The Annotated Transformer Part1

pythonの環境構築(仮想環境推奨)

"venv"という名前の仮想環境を作成 #pythonのバージョンは3.9にしてください。

```
py -3.9 -m venv venv
```

仮想環境のアクティベート

- Linux, Mac

```
source ./venv/bin/activate
```

- Windows

```
source ./venv/Scripts/activate
```

必要なmoduleのインストール #※めっちゃ時間かかります

```
pip install -r requirements.txt
```

仮想環境のディアクティベート

```
deactivate
```

内容目標

- Transformerがどのようなアーキテクチャで構成されているのかを理解する。
- 機械学習、特にTransformerの仕組みをふわっと理解する(詳しいアルゴリズムの解説はしません)。
- Transformerがどのような処理を行っているのかの概要を理解する。

Transformer

Transformer とは、系列情報を処理する、注意機構(Attention)を主体とした深層学習モデルです。
提案された当初は、英語→ドイツ語の翻訳といった自然言語処理の系列変換タスクでのみ注目されていましたが、今日ではChatGPTをはじめとした様々なタスクやモダリティに応用され、その有用性が広く認識されています。

"The Annotated Transformer"[1]とは?

上記で述べたとおり、 Transformer の最も重要な構成要素は"Attention"です。
その"Attention"の仕組みを論文"Attention is All You Need"[2]に基づいて、コードを交えて解説している良い資料として、"The Annotated Transformer"あります。
今回の勉強会では、この資料を基にpythonでコードを書きながら、Transformerについての理解を深めていくことを目的としています。
"The Annotated Transformer"は三部構成となっており、今回はPart1のモデルアーキテクチャの部分をやっていきます。
本日の予定としては、まず Transformer の背景について簡単に説明した後、エンコーダ・デコーダの実装、Attention機構の実装・・・と進めていこうと考えています

背景

Transformerの最大の特徴は、系列変換の並列計算の高速化です。
少し難しい言い方をすると、畳み込みニューラルネットワーク（CNN）を基本構成要素として使用し、入力と出力のすべての位置に対して隠れた表現を並列に計算します。これにより、 Extended Neural GPU, ByteNet, ConvS2S の基盤を成しています。
Attentionの構成要素には、Self Attention, Multi Head Attention 等があります。"Self Attention"についてはエンコーダのセクションで、"Multi Head Attention"についてはデコーダのセクションで詳述します。

Part1: モデル構造 Model Architecture

Encoder-Decoder Architecture

最も強力な(2023年では)自然言語処理や音声処理などで使われるモデル構造に、"エンコーダ・デコーダ構造"[3]があります。Transformerでも、このモデル構造が使われています。

ここで、"エンコーダ"とは、入力シンボル表現のシーケンス （x1，...，xn） を連続表現のシーケンス z = （z1，...，zn） にマッピングするものです。通常、入力されるシンボルは離散的なもので、単語や文字などが例として挙げられます。
マッピングされた連続表現のシーケンス z は、ニューラルネットワークの隠れ層での計算に使われます。通常のシンボルの状態だと、ニューラルネットの計算に使いにくいので、計算しやすいベクトル表現にマッピングするみたいなイメージです。

"デコーダ"ではzが与えられると、シンボルの出力シーケンス（y1，...，ym）を1要素ずつ生成します。デコーダの出力が、最終的な出力になります。

要するに、エンコーダは入力を「解釈」し、デコーダはその解釈を基に出力を「生成」する役割を果たします。

Transformerの全体図は、The Annoted Transformer のPart1を参照してください。

program: モデル構造のクラス

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """

    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)

class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return log_softmax(self.proj(x), dim=-1)
```

エンコーダ

Transformerで使われるエンコーダの重要な構成要素の一つに、"Self Attention"があります。
Self Attention という言葉を聞いたことはありますか？
Self Attention では、シーケンス内の各要素の依存関係を捉える事によって、推論の精度の劇的な向上や並列計算を可能にしています。
シーケンスの処理を左から右に線形に行うと、シーケンス内の各要素の依存関係を捉えることができません。しかし、シーケンス内の各トークン同士の相互作用を並列で計算することで、各要素の依存関係を捉える事を実現しています。

エンコーダの構成要素

エンコーダには、２つのサブレイヤーがあり、"Self Attention"と"Feedforward Network"と呼ばれています。
"Self Attention"によって、シーケンス内の依存関係が計算されます。
その後、"Feedforward Network"によって各トークンの表現が変換されます。
つまり、入力されたシンボルを"Self Attention"↦"Feedforward Network"に通すことによって、上記の連続表現のシーケンスzが得られます

多層エンコーダ

深層ニューラルネットワークの特徴として、層を増やせば増やすほど性能が上がるというのは皆さんご存じかと思います。
それと同様に、Transformerでは、エンコーダの多層化(深層化)を行っています。
以下では、その"多層エンコーダ"をプログラムで実装していきます。
実装方針は、"大きい実装から小さい実装を導いていく"という方針です。
具体的には、まずはエンコーダのクラスを作り、各層のエンコーダを作り、２つのサブレイヤー"Self Attention"と"Feedforward Network"を作ります。次にそれらを合成するクラスを作り、多層エンコーダの実装を行います。

program: 多層エンコーダ

ここでは、多層エンコーダの層の数を6とする。理由は知りません。
[補足]
ChatGPT4o miniに聞いたところ、以下の返答が得られました。

> Transformerモデルにおいて、エンコーダとデコーダの層を6に設定しているのは、やや任意の選択であり、
> 実際のところ性能と計算効率のバランスを取るために実験的に決められたものです。この6層という設定は、
> 元のTransformer論文において、十分な複雑な依存関係を学習しつつも、
> 計算資源に対する負荷を適切に管理できる範囲で効果的であることが分かったため採用されました。
>
> ただし、この層の数は固定ではなく、タスクや利用可能な計算リソースに応じて調整可能です。
> 6層以外にも、層を増減させることでパフォーマンスを向上させたり、
> 計算量を削減したりすることができます。

ChatGPT o3に聞いたところ、以下の返答が得られました。

> Ａ経緯概
> Transformer を提案した Vaswani らの元論文では，エンコーダ／デコーダともに N = 6 層を
> 「ベースモデル」の既定値として採用した．これは再帰や畳み込みを持つ従来モデル
> （8 〜 10 層程度が多かった）の計算量と GPU メモリ制約を踏まえ，実用的な訓練時間と
> 性能のバランスが取れる深さとして経験的に選ばれたハイパーパラメータである．
> <cite>NeurIPS Papers
>
> Ｂ実験根
> 同論文の Table 3（行 (C)）では層数 2･4･6･8 を比較したアブレーションが示されている．
> 2 層：BLEU 23.7
> 4 層：BLEU 25.3
> 6 層：BLEU 25.8（基準）

8 層：BLEU 25.5
→6 層から 8 層へ深くしても BLEU 改善はほぼなく，学習ステップ数・パラメータ数（65 M→80 M）
が増えるだけだったため，**「6 層が効果と計算コストの折り合いが最も良い」**という結論に落ち着いた．
<cite>NeurIPS Papers

C 計算費
2017 年当時は P100×8 枚で 12 時間～数日という訓練枠が現実的上限だった．6 層構成なら
3 億 FLOPs 台に収まり，既存 Seq2Seq を凌ぐ BLEU を達成しつつ，より深いモデルより
40 %以上短い学習時間で済むことが社内実験で確認されている．
<cite>NeurIPS Papers

D 発展形
以後の研究では、
・幅（d_model や h）を増す（Transformer-big）
・深さをさらに積む（BERT-base 12 層・GPT-3 96 層など）
といった拡張が行われたが，これはハードウェアと最適化技術の進歩によって
「6 層の制約」が緩和されたためであり，理論的に 6 層でなければならないわけではない．

要約

6 層は「性能が飽和する最小深さで計算資源を節約できる」ことを示す実験結果に基づく
経験的なハイパーパラメータであり，現在は目的・資源に応じて自由に調整される。

恐らく、6層が実際にやってみて丁度良かったからだと思います。

```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])

class Encoder(nn.Module):
    "Core encoder is a stack of N layers"

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):#おそらく、左から順番にシーケンスを処理する部分
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

design of multilayer encoder

エンコーダを単層で実装する場合は、"Self Attention"↦"Feedforward Network"を行うだけで良い。
しかし、多層化することによって、情報は左から右に順番に伝播されていく。
それにより、深層ニューラルネットワークのように、勾配消失や過学習などの問題が出現する。
それらの問題を解決するために、ただ単に伝播させるのではなく、残差接続(Residual Connection)[4]と呼ば

れる手法を用いて加工することで、深いネットワークでの勾配消失を防ぎ、情報が層を越えて伝播しやすくする。
また、残差接続をした後にレイヤー正規化(Layer Normalization)[5]という操作が行われる。
レイヤー正規化は、各層の出力を正規化し、学習を安定させる役割を果たす。これにより、勾配の変動を抑え、学習がスムーズに行われるようになる。

上記の操作を、LayerNormと呼ぶ(多分)
↑は嘘でした。サブレイヤーの処理後、Self Attention と Feedforward Networkの処理後にサブレイヤーをconnectするときに呼ばれるのが、LayerNormでした。

program: LayerNorm

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

残差接続の計算

上記では、情報を右から左へ伝播させるときに、ただ伝播させるのではなく、残差接続(Residual Connection)[4]と呼ばれる手法を用いて加工することによって、深いネットワークでの勾配消失を防ぎ、情報が層を越えて伝播しやすくなる。と書きました。
さらに、別の研究ですが、過学習を防ぐために、ドロップアウト[6]という手法も提案されています。
これは、学習時にニューロンをランダムに無効化することによって、モデルの汎化能力を高めます。

program: add dropout

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

program: complete multilayer encoder

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"

    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

エンコーダの動きの順序

1. 入力 (x, mask) がエンコーダに渡される。
2. エンコーダのforward関数 が呼ばれ、各エンコーダ層（EncoderLayer）が順番に処理される。
3. 各エンコーダ層では、Self Attention と Feedforward Network の2つのサブレイヤーが順番に処理される。
4. 各サブレイヤーは、残差接続 と レイヤー正規化 が適用される。
5. 最終的にエンコーダの出力が正規化(LayerNorm)され、結果が返される。

エンコーダまとめ

これにてエンコーダの実装が完了した。

デコーダ

上記で述べた通り、デコーダは、エンコーダからのシーケンスを受け取って、最終的な結果を返す役割を持っています。
エンコーダ同様、デコーダも多層化することにより、性能向上が図られています。

program: 多層デコーダ

上記のエンコーダ同様、多層デコーダの層を 6 とする。

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."

    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

デコーダのサブレイヤ

エンコーダは"Self Attention"と"Feed-Forward Network"の二つのサブレイヤに分かれているという話は覚えていますか？
デコーダもサブレイヤが複数あるのですが、エンコーダのサブレイヤ"Self Attention"と"Feed-Forward Network"に加えて、"Multi Head Attention(Encoder-Decoder Attention（エンコーダ出力に対する多頭注意）)"の三つのサブレイヤに分かれています。
"Self Attention"と"Feed-Forward Network"の役割はエンコーダと同じですが、"Multi Head Attention"では、エンコーダが作った文脈ベクトル全体を各デコーダ位置から閲覧させ，入力系列との整合を取るという役割があります。
ただし、各サブレイヤの前後に残差接続＋レイヤ正規化が行われる点はエンコーダと一緒です。

program: 各層のデコーダの実装

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"

    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)
```

```
    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

subsequence mask

```
subsequence maskを用いることで、デコーダースタックの自己注意サブレイヤーを修正し、
位置が後続の位置に注意を向けることを防止します。
このマスク処理と、出力埋め込みが1位置分オフセットされているという事実を組み合わせることで、
位置iの予測は位置i未満の既知の出力をのみに依存するように保証されます。
```

と、小難しい説明が書いてありますが、subsequence maskは、文字の通りマスクです。未来のトークンの参照を防ぐという役割を持つマスクです。
何故未来のトークンを隠さないといけないかというと、未来のトークンには、現在訓練中のデータがあり、それを訓練に使ってしまうと、訓練時にだけ存在する手掛かりに依存した重みが形成されてしまい、実用時に性能が急落してしまうからです。
下記のshow_example(example_mask)関数を実行すると、ヒートマップが表示されますが、これはsubsequence maskを視覚的に理解し易いようにする図です。つまり訓練には関係のない図です。
縦軸がクエリ位置で、横軸がキー位置、色は明るいところほどトークンがよく見え、暗いところほど見えないというsubsequence maskの働きを分かりやすいように描画してくれています。

program:

```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = torch.triu(torch.ones(attn_shape), diagonal=1).type(
        torch.uint8
    )
    return subsequent_mask == 0

def example_mask():
    LS_data = pd.concat(
        [
            pd.DataFrame(
                {
                    "Subsequent Mask": subsequent_mask(20)[0][x, y].flatten(),
                    "Window": y,
                    "Masking": x,
                }
            )
            for y in range(20)
            for x in range(20)
        ]
```

```
    )

    return (
        alt.Chart(LS_data)
        .mark_rect()
        .properties(height=250, width=250)
        .encode(
            alt.X("Window:O"),
            alt.Y("Masking:O"),
            alt.Color("Subsequent Mask:Q", scale=alt.Scale(scheme="viridis")),
        )
        .interactive()
    )


show_example(example_mask)
```

Attention

上記では、Transformerを構造的な概念である、エンコーダ・デコーダ構造のレベルで説明と実装を行いました。ざっくり言うと、"大雑把にTransformerとはこういう構成になってるよ"という紹介をしました。
ここからは、エンコーダとデコーダ構造の中身、Transformerの中心演算を担っている"Attention"について、pythonによる実装を通して理解を深めていきます。

Attention 概要

Attention関数は、Query(Q)とKey(K)とValue(V)の集合を出力にマッピングする関数として定義できます。
先ほど出力したsubsequence maskを見ると分かりやすいかもしれませんが、(Q, K)によってVが決まります。
数式で表すと、以下になります。

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

ここで、クエリ、キー、値、および出力はすべてベクトルです。
また、 dk はキー（とクエリ）のベクトル長、すなわち各ヘッドで Q・K が持つ次元数を表します。
Transformer-base なら dk = d_model / h（例：512/8 = 64）です。

出力は、各値に割り当てられた重みによる値の加重和として計算されます。この重みは、
クエリと対応するキーの互換性関数によって計算されます。
私たちは、この特定の注意機能を「スケーラブル・ドットプロダクト・アテンション」と呼びます。
入力は、次元dkのクエリとキー、および次元dvの値から構成されます。クエリとすべてのキーの
ドット積を計算し、それぞれをdkで除算し、ソフトマックス関数を適用して値の重みを取得します。

program: Attention

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Attention機構の構成要素

Attention機構では、①スコア関数 ②ヘッド構成 ③接続形態が核を成しています。
①のスコア関数は文字通り、スコアの計算を行う関数です。どのようにスコアを計算するのかを決めます。
②のヘッド構成は、何層で Attention の計算を行うかを表しており、上記の6層エンコーダ・デコーダでは6になる・・・と思っていたのですが、誤りでした。
正しくは、

```
ヘッド (head) は 1 つの注意層の内部 をいくつに“並列分割”して計算するかを示すパラメータ。
例: “8 heads” なら 同じ層の中で 8 本の Scaled Dot Product（or additive など）Attentionを同時に計算し，結果を結合して次の層に送る。
```

で、Attention全体の話ではなく、個々の話でした。[8][9]
前述した層 (layer) は こうした "self Attention" + "Feed Forward Network" を縦に積む深さを指します。
Transformer-base だとエンコーダ 6 層・デコーダ 6 層が典型例。ヘッド数を増やしても「層が増える」わけではないそうです。
③の接続形態は上記のQ, K, V を「どの系列・モダリティから取るか」を定義する概念。
これまでも何度か出てきた Self Attention は、同一系列内からしか参照しないという接続形態です。
Self Attentionの他にも、"Cross Attention や Casual Attention"などがある。 この三つが Attention 機構の核をなしていますが、補助要素として、上記でも出てきた

```
区分 | 主な目的 | 代表例
正規化 | 勾配爆発・消失を抑え学習安定 | LayerNorm
残差接続 | 深層化による情報劣化を回避 | Add & Norm ブロック
マスク | 因果律保持・パディング除去 | サブシーケンスマスク，パディングマスク
位置符号化 | 系列順序を注入 | Sinusoidal，Learnable PE
ドロップアウト | 汎化性能向上 | Attention Dropout，FFN Dropout
スケーリング | 大次元での数値安定化 | 1/dk1/\sqrt{d_k}1/dk 係数
最適化技巧 | 収束促進・性能向上 | 学習率ウォームアップ，重み減衰 等
```

があり、これらを調整することで、実際に使えるレベルまで引き上げています。

①スコア関数

まずは、①のスコア関数から説明していきたいと思います。
上記の Attention 関数のコメントに、"スケーリングされた Dot Product Attention を計算する"と書いてありました。
Transformer では、スコア関数として Scaled Dot Production を用いるのが主流のようです。
そもそも、有名なスコア関数に、"Additive Attention"[3] と "Dot Product Attention"の二つがあります。
超簡単な説明としては、"Additive Attention"は性能はいいがコストが高く[7]、"Dot Product Attention"は性能はそこそこだがコストが低いというトレードオフの関係があります。
Attention を単層で用いる場合は"Additive Attention"を用いることが多いようですが、 Transformer のような多層構造で Attention を用いる場合はコストの観点から、"Dot Product Attention"が使われるのが主流のようです。

備考: なぜスケールをする必要があるのか

上記の"Dot Product Attention"は、スケーリングをすると書かれていました。
超簡単にいうと、"Additive Attention"の性能に近づけるためにスケーリングを行うのですが、詳しくは参考文献[7]をご覧ください。
簡単に説明すると、

> dkの値が小さい場合、両メカニズムの性能は類似していますが、dkの値が大きくなるにつれ、
> 加法注意はスケーリングなしでドット積注意を上回ります。
> dkの値が大きい場合、ドット積の絶対値が非常に大きくなり、ソフトマックス関数が極端に小さな
> 勾配を持つ領域に押し込まれるためだと推測されます。
> この効果を相殺するため、内積を`1/√dk`でスケーリングします。

だそうです。
なぜ√dkで割るのかですが、

- 内積のスケール問題
  各要素が平均 0、分散 1 のとき、
  Var(q * k) = dkとなり次元が増えるほど値が大きくなる[cite]。
- softmax の飽和回避
  大きすぎるスコアは softmax を極端に尖らせ、勾配がほぼ 0 になり学習が鈍化[[10]]
  (https://glassboxmedicine.com/2019/08/15/the-transformer-attention-is-all-you-need/?utm_source=chatgpt.com)。
- √dkで標準化
  分散 ≈ 1に正規化され、softmax が適度な温度で働く。
  要するにdkは「キー・クエリ空間の次元」を示し、その平方根で割るのは 数値安定性と学習効率を保つための温度スケーリング だそうです。

②ヘッド構成

次に、②のヘッド構成について説明していきます。
上記のデコーダの所では、"Multi Head Attention"の話をしました。
ヘッド構成とは、上述の通り"1 つの注意層の内部 をいくつに“並列分割”して計算するかを示すパラメータ。"でした。

参考文献

[1] Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman(2022). The Annotated Transformer. https://nlp.seas.harvard.edu/annotated-transformer/
[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser(2017). Attention Is All You Need. Advances in Neural Information Processing Systems 30 (NIPS 2017)
[3] Dzmitry Bahdanau, KyungHyun Cho(2014), Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. https://arxiv.org/abs/1409.0473
[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. https://arxiv.org/abs/1512.03385
[5] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton. Layer Normalization. https://arxiv.org/abs/1607.06450
[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. https://jmlr.org/papers/v15/srivastava14a.html [7] Denny Britz, Anna Goldie, Minh-Thang Luong, Quoc Le. Massive Exploration of Neural Machine Translation Architectures. https://arxiv.org/abs/1703.03906 [8] Sebastian Raschka. Understanding and Coding Self-Attention, Multi-Head Attention, Causal-Attention, and Cross-Attention in LLMs. https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention?utm_source=chatgpt.com [9] Peng Jin, Bo Zhu, Li Yuan, Shuicheng Yan. MoH: Multi-Head Attention as Mixture-of-Head Attention. https://arxiv.org/abs/2410.11842 [10] Rachel Draelos. The Transformer: Attention Is All You Need. https://glassboxmedicine.com/2019/08/15/the-transformer-attention-is-all-you-need/?utm_source=chatgpt.com

[11] 森下篤(2024). Visual Studio Code 実践ガイド. 技術評論社
[12] Bill Ludanovic, 鈴木駿, 長尾高弘(2022). 入門 Python3 第二版. O'Reilly Japan
[13] Al Sweigart, 相川愛三(2023). 退屈なことはPythonにやらせよう 第二版. O'Reilly Japan
[14] Al Sweigart, 岡田祐一(2022). きれいなPythonプログラミング. マイナビ

This material benefited from the assistance of ChatGPT.

Kazuma Aoyama(kazuma-a@lsnl.jp), Yoshiteru Taira(yoshiteru@lsnl.jp)