

@halhorn (/halhorn) 2018年12月05日に更新
(/halhorn)

...

ミクシィグループ Advent Calendar 2018 (/advent-calendar/2018/mixi) | 5日目

作って理解する Transformer / Attention

NLP(/tags/nlp) DeepLearning(/tags/deeplearning) TensorFlow(/tags/tensorflow)
Transformer(/tags/transformer)

👍 274

📄

こんにちは。ミクシィ AI ロボット事業部でしゃべるロボットを作っている (<https://www.wantedly.com/projects/92981>)インコです。
この記事は ミクシィグループ Advent Calendar 2018 (<https://qiita.com/advent-calendar/2018/mixi>) の5日目の記事です。

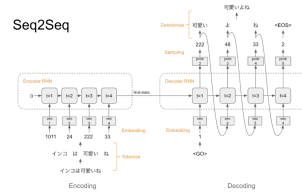
この記事の目的

この記事では2018年現在 DeepLearning における自然言語処理のデファクトスタンダードとなりつつある Transformer を作ることで、Attention ベースのネットワークを理解することを目的とします。

機械翻訳などの Transformer, 自然言語理解の BERT やその他多くの現在 SoTA となっている自然言語処理のモデルは Attention ベースのモデルです。Attention を理解することが今後の自然言語処理 x Deep Learning の必須になってくるのではないのでしょうか。

この記事にかかれているコード (Github) (https://github.com/halhorn/deep_dialog_tutorial/tree/master/deepdialog/transformer)

歴史 - RNN から Attention へ



Seq2Seq

(<https://camo.qiitausercontent.com/48959ffb403f1>)

1be98f452d10e0028cca38f53d0/68747470733a2f2f71696974612d696d6167652d73746f72652e7332e616d617a6f6e6177732e636f6d2f302f36313037392f64373231626564362d326431362d633639342d353537642d6236646131633034373930652e706e67)



(<https://camo.qiitausercontent.com/39708ad89408>)

a2a58c93653bd237551eca8d132e/68747470733a2f2f71696974612d696d6167652d73746f72652e7332e616d617a6f6e6177732e636f6d2f302f36313037392f64373231626564362d326431362d633639342d353537642d6236646131633034373930652e706e67)

かつて 自然言語処理 x Deep Learning と言えば、LSTM や GRU といった RNN (Recurrent Neural Network) でした。
参考：DeepLearning における会話モデル：Seq2Seq から VHRED まで (<https://qiita.com/halhorn/items/646d323ac457715866d4>)

ところが2017年の6月、Attention Is All You Need (<https://arxiv.org/abs/1706.03762>) という強いタイトルの論文が Google から発表され、機械翻訳のスコアを既存の RNN モデル等から大きく引き上げます。

Attention は従来の RNN のモデル Seq2Seq などでも使われていました。（seq2seq で長い文の学習をうまくやるための Attention Mechanism について (<https://qiita.com/halhorn/items/614f8fe1ec7663e04bea>)

Transformer は文章などのシーケンスから別の文章などのシーケンスを予測するモデルとして発表されましたが、Transformer の Encoder 部分を使ったモデルは文章分類などシーケンスからカテゴリを予測する問題等でも高い性能を出しており、特に最近発表された同じく Google の BERT (Bidirectional Encoder Representations from Transformers) (<https://arxiv.org/abs/1810.04805>) は言語理解の様々なタスクのベンチマークである GLUE で圧倒的なスコアを達成しました。

このように Transformer やその基盤になっている Attention はその後広く使われるようになり、現在では「自然言語処理 x Deep Learning といえば Attention だよな」と言っても過言ではないくらいの存在となっています。

利点

Attention は RNN に比べて良い点が3つあります。

- 性能が良い
 - 現在の SoTA の多くが Attention ベース
- 学習が速い
 - RNN は時刻tの計算が終わるまで時刻t+1の計算をできません。このため GPU をフルに使用できないことがあります。
 - Transformer は推論時の Decoder を除いて、すべての時刻の計算を同時に行えるため GPU をフルに使いやすいです。
 - 私の経験でも RNN の学習時は GPU 使用率が0-100%の間を行き来し平均60%程度ですが、Attention ベースのモデルの場合 GPU 使用率は98%前後くらいにピッタリ張り付きます
- 構造がシンプル
 - RNN を使わないため比較的構造が単純です
 - 全結合と行列積くらい知っていれば理解できてしまいます

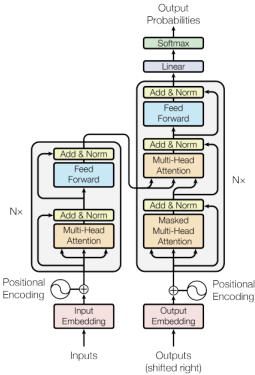
Transformer を知る

日本語の情報であれば Ryobot さんの 論文解説 Attention Is All You Need (Transformer) (<http://deplearning.hatenablog.com/entry/transformer>) を読むのが一番です。
もしくは本家の論文 (<https://arxiv.org/abs/1706.03762>) も大変読みやすいです。

Transformer を作る前にざっと目を通しておくことをおすすめします。

Transformer を作る

「作る」は「理解する」の近道です。
ここからは作ることで Transformer を理解していきましょう。



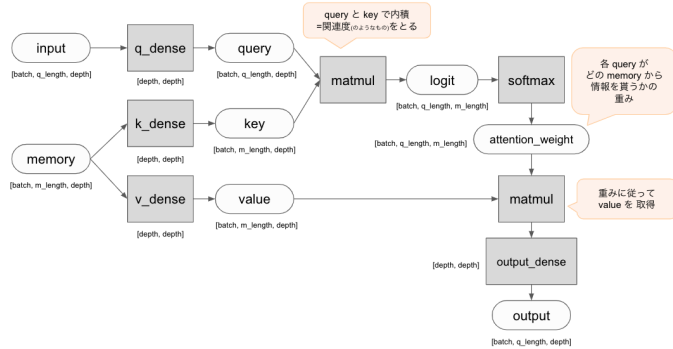
(<https://camo.qiitusercontent.com/ffe49d2d276eda374036114a0abb211fc94016c0/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f33623432393663642d633938352d366132352d336663612d3631643934613034636363392e706e67>)

Transformer も BERT も、基本的な Attention の仕組みを理解できれば難しくはありません。
まずは（実用性はおいておいて）基本的な Attention を作っていきましょう。

基本的な Attention

Attention の基本は query と memory(key, value) です。
Attention とは query によって memory から必要な情報を選択的に引っ張ってくることです。
memory から情報を引っ張ってくる時には、query は key によって取得する memory を決定し、対応する value を取得します。

まずは基本的な Attention として下記のようなネットワークを作ってみましょう。
丸は Tensor, 四角はレイヤーもしくは処理を表しています。



(<https://camo.qiitausercontent.com/dc9367114569469784d8f5a92b23fd4291a562f/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f37633230383137622d336563632d363430322d666266362d6263303035643430383734392e706e67>)

コードで書くと以下ようになります。
transformer/attention.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/attention.py#L125)

基本的なAttention

```
class SimpleAttention(tf.keras.models.Model):
    ...
    Attention の説明をするための、Multi-head ではない単純な Attention です
    ...

    def __init__(self, depth: int, *args, **kwargs):
        ...
        コンストラクタです。
        :param depth: 隠れ層及び出力の次元
        ...

        super().__init__(*args, **kwargs)
        self.depth = depth

        self.q_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='q_dense_layer')
        self.k_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='k_dense_layer')
        self.v_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='v_dense_layer')
        self.output_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='output_dense')

    def call(self, input: tf.Tensor, memory: tf.Tensor) -> tf.Tensor:
        ...
        モデルの実行を行います。
        :param input: query のテンソル
        :param memory: query に情報を与える memory のテンソル
        ...

        q = self.q_dense_layer(input) # [batch_size, q_length, depth]
        k = self.k_dense_layer(memory) # [batch_size, m_length, depth]
        v = self.v_dense_layer(memory)

        # ここで q と k の内積を取ることで、query と key の関連度のようものを計算します。
        logit = tf.matmul(q, k, transpose_b=True) # [batch_size, q_length, k_length]

        # softmax を取って正規化します
        attention_weight = tf.nn.softmax(logit, name='attention_weight')

        # 重みに従って value から情報を引いてきます
        attention_output = tf.matmul(attention_weight, v) # [batch_size, q_length, depth]
        return self.output_dense_layer(attention_output)
```

細かく見ていきます。

input, memory



4964e03/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f32393664613263302d376633612d666665652d356639312d3365663664313639393234302e706e67)

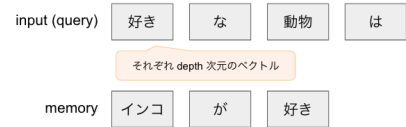
入出力は input と memory です。

- **input** (query): 入力の時系列
- **memory**: 引いてくる情報の時系列

これらの Tensor の shape は [batch_size, q_length, depth] と [batch_size, m_length, depth] です。

length は文章中のトークンの長さ（例えば1文の中の最大単語数など）です。
depth は各単語を Embedding した次元数です。

バッチの中のデータ一つをとってくと、このようなデータが入っています。

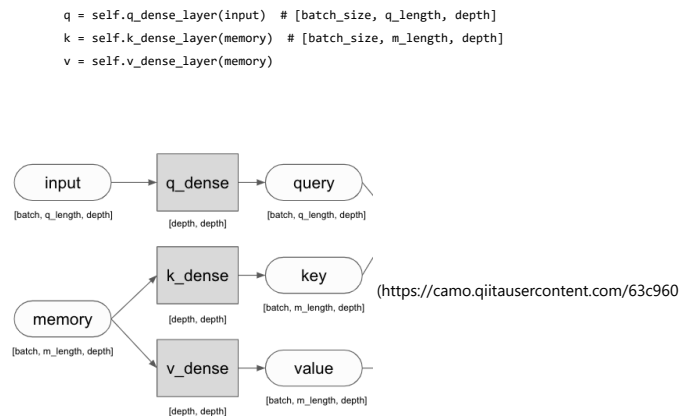


(<https://camo.qiitusercontent.com/d7761f31a240c2730354a200bd38e58726b37aa4/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f36626430306263382d306133352d316234642d636236352d3630656137663234366334362e706e67>)

この図だと、 q_length=4, m_length=3 ですね。

それぞれのブロックは、「好き」などの元々は単語等であったトークンを depth 次元の分散表現に直したものです。

query, key, value を作る



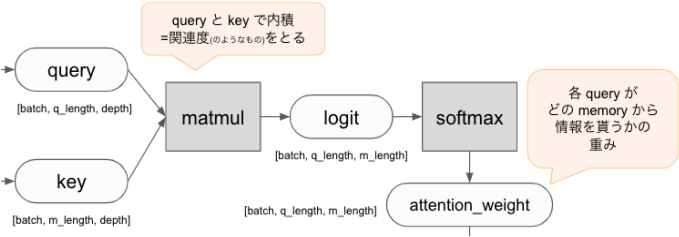
50cba82c024623475f2b5efba2c3ebba0a/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f63313632656163612d633836322d363936652d323963632d3663336666343561323937322e706e67)

input は Dense レイヤーで変換されて query に、 memory は 2 つの Dense レイヤーでそれぞれ key, value に変換されます。

attention weight を計算する

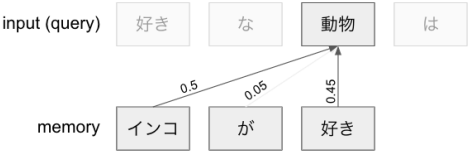
```
# ここで q と k の内積を取ることで、query と key の関連度のようなものを計算します。
logit = tf.matmul(q, k, transpose_b=True) # [batch_size, q_length, k_length]

# softmax を取ることで正規化します
attention_weight = tf.nn.softmax(logit, name='attention_weight')
```



(<https://camo.qiitausercontent.com/2a8276fc2b546123d97def45043824a4adf4329f/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f33633561663761622d393861622d626433662d633266612d3739643432313866393163372e706e67>)

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。（key は memory を変換したものでしたね。）
例えば 動物 というクエリに対して「インコ が50%、 が5%、 好き が45%かなー」などと計算します。



(<https://camo.qiitausercontent.com/0b85036234a8aa52351ecd68c9dcefc96274187e/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f33633561663761622d393861622d62666234322d353233392d303065353535361396334612e706e67>)

この計算は具体的には、query と key の行列積(matmul)をとることで計算されます。
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りますね。大雑把に言うと query と key の近さを計算していると私は理解しています。（ただし query, memory とともにベクトルの長さはそれぞれバラバラなので実際にはそのように単純ではありません。）

行列積をとった後は softmax をかけることで、query ごとの weight の合計が1.0になるように正規化しています。

さて、これで attention_weight ができました。shape は [batch_size, q_length, k_length] です。
[q_length, k_length] の部分を見ると下の様な行列になっています。

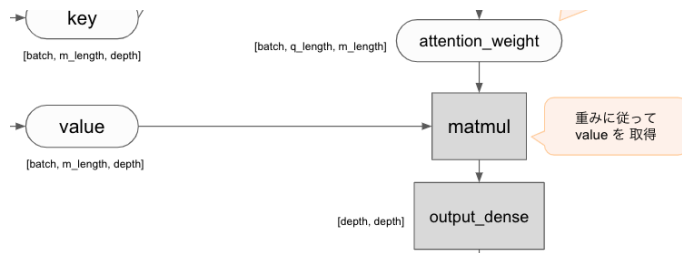
q_0 ← m_0	q_0 ← m_1	...	q_0 ← m_M
q_1 ← m_0	q_1 ← m_1	...	q_1 ← m_M
...
q_N ← m_0	q_N ← m_1	...	q_N ← m_M

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します。
Softmax をかけているので、各行の値は足すと1.0になります。

先程の図の query 「動物」を考えると、表の idx=2 行目が 0.5, 0.05, 0.45 になるということです
ね。

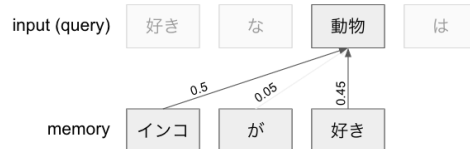
attention weight に従って value から情報を引き出す

```
# 重みに従って value から情報を引いてきます
attention_output = tf.matmul(attention_weight, v) # [batch_size, q_length, depth]
return self.output_dense_layer(attention_output)
```



(<https://camo.qiitusercontent.com/55c6acb98b35897b56162f31013f17eb485ddca7/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f36636232393538362d383637622d633530352d346131622d6535613865303032353232662e706e67>)

重みと value の行列積を取ることで、重みに従って value の情報を引いてきます。

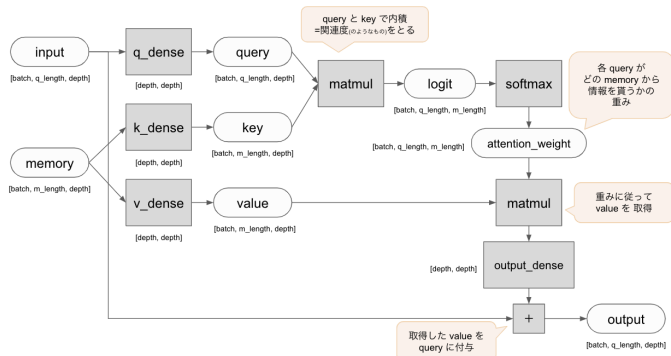


(<https://camo.qiitusercontent.com/0b85036234a8aa52351ecd68c9dcefc96274187e/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f36636232393538362d383637622d633032322d6666234322d353233392d303065353535361396334612e706e67>)

例えば上の図であれば、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 になります。

得られたベクトルを Dense で変換したものがこのレイヤー出力になります。

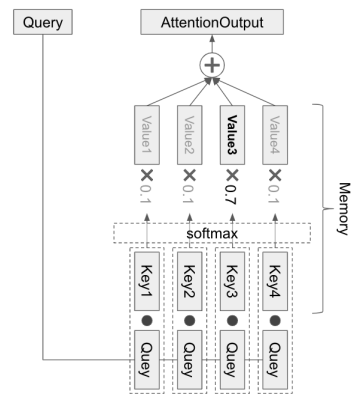
まとめ



(<https://camo.qiitusercontent.com/0df89d309e385fb9b47b74b6fd81833f2c9f5b14/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f34393736316432632d376436382d303139392d353863612d3833333430353330383134312e706e67>)

- 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

別の書き方をするとこんな感じになります。



(<https://camo.qiitausercontent.com/9b8af7118dcd4c006bc531f105f969b1da00dbc9/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f38626631643534342d663834322d343336322d303365392d3333343533333338363563362e706e67>)

Attention の使い方

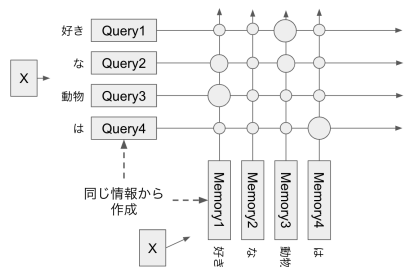
Attention には大きく 2 つの使い方があります。

Self-Attention

input (query) と memory (key, value) すべてが同じ Tensor を使う Attention です。

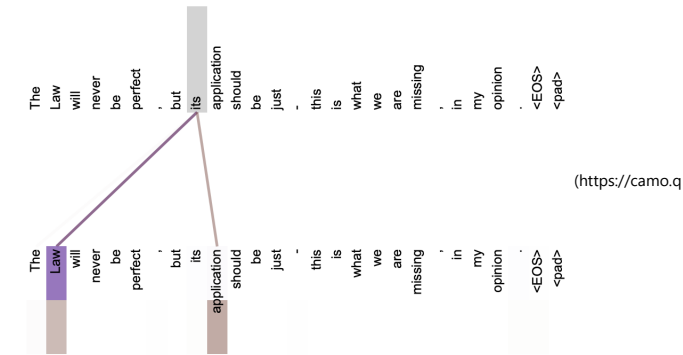
```
attention_layer = SimpleAttention(depth=128)

x: tf.Tensor = ...
attention_output = attention_layer(input=x, memory=x)
```



(<https://camo.qiitausercontent.com/e3841e989665ca207b2bafc5ae1bbb81074e5724/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f30333336313339372d3336361382d616464612d633066312d6530386661386630333031352e706e67>)

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 だよなとか）を獲得するのに使われているなどと論文では分析されています。



iitausercontent.com/d820da9fb7a3caab57af63ec22710ef7cfe36571/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f65326

632366132342d646637322d626563312d343132622d3862613934333566376139642e706e67)
(上段が query. its という query が Law , application という memory から情報を引いてい
ることがわかる)

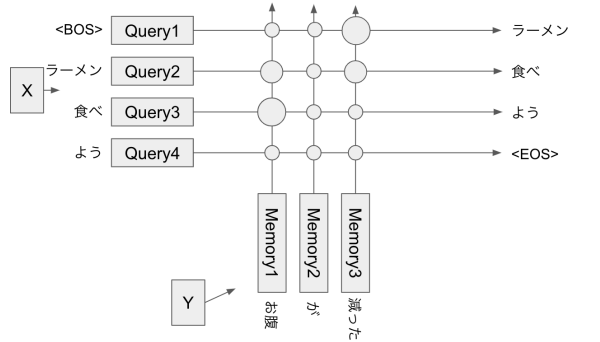
Self-Attention は汎用に使える仕組みで、Transformer の Encoder 部分、Decoder 部分のどちら
でも使われていますし、文章分類などでは後述する SourceTarget-Attention は必要なく Self-Atte
nition のみで作ることができます。

SourceTarget-Attention

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

```
attention_layer = SimpleAttention(depth=128)

x: tf.Tensor = ...
y: tf.Tensor = ...
attention_output = attention_layer(input=x, memory=y)
```



(<https://camo.qiitousercontent.com/4edacdca0a7104ca91b11adb85f3ea31c6ac6/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f35343239393939302d623161382d393536392d636536342d303839383036236303061622e706e67>)

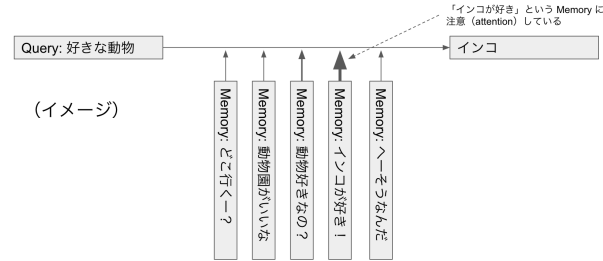
SourceTarget-Attention は Transformer では Decoder で使われます。
Decoder はある時刻 t のトークンを受け取って t+1 の時刻のトークンを予測します。
お腹 / が / 減った → ラーメン / 食べ / よう という発話応答を学習する場合、

- memory 過去の発話 お腹 / が / 減った
- input は <BOS> / ラーメン / 食べ / よう (BOS は Begin of Sentence を表すトークン)

として、出力が input の1時刻あと、つまり ラーメン / 食べ / よう / <EOS> となるようにして学習します。(EOS: End Of Sentence)

生成時には、query <BOS> は memory お腹 / が / 減った から情報を取り出してきて ラーメン を生成します。次に生成された ラーメン query に加え、query <BOS> / ラーメン は同じ memory から情報をとってきて ラーメン / 食べ を生成します。といった具合で1トークンずつ生成を行います。

SourceTarget-Attention は Transformer ではこのように Decoder で使われていますが、End-to-End Memory Network(日本語解説記事) (http://deeplearning.hatenablog.com/entry/memory_networks) ではもっとダイレクトに query を質問：「好きな動物」などとし、memory は「どこ行くー?」「動物園がいいな」「動物好きなの?」「インコが好き!」「ヘーそうなんだ」といった文章列として、query は一番関係ありそうな文(ここだと「インコが好き!」)から情報を引いてきて「インコ」と答えます。



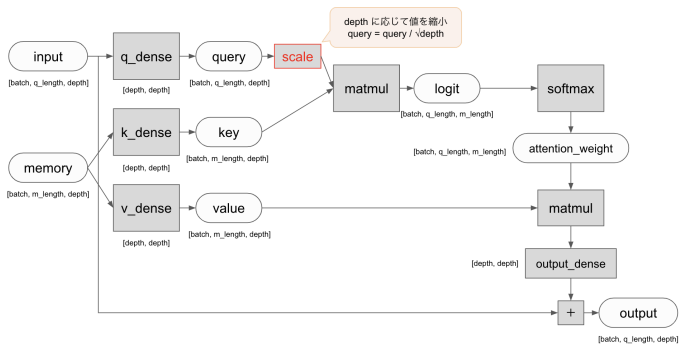
(<https://camo.qiitousercontent.com/d643830c82f8f34705cb71720fde312123d1637f/68747470733>)

a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f38656331343865322d663432652d363634652d633166632d3837353764653536386137342e706e67)

Scaled Dot-production

ここから先のセクションでは基本の Attention に加えて、学習をうまく行うための Transformer に実装されているいろいろな仕組みを紹介していきます。

まずは Scaled Dot-production です。



(https://camo.qiitausercontent.com/03b608cc2a33dd3a485eb440569560d4466b0e45/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f38656331343865322d663432652d363634652d633166632d3837353764653536386137342e706e67)

Softmax 関数は、logit の値が大きいと値がサチってしまい、gradient が0に近くなってしまいます。

Softmax の logit は query と key の行列積です。従って、query, key の次元（depth）が大きいほど logit は大きくなります。

そこで、query の大きさを depth に従って小さくしてあげます。

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

transformer/attention.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/attention.py#L65)

```
# ここで q と k の内積を取ることで、query と key の関連度のようなものを計算します。
q *= depth ** -0.5 # scaled dot-product
logit = tf.matmul(q, k, transpose_b=True) # [batch_size, q_length, k_length]

# softmax を取ることで正規化します
attention_weight = tf.nn.softmax(logit, name='attention_weight')
```

Mask

mask は特定の key に対して attention_weight を0にするために使います。attention_weight は Softmax の出力なので、その入力である logit に対し、mask したい要素は -∞ に値を書き換えてやります。

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 $e^{x_j} \rightarrow e^{-\infty} = 0$ になるからです。

transformer/attention.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/attention.py#L69)

```
# ここで q と k の内積を取ることで、query と key の関連度のようなものを計算します。
logit = tf.matmul(q, k, transpose_b=True) # [batch_size, head_num, q_length, k_length]
logit += tf.to_float(attention_mask) * input.dtype.min # mask は pad 部分などが1, 他は0

# softmax を取ることで正規化します
attention_weight = tf.nn.softmax(logit, name='attention_weight')
```

attention_mask は shape が [batch_size, q_length, m_length] で dtype が bool の Tensor です。0にしたい部分が True となっています。

これによって Mask が True の部分は input.dtype.min ㇿ マイナス無限大な値になります。

attention_weight を0にしたい場合は次の2つがあります。

- PAD を無視したい
- Decoder の Self-Attention で未来の情報を参照できないようにしたい

PAD を無視したい

ニューラルネットに入力される文章（トークン列）の長さは batch によって様々です。

例えば key に入ってくる文章が

```
[
  おはよう,
  インコ / が / 好き,
  お腹 / へった,
]
```

だとしましょう。それぞれ長さは [1, 3, 2] です。このままでは長さがバラバラで Tensor に変換できないので、PAD を追加してあげます。

```
[
  おはよう / <PAD> / <PAD>,
  インコ / が / 好き,
  お腹 / へった / <PAD>,
]
```

これで、このデータは 3 x 3 の Tensor になります。（実際は各トークンは Embedding されたベクトルなので、3 x 3 x depth の Tensor になります）

さて、しかし attention_weight を計算するときには PAD は無視したいところです。

なぜなら PAD がいくつあるかはバッチの作り方（そのバッチ内の他の文の長さ）で異なってくるため、本来お互い干渉するべきではないバッチ内の他の文によって出力が変わってしまうためです。

（たとえば上の例のバッチの他のデータに「インコ / は / かわいい / ん / じゃ / あ / あ / あ / あ」など長いものがあれば上の例の PAD がたくさん増え、もし PAD を無視しないとそれによって計算結果が変わってしまいます。）

そこで、PAD の部分にはマスクをしましょう。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] になります。（後述する Multi-head Attention ではここに head の次元も加わります。）

例えば memory = おはよう / <PAD> / <PAD> に対する mask は次のようになります。（仮に query の長さが4とします）

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

行が query の長さ、列が key の長さになっています。key の idx=1,2 が Mask されていますね。

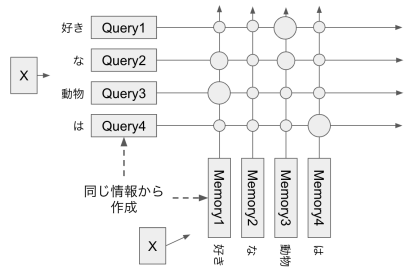
ではこれらの Mask はどうやって作るのか？？というところは最後の全体像でコードを載せます。

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

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を予測します。（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います。

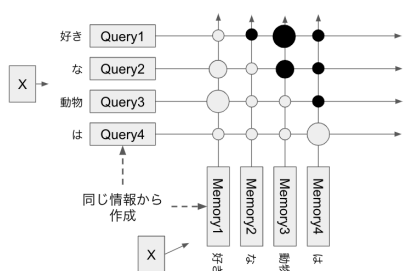
この時 Self-Attention で予測対象の未来の情報が得られてしまうと困ります。



(<https://camo.qiitusercontent.com/e3841e989665ca207b2bafc5ae1bbb81074e5724/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f3033366313339372d336361382d616464612d633066312d6530386661386630333031352e706e67>)

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」を参照できたら困りますね。（推論時には未来の情報は当然与えられないので）

なので、query は自分の時刻より先の memory の情報を参照できなくする必要があります。



(<https://camo.qiitusercontent.com/c1c0194389ab9dbf3831853913dc15b5710255cc/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f38646161313233372d623336392d663233342d376134332d6233356539383634383837392e706e67>)

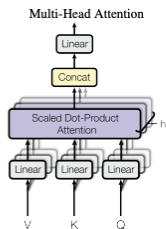
図で書くと上の黒丸のところを mask する必要があります。

つまりこの図の場合 mask は次のようになります。

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

Multi-head Attention

Multi-head Attention は、これまでのような Simple Attention をパラレルに並べるものです。それぞれの attention を head と呼びます。



(<https://camo.qiitusercontent.com/36418044c1d894a4421e3a7089a03105f272c853/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f35613936346631352d343939372d653965372d633133652d3961656162386561316136312e706e67>)

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書かれています。

仕組みは単純で、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 するだけです。
この Multi-head Attention が RNN における LSTM, GRU セルのように Attention ベースのモデルの基本単位になってきます。

さて、これまでの Simple Attention, Scaled Dot-product, Mask, Multi-head Attention を合わせると以下のような Attention モジュールができます。

論文や本家実装では、この他に随所に Dropout を追加して汎化性能を上げているのでこれも追加します。

transformer/attention.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/attention.py#L9)

```

class MultiheadAttention(tf.keras.models.Model):
    ...

    Multi-head Attention のモデルです。
    model = MultiheadAttention(
        hidden_dim=512,
        head_num=8,
        dropout_rate=0.1,
    )
    model(query, memory, mask, training=True)
    ...

    def __init__(self, hidden_dim: int, head_num: int, dropout_rate: float, *args, **kwargs):
        ...

        コンストラクタです。
        :param hidden_dim: 隠れ層及び出力の次元
            head_num の倍数である必要があります。
        :param head_num: ヘッドの数
        :param dropout_rate: ドロップアウトする確率
        ...

        super().__init__(*args, **kwargs)
        self.hidden_dim = hidden_dim
        self.head_num = head_num
        self.dropout_rate = dropout_rate

        self.q_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='q_dense_layer')
        self.k_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='k_dense_layer')
        self.v_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='v_dense_layer')
        self.output_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='output_dense_layer')
        self.attention_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(
        self,
        input: tf.Tensor,
        memory: tf.Tensor,
        attention_mask: tf.Tensor,
        training: bool,
    ) -> tf.Tensor:
        ...

        モデルの実行を行います。
        :param input: query のテンソル
        :param memory: query に情報を与える memory のテンソル
        :param attention_mask: attention weight に適用される mask
            shape = [batch_size, 1, q_length, k_length] のものです。
            pad 等無視する部分が True となるようなものを指定してください。
        :param training: 学習時か推論時のフラグ
        ...

        q = self.q_dense_layer(input) # [batch_size, q_length, hidden_dim]
        k = self.k_dense_layer(memory) # [batch_size, m_length, hidden_dim]
        v = self.v_dense_layer(memory)

        q = self._split_head(q) # [batch_size, head_num, q_length, hidden_dim/head_num]
        k = self._split_head(k) # [batch_size, head_num, m_length, hidden_dim/head_num]
        v = self._split_head(v) # [batch_size, head_num, m_length, hidden_dim/head_num]

        depth = self.hidden_dim // self.head_num
        q *= depth ** -0.5 # for scaled dot production

        # ここで q と k の内積を取ることで、query と key の関連度のようなものを計算します。
        logit = tf.matmul(q, k, transpose_b=True) # [batch_size, head_num, q_length, k_length]
        logit += tf.to_float(attention_mask) * input.dtype.min # mask は pad 部分などが1, 他は0

        # softmax を取って正規化します
        attention_weight = tf.nn.softmax(logit, name='attention_weight')
        attention_weight = self.attention_dropout_layer(attention_weight, training=training)

        # 重みによって value から情報を引いてきます
        attention_output = tf.matmul(attention_weight, v) # [batch_size, head_num, q_length, hidden_dim/head_num]
        attention_output = self._combine_head(attention_output) # [batch_size, q_length, hidden_dim]
        return self.output_dense_layer(attention_output)

    def _split_head(self, x: tf.Tensor) -> tf.Tensor:
        ...

        入力の tensor の hidden_dim の次元をいくつかのヘッドに分割します。
        入力 shape: [batch_size, length, hidden_dim] の時
        出力 shape: [batch_size, head_num, length, hidden_dim/head_num]
        となります。
        ...

        with tf.name_scope('split_head'):
            batch_size, length, hidden_dim = tf.unstack(tf.shape(x))
            x = tf.reshape(x, [batch_size, length, self.head_num, self.hidden_dim // self.head_num])
            return tf.transpose(x, [0, 2, 1, 3])

    def _combine_head(self, x: tf.Tensor) -> tf.Tensor:
        ...

        入力の tensor の各ヘッドを結合します。 _split_head の逆変換です。
        入力 shape: [batch_size, head_num, length, hidden_dim/head_num] の時
        出力 shape: [batch_size, length, hidden_dim]
        となります。
        ...

        with tf.name_scope('combine_head'):

```

```
batch_size, _, length, _ = tf.unstack(tf.shape(x))
x = tf.transpose(x, [0, 2, 1, 3])
return tf.reshape(x, [batch_size, length, self.hidden_dim])
```

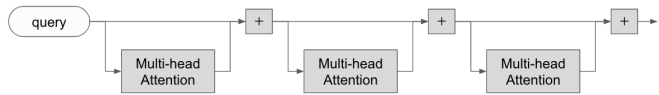
また、Multi-head Attention を継承して Self-Attention も作ります。
これは単に Multi-head Attention の memory に input を渡すだけです。

transformer/attention.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/attention.py#L107)

```
class SelfAttention(MultiheadAttention):
    def call( # type: ignore
        self,
        input: tf.Tensor,
        attention_mask: tf.Tensor,
        training: bool,
    ) -> tf.Tensor:
        return super().call(
            input=input,
            memory=input,
            attention_mask=attention_mask,
            training=training,
        )
```

Hopping

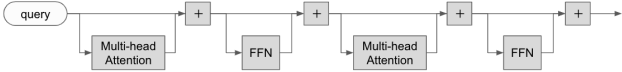
Multi-head Attention を何度も繰り返し適用することで、より複雑な学習ができるようにします。



(<https://camo.qiitausercontent.com/b5d700eb42ed75bc057857486b781607869efa2b/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f32646438323166362d663538332d363839322d333563652d61396233333636361666234642e706e67>)
RNN とは違って、この各 Attention はそれぞれ独立した重みを持っています。
そのため何回 Hopping をするかは固定です。（これを可変にした Universal Transformer (<http://arxiv.org/abs/1807.03819>) というモデルもあります。）

Position-wise Feedforward Network

各 Hopping の Attention のあとは FFN をはさみます。この FFN は全時刻で同じ変換がなされます。



(<https://camo.qiitausercontent.com/63fe2bea14b32918da0a64207e2c0365f761f032/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f3835376533030312d653365622d346336322d366538612d3939636135353131333431312e706e67>)
FFN は2層で、第一層の次元は hidden_dim * 4 で ReLU, 第二層の次元は hidden_dim で Linear な activation function をもちます。

transformer/common_layer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/common_layer.py#L4)

```
class FeedForwardNetwork(tf.keras.models.Model):
    ...
    Transformer 用の Position-wise Feedforward Neural Network です。
    ...

    def __init__(self, hidden_dim: int, dropout_rate: float, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.filter_dense_layer = tf.keras.layers.Dense(hidden_dim * 4, use_bias=True,
                                                         activation=tf.nn.relu, name='filter_layer')
        self.output_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=True, name='output_1')
        self.dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(self, input: tf.Tensor, training: bool) -> tf.Tensor:
        ...
        FeedForwardNetwork を適用します。
        :param input: shape = [batch_size, length, hidden_dim]
        :return: shape = [batch_size, length, hidden_dim]
        ...

        tensor = self.filter_dense_layer(input)
        tensor = self.dropout_layer(tensor, training=training)
        return self.output_dense_layer(tensor)
```

LayerNormalization

画像処理系の Deep Learning では Batch Normalization が有名ですが、Transformer では Layer Normalization を使います。

実は tf.keras には実装が無いので自前で作ります。
transformer/common_layer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/depdialog/transformer/common_layer.py#L49)

```
class LayerNormalization(tf.keras.layers.Layer):
    def build(self, input_shape: tf.TensorShape) -> None:
        hidden_dim = input_shape[-1]
        self.scale = self.add_weight('layer_norm_scale', shape=[hidden_dim],
                                     initializer=tf.ones_initializer())
        self.bias = self.add_weight('layer_norm_bias', [hidden_dim],
                                    initializer=tf.zeros_initializer())
        super().build(input_shape)

    def call(self, x: tf.Tensor, epsilon: float = 1e-6) -> tf.Tensor:
        mean = tf.reduce_mean(x, axis=[-1], keepdims=True)
        variance = tf.reduce_mean(tf.square(x - mean), axis=[-1], keepdims=True)
        norm_x = (x - mean) * tf.rsqrt(variance + epsilon)

        return norm_x * self.scale + self.bias
```

ResidualNormalizationWrapper

Transformer はあちこちに Layer Normalization, Dropout, Residual Connection などの正則化をいれています。
これらを素で書いていくと辛いので、レイヤーを受け取ってそのレイヤーに対して各種 Normalization を施すラッパーを作ります。



(<https://camo.qiitausercontent.com/fc2513d5ec2e84b2eb0667f5b517ffd3b17df0ae/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f30633533666362352d303533342d383063302d616132612d3462646530386662336538652e706e67>)

(本家コード (<https://github.com/tensorflow/models/blob/master/official/transformer/model/transformer.py#L267>) では PrePostProcessingWrapper という名前になっています)

transformer/common_layer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/depdialog/transformer/common_layer.py#L29)

```
class ResidualNormalizationWrapper(tf.keras.models.Model):
    def __init__(self, layer: tf.keras.layers.Layer, dropout_rate: float, *args, **kwargs) -> Non
        super().__init__(*args, **kwargs)
        self.layer = layer
        self.layer_normalization = LayerNormalization()
        self.dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(self, input: tf.Tensor, training: bool, *args, **kwargs) -> tf.Tensor:
        tensor = self.layer_normalization(input)
        tensor = self.layer(tensor, training=training, *args, **kwargs)
        tensor = self.dropout_layer(tensor, training=training)
        return input + tensor
```

Positional Encoding

上記の仕組みだけでは、Transformer は文章内のトークンの順序を学習に使うことができません。

つまり 私 / は / 君 / より / 賢い と 君 / は / 私 / より / 賢い は全く同じデータになってしまいます。

そこで、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 したものに足します。

この Positional Encoding は次のような数式で計算されます。

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

(<https://camo.qiitausercontent.com/6846d47f8b9e270e6a2cc77a82abdcdf540262df/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f386339393063662d333461352d333763392d623836352d6630643930323834323462302e706e67>)

- 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。

試しに偶数番目の次元について計算してみましょう。

$$u_i = \frac{1}{10000^{2i/d_{model}}}$$
$$PE_{p,2i} = \sin(pu_i)$$
$$PE_{p,2i+1} = \cos(pu_i)$$

として

$$PE_{p+k,2i} = \sin((p+k)u_i)$$
$$= \sin(pu_i)\cos(ku_i) + \cos(pu_i)\sin(ku_i)$$
$$= PE_{p,2i}\cos(ku_i) + PE_{p,2i+1}\sin(ku_i)$$

となり、 $PE_{p,2i}$ 及び $PE_{p,2i+1}$ の線形和で表せることがわかりました。

実装は以下のようにしました。(本家のコードとは大きく違います)

$\cos(x) = \sin(x + \pi/2)$ であることを使っています。

transformer/embedding.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/embedding.py#L33)


```

class AddPositionalEncoding(tf.keras.layers.Layer):
    ...

    入力テンソルに対し、位置の情報を付与して返すレイヤーです。
    see: https://arxiv.org/pdf/1706.03762.pdf

    PE_{pos, 2i} = sin(pos / 10000^{2i / d_model})
    PE_{pos, 2i+1} = cos(pos / 10000^{2i / d_model})
    ...

    def call(self, inputs: tf.Tensor) -> tf.Tensor:
        fl_type = inputs.dtype
        batch_size, max_length, depth = tf.unstack(tf.shape(inputs))

        depth_counter = tf.range(depth) // 2 * 2 # 0, 0, 2, 2, 4, ...
        depth_matrix = tf.tile(tf.expand_dims(depth_counter, 0), [max_length, 1]) # [max_length,
        depth_matrix = tf.pow(10000.0, tf.cast(depth_matrix / depth, fl_type)) # [max_length, de

        # cos(x) == sin(x + pi/2)
        phase = tf.cast(tf.range(depth) % 2, fl_type) * math.pi / 2 # 0, pi/2, 0, pi/2, ...
        phase_matrix = tf.tile(tf.expand_dims(phase, 0), [max_length, 1]) # [max_length, depth]

        pos_counter = tf.range(max_length)
        pos_matrix = tf.cast(tf.tile(tf.expand_dims(pos_counter, 1), [1, depth]), fl_type) # [ma

        positional_encoding = tf.sin(pos_matrix / depth_matrix + phase_matrix)
        # [batch_size, max_length, depth]
        positional_encoding = tf.tile(tf.expand_dims(positional_encoding, 0), [batch_size, 1, 1])

        return inputs + positional_encoding

```

本家の実装 (https://github.com/tensorflow/models/blob/master/official/transformer/model/model_utils.py#L28) は論文とは微妙に違います。論文では各時刻の PE は [sin, cos, sin, cos, ...] と交互に並ぶのですが、本家実装では [sin, sin, ..., cos, cos, ...] と並ぶ様になっています。ただ上の式変換でわかるように、要は sin に対応する cos の要素があればいいだけなのでこうしているものと思われる。

ちなみに Positional Encoding は必ずしもこの形である必要があるわけではなく BERT (<https://arxiv.org/abs/1810.04805>) では Positional Encoding を変数にしてしまつて学習で獲得するようにしています。

Token Embedding

文を単語などで分割した各トークンは数値 (int) なのですが、DeepLearning で扱うためにはこれを Embedded Vector にする必要があります。

tensorflow で言語処理をやったことがある人にはおなじみ tf.nn.embedding_lookup です。

本家のコード (https://github.com/tensorflow/models/blob/master/official/transformer/model/embedding_layer.py) を覗いてみると、最後に embedding を隠れ層の次元 (hidden_dim) に応じてスケールさせているようです。

以下のように実装してみます。

[transformer/embedding.py \(https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/embedding.py#L7\)](https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/embedding.py#L7)

```

PAD_ID = 0

class TokenEmbedding(tf.keras.layers.Layer):
    def __init__(self, vocab_size: int, embedding_dim: int, dtype=tf.float32, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.dtype_ = dtype

    def build(self, input_shape: tf.TensorShape) -> None:
        self.lookup_table = self.add_variable(
            name='token_embedding',
            shape=[self.vocab_size, self.embedding_dim],
            dtype=self.dtype_,
            initializer=tf.random_normal_initializer(0., self.embedding_dim ** -0.5),
        )
        super().build(input_shape)

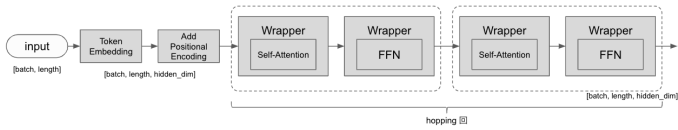
    def call(self, input: tf.Tensor) -> tf.Tensor:
        mask = tf.to_float(tf.not_equal(input, PAD_ID))
        embedding = tf.nn.embedding_lookup(self.lookup_table, input)
        embedding *= tf.expand_dims(mask, -1) # 元々 PAD だった部分を0にする
        return embedding * self.embedding_dim ** 0.5

```

これで細かい部品は出揃いました！

Encoder

Transformer は大きく Encoder と Decoder からなります。
ここでは入力トークン列をエンコードする Encoder を作ります。
Encoder は Self-Attention によって入力をエンコードしていきます。



(<https://camo.qiitausercontent.com/bafd1edb8464cf01419981b80573c7305af57ca2/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e63666d2f302f36313037392f36336639343733382d663066372d333538372d393036312d3739643631373531373432352e706e67>)

Wrapper は ResidualNormalizationWrapper です。

transformer/transformer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/transformer.py#L117)

```
class Encoder(tf.keras.models.Model):
    ...
    トークン列をベクトル列にエンコードする Encoder です。
    ...

    def __init__(
        self,
        vocab_size: int,
        hopping_num: int,
        head_num: int,
        hidden_dim: int,
        dropout_rate: float,
        max_length: int,
        *args,
        **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.token_embedding = TokenEmbedding(vocab_size, hidden_dim)
        self.add_position_embedding = AddPositionalEncoding()
        self.input_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

        self.attention_block_list: List[List[tf.keras.models.Model]] = []
        for _ in range(hopping_num):
            attention_layer = SelfAttention(hidden_dim, head_num, dropout_rate, name='self_attent')
            ffn_layer = FeedForwardNetwork(hidden_dim, dropout_rate, name='ffn')
            self.attention_block_list.append([
                ResidualNormalizationWrapper(attention_layer, dropout_rate, name='self_attention'),
                ResidualNormalizationWrapper(ffn_layer, dropout_rate, name='ffn_wrapper'),
            ])
        self.output_normalization = LayerNormalization()

    def call(
        self,
        input: tf.Tensor,
        self_attention_mask: tf.Tensor,
        training: bool,
    ) -> tf.Tensor:
        ...
        モデルを実行します

        :param input: shape = [batch_size, length]
        :param training: 学習時は True
        :return: shape = [batch_size, length, hidden_dim]
        ...

        # [batch_size, length, hidden_dim]
        embedded_input = self.token_embedding(input)
        embedded_input = self.add_position_embedding(embedded_input)
        query = self.input_dropout_layer(embedded_input, training=training)

        for i, layers in enumerate(self.attention_block_list):
            attention_layer, ffn_layer = tuple(layers)
            with tf.name_scope(f'hopping_{i}'):
                query = attention_layer(query, attention_mask=self_attention_mask, training=train)
                query = ffn_layer(query, training=training)
            # [batch_size, length, hidden_dim]
        return self.output_normalization(query)
```

(参考) Attention ベースの識別器

すこし横道にそれて、Transformer ではなく Attention ベースの識別器の作り方の話です。
多くのタスクでは、問い合わせのカテゴリ分類であったり感情識別であったり、自然言語処理のタスクは文章生成ではなく文書識別です。

文書識別のタスクを行うには後述する Decoder は必要なく Encoder に一層 Dense Layer をかぶせるだけでできます。
詳しくは BERT の論文 (<https://arxiv.org/abs/1810.04805>)が参考になりますが以下のように作ります。

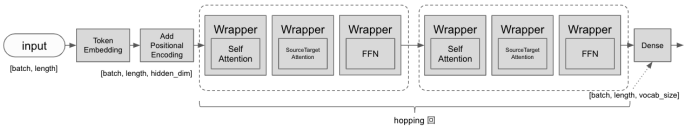
- Encoder への入力(時刻0) の位置に <CLS> というトークンを入れる
 - <CLS> / ああ / 楽しい / ことが / あった のようなデータになります
- Encoder に入力し、 <CLS> の位置の出力のみを取り出します。
- これに Dense Layer をかぶせて識別したいクラスの数の次元に変換します
- あとは softmax をかけたり cross entropy を計算して loss にします

```
encoder_output = self.encoder_layer(...)
cls_vector = encoder_output[:, 0:, :] # cls は必ず文書の最初の位置にあるから `0`
classification_logits = self.dense_layer(cls_vector)
prob = tf.nn.softmax(classification_logits)
```

今回のネットワーク構造では識別器に複数の文を入れることはできませんが、BERT ではそのへんの工夫もされているのでぜひ論文を読んでみてください。

Decoder

Encoder と同様に Decoder を作ります。
Decoder はまず入力に Self-Attention をかけてから、SourceTarget-Attention で Encoder がエンコードした情報を取り込みます。
これによって、時刻 0~t のトークン列を入力として時刻 1~t+1 のトークン列（つまり1時刻未来）を出力します。



(<https://camo.qiitusercontent.com/76e322e3e003247cd78d04b4fbd952b5ac6848d/68747470733a2f2f71696974612d696d6167652d73746772652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f31343033636139612d633061362d376537312d616230342d3065346138636162666132632e706e67>)

transformer/transformer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deep_dialog/transformer/transformer.py#L178)

```

class Decoder(tf.keras.models.Model):
    ...
    エンコードされたベクトル列からトークン列を生成する Decoder です。
    ...

    def __init__(
        self,
        vocab_size: int,
        hopping_num: int,
        head_num: int,
        hidden_dim: int,
        dropout_rate: float,
        max_length: int,
        *args,
        **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.token_embedding = TokenEmbedding(vocab_size, hidden_dim)
        self.add_position_embedding = AddPositionalEncoding()
        self.input_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

        self.attention_block_list: List[List[tf.keras.models.Model]] = []
        for _ in range(hopping_num):
            self_attention_layer = SelfAttention(hidden_dim, head_num, dropout_rate, name='self_a
            enc_dec_attention_layer = MultiheadAttention(hidden_dim, head_num, dropout_rate, name
            ffn_layer = FeedForwardNetwork(hidden_dim, dropout_rate, name='ffn')
            self.attention_block_list.append([
                ResidualNormalizationWrapper(self_attention_layer, dropout_rate, name='self_atten
                ResidualNormalizationWrapper(enc_dec_attention_layer, dropout_rate, name='enc_dec
                ResidualNormalizationWrapper(ffn_layer, dropout_rate, name='ffn_wrapper'),
            ])
        self.output_normalization = LayerNormalization()
        # 注: 本家ではここは TokenEmbedding の重みを転地したものを使っている
        self.output_dense_layer = tf.keras.layers.Dense(vocab_size, use_bias=False)

    def call(
        self,
        input: tf.Tensor,
        encoder_output: tf.Tensor,
        self_attention_mask: tf.Tensor,
        enc_dec_attention_mask: tf.Tensor,
        training: bool,
    ) -> tf.Tensor:
        ...
        モデルを実行します

        :param input: shape = [batch_size, length]
        :param training: 学習時は True
        :return: shape = [batch_size, length, hidden_dim]
        ...

        # [batch_size, length, hidden_dim]
        embedded_input = self.token_embedding(input)
        embedded_input = self.add_position_embedding(embedded_input)
        query = self.input_dropout_layer(embedded_input, training=training)

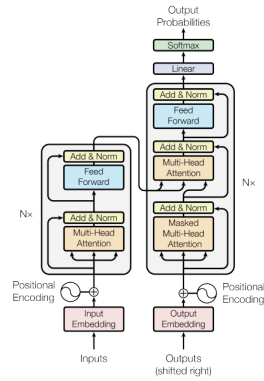
        for i, layers in enumerate(self.attention_block_list):
            self_attention_layer, enc_dec_attention_layer, ffn_layer = tuple(layers)
            with tf.name_scope(f'hopping_{i}'):
                query = self_attention_layer(query, attention_mask=self_attention_mask, training=
                query = enc_dec_attention_layer(query, memory=encoder_output,
                    attention_mask=enc_dec_attention_mask, training=t
                query = ffn_layer(query, training=training)

        query = self.output_normalization(query) # [batch_size, length, hidden_dim]
        return self.output_dense_layer(query) # [batch_size, length, vocab_size]

```

全体像

最後に Encoder と Decoder をつなげると Transformer の出来上がりです！



(<https://camo.qiitausercontent.com/5af7348bde95e4f6c52da9c0f1a2c6a95a64510a/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e63666d2f302f36313037392f65366532316335612d346432392d3033333642d373731312d3834393337653235366332302e706e67>)

transformer/transformer.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/transformer.py#L11)

```

PAD_ID = 0

class Transformer(tf.keras.models.Model):
    '''
    Transformer モデルです。
    '''
    def __init__(
        self,
        vocab_size: int,
        hopping_num: int = 4,
        head_num: int = 8,
        hidden_dim: int = 512,
        dropout_rate: float = 0.1,
        max_length: int = 200,
        *args,
        **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.vocab_size = vocab_size
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate
        self.max_length = max_length

        self.encoder = Encoder(
            vocab_size=vocab_size,
            hopping_num=hopping_num,
            head_num=head_num,
            hidden_dim=hidden_dim,
            dropout_rate=dropout_rate,
            max_length=max_length,
        )
        self.decoder = Decoder(
            vocab_size=vocab_size,
            hopping_num=hopping_num,
            head_num=head_num,
            hidden_dim=hidden_dim,
            dropout_rate=dropout_rate,
            max_length=max_length,
        )

    def build_graph(self, name='transformer') -> None:
        '''
        学習/推論のためのグラフを構築します。
        '''
        with tf.name_scope(name):
            self.is_training = tf.placeholder(dtype=tf.bool, name='is_training')
            # [batch_size, max_length]
            self.encoder_input = tf.placeholder(dtype=tf.int32, shape=[None, None], name='encoder')
            # [batch_size]
            self.decoder_input = tf.placeholder(dtype=tf.int32, shape=[None, None], name='decoder')

            logit = self.call(
                encoder_input=self.encoder_input,
                decoder_input=self.decoder_input[:, :-1], # 入力 EOS を含めない
                training=self.is_training,
            )
            decoder_target = self.decoder_input[:, 1:] # 出力は BOS を含めない

            self.prediction = tf.nn.softmax(logit, name='prediction')

            with tf.name_scope('metrics'):
                xentropy, weights = padded_cross_entropy_loss(
                    logit, decoder_target, smoothing=0.05, vocab_size=self.vocab_size)
                self.loss = tf.identity(tf.reduce_sum(xentropy) / tf.reduce_sum(weights), name='l')

                accuracies, weights = padded_accuracy(logit, decoder_target)
                self.acc = tf.identity(tf.reduce_sum(accuracies) / tf.reduce_sum(weights), name='a')

    def call(self, encoder_input: tf.Tensor, decoder_input: tf.Tensor, training: bool) -> tf.Tensor:
        enc_attention_mask = self._create_enc_attention_mask(encoder_input)
        dec_self_attention_mask = self._create_dec_self_attention_mask(decoder_input)

        encoder_output = self.encoder(
            encoder_input,
            self_attention_mask=enc_attention_mask,
            training=training,
        )
        decoder_output = self.decoder(
            decoder_input,
            encoder_output,
            self_attention_mask=dec_self_attention_mask,
            enc_dec_attention_mask=enc_attention_mask,
            training=training,
        )
        return decoder_output

    def _create_enc_attention_mask(self, encoder_input: tf.Tensor):

```

```
with tf.name_scope('enc_attention_mask'):
    batch_size, length = tf.unstack(tf.shape(encoder_input))
    pad_array = tf.equal(encoder_input, PAD_ID) # [batch_size, m_length]
    # shape broadcasting で [batch_size, head_num, (m|q)_length, m_length] になる
    return tf.reshape(pad_array, [batch_size, 1, 1, length])

def _create_dec_self_attention_mask(self, decoder_input: tf.Tensor):
    with tf.name_scope('dec_self_attention_mask'):
        batch_size, length = tf.unstack(tf.shape(decoder_input))
        pad_array = tf.equal(decoder_input, PAD_ID) # [batch_size, m_length]
        pad_array = tf.reshape(pad_array, [batch_size, 1, 1, length])

        autoregression_array = tf.logical_not(
            tf.matrix_band_part(tf.ones([length, length], dtype=tf.bool), -1, 0)) # 下三角が
        autoregression_array = tf.reshape(autoregression_array, [1, 1, length, length])

    return tf.logical_or(pad_array, autoregression_array)
```

Maskの節ですでてきた各マスクはここで作成されます。
PADの部分については、入力に PAD_ID と等しいかで計算を行い、未来の情報を取得できないよう
するのは tf.matrix_band_part を使用して下三角行列を取得しています。

また build_graph というメソッドでグラフモードでの学習用の loss などのノードもろもろを作
成します。
padded_cross_entropy_loss, padded_accuracy ([https://github.com/halhorn/deep_dialog_tutorial/
blob/master/deepdialog/transformer/metrics.py](https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/metrics.py)) は（時間がなかったので）本家のコード ([http
s://github.com/tensorflow/models/blob/master/official/transformer/utis/metrics.py](https://github.com/tensorflow/models/blob/master/official/transformer/utils/metrics.py))をほぼコピ
ベさせていただきました。
PAD 部分は loss や accuracy の計算に入れないこと、label smoothing と言って正解データを単
純な one-hot ではなく、少しなませたものにするなどの工夫がされています。

今回のモデル等のコード全体は [こちら](https://github.com/halhorn/deep_dialog_tutorial/tree/master/deepdialog/transformer) ([https://github.com/halhorn/deep_dialog_tutorial/tree/m
aster/deepdialog/transformer](https://github.com/halhorn/deep_dialog_tutorial/tree/master/deepdialog/transformer))
また、私は Tensorflow でのモデル構築を TDD (Test Driven Development) でやるのが大好きな
ので、テストのコード ([https://github.com/halhorn/deep_dialog_tutorial/tree/master/test/deepdial
og/transformer](https://github.com/halhorn/deep_dialog_tutorial/tree/master/test/deepdialog/transformer))もあります。

Transformer を学習させる

ここからは作った Transformer を学習させます。
ここでは実験タスクとして、夏目漱石の小説の各文を入れて、その次の文を予測するタスクを学
習させてみましょう。

データの準備

青空文庫から落としてきた夏目漱石の本の文章を使います。
文章から雑に非文部分を除き、文ごとに改行したものです。
(雑な処理で文を分けているので「」内で別れてしまったりいろいろきとうです。)
https://github.com/halhorn/deep_dialog_tutorial/tree/master/data ([https://github.com/halhorn/d
eep_dialog_tutorial/tree/master/data](https://github.com/halhorn/deep_dialog_tutorial/tree/master/data))

トークナイザー

文章 (str) はそのままでは Deep Learning モデルの入力にできません。
なので、下記のように文章を適当なトークンに区切り、更にそれを数値 (ID) に変換していま
す。

- 1. 文章： 吾輩は猫である
- 2. トークナイズされた文章： 我輩 / は / 猫 / である
- 3. トークナイズされた ID： 1025 / 24 / 420 / 320

トークナイズ・ID変換をやる方法はいろいろありますが、Google の Taku Kudo さんの作成され
た sentencepiece (<https://github.com/google/sentencepiece>) を今回は使います。
日本語解説記事 (<https://qiita.com/taku910/items/7e52f1e58d0ea6e7859c>) もあります。

SentencePiece は文章集合から学習させることで、文章をてきとうな大きさのトークンによしな
に分割し、ID に変換してくれます。MeCab などていわる単語に分けるよりも未知語が出に
くく、スペースで単語が区切られているなどの言語的制約も無いので日本語のニューラルネット
用トークナイザーとしてとても使いやすいです。

バッチ作成モジュール

学習用のバッチを作成します。

大雑把に以下のようなことをします。

1. 文章をトークナイザーで ID 列に変換
2. 決められた長さで切り詰める
3. Decoder への入力データは前後に bos と eos を入れる
4. バッチ内の各 ID 列の長さを最大のものに揃える (pad を挿入する)
5. numpy の array にする

プログラムはこちら：https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/preprocess/batch_generator.py (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/preprocess/batch_generator.py)

学習を行う

ここまで作った部品を組み合わせで学習を行います。

学習は jupyter でおこないます。

https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/training.ipynb (https://github.com/halhorn/deep_dialog_tutorial/blob/master/deepdialog/transformer/training.ipynb)

training.ipynb


```

# In[1]:
# カレントディレクトリをリポジトリ直下にするおまじない
import os
while os.getcwd().split('/')[-1] != 'deep_dialog_tutorial': os.chdir('.')
print('current dir:', os.getcwd())

# In[2]:
import tensorflow as tf
from deepdialog.transformer.transformer import Transformer
from deepdialog.transformer.preprocess.batch_generator import BatchGenerator

# # Create Data
# In[3]:
data_path = 'data/natsume.txt'

# In[4]:
batch_generator = BatchGenerator()
batch_generator.load(data_path)

# In[5]:
vocab_size = batch_generator.vocab_size

# # Create Model
# In[ ]:
graph = tf.Graph()
with graph.as_default():
    transformer = Transformer(
        vocab_size=vocab_size,
        hopping_num=4,
        head_num=8,
        hidden_dim=512,
        dropout_rate=0.1,
        max_length=50,
    )
    transformer.build_graph()

# # Create Training Graph
# In[ ]:
save_dir = 'tmp/learning/transformer/'
log_dir = os.path.join(save_dir, 'log')
ckpt_path = os.path.join(save_dir, 'checkpoints/model.ckpt')

os.makedirs(log_dir, exist_ok=True)

# In[ ]:
with graph.as_default():
    global_step = tf.train.get_or_create_global_step()

    learning_rate = tf.placeholder(dtype=tf.float32, name='learning_rate')
    optimizer = tf.train.AdamOptimizer(
        learning_rate=learning_rate,
        beta2=0.98,
    )
    optimize_op = optimizer.minimize(transformer.loss, global_step=global_step)

    summary_op = tf.summary.merge([
        tf.summary.scalar('train/loss', transformer.loss),
        tf.summary.scalar('train/acc', transformer.acc),
        tf.summary.scalar('train/learning_rate', learning_rate),
    ], name='train_summary')
    summary_writer = tf.summary.FileWriter(log_dir, graph)
    saver = tf.train.Saver()

# # Train
# In[ ]:
max_step = 100000
batch_size = 128
max_learning_rate = 0.0001
warmup_step = 4000

# In[ ]:
def get_learning_rate(step: int) -> float:
    rate = min(step ** -0.5, step * warmup_step ** -1.5) / warmup_step ** -0.5
    return max_learning_rate * rate

# In[ ]:
with graph.as_default():
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
    step = 0

# In[ ]:
with graph.as_default():
    for batch in batch_generator.get_batch(batch_size=batch_size):
        feed = {
            **batch,
            learning_rate: get_learning_rate(step + 1),
        }
        _, loss, acc, step, summary = sess.run([optimize_op, transformer.loss, transformer.acc, g
summary_writer.add_summary(summary, step)

```

```

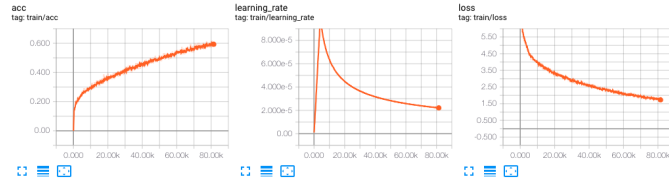
if step % 100 == 0:
    print(f'{step}: loss: {loss},\t acc: {acc}')
    saver.save(sess, ckpt_path, global_step=step)

# In[ ]:

```

learning rate は論文にかかれていたとおり最初0から初めて徐々に増やしていき (Warming-up)、4000ステップを境に減衰させるようにします。

まだ学習途中ですが、accuracy がどんどん上っていますね！



(<https://camo.qiitausercontent.com/dce45ba6c7106b1572ab0924e8c837e716b23414/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36313037392f32643039653264622d333566352d356237352d356163352d3332623763613461333937332e706e67>)

実用的な学習を行うには、training の metrics を見るだけではなく汎化をしているかにもにかしらの方法で見る必要があります。

今回の次の文を生成するタスクや対話学習などでは、ある入力に対し正解の出力がまったくもって一意に決まらないので通常の training set + validation set ではうまく評価できないことが多いです。

このようなタスクの汎化性能を見るのは人が目で見ると今の所ないのであって 1 epoch ごとに適当な入力から何が生成されるか Tensorboard に出す (<https://qiita.com/halhorn/items/2127e5aa3946aa008c32>) のがおすすめです。

今回作(れ)らなかった部分

- ビームサーチ・生成
 - transformer.prediction の値をもとにビームサーチを行うことになります。(今回そこまでやる時間はありませんでした)
 - これが graph mode で作るとクソ難解なコードになります。
- 学習途中の汎化性能の評価
 - 実用的な学習を行うには、training の metrics を見るだけではなく汎化をしているかにもにかしらの方法で見る必要があります。
 - 今回の次の文を生成するタスクや対話学習などでは、ある入力に対し正解の出力がまったくもって一意に決まらないので通常の training set + validation set ではうまく評価できないことが多いです。
 - このようなタスクの汎化性能を見るのは人が目で見ると今の所ないのであって 1 epoch ごとに適当な入力から何が生成されるか Tensorboard に出す (<https://qiita.com/halhorn/items/2127e5aa3946aa008c32>) のがおすすめです。
- Multi-head Attention のメモリのキャッシュ
 - 自己回帰的に生成を行う場合に何度も同じ memory を変換するコストをなくするため、本家のコードではキャッシュ (https://github.com/tensorflow/models/blob/master/official/transformer/model/attention_layer.py#L110)の仕組みが実装されています。

参照

- この記事にかかれていたコード (tf.keras ベース) (https://github.com/halhorn/deep_dialog_tutorial/tree/master/deepdialog/transformer)
 - テスト (https://github.com/halhorn/deep_dialog_tutorial/tree/master/test/deepdialog/transformer)
- 本家 Google の Transformer (旧 tf.layers ベース) (<https://github.com/tensorflow/models/tree/master/official/transformer>)

さいごに

ミクシィ AI ロボット事業部では DeepLearning で対話学習を行う R&D エンジニア及び実装・インフラまわりのエンジニアを募集しています。

ちょうどTransformerのことを学習していたため大変参考になりました
よかったら、学習にかかった時間などを教えてもらえるとありがたいです。



halhorn (/halhorn)
(/halhorn) 1478 contribution

2018-12-09 21:39
いいね 1

お役に立てて幸いです。
Attentionの仕組み時代は元から知っていたり、会社のチームで先にTransformerを使っている方がいたりしたので大体の仕組み元々知ってる状態でした。単体での勉強はryobot氏の記事を読んで本家の実装読んで自分でも実装しつつ分からないところは論文と比較したくらいなので数週間くらいでしょうか。



kotu931226 (/kotu931226)
(/kotu931226) 4 contribution

2018-12-09 21:49
いいね 0

書き方が紛らわしくてすみませんでした。
Transformerが80k step学習するのににかかった時間を教えて貰いたいです。

ちなみに自分もryobot氏の記事を読んでいたので理解するのに1か月以上かかってしまいました。
数週間で理解できるのは羨ましいですね。



halhorn (/halhorn)
(/halhorn) 1478 contribution

2018-12-09 22:25
いいね 0

おっとすみません。私の学習ではなく機械の学習でした。
80kまででGTX1080の1枚で半日〜一日くらいだったと思います。



kotu931226 (/kotu931226)
(/kotu931226) 4 contribution

2018-12-10 11:34
いいね 1

回答ありがとうございます!!
やはりTransformerで学習させるとそれくらい時間かかるものですね
参考にさせていただきます。



taxio (/taxio)
(/taxio) 5 contribution

2019-04-08 15:46
いいね 0

解説記事ありがとうございます。論文を読みながら分からないところを補完するのに活用させてもらっています。

質問なのですが、文章生成時はDecoderは1つ先の単語しか予測できないので、文章全体の生成を終了させるには、その出力を元にもう一度Decoderに入力するという処理を文字列の長さ分(論文では50?)だけ繰り返すという認識で合ってますでしょうか？

また、この再入力のときにビームサーチなどの処理を適用するということでしょうか？



halhorn (/halhorn)
(/halhorn) 1478 contribution

2019-04-08 17:57
いいね 1

質問なのですが、文章生成時はDecoderは1つ先の単語しか予測できないので、文章全体の生成を終了させるには、その出力を元にもう一度Decoderに入力するという処理を文字列の長さ分(論文では50?)だけ繰り返すという認識で合ってますでしょうか？

およそそのとおりです。
1トークン生成したら、その生成結果を次の時刻に入れてまた次の時刻のトークンを生成します。
違うのは、EOS (End of Sentence) というトークンが生成された場合そこで生成をストップする点ですね。もしくは50トークンなど最大値まで生成した場合そこでも終わりです。

また、この再入力のときにビームサーチなどの処理を適用するということでしょうか？

はい。実用上はビームサーチを行わないとまっとうな品質にはならないと思います。

1トークンずつ生成 + ビームサーチのコードの書き方によって生成の速度が大きく変わります。



taxio (/taxio)
(/taxio) 5 contribution

2019-04-08 19:41
いいね 1

ありがとうございます！

論文を読んでもDecoderの振る舞いがいまいちピンとこなかったので、参考になりました！