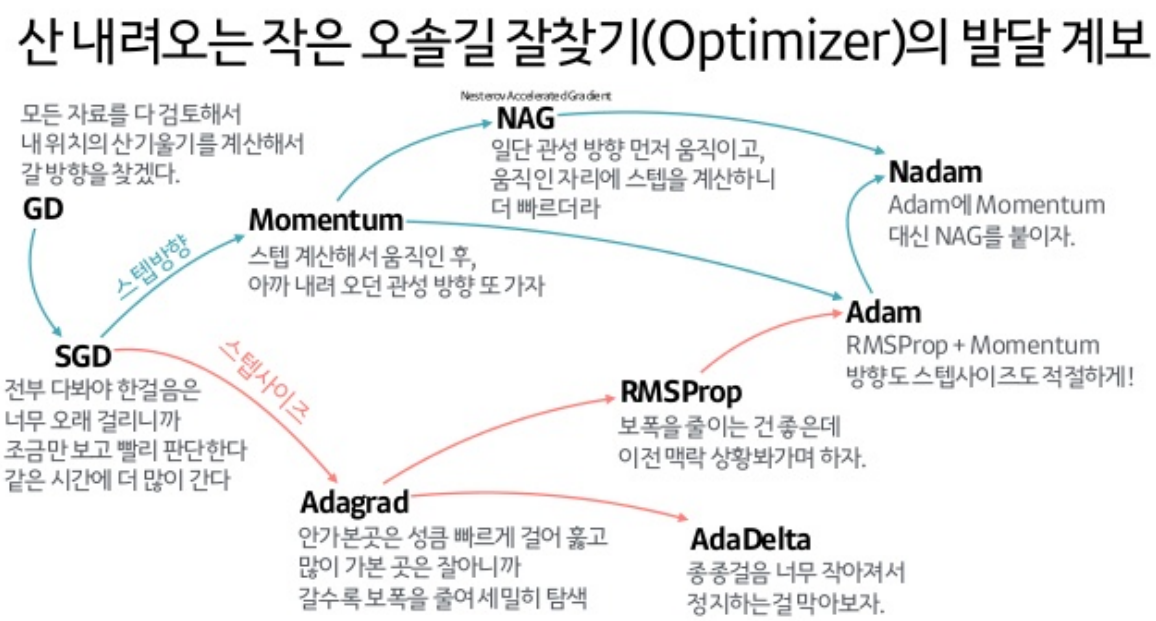
수식은 f(x)=max(ax, x)로 아주 간단합니다. a는 하이퍼파라미터로 Leaky('새는') 정도를 결정하고, 일반적으로는 0.01의 값을 가집니다. (여기서 말하는 '새는 정도'라는 것은 입력값의 음수일 때의 기울기를 비유)



(출처: https://wikidocs.net/60683)

그래프에서는 새는 모습을 확실히 보여주기 위해 a를 0.1로 잡았다. 위와 같이 입력값이 음수라도 기울기가 0이 되지 않으면 ReLU는 죽지 않는다!

Optimizer in PyTorch



Code1 : MNIST_softmax

```
optimizer = torch.optim.Adam(linear.parameters(), lr=0.001)
```

(07-2) 에서의 코드와 차이점은 SGD가 아니라 Adam optimizer를 사용했다는 점!

Code2 : MNIST_nn

이전까지 다뤘던 내용들을 코드에 모두 반영한다면?

```
# nn layers
linear1 = torch.nn.Linear(784, 256, bias=True)
linear2 = torch.nn.Linear(256, 256, bias=True)
linear3 = torch.nn.Linear(256, 10, bias=True)
relu = torch.nn.ReLU()
```

```
# Initialization
torch.nn.init.normal_(linear1.weight)
torch.nn.init.normal_(linear2.weight)
torch.nn.init.normal_(linear3.weight)
# 결과 생략
```

```
# model
model = torch.nn.Sequential(linear1, relu, linear2, relu, linear3).to(device)

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

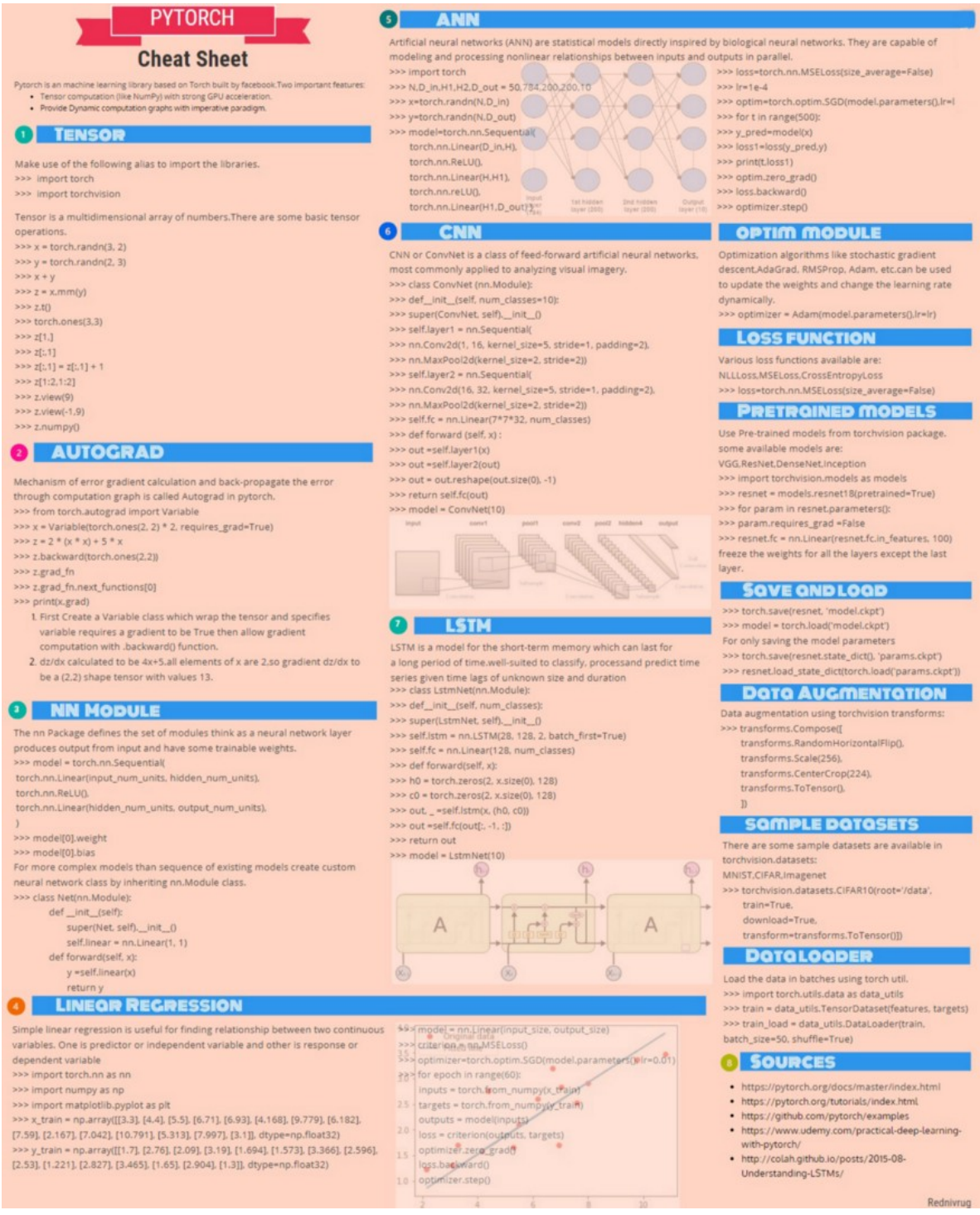
```
total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    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)

        optimizer.zero_grad()
        hypothesis = model(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 = 129.379852295
Epoch: 0002 cost = 36.195644379
Epoch: 0003 cost = 23.019363403
Epoch: 0004 cost = 15.959907532
Epoch: 0005 cost = 11.481480598
Epoch: 0006 cost = 8.563683510
Epoch: 0007 cost = 6.402721882
Epoch: 0008 cost = 4.730027199
Epoch: 0009 cost = 3.540746927
Epoch: 0010 cost = 2.647785187
Epoch: 0011 cost = 2.033876657
Epoch: 0012 cost = 1.644687057
```

```
Epoch: 0013 cost = 1.179143310
Epoch: 0014 cost = 1.109077334
Epoch: 0015 cost = 0.785691142
Learning finished
```

```
Accuracy: 0.9451000094413757
Label:  9
Prediction:  9
```

Cheat Sheet



PYTORCH

Cheat Sheet

Pytorch is an machine learning library based on Torch built by facebook.Two important features:

- Tensor computation (like NumPy) with strong GPU acceleration.
- Provide Dynamic computation graphs with imperative paradigm.

1 TENSOR

Make use of the following alias to import the libraries.

```
>>> import torch
>>> import torchvision
```

Tensor is a multidimensional array of numbers.There are some basic tensor operations.

```
>>> x = torch.randn(3, 2)
>>> y = torch.randn(2, 3)
>>> x + y
>>> z = x.mm(y)
>>> z.t()
>>> torch.ones(3,3)
>>> z[1,]
>>> z[:,1]
>>> z[:,1] = z[:,1] + 1
>>> z[1:2,1:2]
>>> z.view(9)
>>> z.view(-1,9)
>>> z.numpy()
```

2 AUTOGRAD

Mechanism of error gradient calculation and back-propagate the error through computation graph is called Autograd in pytorch.

```
>>> from torch.autograd import Variable
>>> x = Variable(torch.ones(2, 2) * 2, requires_grad=True)
>>> z = 2 * (x * x) + 5 * x
>>> z.backward(torch.ones(2,2))
>>> z.grad_fn
>>> z.grad_fn.next_functions[0]
>>> print(x.grad)
```

1. First Create a Variable class which wrap the tensor and specifies variable requires a gradient to be True then allow gradient computation with .backward() function.
2. dz/dx calculated to be 4x+5.all elements of x are 2.so gradient dz/dx to be a (2,2) shape tensor with values 13.

3 NN MODULE

The nn Package defines the set of modules think as a neural network layer produces output from input and have some trainable weights.

```
>>> model = torch.nn.Sequential(
 torch.nn.Linear(input_num_units, hidden_num_units),
 torch.nn.ReLU(),
 torch.nn.Linear(hidden_num_units, output_num_units),
 )
>>> model[0].weight
>>> model[0].bias
```

For more complex models than sequence of existing models create custom neural network class by inheriting nn.Module class.

```
>>> class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.linear = nn.Linear(1, 1)
        def forward(self, x):
            y =self.linear(x)
            return y
```

4 LINEAR REGRESSION

Simple linear regression is useful for finding relationship between two continuous variables. One is predictor or independent variable and other is response or dependent variable

```
>>> import torch.nn as nn
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168], [9.779], [6.182], [7.59], [2.167], [7.042], [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)
>>> y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573], [3.366], [2.596], [2.53], [1.221], [2.827], [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)
```

```
>>> model = nn.Linear(input_size, output_size)
>>> criterion = nn.MSELoss()
>>> optimizer=torch.optim.SGD(model.parameters(),lr=0.01)
>>> for epoch in range(60):
    inputs = torch.from_numpy(x_train)
    targets = torch.from_numpy(y_train)
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

5 ANN

Artificial neural networks (ANN) are statistical models directly inspired by biological neural networks. They are capable of modeling and processing nonlinear relationships between inputs and outputs in parallel.

```
>>> import torch
>>> N,D_in,H1,H2,D_out = 50,784,200,200,10
>>> x=torch.randn(N,D_in)
>>> y=torch.randn(N,D_out)
>>> model=torch.nn.Sequential(
    torch.nn.Linear(D_in,H),
    torch.nn.ReLU(),
    torch.nn.Linear(H,H1),
    torch.nn.reLU(),
    torch.nn.Linear(H1,D_out),)
>>> loss=torch.nn.MSELoss(size_average=False)
>>> lr=1e-4
>>> optim=torch.optim.SGD(model.parameters(),lr=l
>>> for t in range(500):
>>> y_pred=model(x)
>>> loss1=loss(y_pred,y)
>>> print(t,loss1)
>>> optim.zero_grad()
>>> loss.backward()
>>> optimizer.step()
```

6 CNN

CNN or ConvNet is a class of feed-forward artificial neural networks, most commonly applied to analyzing visual imagery.

```
>>> class ConvNet (nn.Module):
>>> def__init__(self, num_classes=10):
>>> super(ConvNet, self).__init__()
>>> self.layer1 = nn.Sequential(
>>> nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
>>> nn.MaxPool2d(kernel_size=2, stride=2))
>>> self.layer2 = nn.Sequential(
>>> nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
>>> nn.MaxPool2d(kernel_size=2, stride=2))
>>> self.fc = nn.Linear(7*7*32, num_classes)
>>> def forward (self, x) :
>>> out =self.layer1(x)
>>> out =self.layer2(out)
>>> out = out.reshape(out.size(0), -1)
>>> return self.fc(out)
>>> model = ConvNet(10)
```

7 LSTM

LSTM is a model for the short-term memory which can last for a long period of time.well-suited to classify, processand predict time series given time lags of unknown size and duration

```
>>> class LstmNet(nn.Module):
>>> def__init__(self, num_classes):
>>> super(LstmNet, self).__init__()
>>> self.lstm = nn.LSTM(28, 128, 2, batch_first=True)
>>> self.fc = nn.Linear(128, num_classes)
>>> def forward(self, x):
>>> h0 = torch.zeros(2, x.size(0), 128)
>>> c0 = torch.zeros(2, x.size(0), 128)
>>> out, _ =self.lstm(x, (h0, c0))
>>> out =self.fc(out[:, -1, :])
>>> return out
>>> model = LstmNet(10)
```

OPTIM MODULE

Optimization algorithms like stochastic gradient descent,AdaGrad, RMSProp, Adam, etc.can be used to update the weights and change the learning rate dynamically.

```
>>> optimizer = Adam(model.parameters(),lr=lr)
```

LOSS FUNCTION

Various loss functions available are:
NLLLoss,MSELoss,CrossEntropyLoss

```
>>> loss=torch.nn.MSELoss(size_average=False)
```

PRETRAINED MODELS

Use Pre-trained models from torchvision package. some available models are:
VGG,ResNet,DenseNet,Inception

```
>>> import torchvision.models as models
>>> resnet = models.resnet18(pretrained=True)
>>> for param in resnet.parameters():
>>> param.requires_grad =False
>>> resnet.fc = nn.Linear(resnet.fc.in_features, 100)
```

freeze the weights for all the layers except the last layer.

SAVE AND LOAD

```
>>> torch.save(resnet, 'model.ckpt')
>>> model = torch.load('model.ckpt')
```

For only saving the model parameters

```
>>> torch.save(resnet.state_dict(), 'params.ckpt')
>>> resnet.load_state_dict(torch.load('params.ckpt'))
```

DATA AUGMENTATION

Data augmentation using torchvision transforms:

```
>>> transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.Scale(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    ])
```

SAMPLE DATASETS

There are some sample datasets are available in torchvision.datasets:
MNIST,CIFAR,Imagenet

```
>>> torchvision.datasets.CIFAR10(root='/data',
    train=True,
    download=True,
    transform=transforms.ToTensor())
```

DATALOADER

Load the data in batches using torch util.

```
>>> import torch.utils.data as data_utils
>>> train = data_utils.TensorDataset(features, targets)
>>> train_load = data_utils.DataLoader(train,
batch_size=50, shuffle=True)
```

8 SOURCES

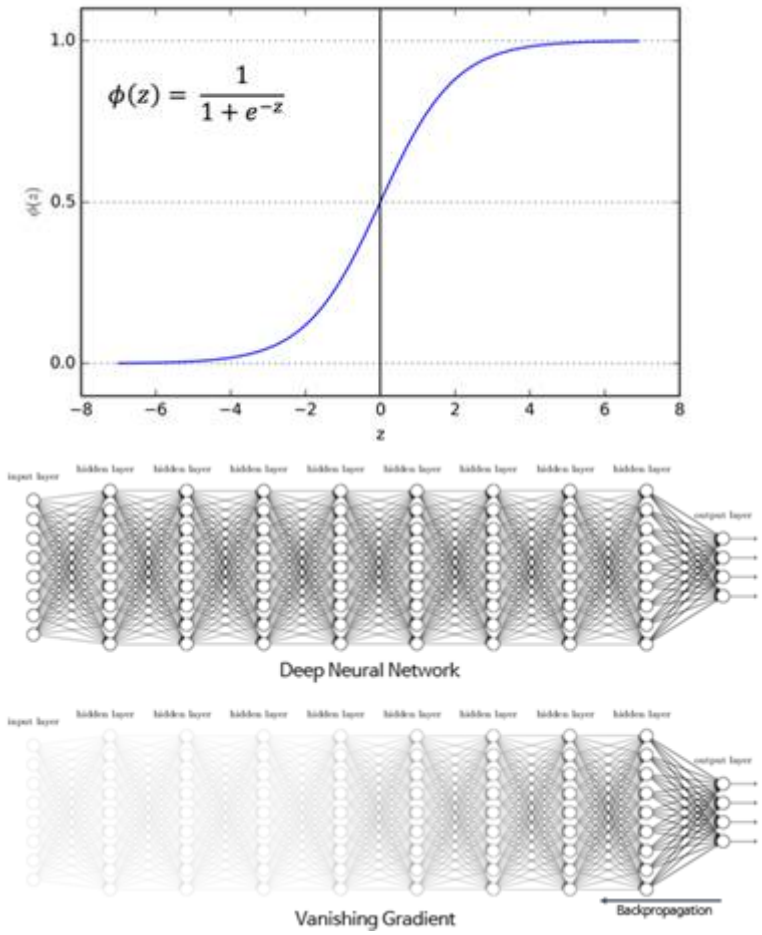
- https://pytorch.org/docs/master/index.html
- https://pytorch.org/tutorials/index.html
- https://github.com/pytorch/examples
- https://www.udemy.com/practical-deep-learning-with-pytorch/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Rednivrug

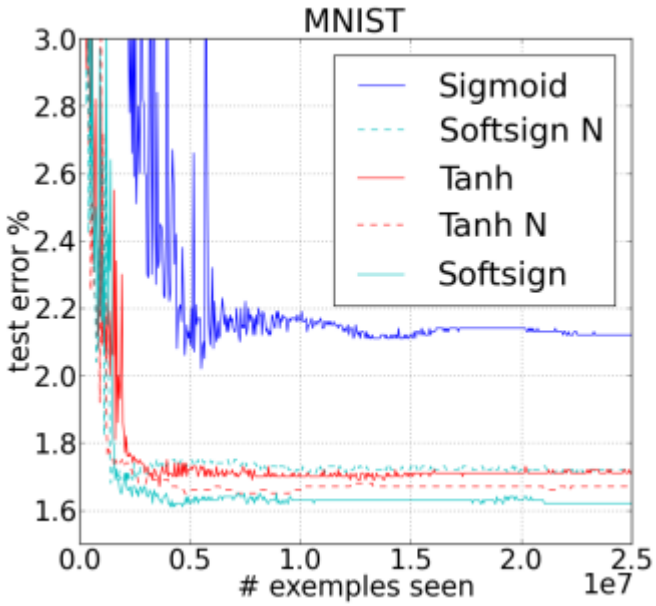
(9-2) Weight Initialization

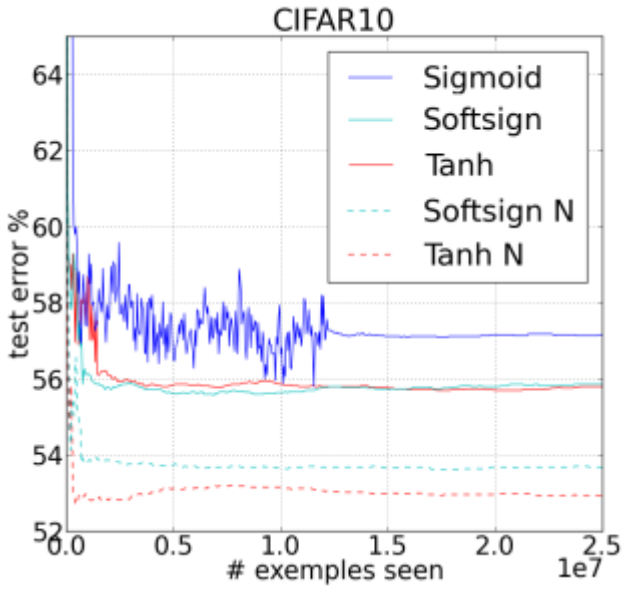
Why?

기존에는 무작위수를 사용하여 가중치 초기화를 했다. 이 경우, 활성화 함수로 시그모이드를 사용하는 신경망에 무작위로 생성한 입력 데이터를 넣는다.



시그모이드의 양 극단은 기울기가 0으로 수렴하므로 무작위수 가중치로 여러 층의 신경망을 사용할 경우 back propagation(역전파)의 과정에서 gradient vanishing(기울기 소실)의 문제가 발생할 수 있다.

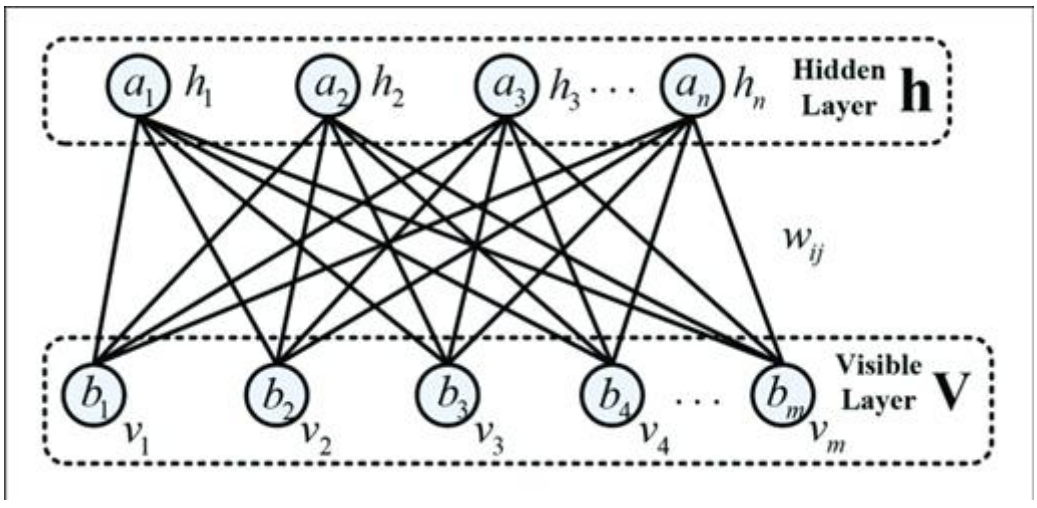




다양한 활성화 함수와 초기화 전략을 이용한 MNIST와 CIFAR10 트레이닝의 Test error를 나타낸 그래프이다. 뒤에 N이 붙은 점선들이 weight 초기화를 사용한 것이며, 더 효과적인 결과를 얻어냄을 알 수 있다. 따라서, Weight Initialization을 사용하는 것이 딥러닝의 성능을 증가시킨다.

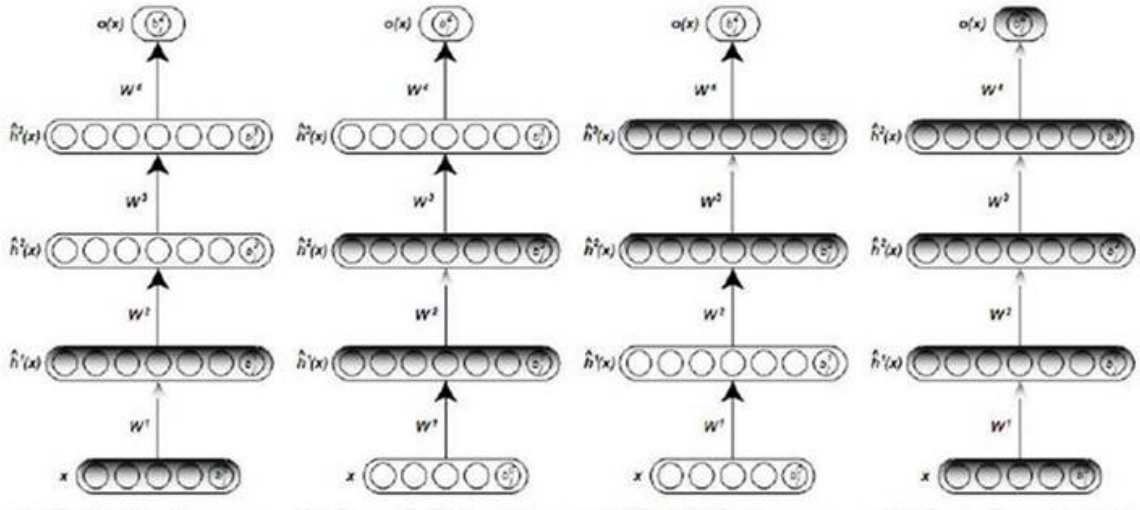
RBM/DBN

RBM



같은 layer끼리는 연결이 안 돼 있고 다음 layer와는 완전히 연결돼 있는 형태

DBN – RBM에 Pre-training 도입



(a) First hidden layer pre-training (b) Second hidden layer pre-training (c) Third hidden layer pre-training (d) Fine-tuning of whole network

앞에 두 layer를 Pre-training하고 그 다음 두 layer, 이렇게 차근차근 training을 한 후 마지막 Fine-tuning을 하는 방식

(하지만, RBM과 DBN은 새로운 initialization 방식이 소개돼 요즘 잘 사용하지 않음)

세이비어 초기화(Xavier Initialization)

이 방법은 균등 분포(Uniform Distribution) 또는 정규 분포(Normal distribution)

이전 층의 뉴런 개수와 다음 층의 뉴런 개수로 식을 세운다. 이전 층의 뉴런의 개수를 $\mathrm{n}_{in'}$ 다음 층의 뉴런의 개수를 n_{out}

$$\mathrm{W} \sim \mathrm{Uniform}\left(-\sqrt{\frac{6}{\mathrm{n}_{in}+n_{out}}},+\sqrt{\frac{6}{\mathrm{n}_{in}+n_{out}}}\right)$$

다시 말해 $\sqrt{\frac{6}{\mathrm{n}_{in}+n_{out}}}$ 를 m이라고 하였을 때, −m과 +m 사이의 균등 분포를 의미

$$\sigma = \sqrt{\frac{2}{\mathrm{n}_{in}+n_{out}}}$$

$$\mathrm{W} \sim \mathrm{N}(0,\left(\sqrt{\frac{2}{\mathrm{n}_{in}+n_{out}}}\right)^2)$$

- 세이비어 초기화는 여러 층의 기울기 분산 사이에 균형을 맞춰서 특정 층이 너무 주목을 받거나 다른 층이 뒤쳐지는 것을 막는다.
- 시그모이드 함수나 하이퍼볼릭 탄젠트 함수와 같은 S자 형태인 활성화 함수와 함께 사용할 경우 효과적
- ReLU 함수는 He 초기화와 효과적

Xavier을 사용한 code

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

```
# 앞과 동일한 MNIST 자료 불러오기 과정
```

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

```
# nn layers
linear1 = torch.nn.Linear(784, 256, bias=True)
linear2 = torch.nn.Linear(256, 256, bias=True)
linear3 = torch.nn.Linear(256, 10, bias=True)
relu = torch.nn.ReLU()
```

```
# xavier initialization
torch.nn.init.xavier_uniform_(linear1.weight)
torch.nn.init.xavier_uniform_(linear2.weight)
torch.nn.init.xavier_uniform_(linear3.weight)
```

```
# model
model = torch.nn.Sequential(linear1, relu, linear2, relu, linear3).to(device)
```

```
# define cost/loss & optimizer
criterion = torch.nn.CrossEntropyLoss().to(device)    # Softmax is 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)

        optimizer.zero_grad()
        hypothesis = model(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')
```

```
# Test the model using test sets
with torch.no_grad():
    X_test = mnist_test.test_data.view(-1, 28 * 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = model(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())

    # Get one and predict
    r = random.randint(0, len(mnist_test) - 1)
    X_single_data = mnist_test.test_data[r:r + 1].view(-1, 28 *
28).float().to(device)
    Y_single_data = mnist_test.test_labels[r:r + 1].to(device)

    print('Label: ', Y_single_data.item())
    single_prediction = model(X_single_data)
    print('Prediction: ', torch.argmax(single_prediction, 1).item())
```

```
Accuracy: 0.9804999828338623
Label:  8
Prediction:  8
```

He 초기화(He initialization)

He 초기화는 세이비어 초기화와 다르게 다음 층의 뉴런의 수를 반영하지 않는다.

$$W \sim \text{Uniform}\left(-\sqrt{\frac{2}{n_{in}}}, +\sqrt{\frac{2}{n_{in}}}\right)$$

$$\sigma = \sqrt{\frac{2}{n_{in}}}$$

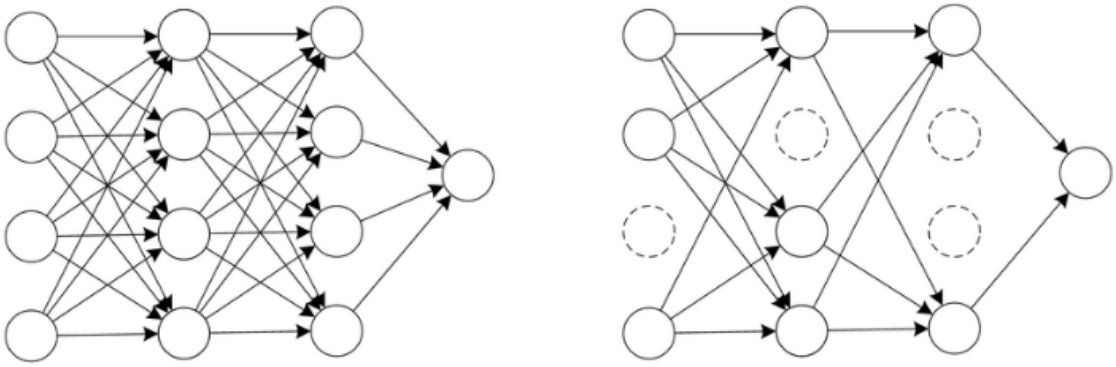
$$W \sim N(0, \left(\sqrt{\frac{2}{n_{in}}}\right)^2)$$

- ReLU 계열 함수를 사용할 경우에는 He 초기화 방법이 효율적

(9-3) Dropout

- Overfitting을 막는 방법?: 과적합은 인공신경망이 Train data에 지나치게 적응되어, 그 외의 데이터에 대한 적응력이 떨어지는 상태를 말함. 이는 error가 증가해 모델 성능 저하를 불러옴.

- Dropout이란 과적합을 억제하기 위한 방법 중 하나로, 뉴런을 임의로 삭제하며 학습하는 방법.
- 이때 특정 비율을 정하여 사용한다. 가령 드롭아웃의 비율을 0.5로 한다면 학습 과정마다 랜덤으로 절반의 뉴런을 사용하지 않고, 절반의 뉴런만을 사용한다.



(a) Standard Neural Network (b) Network after Dropout

- Dropout은 학습 시에 인공 신경망이 특정 뉴런 또는 특정 조합에 너무 의존하는 것을 방지해주고, 매번 랜덤 선택으로 뉴런들을 사용하지 않으므로 서로 다른 신경망들을 앙상블하여 사용하는 것 같은 효과를 내어 과적합을 방지한다.

```
# nn layers
linear1 = torch.nn.Linear(784, 512, bias=True)
linear2 = torch.nn.Linear(512, 512, bias=True)
linear3 = torch.nn.Linear(512, 512, bias=True)
linear4 = torch.nn.Linear(512, 512, bias=True)
linear5 = torch.nn.Linear(512, 10, bias=True)
relu = torch.nn.ReLU()
dropout = torch.nn.Dropout(p=drop_prob) #Dropout 사용 함수 #p=사용하지 않을 비율 설정
```

```
# model
model = torch.nn.Sequential(linear1, relu, dropout,
                            linear2, relu, dropout,
                            linear3, relu, dropout,
                            linear4, relu, dropout,
                            linear5).to(device)
```

- Dropout은 신경망 학습 시에만 사용하고, 예측 시에는 사용하지 않는다. 따라서 코드에서는 train과 evaluation을 각각 선언해야 함.

```
total_batch = len(data_loader)
model.train()    # set the model to train mode (dropout=True)
```

```
# Test model and check accuracy
with torch.no_grad():
model.eval()    # set the model to evaluation mode (dropout=False)
...
```

(9-4) Batch Normalization

위에서 언급한 **ReLU** 계열의 함수나, **He, Xavier**와 같은 **Weight 초기화 방법**,

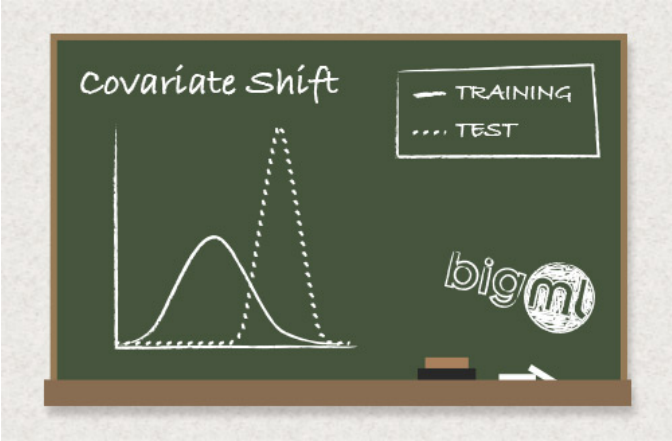
그리고 **작은 Learning rate 값**을 사용하는 것만으로도 어느 정도 **Gradient Vanishing**과 **Exploding**을 완화시킬 수 있다.

하지만, 위의 방법을 사용하더라도 Training 과정 중에 언제든 **Gradient Vanishing**과 **Exploding**이 발생할 수 있다.

Batch Normalization (배치 정규화)은 Gradient Vanishing과 Exploding을 예방하는 또 다른 방법이다.

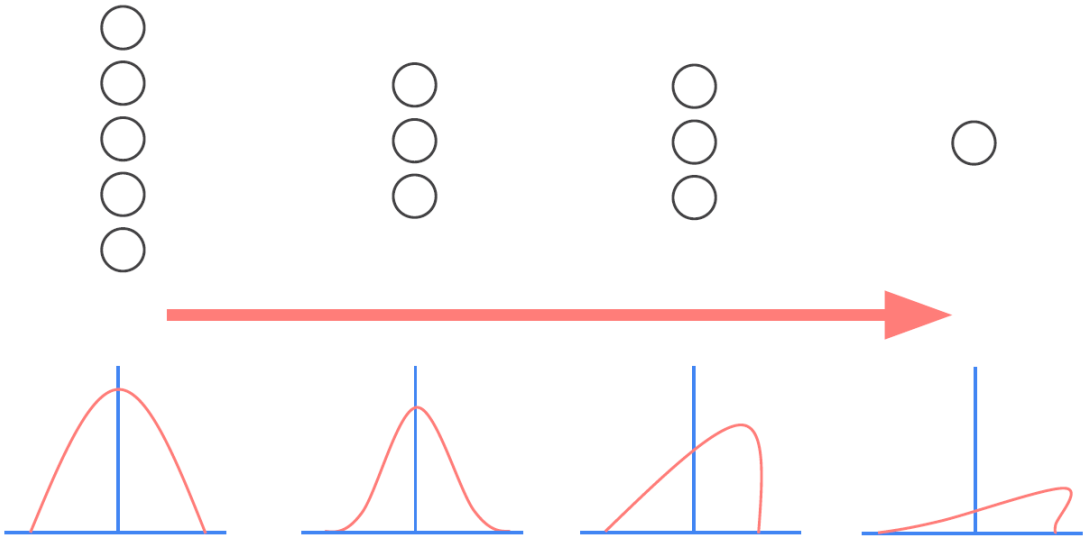
내부 공변량 변화 (Internal Covariate Shift)

- **Covariate Shift (공변량 변화)란?**



  - Train Data와 Test data의 데이터 분포가 다른 경우를 의미함.

- **Internal Covariate Shift (내부 공변량 변화)란?**



- Training 과정에서 **각 Layer를 지날 때 마다 입력 데이터의 분포가 달라지는 현상**을 말한다.

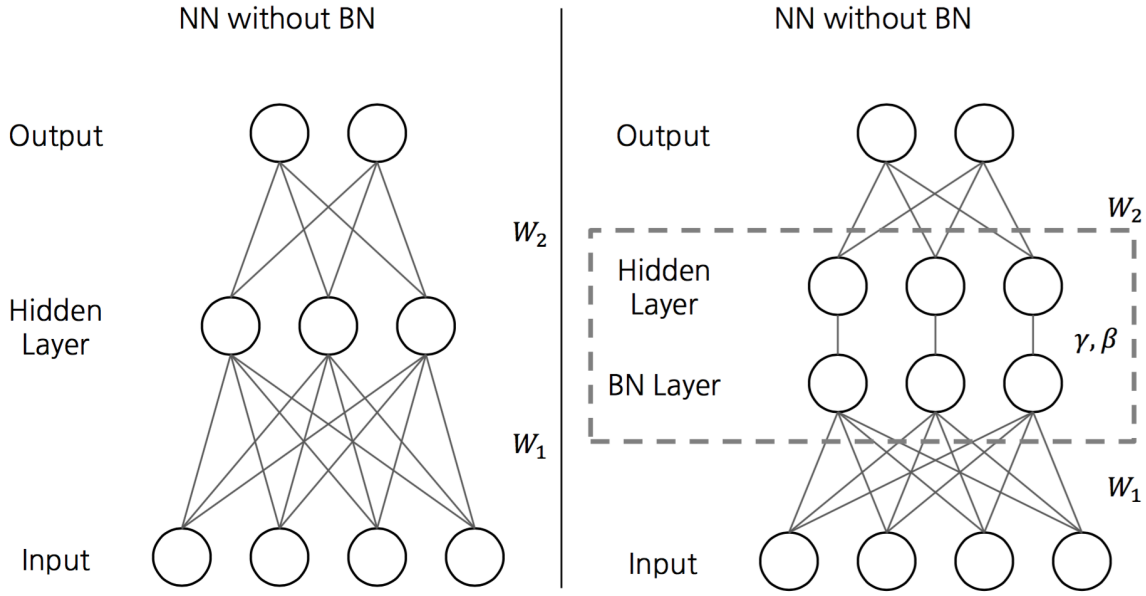
Why?

이전 Layer들의 학습에 의해 이전 Layer의 weight 값이 바뀌었기 때문에

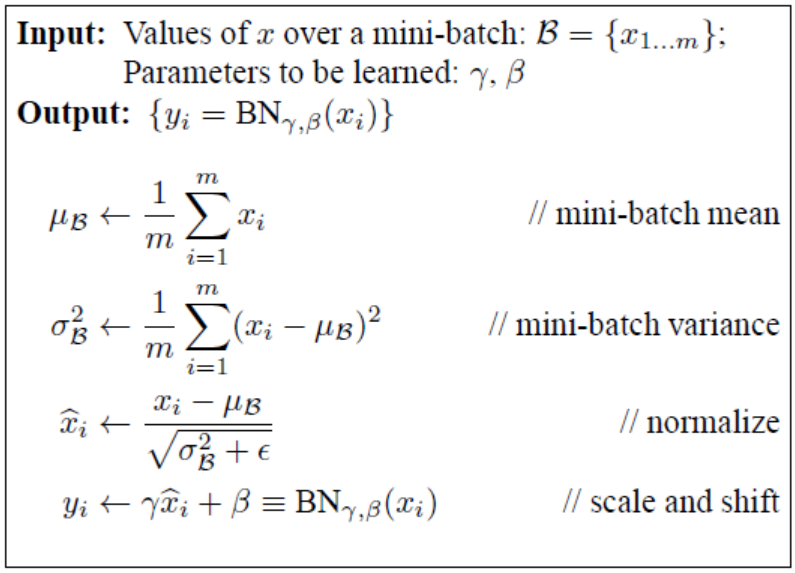
현재 Layer에 전달되는 입력 데이터의 분포 ≠ 현재 Layer가 학습했던 시점의 분포

- **Batch Normalization**

Internal Covariate Shift를 완화하기 위해 입력 데이터들을 정해진 Batch 단위로 쪼개 정규화한다.



위와 같이 입력 데이터들은 Hidden Layer의 활성화 함수를 지나기 전에 Batch Normalization Layer를 먼저 통과한다.



- Batch Normalization에 관한 공식

m = mini-batch에 있는 샘플의 수

- Batch Normalization은 각 미니 배치 B에 대한 평균과 분산을 구하고 이를 통해 입력 데이터를 Normalize 한다.
- 이후 스케일 매개변수(γ)와 시프트 매개변수(β)를 이용해 조정한 값을 최종적으로 Output으로 사용하여 Layer의 Input으로 사용한다.

Batch Normalization의 장점

- Sigmoid와 같은 함수를 사용하더라도 Gradient Vanishing 이 크게 개선된다.
- Weight Initialization에 덜 민감해진다.
- 더 큰 Learning rate를 사용할 수 있어, 학습 속도를 개선시킬 수 있다.
- Mini-batch마다 평균과 표준편차를 계산하므로 train data에 일종에 잡음을 넣게 되어 Overfitting을 방지하는 부수적 효과가 있다. 하지만 Drop out과 함께 사용하는 것이 더 좋다.

Batch Normalization의 단점 및 한계

- 모델을 복잡하게 하여 test data에 대한 예측 시에 실행 시간이 길어진다.
- Mini- Batch 크기에 의존적이어서 그 크기가 너무 작을 경우 train에 악영향을 줄 수 있다.

Batch Normalization에서 주의할 점!

Batch Normalization은 Train data와 Test data에서 적용방법이 약간 다르다.

Train Data에서는 **현재 보고 있는 mini-batch에서 평균과 표준 편차**를 구하지만,

Test Data를 이용하여 Inference를 진행할 때는 **training 과정의 K개의 Mini-batch에서 얻은 K개의 표본평균들의 평균 (Learning mean)과, K개의 표본분산들의 평균 (Learning Variance)를 사용한다.**

위와 같이 다르게 적용하는 이유 - https://kjhov195.github.io/2020-01-09-batch_normalization/

```
for epoch in range(training_epochs):
```

```
    bn_model.train() #model이 train set인 경우

    for X, Y in train_loader:
        X = X.view(-1, 28 * 28).to(device)
        Y = Y.to(device)

        bn_optimizer.zero_grad()
        bn_prediction = bn_model(X)
        bn_loss = criterion(bn_prediction, Y)
        bn_loss.backward()
        bn_optimizer.step()


    with torch.no_grad():
        bn_model.eval() #model이 test set인 경우
```

PyTorch에선 위와 같이 간단한 코드로 data의 종류(train,test)에 따라 알맞은 Batch Normalization을 수행하게 할 수 있다.

숙제 2문제

이번 주에 배운 여러 기법을 바탕으로 이전에 봤던 MNIST 데이터셋을 이용하여 딥러닝 구조를 PyTorch로 짜는 문제입니다 :)

Q1-1) 아래에 주어진 주석을 기반으로 하여 코딩을 해주세요.

- 모두의 딥러닝 시즌2 github 코드에 힌트가 있습니다 :) https://github.com/deeplearningzerotoall/PyTorch

```
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import matplotlib.pylab as plt
import random


# 파라미터 설정 (learning rate, training epochs, batch_size)
learning_rate = 0.1
training_epochs = 15
batch_size = 100


#train과 test set으로 나누어 MNIST data 불러오기
```

```
'Fill this blank!


#dataset loader에 train과 test할당하기(batch size, shuffle, drop_last 잘 설정할 것!)

'Fill this blank!


# Layer 쌓기 (조건: 3개의 Layer 사용, DropOut 사용 (p=0.3), ReLU 함수 사용, Batch normalization하기)
# 각 Layer의 Hidden node 수 : 1st Layer (784,100), 2nd Layer(100,100),3rd Layer(100,10)

'Fill this blank!'


#xavier initialization을 이용하여 각 layer의 weight 초기화

'Fill this blank!'



# torch.nn.Sequential을 이용하여 model 정의하기(쌓는 순서: linear-Batch Normalization Layer - ReLU- DropOut)

'Fill this blank!'



# Loss Function 정의하기 (CrossEntropy를 사용할 것!)

'Fill this blank!'




#optimizer 정의하기 (Adam optimizer를 사용할 것!)

'Fill this blank!'




#cost 계산을 위한 변수 설정
train_total_batch = len(train_loader)



#Training epoch (cost 값 초기 설정(0으로)과 model의 train 설정 꼭 할 것)
for epoch in range(training_epochs):

    'Fill this blank!'


#train dataset을 불러오고(X,Y 불러오기), back propagation과 optimizer를 사용하여 loss를 최적화하는 코드
```

```
    for X, Y in train_loader:

        'Fill this blank!'

        avg_cost += bn_loss / train_total_batch


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

print('Learning finished')



#test data로 모델의 정확도를 검증하는 코드 (model의 evaluation mode 설정 꼭 할 것)
#X_test 불러올 때 view를 사용하여 차원 변환할 것/ Y_test를 불러올때 labels사용
#accuracy의 초기 값 설정(0으로) 꼭 할 것

with torch.no_grad():

    'Fill this blank!'




    print("Accuracy: ", bn_acc.item())



    ##Test set에서 random으로 data를 뽑아 Label과 Prediction을 비교하는 코드
    r = random.randint(0, len(mnist_test)-1)
    X_single_data = mnist_test.test_data[r:r + 1].view(-1, 28 *28).float()
    Y_single_data = mnist_test.test_labels[r:r + 1]

    print('Label: ', Y_single_data.item())
    single_prediction = bn_model(X_single_data)
    print('Prediction: ', torch.argmax(single_prediction, 1).item())
```

Q1-2) 지금까지는 Layer의 수를 바꾸거나, Batch Normalization Layer를 추가하는 등 Layer에만 변화를 주며 모델의 성능을 향상 시켰습니다.

이번 문제에서는 위에서 만든 모델에서 있던 Layer 들의 Hidden node 수를 증가 또는 감소 (ex: 200, 300, 50...) 시켰을 때, train set에서의 cost와 test set에서 Accuracy가 기존 결과와 비교하였을 때 어떻게 달라졌는지 비교해주시면 됩니다.

- **Hidden node 수 변화 시 주의해야 할 점**
    - 각 Layer의 앞 뒤 node 숫자 일치시키기
    - Batch Normalizaion Layer의 node 숫자와 일치시키기

```
linear1 = torch.nn.Linear(784, 100, bias=True)
linear2 = torch.nn.Linear(100, 100, bias=True)
linear3 = torch.nn.Linear(100, 10, bias=True)


bn1 = torch.nn.BatchNorm1d(100)
bn2 = torch.nn.BatchNorm1d(100)
```

```
linear1 = torch.nn.Linear(784, 200, bias=True)
linear2 = torch.nn.Linear(200, 150, bias=True)
linear3 = torch.nn.Linear(150, 10, bias=True)


bn1 = torch.nn.BatchNorm1d(200)
bn2 = torch.nn.BatchNorm1d(150)
```