인공지능 기초

Lee Jun Hyeok (wnsx0000@gmail.com)

April 28, 2025

목차

Part I

ML

1. ML

1.1. ML

1.1.1. ML

1. ML
ML(Machine Learning)은 프로그램이 경험을 통해 학습하여 동작이 정의되는 프로그램 또는 알고리즘임.

일반적인 프로그램에서 사용되는 명시적 프로그래밍(Explicit Programming)에서는 단순히 개발자가 모든 경우에 대한 동작을 직접 정의하는데, 이 경우 모든 경우에 대한 동작을 전부 정의하기는 어려움. 이에 따라 머신러닝이 등장했음.

ML에서 가설(Hypothesis)은 모델이 입력값에 따른 출력값을 계산하기 위해 사용하는 수학적 함수 또는 그 출력값임.

ML에서 Epoch는 전체 데이터를 활용해 학습한 반복 횟수를 말하고(데이터셋 한 바퀴), Batch size는 한 번에 학습하는 양을, Iteration은 batch만큼 학습하는 횟수를 말함.

ML에서 feature는 모델이 예측하려는 결과를 설명하는 특성으로, tensor에서의 각 변수를 의미함.

ML에서의 학습은 데이터를 통해 모델을 개선 및 최적화하는 과정임. 학습은 지도학습, 비지도학습, 강화학습으로 분류할 수 있음.

2. 지도학습
지도학습(Supervised Learning)은 정답이 존재하는 데이터를 사용하는 학습으로, ML에서의 가장 흔한 학습 형태임. 지도학습에는 regression, binary classification, multi-label classification 등이 있음.

Regression은 입력값과 출력값에 대한 관계를 모델링하고 예측하는 문제임. regression을 수행하는 알고리즘으로는 linear regression 등이 있음.

Classification은 데이터가 사전에 분류되어 있을 때(label(정답) 존재) 어떤 데이터가 어떻게 분류되어야 하는지를 판단하는 문제임. classification을 수행하는 알고리즘으로는 logical regression 등이 있음. classification 중 class(label)가 2개인 것을 binary regression이라 하고, 여러 개인 것을 multinomial classification이라고 함.

3. 비지도학습
비지도학습(Unsupervised Leaning)은 정답이 존재하지 않는 데이터를 사용하는 학습임. 비지도학습에는 clustering, dimension reduction 등이 있음.

clustering은 데이터가 사전에 분류되어 있지 않을 때 데이터들을 분류하는 문제임.

2. Tensor

2.1. Tensor

2.1.1. Tensor

1. Tensor
Tensor(텐서)는 다차원 배열을 일반화한 개념임.

tensor의 차원은 해당 다차원 배열이 가지는 각각의 축을 의미하고, 차원(축)의 개수는 Rank라고 함.

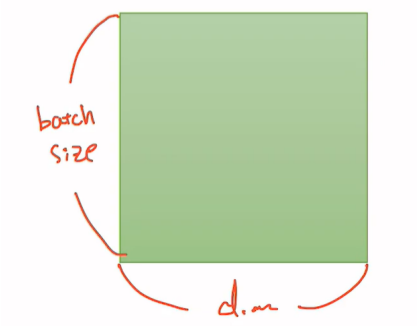
1차원 tensor는 Vector(백터), 2차원 tensor는 Matrix(행렬)라고 부름. 차원이 없고 하나의 값만을 가지는 tensor는 Scalar(스칼라)라고 부름.

ML에서는 tensor의 형태로 데이터를 처리함.

2. Tensor Shape Convention
tensor의 shape은 tensor의 각 차원에 대한 데이터의 구조와 크기를 말함. tensor t의 shape은 |t|로 표기하고, 그 값은 순서쌍으로 나타냄.

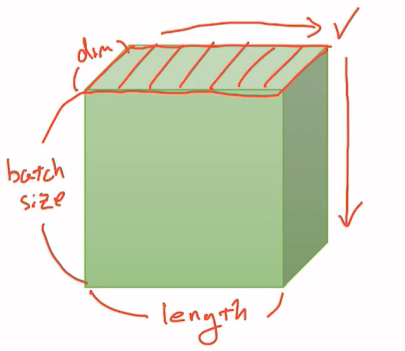
pytorch에서 tensor를 다룰 때는 대체로 차원에 따라 아래와 같은 convention으로 shape을 나타냄.

1) 2차원 tensor
2차원 tensor는 shape을 |t| = (batch size, dim)으로 표기함. 이때 batch size가 세로, dim이 가로임.



2) 3차원 tensor(NLP)
NLP에서 사용하는 3차원 tensor는 shape을 |t| = (batch size, length, dim)으로 표기함. 이때 batch size가 세로, length가 가로, dim이 깊이임.

NLP에서는 자연어를 처리하므로, batch size는 하나의 문장(또는 처리 단위)을 나타내고, length는 특정 문장에 포함되는 단어(또는 토큰)의 개수를, dim은 특정 단어를 나타냄. 즉, 위에서 봤을 때의 하나의 면이 하나의 문장을 나타냄.



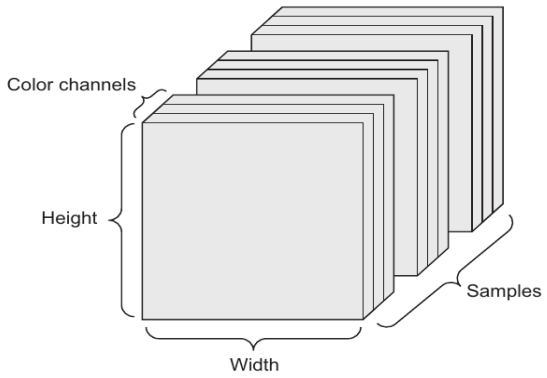
3) 4차원 tensor(computer vision)
conputer vision에서 사용하는 4차원 tensor는 channels first인 경우와 channels last인 경우에 따라 shape이 다름.

Channels First인 경우 shape을 |t| = (batch size, width, height, channels)로 표기함. 이때 width가 세로, height가 가로, channels가 깊이임. 이 경우 각 channel에 대한 면이 각각 존재하는 것으로 생각할 수 있음. 프레임워크에 따라 channels first를 사용하기도 하고 channels last를 사용하기도 하는데, pytorch는 channels first를 사용함.

Channles Last인 경우 shape을 |t| = (batch size, channels, width, height)로 표기함. 이때 channels가 세로, width가 가로, height가 깊이임.

이미지는 가로와 세로로 위치를 표현하므로, batch size는 각 이미지를 나타내고, width와 height는 해당 이미지의 가로와 세로를, channels는 데이터의 채널 수(ex. RGB이면 3)를 나타냄.

물론 흑백 이미지와 같이 channels이 하나인 경우 3차원 tensor를 사용할수도 있음.



물론 tensor를 수학적으로 더 엄밀하게 정의할 수 있지만, 여기에서는 단순히 데이터의 묶음 정도로 이해할 수 있음. 또한 차원과 rank는 선형대수학에서의 그것과는 의미가 다름.

tensor를 다룰 때는 그 shape을 잘 파악하는 것이 중요함.

2.2. Tensor Manipulation

2.2.1. Pytorch

Pytorch는 facebook에서 개발한 오픈소스 딥러닝 프레임워크임.

본 필기는 pytorch를 기반으로 함.

2.2.2. Tensor Manipulation

pytorch를 활용하여 아래와 같이 tensor를 조작할 수 있음.

1. 생성 및 정보 확인
아래와 같이 tensor를 생성하고, 차원/shape을 확인할 수 있음.

tensor는 []로 묶어서 표기함. 이때 $(a_1, a_2, \cdots, a_n)$이라면 a_1에서 a_n 순서로 가장 바깥쪽 차원부터 안쪽 차원까지의 원소의 개수임.

tensor는 하나의 자료형만을 가질 수 있음. 다른 자료형의 값을 작성하면 자동으로 형변환됨.

zeros()에 requires_grad=True를 옵션으로 지정하여 gradient 계산에 사용할 tensor를 생성할 수 있음음.

```
t = torch.Tensor(2, 3, 5) # (2, 3, 5)인 tensor 생성
t = torch.Tensor([[1, 2, 3], [4, 5, 6]]) # 값으로 tensor 생성
t = torch.FloatTensor([[1, 2, 3], [4, 5, 6]]) # float tensor 생성
t = torch.LongTensor([[1, 2, 3], [4, 5, 6]]) # tensor 생성
t = torch.BoolTensor([True, False, False]) # tensor 생성
t = torch.zeros(2, 3, 4) # 2x3x4, 0으로 채운 tensor 생성
t = torch.ones(2, 3, 4) # 2x3x4, 1로 채운 tensor 생성
t = torch.empty(2, 3, 4) # 2x3x4, 초기화되지 않은 tensor 생성
t = torch.arange(0, 10) # 0~9. 범위 값으로 tensor 생성
t = torch.eye(4) # 4x4, 단위 행렬 생성

t.dim() # rank
t.size() # shape
t.shape # shape
```

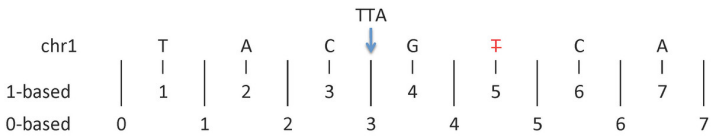
2. 추출

아래와 같이 인덱싱 및 슬라이싱하여 특정 부분을 추출할 수 있음. 이때 추출된 결과는 tensor임.

[]에 인덱스를 작성하여 원소를 추출할 수 있음. 또한 인덱스로 tensor나 :를 작성하여 슬라이싱을 할 수 있음.

논리형 tensor를 인덱스로 작성하면 대응되는 위치의 값이 참인 경우에만 추출함. tensor에 관계 연산을 수행하면 해당 tensor의 값 별 연산을 수행해 그 결과를 논리형 tensor로 반환하는데, 이를 인덱스로 작성하면 해당 조건에 맞는 원소만을 추출할 수 있음. 이를 boolean masking이라고 함.

:는 start:stop:step 꼴로 작성하는데, start는 시작 인덱스(포함), stop는 종료 인덱스(제외), step는 슬라이싱 간격임. start:stop꼴로 작성하면 step은 1로 지정됨. start/stop 값을 생략하면 시작/끝 값으로 지정됨. start/stop 값으로 음의 정수를 작성하면 뒤에서부터 선택함. 파이썬은 0-based 인덱스를 사용하므로, 자연스럽게 정수의 절댓값만큼 앞 또는 뒤로 이동하며 원소가 선택됨.



	1-based	0-based
Indicate a deletion	chr1:5-5 T/-	chr1:4-5 T/-
Indicate an insertion	chr1:3-4 -/TTA	chr1:3-3 -/TTA

```
t = torch.FloatTensor([[1, 2, 3], [4, 5, 6]]) # tensor 생성

t[0] # [1, 2, 3]
t[:, t > 4]
t[:, [0, 2]]
t[1,2]
t[:, 1:3]
t[:, ::2]
```

3. broadcasting

Broadcasting은 서로 크기가 다른 두 tensor를 연산할 때 tensor의 크기가 자동으로 조정되는 기능임.

덧셈, 뺄셈, 단순 곱셈 등을 수행할 때 작은 쪽이 큰 쪽으로 크기가 맞춰짐. 이때 작은 쪽은 기존의 값을 복사하여 연장됨. 차원이 맞지 않으면 차원이 늘어남. 특히 scalar를 활용한 간단한 연산에서 쉽게 활용할 수 있음.

```
# vector + scalar
m1 = torch.tensor([1, 2])
m2 = torch.tensor(3)
rst = m1 + m2

# vector + vector
m1 = torch.tensor([[1, 2]]) # (1, 2)
m2 = torch.tensor([[2], [3]]) # (2, 1)
rst = m1 + m2 # 둘 다 (2, 2)로 늘어나 계산됨
```

물론 broadcasting이 자동으로 수행되어 의도치 않은 결과가 나올 수 있으므로 주의하는 것이 좋음.

4. 곱셈

*tensor끼리 *로 연산하거나 mul() 메소드를 사용하면 위치가 대응되는 원소끼리의 단순 곱셈이 수행됨. matmul() 메소드를 사용하면 행렬 곱이 수행됨.*

단순 원소별 곱셈은 크기가 맞지 않는 경우 broadcasting이 수행되지만, 행렬 곱에서 크기가 맞지 않으면 런타임 에러가 남.

```
m1 = torch.tensor([[1, 2]]) # (1, 2)
m2 = torch.tensor([[2], [3]]) # (2, 1)
m1.mul(m2) # 단순 원소별 곱
m1 * m2 # 단순 원소별 곱
m1.matmul(m2) # m1 * m2 행렬 곱
```

5. 기본 연산

평균, 최대/최소, 합계 등을 계산할 수 있음. 인자에 dim을 지정하지 않으면 전체 데이터에 대해 계산함.

1) mean()

mean()으로 평균을 계산할 수 있음. 이때 mean()은 정수형 tensor에 사용하면 런타임 에러가 발생함. 실수에 대해서만 계산할 수 있음. 또한 인자로 dim을 지정하여 어떤 차원 방향으로 평균을 계산할지를 지정할 수 있음. 예를 들어, (2, 3)일 때 dim=0이면 행, dim=1이면 열, dim=-1이면(뒤에서부터 셈) 열 방향에 대해 평균을 계산함. 참고로 다른 메서드들에서도 dim을 지정할 때 동일한 방식을 사용하면 됨.

```
t = torch.tensor([[1, 2], [3, 4]])
t.mean()
t.mean(dim=1)
```

2) sum()

sum()으로 합계를 계산할 수 있음. mean()과 동일한 원리로 dim을 지정하여 계산할 수 있음.

```
t.sum()
t.sum(dim=1)
```

3) max()

max()로 최댓값을, argmax()로 최댓값의 인덱스를 얻을 수 있음. mean()과 동일한 원리로 dim을 지정하여 계산할 수 있음. max()를 사용하면 최댓값(max)과 최댓값의 인덱스(argmax)가 담긴 객체가 반환되고, argmax() 메서드를 사용하면 최댓값의 인덱스만을 반환받을 수 있음.

```
t.max(dim=1)
t.argmax()
```

6. 구조 변형

tensor의 구조를 변형할 수 있음.

1) view()
view()로 shape을 지정하여 tensor를 해당 shape으로 변형할 수 있음. 이때 -1을 지정하면 해당 부분에는 수가 자동으로 지정됨.

이때 당연하게도 지정하는 수들의 곱이 기존 수들의 곱과 동일해야 함. -1이 존재한다면 다른 수들의 곱이 기존 수들의 곱의 약수여야 함.

```
t = torch.tensor([[[1, 2, 2], [3, 4, 2]], [[3, 1, 1], [2, 3, 5]]]) # (2, 2, 3)
t.view([-1, 3]) # (4, 3)으로 반환
```

2) squeeze()
squeeze()로 원소의 개수가 1개인 차원을 없애도록 변형할 수 있음. 원소의 개수가 1개이면 해당 차원이 존재하지 않는 것과 구조가 다르지 않음. dim을 지정하면 해당 차원의 원소의 개수가 1개일 때 해당 차원만을 제거함.

```
t = torch.tensor([[[1, 2, 2], [3, 4, 2]]]) # (1, 2, 3)
t.squeeze() # (2, 3)
t.squeeze(dim=0) # (2, 3)
t.squeeze(dim=2) # (1, 2, 3)
```

3) unsqueeze()
unsqueeze()는 지정한 위치에 원소의 개수가 1인 차원을 추가함. 즉, squeeze()의 반대 작업을 수행함. 이때 인자(dim)로 차원을 반드시 지정해야 함.

```
t = torch.tensor([[1, 2, 2], [3, 4, 2]]) # (2, 3)
t.unsqueeze(1) # (2, 1, 3)
```

7. Type casting
아래와 같이 float(), long() 등의 메소드를 사용하여 tensor의 자료형을 변환할 수 있음.

```
lt = torch.LongTesor([1, 2, 3, 4])
lt = lt.float() # float으로 변환
lt = lt.long() # long으로 변환
```

8. Concatenate
여러 tensor들을 연결할(concatenate) 수 있음.

1) cat()
cat()으로 여러 tensor들을 연결할 수 있음. 인자에 연결할 tensor들을 list 또는 tuple로 묶어서 작성함. dim을 지정하여 어느 방향으로 연결할지를 지정할 수 있고, 기본값은 0임. 이때 tensor의 rank는 변화하지 않고, 지정한 차원이 가지는 원소의 개수만 변화함.

cat()을 사용하지 않고 단순히 []로 두 tensor를 묶는다면, cat()처럼 각 tensor의 값들이 사용되어 묶이는 것이 아니라 해당 tensor들로 이뤄진 tensor가 반환됨.

```
t1 = torch.tensor([[1, 2], [3, 4]])
t2 = torch.tensor([[5, 6], [7, 8]])
cat([t1, t2]) # 행 방향 결합. dim=0과 동일함.
cat((t1, t2)) # 행 방향 결합. dim=0과 동일함.
cat([t1, t2], dim=1) # 열 방향 결합.
```

2) stack()
stack()으로 여러 tensor를 쌓을 수 있음. dim을 지정하여 어느 방향으로 쌓을지를 지정할 수 있고, 기본값은 0임. stack()의 동작은 tensor를 기하학적으로 생각했을 때 말 그대로 쌓는 것으로 이해할 수 있음. 즉, 항상 새로운 차원이 생성되고, 이는 각 tensor를 unsqueeze한 뒤 cat()하는 것과 동일함.

```
torch.stack([t1, t2, t3], dim=1) # 열 방향으로 쌓음.
torch.cat([t1.unsqueeze(n), t2.unsqueeze(n), t3.unsqueeze(n)], dim=n)
```

9. Ones/Zeros
아래와 같이 특정 tensor와 동일한 shape을 가지면서 모든 값이 0이나 1인 tensor를 생성할 수 있음.

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
torch.ones_like(x) # 모든 값이 1
torch.zeros_like(x) # 모든 값이 0
```

10. In-place operation
In-place operation은 아래와 같이 메서드 식별자의 맨 뒤에 _를 붙인 것임. 해당 메서드를 사용하면 연산 직후 그 반환값을 해당 tensor의 값으로 바로 대입함.

```
t1.mul(t2) # 단순 연산
t1.mul_(t2) # 연산 후 그 반환값을 t1에 바로 대입
```

참고로 in-place operation을 사용하더라도 기존 연산에 비해 유의한 성능적 이점이 있지는 않음.

11. scatter()
아래와 같이 scatter()를 사용하여 tensor의 값을 지정한 위치로 삽입할 수 있음. 첫 번째 인자에는 어떤 방향으로 수행할 것인자, 두 번째 인자에는 값을 삽입할 위치를 나타내는 tensor를, 세 번째 인자에는 삽입할 값을 가진 tensor를 작성함. 이때 두 번째와 세 번째 tensor의 크기가 동일해야 함.

```
src = torch.tensor([[1, 2, 3], [4, 5, 6]])  # 넣을 값들
index = torch.tensor([[0, 1, 2], [2, 0, 1]])  # 값을 넣을 위치

torch.zeros(2, 3).scatter_(1, index, src) # [[1., 2., 3.], [5., 6., 4.]]
```

3. Regression&Classification

3.1. Linear Regression

3.1.1. Linear Regression

Linear Regression은 종속 변수와 하나 이상의 독립변수 사이에 존재하는 선형 관계를 찾는 방법임. simple/multivariate linear regression으로 나눌 수 있음.

1. Simple Linear Regression
Simple Linear Regression(단순선형회귀)는 입력(독립변수, x)과 출력(종속변수, y) 사이의 관계를 가장 잘 설명하는 최적의 직선 $y = Wx + b$를 찾는 regression임.

simple linear regression은 multivariate linear regression의 특수한 경우로 생각할 수 있으므로, 주요 내용은 multivariate linear regression에 정리함.

2. Multivariate Linear Regression
Multivariate Linear Regression(다중선형회귀)는 여러 개의 입력(독립변수)과 출력(종속변수)의 관계

를 가장 잘 설명하는 최적의 직선 $y = W_1x_1 + \cdots + W_nx_n + b$를 찾는 regression임.

가설은 아래와 같음. x는 행이 batch를 나타내고 열이 각 변수를 나타내는 tensor임. W(Weight)는 행이 각 변수를 나타내고 열이 출력 차원을 나타내는 tensor임. b(Bias)는 단순히 더해지는 상수 tensor임. y는 행이 각 batch를 나타내고 열이 출력 차원을 나타내는 tensor임.

$$H(x) = xW + b$$

3. 구현
pytorch에서는 아래의 과정으로 single/multivariate linear regression을 수행할 수 있음.

1) 데이터 정의
x, y 각각에 대한 tensor를 정의함.

2) 모델 정의
모델과, W와 b 각각에 대한 tensor를 정의함. 이때 W와 b는 gradient를 계산하여 최적화할 것이므로 정의 시에 requires_grad=True 옵션을 지정함.

3) Cost Function 계산
cost function을 정의하고 계산함.

Cost(Loss)는 모델이 정답과 얼마나 가까운지를 나타내는 값으로, cost가 0에 가까울수록 모델이 정답과 가까움. cost/loss를 나타내는 함수를 Cost/Loss Function이라고 함. linear regression은 cost function의 값을 최소화하는 직선(W와 b)을 찾는 과정임.

linear regression에서는 아래에서 설명할 MSE를 cost function으로 사용할 수 있음.

4) 학습
gradient descent를 반복 수행하며 cost가 최소가 되도록 W와 b를 학습(최적화)함.

```
import torch
import torch.optim as optim

x_train = torch.FloatTensor([[1, 2, 3], [3, 5, 6], [8, 9, 10]])
y_train = torch.FloatTensor([[2], [4], [6]])

W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)

optimizer = optim.SGD([W, b], lr=0.01)

iter_epoch = 100
for epoch in range(1, iter_epoch + 1) :
    hypothesis = x_train.matmul(W) + b # 행렬 곱 수행
    cost = torch.mean((hypothesis - y_train) ** 2) # MSE

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

3.1.2. MSE

linear regression에서는 MSE(Mean Squared Error, 평균제곱오차)로 cost를 계산할 수 있음. 이는 단순히 예측값($H(x_i)$)과 실제 값(y_i)의 차이를 제곱한 것에 대한 평균으로, m개의 입/출력값에 대해 아래의 수식으로 계산할 수 있음. 즉, 아래의 수식을 cost function으로 사용함.

$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x_i) - y_i)^2$$

pytorch에서는 torch.nn.functional에 있는 mse_loss()를 사용할 수 있음. 참고로, 2차원 tensor뿐만 아니라 3차원 이상의 tensor에 대해서도 적용이 가능함.

```
import torch.nn.functional as F

cost = F.mse_loss(predection, y_train) # 예측값과 실제 입력값 작성
```

3.1.3. Gradient Descent

1. Gradient Descent

Gradient Descent(GD, 경사하강법)은 함수의 기울기(변화율, gradient)를 이용해 점진적으로 최소인 가지는 지점을 찾는 방법임. 이때 편미분을 활용하므로 기울기는 아래와 같이 표기함.

$$\frac{\partial y}{\partial W} = \nabla W$$

gradient descent에서는 cost function을 편미분하여 각각에 대한 현재 위치(기존 W, b 대입)에서의 변화율을 구하고, 지정한 양의 상수 α와 곱한 값을 기존의 값에서 뺌. 따라서 특정 값에서 함수가 감소이면 양의 방향으로, 증가이면 음의 방향으로 이동하게 되므로, W와 b에 대해 반복 연산하면 최소인 지점으로 최적화됨. 이때 α는 한 번에 얼마만큼 이동할 것인지를 결정하는 양의 상수로(음수이면 최대를 구하게 됨.), Step Size 또는 Learning Rate(LR)라고 함.

$$\nabla W = \frac{2}{m}\sum_{i=1}^{m}((Wx_i + b) - y_i)x_i,\ \ W_{new} : W - \alpha\nabla W$$

$$\nabla b = \frac{2}{m}\sum_{i=1}^{m}((Wx_i + b) - y_i),\ \ b_{new} : b - \alpha\nabla b$$

gradient descent는 최소인 지점을 찾는 것이 목표이지만, 사실 학습 시작점으로부터 가까운 극소(local minimum)를 찾을 뿐, 항상 최소를 찾는 것은 아님. 물론 cost function의 개형에 따라 항상 최솟인 지점을 구하는 것이 가능할 수 있음.

편미분과 chain rule을 사용하면 이를 실제로 간단하게 계산할 수 있음. 중간 과정에 대한 수식을 작성하고(특히 각 중간 결과를 z 등으로 명시해야 편함.), chain rule을 사용해 필요한 변수들에 대해 미분하면 됨.

2. 코드

이를 코드로 나타내면 아래와 같음.

```
W_gradient = 2 * torch.mean((W * x_train + b - y_train) * x_train)
b_gradient = 2 * torch.mean(W * x_train + b - y_train)
lr = 0.1
W -= lr * W_gradient
b -= lr * b_gradient
```

pytorch는 이 과정을 수행해주는 함수들을 제공함. 아래와 같이 학습 가능한 변수들과 LR을 지정하여 optimizer를 생성한 뒤, 함수들을 호출해 gradient descent를 수행할 수 있음.

```
import torch.optim as optim
optimizer = optim.SGD([W, b], lr=0.15) # 학습할 변수와 lr 지정
...
cost = F.mse_loss(hypothesis, y_train) # cost function 정의
...
optimizer.zero_grad() # 각 변수에 대한 gradient 값을 0으로 초기화
cost.backward() # cost 객체의 정보를 활용하여 각 변수에 대한 gradient를 계산
optimizer.step() # optimizer의 알고리즘에 따라 변수 최적화(학습)
```

실세계에서의 cost function은 대체로 복잡하고 비선형인 경우가 많으므로 미분하여 최소인 지점을 찾기 어려움. 이에 따라 경사하강법에서는 반복 연산을 통해 최소인 지점을 대략적으로 찾고, epoch가 커질수록 더 정확한 값을 얻을 수 있음.

3.2. Classification

3.2.1. One-hot Encoding

One-hot Vector는 하나의 원소만 1이고, 나머지는 0인 벡터임. 각 행이 one-hot vector인 matrix를 one-hot matrix라고 함. One-hot Encoding은 데이터를 one-hot으로 변환하는 것임. 이는 주로 범주형 데이터를 나타낼 때 사용함.

pytorch에서는 아래와 같이 단순 숫자로 표현된 범주형 데이터를 one-hot 벡터로 변환할 수 있음. 특히 2차원 이상의 정수형 tensor에 대해서는 제공되는 one_hot()을 사용하는 것이 편리함. 이때 num_classes로는 범주의 개수(one hot 벡터의 차원)를 지정함.

```
# 직접 인코딩하기
y = torch.Tensor([3, 1, 2, 4, 5])
eye_matrix = torch.eye(len(y))
y_one_hot = eye_matrix[y.long() - 1]

# 함수 사용하기
import torch.nn.functional as F

y = torch.Tensor([[1, 2], [0, 1]])
y_one_hot = F.one_hot(y, num_classes=3)
```

3.2.2. Logistic Regression

1. Logistic Regression

Logistic Regression은 binary classification을 수행하기 위해 linear regression의 방식을 범주형 데이터(classification)에 적용한 알고리즘임. 출력값은 항상 2가지 범주로 분류되고, 이 두 범주는 0과 1로 구현됨.

정리하면, logistic regression은 단순히 각 batch별 출력 차원이 1인 linear regression의 결과를 sigmoid를 사용해 0과 1 사이로 변환한 것임.

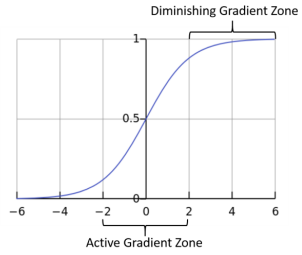
2. 가설

가설은 아래와 같음. 기존 linear regression의 선형변환의 결과를 sigmoid 함수에 넣은 것임.

$$Z(x) = xW + b, \;\; H(x) = \frac{1}{1 + e^{-Z(x)}}$$

sigmoid 함수는 아래와 같은 개형을 가짐. 즉, linear regression 가설의 값을 0과 1 사이의 값으로 변환함.

$$A = \frac{1}{1+e^{-x}}$$



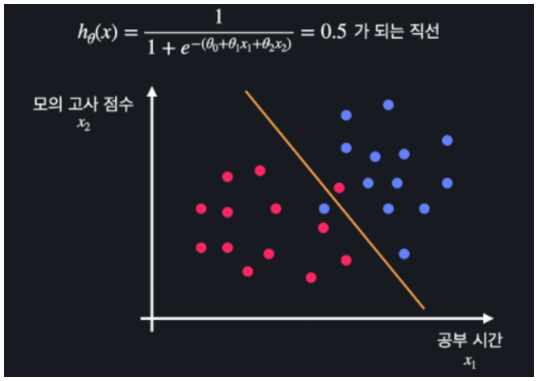
sigmoid 함수는 선형변환의 값을, 예측값이 1일 확률로 변환하는 것으로 생각할 수 있음. 당연하게도 1에서 빼서 정답이 0일 확률도 구할 수 있음.

$$H(x) \approx P(X=1) = 1 - P(X=0)$$

3. 결정 경계

결정 경계(Decision Boundary)는 classification에서 각 class의 영역을 나누는 경계임. binary classification에서는 각 class에 대한 확률이 동등하다면 H(x)=0이 되는 지점이 결정 경계임.

logistic regression의 결과는 각 데이터를 좌표평면에 나타냈을 때 결정 경계가 되는 하나의 직선을 찾은 것으로 볼 수 있음. 당연하게도 $xW + b = 0$인 지점을 기준으로 데이터가 분류됨. 이때 logistic regression은 linear regression을 기반으로 하기 때문에, 데이터가 선형적으로 분류되지 않으면 적용할 수 없음.



4. Cost Function

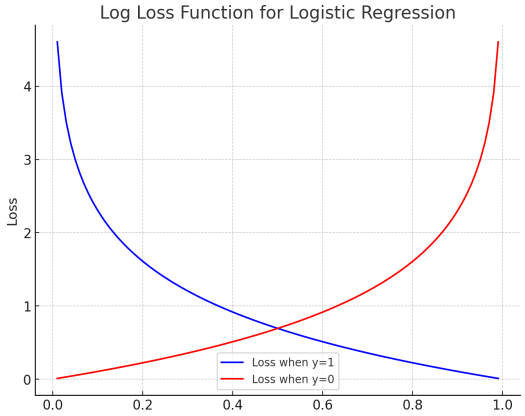
아래와 같은 수식을 cost function으로 사용함. 이를 BCE(Binary Cross Entropy)라고 함.

$$cost(W,b) = \frac{1}{m}\sum C(H(x),y)$$

$$C(H(x),y) = -y\log(H(x)) - (1-y)(\log(1-H(x)))$$

즉, 각 데이터에 대해 함수 C의 값을 구하여 평균을 냄. 이때 함수 C는 아래의 수식을 간단하게 나타낸 것인데, y=1일 때는 $-\log(x)$를 사용하여 가설의 값이 1에 가까울수록 0에 가까운 값을 반환하도록 하고, y=0일 때는 $-\log(1-x)$를 사용하여 가설의 값이 0에 가까울수록 0에 가까운 값을 반환하도록 한 것임.

$$C(H(x),y) = \begin{cases} -\log(H(x)) & if\ y=1 \\ -\log(1-H(x)) & if\ y=0 \end{cases}$$



이런 방식 대신 MSE를 cost function으로 사용하면 sigmoid에 의해 그 개형은 굴곡이 많은 형태가 된다고 함. 즉, gradient descent 수행 시에 적절하지 않은 local minimum으로 최적화될 수 있음.

이는 직접 수식을 쓸 수도 있지만, torch에서 제공하는 함수를 사용할 수 있음.

```
cost = (-1) * torch.mean(y_data * torch.log(hypothesis) +
                         (1 - y_train) * torch.log(1 - hypothesis))

import torch.nn.functional as F
cost = F.binary_cross_entropy(hypothesis, y_train)
```

5. 작업 과정

아래의 과정으로 logistic regression을 수행할 수 있음.

1) 데이터 정의

x와 y 각각에 대한 tensor를 정의함. y가 0과 1만을 값으로 가진다는 것을 제외하면 linear regression과 동일함.

2) 모델 정의

모델과, W와 b 각각에 대한 tensor를 정의함. 이때 W와 b를 직접 정의할 수도 있지만, nn.Module을 사용하여 모델을 생성하는 것이 간편함. sigmoid 함수 또한 torch.exp() 등을 사용하여 직접 작성하는 것보다 torch.sigmoid() 또는 nn.Sigmoid class를 사용하는 것이 편리함.

3) Cost Function 계산

cost function을 정의하고 계산함.

4) 학습

linear regression과 동일하게 gradient descent를 반복 수행하여 cost가 최소가 되도록 학습함. optimizer가 알아서 계산해주므로 여기에서 실질적인 미분은 다루지 않음.

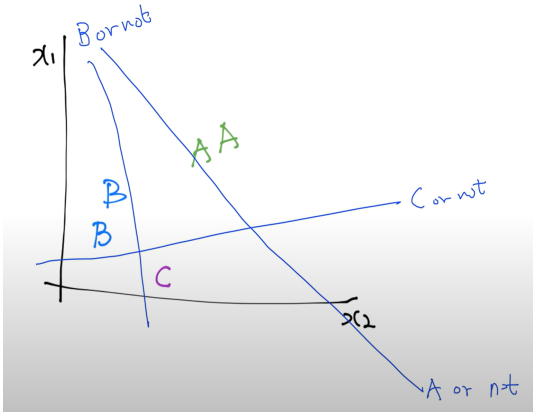
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class MyLogisticModel(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()
    def forward(self, x) :
        return self.sigmoid(self.linear(x))

x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]] # (6, 2)
x_train = torch.FloatTensor(x_data)
y_data = [[0], [0], [0], [1], [1], [1]] # (6, 1)
y_train = torch.FloatTensor(y_data)

model = MyLogisticModel()

optimizer = optim.SGD(model.parameters(), lr=1)

nb_epochs = 1000
for epoch in range(1, nb_epochs + 1) :
    # hypothesis
    hypothesis = model(x_train)

    # cost
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # learning
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
```

3.2.3. Softmax Classification

1. Softmax Classification

Softmax Classification 또는 Multinomial Logistic Regression는 logistic regression을 확장하여 multinomial classification을 수행하는 알고리즘임. 즉, 출력값은 여러 개의 범주로 분류됨.

softmax classification의 기본 원리는 여러 번의 binary classification을 수행하는 것임. 여러 개의 범주가 존재할 때, 각 범주 별로 해당 범주에 속하는지 속하지 않는지에 대해 binary classification을 수행함. 즉, 각 범주에 대한 결정 경계를 나타내는 직선을 구하는 것임. 이에 따라 W는 각 열이 범주에 대한 가중치를 나타내는 matrix가 됨.



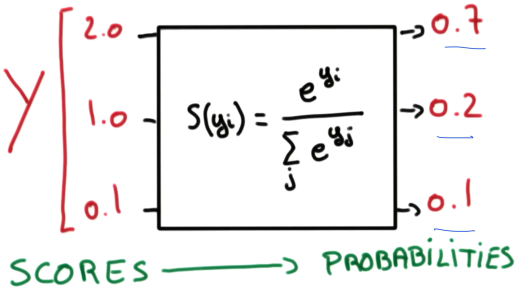
정리하면, softmax classification은 단순히 각 batch별 출력 차원이 여러 개인 linear regression의 결과를 softmax를 사용해 합이 1이고, 0과 1 사이가 되도록 변환한 것임.

2. Softmax 함수

softmax classification에서는 sigmoid 함수 대신 softmax 함수를 사용하여 선형변환에 의해 도출된 값을 확률화함. 물론 sigmoid 함수를 사용해도 0과 1 사이의 수를 얻을 수는 있지만 각 범주에 대한 확률을 얻을 수는 없음.

softmax 함수는 아래와 같은 수식을 가짐. 이는 각 y값(선형변환의 출력값)에 대해 전체의 합이 1, 각각의 값이 0과 1 사이인 값을 도출함. 즉, 각 y값에 해당하는 범주에 속할 확률을 구할 수 있음.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$



코드 상에서 이를 직접 작성할 수도 있지만, pytorch에서 제공하는 F.softmax()로 간단히 구할 수 있음.

```
import torch.nn.functional as F

z = torch.FloatTensor([1, 2, 3])
hypothesis = F.softmax(z, dim=1)
```

3. 가설

입력값을 각 결정 경계에 대한 선형변환에 넣고, softmax 함수를 사용하여 각 범주에 대한 확률을 얻음. 즉, softmax classification은 각 범주에 대한 이산확률분포를 구하는 과정이고, 가설은 아래와 같음.

$$Z(x) = xW + b, \;\; H(x) = S(Z(x))$$

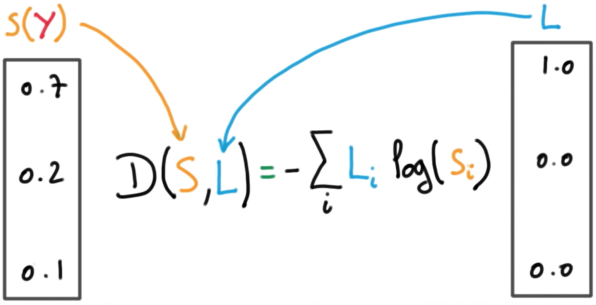
4. Cost Function

cross-entropy를 cost function으로 사용함.

Cross-Entropy는 두 이산확률분포 사이의 유사성을 계산하여 cost를 반환하는 방식으로, 아래와 같은 수식을 가짐. 여기에서 y는 one-hot 벡터로 구성된 정답임. $\hat{y}$는 예측값으로, 각 값이 범주에 대한 확률

값을 가지는 벡터임. N은 범주의 개수임. 아래의 수식을 통해 각 범주 별 연산을 수행함.

$$L = \sum -y \log(\hat{y})$$



*cost function*은 예측값이 정답에 가까울수록 작은 값을, 멀수록 큰 값을 반환해야 함. 위의 수식에서는 정답 벡터에서 0인 부분은 전부 무시하고, 1인 부분에 대해서 값을 확인함. 이때 $-\log(x)$는 입력이 0에 가까우면 ∞을, 1에 가까우면 0을 내보내므로, 예측값의 확률에 따라 적절한 cost가 반환됨.

실제로는 여러 batch를 사용하므로 y와 $\hat{y}$가 matrix임. 이 경우 각 행 별로 cross-entropy를 계산한 뒤 평균을 내면 cost를 얻을 수 있음. 이는 직접 수식을 쓸 수도 있지만, 아래와 같이 torch에서 제공하는 함수를 사용할 수 있음. log_softmax()는 softmax()에 log를 씌우는 함수이고, nll_loss()는 softmax()의 결과와 y를 인자로 작성하면 cost를 계산해 주는 함수이고, cross_entropy()는 선형변환의 결과와 y를 인자로 작성하면 cost를 계산해 주는 함수임.

아래의 코드에서 z는 선형변환의 출력값(matrix), y는 예측 범주를 저장하는 행벡터(0부터 시작), hypothesis는 $\hat{y}$, y_one_hot은 y임. cost를 직접 계산하는 경우 y를 one-hot matrix로 변환해야 하고, null_loss()나 cross_entropy()를 사용하는 경우 단순히 y를 그대로 사용함.

```
import torch.nn.functional as F

y = torch.tensor([3, 0, 2]) # 여기서 y는 단순히 범주 값을 저장함.
hypothesis = F.softmax(y, dim=1)

# y를 one-hot matrix로 변환
y_one_hot = torch.zeros_like(hypothesis)
y_one_hot.scatter_(1, y.unsqueeze(1), 1)

# 상황에 따라 알맞은 방법으로 cost를 계산하자
cost1 = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()
cost2 = (y_one_hot * -F.log_softmax(z, dim=1)).sum(dim=1).mean()
cost3 = F.nll_loss(F.log_softmax(z, dim=1), y)
cost4 = F.cross_entropy(z, y)
```

상황에 따라 다를 수 있지만, 대부분의 경우 model이 마지막에 linear layer를 거치도록 하고, cross_entropy()를 사용해 cost를 계산함. 이때 cross_entropy()에는 첫 번째 인자로 (batch_size, num_classes, ...) 형태의 예측 데이터를, 두 번째 인자로 (num_classes) 형태의 정수형 정답 데이터를 넣어야 함. tensor가 2차원이거나 3차원 이미지 데이터(channels가 num_classes임.)인 경우는 자연스럽게 사용할 수 있지만, 3차원 이상의 데이터의 경우 tensor의 구조를 변형해야 할 수 있음. 예를 들어, 출력이 one hot encoding된 3차원 문자열 데이터이고, 정답이 label을 값으로 가지는 2차원 데이터인 경우 아래와 같이 가장 안쪽 데이터와 데이터의 순서를 유지한 채 구조를 바꿀 수 있음.

```
output = output.view(-1, num_classes)
y = y.view(-1)

F.cross_entropy(output, y)
```

참고로 cross-entropy의 개형은 이차함수 형태임.즉, gradient descent로 항상 최솟값을 구할 수 있음.

logistic regression에서 사용한 cost function은 cross-entropy를 범주가 2개인 상황에 적용한 것임.

5. 작업 과정
아래의 과정으로 softmax classification을 수행할 수 있음.

1) 데이터 정의
입력(x)과 출력(y) 각각에 대한 tensor를 정의함. 입력은 여러 개의 변수를 나타낼 수 있어야 하므로 열은 변수를, 행은 case를 나타내는 행렬로 정의함. 출력은 예측 범주를 저장하는 행벡터로 정의함. 출력은 필요 시 one-hot으로 변환함.

2) 모델 정의
모델과, W와 b 각각에 대한 tensor를 정의함. 이때 W와 b를 직접 정의할 수도 있지만, nn.Module을 사용하여 모델을 생성하는 것이 간편함. 상황에 따라 다르게 정의할 수도 있지만, 뒤의 함수에서 선형변환의 출력값이 필요하므로 단순히 이를 계산하도록 함.

3) Cost Function 계산
cost function을 정의하고 계산함. 상황에 따라 다를 수 있지만, 여기에서는 cross_entropy()를 사용함.

4) 학습
linear regression과 동일하게 gradient descent를 반복 수행하여 cost가 최소가 되도록 학습함. optimizer가 알아서 계산해주므로 여기에서 실질적인 미분은 다루지 않음.

```
import torch
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim

class MySoftmaxModel(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear = nn.Linear(4, 3)
    def forward(self, x) :
        return self.linear(x)

# 3개의 범주로 classification함.

x_data = [[1, 2, 1, 1],
          [2, 1, 3, 2],
          [3, 1, 3, 4],
          [4, 1, 5, 5],
          [1, 7, 5, 5],
          [1, 2, 5, 6],
          [1, 6, 6, 6],
          [1, 7, 7, 7]]
x_train = torch.FloatTensor(x_data) # (8, 4)
y_data = [2, 2, 2, 1, 1, 1, 0, 0]
y_train = torch.LongTensor(y_data) # (8,)

model = MySoftmaxModel()
optimizer = optim.SGD(model.parameters(), lr=0.1)

nb_epochs = 1000

for epoch in range(1, nb_epochs + 1) :
    # hypothesis
    hypothesis = model(x_train)

    # cost
    cost = F.cross_entropy(hypothesis, y_train)

    # learning
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
```

모델의 예측값 자체를 구하려면 선형변환의 출력값애 argmax()를 수행하면 됨. 이를 정답과 비교하여 정확도를 구할 수 있음.

3.2.4. Cross-entropy에 대한 구체적인 정의

1. Cross-entropy에 대한 구체적인 정의
cross-entropy에 대한 좀 더 구체적인 정의를 살펴보자.

어떤 사건에 대한 확률이 P(x)일 때, 해당 사건의 Information은 $-\log P(x)$*로 정의됨. Entropy는 여러 사건에 대한 information의 평균량을 나타내고,* $-\sum_x P(x)\log P(x) = \sum_x P(x)(-\log P(x))$*로 정의됨. 즉, information에 대한 가중합임.*

Cross Entropy는 두 확률분포 사이의 차이를 나타내는 값으로, 아래와 같이 두 확률분포를 사용하여 entropy를 계산한 것임.

$$H(P,Q) = \sum_x P(x)(-\log Q(x))$$

추가로, Negative Log Likelihood는 $-\log Q(x)$로 정의됨. classification에서 어떤 y와 softmax 등에 의한 확률 값 $\hat{y}$에 대해 cross entropy를 계산하는 경우 정답에 해당하는 부분만 1이고 나머지는 0이므로, 그 결과가 정답으로 예측한 값에 대한 negative log likelihood와 동일함. 아래는 그 예시임.

Example:

Let the target distribution be one-hot encoded:

$$P(x) = [0, 0, 1]$$

This means the true class is the third one.

Let the model's predicted distribution be:

$$Q(x) = [0.1, 0.1, 0.8]$$

The cross-entropy becomes:

$$H(P,Q) = -(0 \cdot \log(0.1) + 0 \cdot \log(0.1) + 1 \cdot \log(0.8)) = -\log(0.8)$$

This is numerically identical to the NLL for the true class:

$$\text{NLL} = -\log(Q(x = \text{true class})) = -\log(0.8)$$

2. Pytorch에서의 구현

pytorch에서는 cross-entropy를 softmax에 negative log likelihood를 적용한 것으로 정의함. 즉, 실제 label에 해당하는 확률의 negative log likelihood 값을 loss 값으로 함. 물론 실제로 정의가 다르기는 하지만, 여기에서는 그 결과가 같으므로 편의를 위해 이렇게 설계했다고 함.

3.3. Linear Layer

3.3.1. Linear Layer

1. Linear Layer

Linear Layer 또는 FC(Fully Connected Layer)는 뒤에서 다룰 NN을 구성하는 주요 layer로, linear regression의 $y = xW + b$ 연산을 수행함.

이때 linear layer의 연산은 x의 각 행벡터를 W를 사용해 $\hat{y}$의 행벡터로 변환하는 것으로 이해할 수 있음. 즉, 단순히 W에 따라서 벡터의 값과 크기가 예측값으로 변환됨.

linear layer를 사용하면 regression을 수행할 수 있고, 그 결과를 sigmoid나 softmax에 넣어 classification을 수행할 수 있음.

2. nn.Linear

pytorch에서는 nn.Linear로 linear layer를 생성할 수 있음.

nn.Linear는 linear layer를 생성하는 class로, torch.nn에 포함되어 있음. 즉, 가설(W와 b)을 구성하고 가설에 따른 선형 연산을 수행함.

nn.Linear에서 수행하는 연산은 아래와 같음. 아래와 같은 연산을 수행함. 참고로 이때 W는 실제로 연산에 사용되는 weight를 단순히 transpose한 값으로, 앞에서 설명한 linear regression의 수식과 동일함.

$$y = xW^T + b$$

객체 생성 시에 첫 번째 인자에는 입력(x) 데이터의 차원을, 두 번째 인자에는 출력(y) 데이터의 차원을 작성함. 또한 필요하다면 bias 옵션을 False로 지정해 bias를 제외할 수 있음(True가 기본값). 생성된 Linear 객체에는 W와 b 등이 포함되어 있어서, 생성한 객체를 식별자로 하고 인자로 입력(x) 데이터를 작성하면 수식에 따른 예측 출력값이 반환됨.

linear layer가 가진 W의 각 열과 곱해지는 것은, 입력 tensor에서 가장 안쪽 차원의 행임. 입력 tensor

가 2차원인 경우 단순히 행렬 곱이 수행되고, 3차원 이상인 경우에도 가장 안쪽 행렬과의 행렬 곱이 수행되는 것으로 이해할 수 있음. 다시 말해, 가장 안쪽 차원(행)의 값이 linear layer에 지정한 출력 크기의 값으로 대치되는 것으로 이해할 수 있음.

그 코드는 아래와 같음.

```
    import torch.nn as nn

    m = nn.Linear(4, 2)
    m(x) # y=xW^T + b 반환
```

Part II

DL

1. DL

1.1. DL

1.1.1. NN

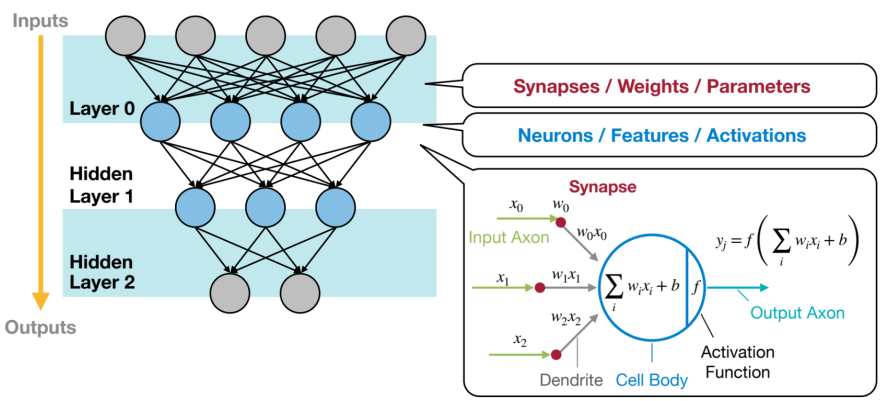
1. 뉴런

뉴런(Neuron)은 다수의 입력을 통해 하나의 출력을 내보내는 NN의 기본 구성 요소임. 이는 인간의 신경망(뇌)이 뉴런으로 구성되어 있다는 점에서 착안한 알고리즘임. 뉴런은 노드, feature, activation 등으로도 부름.

뉴런은 입력값들에 대응되는 가중치(W)를 곱하고 모두 더한 결과에 b가 더한 뒤, 활성 함수에 해당 값을 넣어 그 결과를 출력하는 식으로 동작함.

입력에 곱해지는 W(Weight)를 각 layer 사이 뉴런들의 연결로 이해할 수 있고, 이는 Synapse, Parameter라고도 부름. 이는 연결 전체를 의미하거나 각각의 개별 연결을 의미함.

또는 parameter를 NN에서 최적화가 수행되는 대상으로(W와 b) 이해할 수 있음. 이와 구분하여 learning rate, minibatch size 등 DL을 위해 지정하는 값들을 hyper parameter라고 함.



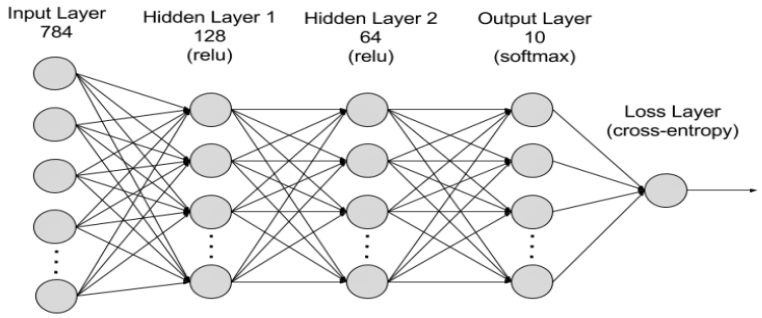
2. NN

NN(Neural Network) 또는 ANN(Artifical Neural Netowrk, 인공신경망)은 뉴런을 여러 층(Layer)으로 쌓아 인간의 신경망 구조를 구현한 ML 모델임.

NN에는 은닉층이 존재하지 않는 NN인 SLP(Single Layer Perceptron)와, 은닉층이 존재하는 MLP(Multi-Layer Perceptron)가 있음. NN 중 많은 뉴런과 층을 사용하는 것을 특히 DNN(Deep Neural Network)이라고 함.

NN의 층은 입력층, 출력층, 은닉층으로 구분됨. 입력층(Input Layer)은 데이터를 입력받는 층으로 단순히 데이터를 NN에 전달함. 출력층(Output Layer)은 결과를 출력하는 층으로 regression이나 classification 등을 통해 결과를 도출함. 은닉층(Hidden Layer)은 입력층과 출력층 사이에 존재하여 W와 b, 활성 함수에 의한 실질적인 학습이 수행되는 층임. 은닉층이 존재하지 않으면 비선형적인 관계에 대한 학습이 어려움.

NN은 각 층 사이의 연산에 대한 W, b와 활성 함수를 정의함으로써 구현됨. 예를 들어, 아래의 NN에서 입력층과 은닉층1 사이에는 구조가 (784,128)인 W와 (128,)인 b, ReLU가 활성 함수로 사용됨. 당연하게도 다음 계층의 뉴런을 k개로 하려면 W, b, 활성 함수의 결과가 k가지로 도출되어야 함.



학습 시에 NN의 각 W와 b를 최적화하는 데에는 backpropagation을 사용함.

3. Backpropagation

Progagation(전파)는 NN에서 데이터가 각 층을 통해 이동하는 것을 말함. propagation에는 forward propagation과 backpropagation이 있음.

Forward Propagation(순전파)는 입력층에서 출력층 방향의 propagation으로, 출력값을 계산하는 propagation임. Backpropagation(역전파)는 출력층의 결과와 실제 정답 사이의 오차를 활용하여 순전파의 반대 방향으로 이동하며 W와 b를 최적화하는 propagation임. NN을 활용하려면 학습에 따라 W와 b를 최적화할 수 있어야 하는데, 이를 backpropagation으로 수행할 수 있음.

Chain Rule을 활용하여 backpropagation을 수행할 수 있음. x와 y 사이의 미분가능한 관계식과, y와 z 사이의 미분가능한 관계식이 존재한다면 아래와 같이 chain rule에 의해 x와 z 사이의 관계식을 계산할 수 있음.

$$\frac{\partial x}{\partial z} = \frac{\partial x}{\partial y} \cdot \frac{\partial y}{\partial z}$$

각 층에서 사용하는 W와 b에 대한 cost function의 변화율(gradient)을 계산하면 gradient descent로 학습할 수 있음. cost function을 L(여기에서는 제곱 오차로 가정함.), i번째 층에서의 W, b, 출력값, 활성 함수를 각각 W_i, b_i, y_i, f_i, 최종 출력값과 정답을 y_{pred}, y_{true}라 할 때, 아래의 수식이 성립함.

$$L = \frac{1}{2}(y_{pred} - y_{true})^2$$

$$y_i = f_i(y_{i-1}W_i + b_i)$$

아래와 같이 바로 뒤에 오는 층의 값인 $\frac{\partial L}{\partial y_{i+1}}$를 알고 있을 때, 각 관계식이 모두 존재하므로 각 층의 W에 대한 L의 gradient를 항상 계산할 수 있음. 또한 출력층 바로 앞쪽 은닉층에 대한 관계식도 간단히 구할 수 있으므로 수학적 귀납법에 의해 모든 gradient를 항상 구할 수 있음. b 또한 동일한 방식으로 구할 수 있음.

$$\frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial y_{i+1}} \cdot \frac{\partial y_{i+1}}{\partial y_i}$$

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial W_i}$$

물론 pytorch에서 이 과정은 아래의 코드만 작성하면 알아서 수행됨.

```
optimizer.zero_grad() # gradient 초기화
cost.backward() # backpropagation 수행. 각 층에 대한 gradient 계산
optimizer.step() # 계산한 gradient를 각 층의 W와 b에 반영
```

SLP만을 사용한 초창기에는 선형적인 학습으로 AND/OR 연산을 수행하는 모델을 만들 수는 있었지만, 근데 XOR 연산과 같이 비선형적인 모델이 필요한 문제는 풀 수 없었음. 또한 Marvin Minsky에 의해 SLP만으로 이를 풀 수 없고, MLP를 사용해야 함이 수학적으로 증명되었음. 처음에는 MLP에서 각 가중치를 최적화하는 방법을 찾지 못했지만, 이후 backpropagation이 등장하면서 MLP의 가능성이 다시 열렸음.

circuit theory에 의하면, 어떤 기능을 수행하는 NN이 shallow해지면, 그만큼 더 많은 hidden unit을 활용해야 동일한 기능을 수행할 수 있음.

1.1.2. NN 모델 정의

pytorch에서 제공하는 torch.nn(neural network) 또는 nn.Sequential 모듈 등을 사용하여 모델을 정의할 수 있음.

1. nn.Module
Module은 neural network 모델을 정의하는 class로, torch.nn에 포함되어 있음.

모델 구성 시에는 nn.Module을 상속받는 class를 정의하고, ___init___()과 forward()를 오버라이드함. 이렇게 정의된 class에 대한 객체를 생성하여 그 식별자로 호출하면 내부적으로 forward()가 호출되고, 가설에 따른 연산 결과가 반환됨. 즉, 모델을 생성하고, 해당 모델에 정의된 연산을 수행함. nn.Linear, nn.Conv2d, nn.RNN 등 nn.Module을 상속받는 class들은 동일하게 식별자를 호출하는 식으로 사용함.

코드는 아래와 같음.

```
import torch.nn as nn

class MyModel(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.fc1 = nn.Linear(10, 50)  # 입력 10, 출력 50
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(50, 1)  # 출력 1

    def forward(self, x) :
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

model = MyModel()
...
hypothesis = model(x_train)
```

2. nn.Sequential

nn.Sequential은 여러 계층을 가지는 모델을 간단히 생성하는 class로, torch.nn에 포함되어 있음.

아래와 같이 단순히 계층을 순서대로 작성하여 모델을 생성할 수 있음. nn.Module를 활용할 때와 같이 forward()를 작성할 필요가 없이, 단순히 객체만 생성하면 됨.

```
model = nn.Sequential(
    nn.Linear(10, 20),  # 입력 10, 출력 20
    nn.ReLU(),          # 활성 함수
    nn.Linear(20, 1)    # 입력 20, 출력 1
)
...
hypothesis = model(x_train)
```

1.1.3. DL

DL(Deep Learning)은 NN(DNN)을 사용하여 학습하는 ML의 한 분야임. DL에서는 NN, RNN, CNN 등을 NN으로 사용하여 학습을 수행함.

DL의 과정은 아래와 같음.
1) NN 아키텍처 설계
2) 학습을 수행하고, overfitting이 일어나기 전까지 model size(깊이, 넓이)를 늘림.
3) overfitting이 일어나면 regularization 등을 수행해 해결하고, 다시 2번으로 돌아감.

또한, underfitting이 발생했으면 NN을 더 넓고 깊게 구성함.

기본적으로 DL은 정의된 방식에 따라 입력에 W를 곱해 출력을 생성하고, 정답과의 loss를 계산하여 최적의 W를 찾는 과정을 반복해 학습을 수행함.

코드 상에서의 과정은 아래와 같음.
1) 라이브러리 import
2) GPU 사용 설정
3) 학습 관련 parameter(lr, epochs, batch_size 등) 정의
4) 데이터셋 및 DataLoader 등 구성
5) NN 모델 정의
6) loss function, optimizer 선택
7) 학습 및 역전파 수행
8) 평가

1.2. 활성 함수

1.2.1. 활성 함수

활성 함수(Activation Function)는 뉴런의 선형 연산(xW+b) 결과를 비선형적으로 변환하는 함수로, sigmoid 함수, softmax 함수, ReLU 함수 등이 있음. 활성 함수는 입력과 출력에 대한 비선형적인 관계를 모델링할 수 있도록 함.

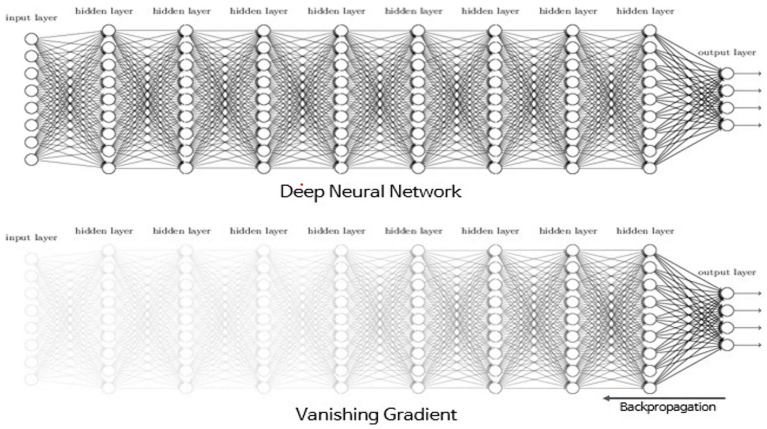
활성 함수의 출력값, 또는 특정 layer의 출력값을 Activation이라고 함. 이때 앞 layer의 출력은 뒷 layer의 입력이므로, 특정 layer에 대해 입력으로 들어가는 activation을 Input Activation, 출력으로 나오는 activation을 Output Activation이라고 함.

activation function이 존재하지 않는다면, network가 단순히 여러 행렬을 곱하는 연산이 됨.

1.2.2. ReLU

1. Vanishing Gradient Problem
역전파 시에 활성 함수를 미분하여 gradient를 구하게 됨. 그런데 sigmoid 함수는 입력값이 극단에 가까워질수록 출력값이 급격히 작아지므로, 역전파 시에 입력층에 가까운 층까지 값이 전달되지 못할 수

있음. 이와 같이 여러 개의 층을 거치며 gradient 값이 소실되는 현상을 Vanishing Gradient Problem 이라고 함.



2. ReLU

ReLU 함수는 값이 0보다 크거나 같을 때는 원래의 값을, 0보다 작을 때는 0을 출력하는 함수임. 즉, 수식은 아래와 같음.

$$f(x) = \begin{cases} x & if\ y \geq 0 \\ 0 & if\ y < 0 \end{cases} = \max(0, x)$$

물론 이 경우에도 입력값이 0보다 작을 때 gradient가 소실되기는 하지만, 실제로 사용해 보면 잘 동작한다고 함.

ReLU는 gradient가 0 또는 1로만 계산되기 때문에 연산이 굉장히 간단함. 또한 0에서 미분이 불가능해도 실제로 값이 정확히 0인 경우는 잘 없고, 0이더라도 gradient를 단순히 0 또는 1로 결정해 주면 됨.

pytorch에는 relu(), leaky_relu() 등의 함수로 ReLU를 사용할 수 있음. 활성 함수로는 sigmoid보다 ReLU를 주로 사용하고, sigmoid는 마지막의 classification 등에 사용함.

```
torch.nn.relu(x)
torch.nn.leaky_relu(x, 0.01)
```

1.2.3. 기타 활성 함수들

아래와 같이 다양한 활성 함수들이 있음.

1) sigmoid : dynamic range가 제한되어 있어 quantize가 쉽지만 gradient vanishing이 발생함.

sigmoid 함수 $\sigma = \frac{1}{1+e^{-1}}$는 미분하면 $\sigma(1-\sigma)$가 됨.

2) ReLU : 값이 음수이면 gradient가 없어지지만, 이에 따라 sparsify하기 쉬움. 또한 gradient가 항상 0또는 1이므로 개별 gradient값을 저장해둘 필요가 없이, bit mask를 수행하여 gradient가 0이면 0을 출력하고, 1이면 이전 featrue map 값을 출력함. 대신 dynamic range에 제한이 없어 quantize가 어려움.

3) ReLU6 : 6을 넘어가는 입력을 6으로 맞추는 ReLU. dynamic range를 제한해 quantize하기 쉽게 함.

4) leaky ReLU : 음수인 입력에 대해 작은 gradient를 주는 ReLU. 음수 입력에 대한 gradient가 없어지지는 않지만, sparsity 관점에서의 이점은 없어짐. 아래와 같이 음수 쪽에 a(ex. 0.01)를 곱하고, 미분 시에 음수에 대한 gradient가 a가 됨.

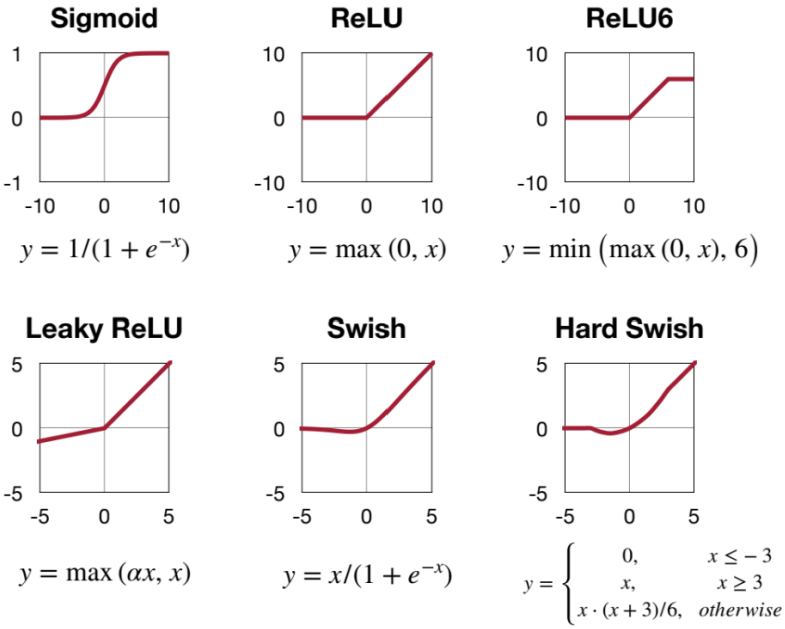
5) swish : ReLU와 개형이 유사하지만, 성능적인 이점이 있음. 다만 하드웨어적인 구현이 어려움.

6) hard swish : swish와 유사하지만 구현이 간단하도록 한 것.

7) tanh : sigmoid와 유사하지만 -1부터 1 사이의 값을 가짐. 수식은 아래와 같음.

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

tanh 함수 f를 미분하면 $1 - f^2$*이 됨.*



2. DL 관련 고려사항

2.1. 학습 데이터셋 구성

2.1.1. 데이터셋 분리

1. 평가

학습 이후에는 유의한 결과가 도출되었는지 평가(Evalutation) 또는 테스트해야 함.

아래와 같이 학습 이후 별도의 training set으로 출력값을 얻고, 해당 값을 정답과 비교하여 데이터셋에 대해 평균을 내면 모델의 정확도(Accuracy)를 계산할 수 있음.

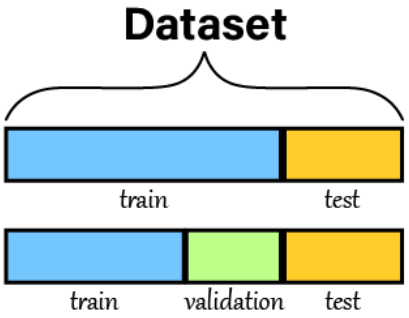
이때 torch.no_grad()를 호출하여 autograd의 동작을 비활성화하여 테스트 시의 연산 속도를 높일 수 있음.

```
# 평가
x_test = torch.FloatTensor([[1, 2], [2, 3], [3, 1]])
y_test = torch.FloatTensor([[0], [0], [0]])

with torch.no_grad() :
    hypothesis = model(x_test)
    prediction = hypothesis >= torch.FloatTensor([0.5])
    rst = torch.mean((prediction == y_test).float())
```

2. 데이터 분리

테스트 시에 사용하는 데이터셋을 training set으로 그대로 활용하면 유의미한 테스트를 하기 어렵고, overfitting이 발생할 수 있음.

그래서 데이터셋을 분리하여 일부는 training set으로, 나머지는 test set으로 활용함. 또한 training set은 다시 training set과 validation(development) set으로 분리함. validation set은 epoch 또는 특정 주기마다 사용하여 lr, λ 값 등을 튜닝할 때 등에 활용하는데, test set 사용 이전에 확인용으로 사용하는 데이터셋임.



2.1.2. 학습 데이터셋 구성

pytorch에서 제공하는 torch.utils.data 모듈을 사용하여 학습 데이터셋을 구성할 수 있음.

1. Dataset

Dataset은 pytorch에서 데이터셋을 구성하는 기본 class로, torch.utils.data에 포함되어 있음.

데이터셋 구성 시에는 Dataset을 상속받는 class를 정의하고, 전체 데이터를 필드로 저장하고, ___len___()과 ___getitem()___()을 오버라이드함. ___len___()은 데이터셋의 총 데이터 개수(batch size)를 반환하도록 구현하고, ___getitem()___은 지정한 인덱스에 대한 입력 데이터와 출력 데이터를 tuple로 반환하도록 구현함.

```
from torch.utils.data import Dataset

class MyDataset(Dataset) :
    def __init__(self) :
        self.x_data = [[73, 80, 75], [93, 88, 93]] # list로 작성 가능
        self.y_data = [[152], [185]]
    def __len__(self) :
        return len(self.x_data)
    def __getitem__(self, index) :
        x = torch.FloatTensor(x_data[index])
        y = torch.FloatTensor(y_data[index])
        return x, y

dataset = MyDataset()
```

2. DataLoader

DataLoader는 데이터를 특정 size의 batch(minibatch) 단위로 가져올 수 있도록 하는 class로, torch.utils.data에 포함되어 있음.

아래와 같이 Dataset 객체와, batch_size(각 minibatch 크기), shuffle(데이터 학습 순서 섞음.) 여부 등을 지정함. 이때 통상적으로 batch size는 2의 거듭제곱수로, shuffle은 True로 지정함.

```
from torch.utils.data import DataLoader

dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

DataLoader 객체는 생성 시에 지정한 Dataset 객체의 ___getitem()___ 반환값을 활용하여 batch size 개수만큼의 데이터를 반환함. epoch 반복문 안에 DataLoader 객체의 값을 받는 반복문을 작성하거나, enumerate()를 사용한 반복문을 작성하여 minibatch 별 학습을 수행할 수 있음. Dataset를 DataLoader에 넣지 않고 바로 enumerate()에 넣을 수도 있지만, 이 경우 한 번에 하나의 데이터만이 반환됨.

```
for epoch in range(nb_epochs + 1):
    for batch_index, batch_data in enumerate(dataloader) :
            x_train, y_train = batch_data

            hypothesis = model(x_train)
            cost = F.mse_loss(hypothesis, y_train)

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

3. Minibatch Gradient Descent

Minibatch Gradient Descent는 전체 데이터를 minibatch라는 단위로 균일하게 나눈 뒤 각 minibatch를 사용하여 학습하는 gradient descent임.

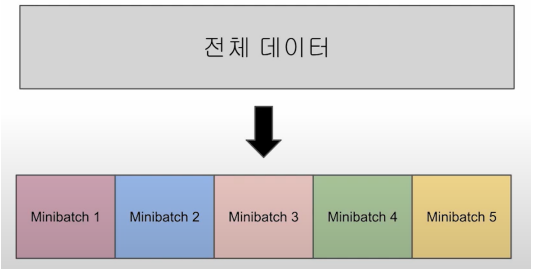
학습을 통해 유효한 결과를 내려면 많은 양의 데이터를 활용해야 하는데, 전체 데이터에 대해 한 번에 연산을 수행하는 것은 너무 많은 시간이 걸리고 하드웨어적인 제약이 존재할 수 있음. 이에 따라 한 번의 epoch(전체 데이터) 내에서 데이터를 여러 minibatch로 나누어 학습시킴. 또한 이렇게 학습하면 새로운 데이터를 추가하기에도 편리함. 이 경우 단순히 전체 데이터를 사용할 때처럼 epoch에 따라 cost가 매끄럽게 줄어드는 대신, 거칠게(변동성이 있게) 줄어들 수 있음.

이때 minibatch size는 대체로 2의 거듭제곱으로 지정함. 이는 memory 활용에서의 이점이 있다고 함. 또한 batch size를 지정하고 남은 부분은 batching하고 남는 부분은 버리기도 하고, 그걸로만 돌리기

도 한다고 함. 하지만 그 부분이 작고, 어차피 random하게 뽑아서 돌리는 경우가 많으므로 큰 영향이 있지는 않음.

이와 유사하게 데이터가 순차적으로 주어질 때 각 부분씩 실시간으로 학습시키는 방식을 Online Learning이라고 함.

pytorch에서 제공하는 Dataset과 DataLoader class를 사용해 이를 간단히 구현할 수 있음.



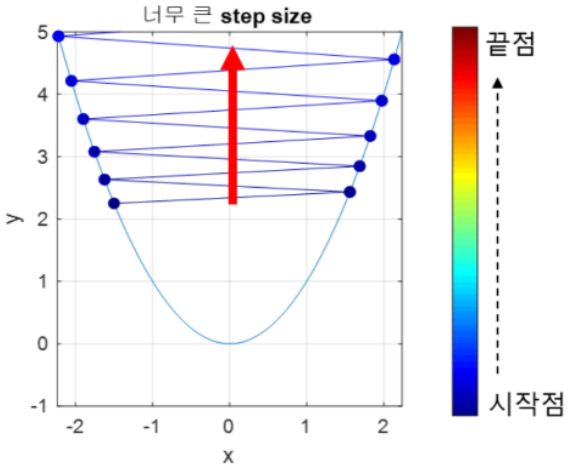
SGD(Stochastic Gradient Descent)는 전체 데이터셋이 아니라, batch별로 처리를 할 때마다 gradient descent를 수행하는 방식임.

2.2. Learning Rate

2.2.1. Learning Rate 설정

gradient descent에서는 적절한 learning rate를 설정하는 것이 중요함.

learning rate가 너무 크면 overshooting이 발생함. 이는 매 연산 시에 최소로부터 더 멀어지는 방향으로 이동하게 되는 현상을 말함. epoch 별 cost를 찍어 보면 cost가 점점 커짐.



learning rate가 너무 작으면 최적화에 너무 많은 시간이 필요하게 되거나 local minimum을 찾게 됨. cost를 찍어 보면 cost에 변화가 너무 적음.

learning rate는 단순히 경험적으로 설정해야 함. 주로 0.01정도로 시작하고 cost의 변화를 확인하여 조정함.

fine-tuning 시에는 pre-train에 의해 이미 loss가 줄어들어 있으므로 learning rate를 작게 하는 것이 좋다고 함. 물론 실제로 어떤 값이 최적인지는 경험적으로 확인해야 함.

2.2.2. Learning Rate Scheduler

Learning Rate Schedular는 학습 중에 lr을 동적으로 조정하는 도구임. 고정된 lr을 사용하는 것보다

점진적으로 변화를 주는 것이 모델의 성능이 개선되는 경우가 많다고 함.

pytorch에서는 torch.optim.lr_scheduler에 포함된 lr schedular를 활용할 수 있음. 아래와 같이 사용함. StepLR()은 지정한 수의 epoch(step_size)마다 lr에 gamma 값을 곱함.

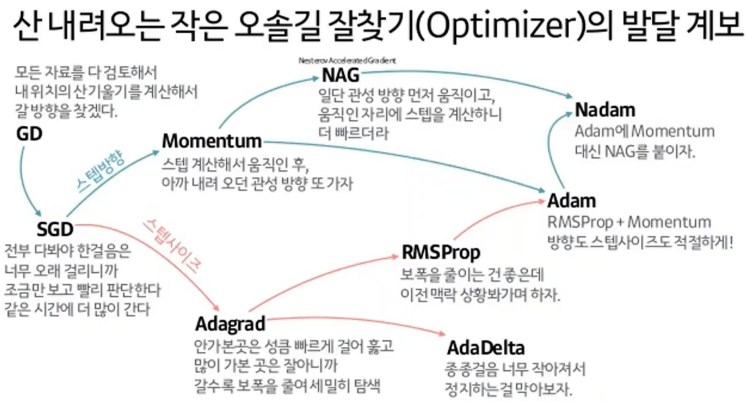
epoch마다 learning rate를 점진적으로 줄이는 기법을 Learning Rate Decay라고 함. 0.95^{epoch}*을 곱하거나,* $\frac{k}{\sqrt{epoch}}$*을 곱하거나, discrete하게 지정하거나, manual하게 지정하는 등 여러 가지 방법이 존재함.*

```
lr_sche = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.9)
...
for epoch in range(1, epochs + 1) :
    lr_sche.step()
    ...
```

2.3. Optimizer

2.3.1. Optimizer

pytorch에서 제공하는 Optimizer는 cost function을 활용하여 모델을 학습(최적화)함. torch.optim에는 pytorch에서 제공하는 다양한 optimizer들이 있음. 각 optimizer 별로 나름의 알고리즘을 사용함.



optimizer별로 정의하는 방법은 다를 수 있지만, 아래와 같이 모델의 parameter와 learning rate를 전달해야 하는 것은 동일함. 또한 동일한 코드로 학습을 수행할 수 있음.

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# 다른 optimizer를 사용하더라도 아래의 코드로 학습을 할 수 있음.
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

loss function이 여러 극소를 가지는 경우가 존재할 수 있으므로, Adam optimizer 등에서는 gradient에 대한 momentum을 적용하는 기법을 사용함. 즉, 이전 step의 gradient들을 저장해 두고, 현재 시점의 gradient만이 아니라 이전 gradient들에도 가중치를 줘서 활용함. 이전 gradient 값이 충분히 컸다면 갑자기 값이 작아지는 지점을 벗어날 수 있게 됨.

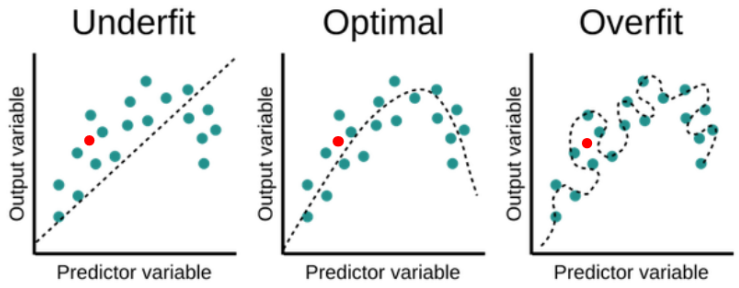
2.4. Overfitting

2.4.1. Overfitting

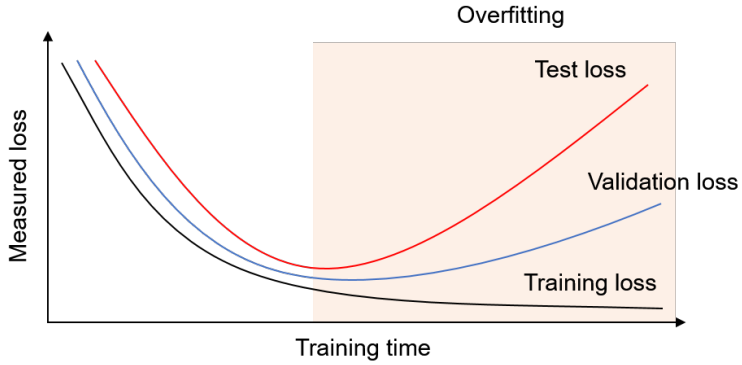
1. Overfitting

Overfitting은 학습이 사용된 데이터에 대해 과하게 최적화되어 일반적인 경우에 대한 설명력이 떨어지는 현상을 말함. 즉, 학습 데이터에만 잘 들어맞고 다른 데이터에는 잘 동작하지 않는 경우를 말함. 이는 그래프로 보면 어떤 일반적인 직선을 찾은 것이 아니라, 데이터 분포에 딱 맞는 곡선을 찾은 것으로 생각할 수 있음.

이는 High Variance Problem이라고도 함. 반면 model이 너무 단순한 경우 underfitting이 일어날 수 있는데, 이는 High Bias Problem이라고 함. high bias problem은 더 학습시키거나, model을 더 복잡하게 구성해 해결할 수 있음.



epoch가 커질수록 training set에 대한 cost는 줄어들지만, test/validation set에 대한 cost는 줄어들었다가 다시 늘어나게 됨. 이렇게 cost가 다시 늘어나는 지점이 overfitting이 발생하는 지점임. test set을 사용하기 전에 validation set으로 overfitting이 어느 정도 발생하는지는 확인해 볼 수 있음. 물론 validation set이 test set과 데이터 분포가 일치하지는 않기 때문에 overfitting 발생 지점을 항상 일치하는 것은 아님.



2. Overfitting 방지

overfitting을 방지하는 방법으로는 아래와 같은 것들이 있음.

1) 더 많은 데이터를 사용함.

2) 일부 feature를 제거함.
feature가 많고 구체적이라면 모델이 데이터를 너무 잘 설명하게 될 수 있음.

3) Regularization
Regularization은 overfitting을 방지하기 위해 모델을 제한하는 기법임. L1 regularization, L2 regularization, dropout 등이 있음.

3. Regularization

Regularization(정규화, 정칙화) 또는 Weight Decay는 model에 패널티를 부여하여 overfitting을 방지하는 기법임. regularation으로는 L1 regularization과 L2 regularization이 있음. 이 두 방식 모두 최적화 시에 W가 0에 가까워지도록 해 영향이 적은 feature를 제거함.

L1/L2 regularization은 각각 아래의 수식과 같이 cost function에 $\lambda|W|$와 $\lambda||W||^2$를 패널티로 더함. 즉, 각각 L1 norm의 제곱에 λ를 곱한 값을, L2 norm의 제곱에 λ를 곱한 값을 더함. 이때 λ는 패널티의 크기를 지정하는 양의 실수로, hyper parameter임.

$$L' = L(x; W) + \lambda |W|$$

$$L' = L(x; W) + \lambda ||W||^2$$

이에 대해 gradient descent를 적용하는 것을 생각해 보면, W가 양수일 때는 cost function의 기울기가 더 크게 계산되고, W가 음수일 때는 cost function의 기울기가 더 작게 계산됨. 즉, 두 방식은 모두 패널티를 더해 cost function의 기울기를 W가 0에 가까워지도록 조정함. 결과적으로 W가 전반적으로 0에 가까워져 영향이 적은 feature는 제거됨.

이때 L1은 λ가 충분히 작아지면 W는 0으로 최적화되고, L2는 λ가 충분히 작아져도 0으로 최적화되지는 않음. 각 loss를 미분한 것을 생각해 보면 L1에서는 graient가 $\lambda(sign(W))$이고, L2에서는 gradient가 $2\lambda W$이므로 쉽게 이해할 수 있음. 또한 L2는 w의 크기에 따라 패널티 값이 정해지기 때문에 W 값이 부드럽게 0에 근접하게 됨.

어떨 때는 L1, 어떨 때는 L2가 더 나아서, 실험적으로 판단해야 함.

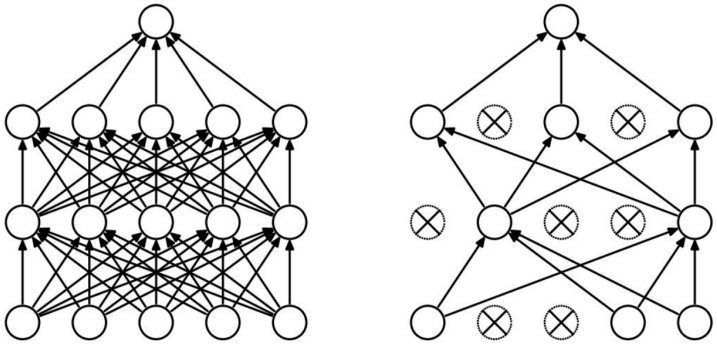
2.4.2. Dropout

1. Dropout

Dropout은 학습 시에 NN에 존재하는 각 뉴런(노드) 중 설정된 조건에 따라 일부 뉴런을 해당 epoch에서 일시적으로 사용하지 않는 기법임. 즉, 현재의 epoch에서는 정해진 뉴런들을 사용하지 않은 채 backpropagation을 수행하고, 그 다음 epoch에서는 사용하지 않을 뉴런들을 새롭게 결정함.

dropout은 여러 형태의 NN을 사용한 효과를 낼 수 있고, 또한 overfitting을 방지할 수 있음.

구체적으로는, activation과 동일한 크기의 tensor를 생성하고, 확률 값에 따라 각 값을 0과 1로 구성하여 이후 activation에 element-wise하게 곱함. 즉, 0에 해당되는 부분이 제거됨.



(a) Standard Neural Net (b) After applying dropout.

2. 코드

pytorch에서는 아래와 같이 nn.Dropout class의 객체를 층 중간에 넣어서 구현할 수 있음. 이때 p에는 전체 뉴런 개수에 대한 사용하지 않을 뉴런의 비율을 지정함.

```
...
dropout = torch.nn.Dropout(p=drop_prob)
model = torch.nn.Sequential(linear1, relu, dropout,
                            linear2, relu, dropout,
                            linear3, relu, dropout,
                            linear4)
...
```

물론 이는 학습 시에 사용하는 기법으로, 테스트 시에는 사용하지 않음. 이는 모델에 대해 train()과 eval() 메서드를 사용하면 자동으로 적용됨. 객체에 대해 학습/테스트 모드를 지정하는 것임.

```
# learning (dropout 적용)
model.train()
...

# test (dropout 적용 x)
model.eval()
...
```

2.5. Normalization

2.5.1. Normalization

1. Preprocessing
학습 시에는 데이터에 대해 적절한 Preprocessing(전처리)가 필요할 수 있음.

일반적인 데이터에 대해서도 preprocessing을 수행하면 더 좋은 결과를 얻을 수 있지만, W 등의 각 값이 동등하지 않은 경우 그 효용이 커짐. W의 원소 중 어떤 것의 변화율은 작은데 다른 것의 변화율은 그에 비해 너무 크다면, 적절한 lr을 사용했음에도 gradient descent 시에 값이 overshooting되거나 결과가 제대로 도출되지 않을 수 있음.

preprocessing으로는 zero centering, normalization, standardization 등을 수행할 수 있음.

2. Normalization
Normalization(정규화)는 데이터의 값을 특정 범위로 맞추는 기법임. 이는 특정 feature의 영향이 과하게 커지는 것을 방지함. 각 원소에 대해 아래의 수식을 적용하여 0과 1 사이로 normalization할 수 있음. 특히 실세계의 데이터는 적절히 scaling하는 것이 중요함.

$$\frac{x_i - x_{min}}{x_{max} - x_{min}}$$

3. Standardization
Standardization(표준화)는 데이터의 평균을 0으로, 표준편차를 1로 조장하는 기법임. 이는 변동이 큰 feature의 영향이 과하게 커지는 것을 방지함. 각 원소에 대해 아래의 수식을 적용하여 standarization 할 수 있음. m은 평균, σ는 표준편차임.

$$\frac{x - m}{\sigma}$$

standarization은 z-score normalization, 혹은 그냥 normalization이라고도 함.

normalization, standarization, regularization 모두 동일하게 '정규화'로 번역하는 경우가 있지만, 구분 가능한 개념임.

2.6. Weight Initialization

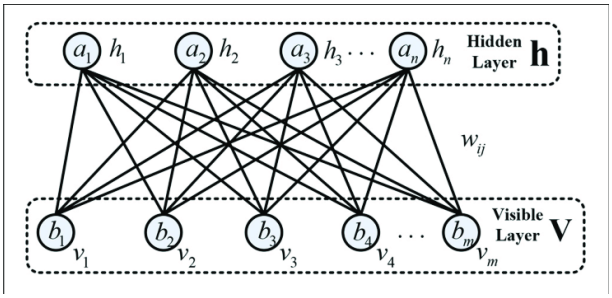
2.6.1. Weight Initialization

1. Weight Initialization
적절한 weight initialization 기법을 적용하면 훨씬 정확하고 빠른 학습을 수행할 수 있고, vanishing gradient problem을 방지할 수 있음.

2. RBM
RBM(Restricted Boltzmann Machine)은 2개의 층에 대해서, 아래의 두 가지 연산을 가지는 모델임. 이때 층 내부에는 연결이 존재하지 않고, 반대편 층에 대해서는 모든 연결이 존재함.

1) forward(encoding) : forward 방향으로 가중치를 활용하여 값을 계산해 넘기는 연산.
2) backward(decoding) : backward 방향으로 가중치를 활용하여 값을 계산해 넘기는 연산.

RBM에서는 forward를 수행한 뒤 backward를 수행하여, 그 결과값과 원래의 입력값의 차이를 계산함. 이 차이가 최소가 되도록 두 층 사이의 가중치를 최적화함(gradient descent 등을 활용함.).



3. DBN
DBN(Deep Belief Network)은 여러 개의 RBM을 쌓아 만든 NN으로, Hinton이 제안한 방법임. 즉, NN의 전체 층 중에서 이웃한 두 개의 층을 RBM으로 생각하여 입력층에서 출력측 방향으로 각 W를 하나씩 초기화함.

이렇게 DBN을 구성하는 것을 pre-tuning이라고 하고, 구성된 DBN을 활용하여 학습하는 것을 fine-tuning이라고 함.

물론 현재는 복잡한 DBN 대신 Xavier/He initialization 등을 사용함. 이는 더 단순하면서도 효과적임.

4. Xavier/He initialization
Xavier/He initialization은 정규분포(Normal Distribution) 또는 균등분포(Uniform Distribution)에서 추출한 수를 가중치의 값으로 하는 방식임. 이때 Xavier initialization은 가중치의 입력과 출력의 개수에 따라 확률분포가 결정되고, He initialization는 입력의 개수에 따라 확률분포가 결정됨.

그 수식은 아래와 같음. n_{in}은 입력 neuron의 개수, n_{out}은 출력 neuron의 개수임. 각각 해당되는 normal distribution에서 임의의 값을 꺼내 initialization value로 함.

- Xavier Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}+n_{out}}}$$

- He Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

- Xavier Uniform initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in}+n_{out}}}, +\sqrt{\frac{6}{n_{in}+n_{out}}})$$

- He Uniform initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}})$$

pytorch에서는 아래와 같이 nn.init.xavier_uniform_()을 사용하여 초기화할 수 있음.

```
import torch

linear1 = torch.nn.Linear(100, 90, bias=True)
linear2 = torch.nn.Linear(90, 110, bias=True)
torch.nn.init.xavier_uniform_(linear1.weight)
torch.nn.init.xavier_uniform_(linear2.weight)
```

참고로 W를 모두 0으로 초기화하는 방식은 적절하지 않음. gradient는 뒤쪽 층의 gradient를 곱하는 방식으로 계산되는데, W가 0이면 모든 gradient 값이 계속해서 0으로 계산될 수 있음.

균등분포(Uniform Distribution)는 모든 값이 동일한 확률을 갖는 확률분포로, 이산 균등분포와 연속 균등분포로 구분됨.

2.6.2. Normalization Layer

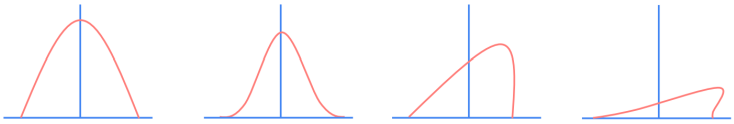
1. Vanishing/Exploding Gradient Problem
vanishing gradient problem은 앞에서 다룬 것과 같이 gradient가 소실되는 문제이고, Exploding Gradient Problem은 반대로 gradient가 과하게 커지는 문제임.

이 문제들은 아래의 기법으로 해결할 수 있음.

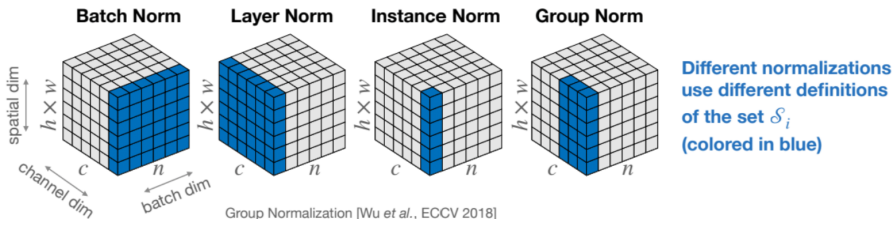
1) 적절한 활성 함수 사용(ReLU 함수 등).
2) 적절한 initialization 기법 사용.
3) exploding gradient problem의 경우 작은 learning rate을 사용함.
4) batch normalization 사용.
5) residual connection 사용.

2. Normalization Layer
NN에서 normalization을 수행하는 layer를 Normalization Layer라고 함. NN 중간에서의 normalizatoin은 학습을 더 빠르고 정확하고 안정적으로 수행할 수 있게 함.

internal covariate shift는 vanishing/exploding gradient problem을 유발하고, 안정적인 학습을 방해함. Covariate Shift는 데이터셋끼리의(training set과 test set) feature 분포가 다른 현상으로, 모델이 적절한 예측을 수행하기 어렵게 함. Internal Covariate Shift는 NN에서 각 층마다 발생하는 covariate shift임. 즉, 학습 시에 각 layer에 대한 데이터 분포가 달라지는 문제이고, 이는 NN이 깊어질수록 심해짐. 특정 layer의 데이터 분포가 변화는 그 다음 layer에 대한 입력 데이터 분포 변화이므로 쉽게 이해할 수 있음. normalization layer을 거쳐 데이터 분포를 맞추면 internal covariate shift를 해결할 수 있음.



normalization 대상에 따라 Batch/Layer/Instance/Group Normalization이 존재함.



Group Normalization [Wu *et al.*, ECCV 2018]

3. Batch Normalization
Batch Normalization은 NN의 layer에서 각 batch에 대해 normalization을 수행하는 기법임.

batch normalization은 아래와 같은 과정으로 계산됨.

1) 각 minibatch별로 값의 평균과 분산을 계산하여 표준화함. 이때 학습 시에 매 minibatch에 대해 계산한 평균과 분산은 sample mean/variance라고 하고, 학습을 반복하며 얻은 sample mean/variance들에 대한 평균을 계산한 것을 learning mean/variance라고 함. 학습이 끝나게 되면 learning mean/variance는 고정된 값을 가지므로 inference 시에는 이를 활용하여 batch normalization을 수행함.

2) 연산 후 표현 능력이 손실되었을 수 있으므로, 표준화한 값에 스케일링 파라미터(값 범위 조정) γ를 곱하고, 시프트 파라미터(값 평행이동) β를 더함. γ와 β는 가중치와 마찬가지로 역전파 시에 최적화함.

batch normalization을 적용한 경우 β가 bias처럼 동작하므로, 이후 linear layer에서는 bias를 사용하지 않을 수 있다고 함.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

4. 코드

pytorch에서는 아래와 같이 nn.BatchNorm1d class로 normalization layer(batch normalization)를 구현할 수 있음.

dropout처럼 학습 시에는 train() 메서드를 호출하고, 테스트 시에는 eval() 메서드를 호출해 모드를 지정해 줘야 함. sample mean/variance는 매번 feature 별로 계산하므로 동일한 데이터라도 minibatch의 구성이 달라지면 다른 값이 도출될 수 있는데, 테스트 시에는 매번 다른 값이 나와서는 안 됨. 이에 따라 테스트 시에는 learning mean/variance를 사용하여 batch normalization을 함. train()을 호출하면 sample mean/variance를 사용하고 그 leaning mean/variance를 계산하고, eval()을 호출하면 계산되어 있는 leaning mean/variance를 사용함.

batch normalization은 선형변환 계산 이후 활성함수 사용 직전에 적용함.

```
linear1 = torch.nn.Linear(784, 32, bias=True)
linear2 = torch.nn.Linear(32, 32, bias=True)
linear3 = torch.nn.Linear(32, 10, bias=True)
relu = torch.nn.ReLU()
bn1 = torch.nn.BatchNorm1d(32) # 정규화할 요소의 개수 지정
bn2 = torch.nn.BatchNorm1d(32)
...
model = torch.nn.Sequential(linear1, bn1, relu,
                            linear2, bn2, relu
                            linear3)
...
# learning
model.train()
...
# test
model.eval()
```

batch normalization layer는 CNN 등에서 주로 사용됨.

2.7. 요약

2.7.1. 요약

앞에서 살펴본 기법들을 모두 적용하여 MNIST 데이터셋을 학습하는 코드는 아래와 같음.

```
import torch
```

```
import torchvision.datasets as dsets
import torchvision.transforms as transform
from torch.utils.data import DataLoader, Subset
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import time

# parameter
learning_rate = 0.001
batch_size = 128
epochs = 10
subset_size = 20000
test_size = 2000

# dataset
mnist_train = dsets.MNIST(root="MNIST_data/", train=True,
                          transform=transform.ToTensor(), download=True)
mnist_train = Subset(mnist_train, list(range(subset_size)))
dataloader_train = DataLoader(mnist_train, batch_size=batch_size, shuffle=True)

mnist_test = dsets.MNIST(root="MNIST_data/", train=False,
                          transform=transform.ToTensor(), download=True)
mnist_test = Subset(mnist_test, list(range(test_size)))
dataloader_test = DataLoader(mnist_test, batch_size=len(mnist_test), shuffle=True)

# model
class MnistModel(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            # |x| = (batch_size, 32, 28, 28)
            nn.ReLU(),
            nn.MaxPool2d(2) # |x| = (batch_size, 32, 14, 14)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            # |x| = (batch_size, 64, 14, 14)
            nn.ReLU(),
            nn.MaxPool2d(2) # |x| = (batch_size, 64, 7, 7)
        )

        self.linear = nn.Linear(64 * 7 * 7, 10)
        nn.init.xavier_uniform_(self.linear.weight)

    def forward(self, x) : # |x| = (batch_size, 1, 28, 28)
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.linear(x)

        return x
```

```
model = MnistModel()

# optimizer
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# train
model.train()
train_start_time = time.time()
for epoch in range(1, epochs + 1) :
    for batch_index, batch_data in enumerate(dataloader_train) :
        x_train, y_train = batch_data
        x_train = x_train.view(-1, 1, 28, 28)

        # hypothesis
        hypothesis = model(x_train)

        # cost
        cost = F.cross_entropy(hypothesis, y_train)

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

    print('Epoch {:4d}/{}, Cost: {:.6f}'.format(epoch, epochs, cost.item()))
train_end_time = time.time()
print("train time :" , (train_end_time - train_start_time), "s")

# test
model.eval()
with torch.no_grad() :
    for batch_index, batch_data in enumerate(dataloader_test) :
        x_test, y_test = batch_data
        x_test = x_test.view(-1, 1, 28, 28)

        # hypothesis
        hypothesis = model(x_test)
        prediction = hypothesis.argmax(dim=1).long()
        accuracy = (y_test == prediction).float().mean()

        # cost
        cost = F.cross_entropy(hypothesis, y_test)

        print('test accuracy: {}, Cost: {:.6f}'.format(accuracy, cost.item()))
```

3. CNN

3.1. Convolution&Pooling

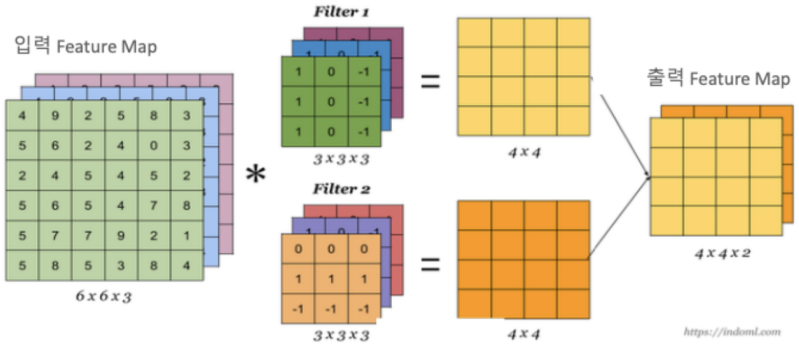
3.1.1. Convolution

1. Convolution
Convolution은 데이터에 대해 filter를 stride만큼 이동시키면서, 대응되는 위치에 있는 원소끼리 곱한

값을 모두 더한 결과를 출력으로 하는 연산임. 이때 Filter(Kernel)는 연산에 사용되는 별개의 tensor 이고, Stride는 filter를 이동시키는 간격임. convolution은 데이터의 feature를 추출하는 기능을 함.

convolution의 결과로 생성된 출력 데이터를 Feature Map이라고 함.

입력 데이터에 따라 filter도 여러 개의 channel을 가짐. filter의 channel 수는 입력 데이터의 channel 수와 같음. 또한 filter의 개수는 출력 데이터의 channel 수와 같음.



2. Receptive Field

convolution 시에 특정 뉴런이 입력 이미지에서 영향을 받는 영역을 Receptive Field라고 함. 이는 해당 뉴런이 속한 layer에 대한 filter 영역만이 아니라, 뉴런이 구성되는 데에 관여한 전체 영역을 의미함.

각 convolution에서 동일한 크기 k의 filter를 사용한다면, L개의 layer를 거치는 경우 receptive field의 크기는 $L \cdot (k-1)+1$임. 예를 들어, layer 1과 2 모두에서 3×3 filter를 stride 1로 한다면, layer 2의 뉴런이 가지는 receptive field의 크기는 5×5임.

어떤 부분에 대한 receptive field가 클수록 해당 부분이 많은 픽셀 간의 연관성을 반영하게 됨. 이에 따라 큰 receptive field를 구현하려면 깊은 NN을 사용할 수 있는데, 이에 따라 gradient vanishing problem이 발생할 수 있음. 또는 stride를 주는 방법도 있는데, 이 경우에는 accuracy가 떨어짐. 이런 tradeoff를 잘 고려하여 모델을 설계해야 함.

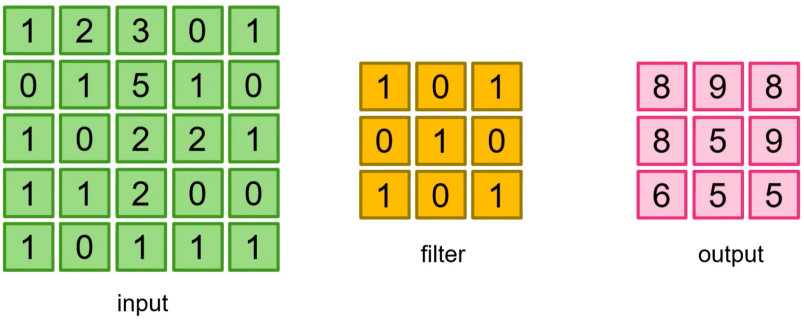
참고로 CNN에서는 특정 부분 주위의 픽셀들이 receptive field에 포함되지만, transformer는 해당 데이터 내의 모든 픽셀을 고려함. 즉, global receptive field가 구현됨.

3. Padding

convolution 연산 시에 입력 데이터에 Padding을 추가할 수 있음. padding 값이 n이라면 입력 데이터 둘레로 n개의 pad를 추가한 것임.

이때 pad의 값은 전부 0으로 하거나(zero-padding), 기존의 값을 기반으로 생성하여 넣기도 함(replication-padding 등).

convolution 연산에서는 입력 데이터에 비해 출력 데이터의 크기가 줄어들게 되므로 padding을 추가하여 출력 크기를 조정할 수 있음. 또한 입력 데이터에서 가장자리에 있는 부분은 내부의 부분에 비해 덜 사용될 수 있는데, padding을 추가하면 이를 개선할 수 있음.



4. Output Size

convolution에 의한 output size는 아래와 같이 계산할 수 있음. 이때 size는 한 변의 길이를 말함. FC

에 사용할 linear layer 정의 시에 W의 크기를 지정해야 하므로 output size를 계산할 수 있어야 함.

$$Output\ size = \frac{Input\ size\ -\ Filter\ size\ +\ (2\ *\ Padding)}{Stride} + 1$$

참고로 covolution layer에서의 parameter의 개수(연산량)는 filter의 전체 크기와 input channels, output channels를 곱해서 계산할 수 있음.

5. 1D Convolution vs. 2D Convolution
1D Convolution은 filter를 하나의 축(방향)에 대해서만 이동하며 수행하는 convolution이임. 오디오, 주가 데이터 등에 대해서 적용할 수 있음. 이 경우 데이터의 shape은 (batch_size, channels, width)임.

2D Convolution은 filter를 두 개의 축(방향)에 대해서 이동하며 수행하는 convolution임. 이미지 데이터 등에 대해서 적용할 수 있음. 이 경우 데이터의 shape은 (batch_size, channels, height, width)인데,

3.1.2. Grouped Convolution

1. Grouped Convolution
Grouped Convolution은 여러 channel을 가지는 입력 데이터를 channel에 대해 여러 group으로 나누어 각각에 대해 convolution을 수행하는 기법임.

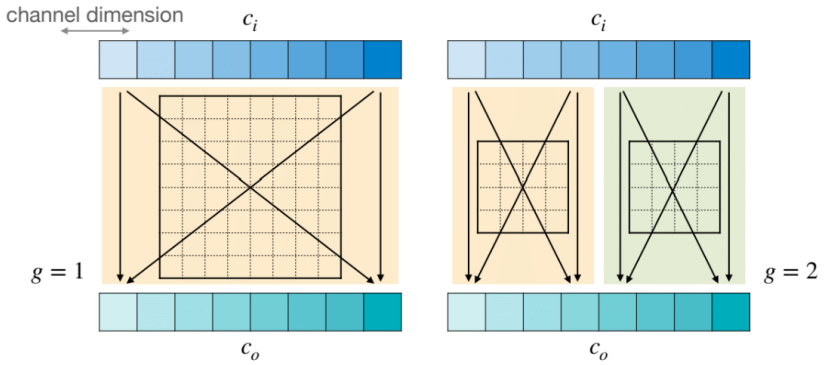
filter의 크기를 K, input channels를 C_{in}, output channels를 C_{out}, output width/height를 W_{out}, H_{out} 이라고 하면 convolution의 연산량은 아래와 같음.

$$(K \cdot K \cdot C_{in}) \cdot (H_{out} \cdot W_{out} \cdot C_{out})$$

group의 개수를 G라 하자. channel에 따라 여러 group으로 나누면 연산량이 아래와 같음. 데이터를 여러 group으로 나누면 연산량이 전체 부분에 대한 곱셈으로 계산되는 대신, 각 부분에 대한 덧셈으로 계산되므로 연산량이 줄어듦. 또한 병렬적으로 계산하므로 환경에 따라 더 효율적일 수 있음.

$$G \cdot (K \cdot K \cdot \frac{C_{in}}{G}) \cdot (H_{out} \cdot W_{out} \cdot \frac{C_{out}}{G})$$

groupt convolution을 적용하면 출력 feature의 개수는 유지되는데 parameter가 줄어듦. 즉, 출력 결과는 같은데 연산량이 적음. 다만 각 group에 대해 독립적으로 연산이 수행되므로 해당 channel들 사이의 상관관계를 반영하지 못함.



2. Depthwise Convolution
Depthwise Convolution은 grouped conovolution의 극단적 적용으로, 입력 channel 각각에 대한 group을 생성하는 기법임. 즉, input channels와 group 개수가 같음.

이 경우에도 출력 feature의 개수는 유지되지만 parameter가 줄어듦. 대신 channel끼리의 상관관계는 반영하지 못함.

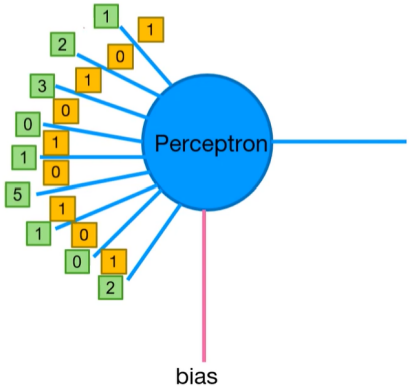
3.1.3. Convolution Layer

1. Convolution Layer

convolution 연산을 뉴런의 관점에서 보면 filter의 각 값이 가중치로 사용되어 곱해지고, 그 이후 bias가 더해지는 것임. DNN에서와 동일하게 filter의 가중치와 bias는 학습 시에 backpropagation에 의해 최적화됨. 이때 해당 가중치가 데이터의 각 부분에 공통적으로 적용되므로 이를 공유 가중치(Shared Weight)라고 하고, 이런 방식을 Parameter Sharing이라고 함.

FC에서는 각 뉴런이 직전 layer의 뉴런과 모두 연결되어 있었지만, convolution layer에서는 각 뉴런이 직전 layer의 국지적인 일부 뉴런들과 연결되어 있음.

linear layer에 비해 훨씬 적은 수의 parameter를 가짐.



2. 코드

pytorch에서는 아래와 같이 nn.Conv2d class로 convolution layer를 생성할 수 있음. 이때 in_channels는 입력 데이터의 channels(filter의 channels), out_channels는 출력 데이터의 channels(filter의 개수), kernel_size는 filter의 크기(ex. 3 또는 (2,3) 등)임. stride는 1, padding은 0, bias는 True가 default임.

가중치와 bias는 내부에서 자체적으로 초기화됨.

이때 입력 데이터로는 channels first인 4차원 tensor(batch_size, channels, height, width)를 넣음. 출력 데이터는 동일하게 channels first인 4차원 tensor가 도출되는데, 이때의 channels는 layer(filter)에 지정한 값이 되고, height/width는 filter size, padding, stride에 의한 값이 됨.

Conv2d에서는 output size가 실수인 경우 소수점 아래 수는 버리는(내림) 식으로 동작함.

```
import torch.nn as nn

conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
                                            padding=0, bias=True)
conv = nn.Conv2d(3, 1, 5, stride=2, padding=0, bias=True)
x = conv(x) # x의 shape은 (batch_size, channel, height, width)
```

아래와 같이 filter를 manual하게 지정하여 feature를 추출하는 것도 가능은 함.

1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

3.1.4. Pooling

1. Pooling
Pooling은 featrue를 요약하면서 크기를 줄이는 연산임. 이는 데이터의 크기를 줄이거나 세부 정보를 제거하는 기능을 함.

pooling에는 max pooling, average pooling 등이 있음. Max Pooling은 filter 내부에서 최댓값을 추출하고, Average Pooling은 filter 내부의 평균값을 추출함.

pooling에 대해서는 filter의 크기와 stride가 같은 것이 일반적임.

pooling에 의한 output size는 convolution과 동일한 방식으로 계산이 가능함. filter의 크기와 stride가 같고 padding이 0이면, pooling은 데이터의 크기를 filter의 크기로 나누는 동작을 함. pooling도 convolution과 마찬가지로 output size를 계산할 수 있음. covolution은 channels를 조정할 수 있었지만, pooling은 기본적으로 channels를 유지하고 height와 width을 줄임.

pooling은 parameter를 가지지 않으므로 backpropagation에 의한 최적화가 수행되지 않음.

2. 코드
pytorch에서는 아래와 같이 nn.MaxPool2d class로 (max) pooling layer를 생성할 수 있음. kernel_size는 영역의 크기, stride는 영역의 이동 간격임. stride는 kernel_size 값, padding은 0이 default임.

이때 입력 데이터로는 channels first인 4차원 tensor(batch_size, channels, height, width)를 넣음. 출력 데이터로는 동일하게 channels first인 4차원 tensor가 도출되는데, 이때 channels는 유지되고 height/width는 filter size, padding, stride에 의한 값이 됨.

```
import torch.nn as nn

pool = nn.MaxPool2d(kernel_size, stride=None, padding=0)
pool = nn.MaxPool2d(3)
x = pool(x)
```

3.2. CNN

3.2.1. CNN

1. CNN
CNN(Convolution Neural Network)은 데이터를 convolution 연산을 활용하는 NN임. CNN은 주로 이미지와 같이 공간적 구조를 가지는 데이터에 사용함.

CNN의 학습도 기존의 NN과 동일하게 수행할 수 있음. 모델을 구성하여 출력 결과를 얻은 뒤 cost function을 호출하고, 역전파에 의해 NN에 존재하는(convolution layer, FC 등) W와 b가 최적화됨.

CNN은 앞쪽 layer에서 뒤쪽 layer로 갈수록 resolution이 작아지고 width(channels)가 커지는 형태로 구성됨. 이에 따라 구체적인 픽셀들로부터 추상적인 데이터(high-level representation)를 추출하게 됨.

2. CNN의 이점
CNN을 적용하는 데이터에 대한 가정은 아래와 같음.

1) Locality of Pixel Dependencies
Locality of Pixel Dependencies는 각 데이터(픽셀)의 종속성이 locality(지역성)을 가진다는 가정임. 즉, 각 데이터는 시간이나 위치에 대해 인접한 데이터들에게만 종속성을 가짐.

2) Stationarity of Statistics
Stationarity(정상성)은 of Statistics는 주로 시간이나 위치에 따른 데이터의 확률 분포가 일정하다는 가정임. 즉, 데이터에 대해서 서로 다른 시간이나 위치에서 동일한 데이터가 등장할 수 있다는 것임. 예를 들어, 사진에서 얼굴은 오른쪽 위에 있을 수도 있고, 왼쪽 아래에 있을 수도 있음.

3) Compositionality
compositionality는 feature들이 합쳐져서 하나의 high-level representation을 나타낼 수 있다는 가정임. 추가로, Manifold hypothesis는 고차원의 데이터가 실제로는 저차원의 manifold 위에 근사적으로 존재

한다는 가정임. 이 두 가정 모두 고차원에서 저차원으로 데이터를 변환하는 식으로 동작하는 CNN의 기반임.

이에 따라 위와 같은 가정을 만족하는 데이터에 대해 DNN과 비교하여 CNN이 가지는 이점은 아래와 같음.

1) 공간적 구조 반영(Spatial Locality 활용)
기존의 NN에서는 기본적으로 1차원 형태의 입력 데이터를 사용하므로 이미지 등의 데이터는 1차원으로 flattern(평탄화)함. 이 경우 데이터의 각 부분에 대한 locality를 활용되지 못하는데, CNN에서는 filter를 사용해 인접한 데이터끼리의 관계를 추출함.

2) Translation Invariance
Translation Invariance(이동 불변성)은 특정 부분이 시간이나 위치가 달라져도 동일하게 학습되는 것을 말함. CNN에서 하나의 filter는 전체 데이터에 대해 적용되므로, 데이터가 서로 다른 시간이나 위치에 있더라도 동일하게 추출됨. 또한 그 형태가 일부 다르더라도 pooling에 의해 동일하거나 유사하게 추출될 수 있음.

3) 사용 가중치 감소
DNN에서와 같이 다차원 데이터를 단순히 flattern하여 학습하면 변수 개수에 따른 가중치가 너무 많아져 연산이 오래 걸리고 overfitting이 발생하기 쉬움. convolution에서는 공유 가중치를 사용하므로 가중치 자체가 훨씬 적고, convolution과 pooling에 의해 데이터를 추출하면 변수를 줄일 수 있음.

3. CNN 구성
CNN은 아래와 같이 구성됨. 이때 각 layer마다의 데이터 shape을 잘 고려해야 함. 헷갈린다면 직접 찍어보자.

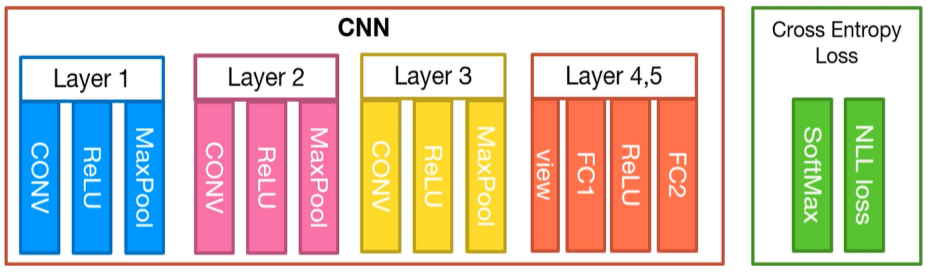
1) convolution을 수행하는 부분
convolution layer, 활성 함수, pooling layer의 순서로 각 layer를 구성함.

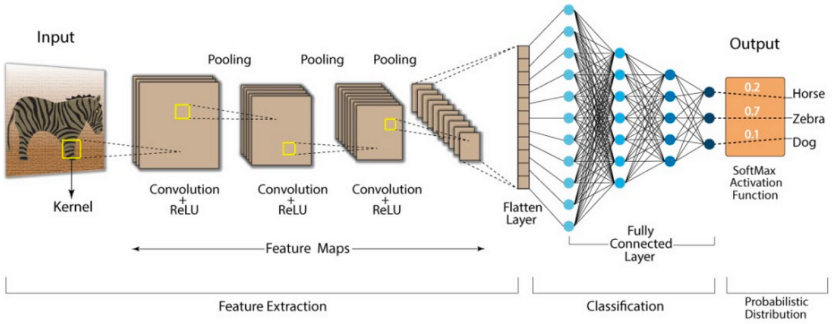
2) FC를 포함하는 부분
convolution의 결과는 주로 다차원이므로 FC에 사용하기 위해 flattern layer를 거쳐 1차원으로 변환한 뒤, FC 및 활성 함수로 구성함.

flattern layer는 아래와 같이 view() 메서드로 간단히 구현할 수 있음.

```
x.view(x.size(0), -1)
```

3) 정답 구성 및 cost function을 계산하는 부분
앞에서 다룬 것처럼 정답을 구성하고(ex. softmax) cost function을 계산함. 이후 backpropagation을 수행함.





이미지 데이터셋은 torchvision의 ImageFolder를 사용해 간단히 불러올 수 있음.

3.2.2. CNN의 활용

본 필기에서는 이미지 데이터의 classification에 적용하는 CNN을 주로 정리하는데, 이외에도 이미지 데이터 등을 활용하는 여러가지 상황과 분야에 CNN을 적용할 수 있음.

아래와 같은 분야들이 존재함.

1) Classification : 주어진 이미지 데이터를 특정 class(label)로 분류하는 것. DenseNet, SqueezeNet, AutoML(NAS, NASNet) 등의 모델이 존재함.

2) Detection : 이미지나 비디오 데이터에서 객체의 위치와 class를 찾는 것.

3) Tracking : 비디오 데이터 등에서 객체의 위치를 지속적으로 추적하는 것. 객체를 detection하고, 각 프레임 간의 연관관계를 추출함. MDNet, ROLO 등의 모델이 존재함.

4) Segmentation : 이미지 데이터 등에서 객체에 해당하는 픽셀을 분리하는 것. FCN, U-Net 등의 모델이 존재함.

5) Image Captioning : 이미지 데이터에 대한 텍스트 설명을 생성하는 것.

6) Super Resolution : 저해상도 이미지 데이터를 고해상도 이미지로 변환하는 것.

7) Generative Model : 새로운 이미지나 비디오 데이터를 생성하는 것.

8) OpenPose : 이미지 데이터 등에서 사람의 자세나 관절 위치를 추출하는 것.

각 분야에서 사용하는 모델은 pytorch 모듈이나 웹에서 쉽게 찾을 수 있음.

3.3. Advanced CNN

3.3.1. Advanced CNN

다양한 CNN의 구현체들을 알아보자. VGG와 ResNet은 뒤에 따로 정리함. 구체적인 구조와 디테일은 필요하면 찾아보자.

1. LeNet
LeNet은 1980년대 CNN 초창기에 개발된 CNN 모델로, MNIST 등 흑백 손글씨 이미지를 학습하는 데에 주로 사용되었음.

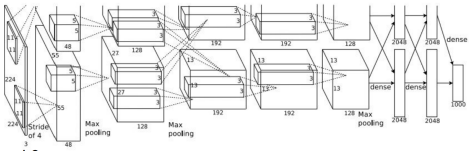
LeNet의 아키텍처는 앞에서 살펴본 기본적인 CNN의 구조와 같음. 여러 겹의 convolution layer와 pooling layer를 거쳐 그 결과를 FC에 넣은 뒤 classification함.

2. AlexNet
AlexNet은 2012년 ILSVRC(ImageNet 데이터셋)에서 등장한 CNN 모델로, DL의 가능성을 보였음.

AlexNet의 아키텍처는 아래와 같음. 여러 겹의 convolution layer와 pooling layer를 거쳐 그 결과를 FC에 넣은 뒤 classification함. 최초로 ReLU를 활성 함수로 사용하였고, 7개의 개별적인 CNN을 구성한 뒤 그 결과를 합치도록(ensemble) 했음.

Case Study: AlexNet

[Krizhevsky et al. 2012]



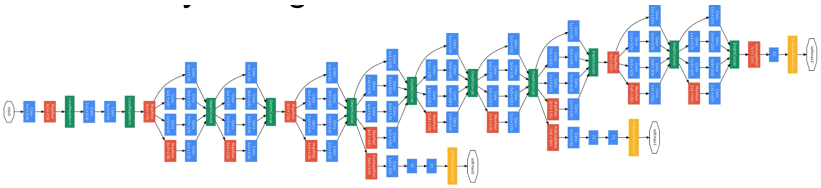
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

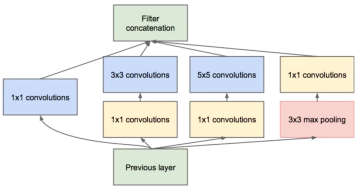
Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

3. GoogLeNet

GoogLeNet은 2014년 google이 ILSVRC(ImageNet 데이터셋)에서 공개한 CNN 모델임.

GoogLeNet 아키텍처는 아래와 같음. 넓고 깊은 CNN을 사용하지만 inception module을 사용해 성능을 개선했음. Inception Module은 서로 다른 크기의 filter를 사용하는 convolution과 pooling을 병렬적으로 적용한 뒤 그 결과를 합치는 구조임.





Inception module

ILSVRC 2014 winner (6.7% top 5 error)

3.3.2. VGG

1. VGG

VGG는 2014년 옥스포드 대학이 ILSVRC에서 공개한 CNN 모델임. 이때 공개한 VGG는 VGG16으로, 뒤에 붙은 숫자는 가중치를 가지는 layer(conv, FC)의 개수를 나타냄. VGG11, VGG19 등도 존재함.

VGG의 아키텍처는 아래와 같음. conv3-256과 같은 표기는 해당 convolution layer에서는 filter의 size가 3이고, channels(depth)가 256이라는 것임. 앞에서 살펴본 CNN 모델들과 같이 여러 겹의 convolution layer와 pooling layer를 거쳐 그 결과를 FC에 넣은 뒤 classification함. 다만 더 규칙적이고 간결한 구조를 가지는데, 각 filter는 size가 3, stride가 1, padding이 1으로 convolution 연산은 데이터의 size를 변경하지 않음. 데이터의 size는 pooling과 FC에 의해서만 변경됨.

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

2. 코드

pytorch에서는 torchvision.models.vgg로 VGG11부터 VGG19까지를 간편하게 구성할 수 있도록 함.

기본적으로 입력은 $3 \times 224 \times 224$인 데이터 집합임.

아래와 같이 각 conv(channels 지정)와 pooling 위치로 dictionary를 구성하여 make_layer()에 넣으면 conv 부분의 layer를 생성할 수 있음. 이를 VGG()에 넣어 모듈을 구성함. 이때 생성된 모델의 각 filter는 크기가 3이고, 모델에 넣을 수 있는 각 입력 데이터는 $(3 \times 224 \times 224)$임. num_classes에는 classification의 가짓수를 지정함.

```
cfgs: Dict[str, List[Union[str, int]]] = {
    "myconv11": [64, "M", 128, "M", 256, 256, "M", 512, 512,
                              "M", 512, 512, "M"],
    "myconv16": [64, 64, "M", 128, 128, "M", 256, 256, 256,
                              "M", 512, 512, 512, "M", 512, 512, 512, "M"]
}

conv_layers = make_layers(cfgs["myconv16"], batch_norm=False)
model = VGG(conv_layers, num_classes=1000)
```

torchvision.models.vgg에서 확인할 수 있는 구체적인 코드는 아래와 같음. 필요하다면 이 코드를 상황에 맞게 수정하여 사용하자.

```
    class VGG(nn.Module):
        def __init__(
            self, features: nn.Module, num_classes: int = 1000,
                            init_weights: bool = True, dropout: float = 0.5
        ) -> None:
            super().__init__()
            _log_api_usage_once(self)
            self.features = features
            self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
            self.classifier = nn.Sequential(
                nn.Linear(512 * 7 * 7, 4096),
                nn.ReLU(True),
                nn.Dropout(p=dropout),
                nn.Linear(4096, 4096),
                nn.ReLU(True),
                nn.Dropout(p=dropout),
                nn.Linear(4096, num_classes),
            )
            if init_weights:
                for m in self.modules():
                    if isinstance(m, nn.Conv2d):
                        nn.init.kaiming_normal_(m.weight, mode="fan_out",
                                                nonlinearity="relu")
                        if m.bias is not None:
                            nn.init.constant_(m.bias, 0)
                    elif isinstance(m, nn.BatchNorm2d):
                        nn.init.constant_(m.weight, 1)
                        nn.init.constant_(m.bias, 0)
                    elif isinstance(m, nn.Linear):
                        nn.init.normal_(m.weight, 0, 0.01)
                        nn.init.constant_(m.bias, 0)

        def forward(self, x: torch.Tensor) -> torch.Tensor:
            x = self.features(x)
            x = self.avgpool(x)
            x = torch.flatten(x, 1)
            x = self.classifier(x)
            return x
```

```
def make_layers(cfg: List[Union[str, int]],
                           batch_norm: bool = False) -> nn.Sequential:
    layers: List[nn.Module] = []
    in_channels = 3
    for v in cfg:
        if v == "M":
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            v = cast(int, v)
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

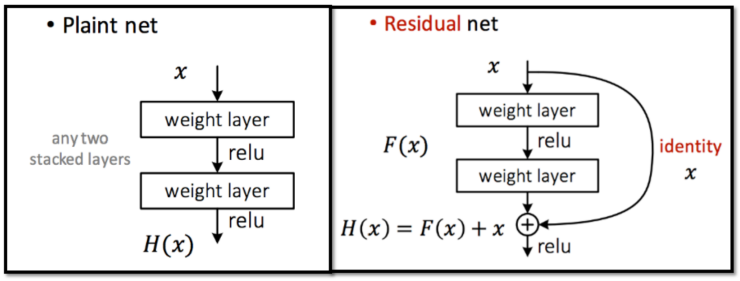
3.3.3. ResNet

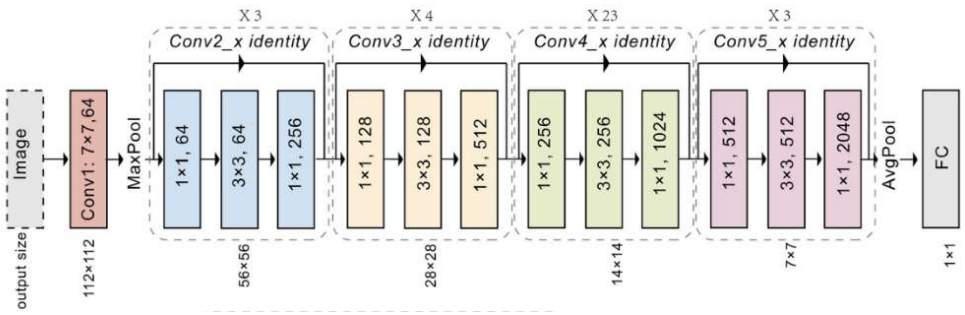
1. ResNet

ResNet은 2015년 마이크로소프트에서 공개한 CNN 모델임. 가중치를 가지는 layer(conv, FC)의 개수에 따라 ResNet18, ResNet50, ResNet152 등이 있음.

ResNet의 아키텍처는 아래와 같음. ResNet은 총 4개의 layer를 가지고, 각 layer는 여러 개의 residual block들로 구성되어 있음. Residual Block은 weight layer를 통과한 값과 통과하지 않은 값의 합을 출력 값으로 하는 layer 묶음임. 또한 이런 연결을 Residual Connection이라고 함.

VGG 등에 비해 ResNet은 훨씬 깊은데, 이는 ResNet이 residual block을 사용하기 때문에 가능함. 기존의 CNN은 너무 깊게 쌓으면 gradient가 소실되는 등의 문제가 존재하지만, residual block을 사용하면 feature가 쉽게 유실되지 않아 모델이 깊어져도 원활한 학습이 가능함. 이때의 이런 연결을 Skip Conncection이라고 함.





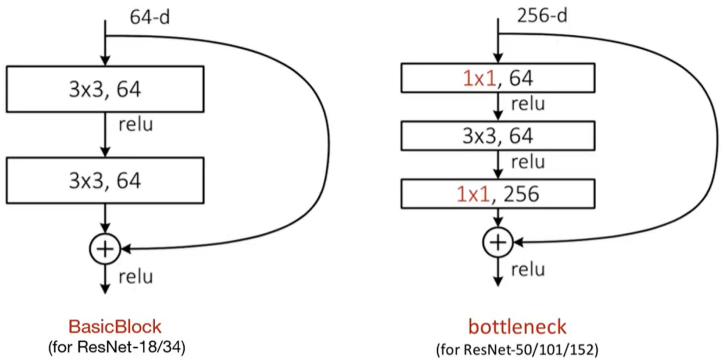
2. Basic block vs. Bottleneck block

residual block에는 basic block과 bottleneck block 등이 있음. ResNet은 basic block들 또는 bottleneck block들로 구성됨.

Basic block은 3×3 filter를 가지는 2개의 convolution layer로 구성된 단순한 residual block임. 이때 padding은 1, stride는 1인 것이 기본으로, 이러면 데이터의 size가 유지됨.

Bottleneck block(Bottleneck)은 1×1 filter를 가지는 2개의 convolution layer와, 그 사이의 3×3 filter를 가지는 하나의 convolution layer로 구성된 residual block임. 1×1 filter는 padding이 0, stride가 1이고, 3×3 filter는 padding이 1, stride가 1인 것이 기본으로, 이러면 데이터의 size가 유지됨.

bottleneck에서는 $n \times 1 \times 1$ filter를 사용하는 convolution layer를 사용하는데, 이를 통해 데이터의 크기를 유지하면서 channels(depth)를 n으로 지정할 수 있음. 특히, channels를 줄인 뒤 다른 convolution layer를 쌓고 다시 channels를 늘리면, 단순히 하나의 convolution layer만을 사용할 때보다 parameter(연산량)이 대폭 감소함.



3. 코드

pytorch에서는 torchvision.models.resnet으로 ResNet을 간편하게 구성할 수 있도록 함.

기본적으로 입력은 $3 \times 224 \times 224$인 데이터 집합임.

아래와 같이 ResNet을 구성할 수 있음. 첫 번째 인자로는 ResNet의 각 block을 basic block으로 할 것인지, bottleneck으로 할 것인지 지정하고, 두 번째 인자로는 각 layer가 몇 개의 block을 가질 것인지 지정하고, 세 번째 인자로는 classification의 가짓수를, 네 번째 인자로는 초기화 기법 적용 여부를 지정할 수 있음.

```
import torchvision.models.resnet as resnet

resnet50 = ResNet(resnet.Bottleneck, [3, 4, 6, 3], 10, True)
```

torchvision.models.resnet에서 구체적인 구현 코드를 확인할 수 있음. 필요하다면 이 코드를 상황에 맞게 수정하여 사용하자. 코드가 길어서 여기에 첨부하지는 않음.

*코드에서 expansion은 input channels에 곱해서 ouput channels를 결정하는 값임. basic block은 channels가 변하지 않으므로 1이고, bottleneck은 channels가 4배가 되므로 4임. 또한 참고로, planes * block.expansion는 해당 block의 output channels임.*

stride가 1이면 block을 거치는 동안 input size와 output size는 동일하게 유지되지만, 2, 3, 4번째 layer의 첫 번째 block에서는 stride가 2로 지정되므로 output size가 감소함.

basic block을 사용하든 bottleneck을 사용하든 convolution layer를 통과한 데이터와 skip connection으로 넘어온 데이터의 shape이 다른 경우, 코드에서는 downsample이라는 layer를 skip connection에 추가로 사용해 이를 맞춤. 특히, stride가 1이 아니거나, bottleneck의 경우 channels가 달라지므로 downsample을 사용해야 함. 코드를 보면 각 layer에서 첫 번째 block에는 downsample이 적용되는 것을 알 수 있음.

사실 이런 기법과 모델이 정확히 왜 더 뛰어난지 완전히 이해하지는 못하고 있음.

4. RNN

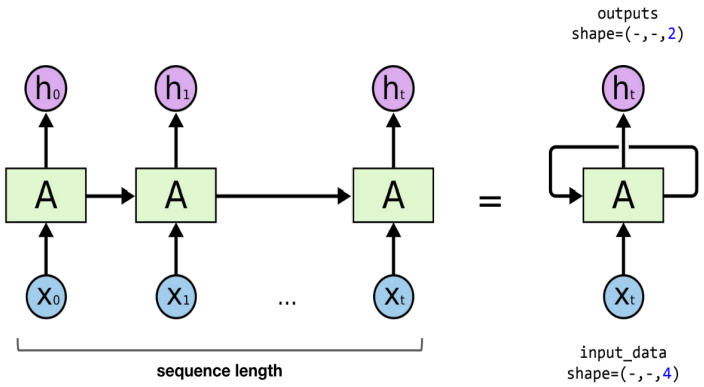
4.1. RNN

4.1.1. Recurrent Layer

1. 셀

셀(Cell, Memory Cell, RNN Cell)은 아래와 같이 연산 결과를 출력 방향뿐만 아니라, 스스로에게 전송하여 다음 연산에 사용하는 RNN의 기본 구성 요소임.

셀에서는 이전에 수행한 연산 결과를 사용하므로 입력 순서가 결과에 반영됨. 셀에서 다음 셀로 전달되는 값은 출력층으로 드러나지 않으므로 Hidden State라고 하고, 그 크기를 Hidden Size라고 함.

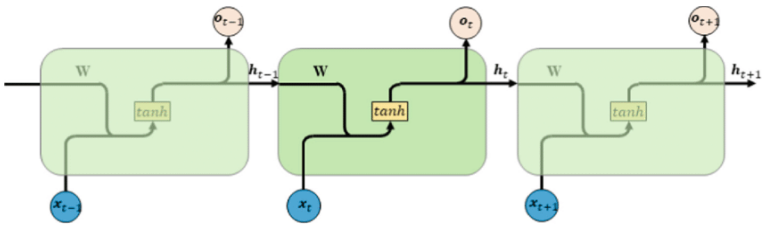


셀은 나름의 W와 b, input과 hidden state를 사용해 연산을 수행함. 셀에서 수행되는 연산을 수식으로 표기하면 아래와 같음. x_t는 입력, h_{t-1}은 hidden state, h_t는 새로운 상태(출력), f_W는 셀에서 수행하는 연산 함수임.

$$h_t = f_W(h_{t-1}, x_t)$$

Vanilla RNN은 가장 기본적인 형태의 셀을 사용하는 RNN임. vanilla RNN의 셀에서 수행되는 연산은 아래와 같음. h_t는 해당 시점에서의 hidden state, y_t는 실질적인 출력값임. 수식에서 확인할 수 있는 것처럼, 총 3가지 가중치가 사용됨. hyperbolic tangent가 사용되는데, 이 함수의 개형은 sigmoid와 거의 유사함. 참고로, 맨 처음의 계산에서는 h_{t-1}의 값을 주로 0으로 함.

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t), \;\; y_t = W_{hy}h_t$$



(a) Vanilla RNN layer

물론 vanilla RNN 외에도 LSTM, GRU 등 다양한 형태의 RNN이 존재함. vanilla RNN은 exploding 및 vanishing gradient problem이 존재하는데, LSTM과 GRU로 이를 어느 정도 해결할 수 있음.

2. Recurrent Layer

Recurrent Layer는 RNN에서 셀로 구성된 layer임.

recurrent layer에서는 동일한 셀로 연산 결과를 전송하므로 시간순으로 표기하면 여러 개의 동일한 셀들의 집합으로 나타낼 수 있는데, 실제로는 hidden state 값을 활용하여 셀이 반복 호출되는 것임. 실질적으로는 하나의 셀이 사용되는 것이므로, 한 세트의 parameter(가중치)만이 존재함. 즉, 긴 sequence가 들어와도 적은 수의 parameter로 학습이 가능함.

3. 코드
pytorch에서 recurrent layer는 아래와 같이 nn.RNN을 사용해 생성할 수 있음. nn.RNN에서는 batch_first=False가 default이지만, 여기에서는 True인 것으로 가정하고 정리함.

이때 입력 데이터로는 3차원 tensor(batch_size, Sequence_length, Input_size)를 넣음(2차원인 단일 batch 데이터도 사용 가능함.). 이때 squence length는 시간에 따라 셀에 들어가는 입력의 총 개수이고, input size는 각 셀에 들어가는 입력의 길이임.

출력 데이터는 3차원 tensor(batch_size, Sequence_length, Hidden_size)임. 이때 hidden size는 hidden state의 길이임. vanilla RNN의 경우 출력 직전에 값이 분기되므로, hidden size는 output size과 동일함. 정리하면, 하나의 batch에서 데이터는 각 시간대마다 행 별로 셀에 입력되서 hidden size 만큼의 데이터로 출력되는 것임.

hidden state는 3차원 tensor(Num_layers, Batch_size, Hiddens_size)임. num layers는 해당 recurrent layer의 개수임. 이는 RNN 객체 생성 시에 지정한 batch_first와는 상관없이 shape이 결정됨.

반환값 중 첫 번째(outputs)는 전체 출력값을 모은 tensor이고, 두 번째(status)는 마지막 셀에서의 hidden state(=출력값)임.

인자에 num_layers를 지정하여 원하는 만큼의 recurrent layer를 쌓을 수 있음.

```
import torch.nn as nn

rnn1 = nn.RNN(input_size, hidden_size, batch_first=True)
rnn2 = nn.RNN(input_size, hidden_size, num_layers=2, batch_first=True)
outputs, status = rnn1(input_data)
```

4.1.2. RNN

1. RNN
RNN(Recurrent Neural Network, 순환 신경망)은 시간적 순환 구조를 가지는 셀(recurrent layer)들로 구성된 NN임.

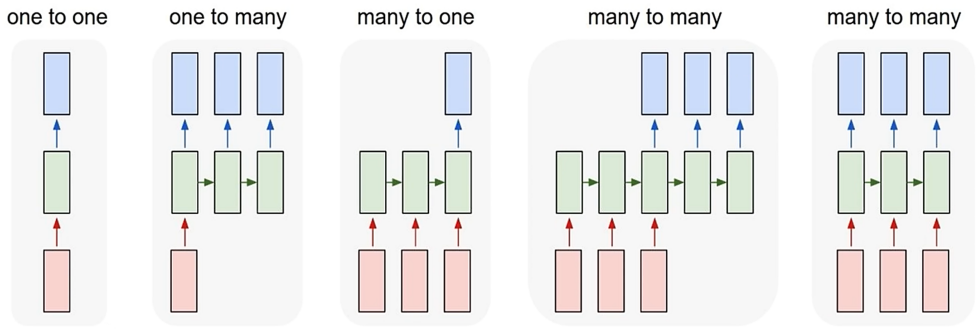
RNN은 sequential 데이터를 처리하는 것을 그 목적으로 함. sequential 데이터는 순서가 중요한 데이터로, 내부적인 순서도 데이터의 일부인 것임. 대표적으로 자연어, 음성 및 비디오 데이터 등이 있음.

RNN의 학습도 기존의 NN과 동일하게 수행할 수 있음. 모델을 구성하여 출력 결과를 얻은 뒤 cost function을 호출하고, 역전파에 의해 NN에 존재하는(recurrent layer, FC 등) W와 b가 최적화됨.

또한 RNN, transfomer 등에서는 입력 데이터를 처리 가능한 단위로 나눠서 처리하는데, 이때의 각 요소를 Token(토큰)이라고 함.

2. RNN 구성
RNN은 사용하는 목적에 맞게 다양한 형태로 그 구조를 구성할 수 있음.

입력과 출력의 개수에 따라 one to many, many to one, many to many 등의 구조를 가질 수 있음. 예를 들어, 아래의 그림에서 one to many는 image captioning 등에, many to one은 classification등에, many to many는 translation과 video 처리 등에 사용됨.



또한 필요 시에 recurrent layer를 여러 개 쌓거나 FC와 softmax 등을 사용할 수 있음.

recurrent layer에서는 기본적으로 출력값과 hidden state의 shape이 동일한데, 원하는 shape의 출력값을 위해 hidden state까지 압축/확장하는 것은 적절하지 않을 수 있음. 이에 따라 recurrent layer들을 거친 이후에 FC layer를 사용해 크기를 맞추는 식으로 모델을 구성함.

3. RNN 활용
RNN은 아래와 같이 sequential 데이터를 처리하는 분야 등에 적용이 가능함.

1) Language Modeling : 주어진 텍스트 데이터에서 단어나 문장의 분포를 학습하여 텍스트를 생성하거나 문맥을 이해하는 것.

2) Machine Translation : RNN을 활용한 언어 번역.

3) Conversation Modeling : 문맥에 맞는 응답을 생성하는 것.

4) Image Captioning : 이미지 데이터에 대한 텍스트 설명을 생성하는 것.

물론 NN에서도 순서 정보를 feature로 해서 sequential 데이터를 학습할 수는 있지만, 순서에 따른 데이터의 의존성을 나타내야 하고, 많은 parameter를 사용해야 하므로 효과적이지 않음.

4.2. RNN 구성 예시

상황에 따라 어떤 식으로 RNN을 구성할 수 있는지 알아보자.

4.2.1. 문자열 데이터 처리

문자열(텍스트) 데이터를 처리하는 RNN 모델은 아래와 같이 구성할 수 있음.

1. 짧은 문자열 데이터
짧은 문자열 데이터에 대해서는 batch당 하나의 문자열을 처리하도록 하고, 각 문자 하나씩을 셀에 입력하여 그 출력으로는 바로 다음에 올 문자를 예측하도록 RNN을 구성함.

이렇게 텍스트 데이터 등을 처리할 때 입력에 대한 바로 다음 값을 예측하는 방식을 Auto-regressive 라고 함.

각 문자에 숫자를 부여하고 one-hot 벡터로 변환하여 입력으로 사용함.

코드는 아래와 같음. 데이터를 구성하는 부분을 제외하고는 기존 NN의 코드와 동일하므로 생략함.

```
sample = "wow it's amazing!"

# data
data_list = list(set(sample)) # data에서 유일한 문자들의 list 추출
data_dic = {d:i for i, d in enumerate(data_list)} # 문자 별로 인덱스 지정

data_dic_rev = {i:d for i, d in enumerate(data_list)}

input_size = len(data_dic)
hidden_size = len(data_dic)

data = [data_dic[c] for c in sample] # data_src를 숫자로 변환
x_train = torch.FloatTensor(data[:-1])
y_train = torch.LongTensor(data[1:])

eye_matrix = torch.eye(len(data_list))
x_one_hot = eye_matrix[x_train.long() - 1]

# model
rnn = nn.RNN(input_size, hidden_size)

# learning
...
for epoch in range(1, epochs + 1) :
    ...
    rst_str = sample[0] + ''.join([data_list[n] for n in prediction])
```

2. 긴 문자열 데이터

긴 문자열 데이터에 대해서는 batch당 부분 문자열 하나를 처리하도록 함. 이때 부분 문자열은 전체 문자열에 대해 지정한 길이의 window에 해당하는 부분임. 즉, 각 부분 문자열을 하나의 batch로 해서 짧은 문자열에서와 동일하게 학습함. 이때 부분 문자열을 셀에 입력하면 그 출력으로는 바로 다음으로 window에 해당되는 부분 문자열을 예측하도록 RNN을 구성함.

각 문자에 숫자를 부여하고 one-hot 벡터로 변환하여 입력으로 사용함. 이때 입력은 3차원 tensor, 출력은 2차원 tensor이므로 cross_entorpy() 사용하는 경우 그 구조를 변경해야 함.

코드는 아래와 같음. 2개의 recurrent layer와 linear layer를 사용해 모델을 구성했음.

```
sentence = ("if you want to build a ship, don't drum up people together to "
"collect wood and don't assign them tasks and work, but rather "
"teach them to long for the endless immensity of the sea.")

sequence_length = 10
char_list = list(set(sentence))
char_dic = {d:i for i, d in enumerate(char_list)}
dic_size = len(char_dic)

# data setting
x_data = []
y_data = []
for i in range(0, len(sentence) - sequence_length) :
    x_str = sentence[i : i + sequence_length]
    y_str = sentence[i + 1 : i + sequence_length + 1]
    x_data.append([char_dic[c] for c in x_str]) # x_data list에 추가
    y_data.append([char_dic[c] for c in y_str]) # y_data list에 추가

x_train = torch.LongTensor(x_data)
y_train = torch.LongTensor(y_data)

eye_matrix = torch.eye(dic_size)
x_one_hot = F.one_hot(x_train, num_classes=dic_size).float()
```

```
# model
class LongCharModel(nn.Module) :
    def __init__(self, input_size, hidden_size, num_layers) :
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers=num_layers)
        self.fc = nn.Linear(hidden_size, hidden_size)

    def forward(self, x) :
        x, status = self.rnn(x)
        x = self.fc(x)
        return x

model = LongCharModel(dic_size, dic_size, 2)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

```
for epoch in range(1, epochs + 1) :
    # hypothesis
    hypothesis = model(x_one_hot)

    # cost
    cost = F.cross_entropy(hypothesis.view(-1, dic_size), y_train.view(-1))

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

    # rst
    prediction = hypothesis.argmax(dim=2)
    rst_str = "" + sentence[0]
    for index, str in enumerate(prediction) :
        if index == 0 :
            rst_str += ''.join([char_list[n] for n in str])
        else :
            rst_str += ''.join([char_list[n] for n in str])[-1]

    if epoch % 10 == 0 :
        print('Epoch {:4d}/{}, Cost: {:.6f}\nExpected string: {}'
                        .format(epoch, epochs, cost.item(), rst_str))
```

4.2.2. Time Series 데이터 처리

Time Series 데이터 처리
times series 데이터를 처리하는 RNN 모델을 구성할 수 있음. Times Series(시계열) 데이터는 일정한 시간 간격으로 배치된 데이터를 말함.

여기에서는 many to one 모델로 time series 데이터를 처리함. 이때 many to one은 단순히 마지막 출력값을 사용하도록 구성할 수도 있지만, 이렇게 구성하면 hidden state의 shape이 출력값의 shape에 따라 제한됨. 이에 따라 recurrent layer의 모든 출력값을 사용하고, recurrent layer 뒤에 FC layer를 거치도록 구성함.

2. 코드
아래와 같이 구성할 수 있음. 여기에서는 데이터를 여러 batch로 쪼개서 사용함.

```
# data
...
x_train, y_train = set_xy_from_data(train_set)
# 각각 (batch, 10, 5), (batch, 10, 1)

# model
class StockDataModule(nn.Module) :
    def __init__(self, input_size, hidden_size, output_size, num_layers) :
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers=num_layers)
        self.fc = nn.Linear(hidden_size, output_size, bias=True)

        nn.init.xavier_uniform_(self.fc.weight)

    def forward(self, x) :
        x, status = self.rnn(x)
        x = self.fc(x)
        return x
model = StockDataModule(data_dim, hidden_dim, output_dim, num_layers=1)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# train
model.train()
for epoch in range(1, epochs + 1) :
    # hypothesis
    hypothesis = model(x_train)

    # cost
    cost = F.mse_loss(hypothesis, y_train)

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

    if epoch % (epochs / 10) == 0 :
        print('Epoch {:4d}/{}, Cost: {:.6f}'
                        .format(epoch, epochs, cost.item()))
```

4.3. Seq2Seq

4.3.1. Word Embedding

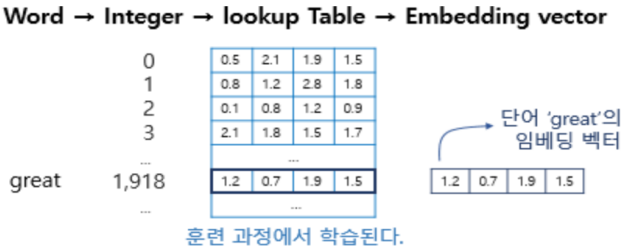
1. Embedding
Embedding 데이터를 벡터화하는 기법 또는 그 결과를 말함. 즉, 데이터를 digitalize함. embedding으로는 one-hot encoding, word embedding, BERT 등이 있음.

2. Word Embedding
Word Embedding은 정수형 label로 구성된 벡터를 dense한 실수형 벡터로 변환하는 기법 또는 그 결과임. 주로 NLP에서 문장의 각 단어를 벡터화할 때 사용함.

앞에서는 정수형 label 벡터를 one hot encoding을 적용해 벡터화했지만, one hot encoding은 차원이 크고 매우 sparce한 결과를 도출하므로 비효율적일 수 있음. 반면에 word embedding은 차원이 작고 dense한 실수형 결과를 도출함.

3. 변환 방식

Embedding Table은 입력 label 벡터가 가질 수 있는 label의 가짓수를 행으로, embedding(출력) 벡터의 크기를 열로 하는 행렬임. embedding table에는 각 label과 그에 대응는 embedding 벡터가 정의되어 있음. embedding table은 embedding layer의 parameter로, backpropagation 시에 최적화됨.



word embedding은 embedding table에서 입력 label 벡터의 각 label에 해당하는 embedding 벡터를 찾아 반환하는 식으로 동작함.

4. 코드

pytorch에서는 아래와 같이 nn.Embedding으로 word embedding layer를 생성할 수 있음. 첫 번째 인자(num_embedding)로는 embedding table의 행(입력 벡터의 크기. ex. 문장에서 단어의 개수)을, 두 번째 인자(embedding_table)로는 embedding table의 열(embedding 벡터의 크기)을 지정함.

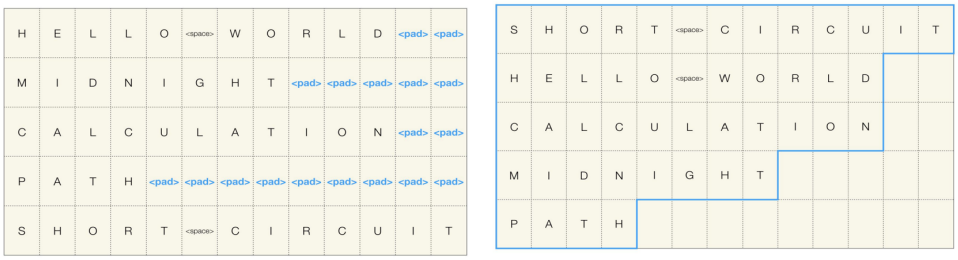
```
import torch.nn as nn

embed = nn.Embedding(vocab_size, embedding_dim)
```

참고로 pytorch에서는 Word2Vec, GloVe 등 사전 훈련된 embedding table을 가져와 사용할 수도 있음.

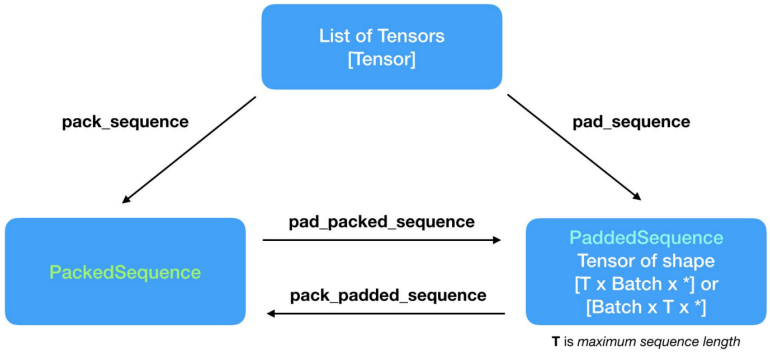
4.3.2. Padding/Packing

padding과 packing은 서로 다른 길이를 가지는 sequential 데이터를 처리하는 기법들임. sequential 데이터는 길이가 제각각이거나 가변적인 경우가 많으므로 적절한 처리가 필요함.

Padding은 가장 긴 sequence를 기준으로 나머지 sequence에 Pad(주로 0)를 추가하는 방식임. padding은 구현이 간단하지만 pad가 많아질수록 계산할 필요가 없는 부분을 계산하게 됨. Packing은 pad를 추가하는 대신 각 sequence의 길이를 저장하고 활용하는 방식임. packing은 구현이 비교적 복잡함.



pytorch에서는 아래의 그림과 같이 4개의 함수를 사용해 기본 tensor, packed sequence, padded sequence 사이에서 상태를 변환할 수 있음. 참고로, pytorch에서 packing을 하려면 데이터가 sequence의 길이에 따라 내림차순으로 정렬되어 있어야 함.

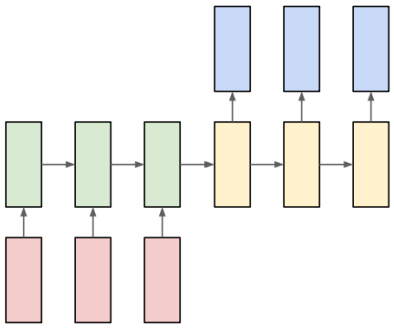


4.3.3. Seq2Seq

1. Seq2Seq

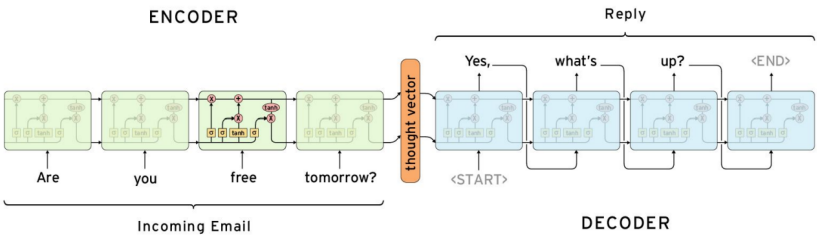
Seq2Seq는 sequence를 입력받아 sequence를 출력하는 RNN 모델임. 주로 번역, 챗봇, 음성 인식 등에 사용됨.

단순한 many-to-many RNN은 입력 sequence와 출력 sequence의 길이가 다른 경우를 처리하기 어려움. 또한 이 경우 출력 sequence가 입력 sequence 전체가 아니라 앞쪽 부분만을 반영하는 결과가 도출됨. seq2seq에서는 입력이 종료된 이후 출력하므로 입력 sequence 전체에 대한 출력 sequence를 내보낼 수 있음.



Seq2Seq

seq2seq는 encoder와 decoder로 구성됨. 이 두 부분은 단순히 셀을 연결한 것임. Encoder에서는 입력을 받아 처리하고, 최종 결과를 고정된 길이의 벡터로 decoder의 hidden state로 전달함. Decoder에서는 출력을 내보냄. 이때 주로 decoder의 첫번째 셀에는 Start Flag를 입력으로 넣고, 그 이후의 셀에는 직전 셀의 출력을 입력으로 넣는 식으로 구현됨.



2. 코드

pytorch에서는 아래와 같이 seq2seq를 구현할 수 있음. 코드가 길어서 모델만 첨부함.

여기에서 데이터 구성 시에는 각 단어를 labeling했고, 문장 별로 batch를 구성했음. 또한 각 batch끼리의 sequence별 길이를 맞추기 위해 padding을 추가했음. 각 문자는 embedding을 적용했음.

아래의 코드에서와 달리 sequence의 길이가 미리 정해져 있는 경우가 아니라면, 각 sequence의 마지막에 end token을 붙여서 학습함. 이후 decoder에서는 end token이 나올 때까지 출력하도록 함. 또한 maximum 길이를 정의해 두고, 그 길이까지 end token이 출력되지 않으면 끝나도록 함.

decoder의 입력 데이터는 teacher focring을 적용해 구성했음. Teacher Forcing은 seq2seq의 encoder에서 각 출력값 대신 실제 정답을 다음 시점의 입력으로 사용하는 기법임. teacher forcing을 사용하면 구현이 간단하고, 학습 초기에 왜곡된 출력을 입력으로 사용하는 상황을 예방할 수 있음. 이때 첫 번째 입력으로는 SOS(Start Of Sentence) 토큰(주로 0으로 구성된 tensor)을 넣음.

여기에서는 기존 방식대로 전체 tensor를 통째로 모델에 넣는 식으로 구현했는데, batch별로 각 2차원 tensor에 대해 반복문을 돌려 따로 처리할 수도 있음.

```
# model
class MyEncoder(nn.Module) :
    def __init__(self, num_embedding, hidden_size) :
        super().__init__()
        self.embed = nn.Embedding(num_embedding, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)

    def forward(self, x) :
        x = self.embed(x) # (batch(문장 개수), input_size, hidden_size)
        _, hidden = self.gru(x)
        return hidden
```

```
class MyDecoder(nn.Module) :
    def __init__(self, num_embedding, hidden_size, output_label_size) :
        super().__init__()
        self.embed = nn.Embedding(num_embedding, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_label_size)

        self.output_label_size = output_label_size

    def forward(self, y, hidden_state) :
        y = self.embed(y)

        sos = torch.zeros(y.size(0), 1, y.size(2))
        y = y[:, :-1, :]
        y = torch.cat((sos, y), dim=1)

        x, _ = self.gru(y, hidden_state)

        x = self.linear(x)
        x = x.view(-1, x.size(2))

        return x
```

```
class Seq2Seq(nn.Module) :
    def __init__(
            self,
            encoder_embedding,
            decoder_embedding,
            hidden_size,
            input_size
            ) :
        super().__init__()
        self.input_size = input_size
        self.output_size = output_size
        self.encoder = MyEncoder(encoder_embedding, hidden_size)
        self.decoder = MyDecoder(decoder_embedding, hidden_size,
                                                    decoder_embedding)

    def forward(self, x, y) :
        hidden = self.encoder(x)
        x = self.decoder(y, hidden)
        return x
```

5. Transformer

5.1. Attention Mechanism

5.1.1. Attention Mechanism

1. Attention Mechanism

Attention Mechanism(Attention)은 decoder에서 encoder로부터 넘어온 벡터뿐만 아니라, encoder에서의 모든 hidden state를 확률적으로 사용하는 기법임. 여기에서는 seq2seq2에 적용하는 상황을 기반으로 정리함.

단순 RNN에서는 연산에 따라 gradient vanishing problem이 발생할 수 있으므로 LSTM/GRU를 사용했고, 이후 sequential 데이터의 더 나은 처리를 위해 seq2seq가 등장했음. 하지만 LSTM/GRU를 사용한 seq2seq 모델에서도 입력을 고정된 길이의 벡터로 압축하므로, gradient vanishing problem이 완전히 해결되지는 못했음. 즉, long-range dependency를 가지는 데이터를 처리하는 데에 한계가 존재했음. 이런 이유 등에 의해 기존 RNN에 attention을 적용한 attention model이 등장함.

또한 attention을 적용하면 학습 중간에 각 hidden state에 대한 확률 값 출력 및 시각화할 수 있으므로, 학습 과정을 파악하기에 용이함.

2. Attention Function

attention은 attention function으로 구현됨. Attention Function은 현재 상태(값)인 Query, 참조할 대상인 Key, attention value 구성 시에 실질적으로 사용할 값인 Value를 입력으로 받아 Attention Value를 반환하는 함수임.

Q, K, V를 유튜브에 비유하면, Q는 검색창에 작성한 텍스트이고, K는 비디오의 제목 등이고, V는 Q와 K의 비교를 통해 검색된 비디오임.

attention function에 대한 수식은 아래와 같음. 왜 이런 식으로 구현되는지는 seq2seq에서의 동작을 보면 쉽게 이해할 수 있음. 이때 $\sqrt{d_k}$는 normalization(scaling)을 위한 것으로, 곱 연산에 의해 값이 과하게 커지는 것을 방지함.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V = attention\ value$$

이때 query와 key는 서로 비교(similarity 계산) 가능해야 하므로 주로 동일한 shape을 가짐. 또한 query와 attention value는 동일한 shape을 가짐. 대부분의 경우 query, key, value, attention value는 동일한 shape을 가짐.

3. Attention을 적용한 seq2seq

seq2seq에 attention을 적용하는 경우, query는 decoder가 가지는 특정 시점의 hidden state, key와 value는 encoder가 가지는 전체 hidden state 각각임. 물론 어떤 모델에서는 key와 value가 서로 다른 값으로 설계될 수 있음.

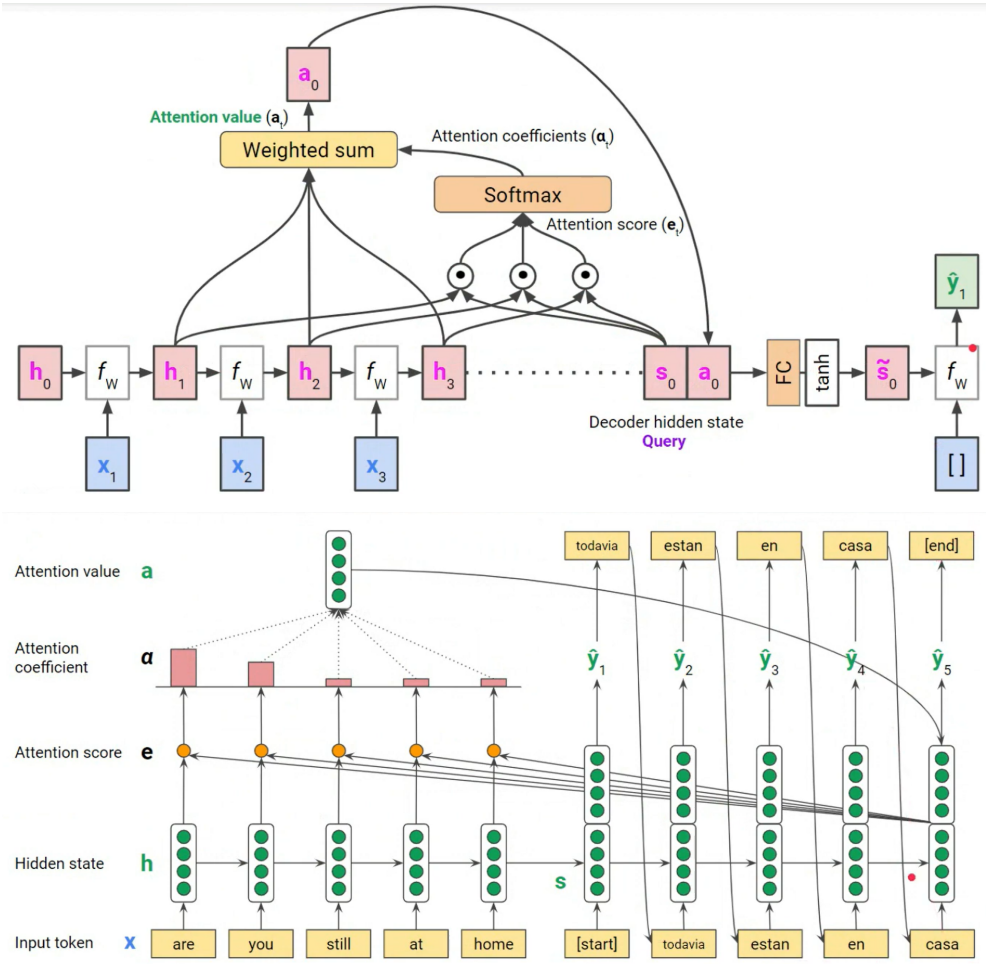
decoder에서 특정 시점의 hidden state에 대해 아래와 같이 attention function을 적용해 attention value를 계산하고, 이를 반영함.

1) decoder의 해당 시점 hidden state(query)와 encoder의 각 hidden state(key) 사이의 similarity(Attention Score)를 계산함. 이는 주로 dot product로 사용해 계산함. 물론 다른 계산 방법도 존재하지만, dot product가 가장 간단하고 성능도 나쁘지 않다고 함.

2) encoder의 hidden state마다 존재하는 attention score 전체에 대해 softmax 함수를 적용해 각 hidden state에 대한 확률분포를 얻음. 이 확률은 similarity에 대한 확률임.

3) encoder의 hidden state(value)와 각각에 대응되는 확률을 곱하고 전부 더해 attention value를 도출함.

4) attention value와 decoder의 해당 시점 hidden state를 연결하고(크기가 2배가 됨.), 이를 FC에 넣어 원래 hidden state의 길이로 다시 압축한 뒤 활성 함수 등을 거쳐 다음 셀로 전달함.



5.2. Transformer

5.2.1. Transfomer layer

1. Transformer layer

Transformer Layer는 self-attention을 통한 token의 contextualize를 수행하는 layer임.

2. Self-attention

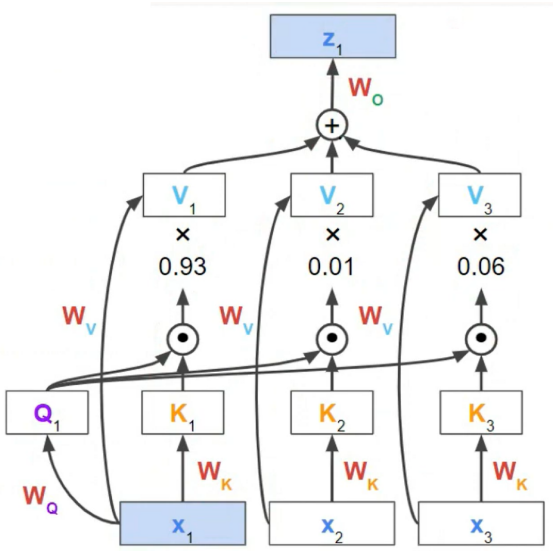
transformer layer에서는 입력을 token으로 분해하고, 각 token들 사이의 관계를 self-attention으로 반영하여 출력함.

Self-attention은 각 token을, 자신을 포함한 전체 token들에 대한 attention으로 재구성하는 기법임. 즉, 특정 구성요소는 자신을 포함한 모든 구성요소를 사용해 표현됨. self-attention은 아래와 같은 과정을 거쳐 수행됨.

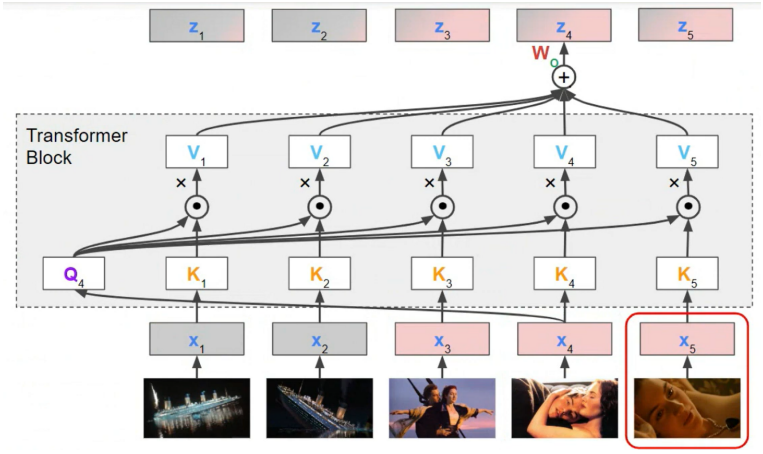
1) 각 token x_i에 대해 가중치 W_Q, W_K, W_V를 곱해 Q_i, K_i, V_i를 생성함. 이때의 W_Q, W_K, W_V는 모든 입력에 대해 공유되는 가중치로, backpropagation에 의해 최적화됨. 또한 생성된 Q_i, K_i, V_i의 크기는 가중치의 크기에 따라 결정되는데, 이는 입력의 길이와 다를 수 있음.

2) 특정 token x_j에 대해 attention을 적용함. 즉, Q_j과 각각의 K_i에 대해 similarity를 dot product로 계산하고, softmax를 거쳐 확률로 변환함. 각 확률과 V_i를 곱한 뒤 모두 더해 해당 token x_j에 대한 attention value를 생성함.

3) 생성된 attention value에 가중치 W_O를 곱해 입력의 크기와 맞추면 token x_j에 대한 결과인 z_j가 도출됨.



각 token은 자기 자신과 높은 similarity를 가지기 때문에 z_j는 x_j와 유사한 값을 가짐. 이런 특성을 시각화하면 아래와 같음. 이때 각 token에는 자신의 값에 전체 sequence에 대한 context(맥락)이 반영되므로 이 과정을 Contextualize라고 함.



모든 token에 대해 이를 수행하여 도출한 결과는 입력과 동일한 shape을 가짐. 이와 같이 transformer layer에서는 token의 개수와 길이를 보존하면서 벡터를 변환하기 때문에 명칭이 transformer임.

3. Transformer layer의 학습

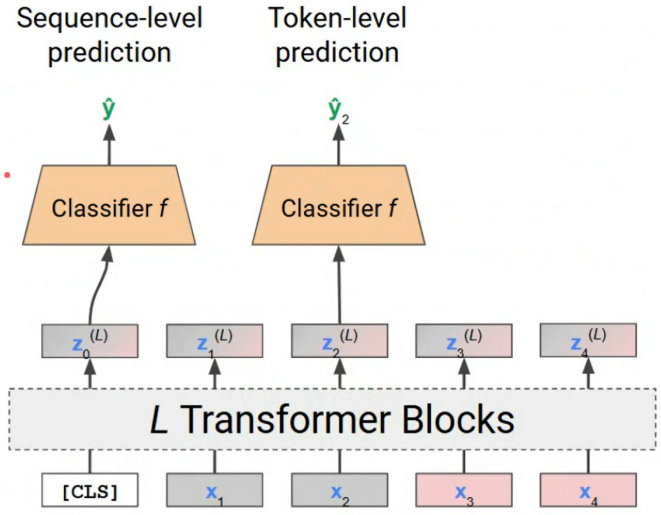
RNN을 사용하지 않고 transformer layer만으로도 학습을 수행하고, regression, classification 등을 수행할 수 있음.

출력 token 마다의 정답이 존재하는 경우(token-level)에는 단순히 각 token을 활용해 loss를 계산하고, backpropagtation으로 W_Q, W_K, W_V, W_O을 최적화하여 학습할 수 있음.

출력 token을 종합한 전체 sequence에 대한 정답이 존재하여(sequence-level) sequence를 대표하는 token을 활용해야 하는 경우, 아래의 방법으로 학습할 수 있음.

1) 각 출력 token에 대한 평균 token을 계산해 예측값으로 활용함. 이 경우 평균 token이 전체 sequence에 대한 모든 정보를 반영하지는 못하므로, 길이가 짧거나 각 token들이 유사한 경우에만 잘 동작함.

2) CLS token을 입력 sequence에 포함시켜 transformer layer를 거치도록 하고, CLS token에 대한 출력을 예측값으로 활용함. CLS(classification) token은 어떠한 의미를 가지지 않는 dummy token으로, transformer layer를 거치면 전체 sequence에 대한 정보를 적은 편향으로 반영할 수 있음.



참고로 행렬을 곱해 특정 차원으로 변환하는 연산을 Projection이라고 함. 이에 따라 DL에서는 단순히 가중치 행렬을 곱하는 과정을 의미하기도 함.

5.2.2. Transformer

1. Transformer

Transformer는 self-attention 기법을 사용해 sequential 데이터를 학습하는 NN 모델임. 여기에는 2017년 논문 Attention Is All You Need에서의 번역기 transformer를 기반으로 정리했음.

transformer 이전에는 RNN으로 NLP를 처리했음. 하지만 RNN은 데이터를 유지하기 어렵고 각 시점이 이전 시점에 종속되므로 병렬 처리가 불가능해 efficiency 확보에 한계가 있음. 또한 CNN을 활용하는 방법도 있지만 당연하게도 전체 context를 반영하지 못함. transformer는 RNN, CNN을 활용하지 않고 sequential 데이터가 가지는 순서와 연관성을 PE와 attention으로 반영함.

transformer는 아래와 같이 여러 개의 transformer layer로 이루어지는데, 각 layer는 그 역할에 따라 encoder와 decoder로 구분됨. encoder-decoder 구조를 가진다는 점은 seq2seq와 동일함.

구체적인 구성은 아래에 따로 정리함.

2. Positional Encoding

Positional Encoding(PE)은 embedding된 token에 순서 정보를 추가하는 기법임. 이는 token별로 위치에 따른 벡터를 생성해 단순히 더하는 식으로 구현됨. 이에 따라 생성된 벡터는 token과 동일한 크기여야 함.

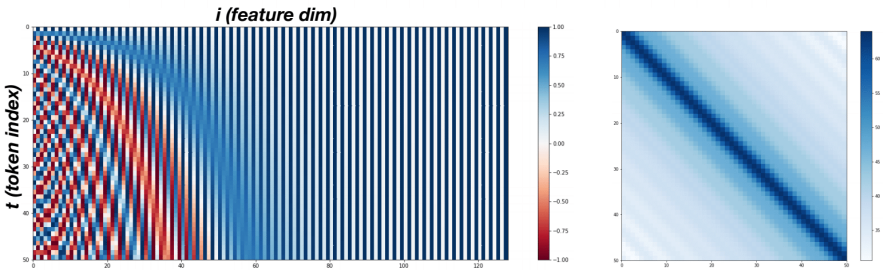
attention만을 적용하면 context 정보는 추가되지만 순서 및 위치 정보는 추가되지 않음. 순서 및 위치 정보를 추가하지 않으면, 동일한 종류의 token으로 구성되어 있지만 그 순서가 다른 두 sequence가 동일하게 처리되는 상황 등이 발생할 수 있음.

텍스트 데이터를 처리하는 경우, PE는 아래의 2가지 조건을 만족하는 벡터를 token별로 생성할 수 있어야 함. 물론 텍스트 데이터가 아니라 다른 데이터를 처리하는 경우에는 해당 데이터에 부합하는 방식을 사용함.

1) 각 위치에 대한 유일한 값이어야 함.
2) 위치가 가까우면 값이 유사해야 함. 이는 텍스트를 데이터를 잘 처리하기 위한 조건으로, 언어에서는 특정 성질의 단어가 대체로 유사한 위치에 옴.

sin/cos함수를 활용해 이를 만족하는 값을 생성할 수 있음. 그 수식은 아래와 같음. d_{model}은 전체 token의 개수이고, pos는 sequence에서의 위치, i는 벡터의 각 원소를 나타내는 인덱스임. i에 의해 주기가 결정되는 것을 생각하면, i가 작은 쪽에서는 pos에 따른 값의 변화가 크고 i가 큰 쪽에서는 변화가 작음. 이에 따라 벡터가 구성되는 것을 이해할 수 있음.

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases} \qquad \omega_k = \frac{1}{10000^{2k/d}}$$

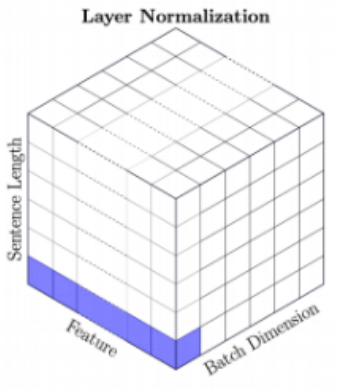


3. Residual Connection/Normalization

transformer에서는 각 layer를 거친 뒤 residual connection에 따른 합 연산과 layer normalization을 수행함.

이전에 다룬 것처럼 residual connection을 적용하면 여러 layer가 쌓이면서도 성능을 보장할 수 있음. 이때당연하게도 residual connection에서 입력과 출력을 더하기 때문에, 각 layer는 입력의 shape을 보존하도록 설계되어 있음.

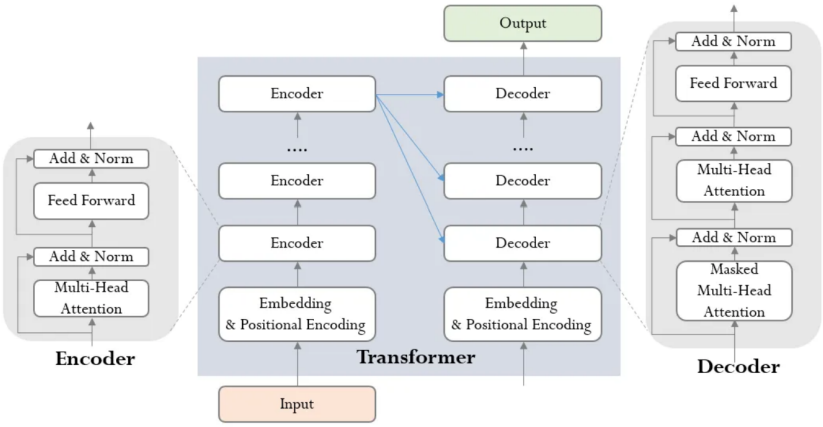
Layer Normalization은 하나의 token에 대해 각 feature값들을 normalize하는 것임. 즉, token이 가지는 값들끼리 normalization을 수행하는 것. 이는 batch normalization과 동일한 수식(평균, 분산, parameter 사용)으로 계산됨. 처음에는 normalization layer를 각 layer 뒤에 오도록(Post-norm) 설계했지만, 현재는 학습 안정성 등의 측면에서 normalization layer를 각 layer 앞에 오도록(Pre-norm) 설계하는 경우가 많다고 함.



4. Stacked Transformer Layers

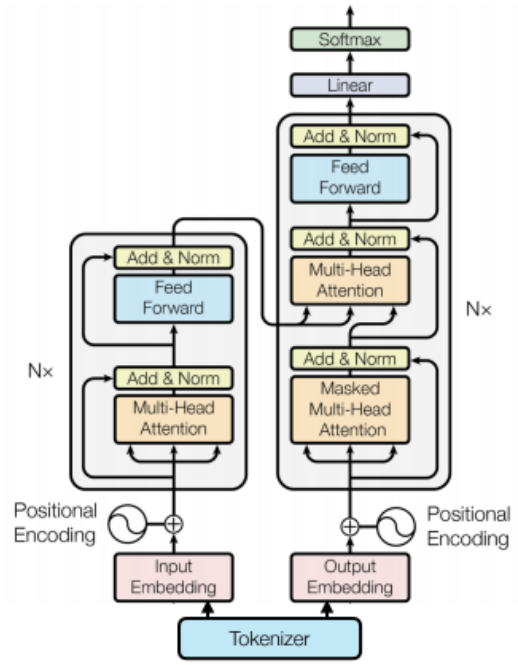
transformer에서는 여러 개의 encoder와 decoder가 각각 세로로 쌓여 동작함. 즉, 출력 결과를 다시 encoder/decoder에 넣는 과정을 여러 번 반복하여 마지막으로 생성된 출력을 값으로 사용함. encoder에서는 마지막에 출력된 결과를 decoder에 전달하고, 모든 decoder는 해당 값을 사용해서 다음 decoder로 출력을 내보내며 예측값을 하나씩 추가하여 전체 예측 결과를 도출함. decoder는 출력으로 EOS token이 나올 때까지 반복됨.

이에 따른 transformer의 실질적인 구조는 아래와 같음.



5.2.3. Transformer 구성

transformer는 아래와 같은 구성을 가짐. 이는 총 9가지 단계를 거쳐 동작함.



1. Step1 : Input Embedding

Tokenizer를 사용하여 입력의 각 sequence를 여러 token으로 분해하고, word embedding으로 각각을 embedding함. 이후 PE를 더함.

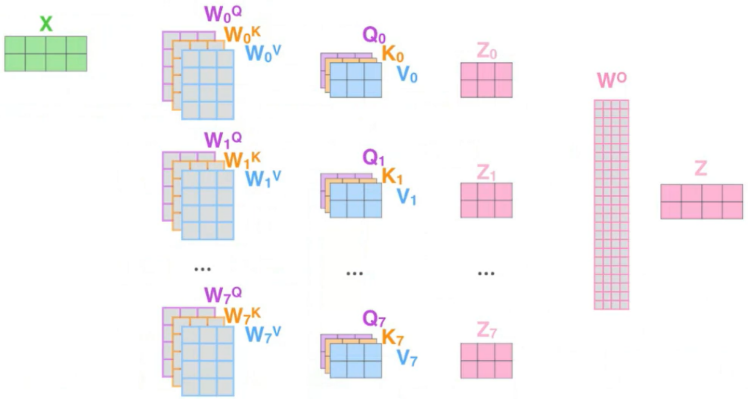
decoder에서 EOS(End Of Sentence) token이 나올 때까지 출력하므로, 각 입력 sequence의 끝에 EOS token을 붙여 학습시켜야 함.

2. Step2 : Multi-head Attention

token들에 대한 multi-head self-attention을 수행함.

Multi-head로 수행한다는 것은 개별적인 가중치(W_0는 공통적으로 사용함.)를 가지는 여러 attention을 병렬적으로 수행한 뒤 그 결과를 합치는 것을 말함. 특히 대명사와 같은 경우 context에 따라 여러 가지 의미를 가질 수 있는데, 하나의 attention만을 계속해서 수행하면 하나의 의미만을 학습하게 될 수 있으므로 multi-head를 적용함.

token으로 행이 구성된 sequence 행렬을 입력으로 받으면, 각 attention에서는 가중치로 Q, K, V를 계산하고 결과적으로 Z 행렬을 도출함. multi-head이므로 각 attention에서 내보낸 Z 행렬들을 열 방향으로 결합하고, 그 결과를 W_0로 크기를 맞춰 출력함.



3. Step3 : Feed Forward Layer

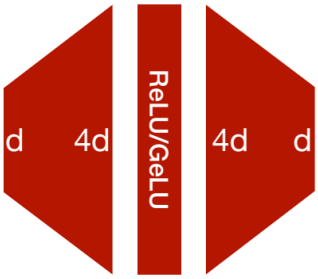
feed forward layer를 거침.

Feed Forward Layer 또는 FFN(Feed Forward Network)는 linear layer, 활성 함수(ReLU), linear layer

로 구성된 layer임. 즉, feed forward layer는 단순히 FC로 생각할 수 있는데, attention은 각 token에 context를 반영하지만 개별 token에 대한 변환까지 잘 수행하지는 못하므로 추가한 것임. 이 부분을 생략하면 결과가 잘 안 나오게 된다고 함.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

기본적으로는 아래와 같이 inverted bottleneck 형태로 구현됨.



4. Step4 : Decoder Input Embedding
encoder와 동일하게 입력의 각 sequence를 여러 token으로 분해하고, 각각을 embedding함. 이후 PE를 더함.

이때 seq2seq에서와 동일하게 autoregressive로 동작함. 입력에 대한 출력은 바로 다음 token에 대한 예측값이고, 이 예측값을 입력에 포함시켜 다시 입력으로 넣어 또 그 다음을 예측함. 이때 RNN에서는 단순히 직전 출력을 입력으로 사용했지만, transformer는 이전 출력 결과들로 구성한 누적 token들을 한 번에 입력으로 사용함.

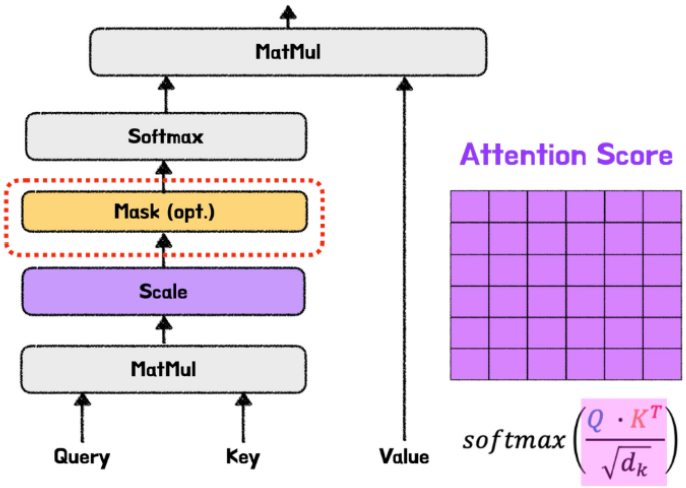
입력 sequence의 첫 번째 token은 SOS(Start Of Sentence) token이고, 그 다음 token부터 예측값으로 구성됨.

5. Step5 : Masked Multi-head Attention
encoder와 동일하게 multi-head attention을 적용하는데, 중간에 입력에서 아직 예측되지 않은 부분에 대해 masked 처리를 함.

transformer의 layer들은 입력과 출력의 크기를 보존함. 그런데 decoder에서는 순차적으로 바로 다음 token을 예측하므로, 단순히 multi-head attention을 적용하면 해당 시점에서 아직 예측하지 않은 token에 대한 값을 사용하게 됨.

이에 따라 masked multi-head attention에서는 softmax를 적용하기 직전에 아직 예측하지 않은 부분의 값을 -inf로 변환(masking)함. 이후 softmax를 적용하면 -inf에 해당하는 부분은 0이 되므로 계산에 활용되지 않음.

이와 같이 masking을 적용한 것을 Casual Self Attention(이때의 mask를 Casual Mask라고 함.), 적용하지 않은 것을 Global Self Attention이라고도 함.



6. Step6 : Encoder-Decoder Attention
encoder의 출력과 decoder의 값을 활용하여 multi-head attention을 수행함. 이때 K와 V는 encoder의 출력을 사용해 생성하고, Q는 decoder의 값을 사용해 생성함. 즉, encoder에서 받은 입력의 의미를 그대로 사용하는 것임.

7. Step7 : Feed Forward Layer
encoder에서와 같은 feed forward layer를 거침.

8. Step8 : Linear Layer
출력을 linear layer에 넣어 각 token을 정답과 같은 크기로 맞춤.

9. Step9 : Softmax Layer
classification을 위해 softmax layer를 사용함.

이후 cross entropy 등을 사용해 loss를 계산함.

decoder의 동작에서, 각 시점에서의 값이 최적일 수도 있지만 틀린 값일 수도 있음. 중간에 틀린 값을 도출했다면 그 뒤의 값들도 틀릴 수 있음. 이에 따라 출력에 대해 확률 상위 n개의 예측값을 뽑아 각각을 활용한 출력값을 계산하도록 구현할 수 있음. 계속해서 상위 n개를 뽑다가 마지막에는 가장 높은 확률을 가지는 것을 출력함. 이를 Beam Search라고 함. 반면 매번 가장 높은 확률의 값 하나에 대해서만 계산하는 것을 Greedy Search라고 함.

5.3. Transformer의 활용

5.3.1. BERT

1. BERT
BERT(Bidirectional Encoder Representations form Transformers)는 구글에서 개발한 사전 훈련된 NLP model로, transformer의 encoder를 활용함. 주로 텍스트 embedding을 생성하는 등의 작업을 수행함.

BERT의 학습에는 사람이 개입할 필요 없이 텍스트 데이터만 입력하면 됨. 그래서 large scale 데이터로 학습이 가능함.

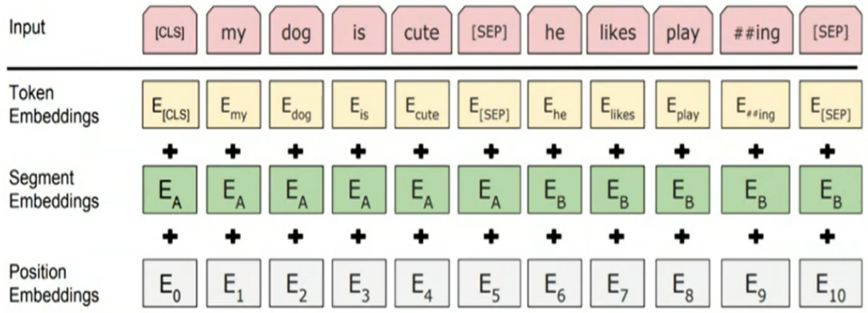
BERT는 오늘날 단어에 대한 embedding을 생성할 때 기본적으로 사용되는 모델임.

2. 학습 과정
학습 시에 입력 sequence는 2개의 문장으로 구성됨. 이 문장들의 각 단어에 대해 아래와 같은 3가지 embedding을 생성해 더한 것을 입력으로 넣음.

1) Token Embedding : 각 단어에 대한 embedding.
2) Segment Embedding : 2개의 문장 중 어디에 속하는 단어인지를 나타내는 embedding.

3) Poisiton Embedding : 전체 sequence에서의 위치를 나타내는 embedding.

이때 첫번째 token은 학습을 위한 CLS token이고, 두 문장은 SEP token으로 구분됨.



BERT에는 2가지 종류의 학습을 수행할 수 있음. MLM이 NSP에 비해 더 중요하다고 함.

1) MLM
MLM(Masked Language Modeling)은 단어에 대한 정확도 학습 기법으로, token의 15%를 MASK token으로 변환한 뒤 해당 token을 예측하는 방식으로 학습함. 빈칸을 뚫어놓고 어떤 단어가 적절한지 맞추는 것.

2) NSP
NSP(Next Sentnece Prediction)은 문장 관계에 대한 정확도 학습 기법으로, sequence의 두 문장이 연속적인지를 판단하는 binary classification을 수행하여 학습함. 이때 입력 sequence의 절반은 실제로 붙어있던 두 문장으로, 나머지 절반은 그렇지 않은 두 문장으로 구성함.

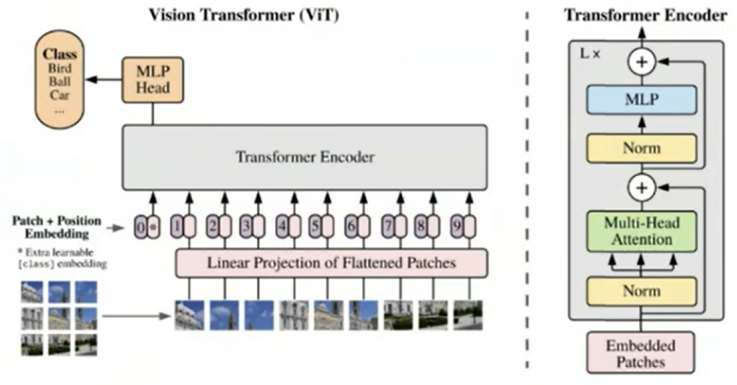
5.3.2. ViT

1. ViT
ViT(Vision Transformer)는 이미지 데이터 처리에 transformer의 encoder를 활용한 모델임.

ViT가 CNN보다 성능상 뛰어나려면 모델이 크고 데이터가 굉장히 많아야 함. CNN은 인접한 데이터끼리의 관계성을 반영하는데, transformer는 인접한 부분뿐만 아니라 모든 token들끼리의 순서와 관계성을 반영함. 전체 부분에 대한 관계성을 파악하느라 학습 시간이 오래 걸리지만, 데이터를 많이 사용하면 인접하지 않은 부분들까지의 관계성을 학습할 수 있어 성능이 좋아짐. 현재는 하드웨어의 개선으로 인해 CNN에 비해 ViT의 이점이 부각됨.

2. 학습 과정
이미지 데이터를 token으로 잘 분해하면 transformer에 넣을 수 있음. ViT에서는 이미지 데이터를 16×16 크기의 단위인 Patch로 분해하고, linear layer를 거쳐 고정된 길이의 벡터로 flatterng함. 이후 PE를 더하는데, 이때 PE는 번역기에서와는 달리 단순히 임의로 초기화하고 이후 학습하는 방식으로 정해짐. d

첫 번째 token으로는 CLS token을 넣어 학습함.



Part III

기타

1. 기타

1.1. 기타 pytorch 관련 정리

1.1.1. GPU 사용

pytorch에서는 device(GPU)를 지정하여 연산을 수행할 수 있음.

특히 nvidia GPU의 경우 cuda를 사용해 간단히 활용할 수 있음. nvidia GPU를 활용할 수 있는 환경에서는 찾아서 적용하자.

1.1.2. 데이터 저장 및 불러오기

1. 데이터셋
pytorch의 객체는 아래와 같이 torch에서 제공하는 save(), load() 메서드를 사용해 저장하고 불러올 수 있음.

```
data = torch.Tensor([1, 2, 3, 4, 5])
torch.save(data, "./test.pt")
data = torch.load("./test,pt")
```

참고로 JPEG 등의 이미지 데이터는 ImageFolder로 불러오고, 아래와 같이 Image class의 save() 메서드를 사용해 저장할 수 있음. 이 경우에도 torch.save()를 사용할 수는 있지만 이는 pytorch가 해석할 수 있는 형태로 저장하는 것으로, JPEG와는 호환되지 않음.

```
data.save("./PyTorch/custom_data/train_data/%d_%d.jpeg"%(data_index, label))
```

2. 모델 데이터
매번 모델의 값을 계산하는 것은 비효율적이므로, pytorch에서는 모델 데이터를 저장하고 불러오는 기능을 제공함.

아래와 같이 모델과 경로를 지정하여 모델 데이터를 저장하고 불러올 수 있음. pth는 모델 데이터의 파일 확장자임. 이때 저장하려는 디렉토리가 존재하지 않으면 에러가 남.

```
model = ConvModel()
torch.save(model.state_dict(), "./model/convmodel.pth") # 저장

new_model = ConvModel()
new_model.load_state_dict(torch.load("./model/convmodel.pth")) # 불러오기
```

1.2. torchvision

1.2.1. torchvision

1. torchvision
torchvision은 유명한 데이터셋들, 모델 아키텍처, 이미지 변환 등을 포함하는 패키지임. torchvision은 Transforms, MNIST, Imagefolder 등을 포함함.

개별화된 DL을 수행하기 위해서는 모델을 구성하는 것 외에도, 자체적인 데이터셋을 구축할 수 있어야 함. torchvision 등에 정의되어 있는 데이터셋의 코드를 참고하여 custom dataset을 만들 수 있음.

2. transforms
transforms는 이미지 데이터에 대한 전처리 및 변환을 수행하는 class임. 이는 torchvision에 포함되어 있음.

transforms에는 아래와 같은 메서드들이 포함되어 있음. toTensor()는 0부터 255까지의 값을 가지는 데이터를 0부터 1까지의 값을 가지는 tensor로 변환함.

```
transforms.Resize((128, 128)),      # 데이터의 크기를 (128, 128)로 조정
transforms.ToTensor(),              # 데이터(0~255)를 tensor(0~1)로 변환
transforms.Normalize(mean, std)     # 데이터를 정규화
```

아래와 같이 전처리 및 변환을 리스트로 작성하여 transforms 객체를 구성할 수 있음. 이는 torchvision 데이터셋 등에 적용할 수 있음.

```
import torchvision
import torchvision.transforms as transforms

trans = transforms.Compose([
    transforms.Resize((128, 128)),
])
...
train_data = torchvision.datasets.ImageFloder(root="/origin_data",
                                                    transform=trans)
```

정규화 및 표준화는 아래와 같이 수행할 수 있음. 각 channel 별로 평균과 표준편차를 계산하고, 255로 나누어 0부터 1까지의 값으로 정규화함. toTensor()에 의해 데이터는 0부터 1까지의 값으로 맞춰져 있으므로 이후 Normalize()로 0부터 1까지의 값으로 표준화를 수행할 수 있음.

```
# mean, std 계산을 위한 dataset load 수행

train_data_mean = trainset.data.mean( axis=(0,1,2) )
train_data_std = trainset.data.std( axis=(0,1,2) )

train_data_mean = train_data_mean / 255
train_data_std = train_data_std / 255

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(train_data_mean, train_data_std)
])

# 이후 해당 transform으로 dataset을 다시 load함
```

1.2.2. MNIST

MNIST은 0부터 9까지의 숫자에 대한 손글씨 데이터셋으로, 우체국에서 손으로 작성한 우편번호를 읽는 모델을 만들기 위해 등장했음.

training set으로 60000개의 데이터가, test set으로 10000개의 데이터가 포함되어 있음. torchvisioin.datasets에서 불러온 MINST 데이터셋은 Dataset 객체이고, 해당 객체의 각 데이터는 입력 데이터와 label(정답)

로 이루어진 tuple로 구성되어 있음. 입력 데이터는 ×28 × 28인 이미지(gray scale)이고, 픽셀 하나 당 하나의 값을 가짐. label은 각 입력 데이터에 해당하는 숫자임(one-hot이 아닌 단순 숫자.).

아래의 코드로 MNIST 데이터셋을 가져올 수 있음. root는 데이터셋의 위치를, train은 training set을 가져올지를(true이면 training set, false이면 test set), transform은 데이터셋 변환을, download는 필요 시 다운로드 여부를 지정함. 불러온 데이터는 주로 DataLoader에 넣어 사용함.

```
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# MNIST 불러오기
mnist_train = dsets.MNIST(root="MNIST_data/", train=True,
                          transform=transforms.ToTensor(), download=True)
mnist_train = Subset(mnist_train, list(range(60000))) # 일부만 사용 가능

# DataLoader 객체 생성하기
dataloader = DataLoader(mnist_train, batch_size=4, shuffle=True,
                          drop_last=True)

# learning
epochs = 1000
for epoch in range(1, epochs + 1) :
    for data, label in dataloader :
        # for batch_index, batch_data in enumerate(dataloader) 로도 가능.
        data = data.view(-1, 28 * 28)
        ...
```

이렇게 얻은 데이터셋을 사용해 softmax classification을 수행할 수 있음. 데이터를 FC에 넣기 전에는 각 이미지를 tensor에서 하나의 행으로 구성하여 처리하기 위해 아래와 같이 shape을 변경함.

```
X = X.view(X.size(0), -1)
```

참고로 matplotlib.pyplot 모듈을 사용하면 데이터를 시각화할 수도 있음.

1.2.3. ImageFolder

ImageFolder는 폴더 구조의 이미지 데이터셋을 불러올 수 있는 class임. 이는 torchvision에 포함되어 있음.

아래와 같이 경로와 transforms를 지정하여 이미지 데이터셋을 불러오면 Dataset 객체로 반환됨. 이때 지정한 경로에 존재하는 폴더들이 개별적으로 label되어 Dataset 객체가 구성됨. 이때 ImageFolder는 기본적으로 지정한 경로 내부에 하나 이상의 폴더가 존재하는 것으로 가정하므로, 폴더가 존재하지 않으면 에러가 남. 생성된 Dataset 객체의 데이터는, 입력 데이터와 대응되는 label(폴더)로 구성된 tuple임.

참고로 ImageFolder로 불러온 데이터가 tensor로 저장되어 있었다면 단순히 사용하면 됨. 반면 tensor가 아닐 때 이를 tensor로 활용하려면 transforms.toTensor()를 사용해 tensor로 변환해야 함.

```
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

trans = transforms.Compose([
    transforms.Resize((64, 128))
])
train_data = torchvision.datasets.ImageFolder(
                    root="./PyTorch/custom_data/origin_data", transform=trans)

for data_index, data_value in enumerate(train_data) :
    data, label = data_value
    ...
```

1.2.4. CIFAR10

CIFAR10은 rgb 이미지 classification을 위한 데이터셋임.

training set으로 60000개의 데이터가, test set으로 10000개의 데이터가 포함되어 있음. torchvisioin.datasets에서 불러온 CIFAR10 데이터셋은 Dataset 객체이고, 해당 객체의 각 데이터는 입력 데이터와 label(정답)로 이루어진 tuple로 구성되어 있음. 입력 데이터는 $3 \times 32 \times 32$인 이미지임. label은 각 입력 데이터에 해당하는 숫자임(one-hot이 아닌 단순 숫자.).

아래의 코드로 CIFAR10 데이터셋을 가져올 수 있음. root는 데이터셋의 위치를, train은 training set을 가져올지를(true이면 training set, false이면 test set), transform은 데이터셋 변환을, download는 필요시 다운로드 여부를 지정함. 불러온 데이터는 주로 DataLoader에 넣어 사용함.

```
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# CIFAR10 dataset
transform = transforms.Compose([
    transforms.ToTensor()
])
train_data = torchvision.datasets.CIFAR10(root='./cifar10', train=True,
                                    download=True, transform=transform)
train_data = Subset(train_data, list(range(60000)))
trainloader = torch.utils.data.DataLoader(train_data,
                                    batch_size=batch_size, shuffle=True)

# learning
epochs = 1000
for epoch in range(1, epochs + 1) :
    for data, label in dataloader :
        # for batch_index, batch_data in enumerate(dataloader) 로도 가능.
        data = data.view(-1, 28 * 28)
        ...
```

1.3. visdom

1.3.1. Visdom

1. Visdom
Visdom은 DL 모델의 학습 상태 등을 시각화하는 도구임.

visdom은 클라이언트-서버를 구조를 기반으로 동작함. 프롬프트에 python -m visdom.server를 입력하면 visdom 서버가 실행되고, 제공되는 포트로 접속하면 코드로 출력한 시각화 결과를 확인할 수 있음.

2. 기본 사용법
여기에 정리한 것들 외에도 다양한 시각화가 가능하므로 필요하면 찾아보자.

1) visdom 객체 생성
visdom을 사용하려면 아래와 같이 Visdom 객체를 생성하고, 해당 객체의 메서드를 사용함. 이때 visdom 서버가 꺼져 있으면 에러가 남.

```
import visdom

vis = visdom.Visom()
```

2) 데이터 이미지 출력
아래와 같이 tensor의 값을 이미지로 출력할 수 있음. image()에는 shape이 (channels, height, width)인 tensor를 넣어 단일 이미지를 출력할 수 있고, images()에는 shape이 (batch_size, channels, height, width)인 tensor를 넣어 여러 이미지를 출력할 수 있음. 그 결과는 visdom 서버에 접속하여 확인할 수 있음.

```
vis.image(t1)
vis.images(t2)
```

3) 선그래프 출력
아래와 같이 X와 Y를 지정하여 line()으로 선그래프를 출력할 수 있음. 이때 X와 Y는 동일한 shape이어야 하는데, 1차원이면 하나의 선이 그려지고 2차원이면 작성한 만큼의 선이 그려짐. 2차원의 경우 각 데이터가 열 단위로 사용되어 선이 그려짐. 또한 Y가 1차원인 경우 X를 지정하지 않으면 0과 1 사이 숫자를 차례대로 사용함.

line()은 해당 그래프의 id를 반환함.

인자에 win으로 그래프의 id를 넣고, update 방식(append는 추가, replace는 대체)을 지정하여 선그래프를 업데이트할 수 있음.

인자에 opts로 title, label, legend 등의 옵션을 지정할 수 있음. 이때 dictonary 형태로 작성함.

```
graph_id = vis.line(X=x, Y=y, env="main") # 선그래프 출력
vis.line(X=x, Y=y, win=graph_id, update="append") # 선그래프 업데이트
graph_id = vis.line(X=x, Y=y, env="main",
                    opts=dict(title="test", legend=["1번", "])) # 옵션 지정
```

4) 환경 지정 Visdom 객체 또는 출력한 각 내용에 아래와 같이 환경(env)을 지정할 수 있음. 환경은 출력 내용을 구분하는 논리적 공간으로, 특정 환경에 대한 일괄적인 처리가 가능함.

close()는 서버에서 해당 이미지들을 삭제함.

```
vis.image(t1, env="main")
...
vis.close(env="main")
```

3. 코드
MNIST에서 epoch마다의 cost 값을 선그래프로 출력하는 상황에서 코드는 아래와 같이 작성할 수 있음.

빈 그래프를 생성하고 그 id를 활용하여 반복적으로 update함.

```
# visdom
vis = visdom.Visdom()
graph_id = vis.line(Y=torch.zeros(1), opts=dict(title="train"))
vis.close(env="main")

def printVisdom(id, X, Y) :
    vis.line(X=X, Y=Y, win=id)
...
for epoch in range(1, epochs + 1) :
    ...
    vis.line(X=torch.Tensor([epoch]), Y=torch.Tensor([cost.item()]),
                                            win=graph_id, update="append")
```

1.4. Docker

1.4.1. Docker

Docker(도커)는 컨테이너 기반의 가상화 시스템임.

virtual machine처럼 os 전체에 대해 가상화하는 대신, 필요한 일부만을 가볍게 가상화한 것이 컨테이너임. docker를 설치하고 그 위에 컨테이너를 올려 테스트 등을 쉽게 수행할 수 있음.

1.5. 파이썬 문법

1.5.1. 객체와 class

파이썬은 객체지향프로그래밍을 지원하고, 그 개념은 java의 그것과 다르지 않음. 더 구체적인 사용 방식과 문법 등은 필요할 때마다 찾아보자.

1. 객체
파이썬에서 모든 것은 객체이고(모든 변수는 객체를 가리킴.), 객체는 고유한 참조(reference, id)를 가지고, 속성(Attribute, java에서의 필드)과 메서드(Method)를 가질 수 있음.

객체는 mutable과 immutable로 구분됨. Immutable은 값을 수정할 수 없는 객체로, 정수, 실수, 문자열, tuple 등이 있음. Mutable은 값을 수정할 수 있는 객체로, list 등이 있음. immutable 객체는 값을 변경할 수 없기 때문에, immutable 객체를 가리키던 변수의 값이 바뀌면 해당 변수는 다른 객체를 가리키게 됨. 참고로 이후 가리키는 변수가 없는 객체는 garbage collector에 의해 처리됨.

파이썬에서 함수 호출 시에는 인자로 작성한 변수에 해당하는 객체가 그대로 매개변수에 대입되는데, 해당 객체가 immutable이라면 값 수정 시에 새로운 객체가 사용되어 call by value처럼 동작하고, 해당 객체가 mutalbe이라면 값을 수정해도 기존 객체가 유지되므로 call by reference처럼 동작함. 이런 특징을 call by assignment라고 함.

2. class 정의
아래와 같이 class를 정의함. 간단하게 상속 및 오버라이드도 할 수 있음. 이때 self는 자기 자신을 가리키는 참조임. 참고로 속성은 정의되어 있지 않아도 객체 사용 시에 작성하여 사용할 수 있음.

```
class MyParentClass:
    name = "MyParentClass"

    def printName(self):
        print(self.name)

class MyChildClass(MyParentClass):
    name = "MyChildClass"

    def printName(self):
        print(self.name)
```

3. 객체 생성
아래와 같이 객체를 생성함. 각 필드와 메서드는 .으로 접근할 수 있음.

```
a = MyChildClass()
a.printName()
```

3. 특수 메서드들
아래와 같은 특수한 메서드들이 있음.

1) ___init___()
아래와 같이 ___init___을 식별자로 메서드를 정의하면 생성자로 취급됨.

```
class MyClass:
    name = "MyClass"

    def __init__(self):
        print("class name is", self.name)
```

2) ___call___()
아래와 같이 ___call___을 식별자로 메서드를 정의하면 객체명을 메서드 식별자처럼 사용해 호출할 수 있고, 이때 ___call___()에 해당되는 작업이 수행됨.

```
class MyPow:
    def __call__(self, x):
        return (x * x)
a = MyPow()
a(5) # 25
```

1.5.2. Tuple

1. Tuple
Tuple은 순서가 존재하는 immutable 데이터 구조임. list와 유사하지만 immutable인 것. 아래와 같이 괄호로 나타냄. tuple을 한 번에 여러 번수에 할당하도록 작성할 수 있는데, 이를 unpacking이라고 함.

```
x = (1, 2, 3)
x[2] # 3

a, b = (5, 6) # unpacking
```

2. 반환 시의 Tuple
아래와 같이 함수에서 return으로 여러 개의 값을 작성하면 자동으로 tuple이 구성되어 반환됨.

```
def testTupleReturn() :
    return 1, 2

a, b = testTupleReturn() # (1, 2)
```

1.5.3. Dictionary

파이썬에서 Dictionary는 key-value 쌍으로 중복되지 않게 데이터를 저장하는 자료구조임. 아래와 같이 여러 방식으로 정의하고 접근할 수 있음.

```
# {}를 사용한 정의
my_dict = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# dict()을 사용한 정의
my_dict = dict(name="John", age=30, city="New York")

my_dict["name"] # "John"

# for문을 활용한 정의
squares = {x: x**2 for x in range(1, 6)}
```

1.5.4. 기타 함수들

자주 사용되는 기타 함수들은 아래와 같음.

1) enumerate(x) : 반복 가능한(iterable) 객체를 인자로 받아 각 인덱스와 값을 tuple로 묶어서 enumerate 객체를 생성해 반환함. enumerate 객체는 아래와 같이 for문에 작성하여 매 루프마다 하나의 tuple(인덱스와 값 쌍)이 사용되도록 할 수 있음.

```
fruits = ['apple', 'banana', 'cherry']

for index, value in enumerate(fruits):
    print(index, value)
```

1.5.5. 모듈 설치 위치 확인하기

아래와 같이 파이썬 모듈이 설치된 위치를 확인할 수 있음. 구체적인 동작과 구현이 궁금하다면 코드를 직접 뜯어보자.

```
import torchvision.models.vgg as vgg

print(vgg.__file__)
```

1.5.6. 타입 어노테이션

타입 어노테이션(Type Annotation)은 변수, 함수의 반환값 등에 타입을 명시적으로 지정하는 문법임.

아래와 같이 지정함.

```
a : int = 1

intnumbers : Set[int] = {1, 2, 3, 4, 5}

def sum_data(numbers : List[int]) -> int :
    return sum(numbers)
```

물론 런타임에 값이 지정한 타입과 일치하지 않아도 에러가 나지는 않음. mypy 등의 정적 타입 검사기를 사용하면 실행 전에 타입 어노테이션에 따른 검사를 수행할 수 있음.

1.6. 기타 DL 관련 내용

1.6.1. Transfer Learning

Transfer Learning은 train한 네트워크를 다른 task에 활용하는 것임.

전체 네트워크를 fine-tuning해서 활용할 수도 있지만, 앞쪽은 freeze하고 마지막 부분의 layer만 추가로 학습시킬 수도 있음. 또는 필요한 만큼의 뒤쪽 layer만 학습시킬 수 있음.

이는 pre-train 시에 활용한 데이터셋이 훨씬 커야 한다고 함.

1.6.2. Multitask Learning

Multitask Learning은 하나의 model이 여러 개의 관련 task를 동시에 처리하도록 하는 기법임.

예를 들어, 사진에서 보행자, 차, stop sign, 신호등을 찾아내는 작업을 생각할 수 있음. 이 경우 단순히 각각에 대한 확률을 도출하도록 하면 됨. 길이가 4인 벡터를 도출하고, 각각에 대해 BCE를 적용함.

1.6.3. LLM

LLM(Large Language Model)은 방대한 양의 데이터로 pre-train된 초대형 language model임.

매우 유연하고 높은 성능을 보이지만, Hallucination와 Outdated Information, Lack of in-depth Knowledge 등의 문제점이 존재함.

RAG, prompt engineering 등으로 이를 개선할 수 있음.

1.6.4. Retrival Augmented Generation

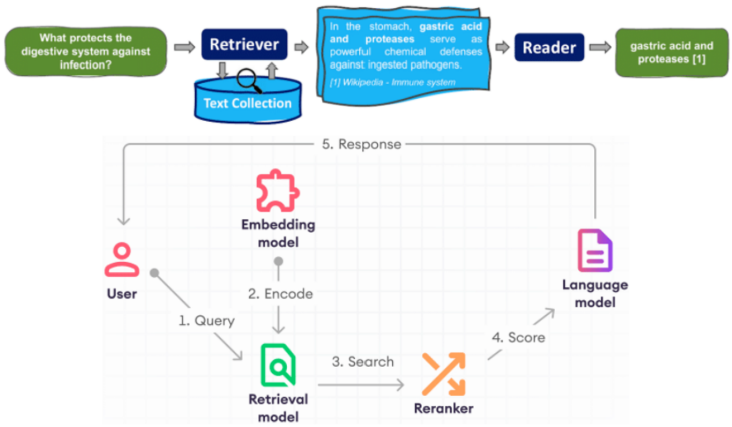
Retrival Augmented Generation(RAG)는 LLM 내부에 저장된 데이터만이 아니라 외부(DB 등)에 저장된 데이터를 활용하는 기법임.

LLM에 직접 데이터를 저장한다면, 해당 데이터는 유지 또는 수정되기 어려움.

아래와 같이 사용자로부터 query가 들어오면 embedding model을 활용해 데이터를 vector화하고, retriever로 query에 맞는 데이터를 선택한 뒤 LLM에 전달함. 또한 추가로 reranker를 사용해 query와의 관련성을 조정할 수 있음.

구체적으로는, RAG에서는 문서를 chunking해서 여러 개의 chunk를 생성하고, 나름의 encoder로 각 chunk를 벡터화하여 DB에 저장함. 이후 query가 들어오면 동일하게 벡터화한 뒤 DB의 벡터 중에서 유사도가 높은 것(dot product 값이 큰 것)을 선택하고, 해당 벡터에 대응되는 chunk를 query와 함께 프롬프트에 입력함.

Langchain 등의 tool을 사용해 RAG를 활용할 수 있음.



1.6.5. Prompt Engineering

Prompt Engineering은 model이 원하는 결과를 생성하도록 prompt를 잘 설계하고 입력하는 기법임.

단순하게는 입력을 구체적으로 적는 방법을 사용할 수 있고, 또한 아래와 같은 기법들을 활용할 수 있음.

1. Zero-shot Prompting
Zero-shot Prompting은 각 task 별로 하나의 model을 구성해 사용하던 방식과 달리, 하나의 foundation model로 추가적인 학습 없이 여러가지 task를 처리할 수 있도록 한 기법임.

2. Few-shot Prompting
Few-shot Prompting은 추가적인 학습 없이, 몇 가지 예시를 제공하여 어떤 새로운 task를 처리할 수 있도록 한 기법임. 이는 입력된 예시들의 맥락(context)를 활용하므로 In-context Learning이라고도 함.

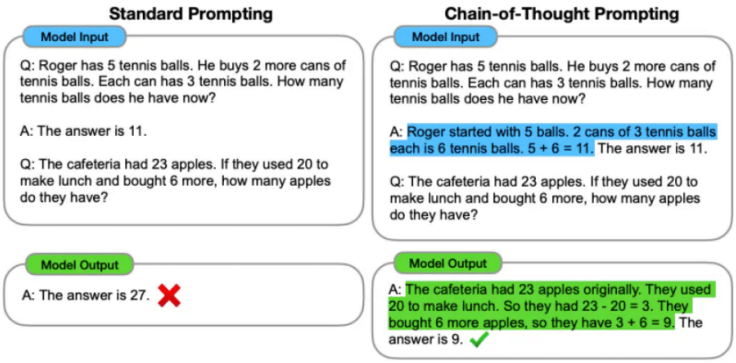
few-shot prompting 시에는 아래와 같은 경험적 노하우가 있다고 함.

1) classification에 대한 예시를 제공할 때는, 각 class에 대한 예시의 개수가 서로 같은 것이 좋다고 함.

2) text 예시 등을 제공할 때는, 각 예시에서의 format을 서로 맞추는 것이 좋다고 함.

3. Chain-of-Thought
Chain-of-Thought는 model이 여러 중간 단계를 거쳐 복잡한 추론 과정을 해결할 수 있도록 유도하는 기법임. 즉, thinking process를 알려주는 것임.

단순히 입력으로 "Let's think step by step."이라는 문장을 추가하기만 해도 복잡한 문제가 zero-shot으로 해결되기도 한다고 함.



1.7. Linear Algebra

1.7.1. 열공간

행렬과 벡터의 곱을 생각하자. 벡터 $x = (x_1, x_2, x_3)^T$와 각 열이 c_1, c_2, c_3인 행렬 A의 곱 Ax는 $x_1c_1 + x_2c_2 + x_3c_3$ 꼴로 표현할 수 있음. 즉, 행렬과 벡터의 곱은 해당 행렬 열공간의 원소로 볼 수 있음.

이때 어떤 행렬의 열공간이 가지는 차원은 해당 행렬의 rank와 같음. rank는 일차독립인 열의 개수로 구할 수 있고, 기본연산을 적용해 일차독립인 열의 개수를 구할 수 있음. 이 연산은 행에 대해서도 성립함.

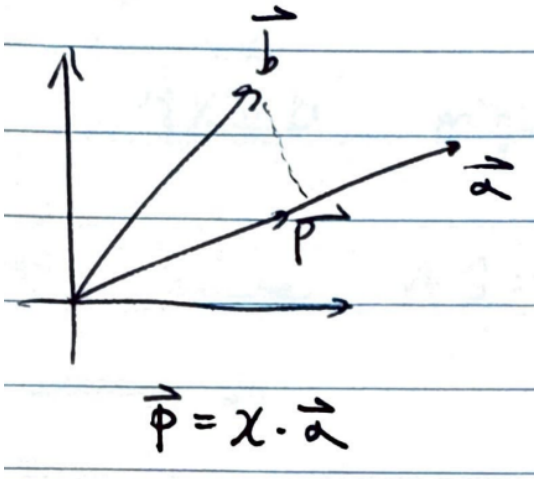
CR decomposition을 사용해 행렬 어떤 행렬의 곱으로 decomposition해서 rank를 찾는 방법도 존재함.

1.7.2. Projection

1. 벡터에 대한 projection
벡터 a와 b가 존재할 때, b에 대응되는 projection matrix를 곱해 b를 a로 projection할 수 있음. projection matrix는 아래와 같음. 이는 $p = xa, a^T(b - p) = 0$을 정리하여 구할 수 있음.

$$P = \frac{aa^T}{a^Ta}, p = \frac{aa^T}{a^Ta}b$$

이때 projection matrix P에 대해 $P^2 = P, P^T = P$가 성립함.



2. 평면에 대한 projection
평면에 대한 projection도 직선에 대한 projection과 유사하게 계산할 수 있음.

어떤 평면에 대해 기저가 $\{a_1, a_2\}$이고, 기저로 열을 구성한 행렬 A가 있다고 하자. 정사영된 벡터 p를 해당 평면의 기저로 표현하면, $p = x_1a_1 + x_2a_2 = Ax$와 같이 나타낼 수 있음. 이때 $a_1^T(b - Ax) = a_0^T(b - Ax) = 0$이므로, 정리하면 $A^T(b - Ax) = 0$임. 이를 정리하면 projection matrix는 아래와 같음.

$$P = A(A^TA)^{-1}A^T, p = A(A^TA)^{-1}A^Tb.$$

여기에서도 $P^2 = P, P^T = P$가 성립함.

SAFREE에서는 projection을 활용해 부적절한 token을 제거함. 부적절한 token의 집합으로 unsafe space를 정의해 새로 입력된 token이 해당 공간과 얼마나 가까운지(요소가 얼마나 포함되어 있는지) projection으로 확인함. safe space를 구성하여 문제가 발생한 token은 해당 공간에 projection해 사용함.

1.8. 참고자료

1.8.1. 참고자료

pytorch, ML, DL, CNN, RNN
1. https://deeplearningzerotoall.github.io/season2/lec_pytorch.html

2. https://hunkim.github.io/ml/

transformer
1. https://www.youtube.com/watch?
v=o3UEbQ24zhQ&list=PL0E_1UqNACXA5u65LBjzFCAVSZ4xuBWqj&index=23
2. https://www.youtube.com/watch?v=AA621UofTUA

CNN을 적용하는 데이터의 가정
https://seoilgun.medium.com/cnn%EC%9D%98-stationarity%EC%99%80-locality-610166700979