特定の key に対して attention_weight を0にするMask処理を行う。
→maskしたい要素のweightを**－∞**に置き換える。
attention_weight を0にするケースとして以下の2つがある。
①PAD を無視する。
②Decoder の Self-Attention で未来の情報を参照できないようにする。
※**マスクは attention_weight に適用する**ものですので、shape は [batch_size, q_length, m_length] となる。

なぜ -∞ かというと、 $softmax_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$ なので消したい x_j が -∞ になると $e^{x_j} \to e^{-\infty} = 0$ になるから

Softmax 関数はlogit値が大きいほど勾配が0に近くなってしまう。
query, key の次元が大きいほど logit は大きくなるので、query の大きさを depth に従って小さくするため√depthで割っている。

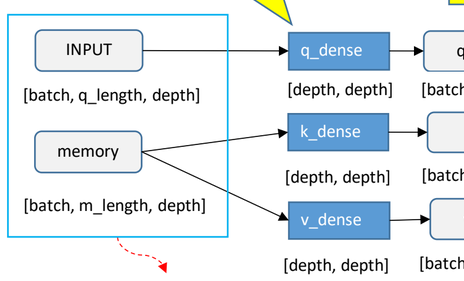
$$attention\_weight = softmax\left(\frac{qk^T}{\sqrt{depth}}\right)$$

全結合層(Dense)で変換
input → query
memory → key , valueを得る。

queryとkeyで内積を取り関連度情報を得る。
→行列積なので、query と key のそれぞれのベクトルが似た方向を向いていれば値は大きくなる。
感覚的にはquery と key の近さを計算しているような感じ

query ごとの weight の合計が1.0になるように

各queryがどのmemoryから情報を取得するかの重み



[q_length, k_length] の部分を見ると下のような行列になっている。

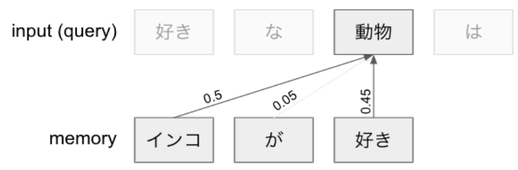
q_0 ← m_0	**q_0 ← m_1**	**...**	**q_0 ← m_M**
q_1 ← m_0	q_1 ← m_1	...	q_1 ← m_M
...	...	...	...
q_N ← m_0	q_N ← m_1	...	q_N ← m_M

q_i ← m_j は i 番目のquery が j 番目の memory から情報を引いてくる重みを表す。
Softmax をかけているので、各行の値の合計が1.0になる。

attention weight に従って value から情報を引き出す

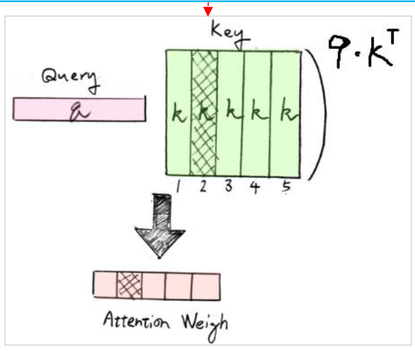
得られたベクトルを Dense で変換したものを出力

attention_weightと value の行列積を取ることで、重みに従って value の情報を取得する。



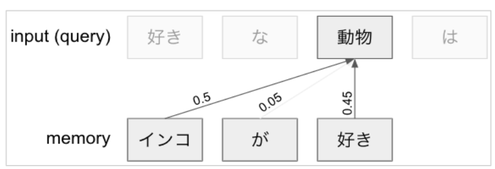
動物 の query (つまり query[batch, 2, :]) の位置の値は
value(インコ) * 0.5 + value(が) * 0.05 + value(好き) * 0.45
となる。

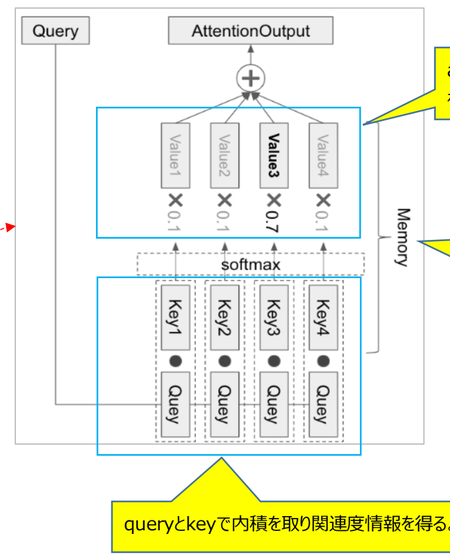
バッチの中のデータは以下のようなイメージ



Attentionとは queryquery **(検索クエリ) に一致する** key **を索引し，対応する** value **を取り出す操作**で、辞書オブジェクト機能と同じと考えればいい。
※query は memory のどこから情報を引いてくるかを決めるために key を使います。

例えば 動物 というクエリに対して「インコ が50%、がが5%、好き が45%」





attention_weightとvalueの内積を取ってvalueを抽出

Memoryをkey，valueに分ける。

queryとkeyで内積を取り関連度情報を得る。

●attention は query によって memory (=key, value) から情報を引き出す仕組み
●key, value は memory を変換してつくる
●query と key から attention weight を計算する
●attention weight に従って value から情報を引き出す

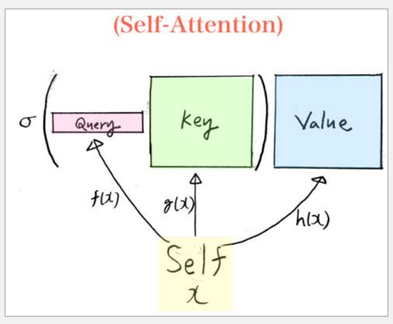
Attensionの種類

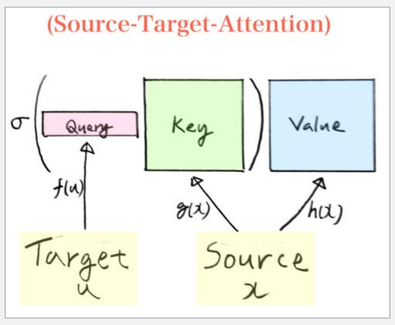
①Self-Attention

input (query) と memory (key, value) すべてが同じ Tensor を使う Attention
Self-Attention は言語の文法構造であったり、照応関係（its が指してるのは Law だよねとか）を獲得するのにも使われている
Self-Attention は汎用に使える仕組みで、Transformer の Encoder 部分、Decoder 部分のどちらでも使われている。
文章分類は SourceTarget-Attention は必要なく Self-Attention のみで作ることができる。

②SourceTarget-Attention

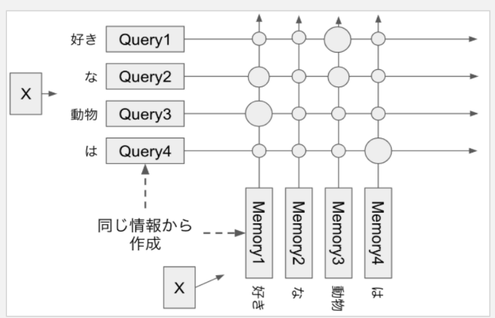
input (query) と memory (key, value) の２つが別の Tensor を使う Attention

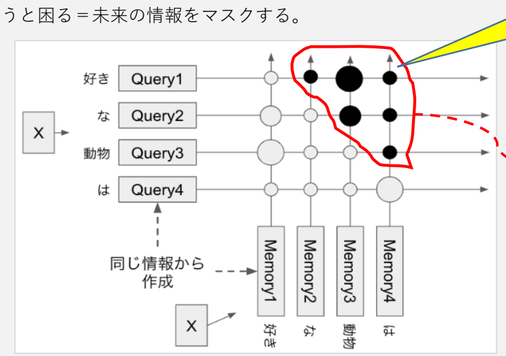




Decoder の Self-Attention で未来の情報を参照できないようにする

Decoder は推論時はある時刻tの入力からt+1を予測し、その予測を次の入力に回してさらにそのさきを予測する。（自己回帰）
一方学習のときにはすべての時刻で同時に次の時刻のトークンを予測する学習を行う。
→この時 Self-Attention で予測対象の未来の情報が得られてしまうと困る＝未来の情報をマスクする。





各query に対して自分の時刻より先の memory の情報を参照できないようにMaskする。

この例だと以下のようなMASKになる。

```
[[False,  True,  True,  True],
 [False, False,  True,  True],
 [False, False, False,  True],
 [False, False, False, False]]
```

上の図で、 query が「好き / な」までしかまだ生成されていないのに、 memory の未来の情報「動物」を参照できたら困る（推論時には未来の情報は当然与えられないので）

Multi-head Attention

Multi-head Attention は、これまでのような Simple Attention をパラレルに並べたもの。
それぞれの attention を head と呼ぶ。

※論文では一つの大きな Attention を行うよりも、小さな複数の head に分けて Attention を行うほうが性能が上がったとされている。
※query, key, value をそれぞれ head_num 個に split してからそれぞれ attention を計算し、最後に concat するという仕組み

Positional Encoding

Transformer は文章内のトークンの順序を学習に使うことができないので位置情報を付加する必要がある。
各トークンがどの位置にあるのかを表すための値、Positional Encoding を各トークンを Embedding したものに足す。
Positional Encoding は次のような数式で計算される。

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

pos: 時刻
2i, 2i+1: Embedding の何番目の次元か

> このような数式を選んだ理由としては、この式が相対的な位置（PE_{pos+k}）を PE_{pos} の線形関数で表現可能（=ニューラルネットが学習しやすいはず）だからとのことです。