THE UNIVERSITY OF CHICAGO

ML Journal Club

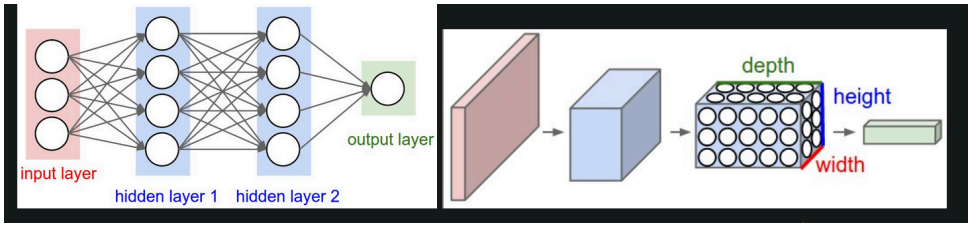
5.15.24

Convolutional architectures

(very short)

CNNs

- Local filter
  - For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter).



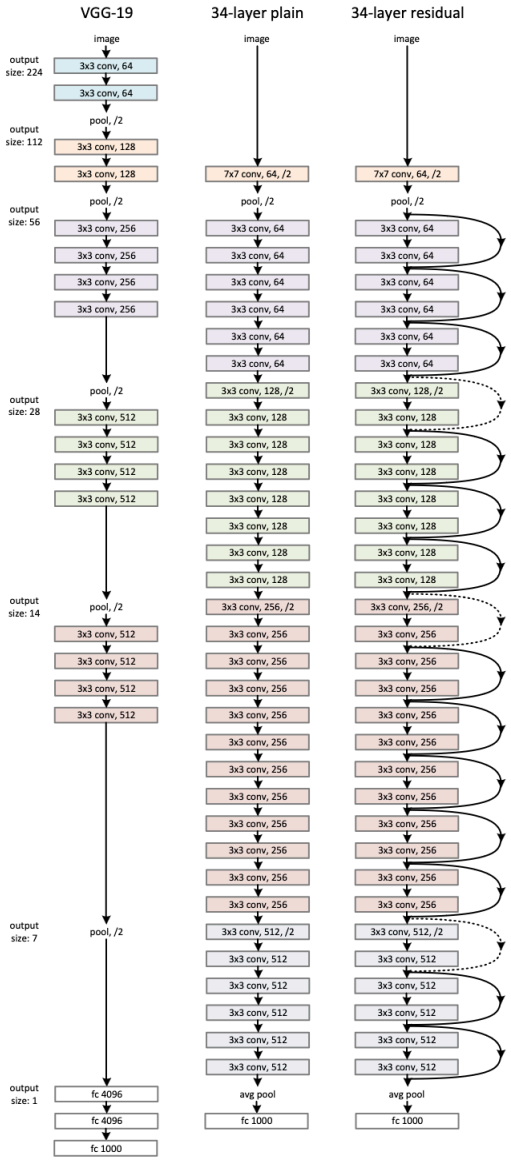
CNN filters

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Don’t implement from scratch, just use a good architecture from ImageNet

- e.g. ResNet (right) or VGG



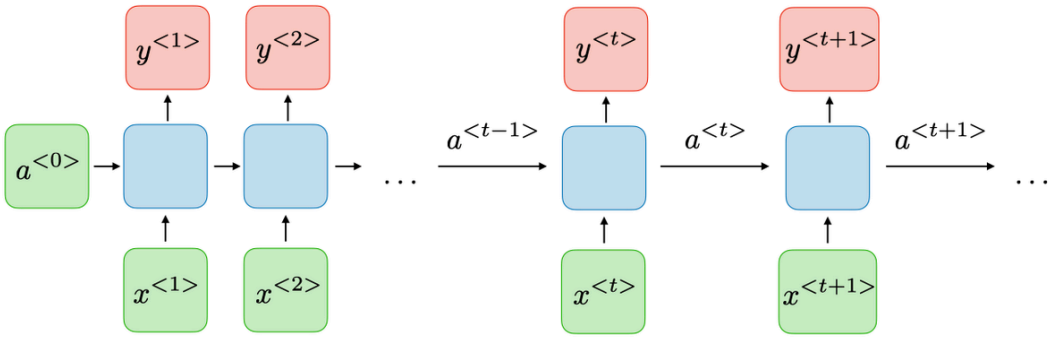
Recurrent neural networks

A short intro

Generic RNNs

- Idea: learn a dynamical system $h^{(t)} = f_\theta(h^{(t-1)}, x^{(t)})$

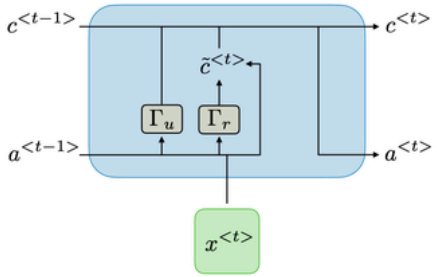
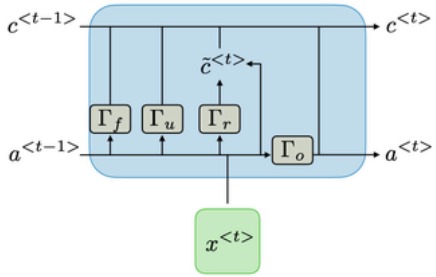
```
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```



https://karpathy.github.io/2015/05/21/rnn-effectiveness/
https://web.stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Plain RNNs have problems

- vanishing or exploding gradients (due to multiplication in chain rule through hidden states)
- hard to maintain long-range dependencies
- instead other architectures: LSTM (Hochreiter & Schmidhuber 1997), GRU

Characterization	**Gated Recurrent Unit** (GRU)	**Long Short-Term Memory** (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$	$\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \star c^{<t>}$
Dependencies		

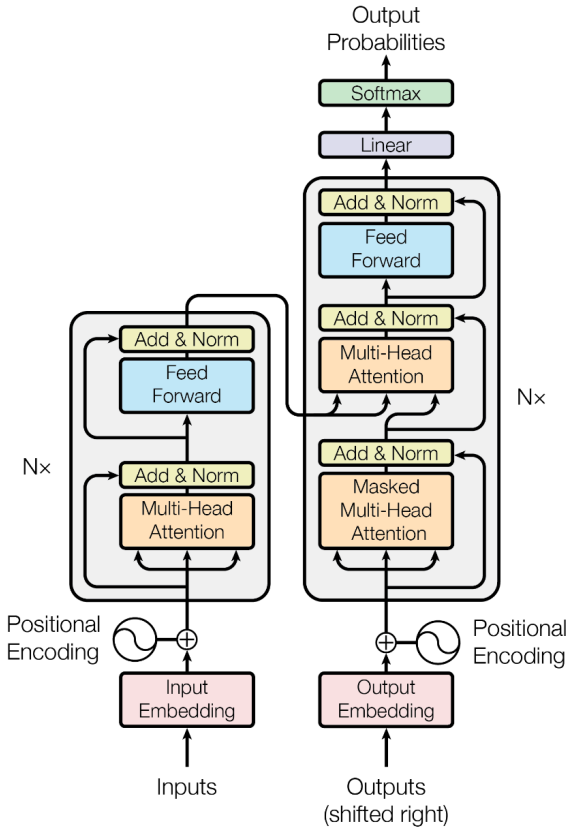
Each gate Γ allows in/out of information by sigmoid: $\Gamma = \sigma(W_1 x^{(t)} + W_2 a^{(t-1)} + b)$

https://statmech.stanford.edu/teaching/2023a_chem263/lecture07_notes.pdf
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Transformers and attention

Transformers overview

- "Attention is all you need" (Vaswani et al. 2017)*
- Main parts
  - embedding
  - positional encoding
  - multi-head attention
    - multiple layers
- Use cases:
  - seq-to-seq (encoder + decoder)
  - BERT (encoder)
  - GPT (decoder)



Listing order is random, interestingly
http://nlp.seas.harvard.edu/annotated-transformer/

Why self-attention?

- Previous architectures (RNN, LSTM etc.) for sequence-to-sequence tasks had a hard time keeping track of long range dependencies
  - info has to back propagate through all the hidden states as the input is consumed
- Attention mechanism: (Bahdanau et al. 2014)
  - model output vector in sequence $x_i^{\ell+1}$ as a weighted sum of input vectors in sequence: $x_i^{\ell+1} = \sum_j \alpha_{ij} x_j^{\ell}$
  - this means x_i can "attend" to all other vectors at once, without having to propagate through some hidden layers
  - note: in this form, you ***lose positional information***

How is self-attention implemented?

- Recall that we need a way of assigning weights to each of the input vectors x_i (n vectors total, can write as $X \in \mathbb{R}^{n \times d}$)
- We use a softmax: $\alpha_{ij} = \textsf{softmax}_j(S) = \frac{e^{S_{ij}}}{\sum_j e^{S_{ij}}}$ where $S \in \mathbb{R}^{n \times n}$ is a similarity matrix between some representation of vectors x_i and x_j
- Similarity matrix is also now learnable by using a linear projection:

$$S = \frac{(XQ)(XK)^\top}{\sqrt{d_k}} \text{ where } Q, K \in \mathbb{R}^{d \times d_k} \text{ or equivalently } S_{ij} = \frac{(x_i Q)(x_j K)^\top}{\sqrt{d_k}}$$

  - the denominator scaling is just to make sure things don't blow up
  - finally, we also let the actual value of the input vector be learned as XV where $V \in \mathbb{R}^{n \times d_v}$

Self-attention (cont.)

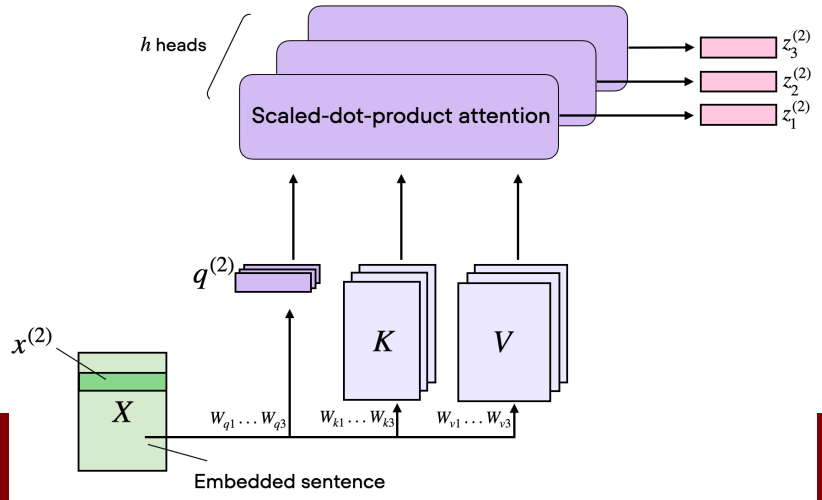
- Putting this all together:

$$\mathsf{SelfAttention}(X) = \mathsf{softmax}_j\left(\frac{(XQ)(XK)^\top}{\sqrt{d_k}}\right)(XV) \in \mathbb{R}^{n \times d_v}$$

- Trainable parameters: Q, K, V (traditionally called "query", "key" and "value")
-

Multihead self-attention (MHSA)

- Self-attention in principle allows you to learn a single feature/channel across the input vectors
- What if you need to know more? treat the self-attention as "head" and then stack them together
  - $\text{MultiHead}(X) = [\text{head}_1, \ldots, \text{head}_h]W^O$ where $\text{head}_i = \text{Attention}(XQ_i, XK_i, XV_i)$ and $Q_i, K_i \in \mathbb{R}^{d \times d_k}$, $V_i \in \mathbb{R}^{d \times d_v}$, and final linear projection $W \in \mathbb{R}^{hd_v \times d_{out}}$
  - typically you choose $d_k = d_v = d/h$ so the total dimensionality is divided among h heads (e.g. $d = 64$, $h = 8$, $d_k = 8$)



MHSA (cont.)

- Already implemented in Pytorch
- Notes
  - computational cost scales quadratically with input sequence length (since you need to make an n x n matrix)
  - many different versions which try to increase performance
  - invariant to permutation of inputs
  - can't have arbitrary sequence length (maximum with padding)

MultiheadAttention

CLASS torch.nn.MultiheadAttention(*embed_dim*, *num_heads*, *dropout=0.0*, *bias=True*, *add_bias_kv=False*, *add_zero_attn=False*, *kdim=None*, *vdim=None*, *batch_first=False*, *device=None*, *dtype=None*) [SOURCE]

Allows the model to jointly attend to information from different representation subspaces.

Method described in the paper: Attention Is All You Need.

Multi-Head Attention is defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \dots, head_h)W^O$$

where $head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

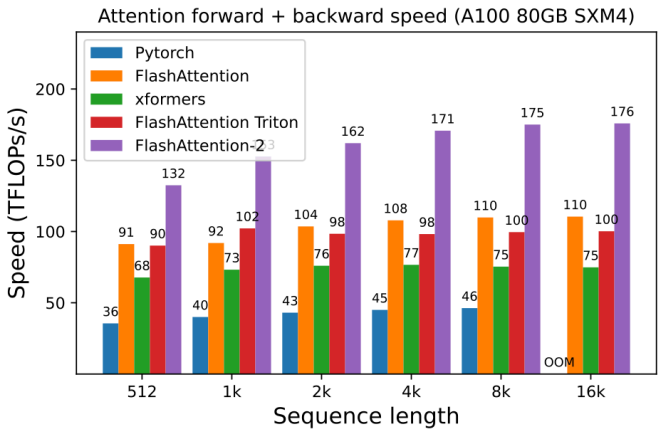
`nn.MultiHeadAttention` will use the optimized implementations of `scaled_dot_product_attention()` when possible.

In addition to support for the new `scaled_dot_product_attention()` function, for speeding up Inference, MHA will use fastpath inference with support for Nested Tensors, iff:

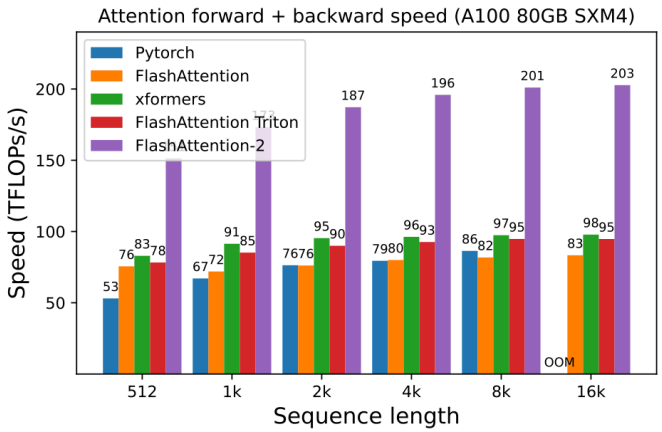
```
]: import torch.nn as nn

class SelfAttention(nn.Module):

    def __init__(self, d_in, d_out_kq, d_out_v):
        super().__init__()
        self.d_out_kq = d_out_kq
        self.W_query = nn.Parameter(torch.rand(d_in, d_out_kq))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out_kq))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out_v))

    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value

        attn_scores = queries @ keys.T  # unnormalized attention weights
        attn_weights = torch.softmax(
            attn_scores / self.d_out_kq**0.5, dim=-1
        )

        context_vec = attn_weights @ values
        return context_vec
```

Flash-Attention

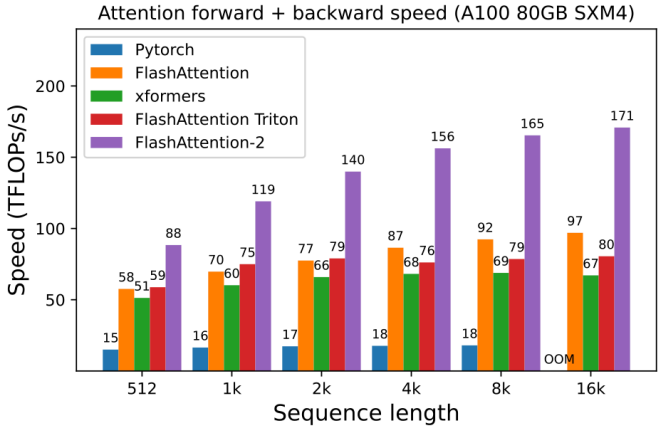
You don't really need linear model



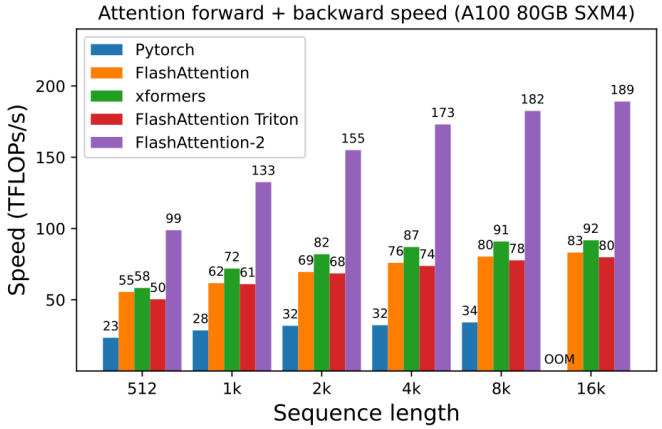
(a) Without causal mask, head dimension 64



(b) Without causal mask, head dimension 128



(c) With causal mask, head dimension 64



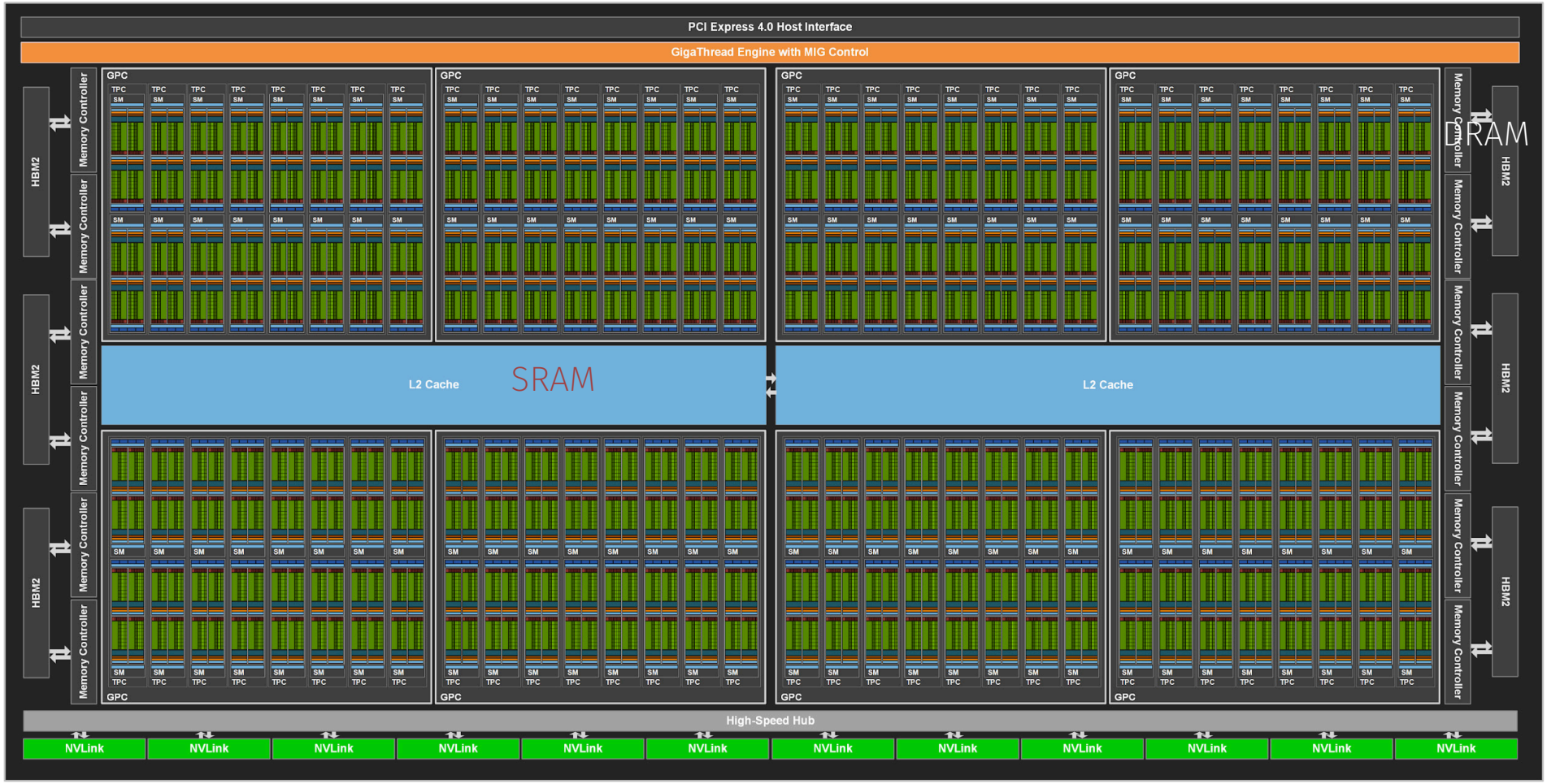
(d) With causal mask, head dimension 128

Figure 4: Attention forward + backward speed on A100 GPU

Flash-Attention

SRAM: fast, expensive, power-efficient (L1/L2 cache)

DRAM: slow, cheap, power-hungry (HBM, DDR)



A100 architecture (GA100)

Flash-Attention

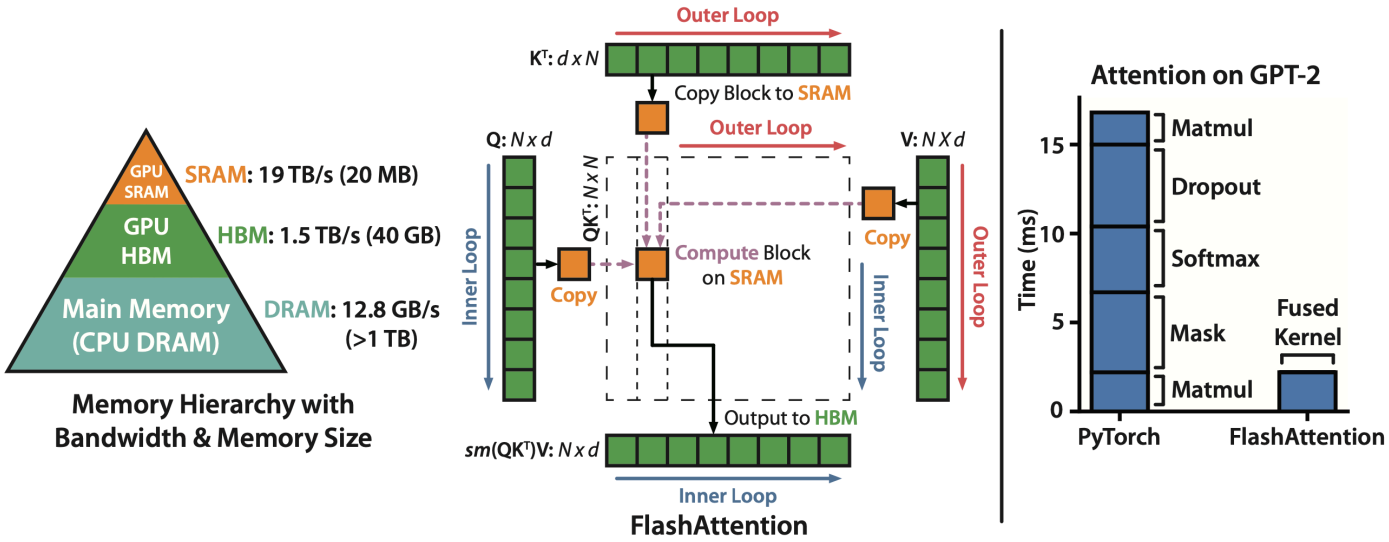
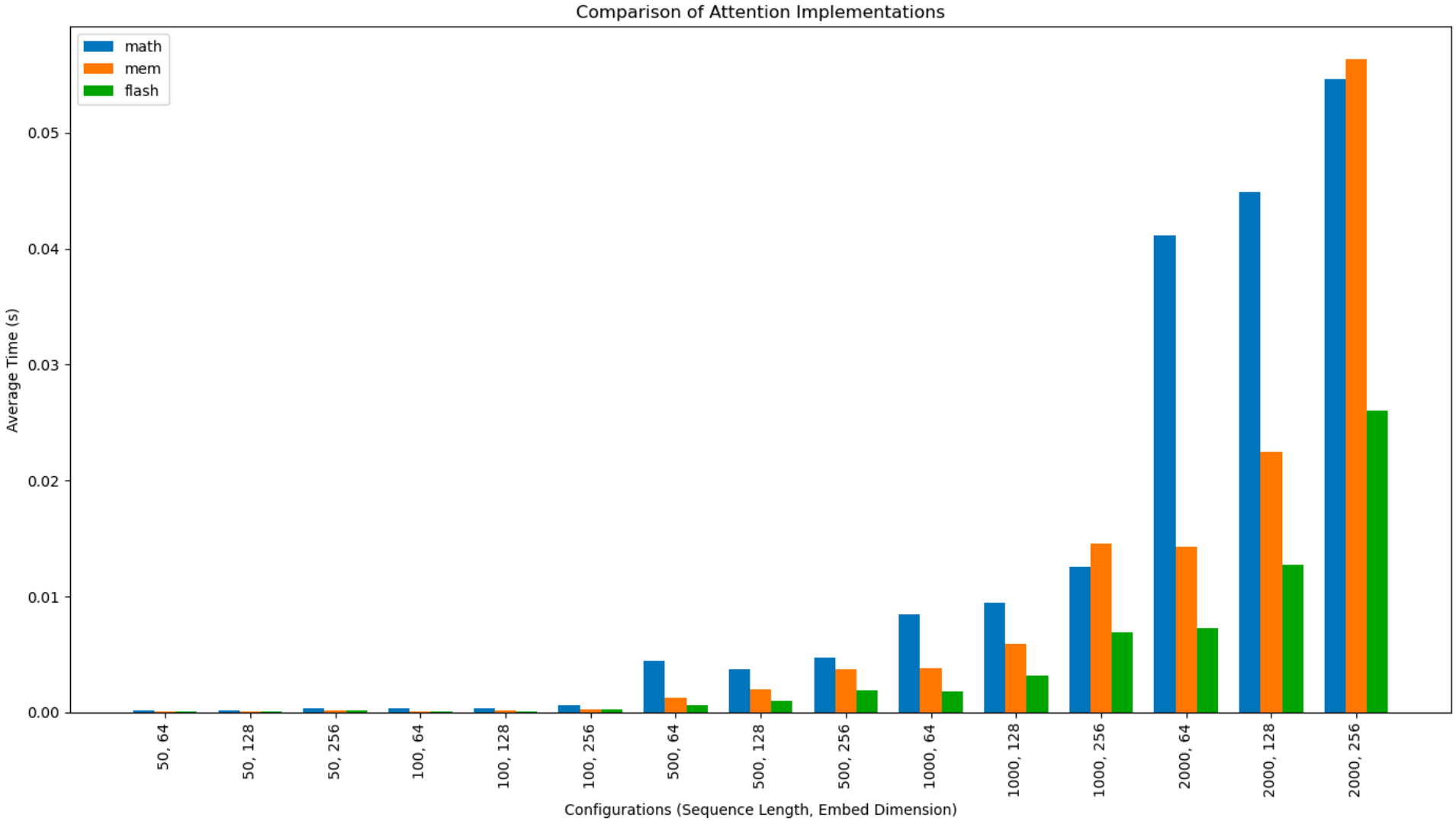
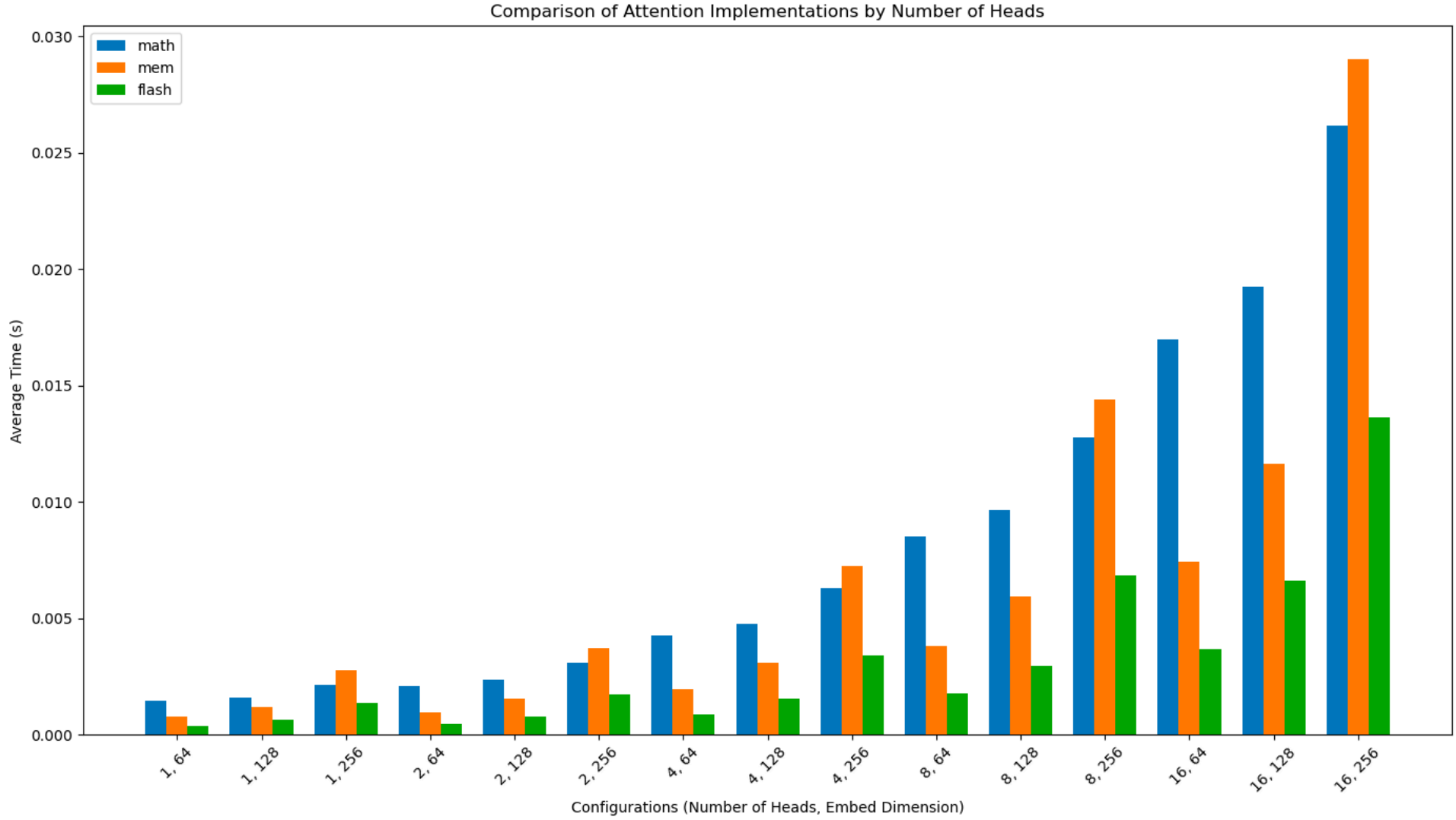


Figure 1: **Left:** FLASHATTENTION uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FLASHATTENTION loops through blocks of the **K** and **V** matrices and loads them to fast on-chip SRAM. In each block, FLASHATTENTION loops over blocks of **Q** matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM. **Right:** Speedup over the PyTorch implementation of attention on GPT-2. FLASHATTENTION does not read and write the large $N \times N$ attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

MHSA (cont.)


Comparison of Attention Implementations
math
mem
flash
Average Time (s)
0.00
0.01
0.02
0.03
0.04
0.05
50, 64
50, 128
50, 256
100, 64
100, 128
100, 256
500, 64
500, 128
500, 256
1000, 64
1000, 128
1000, 256
2000, 64
2000, 128
2000, 256
Configurations (Sequence Length, Embed Dimension)

MHSA (cont.)


Comparison of Attention Implementations by Number of Heads
math
mem
flash
Average Time (s)
0.030
0.025
0.020
0.015
0.010
0.005
0.000
1, 64
1, 128
1, 256
2, 64
2, 128
2, 256
4, 64
4, 128
4, 256
8, 64
8, 128
8, 256
16, 64
16, 128
16, 256
Configurations (Number of Heads, Embed Dimension)

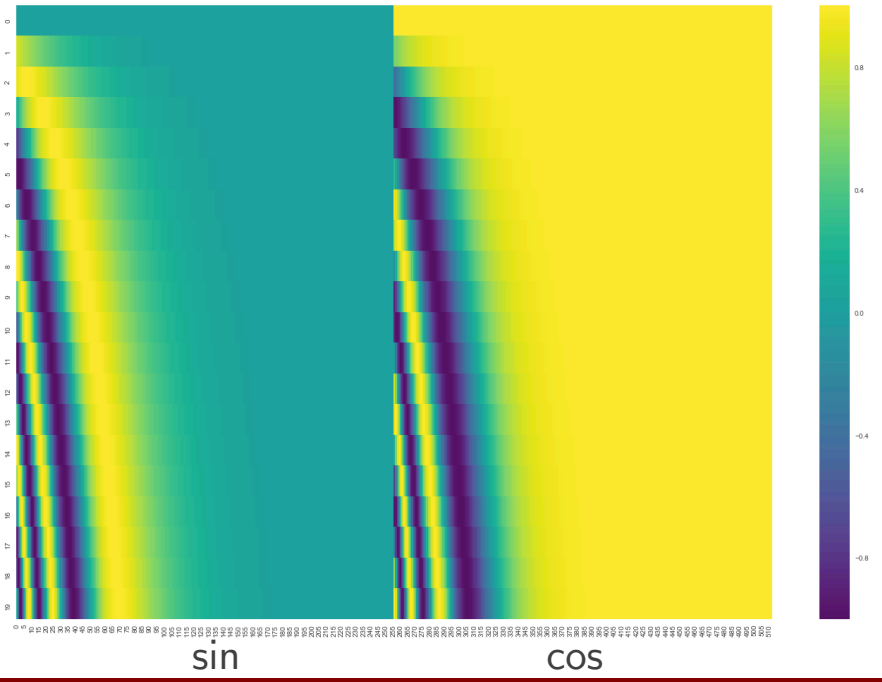
Embedding

- Takes length n input (e.g. seq of works) and converts into vectors of fixed length
  - simplest: use a lookup table (previously trained, e.g. word2vec)
  - can think of a one-hot encoding -> vector
- "I want to eat pie" (5 words) -> $h_i \in \mathbb{R}^{d_{embed}} | i = \{1,\ldots,5\}$
- Example

```
# need to specify size of dictionary 'max_atomic'
embedder = torch.nn.Embedding(max_atomic, 64)
x = torch.LongTensor([1, 6, 7]) # shape (n_batch,)
h = embedder(x) # shape (n_batch, 64)
```

Positional encoding

- Recall that because we are taking weighted sum of inputs, the attention mechanism has no sense of ordering unless we tell it
- In the original transformer, they use an encoding about the position $PE_{\text{pos},2i} = \sin(\text{pos}/10000^{2i/d})$ and $PE_{\text{pos},2i+1} = \cos(\text{pos}/10000^{2i/d})$ (very confusing to look at)
- in principle can be learned



Rotary Positional Embedding

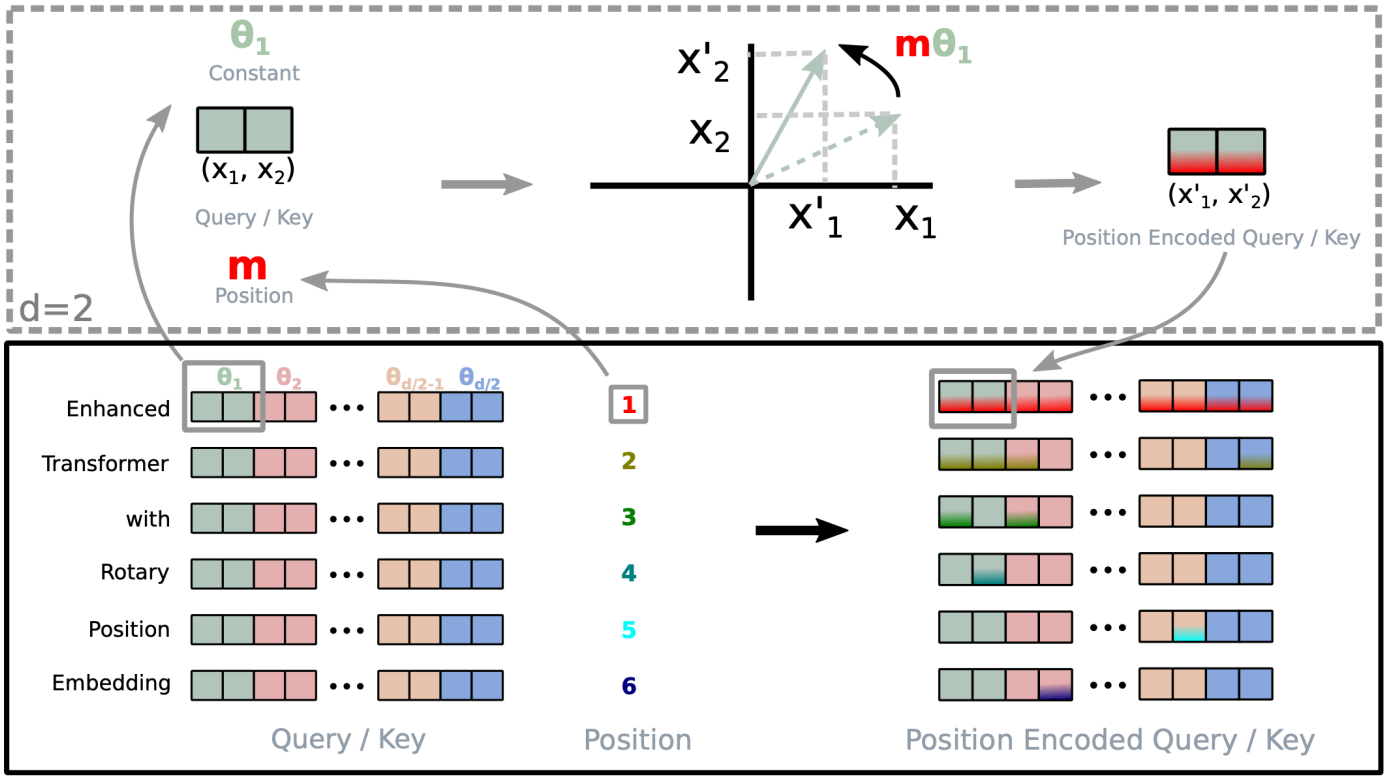


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Extrapolative positional encoding to longer sequences, widely applied to current day LLMs (LLaMA, GLM, ESM-2)

Rotary Positional Embedding

Let $\mathbb{S}_N = \{w_i\}_{i=1}^N$ be a sequence of N input tokens with w_i being the i^{th} element. The corresponding word embedding of $\mathbb{S}_N$ is denoted as $\mathbb{E}_N = \{\boldsymbol{x}_i\}_{i=1}^N$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the d-dimensional word embedding vector of token w_i without position information. The self-attention first incorporates position information to the word embeddings and transforms them into queries, keys, and value representations.

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_m, m) \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_n, n) \\ \boldsymbol{v}_n &= f_v(\boldsymbol{x}_n, n), \end{aligned} \tag{1}$$

General PE form

$$f_{t:t\in\{q,k,v\}}(\boldsymbol{x}_i, i) := \boldsymbol{W}_{t:t\in\{q,k,v\}}(\boldsymbol{x}_i + \boldsymbol{p}_i)$$

$$\begin{cases} \boldsymbol{p}_{i,2t} &= \sin(k/10000^{2t/d}) \\ \boldsymbol{p}_{i,2t+1} &= \cos(k/10000^{2t/d}) \end{cases}$$

Additive PE

$$\langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n)\rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n)$$

$$f_q(\boldsymbol{x}_m, m) = (\boldsymbol{W}_q \boldsymbol{x}_m)e^{im\theta}$$

$$f_k(\boldsymbol{x}_n, n) = (\boldsymbol{W}_k \boldsymbol{x}_n)e^{in\theta}$$

$$g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n) = \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}]$$

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

Multiplicative PE

Rotary Positional Embedding

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m$$

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

Can be seen as multiplying a rotation matrix in 2D case

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

Effectively equals to

Rotary Positional Embedding

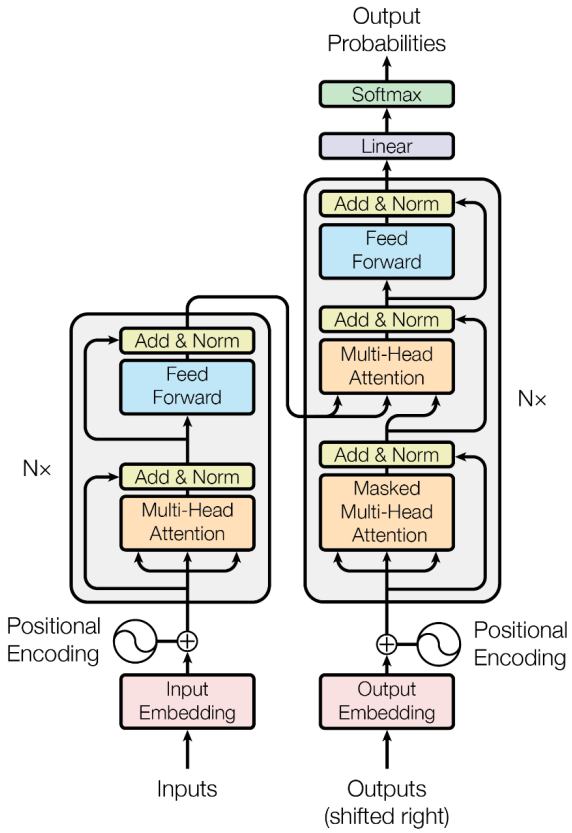
Table 4: Pre-training strategy of RoFormer on Chinese dataset. The training procedure is divided into various consecutive stages. In each stage, we train the model with a specific combination of maximum sequence length and batch size.

Stage	Max seq length	Batch size	Training steps	Loss	Accuracy
1	512	256	200k	1.73	65.0%
2	1536	256	12.5k	1.61	66.8%
3	256	256	120k	1.75	64.6%
4	128	512	80k	1.83	63.4%
5	1536	256	10k	1.58	67.4%
6	512	512	30k	1.66	66.2%

Can extrapolate to much longer sentences

What else

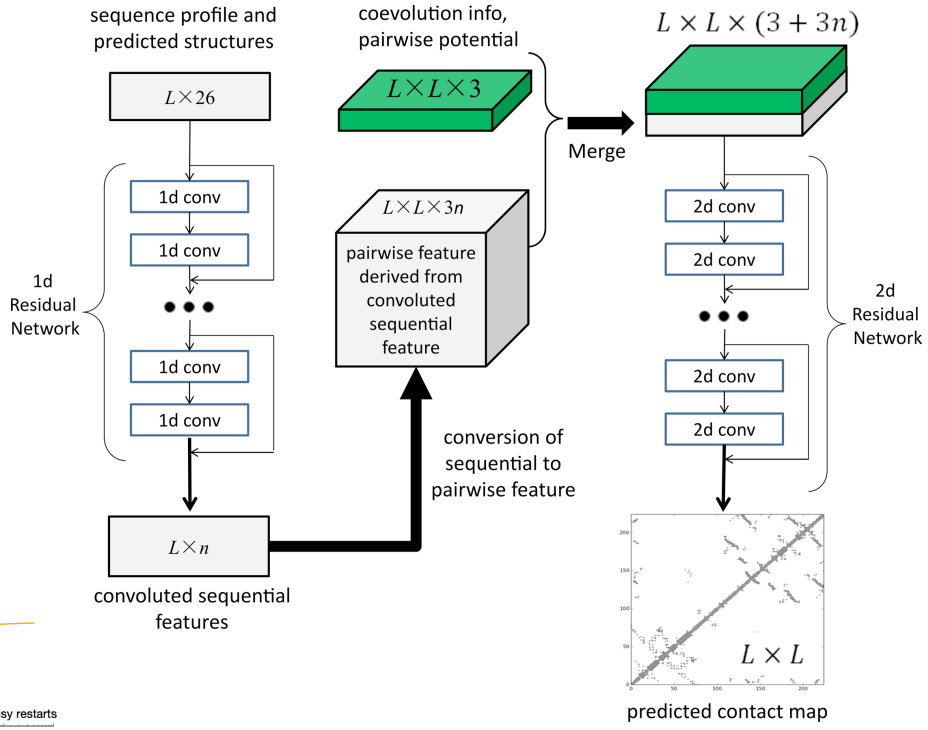
- Important other notes: one attention layer also includes
  - residual connections
  - layer norm after MHSA
  - Feedforward network (2 layer MLP with ReLU)
- Stack multiple attention layers together
- Decode: masked attention (can't see future information)
  - basically apply a lower triangular matrix to attention weights

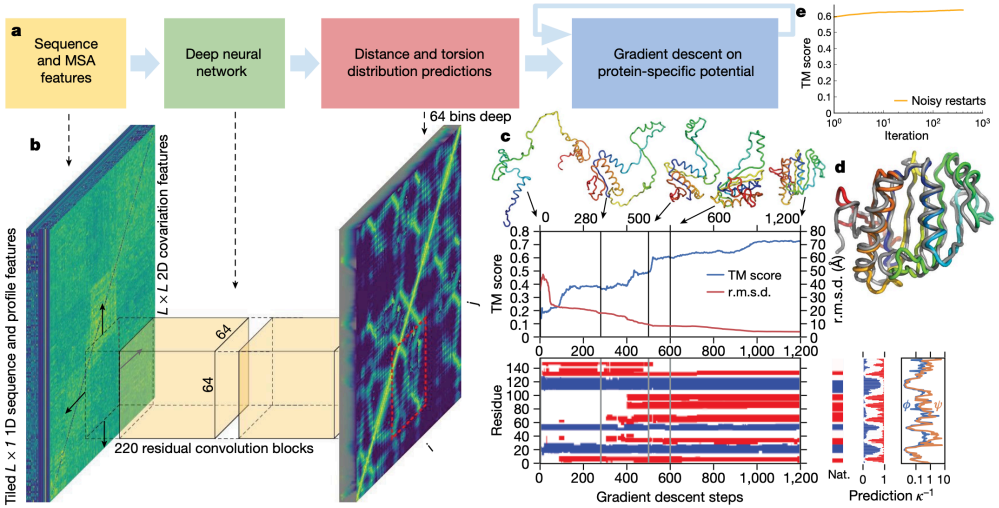


Applications in science

CNNs

- AlphaFold 1
- Wang et al. PLos Comp. Bio 2017

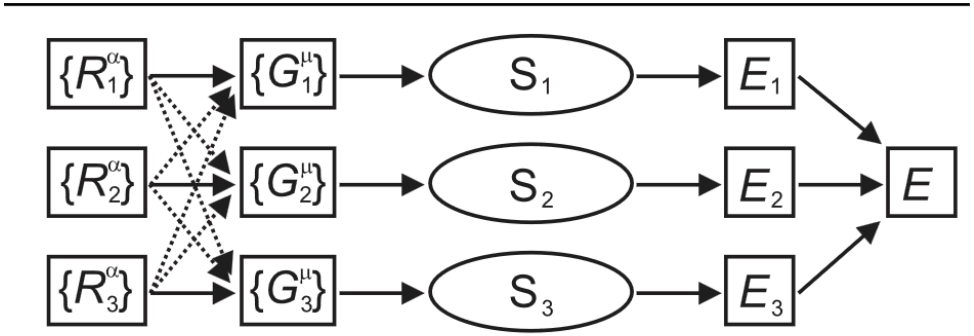




More about embeddings

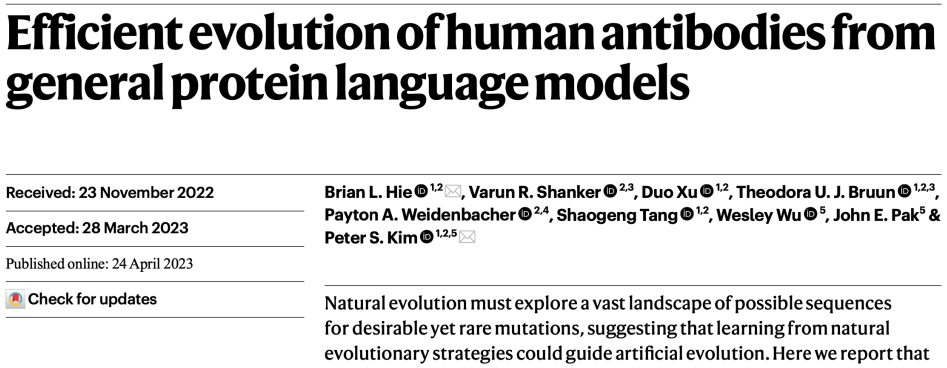
- Behler-Parrinello symmetry functions (working in 3D space)
  - encode local information
  - $G_i^1 = \sum_{j \neq i} e^{-\eta(R_{ij}-R_s)^2} f_c(R_{ij})$, f_c goes to 0 at chosen cutoff distance
  - Angular term

$$G_i^2 = C \sum_{j,k \neq i} (1 + \lambda \cos\theta_{ijk})^{\xi} \times e^{-\eta(R_{ij}^2+R_{jk}^2+R_{ki}^2)} f_c(R_{ij}) f_c(R_{jk}) f_c(R_{ki})$$



Embeddings

- Protein sequence embeddings (ESM)
  - amino acid -> vector
  - transformer architecture
- Protein structure
  - GearNet
- Protein surface
  - MaSIF
  - uses convolutions over geodesic on protein surface
- many, many applications



Efficient evolution of human antibodies from general protein language models

Received: 23 November 2022

Accepted: 28 March 2023

Published online: 24 April 2023

Check for updates

Brian L. Hie [1,2], Varun R. Shanker [2,3], Duo Xu [1,2], Theodora U. J. Bruun [1,2,3], Payton A. Weidenbacher [2,4], Shaogeng Tang [1,2], Wesley Wu [5], John E. Pak [5] & Peter S. Kim [1,2,5]

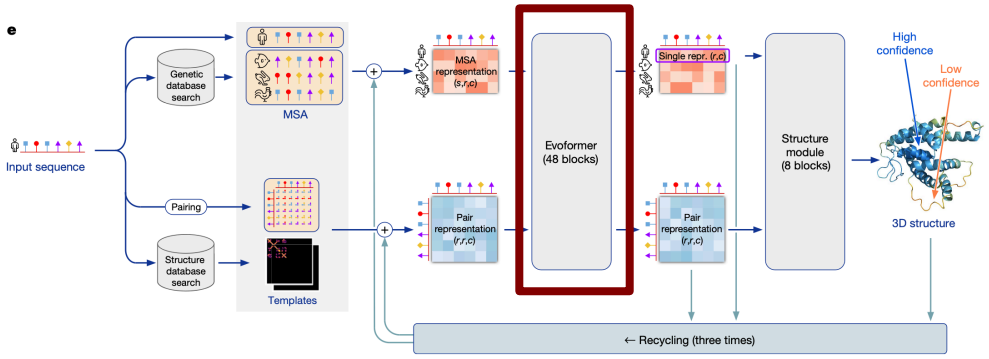
Natural evolution must explore a vast landscape of possible sequences for desirable yet rare mutations, suggesting that learning from natural evolutionary strategies could guide artificial evolution. Here we report that

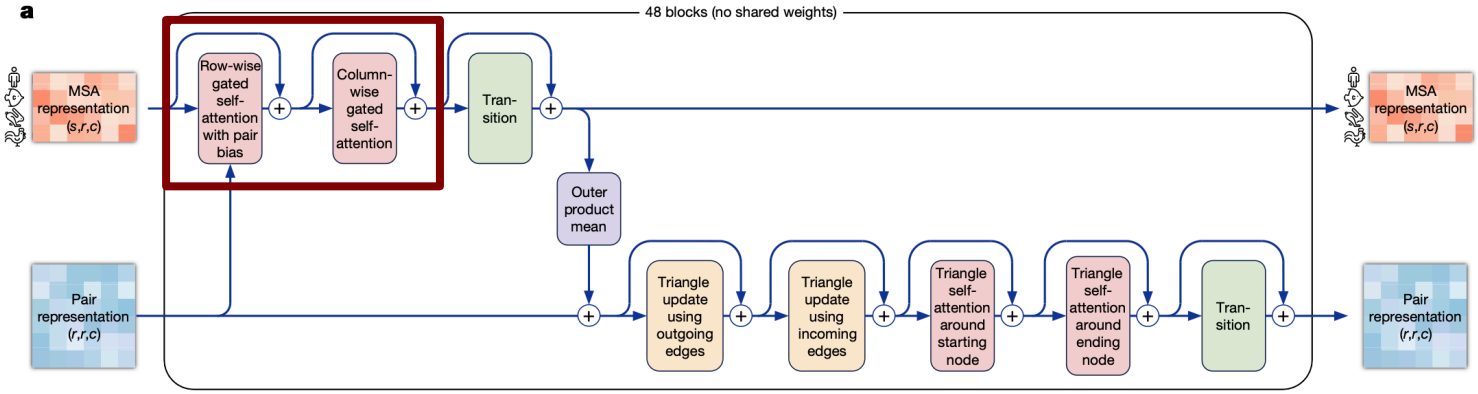
Variations/applications of transformers

- Very generic, e.g. Vision transformer
  - just do attention over patches of an image
- SE(3) Transformer (Fuchs, ... Welling 2020)
  - used in RoseTTAFold
- Equiformer (Liao and Smidt 2022)
  - used in EquiFold
- GPT (transformers scale well)
  - GPT-1 had 12 self-attention heads, 768 dims
  - GPT-3 has 96 layers, 12,288 dims
- Graph transformers: allow you to see the whole graph structure at once
  - need special position encoding (e.g. Laplacian eigenvectors)
- Language models in other domains
  - chemistry: SMILES (MolFormer - Ross et al. 2021)
  - protein sequences: ESM (

Variations on a theme

- Evoformer (Alphafold2)
  - attention within sequence positions and across sequences

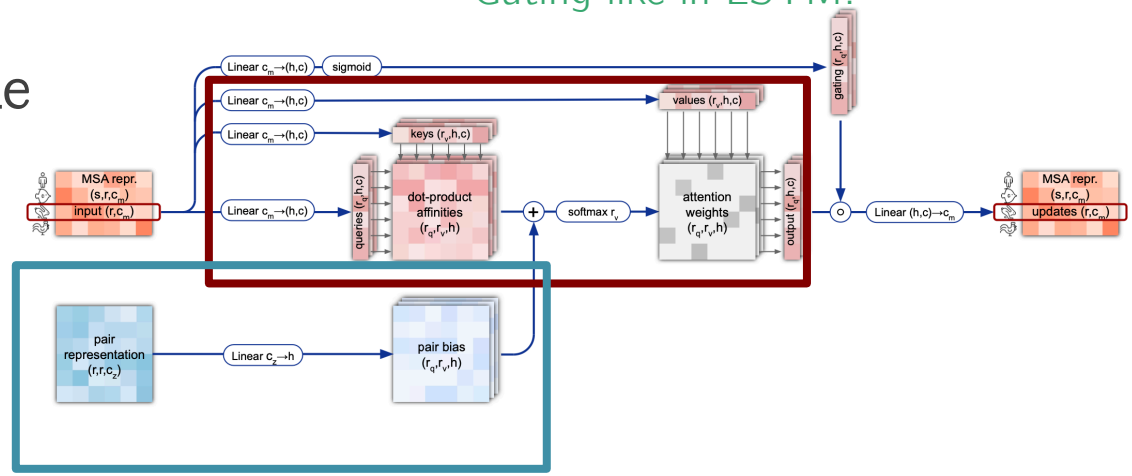




Gating like in LSTM!

Variations on a theme

- Evoformer (Alphafold2)



Algorithm 7 MSA row-wise gated self-attention with pair bias

def MSARowAttentionWithPairBias($\{\mathbf{m}_{si}\}, \{\mathbf{z}_{ij}\}, c = 32, N_{\text{head}} = 8$) :

 # Input projections

1: $\mathbf{m}_{si} \leftarrow \text{LayerNorm}(\mathbf{m}_{si})$

2: $\mathbf{q}^h_{si}, \mathbf{k}^h_{si}, \mathbf{v}^h_{si} = \text{LinearNoBias}(\mathbf{m}_{si})$ $\quad \mathbf{q}^h_{si}, \mathbf{k}^h_{si}, \mathbf{v}^h_{si} \in \mathbb{R}^c,\ h \in \{1, \ldots, N_{\text{head}}\}$

3: $b^h_{ij} = \text{LinearNoBias}(\text{LayerNorm}(\mathbf{z}_{ij}))$

4: $\mathbf{g}^h_{si} = \text{sigmoid}\left(\text{Linear}(\mathbf{m}_{si})\right)$ $\quad \mathbf{g}^h_{si} \in \mathbb{R}^c$

 # Attention

5: $a^h_{sij} = \text{softmax}_j\left(\frac{1}{\sqrt{c}}\, \mathbf{q}^{h\top}_{si} \mathbf{k}^h_{sj} + b^h_{ij}\right)$

6: $\mathbf{o}^h_{si} = \mathbf{g}^h_{si} \odot \sum_j a^h_{sij} \mathbf{v}^h_{sj}$

Gating like in LSTM!

 # Output projection

7: $\tilde{\mathbf{m}}_{si} = \text{Linear}\left(\text{concat}_h(\mathbf{o}^h_{si})\right)$ $\quad \tilde{\mathbf{m}}_{si} \in \mathbb{R}^{c_m}$

8: **return** $\{\tilde{\mathbf{m}}_{si}\}$