

# Transformerの最前線

～置込みニューラルネットワークの先へ～

オムロン サイニックエックス



The 28th  
Symposium on Sensing  
via Image Information

**SSII  
2022**



# Transformerの最前線

～ 置込みニューラルネットワークの先へ～

2022.6.8

牛久 祥孝 (オムロンサイニックスエックス)

# 自己紹介

**2013.6～2013.8**

Microsoft Research Intern  
博士(情報理工学)、東京大学

**2014.4～2016.3**

NTT CS研 研究員

**2016.4～2018.9**

東京大学 講師 (原田牛久研究室)

**2016.9～**

産業技術総合研究所 協力研究員

**2016.12～2018.9**

国立国語研究所 共同研究員

**2018.10～**

オムロンサイニックス株式会社 Principal Investigator

**2019.1～**

株式会社 Ridge-i 取締役 Chief Research Officer

**2020.4～**

津田塾大学 非常勤講師

**2021.7～**

東北大学 非常勤講師

## 所属団体

ACM、IEEE、電子情報通信学会、情報処理学会、日本ロボット学会、人工知能学会、  
応用物理学会、建築情報学会、日本ディープラーニング協会

### 画像キャプション生成

[Ushiku+, ACMMM 2012]

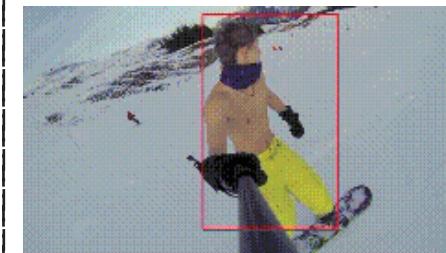
[Ushiku+, ICCV 2015]



A yellow train on the tracks  
near a train station.

### 動画の特定区間と

キャプションの相互検索  
[Yamaguchi+, ICCV 2017]



A guy is skiing with no shirt on  
and yellow snow pants.

# 本講演について

Transformerの基本的な動作から応用範囲、最近のMLP系ネットワークまでの俯瞰

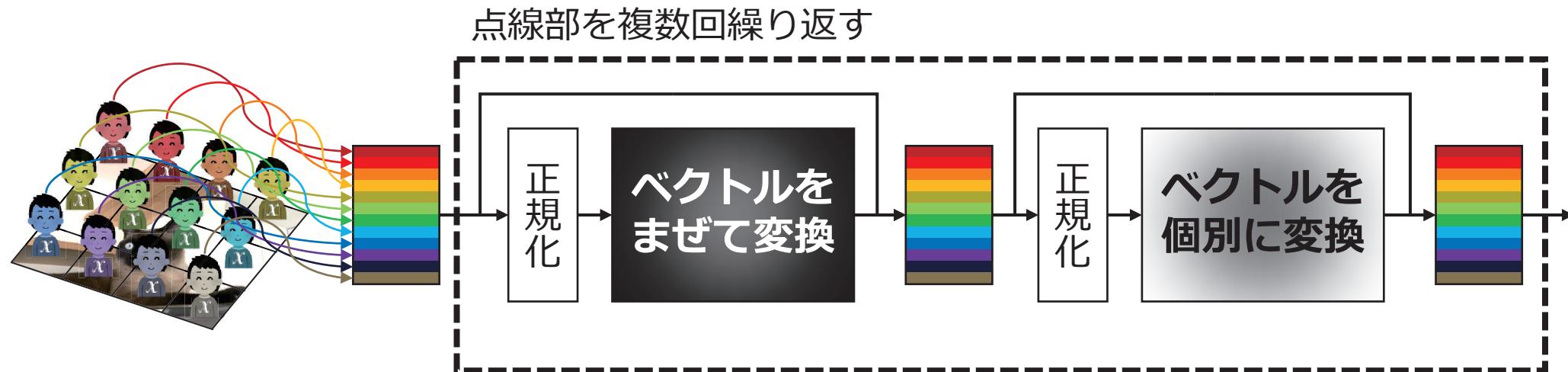
- そもそもTransformerって？
- Transformer旋風
- Transformerはオワコン？！
- Transformerのノウハウ



←本講演の資料の最新バージョン  
(サブタイトルにも掲載しております)

# 色々聞きすぎて良く分からぬ！という人のために

- Transformerはトークン（単語やパッチなど）をResidual接続しながら2つのモジュールを繰り返しているだけ！
  - トークンを混ぜるToken Mixer
  - トークンを変換するMLP
- 話題になったMLP系も実は基本的に同じ構造！

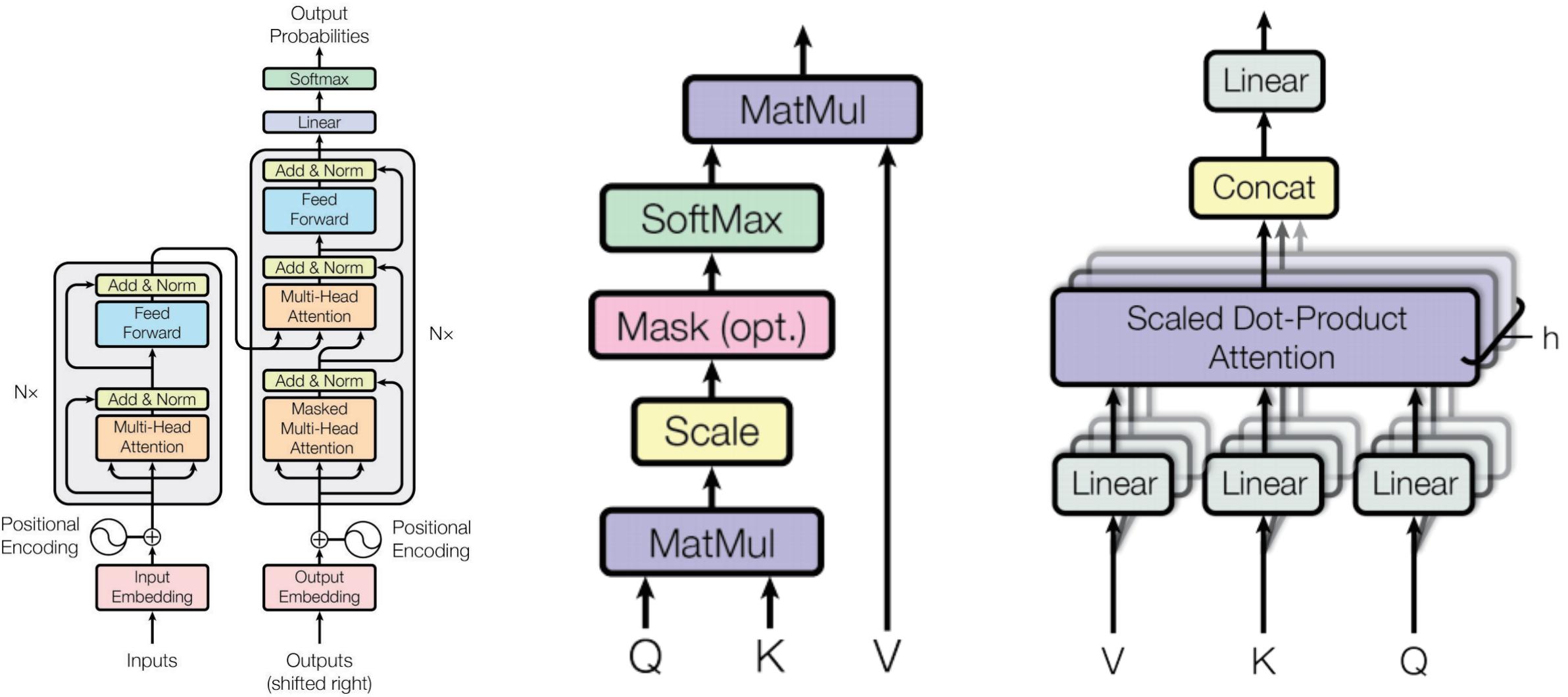


- ...あとは色々とノウハウがあるので気を付けましょう

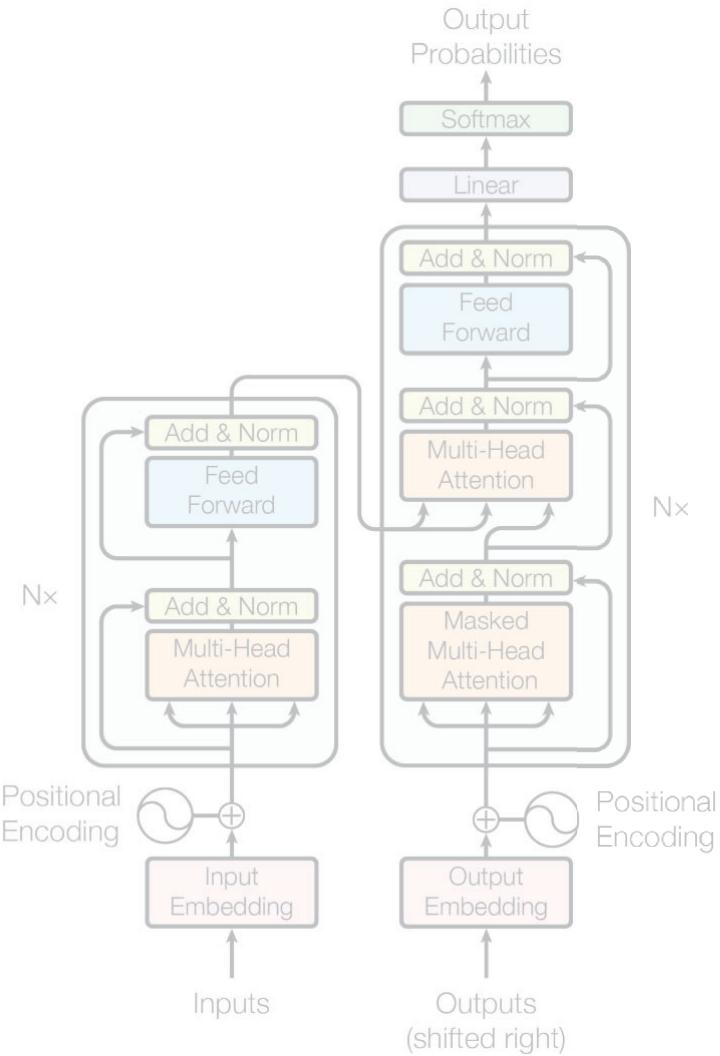


そもそもTransformerって？

# Transformer系論文でよく見る図

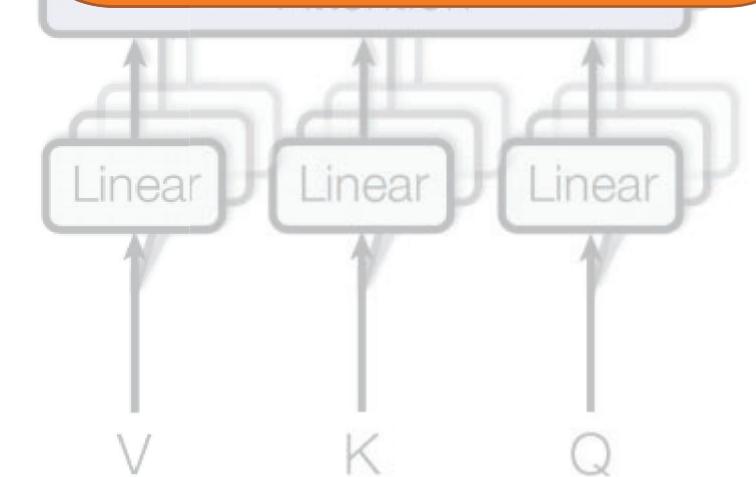


# Transformer系論文でよく見る図



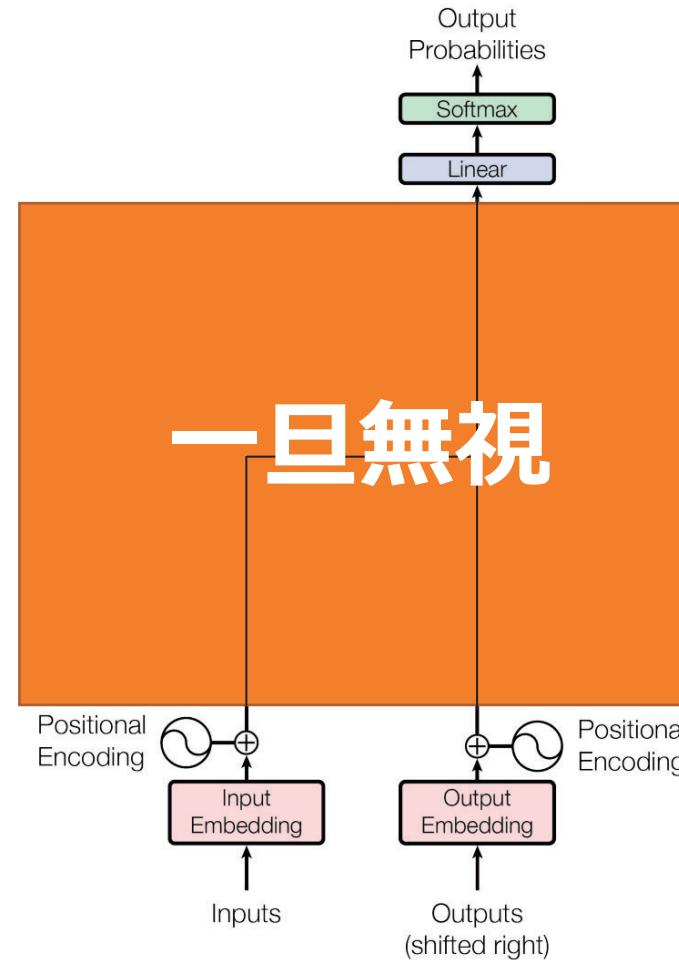
綺麗な図だし  
分かりやすく見える  
けど分かりにくい

※個人の感想です



# Transformerとは

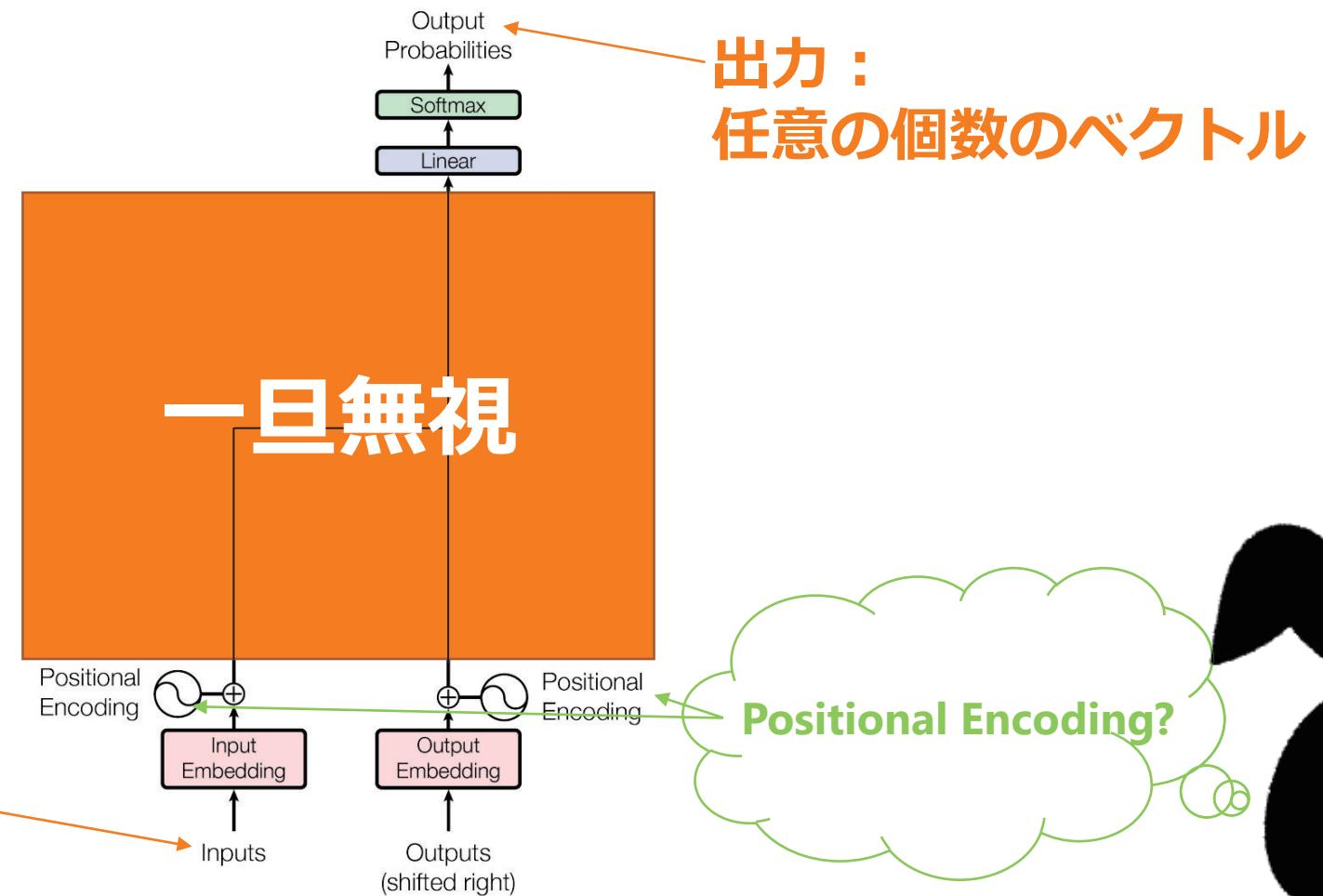
任意の個数のベクトルを任意の個数のベクトルに変換する  
ネットワーク



# Transformerとは

任意の個数のベクトルを任意の個数のベクトルに変換する  
ネットワーク

入力：  
任意の個数のベクトル



# Positional Encoder

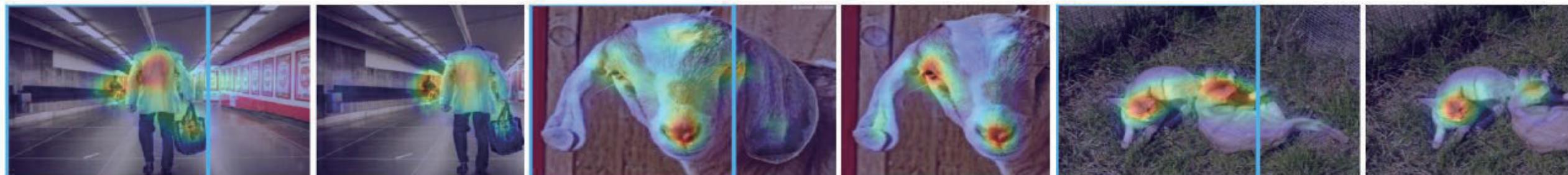
入力ベクトルの**位置**と出力ベクトルの**位置**を表すベクトル

- 単語の分散表現なら「文中の何単語目？」
- 画像の特徴量なら「画像内のどこの座標？」
- 時系列の特徴量なら「何秒時点の情報？」



**RNN**や**CNN**は各特徴量の位置情報を知っている

- 「**RNN**も**CNN**も各要素の**相対位置**を知っている」 [Shaw+, NAACL-HLT'18][Dai+, ACL'19]
- 「**CNN**は各要素の**絶対位置**を学習している」 [Islam+, ICLR'20]



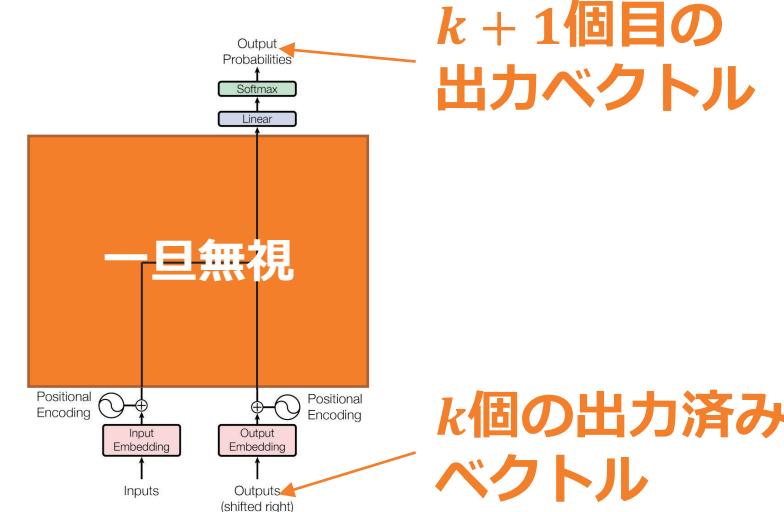
画像の顕著性推定で、画像全体を入力 vs. 一部の画像を入力→それぞれ“画像の中央”は顕著性が大きい

# Autoregressive vs. Non-autoregressive

出力のベクトルを...

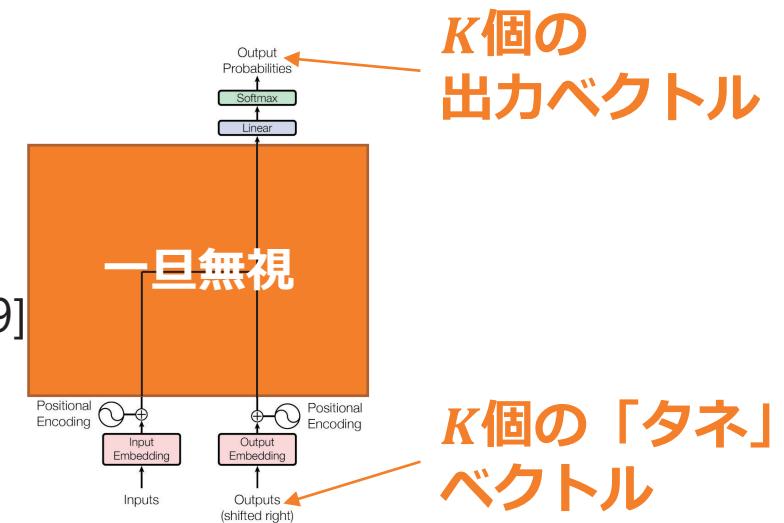
- **Autoregressive= 1個ずつ推定する**

- 入力のベクトル群と出力済みのベクトル群を用いて新たなベクトルを出力
- 「終わり」を意味する単語が出るまで繰り返す
- 元のTransformer [Vaswani+, NIPS'17] はAutoregressive
- 精度は良いが遅い

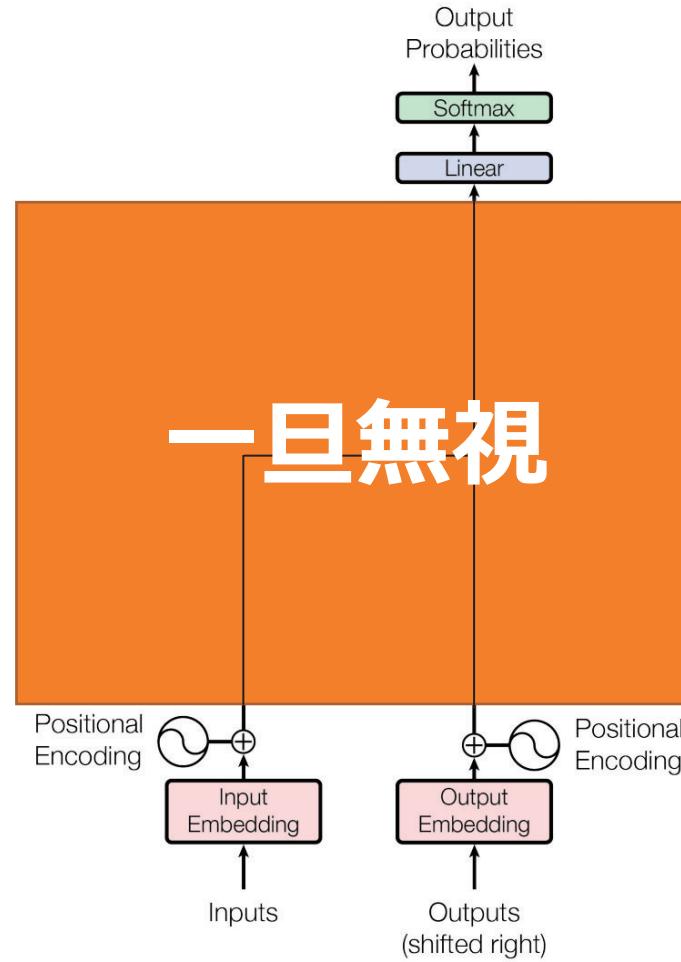


- **Non-autoregressive=全部一気に推定する**

- 何らかの方法で複数個の出力ベクトル用の「タネ」となるベクトルを入力 [Gu+, ICLR'18][Gu+, AAAI'19]
  - 並列で1回で計算できるので速いが、精度が低下
- 一気に推定するのを繰り返して単語数を調整 [Ghazvininejad+, EMNLP'19][Gu+, NeurIPS'19]



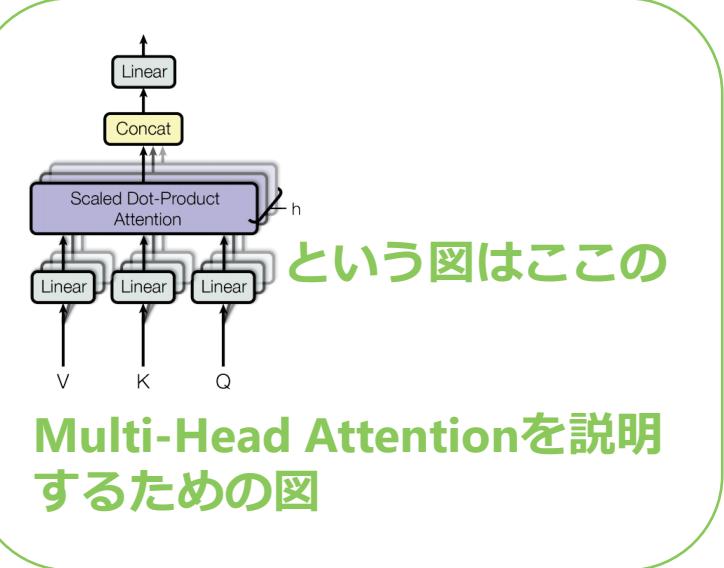
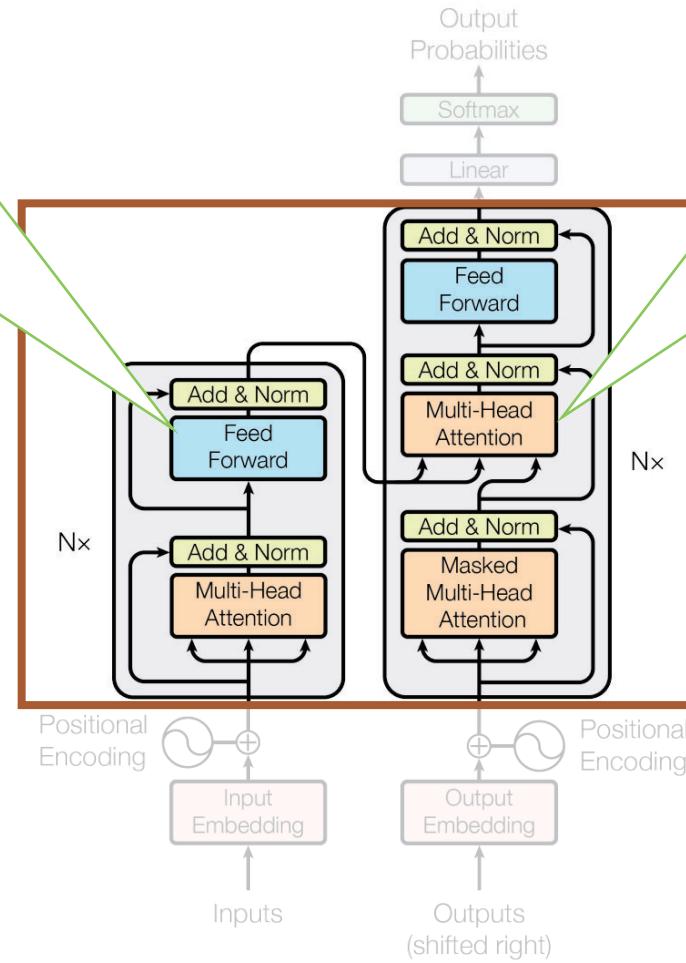
# きっとここまでok



# 次は無視していた部分の中身

ここは

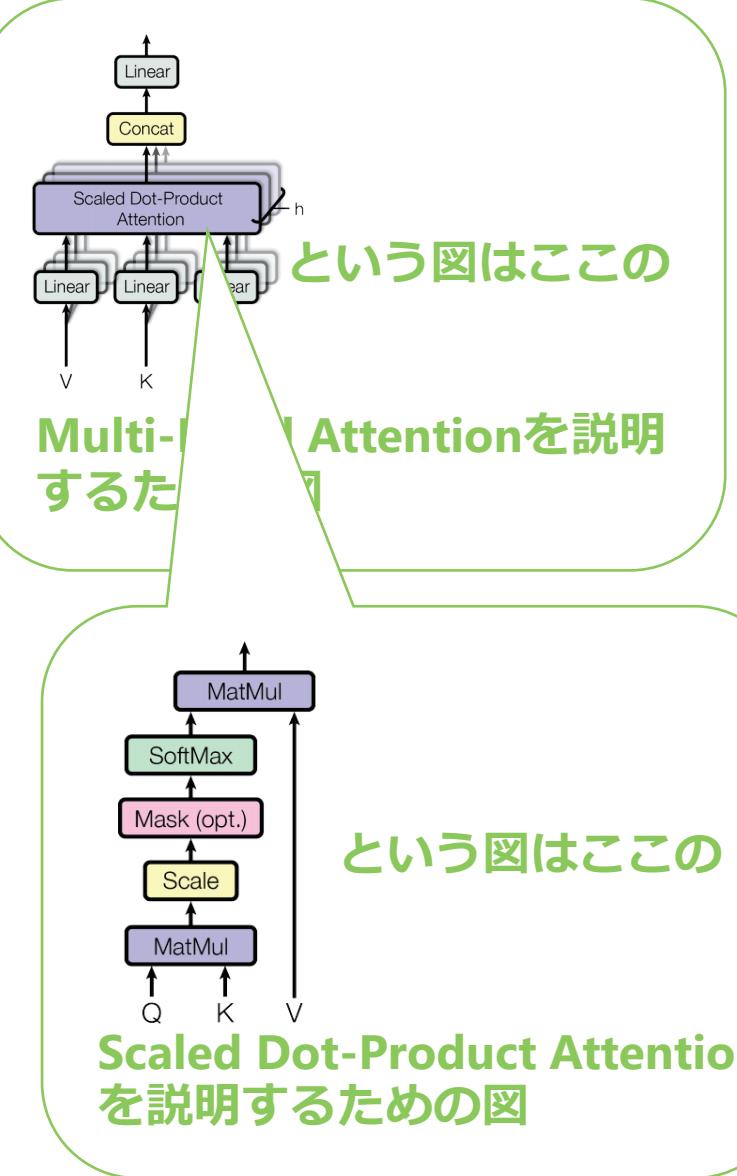
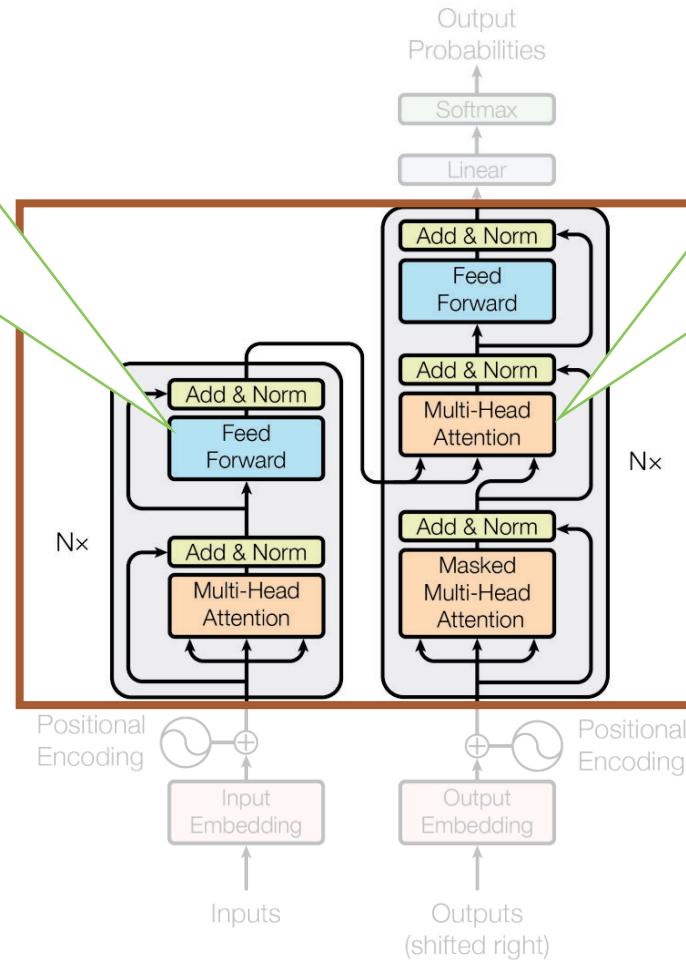
- Residual接続
- 各ベクトルにMLPを適用  
= 1x1 conv と等価



# 次は無視していた部分の中身

ここは

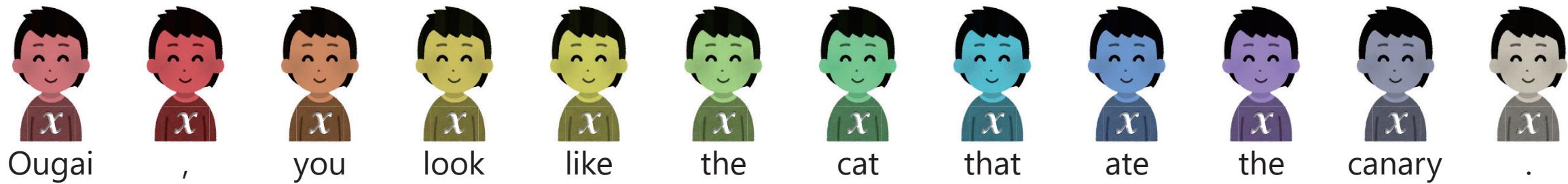
- Residual接続
- 各ベクトルにMLPを適用  
= 1x1 conv と等価



# Multi-head Attention

おさらい： 扱うのは任意の個数のベクトル

- 自然言語処理であれば单語の分散表現の系列 + 位置情報



- 画像認識であればタテ×ヨコにチャネル数分の次元のベクトル（局所特徴量）が並んだ数列 + タテ・ヨコの位置情報

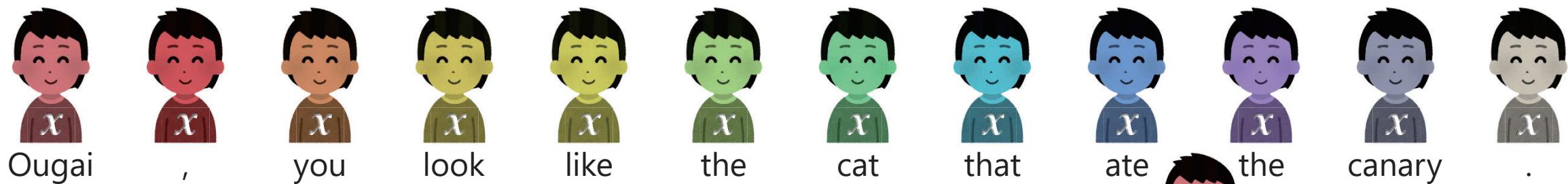


鷗外さん♂

# Multi-head Attention

おさらい： 扱うのは任意の個数のベクトル

- 自然言語処理であれば单語の分散表現の系列 + 位置情報



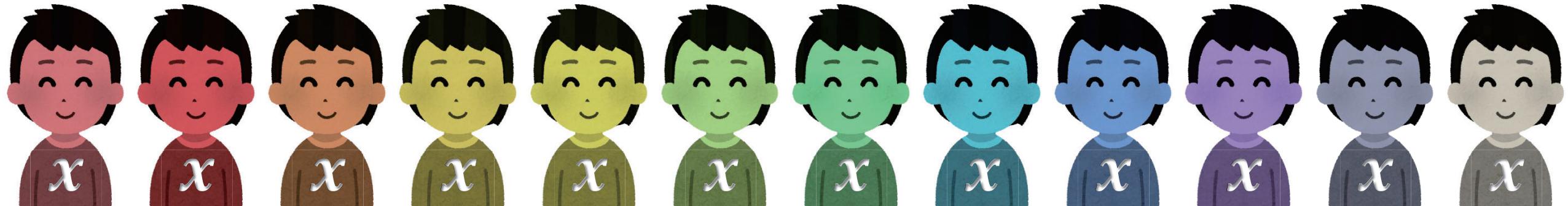
- 画像認識であればタテ×ヨコにチャネル数分の次元のベクトル（局所特徴量）が並んだ数列 + タテ・ヨコの位置情報



# このベクトル に注目して考える

1. 他のベクトルを「検索」するためのクエリ $q = W^Q x$ を計算

$$q = W^Q x$$

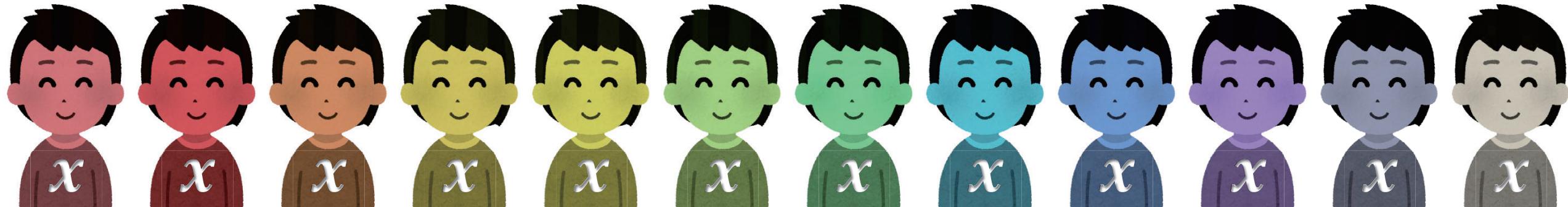


# このベクトル に注目して考える

2. 「類似検索」対象としての各ベクトルのキー  $k = W^K x$  を計算

$k \quad k \quad k$

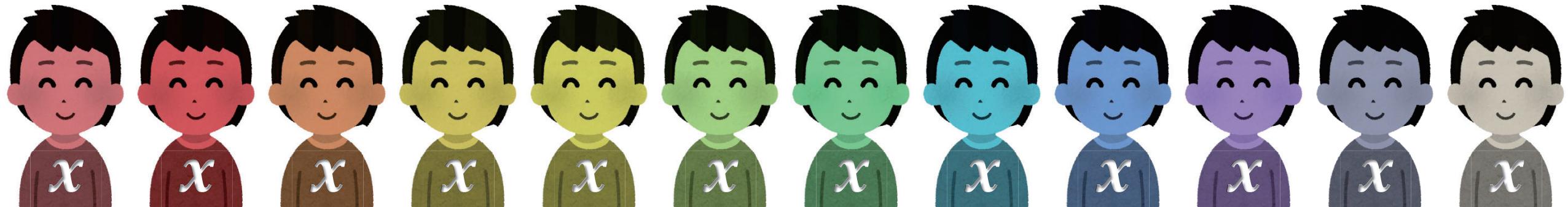
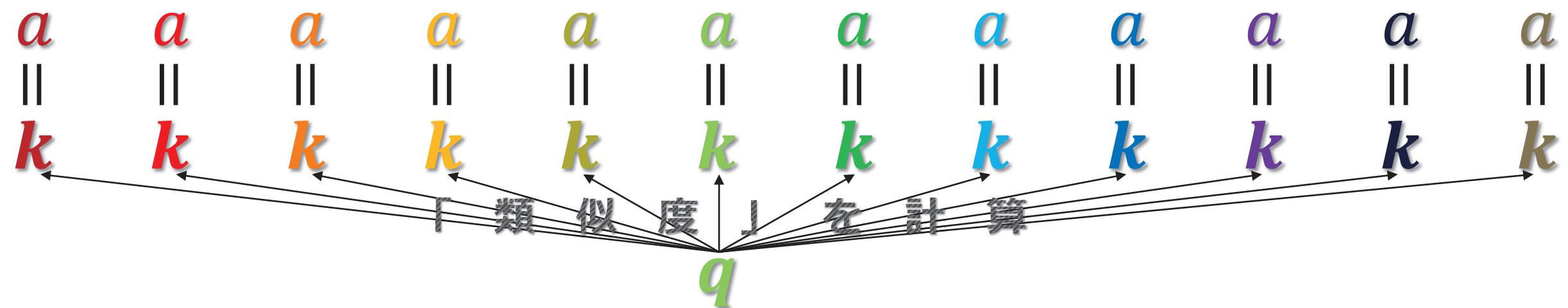
$q$



# このベクトル に注目して考える

3. クエリとキーの内積を次元数 $d$ とsoftmaxで正規化した「類似度」

$$a = \text{softmax}(q^\top k / \sqrt{d})$$
 を計算

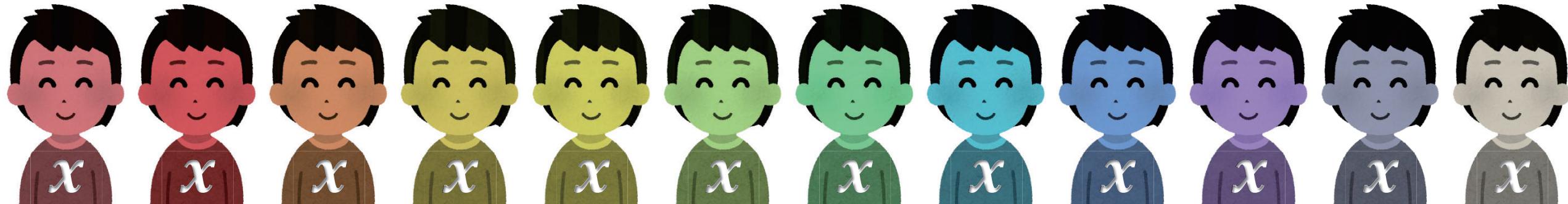


# このベクトル に注目して考える

4. 各ベクトルの代表値  $v = W^T x$  を計算

$v$      $v$

$a$      $a$



# このベクトル に注目して考える

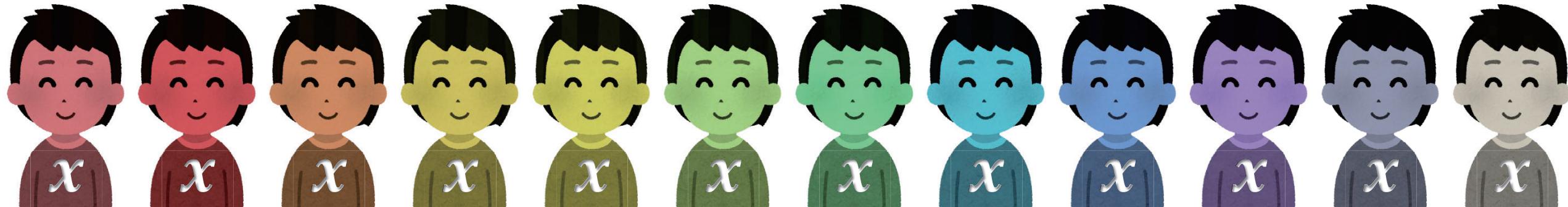
5. 類似度 $a$ で重みづけした和 $\sum av$ を計算→ベクトル  に加算 (residual接続有)



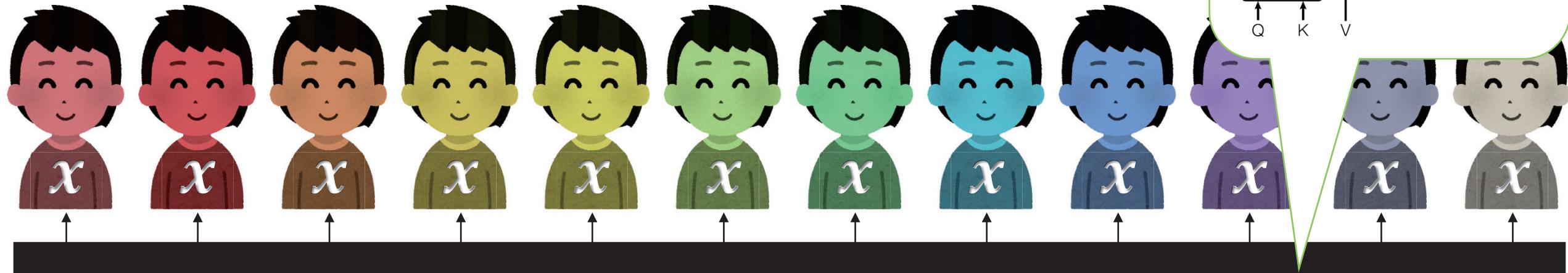
更新後のベクトル

重み付け和を計算

$v$												
.	.	.	.	.	.	.	.	.	.	.	.	.
$a$												

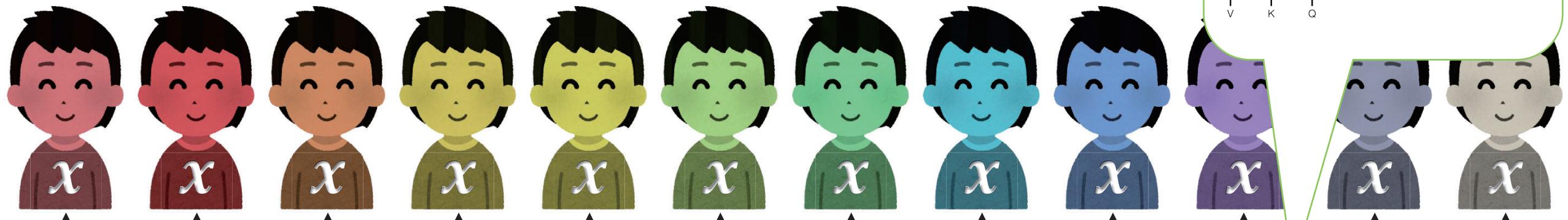


# Scaled Dot-Product Attention

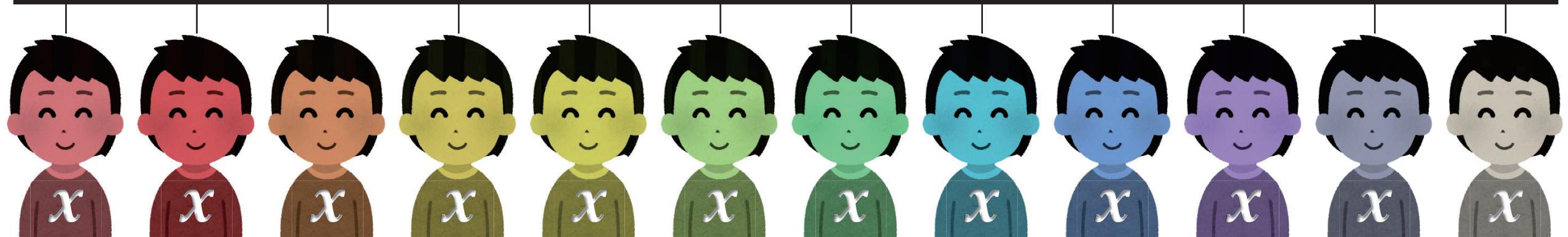


手順1~5をすべてのベクトルに対して実行する

# Multi-head Attentionでは



行列  $W^Q, W^K, W^V$  を  $h$  個用意し、それぞれを用いながら  
手順1~5をすべてのベクトルに対して実行する



# Transformer まとめ

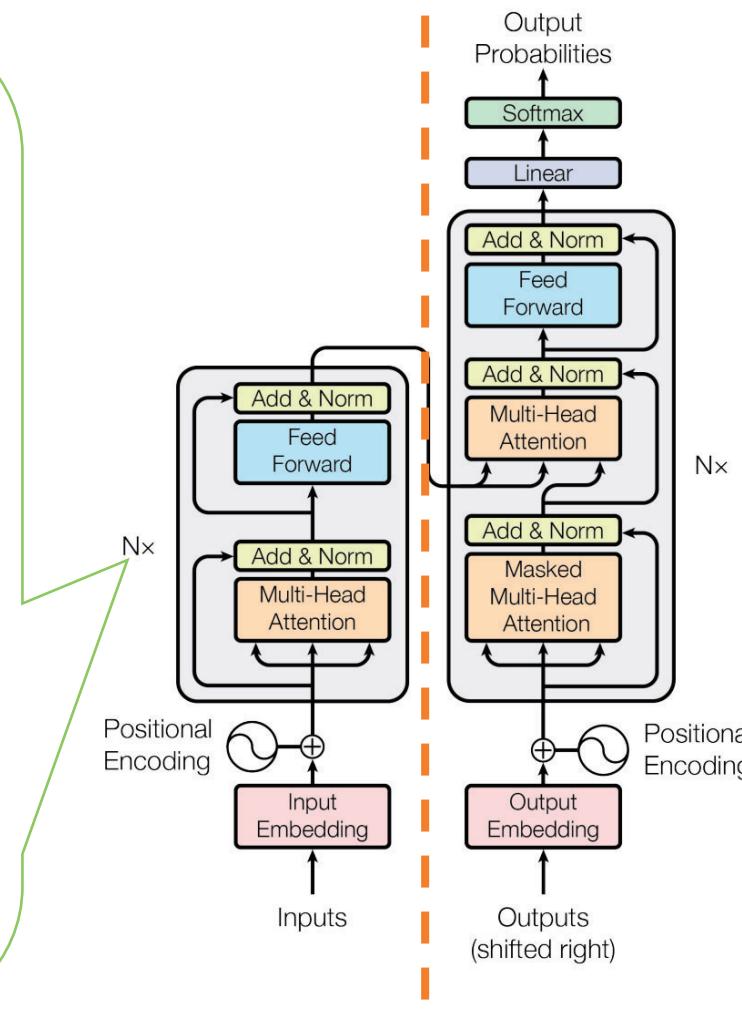
任意の個数のベクトルを **Encoder|Decoder** 形式で**変形**する技術

N回繰り返す

例えばGPT-3では

- 96回繰り返す
- 各回96-headsの  
Attention (128次  
元)  
→パラメータ数 175B

※2022年4月にGoogle  
が公開したPaLMは540B  
[Chowdhery+, 2022]



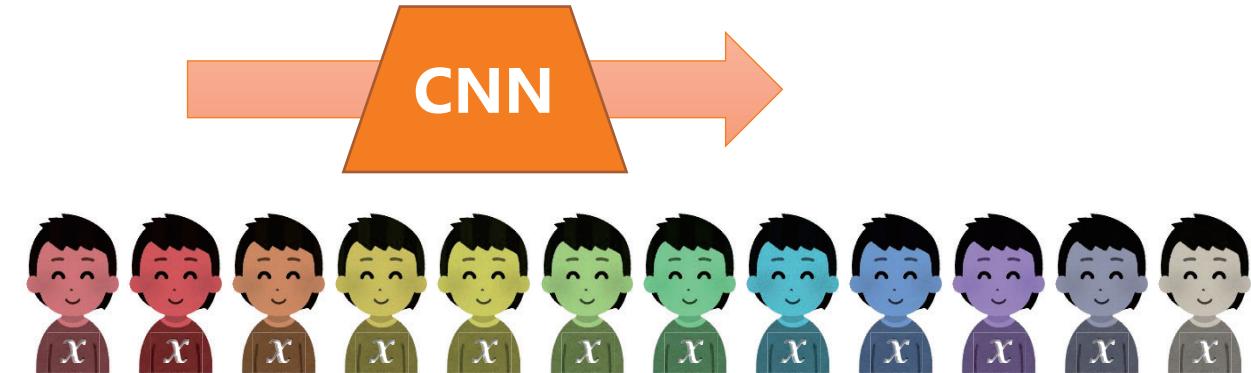
# (Attentionの無い) CNNやRNNとの違い

**Transformer**は...

全 $n$ 個のベクトルから全 $n$ 個のベクトルへのアテンションを計算  
 $O(n^2)$ だが一番広域に情報がロス無く伝達可能

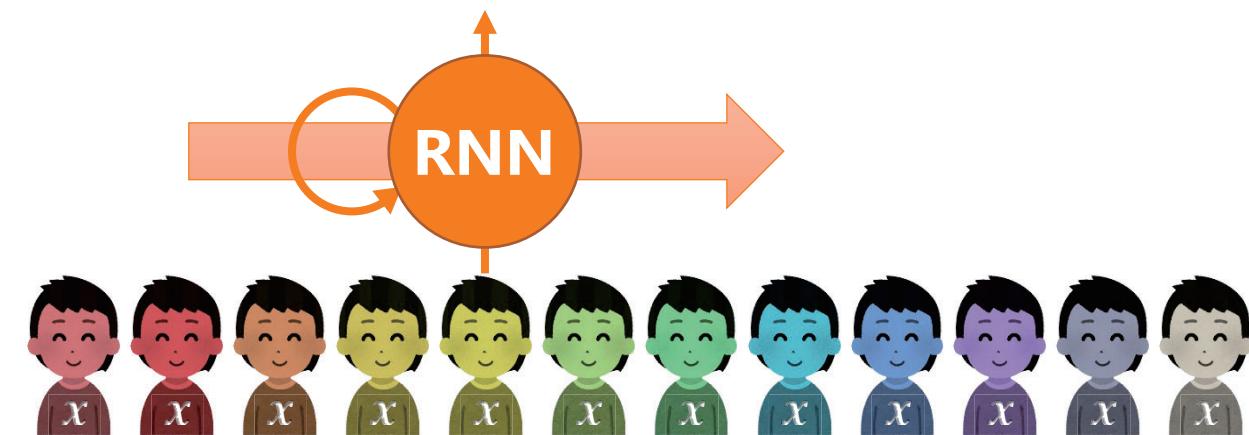
**CNN**は...

3つなど、全体からすれば少数の  
ベクトルだけの畳込み計算を走査  
 $O(n)$ だが情報が伝わるのは近隣だけ



**RNN**は...

ベクトルを一つずつ走査しながら  
内部のセルに変数（記憶）を保存  
 $O(n)$ だが長い系列は不得手





Transformer旋風

# Computer Vision以外

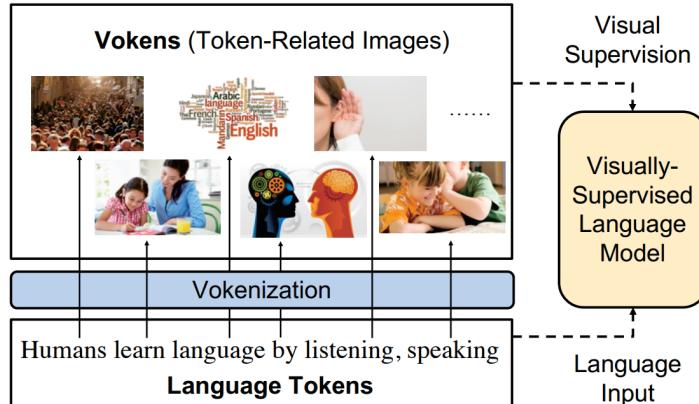
Self-attentionは更に広いので省略

- **自然言語処理**
  - 翻訳かつ原著 [Vaswani+, NIPS'17] その他多数！！！
  - 言語モデル GoogleのBERT [Devlin+, NAACL-HLT'19], PaLM [Chowdhery+, 2022], DeepMindのGopher [Rae+, 2022], OpenAIのGPT-2/3 [Radford+, 2019][Brown+, NeurIPS'20] など多数
  - 2兆トークンからなるデータベースの検索 [Borgeaud+, 2021]
- **音声処理・信号処理**
  - 表現学習 HuBERT [Hsu+, TASLP'21], SSAST [Gong+, AAAI'22]
  - 音声認識 [Lüscher+, INTERSPEECH'19]
  - 音楽生成 [Huang+, ICLR'19][Choi+, ICML'20]
  - 時系列予測 [Li+, NeurIPS'19][Wu+, NeurIPS'20]
- **テーブルデータ**
  - FT-Transformer [Gorishniy+, NeurIPS'22] ※ただし表データは依然としてGradient Boostingが強い
- **Bio/Chem-informatics**
  - 分子構造解析 [Fuchs+, NeurIPS'20][Rong+, NeurIPS'20]
- **エージェント・ロボティクス**
  - マルチエージェント通信 [Inala+, NeurIPS'20]
  - One Shotで模倣学習 [Dasari+Gupta, CoRL'20]
  - タスク系列の強化学習 Scene Memory Transformer [Fang+, CVPR'19], Decision Transformer [Chen+, NeurIPS'21], Trajectory Transformer [Janner+, NeurIPS'21], Gato [Reed+, 2022]

# Vision & Language

## 表現学習→基盤モデル

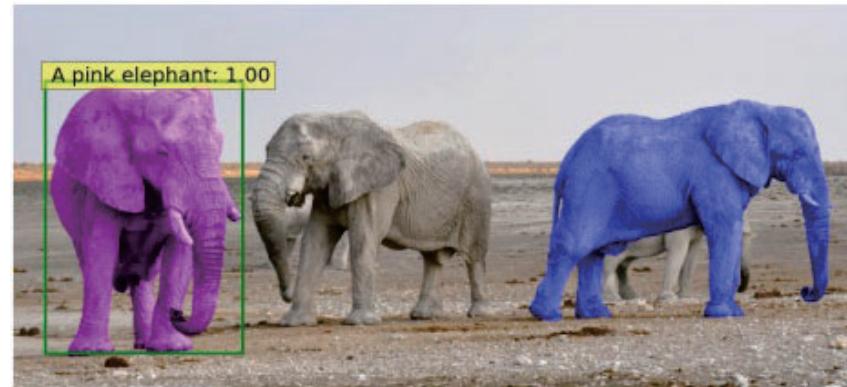
VideoBERT [Sun+, ICCV'19]  
LXMERT [Tan+Bansal, EMNLP'19]  
ViLBERT [Lu+, NeurIPS'19]  
VL-BERT [Su+, ICLR'20]  
UNITER [Chen+, ECCV'20]  
OSCAR [Li+, ECCV'20]  
Voken [Tan+Bansal, EMNLP'20]  
COOT [Ging+, NeurIPS'20]  
Perceiver [Jaegle+, ICML'21]  
PolyViT [Likhosherstov+, 2021]  
Flamingo [Alayrac+, 2022]



[Tan+Bansal, EMNLP'20]

## 参照表現理解

MDETR [Kamath+, ICCV'21]



## キャプション生成

[Zhou+, CVPR'18][Li+, ICCV'19][Cornia+, CVPR'20]



**GT:** A cat looking at his reflection in the mirror.  
**Transformer:** A cat sitting in a window sill looking out.

**$\mathcal{M}^2$  Transformer:** A cat looking at its reflection in a mirror.

**GT:** A plate of food including eggs and toast on a table next to a stone railing.

**Transformer:** A group of food on a plate.

**$\mathcal{M}^2$  Transformer:** A plate of breakfast food with eggs and toast.

[Cornia+, CVPR'20]

## TextVQA [Kant+, ECCV'20]



What sponsor is to the right of the players?

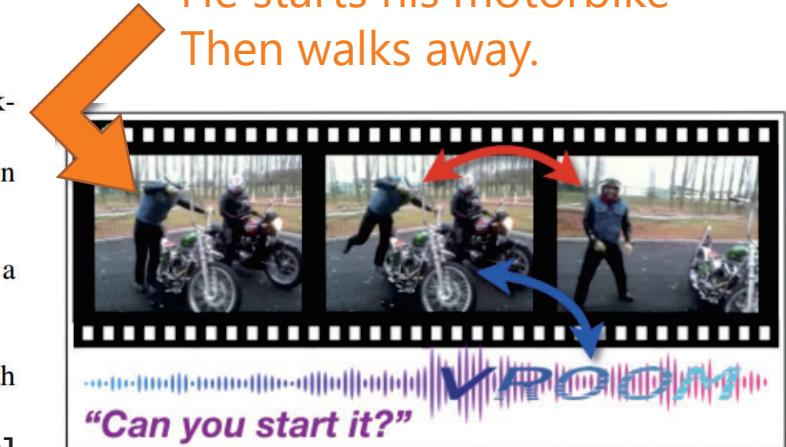
- WARNER    ● AHL  
● AHL    ● RBC

Previous Model:  
● AHL X

Our Model:  
● RBC ✓

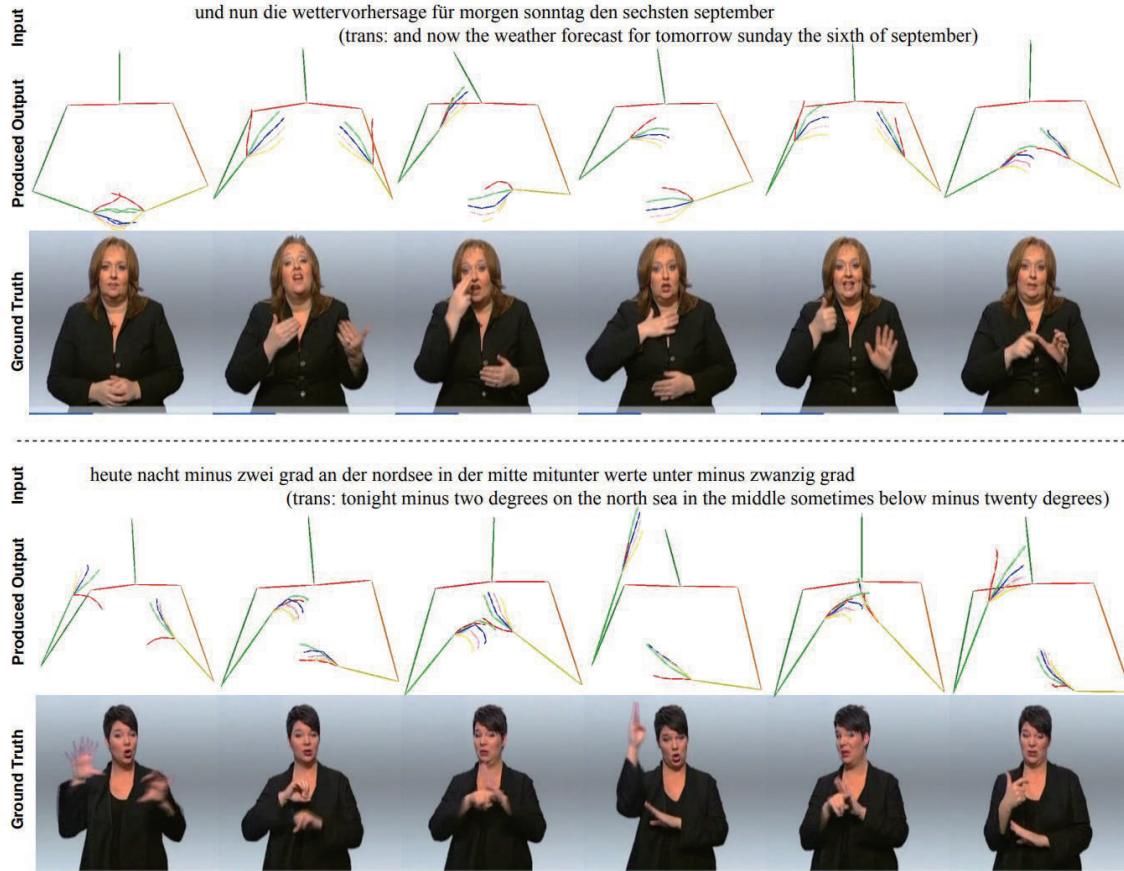
## 動画検索 [Gabeur+, ECCV'20]

He starts his motorbike  
Then walks away.

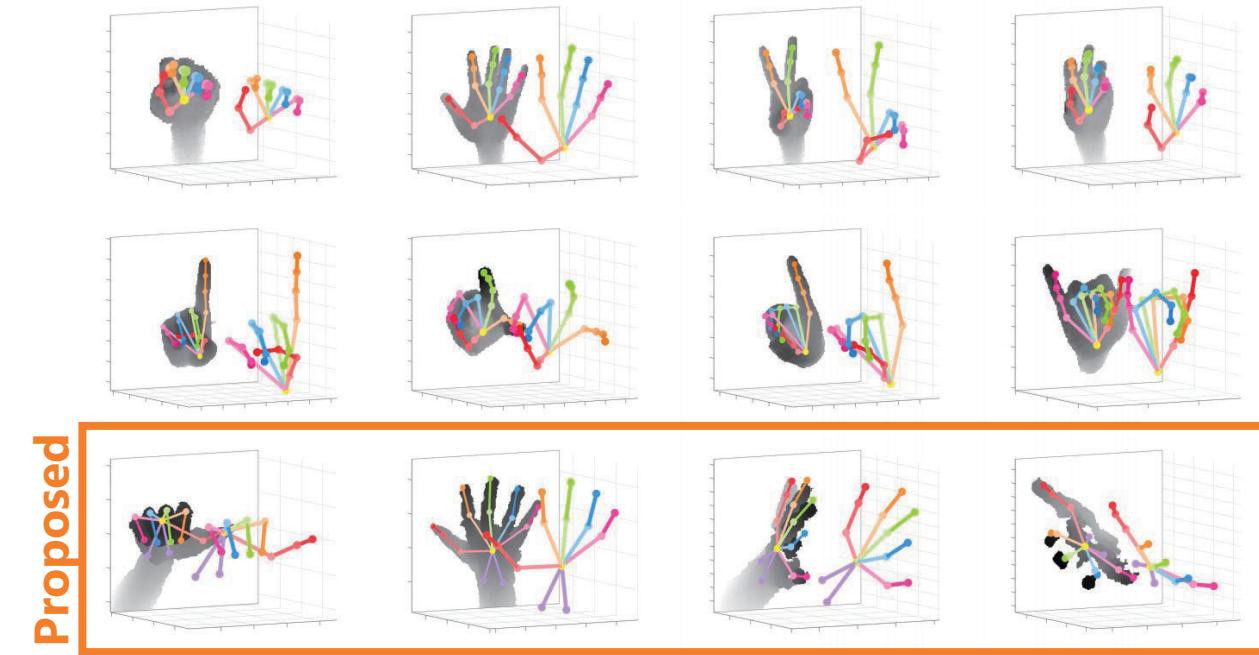


# 姿勢推定系

手話の認識 [Saunders+, ECCV'20]



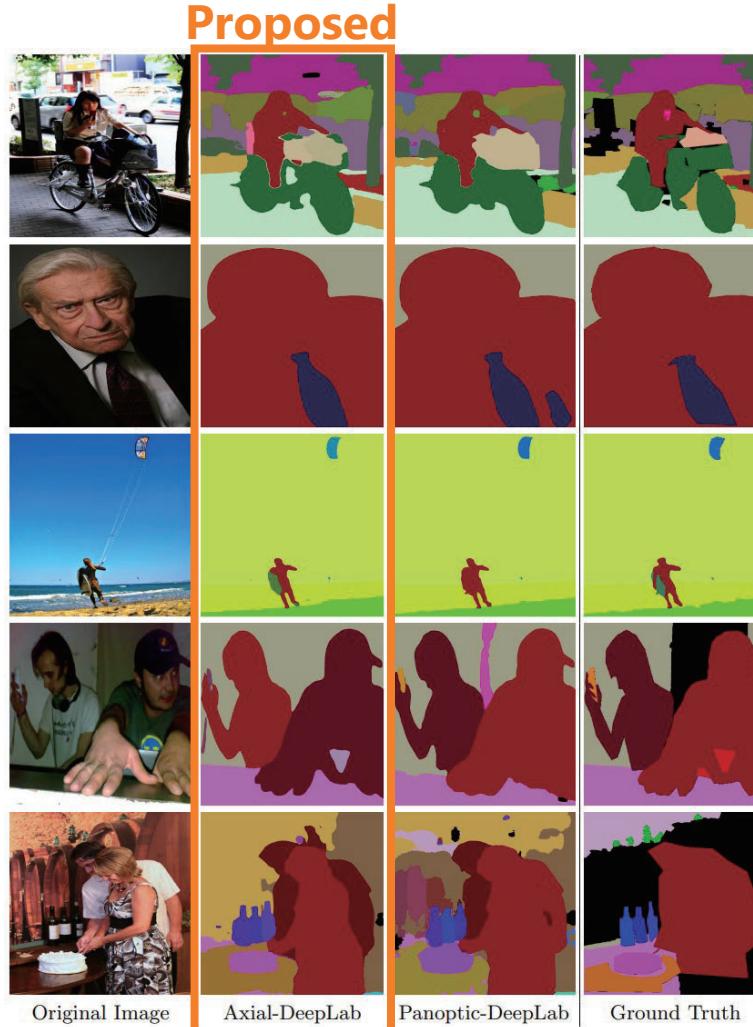
手の姿勢 [Huang+, ECCV'20]



# 領域分割系

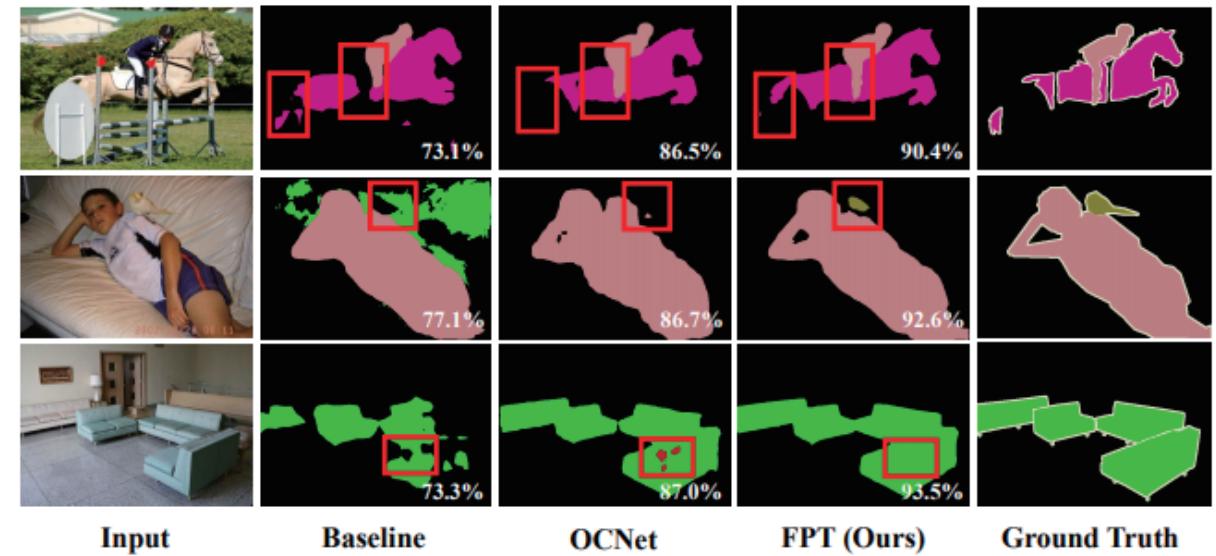
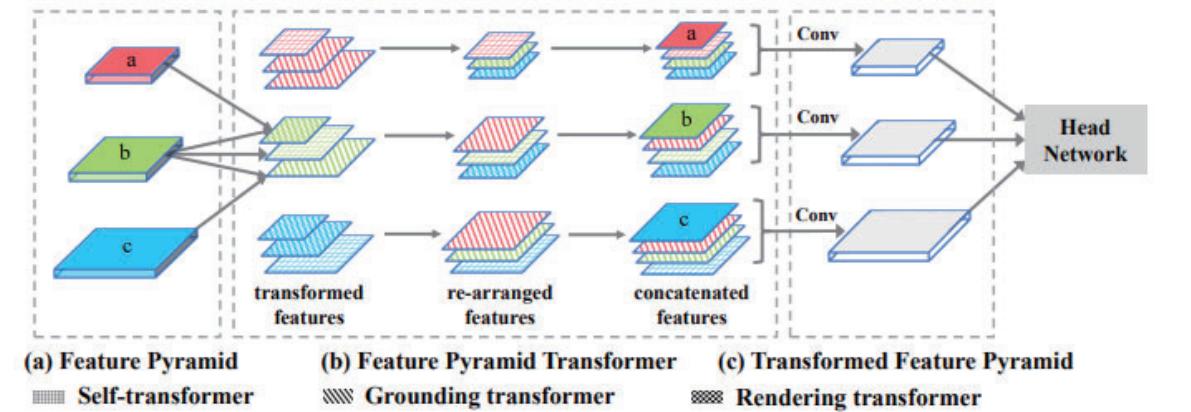
## Panoptic Segmentation

[Kirillov+, ECCV'20]



## Semantic Segmentation にも Instance Segmentation にも [Zhang+, ECCV'20]

See also  
[Liang+, CVPR'20]



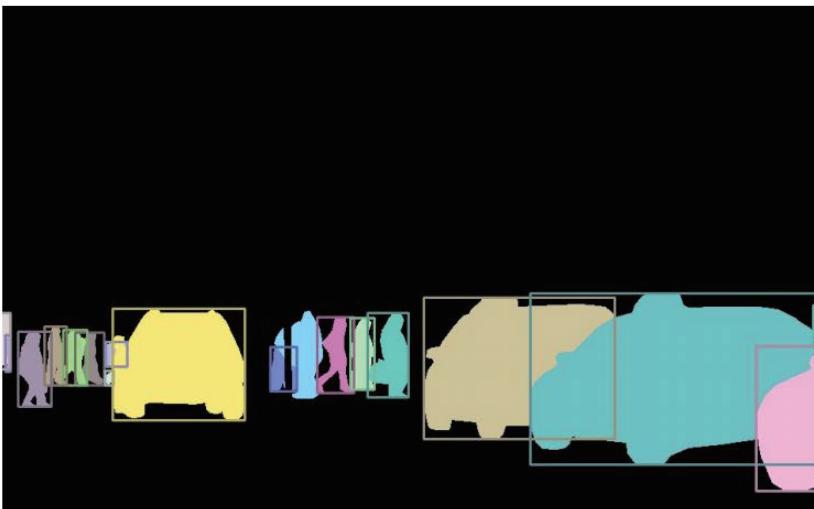
# 注：xxxx Segmentationがたくさんある？？？



(a) image



(b) semantic segmentation



(c) instance segmentation

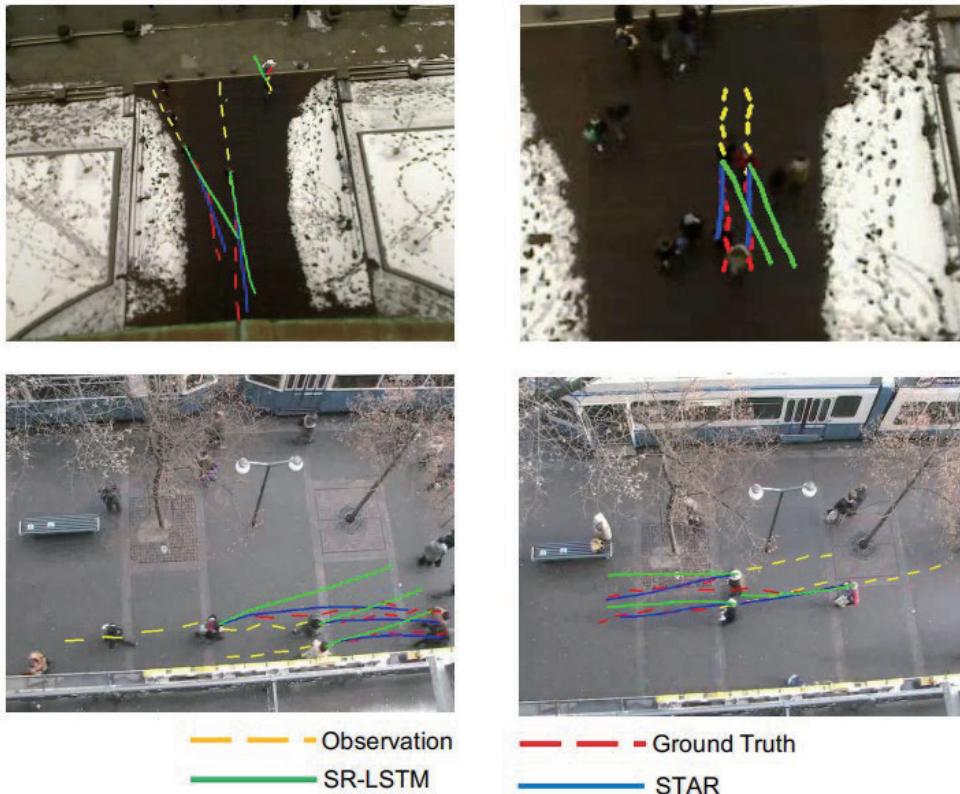


(d) panoptic segmentation

[Kirillov+, CVPR 2019]

# 動画像理解

## 経路予測 [Yu+, ECCV'20]



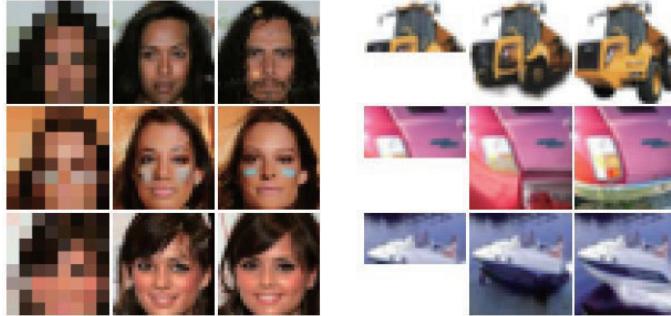
## 動作認識 [Girdhar+, ECCV'20]



# その他の例

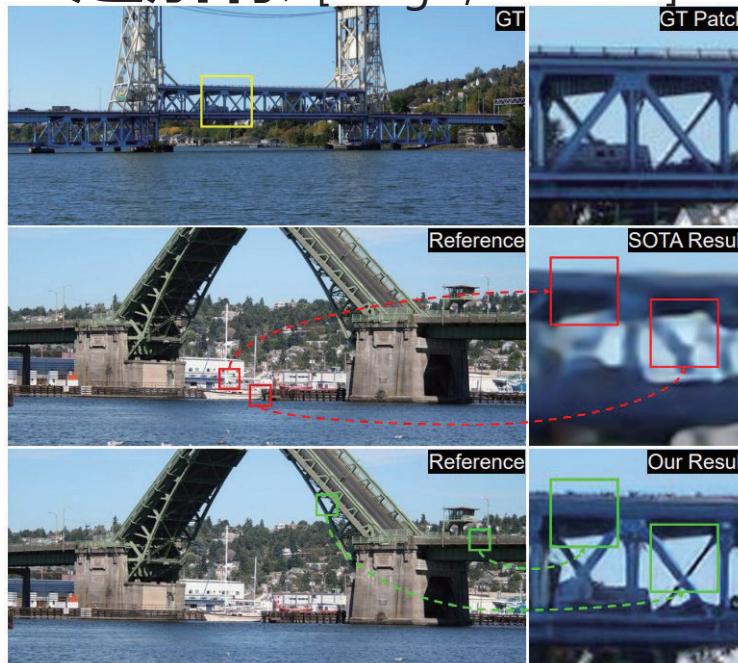
## 画像生成モデル 超解像 + 補完

[Parmar+, ICML'18]



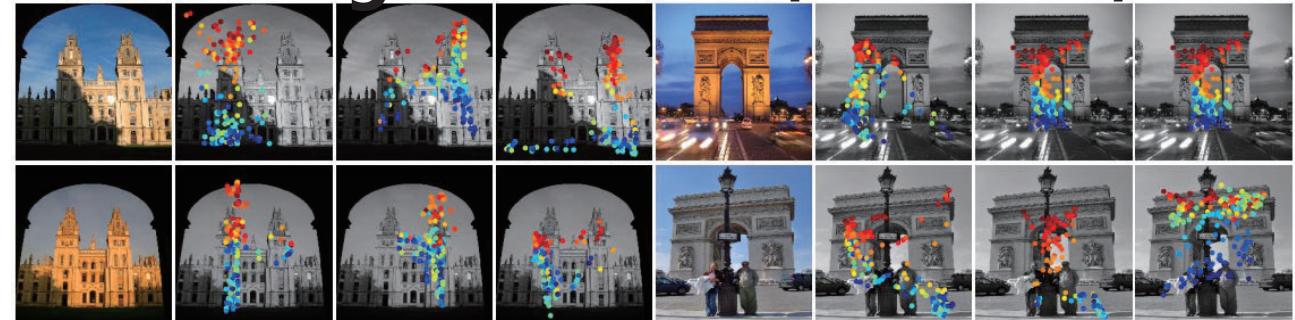
## 超解像

[Yang+, CVPR'20]



## 特定物体検索 Fine-grained識別

[Kim+, ECCV'20]



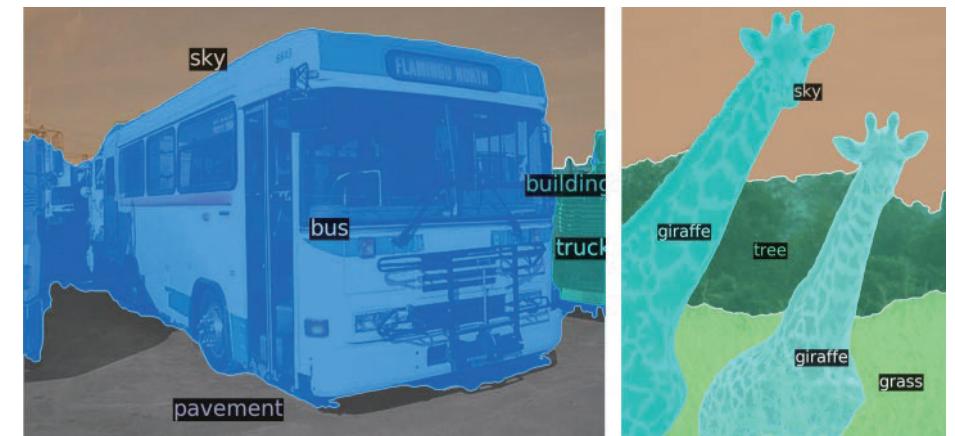
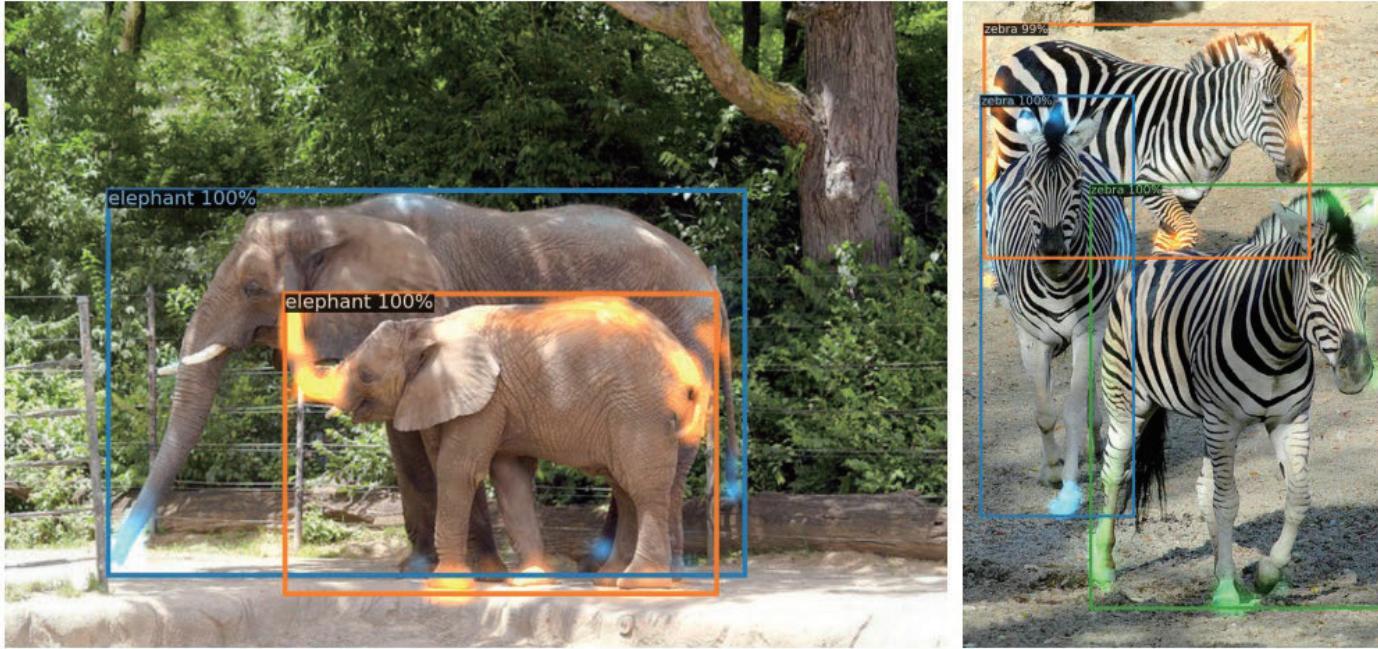
## スケッチで検索

[Ribeiro+, CVPR'20]

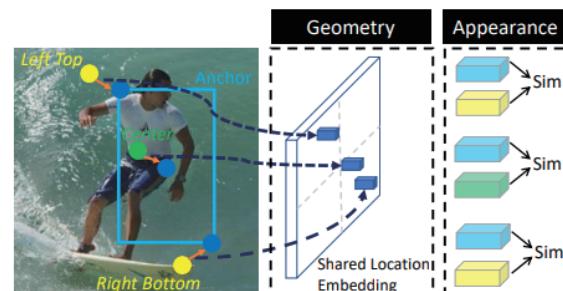
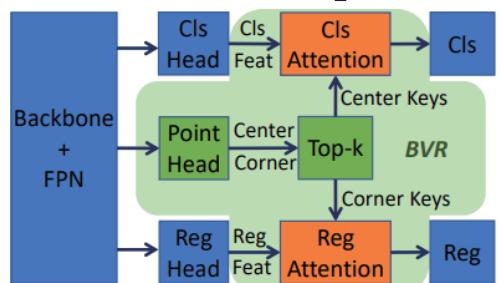


# 物体検出 (DETR)

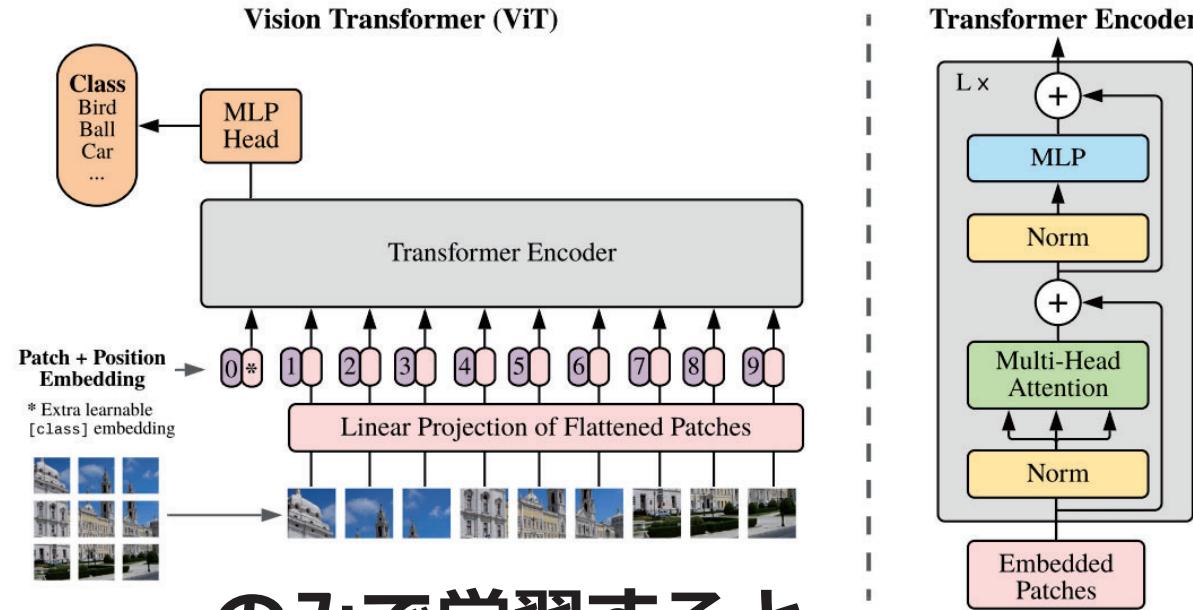
物体検出（左）とPanoptic Segmentation（右） [Carion+, ECCV'20]



物体検出は[Chi+, NeurIPS'20]も提案



# Vision Transformer [Dosovitskiy+, ICLR'21]

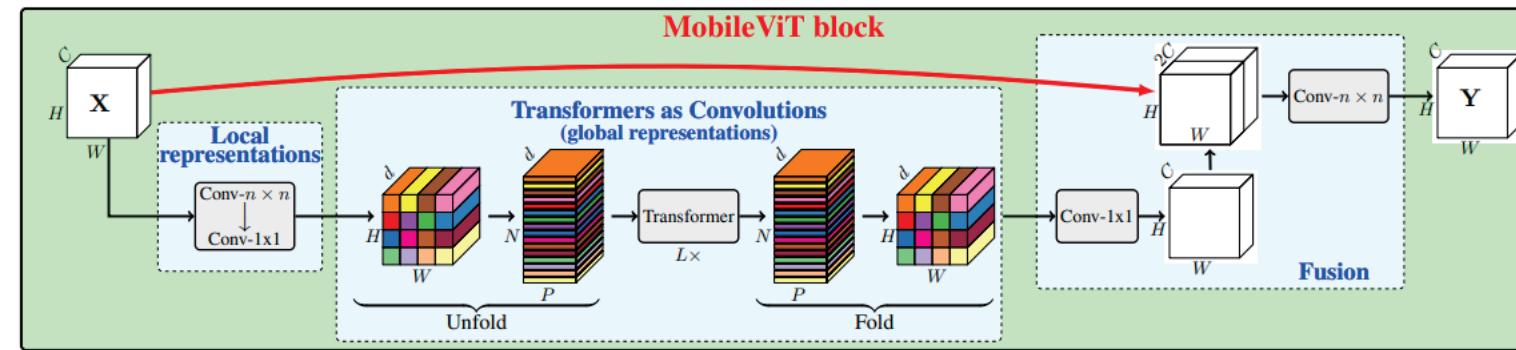
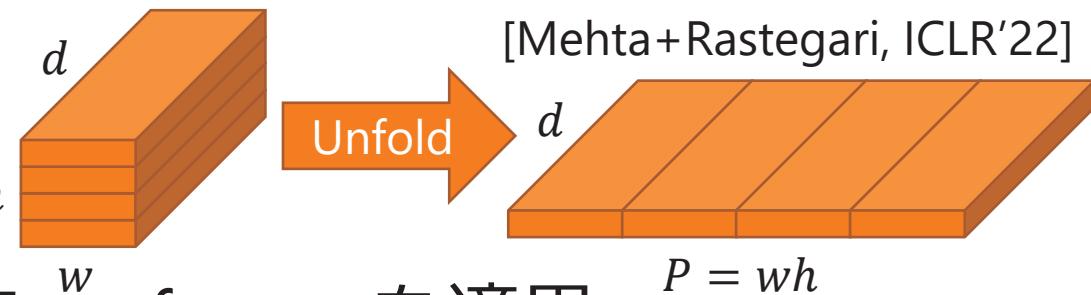


- **画像からTransformerのみで学習すると**
  - ResNet152層とほぼ同等の精度かつ25%程度の学習時間
  - JFT-300Mという大規模データセットが必要
  - 知識蒸留を組み合わせて、ImageNetの1000クラス画像データのみの学習でも EfficientNetを超えたDeiTも有名 [Touvron, ICML'21]
- **エンコーダのみのTransformer**
  - 実はオリジナルのTransformerよりも構造が単純で理解も容易

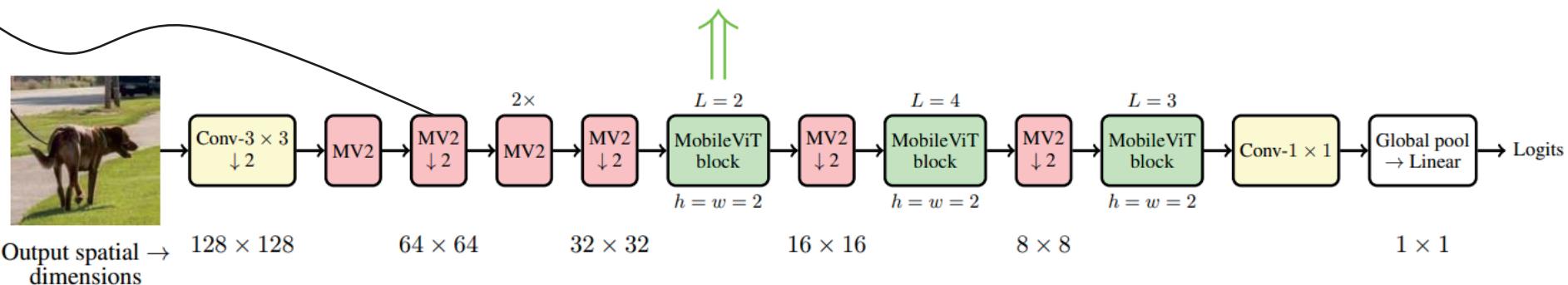
# MobileViT

- 中央のブロックでは...

- 各パッチを左右方向に展開 (Unfold) <sup>$h$</sup>
- 縦にスライスしたベクトル集合毎にTransformerを適用
  - 言い換えると、 $d$ 次元ベクトルが $N$ 個ある集合上でのTransformerを $P$ 回適用
- もう一度各パッチを正方形に戻す (Fold)



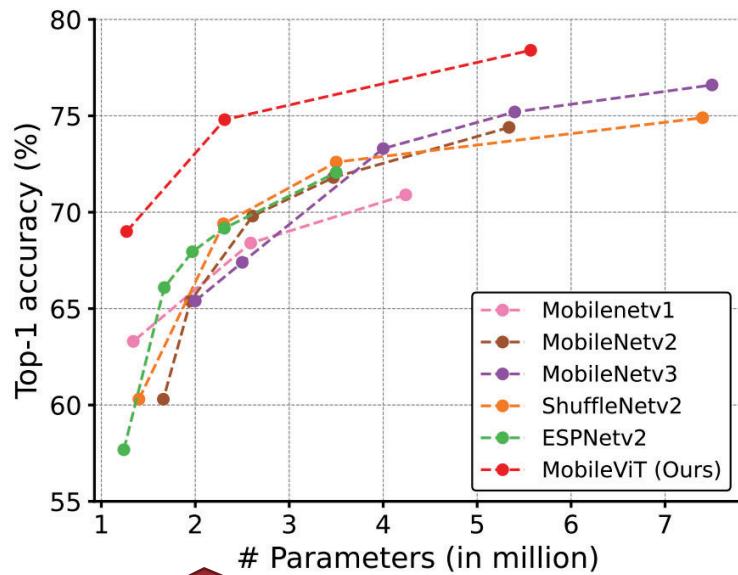
MV2: MobileNet v2 block  
 $\downarrow 2$ : down sampling



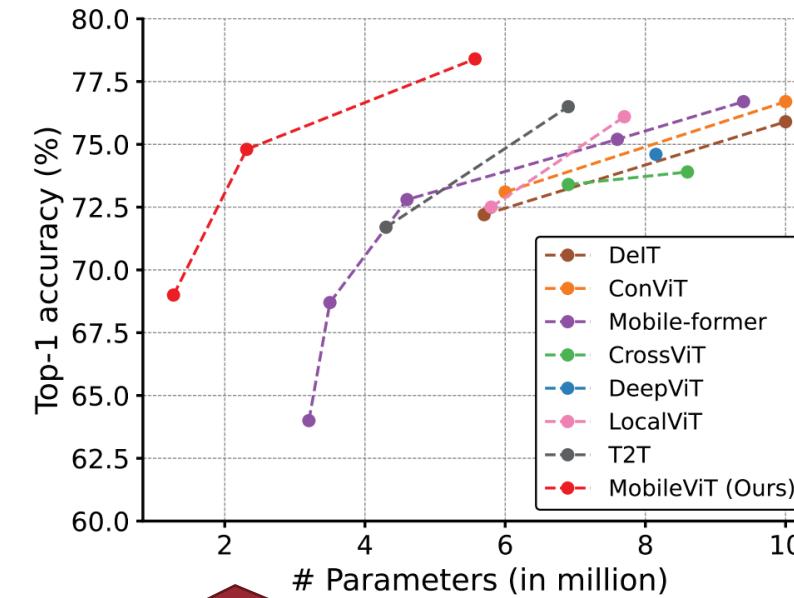
# MobileViT

[Mehta+Rastegari, ICLR'22]

- ImageNetの1000クラス画像データだけで学習



他のMobile系ネットワーク  
より高精度



他のViT系ネットワーク  
より少パラメータ&高精度

# Vision Transformerで物体検出やセグメンテーションを行うには



[Dosovitskiy+, ICLR'21]

- ViTのような粗いパッチだけだと...細かいバウンディングボックスの位置決めやセグメンテーションの精度が落ちる
- 一方で細かいパッチを多く作ってしまうと、アテンションの計算で時間がかかる（パッチ数の $2$ 乗オーダー）

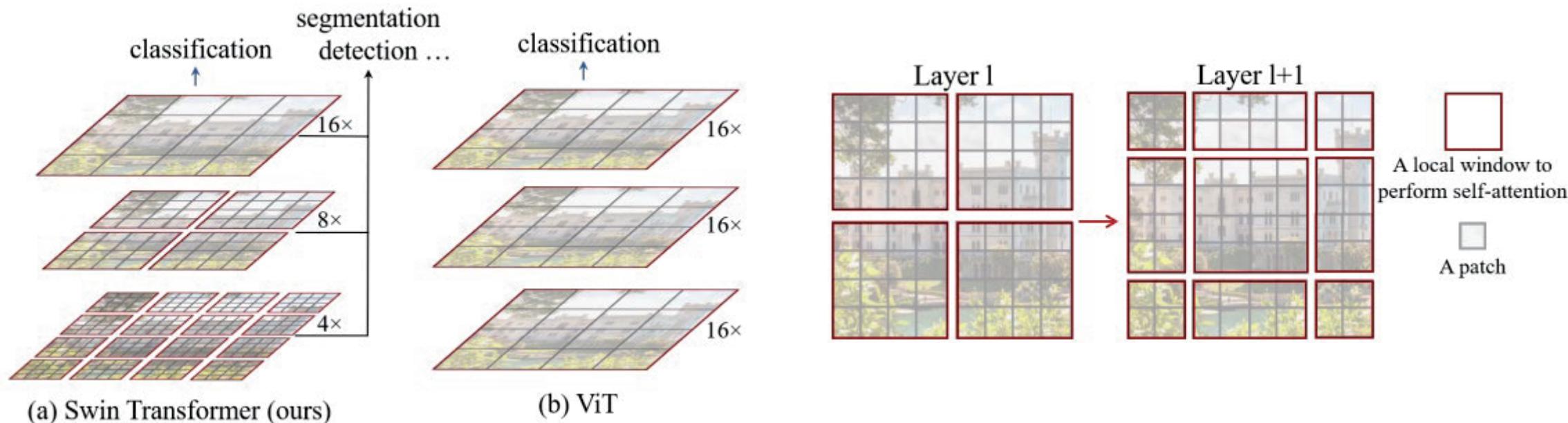
# Swin Transformer

Best paper!

[Liu+, ICCV'21]

- **CNNでおなじみのピラミッド構造を持ち込む**

- アテンション計算を各ローカルウィンドウに限定して計算量削減
- これらのレイヤーと少し区切りをずらしたレイヤーを交互に挟む  
→ローカルウィンドウを超えた受容野を達成



- 物体検出で評価（セグメンテーションはSwin-Unet [Cao+, 2021]）

# Vision Transformer と CNN の構造比較

[Raghu+, NeurIPS'21]

- **ViTとCNNの内部表現構造**

- ViTはより均一な表現を持ち、下位層と上位層の類似性が高い
  - 両モデルに顕著な差異がある

- **局所的・大域的な情報の利用方法**

- ViTはResNetよりも大域的な情報を低レイヤーに取り込んでいる
  - ただし下位層に局所的な情報を取り込むことは依然として重要で、大規模な事前学習データが必要

- **ViTのネットワーク構造**

- ViTのスキップ接続はResNetsよりもさらに影響力が強く、性能と表現の類似性に強い影響を与える

- **ViTの位置情報**

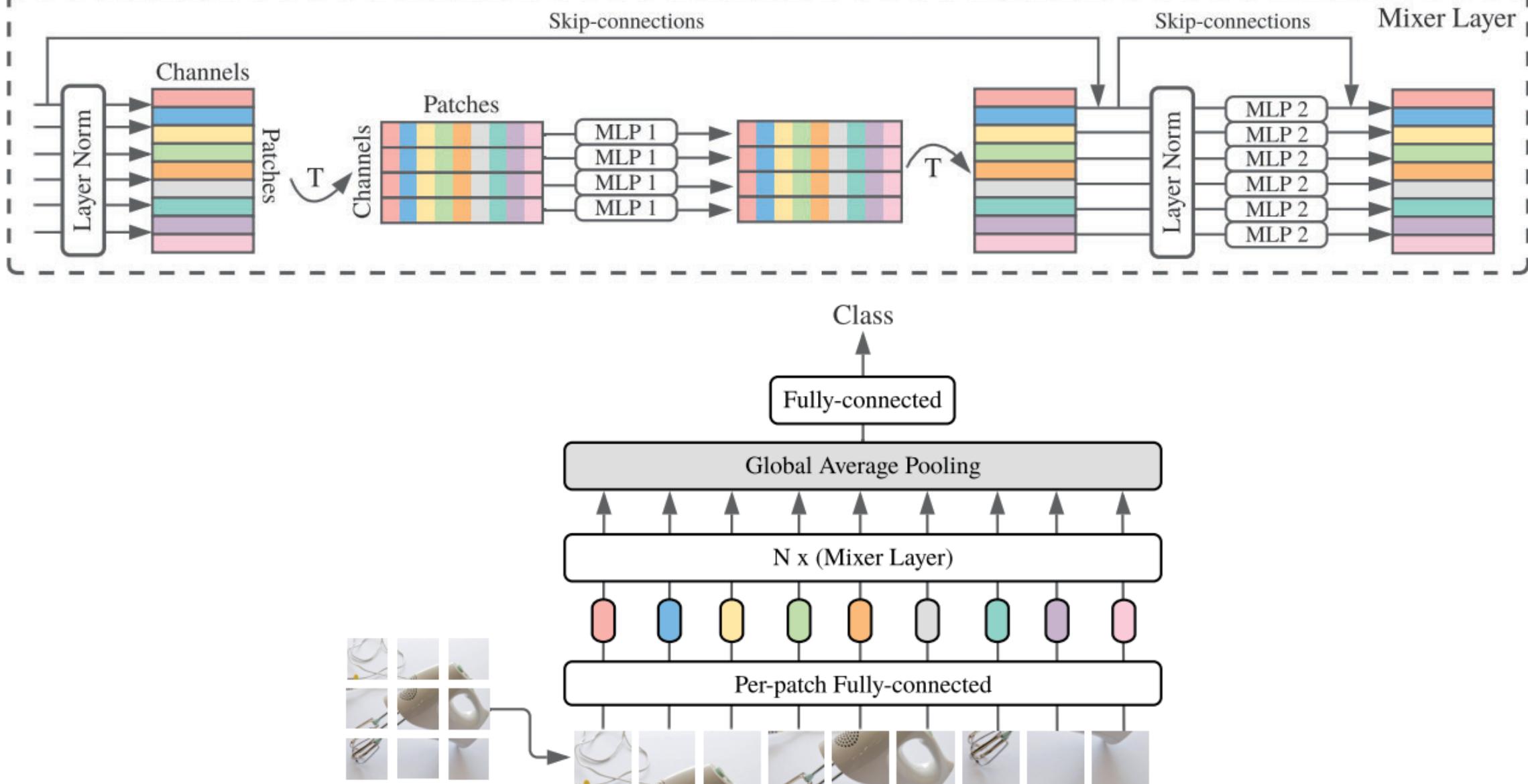
- ViTの方が画像内での位置情報が保持されている



# Transformerはオワコン？！

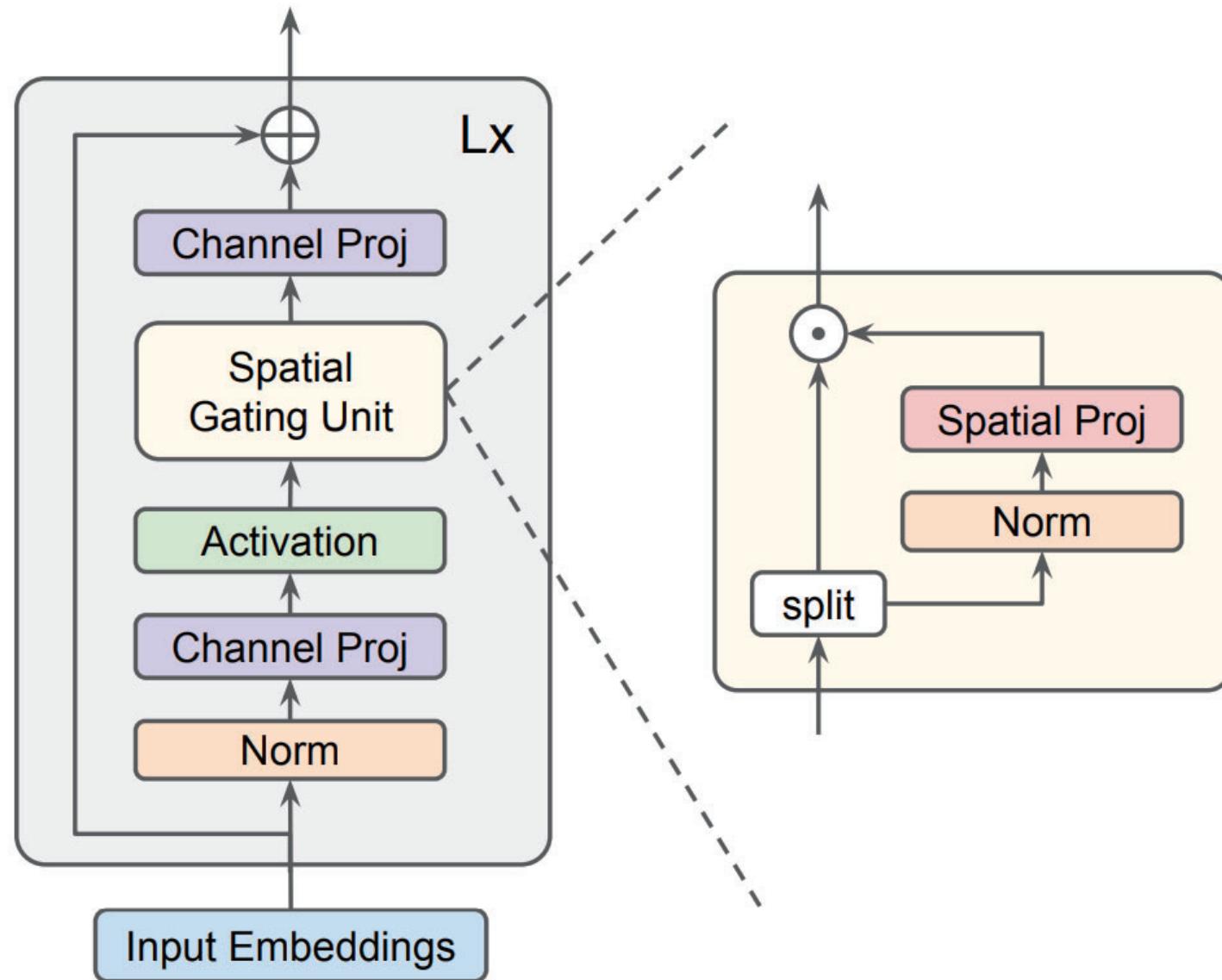
# MLP-Mixer

[Tolstikhin+, NeurIPS'21]



# gMLP

[Liu+, NeurIPS'21]



# gMLPによる画像分類精度比較では

Model	ImageNet Top-1 (%)*	Input Resolution	Params (M)	MAdds (B)
ConvNets				
ResNet-152 [16]	78.3	224	60	11.3
RegNetY-8GF [39]	81.7	224	39	8.0
EfficientNet-B0 [17]	77.1	224	5	0.39
EfficientNet-B3 [17]	81.6	300	12	1.8
EfficientNet-B7 [17]	84.3	600	66	37.0
NFNet-F0 [33]	83.6	192	72	12.4
Transformers				
ViT-B/16 [7]	77.9	384	86	55.4
ViT-L/16 [7]	76.5	384	307	190.7
DeiT-Ti [8] (ViT+reg)	72.2	224	5	1.3
DeiT-S [8] (ViT+reg)	79.8	224	22	4.6
DeiT-B [8] (ViT+reg)	81.8	224	86	17.5
MLP-like <sup>†</sup>				
Mixer-B/16 [20]	76.4	224	59	12.7
Mixer-B/16 (our setup)	77.3	224	59	12.7
Mixer-L/16 [20]	71.8	224	207	44.8
ResMLP-12 [22]	76.6	224	15	3.0
ResMLP-24 [22]	79.4	224	30	6.0
ResMLP-36 [22]	79.7	224	45	8.9
gMLP-Ti (ours)	72.3	224	6	1.4
gMLP-S (ours)	79.6	224	20	4.5
gMLP-B (ours)	81.6	224	73	15.8

ほとんどすべての  
CNN画像分類  
よりも

Transformerの  
画像分類  
よりも

(他のMLP系よりも)  
gMLPの方が  
精度が良い/  
パラメータが少ない

で出てくる感想が

Transformerってオワコンでは！？

# 結論から言うと

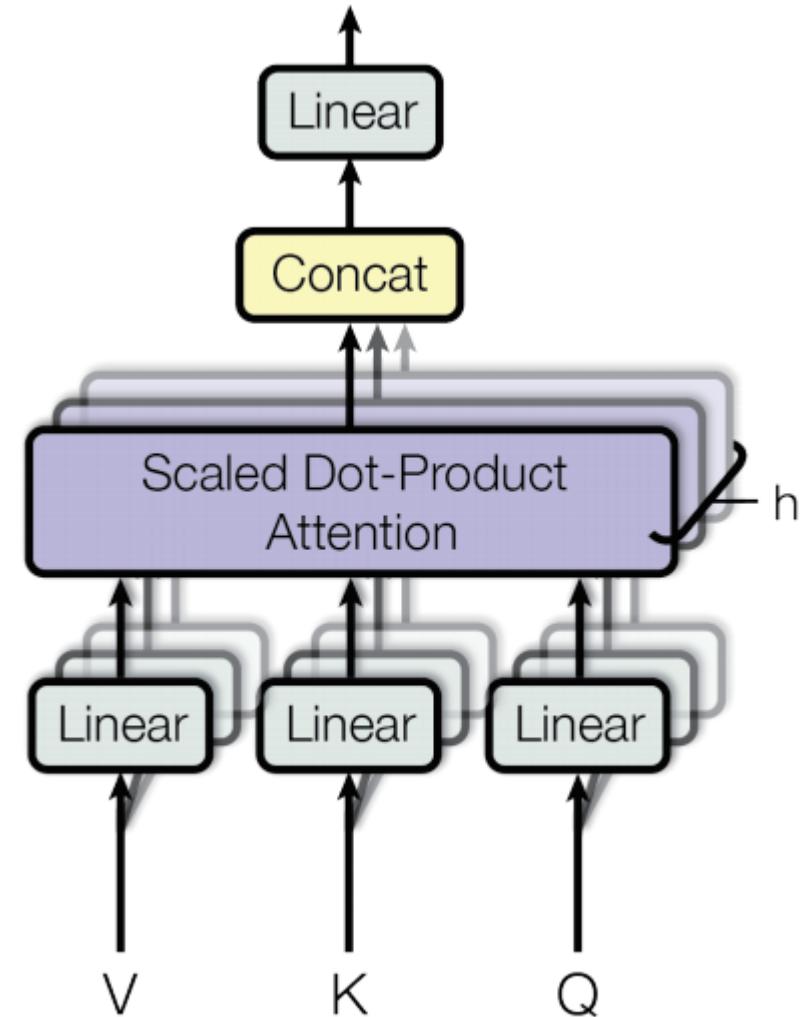
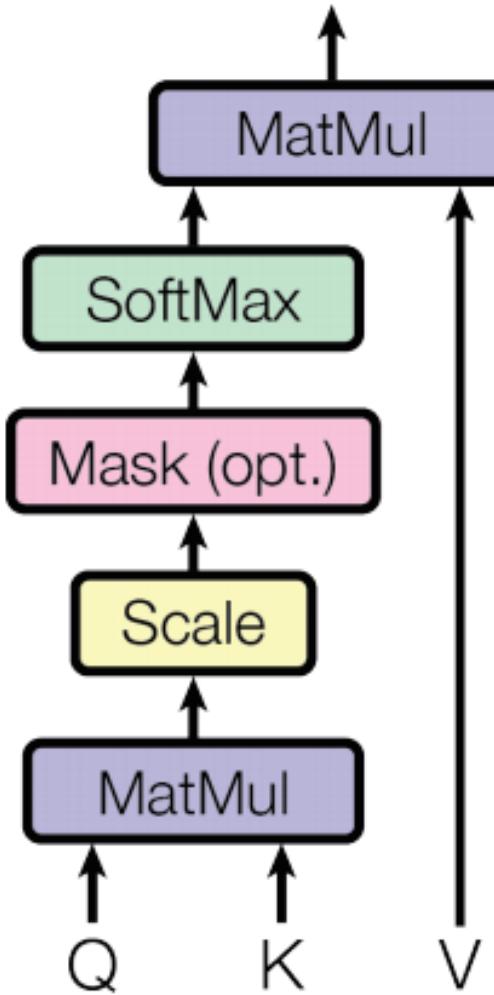
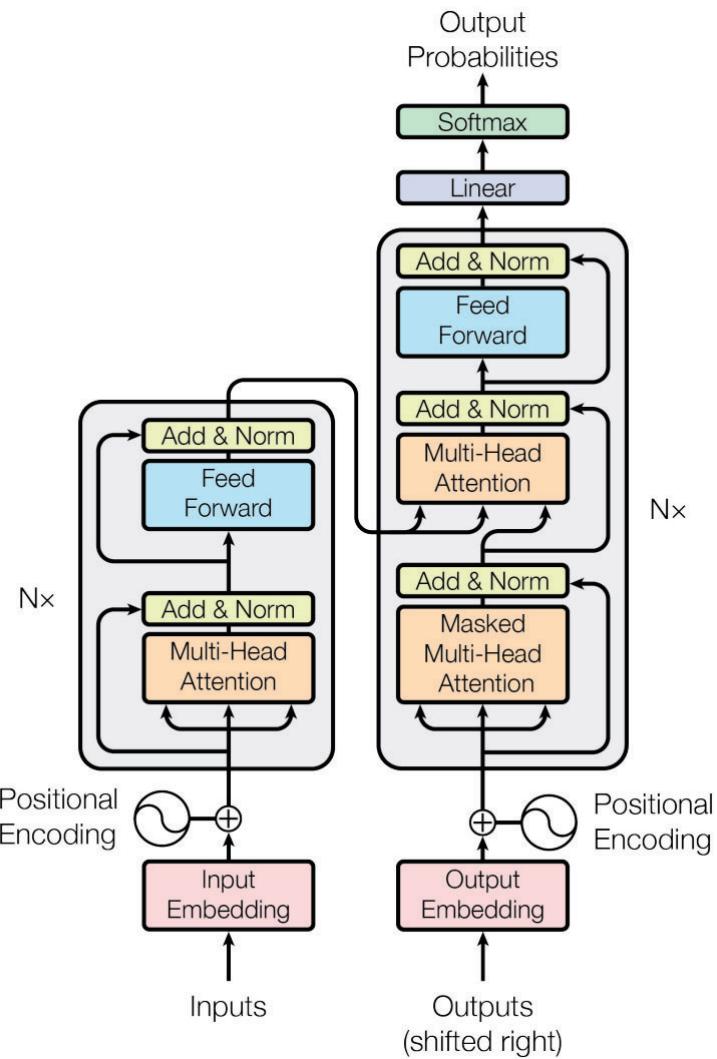
TransformerとMLP-MixerとgMLPは  
本質的にやっていることは余り違いません

単にAttention is All You Needではなかっただけです



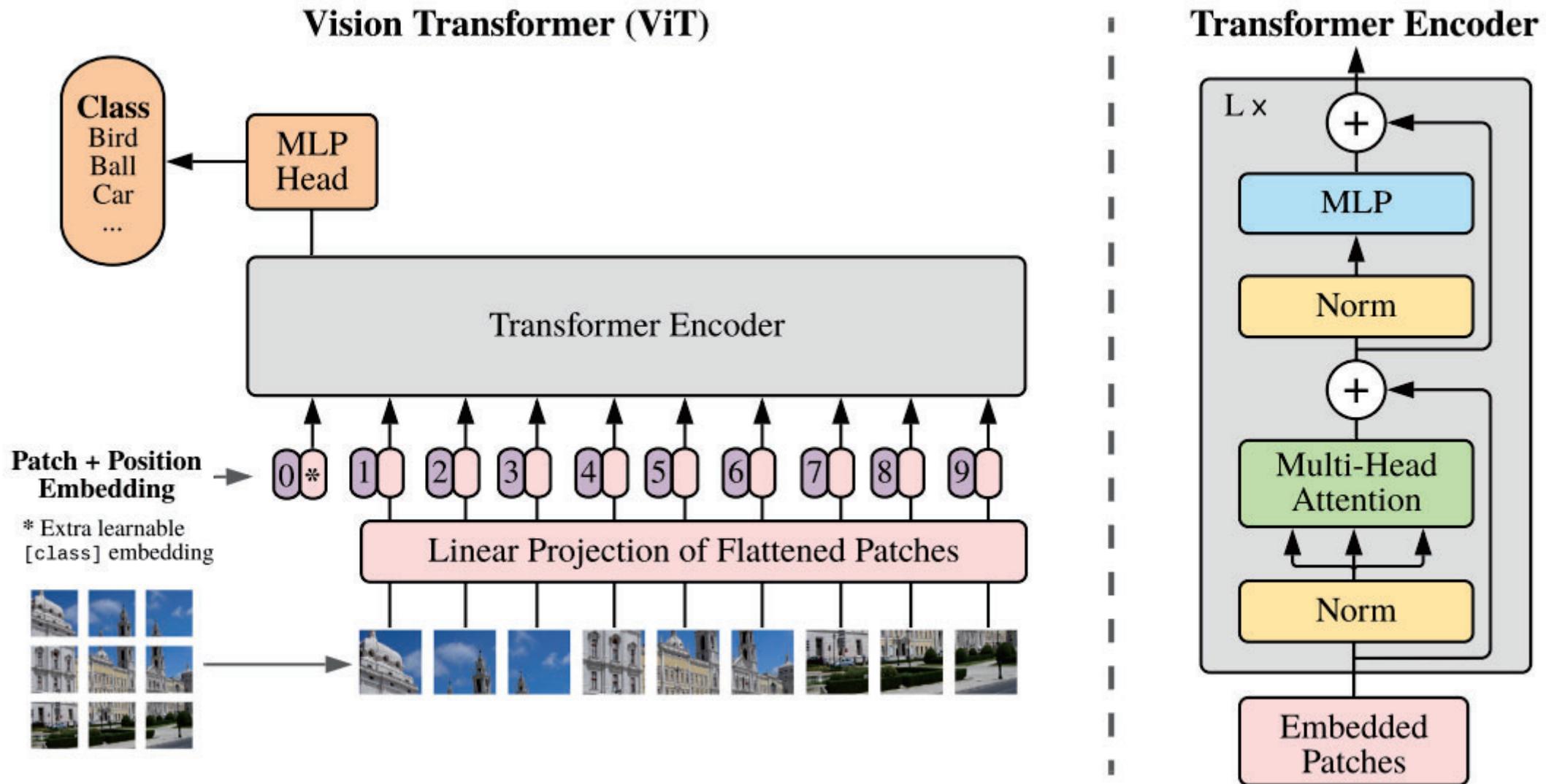
# Transformer

[Vaswani+, NIPS'17]

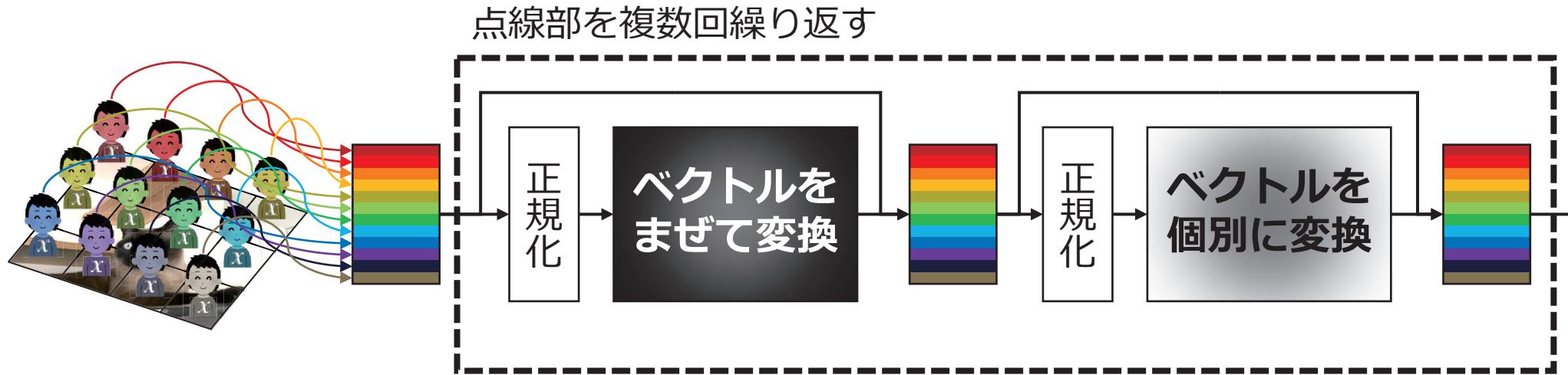


# Vision Transformer

[Dosovitskiy+, ICLR'21]

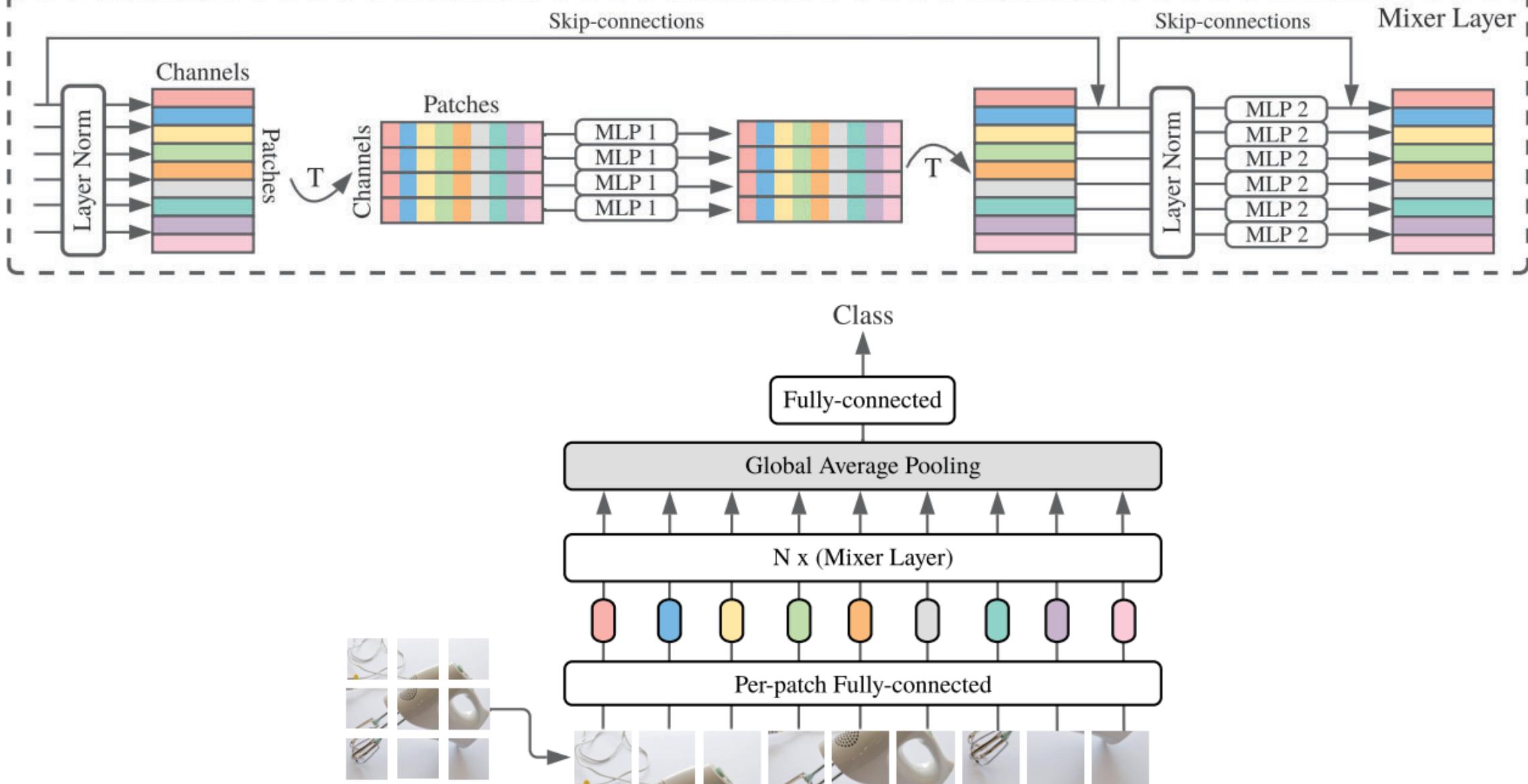


# 要するに

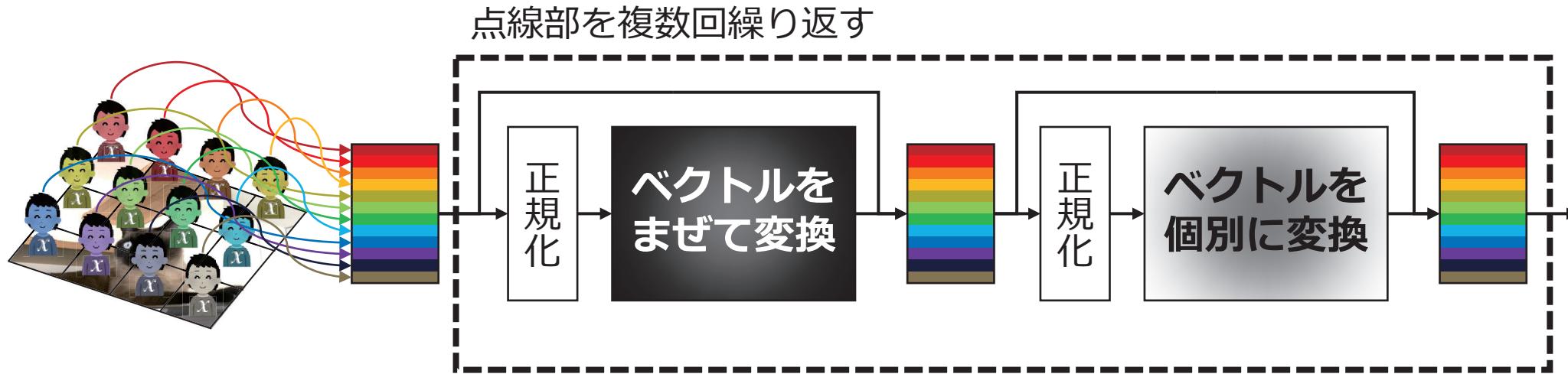


# MLP-Mixer

[Tolstikhin+, NeurIPS'21]

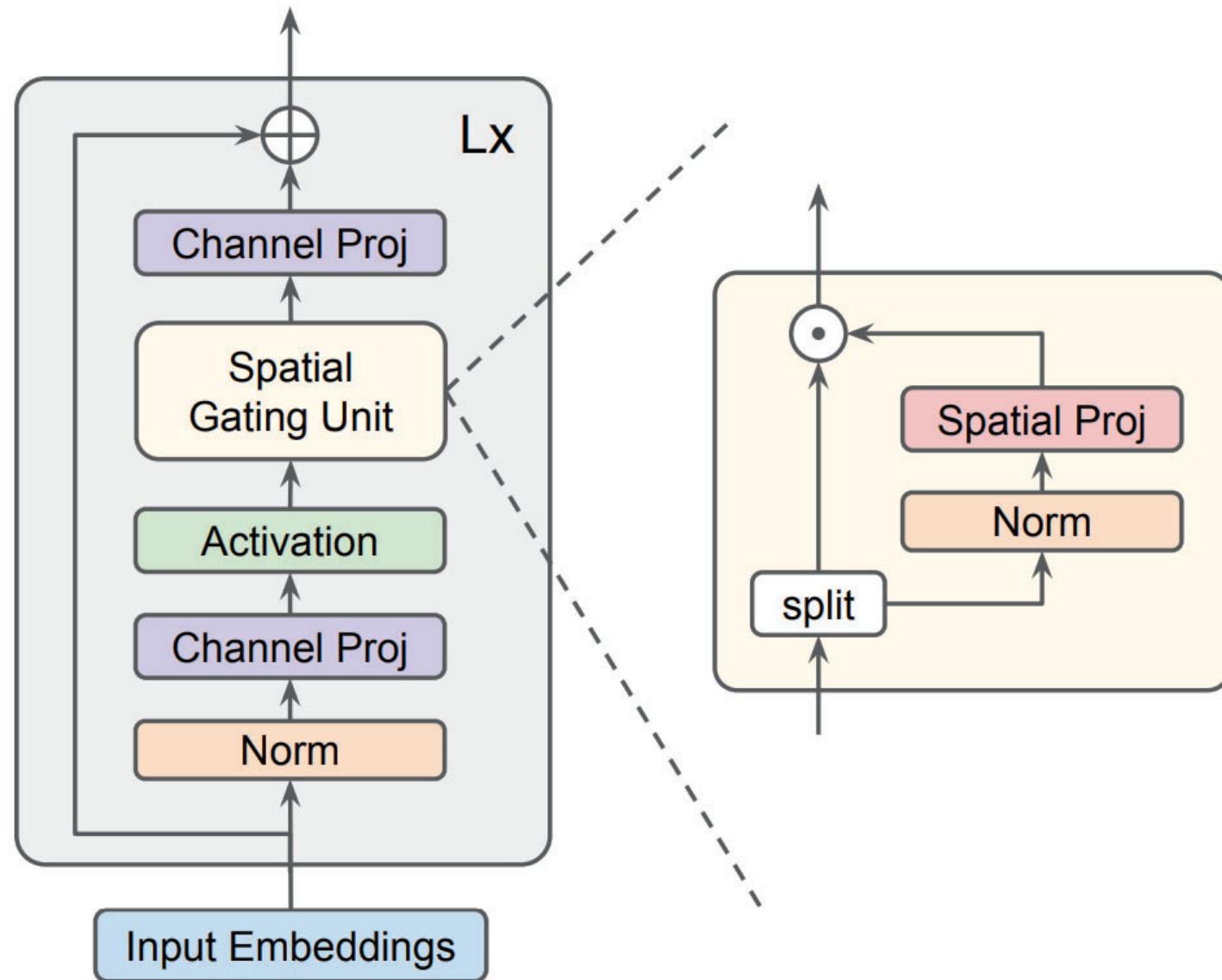


# 要するに

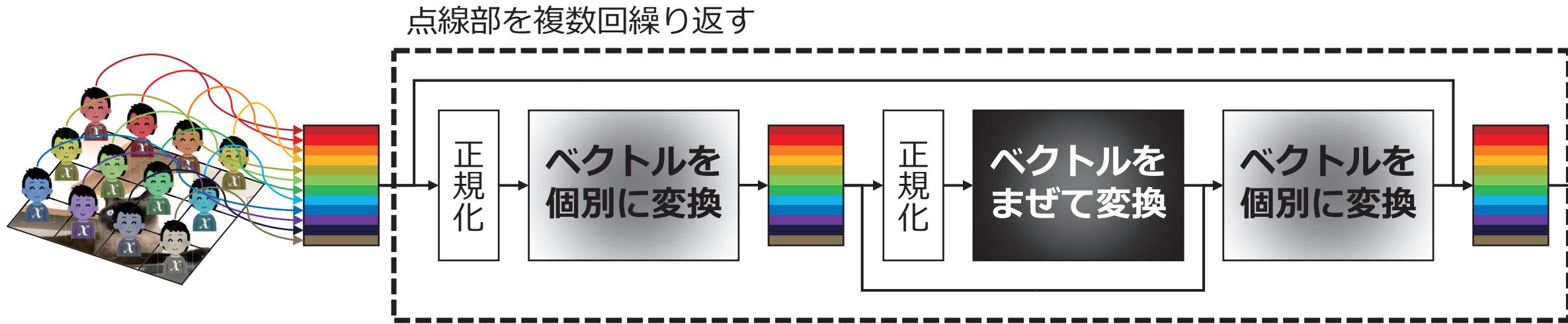


# gMLP

[Liu+, NeurIPS'21]



# 要するに



# 同じところ vs. 変わったところ

- **みんな同じところ**
  - ベクトルの集合を**変換**するモジュールを繰り返し適用する
    - **変換**はベクトルをまとめて変換するか、ベクトルを個別に変換するかの2通り
  - ベクトルを個別に変換する方法は**MLP**
  - 誤差消失や爆発を防ぐための正規化 (**Layer Normalization**)
  - 同じ目的で導入されている**Skip Connection**
- **Transformerから変わったところ**
  - ベクトルをまとめて変換する方法が**Attention**から行列積になった
  - ベクトルの**位置情報**をベクトル自身が保持 (Transformer)
    - ベクトルのインデックスに併せてネットワークが保持 (MLP系)

# 同じところ vs. 変わったところ

- **みんな同じところ**

- ベ

- 番

- gMLPの論文内の報告

- 「ネットワーク内はほとんどMLPだけど、

- ちょっとだけAttention入れた手法 (aMLP) は  
さらに高性能**

- ベ

- 読

- 同

- **Transformerから変わったところ**

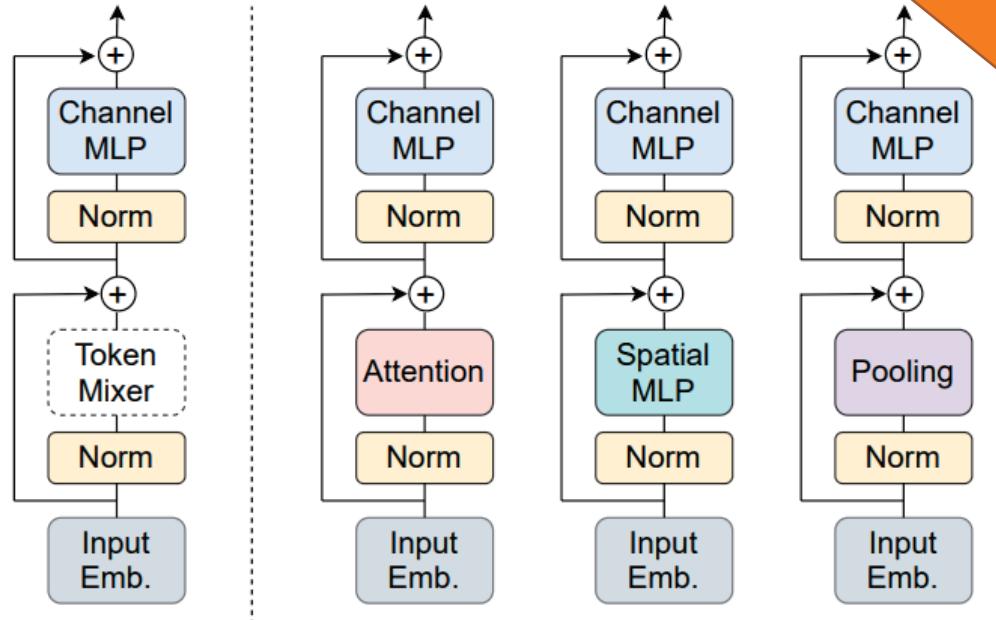
- ベクトルをまとめて変換する方法が**Attention**から**行列積**になった

- ベクトルの**位置情報**をベクトル自身が保持 (Transformer)

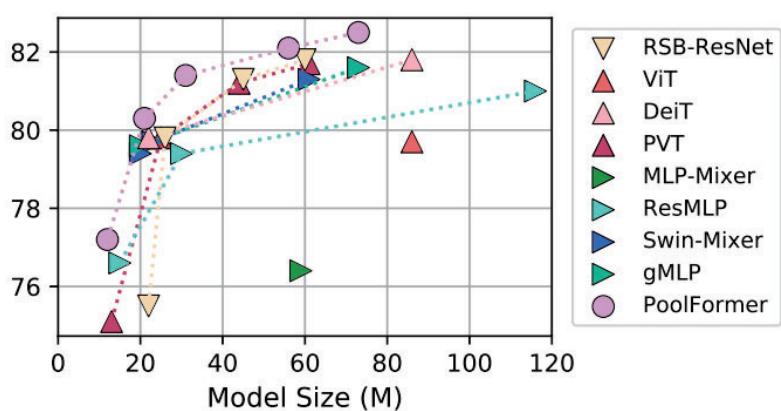
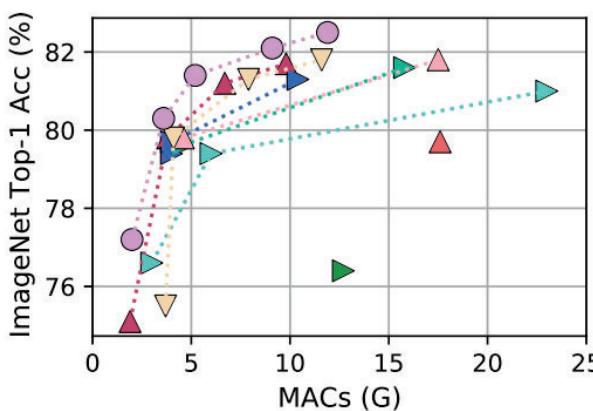
- ベクトルのインデックスに併せてネットワークが保持 (MLP系)

# なんてことを2021年頃から言っていたら...

## MetaFormer [Yu+, CVPR'22]

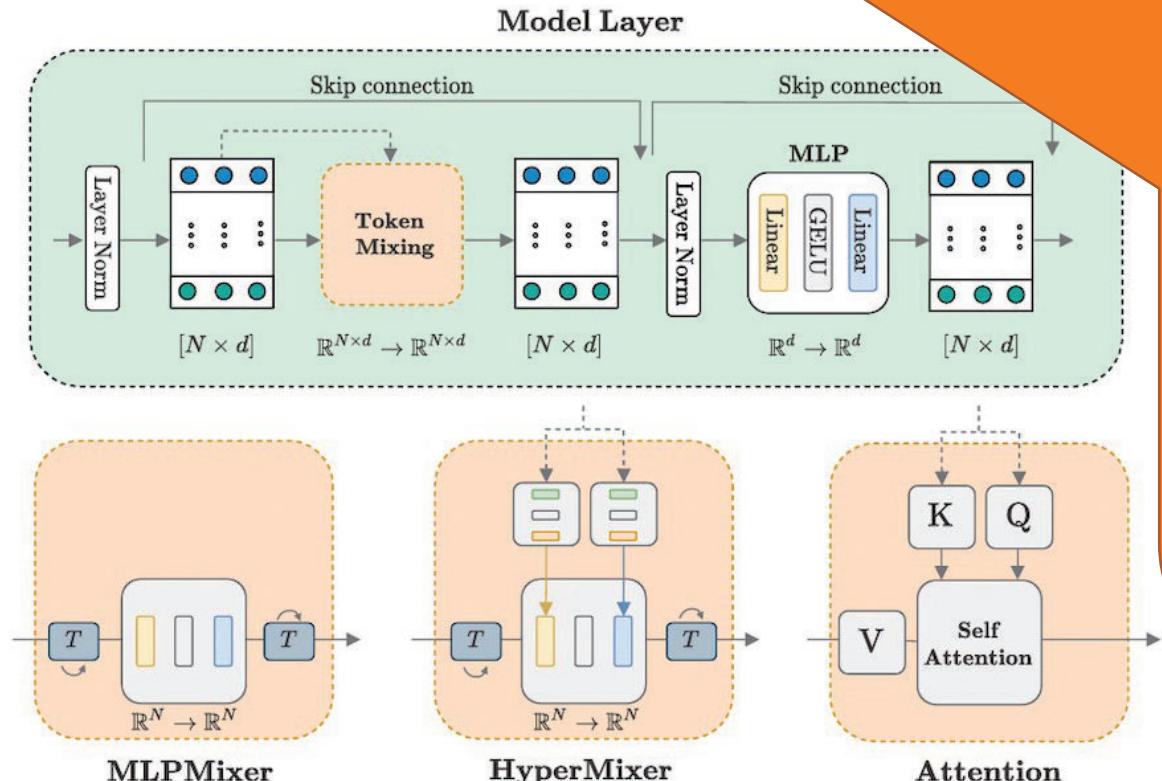


- MLP-like modelとTransformerの違いはToken Mixerの部分だけ  
→まさに「ベクトルをまぜて変換」
- 混ぜるのPoolingでも良くない？
- ImageNetの1000クラスデータの学習でViT系やMLP系の手法よりも高精度だよ



# なんてことを2021年頃から言っていたら...

## HyperMixer [Mai+, 2022]



- MLP-MixerとTransformerの違いはToken Mixingの部分だけ  
→まさに「ベクトルをまぜて変換」
- MLP-Mixerと違って位置不变なToken Mixingを行うHyperMixerを作ったよ
- 自然言語処理の各タスクで良好な精度を実現したよ

Model	MNLI	SNLI	QQP	QNLI	SST	# Params
<i>Baselines</i>						
	Validation set results (average accuracy / standard deviation over 10 seeds)					
MLPMixer	63.9 (0.34)	79.6 (0.11)	83.7 (0.42)	68.1 (2.1)	80.1 (0.67)	11 M
gMLP	60.8 (0.95)	80.5 (0.55)	82.8 (0.21)	60.5 (0.49)	78.7 (0.74)	11 M
FNet	59.7 (0.27)	75.3 (0.46)	79.4 (0.28)	59.9 (0.46)	79.7 (0.71)	11 M
Transformer	<b>65.4 (0.51)</b>	80.9 (0.4)	82.8 (0.22)	67.3 (2.03)	79.0 (0.86)	11 M
HyperMixer (ours)	63.9 (1.16)	<b>81.6 (0.43)</b>	<b>85.3 (0.19)</b>	<b>77.2 (1.07)</b>	80.5 (1.14)	11 M

# もう一度結論を言うと

TransformerとMLP-MixerとgMLPは  
本質的にやっていることは余り違いません

単にAttention is All You Needではなかっただけです





# Transformerのノウハウ

# Transformerの基本性能向上を心がけた軌跡

- ELU [Clevert+, ICLR 2016]
- GeLU [Hendrycks+Gimpel, 2016]
- Swish [Ramachandran+, ICLR WS 2018]
- SELU [Klambauer+, NIPS 2017]
- GLU [Dauphin+, ICML 2017]
- RMS [Zhang+Sennrich, NeurIPS 2019]
- ReZero [Bachlechner+, 2020]
- Fixup [Zhang+, ICLR 2019]
- Adaptive Softmax [Joulin+, ICML 2017]
- Mixture of Softmaxes [Yang+, ICLR 2018]

...が無駄に続けられている近年？！

- **Transparent Attention [Bapna+, EMNLP 2018]**
- **Evolved Transformer [So+, ICML 2019]**
- **Synthesizer variants [Tay+, 2020]**
- **Funnel Transformer [Dai+, NeurIPS 2020]**
- **Lightweight and Dynamic convolution [Wu+, ICLR 2019]**
- **Mixture of Experts Transformer [Shazeer+, NeurIPS 2018][Lepikhin+, ICLR 2021]**
- **Switch Transformer [Fedus+, 2021]**
- **Product Key Memory [Lample+, NeurIPS 2019]**
- **Universal Transformer [Dehghani+, ICLR 2019]**

# ツッコミ元について

- Google Researchの中の人16人で
- 30を超えるTransformerの改善手法を
- 50を超えるバリエーションと
- 9の観点で評価した結果

**大半の手法が元のTransformerと大差なかったよと言う話**

Do Transformer Modifications Transfer Across  
Implementations and Applications?

---

Sharan Narang\* Hyung Won Chung Yi Tay William Fedus  
Thibault Fevry† Michael Matena † Karishma Malkan† Noah Fiedel  
Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li  
Nan Ding Jake Marcus Adam Roberts Colin Raffel

# 改善方法の分類

1. 活性化関数
2. 正規化
3. 深さ
4. 埋め込み
5. パラメータ共有
6. Softmaxの改良
7. 全体のアーキテクチャ

※ 必ずしもTransformerの改善を意図して発表された手法だけでは無いので注意

# 1. 活性化関数の改善の歴史

- **ELU [Clevert+, ICLR 2016]**
  - 負の部分だけ指数関数で-1に漸近し、基本的に滑らかな形をとる。
  - 被引用数3000超！
- **GeLU [Hendrycks+Gimpel, 2016]**
  - ReLUに似た形だけど導関数が滑らか。
  - BERTにもGPT-3にも使われているけどICLRからはリジェクトされている。頑張れ。
- **Swish [Ramachandran+, ICLR WS 2018]**
  - ReLUに似た形だけど導関数が滑らか。あれ言っていることがGeLUと変わらない。
- **SELU [Klambauer+, NIPS 2017]**
  - Self-Normalizing Neural Networksを提案する論文の一部。 ELUを正の部分でも負の部分でも定数倍したもの。
  - SELU自体の評価は行われていない。査読者もツッコんでいるのに何故通った。
- **GLU [Dauphin+, ICML 2017]**
  - LSTMのGate部分を持ってきた活性化関数。
  - 線形変換 + 活性化関数（シグモイド）を通した0~1の値をもつGateと、別途計算した線形変換の要素積。

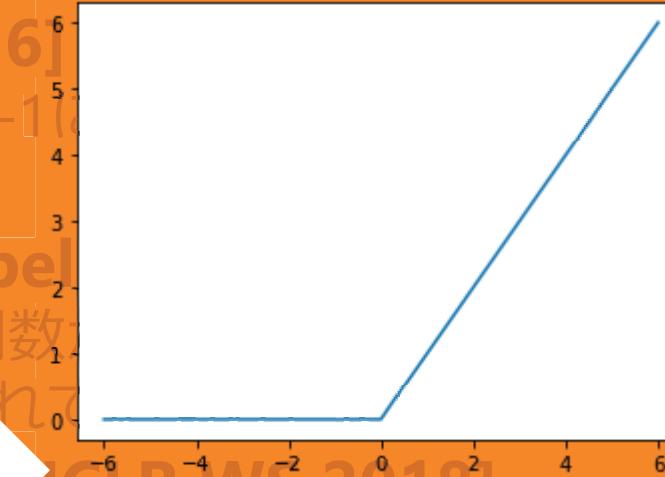
# 要するに...

## 1. 活性化関数の改善の歴史

ReLU

- **ReLU** [Clevert+, ICLR 2016]

– 負の部分だけ指数関数で-1000倍する  
– 被引用数3000超！



かな形をとる。

- **GeLU** [Hendrycks+Gimpel, ICLR 2019]

– ReLUに似た形だけど導関数が滑らか。  
– BERTにもGPT-3にも使われていて、  
– ジャストされている。頑張れ。



GeLUと変わらない。

- **Swish** [Ramachandran et al., ICLR WS 2018]

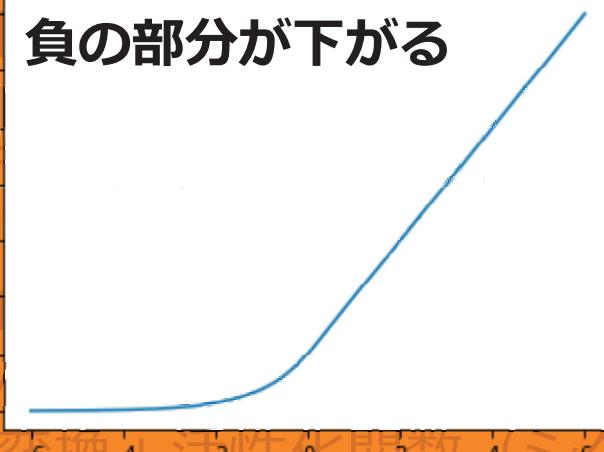
– ReLUに似た形だけど導関数が滑らか。あれ言っていることがGeLUと変わらない。



大体ReLUで滑らか

- **SELU** [Selu, 2017]

– Self-Normalizing FunctionでReLUを上回る  
– SELUはReLUの改良版



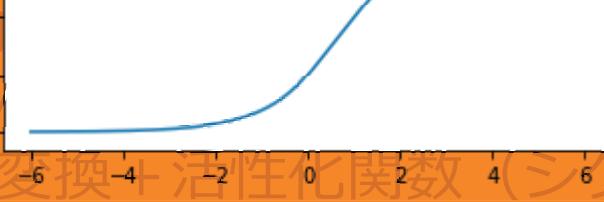
負の部分が下がる



大体ReLUで滑らか

- **GLU** [GLU, 2017]

– LSTMで使われる活性化関数。  
– 線形変換+活性化関数(シグモイド)を通した



ELU/SELU



大体ReLUで滑らか

GeLU/Swish

# そしてぶつちやけ...

## • ELU [Clevert+, ICLR 2016]

- 負の部分だけ指数関数で-1に漸近し、基本的に滑らかな形をとる。
- 被引用数3000超！

## • GeLU [Hendrycks+Gimpel, 2019]

- ReLUに似た形だけど導関数が滑らか
- BERTにもGPT-3にも使われている

はリジェクトされている。頑張れ。

## • Swish [Ramachandran+, ICLR 2017]

- ReLUに似た形だけど導関数が滑らか

っていることがGeLUと変わらない。

## • SELU [Klambauer+, NIPS 2017]

- Self-Normalizing Neural Networks論文の一部。ELUを正の部分でも負の部分でも定数倍したもの。

論文の一部。ELUを正の部分でも負の部分でも定数倍したもの。

- SELU自身の評価は行われていない

ソックンでいるのに何故通った。

## • GLU [Dauphin+, ICML 2017]

- LSTMのGate部分を持ってきた活性化関数。
- 線形変換 + 活性化関数（ReLU）を通りたがるの値をもつGateと、別途計算した線形変換の要素積。

精度大して上がりません



# ここで先に：実験概要

- **転移学習課題：Text-to-Text Transfer Transformer (T5)**
  - テキストを入れてテキストを出す複数の課題のための事前学習
  - 6.1TBのテキストをフィルタリングして約750GBにしたColossal Clean Crawled Corpus (C4)を利用
  - 本論文は次の3つの課題を採用
    - QAや推論などの複合タスク  
SuperGLUE [Wang+, NeurIPS 2019]
    - 文の要約  
XSum [Narayan+, EMNLP 2018]
    - 質問応答  
WebQuestions [Berant+, EMNLP 2013]
- **機械翻訳課題：WMT'14の英独翻訳タスク**



# ここで先に：実験概要

おことわり

本論文の実験はNLP課題です



# 活性化関数にまつわる実験結果

- パラメータ数と計算量が揃う様に調整
  - Final loss = 文法モデル性能、ここだけ lower is better
  - SGLUE = 複合課題
  - XSum = 要約課題
  - WebQ = 質問応答
  - WMT EnDe = 機械翻訳

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	70.97	17.78	23.02	26.62
GeLU	223M	11.1T	3.58	2.179 ± 0.003	1.838	<b>73.67</b>	<b>17.86</b>	<b>25.13</b>	26.47
Swish	223M	11.1T	3.62	2.186 ± 0.003	1.847	<b>72.03</b>	17.74	<b>24.34</b>	<b>26.75</b>
ELU	223M	11.1T	3.56	2.270 ± 0.007	1.932	65.86	16.73	23.02	26.08
GLU	223M	11.1T	3.59	2.174 ± 0.003	<b>1.814</b>	<b>67.86</b>	<b>17.42</b>	24.34	<b>27.12</b>
SeLU	223M	11.1T	3.55	2.315 ± 0.004	1.948	66.13	16.76	22.75	25.99

- 結果：共通して性能が向上した手法がない
  - 性能向上 = 太字と言っているけどしばしば間違っているので注意

# 精度が良くなつた活性化関数もある

- **GLUの発展形 [Shazeer, 2020]**

- GLUは線形変換 + 活性化関数によるゲートと線形変換の要素積
  - GLU自体は活性化関数をシグモイドとしていた

$$\text{GLU}(x, W, V, b, c) = \sigma(xW + b) \otimes (xV + c)$$

- **以下のバリエーションを試してみた**

- GeGLU : 活性化関数が**GeLU** (ReLUみたいな形の滑らかなやつ)
  - ReGLU : 活性化関数が**ReLU**
  - SwiGLU : 活性化関数が**Swish** (ReLUみたいな形の滑らかな奴)
    - パラメータ数5400億の超巨大言語モデルPaLMでも利用 [Chowdhery+, 2022]
  - LiGLU : 活性化関数なし (双線形形式)
  - あれ、ELUやSELUとも組合せてみないの... ?

# GLUのバリエーションの評価

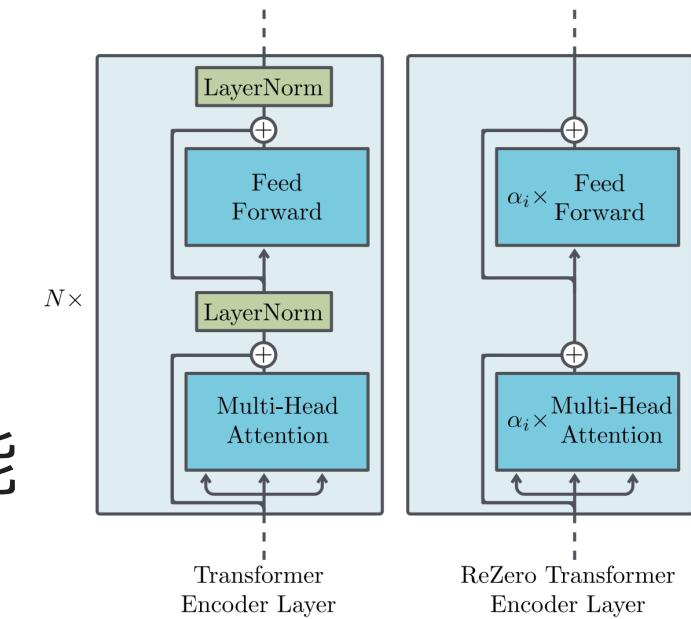
- 評価方法は前述通り
  - Final loss = 文法モデル性能、ここだけ lower is better
  - SGLUE = 複合課題
  - XSum = 要約課題
  - WebQ = 質問応答
  - WMT EnDe = 機械翻訳

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
GLU	223M	11.1T	3.59	$2.174 \pm 0.003$	<b>1.814</b>	<b>67.86</b>	<b>17.42</b>	24.34	<b>27.12</b>
GeGLU	223M	11.1T	3.55	$2.130 \pm 0.006$	<b>1.792</b>	<b>74.86</b>	<b>18.27</b>	<b>24.87</b>	<b>26.87</b>
ReGLU	223M	11.1T	3.57	$2.145 \pm 0.004$	<b>1.803</b>	<b>73.40</b>	<b>18.36</b>	<b>24.87</b>	<b>27.02</b>
SwiGLU	223M	11.1T	3.53	$2.127 \pm 0.003$	<b>1.789</b>	<b>74.21</b>	<b>18.20</b>	<b>24.34</b>	<b>27.02</b>
LiGLU	223M	11.1T	3.59	$2.149 \pm 0.005$	<b>1.798</b>	<b>73.97</b>	<b>17.97</b>	<b>24.34</b>	26.53

- GLUとLiGLUは少し下がる結果もあるが...  
他のバリエーションは一貫して効果アリ

## 2. 正規化の改善の歴史

- **Vanilla Transformer : LayerNorm [NIPS DLS 2016]**
  - ベクトルの要素ごとの平均と分散で正規化。
- **RMS [Zhang+Sennrich, NeurIPS 2019]**
  - LayerNormが遅いので平均抜きで正規化。
- **ReZero [Bachlechner+, 2020]**
  - タイトルの出だしが“ReZero is All You Need...”  
出たよ○○ is All You Need系論文。
  - 元のTransformer（左側）の変換部分に学習可能  
パラメータ $\alpha$ （初期値ゼロ）を掛ける（右側）。
- **Fixup [Zhang+, ICLR 2019]**
  - 正規化を一切せずに、Residualブロックの初期値を0とか1にするだけでもBatchNormやLayerNormと近い精度を達成。



# 正規化にまつわる実験結果

- **出場選手**

- RMS Norm 「LayerNormの計算早くするぜ」
- ReZero 「Residualブロックの変換部分を $\alpha$ （初期値0）倍するぜ」
- Fixup 「初期値ちゃんと考えると正規化不要だぜ」

効果のあった手法はどれでしょうか？

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
RMS Norm	223M	11.1T	3.68	$2.167 \pm 0.008$					
Rezero	223M	11.1T	3.51	$2.262 \pm 0.003$					
Rezero + LayerNorm	223M	11.1T	3.26	$2.223 \pm 0.006$					
Rezero + RMS Norm	223M	11.1T	3.34	$2.221 \pm 0.009$					
Fixup	223M	11.1T	2.95	$2.382 \pm 0.012$					

# 正規化にまつわる実験結果

- **出場選手**
  - RMS Norm 「LayerNormの計算早くするぜ」
  - ReZero 「Residualブロックの変換部分を $\alpha$ （初期値0）倍するぜ」
  - Fixup 「初期値ちゃんと考えると正規化不要だぜ」

## 結果発表

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
RMS Norm	223M	11.1T	3.68	$2.167 \pm 0.008$	<b>1.821</b>	<b>73.73</b>	<b>17.94</b>	<b>24.07</b>	<b>27.14</b>
Rezero	223M	11.1T	3.51	$2.262 \pm 0.003$	1.939	57.21	15.64	20.90	26.37
Rezero + LayerNorm	223M	11.1T	3.26	$2.223 \pm 0.006$	1.858	68.16	17.58	23.02	26.29
Rezero + RMS Norm	223M	11.1T	3.34	$2.221 \pm 0.009$	1.875	65.14	17.32	23.02	26.19
Fixup	223M	11.1T	2.95	$2.382 \pm 0.012$	2.067	55.71	14.42	23.02	26.31

- **RMS Normのみ効果アリ**
  - というかReZeroが大変な状況

### 3. 深さについての検証

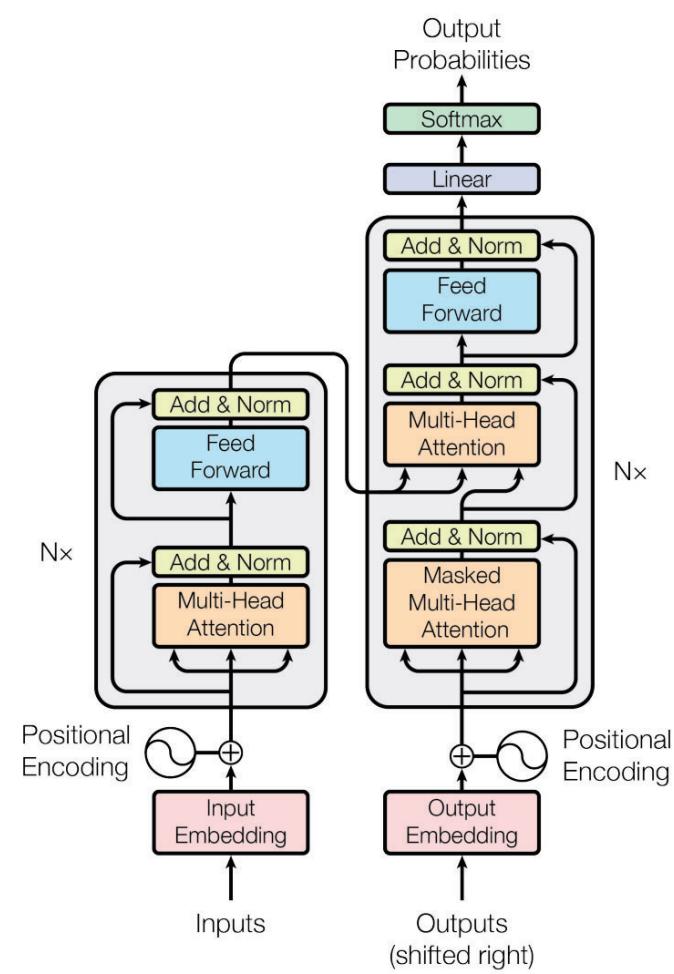
- 右図のFeed Forward部分

- 線形変換その1 → ReLU → 線形変換その2
- パラメータ数のトレードオフを調べたい
  - 全体の層数 (右図におけるN)
  - 真ん中の部分の次元数  $d_{ff}$
  - Multi-Head Attentionのヘッド数  $H$

- 調べた結果

- Vanillaは 12 layers,  $d_{ff} = 3072, H = 12$

– 層数が深い方が精度良さげだが、1秒当たりのステップ計算が遅い。



Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
24 layers, $d_{ff} = 1536, H = 6$	224M	11.1T	3.33	$2.200 \pm 0.007$	1.843	<b>72.55</b>	17.75	<b>25.13</b>	<b>26.89</b>
18 layers, $d_{ff} = 2048, H = 8$	223M	11.1T	3.38	$2.185 \pm 0.005$	<b>1.831</b>	<b>74.74</b>	16.83	<b>24.34</b>	<b>27.10</b>
8 layers, $d_{ff} = 4608, H = 18$	223M	11.1T	3.69	$2.190 \pm 0.005$	1.847	<b>72.17</b>	17.69	<b>23.28</b>	<b>26.85</b>
6 layers, $d_{ff} = 6144, H = 24$	223M	11.1T	3.70	$2.201 \pm 0.010$	1.857	70.28	17.59	<b>24.60</b>	<b>26.66</b>

## 4. 埋め込み方法についての検証

- **InputとOutputは語彙×埋め込み次元のパラメータ**
  - NLPだとパラメータ数に影響が大きい
- **行列分解 (ALBERT [Lan+, ICLR 2020] より)**
  - 語彙×埋め込み次元 → 語彙×内部次元と内部次元×埋め込み次元
- **エンコーダの入出力での埋め込み [Chung+, ICLR 2021]**
  - 共有 (Tied) か非共有 (Untied) か
- **頻度による埋め込み次元の調整 [Baevski+Auli, ICLR 2019]**
  - 低頻度な単語は低次元のベクトルに埋め込む

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	70.97	17.78	23.02	26.62
Factorized Embedding	227M	9.4T	3.80	2.208 ± 0.006	1.855	69.37	15.92	22.75	26.50
Factorized & shared embeddings	202M	9.1T	3.92	2.320 ± 0.010	1.952	68.27	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248M	11.1T	3.55	2.192 ± 0.002	1.840	69.36	17.72	<b>24.34</b>	26.49
Tied decoder input and output embeddings	248M	11.1T	3.57	2.187 ± 0.007	<b>1.827</b>	<b>73.28</b>	17.74	<b>24.87</b>	<b>26.67</b>
Untied embeddings	273M	11.1T	3.53	2.195 ± 0.005	<b>1.834</b>	70.94	17.58	<b>23.28</b>	26.48
Adaptive input embeddings	204M	9.2T	3.55	2.250 ± 0.002	1.899	65.72	16.21	<b>24.07</b>	<b>26.66</b>

- 実験結果：デコーダの入出力を共有（エンコーダとは非共有）すると○

## 5. パラメータ共有方法についての検証

- **ALBERT [Lan+, ICLR 2020]** より
  - 各層のパラメータを全部共有する
  - 先ほどの埋め込みの分解と共有も試す
  - エンコーダだけ/デコーダだけで共有

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
Block sharing	65M	11.1T	3.91	$2.497 \pm 0.037$	2.164	62.05	14.53	21.96	25.48
+ Factorized embeddings	45M	9.4T	4.21	$2.631 \pm 0.305$	2.183	58.85	14.00	19.84	25.27
+ Factorized & shared embeddings	20M	9.1T	4.37	$2.907 \pm 0.313$	2.385	52.34	11.37	19.84	25.19
Encoder only block sharing	170M	11.1T	3.68	$2.298 \pm 0.023$	1.929	66.21	16.23	23.02	26.23
Decoder only block sharing	144M	11.1T	3.70	$2.352 \pm 0.029$	2.082	65.58	16.13	<b>23.81</b>	26.08

## • 実験結果：大体ダメ

- ただしALBERTでは文の順番を当てる損失も入れているが、この論文では対象としていないのでALBERT自身との比較ではない

## 6. Softmax

- **Adaptive Softmax [Joulin+, ICML 2017]**
  - 単語の頻度に応じて語彙をクラスタリング→階層的識別で高速化
  - 低頻度語はさらに射影して軽量化 + 高速化
- **Mixture of Softmaxes [Yang+, ICLR 2018]**
  - Softmaxを $K$ 通り計算して重みづけ和による事後確率計算

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	$2.182 \pm 0.005$	1.838	70.97	17.78	23.02	26.62
Adaptive softmax	204M	9.2T	3.60	$2.364 \pm 0.005$	1.982	<b>71.28</b>	16.67	21.16	25.56
Adaptive softmax without projection	223M	10.8T	3.43	$2.229 \pm 0.009$	1.914	69.52	17.10	23.02	25.72
Mixture of softmaxes	232M	16.3T	2.24	$2.227 \pm 0.017$	<b>1.821</b>	<b>74.57</b>	17.62	22.75	<b>26.82</b>

- 実験結果：
  - Mixture of Softmaxesは、性能が良くなったタスクもあるけど計算速度が40%低下

## 7. 全体のアーキテクチャの改善の歴史

- **Transparent Attention [Bapna+, EMNLP 2018]**
- **Evolved Transformer [So+, ICML 2019]**
- **Synthesizer variants [Tay+, 2020]**
- **Funnel Transformer [Dai+, NeurIPS 2020]**
- **Lightweight and Dynamic convolution [Wu+, ICLR 2019]**
- **Mixture of Experts Transformer [Shazeer+, NeurIPS 2018][Lepikhin+, ICLR 2021]**
- **Switch Transformer [Fedus+, 2021]**
- **Product Key Memory [Lample+, NeurIPS 2019]**
- **Universal Transformer [Dehghani+, ICLR 2019]**

# 実験結果

いよいよ多くて訳が分からん

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	70.97	17.78	23.02	26.62
Transparent attention	223M	11.1T	3.33	2.181 ± 0.014	1.874	51.31	10.40	21.16	<b>26.80</b>
Dynamic convolution	257M	11.8T	2.65	2.403 ± 0.009	2.047	53.16	12.67	21.16	17.03
Lightweight convolution	224M	10.4T	4.07	2.370 ± 0.010	1.989	60.27	14.86	23.02	24.73
Evolved Transformer	217M	9.9T	3.09	2.220 ± 0.003	1.863	<b>71.80</b>	10.76	<b>24.07</b>	26.58
Synthesizer (dense)	224M	11.4T	3.47	2.334 ± 0.021	1.962	58.46	14.27	16.14	<b>26.63</b>
Synthesizer (dense plus)	243M	12.6T	3.22	2.191 ± 0.010	1.840	<b>71.11</b>	16.96	<b>23.81</b>	<b>26.71</b>
Synthesizer (dense plus alpha)	243M	12.6T	3.01	2.180 ± 0.007	<b>1.828</b>	<b>72.12</b>	17.02	<b>23.28</b>	26.61
Synthesizer (factorized)	207M	10.1T	3.94	2.341 ± 0.017	1.968	59.75	15.39	<b>23.55</b>	26.42
Synthesizer (random)	254M	10.1T	4.08	2.326 ± 0.012	2.009	51.17	10.35	19.56	26.44
Synthesizer (random plus)	292M	12.0T	3.63	2.189 ± 0.004	1.842	<b>71.81</b>	17.04	<b>24.87</b>	26.43
Synthesizer (random plus alpha)	292M	12.0T	3.42	2.186 ± 0.007	<b>1.828</b>	<b>73.13</b>	17.08	<b>24.08</b>	26.39
Universal Transformer	84M	40.0T	0.88	2.406 ± 0.036	2.053	66.77	14.09	19.05	23.91
Mixture of experts	648M	11.7T	3.20	2.148 ± 0.006	<b>1.785</b>	<b>72.06</b>	<b>18.13</b>	<b>24.08</b>	<b>26.94</b>
Switch Transformer	1100M	11.7T	3.18	2.135 ± 0.007	<b>1.758</b>	<b>73.51</b>	<b>18.02</b>	<b>26.19</b>	<b>26.81</b>
Funnel Transformer	223M	1.9T	4.30	2.288 ± 0.008	1.918	64.58	16.26	22.75	23.20
Weighted Transformer	280M	71.0T	0.59	2.378 ± 0.021	1.989	66.20	16.98	23.02	26.30
Product key memory	421M	386.6T	0.25	2.155 ± 0.003	<b>1.798</b>	<b>73.18</b>	17.04	<b>23.55</b>	<b>26.73</b>

## 7. 全体のアーキテクチャの改善の歴史

- **Transparent Attention [Bapna+, EMNLP 2018]**
- **Evolved Transformer [So+, ICML 2019]**
- **Synthesizer variants [Tay+, 2020]**
- **Funnel Transformer [Dai+, NeurIPS 2020]**
- **Lightweight and Dynamic convolution [Wu+, ICLR 2019]**
- **Mixture of Experts Transformer [Shazeer+, NeurIPS 2018][Lepikhin+, ICLR 2021]**
- **Switch Transformer [Fedus+, 2021]**
- **Product Key Memory [Lample+, NeurIPS 2019]**
- **Universal Transformer [Dehghani+, ICLR 2019]**

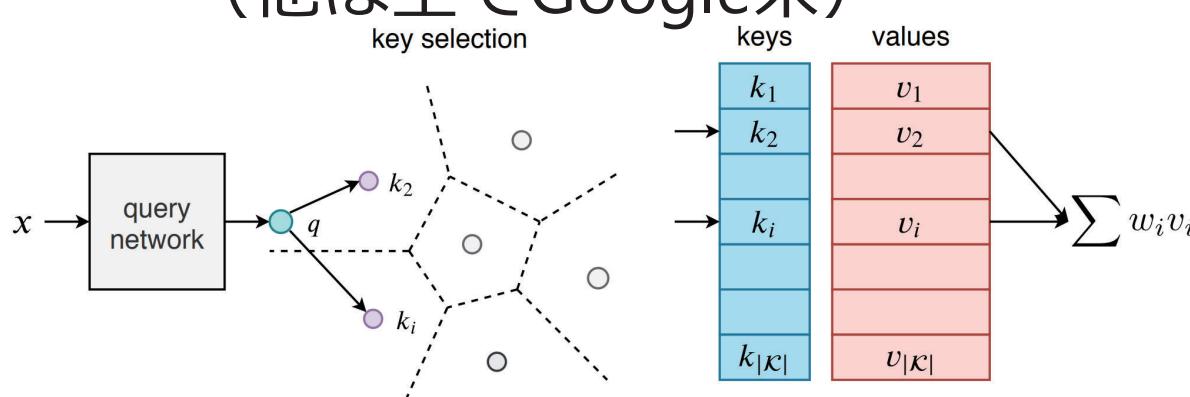
## 精度が良くなつたもの/悪くなつたもの

- **Transparent Attention [Bapna+, EMNLP 2018]**
- **Evolved Transformer [So+, ICML 2019]**
- **Synthesizer variants [Tay+, 2020]**
- **Funnel Transformer [Dai+, NeurIPS 2020]**
- **Lightweight and Dynamic convolution [Wu+, ICLR 2019]**
- **Mixture of Experts Transformer [Shazeer+, NeurIPS 2018][Lepikhin+, ICLR 2021]**
- **Switch Transformer [Fedus+, 2021]**
- **Product Key Memory [Lample+, NeurIPS 2019]**
- **Universal Transformer [Dehghani+, ICLR 2019]**

# Product Key MemoryとSynthesizer variants

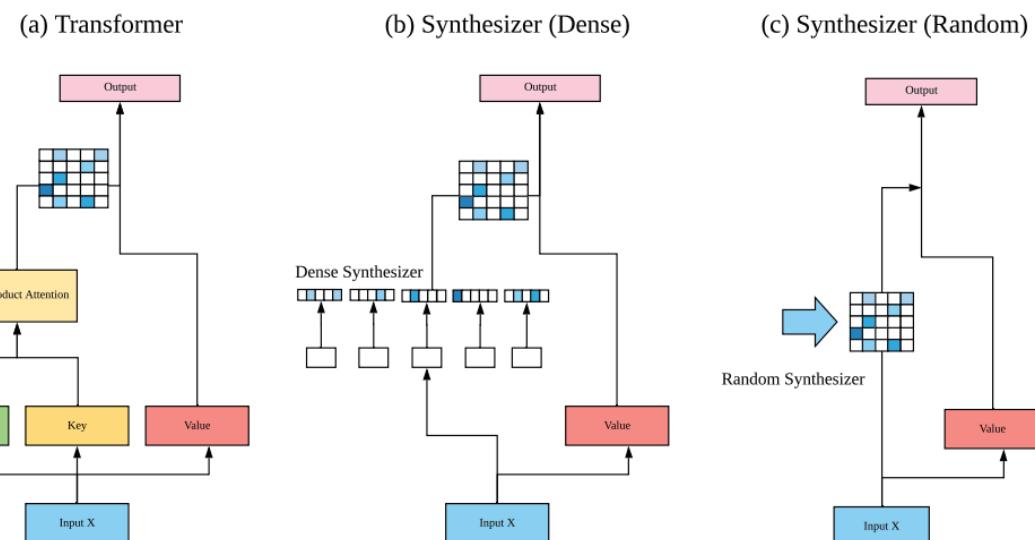
- **Product Key Memory** [Dehghani+, ICLR 2019]

- 大量のkeyを別途学習しながら multi-head attentionぽいことをやるPKMの提案
- 一部の層のFFNをPKMに変えると、より小規模なネットワークで精度・速度up!
- 前頁では本論文と[Wu+, ICLR 2019]だけがFacebookの論文（他は全てGoogle系）



- **Synthesizer** [Tay+, 2020]

- アテンション行列を  $QK^\top$  から
  - 入力  $X$  の線形変換 + ReLU + 線形変換に (Dense)
  - もう乱数でいいや (Random)
- Performer [Choromanski+, ICLR'21] は  $q$  と  $k$  のカーネルでアテンションを近似



# Switch TransformerとMixture of Experts Transformer

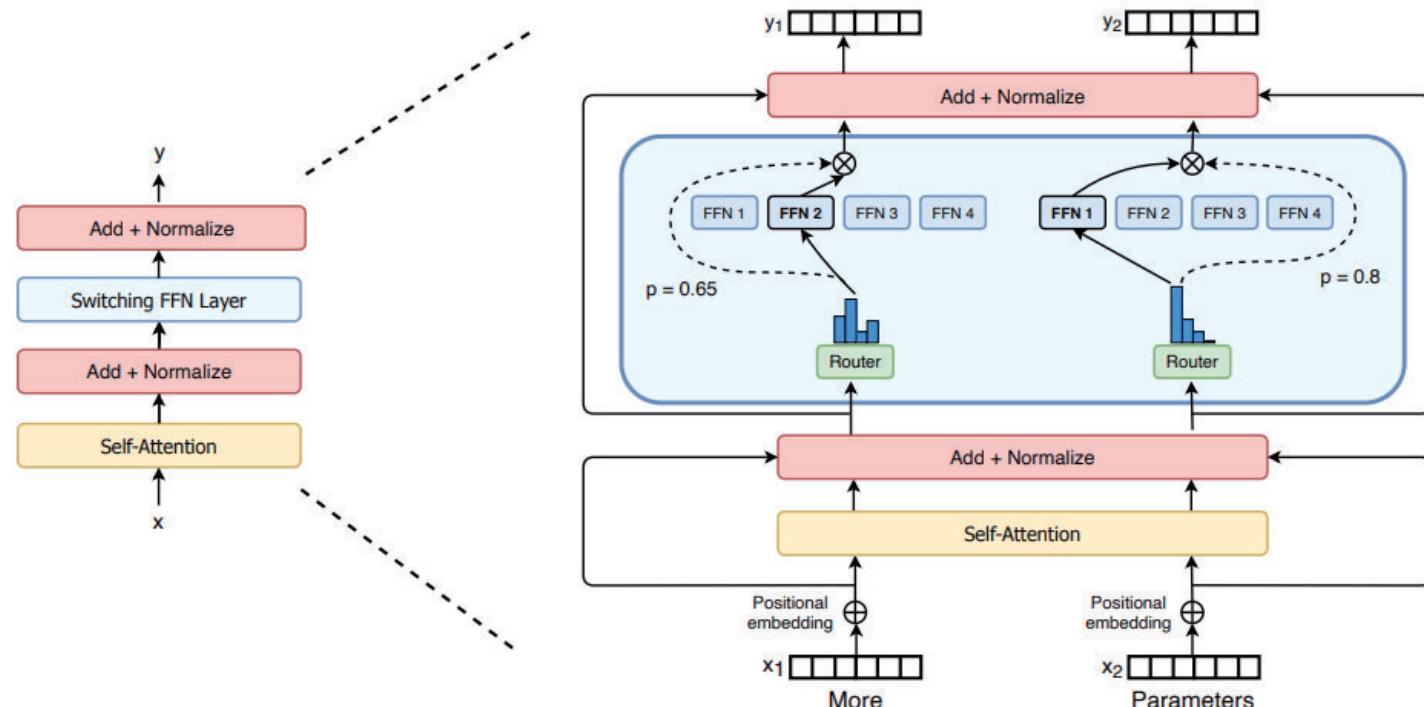
[Shazeer+, NeurIPS 2018][Lepikhin+, ICLR 2021][Fedus+, 2021]

- **Switch Transformer : 1.6兆個のパラメータ**

- と聞くと大変そうだが、FFNが複数ある (Mixture of Experts)
- Switch Transformerでは選択的にこのFFNのうち一つを選ぶ
- ので、全パラメータを毎回の学習や推論に使うわけではない

- **余談**

- これらの一連の論文は Googleによるもの
- 特にNorm Shazeerは
  - Transformer原著の第2著者
  - これらの論文にも著者として入っている



# まとめ

- 近年のTransformerの改善手法の大半が元のTransformerと比べて大差ない
  - 複数課題での汎用性が無い、ソースコードもほとんど変わらない

One possible explanation for this is that the originally-proposed Transformer architecture was near-perfect, and there wasn't much that could be done to improve it.

(これは、当初提案されたTransformerのアーキテクチャが完璧に近く、改良の余地があまりなかったことが理由として考えられます。)



著者ら

- ありがたい格言：新たな改善手法を考えた時は  
**「複数実装をベースに使え」 「cvも含む複数の課題で評価せよ」**  
**「ハイパーパラメータを揃えよ」 「最良値じゃなく平均+分散」**

# それでも：効果が確認された手法

- 実は...Vanilla Transformerでも入っている工夫がある
  - LayerNormが後 → LayerNormが先 [Baevski+Auli 2019][Xiong+, 2020]
  - 絶対値による位置埋め込み → 相対的な位置埋め込み [Raffel+, 2019]
- 活性化関数：GLUとGeLU/Swishの組合せ
- 正規化：RMS Norm
- デコーダにおける入出力の分散表現の共有
- アーキテクチャの工夫
  - Mixture of Experts Transformer
  - Switch Transformer
  - Product key memory
  - Synthesizer variants

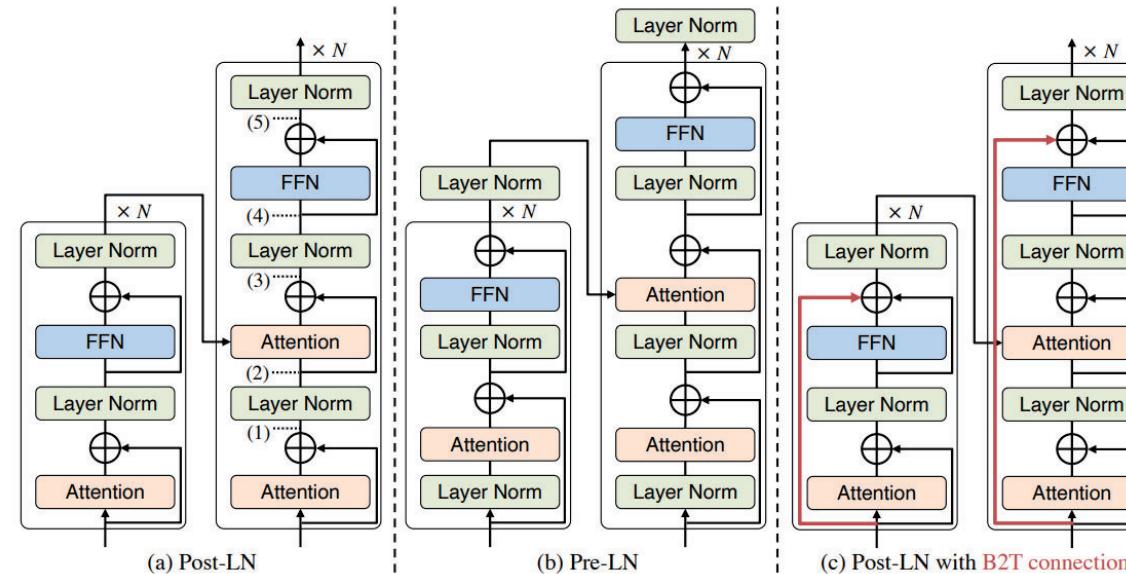


その他の

Transformerのノウハウ

# Layer Normalization が後か先か問題

- Post-LN: 性能が高いが、訓練が不安定 →誤差が途中で消失してしまうのが問題
- Pre-LN: 訓練が安定するが、性能が低い →入力と出力が余り変わらないのが問題



- **Bottom-to-Top (B2T) connection** [Takase+, 2022]
  - Post-LNの変換能力とPre-LNより優れた訓練安定性を両立
- **DeepNorm** [Wang+, 2022]
  - Post-LNベースで、Residual接続がそのまま（1倍）加算されるのを定数倍大きくする
  - パラメータの初期値（の一部を）定数で割って小さくする
  - 1000層のTransformerでも訓練できるようになった

# Positional Encoderの改良とWarmup

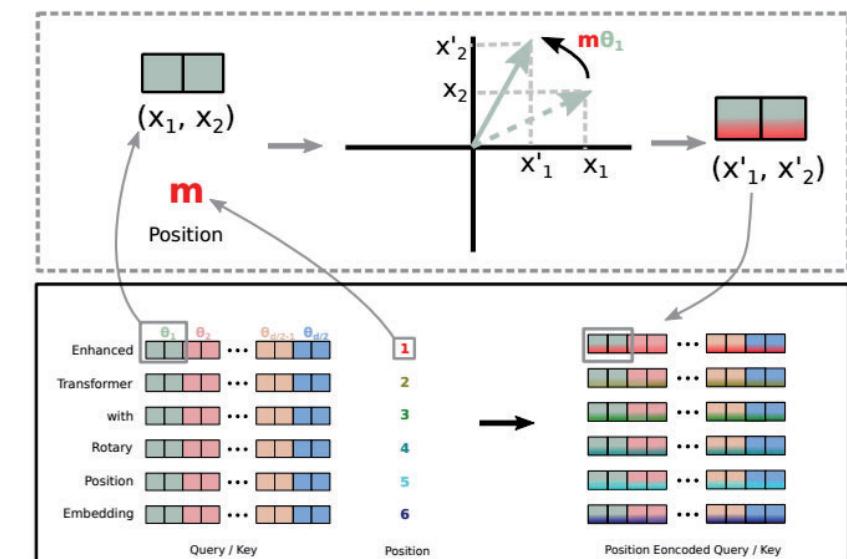
## Positional Encoder

- 元々は絶対位置の埋め込み
- 相対位置による位置埋め込み [Shaw+, NAACL'18] [Raffel+, 2019]
- 絶対位置の位置埋め込みにシフト不変性を導入 [Kiyono+, EMNLP'21]
- RoPE (Rotary Position Embedding) [Su+, 2021]

## 学習率のWarmup

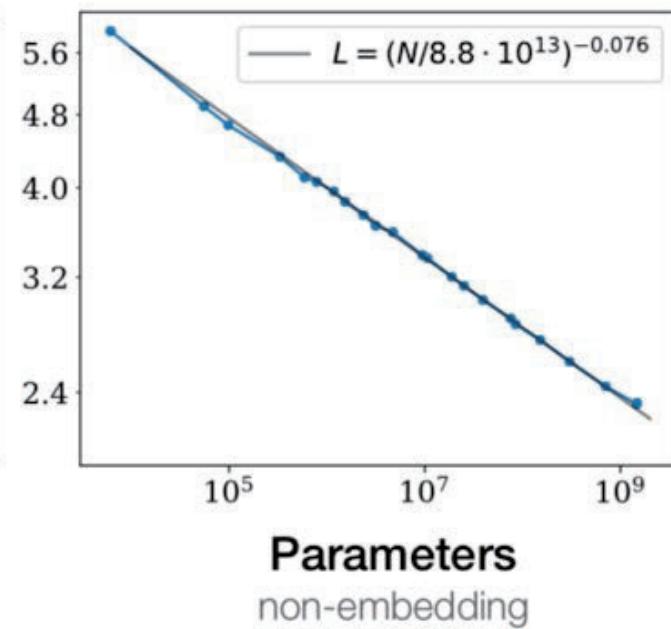
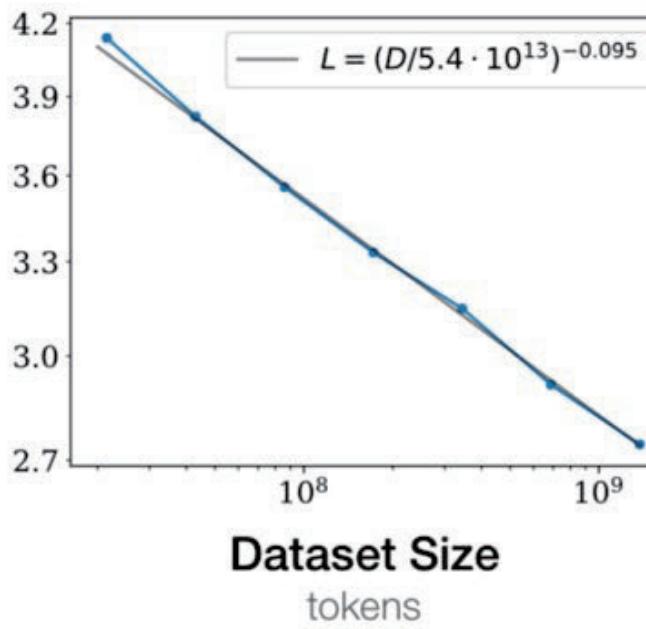
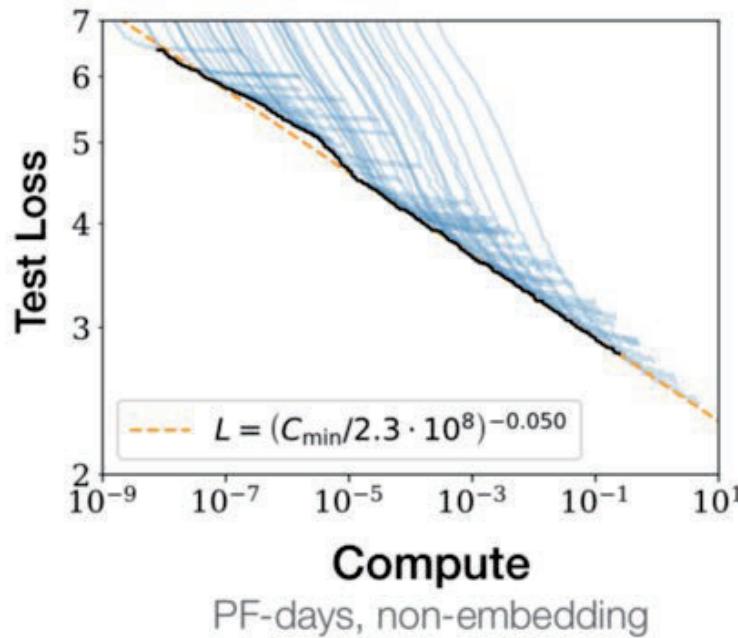
- 最初は $step\_num$  に対して線形に増加
- $step\_num$ が $warmup\_step$ と等しくなると  $step\_num$ の平方根に逆比例して0に漸近

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$



# べき乗則の思い出

- OpenAIによるGPT-3で観測された法則 [Henighan+, 2020]
- 計算予算、データセットサイズ、パラメータが $t$ 倍になると損失が $at^b$ 倍小さくなる ( $a, b$ は定数)

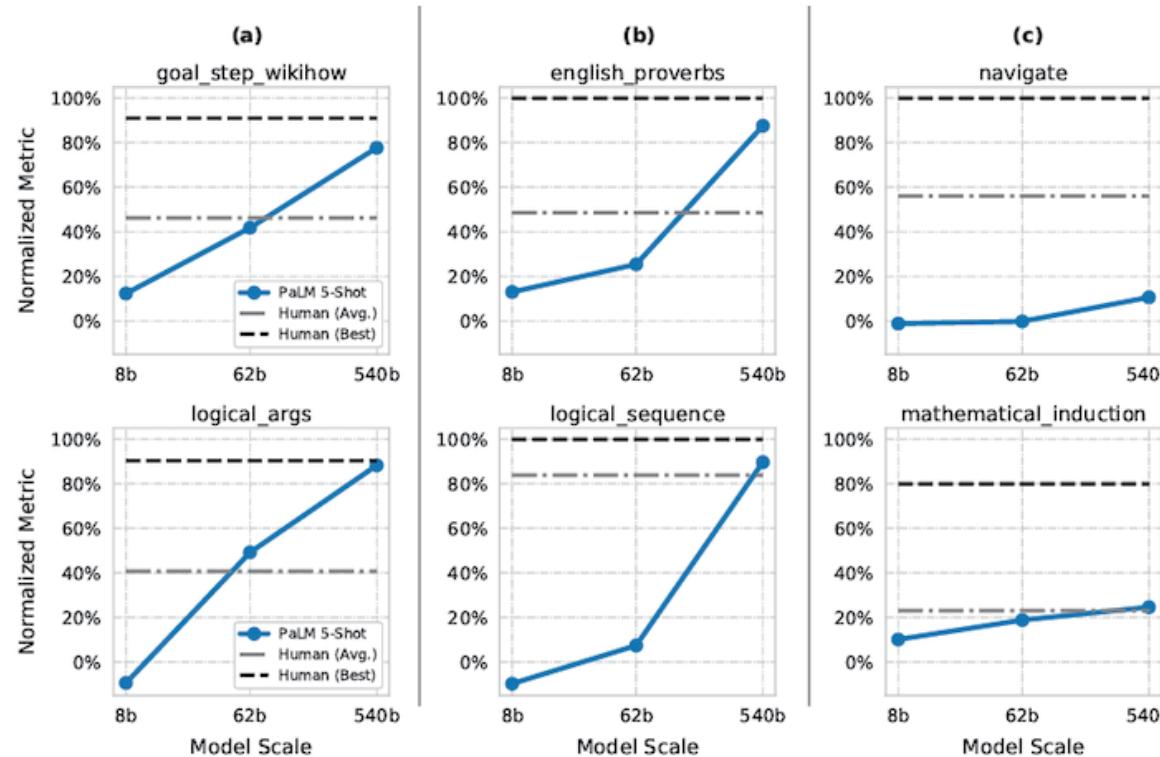


つまり...

リソースの多い奴が勝つ

# べき乗則の思い出

- GoogleによるPaLMでは...
- パラメータが8B→62B→540Bと増えた時に非連続変化

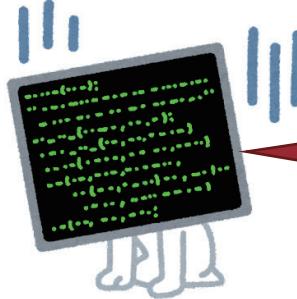


つまり...

更に

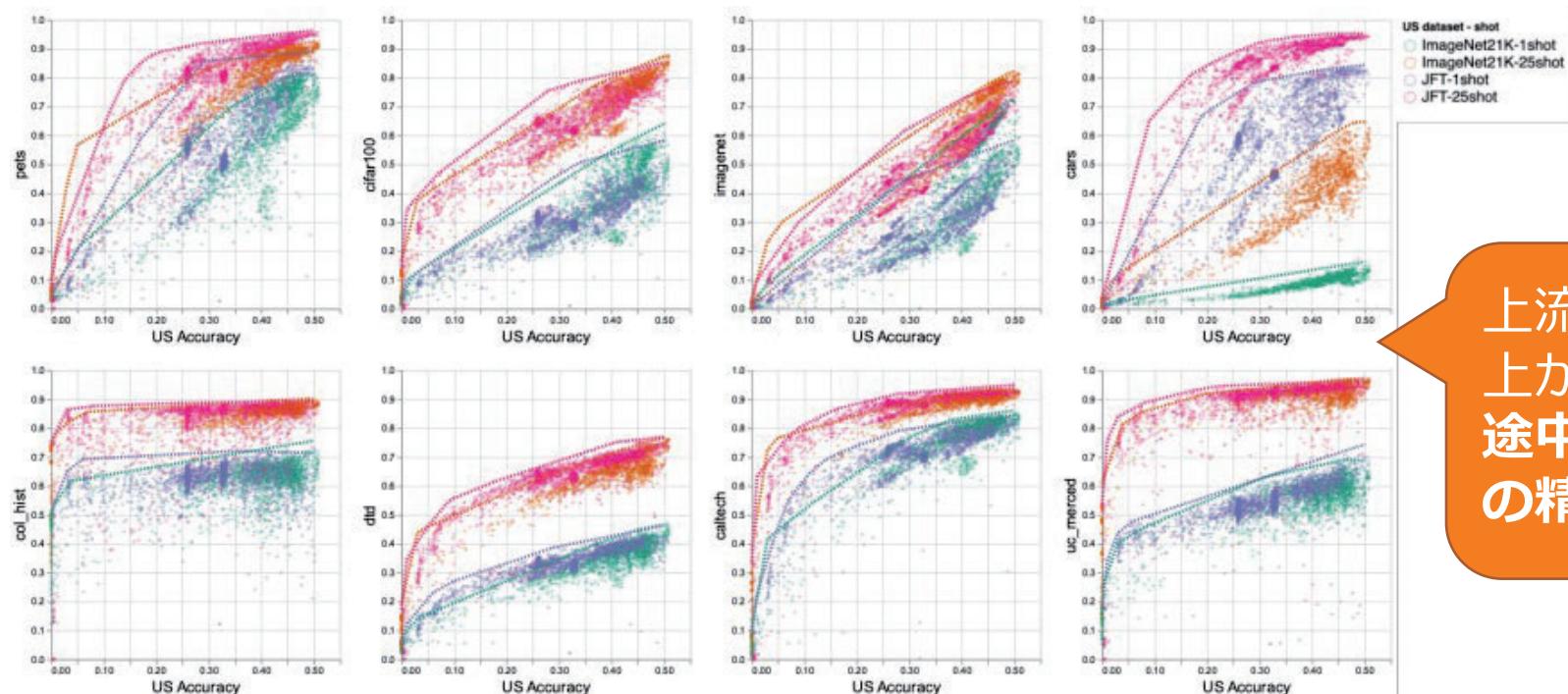
リソースの多い奴が勝つ

# 出てくるかもしれない諦観



巨大な基盤モデルを大規模データセットで訓練して  
後続タスク学習できるところが強いんだ...

- **Exploring the Limits of Large Scale Pre-training** [Abnar+, ICLR'22]
  - JFT-300Mを上流タスクとして学習した時の精度（横軸）
  - vs. 続いて下流で各タスクを学習したときの精度（縦軸）



上流タスクの精度が  
上がっても、  
途中から下流タスク  
の精度は頭打ち

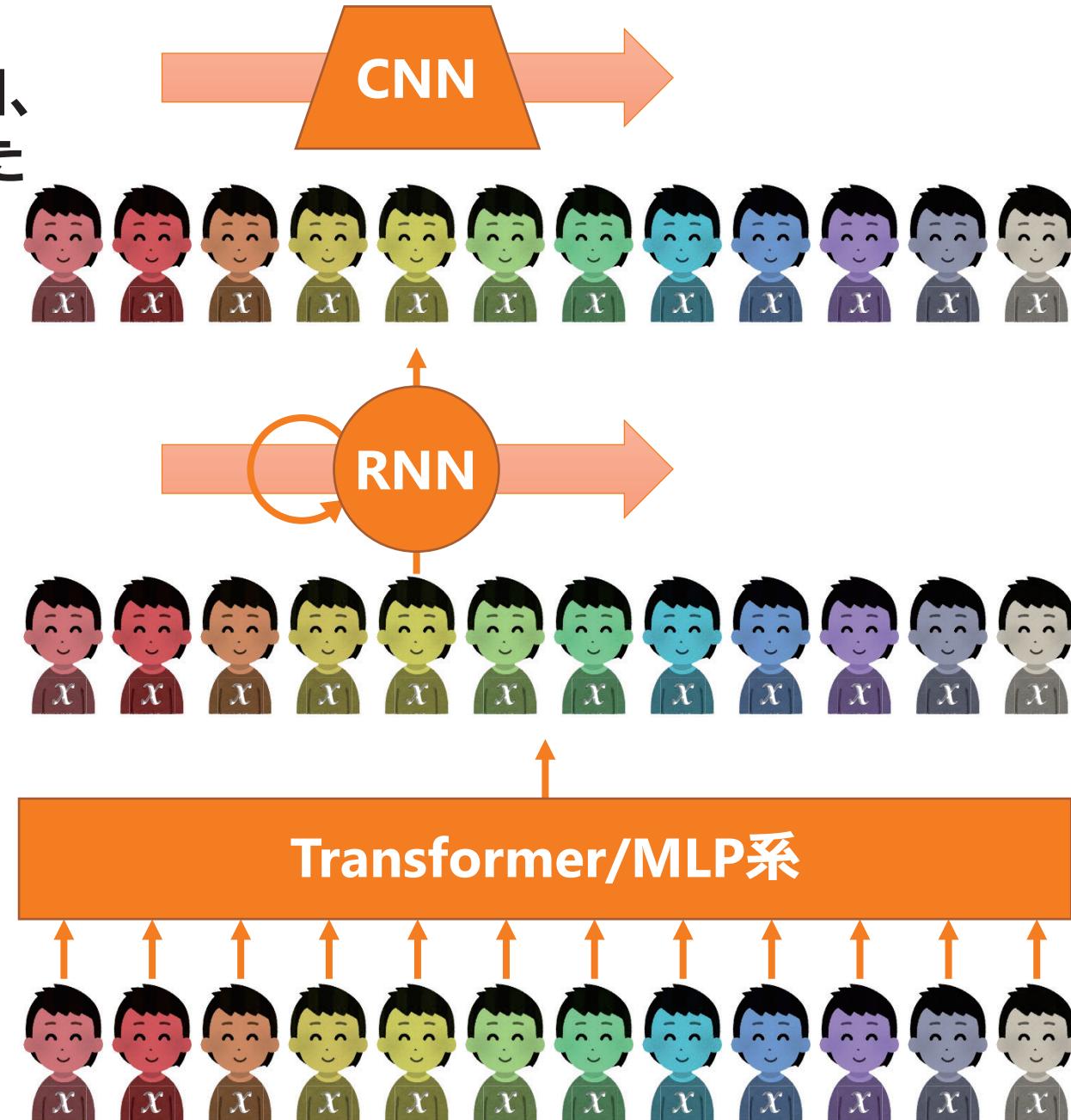
# さいごに

Transformerの基本的な動作から応用範囲、  
最近のMLP系ネットワークまでを俯瞰した

- そもそもTransformerって？
- Transformer旋風
- Transformerはオワコン？！
- Transformerのノウハウ

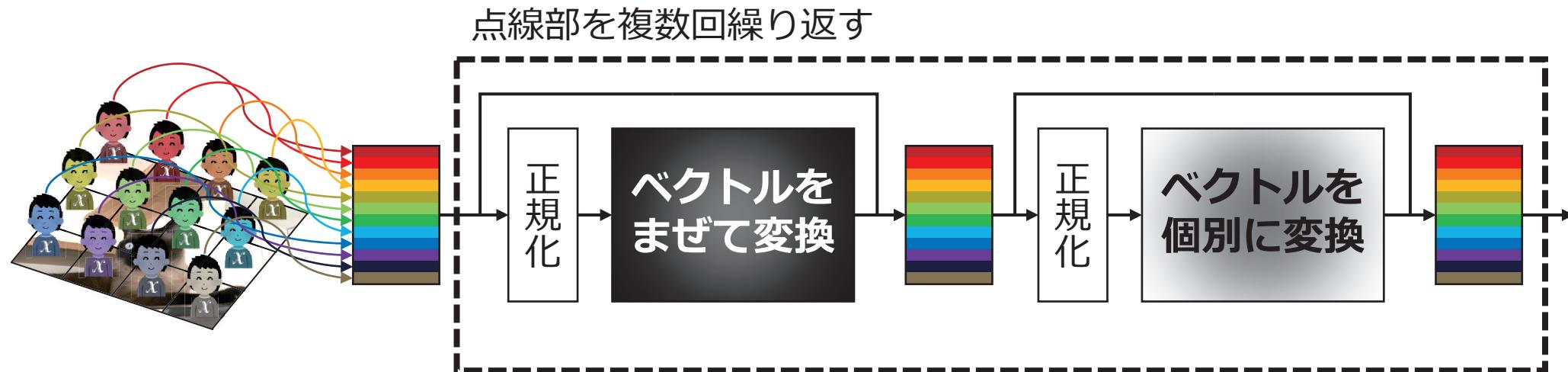
帰納バイアスと必要データ量・計算機の  
トレードオフ

- ベクトルを個別に変換
- ベクトルをまぜて変換  
→ アテンションは有効だけど  
**全てじゃない**



# 色々聞きすぎて良く分からぬ！という人のために

- Transformerはトークン（単語やパッチなど）をResidual接続しながら2つのモジュールを繰り返しているだけ！
  - トークンを混ぜるToken Mixer
  - トークンを変換するMLP
- 話題になったMLP系も実は基本的に同じ構造！



- ...あとは色々とノウハウがあるので気を付けましょう