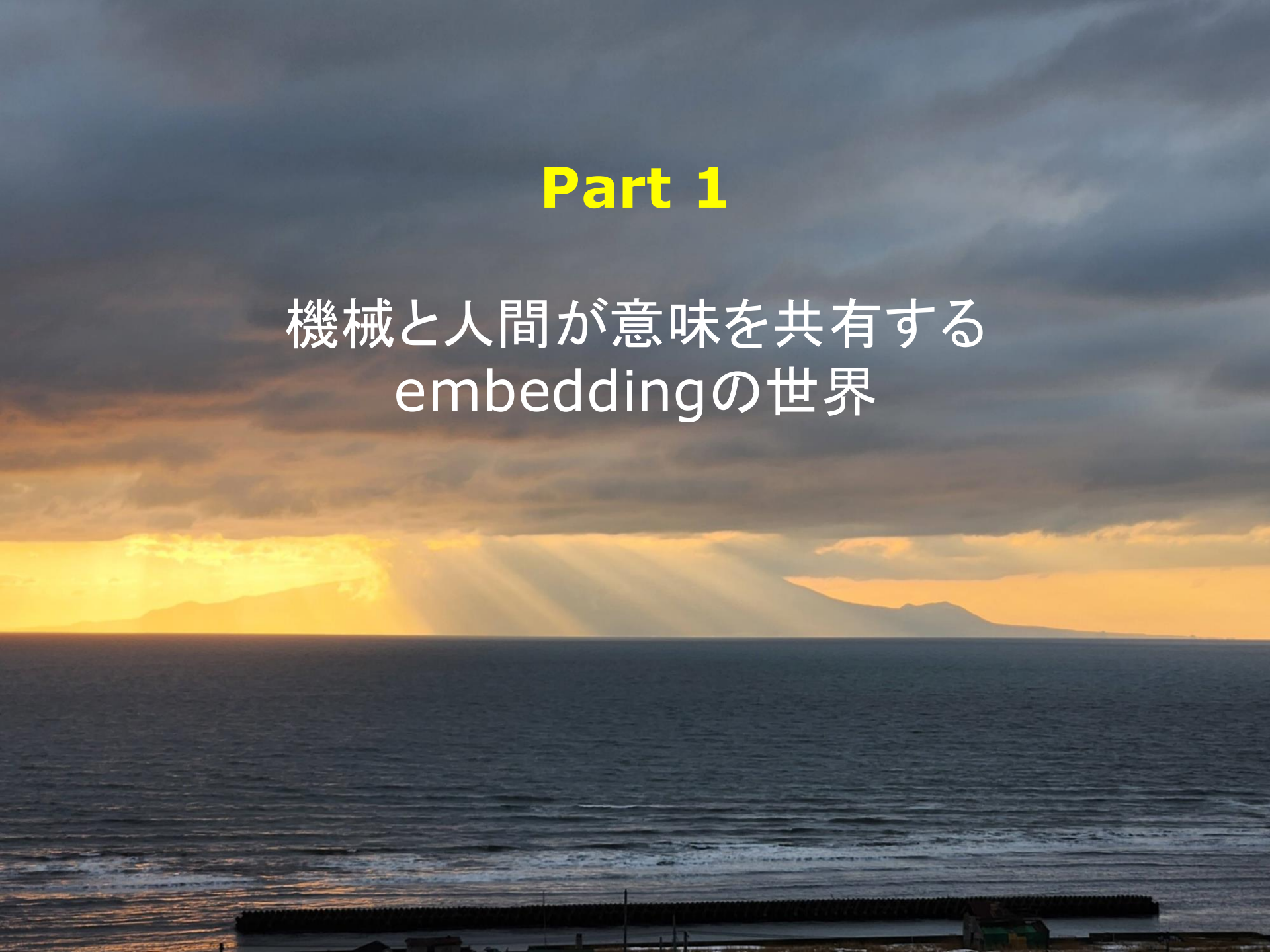
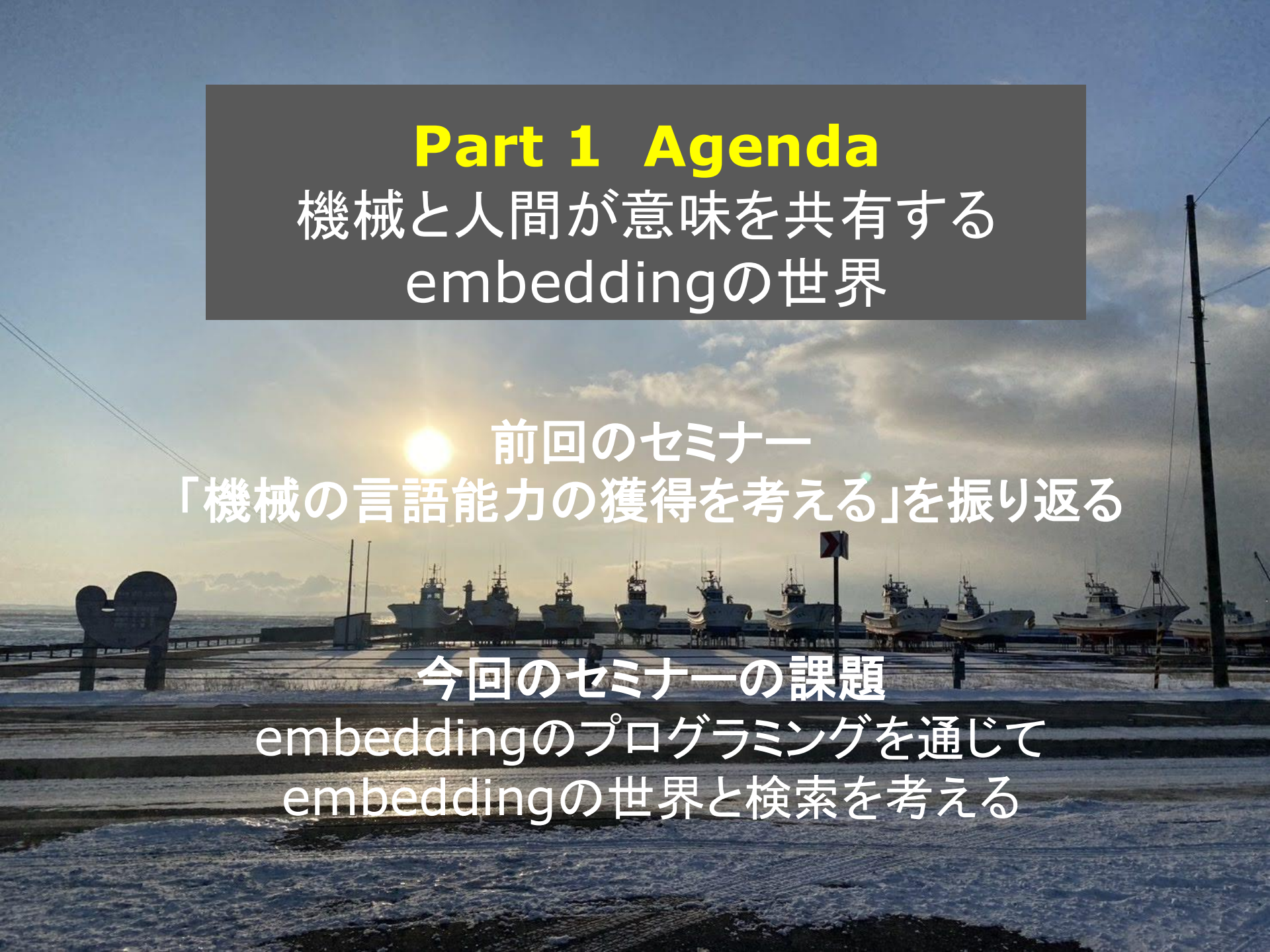

embeddingプログラミングの基礎
機械と人間が意味を共有する
embeddingの世界


Agenda
embeddingプログラミングの基礎
Part 1
機械と人間が意味を共有する
embeddingの世界
Part 2
embeddingプログラミング入門
Part 3
embeddingと検索技術の新しい展開



Part 1

機械と人間が意味を共有する
embeddingの世界



Part 1 Agenda

機械と人間が意味を共有する embeddingの世界

前回のセミナー
「機械の言語能力の獲得を考える」を振り返る

今回のセミナーの課題
embeddingのプログラミングを通じて
embeddingの世界と検索を考える

前回のセミナー
「機械の言語能力の獲得を考える」
を振り返る

「機械の言語能力の獲得」と
「意味の理解能力の獲得」

前回のセミナーは、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、その中核を「意味を理解する」能力の獲得と見なして議論を展開しました。

その中心問題は、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いであり、この問に対して「意味の分散表現論」の発展が一つの答えを与えることになります。

この四半世紀のAI技術の理論史は、「意味とは何か」を探求する「意味の分散表現論」すなわちembedding論の発展史であると考えることができます。

意味の分散表現論の系譜

- 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

2010年 Coecke DisCoCat論文

2018年 Bradley DisCoCat解説論文

2020年 Bradley DisCoCat批判

「翻訳モデル」から「大規模言語モデル」への移行

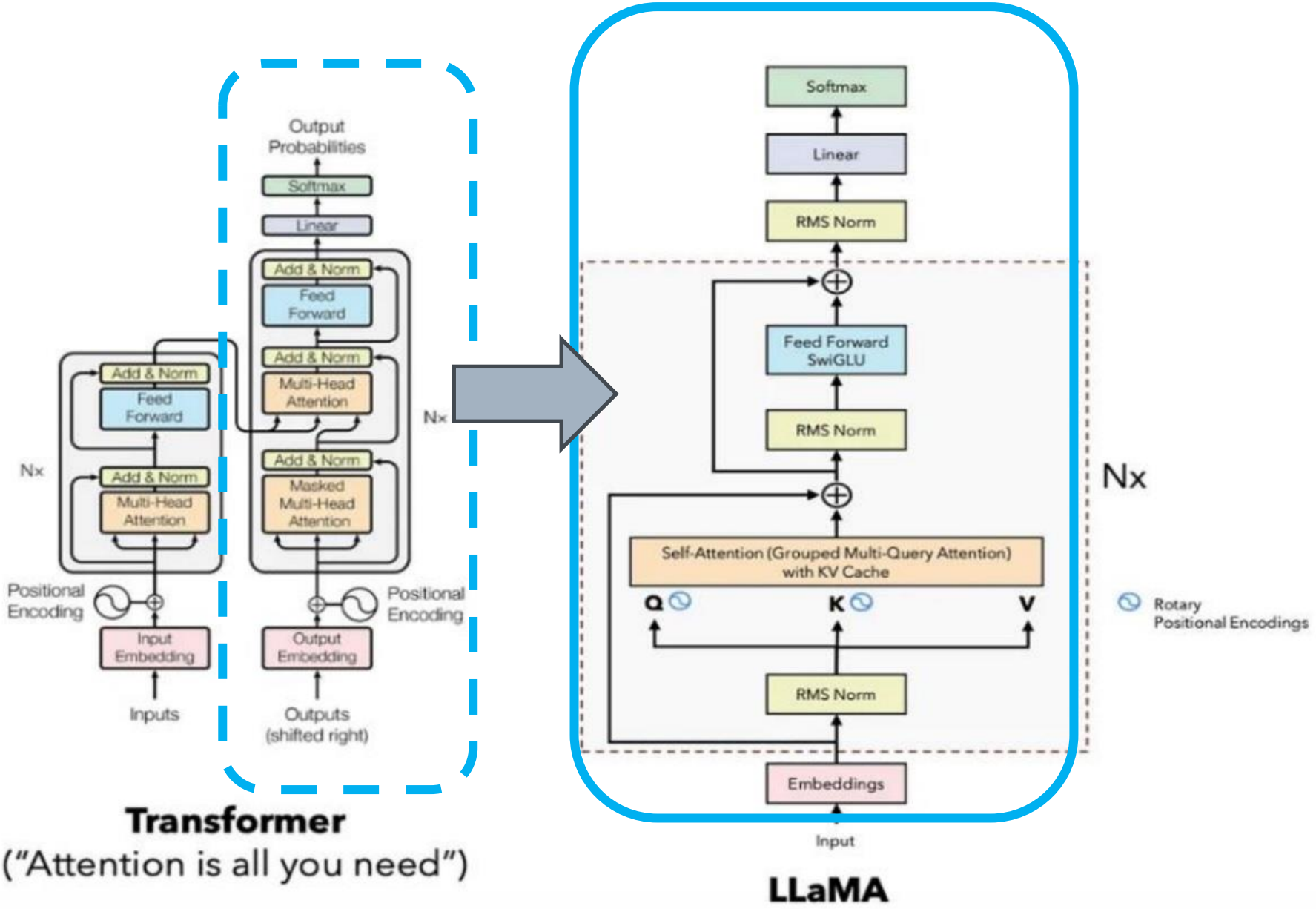
語の意味の分散表現が文の意味の分散表現へと進み、それをベースとした「翻訳モデル」がAttention メカニズムの導入により発展します。 変化はさらに続きます。

Transformerを頂点とした「翻訳モデル」がencoder-only / decoder-only アーキテクチャーに分解・解体する中で、後者のアーキテクチャーの「勝利」として、「大規模言語モデル」が成立します。

「機械の言語能力の獲得」という機械の能力の画期的な拡大を可能としたのは、技術的には、強力な「大規模言語モデル」の成立によるものです。

大規模言語モデルの成立が、
「機械の言語能力の獲得」を可能にした

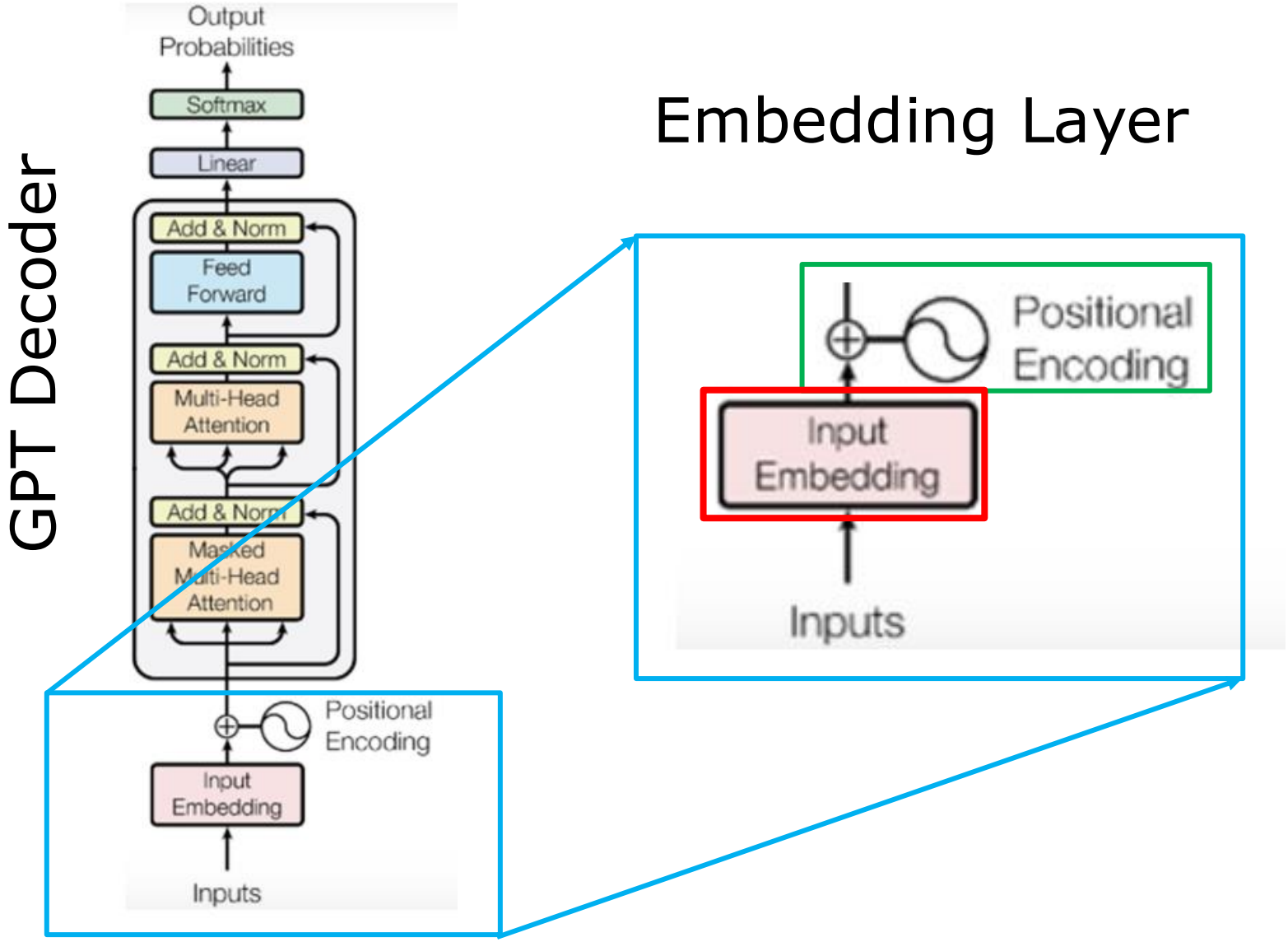
「翻訳モデル」から「大規模言語モデル」へ



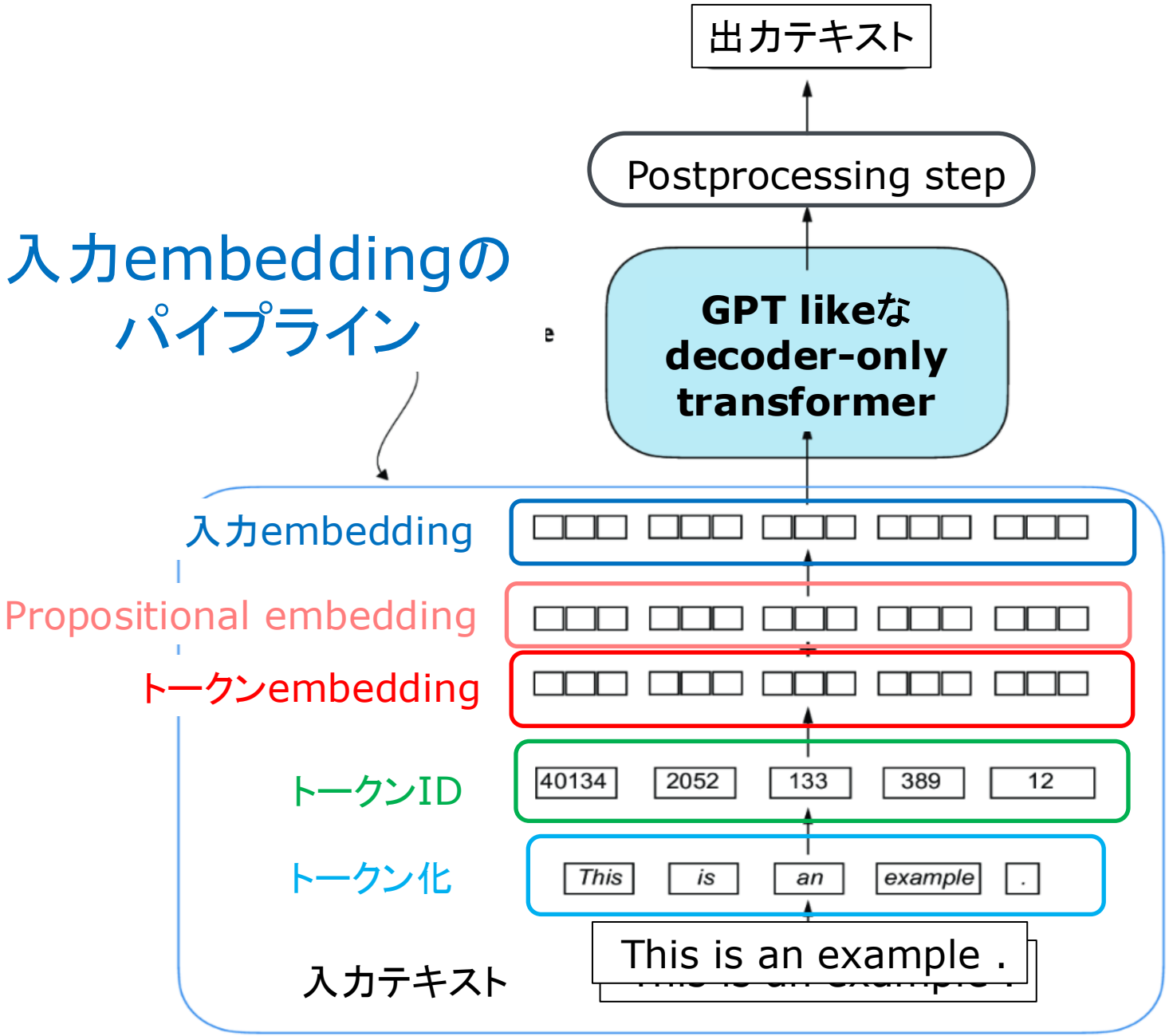
大規模言語モデルの成功の要因

「大規模言語モデル」の成功には、多くの要因があるのですが、前回のセミナーでは、その中で特に、三つの要因を挙げました。

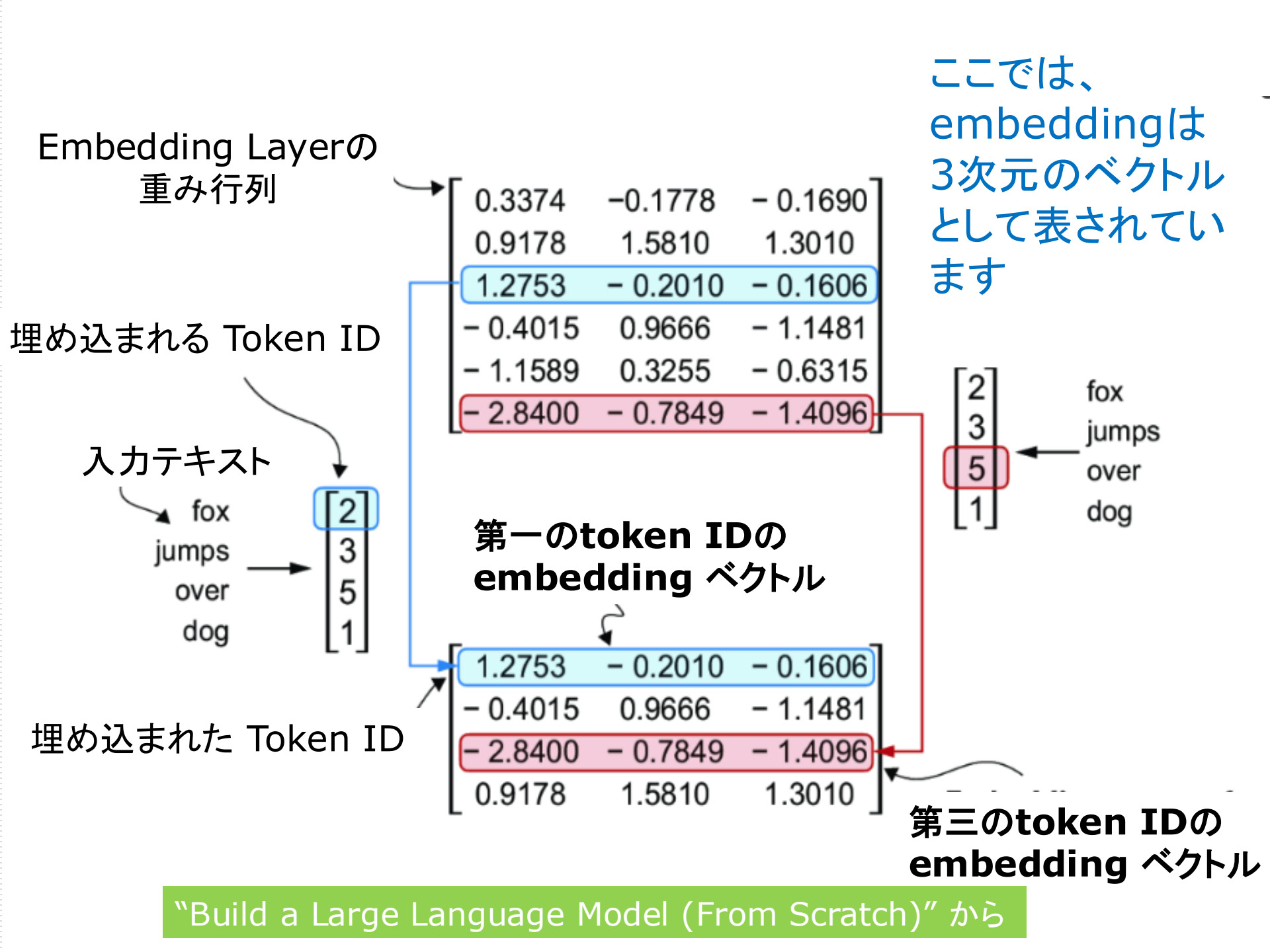
- **Next token prediction**に特化したシンプルでスケーラブルなアーキテクチャーの採用
- **Self-Supervised Learning**によるインターネット上のラベルのない大量の文字データからのembeddingの生成能力
- In-Context learningと**Retrieval-Augmented Generation**による、柔軟な機能拡張の能力


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

入力embeddingのパイプライン



"Build a Large Language Model (From Scratch)" から



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

LLMとembeddingの不思議

繰り返しになりますが、機械の言語能力の獲得は、 LLMがembeddingを通じて、人間のことばの意味を、理解できるようになったからだと考えています。

ただ、LLMの振る舞いにも、embeddingの役割にも、いろいろ不思議なことがあります。

例えば、LLMが Next token predictionマシンであるという認識と、それがコンテキストを含む言葉の意味を理解する能力として現れるという認識には、ギャップがあるように思います。

また、embeddingは、それ自身で既に「世界」についての情報を豊富に含んでいるように思えます。そうした特徴は外付けのRAGとは独立なものです。それは、embedding の生成メカニズムそのものに根ざしているはずです。

前回のセミナーで語れなかったこと

残念ながら、前回のセミナーでは、こうしたLLMの振る舞いの不思議さや、embeddingが抱えている情報の豊ささを、十分にふれることはできませんでした。

それについては、今回のセミナーを含めて、後続のセミナーで補っていこうとおもいます。

今回のセミナーの課題

embeddingのプログラミングを通じて
embeddingの世界と検索を考える

人間と機械が意味を通じ合うための「共通言語」としてのembeddingの重要性

僕は、Embeddingの発見を「この4半世紀のAI研究の白眉」と高く評価し、それを「人間と機械の共通言語」と呼んでいます。また、人間にとってEmbeddingは、音声や文字に次ぐ「ことばの第三の形態」であるとも位置づけています。

人間がことばの意味を機械に伝えることが可能となり、また、機械が人間のことばの意味を理解できるようになったことは、人間にとっても機械にとっても、大きな飛躍でした。

それは、embedding技術によって初めて可能になりました。

embeddingは目立たない

機械の言語能力を可能にしたLLMの内部では、embeddingは文字通り、文字の代わりに縦横無尽に飛び回っています。

embeddingを直接理解するのは困難なので、我々は「それ」がある文字列トークンの「意味の表現」であると頭の中で読み替えてそのLLM内部での働きを理解します。両者は、本来同じものなので、そうした読み替えが間違っているわけではありません。

embeddingは重要なのですが、容易に等価な他の概念に置き換えられて理解されるので、それ自体はあまり目立たないのです。そのことは、embeddingの理解を、少し難しくします。

今回のセミナーでのembeddingへのアプローチ

今回のセミナーでは、複雑なLLMの世界をいったん離れて、LLMとは直接関係のない世界に舞台を設定して、人間と機械が意味を通じ合うための「共通言語」としてのembeddingの働きを、まず、確認したいと考えています。

今回のセミナーでは、LLMを利用したコード開発の経験をもつIT技術者に、直接embeddingをハンドルするembeddingプログラミングのサンプルを紹介します。

embeddingプログラミングの基礎を学ぶことは、LLMのより深い理解にも、そのより高度な利用にembedding技術を活かすことに役立つと考えています。

embeddingのプログラミングを通じて embeddingの世界と検索を考える

今回の「embeddingプログラミングの基礎」は、内容的には次の二つのコンテンツから構成されています。

- embedding プログラミング入門
- embeddingと検索技術の新しい展開

embedding技術利用のフォーカスの一つに検索技術が登場していることに注目してください。残念ながら、この議論を、今回は語ることができませんでした。別のセミナーに譲りたいと思います。以下に基本的な観点のみを述べたいと思います。

embeddingが
グローバルな検索の世界に与える影響

AIの登場がITの世界に与える影響については、多くのIT技術者が自分の経験を通じて、多くのことを既に知っていると思います。

今回のセミナーでは、embeddingの登場が検索に与える影響を見ていきたいと思います。

検索の世界は、今、急速な変化の只中にあります。その変化を特徴づけるのは、機械が「ことばの意味を理解し始めた」という変化です。

「機械と人間が意味を共有するembeddingの世界」の登場が、検索の世界を変えようとしています。

21世紀のIT技術の二度目の大変化

それは、LLMの能力がRAGによって大きく拡大したという、狭いAI技術の変化の話ではないのです。

振り返ってみれば、GAFAM等のBig Techの成立に結果した21世紀初頭のITの世界の大きな変化は、Googleのグローバルな検索技術の登場によって先導されました。

その成功は、PageRank・MapReduceというアルゴリズムの実行を可能とするWebスケールの大規模分散システムと、「広告モデル」という新しいビジネス・モデルによって支えられたものでした。

四半世紀の時を経て、21世紀のIT技術・ITビジネスは二度目の大きな変化を迎えようとしています。

どのような変化を展望するのか

もっとも、予想される変化を現在のIT技術・ITビジネスの延長で考える必要はないと思います。

「機械の言語能力の獲得」は、機械にとっても人間にとっても歴史的な大事件です。より具体的に、「機械と人間が意味を共有するembeddingの世界」の進展は、機械にとっても人間にとっても重要な意味を持ちます。

それは、人類の情報共有と情報蓄積のスタイルが、大きく変わるだろうという展望を我々が持ちうることだと思います。

AIがプログラムを生成する時代に
embedding プログラミングを学ぶことの意味

今回のセミナーでは、直接embeddingを操作するembeddingプログラミングの紹介にフォーカスしています。

AIが、ほとんどどんなプログラムも作ってくれる時代に、プリミティブなembeddingプログラミングを学ぶことの意味を確認しておきましょう。

単なるプログラミングテクニックとしてだけでなく、「機械がどのように言葉の意味をembeddingとして捉えているのかを知ることは、機械の言語理解の本質に触れることであり、AIを活用する上での強力な土台となります。

信頼性の高いAIシステムの
根幹技術であるため

LLMの弱点であるハルシネーションを抑制し、情報の透明性と正確性を担保する手法として「RAG（検索拡張生成）」が重要視されています。

RAGのパイプラインでは、テキストをEmbeddingに変換してベクトルデータベースに登録し、ユーザーの入力とのコサイン類似度を用いて関連情報を検索（Vector Search）するというプロセスが必須のステップとして組み込まれています。

AIがコード自体を書いてくれるとしても、この「データをどのようにチャンクに分割し、ベクトル化して検索させるか」というシステム全体の設計やチューニングは人間が行う必要があり、その裏側にある技術を理解しているかどうかがシステムの品質を左右します。

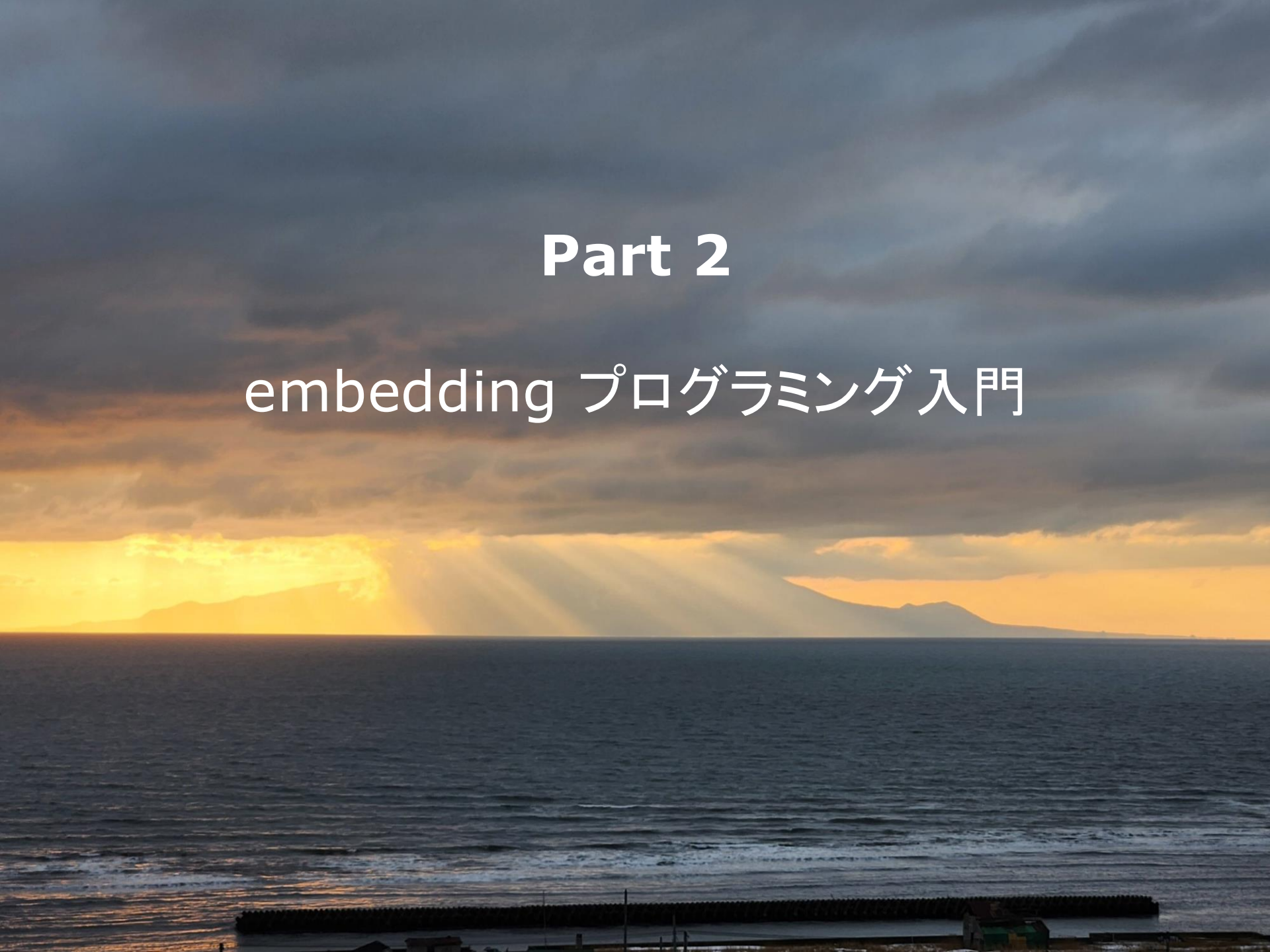
より高度なAIシステムへの進化に対応するため

AI技術は、単純なベクトル検索を行うものから、クエリの書き換えや再順位付けを行う「Advanced RAG」、さらにはAIが自律的に検索対象や再検索を判断する「Agentic RAG」へと急速に進化しています。

また、今後は「embeddingの共有・蓄積・検索」が情報の世界に大きなインパクトを与えると予想されています。

AIにプログラミングを任せる時代だからこそ、ブラックボックスになりがちな「ベクトル空間における意味の検索」の仕組みを人間が直接理解しておくことで、AIに対してより高度な指示を出し、生成されたシステムの評価や改善を正確に行うことができるようになると考えています。


Part 1 の終わり



Part 2

embedding プログラミング入門

Part 2 Agenda
embedding プログラミング入門

embedding体験入門

Amazon レビュー情報からの意味的検索

embedding検索の拡大

embedding体験入門

embedding体験入門

このセッションは、後のセッションで紹介する embedding プログラミングの導入として、embeddingとはどんなものなのかの「イメージ」を持ってもらうことを目的としています。

ただ、それが架空の存在ではなく具体的なものであることを示すために、embeddingを具体的に生成・操作するプログラムの一部をサンプルとして提示しています。（OpenAIのAPIを使っています）

embedding ＝　ことばの意味の表現

最初に確認したいのは、embedding は機械が理解可能な形で機械にことばの意味を伝えるも意味の表現だということです。

「機械が理解可能な形」と書きましたが、コンピュータが理解できるのは、「命令」にせよ「データ」にせよ、最終的には、0か1かの二種類の信号の並びだけです。

ですので、ことばの意味の表現としてのembeddingは、最終的には、0か1の信号の並びとしてコンピュータに伝えられなければなりません。

しかし、意味を0か1の信号の並びにしろと言われても、途方に暮れてしまいます。

「コンピュータが理解できるのは、0か1かの二種類の信号の並びだけ」と言ったのは、少し言い過ぎだったかもしれません。

コンピュータは、「命令」と「データ」の違いを理解できます。

また、「データ」の中で、「文字列データ」と「数値データ」の区別もできます。「文字列データ」を他のデータと区別できるのは、ASCIIやUnicodeといった「文字コード」をコンピュータが理解できるからです。

ただ、ことばの意味を表現するembeddingは、「データ」の一種なのですが、「文字コード」とも「数値データ」とも違う形をしています。

embedding ＝　多次元ベクトル

少し飛躍があるのですが（前回のセミナーの「意味の分散表現論の系譜」を参照ください）、ことばの意味を表現する embedding は「多次元ベクトル」の形をしています。

それは、多数の浮動小数点の数値データの並びなのです。
それなら、コンピュータは理解することができます。

数字の並びと言っても、その並びの数は、とても大きいものです。

文字列の表現と
その文字列の意味の表現は違うものです

コンピュータに「文字コード」を用いて文字列の表現を伝えることと、コンピュータにembeddingを用いてその文字列の意味の表現を伝えることは、違うことです。

コンピュータに'cat'の文字列を伝えたいなら、ASCIIコードで、16進数で、**63 61 74** の3バイトを伝えればいいです。

コンピュータに'猫'の文字列を伝えたいなら Unicodeで、16進数で、**U+732B** の3バイトを伝えればいいです。

ところが、それらの文字列の意味を表現する embedding はこうなります。とんでもない数の数字を送らなければなりません。

“cat”の embedding (ada-002)

```
# "cat" のembedding

from openai import OpenAI
client = OpenAI()

client.embeddings.create(
  model="text-embedding-ada-002",
  input="cat",
  encoding_format="float"
)
```

“cat”のembedding (ada-002) 1536次元


```
CreateEmbeddingResponse(data=[Embedding(embedding=[-0.0070539117, -0.01734057, -0.009698243, -0.030739458, -0.0124843605, 0.0030
714646, -0.0051114275, -0.04111828, -0.014561542, -0.021268075, 0.019240519, 0.050759807, -0.0012246867, 0.0025521691, -0.038452
68, -0.006057857, 0.03547515, -0.004622262, 0.0023749352, -0.013455602, -0.018942766, 0.009053111, 0.015894342, -0.008705732, -
0.0147316875, 0.007142529, 0.01315076, -0.013228743, 0.0028676456, 0.004898747, 0.004033845, -0.01680178, -0.015752554, -0.04304
6586, -0.027123885, -0.004278428, 0.008074779, -0.00993928, 0.022076262, -0.009124004, 0.0049200146, 0.00036133575, -0.01207317
8, 0.0132003855, -0.0038069857, 0.0069901077, -0.02209044, -0.0044804746, 0.001356726, 0.013831339, 0.0025628032, 0.008138584, -
0.011335884, 0.010371732, -0.005093704, 0.0028144754, 0.007996797, -0.01271831, 0.0131365815, 0.0022756841, 0.015951056, 0.00413
66406, -0.0018467779, 0.022756841, 0.0112153655, -0.0034011197, -0.007259503, 0.00040896737, -0.0041047386, 0.0050795255, 0.0323
8419, 0.028045503, 0.023621744, -0.005728202, -0.009584812, -0.007149618, -0.008663196, 0.011357153, -0.0006473471, -0.00534537
6, 0.03595723, -0.03261105, -0.016319703, 0.0056324955, 0.019765131, 0.006391057, -0.009343775, 0.0094572045, -0.010244123, 0.01
0577323, 0.021041216, 0.026400771, -0.012335484, 0.03760196, -0.010329195, 0.019524094, -0.01691521, 0.0249829, -0.006866044, -
0.059834186, 0.0058522657, 0.005207134, -0.04298987, -0.0076210606, -0.003746726, -0.009138184, 0.01847487, 0.0009987134, 0.0175
53251, -0.0026177457, -0.0075005414, 0.05750888, 0.011860497, -0.018205473, 0.005958606, -0.0052567595, -0.0049271043, -0.024259
785, -0.0060082315, -0.0159227, 0.010953059, 0.009372132, 0.038849685, 0.00729495, 0.018886052, 0.01901366, -0.007812473, 0.0136
54104, -0.013384709, 0.017156247, 0.02906637, 0.0006349407, 0.013909321, 0.009747868, 0.0009225028, 0.027591784, -0.041515283, -
0.0027701668, -0.004778228, -0.06278336, -0.0023518947, 0.019368127, -0.019538272, 0.0156107675, -0.010648217, 0.023706814, 0.02
4713503, 0.0037112792, 0.012066089, 0.0022898628, 0.0052106786, -0.008996396, 0.027152244, -0.017609967, -0.007372933, 0.0303140
```


・・・ 中略 ・・・


```
943, -0.0056892103, 0.023465777, -0.015837627, 0.022515804, 0.025876159, -0.0062244567, -0.017709218, -0.015667483, 0.006068491,
-0.010088157, 0.011286259, -0.007571435, -0.020544961, 0.011428046, -0.019708417, 0.016645813, 0.0011936708, -0.01027957, 0.0038
956027, 0.028017145, 0.022303123, 0.02464261, 0.010322106, -0.0152563, -0.004179177, -0.026670167, -0.034964718, 0.010747467, -
0.0031494475, -0.0073020393, -0.03734674, 0.0005578439, 0.034624428, 0.008755358, -0.0059337933, 0.010747467, 0.015582411, -0.00
015142428, -0.033631917, 0.0076848646, -0.040948134, -0.013398888, -0.009478472, 0.008358354, 0.005760104, 0.011484761, 0.012406
377, -0.014065287, -0.013888054, -0.009783315, 0.036751237, -0.013795892, -0.011839229, -0.0033231368, 0.01193848, 0.021424042,
0.007592703, -0.017780112, 0.0076919543, 0.003365673, 0.0017227142, 0.000435774, 0.014122003, 0.010378821, -0.025635121, 0.00493
0649, 0.0044592065, -0.006313074, -0.001993882, 0.004001943, 0.033887133, 0.01327128, -0.00541627, -0.034340855, 0.009010575, -
0.0027949796, 0.012654505, -0.019850204, -0.020417353, -0.017936077, 0.01922634, 0.020587496, -0.011924301, 0.008656107, 0.01614
9558, -0.00397713, 0.02530901, -0.0017351205, -0.014235432, -0.021282254, 0.017099533, 0.019027838, 0.013115314, 0.014547364, -
0.0027045903, 0.016957745, 0.007436737, 0.038963113, -0.028215647, -0.012994794, 0.008691554, -0.017156247, -0.020970322, -0.023
012059, -0.017808469, -0.024132177, -0.0112153655, 0.020275565, 0.02155165, -0.027052993, 0.041260067, 0.017312214, 0.021069573,
0.0022331479, -0.023848603, -0.00015064888, 0.035758726, 0.022359837, -0.017780112, 0.01337762, 0.01758161, -0.014561542, -0.010
839629, -0.018730085, -0.022416553, -0.0075997924, -0.011555655, 0.007550167, -0.01381007, 0.00917363, 0.02065839, 0.00884752,
0.019836025, -0.0072098775, -0.03017231, -0.01348396, 0.031505108, -0.0020470524, -0.012200786, -0.012626148, 0.0069369376, 0.01
7893542, -0.021735974, -0.010818361, 0.019084554, -0.010215766, 0.001627894, -0.019368127, 0.0014905377, 0.0031246347, -0.012413
467, 0.00447693, -0.001633211, -0.017326392, 0.010563144, 0.020034527, -0.014327594, -0.023338169, -0.014228343], index=0, objec
t='embedding')], model='text-embedding-ada-002-v2', object='list', usage=Usage(prompt_tokens=1, total_tokens=1))
```

CreateEmbeddingResponse(data=[Embedding(embedding=[-0.0070539117, -0.01734057, -0.009698243, -0.030739458, -0.0124843605, 0.0030714646, -0.0051114275, -0.04111828, -0.014561542, -0.021268075, 0.019240519, 0.050759807, -0.0012246867, 0.0025521691, -0.03845268, -0.006057857, 0.03547515, -0.004622262, 0.0023749352, -0.013455602, -0.018942766, 0.009053111, 0.015894342, -0.008705732, -0.0147316875, 0.007142529, 0.01315076, -0.013228743, 0.0028676456, 0.004898747, 0.004033845, -0.01680178, -0.015752554, -0.043046586, -0.027123885, -0.004278428, 0.008074779, -0.00993928, 0.022076262, -0.009124004, 0.0049200146, 0.0003613357 5, -0.012073178, 0.0132003855, -0.0038069857, 0.0069901077, -0.02209044, -0.0044804746, 0.001356726, 0.013831339, 0.0025628032
...

...

0.017780112, 0.01337762, 0.01758161, -0.014561542, -0.010839629, -0.018730085, -0.022416553, -0.0075997924, -0.011555655, 0.007550167, -0.01381007, 0.00917363, 0.02065839, 0.00884752, 0.019836025, -0.0072098775, -0.03017231, -0.01348396, 0.031505108, -0.0020470524, -0.012200786, -0.012626148, 0.0069369376, 0.017893542, -0.021735974, -0.010818361, 0.019084554, -0.010215766, 0.001627894, -0.019368127, 0.0014905377, 0.0031246347, -0.012413467, 0.00447693, -0.001633211, -0.017326392, 0.010563144, 0.020034527, -0.014327594, -0.023338169, -0.014228343], index=0, object='embedding')], model='text-embedding-ada-002-v2', object='list', usage=Usage(prompt_tokens=1, total_tokens=1))

“猫”の embedding (ada-002)

```
# "cat" のembedding

from openai import OpenAI
client = OpenAI()

client.embeddings.create(
  model="text-embedding-ada-002",
  input="猫",
  encoding_format="float"
)
```

“猫”のembedding (ada-002) 1536次元


```
CreateEmbeddingResponse(data=[Embedding(embedding=[-0.0070539117, -0.01734057, -0.009698243, -0.030739458, -0.0124843605, 0.0030
714646, -0.0051114275, -0.04111828, -0.014561542, -0.021268075, 0.019240519, 0.050759807, -0.0012246867, 0.0025521691, -0.038452
68, -0.006057857, 0.03547515, -0.004622262, 0.0023749352, -0.013455602, -0.018942766, 0.009053111, 0.015894342, -0.008705732, -
0.0147316875, 0.007142529, 0.01315076, -0.013228743, 0.0028676456, 0.004898747, 0.004033845, -0.01680178, -0.015752554, -0.04304
6586, -0.027123885, -0.004278428, 0.008074779, -0.00993928, 0.022076262, -0.009124004, 0.0049200146, 0.00036133575, -0.01207317
8, 0.0132003855, -0.0038069857, 0.0069901077, -0.02209044, -0.0044804746, 0.001356726, 0.013831339, 0.0025628032, 0.008138584, -
0.011335884, 0.010371732, -0.005093704, 0.0028144754, 0.007996797, -0.01271831, 0.0131365815, 0.0022756841, 0.015951056, 0.00413
66406, -0.0018467779, 0.022756841, 0.0112153655, -0.0034011197, -0.007259503, 0.00040896737, -0.0041047386, 0.0050795255, 0.0323
8419, 0.028045503, 0.023621744, -0.005728202, -0.009584812, -0.007149618, -0.008663196, 0.011357153, -0.0006473471, -0.00534537
6, 0.03595723, -0.03261105, -0.016319703, 0.0056324955, 0.019765131, 0.006391057, -0.009343775, 0.0094572045, -0.010244123, 0.01
0577323, 0.021041216, 0.026400771, -0.012335484, 0.03760196, -0.010329195, 0.019524094, -0.01691521, 0.0249829, -0.006866044, -
0.059834186, 0.0058522657, 0.005207134, -0.04298987, -0.0076210606, -0.003746726, -0.009138184, 0.01847487, 0.0009987134, 0.0175
53251, -0.0026177457, -0.0075005414, 0.05750888, 0.011860497, -0.018205473, 0.005958606, -0.0052567595, -0.0049271043, -0.024259
785, -0.0060082315, -0.0159227, 0.010953059, 0.009372132, 0.038849685, 0.00729495, 0.018886052, 0.01901366, -0.007812473, 0.0136
54104, -0.013384709, 0.017156247, 0.02906637, 0.0006349407, 0.013909321, 0.009747868, 0.0009225028, 0.027591784, -0.041515283, -
0.0027701668, -0.004778228, -0.06278336, -0.0023518947, 0.019368127, -0.019538272, 0.0156107675, -0.010648217, 0.023706814, 0.02
4713503, 0.0037112792, 0.012066089, 0.0022898628, 0.0052106786, -0.008996396, 0.027152244, -0.017609967, -0.007372933, 0.0303140
```


・・・ 中略 ・・・


```
8733, 0.015948033, 0.031079568, 0.012589859, 0.022690719, 0.011872131, 0.0046849814, 0.010614462, -0.024969008, -0.04266856, 0.0
45434114, 0.019108666, -1.7824865e-05, -0.021031385, 0.016659174, 0.011121481, 0.0090473145, 0.00086423586, -0.013775096, 0.0014
609701, 0.0022684133, -0.025021685, 0.038217332, -0.044327892, -0.01579, -0.00428661, 7.7935554e-05, -0.009244855, -0.008059616,
0.022150777, -0.014183345, -0.02627277, -0.014196514, 0.04717246, -0.0011835916, -0.0065188077, -0.0043623336, 0.004227348, 0.03
8375363, 0.023112137, -0.020504612, -0.012016994, 0.002446199, 0.022756565, 0.0062620062, 0.012609612, 0.008066202, 0.003822391
6, 0.007104842, 0.025601136, -0.012260626, -0.0026420925, 0.017475672, 0.02473196, -0.008112294, -0.0074472437, -0.022229793, -
0.006640624, -0.018120969, -0.0047475356, -0.025074363, -0.015697815, -0.010983203, 0.01598754, 0.0251007, -0.0013663158, 0.0081
97894, 0.026694188, -0.0062587135, 0.0076447832, -0.012649121, -0.01485498, -0.01996467, 0.036478985, 0.02913051, 0.019319376,
0.025825014, -0.019319376, 0.012346227, 0.002255244, 0.0066900086, -0.035530794, 0.009527994, 0.014433562, -0.0046882736, -0.020
109534, -0.009771626, -0.016698683, -0.033239335, 0.0032659885, 0.015197381, 0.03065815, -0.019398391, 0.032949608, 0.002121904
9, -0.004872644, 0.0017548103, 0.0021087355, 0.01542126, 0.027365822, 0.0033104348, -0.013057369, 0.006545146, 0.016764529, -0.0
25640642, 0.001470847, 0.003776299, -0.040324423, 0.0006967387, -0.016777698, 0.003684114, -0.0023935218, -0.016237756, 0.046250
61, 0.004451226, 0.012708383, -0.0011259759, -0.021531818, -0.0049549523, 0.021597665, -0.0064661303, -0.0006103151, -0.00156138
62, 0.022756565, 0.007308966, -0.01352488, -0.024086665, 0.031132245, -0.0067393933, -0.0021186124, -0.0018568725, 0.008507373,
0.0067821937, -0.017804904, 0.007802815, -0.019740794, -0.012952015, 0.03445091, 0.018595064, -0.0056891413, -0.0033499429, -0.0
2686539], index=0, object='embedding')], model='text-embedding-ada-002-v2', object='list', usage=Usage(prompt_tokens=3, total_to
kens=3))
```

CreateEmbeddingResponse(data=[Embedding(embed
ding=[-0.0075657675, -0.009231685, -0.009896735,
-0.0043261177, -0.01141779, 0.006953395, -
0.011760192, -0.025048025, -0.006953395, -
0.019925164, 0.013998975, 0.030026022,
0.003624852, 0.00029260557, -0.033107642,
0.015368583, 0.014525747, 0.0024527838,
0.02949925, -0.03769056, -0.0061138514, -
0.005004337, -0.009396302, -0.021531818, -
0.014183345, 0.004609258, 0.018963804, -
0.023625739, 0.0066175773, -0.0034371894,
0.022400994, -0.017818075, -0.012128933, -
0.032054096, -0.009001222, -0.020346582,
0.0014280468, -0.0114968065, 0.006127021, -
0.008922206, 0.001449447, 0.010798832, -
0.009699196, 0.01997784, 0.0015679707, -
0.004365626, -0.023270167, -0.00704558, -
0.012003824, 0.013254909, 0.00012819498,

...

0.013057369, 0.006545146, 0.016764529, -0.025640642, 0.001470847, 0.003776299, -0.040324423, 0.0006967387, -0.016777698, 0.003684114, -0.0023935218, -0.016237756, 0.04625061, 0.004451226, 0.012708383, -0.0011259759, -0.021531818, -0.0049549523, 0.021597665, -0.0064661303, -0.0006103151, -0.0015613862, 0.022756565, 0.007308966, -0.01352488, -0.024086665, 0.031132245, -0.0067393933, -0.0021186124, -0.0018568725, 0.008507373, 0.0067821937, -0.017804904, 0.007802815, -0.019740794, -0.012952015, 0.03445091, 0.018595064, -0.0056891413, -0.0033499429, -0.02686539], index=0, object='embedding')], model='text-embedding-ada-002-v2', object='list', usage=Usage(prompt_tokens=3, total_tokens=3))

embeddingから意味を読み取ることは困難
ただ、それらの「意味の近さ」は知ることができる

embeddingの数字の並びから、人間がその意味を読み取ることは、困難です。

ただ、二つのembeddingが与えられた時、そのembeddingが表現する「意味の近さ」を知ることはできます。

embeddingはベクトルなので、二つのembedding A, B の内積 $A \cdot B$を考えることができます。ベクトルA, Bのなす角をθ とすると次の式が成り立ちます。

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

「意味の近さ」を表す cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$ を、cosine similarityといいます。

cosine similarityは、−1（ベクトルが完全に反対）から+1（ベクトルが完全に同一）までの値をとります。0はベクトルが直交していて相関がないことを表します。

cosine similarityの値は、二つのembeddingの「意味の近さ」（あるいは、「意味の遠さ」）の程度を表すと解釈できます。それが1に近いほど、意味が近いと考えられるのです。

“cat” と “猫” のembedding のcosine similarity を計算してみましょう。

“cat” と “猫” のembedding (ada-002) はよく似ている

```
text_ja = '猫'
text_en = "cat"

client = OpenAI()
resp = client.embeddings.create(
    model="text-embedding-ada-002",
    input=[text_ja, text_en],
    encoding_format="float",
)

emb_ja = resp.data[0].embedding
emb_en = resp.data[1].embedding
```

```
# cosine similarity
    sim = cosine_similarity(emb_ja, emb_en)

    print("Japanese:", text_ja)
    print("English :", text_en)
    print(f"Embedding dim (JA): {len(emb_ja)}")
    print(f"Embedding dim (EN): {len(emb_en)}")
    print(f"Cosine similarity : {sim:.6f}")
```

```
## model="text-embedding-ada-002
Japanese: 猫
English : cat
Embedding dim (JA): 1536
Embedding dim (EN): 1536
Cosine similarity : 0.845886
```

絵文字 🐈 と'cat'の embeddingもよく似ている

面白いのは、絵文字 🐈 と'cat'のembeddingもよく似ていることもわかります。

---

model="text-embedding-ada-002

絵文字: 🐈

English : cat

Embedding dim (JA): 1536

Embedding dim (EN): 1536

Cosine similarity : 0.851908

文の意味 = 文のembedding

これまでは、’cat’ や ‘猫’ といった語のembeddingを見てきたのですが、文の意味を表す文のembeddingを取得することができます。

“I am a cat” と “吾輩は猫である。” の embedding (ada-002)の類似度

```
text_ja = '吾輩は猫である。'
text_en = "I am a cat."

client = OpenAI()

resp = client.embeddings.create(
    model=“text-embedding-ada-002”,
    input=[text_ja, text_en],
    encoding_format="float",
)

emb_ja = resp.data[0].embedding
emb_en = resp.data[1].embedding
```

```
 # cosine similarity
    sim = cosine_similarity(emb_ja, emb_en)

    print("Japanese:", text_ja)
    print("English :", text_en)
    print(f"Embedding dim (JA): {len(emb_ja)}")
    print(f"Embedding dim (EN): {len(emb_en)}")
    print(f"Cosine similarity : {sim:.6f}")
----------------------------
## model="text-embedding-ada-002
Japanese: 吾輩は猫である。
English : I am a cat.
Embedding dim (JA): 1536
Embedding dim (EN): 1536
Cosine similarity : 0.881660
```

embeddingは、一通りに決まるわけではないこと

一つ注意してほしいことがあります。
ある文のembeddingは一通りに決まるわけではないのです。

OpenAIにはOpenAIの、GoogleにはGoogleのembeddingが存在します。それだけでなく、OpenAIの中にも複数のembeddingの仕方が存在します。

これまで、OpenAIのtext-embedding-ada-002 というembeddingモデルを使ってきたのですが、同じOpenAIのtext-embedding-3-smallというembeddingモデルを使った結果を次に示します。

“I am a cat” と “吾輩は猫である。” の embedding(3-small)の類似度

```
text_ja = '吾輩は猫である。'
text_en = "I am a cat."

client = OpenAI()

resp = client.embeddings.create(
    model="text-embedding-3-small",
    input=[text_ja, text_en],
    encoding_format="float",
)

emb_ja = resp.data[0].embedding
emb_en = resp.data[1].embedding
```

```
 # cosine similarity
    sim = cosine_similarity(emb_ja, emb_en)

    print("Japanese:", text_ja)
    print("English :", text_en)
    print(f"Embedding dim (JA): {len(emb_ja)}")
    print(f"Embedding dim (EN): {len(emb_en)}")
    print(f"Cosine similarity : {sim:.6f}")
----------------------------
## model="text-embedding-3-small"
Japanese: 吾輩は猫である。
English : I am a cat.
Embedding dim (JA): 1536
Embedding dim (EN): 1536
Cosine similarity : 0.531995
```

LLMを使った翻訳 1

```
text = "吾輩は猫である。"
client = OpenAI()

response = client.responses.create(
    model="gpt-5.2",
    input=f"次の日本語を英語に翻訳してくださいT:\n\n{text}",
)
print(response.output_text)
```

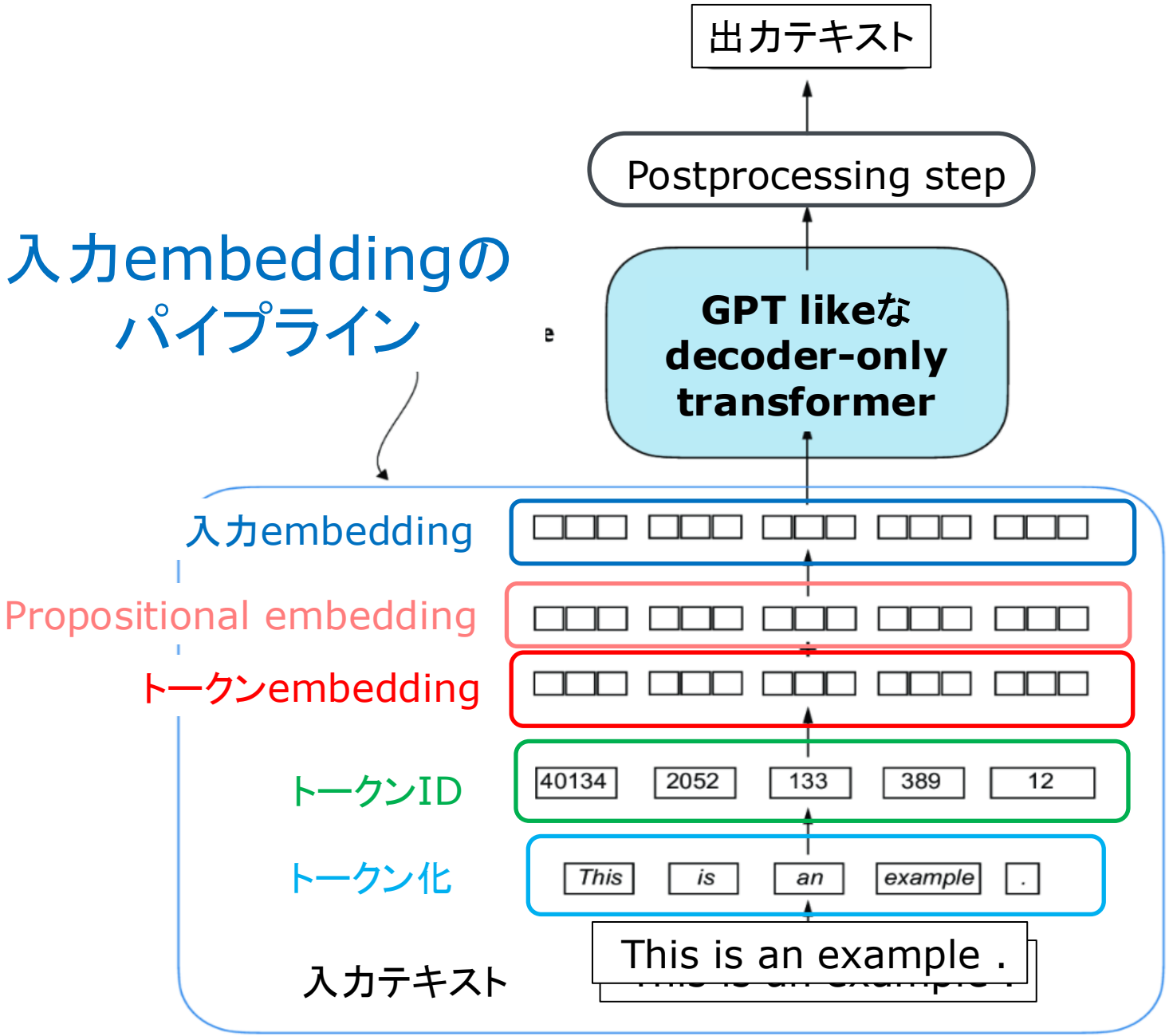
I am a cat.

LLMを使った翻訳 2

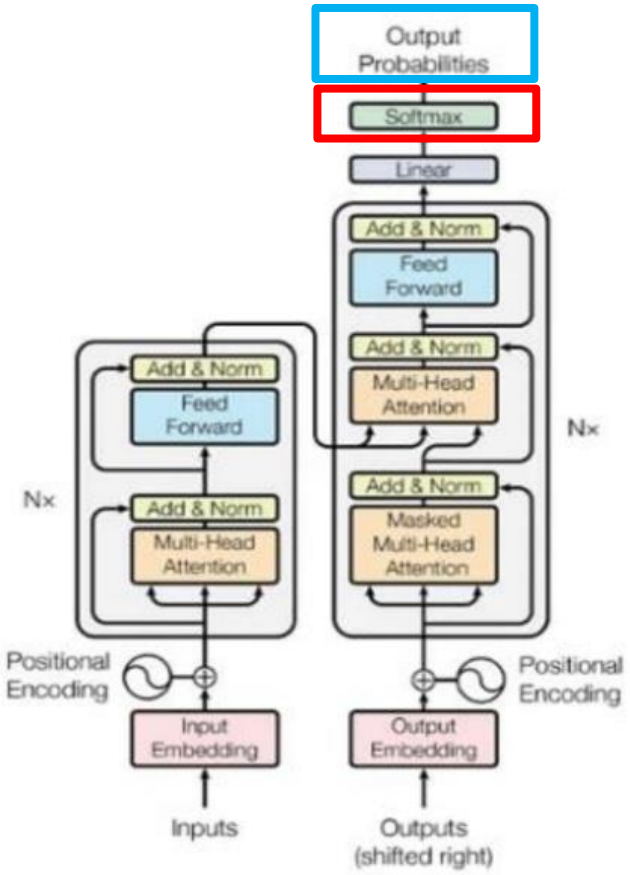
```
text = "国境の長いトンネルを抜けると雪国であった。”
client = OpenAI()

response = client.responses.create(
    model="gpt-5.2",
    input=f"次の日本語を英語に翻訳してくださいT:\n\n{text}",
)
print(response.output_text)
```

When the long tunnel at the border was passed through, it was the snow country.

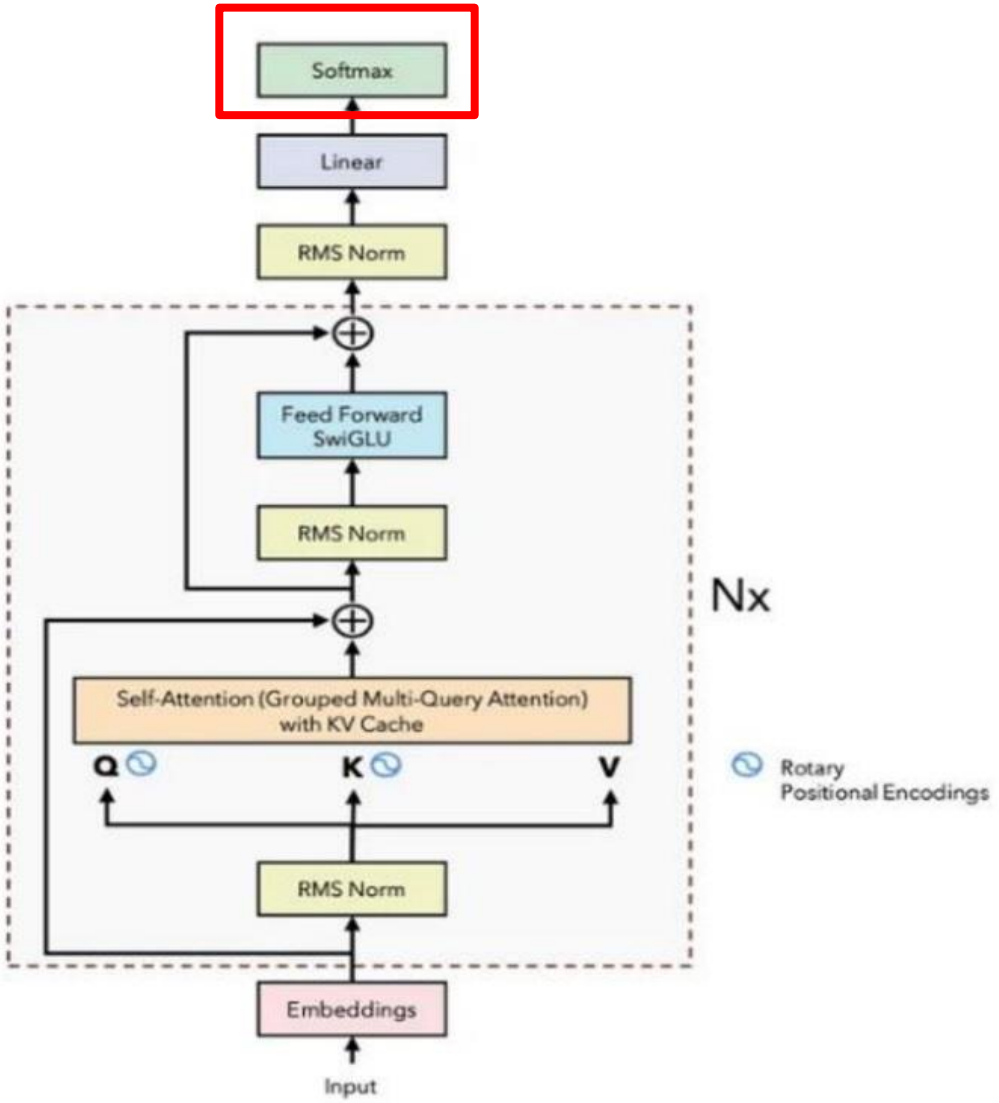
入力embeddingのパイプライン



"Build a Large Language Model (From Scratch)" から



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。

Amazon レビュー情報からの意味的検索

データを準備する

Amazon レビュー情報からの意味的検索 1

このセッションと次のセッションでは、embeddingを利用した、Amazon レビュー情報からの意味的検索の簡単なサンプル・プログラムを紹介しようと思います。

今回のセッションでは、必要なデータの準備をします。
次回のセッションでは、embeddingを利用した意味的検索プログラムを紹介します。

どのような検索が可能なのか？

このサンプル・プログラムで、どのような検索ができるのか、まず。例を見ておきましょう。

results = search_reviews(df, "spoilt", n=1)

Disappointed: The metal cover has severely disformed. And most of the cookies inside have been crushed into small pieces. Shopping experience is awful. I'll never buy it online again.

“spoilt（ダメだった）”で検索して、レビューの本文が返ってくるのですが、この単語 spoilt は、この本文に含まれているわけではありません。

“bad delivery” で検索しています。

results = search_reviews(df, "bad delivery", n=1)

great product, poor delivery: The coffee is excellent and I am a repeat buyer. Problem this time was with the UPS delivery. They left the box in front of my garage door in the middle of the drivewa

商品について、次のような検索も可能です。

results = search_reviews(df, "delicious beans", n=3)

results = search_reviews(df, "delicious beans", n=3)

Delicious!: I enjoy this white beans seasoning, it gives a rich flavor to the beans I just love it, my mother in law didn't know about this Zatarain's brand and now she is traying different seasoning

Fantastic Instant Refried beans: Fantastic Instant Refried Beans have been a staple for my family now for nearly 20 years. All 7 of us love it and my grown kids are passing on the tradition.

Delicious: While there may be better coffee beans available, this is my first purchase and my first time grinding my own beans. I read several reviews before purchasing this brand, and am extremely

元のAmazon レビューは、どのようなデータか？

次のような54万件以上のcsv形式のレビューデータです。
"data/fine_food_reviews_1k.csv"
サンプル・プログラムでは、そのうちの10,00件のデータを使います。

	A	B	C	D	E	F	G	H	I	J
名前ボックス		ProductId	UserId	ProfileName	HelpfulnessN	HelpfulnessD	Score	Time	Summary	Text
2	1	B001E4KFG0	A3SGXH7AU	delmartian	1	1	5	1303862400	Good Quality	I have bought several of the Vitality canned dog food products and have found them all to be of good
3	2	B00813GRG4	A1D87F6ZCV	dll pa	0	0	1	1346976000	Not as Adver	Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually small sized unsalted. N
4	3	B000LQOCH	ABXLMWJIX	Natalia Corre	1	1	4	1219017600	"Delight" say	This is a confection that has been around a few centuries. It is a light, pillowy citrus gelatin with nuts
5	4	B000UA0QIQ	A395BORC6F	Karl	3	3	2	1307923200	Cough Medic	If you are looking for the secret ingredient in Robitussin I believe I have found it. I got this in addition
6	5	B006K2ZZ7K	A1UQRSCLF	Michael D. Bi	0	0	5	1350777600	Great taffy	Great taffy at a great price. There was a wide assortment of yummy taffy. Delivery was very quick. If
7	6	B006K2ZZ7K	ADT0SRK1M	Twoapennyth	0	0	4	1342051200	Nice Taffy	I got a wild hair for taffy and ordered this five pound bag. The taffy was all very enjoyable with many f
8	7	B006K2ZZ7K	A1SP2KVKFX	David C. Sulli	0	0	5	1340150400	Great! Just a	This saltwater taffy had great flavors and was very soft and chewy. Each candy was individually wrap
9	8	B006K2ZZ7K	A3JRGQVEQN	Pamela G. W	0	0	5	1336003200	Wonderful, ta	This taffy is so good. It is very soft and chewy. The flavors are amazing. I would definitely recommer
10	9	B000E7L2R4	A1MZYO9TZK	R. James	1	1	5	1322006400	Yay Barley	Right now I'm mostly just sprouting this so my cats can eat the grass. They love it. I rotate it around w
11	10	B00171APVA	A21BT40VZC	Carol A. Reed	0	0	5	1351209600	Healthy Dog	This is a very healthy dog food. Good for their digestion. Also good for small puppies. My dog eats he
12	11	B0001PB9FE	A3HDKO7OW	Canadian Fan	1	1	5	1107820800	The Best Hot	I don't know if it's the cactus or the tequila or just the unique combination of ingredients, but the flavo
13	12	B0009XLVG0	A2725IB4YY	A Poeng "Spa	4	4	5	1282867200	My cats LOVI	One of my boys needed to lose some weight and the other didn't. I put this food on the floor for the c
14	13	B0009XLVG0	A327PCT23Y	LT	1	1	1	1339545600	My Cats Are I	My cats have been happily eating Felidae Platinum for more than two years. I just got a new bag and
15	14	B001GVISJM	A18ECVX2RJ	willie "roadie	2	2	4	1288915200	fresh and gre	good flavor! these came securely packed... they were fresh and delicious! i love these Twizzlers!

548235	548234	B0040Q0RDI	A1QHOOQ7C	Janice Edwar	4	6	3	1327363200	Fresh corn!	The fresh tasting corn, no additives! I can't believe Amazon can offer such good tasting corn in
548236	548235	B0040Q0RDI	A21PSVLL8F	Creamed Cor	0	0	3	1348012800	Neutral on th	Libbys Libbys Libbys.... you can do better. The sweet corn is exactly what it says it is. Its sweet
548237	548236	B0040Q0RDI	AZ9F02WPX	looneylady	1	2	5	1331683200	food	This was a great product. I like the size because i can open a can and it is the right size for me.
548238	548237	B0054E37FC	A3RF10DUE2	2 under 2	0	0	3	1339718400	Did not like fl	This drink mix works as well as products similar to it. The powder is difficult to get disolved com
548239	548238	B001E95KNI	A2X61KKXXF	Sean P. Logu	5	5	5	1197763200	Fun way to a	I'm just getting into the whole plank cooking thing, and so far I'm really enjoying it. It is a great
548240	548239	B001E95KNI	A2HDI47ZJU	Pammie P.	1	1	5	1259107200	Order them. (	I was extremely pleased when I recv'd these. They were much larger and thicker than anticipat
548241	548240	B001E95KNI	A1A0Z3FEDX	Cc23rdb	0	0	5	1337990400	Cedar grilling	I wanted to try a new way to cook meat and add more flavor to pork. I purchased a set of the ce
548242	548241	B001E95KNI	A2ANMHP9F	JK	0	0	5	1316304000	Great flavor,	I really enjoy these planks, especially for vegetables. I cook everything from beets, carrots, bro
548243	548242	B007NVH0D	A1IU7S4HCK	Joanna Dane	0	0	4	1334188800	Yes, it is a CL	This coffee, packed in the new VUE format for Keurig Vue machines, is absolutely a clone of co
548244	548243	B001EO5UV0	A3T6HT67E2	J. Brown	0	0	3	1278892800	Not really Pe	Don't buy this if you are looking for a peppermint tea as one cannot taste any peppermint as th
548245	548244	B000KJVJNS	A2SRXVZQB	Mackenzie M	5	5	5	1224115200	Everyone lov	We ordered the Ferrero Rocher as a favor during our recent wedding. Not one was left and eve
548246	548245	B000KJVJNS	ACVIOP87ES	Jill	0	0	5	1342396800	Yummy as all	These were a Birthday gift for my mother! She loves chocolate! Perfect gift for her!I have never

先頭の数行を見る

Id,ProductId,UserId,ProfileName,HelpfulnessNumerator,HelpfulnessDenominator,Score,Time,Summary,Text

1,B001E4KFG0,A3SGXH7AUHU8GW,delmartian,1,1,5,1303862400,Good Quality Dog Food,I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product looks more like a stew than a processed meat and it smells better. My Labrador is finicky and she appreciates this product better than most.

2,B00813GRG4,A1D87F6ZCVE5NK,dll pa,0,0,1,1346976000,Not as Advertised,"Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually small sized unsalted. Not sure if this was an error or if the vendor intended to represent the product as ""Jumbo""."

3,B000LQOCH0,ABXLMWJIXXAIN,"Natalia Corres ""Natalia Corres""",1,1,4,1219017600,"""Delight"" says it all","This is a confection that has been around a few centuries. It is a light, pillowy citrus gelatin with nuts - in this case Filberts. And it is cut into tiny squares and then liberally coated with powdered sugar. And it is a tiny mouthful of heaven. Not too chewy, and very flavorful. I highly recommend this yummy treat. If you are familiar with the story of C.S. Lewis' ""The Lion, The Witch, and The Wardrobe"" - this is the treat that seduces Edmund into selling out his Brother and Sisters to the Witch."

4,B000UA0QIQ,A395BORC6FGVXV,Karl,3,3,2,1307923200,Cough Medicine,If you are looking for the secret ingredient in Robitussin I believe I have found it. I got this in addition to the Root Beer Extract I ordered (which was good) and made some cherry soda. The flavor is very medicinal.

data/fine_food_reviews_1k.csv ファイルの1行目と2行目を詳しく見る

Id,ProductId,UserId,ProfileName,HelpfulnessNumerator,HelpfulnessDenominator,Score,Time,Summary,Text

1,B001E4KFG0,A3SGXH7AUHU8GW,delmartian,1,1,5,1303862400,Good Quality Dog Food,I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product looks more like a stew than a processed meat and it smells better. My Labrador is finicky and she appreciates this product better than most.

Id, ProductId, UserId, ProfileName.

1,B001E4KFG0,A3SGXH7AUHU8GW,delmartian,

HelpfulnessNumerator,HelpfulnessDenominator, Score,Time,

1,1,5,1303862400,

Summary,

Good Quality Dog Food,

Text

I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product looks more like a stew than a processed meat and it smells better. My Labrador is finicky and she appreciates this product better than most.

レビュー・データの読み込み

```
embedding_model = "text-embedding-3-small"
embedding_encoding = "cl100k_base"
max_tokens = 8000  # the maximum for text-embedding-3-small is 8191
```

```
# load & inspect dataset
input_datapath = "data/fine_food_reviews_1k.csv"  # to save space, we provide
df = pd.read_csv(input_datapath, index_col=0)
df = df[["Time", "ProductId", "UserId", "Score", "Summary", "Text"]]
df = df.dropna()
df["combined"] = (
    "Title: " + df.Summary.str.strip() + "; Content: " + df.Text.str.strip()
)
df.head(2)
```

https://github.com/openai/openai-cookbook/blob/365dfaa2ef36e0a6b7639ba8d211a451e0e90455/examples/Get_embeddings_from_dataset.ipynb

df.head(2) の出力

	Time	ProductId	UserId	Score	Summary	Text	combined
0	1351123200	B003XPF9BO	A3R7JR3FMEBXQB	5	where does one start...and stop... with a tre...	Wanted to save some to bring to my Chicago fam...	Title: where does one start...and stop... wit...
1	1351123200	B003JK537S	A3JBPC3WFUT5ZP	1	Arrived in pieces	Not pleased at all. When I opened the box, mos...	Title: Arrived in pieces; Content: Not pleased...

```
df = df[["Time", "ProductId", "UserId", "Score", "Summary", "Text"]]
```

	Time	ProductId	UserId	Score	Summary	Text	combined
0	1351123200	B003XPF9BO	A3R7JR3FMEBXQB	5	where does one start...and stop... with a tre...	Wanted to save some to bring to my Chicago fam...	Title: where does one start...and stop... wit...
1	1351123200	B003JK537S	A3JBPC3WFUT5ZP	1	Arrived in pieces	Not pleased at all. When I opened the box, mos...	Title: Arrived in pieces; Content: Not pleased...

```
df["combined"] = (
    “Title: ” + df.Summary.str.strip() +
    "; Content: " + df.Text.str.strip()
```

サンプル・プログラムの情報
OpenAI API doc

https://developers.openai.com/api/docs/guides/embeddings#embedding-models

Vector embeddings

Learn how to turn text into numbers, unlocking use cases like search.

New embedding models

`text-embedding-3-small` and `text-embedding-3-large`, our newest and most performant embedding m
now available. They feature lower costs, higher multilingual performance, and new parameters to control the c

What are embeddings?

OpenAI's text embeddings measure the relatedness of text strings. Embeddings are commonly used fo

- **Search** (where results are ranked by relevance to a query string)
- **Clustering** (where text strings are grouped by similarity)
- **Recommendations** (where items with related text strings are recommended)
- **Anomaly detection** (where outliers with little relatedness are identified)
- **Diversity measurement** (where similarity distributions are analyzed)
- **Classification** (where text strings are classified by their most similar label)

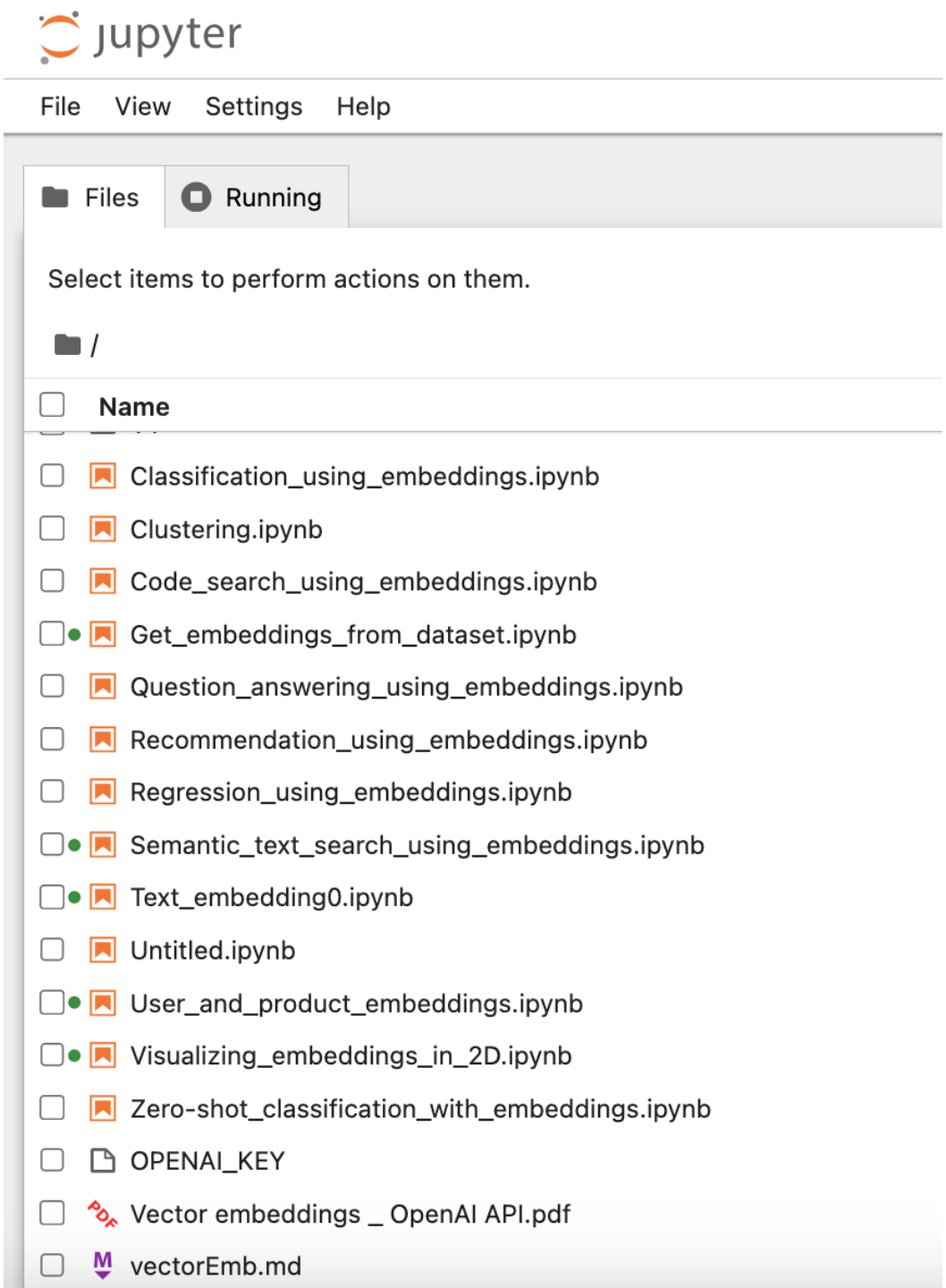
Github
OpenAI-cookbook

Semantic text search using embeddings

https://github.com/openai/openai-cookbook/blob/main/examples/Semantic_text_search_using_embeddings.ipynb

Get embeddings from dataset

https://github.com/openai/openai-cookbook/blob/365dfaa2ef36e0a6b7639ba8d211a451e0e90455/examples/Get_embeddings_from_dataset.ipynb


jupyter
File View Settings Help
Files
Running
Select items to perform actions on them.
/
Name
Classification_using_embeddings.ipynb
Clustering.ipynb
Code_search_using_embeddings.ipynb
Get_embeddings_from_dataset.ipynb
Question_answering_using_embeddings.ipynb
Recommendation_using_embeddings.ipynb
Regression_using_embeddings.ipynb
Semantic_text_search_using_embeddings.ipynb
Text_embedding0.ipynb
Untitled.ipynb
User_and_product_embeddings.ipynb
Visualizing_embeddings_in_2D.ipynb
Zero-shot_classification_with_embeddings.ipynb
OPENAI_KEY
Vector embeddings _ OpenAI API.pdf
vectorEmb.md

embeddingを利用したテキストの意味的検索

embeddingを利用したテキストの検索

今回のセッションでは、embeddingを利用した意味的検索がどのようにして行われるのかを、見ていきたいと思います。

基本的なアイデアは、文字列の集まりAの中から、ある特定の文字列 x を含むAの要素を探し出す「文字列検索」のスタイルから、意味の集まりA'の中から、ある意味x'に意味的に近いものを探すという「意味検索」に切り替えるというものです。

文字列検索と意味の検索

我々は、文字列の検索に慣れすぎているので、意味の検索と言ってもあまりイメージが湧かないかもしれません。ここでは、簡単な例で文字列の検索と意味の検索を比較してみようと思います。

まずは文字列検索から。文字列の集まりAを次のものとしましょう。

```
A = {
      “AIは素晴らしい！”,
      “うちの猫はタマと言います”,
      “うちのペットはいつも寝ています”,
      “金魚は夜は休むのかな？”
    }
```

Aに対して文字列“猫”で検索をかけると、“うちの猫はタマと言います”がヒットします。ただ、文字列“ねこ”で検索をかけても何もヒットしません。文字列“ねこ”は、Aの文字列の集まりには含まれていないからです。

意味の検索は、それとは違います。意味の検索の対象となる集まりA’は、次のものからなります。

A’ = {
“AIは素晴らしい！” の意味,
“うちの猫はタマと言います”の意味 ,
“うちのペットはいつも寝ています”の意味 ,
“金魚は夜は休むのかな？”の意味
}

意味の集まりA′に対して"猫"で「意味の検索」するというのは、意味の集まりA′の中から、"猫"の意味に近いものを探すということです。

きっと "うちの猫はタマと言います"だけでなく、"うちのペットはいつも寝ています"を、意味の近さから見つける可能性が生まれます。

文字列の検索とは異なって、"猫"の意味と"ねこ"の意味は、ほとんど同じですから、"ねこ"での意味検索も"猫"での意味検索と同じ結果になるはずです。

意味の集まりA'に対して、「寝ている」で意味検索をかけたとしましょう。

この時、文字列のレベルでは近い
"うちのペットはいつも寝ています"だけでなく、

"金魚は夜は休むのかな？"や
"うちの猫はタマと言います"

までヒットする可能性があります。

文字列の検索より、意味の検索の方が柔軟で強力です。

文字列の検索から意味の検索へ

文字列の検索から意味の検索へ移行するのに必要なことをまとめてみましょう。

- まず、検索の対象を、文字列の集まりAから意味の集まりA′に変換します。
- 次に検索すべきクエリー文字列xを、xの意味x′ に変換します。
- そして、A′の要素をひとつづつ、x′とどれくらい意味が近いかを計算します。
- A′の要素全てとx′の意味の近さの計算が終わったら、最後に、意味が近いものを何個か選びます。

文字列の意味は embeddingで表現され、意味の近さは、embeddingのcosine similarityで計算できますので、先の文字列けの検索から意味の検索への移行は、次のようなプロセスで実行できることがわかります。

- 検索の対象である文字列の集まりAから、そのembeddingの集まりA′を作成します。
- 検索すべき文字列xから、xのembedding x′を作成します。
- A′の要素をひとつづつ、x′とどれくらい意味が近いかをcosine similarityで計算します。
- A′の要素全てとx′のcosine similarityの計算が終わったら、最後に、意味が近いものを何個か選びます。

Googleの検索とはなんであったのか

話は飛ぶのですが、文字列検索の王者はGoogleです。

Googleの検索は、Googleのcrawlerがインターネット上で網羅的に収集した文字列の巨大な集まりAから、文字列xを含むAの要素のリストを作ることから始まります。

それは、ある言葉が本の何ページ目に出てくるかをまとめる、本の末尾の索引（Index）ページを作る作業に似ています。

ただ、Googleの検索の働きは逆で、ある言葉が与えられた時、その言葉が出てくるページを探します。同じIndexづくりでも、それを「逆引きIndex」ということがあります。

Googleの検索は、「逆引きIndex」で作成された、ある文字列xを含む文字列のリストに優先度をつける「ページ・ランク」アルゴリズムとそれを計算する大規模分散システム「MapReduce」によって支えられていました。

加えて、その巨大な文字列検索システムは「広告モデル」という、ビジネスモデルと結びついていました。

Webスケールに対応した「文字列の検索」技術は、21世紀初頭のIT技術の進化を牽引した技術です。

問題は、この「文字列検索」から「意味検索」への変化が、今まさに進行中だということです。そのインパクトについては、このセミナーの次のパートで触れたいと思います。

前回作成した、データ
data/fine_reviews_1k.csv

```
embedding_model = "text-embedding-3-small"
embedding_encoding = "cl100k_base"
max_tokens = 8000  # the maximum for text-embedding-3-small is 8191
```

```
# load & inspect dataset
input_datapath = "data/fine_food_reviews_1k.csv"  # to save space, we provide
df = pd.read_csv(input_datapath, index_col=0)
df = df[["Time", "ProductId", "UserId", "Score", "Summary", "Text"]]
df = df.dropna()
df["combined"] = (
    "Title: " + df.Summary.str.strip() + "; Content: " + df.Text.str.strip()
)
df.head(2)
```

https://github.com/openai/openai-cookbook/blob/365dfaa2ef36e0a6b7639ba8d211a451e0e90455/examples/Get_embeddings_from_dataset.ipynb

```
df = df[["Time", "ProductId", "UserId", "Score", "Summary", "Text"]]
```

	Time	ProductId	UserId	Score	Summary	Text	combined
0	1351123200	B003XPF9BO	A3R7JR3FMEBXQB	5	where does one start...and stop... with a tre...	Wanted to save some to bring to my Chicago fam...	Title: where does one start...and stop... wit...
1	1351123200	B003JK537S	A3JBPC3WFUT5ZP	1	Arrived in pieces	Not pleased at all. When I opened the box, mos...	Title: Arrived in pieces; Content: Not pleased...

```
df["combined"] = (
    “Title: ” + df.Summary.str.strip() +
    "; Content: " + df.Text.str.strip()
```

検索対象をembedding化したデータの作成

data/fine_food_reviews_with_embeddings_1k.csv

データに”combined”フィールドから作成した”embedding“というフィールドを追加し、それを新しいcsvファイルとして保存します。

```
df["embedding"] = df.combined.apply(lambda x: get_embedding(x, model=embedding_model))
df.to_csv("data/fine_food_reviews_with_embeddings_1k.csv")
```

検索対象をembedding化したデータの作成
data/fine_food_reviews_with_embeddings_1k.csv

データに”combined”フィールドから作成した”embedding“というフィールドを追加し、それを新しいcsvファイルとして保存します。

```
df[“embedding”] =
        df.combined
        .apply(lambda x:
get_embedding(x,model=embedding_model))

df.to_csv("data/fine_food_reviews_with_embeddings_1k.csv")
```

get_embedding

get_embedding は、openai/embeddings_utils.py で次のように定義されています。float のリストを返します。

```
def get_embedding(text: str, engine="text-similarity-davinci-001", **kwargs) -> List[float]:

    # replace newlines, which can negatively affect performance.
    text = text.replace("\n", " ")

    return openai.Embedding.create(input=[text], engine=engine, **kwargs)["data"][0]["embedding"]
```

データのembeddingの形式を変換する

cosine_similarityを利用するために、List形式のembeddingをarray形式に変換します。literal_evalは型変換を安全に行うために挿入されています。

```
import pandas as pd
import numpy as np
from ast import literal_eval

datafile_path =
"data/fine_food_reviews_with_embeddings_1k.csv"

df = pd.read_csv(datafile_path)
df["embedding"] =
df.embedding.apply(literal_eval).apply(np.array)
```

search_reviews

search_reviewsは、ここでは product_descriptionと呼ばれているクエリと文書の埋め込み表現のcosine_similarityを比較し、上位 top_n 件の最適な一致結果を表示します。

```
from utils.embeddings_utils import get_embedding, cosine_similarity

def search_reviews(df, product_description, n=3, pprint=True):

    #クエリーのembeddingを求めている
    product_embedding = get_embedding(
        product_description,
        model="text-embedding-3-small"
    )
```

embeddingフィールドのそれぞれのembeddingとクエリーのembeddingとのcosine_similarity を求め、それをsimilarityフィールドに書き込んでいる。このフィールドは sortに利用される。

```
df["similarity"] = df.embedding.apply(lambda x: cosine_similarity(x, product_embedding))

results = (
    df.sort_values("similarity", ascending=False)
    .head(n)
    .combined.str.replace("Title: ", "")
    .str.replace("; Content:", ": ")
)
if pprint:
    for r in results:
        print(r[:200])
        print()
return results
```


Part 2 の終わり

Part 3

embeddingと検索技術の新しい展開

Part 3 Agenda
embeddingと検索技術の新しい展開

文字列検索と画像検索

Vector Search の紹介

マトリョーシカ -- embedding技術の革新


文字列検索と画像検索
Vector Search とembedding

文字列検索と画像検索
Vector Searchとembedding

これまで、文字列検索へのembeddingの利用を紹介してきたのですがembeddingの検索は、もっと広い応用分野をもっています。

次回のセッションでは、ベクトルで表現されているembeddingを検索するVector Search技術を見ていこうと思います。このVector Search技術は、元はGoogleの画像検索で用いられていた技術です。

文字列と画像はずいぶん違うものなので、両者が同じ検索技術で統合できるというのは意外かもしれません。このセッションでは、Vector Searchへの導入編として、その背景を説明したいと思います。

あるものを数字のベクトルで表す

その統合の鍵は、あるものを数字のベクトルで表すというアイデアにあります。

すでに、見てきたように現代のAIの中心的な技術は、意味を多次元の数字のベクトルで表すというembedding技術です。

ただ、数字のベクトルで表現されるのは、こうした意味のembeddingだけではありません。

身近な例で言えば、図形はベクトルで表現できます。次にそれを見てみましょう。

画像はベクトルで表現できる

先に文字列の「意味」を、数字からなる多次元のベクトルという表現で表すというembedding技術を見てきたのですが、そこでの認識の飛躍と比べると、画像がベクトルで表現されるというのは、驚きは少ないかもしれません。

例えば、64 x 64 のマス目に白と黒のオセロのチップを置いて、模様を描くことができます。マス目一つを1ビットが占める場所とし、白いチップが0、黒いチップが1 だと考えると、この白黒の模様は、横に1バイトの列が8個並んでいて、それが縦に64段積み重なったものと考えることができます。これは、白黒画像をバイトの列で表現していることと同じことです。

1024 x 1024 のカラー画像も、RGBに分解すれば、1024 x 1024ビットで表現されるベクトルが、RGBの分の3個分あれば表現できます。

ここでは、画像を離散的なものとして「デジタイズ」したことが大きく効いています。例えば、円や球は、連続的なものなので、必ずしも単純に数字の並び、有限次元の数字のベクトルとしてとして表現されるとは限りません。

言葉の意味がベクトル表現にたどり着いた経路と、画像がデジタイズによってベクトル表現にたどり着いた経路は、このように異なるのですが、最終的にはいずれもベクトルによる表現に落ち着いたのは面白いことです。これらを一括して、embedding表現と考えることができます。

同じ文字列の検索と
意味が一致する文字列の検索

文字列の意味のベクトル表現と画像のベクトル表現を、embeddingとしてひとくくりにすることに抵抗がある人もいると思います。気持ちはわかります。ただ、この二つのベクトル表現を検索との関係で捉え直すと、意外な共通性があることに気づきます。

文字列の検索では、二つの文字列が「一致」することが基本です。また、その「一致」は容易に検出可能であるように見えます。文字列の一致を条件とすると、文字列"猫"の検索は、文字列"ねこ"の検索とも文字列"cat"の検索とも違うものになります。

一方、文字列ではなくその意味での検索を考えると、文字列"猫"の意味の検索は、文字列"ねこ"の意味の検索とも文字列"cat"の意味の検索とも一致するように見えます。

それでは、意味の検索は「意味の一致」を条件にしているのでしょうか？ 確かに、先の「猫」「ねこ」「cat」の意味の検索の例は、等しい意味の検索のように見えます。

ところが、よく考えてみると、二人の人間が、同じ文字列「ねこ」を使ったとしても、私にとっての「ねこ」の意味と、あなたにとっての「ねこ」の意味が一致しているとは限りません。

意味の世界では、「意味の一致」とは何かを、説明することはとても難しいのです。

同じ画像の検索と
イメージが一致する画像の検索

今度は、文字列ではなく、同じ画像を検索することを考えてみましょう。ここでも、文字列の検索と同じように、二つの画像が「一致」の検出が基本であるように思えます。

RGBのカラーで表現される1024 x 1024ドットの画像があったとします。「一致」を条件にする限り、二つの画像が1ドットだけ違っていても、二つの画像は違うものになります。

その不一致に気づくためには文字列の不一致に気づくより、はるかに多くのチェックが必要になります。しかし、こうした違いは表面的なものです。

先に、文字列に対して、その文字列の意味を考えたように、今度は、画像に対してその「画像の意味」にあたるものを考えてみましょう。

文字列とその意味の場合、意味とは文字列によって「表現」されているものです。文字列は意味の「表現」です。画像の意味にあたるものは、画像によって表現されているものと考えることにしましょう。

猫の画像によって表現されているものは、その画像を生み出している猫の姿・かたちの頭の中の「イメージ」です。画像を生み出す頭の中の「イメージ」は、表現としての画像のように具体的なものではなく、意味と同じように抽象的な世界に存在します。

意味の世界で「意味の一致」を語るのが難しかったように、イメージの世界でも、「イメージの一致」を語ることは、簡単ではありません。

「一致」の世界から「近似」の世界へ

ただ、探すべきものに厳密に一致するものを探すことをやめ、探すべきものに近いものを探すことにすると、検索の世界は、全く異なるアプローチを指針とする、全く新しいスタイルの検索へと変化します。

この「一致」の検索から「近似」の検索への変化は、一見すると正確な検索から曖昧さを含んだ検索への後退のように感じられるかもしれません。

しかし、そうした認識は、意味の検索やイメージの検索に関して言えば、正しいものではありません。なぜなら、意味やイメージの世界には、有限なアルゴリズムで計算可能な「完全な一致」の概念は、そもそも存在しないからです。

近似を計算可能にするベクトル表現

ある領域で、一致を原理とする検索が不可能なら、そこでは、近似的検索を正確な検索に対する便宜的なものと考える必要はありません。そこでは、近似こそが検索の本質なのです。

その意味では、近似的検索にとって、近似を容易に計算可能にするベクトル表現の採用は、本質的に重要な意味を持っています。

Vector検索は、デジタイズ化された画像検索が先行しましたが、そのアプローチは、embeddingによる意味検索とアプローチは同じものです。

検索の世界の飛躍的拡大と高速化

重要なことは、Vector Searchとembedding技術が、今までの検索技術の対象領域を飛躍的に拡大しつつあることです。

それは、次回のセッションで紹介するGoogleのblogのタイトル “Find anything blazingly fast with Google‘s vector search technology” によく表れています。

注意して欲しいのは、“Find anything blazingly fast" という表現です。

「どんなものでも、驚くほど早く見つけ出す」

ここで述べられているのは、単なる画像検索技術の話ではないのです。「どんなものでも、驚くほど早く見つけ出す」技術が登場しているのです。

マルチ・モーダルなAIの時代のembedding検索

多様な対象に対応したVector Search は、マルチ・モーダルなAIの時代の、embeddingを用いた「意味の検索」の基本技術になろうとしています。

そこでは、文字”猫”と"ねこ"の違いや、本物の猫と絵に描かれた猫の違いを超え、猫の鳴き声や猫の思い出も、一緒に「猫」の意味に包んでしまう検索が行われるでしょう。 我々が、記憶の中で、猫を想起するように。

それはまた、ビジネスの世界に、大きな変化をもたらすことになると思います。


Vector Search
embeddingを検索する
2024年3月マルレク「マトリョーシカとトロピカル」から再掲

Vector Search
embeddingを検索する技術

このセッションでは、ベクトルで表現されているembeddingを検索するVector Search技術を見ていこうと思います。

これまで、文字列検索へのembeddingの利用を紹介してきたのですがembeddingの検索は、もっと広い応用分野をもっています。今回紹介するVector Search技術は、元はGoogleの画像検索で用いられていた技術です。

以下は、全面的に、次のblog記事の紹介です。

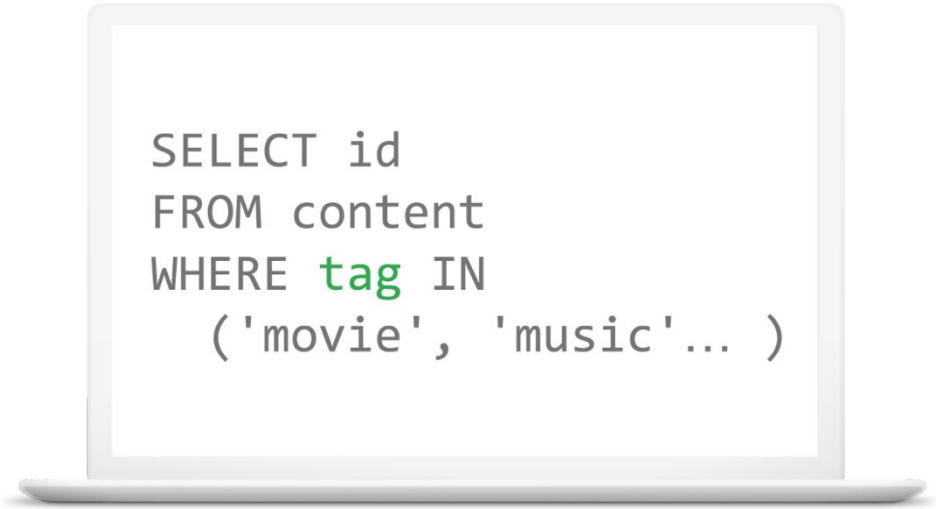
“Find anything blazingly fast with Google's vector search technology”

https://cloud.google.com/blog/topics/developers-practitioners/find-anything-blazingly-fast-googles-vector-search-technology

キーワード検索

「長年にわたり、リレーショナル・データベースと全文検索エンジンは、現代のITシステムにおける情報検索の基盤となってきた。

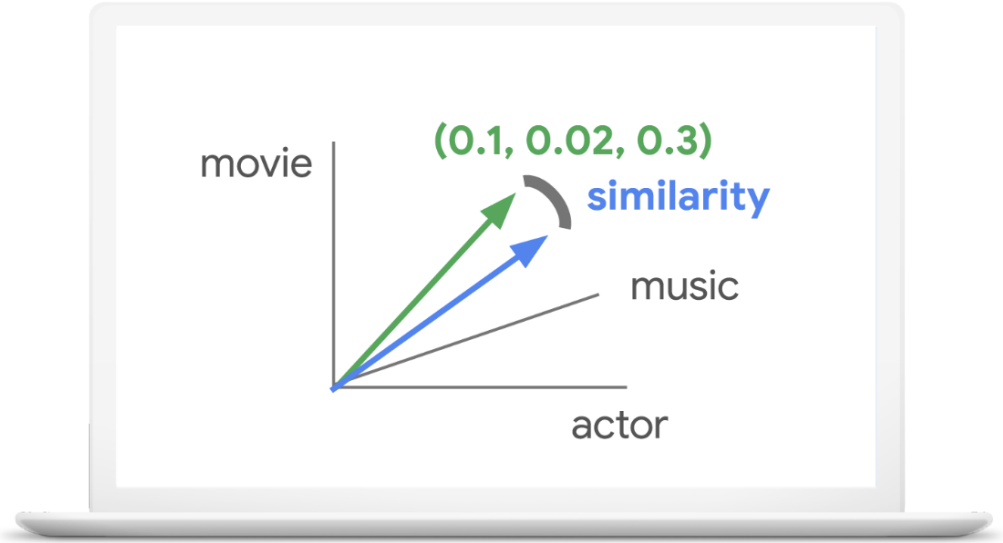
例えば、各コンテンツ（画像やテキスト）や各エンティティ（製品、ユーザー、IoTデバイス、その他何でも）に、「映画」「音楽」「俳優」といったタグやカテゴリーキーワードを追加する。そして、それらのレコードをデータベースに追加し、それらのタグやキーワードで検索を実行できるようにする。」



ベクトル類似検索

「対照的に、ベクトル検索はベクトル（各ベクトルは数字のリスト）を使ってコンテンツを表現し、検索する。数字の組み合わせによって、特定のトピックとの類似性が定義される。

例えば、ある画像（または任意のコンテンツ）に、「映画」が10%、「音楽」が2%、「俳優」関連のコンテンツが30%含まれている場合、それを表すために[0.1, 0.02, 0.3]というベクトルを定義することができる。」



Vector Searchはビジネスを変える

「ベクトル検索は、画像やテキストコンテンツだけに適用できるわけではない。それぞれのものを表すベクトルを定義すれば、ビジネスにおけるあらゆるものの情報検索にも利用できる。

以下はその例である:

- 類似ユーザーの検索: ユーザーのアクティビティ、過去の購入履歴、その他のユーザー属性を組み合わせて、ビジネスにおける各ユーザーを表すベクトルを定義すれば、指定したユーザーに類似するすべてのユーザーを見つけることができる。例えば、類似した商品を購入しているユーザー、ボットである可能性が高いユーザー、デジタルマーケティングでターゲットにすべき潜在的なプレミアム顧客であるユーザーなどを確認することができる。

- 類似商品やアイテムの検索：説明文、価格、販売場所などの商品の特徴から生成されたベクトルを使って、類似商品を見つけ、多くの質問に答えることができる。例えば、"この商品に似ていて、同じユースケースに使えそうな商品は他にあるか？"、"この地域で過去24時間に売れた商品は何か？" (時間と近さに基づいて)
- 欠陥のあるIoTデバイスを見つける：欠陥デバイスの特徴を信号からとらえたベクトル検索により、潜在的に欠陥のあるデバイスを即座に見つけ、プロアクティブなメンテナンスを行うことができる。
- 広告の発見：明確に定義されたベクトルにより、視聴者にとって最も関連性の高い広告や適切な広告を、高いスループットでミリ秒単位で見つけることができる。

- セキュリティ脅威の発見 コンピュータウィルスのバイナリのシグネチャや、ウェブサービスやネットワーク機器に対する悪意のある攻撃行動をベクトル化することで、セキュリティの脅威を特定することができる。
- …その他多数： 今後数年のうちに、あらゆる業界においてベクトル検索のさまざまなアプリケーションが登場し、ベクトル検索はリレーショナル・データベースと同じくらい重要な技術になると思われる。」

高速でスケーラブルな
Vector Searchサービスを構築する

「100万アイテムのプールから類似アイテム（この場合は魚の画像）を見つけるのに約20秒かかる。このレベルのパフォーマンスは、特にMatchIt Fastのデモと比較すると、それほど印象的なものではない。

BigQueryは業界最速のデータウェアハウスサービスの1つなのに、なぜベクトル検索にこれほど時間がかかるのだろうか？」

「これは2つ目の課題を示している。

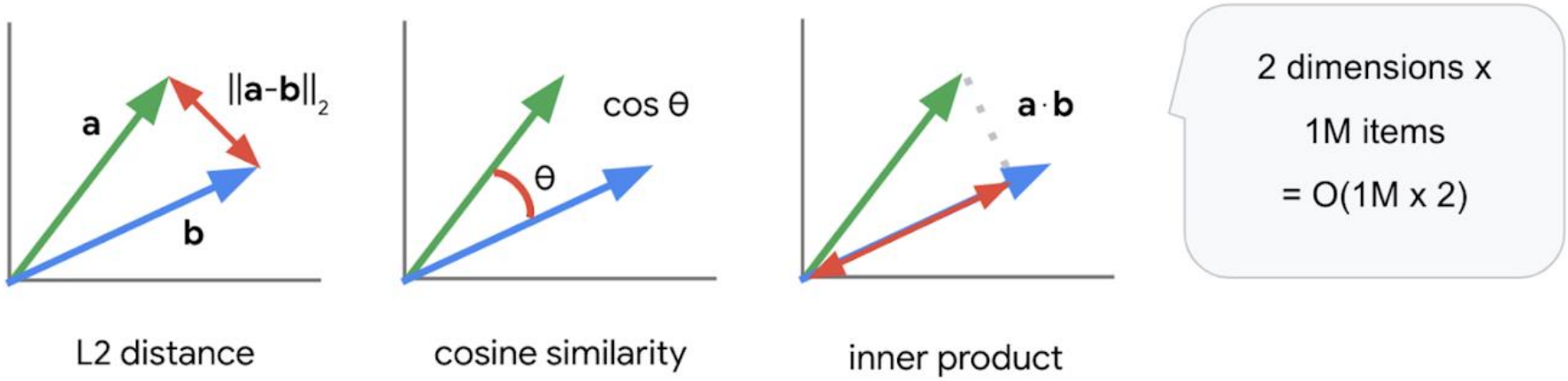
高速でスケーラブルなベクトル検索エンジンを構築するのは簡単なことではない。

素朴な方法で実装すれば、いずれもベクトルの数に次元数を掛けた数に比例した計算が必要になる。

例えば、1024個の要素を持つベクトルと1M個のベクトルを比較した場合、計算数は1024×1M＝1.02Bに比例する。

これは、1回の検索ですべてのエンティティに目を通すために必要な計算であり、上記のBigQueryのデモに時間がかかる理由である。」

ベクトル間の類似度を計算するための最も広く使われているメトリクスは、L2距離（ユークリッド距離）、コサイン・シミラリティ、内積（ドット積）である。



Calculating vector similarity

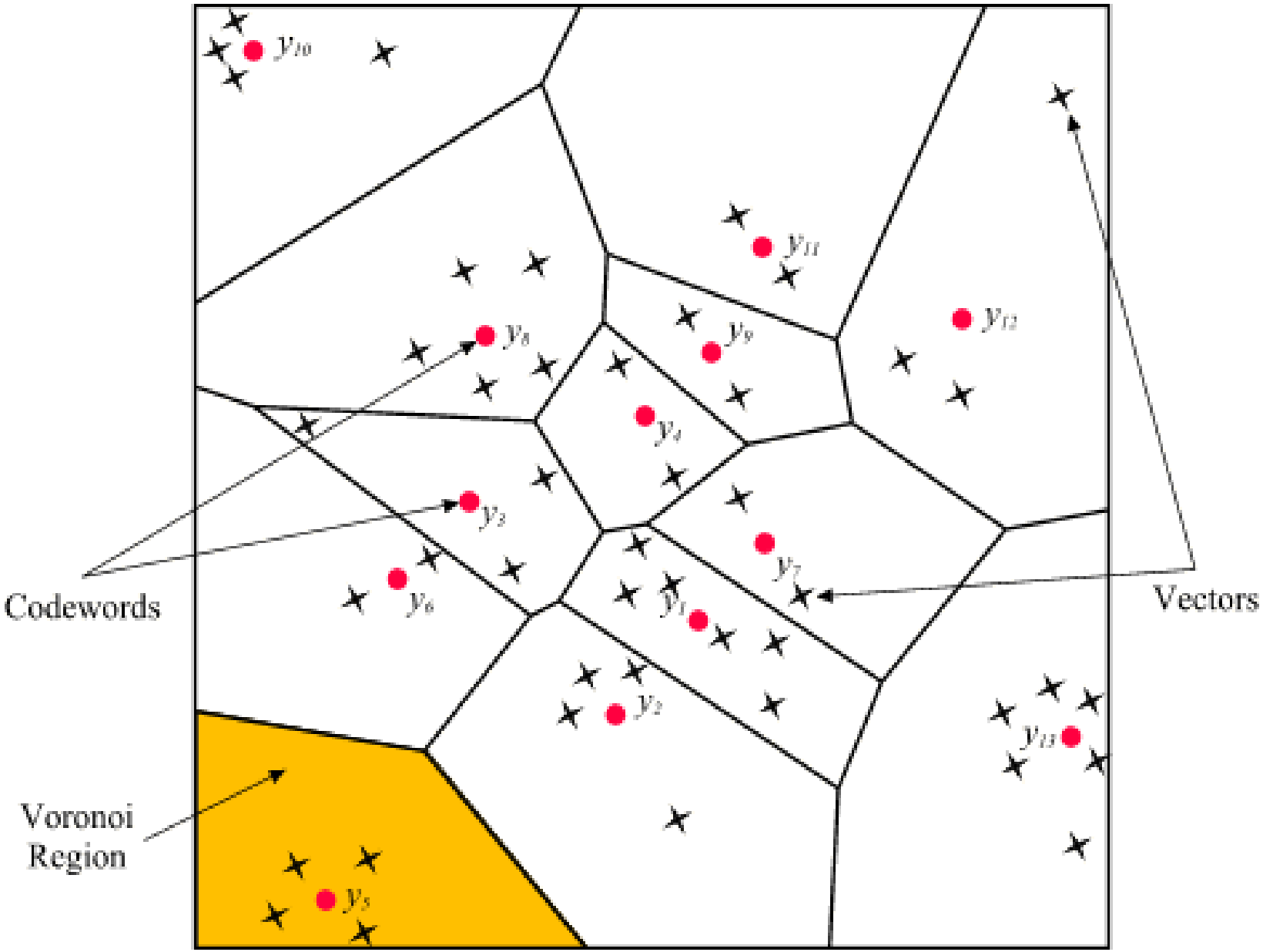
Approximate Nearest Neighbor (ANN)と Vector Quantization (VQ)

ベクトルを1つずつ比較する代わりに、近似最近傍（ANN）アプローチを使用して検索時間を短縮することができる。

多くのANNアルゴリズムはベクトル量子化（VQ）を使用しており、ベクトル空間を複数のグループに分割し、各グループを表す「コードワード」を定義し、それらのコードワードのみを検索する。

このVQテクニックはクエリー速度を劇的に向上させ、リレーショナル・データベースや全文検索エンジンにインデックス付けが不可欠であるように、多くのANNアルゴリズムに不可欠な部分である。

Vector Quantizationの例



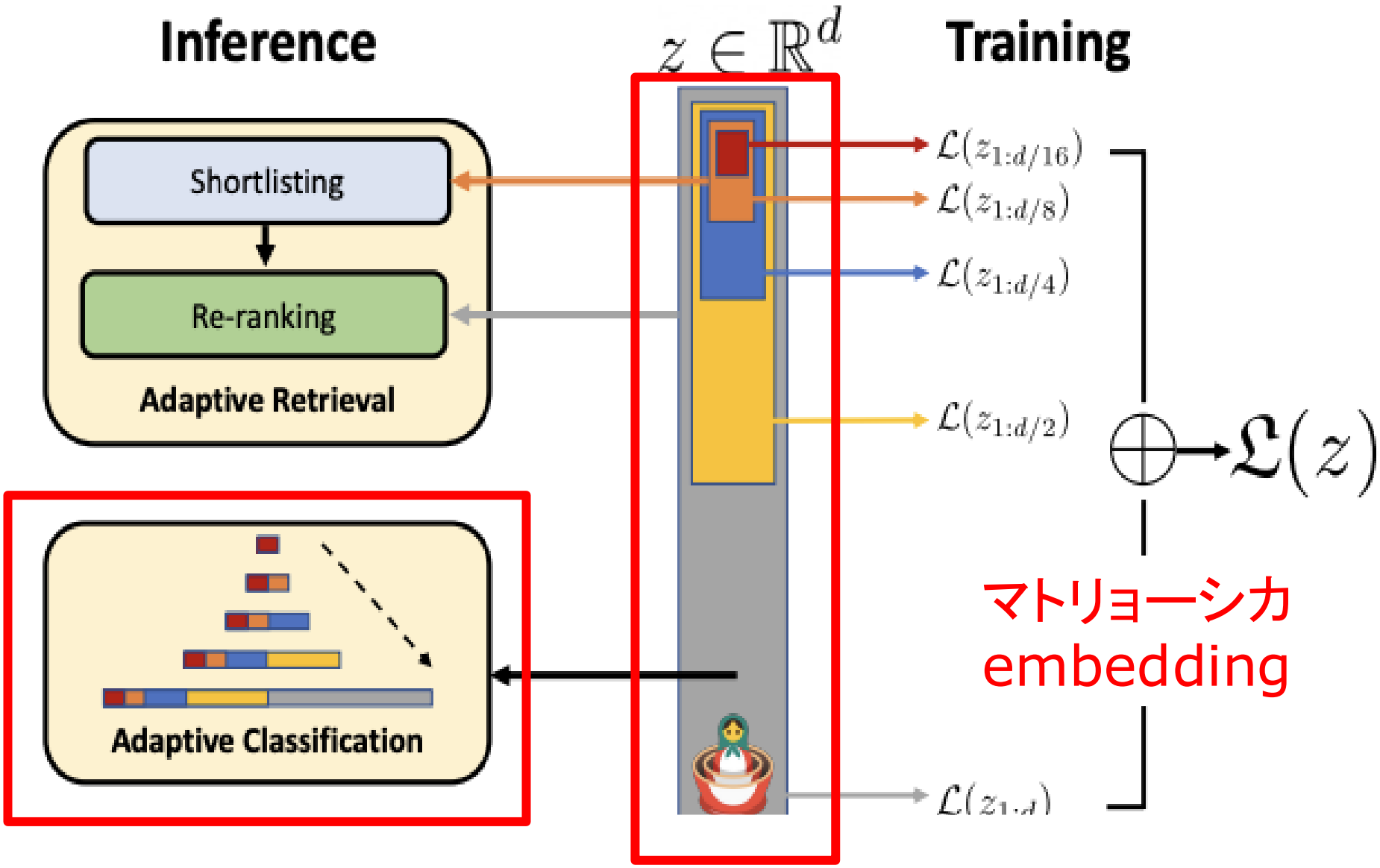
参考文献

Vector Quantization については、次のページを参照ください。
https://www.mqasem.net/vectorquantization/vq.html

1995年のものですが、次の論文も参考になります。
“Solving Multiclass Learning Problems via Error-Correcting Output Codes”
https://arxiv.org/abs/cs/9501101

マトリョーシカ -- embedding技術の革新

embedding技術の革新



Matryoshka Representation Learning

embedding技術の発展のなかで最も重要なものは、Aditya Kusupati らが2022年に発表した「マトリョーシカ表現学習」と呼ばれるものです。

"Matryoshka Representation Learning"
https://arxiv.org/abs/2205.13147

マトリョーシカ embedding
embeddingの次元を増減させる

それは、いったん大きな「次元」で学習したembeddingを、目的に応じてその「次元」を柔軟に変更することができる技術です。

例えば、2048次元の高い精度のembeddingを作成したとしても、そのembeddingをスマホの上で利用するのはメモリーやハードウェアの制約で難しくなります。

ただ、同じ対象についての推論が、多少精度が落ちても、32次元のembeddingで可能となれば、それは非力なスマートフォンでも実行可能になります。

MRLはAdaptive

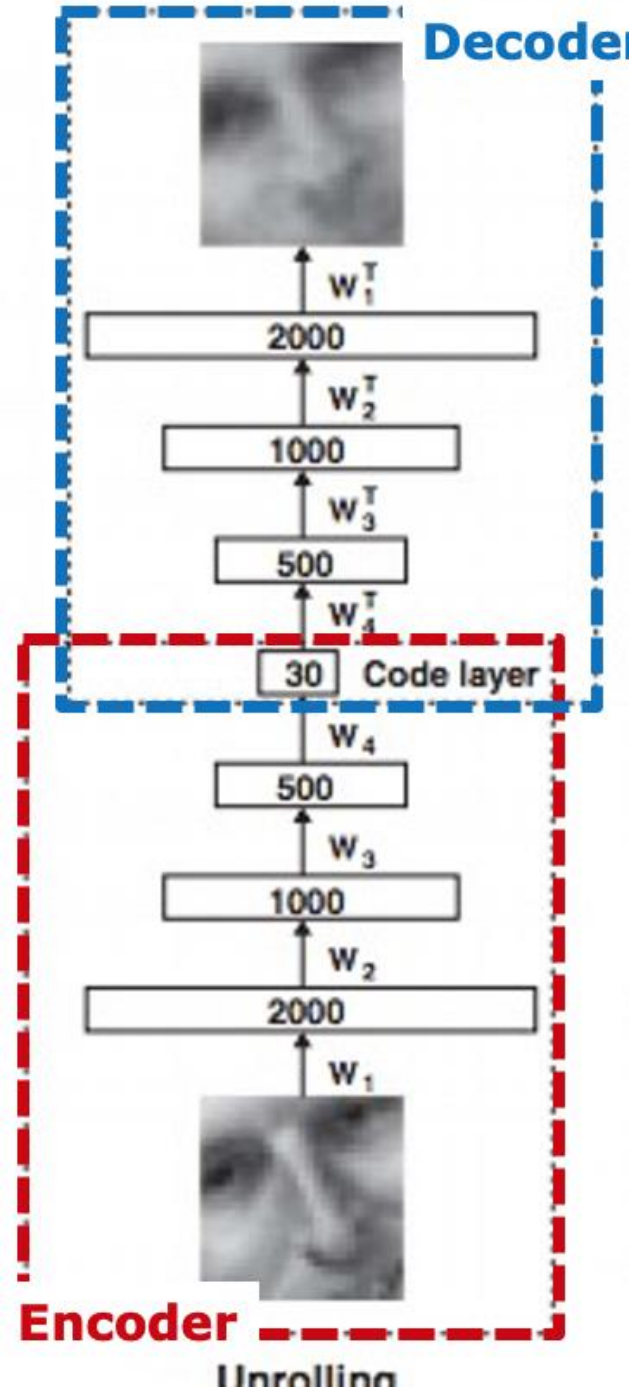
問題は、embeddingの次元が変わるたびに、embeddingを計算し直すというのは、現実的ではないことです。

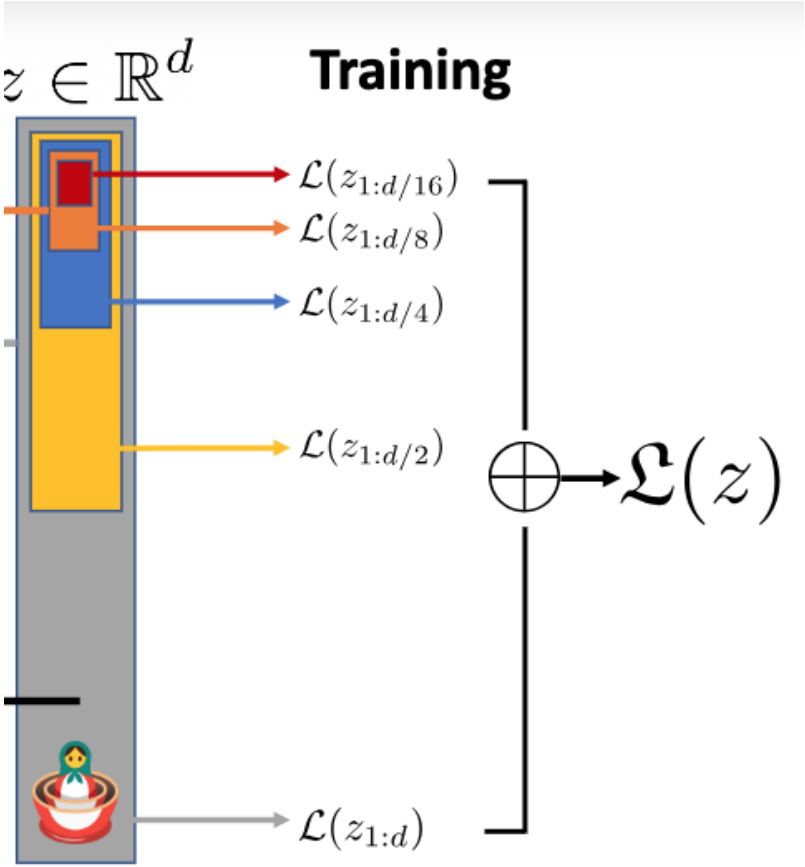
Matryoshka Representation Learningという名前が示すように、MRLの基本的な構成要素は、複数の次元に対応したembeddingを一回の学習で生成しようという技術です。

その実践的な真価は、いったんMRLで生成されたembeddingを、状況に応じて次元を変えて利用できるところにあります。MRLが、Adaptiveだといわれるゆえんはそこにあります。

次元を下げるだけでなく、逆に、必要に応じて、embeddingの次元を上げて精度の高い推論を行うことも、MRLでは可能です。

HintonのAutoencoder と Matryoshka Representation Learning





なぜ、「マトリョーシカ」？

実装としては、大きな次元でのembeddingの内側に、小さな次元のembeddingが「入れ子」で入っているような形になっています。「マトリョーシカ」という名前は、その「入れ子」構造にちなんだものです。

もっとも、実際のマトリョーシカの場合には、中に入れ子で入っている小さな人形の数はあらかじめ決まっているのですが、MRLの場合には、そうではありません。

上位のembedding情報だけをピックアップして、下位の情報を切り捨てることことで、任意の次元のembeddingを得ることができます。（もちろん、学習時に指定した次元以上のembeddingが得られるわけではありません。）

2048次元　1024次元　512次元　256次元　128次元

Commons:Featured picture candidates/Image:Russian-Matroshka.jpg

128次元　64次元　32次元　16次元　8次元

Commons:Featured picture candidates/Image:Russian-Matroshka.jpg

embeddingモデルとMRL

その技術は、オープンソースとして次のサイトで公開されています。

https://github.com/RAIVNLab/MRL

MRLは、OpenAIのembeddingモデル text-embedding-3 をはじめとして、現在のembeddingモデルの基礎になっています。

MRL: Matryoshka Representation Learning 論文 Abstract

学習された表現は、現代のML（Machine Learning）システムの中心的なコンポーネントである。それは、多くの下流タスクに対応している。

表現を学習する場合、それぞれの下流タスクの計算量や統計的制約が未知であることが多い。

このような状況では、厳密に固定した次元での表現は、手元のタスクに適合するには、ある場合には過剰で、ある場合には不足している可能性がある。そこで、次のような問いが生まれる。

「計算資源が変化する複数の下流タスクに適応できる柔軟な表現を設計できないか？」

https://arxiv.org/abs/2205.13147

coarse-to-fine representation

この論文の主要な貢献は「マトリョーシカ表現学習」(MRL: Matryoshka Representation Learning)である。

それは、異なる粒度で情報をエンコードし、単一の埋め込みで下流タスクの計算上の制約に適応することができる。

MRLは既存の表現学習パイプラインに最小限の変更を加えるだけで、推論やデプロイ時に追加コストは発生しない。

MRL は独立に学習された低次元表現と同等以上の精度と豊かさを持つ、粗いものから細かいものまでの(coarse-to-fine)表現を学習する。

MRLの柔軟性

学習されたマトリョーシカ表現の柔軟性により、以下のことが実現される。

(a) ImageNet-1Kの分類において、同レベルの精度で最大14倍の埋め込みサイズの削減、

(b) ImageNet-1Kと4Kの大規模検索において、最大14倍の高速化、

(c) ロングテールのfew shot分類において、最大2%の精度向上。

これらは、すべて元の表現と同等の頑健さをもって実行される。

様々なモダリティへの拡張

最後に、MRLは視覚（ViT、ResNet）、視覚＋言語（ALIGN）、言語（BERT）といった様々なモダリティのウェブスケール・データセット（ImageNet、JFT）にもシームレスに拡張できることを示す。

MRLのコードと事前学習済みモデルは https://github.com/RAIVNLab/MRL でオープンソース化されている。

MRLの応用

マトリョーシカ表現は精度を大きく損なうことなく、大規模な分類や検索の効率を向上させる。

マトリョーシカ表現には様々な応用が考えられるが、本研究では、実世界のMLシステムの主要な構成要素である大規模分類と検索に焦点を当てる。

分類への応用

分類では、MRLで学習したモデルから得られる可変サイズの表現を用いて適応的カスケードを行い、特定の精度を達成するために必要な埋め込みの平均次元を大幅に削減する。

例えばImageNet-1Kの場合、MRL + アダプティブ分類により、ベースラインと同じ精度で最大14倍の表現サイズを削減することができる（セクション4.2.1）。

検索への応用

同様にMRLを適応的検索システムにも利用する。

クエリが与えられた場合、クエリ埋込みの最初の数次元を使って検索候補を絞り込み、その後、より多くの次元を使って検索された集合を再ランク付けする。

このアプローチをシンプルに実装することで、標準的な埋め込みベクトルを用いたシングルショット検索システムと比較して、理論的には128倍（FLOPS換算）、実時間では14倍の高速化を実現する。

最後に、MRLは明示的に粗から細への表現ベクトルを学習するため、直感的には様々な次元間でより多くの意味情報を共有するはずである（図5）。

これはロングテールの継続学習設定において、元の埋め込みと同程度の頑健性を持ちながら、最大2%の精度向上に反映される。

さらに、MRLは粗視化から微視化までの性質を持つため、インスタンス間の分類の難しさや情報のボトルネックを分析する手法としても利用できる。

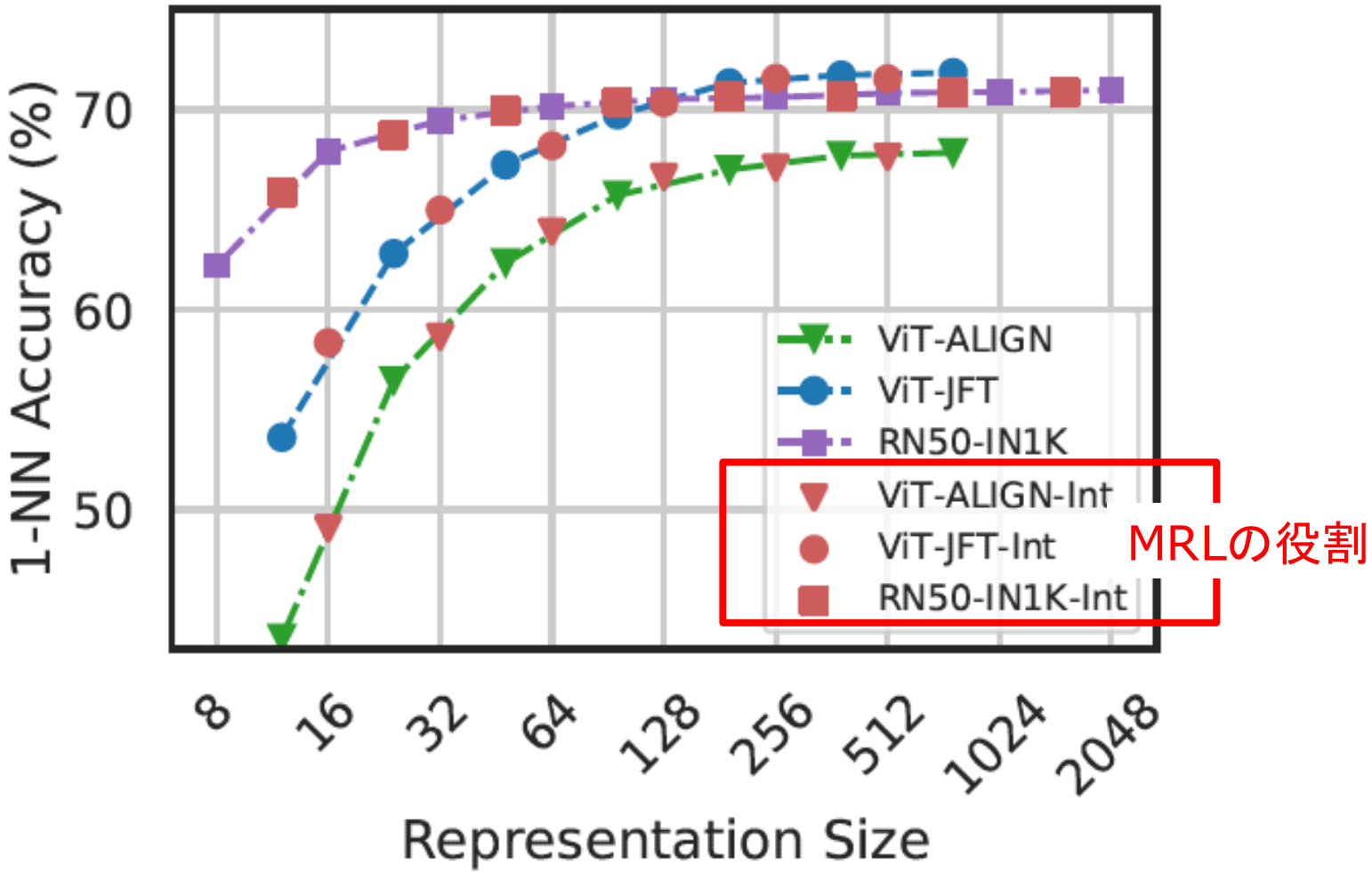


Figure 5: Despite optimizing MRL only for O(log(d)) dimensions for ResNet50 and ViT B/16 models; the accuracy in the intermediate dimensions shows interpolating behaviour.

新しい埋め込みモデルと API の更新
OpenAI

2つの新しい埋め込みモデルを発表します。小さく高効率な text-embedding-3-small モデルと、大きく強力な text-embedding-3-large モデルです。

埋め込みとは、自然言語やコードなどのコンテンツ内で概念を表す数列のことです。埋め込みは、機械学習モデルなどのアルゴリズムがコンテンツ間の関係を理解し、クラスタリングや検索などのタスクを実行することを容易にします。ChatGPT と Assistants API、多くの検索拡張生成（RAG）開発者ツールの知識検索といったアプリケーションを駆動しています。

https://openai.com/ja-JP/index/new-embedding-models-and-api-updates/

埋め込み短縮のネイティブサポート

ベクターストアに保存して検索する場合など、より大きな埋め込みを使用する場合は、一般的に、小型の埋め込みを使用するよりもコストがかかり、より多くの計算、メモリ、ストレージを消費します。

当社の新しい埋め込みモデルは、いずれも開発者が埋め込みを使用する際の性能とコストを釣り合わせられるようにする手法Aで学習を行っています。具体的には、dimensions API パラメーターを渡すことで、埋め込みがその概念を表す特性を失うことなく埋め込みを短縮できます（シーケンスの末尾からいくつかの数字を削除）。

例えば、MTEB ベンチマークでは、text-embedding-3-large 埋め込みを256サイズに短縮しても、1536サイズの短縮されていない text-embedding-ada-002 埋め込みの性能を上回ることができます。

	ada v2	text-embedding-3-small		text-embedding-3-large		
Embedding size	1536	512	1536	256	1024	3072
Average MTEB score	61.0	61.6	62.3	62.0	64.1	64.6

これで非常に柔軟な使い方が可能になります。

例えば、1024ディメンションまでの埋め込みしかサポートしていないベクターデータストアを使用する場合、開発者は最適な埋め込みモデル text-embedding-3-large を使用し、dimensions API パラメーターに1024を指定することで、埋め込みを3072ディメンションから短縮し、精度と引き換えにベクターサイズを小型化できます。


Part 3 の終わり

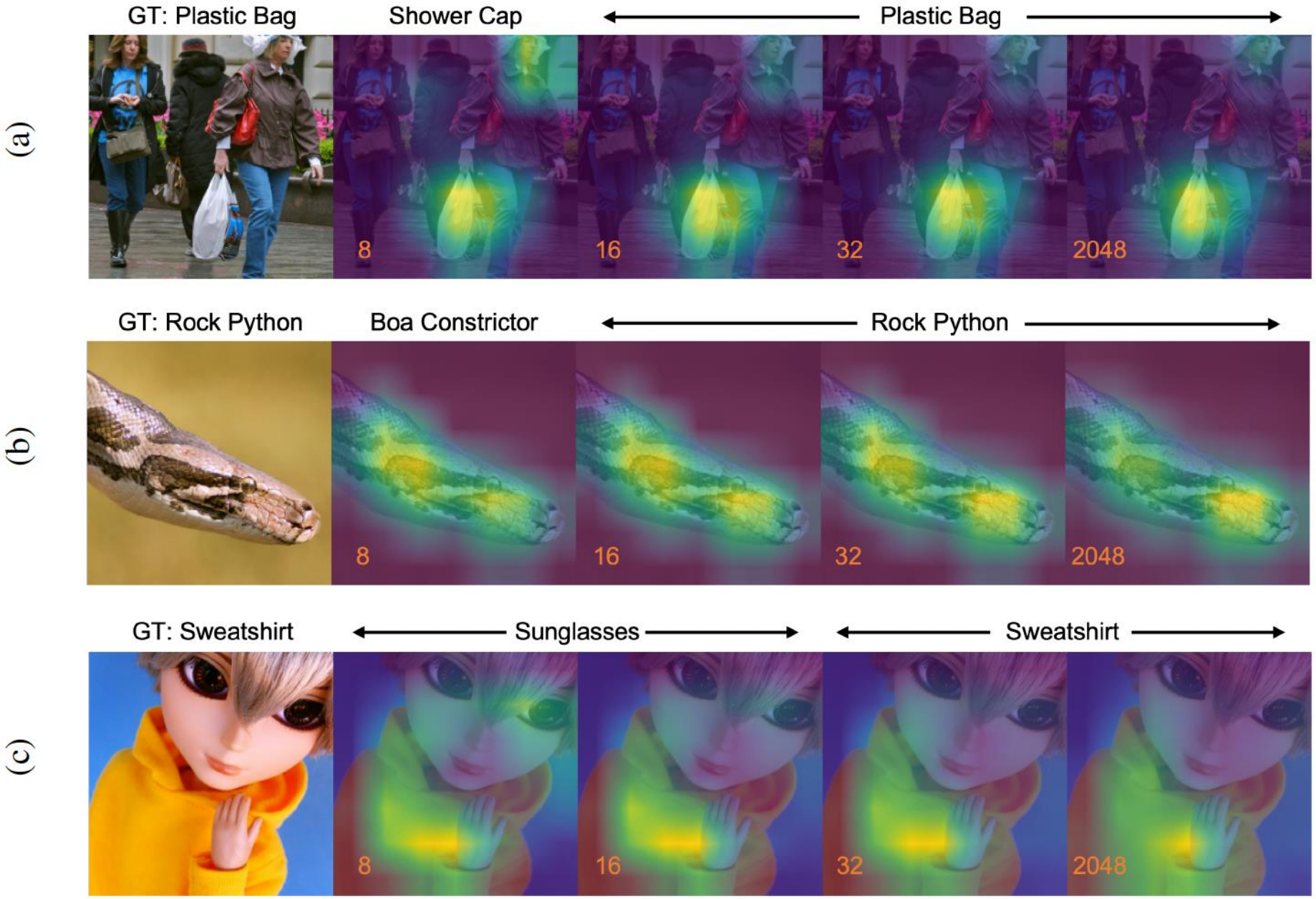

Appendix


Adaptive Classification

表現の次元と予測の精度

表現の次元が変わるにつれて、予測の精度がどう変わるかをみてみましょう。

表現の次元が、低くても(d=32)でも、かなりの精度を持ち得ることが示されています。


(a)
GT: Plastic Bag
Shower Cap
Plastic Bag
8
16
32
2048
(b)
GT: Rock Python
Boa Constrictor
Rock Python
8
16
32
2048
(c)
GT: Sweatshirt
Sunglasses
Sweatshirt
8
16
32
2048

(a)

GT: Plastic Bag

正解は”Plastic Bag”。

8次元の表現では、シーン内に（より大きな視野では右上に）他の関連物体が存在するために混乱して、“Shower Cap”を予測している。

16次元以上（32次元、2048次元）では、正しく”Plastic Bag” と答えている。

(b)

GT: Rock Python

正解は、”Rock Python”

8次元モデルは 同じスーパークラスに属する”Boa”と混同している。

16次元以上（32次元、2048次元）では、正しく”Rock Python” と答えている。

正解は、”SweatShirt”

8次元と16次元のモデルでは、人形の目（”Sunglasses”）にピントが合っていて、”SweatShirt”にピントが合っていない。高次元では正しくピントが合っている

Adaptive Classification

ImageNet-1K には、1,281,167枚のラベル付き訓練画像と、1,000クラスにわたる50,000枚のラベル付き検証画像が含まれている。

ImageNet-1K検証セットのすべての画像に対して、分類にうまく機能する最小の表現を使用することを試みる。

ImageNet-1K検証セットの10Kサイズのサブセットを使用して、表現サイズをm_iからm_{i+1}に増加させるポリシーを学習させる。

このポリシーは、表現サイズm_iを用いた予測信頼度p_iが学習した閾値t_i^*を超えるかどうかに基づいている。

$p_i \geq t_i^*$ ならば、表現サイズ m_i からの予測を使用し、そうでなければ、表現サイズを m_{i+1} に増加する。

閾値の設定

最適な閾値 t_i^*を学習するために、0から1の間でグリッド探索を行った（100 サンプル）。

各閾値t_kについて、10K画像サブセットに対する分類精度を計算し、t_i^*を最も精度の良い最小の閾値t_kに設定した。

この手順で、連続するモデルの閾値、すなわち

$\{ t_j^* \mid j \in \{8,\ 16,\ 32,\ 64, \ldots, 2048\}$を取得することができる。

推論には、ImageNet-1K検証セットの残りの4万サンプルを使用した。

最小サイズの表現（$m = 8$）から開始し、計算された予測信頼度p_8と学習された最適閾値t_8^*を比較した。$p_8 \leq t_8^*$ならば、$m = 16$ に増やし、$m = d = 2048$ までこの手順を繰り返した。

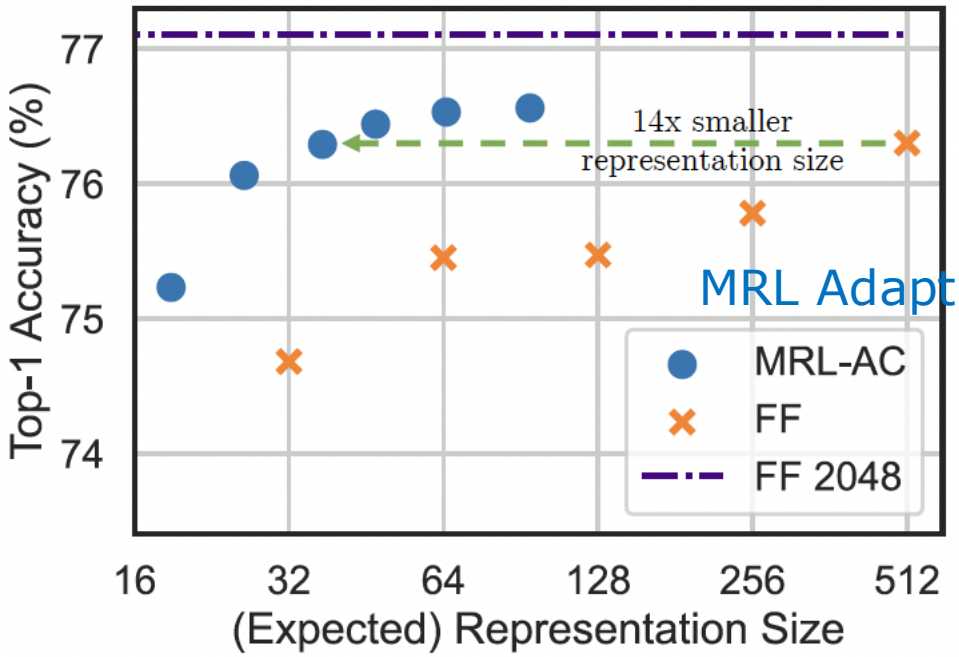
表3と図6に示すように、ImageNet-1Kで76.3%の分類精度を達成するためには、期待値として約37の大きさの表現が必要であり、これはFF-512のベースラインよりも約14倍小さい。

512/37 = 13.833…

表3

Expected Rep. Size	Accuracy
13.43 ± 0.81	73.79 ± 0.10
18.32 ± 1.36	75.25 ± 0.11
25.87 ± 2.41	76.05 ± 0.15
36.26 ± 4.78	76.28 ± 0.16
48.00 ± 8.24	76.43 ± 0.18
64.39 ± 12.55	76.53 ± 0.19
90.22 ± 20.88	76.55 ± 0.20
118.85 ± 33.37	76.56 ± 0.20

図6



MRL Adaptive Classification

MRL論文で利用されているDatasetと モデルの訓練方法

MRL論文には、多数の図表が掲載されています。(図1~図12, 表1~表31)

これらの図表を理解するために、MRL論文で利用されている基本的なdataset とモデルの訓練方法を、まとめてみました。

Datasets

ImageNet-1K *[76] contains 1,281,167 labeled train images, and 50,000 labelled validation images across 1,000 classes. The images were transformed with standard procedures detailed by FFCV [56].*

ImageNet-4K *dataset was constructed by selecting 4,202 classes, non-overlapping with ImageNet-1K, from ImageNet-21K [16] with 1,050 or more examples. The train set contains 1,000 examples and the query/validation set contains 50 examples per class totalling to ~4.2M and ~200K respectively. We will release the list of images curated together to construct ImageNet-4K.*

JFT-300M *[85] is a large-scale multi-label dataset with 300M images labelled across 18,291 categories.*

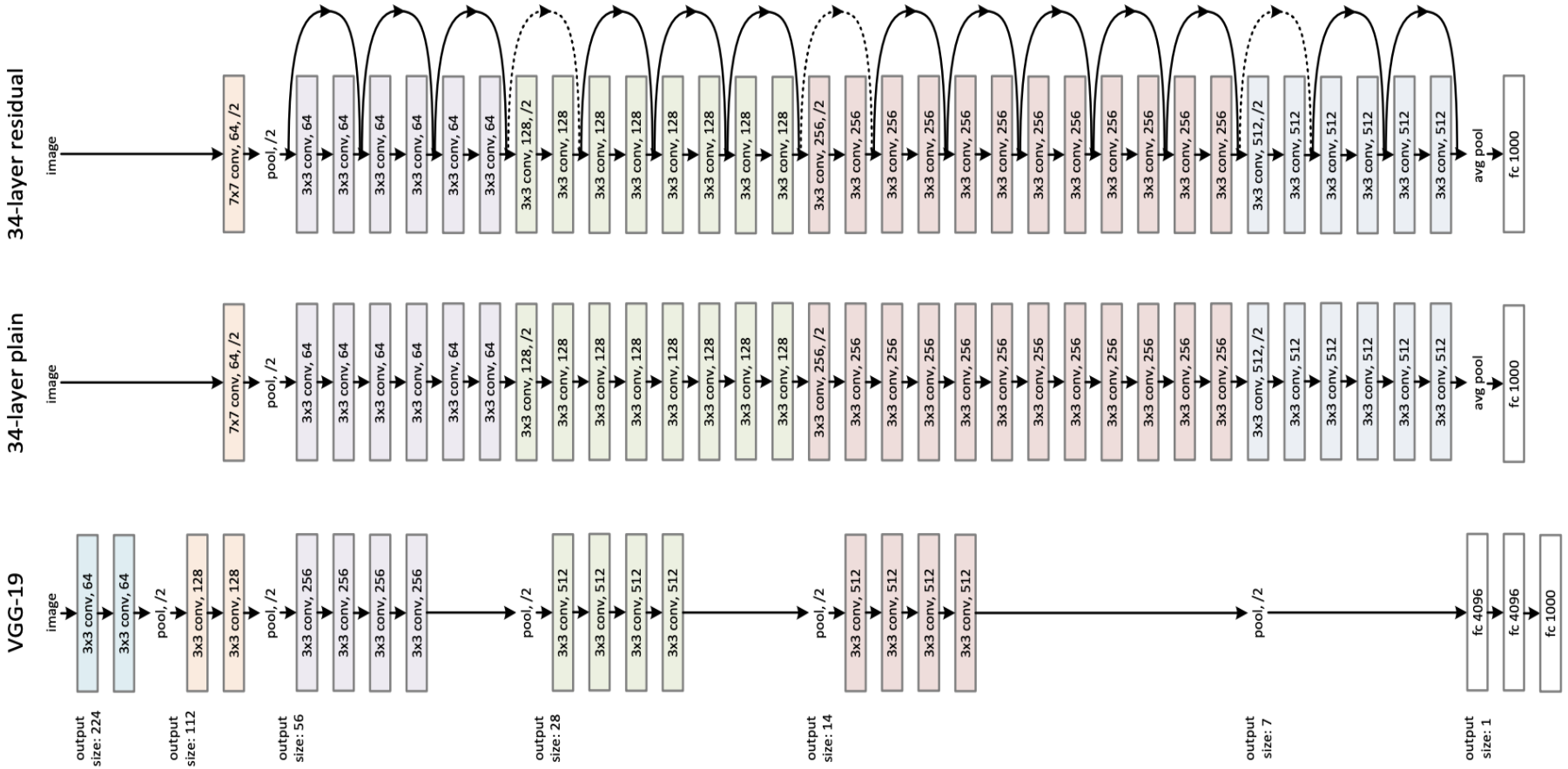
ALIGN *[46] utilizes a large scale noisy image-text dataset containing 1.8B image-text pairs.*

Model Training with ResNet50

We trained all ResNet50–MRL models using the efficient dataloaders of FFCV [56]. We utilized the rn50_40_epochs.yaml configuration file of FFCV to train all MRL models defined below:

- MRL: ResNet50 model with the fc layer replaced by MRL_Linear_Layer(efficient=False)
- MRL–E: ResNet50 model with the fc layer replaced by MRL_Linear_Layer(efficient=True)
- FF–k: ResNet50 model with the fc layer replaced by torch.nn.Linear(k, num_classes), where k $\in$ [8, 16, 32, 64, 128, 256, 512, 1024, 2048]. We will henceforth refer to these models as simply FF, with the k value denoting representation size.

ResNet50



"Deep Residual Learning for Image Recognition"
https://arxiv.org/abs/1512.03385


Adaptive Retrieval

Adaptive Retrieval 概要

MRLのAdaptive Retrievalは、前回見たような Vector Search(nearest neighbour search)の一層の高速化を目指したものです。基本的なアイデアは、次のようなものです。

- まず、与えられたクエリ画像に対して、16次元のような低い次元表現($D_s = 16$)を用いて 、データベースから、例えば、200個の画像の候補を取得します。(それをショートリスト K = 200と呼んでいます) これは高速に可能です。

- 次に、その200個の画像に対して、2048次元のような高容量表現($D_r = 2048$)を用いて、クエリ画像との近さの再ランキングを行います。高次元の2048次元の200画像を素朴に再ランク付けするだけだったら、400KFLOPしかかからないといいます。こうして、最終的な絞り込みを行います。

実世界のシナリオでは、トップランキングの性能が重要な目的です。kが限られた重要な領域をカバーするmAP@kで測定すると、Adaptive Retrievalは、最初から固定次元（例えば、2048次元）の表現によるシングルショット検索よりも、計算量とメモリが大幅に向上することが示されます。

メトリクスのために、次のmAP@kとP@k利用しています。

- 平均平均精度 mean average precision (mAP@k)
- 精度(P@k): $P@k = \frac{correct_{pred}}{k}$
  ここで、$correct_{pred}$は、長さkのショートリストを用いて、クエリ集合全体で正しいラベルを持つNN 最近傍 nearest neighborsの平均検索数である。

MRL Model

- MRL:
  ResNet50のfc層を
  MRL_Linear_Layer (efficient =False)に置き換えたモデル。
- MRL-E:
  fc層をMRL_Linear_Layer(efficient=True)に置き換えた
  ResNet50モデル。
- FF-k:
  fc層をtorch.nn.Linear(k, num_classes)で置き換えた
  ResNet50モデル、k∈[8, 16, 32, 64, 128, 256, 512,
  1024, 2048]。以降、これらのモデルを単にFFと呼び、kの値
  は表現サイズを表す。

4.3 Retrieval

学習された表現を用いた最近傍探索は、多くの検索や探索アプリケーションに力を与えている。

このセクションでは、事前学習されたResNet50モデルの画像検索性能について、2つの大規模データセットImageNet-1K とImageNet-4Kを用いて説明する。

- ImageNet-1Kのデータベースサイズは約1.3Mで、クエリ集合は1000クラスを一様に網羅した50Kのサンプルで構成されている。
- ImageNet-4Kのデータベースサイズは約4.2Mで、クエリ集合は4202クラスを一様に網羅した200Kのサンプルで構成されている。

検索の計算コスト

ResNet50の1回のフォワードパスは4GFLOPsかかるのだが、ImageNet-1Kに対する正確な検索は、1クエリあた2.6GFLOPsかかる。検索オーバヘッドは総コストの40%であるが、検索コストはデータベースのサイズが大きくなるにつれて直線的に増加する。

ImageNet-4Kでは、正確な検索コストが計算のボトルネックとなる。（クエリーあたり8.6GFLOPs）。これらの大規模なデータベースでは、メモリとディスク使用量も、ボトルネックになることが多い。

しかし、ほとんどの実世界のアプリケーションでの正確な検索コスト$O(dN)$は、HNSWのような近似最近傍探索 approximate nearest neighbor search（ANNS）に置き換えられ、コストは$O(dlog(N))$となる。

検索性能

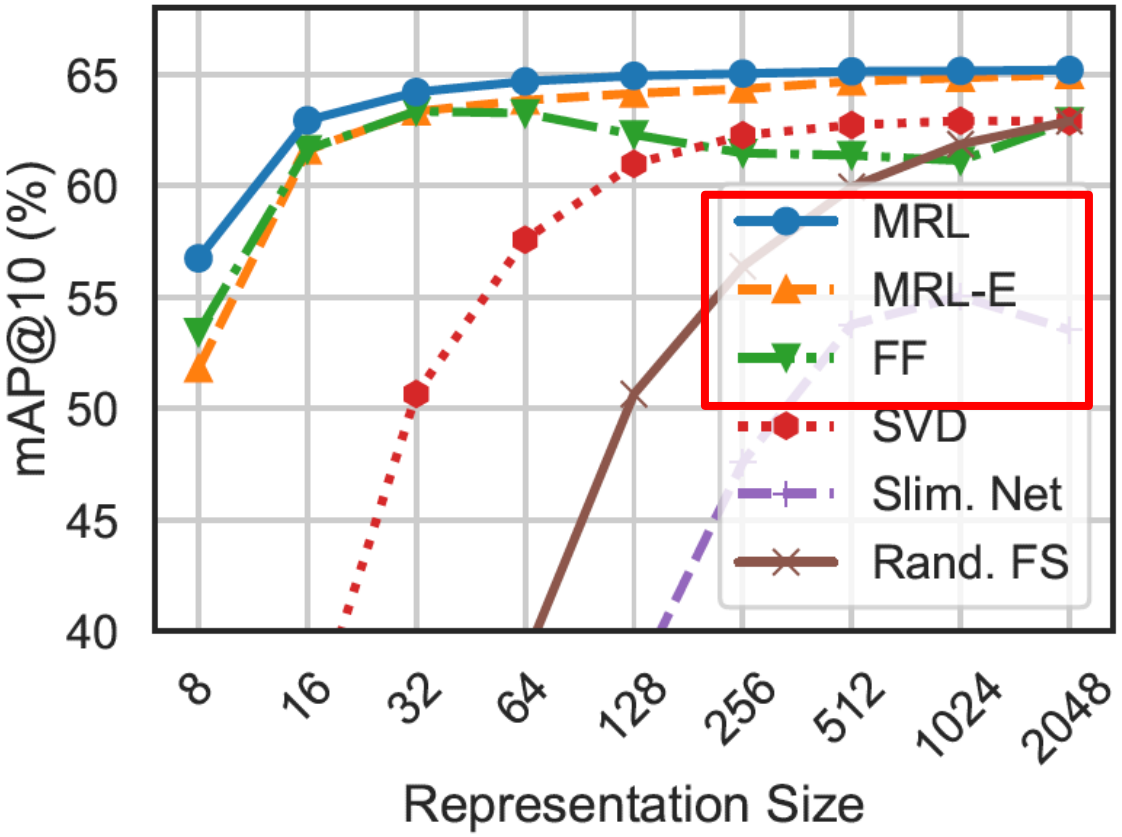
画像検索の目的は、事前に学習されたモデルから得られた表現を用いて、クエリと同じクラスに属する画像を検索することである。

このセクションでは、大規模な関連画像検索のセットアップを包括的に捉える平均平均精度@10（mAP@10）を用いて検索性能を比較する。MFLOPsで厳密検索を用いたクエリあたりのコストを測定する。全ての埋め込みは単位正規化され、L2距離メトリックを用いて検索される。

最後に、k = {10, 25, 50, 100}のmAP@kとP@kにまたがる広範なメトリクスのセットと、厳密探索とHNSWの実際のウォールクロック時間を報告する。詳細は付録EとFを参照。

図7はImageNet-1KにおけるResNet50表現のmAP@10性能を、MRL、MRL-E、FF、slimmableネットワークと、SVDとランダム特徴選択を用いたポストホック圧縮の次元数で比較したものである。

図7



マトリョーシカ表現が最も精度が高く、FFベースラインよりも最大3%優れている。分類と同様に、ポストホック圧縮とスリマブルネットワークベースラインは、≤256次元で検索mAP@10の大幅な低下に悩まされる。付録Eでは、同じモデルのImageNet-4KでのmAP@10について述べる。

MRLモデルは、ウェブスケールのデータベースに対して複数のモデルのフォワードパスを追加することなく、様々な粒度で正確な検索を実行することができる。

FFモデルは独立したデータベースを生成するため、その保存や切り替えに法外なコストがかかる。

マトリョーシカ表現は、Adaptive Retreaval（AR）を可能にし、すべてのデータと下流タスクにフルキャパシティ表現（d = 2048）を使用する必要性を軽減する。

最後に、ANNSパイプラインの一部として使用されている全てのベクトル圧縮技術[60, 45]はマトリョーシカ表現と相補的であり、効率対精度のトレードオフをさらに改善することができる。

4.3.1 Adaptive Retrieval

与えられたクエリ画像に対して、$D_s = 16$のような低次元表現を用いてデータベースから画像のショートリストK = 200を取得し、その後 $D_r = 2048$のような高容量表現を用いて再ランキングを行う。

実世界のシナリオでは、トップランキングの性能が重要な目的であり、kが限られた重要な領域をカバーするmAP@kで測定すると、Adaptive Retrievalは固定次元の表現によるシングルショット検索よりも計算量とメモリが大幅に向上する。

最後に、他の検索パイプラインと同様に、Adaptive Retrievalの最も高価な部分は、ショートリストのための最近傍探索(nearest neighbour search)である。

例えば、2048次元の200画像を素朴に再ランク付けする場合でも、400KFLOPしかかからない。

すべてのAdaptive Retrieval実験について、クエリあたりの正確な検索コストを報告するが、パイプラインのショートリスト化コンポーネントは、ANNS（HNSW）を使用して高速化することができる。

付録Iでは、正確な検索の計算コスト、HNSWインデックスのメモリオーバーヘッド、および両実装のウォールクロック時間について詳述する。ショートリストに32近傍のHNSWを使用しても、検索時の精度は低下しない。

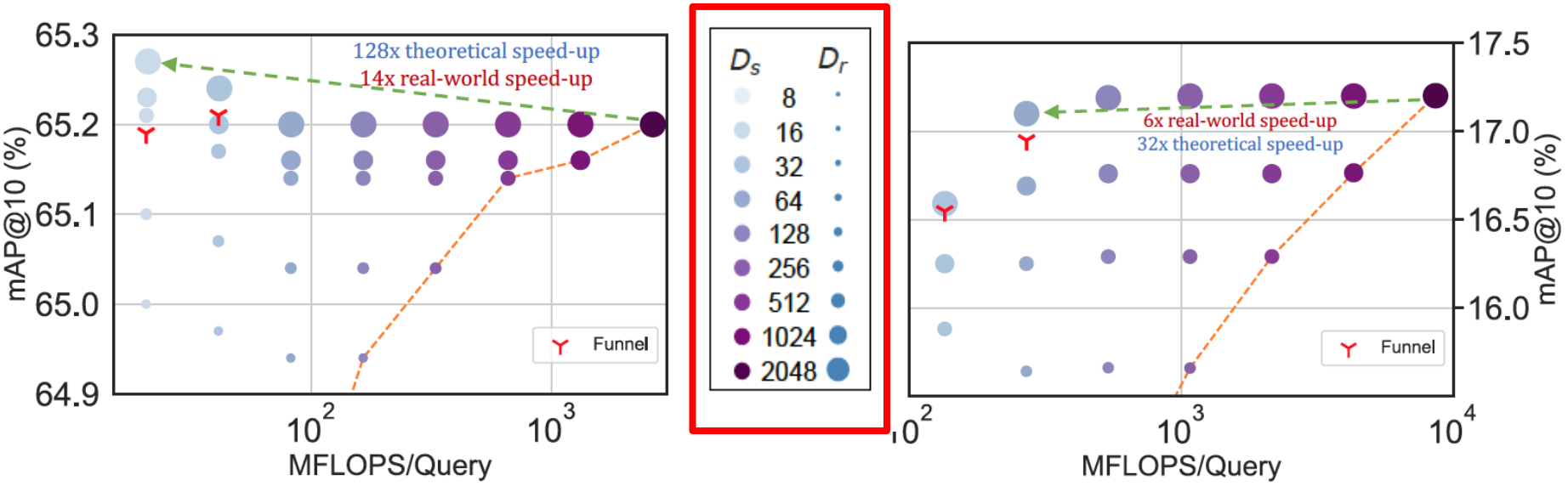
図8は、マトリョーシカ表現を用いた適応検索の計算量対精度のトレードオフを、ImageNet-1KのResNet50を用いた固定特徴によるシングルショットと比較して示している。

全てのAR設定は、様々な表現サイズにおいて、シングルショット検索のパレートフロンティアより上にあることが観察された。

特にImageNet-1Kでは、Ds = 16 & Dr = 2048のARモデルは、d = 2048のシングルショット検索と同等の精度を持ちながら、理論的には約128倍効率的で、実際には約14倍高速であることが示された（同じハードウェア上でHNSWを使用した場合との比較）。

ImageNet-4Kでも同様の傾向が見られるが、データセットの難易度が上がっているため、Ds = 64が必要である。この結果、理論値で約32倍、実測値で約6倍の高速化が得られた。

最後に、K = 200は我々の適応検索実験に有効であるが、付録K.2でショートリストのサイズkについてアブレーションを行ったところ、精度の向上がある点を境に止まることがわかったため、マトリョーシカ表現学習と適応検索のユースケースがさらに強化された。



(a) ImageNet-1K

(b) ImageNet-4K

図8

Funnel Retrieval

MRLでは、Dsを増加させながらk-NNのショートリストを逐次改善・改良するために、*Funnel Retrieval*と呼ぶ単純なカスケード・ポリシーが使われている。これは、DsとDrを手動で選択することへの依存を取り除く試みである。

Dsでショートリストを検索し、Drを増加させながら（再ランクカスケード）、同時にショートリストの長さを減少させながら（ショートリストカスケード）、ショートリストを5回再ランク付けすることを行う。

E　Image Retrieval
データベースの作成

各データセットに対してそれぞれN個とQ個のサンプルを含むデータベースとクエリーセットを生成した。

k-最近傍 k-nearest neighbors（k-NN）のショートリストを取得する表現サイズをDsで指定する。データベースは[N, Ds]配列、クエリ集合は[Q, Ds]配列、近傍集合は[Q, k]配列である。

FAISSとHNSWの利用

効率的な類似検索のためのライブラリであるFAISS [47]を用いて検索を行った。k-NNのショートリストを得るために、データベースを検索するためのインデックスを作成した。

faiss.IndexFlatL2でL2距離メトリックを用いた網羅的NN検索を行い、faiss.IndexHNSWFlatでHNSW[47]による近似NN検索(ANNS)を行った。

特に断りのない限りM＝32のHNSWを使用し、以後HNSW32と呼ぶ。

厳密探索インデックスは高速なk-NN探索計算のためにGPUに移したが、HNSWインデックスは現在GPUサポートがないためCPUに残した。

表 20 に、インデックス構築のためのウォールクロック時間とインデックスサイズを示す。HNSWは、高速なNN検索と大きなインデックスフットプリントをトレードオフにしている（付録Kで詳述）。データベースとクエリ・ベクトルは、インデックスを構築し検索を実行する前にfaiss.normalize_L2で正規化されている。

Table 20: FAISS [47] index size and build times for exact k-NN search with L2 Distance metric and approximate k-NN search with HNSW32 [62].

Rep. Size	Exact Search				HNSW32			
	ImageNet-1K		ImageNet-4K		ImageNet-1K		ImageNet-4K	
	Index Size (MB)	Index Build Time (s)	Index Size (MB)	Index Build Time (s)	Index Size (MB)	Index Build Time (s)	Index Size (MB)	Index Build Time (s)
8	40	0.04	131	0.33	381	4.87	1248	24.04
16	80	0.08	263	0.27	421	6.15	1379	33.31
32	160	0.16	525	0.52	501	6.80	1642	37.41
64	320	0.38	1051	1.05	661	8.31	2167	47.23
128	641	0.64	2101	2.10	981	11.73	3218	89.87
256	1281	1.27	4202	4.20	1622	17.70	5319	102.84
512	2562	2.52	8404	8.39	2903	27.95	9521	158.47
1024	5125	5.10	16808	17.20	5465	44.02	17925	236.30
2048	10249	10.36	33616	41.05	10590	86.15	34733	468.18

MRLのVector Searchのエンジン FAISSとHNSM

Adaptive Retrievalの最も計算コストがかかる部分は、ショートリストのための最近傍探索(nearest neighbour search)です。

MRLでは、このVector Searchのエンジンとして、FAISSとHNSMを使っています。

FAISSとHNSMの概要を紹介します。

Facebookが提供するオープンソース FAISS

Faiss は、密なベクトルの効率的な類似性検索とクラスタリングのためのライブラリである。RAMに収まらないようなものまで、あらゆるサイズのベクトル集合を探索するアルゴリズムが含まれている。また、評価とパラメータチューニングのためのサポートコードも含まれている。

FaissはC++で書かれており、Python/numpy用の完全なラッパーが用意されている。最も有用なアルゴリズムのいくつかはGPU上で実装されている。主にMetaのFundamental AI Researchグループで開発されている。

https://github.com/facebookresearch/faiss

FAISS についての論文 2017年

"Billion-scale similarity search with GPUs"
https://arxiv.org/abs/1702.08734

類似検索は、画像や動画のような複雑なデータを扱う特殊なデータベースシステムで応用される。これらのデータは一般的に高次元の特徴で表現され、特定のインデックス構造を必要とする。

本稿では、このタスクにGPUをより効果的に活用する問題に取り組む。GPUはデータ並列タスクを得意とするが、先行するアプローチは、k-min選択のような並列性が低いアルゴリズムや、メモリ階層をうまく利用できないアルゴリズムがボトルネックとなっている。

我々は，理論上のピーク性能の最大55%で動作するk-選択アルゴリズムを提案し，従来のGPU技術よりも8.5倍高速な最近傍実装を可能にする。また、ブルートフォース検索、近似検索、積量子化に基づく圧縮領域検索の最適化設計を提案することで、さまざまな類似検索シナリオに適用する。これらのすべてのセットアップにおいて、我々は大きなマージンをもって最先端技術を凌駕した。

我々の実装は、Yfcc100Mデータセットの9,500万画像に対する高精度k-NNグラフの構築を35分で、10億ベクトル連結グラフの構築を4台のMaxwell Titan X GPUで12時間未満で可能にしている。我々は、比較と再現性のために、我々のアプローチをオープンソース化した。

HNSWについての論文
Hierarchical Navigable Small World

"Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs"

https://arxiv.org/abs/1603.09320

Federal state budgetary institution of science Institute of Applied Physics of the Russian Academy of Sciences 2016-2018

我々は、制御可能な階層構造を持つナビゲート可能なスモールワールドグラフ（Hierarchical NSW, HNSW）に基づく近似K-最近傍探索のための新しいアプローチを提案する。

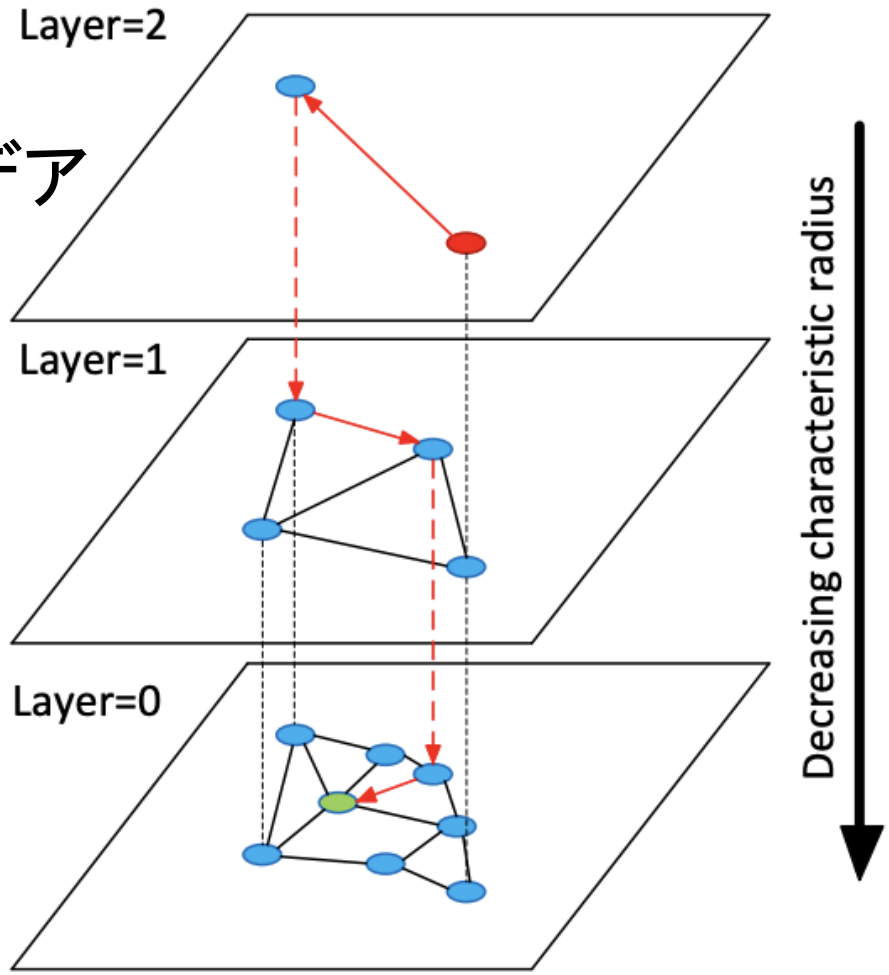
提案する解法は完全にグラフベースであり、一般的な近接グラフ手法の粗い探索段階で用いられるような、追加の探索構造を必要としない。

階層的NSWは、格納された要素のネストされたサブセットに対して、近接グラフの階層セット（レイヤー）からなる多レイヤー構造をインクリメンタルに構築する。

要素が存在する最大の層は、指数関数的に減衰する確率分布でランダムに選択される。これにより、以前に研究されたNSW（Navigable Small World）構造に似たグラフを生成することができる。

スケールの分離を利用して上位層から探索を開始することで、NSWと比較して性能が向上し、対数的な複雑さのスケーリングが可能となる。

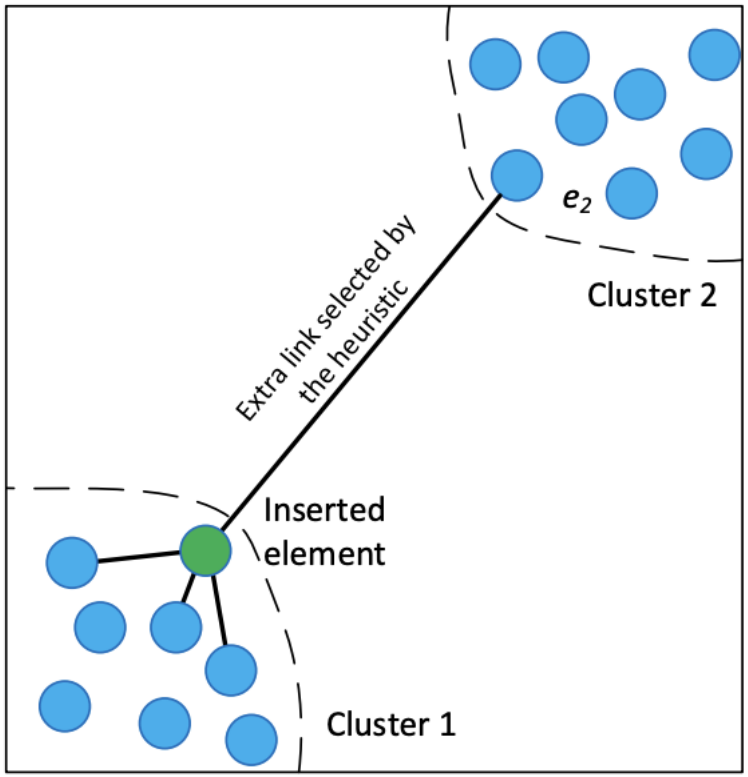
階層的NSWのアイデア



検索は最上層（赤）の要素から検索を開始する。赤い矢印は、エントリーポイントからクエリ（緑）までの貪欲なアルゴリズムの方向を示す。

近接グラフの近傍を選択するためのヒューリスティックを追加採用することで、高リコール時や高度にクラスタ化されたデータの場合に性能が大幅に向上する。

性能評価により、提案する一般的な計量空間探索インデックスが、これまでのオープンソースの最先端ベクトルオンリーアプローチを強く凌駕できることが実証された。このアルゴリズムはスキップリスト構造と類似しているため、バランスの取れた分散実装が可能である。



孤立した2つのクラスターについて、グラフの近傍を選択するために使用されるヒューリスティックの図解。

クラスタ1の境界に新しい要素が挿入される。この要素の最近傍はすべてクラスタ1に属しており、クラスタ間のドロネーグラフの辺が欠けている。しかし、ヒューリスティックはクラスタ2から要素e2を選択するため、挿入された要素がクラスタ1の他の要素と比較してe2に最も近い場合、大域的な接続性が維持される。


Appendixの終わり