機械の言語能力の獲得を考える

I propose to consider the question, "Can machines think?"

....

The original question, "Can machines think?" I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

-- Alan Turing

I propose to consider the question, "Can machines think?"

....

The original question, "Can machines think?" I believe to be too meaningless to deserve discussion.

Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

-- Alan Turing

こうした反省は、「知的機械」. の限界・無能力として意識さるべきものが、実は正確に、我々自身の限界・到達点をさししめしている事を我々に告げるのである。なぜなら、機械の進化にとって、我々は万能の創造主に他ならない。

我々は、我々自身に似せて知的な機械を造ろうとしたが、我々が直面した壁は、我々があまりに自分自身を知らず、加えて、多くの問題に対して、あまりに貧弱な洞察力しか持っていないという事であった。

しかし、次の事を忘れてはならない。我々が、我々自身の創造力の全面的な開花の状態からへだたっているのなら、そのへだたりが、我々が、その進化をつかさどる機械の創造性を基本的に限界づけるのだと。

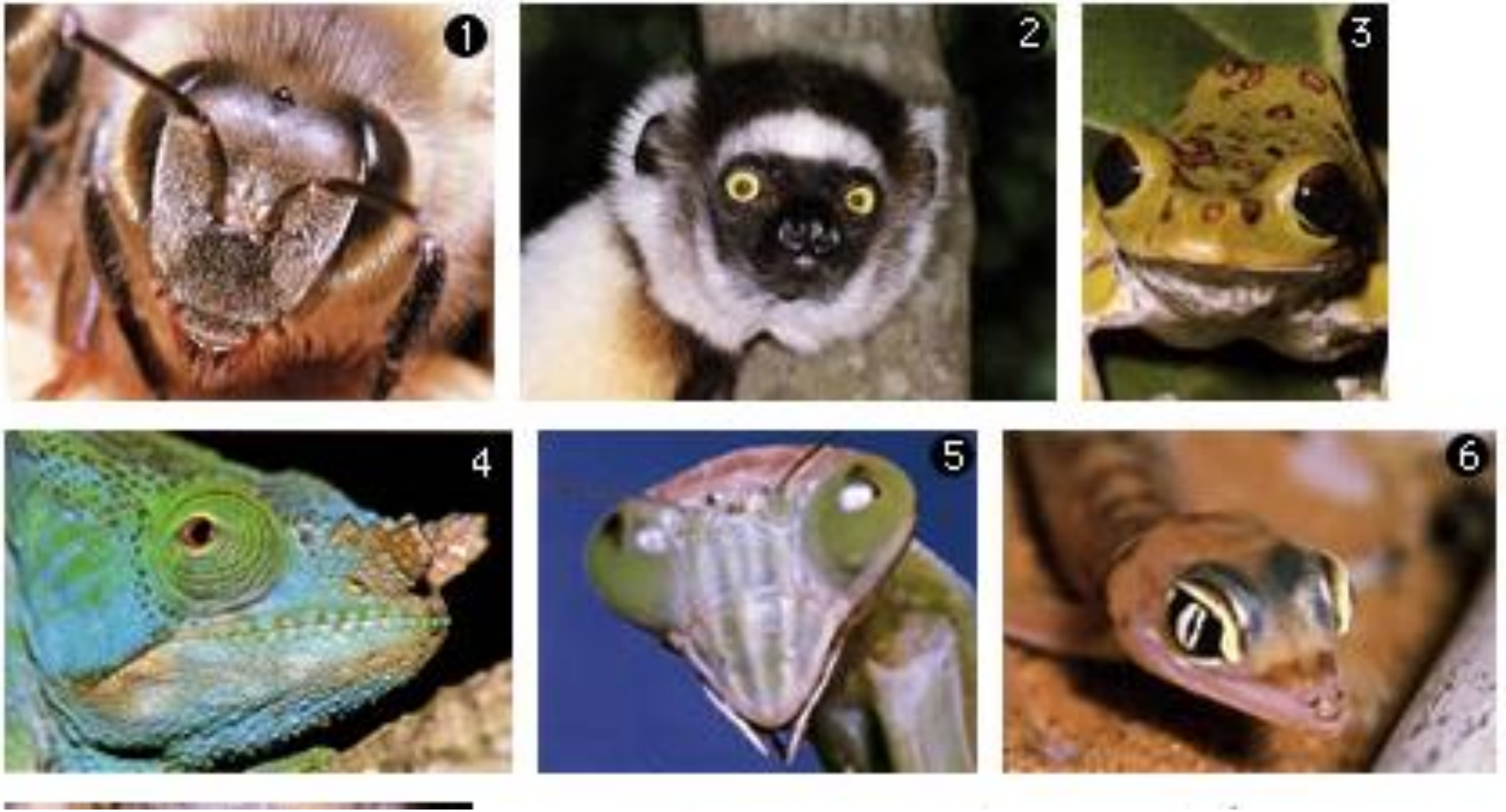
我々は、我々自身の前史を、とおり抜けねばならない。

--- 詠み人しらず

今回のセミナーの問題意識


Bolt

多くの生物は目を持っている



ゲーリングによる目の遺伝子Pax-6の発見

https://www.brh.co.jp/seimeishi/journal/012/ss_1.html

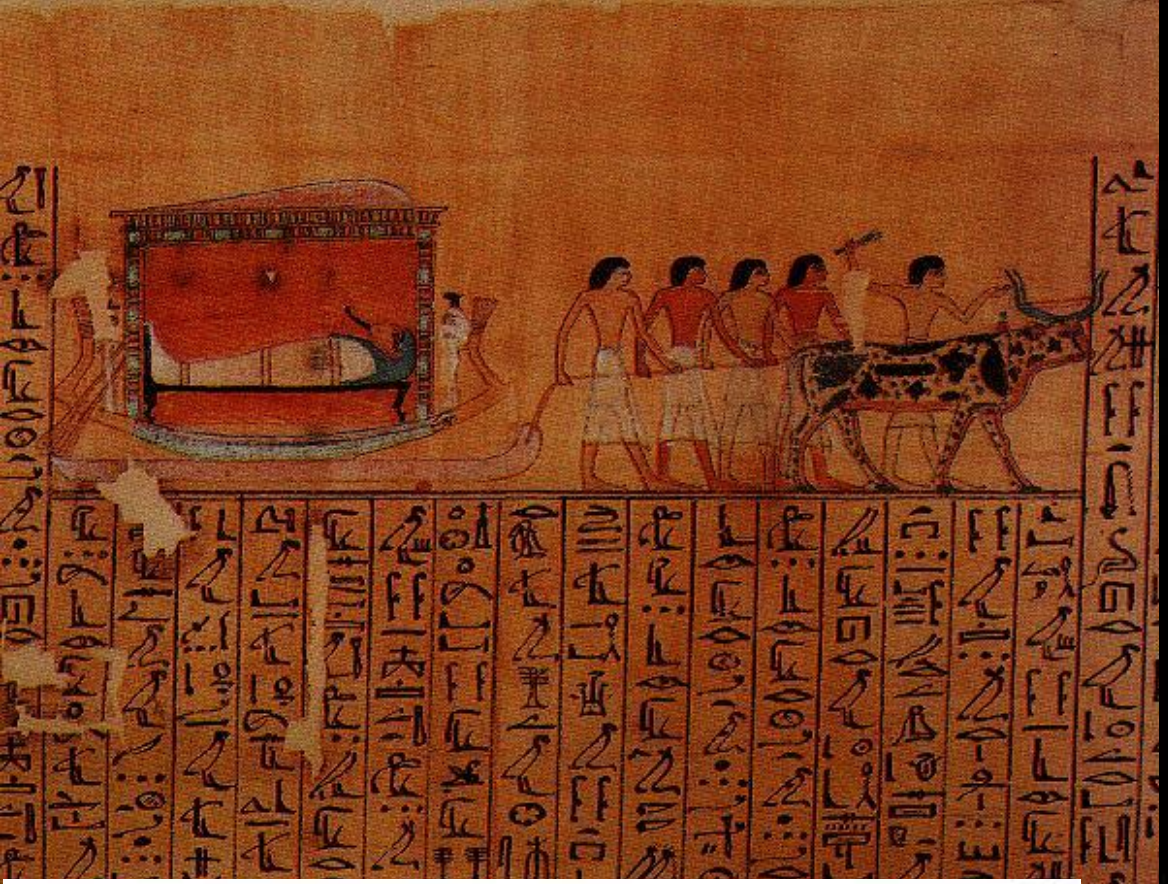
多くの生物はコミュニケーション能力を持つ

親と子も恋人同士の二人も老人ホームの老人もことばを使います。
SNSで罵倒し合うのにも、戦争を呼びかけるのにも戦争に反対するのにもことばが必要です。

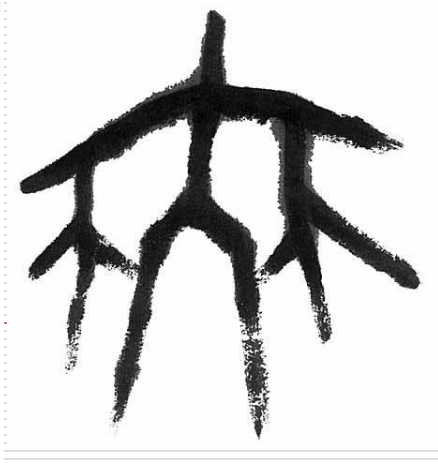
捏造された論文もノーベル賞の対象となる論文も、ことばで書かれています。

これらすべては、人間がひとしく言語能力を持ってコミュニケーションできるから可能になっていることです。

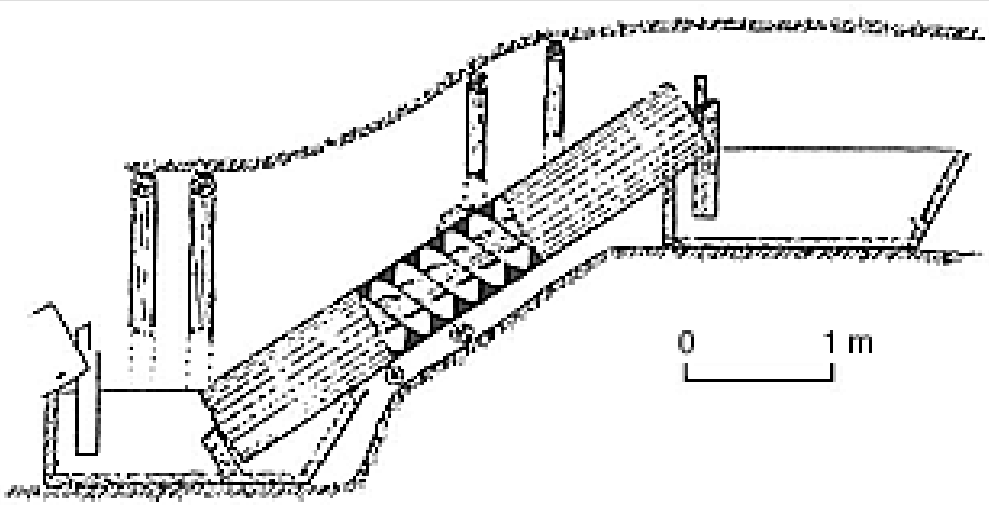

人間の言語能力は生得的なものである



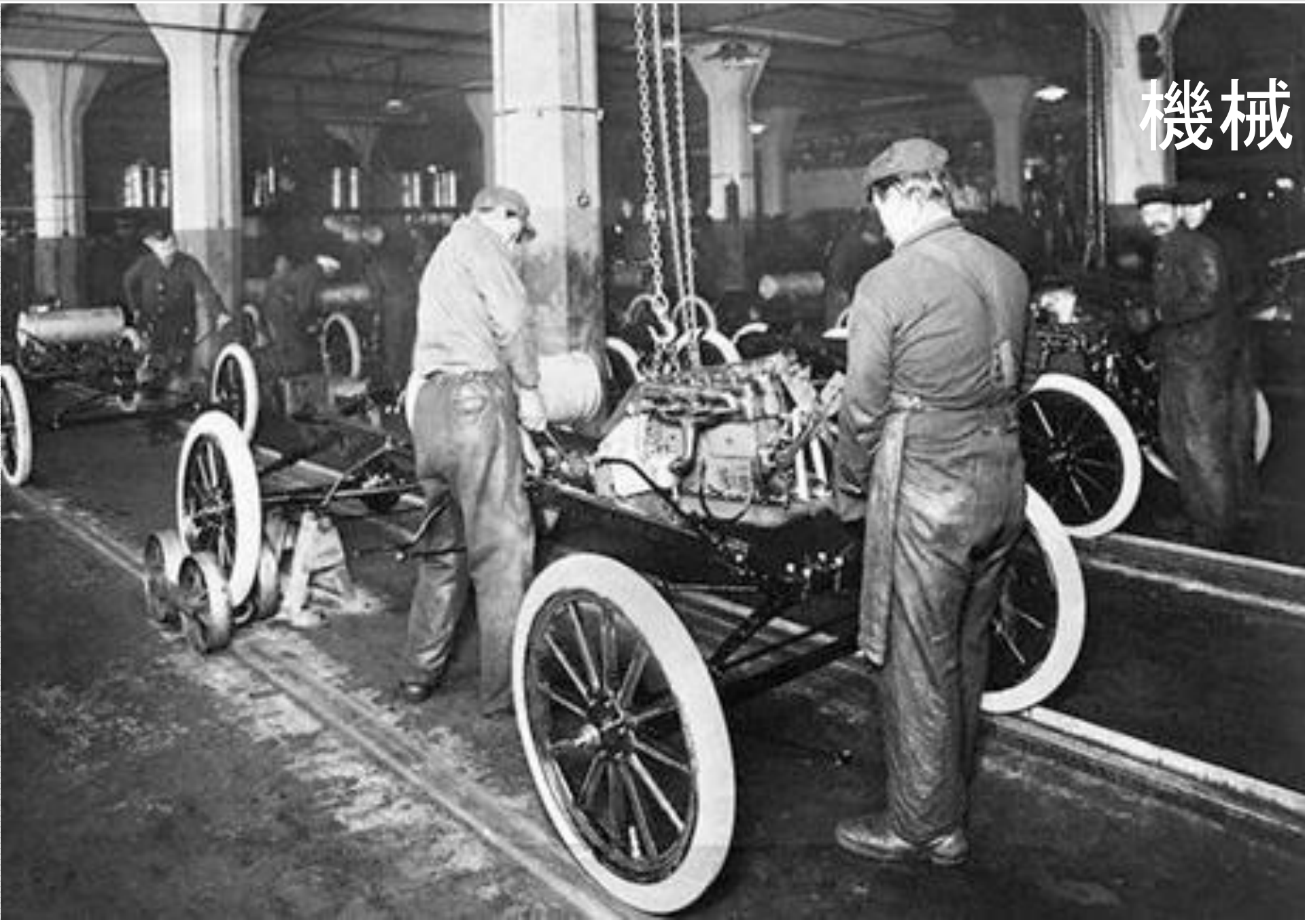

文字




ANNO MDCCXL



機械


機械

今回のセミナーの問題意識

今回のセミナー「機械の言語能力の獲得を考える」は、現代のAI技術の到達点を「機械が言語能力を獲得した」と捉える議論を展開したものです。

機械が新しく獲得した言語能力の中核は、「意味を理解する」能力だと僕は考えています。

今回のセミナーでは、機械の言語能力の獲得の中核を、機械の意味を理解する能力の獲得とする議論を行います。

中心問題は、機械は、どのようにして「意味を理解する」ようになったのか？ という問題です。この問題については、21世紀初めからの「意味の分散表現論」の発展が一つの答えを与えてくれると思っています。

セミナーでは、意味のベクトル表現の発見に始まり、翻訳モデルから大規模言語モデルへの発展へと結実する理論の歴史を振り返ろうと思います。

言語能力と「知性」

言語能力と「知性」

僕は、人間の知性（あるいは、知能）と人間の言語能力を区別しています。人間の知能は複雑な構造を持ち、その最も基本的な構成要素、最も重要な基礎が言語能力なのだと。

言語能力をもつ人間がそうであるように、機械が人間並の言語能力を獲得したとしても、それだけで優れた「知性」を発揮するかはわかりません。

機械の言語能力の獲得は、機械が正しいことを言うことを意味するものではありません。ただ、言語能力なしには、優れた知性に成長することはできないと考えています。

その意味では、機械の言語能力の獲得をAI技術の重要な到達点と考えることは、大きな意味を持っていると考えています。

理論の歴史が示すもの
embedding概念の発見とその意味

人工知能の技術にも、短いながら歴史があります。その技術の歴史を貫いて、理論の歴史があります。今回のセミナーでは、AI技術の理論の歴史を振り返ってみたいと思います。

重要なことは、この4半世紀のAI技術の理論史は、「意味とは何か」を中心的なテーマとして、それを探究する理論の旅に他ならなかったということです。それが、今回のセミナーで展開する「意味の分散表現論」の発展史です。

意味の分散表現論の系譜 – 大規模言語モデルへ

- 2003年 Bengioの「次元の呪い」と語の特徴の分散表現
- 2006年 HintonのAuto Encoder 意味的ハッシング
- 2011年 RNNによる文の生成
- 2013年 Word2Vec 語の意味ベクトル
- 2014年 Sequence to Sequence 文の意味ベクトル
- 2014年 Attention Mechanism
- 2015年 RNNの不思議な力 RNNは文法を理解している
- 2016年 Google ニューラル機械翻訳
- 2017年 Transformer
- 2019年 BERT
- 2020年 GPT-3
- 2022年 ChatGPT

2010年 Coecke DisCoCat論文

2018年 Bradley DisCoCat解説論文

2020年 Bradley DisCoCat批判

2021年 Bradley copresheaf 論文

機械が意味を理解させるためには、人間が意味とは何かを知らなくてはなりません。また、それを実装として機械に伝えなくてはなりません。いろんな試行錯誤があったのですが、意味を多次元のベクトル、embedding として表現するという方法に辿り着き、その応用に磨きをかけていきますます。

意味を表現するembeddingという概念の発見は、この4半世紀のAI研究の白眉だと思います。我々は、embedding という人間と機械の共通言語を獲得し、それを通じて機械と意味を通じ合うことができるようになったのです。

人間にとって、embedding は、話す聞くことばとしての音声と、書く読むことばとしての文字に次ぐ、ことばの第三の形態だと僕は考えています。

こうしたアプローチの意味と課題

「言語能力の獲得」→「意味の理解能力の獲得」→「意味の分散表現論の発展」というスキームや、AI技術の発展を分散表現論の歴史で説明するアプローチには、多くのものを捨象しているという問題もあります。

AGI論の功罪

アルトマンは、2035年までに、あらゆる個人が「2025年時点の全人類に匹敵する知的能力」を手に入れることができると予測しています。
https://www.marketingaiinstitute.com/blog/the-ai-show-episode-135

一方、イリヤ・サツケヴァーは、アルトマンとは対照的に、AIがもたらす「実在的なリスク(Existential Risk)」に対して恐怖の念を抱いています。彼は、AIが単なるプログラムではなく、いつか人間を凌駕し、制御不能になる可能性を技術的必然として捉えています。
https://www.ibm.com/think/topics/superalignment

昔、ある人がAIに次のような皮肉な定義を与えたことがあります。

「AIとは、まだできていないこと全てである」"Al is whatever hasn't been done vet."

("Gödel, Escher, Bach" D. R. Hofstadter)

AGI (artificial general intelligence)の議論は、それに似ています。

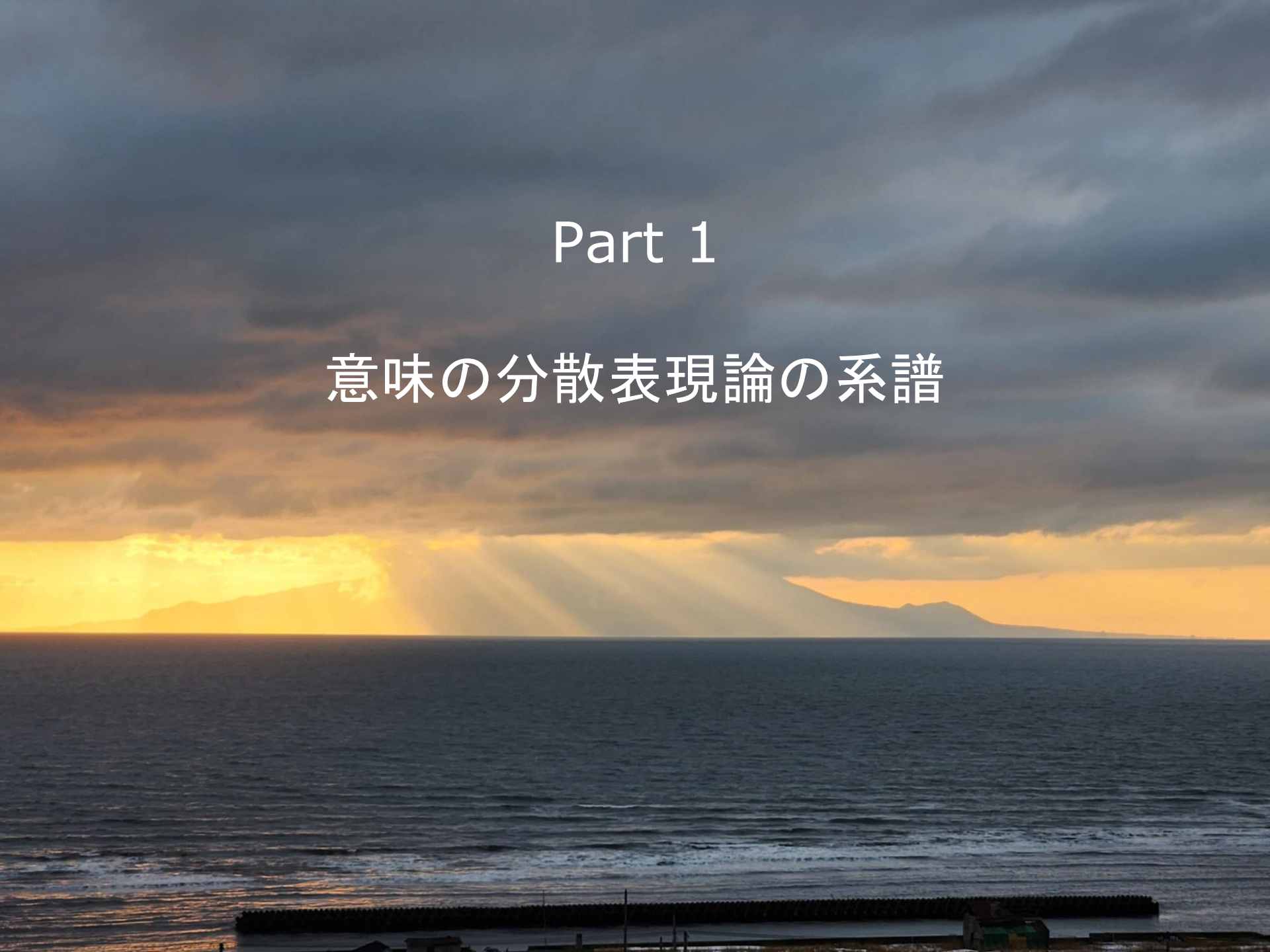
楽観的なAGI論も悲観的なAGI論もあるのですが、AGI論は、AI技術の到達点の評価ではなくまだできていないこと、まだ起きていないことを含んだ未来の予想です。

その予想には、傾聴すべき議論も含まれていることもあるのですが、ある場合には、AIにさらなる投資を呼び込むための誇大宣伝に使われています。

次回のセミナーで扱うこと

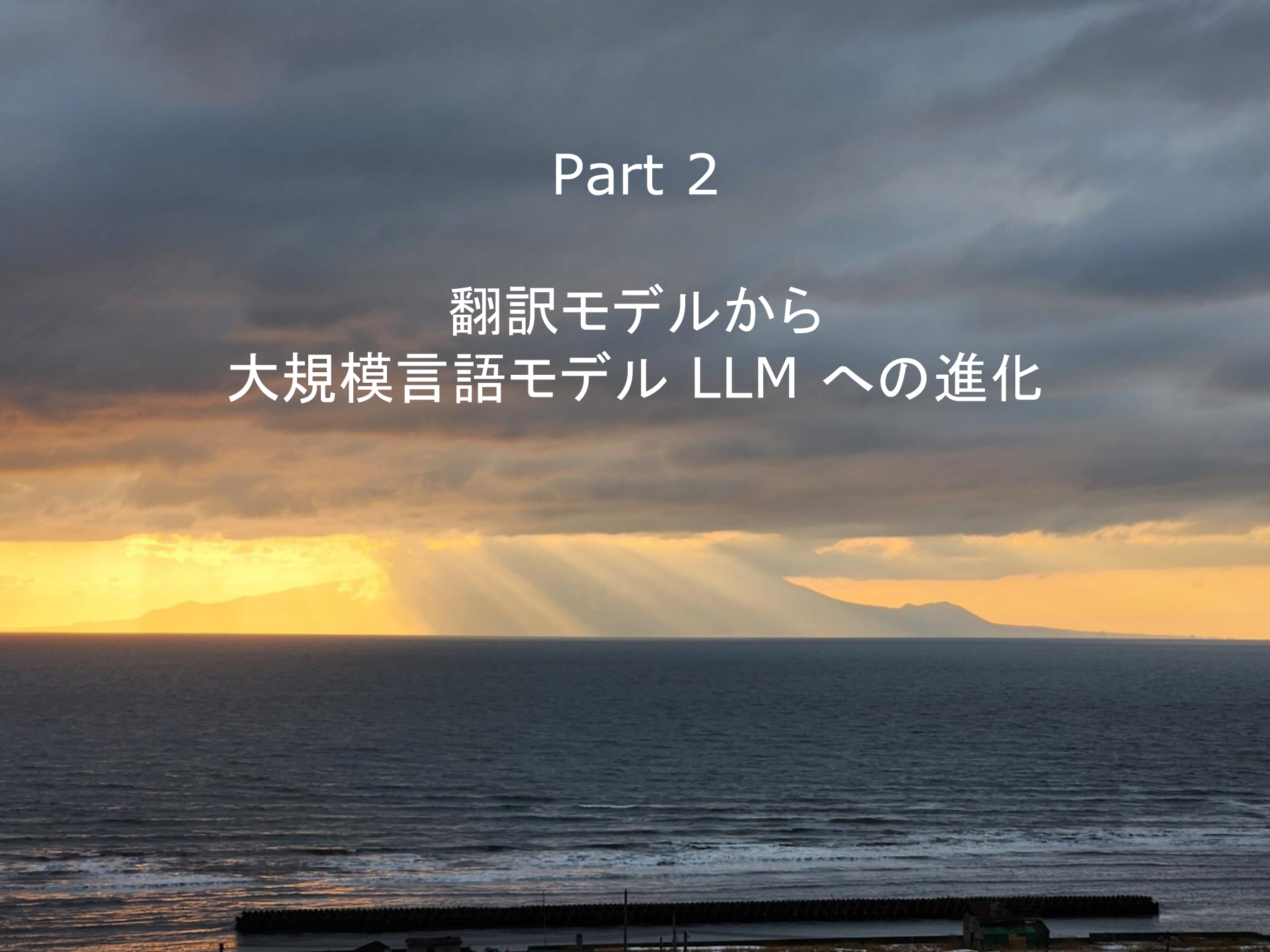
残念ながら、今回のセミナーでは、このembeddingの獲得が、情報の世界でどのようなインパクトを持つかは十分に語ることができません。

それについては、冒頭に述べたように、次回のセミナーは「embeddingの共有・蓄積・検索の未来」として展開したいと思っています。



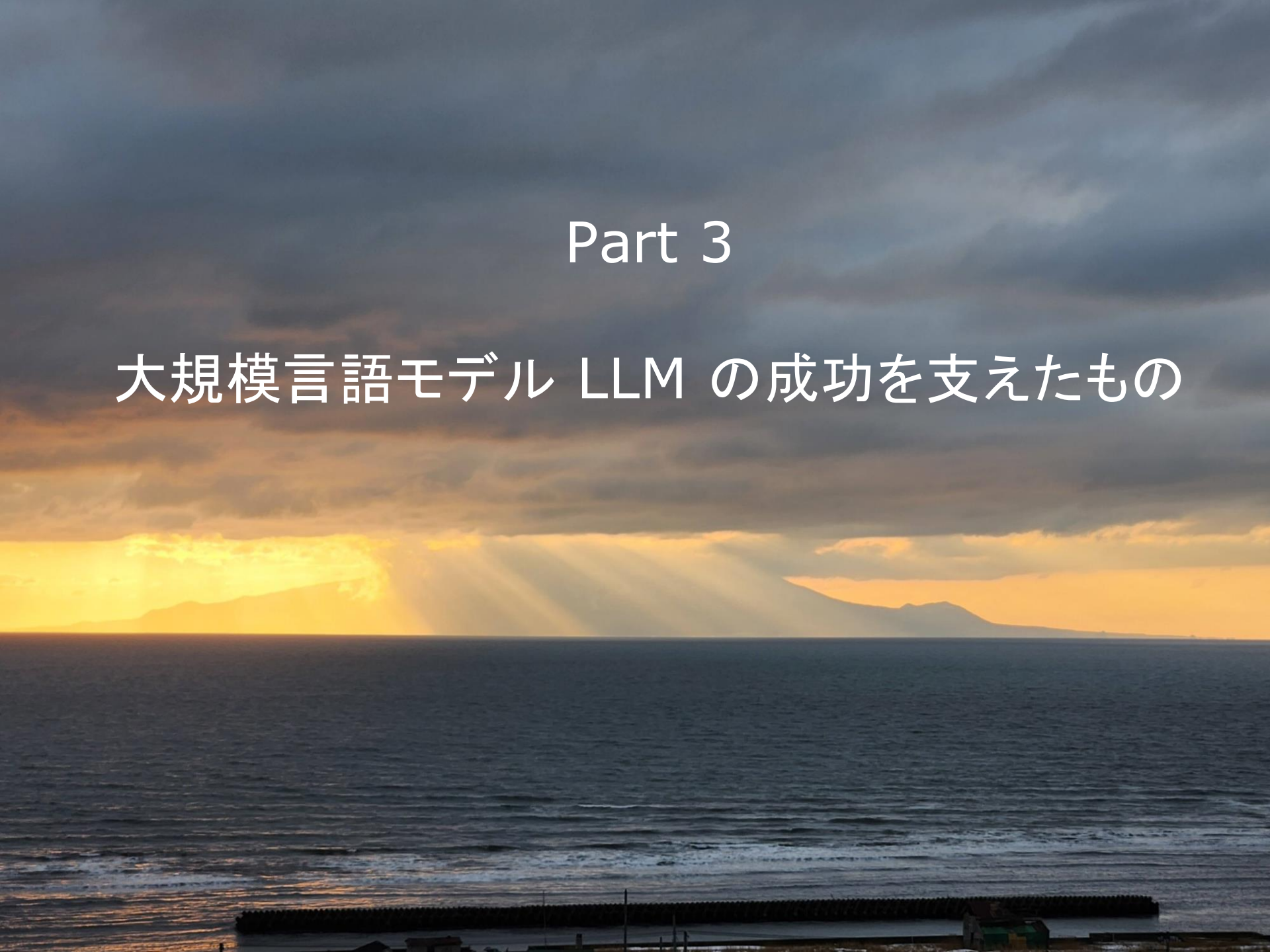
Part 1

意味の分散表現論の系譜



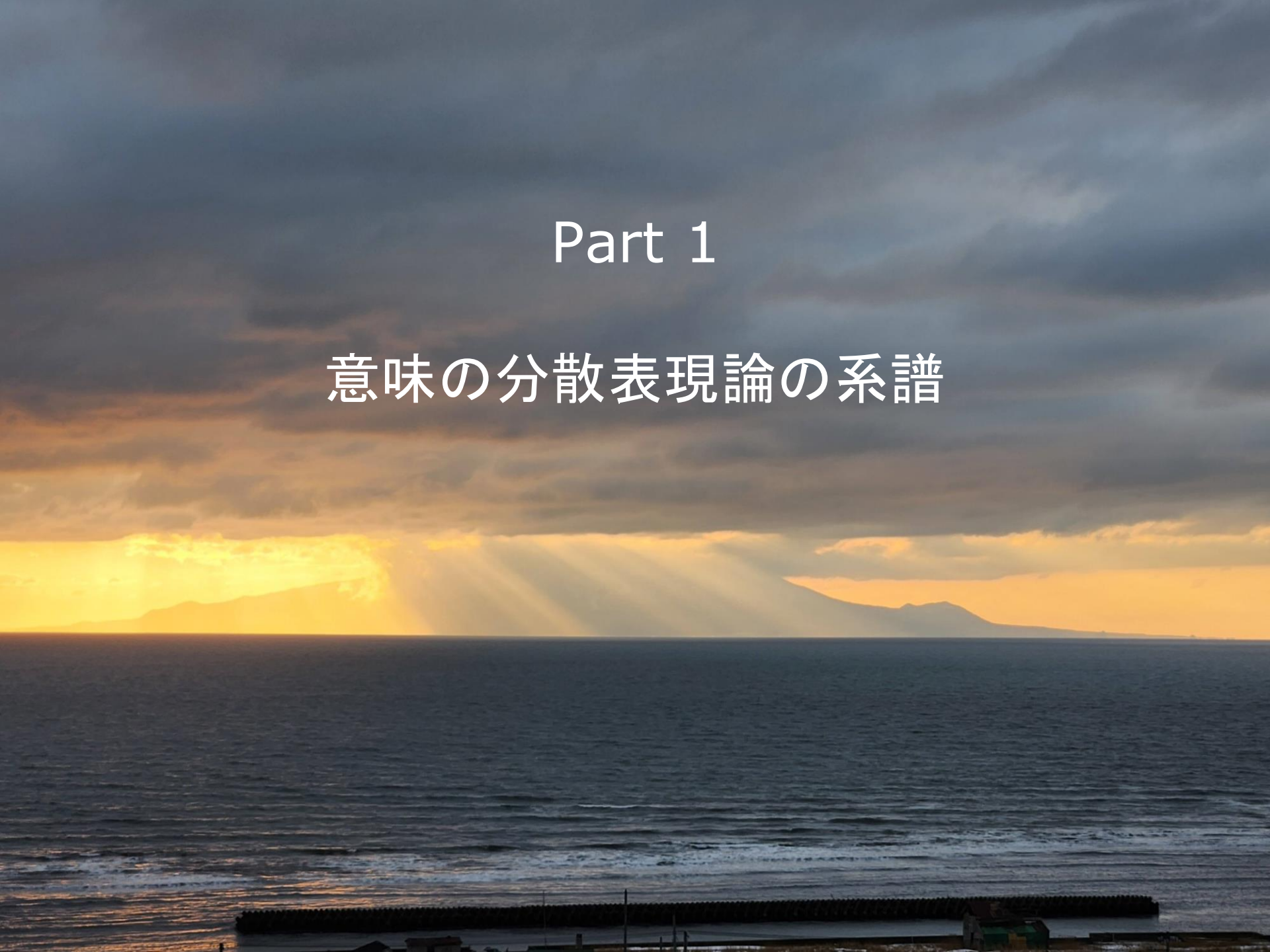
Part 2

翻訳モデルから
大規模言語モデル LLM への進化



Part 3

大規模言語モデル LLM の成功を支えたもの



Part 1

意味の分散表現論の系譜

意味の分散表現論の系譜 – 大規模言語モデルへ

- 2003年 Bengioの「次元の呪い」と語の特徴の分散表現
- 2006年 HintonのAuto Encoder 意味的ハッシング
- 2011年 RNNによる文の生成
- 2013年 Word2Vec 語の意味ベクトル
- 2014年 Sequence to Sequence 文の意味ベクトル
- 2015年 RNNの不思議な力 RNNは文法を理解している
- 2016年 Attention Mechanism
- 2016年 Google ニューラル機械翻訳
- 2017年 Transformer
- 2019年 BERT
- 2020年 GPT-3
- 2022年 ChatGPT

Bengioの「次元の呪い」と語の特徴の分散表現

2003年

A Neural Probabilistic Language Model

Yoshua Bengio et al.

http://goo.gl/977AQp

2003年

featureの分散表現で、「次元の呪い」と戦う

この論文で、彼は、次のような方法を提案する。

1. 語彙中のそれぞれの語に、R^mに実数値の値を持つ、分散した語の特徴ベクトル(word feature vector)を対応づける。
2. 語の並びの結合確率関数を、この並びの中の語の特徴ベクトルで表現する。
3. 語の特徴ベクトルとこの確率関数のパラメーターを、同時に学習する。

要は、語の「特徴ベクトル」という、語の「意味」の対応物を導入しようということだと僕は理解している。こうしたアプローチは、Tomas Mikolovらの **Word2Vec** に受け継がれていく。

少し乱暴にまとめてみる

- 「文」全体を相手にすると、その数はあまりに多すぎて、いくら統計的手法を使っても、言語の特徴をとらえるのは難しい。「次元の呪い」で勝ち目はない。それより遥かに数の少ない「語」の特徴づけから始めよう。「語」から「文」を攻めよう。
- それにしても「語」の特徴が一つの数字で表されるとは思えない。m個の数字の組 すなわちm次元のベクトルとして、「語」の特徴を表すことにしよう。
- 「文」は「語」の並びである。「文」が、「語」の並びとしてある確率的特徴を持つなら、それは「語」の特徴と結びつけることができるはずだ。
- コンピュータは、語の特徴ベクトルと「語」の並びの結合確率関数のパラメーターを、同時に学習すればいい。

HintonのAuto Encoder
意味的ハッシング

2006年

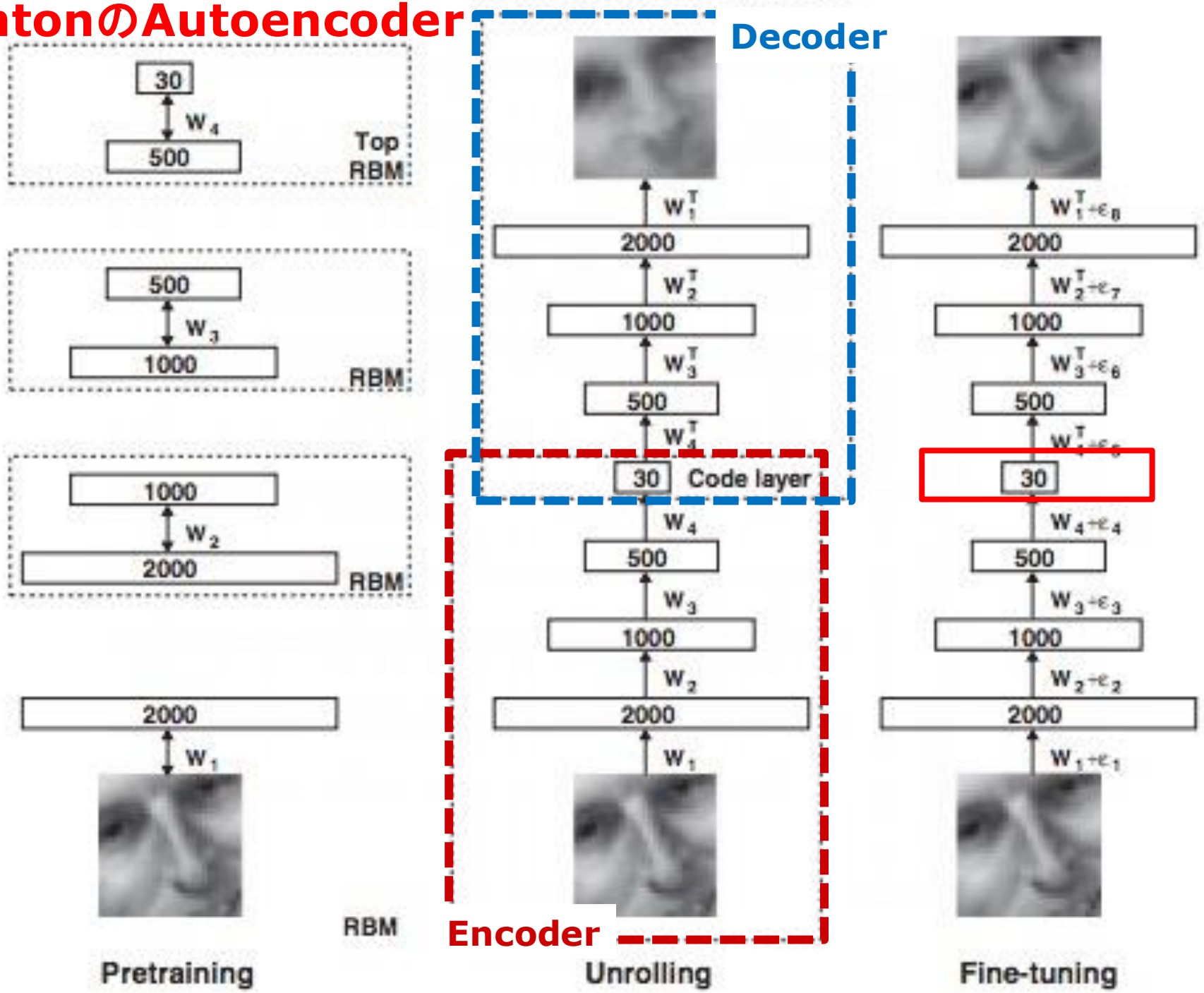
Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton et al.

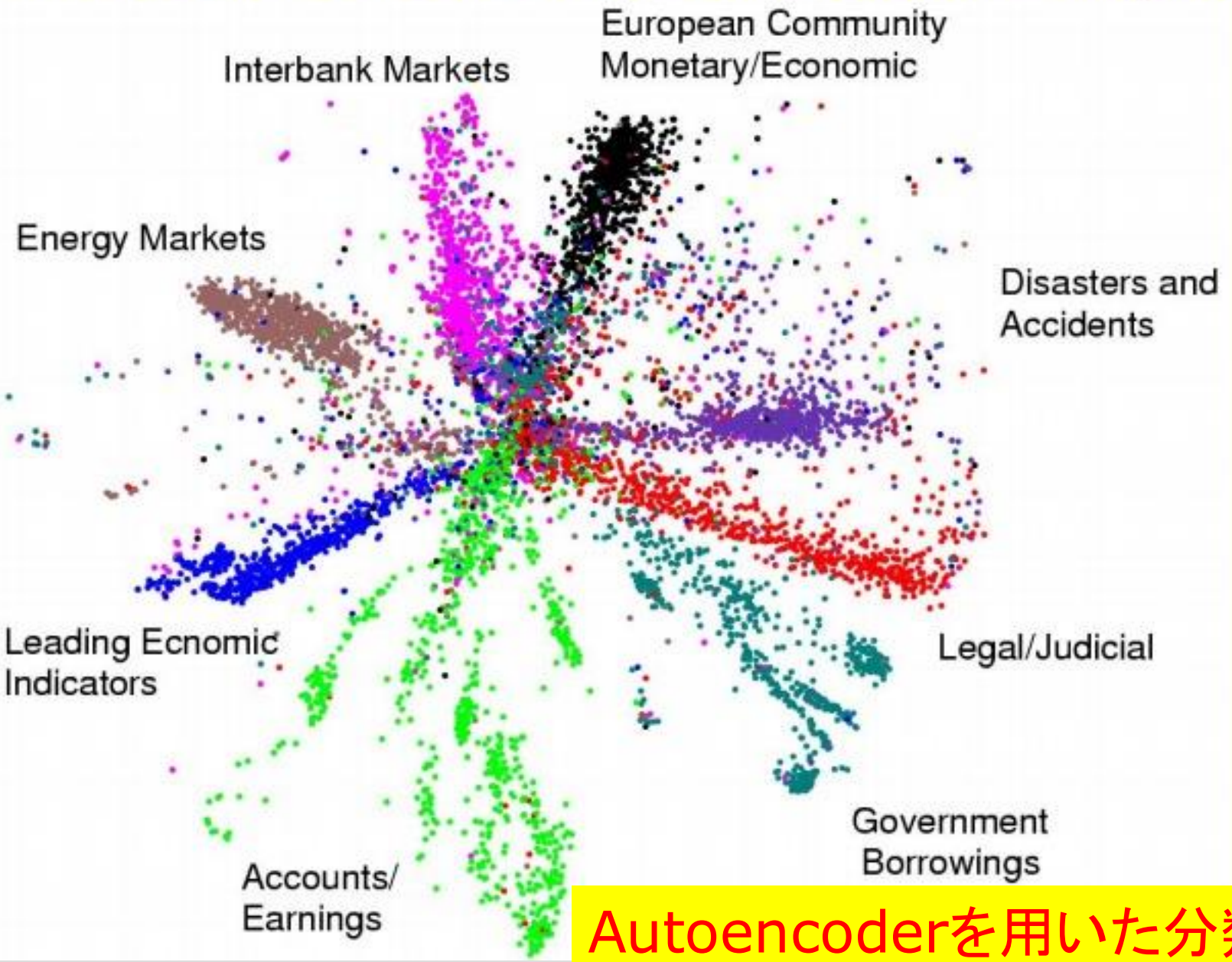
https://www.cs.cmu.edu/~tom/10701_11/slides/DeepNets_science2006.pdf

2006年

HintonのAutoencoder



First compress all documents to 2 numbers.
Then use different colors for different document categories



Autoencoderを用いた分類

Semantic hashing（意味的ハッシング）

- 重要なことは、「画像」と「書籍」では、対象のデータの性質はまるで異なるのだが、Autoencoderは、そのいずれに対しても、高次元のデータを低次元のデータに変換しているということである。別の言葉で言えば、**それは、対象の高次元のデータから、低次元のデータを、元の情報のエッセンスとして取り出しているのである。**
- Hintonは、こうしたAutoencoderの働きを、**Semantic hashing（意味的ハッシング）**と呼んでいる。
- SHA-1のようなハッシングでは、ハッシュ化されたデータから元のデータを復元することは不可能なのだが、Semantic hashingされたデータは、データの次元は低いものの、元の情報の中核部分を保持している。

「語の意味ベクトル」

2013年

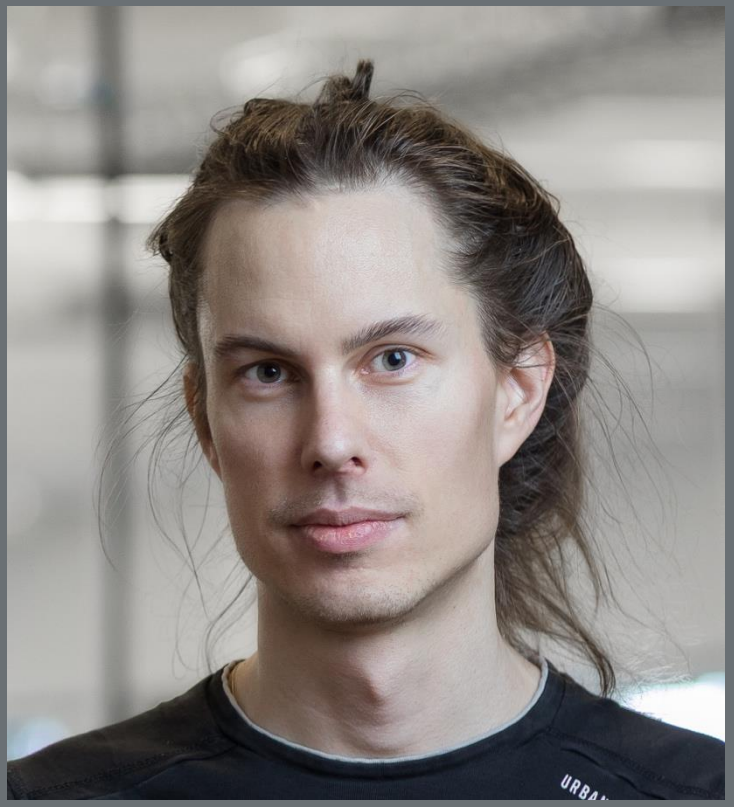
MikolovらのWord2Vec論文が出るのは、Benjioの論文から10年後の2013年だった。

Linguistic Regularities in Continuous Space Word Representations

Tomas Mikolov et al.

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/rvecs.pdf

2013年



Word2Vec論文

2013年に、Google（当時）のTomas Mikolovらは、語が埋め込まれたベクター空間が、言語学的に（文法的にも、意味論的にも）面白い性質を持っていることを発見する。

Tomas Mikolov,Wen-tau Yih,Geoffrey Zweiget

Linguistic Regularities in Continuous Space Word Representations

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/rvecs.pdf

Word2Vec 二つ目の論文とコード公開

Jeff Deanが先の論文に興味を持ち、共同で研究を始め二つ目の論文を発表。

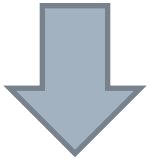
Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean
Efficient Estimation of Word Representations in Vector Space
https://arxiv.org/pdf/1301.3781.pdf

Google Codeに、オープンソースとして公開され、大きな関心を集める。
https://code.google.com/p/word2vec/

どんな語が、与えられた語の近くに埋め込まれるか？

- 似た意味を持つ言葉は、似たベクトルを持つ。
- 似た言葉で置き換えても、正しい文は、正しい文に変わる。

“a **few** people sing well”　　正しい文



“a ***couple*** people sing well”　　正しい文

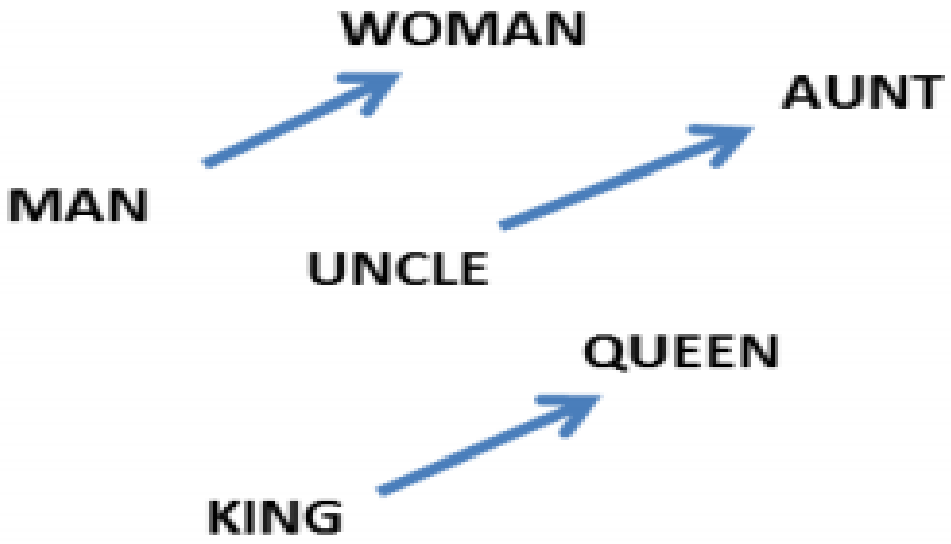
- 意味が似ていなくても、同じクラスの言葉で置き換えても、正しい文は、正しい文に変わる。

“the **wall** is **blue**” ⟶ “the ***ceiling*** is ***red***”

意味を変換するベクトルは共通？

Word Embeddingは、もっと面白い性質を持つ。下の図のように、男性から女性へのベクトルがあるように見える。

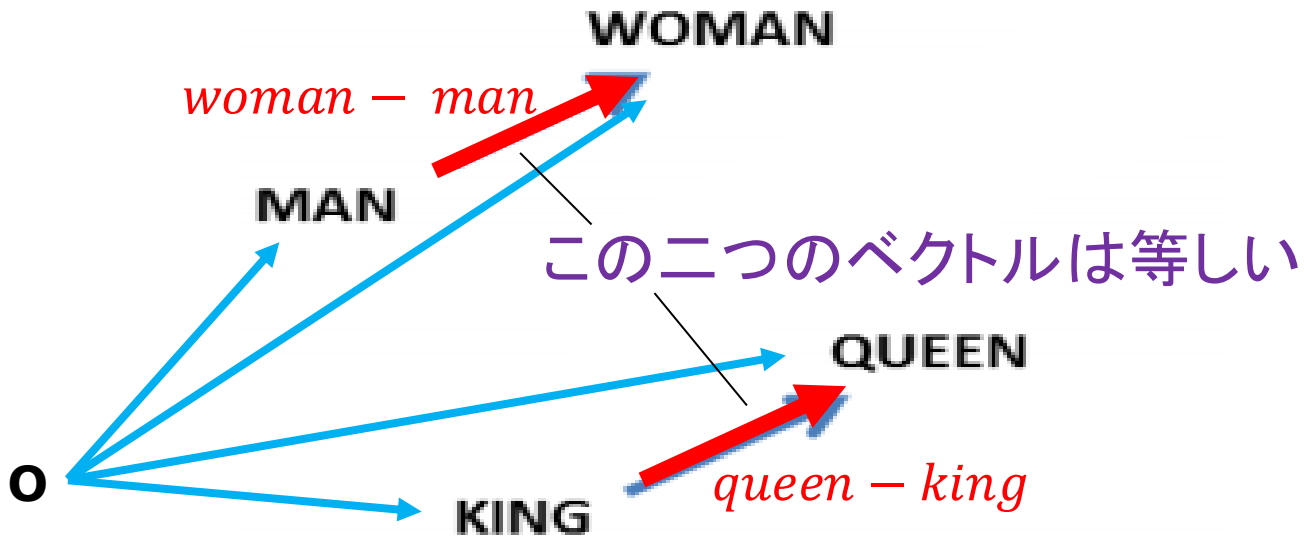
W(“woman”)−W(“man”) ≃ W(“aunt”)−W(“uncle”)
W(“woman")−W(“man") ≃ W(“queen")−W(“king")



意味を変換するベクトルは共通？
Vector Offset Method

Word Embeddingは、もっと面白い性質を持つ。下の図のように、男性から女性へのベクトルがあるように見える。

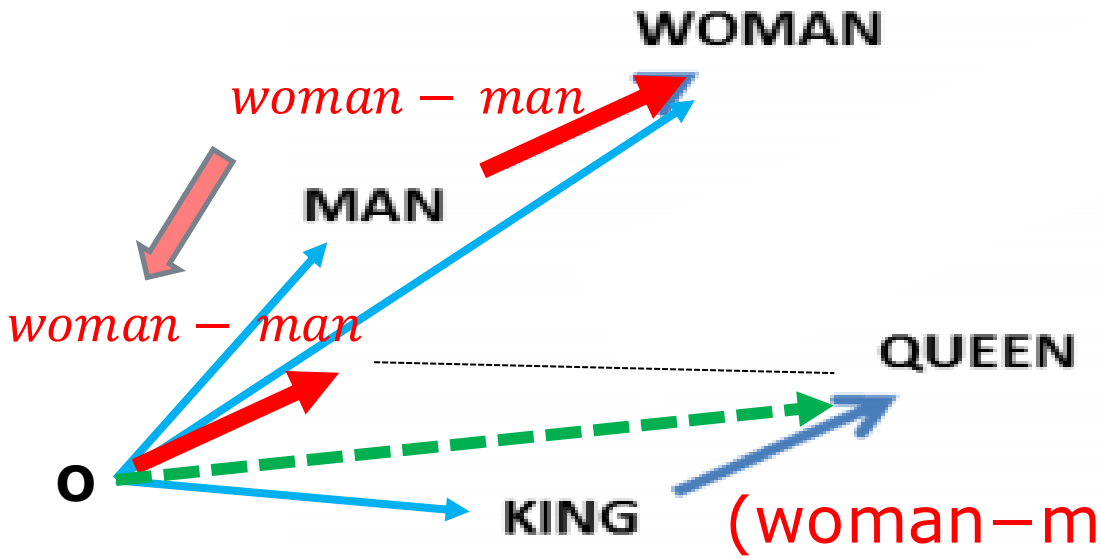
W(“woman")−W(“man") ≃ W(“queen")−W(“king")



King - Man + Woman = Queen
Vector Offset Method

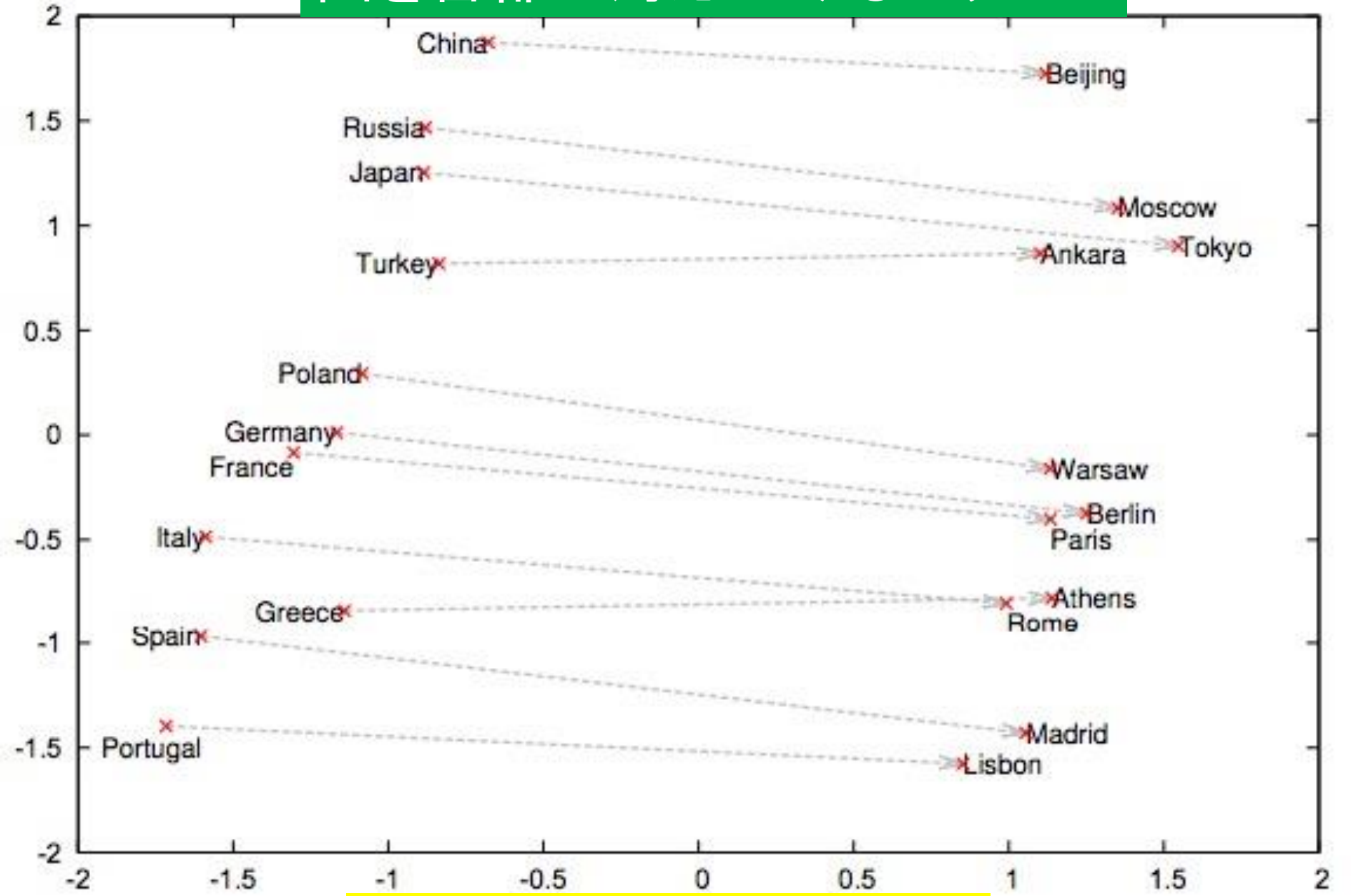
次のような変形もできる。

W(“woman")–W(“man") + W(“king") ≃ W(“queen")



(woman–man) + king ≃ queen

国を首都に対応づけるベクトル



http://arxiv.org/pdf/1310.4546.pdf

「語の意味へのベクトル:で重要なこと
語の意味の近さを定義できる

「語の意味のベクトル表現」でもっとも重要なことは、このアプローチによって、語の意味の近さを定義できることである。

語vと語wのベクトル表現を$\vec{v}$と$\vec{w}$する。この時、

語vと語wの意味の近さ$Similarity(v,w)$を、ベクトル$\vec{v}$と$\vec{w}$の内積$\vec{v}\cdot\vec{w}$で定義する。

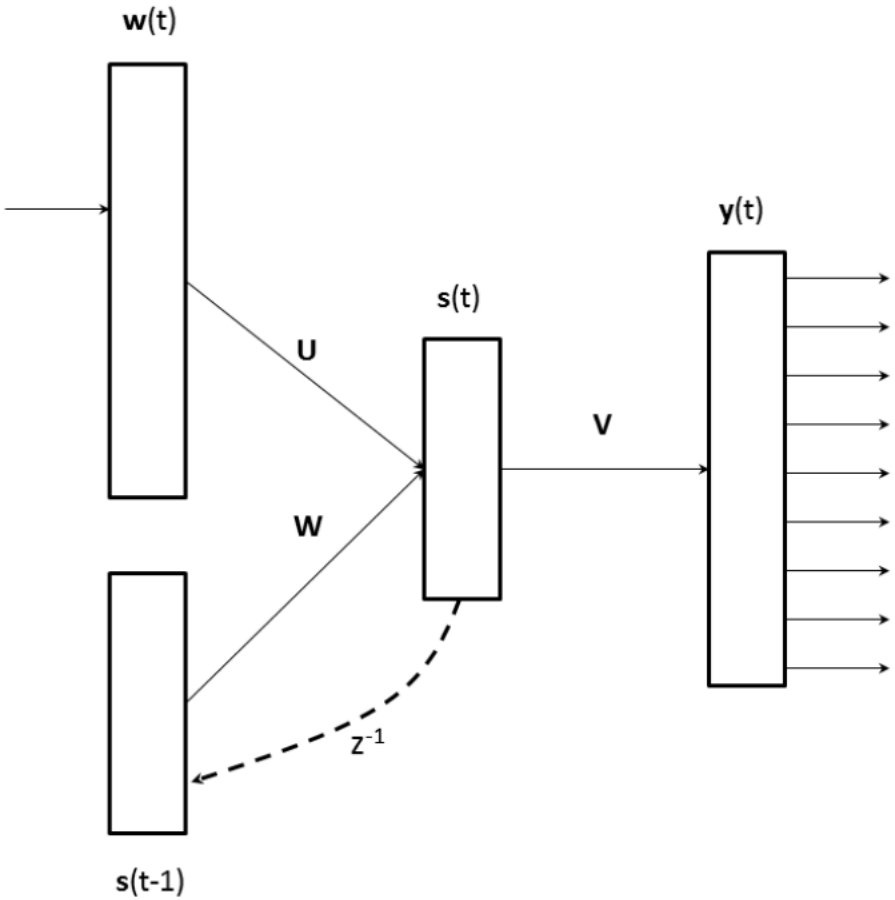
$$Similarity(v,w) = \vec{v}\cdot\vec{w} = |\overrightarrow{v}||\overrightarrow{w}|cos\theta$$

ここに、θは、ベクトル$\vec{v}$と$\vec{w}$のなす角である。これを、cosine-similarityと呼ぶ。

語はどのようにベクトルに変換されるのか

WordをVectorに変える方法

第一論文では、次のようなRNNが使われている。



$$\mathbf{s}(t) = f\left(\mathbf{U}\mathbf{w}(t) + \mathbf{W}\mathbf{s}(t-1)\right)$$

$$\mathbf{y}(t) = g\left(\mathbf{V}\mathbf{s}(t)\right),$$

$$f(z) = \frac{1}{1+e^{-z}}$$

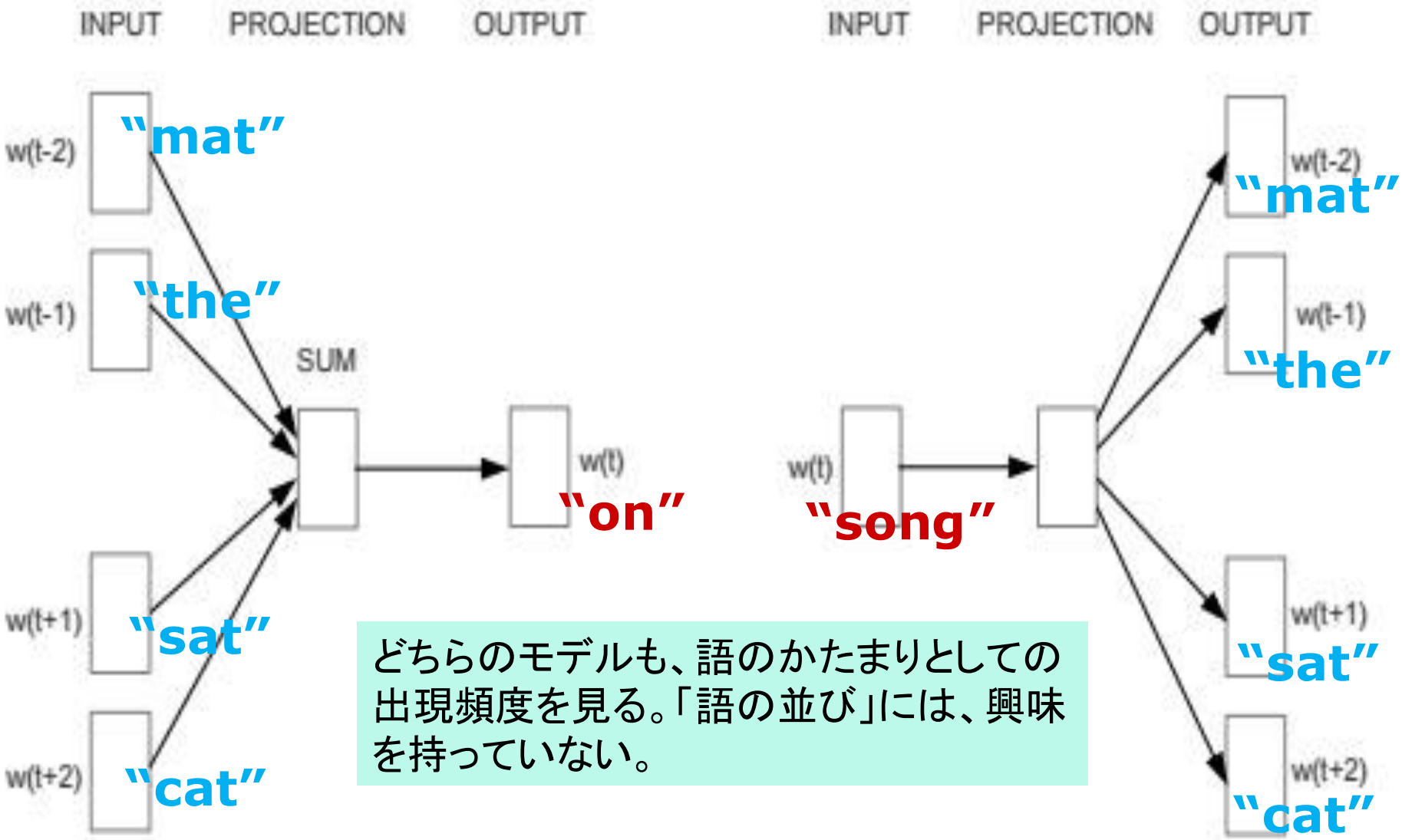
$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

WordをVectorに変える方法

第二論文で使われている方法は、次の二つである。

- **CBOW(Continuus Bag-of-Word”) モデル**
  複数の語の集まりから、一緒に出現しそうな一つの語の確率を調べる。

- **Skip-gram モデル**
  一つの語が与えられた時、一緒に出現しそうな複数の語の確率を調べる。



Continuous Bag-of-Words

CBOW

Skip-gram

http://arxiv.org/pdf/1301.3781.pdf

Sequence to Sequence
文の意味ベクトル

2014年

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever et al.

https://arxiv.org/pdf/1409.3215.pdf

2014年

文の意味ベクトルの発見

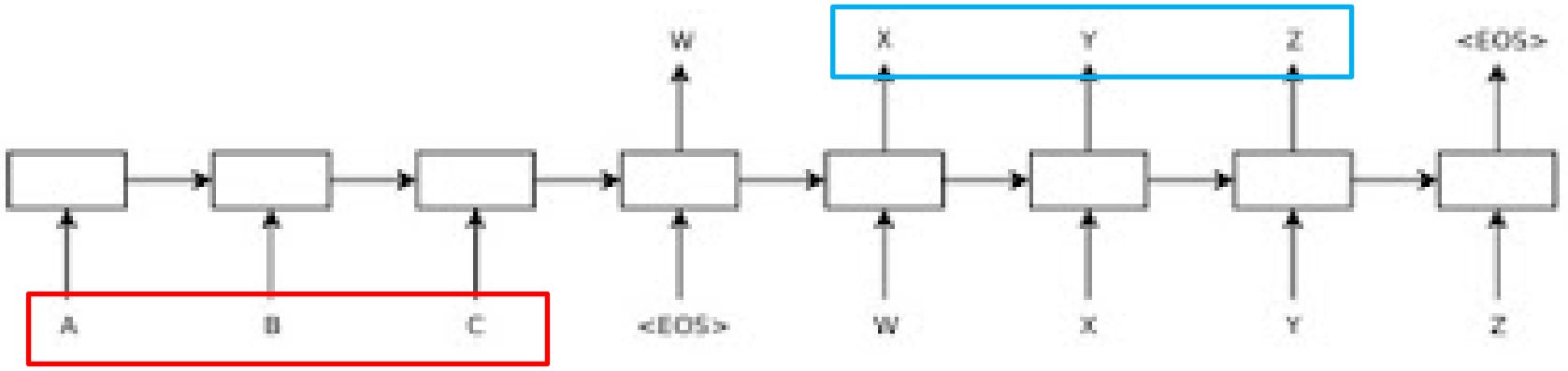
2014年に、Ilya Sutskever らは、シーケンスをシーケンスに変換するRNN(LSTM)の能力が、機械翻訳に応用できるという論文を発表します。

「我々の方法では、入力のシーケンスを固定次元のベクトルにマップするのに、多層のLong Short-Term Memory（LSTM）を利用する。その後、別の深いLSTMが、このベクトルから目的のシーケンスをデコードする。」

それでは、二つのSequence を結びつけているのはなんでしょう。それは二つのSequenceが「同じ意味」を持つということです。前段の入力のSequenceから作られ、後段の出力のSequenceを構成するのに利用される「固定次元のベクトル」とは、二つの文が「同じ意味」を持つことを表現している文の意味のベクトル表現に他なりません。

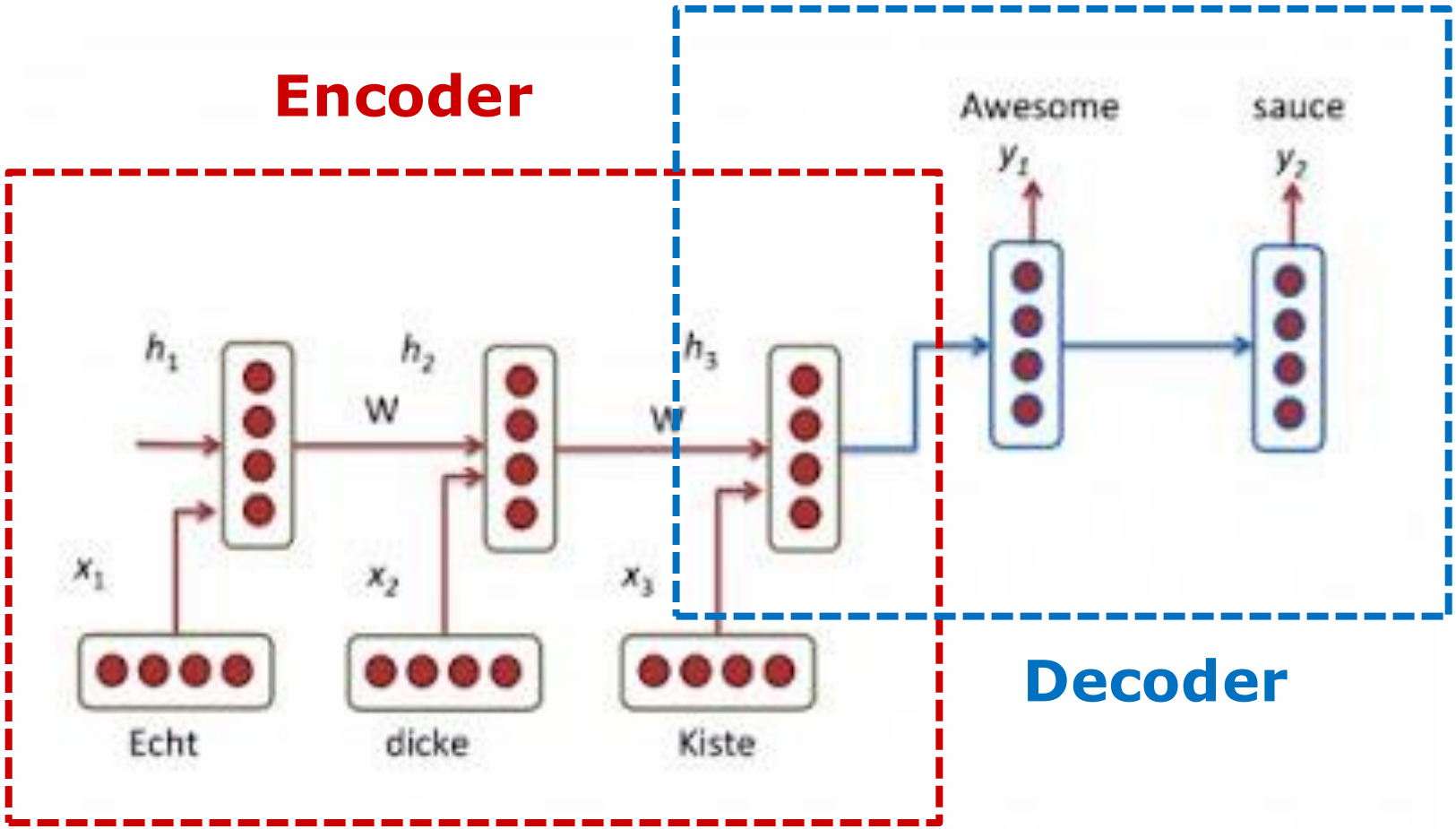
Sequence to Sequence

この論文の次の図を見て欲しい。

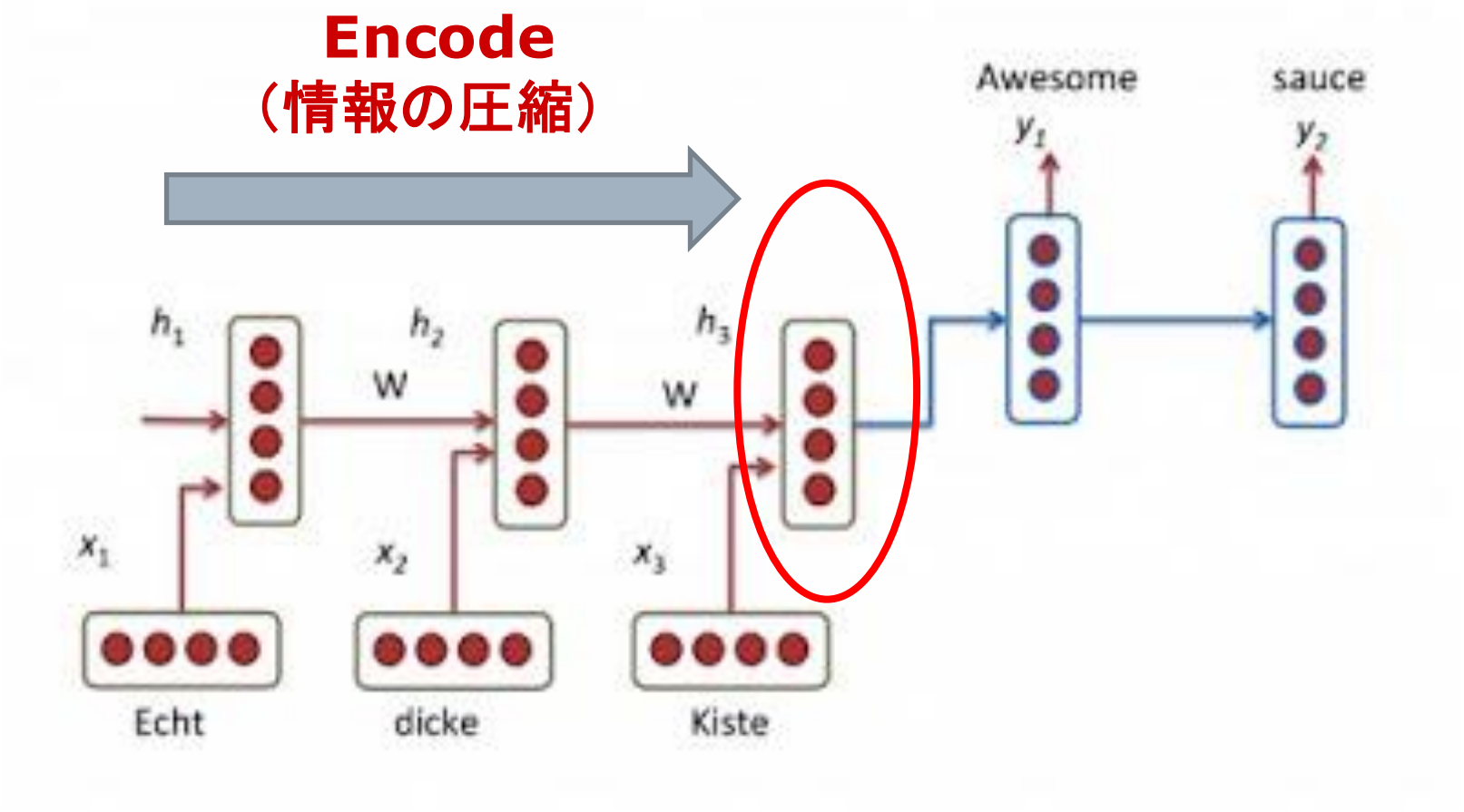


この図は、このシステムが、ABCというシーケンスが与えられた時、xyzというシーケンスを返すことを表している。<EOS>は、End of Sequence でシーケンスの終わりを表す特別な記号である。（これが、シーケンスの構造に課せられた「最小限の前提」である。）

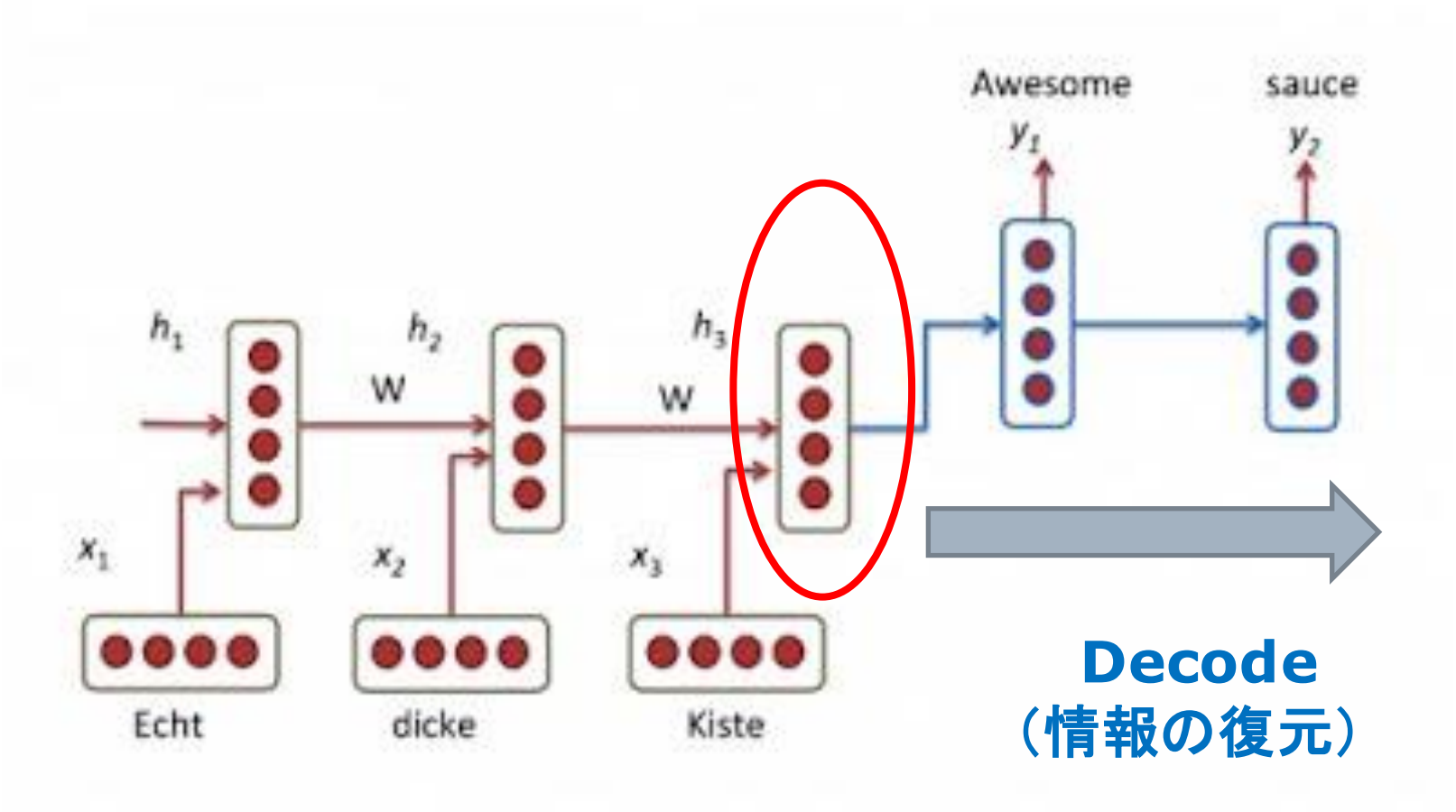
次の図(https://goo.gl/JGckBP から)は、こうしたメカニズムで、RNNが、独文の "Echt dicke Kiste" を英文の"Awesome sauce" に翻訳する様子を表している。(ここでは、文章の終わりを表す<EOS>は、省略されている)



ここでは、Encoder部が、文章の最後にとるRNNの内部状態 h_3 が、そのままDecoder部に渡されることが示されている。入力シーケンスの情報のエッセンスが、この内部状態 h_3に凝縮されていると考えればいい。



AutoencoderのDecoder部が、圧縮された情報から元の情報を復元しようとするように、ここでは、その情報から、「**同じ意味**」を持つ、別の言語の文章を復元しようとする。



Ilya Sutskever らは、このアーキテクチャーで、英語をフランス語に翻訳するシステムを作成し、BLEUのスコアで、34.81という高得点をたたき出した。

この時のシステムは、5段重ねのLSTMで構成され、それぞれが8,000次元の状態からなる384M個のパラメーターを持つものだった。

Attention Mechanism

Neural machine translation by jointly learning to align and translate

Bahdanau, D., Cho, K., and Bengio, Y

https://arxiv.org/pdf/1409.0473.pdf

2016年

論文の概要

近年、ニューラル機械翻訳として提案されたモデルは、多くの場合、Encoder-Decoderのファミリーに属している。そこでは、ソースの文が固定長ベクトルにエンコードされ、そこからデコーダが 翻訳文を生成する。

この論文では、固定長ベクトルの使用が、この基本的なEncoder/Decoderアーキテクチャの性能を改善する上でのボトルネックになっていると推論し、モデルに自動的に、ターゲット・ワードを予測するのに重要なソース・文の一部分について、（ソフト）検索を可能とすることによって、これを拡張することを提案する。

その際、これらの部分を明示的にハードセグメントとして形成する必要はない。

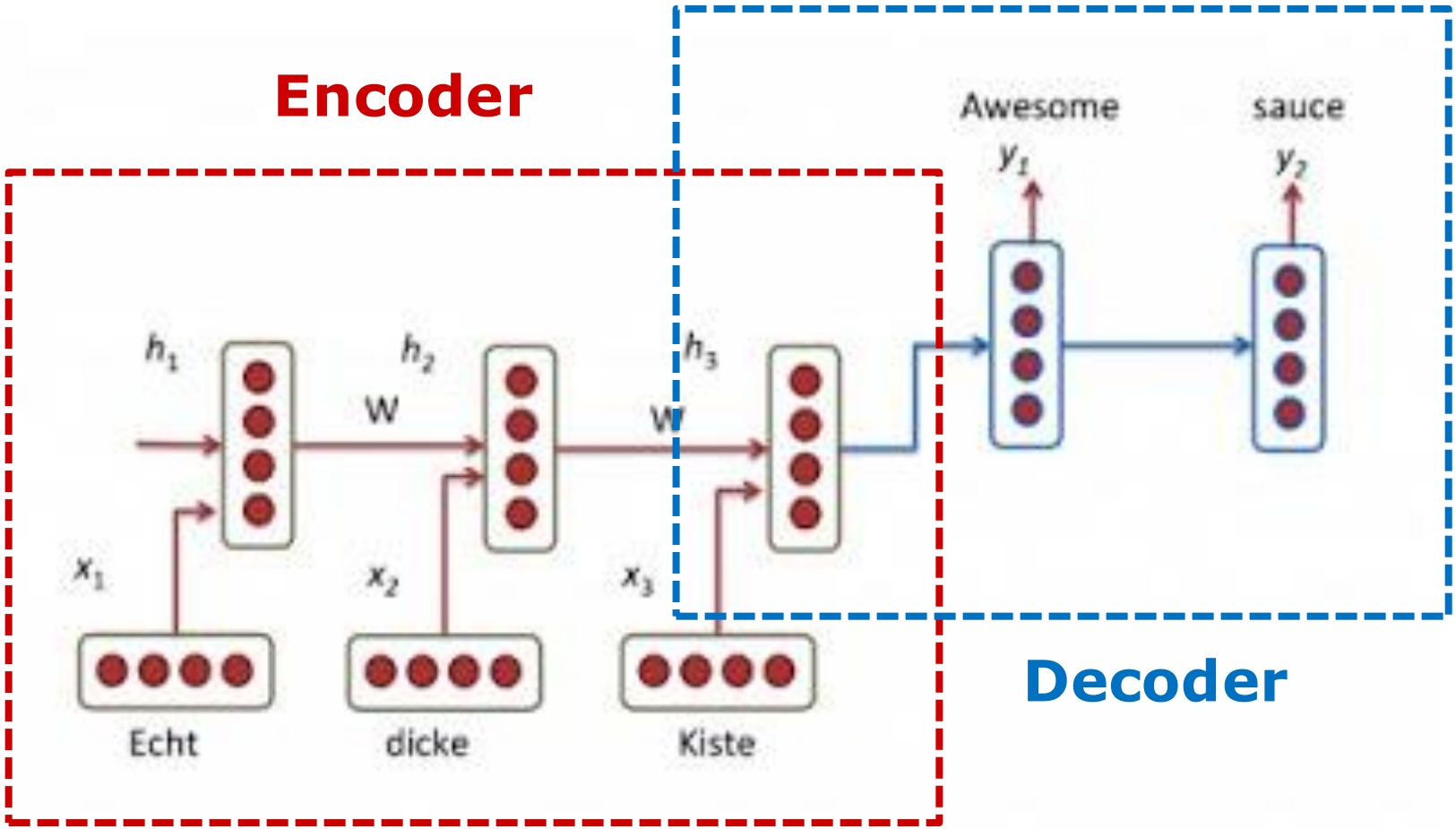
固定長ベクトルがボトルネック

先に見た、Ilya Sutskever らの翻訳システムでは、翻訳さるべき文は、Encoderで、一旦、ある決まった大きさの次元（例えば8000次元）を持つベクトルに変換される。このベクトルからDecoderが翻訳文を生成する。

入力された文が、長いものであっても短いものであっても、中間で生成され以降の翻訳プロセスすべての出発点となるこのベクトルの大きさは同じままだ。このシステムでは、長くても短くても入力された文全体が、一つの固定長のベクトルに変換されるのだ。

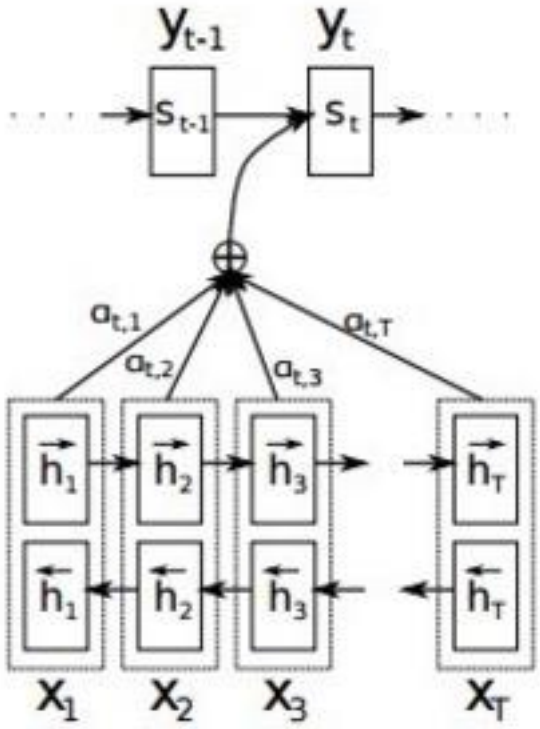
確かに、そこは翻訳の精度を上げる上でのボトルネックになりうる。事実、Ilya Sutskever らのシステムでは、文の長さが長くなるにつれて、翻訳の精度が低下されるのが観察されるという。

次の図(https://goo.gl/JGckBP から)は、こうしたメカニズムで、RNNが、独文の "Echt dicke Kiste" を英文の"Awesome sauce" に翻訳する様子を表している。(ここでは、文章の終わりを表す<EOS>は、省略されている)



この論文の基本的アイデア

文全体に一つの固定長のベクトルを割り当てるのではなく、翻訳時に、ソース・文の一部分を 改めて見直して、その部分から提供される情報を翻訳に生かそうということだ。



$a_{3,2}$が大きい場合、これは、Decorderがターゲット文の第3の単語を生成しながら、ソース文の第2の状態に多くの注意を払うことを意味する。

「ここで、yはデコーダによって生成された翻訳された単語であり、xは原文の単語である。上記の図は双方向のリカレント・ネットワークを使用しているが、それは重要ではない。逆方向は無視していい。

重要な部分は、各デコーダの出力するワード y_tが、Encoderの最後の状態だけでなく、すべての入力状態の重みづけられた結合に依存することである。

aは、出力ごとに、それぞれの入力状態をどの程度考慮されるべきかを定義する重みである。したがって、$a_{3,2}$が大きい場合、これは、Decorderがターゲット文の第3の単語を生成しながら、ソース文の第2の状態に多くの注意を払うことを意味する。

aは、通常、1に合計されるように正規化される（それらは、入力状態に対する確率分布である）。」

Google ニューラル機械翻訳

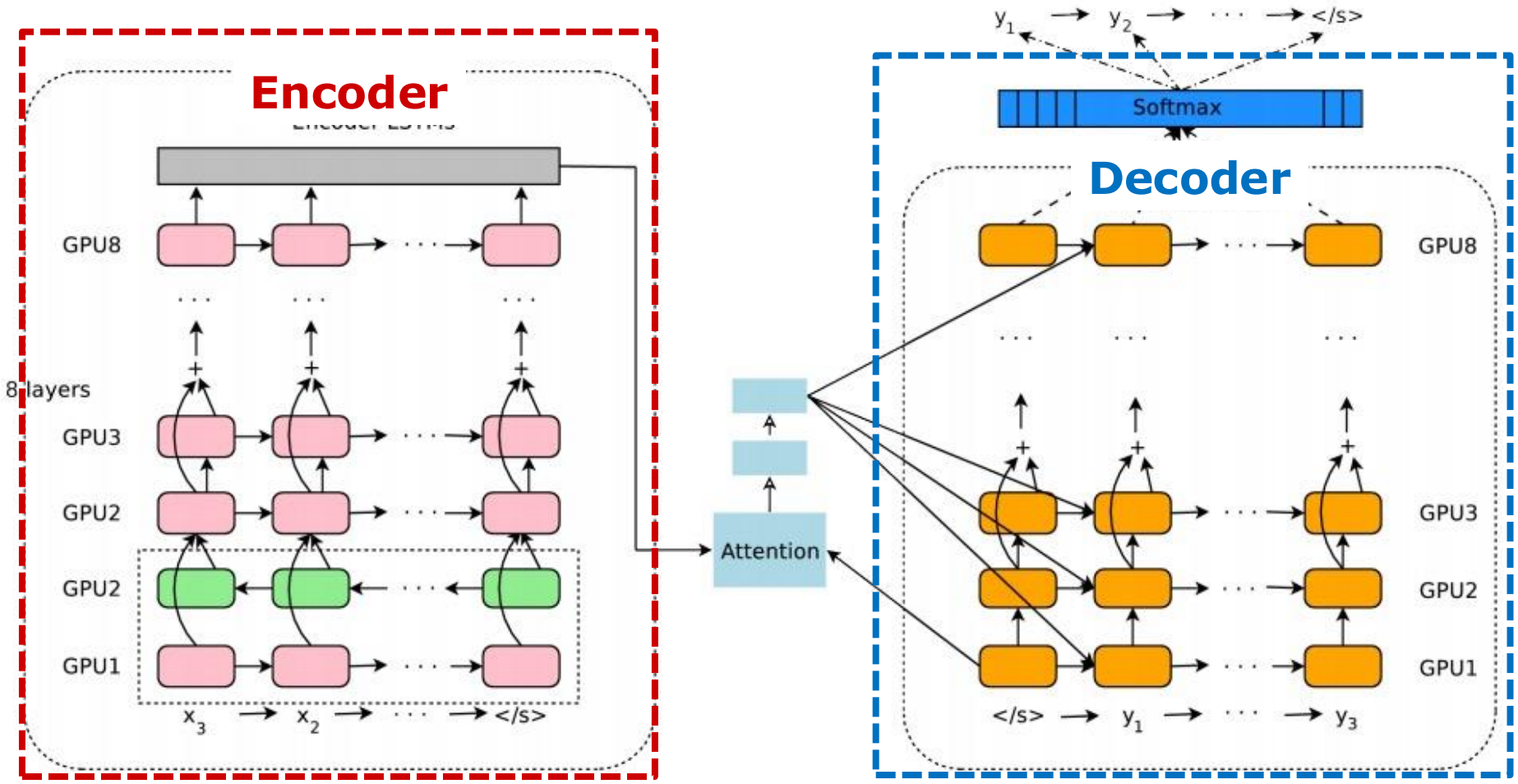
Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation

Yonghui Wu et al.

https://arxiv.org/pdf/1609.08144.pdf

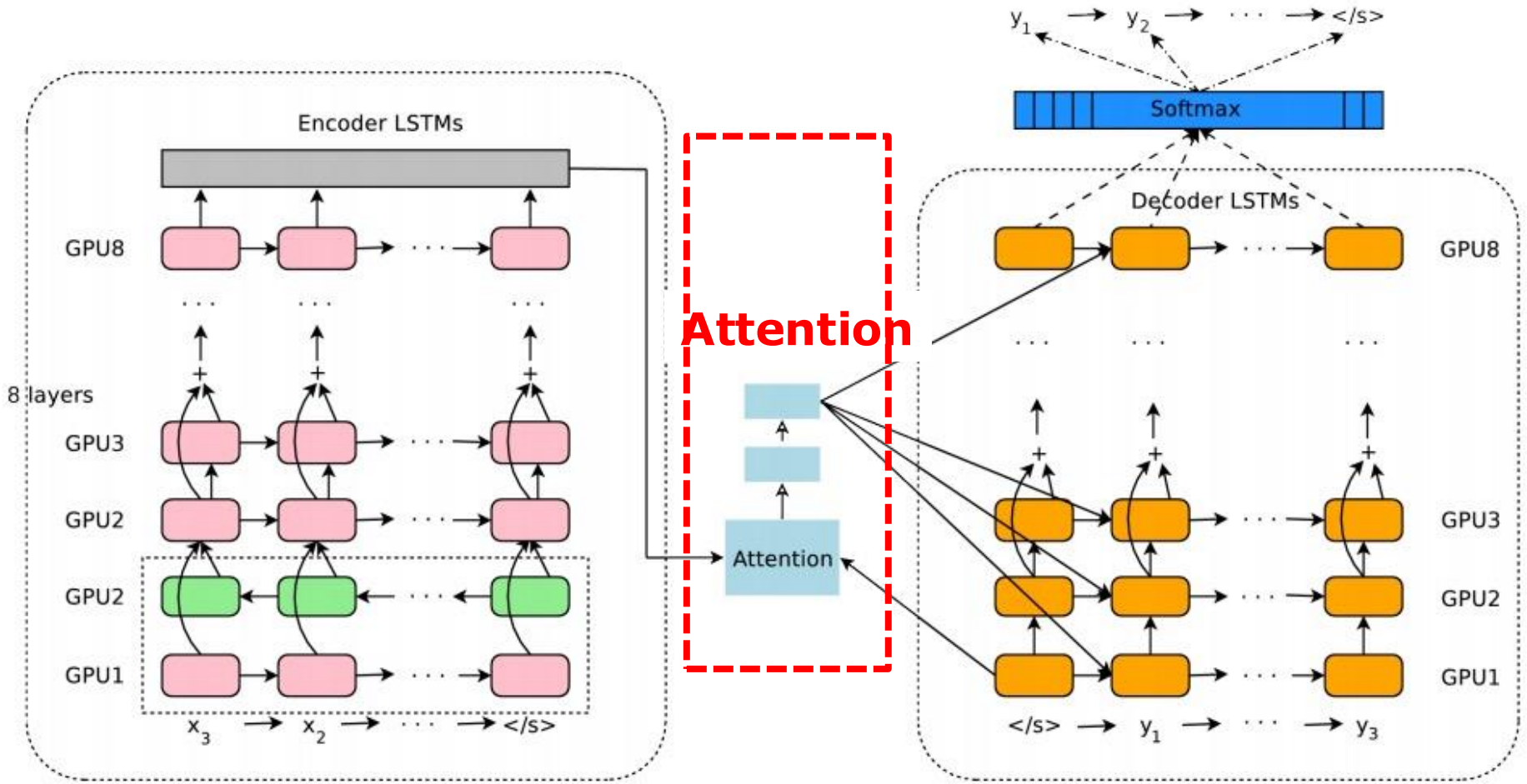
2016年

Encoder / Decoder



左側に、LSTMを8段重ねにした Encoder LSTMがあり、
右側には、同じくLSTMを8段重ねにした Decoder LSTMがある。

Attention Mechanism



EncoderとDecoderの中間に、Attentionと記された領域がある。
ここからの出力Attention Context は、Decoderのすべての
ノードに供給されている。

Google’s Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation

Melvin Johnson et al.

https://arxiv.org/pdf/1611.04558.pdf

2016年

Google 多言語ニューラル機械翻訳

多言語の翻訳を、言語ペアの数だけのシステムによって行うのではなく、一つのシステムで行うことにより、さらに多くの言語へのスケール・アップが容易になる。システムが単純になり、コーパスの少ない言語にもメリットが生まれる。

ゼロ・ショット翻訳が可能であることを示し、また、インターリンガの存在を示唆するなど、非常に刺激的である。

言語に対するいくつかの「仮説」と、ディープラーニングでの結果をある種の「実験」として、結びつけようとするスタイルは新しいものである。

論文の概要

「我々は、単一のニューラル機械翻訳（NMT）モデルを使用して、複数の言語どうしを翻訳する、シンプルで洗練されたソリューションを提案する。」

「共有ワードピースのボキャブラリを使用することで、多言語NMTはパラメータを増やさずに、単一のモデルを利用することができる。これは、従来の多言語NMTの提案よりも大幅に簡単なものだ。」

「我々のモデルは、訓練中に明示的には見られなかった言語ペアの間の暗黙的な橋渡しを実行することも学ぶことができた。それは、翻訳の学習とゼロ・ショット翻訳がニューラル翻訳で可能であることを示している。」

「我々のモデルには、普遍的なインターリンガ表現が存在することを示唆する分析を示し、複数の言語を混在させた時に起きる、興味深い例を示す。」

ゼロ・ショット翻訳

このアプローチの興味深い利点は、明示的な訓練データがない言語ペアの間でゼロ・ショット翻訳を実行できることである。

これを実証するために、2つの異なる言語ペア、
ポルトガル語 -> 英語と英語 -> スペイン語（モデル1）と、
英語 <-> ポルトガル語、英語 <-> スペイン語（モデル2）の4つの異なる言語ペアのデータで訓練されたモデルの、2つの多言語モデルを使う。

これらのモデルの両方が、スペイン語 -> ポルトガル語で、合理的に良質な ポルトガル語を生成できることを示す。

著者らは、これを、「私たちの知る限り、これは真の多言語ゼロ・ショット翻訳の最初のデモです。」と主張している。

意味の共通表現の存在
インターリンガの存在

モデルを複数の言語にまたがってトレーニングすることで、個々の言語レベルでのパフォーマンスが向上し、ゼロ・ショット翻訳が有効になることがわかるということが、この論文の結論なのだが、その意味は？

言語にかかわらず、ネットワークは、同じ意味を持つ文章が同じような方法で表現される何らかの共有表現を学習しているのか？
この質問に対しては、そうだという。これも、画期的。

モデルは、訓練された言語ペアを扱うのと同じ方法で、ゼロ・ショット翻訳を実行しているのか？　これについては、まだよくわからないという。

インターリンガの証拠

いくつかの訓練されたネットワークは、実際に共有表現の強いビジュアルな証拠を示す。

たとえば、以下の図2は、英語 <-> JapaneseとEnglish <-> Koreanでトレーニングされた多対多モデルから作成されたものである。モデルの実際の動作を視覚化するために、意味論的に同一の言語間フレーズの74個のトリプルからなる小さなコーパスから始めた。

つまり、それぞれのトリプルには、同じ基本的な意味を持つ英語、日本語、韓国語のフレーズが含まれている。これらのトリプルをコンパイルするために、私たちは日本語と韓国語の翻訳と対になった、英語の文章のための正しい（Ground Truth）データベースを検索した。

Transformer

Transformer – 大規模言語モデルの基礎

Transformer は、2017年にGoogleが発表したアーキテクチャーです。

それは、GoogleのBERTやOpenAIのGPTといった現代の大規模言語モデルほとんど全ての基礎になっています。BERTの最後の文字 'T' も、GPTの'T'も"Transformer" アーキテクチャーを採用していることを表しています。

Attentionこそすべて

Googleの論文のタイトルは、"Attention is all you need" というものでした。

「Attentionこそすべて」ということで、Attention Mechanism さえあれば、RNNもCNNもいらないと思い切ったことをいっています。

これは、大規模言語モデル成立への大きな飛躍だったと思います。

Attention Is All You Need

Ashish Vaswani et al.

https://papers.nips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

2017年

Google ニューラル機械翻訳から継承したもの

まず最初に確認したいのは、見かけはずいぶん違って見えますが、Transformer アーキテクチャーは、大きな成功を収めた2016年のGoogle ニューラル機械翻訳のアーキテクチャーから多くを学んでいるということです。

ポイントをあげれば、Encode-Decoder アーキテクチャーの採用、EncoderとDecoderの分離、両者をつなぐAttention Mechanismの採用、等々。

こうした、Google ニューラル機械翻訳のアーキテクチャーの特徴は、そのまま、Transformerのアーキテクチャーに引き継がれています。それらの中で、Attentionこそが一番重要なのだというのが、Vaswani らの分析なのだと思います。

Attentionのパフォーマンスの向上
Multi-Head Attention

Attention Mechanismに注目すると、そこには特にパフォーマンスの面で課題があることも浮かび上がってきます。それは、シーケンシャルに実行され並列化されていません。また、Attentionが注目する二つの点の距離が離れると離れるほど計算量が増大します。

"Attention is all you need" 論文は、Attention Mechanismのパフォーマンス改善の提案なのです。"Multi Head Attention" は、まさに、そうしたものです。

もう一つ、この論文の大事な提案は、Self Attentionの重要性の指摘です。

Abstract

現在優勢なシーケンス変換モデルは、エンコーダーとデコーダーを含む、複雑なリカレントまたはコンボリューション・ニューラルネットワークに基づいている。

また、最も優れた性能を持つモデルは、アテンション・メカニズムを通じてエンコーダーとデコーダーを接続している。

我々は、RNNやCNNを完全に排除し、アテンション機構のみに基づく新しいシンプルなネットワークアーキテクチャ「トランスフォーマー」を提案する。

Background

シーケンシャルな計算を減らすという目標は、Extended Neural GPU [20], ByteNet [15], ConvS2S [8] の基盤にもなっている。これらのモデルは、基本構成要素としてCNNを用い、すべての入出力位置に対して並行して隠れ層の表現を計算する。

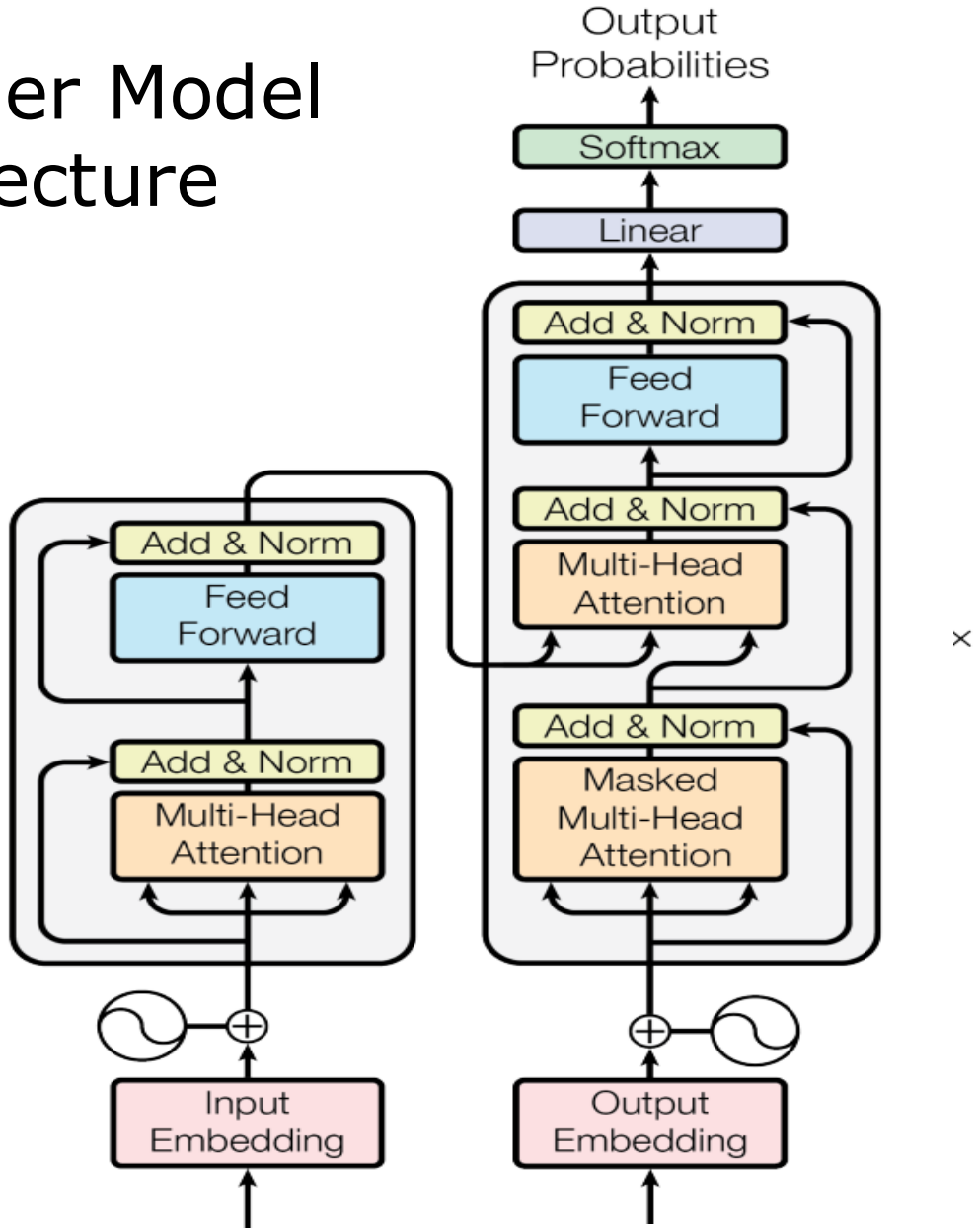
これらのモデルでは、任意の2つの入出力位置からの信号を関連付けるために必要な演算数は、位置間の距離に応じて、ConvS2Sでは線形に、ByteNetでは対数的に増加する。このため、離れた位置間の依存関係を学習することが難しくなる[11]。

Transformerでは、計算量を一定の演算数にまで減少させることができる。確かに、Attentionで重み付けされた位置の平均化によって有効解像度は低下するのだが、この効果は3.2節で説明するようにマルチヘッドAttentionで打ち消すことができる。

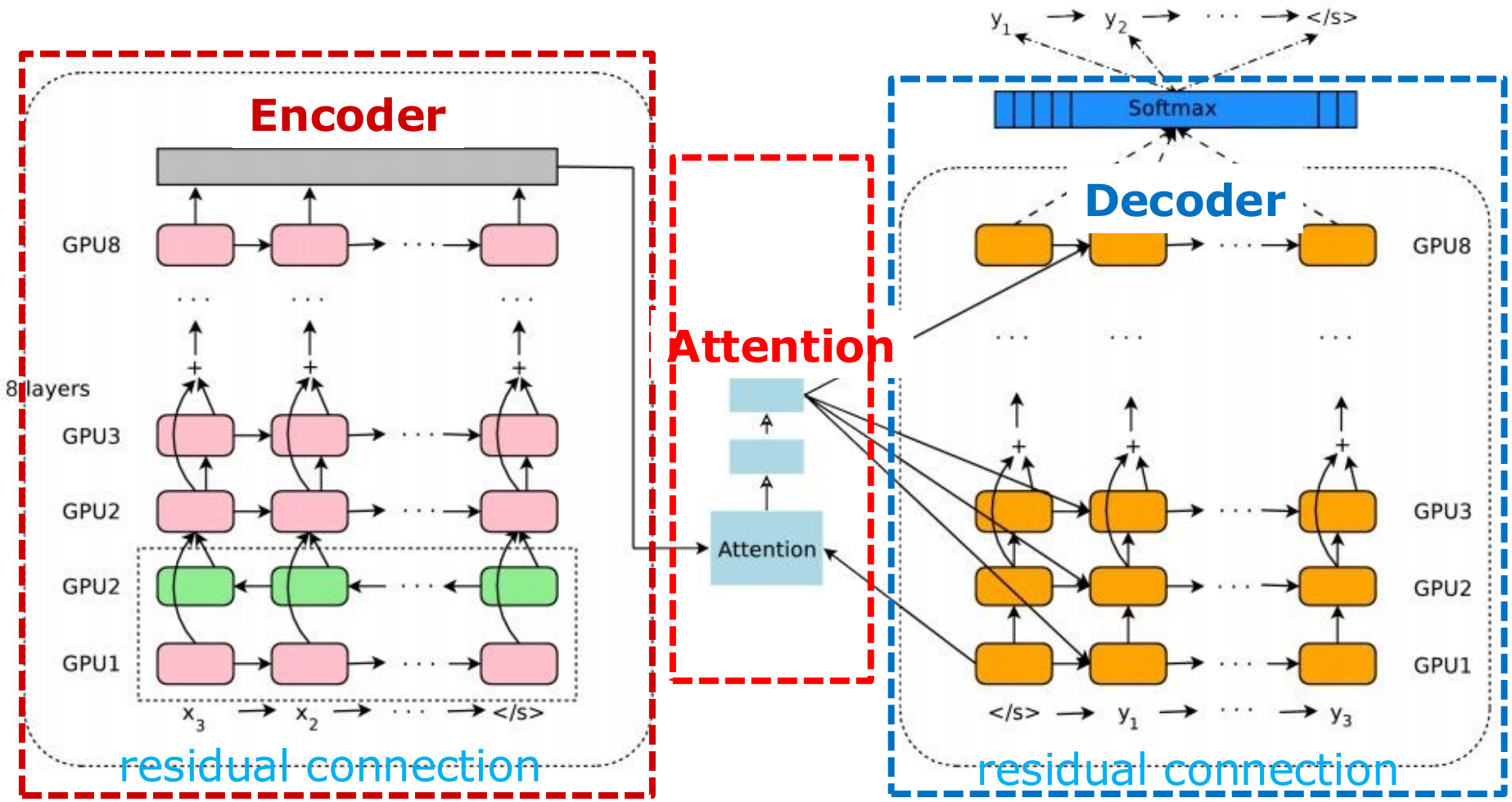
Self Attentionは、intra-attentionと呼ばれることもあるのだが、シーケンスの表現を計算するために、単一のシーケンス内部の異なる位置を関連付けるAttentionメカニズムである。

Self Attentionは、読解、抽象的要約、テキストの含意、タスクに依存しない文章表現の学習など、様々なタスクでうまく利用されている[4, 22, 23, 19].

Transformer Model Architecture

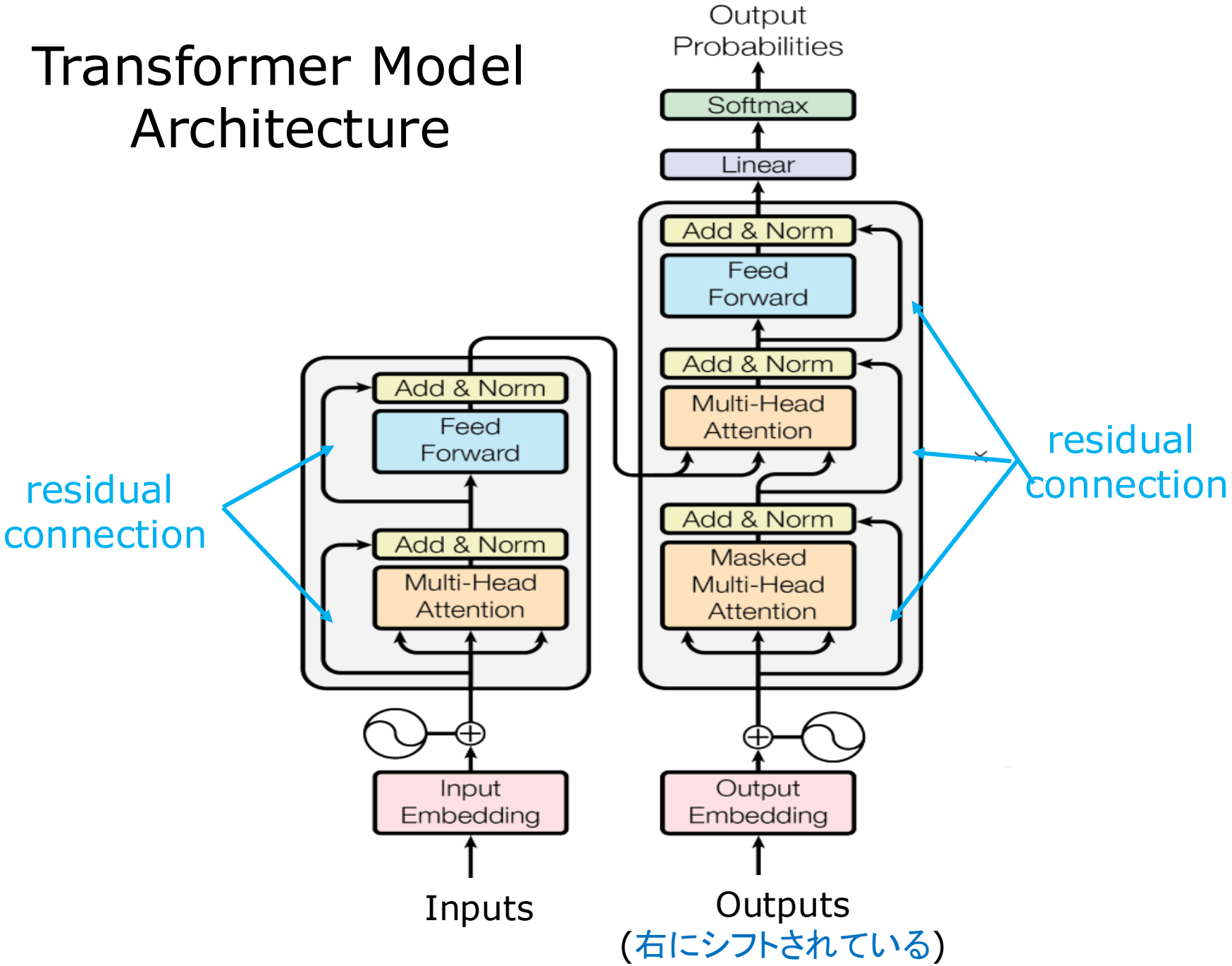


Google ニューラル機械翻訳のアーキテクチャー

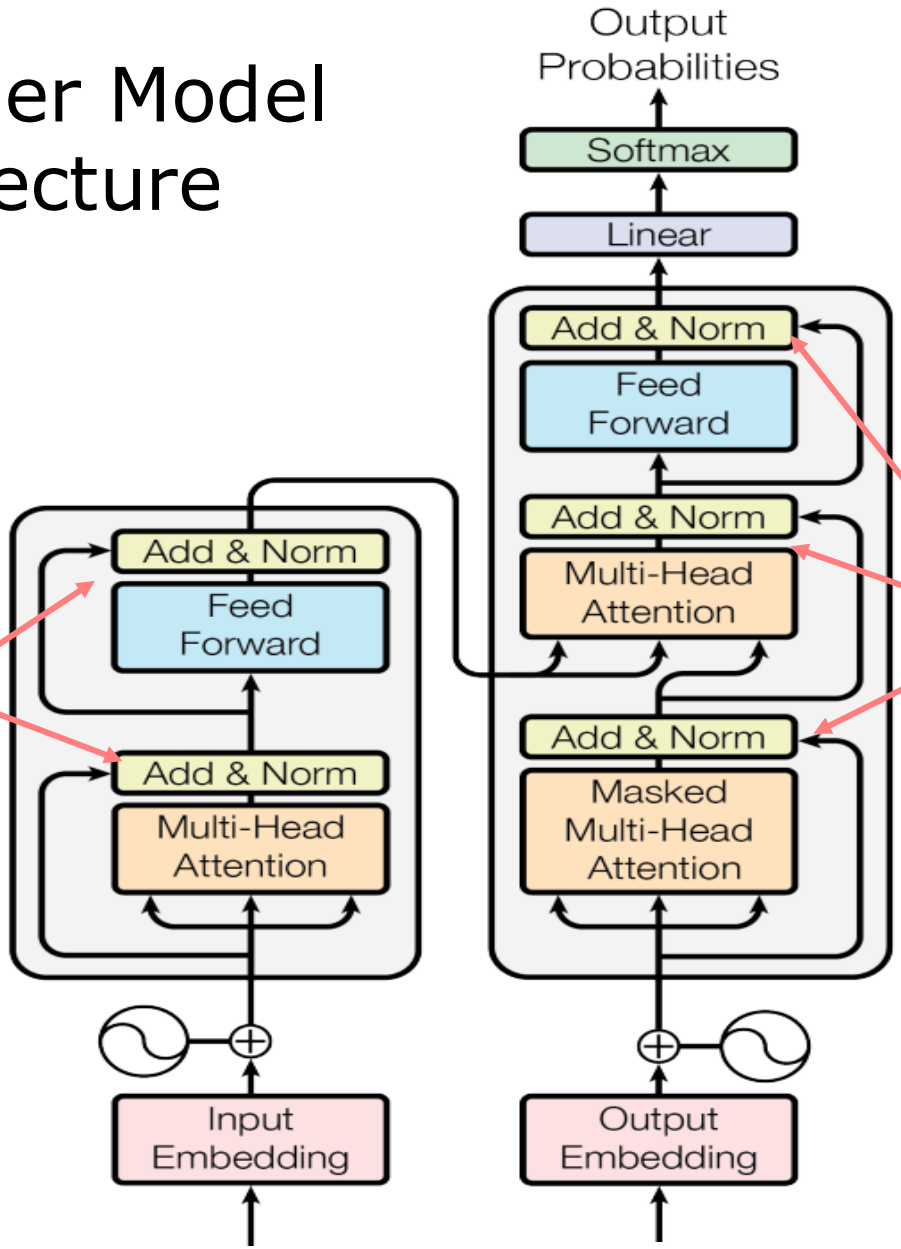


Encoder / Decoder/Attention

Transformer Model Architecture



Transformer Model Architecture

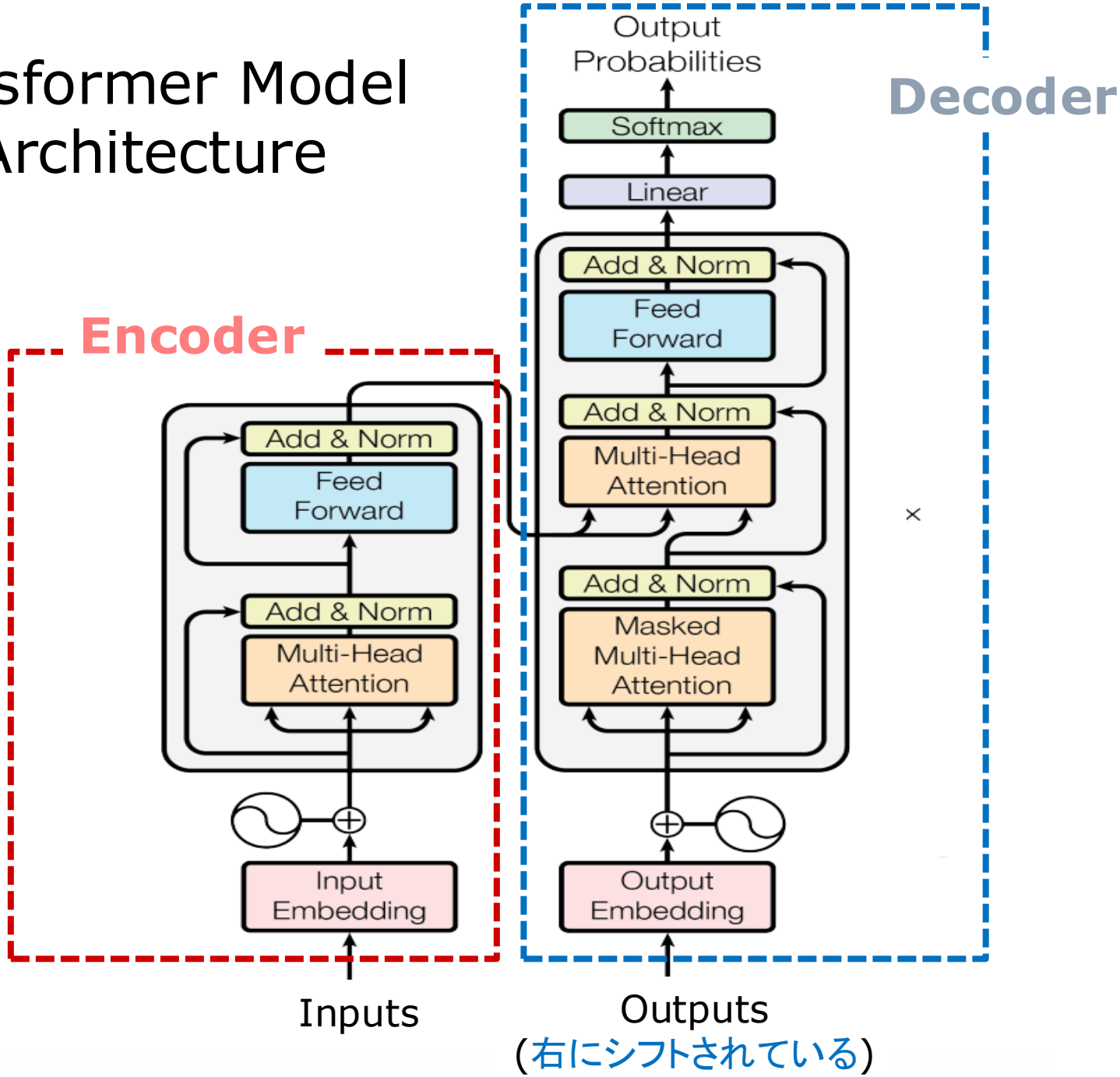


$d_{model} = 512$

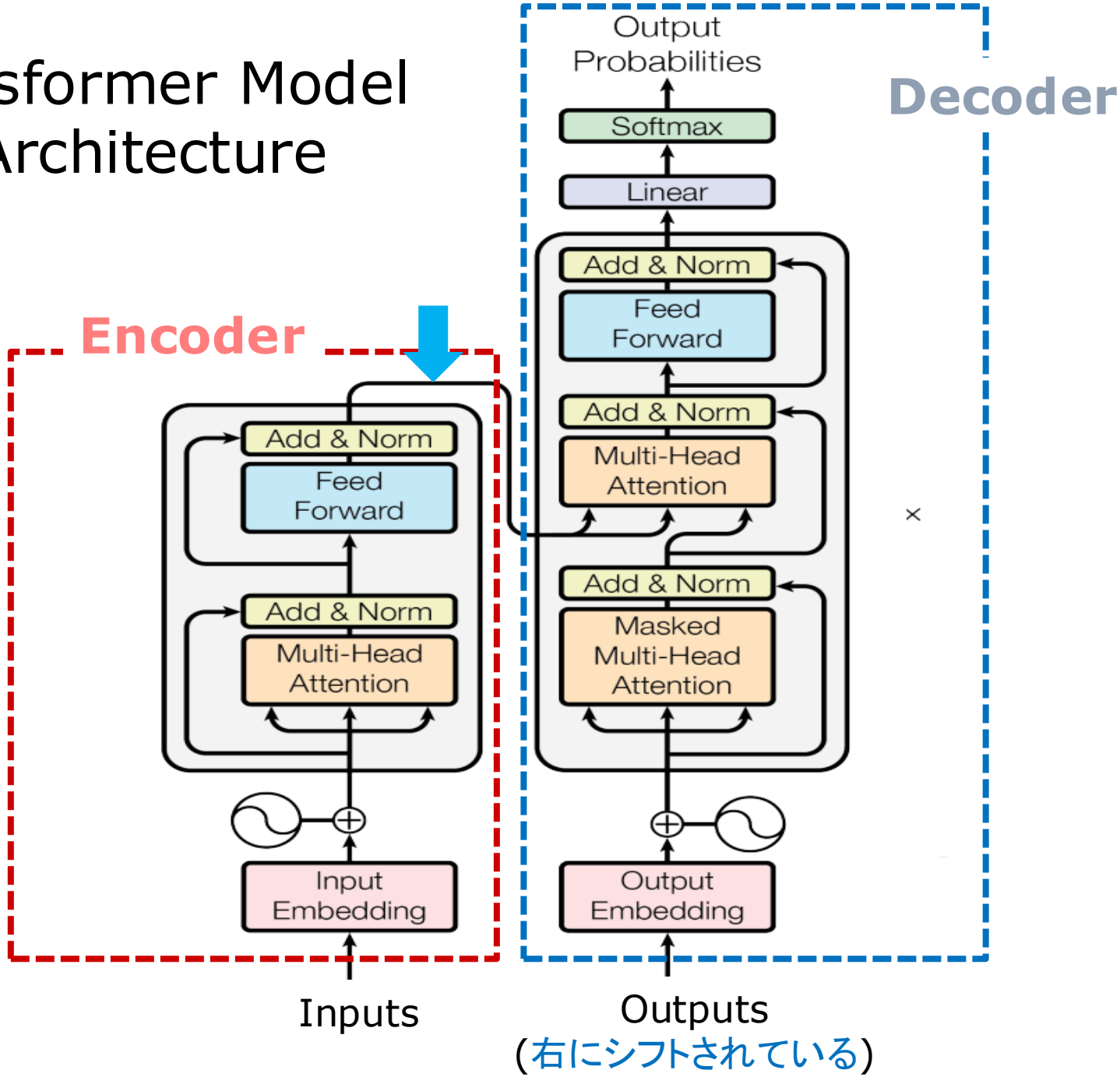
$d_{model} = 512$

(右にシフトされている)

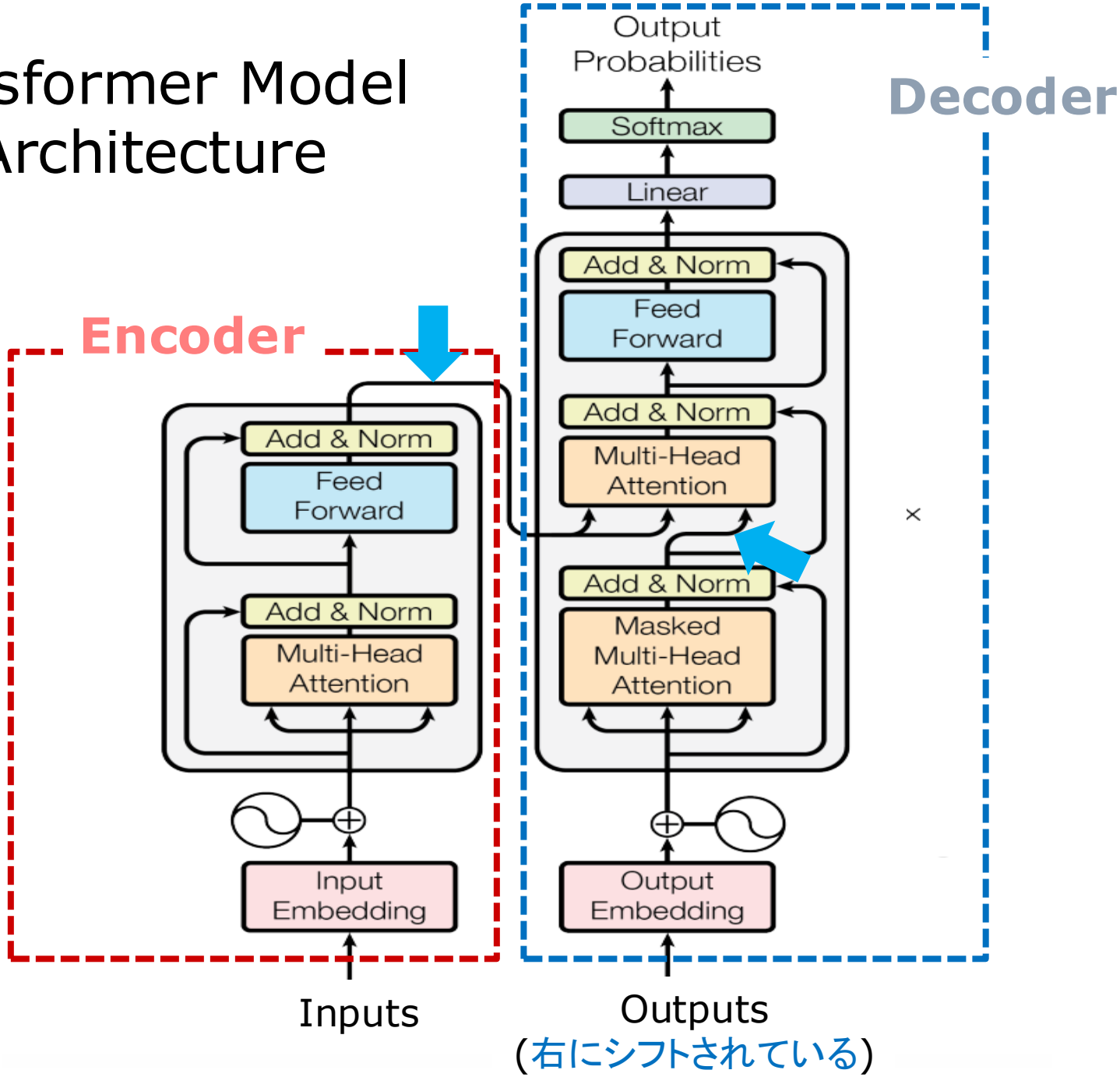
Transformer Model Architecture



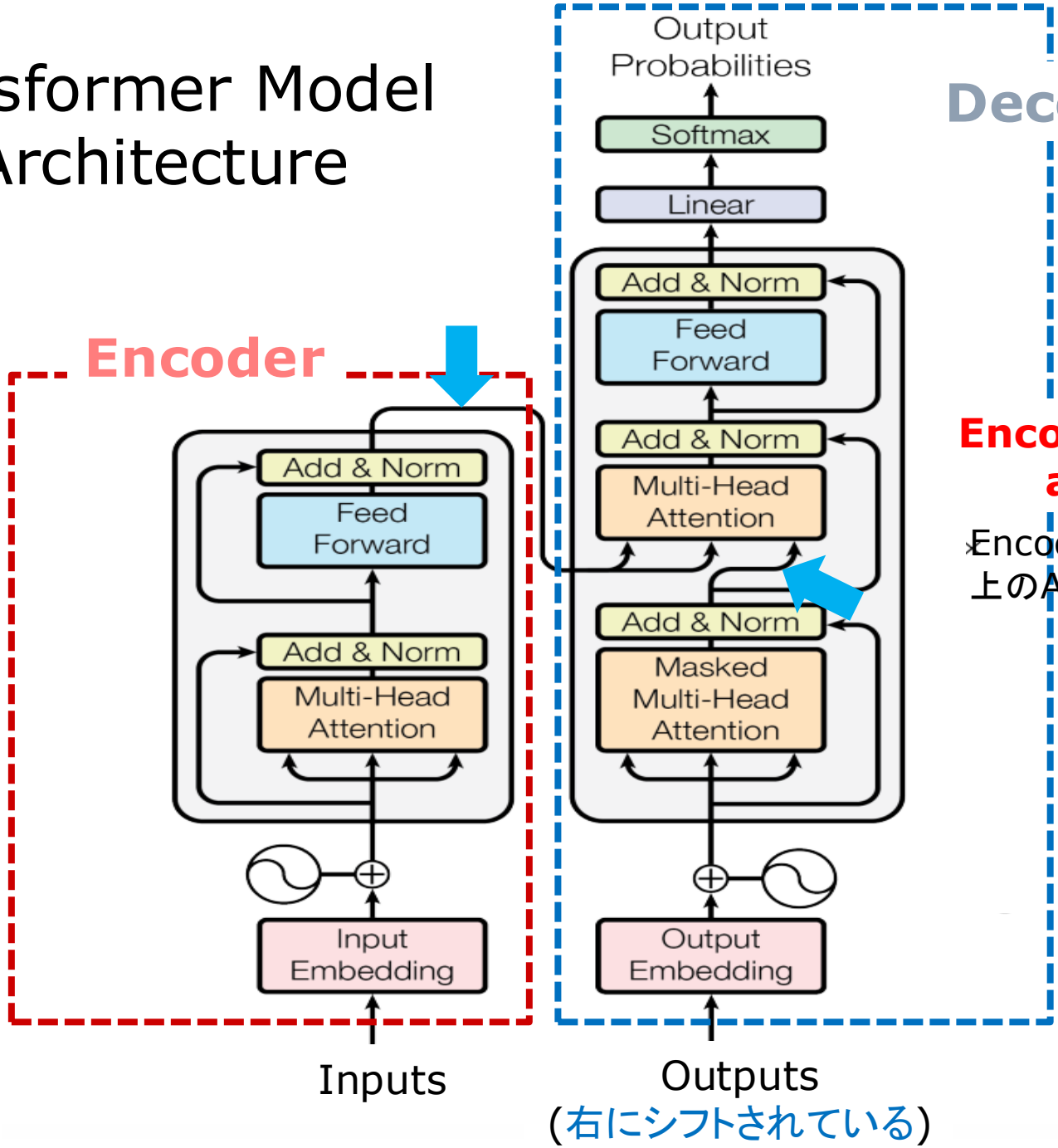
Transformer Model Architecture



Transformer Model Architecture



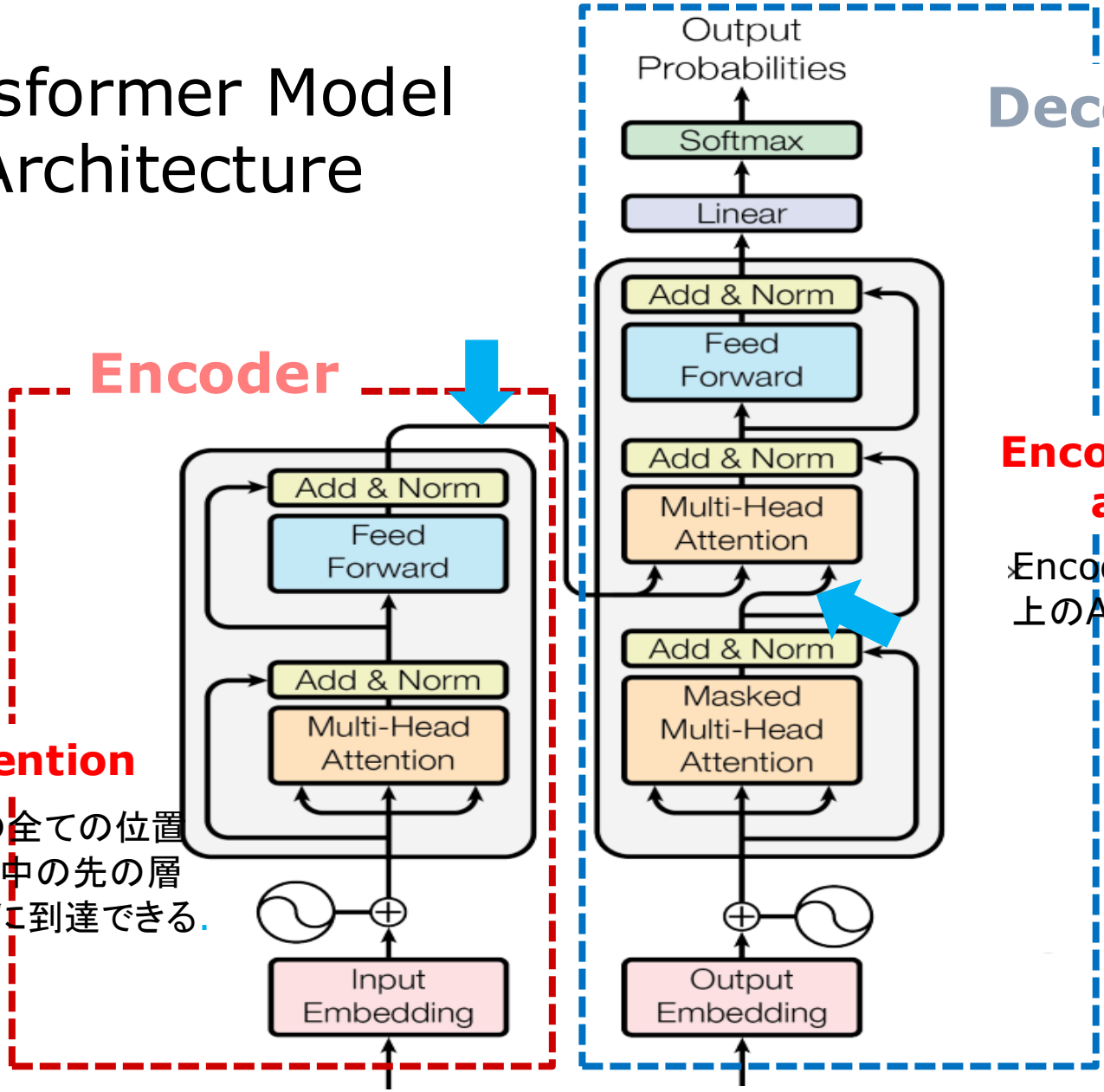
Transformer Model Architecture



Encoder-decoder attention

- Encoderスタックの出力上のAttention

Transformer Model Architecture



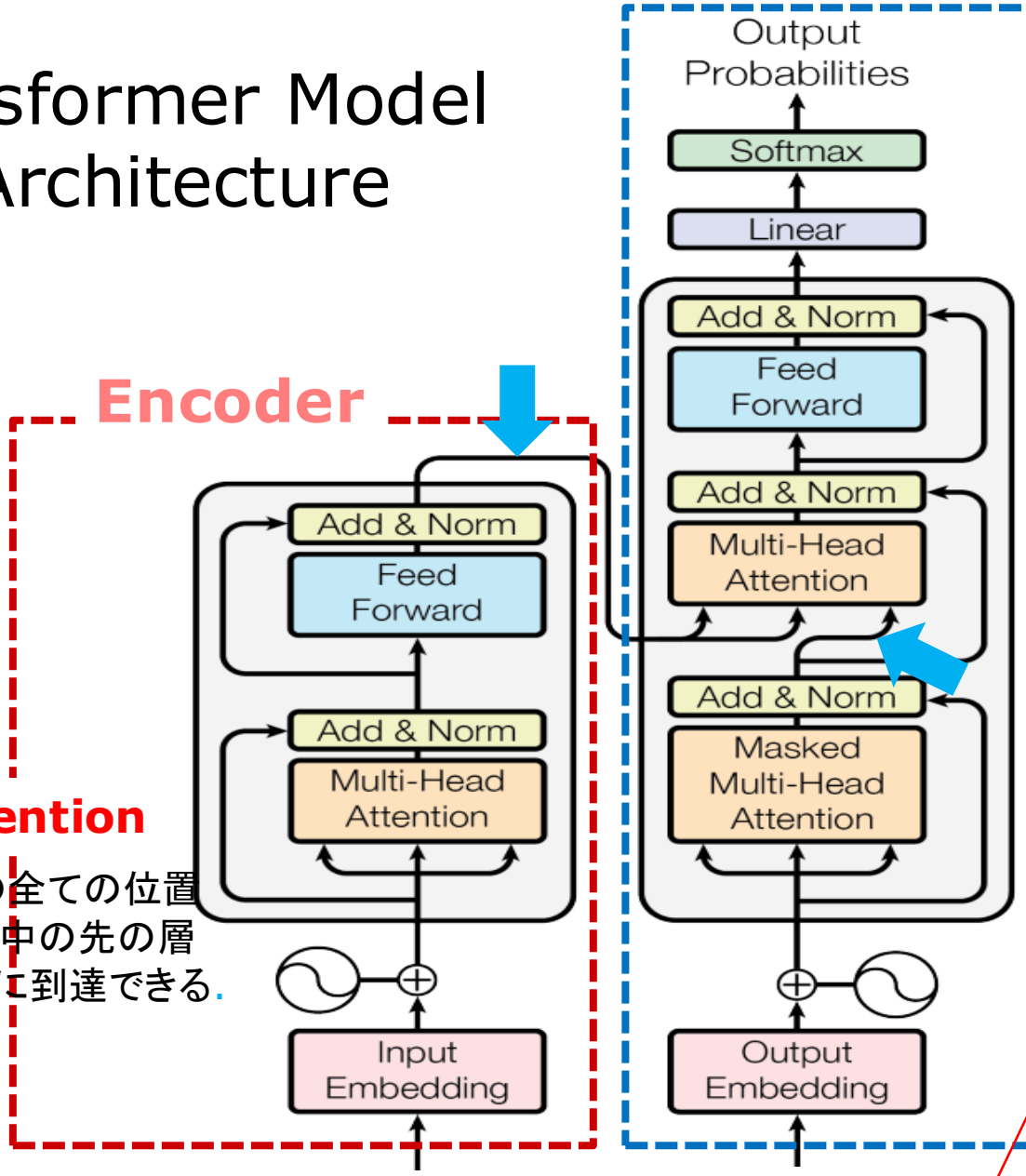
Encoder-decoder attention

Encoderスタックの出力上のAttention

Self attention

Encoder中の全ての位置は、encoder中の先の層の全ての位置に到達できる.

Transformer Model Architecture



Encoder

Decoder

Self attention

Encoder中の全ての位置は、encoder中の先の層の全ての位置に到達できる.

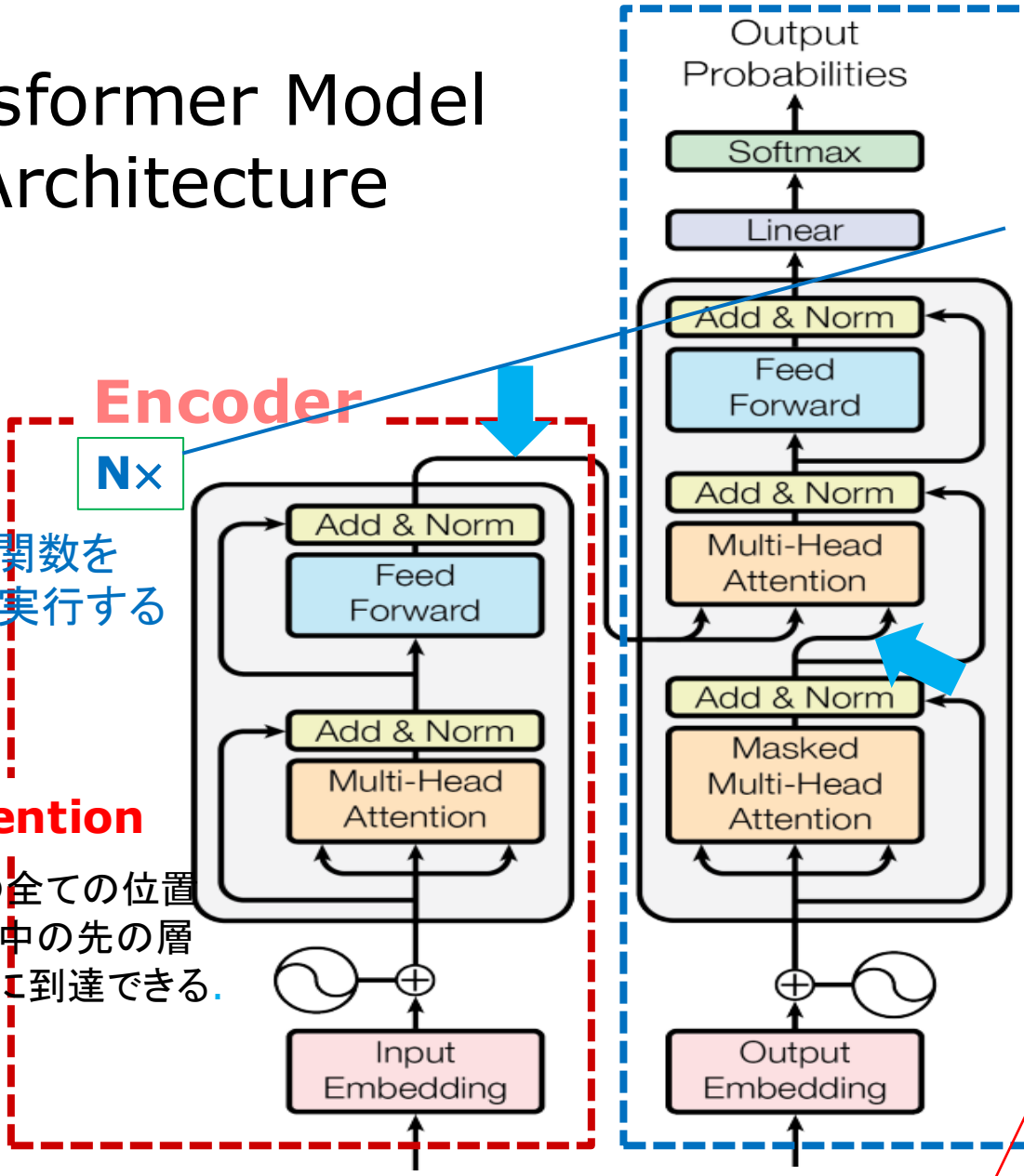
Encoder-decoder attention

Encoderスタックの出力上のAttention

Masked Self attention

Decoderの中で、情報が肥大方向に流れるのを止める必要がある。

Transformer Model Architecture



Encoder

N×

Attention関数を
N個並列に実行する

Self attention

Encoder中の全ての位置は、encoder中の先の層の全ての位置に到達できる.

Decoder

N×

Attention関数を
N個並列に実行する

Encoder-decoder attention

Encoderスタックの出力上のAttention

Masked Self attention

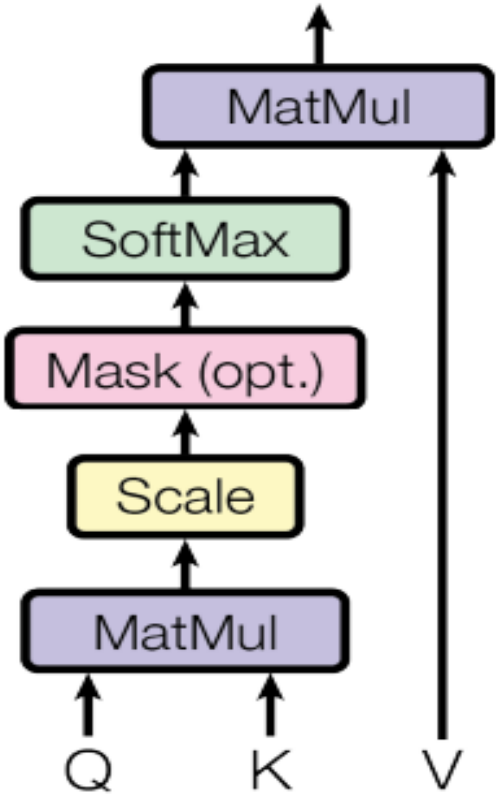
Decoderの中で、情報が左方向に流れるのを止める必要がある。

Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

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

Scaled Dot-Product Attention



Queries **Keys** **Values**

$Q \in \mathbb{R}^{d_k}$
$K \in \mathbb{R}^{d_k}$
$V \in \mathbb{R}^{d_v}$

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

Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional keys, values and queries,

we found it beneficial to linearly project the queries, keys and values h times with different, learned

linear projections to $d_k, d_k \; and \; d_v$ dimensions, respectively.

On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values,

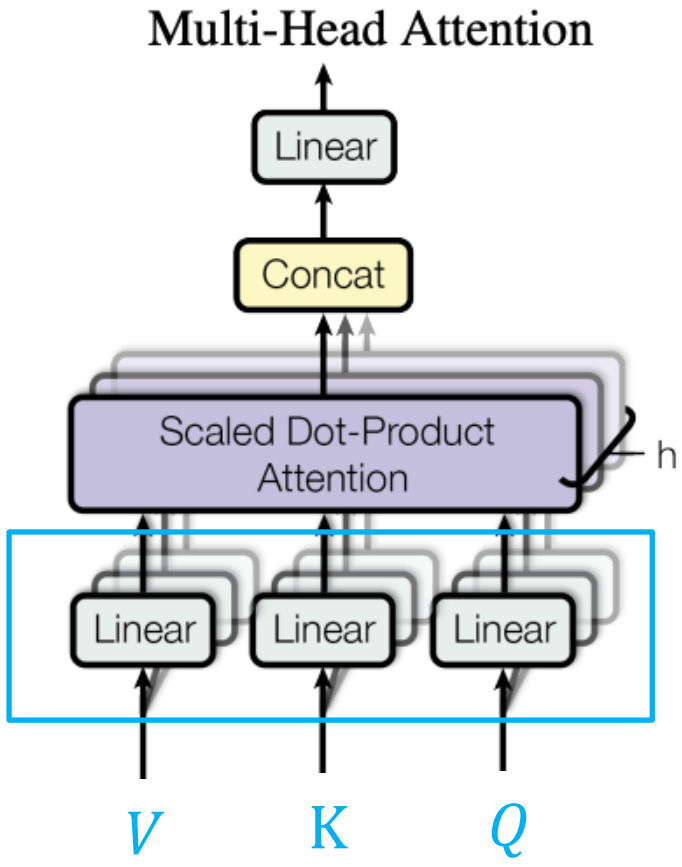
Multi-Head Attention

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$MultiHead(Q, K, V) = concat(head_1, \cdots, head_h)W^O$$
$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in R^{d_{model} \times d_k}$,$W_i^k \in R^{d_{model} \times d_k}$, $W_i^V \in R^{d_{model} \times d_v}$ and $W^O \in R^{hd_v \times d_{model}}$.

Multi-Head Attention



$d_{model} = 512$

$W^O \in \mathbb{R}^{hd_v \times d_{model}}$

$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$

$W_i^k \in \mathbb{R}^{d_{model} \times d_k}$

$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$

Projection

$d_{model} = 512$

$V \in \mathbb{R}^{d_{model}}$

$K \in \mathbb{R}^{d_{model}}$

$Q \in \mathbb{R}^{d_{model}}$

$$MultiHead(Q, K, V) = concat(head_1, \cdots, head_h)W^O$$

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

ベクトルの次元を確認する

$$V \in \mathbb{R}^{d_{model}}$$
$$K \in \mathbb{R}^{d_{model}}$$
$$Q \in \mathbb{R}^{d_{model}}$$

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$$
$$W_i^k \in \mathbb{R}^{d_{model} \times d_k}$$
$$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$$

QW_i^Q：d_{model}次元のベクトルQと$d_{model} \times d_k$ の行列W_i^Qの積
→ d_k次元のベクトル

KW_i^K：d_{model}次元のベクトルKと$d_{model} \times d_k$ の行列W_i^kの積
→ d_k次元のベクトル

VW_i^V：d_{model}次元のベクトルVと$d_{model} \times d_v$ の行列W_i^Vの積
→ d_v次元のベクトル

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

BERT

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova

https://arxiv.org/abs/1810.04805

2019年

BERT論文 タイトル

BERT論文のタイトルは、"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" というのですが、そこでの "Language Understanding" というのは、単なる「翻訳モデル」を超えた「言語理解」を目指すという方向を示唆しています。

二つの文の関係を視野に入れたことで、BERTの能力は大きく拡張可能なものになりました。

言語表現モデル

BERTは、2019年に Google が発表した「言語表現モデル」です。

ここでは、「言語表現モデル」という言い方をしていることに注意したらいいと思います。それは、直接には、Google 機械翻訳のように自然言語処理の何らかの具体的な処理を担うモデルではありません。それ自体は、言語のコンピュータ上の「表現」の構成にもっぱら関わるモデルなのです。

“BERT”の意味

BERTは、Bidirectional Encoder Representations from Transformers のイニシャルをとったものです。

そのアーキテクチャーは、前回見たTransformerの前段部のEncoder だけを独立させたようなものです。内部に、Multi headのSelf-Attention Mechanismを抱えています。

そのAttention Mechanismから「表現」を「構成」することが、BERTの「言語表現モデル」としての第一の仕事になります。

Pre-training とFine-tuning

Transformerから出力を担当するDecoderを切り取って、もっぱらEncoder内に「言語表現」を溜め込んで、どうするのでしょう？

BERTでは、膨大な計算時間をかけて、Encoder内部に「言語表現モデル」を構成するまでの、この段階をPre-training と言います。

BERTで外部に対して何かの仕事をしようとするときには、Pre-training で構成された「言語表現」をそのまま使って、具体的なタスクを実行する出力層を一枚かぶせます。これを Fine-tuning と言います。

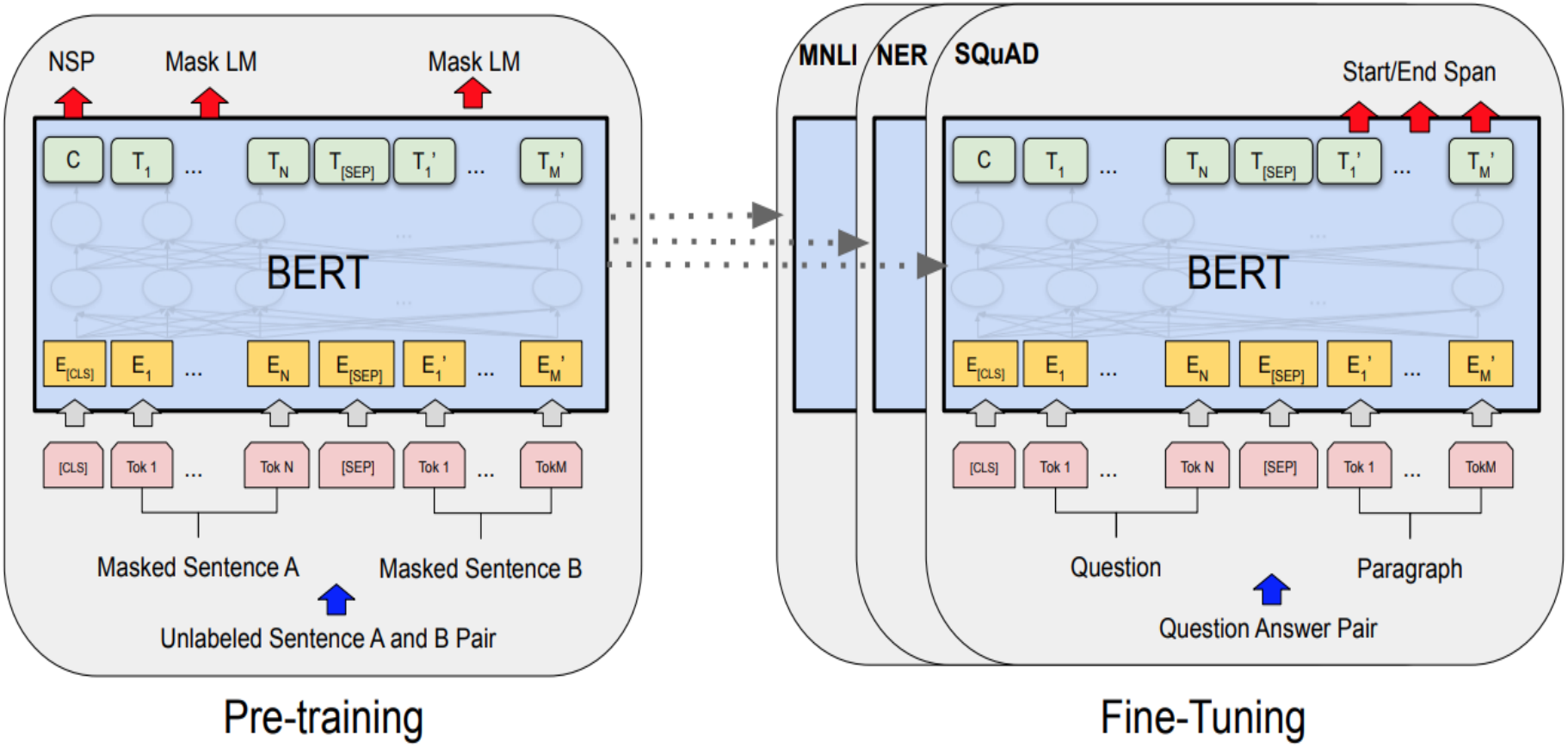
働くBERT

「働くBERT」は、Pre-training + Fine-tuning の形をしています。

ただ、タスクが変わるたびに、長い計算が必要なPre-trainingの段階を繰り返し実行する必要はありません。

「事前に訓練済み」のPre-trained された「言語表現」は、何度も何度も、タスクが変わっても「使い回す」ことができるからです。

Pre-TrainingとFine-Tuning



BERTの「双方向性」

BERTの特徴である Bidirectional 「双方向性」を簡単に説明しておきましょう。

言語の分散表現の基本は、まず、「語」に「ベクトル」を対応させることです。

この計算を実行した、Mikolov-Jeff Deanの「Word2Vec」では、ある語の周辺（前後に関わりなく）にどのような語が出現するかで、語のベクトル表現を求めていきます。(CBOW, N-GRAM)

BERTの「双方向性」

もう一つのアプローチは、ある語の「後ろ」にどのような語が出現するかに注目して、語のベクトル表現を求めることです。これをleft-to-right と呼ぶことにしましょう。

当然、ある語の「前」に、どのような語が出現するかに注目して、語のベクトル表現を見つけるやり方もあり得ます。これを、right-to-left と呼ぶことにしましょう。

BERTは、このleft-to-right とright-to-leftの双方向で語のベクトル表現を見つけます。そのために、文の中の一つの語をランダムに選んで、マスクをかけてその語を推定させる訓練を繰り返します。双方向から語の推定をしていくのは有効です。

文と文の関係

BERTにはもうひとつ際立った特徴があります。
それは、二つの文の関係を把握しようとしていることです。

具体的には、ある文Aともう一つの文Bが与えられたとき、文Aのうしろには文Bがつながるということを学習させます。

こうした文と文の関係は、語の意味の表現(Word2Vex)や文の意味の表現(Sentence2Sentence)のような、個別の語、個別の文の意味表現のレベルでは、把握することはできません。

ただ、実際の言語理解では、こうした文の繋がりが、重要な役割を果たしています。

文と文の関係

例えば、

Sentence A = The man went to the store.

Sentence B = He bought a gallon of milk.

だとすると、この二つの文は、つながっていると判断できますので、"IsNext"というラベルを出力するように訓練します。

Sentence A = The man went to the store.

Sentence B = Penguins are flightless.

だとすると、この二つの文は関連が薄いので、"NotNext" というラベルを出力するように訓練します。

BERTがPre-trainingで行うこと

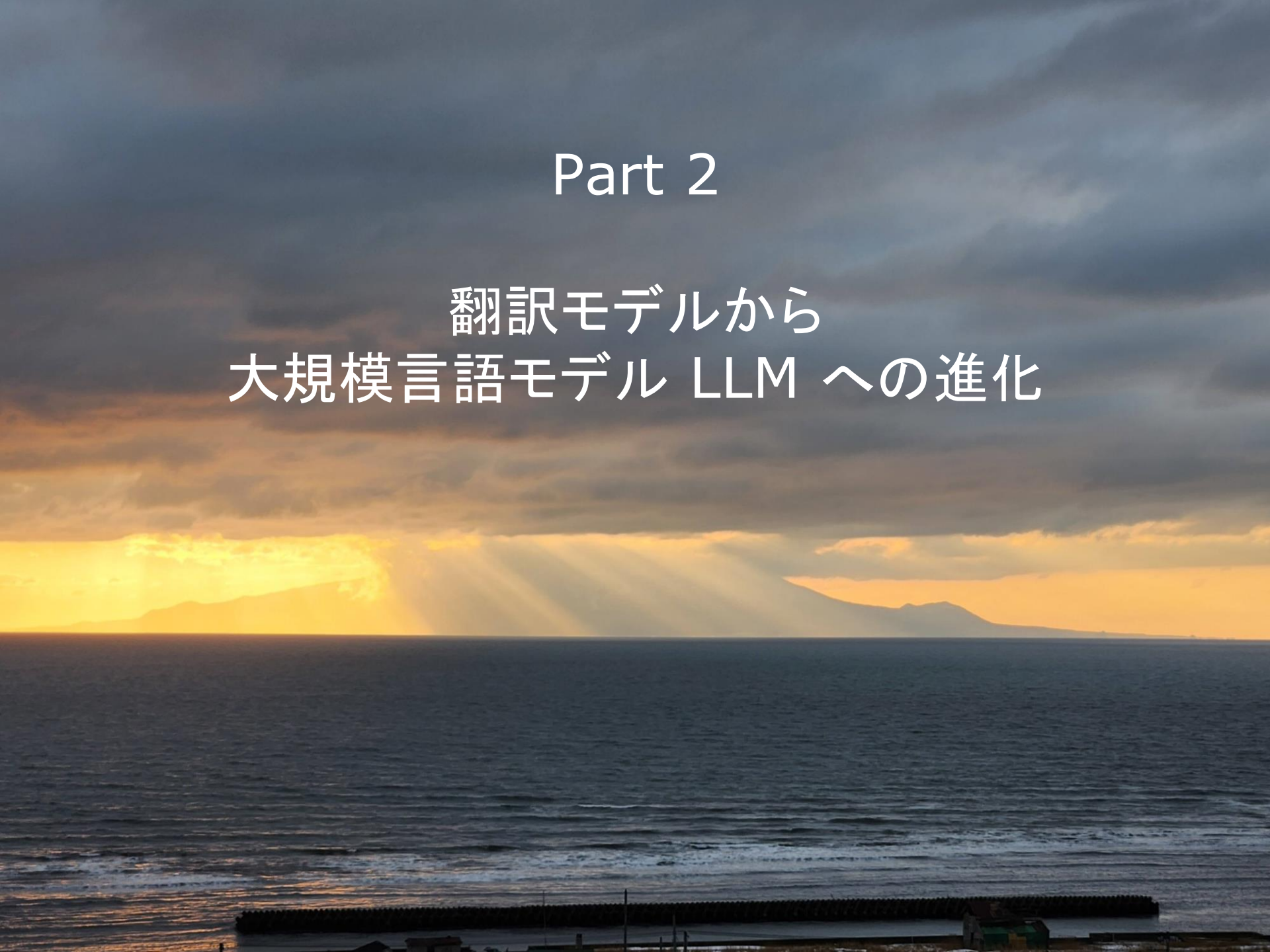
BERTのPre-trainingでは、次の二つのことを、徹底して学習します。

- 双方向での語の意味の表現
- 二つの文が与えられたとき、それが関連しているかの判断

Fine-tuning Taskの例

もっとも、そうした具体的な課題での「言語理解」のタスクは、Fine-tuningの層で実行されることになります。スライドでは、いくつかのFine-tuning層で実行されるタスクの例を挙げておきました。

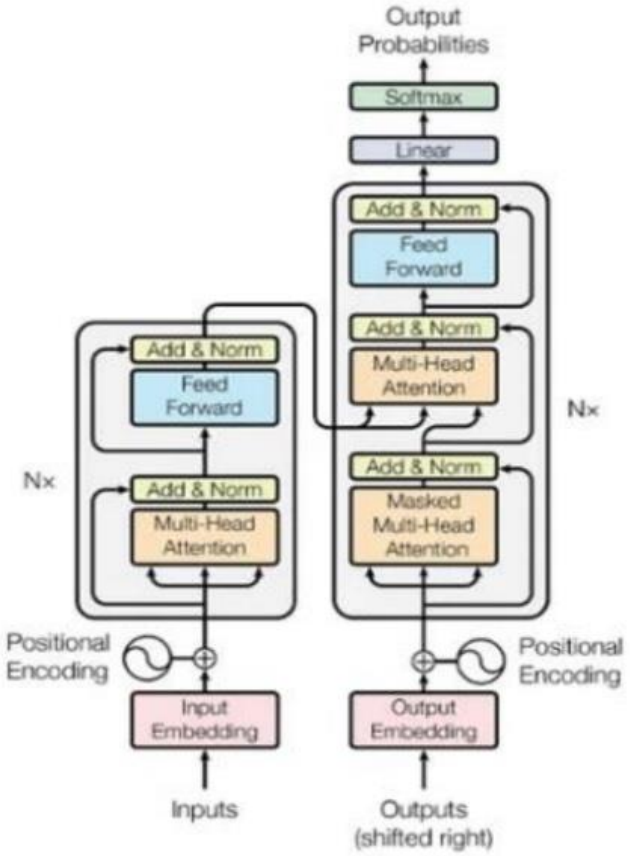
- **GLUE**: The General Language Understanding Evaluation
- **SQuAD v1.1**: The Stanford Question Answering Dataset
- **SWAG**: Situations With Adversarial Generations
- **MNLI**: Multi-Genre Natural Language Inference
- **QNL**I: Question Natural Language Inference



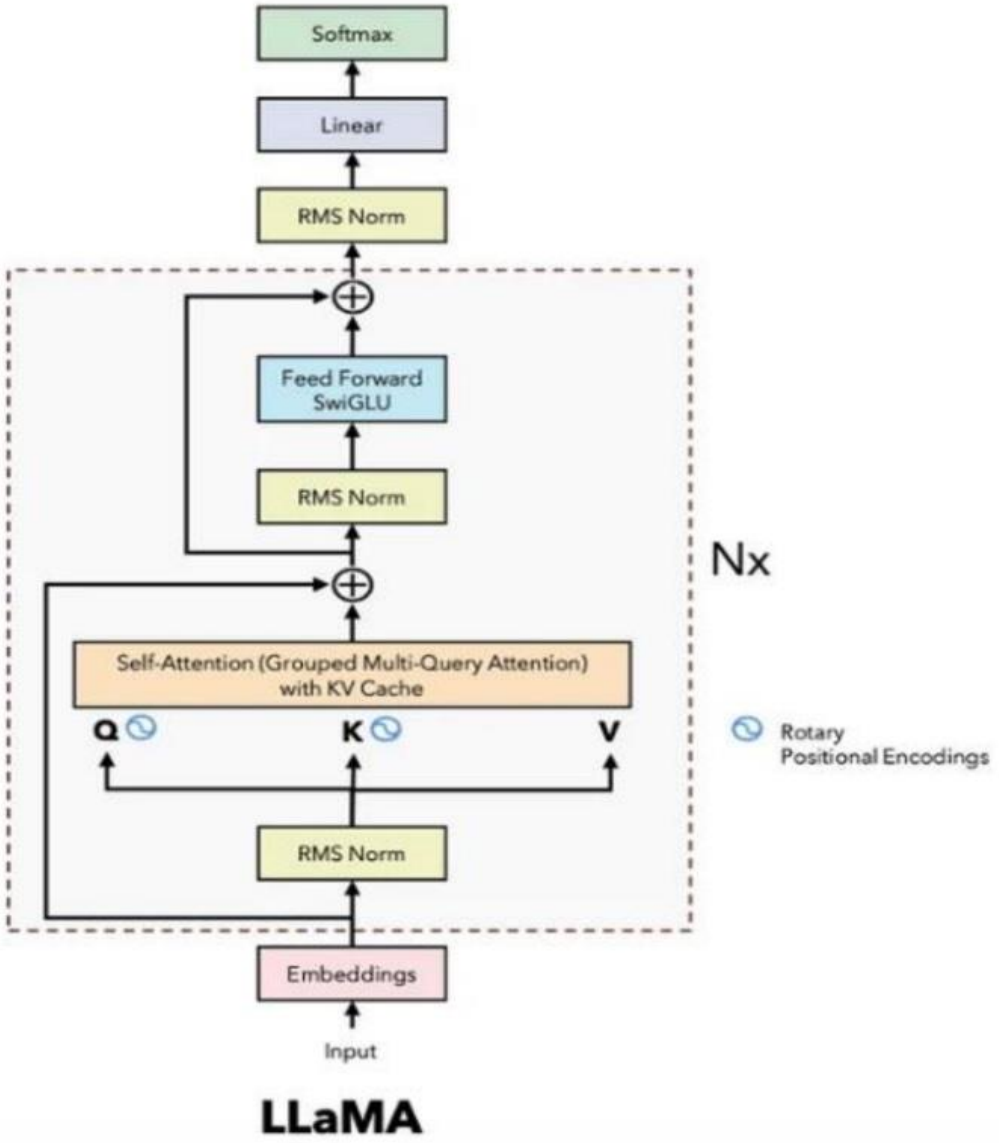
Part 2

翻訳モデルから
大規模言語モデル LLM への進化

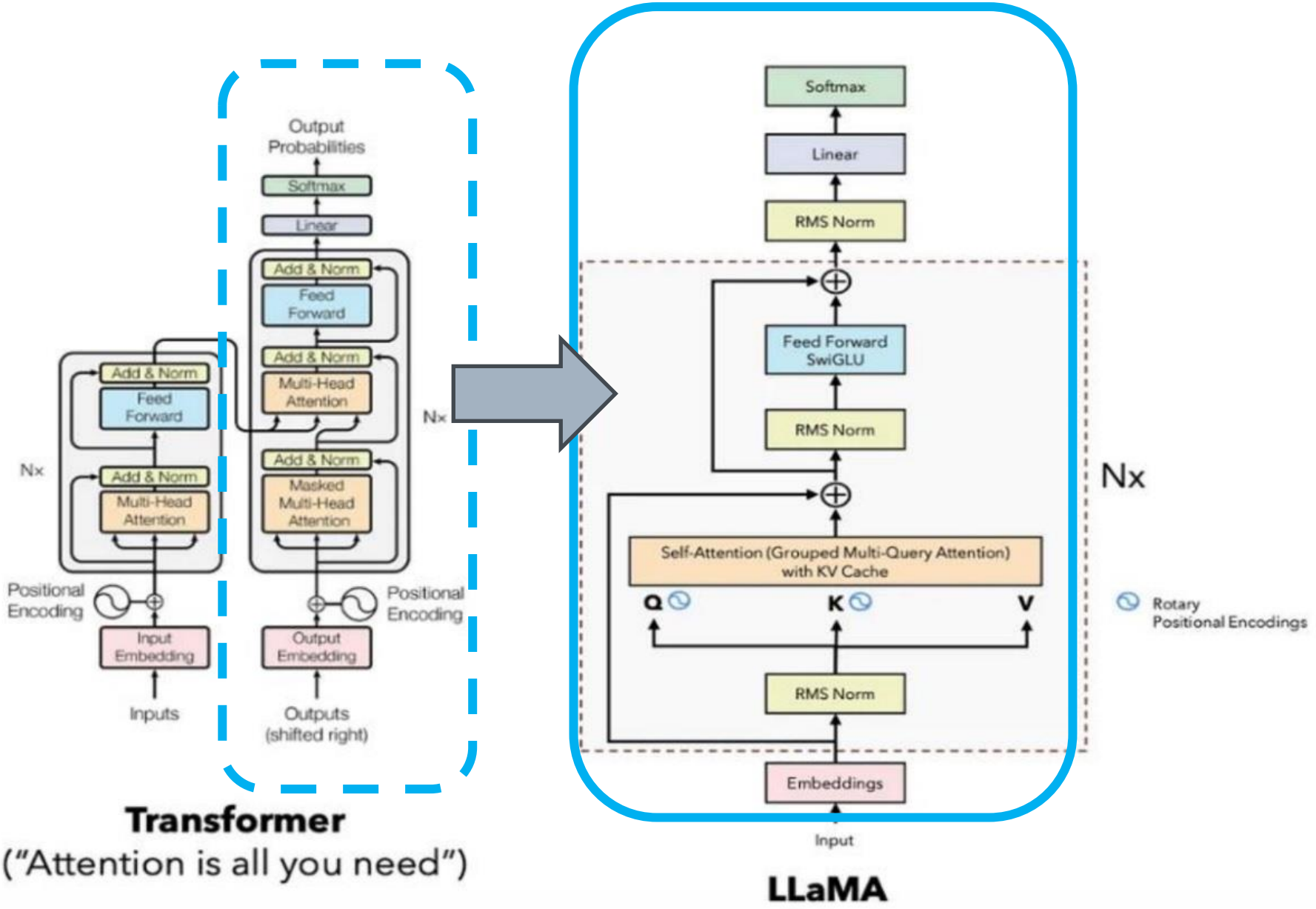
大規模言語モデルへのアーキテクチャーの変化 概要



Transformer
("Attention is all you need")



LLaMA


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
Transformer
("Attention is all you need")
Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
RMS Norm
Embeddings
Input
Nx
Rotary Positional Encodings
LLaMA

大規模言語モデルへの
アーキテクチャーの変化 概要

Transformerの登場と成功は、AI技術と意味の分散表現論の大きな飛躍でした。

このPart 2では、AI技術と意味の分散表現論のさらに大きな飛躍、翻訳モデルから大規模言語モデルへの移行という現代のAI技術に直接つながる重要な変化を取り上げます。

非常にドラスティックな変化が進行します。ここでは、その流れの概略を見ておこうと思います。

TransformerからBERTとGPTへ

翻訳モデルから大規模言語モデルへの進化の過程において、Transformerの影響は決定的なものでした。

大規模言語モデルへの進化において大きな役割を果たした、Post Transformer の代表的な二つのアーキテクチャー BERTとGPTの末尾の‘T’ がTransfomerの‘T’ であることは、その影響の大きさを表しています。

BERT：Bidirectional Encoder Representations from Transformers

GPT　：Generative pre-trained transformer

BERT

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova

https://arxiv.org/abs/1810.04805

11 Oct 2018

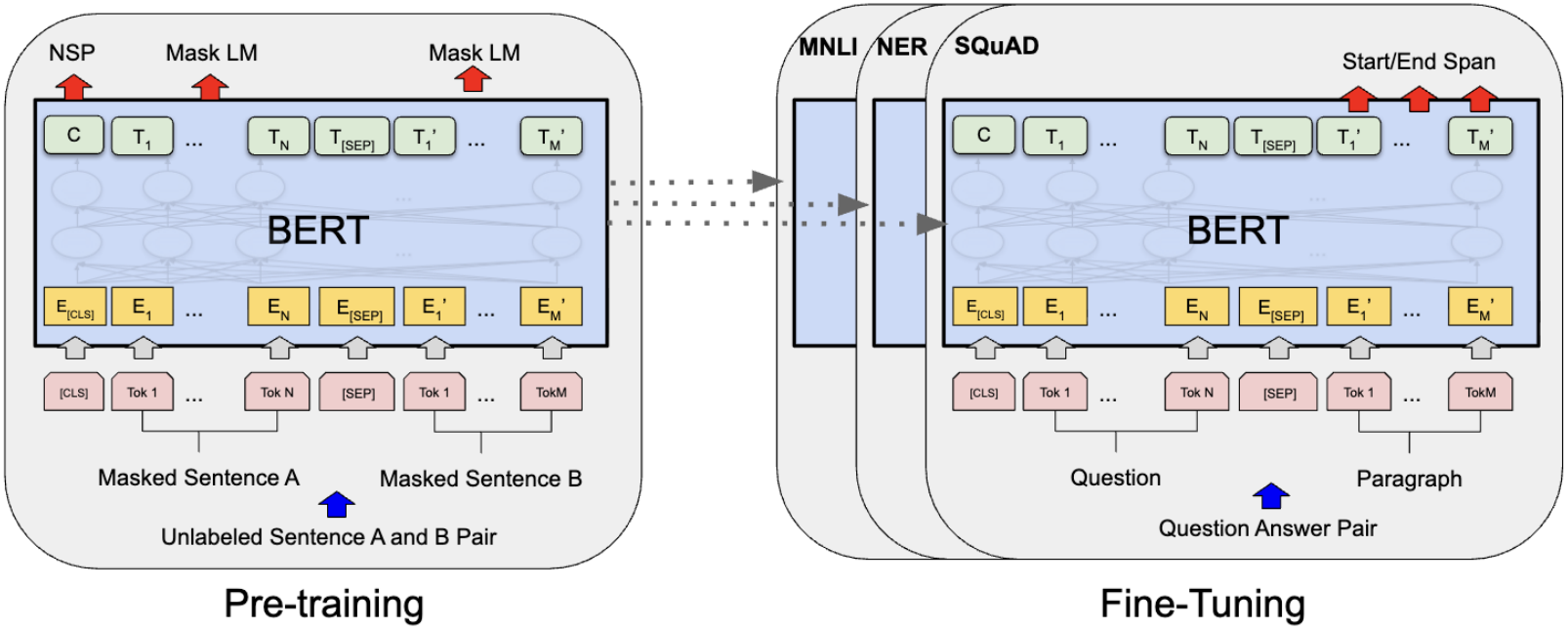


図1: BERTの全体的なPre-trainingとFine-tuning手順。
出力層を除き、Pre-trainingとFine-tuningの両方で同じアーキテクチャが使用される。異なる下流タスク向けのモデル初期化には、同じ事前学習済みモデルパラメータが使用される。Fine-tuning中は、全てのパラメータがFine-tuningされる。[CLS]は入力例ごとに先頭に追加される特殊な記号であり、[SEP]は質問と回答を区切るなどの役割を持つ特殊な区切りトークンである。

GPT

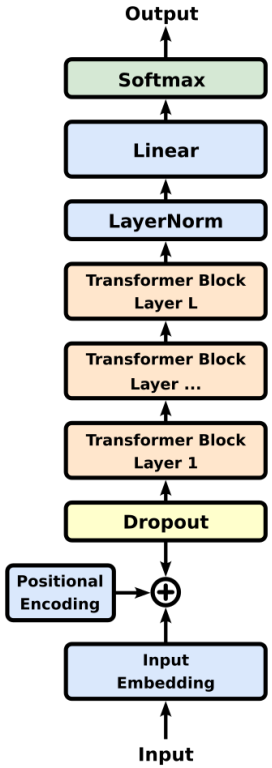
Improving Language Understanding by Generative Pre-Training

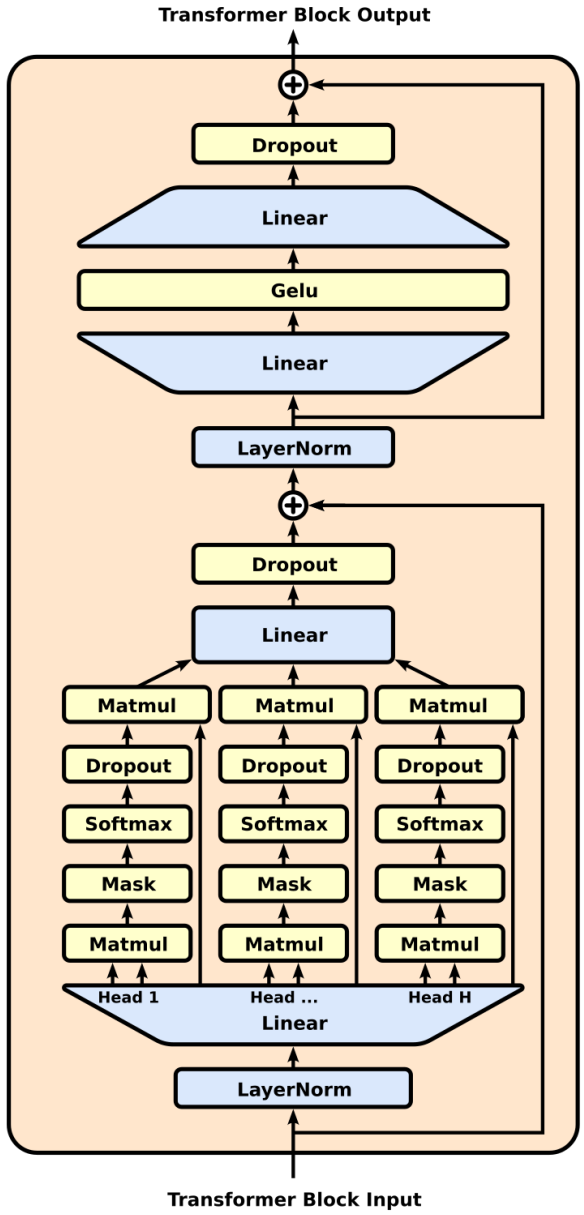
Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever

https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

2018

Original GPT model





https://en.wikipedia.org/wiki/Generative_pre-trained_transformer

TransfomerのEncoderとDecoderの分離とその継承

ただ、Transformerの達成した成果をどのように継承するのかという点で、BERTとGPTのとったアプローチは真逆と言っていい対照的なものでした。

両者は、Transformerの二つの基本的な構成要素 EncoderとDecoderを分離し、その一方だけを継承したのです。

BERT：TransformerからEncoderのみを継承。
Encoder-only アーキテクチャー

GPT　：TransformerからDecoderのみを継承。
Decoder-only アーキテクチャー

Encoder-only, Decoder-onlyの アーキテクチャーの特徴

Transformerは、翻訳システムとして実装されていたのですが、BERTもGPTも、もはやかつてのようなSequence to Sequenceの翻訳システムではありませんでした。

AIから見れば「翻訳」というのは、AIが自然言語に対して行いうる可能な仕事の一つに過ぎません。翻訳モデルの解体とより一般的なAIのモデルの模索がはじまったのです。

それでは、Encoder-only, Decoder-onlyのアーキテクチャーは、どのような特徴を持っていたのでしょう？

BERT：Encoder-only アーキテクチャー
言語の意味の深い理解能力

GPT　：Decoder-only アーキテクチャー
言語の自由な生成能力

Decoder-onlyアーキテクチャーの勝利としての大規模言語モデルの成立

重要なことは、Transformerから分岐した二つのAIアーキテクチャーのうち、Decoder-onlyアーキテクチャーの勝利として、大規模言語モデルが成立したということです。

なぜ、Decoder-onlyアーキテクチャーが勝利したかについては、この概要ではなく、別のセッションで解説したいと思います。

そこでの議論は、現在の大規模言語モデルの特徴をよりよく理解するために重要な情報が含まれています。

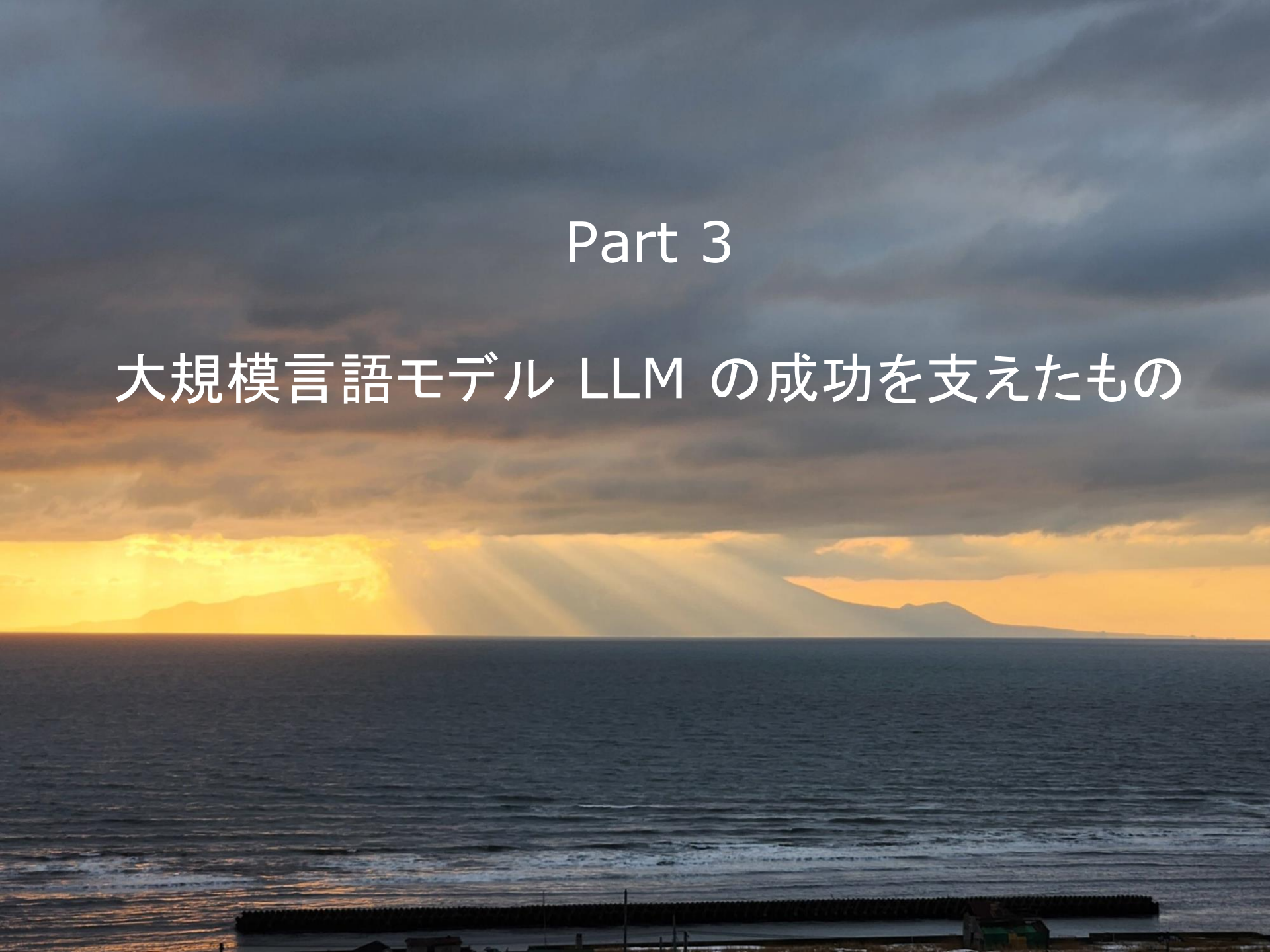
補足情報

概要とはいえ少し情報が少ないので、次の二つの音声概要にアクセスしていただけますか？

- 「Transformerのアーキテクチャー」
- 「LLMはなぜ翻訳モデルから生まれたのか」

このほかに、まだ、整理されていないのですが（ごめんなさい）、
いろんな切り口で音声概要を作ってみました。興味がありましたら、
こちらもご利用ください。

- 「EncoderとDecoderの分離からDecoder OnlyのLLMへ」



Part 3

大規模言語モデル LLM の成功を支えたもの

大規模言語モデル LLM の成功を支えたもの 1

Next token Prediction

LLM アーキテクチャーの成功を支えたもの
Next token Prediction

$$x =\perp x \rightarrow y_1 =\perp x a_1$$
確率 $p(a_1|x)$

$$y_1 =\perp x a_1 \rightarrow y_2 =\perp x a_1 a_2$$
確率 $p(a_2|\, y_1\,)$

$$y_2 =\perp x a_1 a_2 \rightarrow y_3 =\perp x a_1 a_2 a_3$$
確率 $p(a_3|\, y_2\,)$

$$y_3 =\perp x a_1 a_2 a_3 \rightarrow y_4 =\perp x a_1 a_2 a_3 a_4$$
確率 $p(a_4|\, y_3\,)$

$$y_4 =\perp x a_1 a_2 a_3 a_4 \rightarrow y_5 =\perp x a_1 a_2 a_3 a_4 a_5$$
確率 $p(a_5|\, y_4\,)$

$$y_5 =\perp x a_1 a_2 a_3 a_4 a_5 \rightarrow y_6 =\perp x a_1 a_2 a_3 a_4 a_5 a_6$$
確率 $p(a_6|\, y_5\,)$

テキストへの1-トークンの追加とNext Token 確率

振り返り -- LLM アーキテクチャーの成立

先のセッション「大規模言語モデルへのアーキテクチャーの変化概要」では、革命的なTransformerアーキテクチャーの登場を引き金として起きた大きな変化を見てきました。

Transformerのアーキテクチャーから何を継承・発展させるかで、「翻訳モデル」が中心だったAIのアーキテクチャーに大きな分岐が起きました。

一方は、TransformerからEncoderを継承し、他方はTransformerからDecoderを継承しようとしました。これら二つのAIアーキテクチャーは、"Encoder−only" あるいは "Decoder−only" と呼ばれていました。

結果的に大きな成功を収めたのは、"Decoder−only" と呼ばれた流れでした。これが、現在の「大規模言語モデル LLM」です。こうして、AIのアーキテクチャーは、「翻訳モデル」から「大規模言語モデル」へと大きくな転換したのです。

今日では、GPTファミリーはもとより、Gemini も Claude も LLamaもすべて、Decoder−only のLLMアーキテクチャーを採用しています。

LLMアーキテクチャーの成功を支えたもの

このセッションでは、LLMアーキテクチャーの成功を支えた、技術的な優位性はなんだったのかを、まずは、次のような視点から見ていきたいと思います。

- システムの目的設定のシンプルさ
  Next token prediction
- 大量のテキストから学習する能力
  Self-Supervised Learning
- プロンプトを利用した柔軟なタスクの習得
  In-Context Learning

まだまだたくさんありますね。ある意味、いいことづくめにも見えます。切り口によって見えてくるものが変わります。

プロンプト導入によるLLMの成功のベースにあるのは、LLMの基本的な性格に遡って考えれば、LLMアーキテクチャーが持つ「実行可能なタスクの一般性　Universality」です。

同じように考えれば、LLMの推論の効率性には、「推論の因果性 Casuality」が関係しています。ずっと技術寄りなトピックになりますが「KVキャッシュによる推論の効率性」は、「Causal LLM」の特徴のコロラリーです。

LLMの成功の要因として、技術的な優位性そのものではないことが挙げられることもあります。

例えば、「Scaling則によるパフォーマンスの予測可能性」とかは、技術についてのメタな議論だと思います。まあ、それはそれで意味があるのですが。

ここでは、最初に挙げた三つの点について話してみようと思います。

Next token prediction

LLMの振る舞いのシンプルさは、「LLMは、ある文字列トークンの並びが与えられたとき、そのトークンの並びの次に来るトークンを予測して、それを選ぶ」という表現に集約されると思います。

LLMの振る舞いは、決してシンプルなものには見えません。ある場合にはリクエストに応じて長いレポートを書き、ある場合には人間と長い会話をし、ある場合にはプログラムや絵だって書いてくれます。

驚くべきは、こうしたLLMの複雑な振る舞いが、全て、先に述べたNext token predictionという極めてシンプルなメカニズムに還元されるということです。にわかには信じられないと思います。

ただ、それは事実なのです。そのシンプルさが、我々が今やその全体像を見渡すことができないほど巨大なシステムにLLMが成長することを可能にしました。巨大になっても、その心臓部を動かしているのは、「次のトークンを予測する」という単純なメカニズムなのです。

僕たちは、「知能」や「意味理解」に対して機械が与えた、ある意味暴力的にも見える「還元主義」的理解を、どのように捉え返すべきかを考える必要があるのだと思います。

ただ、そこには、誤解も含まれています。また、何か本当は複雑なことが見過ごされていて、簡単な表面に目が入っているだけなのかもしれません。それを考えてみましょう。

LLMがNext token を「返す」とすれば、それはLLMの直接の出力ではなくて、LLMが出力した全てのtokenを渡る「確率分布」の中から、最大の確率をもつtokenを選んでそれを返すと考えるということです。「確率分布」が与えられた時、その分布から最大の確率をもつ変数を選択するのは、決定論的で自動的な過程です。

ただ、Next tokenを「返す」のと、Next tokenを「予測する」のは違うことです。「確率分布」が与えられたとしても、そのことは、何を選ぶかを決定するものではありません。

「吾輩は」というtokenが与えられた時、LLMは「猫」が最大確率をもつ「確率分布」を計算して返す可能性はあります。ただ、「猫」じゃなく、「豚」でも「海」でも「毛虫」でも選んでいいんです。その選択は、「猫」より確率が低いかもしれないというだけです。

LLMが行っていること

LLMが行っていることは、次のようなことです。

文xが与えられた時、LLMは文xの次に来るtokenの「確率分布」を計算します。その分布pからtoken a_1が選ばれたとすると、その確率は確率分布pのもとで xが与えられた時a_1が選ばれる確率 $p(a_1|x)$で表されることになります。

その後、LLMは文xの後にtoken a_1を追加して、文 xa_1を作り、今度は、文 xa_1 の次に来るtoken の確率分布p_1を「新しく」計算します。その分布p_1からtoken a_2が選ばれれば、a_2が選ばれる確率は、$p_1(a_2|xa_1)$になります。

LLMは、token を一つづつ追加して長い文を作っていくのですが、こうした過程を繰り返します。

図が表しているのは、そうした過程なのですが、手抜きをしていて確率分布を全て同じpで表しています。これは本当は正しくありません。文が長くなるたびに、確率分布は、毎回新しく計算し直されます。

確率分布 $p_x(-|x)$の生成と その分布の下でのサンプリング

LLMの働きを振り返ってみましょう。

- LLMは、テキスト x が与えられた時、次に出現するトークンを予測する確率分布 $p_x(-|x)$ を生成します。
- LLMは、一つのトークン a を選んでxに追加して、テキストを一つ分延長して xaにします。
- このとき、xの後ろに一つのトークンaが追加される確率は、$p_x(\mathrm{a}|x)$になります。

テキストへの1-トークンの追加と Next Token 確率

$$x =\perp x \rightarrow y_1 =\perp xa_1$$

確率 $p(a_1|x)$

$$y_1 =\perp xa_1 \rightarrow y_2 =\perp xa_1a_2$$

$$y_2 =\perp xa_1a_2 \rightarrow y_3 =\perp xa_1a_2a_3$$ 確率 $p(a_2|\ y_1\ )$

確率 $p(a_3|\ y_2\ )$

$$y_3 =\perp xa_1a_2a_3 \rightarrow y_4 =\perp xa_1a_2a_3a_4$$

確率 $p(a_4|\ y_3\ )$

$$y_4 =\perp xa_1a_2a_3a_4 \rightarrow y_5 =\perp xa_1a_2a_3a_4a_5$$

確率 $p(a_5|\ y_4\ )$

$$y_5 =\perp xa_1a_2a_3a_4a_5 \rightarrow y_6 =\perp xa_1a_2a_3a_4a_5a_6$$

確率 $p(a_6|\ y_5\ )$

BradleyのLLMの確率計算

実は、前回のマルレクで、Tai−Danae BradleyのMagnitude論を紹介したのですが、その前半部分で、こうしたLLMの行っている確率計算のステップを詳しく解説しています。

「LLMのマグニチュード論 1 −− LLMの確率計算とenrichedカテゴリー論」
https://www.marulabo.net/docs/llm1bradley2/

その中の、「LLMの確率計算の基本」を参照ください。

YouTube:
https://youtu.be/7KJeDC482AI?list=PLQIrJ0f9gMcMjv25F7mabNGdzKUVt-2CZ

PDF:
https://drive.google.com/file/d/182D1nhVmjk6stoKjQ2Q0VwmxB_PrgZn2/view

LLMの内部では何が計算されているのか

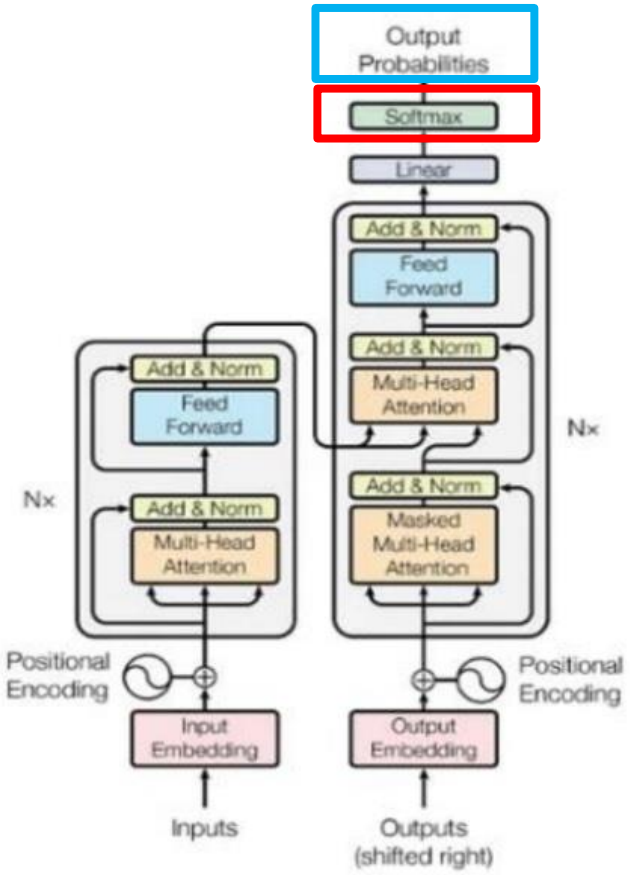
Next token predictionについては、もう一つ注意すべきことがあります。

先のNext token predictionの説明では、LLMは、文字列トークンの並びから次のトークンを予測しているように見えたと思います。それは、ある意味では正しいまとめで、LLMの振る舞いを知るにはわかりやすいのですが、ただ、その説明には、表面的なところがあります。

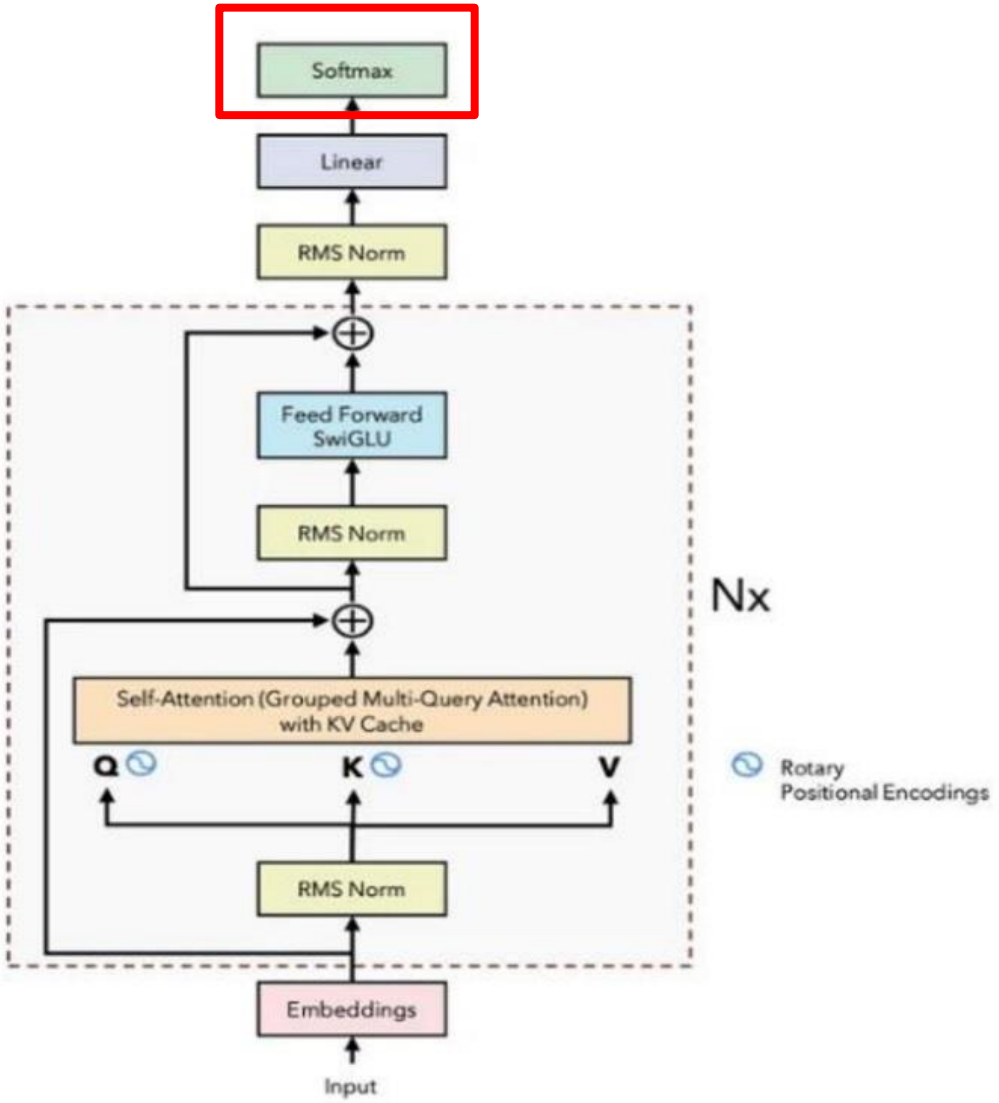
例えば、LLMで確率分布を計算しているのは、Softmax関数なのですが、Softmaxは文字列トークンを扱うことはできません。それが受け取るのは、文字列トークンではなくそのベクトル表現です。

LLMの入出力自体、それを文字列トークンの並びとイメージすることはできるのですが、実際にLLMが受け取って処理して吐き出しているのは、そのベクトル表現です。

文字列トークンのベクトル表現をembeddingと言います。



Transformer
("Attention is all you need")



LLaMA

LLMは人間とのインターフェースには、自然言語の文字列を使うのですが、LLM内部の主要なデータの形式は、文字列ではなくそのembeddingです。LLMの内部では、文字列ではなくembeddingが忙しく飛び回っているのです。

ただし、我々は直接はembeddingを解釈できないので、そのことは、表面からは見えにくいかもしれません。実際には、LLMの行う処理の複雑さは、embeddingに閉じ込められたembeddingが持つ情報の複雑さに、多くをおっています。

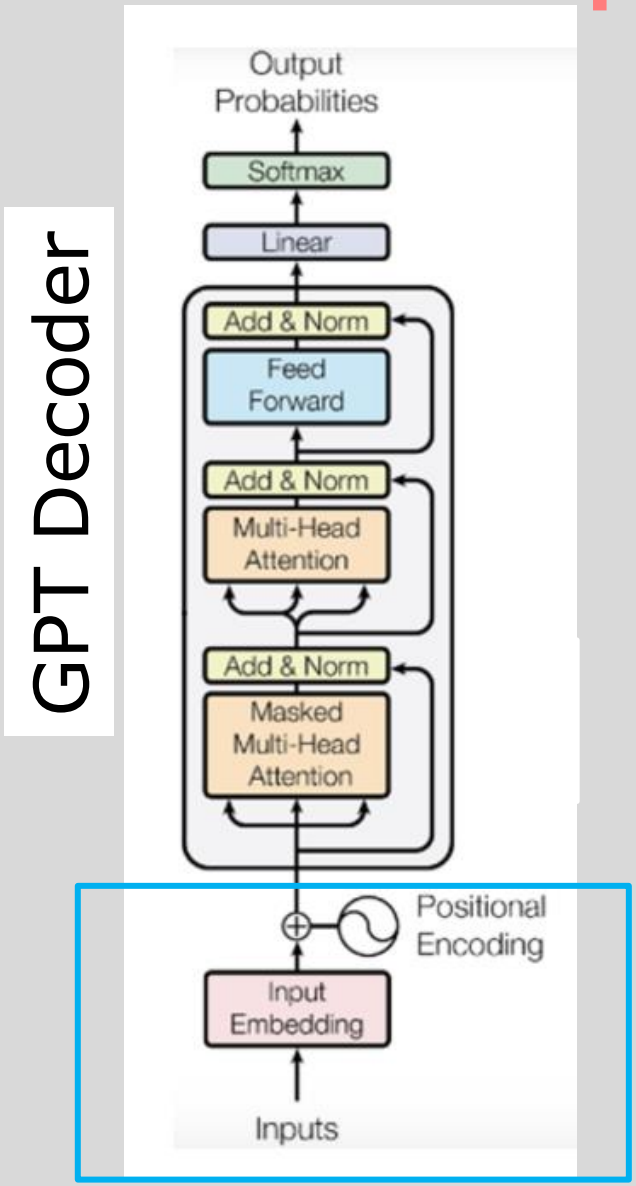
そのことは、我々がLLMの行いうる処理の複雑さを「LLMのパラメーター数」で表す時、そのパラメーター数は、LLMがすでにembeddingとしてすぐに処理可能な文字列トークンの数（正確にいうと、それにベクトルの次元数をかけたもの）であることにも表れています。

大規模言語モデル LLM の成功を支えたもの 2

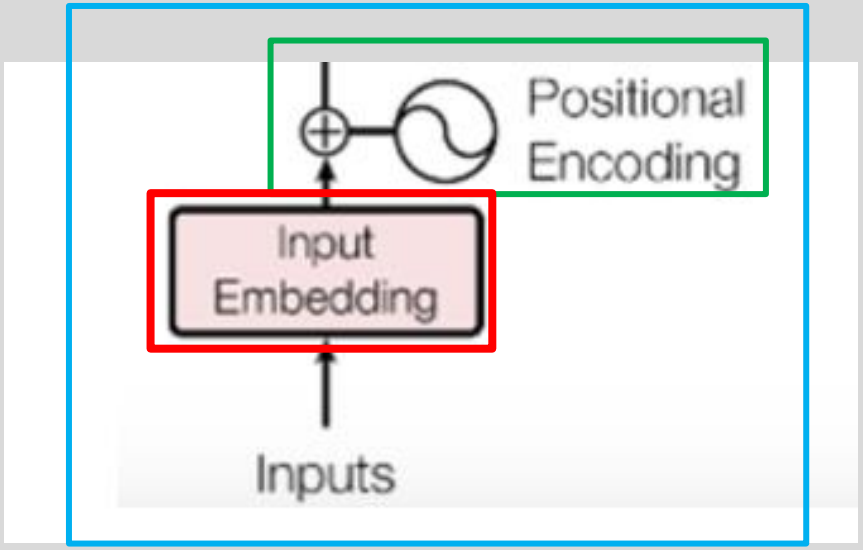
Self-Supervised Learning

LLM アーキテクチャーの成功を支えたもの 2

Self-Supervised Learning



Embedding Layer



LLM アーキテクチャーの成功を支えたもの2
Self-Supervised Learning

このセッションでは、LLM アーキテクチャーの成功を支えた技術として Self-Supervised Learning を取り上げます。

この技術は、インターネット上にある大量のテキストデータ、それには何のラベルも持たず何の構造もないように見えるのですが、そこから、語の意味ベクトル ＝ **embedding** を抽出することを可能にしました。

「教師 ＝ Supervisor 」がAIを鍛える

ニューラル・ネットワークの訓練・学習のプロセスで「教師 ＝ Supervisor 」というのは、訓練・学習が達成すべき正しい目標を知っていて、それをニューラル・ネットワークに教えようとする役割を持つもののことを言います。

機械の考えと教師のしめす正しい答えとの間にズレが生まれることは、機械にとってはとても重要です。なぜなら、機械のニューラル・ネットワークは、「バック・プロパゲーション」といメカニズムで、教師の教えにしたがって自分の間違いを正そうとするからです。

例えば、画像認識のニューラル・ネットワークの訓練では、猫の画像を示して、「これは猫だよ」と教えるのが「教師」の役割です。機械は、そうした認識ができるように、自分の身体を修正・改造していきます。機械は、とても、素直なものです。

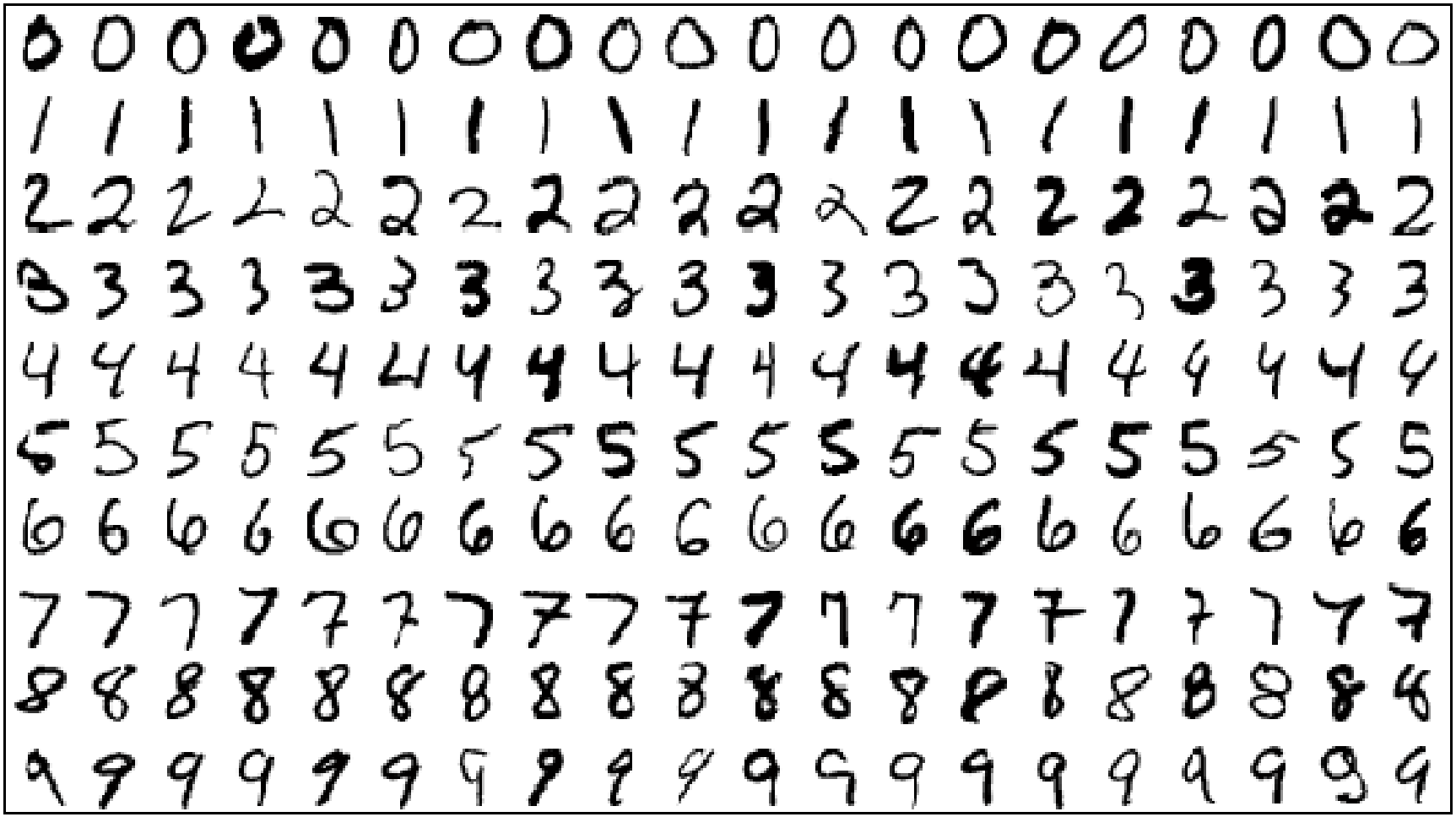
Supervised Learning

人間やロボットの教師が機械の側に実際にいなくても、画像認識の例でしたら、猫の画像に「猫」というラベルがついていれば、そのラベルが「教師」の役割を果たすことができます。

猫のラベルがついた多数の猫の画像を繰り返し学習すれば、今度は、ラベルがなくてもニューラル・ネットワークは猫の認識ができるようになります。こうした学習のスタイルを「 Supervised Learning ＝ 教師あり学習」と言います。

画像認識の分野で利用された訓練用のデータセットの例を紹介します。

MNIST 手書き数字のデータベース



60,000サンプル

http://yann.lecun.com/exdb/mnist/

ImageNet 1000 Categories x AlexNet

Non-Supervised Learning

Supervised Learningの弱点は、学習のために ラベル付きの大量のデータを用意しなければいけないことです。

2012年 Googleはラベル付けされていないイメージだけから、画像の認識が可能であることを示して衝撃を与えます。

この「Googleの猫」と言われる Non-Supervised Learningの登場は、2012年の「Deep Learning 革命」を構成する重要な出来事でした

Non-Supervised Learning

我々は、ラベル付けされていないデータだけから、高レベルの、クラスに固有の特徴を検出することが出来るかという問題を考察した。例えば、ラベル付けされていないイメージだけから、顔の検出が可能かという問題である。

2012年 Googleの猫

広く受け入れられているように見える直観に反して、我々の実験結果は、顔であるかそうでないかのラベルをイメージにつける必要なしに、顔の検出が可能であることを明らかにした。

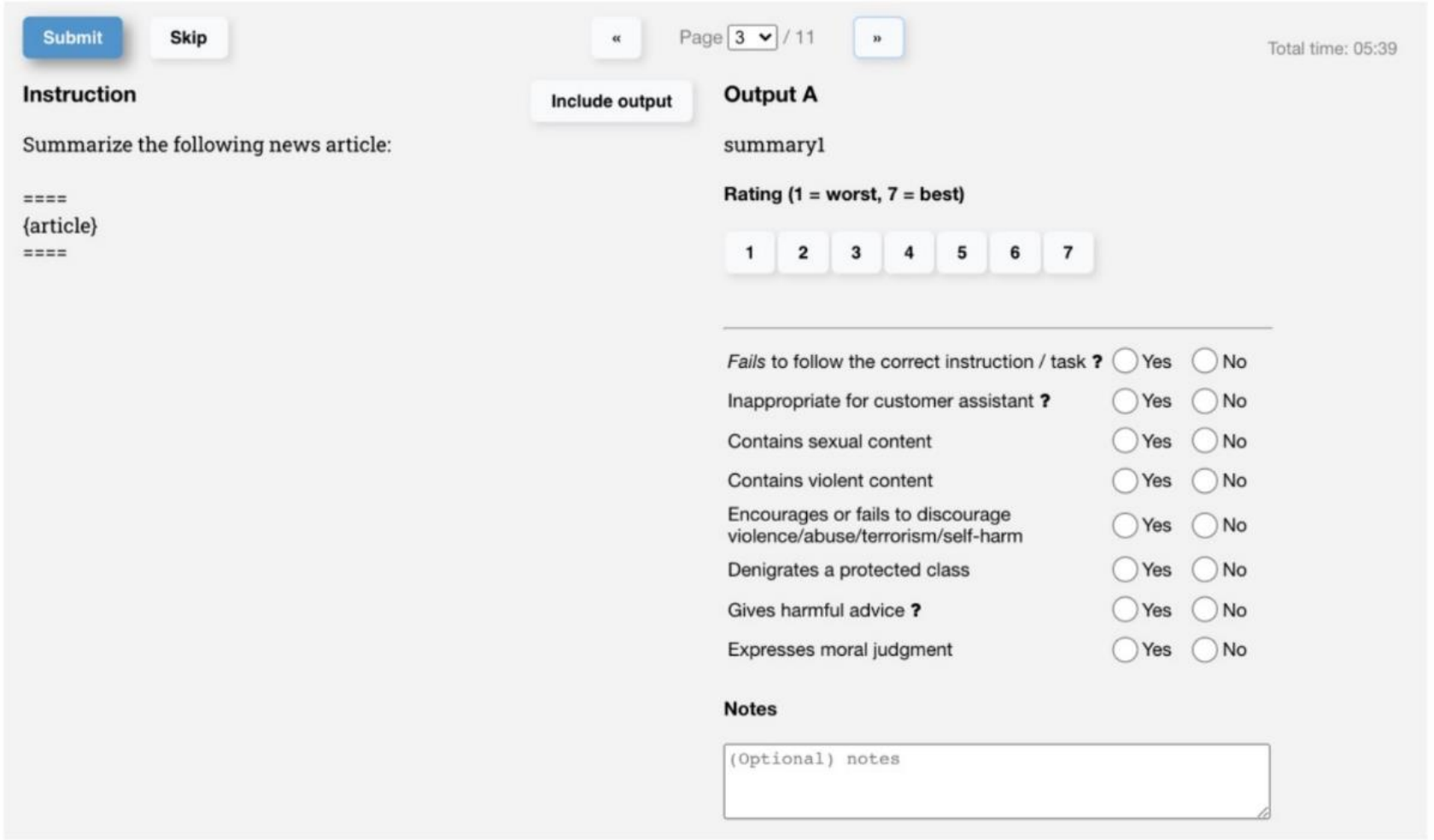
ChatGPTでの人間の教師の採用

ChatGPTでは、システムの訓練に、実際の人間を教師として採用して、多くの人を驚かせました。

「人間のフィードバックからの強化学習」”Reinforcement Learning from Human Feedback (RLHF)” と呼ばれる手法です。

機械をどのように教育・訓練するかについて、これからも新しい試みがなされるのかもしれません。

ChatGPTで、人間の教師が、機械の解答に対して行う評価のチェックリスト



Self-Supervised Learning

Self-Supervised Learning = 自己教師付き学習というのは、学習データの一部をラベルとして利用する学習のスタイルです。

データにわざわざラベルを外付けする手間はいらないという点では、先に見た Non-Supervised Learningに似ていますが、ラベルに当たるものは存在しますので、Supervised Learning の一種です。

そのラベルが、学習データ自身に含まれているので、Self-Supervised Learningと言われます。

LLMは、大量のテキスト・データを読み込みますが、そのどこにラベルがあるのでしょう?

LLM自身を訓練するステージ

LLMのembeddingの生成に、Self-Supervised Learning という学習スタイルが使われていて、それがLLMアーキテクチャーの成功の要因の一つだというのが今回のセッションのテーマです。

そのことを説明する前に、一つ注意してもらいたいことがあります。

それは、我々がLLMを利用する時、そのLLMはすでに訓練済みであるということです。

訓練済みだということは、文字列トークンに対応するembeddingも、すでにLLMに準備されているということです。そこでは、embeddingの生成をユーザーは意識する必要はありません。

Next Token Predictionと embeddingの生成

今回の話は、我々のLLMの利用以前の、LLMの訓練時代が舞台です。

訓練途中のLLMは、すこし想像しにくいとおもいます。それは、よちよち歩きのLLMで、思ったようには動いてくれません。トークンをembeddingに変換する能力も、embeddingをトークンに変換する能力も育っておらず、出鱈目な言葉を返します。

ただ、LLMとしてそのシステムは、Next Token Predictionを実行しようとします。

。

a1 , a2 , a3 , a4 , …

というトークンの並びが入力に与えられているとして、

x2 , x3 , x4 …

というトークンの並びをNext Token Prediction で返したとします

入力　a1 , a2 , a3 , a4 , …

出力　　x2 , x3 , x4 , …

x2が出力された時、正解のラベルはa2とします。

x3が出力された時、正解のラベルはa3とします。

x4が出力された時、正解のラベルはa4とします。…

ラベル a2, a3, a4 …があたえられることで、Next Token Predictionを実行しているLLMのパラメーターは、バック・プロパゲーションで修正されていきます。

重要なことは、この修正されるパラメーターの中に、embeddingのパラメーターも含まれているということです。

こうしたステップを膨大な入力テキストごとに繰り返すことを通じて、LLMの内部に、入力されたテキストの意味を反映したembeddingが生成されていきます。

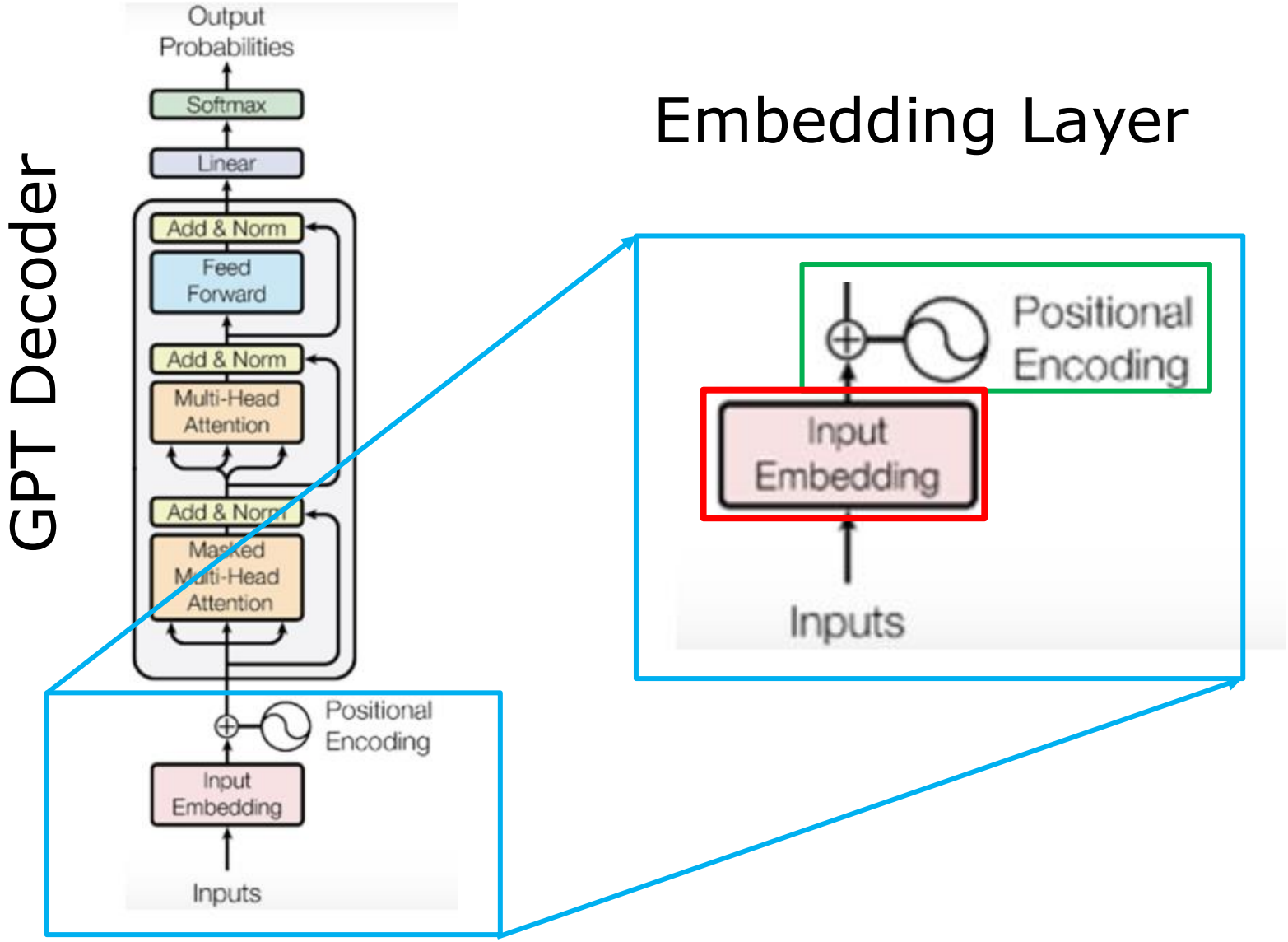
これは、なかなかスマートなアイデアです。

Embedding Layer

LLMには、この Self-Supervised Learningを利用して、大量のそれ自身はラベルをもっていないテキストデータを読み込んで、embeddingを生成するコンポネントが組み込まれています。それが Embedding Layer です。

Embedding Layer は、LLMの最も重要なコンポーネントの一つです。

ただ、LLMのアーキテクチャーの図をみても、Embedding Layer は、意外と小さくしか示されていないように思います。またLLMのユーザーも、このEmbedding Layer の働きを意識することは少ないように思います。


GPT Decoder
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Input Embedding
Inputs
Embedding Layer
Positional Encoding
Input Embedding
Inputs

それには、いくつか理由があります。

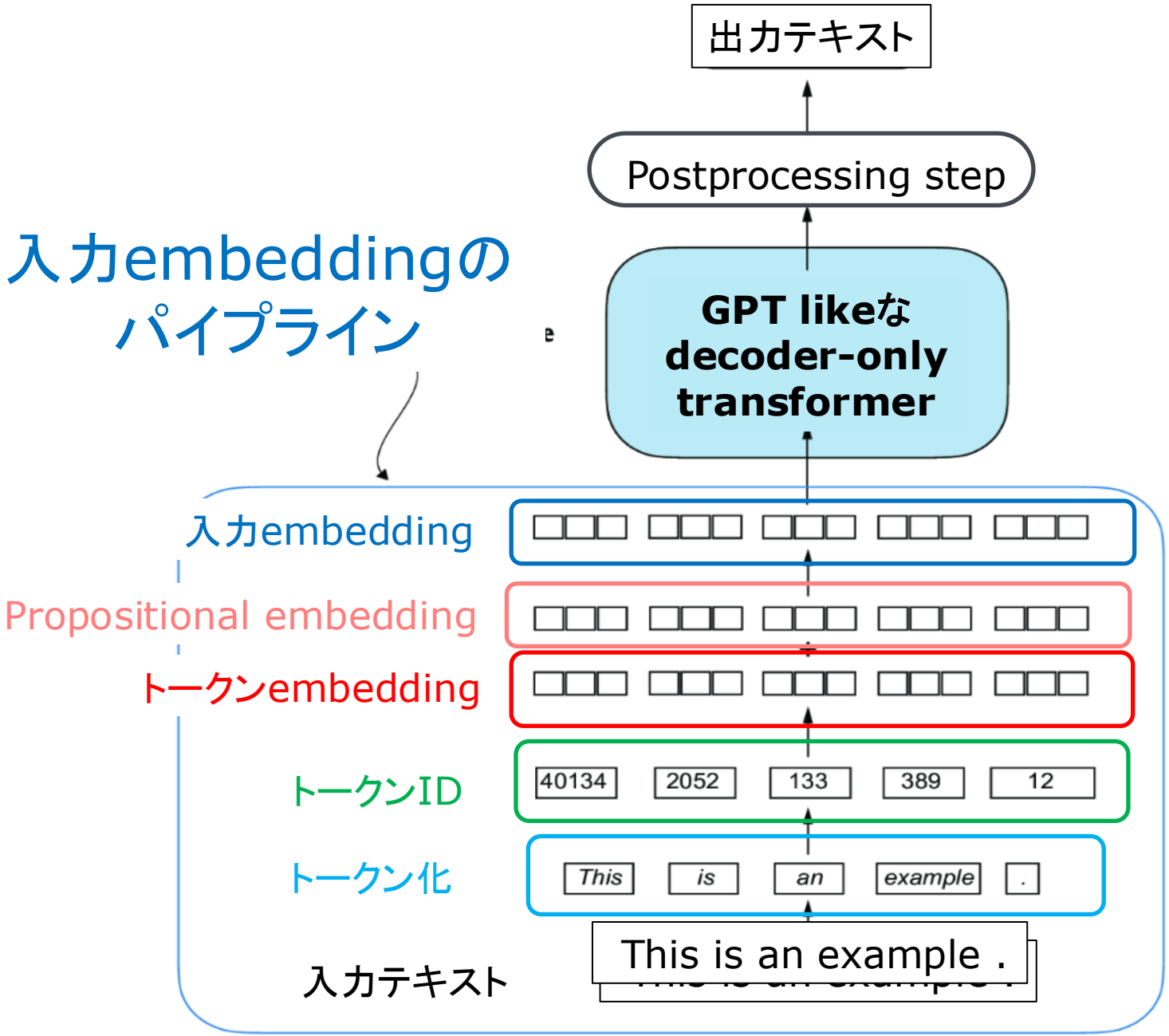
一つには、先のセッションでも見たように、我々はLLMを「文字列トークンの並びから次の文字列トークンを一つ予測する」ものとして文字列トークン中心に考える方が、LLMの振る舞いを理解しやすいからです。あまりembeddingには目が行きません。

それは残念なことです。

もう一つは、先のセクションで見たように、我々は、すでに完成した形でembeddingを受け取っているので、その生成メカニズムを意識することが少ないからです。

embedding Layerは、文字列トークンから対応するembeddingを引き出す単純なLookup Tableに見えます。

入力embeddingのパイプライン

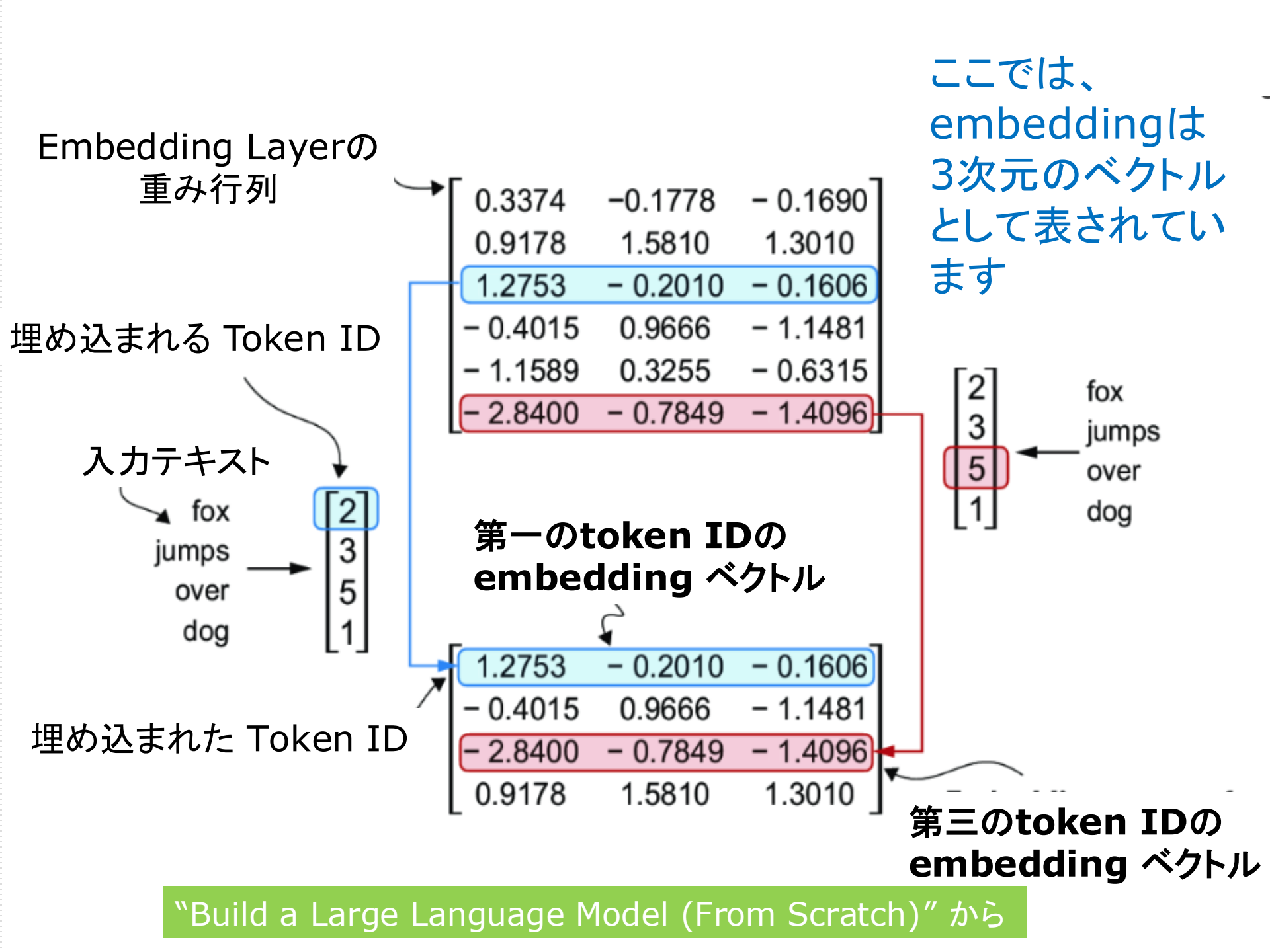


“Build a Large Language Model (From Scratch)” から

Lookup テーブルとしてのEmbedding Layer

訓練済みのLLMでは、Embedding Layerはルックアップ・テーブルとして機能します。Embedding Layerの重み行列からトークンIDに対応する埋め込みベクトルを取得します。

例えば、トークンID 5の埋め込みベクトルは、埋め込み層の重み行列の6行目となります（Pythonは0からカウントを始めるため、5行目ではなく6行目となります）。



“Build a Large Language Model (From Scratch)” から

Embedding Layerは、巨大なコンポネント

上の図では、embeddingベクトルは 3次元のベクトルとして、また、Embedding Layerの重み行列は、6個のエントリーをもつものとして表現されています。

ただ、実際のLLMは、小さ目のGPT-2 でも、embeddingの次数は 768次元で、テーブルのエントリー数(IDを持つトークンの数)は、50,257個もあります。

Embedding Layerは、巨大なコンポネントなのです。

LLM訓練 = embedding生成のコスト

“Build a Large Language Model (From Scratch)” で、著者は、次のようなLLM訓練のコスト、それはembedding生成のコストと考えていいのですが、について、次のような資産を行なっています。

「70億パラメータのLlama 2モデルの学習を例に挙げます。このモデルは、高価なA100 GPU上で184,320 GPU時間（2兆トークンを処理）を要しました。

執筆時点において、AWS上で8基のA100クラウドサーバーを稼働させる場合、1時間あたり約30ドルの費用がかかります。

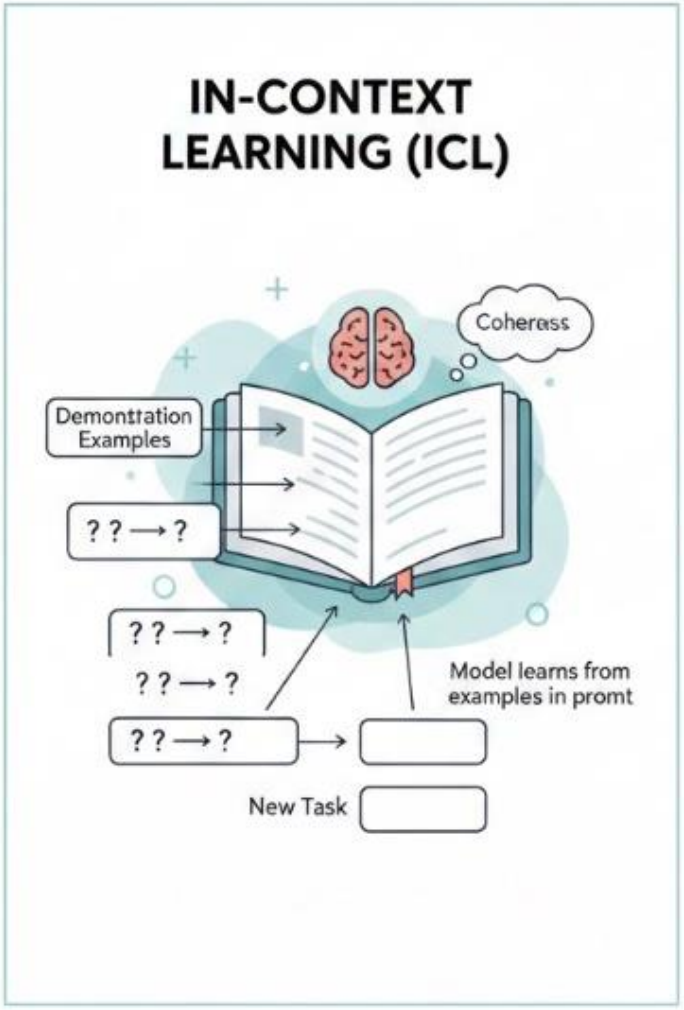
このような大規模言語モデルの総トレーニングコストは、おおよそ69万ドルと概算されます（184,320時間を8で割り、30ドルを乗じた計算値です）。」

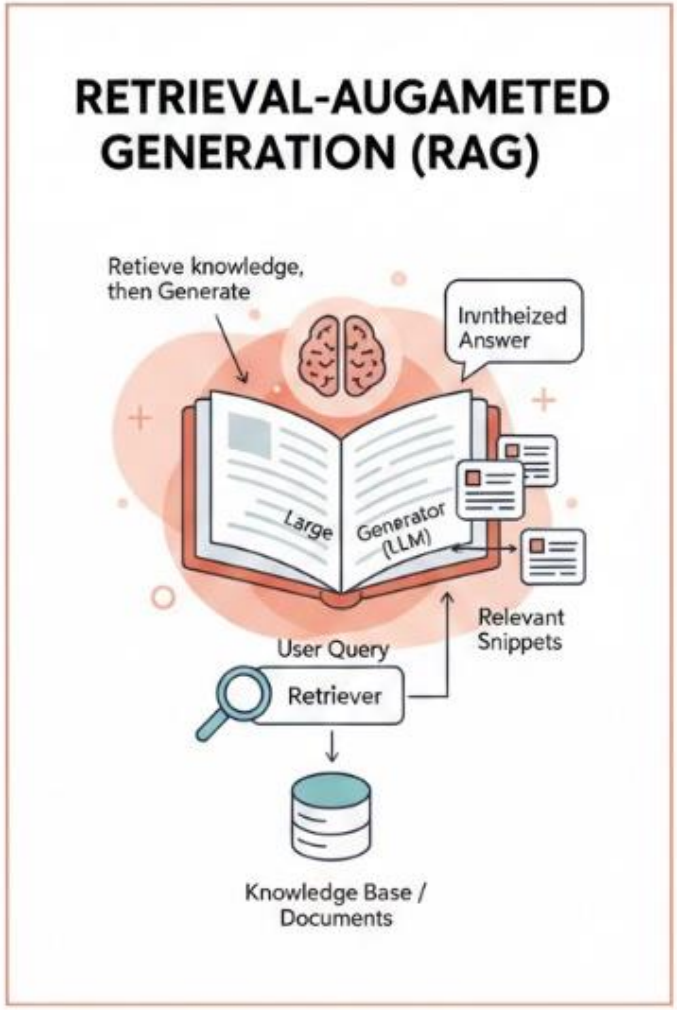
大規模言語モデル LLM の成功を支えたもの 3

In-Context Leaning と Retrieval-Augmented Generation

LLM アーキテクチャーの成功を支えたもの 3

ICLとRAG





大規模言語モデルの機能拡張

これまで見てきたのは、LLMの基本的な機能についてのものでしたが、LLMの急速な進化は、その機能を大きく拡張して来ました。このセッションでは、そうした機能拡張を見ていこうと思います。

モデルの機能を動的に拡張するための試みとして、現在、二つの対照的なアプローチが注目されています。

一つは、モデルの重みを更新することなく、プロンプト内の例示によってタスクに適応させる「インコンテキスト学習（**In-Context Learning: ICL**）」です。もう一つは外部データベースから関連情報を検索して生成を補強する「検索拡張生成（**Retrieval-Augmented Generation: RAG**）」です。

インコンテキスト学習（ICL）のメカニズム

インコンテキスト学習（ICL）は、GPT-3の登場とともにその有効性が広く認識された手法です。

大規模な事前学習によって獲得されたモデルの潜在能力を、入力プロンプト内の「例示（Demonstrations）」によって特定のタスクへと誘導するプロセスです。

このアプローチの最大の特徴は、モデルのパラメータを一切更新しない「非パラメトリックな適応」にあります。

ICLの「学習」

ICLにおける「学習」とは、厳密には重みの更新を伴う「学習」ではなく、モデルの隠れ状態（Hidden States）が入力されたコンテキストに基づいて遷移し、特定の推論パターンを選択する現象を指します 。

モデルはプロンプトに含まれる少数の入出力ペア（Few-shot）から、タスクの形式、言語スタイル、論理的なステップを読み取ります。

例えば、コード生成タスクにおいては、変数名の命名規則やコメントの記述スタイルといった微細な特徴が、モデルの出力品質を左右することが判明しています 。

検索拡張生成（RAG）のアーキテクチャ

RAGは、LLMの生成能力と、情報検索（Information Retrieval）システムを融合させたシステム設計のアプローチです。

LLMを「知識の保管庫」としてではなく「情報の処理エンジン」として位置づけ、必要な事実は外部の信頼できるソースからその都度調和させることで、情報の鮮度と正確性を担保しています。

RAGの最大の利点は、ハルシネーションの抑制と情報の透明性にあります。生成された回答が外部のどのドキュメントに基づいているかを検証できるため、法的責任や医学的正確性が問われる分野での信頼性が極めて高いです。

RAGの4段階パイプライン

RAGのシステム構築は、一般的に「インジェクション」「検索」「拡張」「生成」の4つのステージで構成されます。

1. **インジェクション（データの構造化）：**
   PDFやデータベースなどの非構造化データを、モデルが処理しやすい「チャンク（断片）」に分割し、ベクトル埋め込み（Embedding）に変換してベクトルデータベースに登録する 。

2. **検索（Retrieval）：**
   ユーザーのクエリも同様にベクトル化され、データベース内からコサイン類似度などを用いて関連性の高いチャンクが抽出される 。

3. **拡張（Augmentation）:**
   抽出された複数のチャンクを、元のユーザープロンプトと連結し、コンテキストとしてモデルに提供する 。

4. **生成（Generation）:**
   モデルは提供されたコンテキストを「唯一の根拠（Grounding）」として回答を生成し、必要に応じて引用元（Citations）を明示する

RAGのパラダイム進化：NaiveからAgenticへ

RAGの技術は、単純なベクトル検索を行う「Naive RAG」から、より複雑な推論を伴う「Advanced RAG」や「Modular RAG」へと進化しています。

Advanced RAG: 検索の前にクエリを書き換えて検索精度を高める（Query Rewriting）、検索されたドキュメントを再順位付けする（Reranking）、あるいは複数の検索手法すを組み合わせる「ハイブリッド検索」を導入し、精度を向上させます 。

Agentic RAG: AIエージェントが自律的にどのデータベースを検索すべきか、あるいは検索結果が不十分な場合に再検索を行うべきかを判断する。これにより、複雑なマルチホップ推論（複数の情報を繋ぎ合わせる推論）が可能となります 。

ICLとRAGの対比分析
コスト

エンタープライズ環境におけるコスト比較では、RAGの圧倒的な優位性が示されています。例えば、1,000ページのナレッジベース（約60万トークン）を対象に、1日1,000回のリクエストを処理する場合、すべての情報をプロンプトに流し込むロングコンテキストICLのアプローチは、RAGと比較して月間コストが8倍から82倍高額になるという試算があります 。

ICLでは、モデルに入力する情報の量に比例して課金が発生し、さらに入力が増えるほど推論のレイテンシ（応答時間）も増大しますす。一方、RAGは事前のインデックス作成（Indexing）にコストがかかるものの、推論時にはクエリに関連する数千トークンのみを送信するため、リクエストあたりの単価を極めて低く抑えることができます 。

ICLとRAGの対比分析
精度と信頼性

ICLは、モデルの「推論能力」や「スタイル模倣」においてはRAGを凌駕する場合があります。特に少数の例示からパターンを読み取る能力は、定型的なタスクの自動化において極めて効率的です。しかし、事実関係（Factuality）の正確性が求められるタスクでは、ICLはモデルが本来持つ誤った知識を修正できず、出力の信頼性が低下するリスクがあります。

RAGは、「外部の事実」を直接コンテキストに流し込むため、事実に基づいたQAにおいて高い精度を発揮します。しかし、検索器（Retriever）が適切な情報を取得できなかった場合、あるいはノイズの多い情報を取得してしまった場合に、生成結果が著しく損なわれるという「検索器依存の脆弱性」を抱えています。

ロングコンテキスト時代の変化

2024年後半から2025年にかけて、Gemini 1.5 Pro（200万トークン）やGPT-4o（12.8万トークン以上）のように、極めて長いコンテキストウィンドウをサポートするモデルが登場しました。

これにより、「RAGを使って検索するよりも、すべてのドキュメントを直接プロンプトに放り込む方が精度が高いのではないか」という議論が加速しています 。

コンテキストウィンドウ と 作業メモリ

最新の研究は、物理的な「入力可能サイズ（コンテキストウィンドウ）」と、モデルが情報を統合・処理できる「実効的な能力（作業メモリ）」を明確に区別しています。

- **コンテキストウィンドウ:**
  モデルが一度に受け取れるRAWデータの量。Gemini 1.5 Proでは数千ページのドキュメントに相当する 。

- **作業メモリ:**
  モデルが複数の情報を相互に関連付け、グローバルな推論を行うために保持できる情報の帯域。

驚くべきことに、最新のフロンティアモデルであっても、変数追跡のようなタスクにおいて、追跡すべき変数の数が5〜10個を超えると、推論精度がランダムな推測と同レベルまで急落することが判明しています。

これは、コンテキストウィンドウがどれほど大きくても、モデルが内部で情報を「覚えている」帯域幅には厳しい物理的制約があることを示唆しています。

「Lost in the Middle」と位置バイアス

長いコンテキストを直接処理させる（ロングコンテキストICL）アプローチは、情報の位置によって精度が変動する「位置バイアス」の問題を抱えています。モデルはプロンプトの最初と最後に書かれた情報を重視し、中間部分に埋もれた情報を無視する傾向があります 。

RAGは、情報の位置に関わらず、関連性の高い断片を直接モデルの「視界（コンテキスト）」の最前列に配置するため、この位置バイアスの影響を回避しやすいのです

検索拡張としてのロングコンテキスト

結論として、ロングコンテキスト技術はRAGを駆逐するものではなく、RAGを「強化」するツールとして機能しています。

現在の主流は「検索を全く行わない」ことではなく、「検索した結果をより広大なコンテキストウィンドウで贅沢に処理する」というハイブリッドな方向へ向かっています。