Transformerを用いた予測モデルの構築

文責：河野日生

# **はじめに**

　Transformerは、2017年に発表された論文「[Attention is All You Need](https://drive.google.com/file/d/10Zh4yXouMpgKtoQhsuBU_yTIVnELikLq/view?usp=share_link)」において提案された機械学習モデルである。従来のRNNやLSTMと異なり、Attention(注意)機構を用いることで、長期依存性の学習を効率的に行うことが可能となり、自然言語処理をはじめとする多くの分野で高い性能を示している。ChatGPTをはじめとするLLM(大規模言語モデル)の基盤技術であり、Generative Pre-trained Transformerという名称にもその技術が反映されているのが見てとれる。さらに、Transformerは言語処理のみならず、センサデータを用いた分類タスクや回帰予測タスクにおいても効果を発揮するため、村尾研での研究に大いに役に立つと考えられる。  
　本稿は、筆者がTensorFlowを用いて実装した[TransformerのEncoder部のコード](https://drive.google.com/file/d/1gBKchmKqqH-fPcaT8Bd3BFAl8Tef8b2D/view?usp=share_link)を、村尾研の後輩諸氏に引き継ぐことを目的としている。また、筆者が研究活動を行う中で得たTransformerの知見を整理・再構成している。筆者はTransformerについての理解が大変乏しく浅はかであるため、あくまで本稿はTransformerの利用のきっかけ・取っ掛かり程度としてほしい。本稿が後輩諸氏の研究の一助になることを願う。

# **とにかく動かしてみたい人向け**

* [筆者のコード](https://drive.google.com/file/d/1gBKchmKqqH-fPcaT8Bd3BFAl8Tef8b2D/view?usp=share_link)、[Google Colaboratory](https://colab.research.google.com)、一般公開されている[UCI HAR データセット](https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones)を用いて、Transformerによるモデル構築の流れを手っ取り早く体験することができるJupyter Notebookを用意した。
* [Let's\_Try\_Transformer.ipynb](https://colab.research.google.com/drive/1mtG-YCMq2jbKARBuYHZ5aiA3X6jGiyaR?usp=share_link)を開いて、上から順番に実行すれば良い。
* 自分の手でコードを改良したいときは、必ずコピーしてから行うこと(元のコードは変えないように)。

# **環境構築**

* TensorFlowは他ライブラリとの依存関係やPythonのバージョンとの兼ね合いが複雑であるため、pyenvを用いたPythonのバージョン管理や、pyenv-virtualenv、venv、Dockerなどを用いた仮想環境の構築を強く推奨する。
* 自身のPC上での環境構築が難しければ、[Google Colaboratory](https://colab.research.google.com)や[研究室ワークステーション](https://www.iis.ise.ritsumei.ac.jp/local/workStation)を用いるのが望ましい。
* TensorFlowをインストール後、import tensorflow as tfを実行することでライブラリを読み込むことができる。

　以下は、ローカルのPCで環境構築する場合の注意点を筆者の視点からまとめたものである。詳しくは読者自身の手で調べてほしい。

### **Windows向け**

* 基本的には[公式ドキュメント](https://www.tensorflow.org/install?hl=ja)の通り実行すれば問題ない
* pip install tensorflowを実行
* GPUを用いた計算を行う場合、[GPU サポート | TensorFlow](https://www.tensorflow.org/install/gpu?hl=ja)を参考にしてNVIDIAドライバとそれに対応するCUDA Toolkit、cuDNNをインストールし、パスを通す必要がある。
* TensorFlow含め全てのマテリアルの対応バージョンの整合性を取らないといけないのでかなり面倒。

### **Mac OS向け**

* Mac OSでは、pip install tensorflowでTensorFlowをインストールしたのち、追加でMac OS用の[tensorflow-metal](https://developer.apple.com/metal/tensorflow-plugin/)を、pip install tensorflow-metalでインストールすることで、MacでもGPUを使った計算が可能になる
* インポートは同様にimport tensorflow as tfでOK

print(f"TensorFlow has access to the following devices:\n{tf.config.list\_physical\_devices()}")  
を実行し、  
TensorFlow has access to the following devices: [PhysicalDevice(name='/physical\_device:CPU:0', device\_type='CPU'), PhysicalDevice(name='/physical\_device:GPU:0', device\_type='GPU')]  
と表示されると、GPUを用いた計算が可能な状態になっている。

# **サンプルコードを用いたモデルの構築**

　細かい理屈を抜きにして、とりあえず[筆者のサンプルコード](https://drive.google.com/file/d/1gBKchmKqqH-fPcaT8Bd3BFAl8Tef8b2D/view?usp=share_link)を用いてモデルを構築したい人向けに、サンプルコードの実行方法と各パラメータについて説明する。本コードでは「2値分類」「多クラス分類」「回帰予測」の3つのタスクを実行できるようにした。本コードをライブラリとしてインポートするか、サンプルコード内の関数を自身のソースコードにコピペすれば良い。

### **モデルを構築するまで/モデルに入力するまで**

* 基本的には通常の機械学習モデルの構築方法に準ずれば問題はなく、データの分割や標準化や欠損値処理などを通常の手順通り行えば良い(こういった処理は機械学習において極めて基本的な手順であるため、本稿では割愛する)。
* データの長さが不揃いである場合、以下のようにデータ長が最大のものに合わせてパディングを行う。もしくは固定の最大データ長を予め設定し、それに合わせてパディングやトランケーションを行う。

PAD = 0.0

max\_length\_train = max(len(seq) for seq in X\_train\_scaled)

max\_length\_test = max(len(seq) for seq in X\_test\_scaled)

max\_length = max(max\_length\_train, max\_length\_test)

#シーケンスのパディング

X\_train\_padded = pad\_sequences(X\_train\_scaled,maxlen=max\_length, padding='post', value=PAD, dtype='float32')

X\_test\_padded = pad\_sequences(X\_test\_scaled, maxlen=max\_length, padding='post', value=PAD, dtype='float32')

* モデル構築時にパディング部分を無視する設定を行えば良い(後述)。

### **各パラメータの意味**

| パラメータ | 説明 | 目安 |
| --- | --- | --- |
| max\_length | シーケンスの最大長 (時系列データの時間ステップ数) | 入力データに応じる |
| d\_model | 特徴量の次元数 (入力する特徴のサイズ) | 入力データに応じる (原著論文では512だが、これは単語の意味ベクトルの次元である。村尾研での研究で扱うようなセンサデータでは、多くてもせいぜい10数次元だと考えられる) |
| key\_dim | 各アテンションヘッドの次元数 | 原著論文ではd\_model/num\_headsの値 (とりあえずおよそこの数値から始めて、後から調整すれば良い。TensorFlowのモジュールでは内部のDense層で調整されるので極論好きな数値で良い) |
| num\_heads | アテンションヘッドの数 (Self-Attentionの並列数) | 原著論文では8 (2の倍数が望ましい) |
| ff\_dim | Feed Forward Networkの中間次元数 | 原著論文ではd\_modelの4倍  (とりあえずおよそこの数値から始めて、後から調整すれば良い) |
| num\_transformer\_blocks | Transformer Encoderブロックの数 | 原著論文では6層 (これは自然言語処理の翻訳タスクでの利用を想定している。村尾研で扱うような、少量かつ短いセンサデータであれば1層で十分) |
| dropout | ドロップアウト率 | 0.1 |
| l2\_lambda | L2 正則化の係数 | 1e-4 |
| pad | パディング値 (マスキングする値) | 0.0 (モデル構築前に入力データを何の数値でパディングしたのかを、モデル構築時に渡して上げるイメージ) |
| pooling | プーリングの方法 | 'average' or 'max' (基礎知識のため説明は割愛) |
| task | 実行するタスクの種類 | 'binary' or 'multiclass' or 'regression' (2値分類、多クラス分類、回帰) |
| num\_classes | 多クラス分類の場合のクラス数 (task='multiclass' の場合に必要) | 入力データに応じる |
| regression\_units | 回帰予測の場合の出力ユニット数 (task='regression' の場合に必要) | 入力データに応じる |
| layer\_norm\_epsilon | レイヤー正規化のイプシロン値 | 1e-6 |

### **パラメータの指定→モデルの構築**

config = TransformerConfig(

max\_length=max\_length, *# シーケンスの長さ*

d\_model=X\_train\_final.shape[2], *# 特徴量の次元数*

key\_dim=30, *# 各アテンションヘッドの次元数*

num\_heads=8, *# アテンションヘッドの数*

ff\_dim=X\_train\_final.shape[2] \* 4, *# FeedForward ネットワークの中間次元数*

num\_transformer\_blocks=1, *# トランスフォーマーブロックの数*

dropout=0.1, *# ドロップアウト率*

l2\_lambda=1e-4, *# L2正則化係数*

pad=PAD, *# パディングの値*

pooling='average', *# プーリングの方法 ('average' または 'max')*

task='multiclass', *# タスクの種類(この場合は多クラス分類)*

num\_classes=6 *# クラス数*

)

*# モデルの構築*

model = build\_transformer\_model(config)

### **モデルのコンパイル**

　以下に登場する用語は機械学習全般における基礎的な知識であるため、本稿では説明を割愛し、あくまで本コードの実行における推奨設定のみ掲載する。

model.compile(

optimizer=Adam(learning\_rate=0.001),

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy']

)

* optimizer(最適化アルゴリズム)：基本的にAdamで問題ない
* loss(損失関数)
  + 二値分類の場合：binary\_crossentropy
  + 多クラス分類の場合
    - 正解ラベルが整数：sparse\_categorical\_crossentropy
    - 正解ラベルがone-hot：categorical\_crossentropy
  + 回帰予測の場合：mean\_squared\_errorかmean\_absolute\_error
* metrics(評価関数)
  + 分類の場合：accuracy
  + 回帰の場合：maeやmse

### **学習**

model.fit(

X\_train\_final, y\_train\_final,

epochs=30,

batch\_size=16,

validation\_data=(X\_val, y\_val)

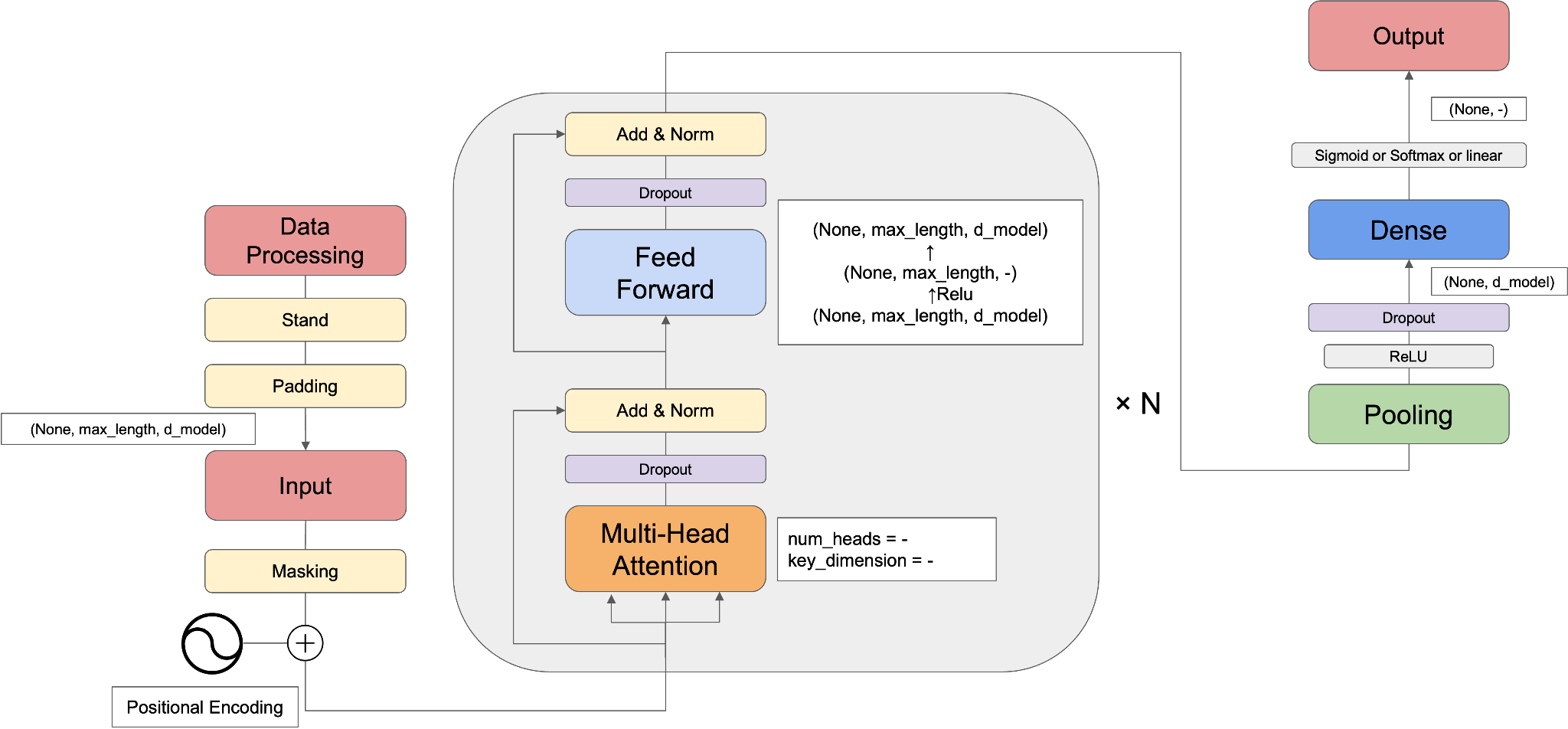
)

* epoch(エポック数)：データ数による。学習曲線などを見ながら適宜調整するのが良い。EarlyStoppingなどのコールバック(一定エポック後に改善が見られなければ学習を打ち切る設定)の導入もアリ。
* batch\_size(バッチサイズ)：データ数や実装環境のメモリ性能などによる。16、32、64などから試し、学習の進行状況や安定性、検証結果を基に最適な値を決定するのが良い。
* [Let's\_Try\_Transformer.ipynb](https://colab.research.google.com/drive/1mtG-YCMq2jbKARBuYHZ5aiA3X6jGiyaR?usp=share_link)にはEarlyStoppingや学習後の学習曲線のプロット、テストデータによる予測やその評価などのコードも加えている。参考にされたし。

# **Transformer/サンプルコードの解説**

　本節では、筆者のサンプルコードとTransformerの概念について詳しい解説を行う。また、筆者が感じた疑問や、読者が読み進める上で当然生じるであろう細かい疑問点については、最後の[疑問点と回答](#_yl7mg03j0kol)に一問一答形式で集約してある。

### **筆者のコードのモデル図**



　Data Processingでデータの前処理や標準化、パディングをしたのち、モデルに入力する。入力時はデータの最大長と特徴量の次元数(d\_model)の形状。Noneになっている部分は一度に入力されるデータの数になる(省略して良いと思う)。その後、マスキングや位置エンコーディングを経てTransformerのEncoder部分に入力される。num\_headsがAttentionヘッドの数、key\_dimensionが一つのヘッドの内部の次元数を示しており、本図では-にしてある。また、Feed Forward Network層の中間次元も-としてある。×NはEncoderブロックの層の数を示しており、1層ならばN=1、 6層ならばN=6となる。

### **位置エンコーディング/get\_positional\_encoding関数**

* Transformerは、Attention機構により入力全体を一括して行列演算で処理するため、計算効率が高い。しかし、RNNやLSTMのように逐次処理する場合と異なり、入力シーケンスの各要素がどの順番に現れるかという情報が失われてしまう。
* そこで、sin波とcos波を用いて、各入力に「これはシーケンスの何番目の要素であるか」という情報を与える。
* 各次元で以下の式によって異なる周波数のsin波・cos波を生成することで、モデルは入力の順序や、隣接する要素間の相対的な位置関係を学習しやすくなる。

### **パディングマスクの作成/create\_attention\_mask関数**

* 現実のデータセットでは、データの長さが一定でない場合が多い。例えば、ジェスチャ認識や行動認識などでは、一回のジェスチャや行動の長さが不揃いであるため、最長のシーケンスに合わせて短いデータを共通の値(基本的には0.0)で埋めるパディング処理を行う。
* しかし、計算時にはこのパディング部分は意味のない情報であるため、無視する必要がある。
* create\_attention\_mask関数は、入力テンソル中の各タイムステップについて、全特徴量がパディング値(config.pad)と一致するかどうかを調べ、パディング部分であれば0.0、そうでなければ1.0のマスク行列を返す。
* このマスク行列は、後続のMultiHeadAttentionなどでパディング部分を計算から除外するために用いられる。

### **Encoder部の構造/transformer\_encoder関数**

　TransformerのEncoderブロックは、Multi Head Attention機構とFeed Forward Netoworkから構成され、さらに残差接続および層正規化が各ブロック内に組み込まれている。Multi Head Attentionレイヤーについては、Transformerの中核となる概念であるため、本節内での簡単な解説に加え、さらに別途[詳しい解説を行う章](#_tkd8mfvqpl4r)を設けている。  
　ここでは、TransformerのEncoderについて全体の流れを確認する。具体的な処理の流れは以下の通りである。

1. **パディングマスクの適用**
   * mask\_layer = Lambda(lambda x: create\_attention\_mask(x, config.pad))(x)  
     により、入力xに対してパディング部分(ここでは値が config.pad、通常は0.0にすることがほとんど)を無視するためのマスクを生成する。
   * 生成されたマスクは、MultiHeadAttentionの計算に渡され、パディングした部分がAttention計算に影響しないようにする。
2. **MultiHeadAttentionレイヤー**
   * attn\_out = MultiHeadAttention(key\_dim=config.key\_dim,　num\_heads=config.num\_heads,　dropout=config.dropout)(x, x, x, attention\_mask=mask\_layer)  
     により、Attentionの計算が実行される。
   * Attentionとは「入力データの中で、どの部分がどの部分に注目すべきか」を決めるアルゴリズム。
   * 引数はx, x, xで全て同じであるため、Self-Attention計算(後述)を行う。
   * **マスク処理：**attention\_mask=mask\_layerが指定されている場合、Attention計算時に特定の入力位置への注意を抑制するマスク処理が適用される。これにより、パディングされた部分を無視したり、未来の情報への計算を制限したりできる。
   * **ドロップアウト:** Attentionの出力に対してドロップアウトが適用される。
   * 最終的に、各ヘッドの出力を結合してattn\_outとして出力する。attn\_outは、入力の各位置に対して、他のすべての位置からの情報を重み付けして集約したものである。
3. **残差接続と層正規化**
   * out1 = LayerNormalization(epsilon=config.layer\_norm\_epsilon)(x + attn\_out)  
     では、x(元の入力)にMultiHeadAttentionの出力(attn\_out)を加算し、その結果に層正規化を適用している。
   * **残差接続**：入力と出力を加算することで、「入力からの変化量」だけを学習するよう促す仕組み。これにより勾配消失問題が緩和され、深いネットワークでも効率的に学習できるようになる。
   * **層正規化**：各特徴量の分布を平均0、分散1に正規化することで学習の安定性を向上させモデルの収束を早める。特にTransformerのような深いネットワークでは、この正規化が勾配のスケールを安定させる重要な役割を果たす。
   * 分散が0に近い場合や0の場合に、ゼロ除算の発生を防ぐため、分母に小さな定数layer\_norm\_epsilonを加える。
4. **Feed Forward Network**
   * ff\_out = Dense(config.ff\_dim, activation='relu', kernel\_regularizer=l2(config.l2\_lambda))(out1)  
     ff\_out = Dense(out1.shape[-1], kernel\_regularizer=l2(config.l2\_lambda))(ff\_out)ff\_out = Dropout(config.dropout)(ff\_out)  
     out2 = LayerNormalization(epsilon=config.layer\_norm\_epsilon)(out1 + ff\_out)  
     で、2層のDense層からなるFeed Forward Networkが適用される。
   * 最初のDense層では中間次元(config.ff\_dim)に変換し、活性化関数reluで非線形性を導入する。第二のDense層では元の次元に戻す。
   * 全体の出力にはDropoutが適用され、最後に残差接続(out1 + ff\_out)と層正規化により出力が整えられる。また、l2\_lambdaによるL2正則化を行っている。
   * これにより、Self-Attentionで得た関係性の情報がさらに高度な特徴に変換され、各タイムステップごとの表現力が強化される。
   * Attention機構は入力の線形な組み合わせで情報を集約する。しかし、現実のデータには複雑なパターンが存在するため、FFNを使うことでAttentionだけでは捉えきれない非線形な関係性や特徴を学習できるようになる。

### **モデルの構築/build\_transformer\_model関数**

　build\_transformer\_model関数は、Transformerモデル全体のアーキテクチャを組み立てるための中心的な部分である。以下の処理が順次行われる。

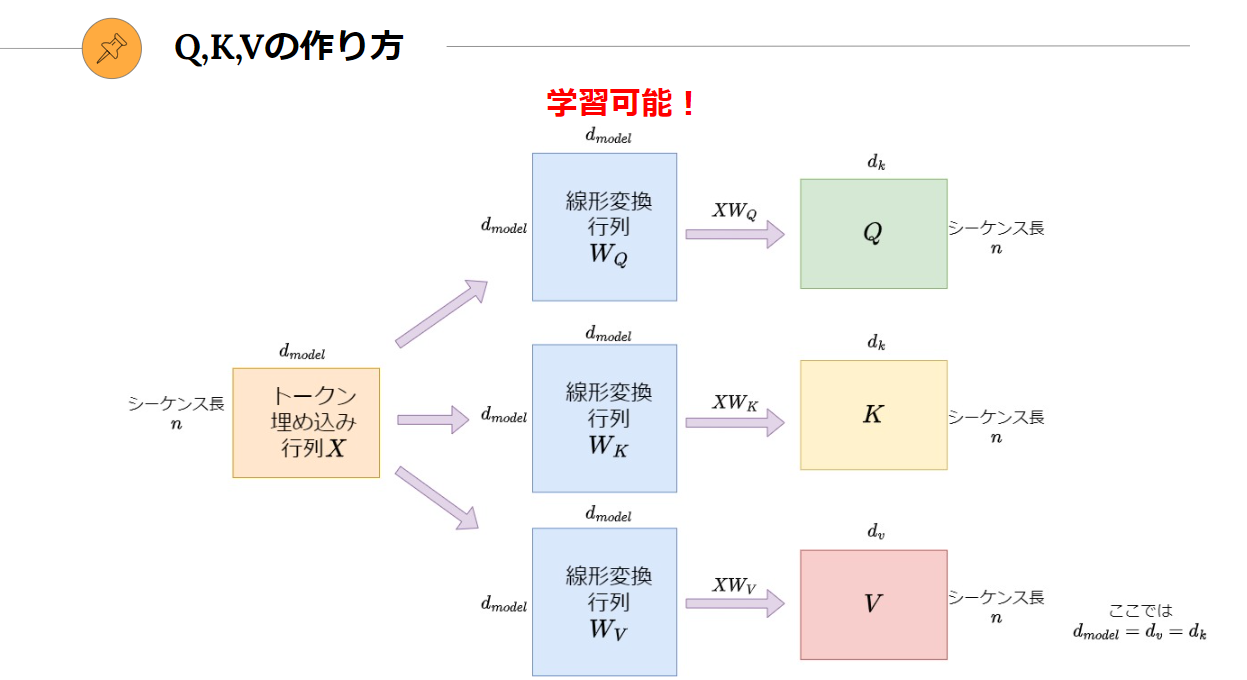
1. **入力層の定義とマスキング処理**
   * inputs = Input(shape=(config.max\_length, config.d\_model), name="input\_layer")  
     x = Masking(mask\_value=config.pad, name="masking\_layer")(inputs)  
     で、シーケンス長(config.max\_length)と各タイムステップの特徴量次元(config.d\_model)を指定して入力の形状を定義する。
   * Masking層を適用することで、入力データ中のパディング値(config.pad)に対応する部分を無視するようにする。
2. **位置エンコーディングの付与**
   * pos\_encoding = get\_positional\_encoding(config.max\_length, config.d\_model)  
     x = x + pos\_encoding[:, :config.max\_length, :]  
     で、[位置エンコーディングの作成](#_3728vk2a3oyq)とその付与を行っている。
3. **Encoderブロックの積み重ね**
   * for \_ in range(config.num\_transformer\_blocks):  
      x = transformer\_encoder(x, config)

では、設定された数(config.num\_transformer\_blocks)だけ、[transformer\_encoder関数](#_c5gjlk3s90tr)を呼び出してEncoderブロックを積み重ねる。

1. **プーリング処理による特徴の集約**
   * Encoderブロックの出力は、シーケンス全体の各タイムステップの情報が並んだ状態である。
   * これをGlobalAveragePooling1DまたはGlobalMaxPooling1Dにより、1つの固定長のベクトルに集約する。
2. **活性化と正則化**
   * プーリング後、ReLU活性化関数を適用することで非線形性を付与し、続いてDropoutにより過学習の抑制を図る。
3. **タスクに応じた出力層の構築**
   * 最後に、分類や回帰などの具体的なタスクに合わせ、Dense層を用いて最終出力を生成する。

# **Multi Head Attention内で何が行われているのか** 　本章は、筆者が論文、公式ドキュメント、様々なテックブログから得られた知見を、入力xがMulti Head Attentionレイヤーに入力されてから出力が生成されるまでの順番に整理・再構成したものである。全てが正しいという保証はなく、筆者の独自の解釈も含む。あくまでも理解のスタートステップとして読んでほしい。

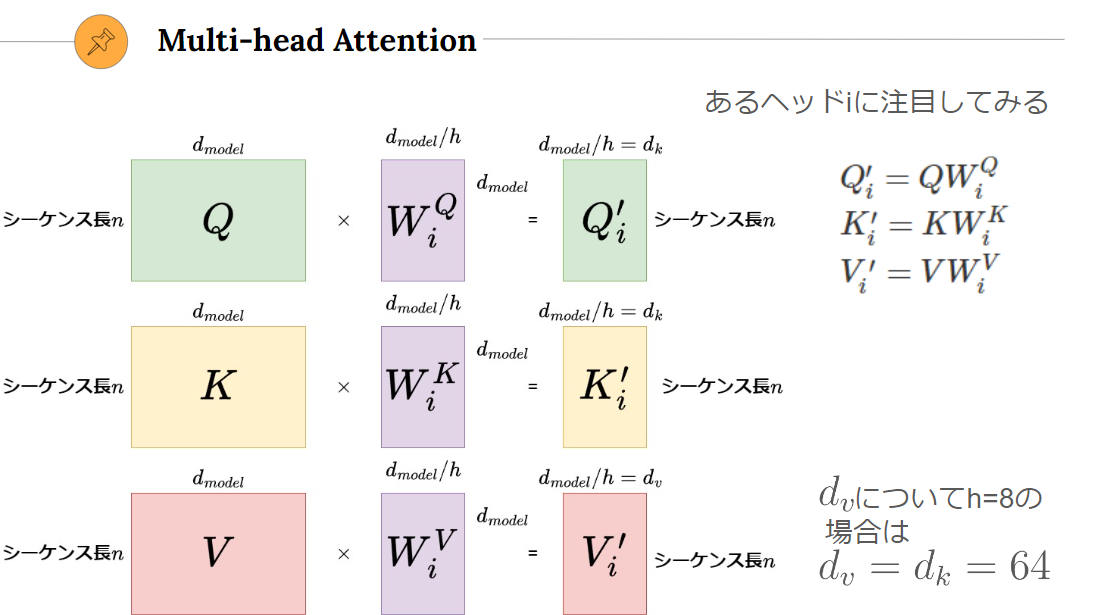
### **入力xから、Query, Key, Valueベクトルを作る**

* + 位置エンコーディングが付与された  
    入力(形状：max\_length, d\_model)に対して、  
    重み行列()(形状：d\_model, num\_heads × key\_dimの行列)を掛けることで、ベクトルが生成される。
  + 重み行列()は最初はランダムな値であり、学習epochを重ねるたびに誤差逆伝播法で修正され、最適な値に決まっていく。つまりTransformerは、入力に対して最適な重み行列のパラメータを見つけるためのアルゴリズムであるというのが筆者の解釈である。(誤差逆伝播法はニューラルネットワーク全般における基礎知識であり、Transformer固有の概念ではないため説明は割愛。)  
    

(画像引用：参考文献1)

### **Query, Key, Valueベクトルを、num\_heads個に分割する**このプロセスでは、Transformerの概念上でのやり方とTensorFlowのMultiHeadAttentionモジュール内部における実装方法に若干の違いがある。しかし、両者は数学的に同じである。本節では両方について述べる。

* + **Transformerの概念上**
    - 原著論文では、生成したQuery, Key, Valueベクトルにさらに重み行列(, の範囲は1からnum\_heads)をかけることで、各アテンションヘッドごとに異なるを生成する。



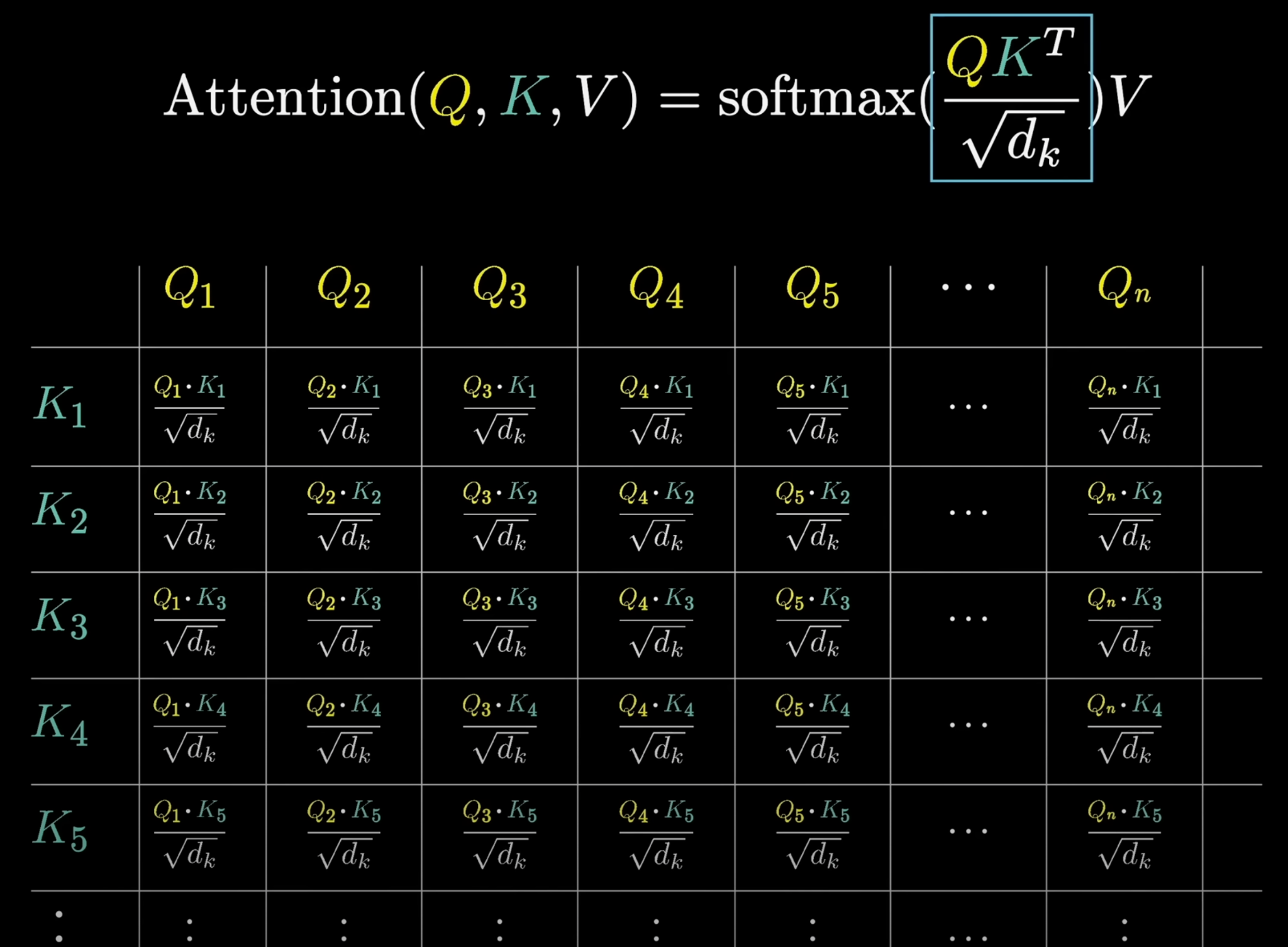
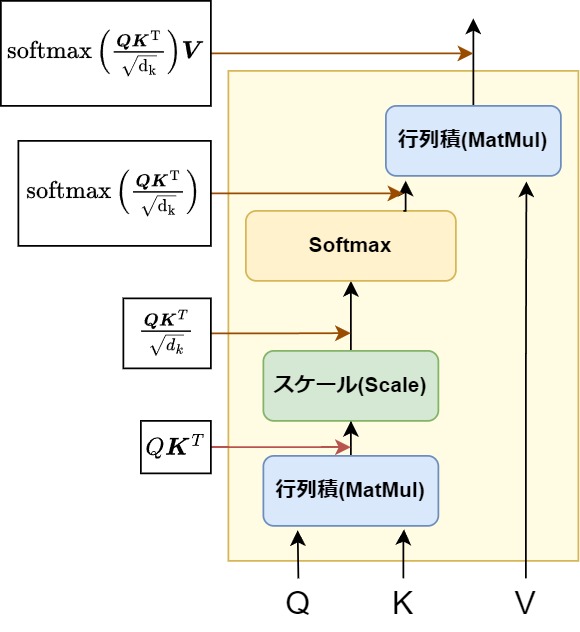
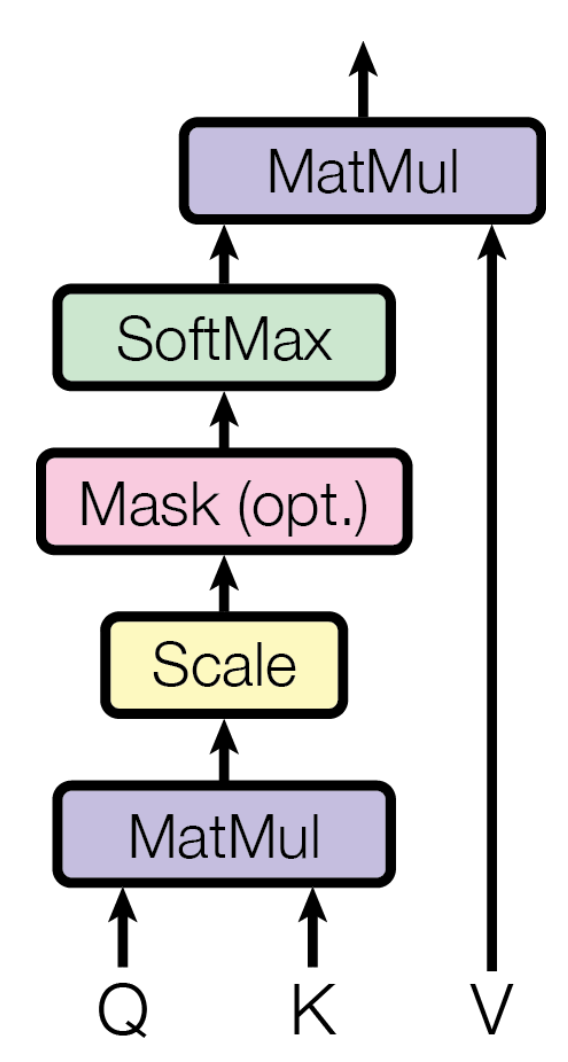
(画像引用：参考文献1)

* + **TensorFlowの実装上**
    - 生成されたQuery、Key、Valueをそのままnum\_heads個に分割する。それぞれが独立してAttention計算(後述)を行う。
    - (形状：max\_length, d\_model)に重み行列(形状：d\_model, num\_heads × key\_dim)をかけることで、直接出力()(形状：max\_length, num\_heads × key\_dim)を生成する。
    - 論文上では、Q, K, Vを生成したあとさらに個別に重み行列を用いているように読めるが、実装上では生成したQ, K, VをそのままDense層(全結合層)で分割・射映することで各アテンションヘッドごとに違う入力を作っている。
    - 各ヘッドごとにとって見てみると、xにをかけた後、さらにをかけるのはとなるが、とすれば同じこととなるのでTensorFlow上ではこう実装されている、と筆者は解釈した。
    - 以降でも、重み行列をかけることは全結合層で線形変換することなので、TensorFlowではDense層に入力することで実装されている。基本的に重み行列をかけること=Dense層に入力することになっていると思えば良い。

### **Scaled Dot-Product Attentionでスコア計算する(Attention計算)**

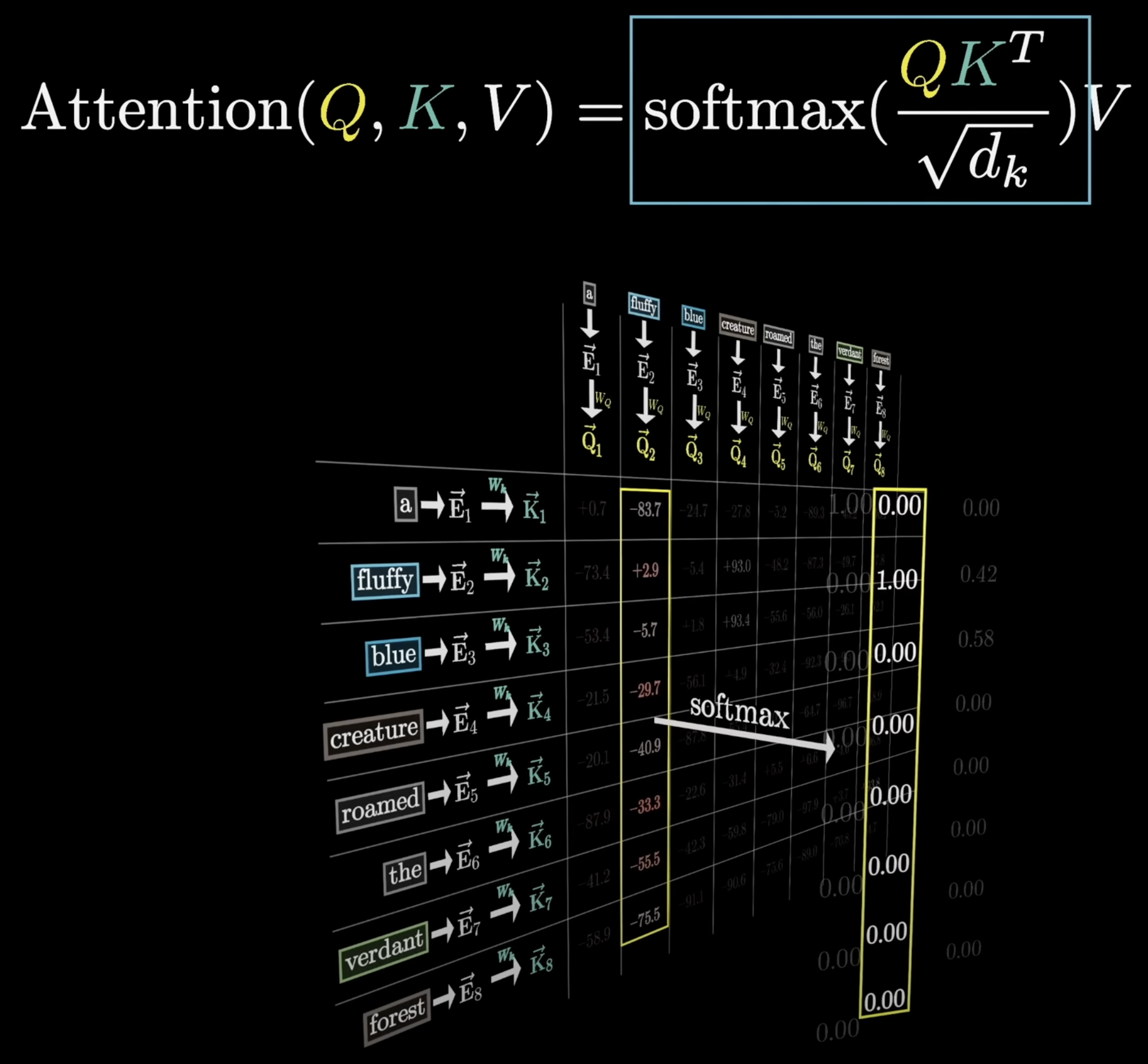
内積、スケーリング、Softmax正規化を経て、各ヘッドごとに重要な情報を抽出する。マスクを適用した上で各要素間の関連性(重み)を計算する。以下の計算式で表される。

　また、



(画像引用：左から原著論文、参考文献1, 参考文献2)

1. **内積による類似度計算**
   * Qと、Kの転置行列の内積を取ることで、類似度を計算する。
2. **スケーリング**
   * は、分散がkey\_dimになるため、次元が増えれば増えるほど、分散が大きくなってしまい、勾配消失が起こりやすくなる。
   * で割ることで、分散が1になるよう標準化を行う。
3. **ソフトマックスによる正規化**
   * softmax関数で、内の各成分を、0.0〜1.0の範囲に変換するこの関数によってベクトルの各成分の合計は常に1.0になる。
   * これは、各位置がどれだけ重要かを示す確率分布である。



(画像引用：参考文献2)

1. **重み付き和による情報の集約**
   * 確率分布を使って、値Vの各要素に「どれだけ注意を向けるか」の重みをかけ、その重み付き和を取る。
   * これは、各クエリに対して「シーケンス内のどの情報をどの程度取り入れるか」を決定するための仕組みである。
   * 具体的には、重みが大きい位置のの情報がより多く出力に反映され、重みが小さい位置の情報は抑えられる。結果として、各クエリに対する出力は、シーケンス全体のの情報の「要約」として得られる。

　以上の流れを、以下の式に示す通り分割した各ヘッドごとに行う。ヘッドはnum\_headsで指定された数ある。

### **ヘッドの統合**

各ヘッドの出力を連結し、重み行列をかける。TensorFlowではDense層を適用して、key\_dimから元のd\_modelに戻す。

### **最終出力**

* + 最終出力の形状はmax\_length × d\_model、すなわち、長さmax\_lengthで特徴量d\_model次元であり、元の入力xと同じ形状に戻る。
  + 入力と出力が同じ形状であるので、特に追加の処理を必要とせずに、Encoderブロック1の出力をEncoderブロック2の入力にできる。筆者のコードではnum\_transformer\_blocksがEncoderブロックの個数となっており、ブロックを積み上げるように繰り返し計算ができる。原著論文では6層。

　以下の表に、入力xがMulti Head Attentionレイヤーに入力されてから出力が生まれるまでどのように形状が変化しているのかを、上記のフェーズごとに対応させてまとめた。

| **MultiHeadAttention内の各ステップ** | **各ステップ時の行列の形状** |
| --- | --- |
| 0. 入力 | (batch\_size, max\_length, d\_model​) |
| 1. Q, K, V生成後(Q, K, Vで同一) | (batch\_size, max\_length, num\_heads × key\_dim) |
| 2.ヘッド分割後 | (batch\_size, num\_heads, max\_length, key\_dim) |
| 3. Attention計算後 | (batch\_size, num\_heads, max\_length, key\_dim) |
| 4. ヘッド統合後 | (batch\_size, max\_length, num\_heads × key\_dim) |
| 5. 最終出力 | (batch\_size, max\_length, d\_model​) |

原著論文では、d\_model(特徴量の次元数512)をnum\_heads(アテンションヘッドの数8)で割った数がkey\_dim(アテンションヘッド内の次元数64)として、行列の形状の厳密な定義がなされていた。

　TensorFlowでは線形変換がDense層によってなされているので、key\_dimがd\_model/num\_headsにならなくても良いし、極論どんな値でも良い感じに線形変換してくれているので、データ量やデータの次元、PCのスペック、エポックごとの学習曲線に応じて好きな値に設定すれば良い。

# **疑問点と回答**

　Transformerを学ぶにあたって疑問に感じた点や、読者が本稿やサンプルコードを読んで考えるであろう疑問点と、その回答をまとめた。

* **なぜ位置エンコーディングにsin波とcos波を用いるのか?**
  + 単に入力順序そのものの値を加えると値が大きくなってしまうため。
  + sinやcosは入力位置に応じて滑らかに値が変化し、–1から1に制限されるため、数値が大きくなりすぎずに順序情報を表現できる。
* **位置エンコーディングの計算に出てくる10000という値の意味は？**
  + 周期関数であるsinやcosは位置が大きくなると値が繰り返されるため、十分な長さのシーケンスでも各位置が一意に表現されるように、低い周波数を与える必要があるから。
  + 10000はその周波数を決定するハイパーパラメータで、別に10000じゃなくて良い。論文では10000だったというだけ。センサデータであることを考えたらもっと小さい値にしてもいいかも。
* **位置エンコーディングにsin波とcos波の両方を使うのはなぜなのか？**
  + 低周波数にすると、特に位置が小さいときに値の変化が少なくなる問題があり、表現の幅を広げるため。
* **パディングの値はなんでも良いのか？**
  + なんでもいい。例えばパディングを-999で行ったとして、モデル構築時にちゃんと引数にその値を渡してあげればマスクが作成されるようにコードを書いた。(はず...)
* **なぜbuild関数の方でもMaskingするのか?**
  + Masking層は入力データ中のパディング部分を無視するために使われ、後続の層が無意味なパディング情報を処理しないようにする。
  + しかし、Attention層はMasking情報を自動で利用しないため、別途Attention計算時にマスクを渡す必要がある。
* **なぜ類似度計算でコサイン類似度ではなく内積を使うのか?**
  + 詳しくは数学の話になるが、コサイン類似度はあくまでベクトルの方向のみを考慮している。しかし、Attention計算ではベクトルの大きさの情報も大事。よって内積をとる。
* **なぜ残差接続は足し算なのか?**
  + 各層での変換結果f(x)と元の入力xを単純に足し合わせることで、もしf(x)がゼロに近い場合でもx自体がそのまま次の層へ伝わるようにする仕組みである。
  + この足し算により、もし学習がうまくいかなかった層でも、元の情報が失われずに残り、勾配が安定して流れるという利点がある。
  + 掛け算を用いると、xの各成分がf(x)と乗算されるため、f(x)の値が0に近い場合、全体が極端に縮小されてしまい、元の情報が失われるリスクがある。
  + また、掛け算は非線形な影響を大きく与えるため、残差接続が本来意図する「変化量だけを学習させる」という効果が得られにくくなる。
* **Self-Attentionじゃないときはどんな時?**
  + 原著論文ではTransformerはEncoder-Decoder構造をとっており、Encoder部分のMulti Head Attentionが1つ、Decoderで2つの計3つ出てくる。Encoder部分の出力と、Decoder部分の1つ目のMulti Head Attentionの出力が、Decoder部のMulti Head Attentionの入力になる。
  + 村尾研で扱うデータではこのような構造はオーバースペックな上、[BERT](https://aclanthology.org/N19-1423.pdf)のようにTransformerのEncoder部分のみを用いたものもメジャーであることから、本稿ではEncoder部分のみに絞って解説を行った。(筆者の研究だけでなく村尾研内でTransformerを用いた他の研究においてもEncoder部分のみの実装だった)

# **参考資料**

　Transformerについて理解を深める一助として、筆者が参考にした資料やテックブログとそのリンクを厳選してまとめた。これ以外にもネット上には多数の解説記事や動画があるため、参考にされたし。

* **原著論文**  
  [Attention is All you Need](https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)
* **参考文献1**(筆者が読んだ中で最も丁寧かつわかりやすい解説)

[Multi-Head AttentionとScaled Dot-Product Attentionの全て：Transformerの核心を徹底解説 - nomulog](https://www.nomuyu.com/multi-head-attention/)

* **参考文献2(**Attention計算がビジュアライズされており、イメージを掴みやすい)

[GPTとは何か Transformerの視覚化 | Chapter 5, Deep Learning](https://youtu.be/KlZ-QmPteqM?si=U2ShN5nkNJ4_Xjze)

[GPT解説2 アテンションの仕組み (Attention, Transformer) | Chapter6, 深層学習](https://youtu.be/j3_VgCt18fA?si=-sr95yATWbkz5v41)

* **TensorFlowのMultiHeadAttentionの公式ドキュメント**

[tf.keras.layers.MultiHeadAttention | TensorFlow v2.16.1](https://www.tensorflow.org/api_docs/python/tf/keras/layers/MultiHeadAttention)

[Neural machine translation with a Transformer and Keras | Text | TensorFlow](https://www.tensorflow.org/text/tutorials/transformer)

* **その他**

[The Illustrated Transformer – Jay Alammar](https://jalammar.github.io/illustrated-transformer/)[**Attention is all you need (Transformer) - Model explanation (including math), Inference and Training**](https://www.youtube.com/watch?v=bCz4OMemCcA&t=1208s)

[ChatGPTのコア技術であるTransformerを高校数学の基礎から最短で理解したい #LLM - Qiita](https://qiita.com/miyakz/items/7f8f49f979bd530e2717)

[これならわかるSelf-Attention](https://zenn.dev/knowhere_imai/articles/1008252132d315)

[【論文解説】Transformerを理解する](https://data-analytics.fun/2020/04/01/understanding-transformer/)

[【Transformerの基礎】Multi-Head Attentionの仕組み | AGIRobots Blog](https://developers.agirobots.com/jp/multi-head-attention/)

[マルチヘッドアテンション (Multi-head Attention) [Transformerの部品] | CVMLエキスパートガイド](https://cvml-expertguide.net/terms/dl/seq2seq-translation/transformer/multi-head-attention/)

[【深層学習】Attention - 全領域に応用され最高精度を叩き出す注意機構の仕組み【ディープラーニングの世界 vol. 24】#095 #VRアカデミア #DeepLearning](https://youtu.be/bPdyuIebXWM?si=_ljli29_C28tvNo3)

[【深層学習】Transformer - Multi-Head Attentionを理解してやろうじゃないの【ディープラーニングの世界vol.28】#106 #VRアカデミア #DeepLearning](https://youtu.be/50XvMaWhiTY?si=gukgLJ8aN8dlpxvz)  
位置エンコーディングについての解説→[Positional Encodingを理解したい - Qiita](https://qiita.com/snsk871/items/93aba7ad74cace4abc62)

MacでのTensorFlow環境構築について↓  
[Tensorflow Plugin - Metal - Apple Developer](https://developer.apple.com/metal/tensorflow-plugin/)  
[Apple SiliconのMacでGPUをTensorflowで用いる方法 - Qiita](https://qiita.com/convolm/items/58c7a4b55313e548ad4f)  
[Mac M1/M2にGPUサポート付きのTensorFlowをインストール](https://zenn.dev/datasciencekun/articles/74b764515ca246)  
[Apple Silicon Mac向けGPU利用のTensorFlow, PyTorch設定](https://trinitas.tech/2024/08/21/apple-silicon-mac%E5%90%91%E3%81%91gpu%E5%88%A9%E7%94%A8%E3%81%AEtensorflow-pytorch%E8%A8%AD%E5%AE%9A/)

# **付録**

　本稿には[サンプルコードのモデル図](#kix.9p14puqenkvb)を添付したが、これの元ファイルのパワポをドライブにアップロードしている。→[サンプルコードのTransformerのモデル図(論文用).pptx](https://docs.google.com/presentation/d/1q9Tb-kdVbDCv6zIfEcu0KiSqVb7LNFty/edit?usp=share_link&ouid=104072017877855438131&rtpof=true&sd=true)  
読者らが将来自分の研究においてTransformerを用い、論文に図を載せる際はこれをダウンロードした上で自由に改良してもらえれば良い。(実際に筆者はISWC2023に投稿した論文にこの図をベースしたモデル図を記載したが、査読レビューでは特に突っ込まれなかったので大丈夫だと思う。多分....)