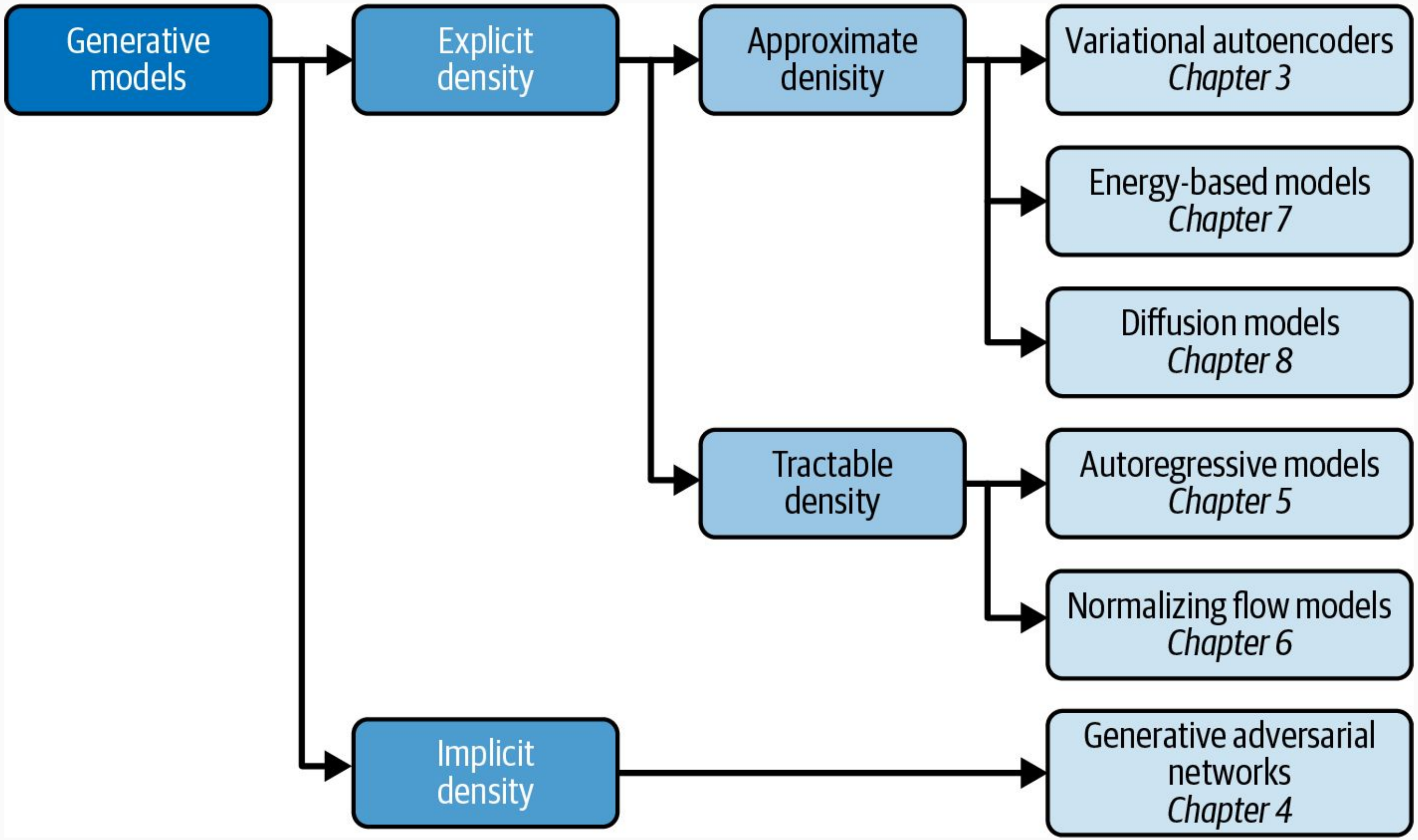
Autoregressive Models

Vijay Raghavan

Family


Generative models
Explicit density
Approximate denisity
Variational autoencoders *Chapter 3*
Energy-based models *Chapter 7*
Diffusion models *Chapter 8*
Tractable density
Autoregressive models *Chapter 5*
Normalizing flow models *Chapter 6*
Implicit density
Generative adversarial networks *Chapter 4*

Autoregressive Models

- Models that simplify the generative modeling problem by treating it as a sequential process.
- Autoregressive models condition predictions on previous values in the sequence, rather than on a latent random variable.
- They attempt to explicitly model the data-generating distribution rather than an approximation of it (as in the case of VAEs).

Agenda

1. Attention Mechanism in action
2. Types of positional embeddings
3. RNN
4. Backpropagation through time (BPTT)
5. RNN and NLP
6. PixelCNN

Attention

Vijay Raghavan

"I went to the bank to withdraw money"

"I went to the bank to withdraw money,"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

"I went to the bank to withdraw money,"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

2. **went and to the bank** - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

"I went to the bank to withdraw money,"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

2. **went and to the bank** - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

3. **to the bank and to withdraw money** - Attention might also be paid between these phrases to recognize that "to the bank" and "to withdraw money" are related purposes, identifying why the subject went to the bank.

"I went to the bank to withdraw money,"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

2. **went and to the bank** - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

3. **to the bank and to withdraw money** - Attention might also be paid between these phrases to recognize that "to the bank" and "to withdraw money" are related purposes, identifying why the subject went to the bank.

4. **withdraw and money** - These words would be closely linked by attention to show the action-object relationship, clarifying what action is being taken with the money.

"I went to the bank to withdraw money,"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

2. **went and to the bank** - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

3. **to the bank and to withdraw money** - Attention might also be paid between these phrases to recognize that "to the bank" and "to withdraw money" are related purposes, identifying why the subject went to the bank.

4. **withdraw and money** - These words would be closely linked by attention to show the action-object relationship, clarifying what action is being taken with the money.

5. **bank and withdraw** - There might be significant attention here to capture the contextual meaning of "bank" as a financial institution where withdrawal happens, rather than any other meaning of "bank."

"I went to the bank to withdraw money"

1. **I and went** - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

2. **went and to the bank** - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

3. **to the bank and to withdraw money** - Attention might also be paid between these phrases to recognize that "to the bank" and "to withdraw money" are related purposes, identifying why the subject went to the bank.

4. **withdraw and money** - These words would be closely linked by attention to show the action-object relationship, clarifying what action is being taken with the money.

5. **bank and withdraw** - There might be significant attention here to capture the contextual meaning of "bank" as a financial institution where withdrawal happens, rather than any other meaning of "bank."

The self-attention mechanism enables the model to focus on different parts of the sentence simultaneously, considering both local and broader context to build a nuanced understanding of the sentence.

Problem Setup & Embedding space

Problem Setup

- Consider an input sequence $x_1, \ldots, x_n$, where each x_i is a vector of dimension d_{model}
- In NLP, x_i could be word/token embeddings
- Stack input vectors into a matrix X of shape (n, d_{model})
- Goal is to compute a new representation z_i for each input element x_i that captures relevant information from the entire sequence

Embeddings - Matrix of shape (n, d_{model})

Token	D1 (Pronoun)	D2 (Verb)	D3 (Preposition)	D4 (Definite Article)	Dn (Noun)
I	0.2212	0.0946	0.5470	0.4375	0.3592
went	0.6540	0.9632	0.6234	0.4815	0.0828
to	0.3356	0.3014	0.8195	0.3616	0.9407
the	0.0885	0.0245	0.3153	0.9066	0.0535
bank	0.0804	0.7651	0.6733	0.3798	0.9523
to	0.3553	0.4538	0.7497	0.5790	0.3795
withdraw	0.1698	0.8050	0.1555	0.3577	0.3163
money	03677	0.4956	0.0420	0.2846	0.9408

Query, Key, and Value Projections

Query, Key, Value

- The *Query (q)* can be thought of as a representation of the current task at hand (e.g., “What word follows *to*?”).
- The *key* vectors *(K)* are representations of each word in the sentence—you can think of these as descriptions of the kinds of prediction tasks that each word can help with.
- The *value* vectors *(V)* are also representations of the words in the sentence—you can think of these as the unweighted contributions of each word.

Step 1: Compute query (Q), key (K), and value (V) matrices using learned projection matrices.

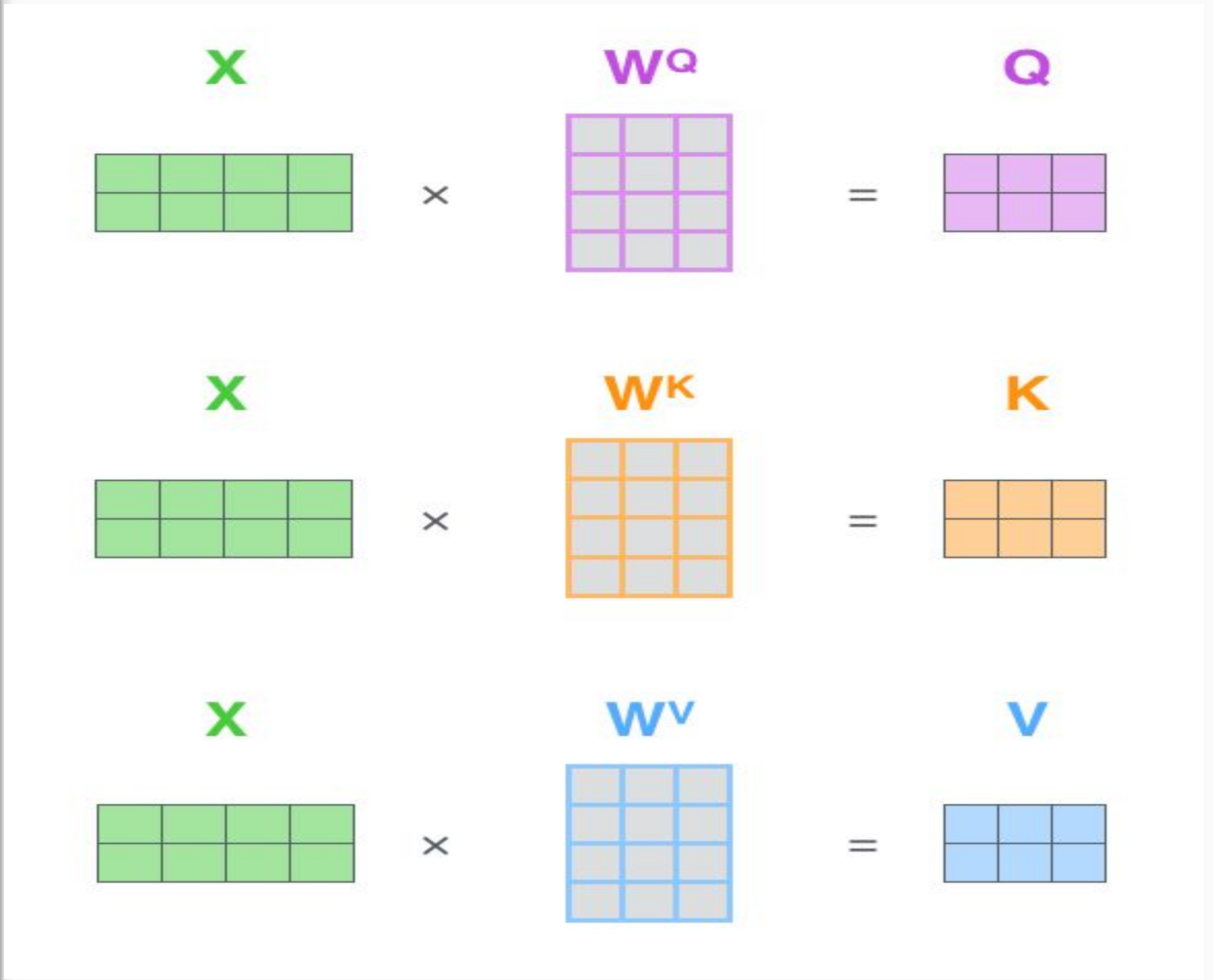
Input sentence: "I went to the bank to withdraw money"

```
Input embeddings (X): [
[ 0.2212, 0.0946, 0.5470, 0.4375, 0.3592], # 'I'
[ 0.6540, 0.4632, 0.6234, 0.4815, 0.0828], # 'went'
[ 0.3356, 0.3014, 0.8195, 0.3616, 0.9407], # 'to'
[ 0.0885, 0.0245, 0.9153, 0.2066, 0.0535], # 'the'
[ 0.8804, 0.7651, 0.6733, 0.3798, 0.6523], # 'bank'
[ 0.3553, 0.4538, 0.7497, 0.5790, 0.3795], # 'to'
[ 0.1698, 0.5050, 0.1555, 0.8577, 0.3163], # 'withdraw'
[ 0.9677, 0.4956, 0.0420, 0.8846, 0.2408]] # 'money'
```

Define three learnable weight matrices: W^Q (query), W^K (key), and W^V (value)

$$W^Q = \begin{bmatrix} 0.1 & -0.2 & 0.3 & 0.4 & -0.1 \\ -0.2 & 0.1 & 0.5 & -0.3 & 0.2 \\ 0.3 & 0.2 & -0.1 & 0.6 & 0.4 \\ -0.4 & 0.5 & 0.2 & -0.1 & 0.3 \\ 0.2 & 0.3 & -0.4 & 0.1 & 0.5 \\ 0.4 & -0.3 & 0.1 & 0.5 & -0.2 \\ -0.1 & 0.4 & -0.5 & 0.2 & 0.3 \\ 0.5 & 0.1 & -0.3 & 0.4 & 0.2 \end{bmatrix} \quad W^K = \begin{bmatrix} 0.1 & 0.2 & -0.3 & 0.4 & 0.1 \\ 0.2 & -0.1 & 0.5 & 0.3 & -0.2 \\ 0.3 & 0.2 & 0.1 & 0.6 & 0.3 \\ 0.4 & 0.5 & 0.2 & 0.1 & -0.4 \\ 0.1 & -0.2 & 0.3 & 0.5 & 0.2 \\ -0.3 & 0.4 & 0.2 & -0.1 & 0.5 \\ 0.2 & 0.3 & -0.4 & 0.1 & 0.4 \\ 0.5 & -0.2 & 0.3 & 0.4 & -0.1 \end{bmatrix} \quad W^V = \begin{bmatrix} 0.1 & 0.2 & 0.3 & -0.4 & 0.2 \\ 0.2 & 0.1 & 0.5 & 0.3 & 0.4 \\ 0.3 & -0.2 & 0.1 & 0.6 & -0.1 \\ 0.4 & 0.5 & -0.2 & 0.1 & 0.3 \\ 0.1 & 0.3 & 0.4 & 0.2 & 0.5 \\ -0.2 & 0.4 & 0.1 & 0.5 & 0.3 \\ 0.3 & -0.1 & 0.5 & 0.2 & -0.4 \\ 0.4 & 0.2 & -0.3 & 0.1 & 0.5 \end{bmatrix}$$

$Q = X \cdot W^Q$, $K = X \cdot W^K$, and $V = X \cdot W^V$



https://jalammar.github.io/illustrated-transformer/

$Q = X \cdot W^Q$, $K = X \cdot W^K$, and $V = X \cdot W^V$

$$Q = \begin{bmatrix} 0.2242 & 0.1739 & 0.2458 & 0.3442 & 0.2701 \\ 0.2303 & 0.0852 & 0.5462 & 0.4000 & 0.0165 \\ 0.5031 & 0.3174 & 0.5037 & 0.7000 & 0.6959 \\ 0.3151 & 0.1561 & 0.1979 & 0.5437 & 0.0204 \\ 0.3861 & 0.1013 & 0.7000 & 0.6140 & 0.2437 \\ 0.3877 & 0.3037 & 0.4035 & 0.5959 & 0.3406 \\ 0.3470 & 0.4098 & 0.0764 & 0.4477 & 0.4706 \\ 0.2415 & 0.1440 & 0.6301 & 0.5205 & -0.0009 \end{bmatrix}$$

$$K = \begin{bmatrix} 0.3804 & 0.1201 & 0.4235 & 0.5457 & 0.2459 \\ 0.5235 & 0.2555 & 0.6974 & 0.7985 & 0.1435 \\ 0.7000 & 0.1909 & 0.8485 & 1.0203 & 0.4841 \\ 0.3143 & 0.0823 & 0.3502 & 0.4820 & 0.1787 \\ 0.8478 & 0.3831 & 0.9538 & 1.1000 & 0.4112 \\ 0.6000 & 0.2502 & 0.6969 & 0.8748 & 0.3449 \\ 0.4772 & 0.2626 & 0.4431 & 0.5693 & 0.1671 \\ 0.6902 & 0.4087 & 0.6252 & 0.8014 & 0.2248 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.2411 & 0.2826 & 0.4303 & 0.2214 & 0.4196 \\ 0.4772 & 0.4701 & 0.7000 & 0.3925 & 0.5180 \\ 0.4772 & 0.4935 & 0.7866 & 0.2476 & 0.7000 \\ 0.2634 & 0.2020 & 0.3951 & 0.2637 & 0.3006 \\ 0.8445 & 0.8134 & 1.1000 & 0.4838 & 0.9519 \\ 0.5112 & 0.5112 & 0.7399 & 0.3656 & 0.6545 \\ 0.3997 & 0.4675 & 0.5147 & 0.1508 & 0.5552 \\ 0.7000 & 0.7898 & 0.8345 & 0.3124 & 0.8206 \end{bmatrix}$$

Step 2: Compute the attention scores using scaled dot-product attention.

Scaled Dot-Product Attention Scores

1. Dot product measures similarity between query and key vectors
    a. If query and key are similar (point in same direction), score will be high
    b. If dissimilar (orthogonal), score will be low
2. Scaling by √dk improves numerical stability
3. Without scaling, dot products can grow large in magnitude for large d_k, pushing softmax into regions with small gradients
4. Result is a matrix of shape (n, n) containing attention scores for each pair of elements

$$\text{Attention Scores} = Q \cdot K^T = \begin{bmatrix} 0.5972 & 0.7679 & 0.9821 & 0.5028 & 1.1207 & 0.8606 & 0.6037 & 0.8108 \\ 0.6848 & 0.9147 & 1.1502 & 0.5765 & 1.2801 & 0.9894 & 0.6702 & 0.9316 \\ 1.2400 & 1.6001 & 2.0718 & 1.0416 & 2.3261 & 1.7805 & 1.2244 & 1.6555 \\ 0.5743 & 0.7437 & 0.9413 & 0.4774 & 1.0542 & 0.8115 & 0.5642 & 0.7672 \\ 1.0546 & 1.3808 & 1.7402 & 0.8742 & 1.9435 & 1.4962 & 1.0276 & 1.4081 \\ 0.9273 & 1.1978 & 1.5328 & 0.7749 & 1.7144 & 1.3168 & 0.9121 & 1.2361 \\ 0.7145 & 0.8924 & 1.1607 & 0.5949 & 1.3006 & 0.9961 & 0.7051 & 0.9409 \\ 0.8844 & 1.1671 & 1.4613 & 0.7328 & 1.6306 & 1.2578 & 0.8577 & 1.1769 \end{bmatrix}$$

Scaled Attention Scores

$$\text{Scaled Attention Scores} = \frac{Q \cdot K^T}{\sqrt{5}} = \begin{bmatrix} 0.2670 & 0.3433 & 0.4393 & 0.2248 & 0.5012 & 0.3849 & 0.2700 & 0.3626 \\ 0.3062 & 0.4090 & 0.5142 & 0.2577 & 0.5724 & 0.4424 & 0.2996 & 0.4165 \\ 0.5545 & 0.7155 & 0.9264 & 0.4658 & 1.0403 & 0.7961 & 0.5474 & 0.7402 \\ 0.2568 & 0.3325 & 0.4209 & 0.2135 & 0.4713 & 0.3629 & 0.2523 & 0.3430 \\ 0.4715 & 0.6175 & 0.7782 & 0.3909 & 0.8690 & 0.6688 & 0.4595 & 0.6295 \\ 0.4146 & 0.5355 & 0.6853 & 0.3465 & 0.7666 & 0.5888 & 0.4078 & 0.5527 \\ 0.3195 & 0.3990 & 0.5190 & 0.2660 & 0.5815 & 0.4454 & 0.3153 & 0.4207 \\ 0.3954 & 0.5218 & 0.6534 & 0.3277 & 0.7291 & 0.5624 & 0.3835 & 0.5262 \end{bmatrix}$$

Step 3: Convert attention scores to attention weights using softmax.

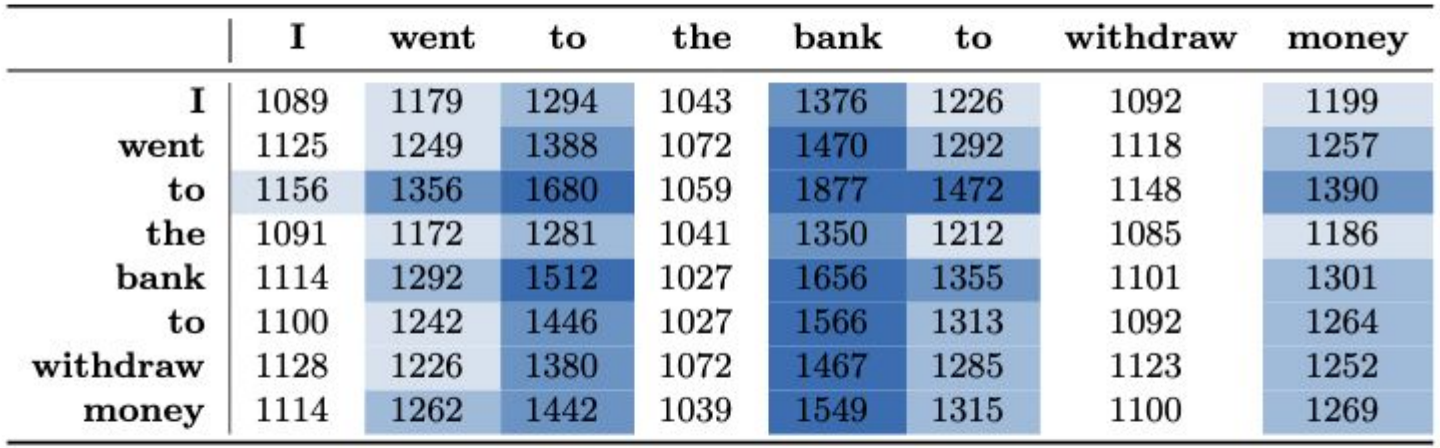
Softmax Attention Weights

- Convert attention scores to weights using softmax function
- $w_{i\square} = \exp(\text{score}(x_i, x_\square)) / \sum_\square \exp(\text{score}(x_i, x_\square))$
- Properties of softmax weights:
  - Non-negative and sum to 1 across each row
  - Higher values indicate more attention
  - Can be interpreted as a probability distribution over the inputs
- Result is an attention weight matrix of shape (n, n)

Weights = softmax(scores)

$$\text{Softmax Attention Weights} = \begin{bmatrix} 0.1089 & 0.1179 & 0.1294 & 0.1043 & 0.1376 & 0.1226 & 0.1092 & 0.1199 \\ 0.1125 & 0.1249 & 0.1388 & 0.1072 & 0.1470 & 0.1292 & 0.1118 & 0.1257 \\ 0.1156 & 0.1356 & 0.1680 & 0.1059 & 0.1877 & 0.1472 & 0.1148 & 0.1390 \\ 0.1091 & 0.1172 & 0.1281 & 0.1041 & 0.1350 & 0.1212 & 0.1085 & 0.1186 \\ 0.1114 & 0.1292 & 0.1512 & 0.1027 & 0.1656 & 0.1355 & 0.1101 & 0.1301 \\ 0.1100 & 0.1242 & 0.1446 & 0.1027 & 0.1566 & 0.1313 & 0.1092 & 0.1264 \\ 0.1128 & 0.1226 & 0.1380 & 0.1072 & 0.1467 & 0.1285 & 0.1123 & 0.1252 \\ 0.1114 & 0.1262 & 0.1442 & 0.1039 & 0.1549 & 0.1315 & 0.1100 & 0.1269 \end{bmatrix}$$

"I went to the bank to withdraw money"



	I	went	to	the	bank	to	withdraw	money
I	1089	1179	1294	1043	1376	1226	1092	1199
went	1125	1249	1388	1072	1470	1292	1118	1257
to	1156	1356	1680	1059	1877	1472	1148	1390
the	1091	1172	1281	1041	1350	1212	1085	1186
bank	1114	1292	1512	1027	1656	1355	1101	1301
to	1100	1242	1446	1027	1566	1313	1092	1264
withdraw	1128	1226	1380	1072	1467	1285	1123	1252
money	1114	1262	1442	1039	1549	1315	1100	1269

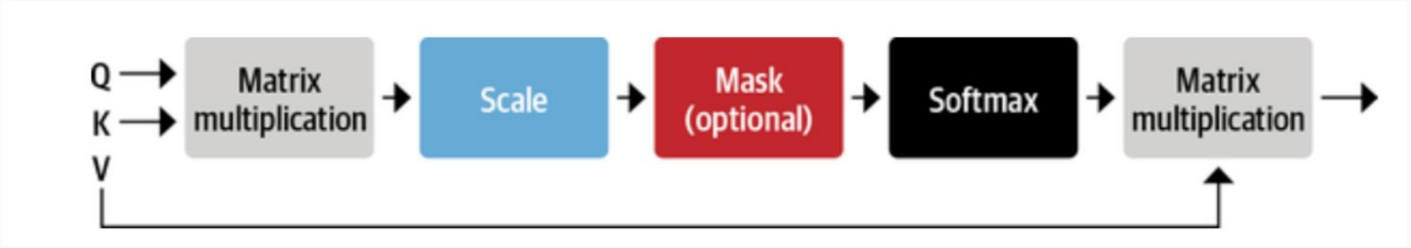
Table 1: Attention weights with color coding (values multiplied by 10000 for readability)

Step 4: Compute the output vectors by taking the weighted sum of the value vectors.

Weighted Aggregation to Compute Outputs

- Weighted sum of value vectors, with weights given by attention
- Can be expressed compactly in matrix form:
  - $Z = W \cdot V$, where Z has shape (n, d_k)
- Each output z_i aggregates information from the entire input sequence, with a focus on the most relevant elements as determined by the attention weights
- Parallel computation across all outputs via matrix multiplication

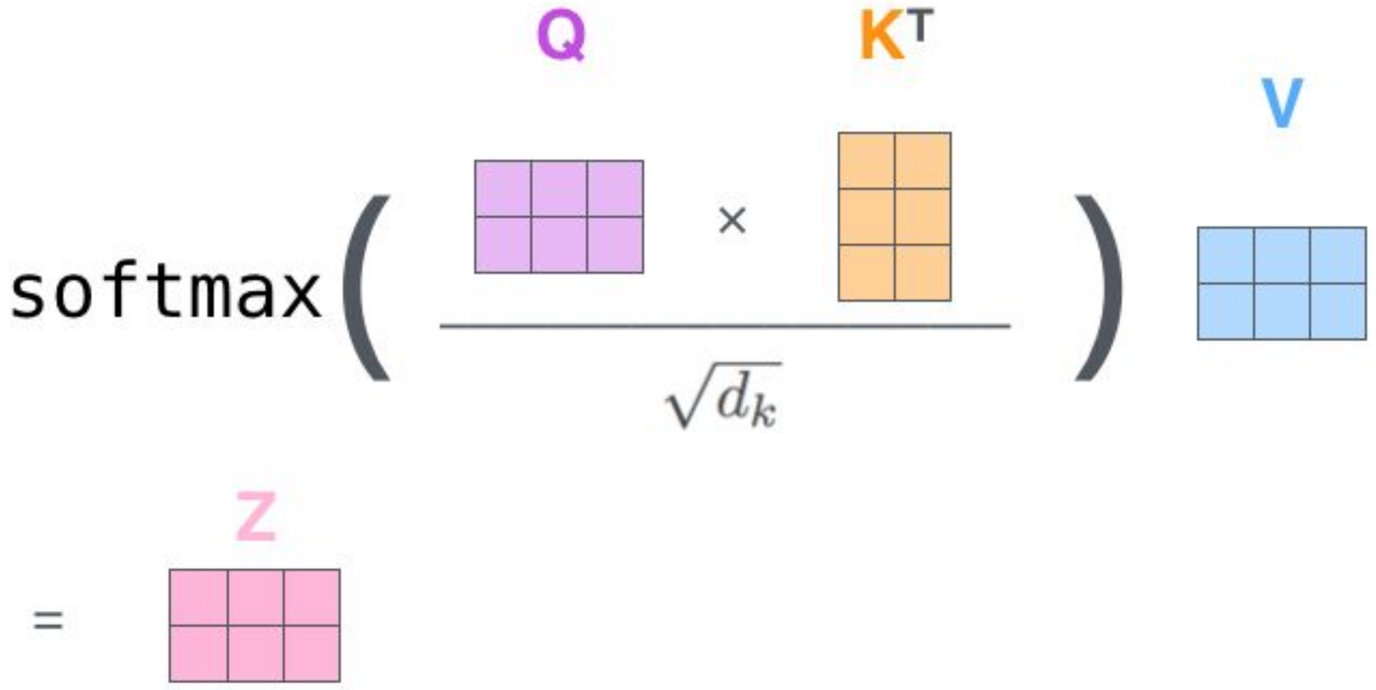
Operations in scaled dot-product attention



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

Tunstall, L., Von Werra, L., & Wolf, T. (2022). *Natural language processing with transformers*. " O'Reilly Media, Inc.".

The self-attention calculation in matrix form

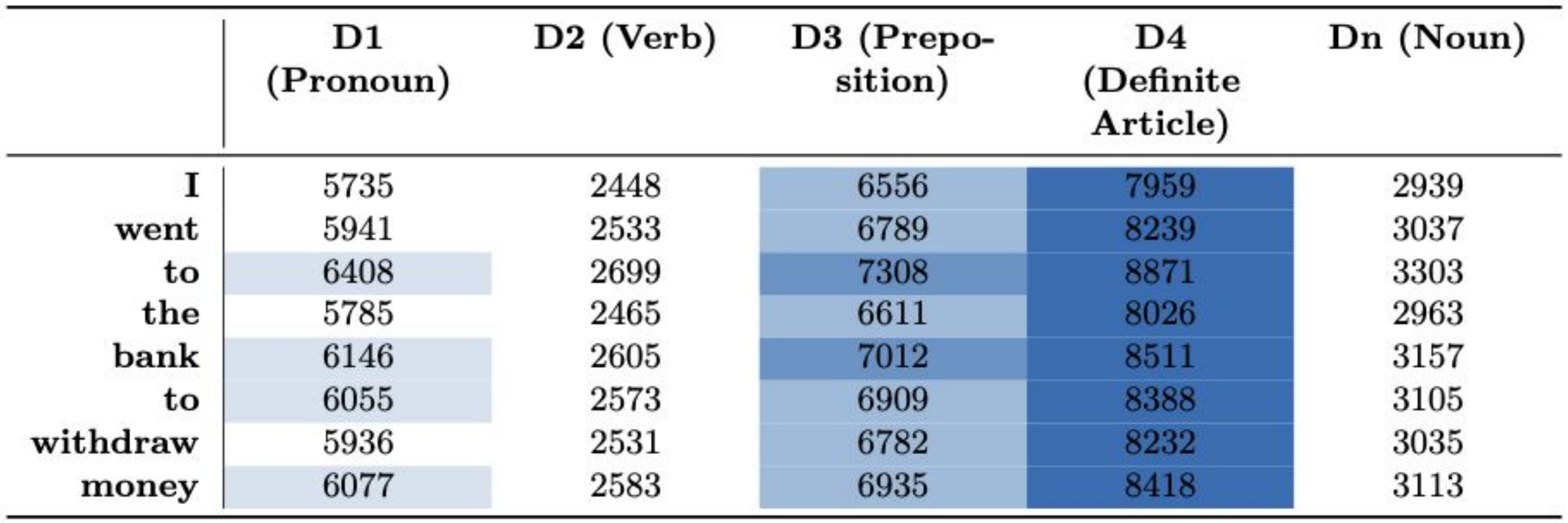


$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

Output (Z) = weights · V

$$\text{Attention Output} = \text{Attention Weights} \cdot K = \begin{bmatrix} 0.5735 & 0.2448 & 0.6556 & 0.7959 & 0.2939 \\ 0.5941 & 0.2533 & 0.6789 & 0.8239 & 0.3037 \\ 0.6408 & 0.2699 & 0.7308 & 0.8871 & 0.3303 \\ 0.5785 & 0.2465 & 0.6611 & 0.8026 & 0.2963 \\ 0.6146 & 0.2605 & 0.7012 & 0.8511 & 0.3157 \\ 0.6055 & 0.2573 & 0.6909 & 0.8388 & 0.3105 \\ 0.5936 & 0.2531 & 0.6782 & 0.8232 & 0.3035 \\ 0.6077 & 0.2583 & 0.6935 & 0.8418 & 0.3113 \end{bmatrix}$$

"I went to the bank to withdraw money"

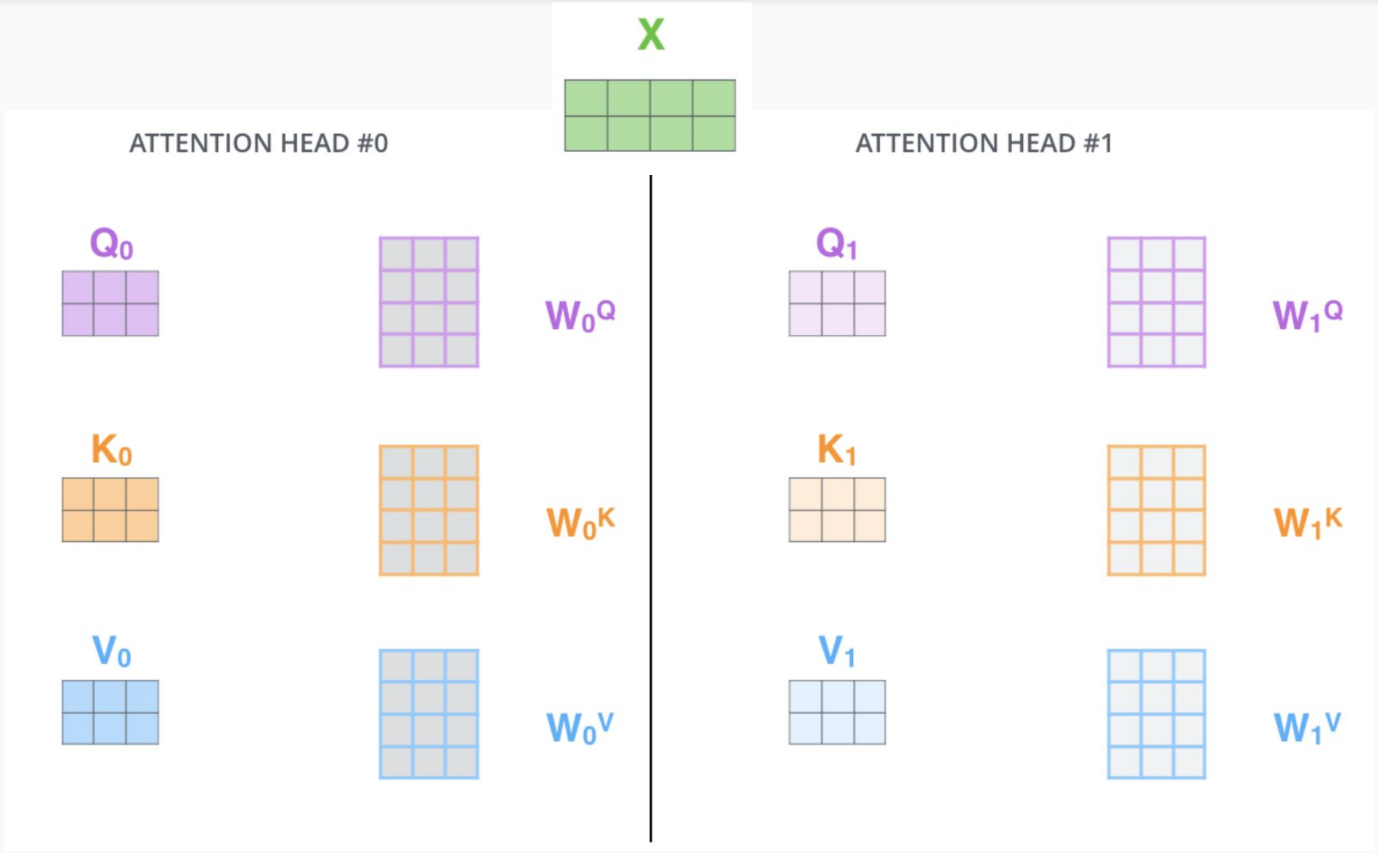


	D1 (Pronoun)	D2 (Verb)	D3 (Preposition)	D4 (Definite Article)	Dn (Noun)
I	5735	2448	6556	7959	2939
went	5941	2533	6789	8239	3037
to	6408	2699	7308	8871	3303
the	5785	2465	6611	8026	2963
bank	6146	2605	7012	8511	3157
to	6055	2573	6909	8388	3105
withdraw	5936	2531	6782	8232	3035
money	6077	2583	6935	8418	3113

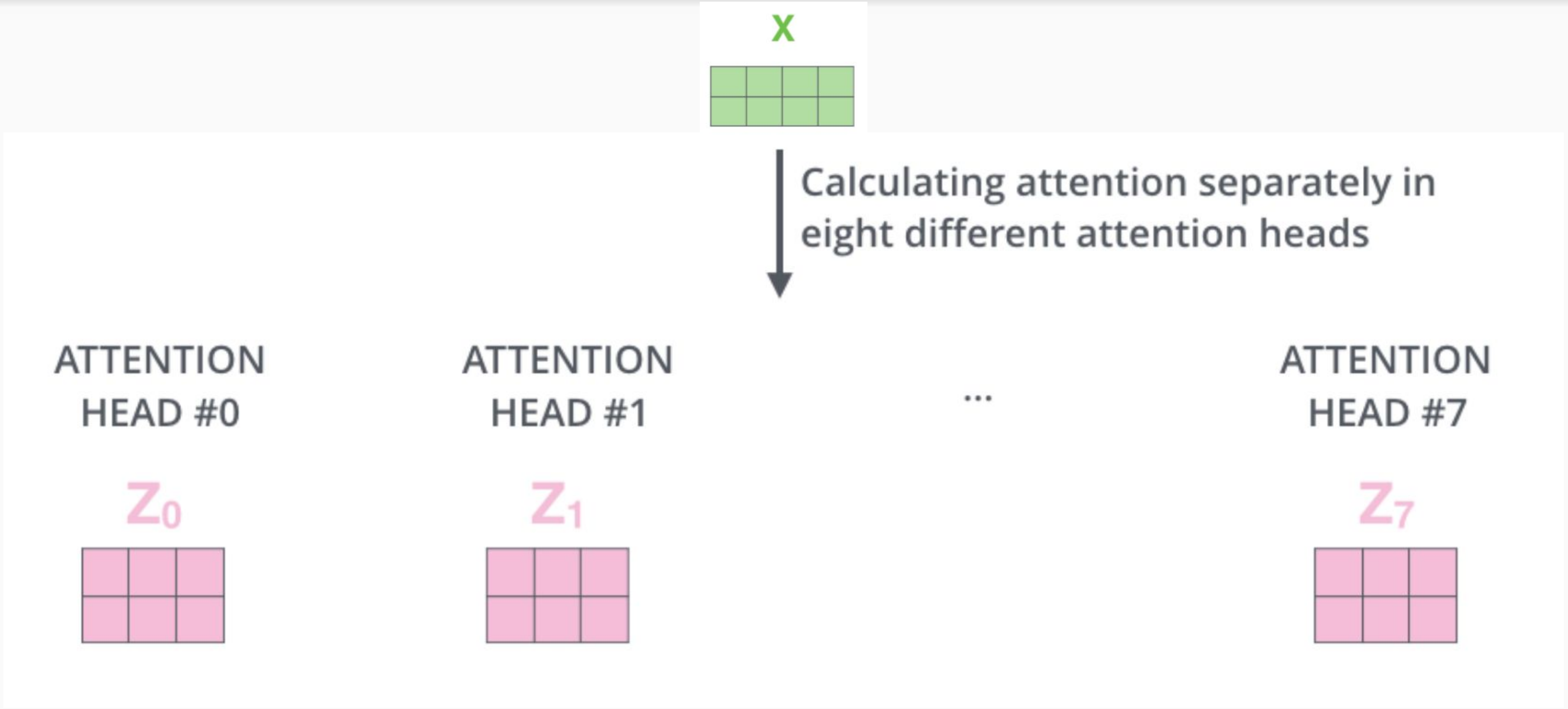
Table 1: Attention Output with color coding and semantic dimension labels (values multiplied by 10000 for readability)

Multi-head Attention

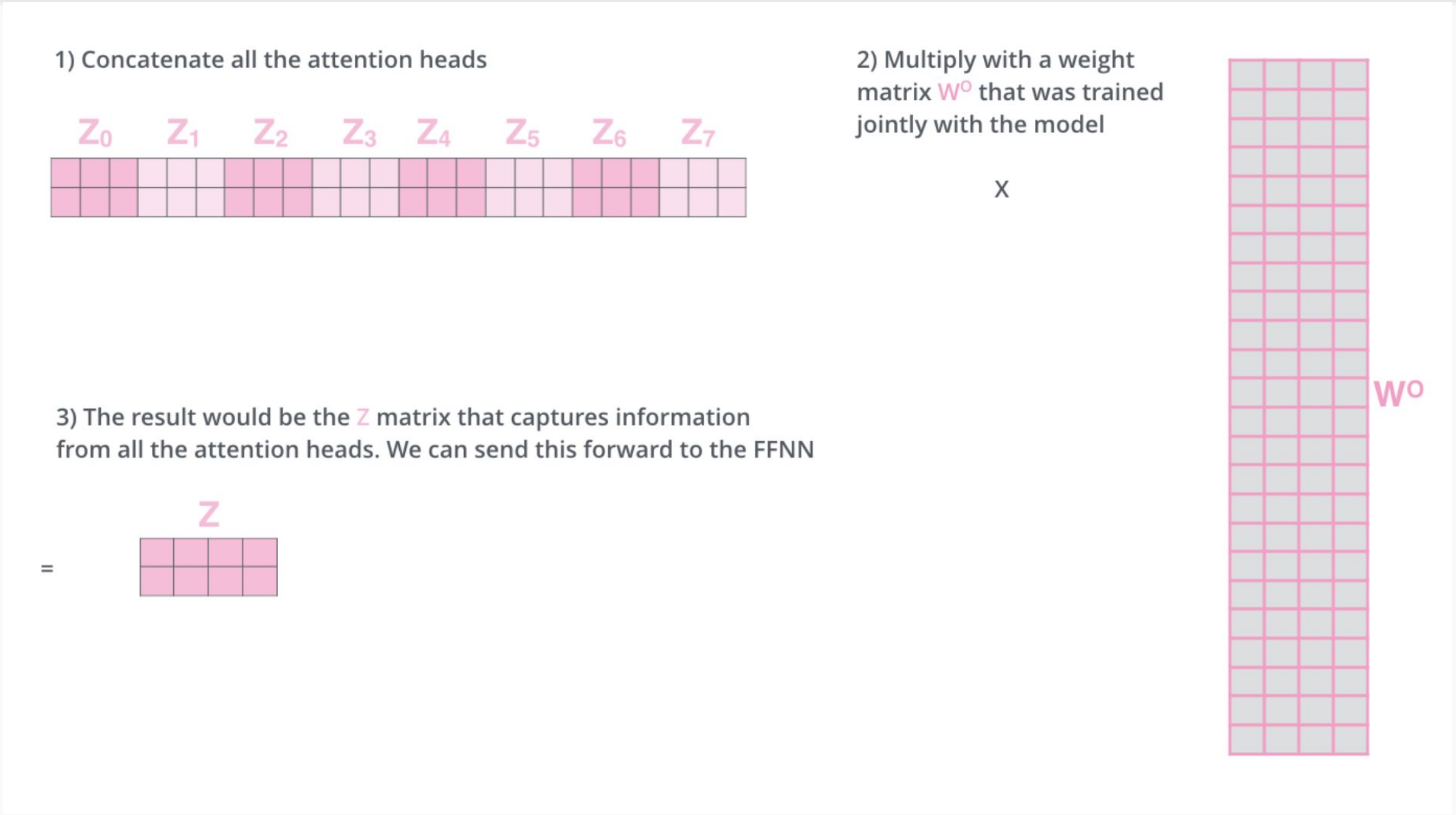
Separate Q/K/V weight matrices for each head



Results from multiple heads


X
Calculating attention separately in eight different attention heads
ATTENTION HEAD #0
ATTENTION HEAD #1
...
ATTENTION HEAD #7
Z0
Z1
Z7

Contatination



1) Concatenate all the attention heads

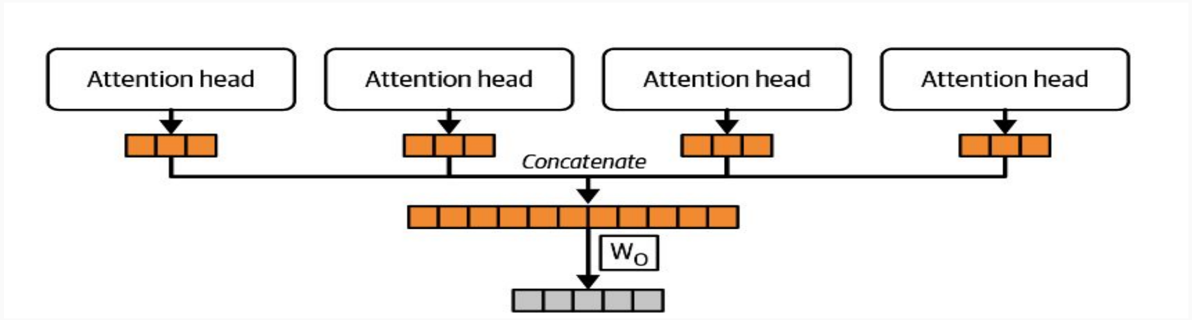
2) Multiply with a weight matrix W^O that was trained jointly with the model

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

Multihead Attention and Concatenation

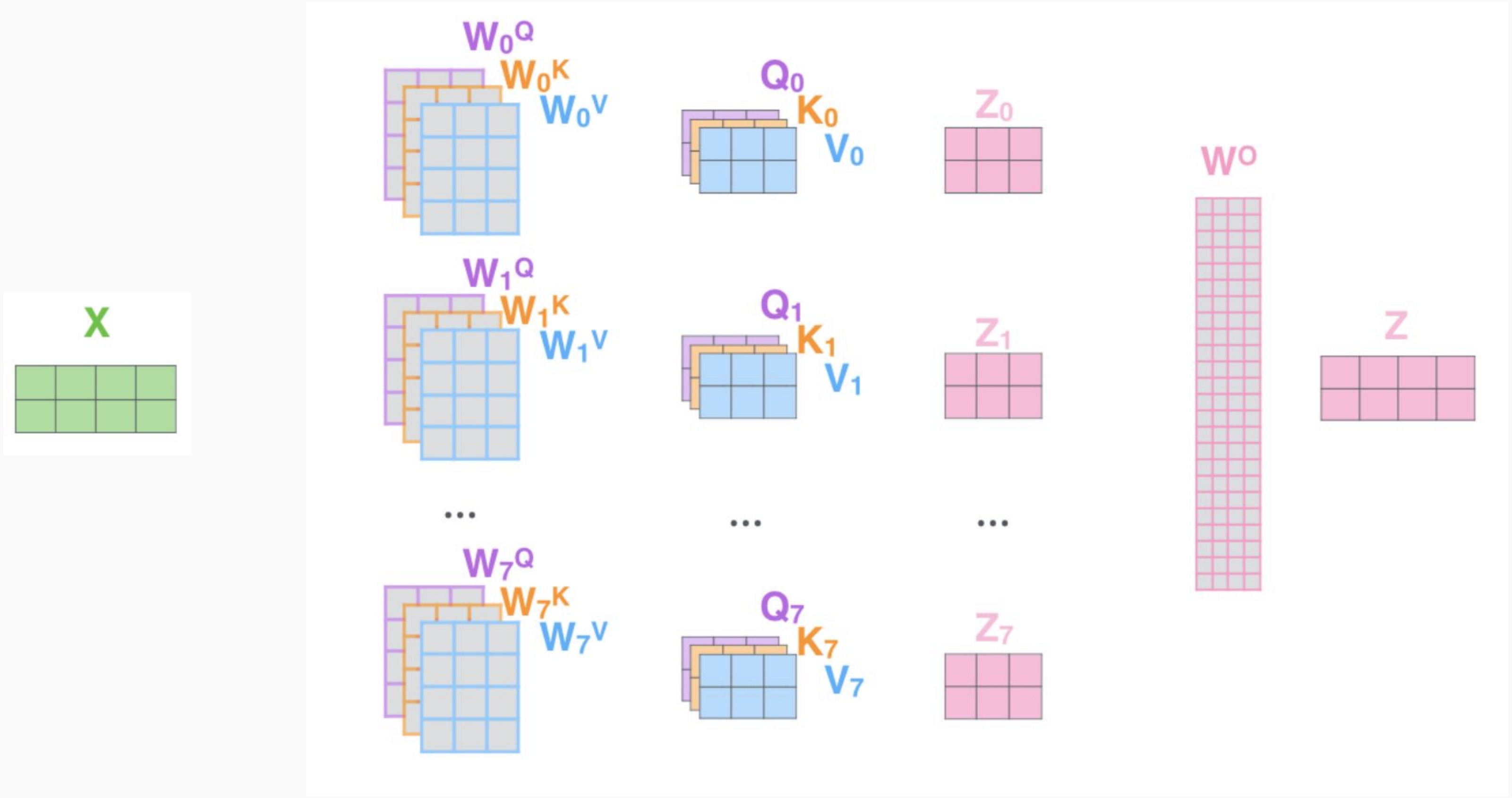
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},\ldots,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

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



Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.".

Recap


X
W_0^Q
W_0^K
W_0^V
W_1^Q
W_1^K
W_1^V
...
W_7^Q
W_7^K
W_7^V
Q_0
K_0
V_0
Q_1
K_1
V_1
...
Q_7
K_7
V_7
Z_0
Z_1
...
Z_7
W^O
Z

"I went to the bank to withdraw money"

Head # 1. *I and went* - Self-attention might link "I" strongly to "went" to establish the subject-verb agreement and understand who is performing the action.

Head # 2. *went and to the bank* - The mechanism would likely focus on connecting "went" to "to the bank" to capture the destination of the action, enhancing the understanding of where the subject went.

Head # 3. *to the bank and to withdraw money* - Attention might also be paid between these phrases to recognize that "to the bank" and "to withdraw money" are related purposes, identifying why the subject went to the bank.

Head # 4. *withdraw and money* - These words would be closely linked by attention to show the action-object relationship, clarifying what action is being taken with the money.

Head # 5. *bank and withdraw* - There might be significant attention here to capture the contextual meaning of "bank" as a financial institution where withdrawal happens, rather than any other meaning of "bank."

Positional Embeddings

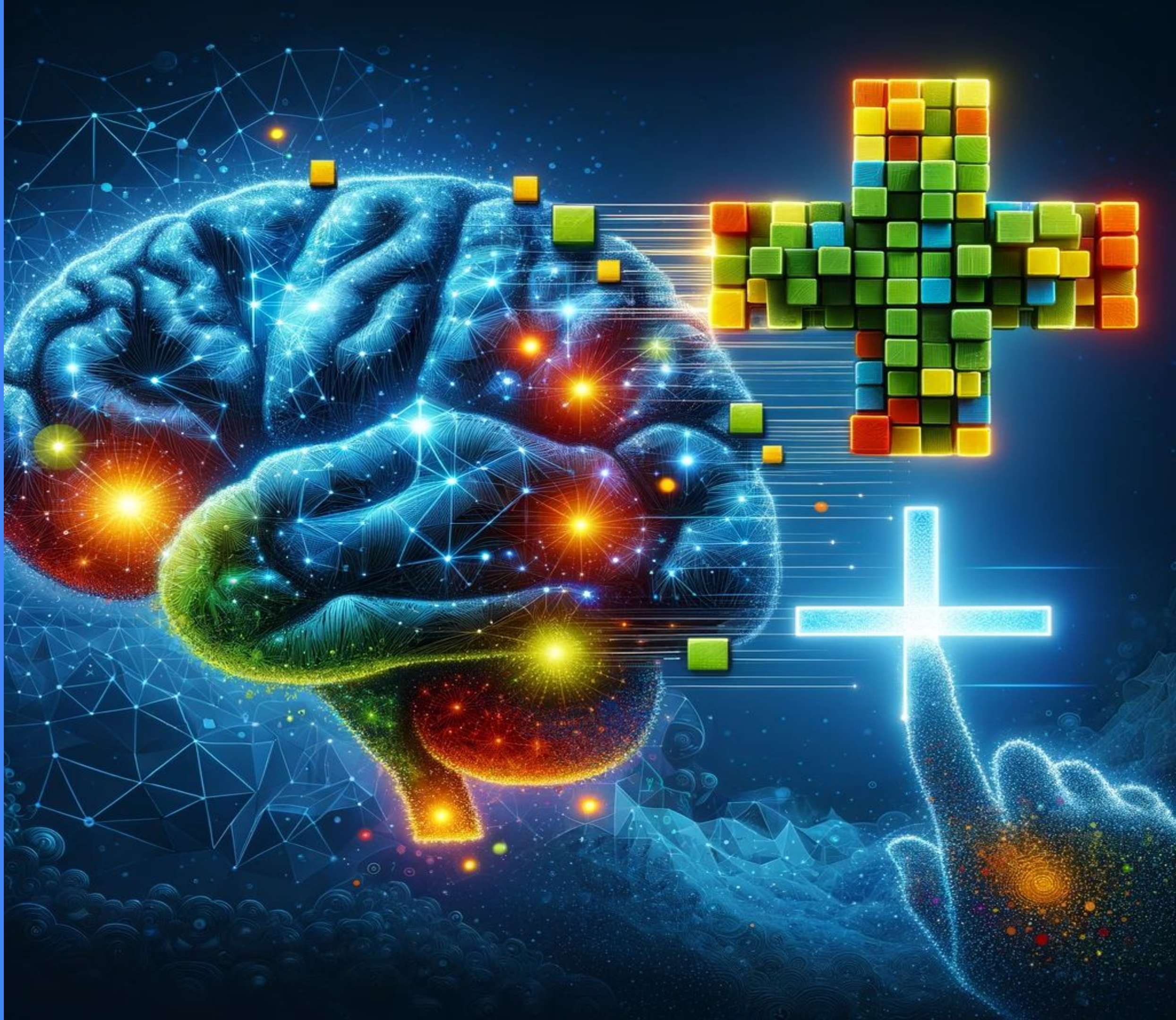
Vijay Raghavan

Mechanics of Positional Embeddings

1. Should be able to distinctly denote the position of the token.
2. Higher magnitude will lead to destruction of semantic meaning between tokens.
3. Balance between mixing up and higher computational complexity.

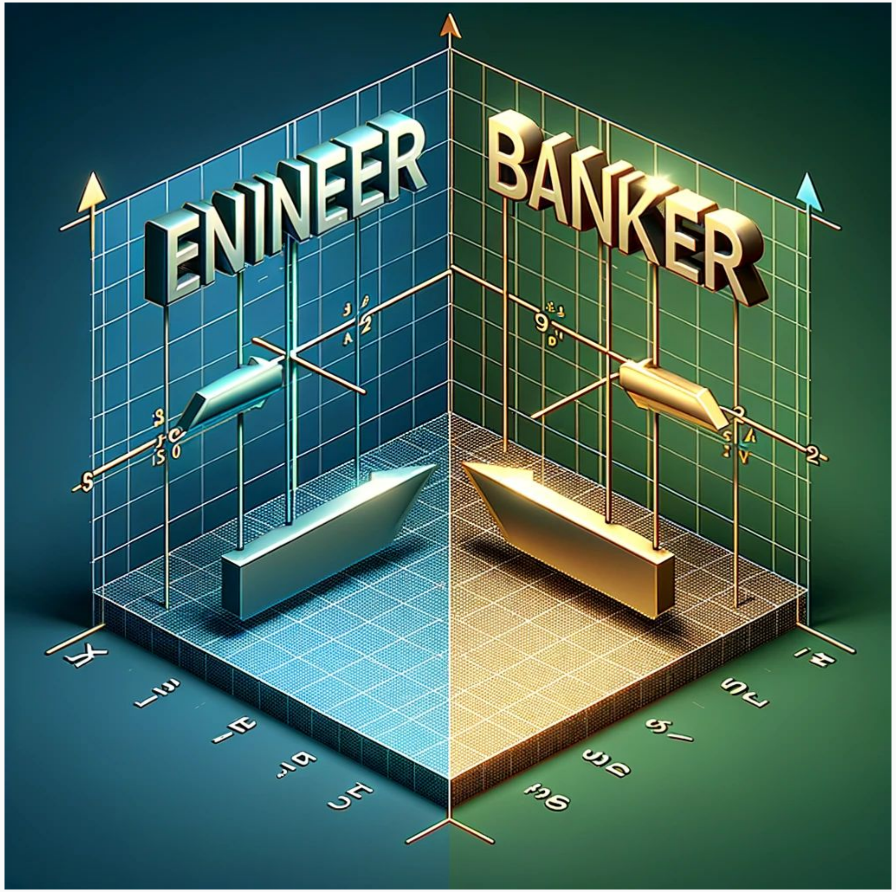
Processing Positional embeddings

1. Additive
2. Concatenation
3. Learned



1. Additive

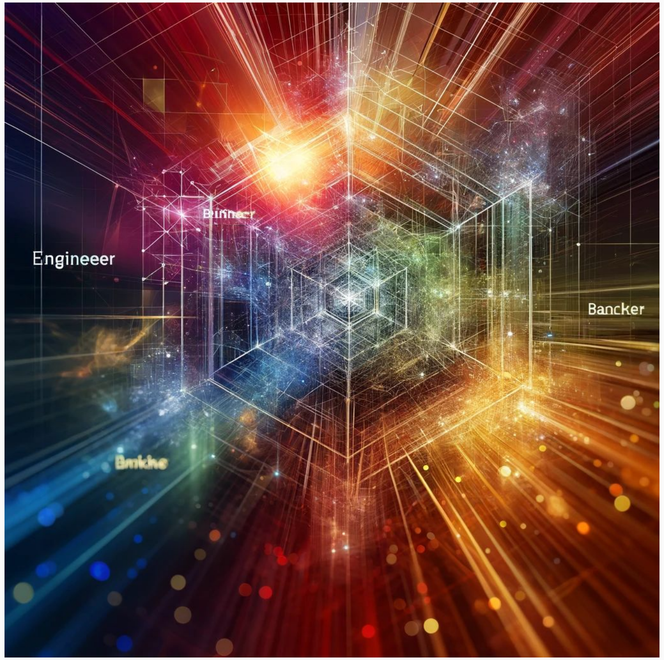
	Engineer	and	Banker
Token Vector	**-0.9748**	**0.4908**	**-0.7905**
Position Vector	**0.1566**	**-0.4797**	**0.1123**
	+	+	+
Additive	**-0.8182**	**0.0112**	**-0.6782**



2. Concatenation

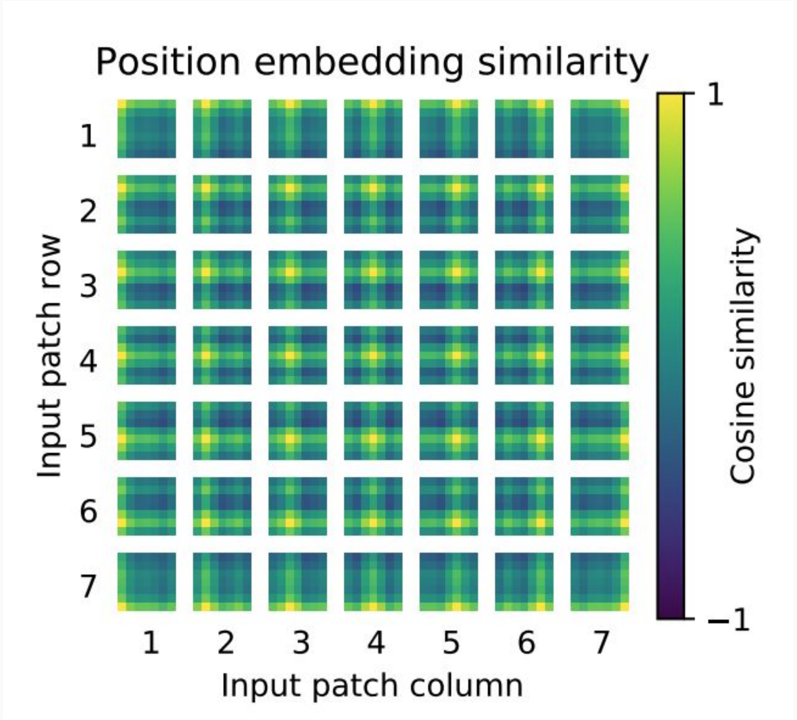
	Engineer	and	Banker
Token Vector	**-0.9748**	**0.4908**	**-0.7905**
Position Vector	**0.1566**	**-0.4797**	**0.1123**

Concat	**-0.9748**	**0.4908**	**-0.7905**	**0.1566**	**-0.4797**	**0.1123**



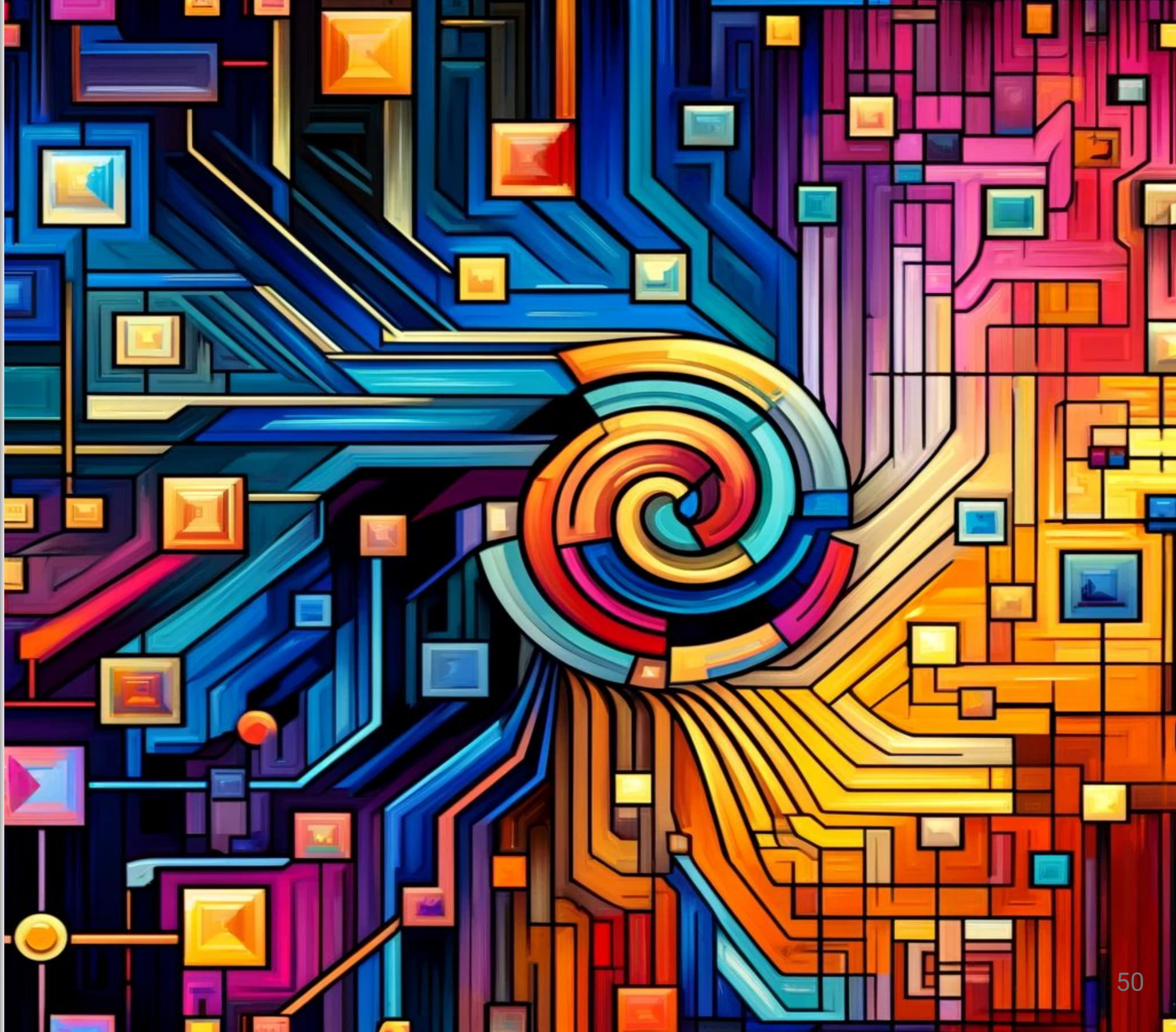
3. Learnt

1. Image Patch Tokenization:
   - The image is divided into fixed-size patches (e.g., 16x16 pixels).
   - Each patch is flattened and projected to a fixed-dimensional vector, forming a sequence of patch embeddings.
2. Adding Positional Embeddings:
   - Positional embeddings, which are vectors of the same dimension as patch embeddings, are added to each patch embedding.
   - These embeddings are initialized as trainable parameters.
3. Learning Process:
   - During training, positional embeddings are learned alongside other model parameters to minimize the loss function.



Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An Image is Worth 16× 16 Words: Transformers for Image Recognition at Scale.

Recurrent neural network (RNN)

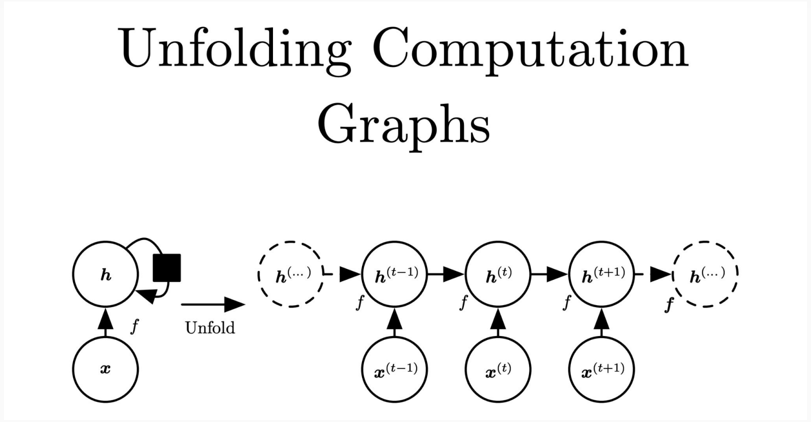


RNN in cubist influenced style

Recurrent Neural Networks

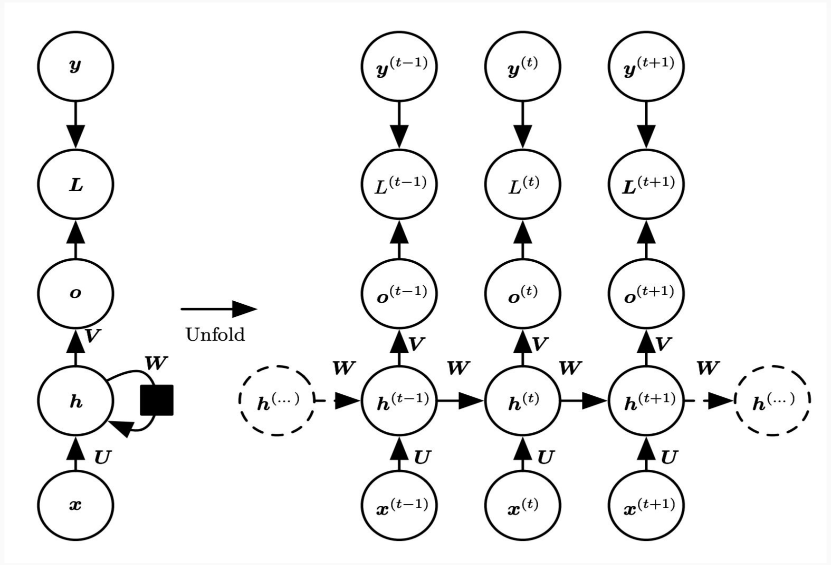
- Process text sequences one word at a time
- Hidden state vector summarizes previous context
- Updated by incorporating vector for current input word
- Tries to represent all potentially relevant context

Recurrent Neural Networks



- Network processes information from the input ***x*** by incorporating it into the state ***h*** that is passed forward through time.
- The black square indicates a delay of a single time step.
- Network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Training Recurrent Hidden Units



$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}),
\end{aligned}$$

- Computational graph to compute the training loss of a recurrent network that maps an input sequence of *x* values to a corresponding sequence of output *o* values.
- Loss *L* measures how far each *o* is from the corresponding training target *y*.
- The loss *L* internally computes *y = softmax(o)* and compares this to the target *y*.
- The RNN has input to hidden connections parametrized by a weight matrix *U* hidden-to-hidden recurrent connections parametrized by a weight matrix *W*, and hidden-to-output connections parametrized by a weight matrix ***V*.**

Introduction to Long Short-Term Memory Networks (LSTMs)

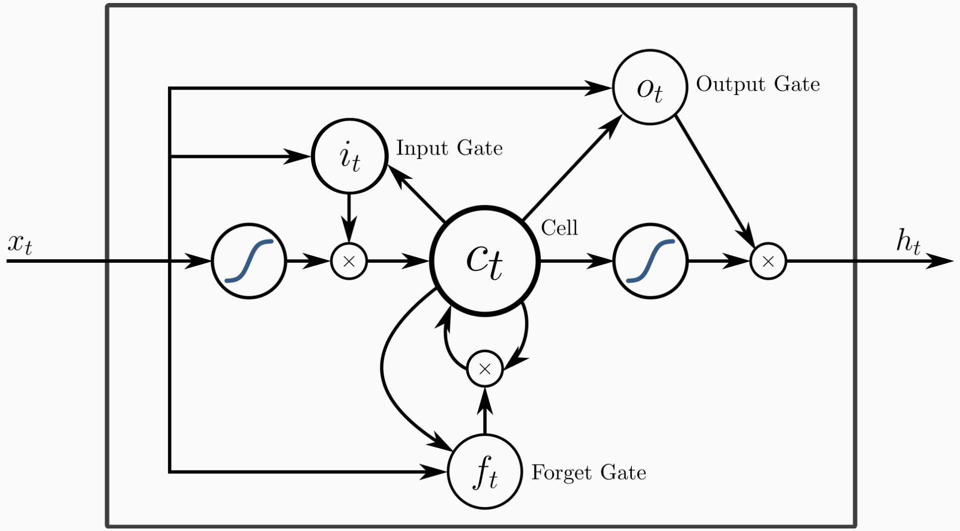
- LSTMs are a type of recurrent neural network (RNN) designed to handle sequential data
- RNNs process data through a recurrent layer (cell) that takes its own output from the previous timestep as input for the next timestep
- Early RNNs used simple tanh activation to scale information between -1 and 1, but suffered from the vanishing gradient problem, limiting their ability to learn from long sequences
- LSTMs, introduced by Hochreiter and Schmidhuber in 1997, address the vanishing gradient problem and can effectively learn from sequences hundreds of timesteps long
- Since then, LSTMs have been improved and widely adopted, with variations like gated recurrent units (GRUs) available in deep learning frameworks like Keras

Applications and Advantages of LSTMs

- LSTMs have been successfully applied to various problems involving sequential data, such as:
    - Time series forecasting: Predicting future values based on historical data
    - Sentiment analysis: Determining the sentiment (positive, negative, neutral) of text data
    - Audio classification: Classifying audio samples into predefined categories
    - Text generation: Creating new text based on learned patterns from training data
- Advantages of LSTMs:
    - Ability to learn long-term dependencies in sequential data
    - Robustness to the vanishing gradient problem, allowing for effective training on long sequences
    - Flexible architecture that can be adapted and improved for specific tasks
    - Wide availability in deep learning frameworks, making them accessible to researchers and practitioners

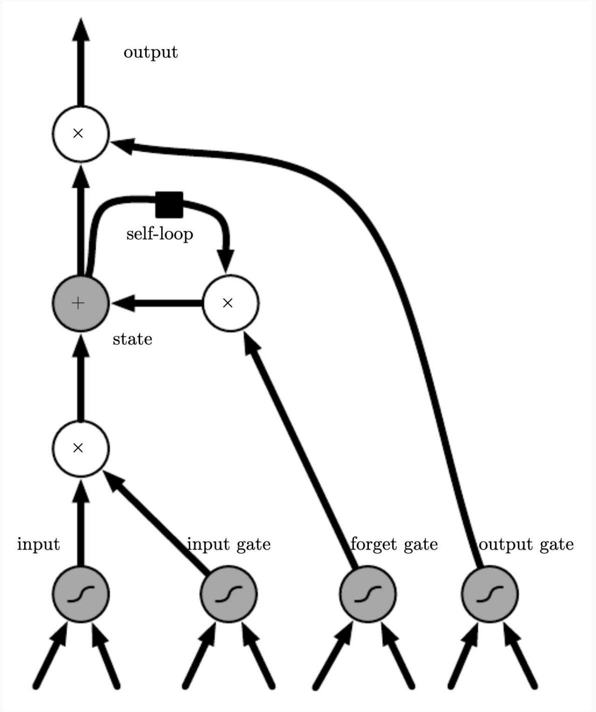
Long short-term memory (LSTM) model (Hochreiter and Schmidhuber, 1997).

1. Forget Gate: Decides what information to discard from the cell state.
2. Input Gate: Decides what new information to store in the cell state.
3. Output Gate: Decides what information to output from the cell state.



https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:Peephole_Long_Short-Term_Memory.svg

Long short-term memory (LSTM) model



- Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks.
- An input feature is computed with a regular artificial neuron unit.
- Its value can be accumulated into the state if the sigmoidal input gate allows it.
- The state unit has a linear self-loop whose weight is controlled by the forget gate.
- The output of the cell can be shut off by the output gate.
- All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity.
- The state unit can also be used as an extra input to the gating units.
- The black square indicates a delay of a single time step.

Forget gate, Input gate and Cell State

- LSTM recurrent networks have “LSTM cells” that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.
- Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information.
- The most important component is the state unit ***s(t)*** , which has a linear self-loop the self-loop weight (or the associated time constant) is controlled by a forget gate unit ***f(t)*** (for time step ***t***), which sets this weight to a value between ***0*** and ***1*** via a sigmoid unit.
- ***x(t)*** is the current input vector and ***h(t)*** is the current hidden layer vector, containing the outputs of all the LSTM cells, and ***b, U, W*** are respectively biases, input weights, and recurrent weights for the forget gates (***f***). The external input gate unit ***g(t)*** is computed similarly to the forget gate with a sigmoid unit to obtain a gating value between ***0*** and ***1***, but with its own parameters:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight ***f(t), b, U*** and ***W*** respectively denote the biases, input weights, and recurrentweights into the LSTM cell.

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

Output gate

The output ***h*(t)** of the LSTM cell can also be shut off, via the output gate ***q*(t)**, which also uses a sigmoid unit for gating which has parameters ***b*o**, ***U*o**, ***W*o** for its biases, input weights and recurrent weights.

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)},$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

Gated recurrent units (GRU)

A single gating unit simultaneously controls the forgetting factor and the decision to update the state unit.

Gated recurrent units, or GRUs

- In this formulation, when the reset gate is close to 0, the hidden state is forced to ignore the previous hidden state and reset with the current input only.
- This effectively allows the hidden state to drop any information that is found to be irrelevant later in the future, thus, allowing a more compact representation.
- On the other hand, the update gate controls how much information from the previous hidden state will carry over to the current hidden state.

The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right)$$

Where ***u*** stands for “update” gate and ***r*** for “reset” gate.

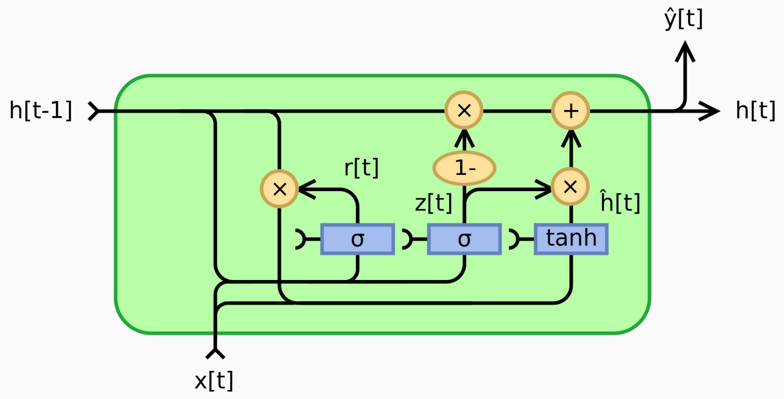
$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right)$$

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right)$$

Cho K, Van Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078. 2014 Jun 3.

Goodfellow I, Bengio Y, Courville A. Deep learning. MIT press; 2016 Nov 10.

Update and Reset



- The reset and update gates can individually “ignore” parts of the state vector.
- The update gates a can linearly gate any dimension, thus choosing to copy it at one extreme of the sigmoid or completely ignore it at the other extreme by replacing it with the new “target state” value.
- The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship between past state and future state.

Backpropagation Through Time (BPTT)

Backpropagation in Standard Neural Networks

- Forward Pass: Data flows through the network layers, and the network produces an output.
- Loss Calculation: The output is compared to the true value to calculate the error (loss).
- Backward Pass: The error is propagated back through the network, and the gradients of the error with respect to the network weights are computed.
- Weight Update: The weights are updated in the direction that reduces the error, typically using gradient descent.

Backpropagation Through Time (BPTT)

- Unrolling the Network: BPTT begins by "unrolling" the RNN across time. This means representing each time step as a separate layer in a large, deep network.
- Forward Pass: Like in standard backpropagation, data is passed forward through the network. In an RNN, this involves processing the entire sequence, time step by time step.
- Loss Calculation: The loss is computed, often at each time step or only at the final step, depending on the task.
- Backward Pass: This is where BPTT diverges from standard backpropagation. The error is propagated back through each time step. Gradients are calculated not just with respect to the weights at each time step, but also with respect to the recurrent connections.
- Accumulating Gradients: Since the same weights are used at each time step, BPTT involves summing the gradients across all time steps.
- Weight Update: Finally, the weights are updated using the accumulated gradients.

RNN - special property

- The unrolled RNN is essentially a feedforward neural network with the special property that the same parameters are repeated throughout the unrolled network, appearing at each time step.
- Then, just as in any feedforward neural network, we can apply the chain rule, backpropagating gradients through the unrolled net.
- The gradient with respect to each parameter must be summed across all places that the parameter occurs in the unrolled net.

BPTT - Network Unrolling

- An RNN is essentially a network that shares weights across different time steps.
- For BPTT, this network is conceptualized as a deep feedforward network where each layer corresponds to a time step in the sequence.
- Imagine a sequence with T time steps. The RNN is "unrolled" into T layers, each representing the network at a different time step.

BPTT - Forward Pass

- Input: At each time step t, the RNN receives an input x_t and the hidden state from the previous time step h_{t-1}
- Computation: The network computes the current hidden state h_t and the output y_t
- Propagation: This process continues, with each step depending on the previous one, until the end of the sequence.

BPTT - Loss Calculation

- The loss function depends on the specific task (e.g., classification, regression).
- It might be computed at each time step or only at the end of the sequence.
- The total loss is often the sum or average of the losses at each relevant time step.

BPTT - Backward Pass

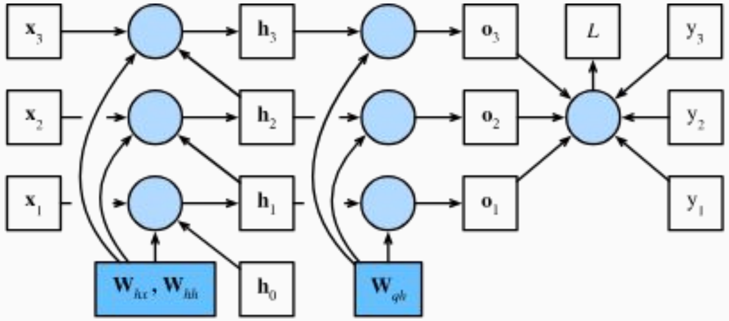
- Gradient Calculation: The gradient of the total loss is calculated with respect to each parameter of the network. This involves applying the chain rule of calculus.
- Through Time: The key aspect is that the gradient computation for each time step includes terms from future time steps. This is because the output at each time step depends on the hidden state, which is influenced by previous time steps.
- Accumulation: Since the network parameters are shared across time steps, gradients computed at each time step are accumulated.

BPTT - Weight Update

- After computing gradients for all time steps, the weights are updated using an optimization algorithm like SGD (Stochastic Gradient Descent).
- This update is based on the accumulated gradients.

BPTT - Computational Graph

- In order to visualize the dependencies among model variables and parameters during computation of the RNN, we can draw a computational graph for the model
- For example, the computation of the hidden states of time step 3, h_3 depends on the model parameters W_{hx} and W_{hh}
- The hidden state of the previous time step h_2, and the input of the current time step x_3

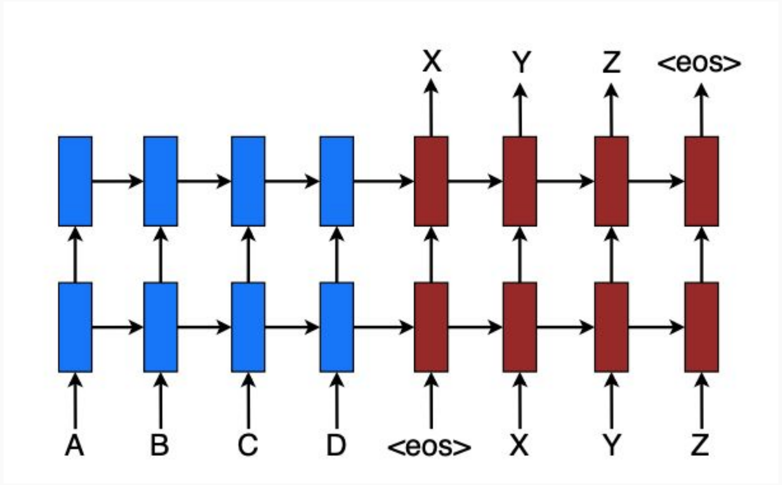


RNN and NLP

RNN and Attention - Luong 2015

- Stacking recurrent architecture for translating a source sequence A B C D into a target sequence X Y Z.
- Attentional mechanism to improve neural machine translation (NMT) by selectively focusing on parts of the source sentence during translation.
    - Global approach which always attends to all source words
    - Local one that only looks at a subset of source words at a time.

Neural machine translation



<eos> marks the end of a sentence.

Attention

Global Attention

- Tries to attend to all source positions per target word
- Creates a variable-length alignment vector to score relevance of each source state
- Context vector is a weighted average of source states per alignment vector
- Tends to have "blurrier" alignments but captures global context
- Tested different alignment scoring functions: Dot product worked best

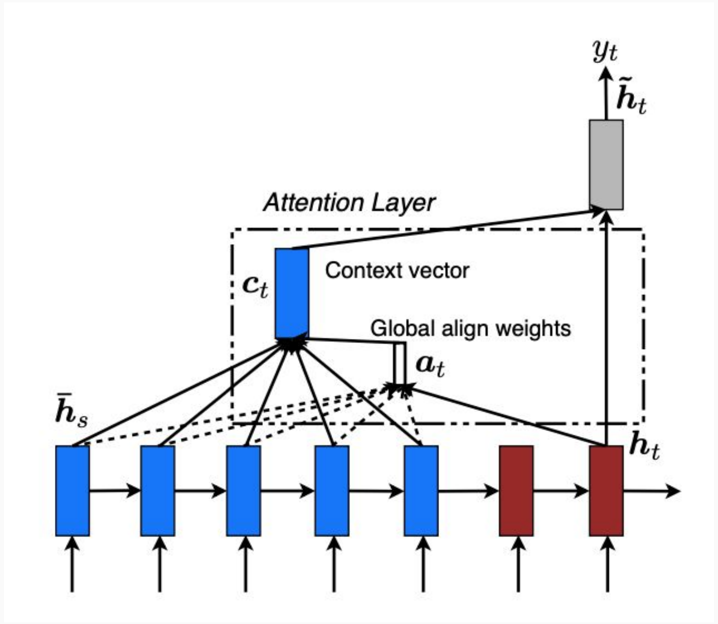
Local Attention

- Predicts a single aligned source position per target word
- Attends to a small fixed-size window around predicted position
- Context vector comes only from local window
- Gives much sharper alignments but limited context
- Monotonic (predict positional increment) and predictive (learn positional model) variants
- Predictive alignment using general scoring function works best

Global attention

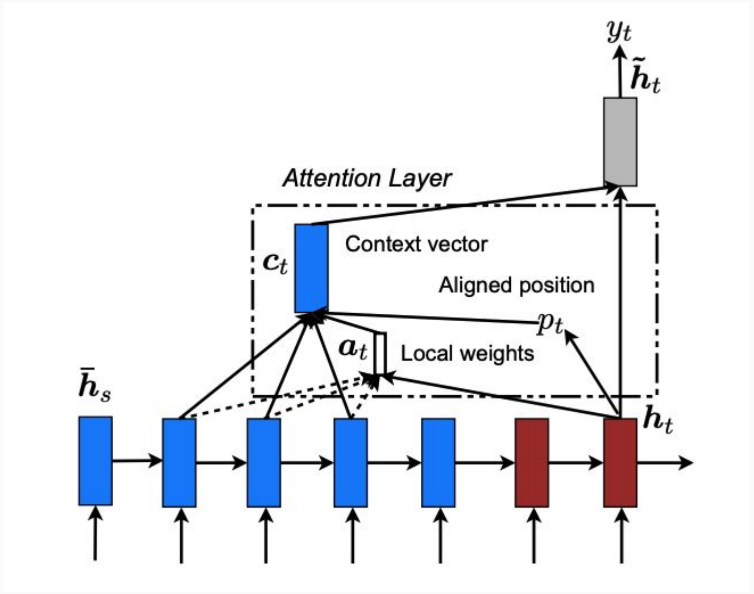
At each time step *t*, the model infers a *variable-length* alignment weight vector a_t based on the current target state *h* and all source states $\bar{h}_s$.

A global context t_s vector c_t is then computed as the weighted average, according to a_t, over all the source states.



Local Attention

- The model first predicts a single aligned position pt for the current target word.
- A window centered around the source position p_t is then used to compute a context vector c_t, a weighted average of the source hidden states in the window.
- The weights at are inferred from the current target state h_t and those source states $\bar{h}_s$ in the window.



Results

- With local attention, they achieve a significant gain of 5.0 BLEU points over non-attentional systems.
- The ensemble model using different attention architectures yielded a new state-of-the-art result in the WMT’15 English to German translation task with 25.9 BLEU points, an improvement of 1.0 BLEU points over the existing best system in that time period.

The basic problem

- Gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization).
- Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding)
- The difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians)

Challenges with RNN Hidden States

- Encoding entire past sequence is inefficient
- Most context not relevant for next step prediction
- Analogous to keeping every previous physical state in working memory

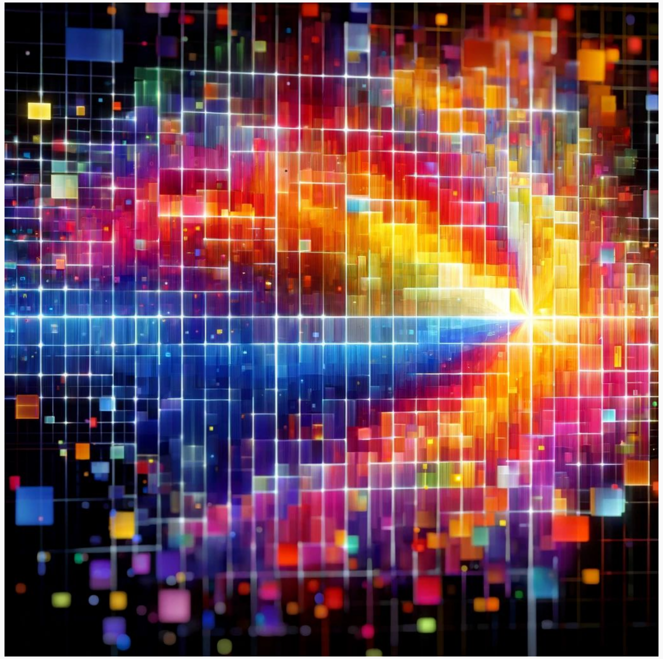
Complications

- Complications arise because sequences can be rather long.
- It is not unusual to work with text sequences consisting of over a thousand tokens. Note that this poses problems both from a computational (too much memory) and optimization (numerical instability) standpoint.
- Input from the first step passes through over 1000 matrix products before arriving at the output, and another 1000 matrix products are required to compute the gradient.

PIxelCNN

Generating images pixel by pixel

- Generates images pixel by pixel by predicting the likelihood of the next pixel based on the pixels before it.
- The model is called *PixelCNN*, and it can be trained to generate images autoregressively.
- PixelCNN uses:
    - *masked convolutional layers*
    - and *residual blocks*.


PixelCNN, visualized as pixels being progressively colored in a vibrant, grid-like pattern

Masked Convolutional Layers in PixelCNN

- Standard convolutional layers cannot be directly used for autoregressive image generation
  - No inherent ordering of pixels, unlike sequential data (e.g., text)
  - PixelCNN introduces masked convolutional layers to enable autoregressive generation

Masking in a PixelCNN

- Masking is used to ensure that the model generates pixels in an autoregressive manner, meaning that the prediction of each pixel depends only on the previously generated pixels.
- This is achieved by applying masks to the convolutional layers in the network.
- The purpose of masking is to prevent the convolutional filters from accessing information from pixels that have not been generated yet.
- By doing so, the model is forced to make predictions based solely on the pixels that come before the current pixel in the specified ordering (usually raster scan order, i.e., row by row from left to right).

Types of Masks

Type A Mask:

- Used in the first convolutional layer that is directly applied to the input image.
- The mask is designed to exclude the current pixel (central pixel) and all pixels that come after it in the ordering.
- This ensures that the prediction of the current pixel does not depend on its own value or future pixels.

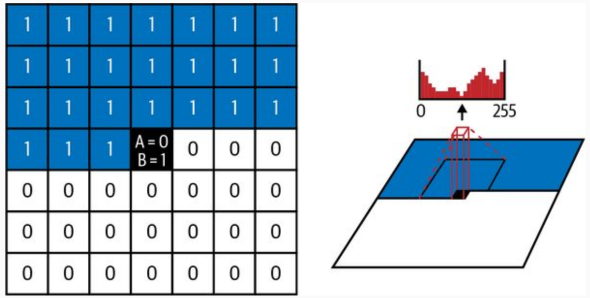
Type B Mask:

- Used in subsequent convolutional layers.
- Similar to Type A Mask, but it includes the current pixel (central pixel) in the receptive field.
- This allows the model to utilize the information from the current pixel when making predictions for future pixels.

Masks and central values

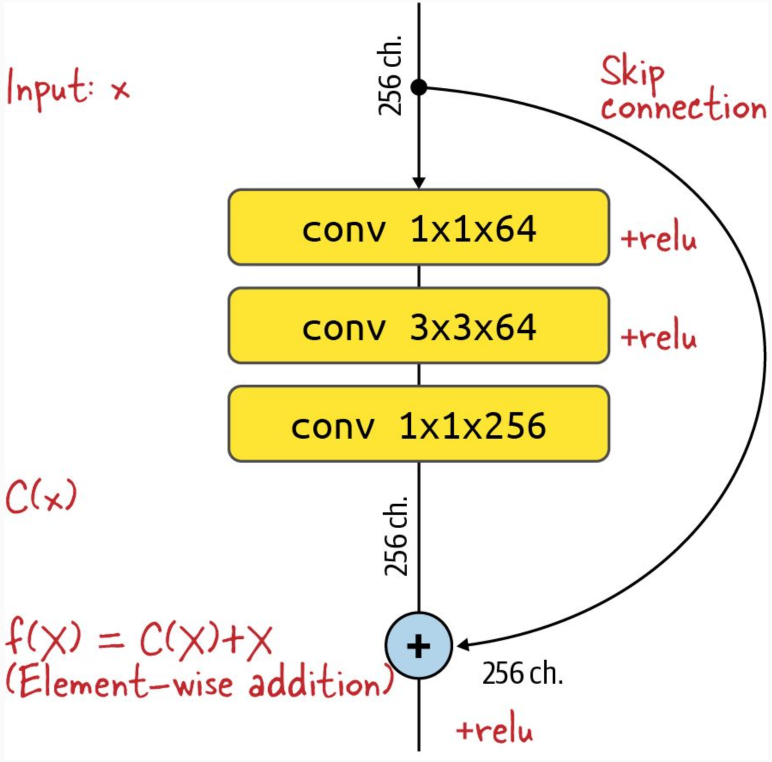
- The masks are typically binary matrices with the same shape as the convolutional filters. The values in the mask are set to either 0 or 1:
  - 0 indicates that the corresponding pixel should be excluded from the convolution operation.
  - 1 indicates that the corresponding pixel should be included in the convolution operation.
- During the forward pass of the network, the masks are element-wise multiplied with the convolutional filters before applying the convolution operation.
- This effectively masks out the undesired pixels and ensures that the model only considers the valid pixels for each prediction.
- By applying these masks, the PixelCNN model can generate pixels sequentially, with each pixel conditioned only on the previously generated pixels.
- This autoregressive property allows the model to capture the dependencies between pixels and generate coherent images.

Left: a convolutional filter mask; right: a mask applied to a set of pixels to predict the distribution of the central pixel value



Skip Connections

- Skip connections convey the signal as is, then recombine it with the data that has been transformed by one or more convolutional layers.
- The combining operation is a simple element-by-element addition.
- The output of the block *f(x)* is the sum of the output of the convolutional path *C(x)* and the skip connection *(x)*.
- The convolutional path is trained to compute *C(x) = f(x) – x*, the difference between the desired output and the input.



Residual blocks and gradients

- The key advantage of using residual blocks is that they allow the network to learn residual functions, which are the differences between the input and the desired output.
- By learning these residual functions, the network can more easily capture the subtle details and nuances in the data.
- Residual blocks also help alleviate the vanishing gradient problem.
- In deep networks without skip connections, the gradients can become very small as they propagate back through many layers, making it difficult to update the weights of the earlier layers.
- With residual blocks, the gradients can flow directly through the skip connections, allowing the network to learn more effectively.

References

1. *Goodfellow I, Bengio Y, Courville A. Deep learning. MIT press; 2016 Nov 10. Chapter 10*
2. *Foster, D. (2022). Generative deep learning. " O'Reilly Media, Inc.". Chapter 5, 9*

This week

1. Midterms - please do not postpone.
2. Homework:
    a. Read the notebook and try running it for a limited number of epochs.
    b. In a few lines please explain why a PixelCNN might need a high number of epochs (and compute) to converge to an acceptable results. Note - there is no right or wrong answer here.