

용어

1. Loss function 종류

현재의 신경망이 훈련데이터를 얼마나 잘 맞추지 못하는가를 표시하는 지표다

- 높은 Accuarcy 를 끌어내는 매개변수 값을 찾는 것
- 최적의 매개변수(가중치와 편향) -> 손실값을 가능한 한 작게 하는 매개변수를 찾는 것
 - 미분을 사용
 - 가중치 매개 변수의 값을 아주 조금 변화시켰을 때 손실 함수가 어떻게 변화하는지
 - 미분이 음수이면 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 줄임, 양수면 반대로 음의 방향으로 변화시켜 손실함수의 값을 줄임
 - 미분 값이 0 이면 가중치 매개변수를 어느 쪽으로 움직여도 손실 함수의 값을 달라지지 않음. -> 가중치 매개변수의 갱신 중단

1) 오차제곱합: MSE

$$MSE = \frac{1}{2m} \sum_i (\hat{y}_i - y_i)^2$$

Y4 - L

2) 교차 엔트로피 오차

$$E = -\sum_k t_k \log y_k$$

- cf) same as our previous J(w)

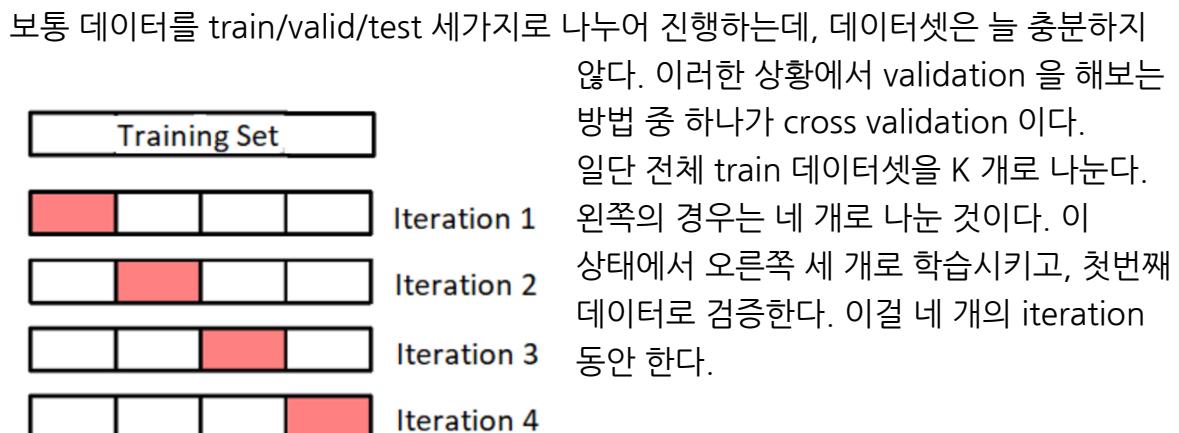
$$J(w) = -\frac{1}{m} \sum_i y^{(i)} \log(g(x^{(i)})) + (1-y^{(i)}) \log(1-g(x^{(i)}))$$

이 경우에는 미분값이 Y4-L/Y4(1-Y4)

3) softmax with loss

이 경우에는 미분값이: Y4-L

2. Cross validation 이란?



3. 소프트맥스 함수

입력값을 정규화(출력의 합이 1 이 되도록 변형)하여 출력한다. 또한, 손글씨 숫자의 경우를 예로 들면, 10 개의 클래스로 분류이므로 Softmax 계층의 입력은 10 개가 된다.

Softmax 수식 쓰고.

Softmax 식에는 왜 지수승이 들어가는가? 확률이니까 양수로 만드려고. 그래서 마이너스 값도 양수로 만들어주려고 지수승으로 올리게 된다.

4. 경사하강법이란

신경망은 학습 단계에서 최적의 매개변수(가중치와 편향)를 찾아내는 것을 목표로 한다. 여기서 최적이란, 손실함수를 최소로 만들어주는 매개 변수를 값을 말한다. 경사하강법은 현 위치에서 기울어진 방향으로 일정 거리만큼 이동한다. 그런 다음 이동한 곳에서도 마찬가지로 기울기를 구하고, 또 그 기울어진 방향으로 나아가기를 반복한다. 이렇게 해서 손실함수의 값을 점차 줄이는 것을 경사하강법이라고 한다. 이때 사용하는 기울기가 **가중치 매개변수에 대한 손실함수의 기울기**이다.

경사하강법을 수식으로 나타내면 다음과 같다.

$$w_{11} = w_{11} - lr * (dL/dw_{11})$$

“parameters = parameters - learning_rate * parameters_gradients”

lr 은 학습률, dL/dw_{11} 은 가중치 매개변수에 대한 손실함수의 기울기

5. 오차역전파법이란

경사하강법에서 가중치매개변수에 대한 손실함수의 기울기를 구하는 쉬운 방법이 역전파법이다. 오차역전파법을 이용하면 느린 수치미분과 달리 기울기를 효율적이고 빠르게 구할 수 있다.

6. 경사하강법과 학습률

학습률이 너무 크면 큰 값으로 발산해버리고, 너무 작으면 거의 갱신되지 않는 채로 경사하강이 끝난다.

내용 문제

1. Lemmatization 이란?

어간 추출. 표제어 추출.

2. Spacy 란?

Python 및 Cython으로 작성된, 고급 자연언어처리를 위한 오픈소스 소프트웨어 라이브러리이다.

3. Activation function 비교 (sigmoid, Tanh, relu)

Activation function이 linear 하면, 레이어를 쌓아도 결국 single layer 수준이다.

그래서 비선형을 가해줘야하는데 sigmoid, tanh, relu 모두 비선형이다.

Sigmoid 와 tanh 모두 vanishing gradient 문제가 있다. 그래프를 보면 x 축 양끝이 기울기가 매우 완만한데, 이는 x의 변화에 대해서 y가 반응을 잘 안한다는 것이다.

Sigmoid의 단점: 역전파때, 결과값이 항상 양수라서 (0과 1 사이의 값) 기울기가 늘 양수거나, 음수인데, 이렇게 되면 지그재그 현상이 발생해서 업데이트 속도가 느려진다. optima를 찾기 힘들다. 기울기가 양수->음수-< 양수 이런식으로 나와야 optima 찾기가 쉽다. tanh는 -1과 1 사이니까 문제 없다.

Tanh는 그냥 scaled sigmoid이다.

$$\tanh(x) = 2 \operatorname{sigmoid}(2x) - 1$$

Relu는 음수의 값은 0으로 만들어주므로 sigmoid와 tanh에 비해 값이 0인 뉴런이 더 많아서 모델이 가벼워진다.

그러나 0이 있기 때문에 dying ReLU 문제가 발생한다.

- That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes).
- This is called **dying ReLu problem**. This problem can cause several neurons to just die and not respond making a substantial part of the network passive.

그래서 leaky ReLu 를 만든다 0 이하의 값이 완전 수평적인 직선은 아니고 살짝 기울어져 있다.

- There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component.
- For example $y = 0.01x$ for $x < 0$ will make it a slightly inclined line rather than horizontal line.
- This is **leaky ReLu**. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.

4. Logistic regression 설명

In logistic regression, the response variable describes the probability that the outcome is the positive case. If the response variable is equal to or exceeds a discrimination threshold, the positive class is predicted

5. Linear regression vs logistic regression

logistic regression 의 sigmoid 함수는 확률값으로 바꾸는 것을 도와준다.

of a single explanatory variable x . We can then express z as follows:

$$z = w_0 + w_1 x$$

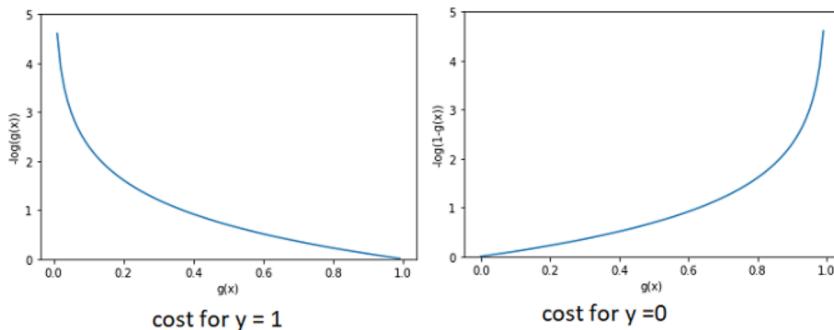
- And the logistic function can now be written as:

$$g(x) = \frac{1}{1 + e^{-(w_0 + w_1 x)}}$$

- Note that $g(x)$ is interpreted as the probability of the dependent variable.
- $g(x) = 0.7$, gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

6. Logistic regression 의 cost function 은 어떠한 모양인가?

linear regression에서 쓰던 loss function 을 그대로 사용할 수는 없다. local minima 가 생기기 때문. 그래서 다음과 같이 정의했는데, 요약을 하자면, $y=1$ 일 때 $g(x)$ 값이 0 으로 갈 수록 코스트가 높고, $y=0$ 일 때 $g(x)$ 값이 0 으로 갈 수록 코스트가 낮다.



- So the cost function for logistic regression is:

$$J(w) = -\frac{1}{m} \sum_i y^{(i)} \log(g(x^{(i)})) + (1-y^{(i)}) \log(1-g(x^{(i)}))$$

7. Regularization 방법 중 하나: weight 를 penalize 한다.

아래와 같이 regularization term 람다를 두고, 람다를 매우 크게 하면, 마치 weight parameter 가 없는 듯한 효과가 난다. 그럼 파라미터가 적어졌다는 건, 모델이 심플해졌다는 것이므로 과적합의 문제가 어느 정도 해소된다.

where $h_{\theta}x_i = \theta_0 + \theta_1x_1 + \theta_2x_2 + \theta_3x_3 + \theta_4x_4$

$$L(x, y) \equiv \sum_{i=1}^n (y_i - h_{\theta}(x_i))^2 + \lambda \sum_{i=1}^n \theta_i^2$$

8. 배치 / 스토카스틱/ 미니배치 간의 비교

배치: 전체 데이터셋에 대한 에러를 구한 뒤 기울기를 계산하여 모델의 파라미터(가중치와 편향)를 계산하는 방법.

장점: 병렬 처리에 유리하다.

단점: 전체 데이터셋과 에러에 대한 정보를 축적하고 있어야 하고 그걸 메모리에 올려야 하기 때문에 메모리가 많이 필요하고, 한 스텝에 모든 학습 데이터 셋을 사용하므로 학습이 오래 걸린다.

스토카스틱: 추출한 데이터 한 개에 대해서 error gradient를 계산하고, 경사하강을 적용하는 방법.

장점: 계산 속도가 빠르다.

단점: 데이터를 한 개씩 처리하기 때문에 GPU의 성능을 전부 활용할 수 없고, 파라미터 업데이트가 너무 자주 일어나서 오버슈팅 문제가 발생함 그래서 global optimal을 못 찾을 수도 있다.

미니배치: 전체 데이터셋 속에서 데이터 m 개를 뽑아서 각 데이터에 대한 기울기를 m 개 구한다음, 그것의 평균을 낸다. 예를 들어 전체 데이터가 1000 개인데, batch size를 10으로 하면 100 개의 mini-batch 가 생기는 거다. 그러면 100 iteration 동안 모델이 업데이트 되고 1 epoch이 끝난다.

장점: SGD 보다 병렬 처리에 유리하고, 전체 학습 데이터를 쓰는 게 아니고 일부분의 학습데이터만을 사용하므로 메모리 사용이 BGD 보다 낫다. 계산도 적당히 빠르고, 파라미터가 너무 자주 업데이트되지도 않는다.

단점: batch size(mini-batch size)를 설정해야 한다 + 에러에 대한 정보를 mini-batch 크기 만큼 축적해서 계산해야 하기 때문에 SGD 보다 메모리 사용이 높다.

9. Optimization-중 매개변수 갱신 방법들 비교해라 (SGD, 모멘텀, AdaGrad, Adam)

SGD

단순하다. 기울어진 방향으로 일정거리만 가겠다는 단순한 방법이다. 그래서 단점은, 비등방성 함수(방향에 따라서 기울기가 달라지는 함수)에서는 탐색 경로가 비효율적이라는 것이다. 만약 최적화하고자 하는 함수가 x 축 방향 기울기는 크고, y 축 방향으로는 작다면 SGD 를 사용해서는 등고선의 최저점에 도달하기 힘들다.

모멘텀

- 가속도 역할을 하는 변수 v 를 더해서, v 의 영향으로 인해 가중치가 감소하던, 혹은 증가하던 방향으로 더 많이 변화하게 해준다.

SGD 가 Oscillation 현상을 겪을 때 더욱 빛을 발한다. Local minima 를 빠져나올 거라고도 기대할 수 있다. 반면 momentum 방식을 이용할 경우 기존의 변수들 θ 외에도 과거에 이동했던 양을 변수별로 저장해야하므로 변수에 대한 메모리가 기존의 두 배로 필요하게 된다.

SGD Nesterov

Momentum 방식에서는 이동 벡터 v_t 를 계산할 때 현재 위치에서의 gradient 와 momentum step 을 독립적으로 계산하고 합친다.

반면, NAG 에서는 momentum step 을 먼저 고려하여, momentum step 을 먼저 이동했다고 생각한 후 그 자리에서의 gradient 를 구해서 gradient step 을 이동한다. NAG 를 이용할 경우 Momentum 방식에 비해 보다 효과적으로 이동할 수 있다

Adagrad

그림상 모멘텀에 비해 지그재그 정도가 적어진다.

신경망 학습에서 학습률 값이 너무 작으면 학습 시간이 길어지고, 너무 크면 발산하여 학습이 제대로 이루어지지 않는다. 이러한 상황에서 Adagrad 의 장점은 각각의 매개변수에 맞추어 학습률을 지정한다는 것이다.

Adagrad 는 h 라는 값을 두는데, 이는 과거의 기울기를 제곱해서 계속 더해간다. 그리고 매개변수를 갱신할 때 $1/\text{루트}(h)$ 를 곱하여 학습률을 조정한다. 따라서 매개변수의 원소 중에서 과거에 많이 움직인 원소의 학습률은 낮춰주는 역할을 한다.

RMSProp

Adagrad 는 과거의 기울기를 제곱해서 계속 더해간다. 그래서 학습을 진행할 수록 갱신이 적게 되어, 어느 순간 갱신되는 정도가 0 이 될 수 있다. 이것을 개선한 기법이 RMSProp 인데, 과거의 모든 기울기를 균일하게 더하는 것이 아니라, 먼 과거의 기울기는 잊고, 새로운 기울기 정보를 더 큰 비율로 반영하며 학습률 값을 정한다.

Adam

모멘텀과 Adagrad 를 합친 방법이다. Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장하며, RMSProp 과 유사하게 기울기의 제곱값의 지수평균을 저장한다.

10. Optimization 과 초기화 방법 설명해라

초기화를 할 때, 데이터가 0 과 1 에 치우쳐 분포하게 되면 역전파의 경로가 점점 작아지다가 사라진다. 이것은 기울기 소실이라 알려진 문제이다.

Sigmoid, Tanh 함수는 좌우대칭이라 중앙 부근이 선형인 함수이다. 그래서 Xavier 초기값이 적당하다. Xavier 초기값이란 앞 계층의 노드가 n 개일 때, 표준편차가 $1/\sqrt{n}$ 인 분포를 사용하여 초기화해준 것이다.

반면, ReLU 를 이용할 때는 He 초기값이 적당하다. He 초기값은 앞 계층의 노드가 n 개 일때, 표준편차가 루트($2/n$)인 정규분포를 사용한다. RELU 는 음의 영역이 0 이라서 더 넓게 분포시키기 위해 2 배의 계수가 필요하다고 볼 수 있다.

11. 초기값을 모두 0 으로 해서는 (정확하는 가중치를 균일한 값으로 초기화해서는 안되는 이유는?)

예를 들어서 TwoLayerNet 에서 첫 번째와 두 번째 층의 가중치가 모두 0 이라고 가정하면, 순전파 때 입력층의 가중치가 0 이기 때문에, 두 번째 층의 뉴런에 모두 같은 값이 전달된다. 두 번째 층의 모든 뉴런에 같은 값이 입력된다는 것은 역전파 때 두 번째 층의 가중치가 모두 똑같이 갱신된다는 것이다. 따라서 가중치들은 같은 초기값에서 시작하고 갱신이 된 후에도 여전히 같은 값을 유지하기 때문에, 이렇게 되면 가중치를 여러 개 갖는 것이 의미가 없다. 따라서 초기값은 무작위로 설정해야 한다.

12. 오버피팅을 방지하는 방법에는 뭐가 있는가?

a. 가중치 감소

간단히 말해서, 가중치 값을 작게 하여 오버피팅이 일어나지 않게 하는 것이다.

b. 드롭아웃

뉴런을 임의로 삭제하면서 학습하는 방법이다. 훈련 때 은닉층의 뉴런을 무작위로 골라 삭제한다. 훈련 때는 데이터를 훌릴 때마다 삭제할 뉴런을 무작위로 선택하고, 시험 때는 모든 뉴런에 신호를 전달한다. 단, 시험때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱해서 출력한다.

13. 각종 임베딩 방법을 비교해보기

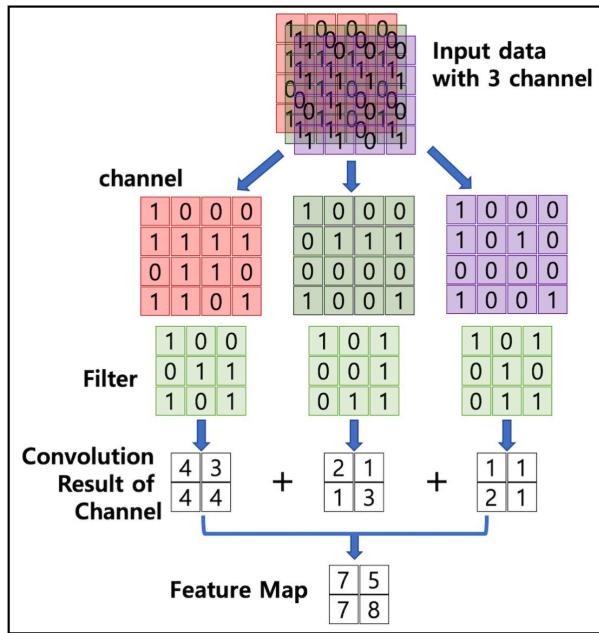
14. CNN 동작 방법 요약

CNN(Convolutional Neural Network)은 이미지의 공간 정보를 유지하면서 인접 이미지와의 특징을 효과적으로 인식하고 강조하는 방식으로 이미지의 특징을 추출하는 부분과 이미지를 분류하는 부분으로 구성됩니다. 특징 추출 영역은 **Filter** 를 사용하여 공유 파라미터 수를 최소화하면서 이미지의 특징을 찾는 **Convolution** 레이어와 특징을 강화하고 모으는 **Pooling** 레이어로 구성됩니다.

CNN은 Filter의 크기, Stride, Padding 과 Pooling 크기로 출력 데이터 크기를 조절하고, 필터의 개수로 출력 데이터의 채널을 결정합니다.

CNN는 같은 레이어 크기의 Fully Connected Neural Network와 비교해 볼 때, 학습 파라미터 양은 20% 규모입니다. 은닉층이 깊어질수록 학습 파라미터의 차이는 더 벌어집니다. CNN은 Fully Connected Neural Network와 비교하여 더 작은 학습 파라미터로 더 높은 인식률을 제공합니다

채널: 컬러이미지는 3개의 RGB 채널로 구성되는데, 이 때의 채널을 가리킴.
높이가 39 픽셀이고, 폭이 31 픽셀인 컬러 사진은 (39, 31, 3)과 같은 형상으로 표현한다. 입력 데이터가 여러 채널을 가질 경우 필터는 각 채널을 순회하며 합성곱을 계산한 후 채널별 피쳐맵을 만든다(즉, 세 개의 피쳐맵이 만들어지는 것이다) 그 후 각 채널별 피쳐맵을 합산하여 최종 피쳐맵으로 반환한다.
입력데이터의 채널 수와 상관없이 **필터별로 1개의 피쳐맵**이 만들어진다.
입력데이터에 적용한 필터 개수는 결국 출력데이터인 feature map의 개수가 된다.



필터: 이미지의 특징을 찾아내기 위한 공용 파라미터이다. 필터를 커널이라고도 한다. CNN에서 학습의 대상이 되는 것이 필터 파라미터이다.

패딩: convolution 레이어에서 filter 와 stride 의 작용으로 feature map 크기는 입력데이터보다 작다. Convolution 레이어에서 출력 데이터의 크기가 줄어드는 것을 방지하는 방법이 패딩이다. 패딩은 입력 데이터의 외각에 지정된 픽셀만큼 특정 값으로 채워 넣는 것을 의미한다. 보통 0 으로 채운다.

풀링 레이어: convolution 레이어의 출력 데이터를 입력으로 받아, 1) 출력 데이터(feature map == activation map)의 크기를 줄이거나 특정 데이터를 강조하는 용도로 사용된다. 풀링 레이어만의 특징은 1) 학습 대상 파라미터가 없으며 2) 풀링 레이어를 통과하면 행렬의 크기가 감소한다는 것 3) 풀링 레이어를 통해 채널수 변경 없음. 이다

15. 그냥 RNN 과 bidirectional RNN 의 차이 설명하기

Hidden state 가 마치 RNN 의 메모리처럼 동작한다. 과거의 input 을 전부 담아놓는 메모리같은 역할을 한다.
 $\text{New hidden state} = \tanh(\text{hidden state} + \text{current input})$
 출력층 = 비선형활성화(new hidden state)

16. RNN 과 LSTM / GRU 설명하기

Hidden state 가 마치 RNN 의 메모리처럼 동작한다. 과거의 input 을 전부 담아놓는 메모리같은 역할을 한다.

$\text{New hidden state} = \tanh(\text{hidden state} + \text{current input})$
 출력층 = 비선형활성화(new hidden state)

RNN 은 backpropagation 과정에서 vanishing gradient problem 을 겪는다. 왜?

Vanishing gradient problem 이 문제가 되는 이유

gradient 는 신경망의 가중치의 값을 업데이트하는 데 사용된다. Gradient 가 작다면, 갱신되는 정도도 작다.

$$\text{new weight} = \text{weight} - \text{learning_rate} * \text{weight_gradient}$$

$$2.0999 = 2.1 - 0.001$$

그래서 앞쪽에 있는 레이어들은 학습이 제대로 안된다. 그래서 자연어 처리의 경우 앞 쪽에 있는 단어들이 최종 결과에 제대로 반영이 안될 수 있다.

LSTM 이 vanishing gradient 문제를 해결하는 방법

Forget gate 와 input gate 두 개를 사용해서 cell state 를 계산한다.

forget gate 는 과거 정보 중에서 중요한 것은 잊어버리지 않으려고 계속 킁한다.

sigmoid(과거 hidden state, current input)

Sigmoid 를 쓴다. Sigmoid 는 0 과 1 사이의 값을 주는데, 유용한 이유는 0 을 주어서 0 을 곱하면 그 수치는 사라지는 거고, 1 을 주어서 1 을 곱하면 그 수치는 보존되는 거다.

Input gate 현재 정보 중에서 중요한 것을 저장한다.

sigmoid(과거 hidden state, current input) *tanh(과거 hidden state, current input)

위와 같이 계산함으로써 sigmoid 는 tanh 중에서 어떤 정보가 중요한지를 결정하게 된다.

New cell state = cell state * forget gate + input gate

forget gate 와 곱함으로써, 0 인 값들은 cell state 에서 지워진다.

Output gate: 다음과 같이 새로 계산될 hidden state 를 결정한다.

sigmoid(과거 hidden state, current input) * tanh(current cell state) => 이게 바로 현재의 hidden state 가 된다. 현재의 hidden state 와 cell state 는 다음 time step 으로 전달된다.

GRU 가 vanishing gradient 문제를 해결하는 방법

Update gate 와 reset gate 를 둔다. Update gate 는 LSTM 의 forget, input gate 와 같다.

reset gate 는 과거 정보가 얼마만큼 지워져야하는지를 결정한다.

17. Seq2seq (encoder - decoder)

인코더 아키텍처와 디코더 아키텍처의 내부는 사실 두 개의 RNN 아키텍처입니다. 입력 문장을 받는 RNN 셀을 인코더라고 하고, 출력 문장을 출력하는 RNN 셀을 디코더라고 합니다. 물론, 성능 문제로 인해 실제로는 바닐라 RNN 이 아니라 **LSTM 셀** 또는 **GRU 셀**들로 구성됩니다. 우선 인코더를 자세히보면, 입력 문장은 단어 토큰화를 통해서 단어 단위로 쪼개지고 단어 토큰 각각은 RNN 셀의 각 시점의 입력이 됩니다. 인코더 RNN 셀은 모든 단어를 입력받은

뒤에 인코더 RNN 셀의 마지막 시점의 은닉 상태를 디코더 RNN 셀로 넘겨주는데 이를 컨텍스트 벡터라고 합니다. 컨텍스트 벡터는 디코더 RNN 셀의 첫번째 은닉 상태로 사용됩니다.

주의할 점

training 시 디코더의 첫번째 입력은 아래에서 들어오는 target 단어와 왼쪽에서 들어오는 컨텍스트 벡터이다.

테스트 과정에서

디코더는 초기 입력으로 문장의 시작을 의미하는 심볼 <sos>가 들어갑니다. 디코더는 <sos>가 입력되면, 다음에 등장할 확률이 높은 단어를 예측합니다. 첫번째 시점(time step)의 디코더 RNN 셀은 다음에 등장할 단어로 je 를 예측하였습니다. 첫번째 시점의 디코더 RNN 셀은 예측된 단어 je 를 다음 시점의 RNN 셀의 입력으로 입력합니다. 그리고 두번째 시점의 디코더 RNN 셀은 입력된 단어 je 로부터 다시 다음에 올 단어인 suis 를 예측하고, 또 다시 이것을 다음 시점의 RNN 셀의 입력으로 보냅니다. 디코더는 이런 식으로 기본적으로 다음에 올 단어를 예측하고, 그 예측한 단어를 다음 시점의 RNN 셀의 입력으로 넣는 행위를 반복합니다.

18. Teacher forcing 이란?

훈련 과정에서는 이전 시점의 디코더 셀의 출력을 현재 시점의 디코더 셀의 입력으로 넣어주지 않고, 이전 시점의 실제값을 현재 시점의 디코더 셀의 입력값으로 하는 방법을 사용할 겁니다. 그 이유는 이전 시점의 디코더 셀의 예측이 틀렸는데 이를 현재 시점의 디코더 셀의 입력으로 사용하면 현재 시점의 디코더 셀의 예측도 잘못될 가능성이 높고 이는 연쇄 작용으로 디코더 전체의 예측을 어렵게 합니다. 이런 상황이 반복되면 훈련 시간이 느려집니다. 만약 이 상황을 원하지 않는다면 이전 시점의 디코더 셀의 예측값 대신 실제값을 현재 시점의 디코더 셀의 입력으로 사용하는 방법을 사용할 수 있습니다. 이와 같이 RNN 의 모든 시점에 대해서 이전 시점의 예측값 대신 실제값을 입력으로 주는 방법을 교사 강요라고 합니다.

19. Seq2seq 의 한계와 attention 이 등장한 이유

Seq2seq 의 한계

첫째, 하나의 고정된 크기의 컨텍스트 벡터에 모든 입력 정보를 압축하려고 하니까 정보 손실이 발생한다. 둘째, RNN 의 고질적인 문제인 기울기 소실(Vanishing Gradient) 문제가 존재한다. 결국 이는 기계 번역 분야에서 입력 문장이 길면 번역 품질이 떨어지는 현상으로 이어질 수 있다. 이에 대한 대안으로 입력 시퀀스가 길어지면 출력 시퀀스의 정확도가 떨어지는 것을 보정해주기 위한 등장한 기법, 어텐션이 등장한다.

어텐션의 기본 아이디어

어텐션의 기본 아이디어는 디코더에서 출력 단어를 예측하는 매 시점(time step)마다, 인코더에서의 전체 입력 문장을 다시 한 번 참고한다는 것이다. 단, 전체 입력 문장을 전부 다

동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야 할 단어와 연관이 있는 입력 단어 부분을 좀 더 집중(attention)해서 보게 된다.

Global attention에 대해 설명을 하겠다.

20. Attention 함수란:

어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구한다. 그리고 구해낸 이 유사도를 가중치로 하여 키와 맵핑되어있는 각각의 '값(Value)'에 반영해준다. 그리고 유사도가 반영된 '값(Value)'을 모두 가중합하여 리턴한다.

21. Attention의 단계

0. Encoder의 매 시점마다 hidden state를 따로 저장해둔다. (seq2seq 때는 이렇게 안했다.)
1. 각각의 hidden state마다 attention score를 계산해둔다.
 - 1-1. Decoder의 특정 시점의 hidden state와 encoder의 모든 시점의 hidden state를 dot product 계산해서 score를 도출한다.
2. Attention score를 softmax(확률값)로 바꾼다.
확률값으로 바꾸면 더 큰 값은 더 크게 극대화시키는 효과가 있다.
3. Softmax 된 score와 encoder의 모든 시점의 hidden state를 곱셈한다.
4. 그 다음 다 값을 더한다. 이게 바로 context vector가 된다.
5. 그 다음 context vector를 decoder에게 준다.

22. Attention이 어떻게 동작하는가? -> BackPropagation

23. 임베딩의 종류

임베딩은 static word embedding과 contextualized word embedding으로 나눌 수 있다. 여기서 static word embedding에 context가 안 들어갔다는 말은, 문맥에 따라서 bank라는 단어가 서로 다른 의미를 가질 수 있음이 반영이 안되었다는 거다. (문맥에 따라서 bank의 의미가 은행일 수도, 강둑일 수도) glove, fasttext 모두에서 bank는 은행, 강둑 등 의미에 상관없이 하나의 벡터값을 갖는다) 그러나 contextualized word embedding을 사용하면 상황에 따라서 다른

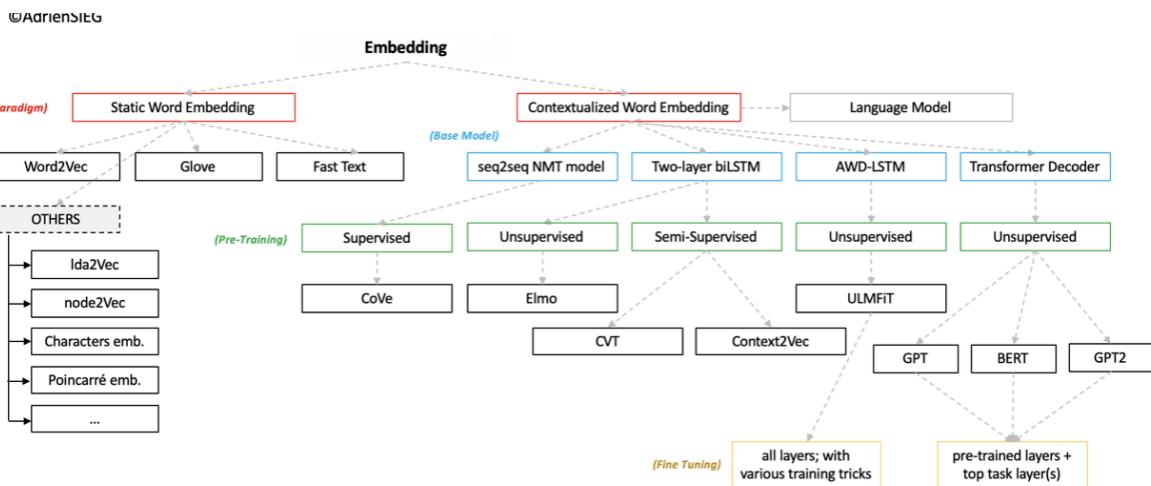
벡터값을 갖게 된다.

static word embedding(word2vec)은 simple, shallow 하다. layer 를 한 개만 사용한다.

dynamic layer(BERT)는 레이어를 12 개 쌓아서 각각 layer 마다 다양한 언어표현들이 학습이 되었다.

24. 임베딩 관점에서 좋은 모델이란?

- a. Bank 잘 포착
- b. Syntactic approach 계층적인 관계가 포착
- c. Semantic approach (의미적인거)
- d. Pragmatic approach (실용적인거)



25. Transfer learning 이란?

각각의 데이터를 갖고, 성능이 얼마나 나오냐?

각각의 분류를 분류에 따라서 모델을 각자 만들고 있다.

태스크마다 서로 다른 모델을 써야 하는데,

아주 큰 데이터에 대해서 학습을 해놓고, 학습된 걸 가놓고 개별적인 태스크에 들어가서는 성능을 잘 하겠지. 이것의 근간이 BERT. 미리 학습된 걸 가져다가 단순히 사용만 하면 되는 방향으로 바뀌게 되었다. multi-lingual transfer learning.

왜 transfer learning 을 쓰는가? 원래는 컴퓨터 비전에서는

초기화 했을 때보다, 외부의 워드 임베딩을 가져다가 사용하는 것이

성능이 더 좋더라. 큰 데이터를 갖고 모든 태스크에

26. Transformer 의 전체적인 구조를 설명해라: 여기도 강 attention score 계산하는 거다.

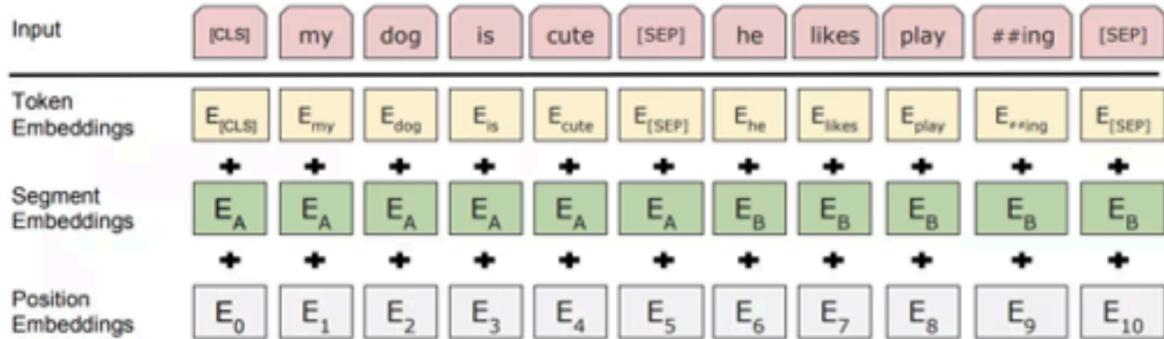
입력의 문장의 길이는 제한되고, 한 번에 두 문장 씩 들어간다. 세 개가 들어가면 안 된다.

트랜스포머의 구조: input 단계 / 인코더 6 개 쌓인거 / 디코더 6 개 쌓인거

27. Transformer 의 문장이 인코더에 들어가기 전 거치는 전처리 과정을 설명해라

Input -> tokenization(wordpiece 를 사용할 것!) -> numericalization -> token

embedding -> padding -> segment embedding -> position embedding



wordpiece 알고리즘

tokenization 을 할 때 단어 기준으로 토큰을 만들어내는 것이 아니라, playing 같은 단어의 경우 play 와 ##ing, 즉, 두 가지 토큰이 생겨나게 된다. 이렇게 하는 이유는 최대한 unknown 토큰을 없애기 위함이다.

segment embedding

transformer 에는 문장 두 개가 들어온다. 해당 토큰이 첫번째 문장에 속하는지, 두 번째 문장에 속하는지를 알려주는 임베딩이다.

Position embedding

순서의 개념을 부여하기 위한 임베딩이다. RNN 이나 LSTM 에는 recurrent unit 이 있지만, 트랜스포머에는 없다. 그래서 포지션에 대한 정보가 반영이 안되어서 이걸 꼭 주기 위해서 포지션 임베딩을 한다.

28. 트랜스포머의 인코더를 설명하시오

인코더의 특징: multi head self-attention 과 feed forward neural net

Attention: 어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구한다. 그리고 구해낸 이 유사도를 가중치로 하여 키와 맵핑되어있는 각각의 '값(Value)'에 반영해준다. 그리고 유사도가 반영된 '값(Value)'을 모두 가중합하여 리턴한다.

Seq2seq 에서 어텐션을 사용할 경우

Query: t 시점의 디코더 셀에서의 은닉 상태

Key: 모든 시점의 인코더 셀의 은닉 상태들

Values: 모든 시점의 인코더 셀의 은닉 상태들

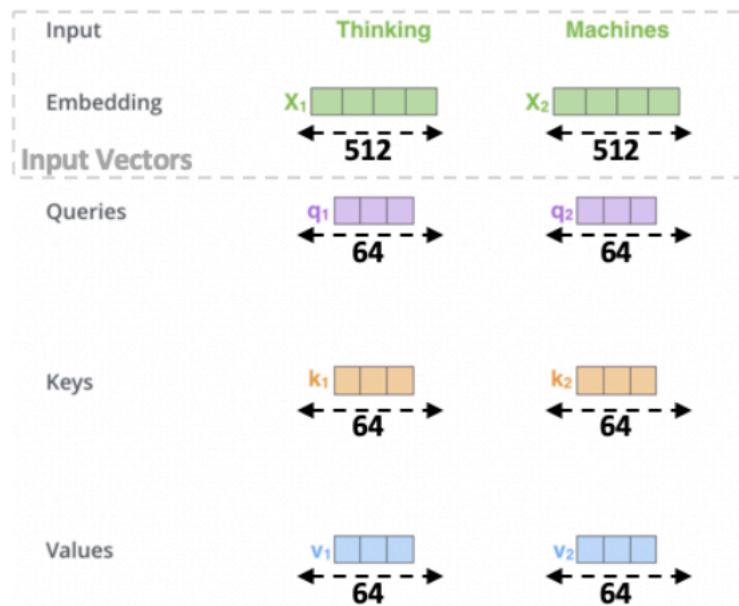
multi head self-attention에서 Q, K, V의 정의:

Query: 입력 문장의 모든 토큰 벡터들 (교수님 용어로는 주목하는 거)

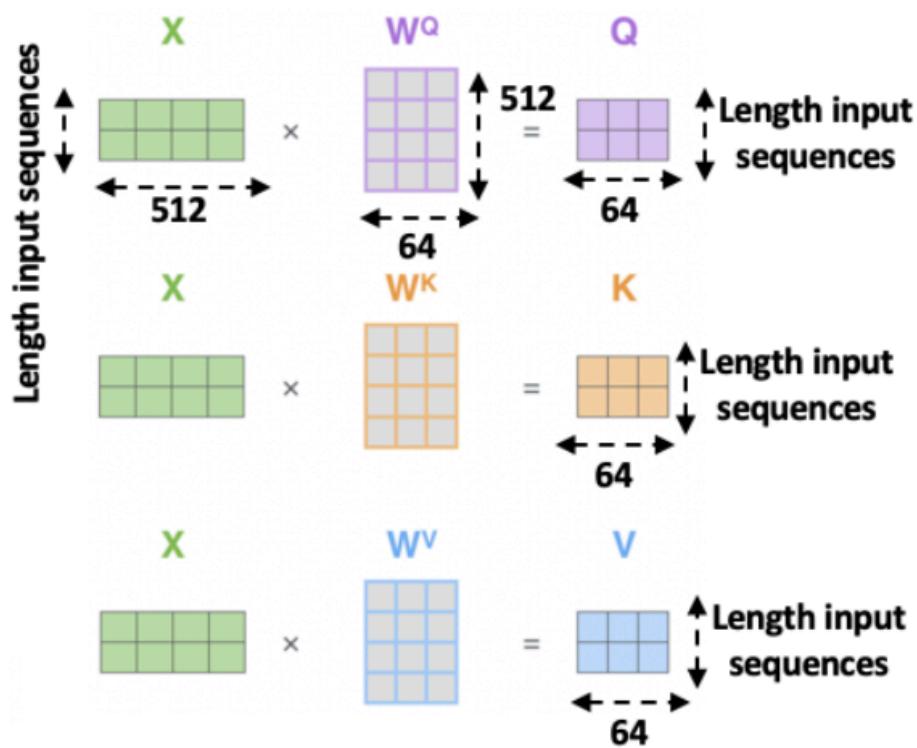
Key: 입력 문장의 모든 토큰 벡터들 (교수님 용어로는 주목 받는 거)

Values: 입력 문장의 모든 토큰 벡터들

** 인코더의 초기 입력은 512 차원을 갖지만, 8 개이 multi head 를 쓰기 때문에 Q, K, V 모두 64 차원의 값이 된다. (아래와 같이)



“Hello World”라는 입력이 있다면, input 의 길이가 2 이고, 다음과 같이 Q, K, V 벡터를 위한 가중치의 형상이 (512, 64)가 된다.



attention score 계산하는 방법:

4

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V$$

$$\text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i = Z$$

Z Length input sequences

1. Query 와 transposed K 를 dot product 로 연산
2. 1 의 값을 루트 K 의 값으로 normalize 한다.
3. 2 를 softmax
4. 3 과 Value 를 곱한다
5. 4 의 값을 가중합한다

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$	x_{13}	$x_{13} * v_3$	

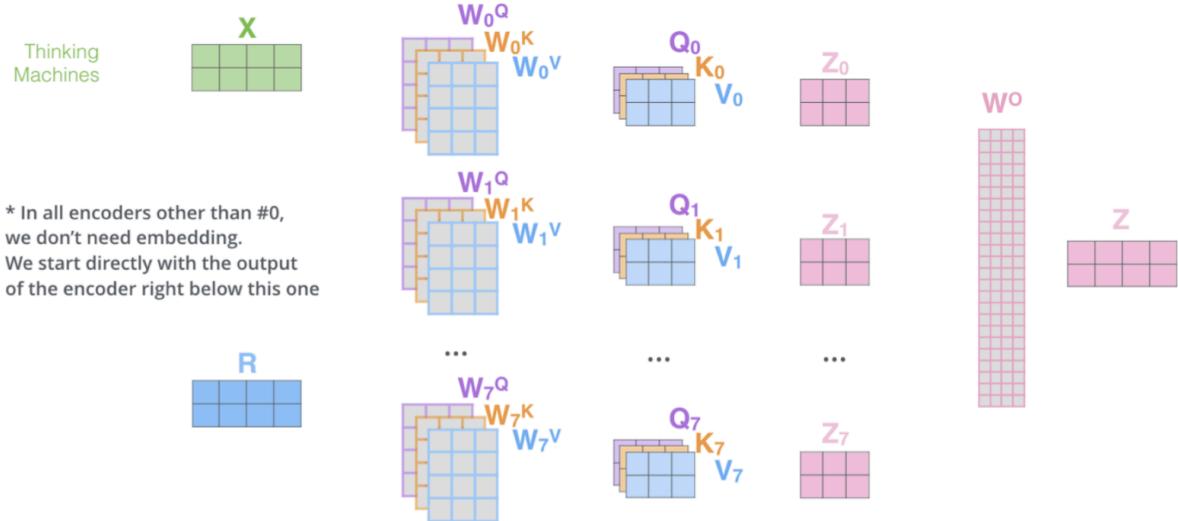
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum [#]
Action		k_1	v_1	$q_2 \cdot k_1$	$q_2 \cdot k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 \cdot k_2$	$q_2 \cdot k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 \cdot k_3$	$q_2 \cdot k_3 / 8$	x_{23}	$x_{23} * v_3$	

Dot product 로 Query 와 Key 를 연산하는 것의 의의:

dot product 은 코사인 유사도를 구하는 것으로 즉, 얼마나 유사하냐를 본다는 것이다.
K 의 차원 값으로 normalize 한 뒤, softmax 그다음 softmax * value 로 한다.

진정한 multihead 로 바꾸면 다음과 같아.

- 1) This is our input sentence* each word*
- 2) We embed
- 3) Split into 8 heads. We multiply X or R with weight matrices
- 4) Calculate attention using the resulting Q/K/V matrices
- 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer



feed-forward 단계:

add 후 normalize 한다. 평균이 0 되고 분산이 1 이 되게 한 다음 feedforward neural net 에 넣는다.

multi-head self-attention 의 장점:

Multi-head 를 쓴으로써 하나의 레이어에서 다양한 특징을 학습할 수 있도록 한다.

The Transformer uses **Multi-Head Attention**, which means it computes attention h different times with **different weight matrices**

H different time 이 가리키는 것이: Pragmatic 측면, semantic 측면, syntactic 측면 등등의 다양한 측면

29. Transformer 의 Decoder 를 설명하시오

핵심 구조는 masked multi-head attention / encoder-decoder attention / feed forward neural net

Masked multi-head attention

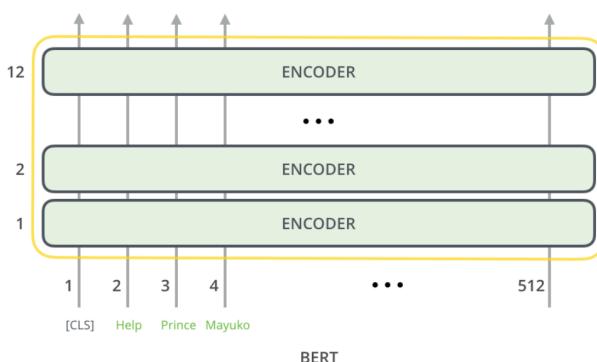
Cheating 을 하지 못하기 위해서 디코더에 input 이 들어오면, 자기꺼하고 자기 앞에 있는 것만 본다. 뒤에꺼는 보지 않는다. 이게 바로 masked multi-head attention. 자기꺼랑 이전꺼 중에 관련 있는 것만 본다. 나머지 값들은 마이너스 무한대로 정한다.

For example, if a sentence “**I play soccer in the morning.**” is given in the decoder, and we want to apply self-attention for “soccer” (let “soccer” be a query), we can only attend “**I**” and “**play**” but cannot attend “**in**”, “**the**”, and “**morning**”.

이렇게 해서 디코더에서 나온 output logit 들에 대해 softmax 를 통과하도록 한다.

30. BERT 와 transformer 의 차이

- 1) transformer 는 sequence to sequence 번역을 하기 위한 구조. 실제 더 많이 사용하는 것은 버트인데, 버트는 트랜스포머의 인코더만 가져다가 classification 등등의 더 많은 다른 업무에 사용한다.
- 2) 트랜스포머에서는 인코더가 6 개, hidden unit 는 512 개, attention 의 multi head 는 8 개
버트는 기본 버전이 인코더 12 개, hidden unit 이 768 개, header 12 개
라지 버전이 인코더 24 개, hidden unit 이 1024 개, header 가 16 개



맨 앞에 <cls> 넣는 거, 그리고 512 크기의 벡터로 표현하는 거, 문장 두 개밖에 못 들어가는 거 다 똑같다.

31. BERT의 특징(Bidirectional Encoder Representational from Transformer)

- 1) 트랜스포머에서 인코더 블록만 떼어 내와서, 많은 데이터에 대해서 학습을 시킨 뒤, 여러가지 NLP 태스크에 사용한다.
- 2) Bi-directional transformer 를 활용하여 토큰의 좌우 컨텍스트를 다 참고할 수 있다.

ELMO: uses the concatenation of independently trained left-to-right and right-to-left LSTM (shallow bidirectional)

GPT: uses a left-to-right Transformer

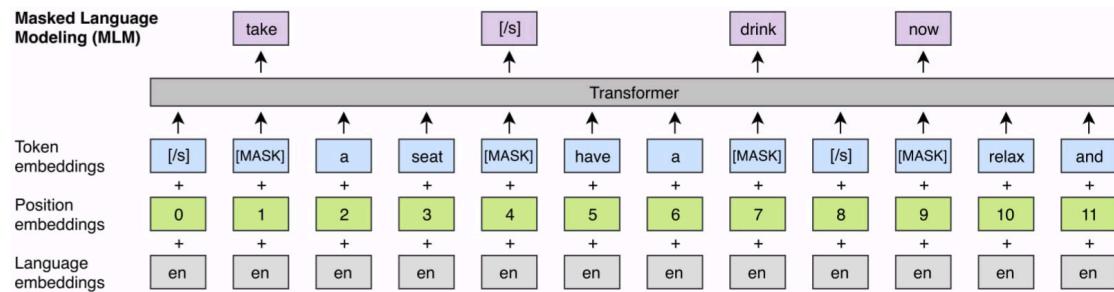
3) Masked language modeling

bidirectional deep transformer 를 학습할 수 있는 이유는 masked language modeling 덕분이다.

각각이 sequence 의 15%를 랜덤으로 뽑아서 마스크를 했다. 그 15%중에서도

- 80% of the time the words were replaced with the masked token [MASK]
- 10% of the time the words were replaced with random words
- 10% of the time the words were left unchanged

이런 식으로 진행했다.



가령 위와 같이 마스킹이 된 input sequence 가 주어졌을 때, 빈 자리의 take 를 맞추고, [/s]를 맞추고, drink 를 맞추고, now 를 맞추고.

4) Next Sentence Prediction

두 문장 씩 넣잖아. 그래서 뒷 문장이 맞는지 아닌지를 맞추는 작업을 하면서 데이터를 학습한다. 50%는 원래 페어인 두 문장을 넣고, 나머지 50 프로는 그냥 랜덤한 페어를 주고.

계산 문제

1. CNN에서의 출력 데이터 shape 문제

CNN은 이미지 특징 추출을 위하여 입력데이터를 필터가 순회하며 합성곱을 계산하고, 그 계산 결과를 이용하여 Feature map을 만듭니다. Convolution Layer는 Filter 크기, Stride, Padding 적용 여부, Max Pooling 크기에 따라서 출력 데이터의 Shape이 변경됩니다.

Convolution 레이어에서의 출력 데이터 크기 계산

H: 입력 데이터 높이

W: 입력 데이터 폭

FH: 필터 높이

FW: 필터 폭

S: Strid 크기

P: 패딩사이즈

$$\text{OutputHeight} = ((H + 2P - FH) / S) + 1$$

$$\text{OutputWeight} = ((W + 2P - FW) / S) + 1$$

(만약 convolution 이후 output size를 input size 그대로 유지하고 싶다면)

$$P: (K - 1)/2$$

학습파라미터 수 계산: 입력채널 수 * FH*FW*출력채널수

Pooling 레이어에서의 출력 데이터 크기 계산

W: 입력 데이터 폭 및 높이

K: 필터 사이즈

S: Strid

$$O = ((W - K) / S) + 1$$

보통 Pooling Size는 stride와 같은 크기로 만들어서 ($K == S$) 모든 요소가 한 번씩 풀링 되도록 한다. 그렇게 되면 위의 식이 아래와 같다는 것이 이해가 되지?

$$\text{OutputRowSize} = \text{InputRowSize} / \text{PoolingSize}$$

$$\text{OutputColumnSize} = \text{InputColumnSize} / \text{PoolingSize}$$

layer	input channel	Filter	output channel	Stride	Pooling	활성함수	Input Shape	Output Shape	파라미터 수
Convolution Layer 1	1	(4, 4)	20	1	X	relu	(39, 31, 1)	(36, 28, 20)	320
Max Pooling Lyaer 1	20	X	20	2	(2, 2)	X	(36, 28, 20)	(18, 14, 20)	0
Convolution Layer 2	20	(3, 3)	40	1	X	relu	(18, 14, 20)	(16, 12, 40)	7,200
Max Pooling Lyaer 2	40	X	40	2	(2,2)	X	(16, 12, 40)	(8, 6, 40)	0
Convolution Layer 3	40	(2, 2)	60	1	1	relu	(8, 6, 40)	(6, 4, 60)	21,600
Max Pooling Lyaer 3	60	X	60	(2, 2)	60	X	(6, 4, 60)	(3, 2, 60)	0
Convolution Layer 4	60	(2, 2)	80	1	1	relu	(3, 2, 60)	(2, 1, 80)	19,200
Flatten	X	X	X	X	X	X	(2, 1, 80)	(160, 1)	0
fully connected Layer	X	X	X	X	X	softmax	(160, 1)	(100, 1)	160,000
합계	X	X	X	X	X	softmax	(160, 1)	(100, 1)	208,320

$$7200 = \\ 20 * 40 * 3 * 3$$

CNN 의 입력 vs FNN 의 입력

CNN Input: (1, 28, 28)

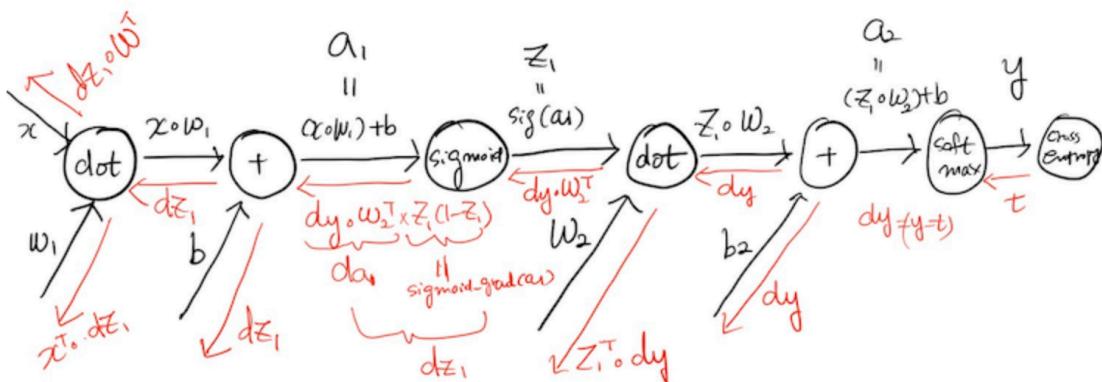
FNN Input: (1, 28*28)

- batch_size, n_iters 주어졌을 때 num_epochs 계산하는 방법

$$\text{num_epochs} = \text{n_iters} / (\text{len}(\text{train_dataset}) / \text{batch_size});$$

- backpropagation 관련한 거

Two Layer Net의 gradient \rightarrow backward



4. RNN, LSTM 형상 이해할 때 주의할 점 (batch_first=True 라고 전제)

RNN의 output:

Output: [batch_size, time_step, hidden_dim]

FC에 넣을 때는 마지막 hidden_dim 만 넣게 되므로

out = self.fc(out[:, -1, :]) 와 같이 적게 된다.

RNN->FC의 output:

Output: [batch_size, output_dim]

여기서 output_dim 은 분류 클래스의 개수가 된다.

5. RNN, LSTM, SEQ2SEQ 형상 비교해서 이해

이 부분 코드는 rnn 모델 정의할 때 batch_first=True 로 설정하지 않아서,

기존의 다른 RNN, LSTM 코드에서는 input 이 [batch_size, time step, input_dim(emb_dim)]

이었는데 seq2seq 코드에서는 input 형상이 [time_step, batch_size, input_dim(emb_dim)]

이렇게 되어있다. 이거 주목!

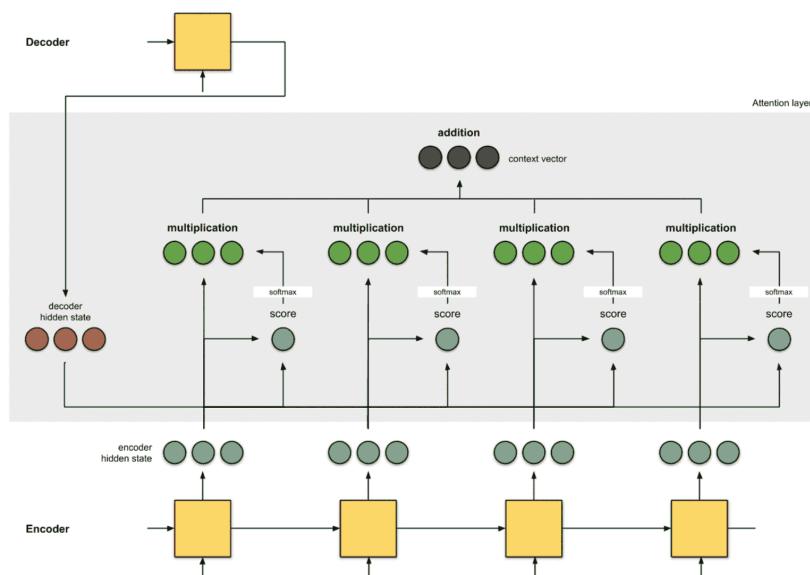
Output의 경우에도:

Batch_first=True: [batch_size, time_step, output_dim]

Batch_first=False: [time_step, batch_size, output_dim]

6. attention score 계산하기

1. Encoder의 매 시점마다 hidden state를 따로 저장해둔다. (seq2seq 때는 이렇게 안했다.)
2. 각각의 hidden state마다 attention score를 계산해둔다.
 - 1-1. Decoder의 특정 시점의 hidden state와 encoder의 모든 시점의 hidden state를 dot product 계산해서 score를 도출한다.
3. Attention score를 softmax(확률값)로 바꾼다.
확률값으로 바꾸면 더 큰 값은 더 크게 극대화시키는 효과가 있다.
4. Softmax된 score와 encoder의 모든 시점의 hidden state를 곱셈한다.
5. 그 다음 다 값을 더한다. 이게 바로 context vector가 된다.
6. 그 다음 context vector를 decoder에게 준다.
7. 위 과정을 decoder의 모든 시점에서 진행한다.



입력시점에서의 decoder_hidden = [10, 5, 10]

encoder score score[^] alignment

1	2	3	4	
[0, 1, 1]	15	0	[0, 0, 0]	
[5, 0, 1]	60	1	[5, 0, 1]	
[1, 1, 0]	15	0	[0, 0, 0]	
[0, 5, 1]	35	0	[0, 0, 0]	

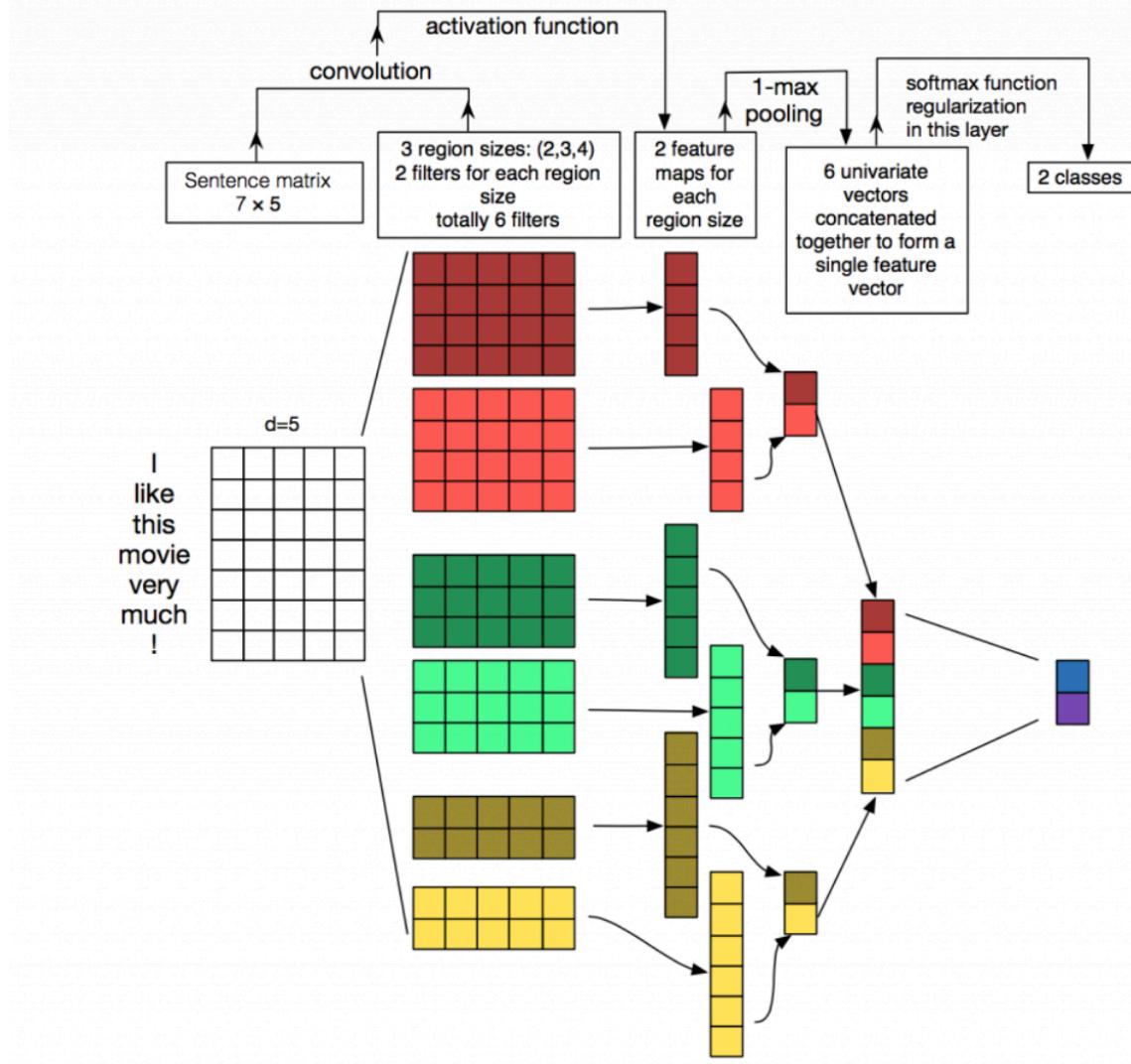
5

$$\text{context} = [0+5+0+0, 0+0+0+0, 0+1+0+0] = [5, 0, 1]$$

유용할 도식

1. CNN 을 NLP 에 적용시킬 방법

filter 하나는 n-gram 의 특징 하나를 확인할 수 있는데, filter 의 수를 늘리면 더 다양한 특징을 확인할 수 있다.



2. DeepLearningWizard-FNN 쪽 도식

- numpy 형상에 관한 것

```
A = np.array([1, 2, 3, 4])
```

np.ndim(A) 출력: 1

-> 배열의 차원수를 구한다

np.shape(A) 출력: (4,)

-> 배열의 형상을 구한다

즉, 배열의 순서대로 0 차원에 있는 원소 개수, 1 차원에 있는 원소 개수, … n-1 차원에 있는 원소 개수

```
B = np.array([[1, 2], [3, 4], [5, 6]])
```

np.ndim(B) 출력: 2 (대괄호 개수가 2 개)

→ 배열의 차원수를 구한다

np.shape(B) 출력: (3, 2)

→ 배열의 형상을 구한다.

즉, 배열의 순서대로 0 차원에 있는 원소는 3 개구, 1 차원에 있는 원소개수는 2 개라서, 형상이 (3, 2)가 된다

Np.dot(A, B) 는 우리가 흔히 아는 행렬 곱셈 해주는 것.

```
dY = ([[1, 2, 3], [4, 5, 6]])
```

Np.sum(dY, axis = 0) 출력: ([5, 7, 9])

Np.sum(dY, axis = 1) 출력: ([6, 15])

CNN resizing 할 때: **view(out.size(0), -1)**

out 의 원래 사이즈는: (100, 32, 7, 7) 즉: (배치수, 채널수, FH, FW)

```
# out.size(0): 100  
# 원하는 사이즈는: (100, 32*7*7) (CNN에서 FC로 바꿀거라서)
```

```
out = out.view(out.size(0), -1)  
이것은 다음과 같다  
out = out.view(100, -1)  
즉, “행은 100으로 해주고, 열은 알아서 맞춰주세요”
```

np.squeeze()란?

크기가 1인 차원을 없애준다.

ex)

```
input = [[1, 1, 1, 1]]  
input.shape 출력: [1, 4]  
input = input.squeeze()  
input은 [1, 1, 1, 1]이 된다  
input.shape 출력: [4]
```

np.unsqueeze(0)

원하는 위치에 차원 1을 더해준다.

ex)

```
input = [1, 1, 1, 1]  
input.shape 출력: [4]  
input = input.unsqueeze(0)  
input.shape은 [1, 4]가 되고  
input은 [[1, 1, 1, 1]]가 된다
```