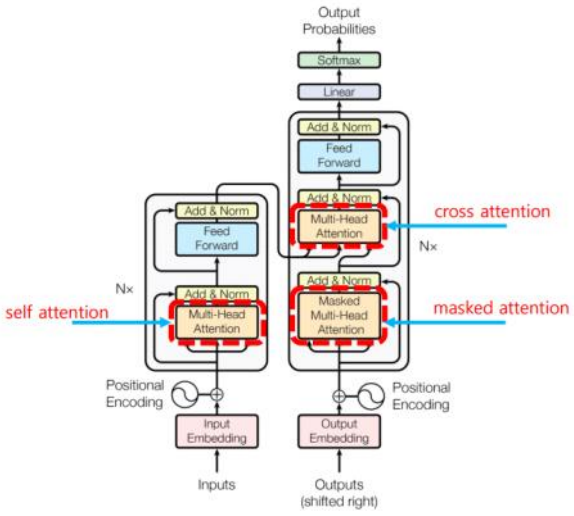
transformers model

2023년 10월 11일 수요일 오전 6:39

Transformers model



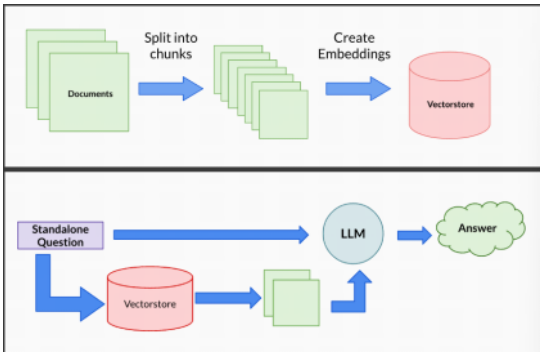
화면 캡처: 2023-10-11 오전 9:41

전체적인 흐름

token -> id-> embedded vector-> transformer-> output vector->id->token

encoder part

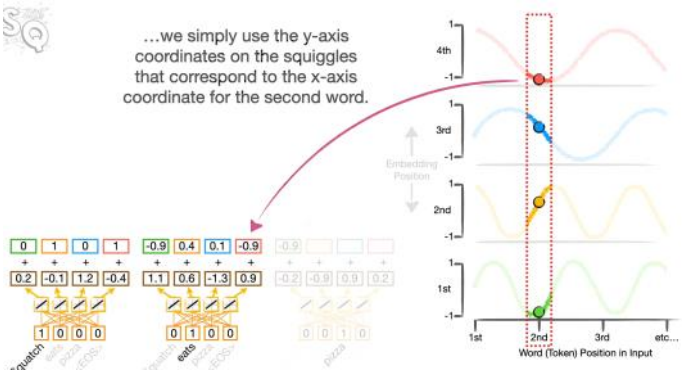
input embedding(positional embedding)



Embedding

token(chunk)[langchain textSplitter]로 여러 token(chunk)이 생길것이다. 이런 chunk를 embedding으로 만들것인데 이게 무슨 의미냐면 embedding으로 만들면 이러한 chunk를 vector representation으로 표현할수있다 즉 vector space로 영역을 옮길수가 있다 즉 선형대수 계산 가능 유사도 계산등 다양한 계산 할수있다는 의미이다
여러 개로 나눈 chunks와 embedding(openaiembedding,huggingfaceEmbedding,,,) 을 가지고 vector space 만드는게 chroma,pinecone,FASSI등이 있다
embedding 인자로 chunks넘겨서 왜 사용 안하냐면 chunks가 너무 많기 때문에 이러한 chunks를 주어진 embedding사용해서 vector store만들어주는 tool이 필요 하기 때문에 사용하는것이다

transformer에서 embedding을 word vec embedding했다고 알고있는데 CBOW, Skip Gram 같은 방식도 있다 어쨌든 입력으로 들어올때 text가 embedding된다는것만 알고 있으면 된다

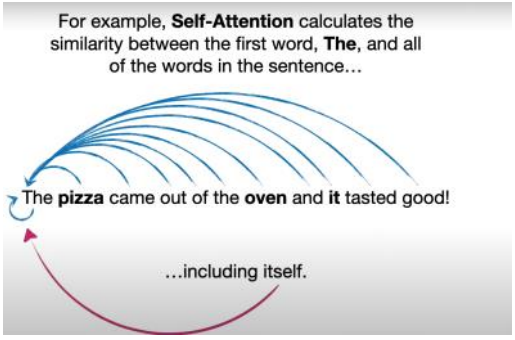


positional encoding
여러 chunk가 있으면 문장의 순서를 모른다 문장은 순서가 중요하다 같은 단어로 이루어진 문장이라도 순서에 따라 전혀 다른 의미가 될수있기 때문에 순서를 알아야 하는데 모델한테 이러한 고유 순서를 알려주는 부분이 positional encoding 부분이라고 보면 된다 어떤식으로 하냐면 sin,cos주기를 이용해서 문장의 순서를 부여 한다 만약 chunk 가 4개라고 하면 sin,cos,sin,cos.. 이런식으로 위에 처럼 있고 빨간색 상자 처럼 주기의 정보를 가져 온다 이렇게 하면 문장의 unique한 sequence를 유지할수있다

최종적으로 embedding 과 positional encoding이 element wise하여 처리 된다

self attention (multi head attention)

attention은 하나의 연산일 뿐이다 가중치가 학습하는게 아님



자신의 chunk도 포함해서 다른 chunk간의 dot product를 통해 유사도계산을 한다

query,key,value에 대한 개념이 나온다

- Query
  - 자신을 포함해서 유사도를 찾고 싶은 대상
- Key
  - 모든 chunks들..? 왜냐하면 자신 포함해서 유사도 계산을 하니까.
- Value
  - query를 대표하는 값

ex_) let's go에 대해 생각해보면 let's에 대한 pos + embedding elementwise 결과있을것이다(go에 대해서도 마찬가지)

let's에 대해 query만들면 자기 자신에 대한 유사도 또한 계산 해야 하기 때문에 let's Key, go Key가 생성이 된다

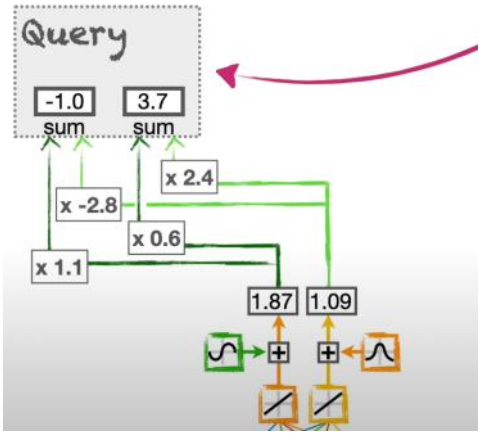
이때 key,value는 다시 사용을 한다 즉 대상에 대한 query만 바뀌고 key,value에 대해서는 재사용을 한다

Q : 입력으로 pos+emb elementwise한 값이 이미 있다(입력하는 문장순서 인지하고있다는의미) 그럼에도 구지 query,key, value 3가지로 나누어서 attention계산하는 이유는 무엇인가

A : 입력을 그대로 사용하면 하나의 chunk에 대한 중요도 파악이 힘들것 같다 즉 그냥 입력만 그대로 사용했을 경우 이 chunk가 핵심인지 아닌지 알수없다는것이다 self attention을 통해 각각의 chunk의 중요도를 반영할수가 있게 되었다 그렇기 때문에 문장 이해하는데 도움이 된다

Q,K,V는 기본적으로 같은 matrix에 대해서 다른 가중치를 곱한거니까 기본적으로 같은것을 표현한다 다만 적용되는 가중치가 다른것이다
밑에서 설명한게 이 질문을 포함하는것 같다

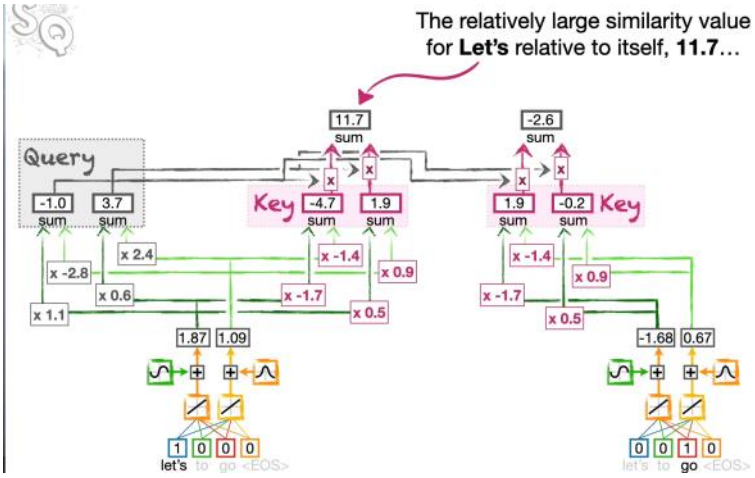
query와 key는 크기가 서로 같아야 하지만 value는 크기 달라도 상관없다
그래서 QK^t인것이다 Q,K가 서로 같은 크기이기 때문에 dot product를 하려면 key를 transpose해야 한다



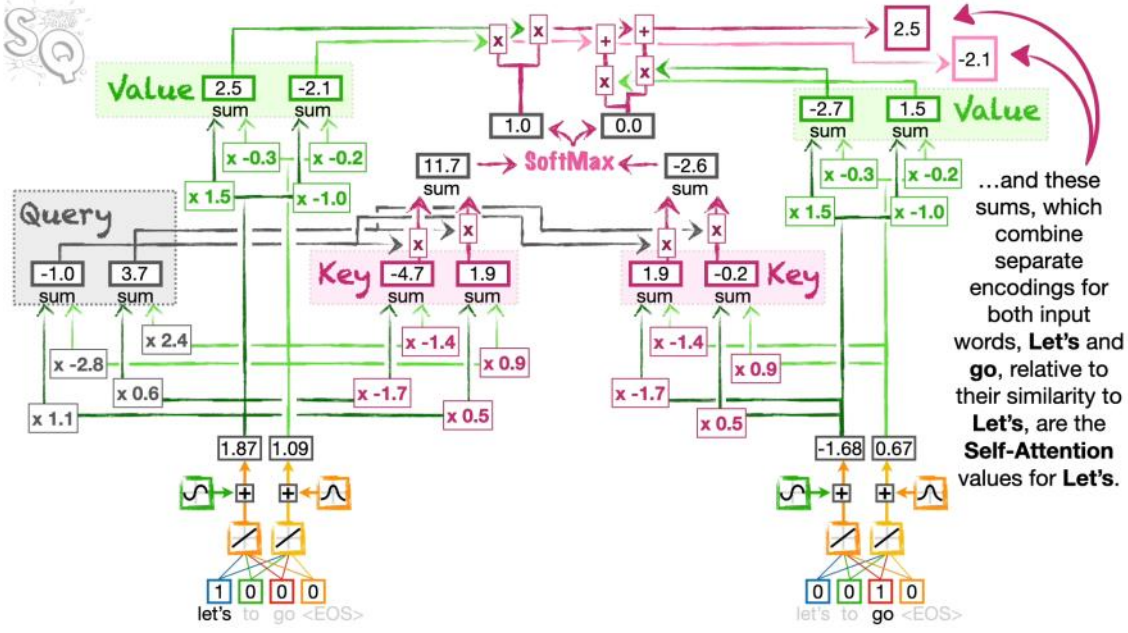
다음 그림은 self attention할때 query 계산하는 그림이다 let's go 두개에 대해 position embedding생성된것에 pair of weight 곱한거가 된다

attention 연산은 가중치가 없다고 했는데 qkv에 가중치 곱한건 무엇인가?

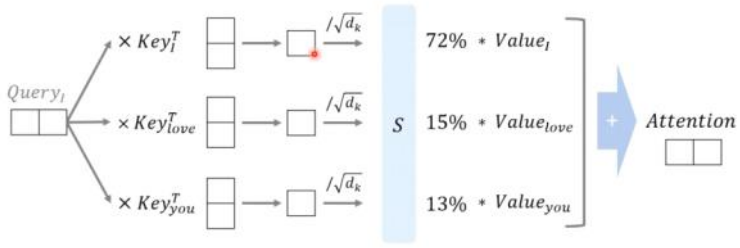
multihead attention 부분에서 가중치 곱한 Q,K,V에 대해 Q',K,'V'를 가진다 그리고 이렇게 곱해진 가중치는 입력 크기와 같게 하기 위해 embed dim*embed dim matrix size의 가중치를 가진다
결국 가중치를 곱해도 입력 차원을 유지한다 그리고 이렇게 곱해진 Q',K',V'은 설정한 multihead 개수 만큼 나눠진다(논문에서는 8개) 어쨌든 head개수만큼 나눠질때 가중치도 서로 다르게 가고 transformer는 이 가중치를 학습한다 즉 위에서 가중치곱한건 multihead까지 고려 한걸로 볼수있을것 같다



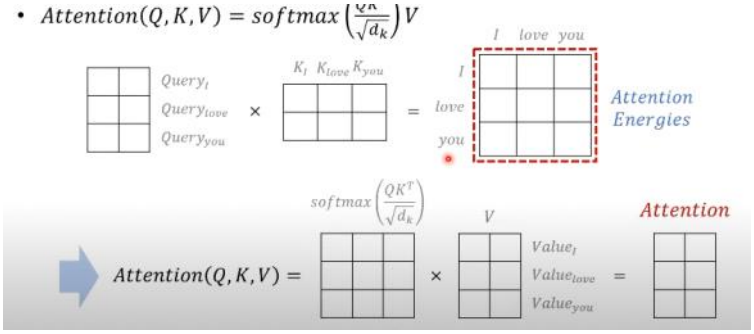
다음 그림은 let's에 대한 query에 대해 let's key, go key가 서로 dot product하는 과정이다
let's자신에 대한 key와도 dot product를 하기 때문에 자신에 대한 유사도 값은 높을수밖에 없다



- $Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$



- $Attention(Q,K,V) = softmax\left(\frac{QK}{\sqrt{d_k}}\right)V$



$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$MultiHead(Q,K,V) = Concat(head_1 \ \ldots head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

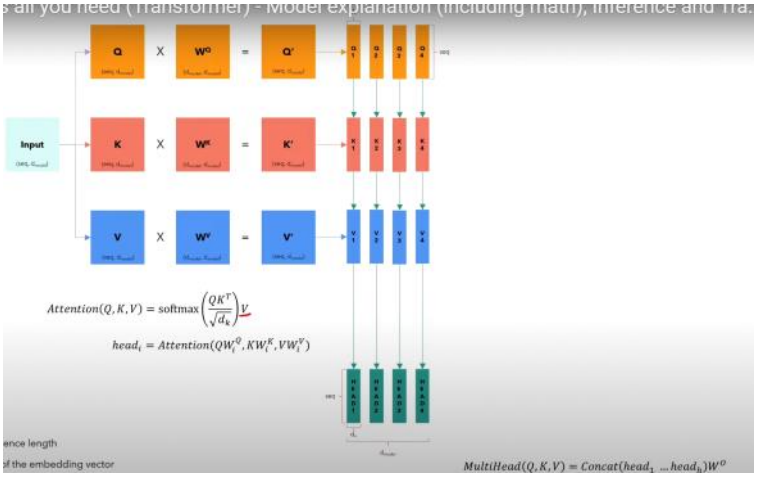
다음 그림과정을 이해하려면 Attention(Q,K,V)식에 대한 이해가 필요하다
QK^t에 대한 dot product구하고 이걸 token크기로 일종의 정규화 과정?을 진행하는것 같다
그리고 계산된 값에 대한 softmax구해준다(0-1까지 확률)

- let's에 대한 value, go에 대한 value (구체적으로 어떻게 구한건지는 잘 모르겠다)

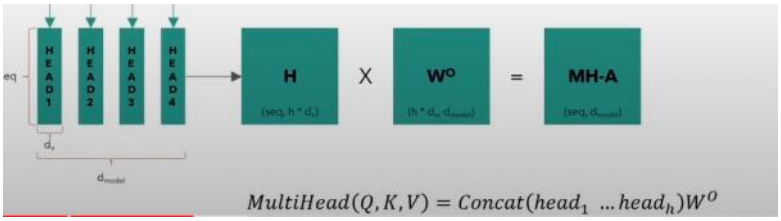
let's와 go에 대한 value를 구한것을 softmax결과 값을 곱해서 해당 query에 대한 attention을 구할수있다
위에 그림은 let's에 대한 self attention value이다

- 이때 입력 embedding과 matrix dimmention은 유지 한다
- attention은 가중치를 학습 하는 단계가 없다 그러면 transformer는 학습 하는 단계가 어디인가
  - linear layer and layer norm ,multihead attention 통해 학습을 한다

multihead attention



multihead attention은 self attention이 여러 개 있는것으로 보면 된다



- $MultiHead(Q,K,V)$를 수행한 뒤에도 **차원(dimension)이 동일하게 유지**됩니다.

$$Concat(head_1,\dots,head_h) = \underbrace{\boxed{head_1}\ \boxed{head_2}\ \boxed{head_3}\ \cdots\ \boxed{head_h}}_{d_{model} = d_v \times h}$$

$$MultiHead(Q,K,V) = \underbrace{\begin{bmatrix}\quad\\ \quad\end{bmatrix}}_{d_{model} = d_v \times h} seq\_len \times \underbrace{\begin{bmatrix}\quad\\ \quad\\ \quad\\ \quad\end{bmatrix}}_{d_{model}} d_{model}$$

head 개수 만큼 나누어서 계산 한 후 concate한다 -> 나눠진 head에는 서로 다른 가중치가 적용되기 때문에 병렬로 돌리는게 의미가 있다
위의 그림을 실제 크기에 예시를 들면 행렬 크기가 (6*512)입력이라고 하고 head가 8개라고 하면
H행렬은 6*(8*64)가 나온다 왜냐하면 head 개수만큼 우선 나눠지니까 8 * 64=512 이다 그리고 입력차원크기와 맞춰주기 위해 가중치가 곱해지는데
차원이 6*512로 유지가 되어야 하니까 가중치 행렬의 크기는 ((8*64),512)가 된다 그렇게 되면 결국 입력 차원을 유지할수있다

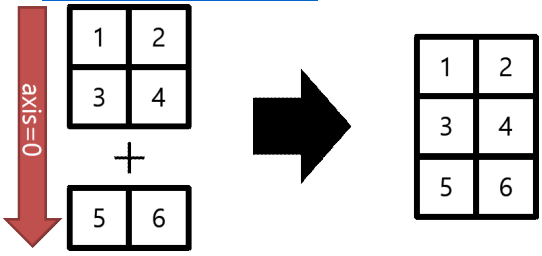
concate & stack

넘파이 알고 쓰자 - stack, hstack, vstack, dstack, column_stack — Everyday Image Processing (tistory.com)

- **np.concatenate((a1,a2...),axis=0)**
  - each row : axis=0
  - each colum : axis=1
  - 합칠 axis제외하고 나머지 shape는 전부 동일해야 한다
  - EX)

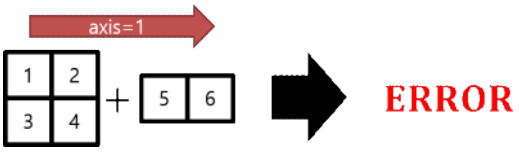
    a = np.array([[1, 2], [3, 4]]) *# a.shape=(2, 2)*
    b = np.array([[5, 6]]) *# b.shape=(1, 2)*

    출처: <https://everyday-image-processing.tistory.com/86>

    axis=0제외하고는 같아야 한다 shape가 2로 같기 때문에 concate할수있다

    만약 axis=1 열 기준으로 concate하려고 하면 error발생한다

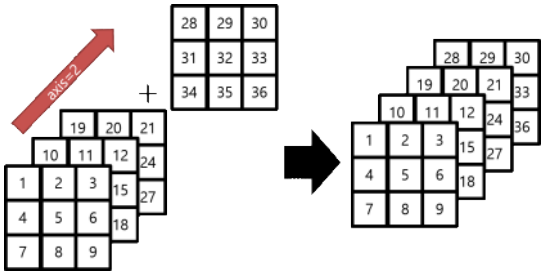
    

    이를 해결하려면 transpose사용해야 한다

  - 3차원의 경우

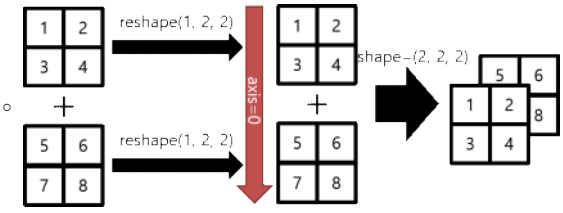
```
a = np.arange(1, 28).reshape(3, 3, 3)
b = np.arange(29, 38).reshape(3, 3, 1)

np.concatenate((a, b), axis=2)
# array([[[ 1,  2,  3, 29],
#         [ 4,  5,  6, 30],
#         [ 7,  8,  9, 31]],

#        [[10, 11, 12, 32],
#         [13, 14, 15, 33],
#         [16, 17, 18, 34]],

#        [[19, 20, 21, 35],
#         [22, 23, 24, 36],
#         [25, 26, 27, 37]]])
```

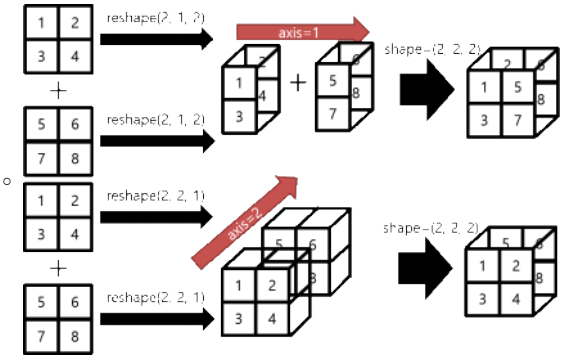
화면 캡처: 2023-10-11 오후 9:28



axis =0,1하면 조건이 맞지 않기 때문에 axis=2로 합쳐야 한다
지금까지 예제 보면서 느낀것은 concate는 axis축 제외하고 shape가 같아야 한다고 했다
axis=0이면 0인덱스끼리 제외하고 나머지 숫자가 같아야하는것 같다 이런식으로 axis=1인경우도 인덱스1빼고 숫자가 같아야 concate가능한것 같다
이 예제도 보면 2 인덱스 제외하고 나머지가 같기 때문에 axis=2로 concate가능한것으로 생각이 된다

결국 핵심은 axis 축 제외한 shape가 같다는 조건이 만족 되면 concate사용할수있다
multihead attention의 경우도 살펴 보면 axis =1로 하면 concate가능할것으로 예상한다

- **np.stack(arrays,axis=0)**
  - concatenate와 다르게 제약이 많은 기능이다 concate는 axis축 제외하고 나머지 shape동일해야 했는데 stack같은경우 모든 shape가 같아야 사용할수있다
  - stack은 지정한 axis를 새로운 axis를 생성한다
  - 

  - 

지정한 축이 1증가한다고 생각하면 될것 같다

stack과 concate의 가장 큰 차이점 : stack함수는 ndim을 1개 넘어서도 배열을 합칠수가 있다
즉 concate가 성공 되려면 reshape를 통해 축이 증가된 상태여야하는데 reshape를 하지 않은 상태에서 concate하면 에러가 난다 왜냐하면 차원을 넘어가게 되기 때문이다 반면 stack은 ndim 즉 차원이 1증가 하기 때문에 차원이 증가하여도 stack을 사용할수가 있다

Bonus

- hstack
  - 두 배열을 가로로 이어붙이는 함수

- vstack
  - hstack과 마찬가지로 두 배열을 이어준다 하지만 axis=0으로 고정되어있다
- dstack
  - axis=ndim으로 고정되어 합치는 함수이다 axis는 ndim-1까지의 범위에서만 사용가능한데 axis=ndim으로 고정한다는것은 stack을 사용한다는 의미이다 즉 dstack은 stack의 방향이 정해진 버전이다

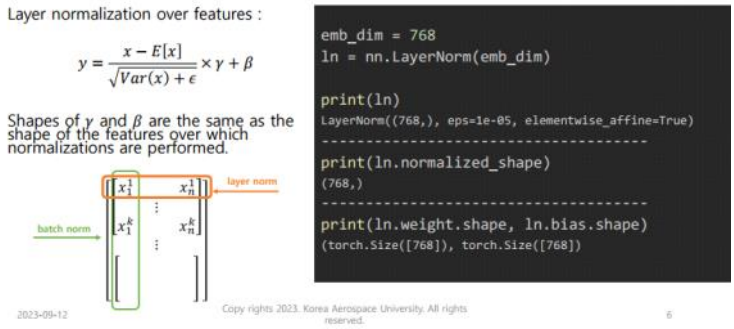
layer normalization

batch normalization과 비슷하기 때문에 batch normalization에 대해서 설명하고 layer normalization설명한다
결국 normalization하는 기법인데 batch에 대해서 normalization하느냐 layer단위로 normalization하느냐에 따라 다른것이다 원리는 비슷하다

우선 batch normalization하지 않으면 vanishing이 발생한다
BN에 대한 직관 : normalization해주면 흩어진 모레알을 모레성 처럼 모았다가 (여기까지가 평균 분산 구해서 정규화 해준거) 다시 흩뿌려준다 얼만큼?(감마만큼의 분산으로) 그리고 β 만큼 shift해준다

gradient vanishing발생하는 지점에 있는 값들을 normalization해줌으로써 방지하는 효과가 있다
감마와 β는 학습되는 가중치이다

다음은 layer normalizatio에 대해 그림만 언급



화면 캡처: 2023-10-11 오후 10:43

```
class LayerNorm(nn.Module):

    def __init__(self,ndim,bias):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(ndim))
        self.bias = nn.Parameter(torch.zeros(ndim)) if bias else None

    def forward(self,input):
        return F.layer_norm(input,self.weight.shape,self.weight,self.bias,1e-5)  #functional (no weight and bias)
```

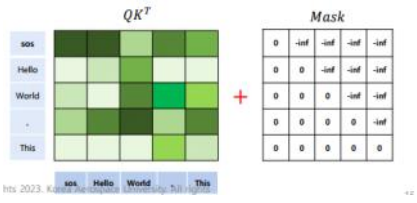
decoder part

아직 decoder파트에서 입력이 어떻게 이루어지고 밑에 cross attention작동과정과 결국 위에서 softmax로 나오는 결과 까지의 진행이 이해하지 못했다

Cross Attention

encoder에서 key,value가져온것과 decoder의 query와 attention수행한다
Masked attention score나온 결과를 Q로 사용하고 인코더에서 나온 k,v에 대해 어텐션 수행?
그렇게 된다면 Q에 대해서는 한정적이지만 k,v에 대해서는 풀로 어텐션한다

masked attention

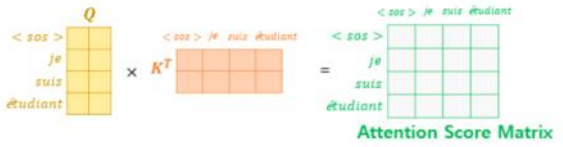


화면 캡처: 2023-10-11 오후 10:50

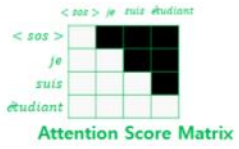
transformer model이 다음 chunk를 predict해야하는데 mask작업을 하지 않으면 cheatting을 하는것과 같게 된다 즉 다음에 올 chunk를 미리 알고 predict하는것과 같다

그래서 mask를 해주는데 순서를 보면 순서대로 masking을 제거하는것을 확인할수있다 -inf하는 이유는 softmax계산할때 제외 시키려고- inf하는것이고 문제 없는것에는 0을 더한다 softmax는 지수 함수를 사용한다 지수 함수에 -inf오면 0에 수렴하게 되니까 masking효과를 볼수있는것이다

softmax(QK^t + Mask)V



이제 자기 자신보다 미래에 있는 단어들은 참고하지 못하도록 다음과 같이 마스킹합니다.




```
def multi_head_masked_attention(Q,K,V,num_head=8,mask=None):
    batch,max_toekn,embbed_dim = Q.size()

    dim_k = embbed_dim//num_head # for softmax normalization

    Q = Q.view(batch,-1,num_head,dim_k).transpose(1,2) # [64,512,8,91] 3차원을 4차원 확장 한거
    K = K.view(batch,-1,num_head,dim_k).transpose(1,2)
    V = V.view(batch,-1,num_head,dim_k).transpose(1,2)


    scores = Q@K.transpose(-2,-1) # dot product

    if mask is not None:
        scores = scores.masked_fill(mask==0,float('-inf')) # QK^T + Mask

    R = torch.softmax(scores,dim=-1)@V

    R = R.transpose(1,2).contiguous().view(batch,-1,embbed_dim) # same input dim

    return R
```

결국

scores = (Q@K.transpose(-2,-1))/math.sqrt(Q.size(-1) # batch 고려해야하기때문
scores.masked_fill(mask==0,-float('inf'))

attention = torch.softmax(scores,dim=-1)@V

main function

X = torch.randn(batch,max_token,embbed_dim)

W_Q = torch.randn(embbed_dim,embbed_dim)
W_K = torch.randn(embbed_dim,embbed_dim)
W_V= torch.randn(embbed_dim,embbed_dim)

Q = X@W_Q
K = X@W_K
V = X@W_V

R = multiple_head_masked_attention(~~~)

sclaed_dot_product_attention

Flash attention?
torch.nn.functional.scaled_dot_product_attention(query,key,value, attn_mask=None,dropout_p=0.0,is_causal=False) -> Tensor

attn_mask = torch.ones(L,S,dtype = torch.bool).tril(diagonal=0) if is)causal else attn_mask

attn_mask = attn_mask.masked_fill(not attn_mask,-float('inf')) if attn_mask.dtype==torch.bool else attn_mask

attn_weight = torch.softmax((Q@K.transpose(-2,-1)#batch가 같이 고려되기 땜이다/math.sqrt(Q.size(-1))) + attn_mask,dim=-1)

attn_weight = torch.dropout(attn_weight,droupout_p)

return attn_weight @ V

Droupout(p=)

during training : 확률 p 만큼 droupout한다
during inference : droupout작동하지않음

class Block(nn.Module)

ln_l = LayerNorm(config.n_embd,bias = config.bias)

attn = CausalSelfAttention(config)

ln_2 = LayerNorm(config,n_embd,bias=config.bias)

mlp = MLP(config)

# skip connection유지
def forward(self,x):
    x = x+ self.attn(sel.ln_l(x))
    x = x+self.mlp(self.ln_2(x))

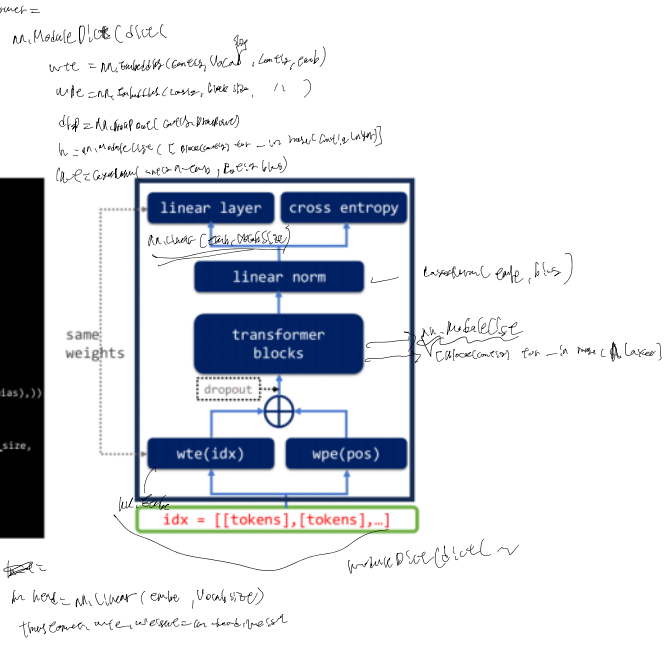
GPT-2 model configuration

```python
@dataclass
class GPTConfig:
    block_size: int = 1024
    vocab_size: int = 50304 # GPT-2 vocab_size of 50257, padded up to nearest multiple of 64 for efficiency
    n_layer: int = 12
    n_head: int = 12
    n_embd: int = 768
    dropout: float = 0.0
    bias: bool = True # True: bias in Linears and LayerNorms, like GPT-2. False: a bit better and faster
```

화면 캡처: 2023-10-12 오전 8:46

GPT-2 model

```python
transformer =
    nn.ModuleDict(dict(
    wte = nn.Embedding(config.vocab_size,
    config.n_embd),
    wpe = nn.Embedding(config.block_size,
    config.n_embd),
    drop = nn.Dropout(config.dropout),
    h = nn.ModuleList([Block(config) for _ in
    range(config.n_layer)]),
    ln_f = LayerNorm(config.n_embd, bias=config.bias),))

lm_head = nn.Linear(config.n_embd, config.vocab_size,
bias=False)

transformer.wte.weight = lm_head.weight
```



화면 캡처: 2023-10-12 오전 8:46

화면 캡처: 2023-10-12 오전 8:46

model - forward

```
        device = idx.device
        b, t = idx.size()
        assert t <= self.config.block_size, f"Cannot forward sequence of length {t}, block size is only {self.config.block_size}"
        pos = torch.arange(0, t, dtype=torch.long, device=device) # shape (t)

        # embedding
        tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
        pos_emb = self.transformer.wpe(pos) # position embeddings of shape (t, n_embd)
        x = self.transformer.drop(tok_emb + pos_emb)
        # transformer blocks and liner normalization
        for block in self.transformer.h:
            x = block(x)
        x = self.transformer.ln_f(x)

        if targets is not None:
            # if we are given some desired targets also calculate the loss
            logits = self.lm_head(x)
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)
        else:
            # inference-time mini-optimization: only forward the lm_head on the very last position
            logits = self.lm_head(x[:, [-1], :]) # note: using list [-1] to preserve the time dim
            loss = None

        return logits, loss
```

화면 캡처: 2023-10-12 오전 8:50

wte-> vocab size,embed

wpe -> block size,embed

F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)

- logits = self.lm_head(x)
- logits.size() = [batch, block, vocab]
- logits.view(-1, logits.size(-1)) = [batch * block, vocab]
- one logit dimension?

- targets.size() = [batch, block]
- targets.view(-1) = [batch * block]

- ignore_index=-1 : do not compute loss if target id is -1

화면 캡처: 2023-10-12 오전 8:51

configure optimizers

```
# start with all of the candidate parameters
param_dict = {pn: p for pn, p in self.named_parameters()}
# filter out those that do not require grad
param_dict = {pn: p for pn, p in param_dict.items() if p.requires_grad}
# create optim groups. Any parameters that is 2D will be weight decayed, otherwise no.
# i.e., all weight tensors in matmuls + embeddings decay, all biases and layernorms don't.
decay_params = [p for n, p in param_dict.items() if p.dim() >= 2]
nodecay_params = [p for n, p in param_dict.items() if p.dim() < 2]
optim_groups = [
    {'params': decay_params, 'weight_decay': weight_decay},
    {'params': nodecay_params, 'weight_decay': 0.0}
]

# Create AdamW optimizer and use the fused version if it is available
fused_available = 'fused' in inspect.signature(torch.optim.AdamW).parameters
use_fused = fused_available and device_type == 'cuda'
extra_args = dict(fused=True) if use_fused else dict()
optimizer = torch.optim.AdamW(optim_groups, lr=learning_rate, betas=betas, **extra_args)
```

화면 캡처: 2023-10-12 오전 8:52