Matrix Magic: Understanding Transformers with Matrices, Math, & Code.

Nadaa Taiyab

What we are going to learn

- How Transfomers work step-by-step:
  - Matrix multiplication
  - Mathematical Notation
  - Architecture Diagrams
  - PyTorch Code
- Essential Background Reading:
  - Intuition Behind Self Attention in Transformer Networks (Ark, YouTube, 40min)
  - Let's build GPT: from scratch, in code, spelled out (Andrej Kaparthy, Youtube, 2hrs)
  - Attention is All You Need (2017)

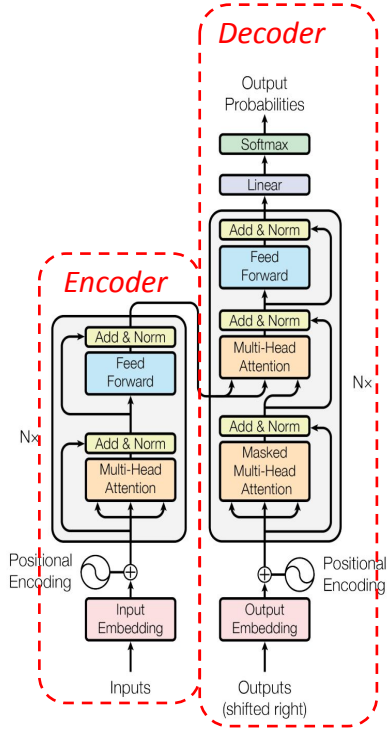


Figure 1: The Transformer - model architecture.

Encoder-Decoder Architecture

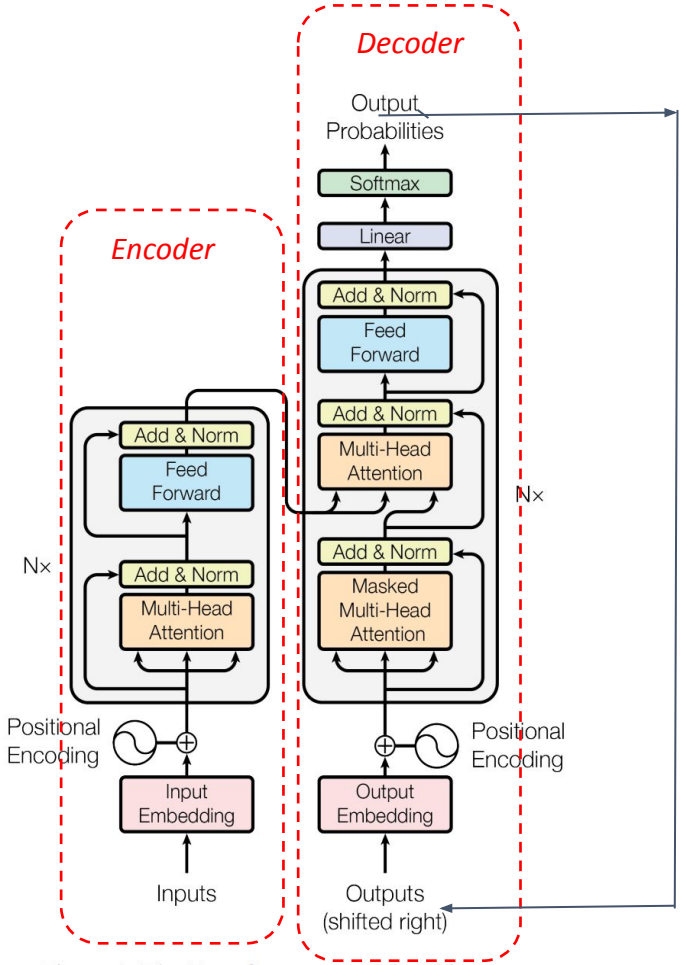
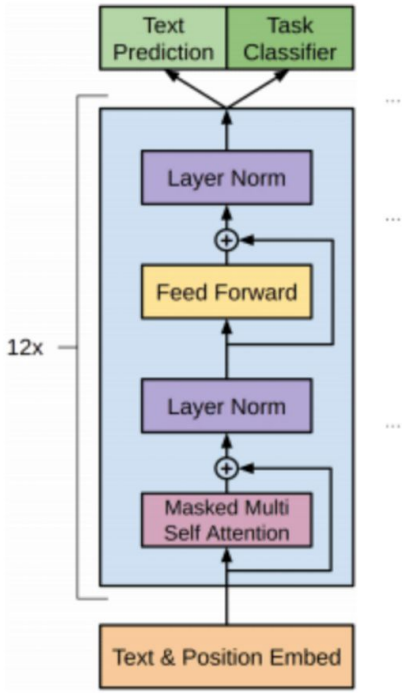


Figure 1: The Transformer - model architecture.

Source: Attention is All You Need (2017)

Outputs become inputs for the next prediction



Decoder-Only Architecture used by GPT-2

Source: Automatic Code Generation using Pre-Trained Language Models (2021)

Decoder Only Transformer Architecture

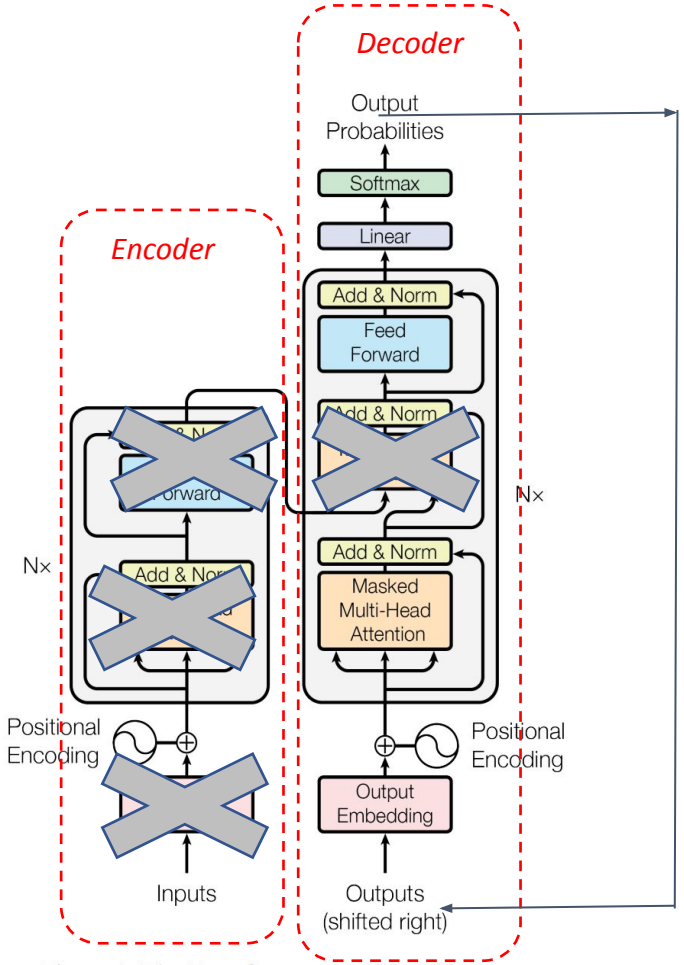
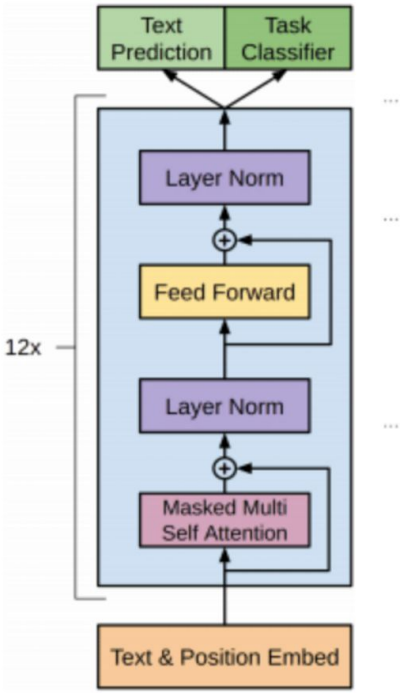


Figure 1: The Transformer - model architecture.

Source: Attention is All You Need (2017)

Outputs become inputs for the next prediction



Decoder-Only Architecture used by GPT-2

Source: Automatic Code Generation using Pre-Trained Language Models (2021)

Tokenize your training data

- **Training Data:**
  - I like to eat soup. I am a good cook. Soup is yummy.
- **Tokenization:**
  - Each word/sub-word, punctuation, and space is a token
- **Vocabulary**
  - [I, like, to, eat, soup, am, a, good, cook, is, yummy, .]

Source: DALL-E

Initialize Token Embeddings

Token Embeddings <---------- $d_{model=8}$ ---------->

Token	Vocab Code	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.26	0.38	0.00	0.27	0.18	0.67	0.65	0.57
like	1	0.23	0.72	0.86	0.96	0.33	0.93	0.23	0.31
to	2	0.89	0.42	0.71	0.64	0.51	0.31	0.63	0.15
eat	3	0.90	0.53	0.41	0.12	0.52	0.28	0.99	0.00
soup	4	0.55	0.03	0.98	0.36	0.01	0.24	0.47	0.69
am	5	0.12	0.33	0.50	0.34	0.41	0.32	0.57	0.04
a	6	0.89	0.34	0.20	0.76	0.04	0.49	0.65	0.54
good	7	0.39	0.21	0.99	0.55	0.51	0.71	0.05	0.78
cook	8	1.00	0.16	0.37	0.49	0.18	0.78	0.94	0.56
.	9	0.97	0.66	0.28	0.82	0.71	0.64	0.18	0.04
is	10	0.56	0.00	0.56	0.29	0.37	0.83	0.95	0.56
yummy	11	0.73	0.67	0.46	0.21	0.18	0.97	0.60	0.38

```
# initialize the embeddings
vocab_size = 12
embeddings_size = 8
emb = nn.Embedding(vocab_size, embeddings_size)

# print the embeddings matrix
print(emb.weight)
```

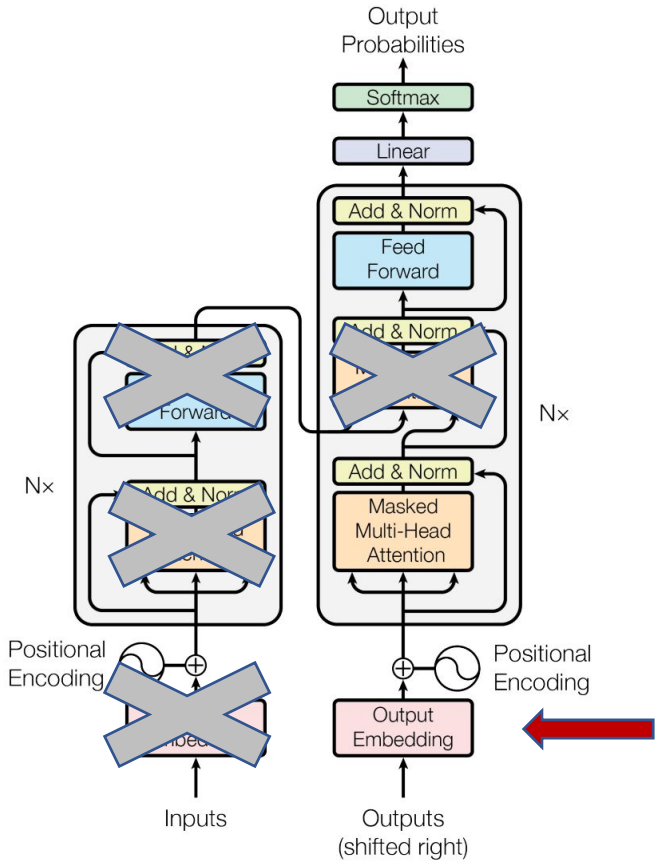


Figure 1: The Transformer - model architecture.

The very old way of vectorizing tokens

One hot encoding (vocab_size, vocab_size)

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
I	1	0	0	0	0	0	0	0	0	0	0	0
like	0	1	0	0	0	0	0	0	0	0	0	0
to	0	0	1	0	0	0	0	0	0	0	0	0
eat	0	0	0	1	0	0	0	0	0	0	0	0
soup	0	0	0	0	1	0	0	0	0	0	0	0
am	0	0	0	0	0	1	0	0	0	0	0	0
a	0	0	0	0	0	0	1	0	0	0	0	0
good	0	0	0	0	0	0	0	1	0	0	0	0
cook	0	0	0	0	0	0	0	0	1	0	0	0
.	0	0	0	0	0	0	0	0	0	1	0	0
is	0	0	0	0	0	0	0	0	0	0	1	0
yummy	0	0	0	0	0	0	0	0	0	0	0	1

- No information about similarity of words. They are all equally different
- Very large, sparse matrices

Pluck Out the Tokens in Your Batch

Embeddings <----------$d_{model=8}$---------->

Token	Vocab Code	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.26	0.38	0.00	0.27	0.18	0.67	0.65	0.57
like	1	0.23	0.72	0.86	0.96	0.33	0.93	0.23	0.31
to	2	0.89	0.42	0.71	0.64	0.51	0.31	0.63	0.15
eat	3	0.90	0.53	0.41	0.12	0.52	0.28	0.99	0.00

```
block_size = 4
num_batches = 1

# Create a tensor with the row numbers
idx = torch.tensor([0,1,2,3])

# Pluck out the rows in the embeddings table
logits = token_embedding_table(idx)
```

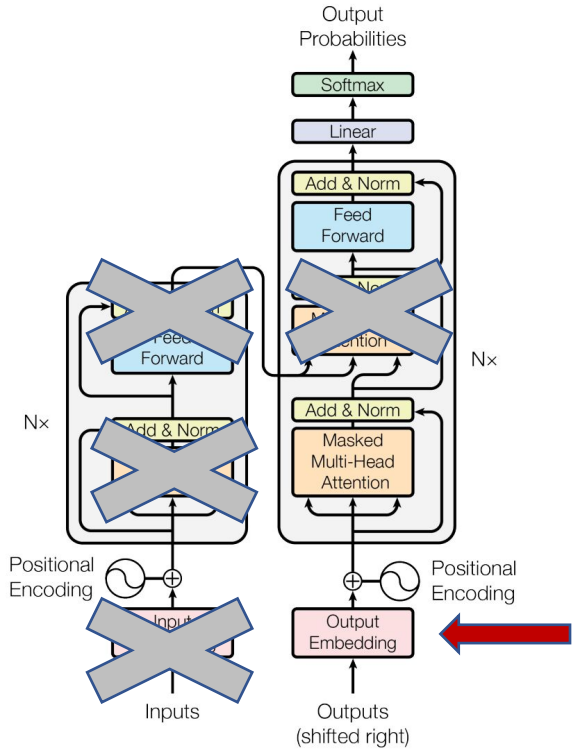


Figure 1: The Transformer - model architecture.

Why positional embeddings or encodings?

- Transformers don’t have inherent sense of the position of the data in the sequence
- All input tokens are processed simultaneously (will see in attention mechanism)
- Positional embeddings (or encodings) give info on relative absolute positions of tokens within a text

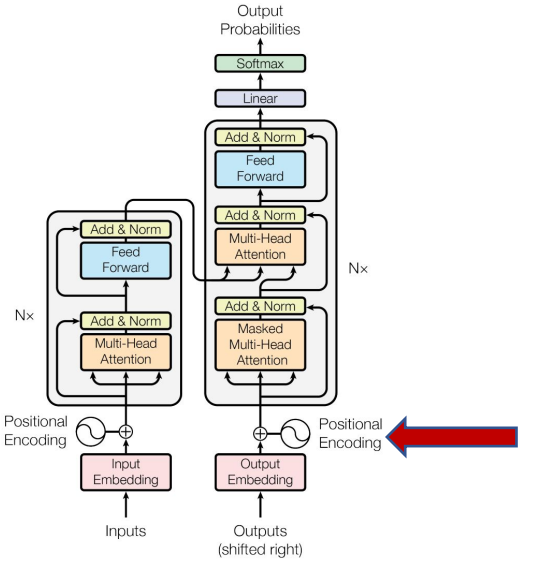


Figure 1: The Transformer - model architecture.

Initialize Positional Embeddings Table

Option 1: Learned Positional Embeddings

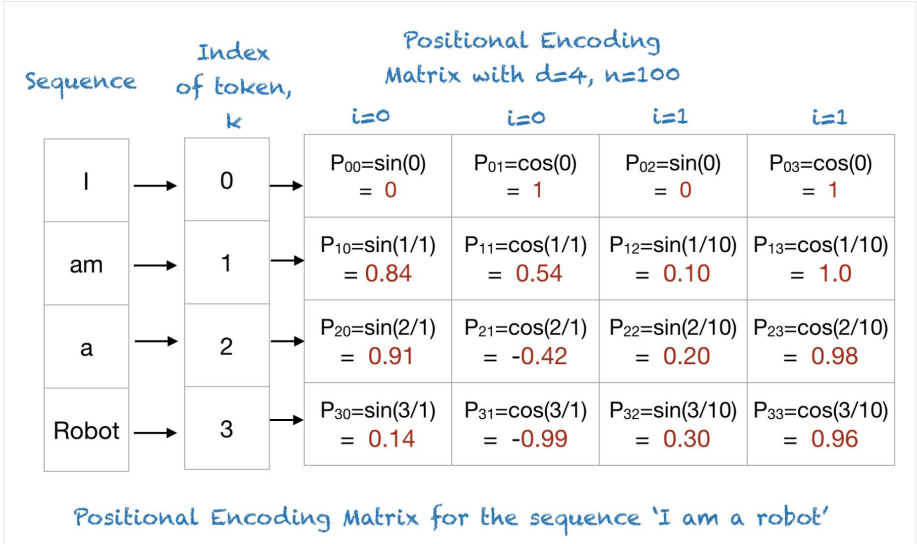
Position Embeddings <---------- $d_{model=8}$ ---------->

Position	d1	d2	d3	d4	d5	d6	d7	d8
0	0.42	0.16	0.05	0.38	0.05	0.79	0.12	0.27
1	0.19	0.50	0.26	0.53	0.71	0.16	0.42	0.32
2	0.32	0.24	0.36	0.86	0.02	0.60	0.31	0.01
3	0.90	0.39	0.44	0.01	0.20	0.66	0.94	0.75

- These weights will be learned by the model

```
position_embedding_table = nn.Embedding(block_size, n_embd)
```

Option 2: Positional Encoding



Source: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Add token and position embeddings

Token Embeddings <---------- $d_{model=8}$ ---------->

Token	Vocab Code	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.26	0.38	0.00	0.27	0.18	0.67	0.65	0.57
like	1	0.23	0.72	0.86	0.96	0.33	0.93	0.23	0.31
to	2	0.89	0.42	0.71	0.64	0.51	0.31	0.63	0.15
eat	3	0.90	0.53	0.41	0.12	0.52	0.28	0.99	0.00

Position Embeddings <---------- $d_{model=8}$ ---------->

Position	d1	d2	d3	d4	d5	d6	d7	d8
0	0.42	0.16	0.05	0.38	0.05	0.79	0.12	0.27
1	0.19	0.50	0.26	0.53	0.71	0.16	0.42	0.32
2	0.32	0.24	0.36	0.86	0.02	0.60	0.31	0.01
3	0.90	0.39	0.44	0.01	0.20	0.66	0.94	0.75

Token Embeddings (4,8) + Position Embeddings (4,8) =

Token	Position	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.68	0.54	0.06	0.65	0.23	1.45	0.76	0.84
like	1	0.43	1.22	1.13	1.49	1.04	1.09	0.65	0.62
to	2	1.20	0.66	1.08	1.50	0.54	0.91	0.94	0.16
eat	3	1.80	0.92	0.84	0.14	0.72	0.94	1.93	0.76

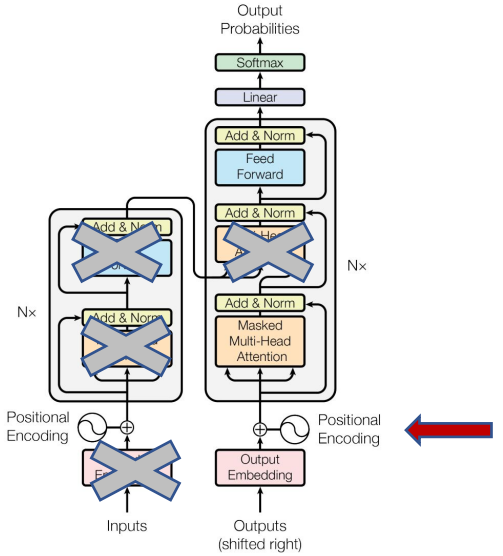
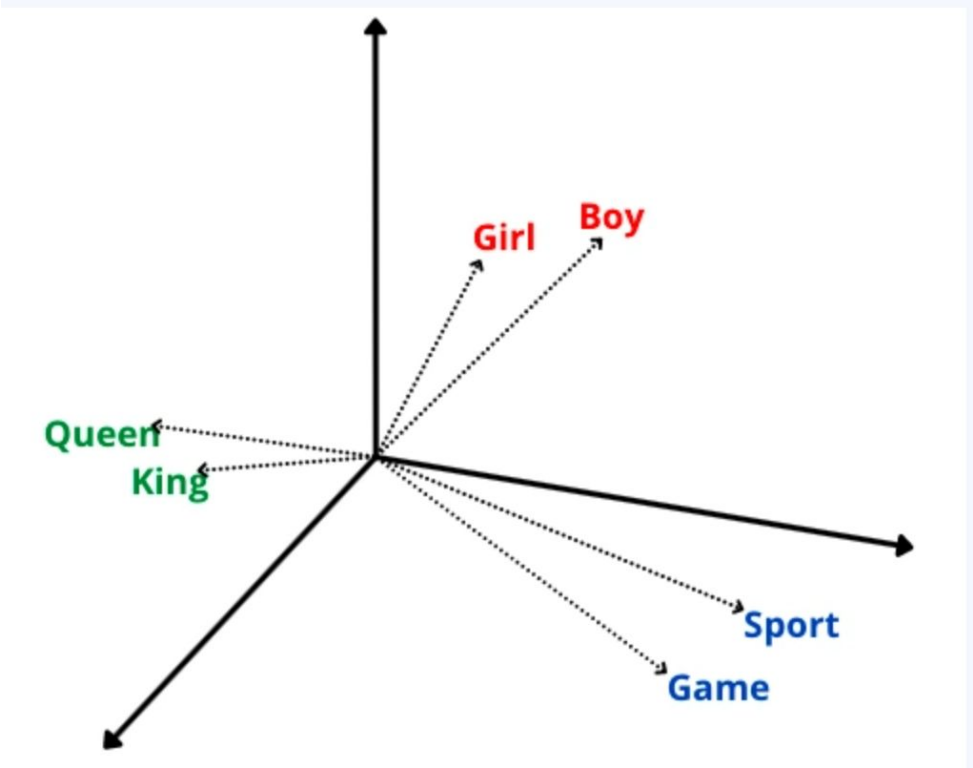


Figure 1: The Transformer - model architecture.

```
# idx is a tensor with the row numbers you want
B, T = idx.shape
# idx and targets are both (B,T) tensor of integers
tok_emb = token_embedding_table(idx) # (B,T,C)
pos_emb = position_embedding_table(torch.arange(T, device=device))# (T,C)
x = tok_emb + pos_emb # (B,T,C)
```

What are token embeddings?



- Embeddings are vectors that represent meaning of the word

- Vectors of words/tokens with similar meanings are closer in space

Example

boy = [10, 8, 9]

girl = [9, 7, 10]

- Fantastic Deep Dive: Stanford CS224N: NLP with Deep Learning | Winter 2021 | Lecture 1 – Intro & Word Vectors

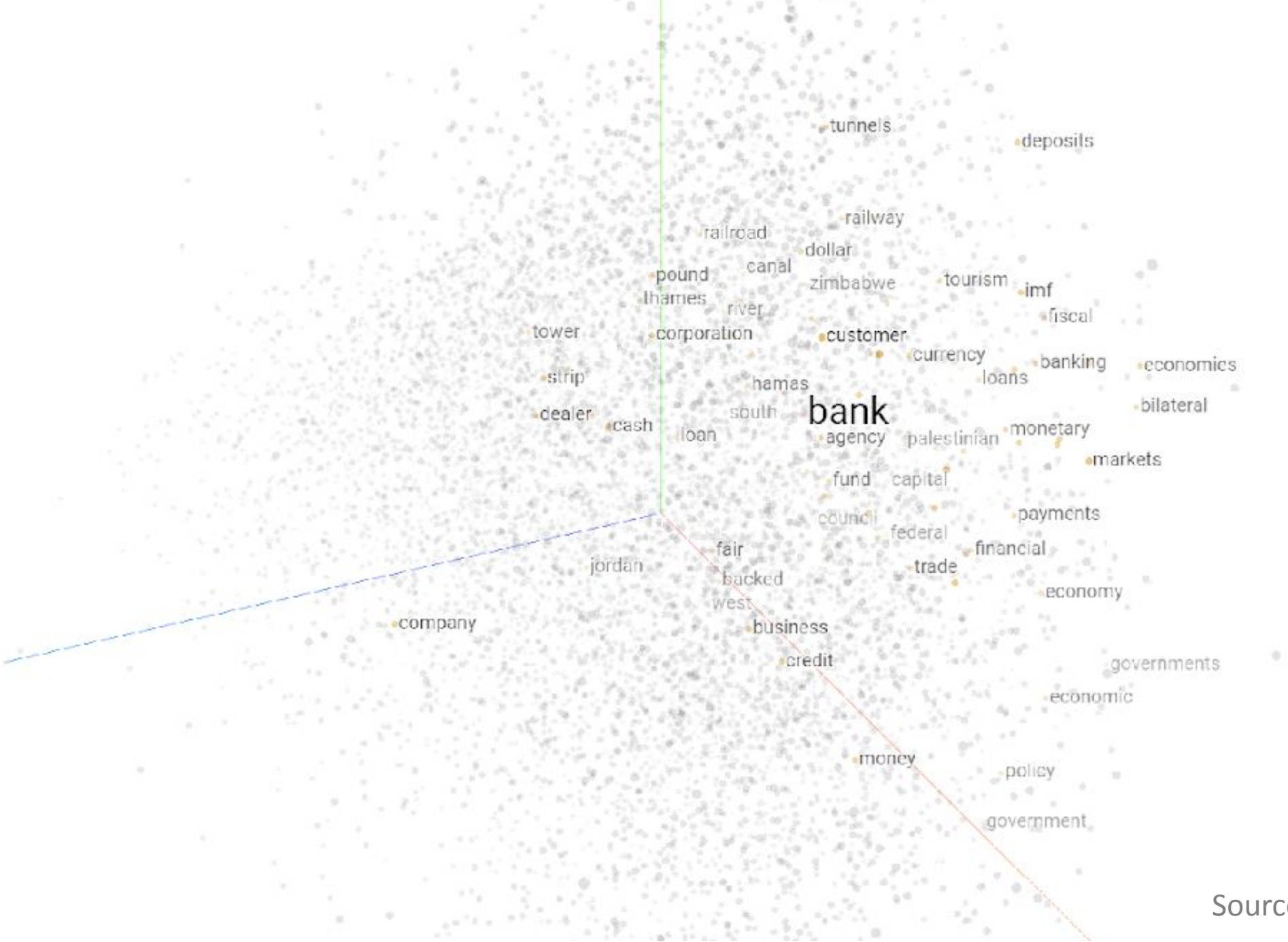
What kind of bank?

I swam across the river to get to the other bank.

I walked across the intersection to get to the other bank.

- River bank or a financial institution or a blood bank?!!

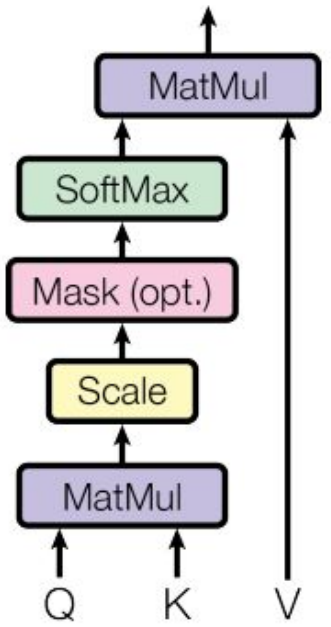
Source: Intuition Behind Self-Attention Mechanism in Transformer Networks (YouTube)



Source: projector.tensorflow.org

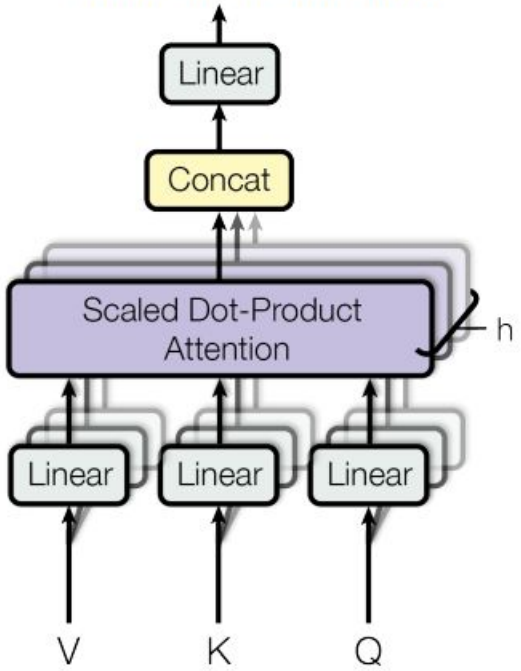
Multi-Headed Attention

Scaled Dot-Product Attention



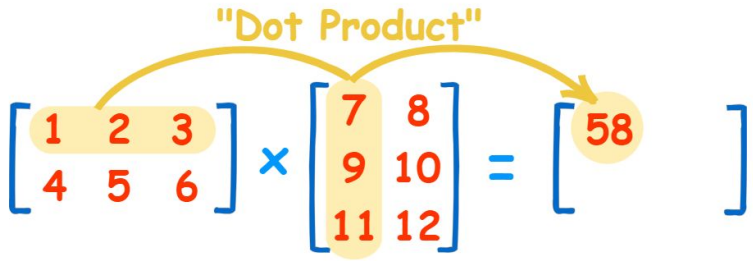
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention

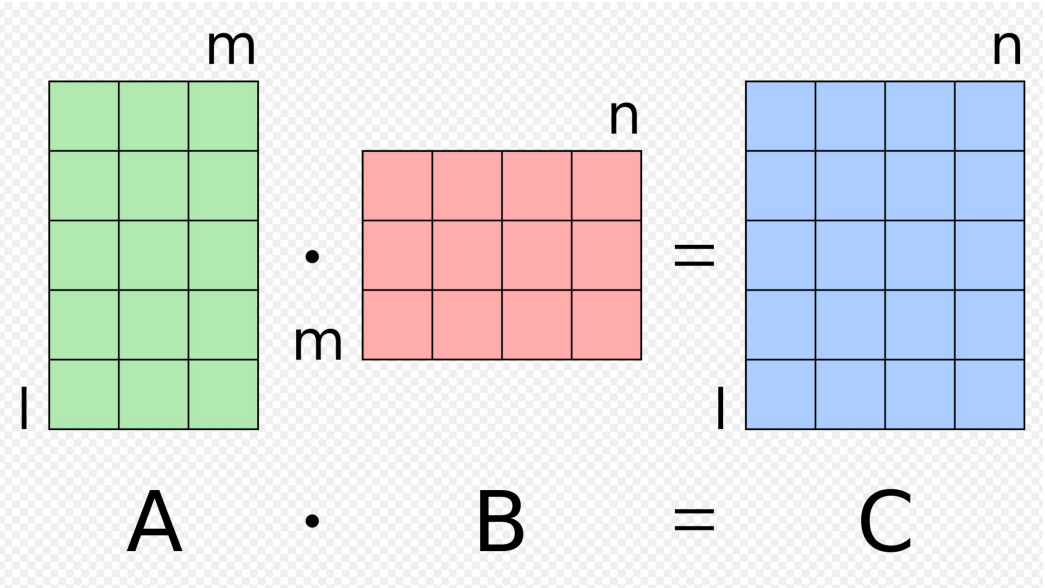


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Matrix Multiplication



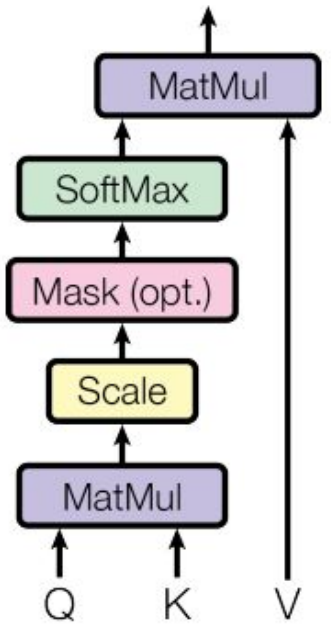
Source: https://www.mathsisfun.com/algebra/matrix-multiplying.html



Source: https://en.wikipedia.org/wiki/Matrix_multiplication

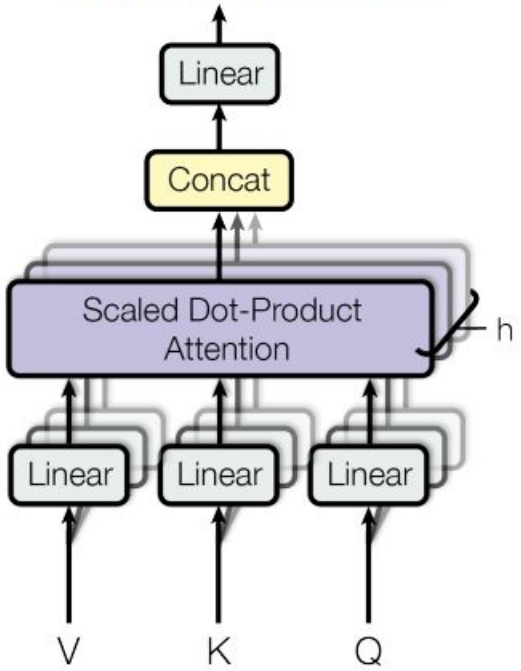
Multi-Headed Attention

Scaled Dot-Product Attention



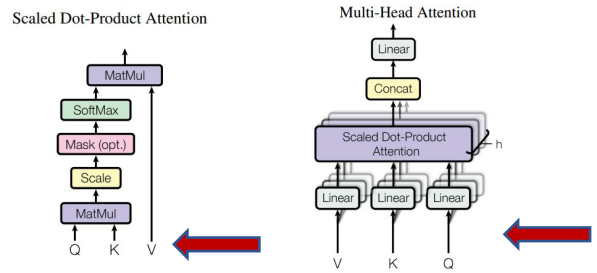
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Query, Key, and Value Matrices



Token Embeddings (4,8) + Position Embeddings (4,8) =

Token	Position	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.68	0.54	0.06	0.65	0.23	1.45	0.76	0.84
like	1	0.43	1.22	1.13	1.49	1.04	1.09	0.65	0.62
to	2	1.20	0.66	1.08	1.50	0.54	0.91	0.94	0.16
eat	3	1.80	0.92	0.84	0.14	0.72	0.94	1.93	0.76

```
head_size = 4 columns
n_embd = 8 columns
```

@

Queries Weights (8,4)

	1	2	3	4
1	0.08	0.34	0.26	0.30
2	0.64	0.24	0.48	0.09
3	0.57	0.33	0.50	0.15
4	0.70	0.11	0.69	0.55
5	0.95	0.36	0.32	0.14
6	0.74	0.63	0.17	0.33
7	0.06	0.64	0.18	0.27
8	0.73	0.97	0.28	0.58

=

Queries (4,4)

	d1	d2	d3	d4
I	2.84	2.76	1.61	1.82
like	4.79	3.06	3.10	2.27
to	3.54	2.62	2.74	2.13
eat	3.35	3.96	2.38	2.20

```
query = nn.Linear(n_embd, head_size)
q = query(x)
```

@

Keys Weights (8,4)

	1	2	3	4
1	0.99	0.23	0.94	0.01
2	0.76	0.19	0.93	0.12
3	0.66	0.57	0.40	0.02
4	0.45	0.32	0.24	0.64
5	0.46	0.38	0.92	0.42
6	0.43	0.56	0.79	0.91
7	0.68	0.52	0.12	0.64
8	0.01	0.61	0.15	0.51

=

Keys (4,4)

	d1	d2	d3	d4
I	2.68	2.31	2.89	2.82
like	4.17	3.18	4.33	3.29
to	4.37	2.80	3.88	2.81
eat	5.17	3.38	4.67	3.01

```
key = nn.Linear(n_embd, head_size)
k = key(x)
```

@

Values Weights (8,4)

	1	2	3	4
1	0.54	0.75	0.47	0.32
2	0.04	0.43	1.00	0.75
3	0.86	0.15	0.86	0.72
4	0.07	0.97	0.93	0.27
5	0.76	0.44	0.22	0.46
6	0.18	0.02	0.23	0.51
7	0.80	0.32	0.13	0.71
8	0.23	0.25	0.42	0.05

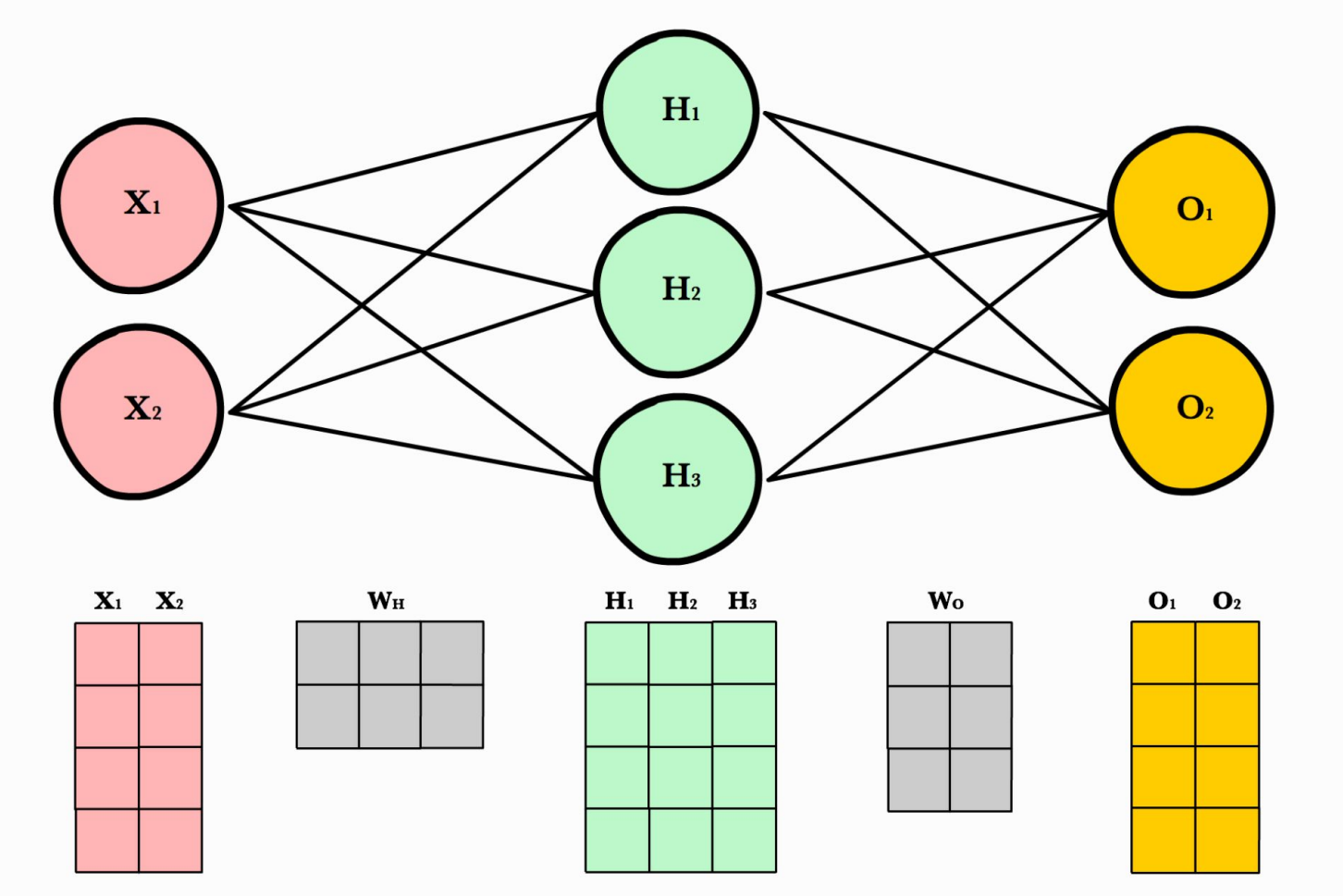
=

Values (4,4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.08	3.39	4.06	3.44
eat	4.18	3.14	3.55	4.12

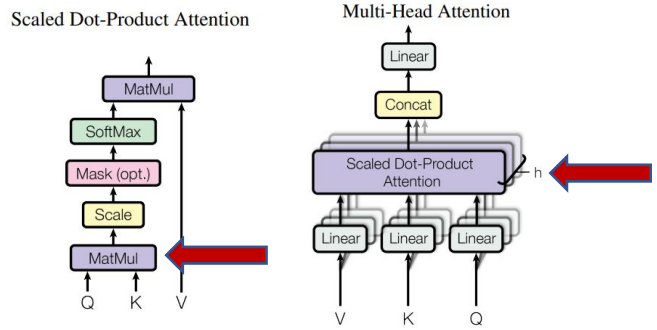
```
value = nn.Linear(n_embd, head_size)
v = v(x)
```

Linear Layers and Matrix Multiplication



Source: https://en.wikipedia.org/wiki/Matrix_multiplication

Attention Matrix



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

Queries (4,4)

	d1	d2	d3	d4
I	2.84	2.76	1.61	1.82
like	4.79	3.06	3.10	2.27
to	3.54	2.62	2.74	2.13
eat	3.35	3.96	2.38	2.20

@

Keys Transposed (4,)

	I	like	to	eat
d1	2.68	4.17	4.37	5.17
d2	2.31	3.18	2.80	3.38
d3	2.89	4.33	3.88	4.67
d4	2.82	3.29	2.81	3.01

$d_q = 4$

$d_k = 4$

Attention Matrix (4, 4)

	I	like	to	eat
I	28.84	40.31	37.30	43.67
like	40.31	57.05	53.14	62.45
to	37.30	53.14	49.84	58.63
eat	43.67	62.45	58.63	69.08

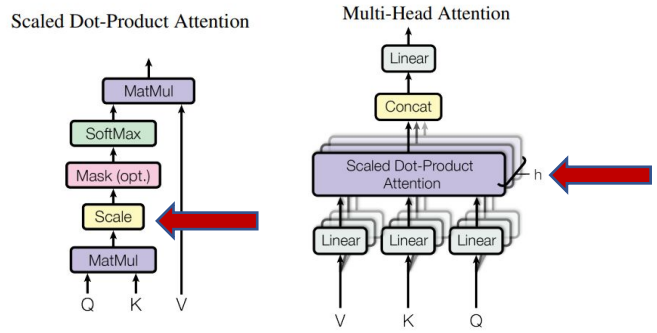
```
wei = q @ k.transpose(-2, -1)
```

Why is scaling necessary?

- Model will start paying too much attention to only the word with the largest numbers (softmax saturation)
- Reduce numerical instability during optimization

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

Scaling



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

Attention Matrix (4, 4)

	I	like	to	eat
I	28.84	40.31	37.30	43.67
like	40.31	57.05	53.14	62.45
to	37.30	53.14	49.84	58.63
eat	43.67	62.45	58.63	69.08

$$\frac{1}{\sqrt{d_k}}$$

$$d_k = 4$$

Scaled Attention (4, 4)

	I	like	to	eat
I	14.42	20.16	18.65	21.83
like	20.16	28.53	26.57	31.22
to	18.65	26.57	24.92	29.32
eat	21.83	31.22	29.32	34.54

```
wei = wei * C**-0.5
```

Encoder vs Decoder

Encoder Attention
No Masking

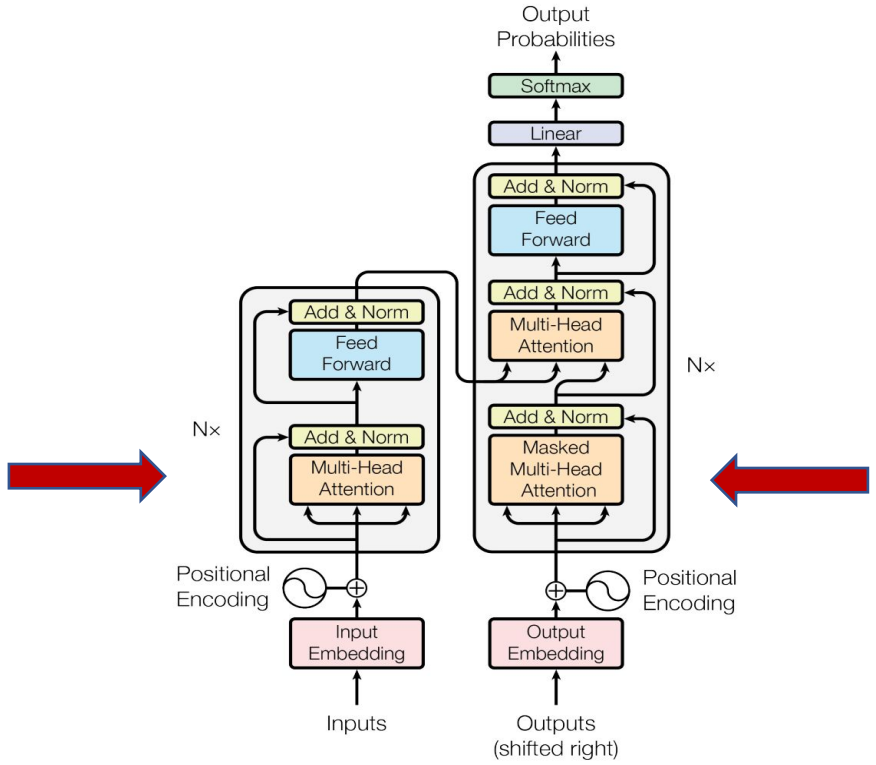
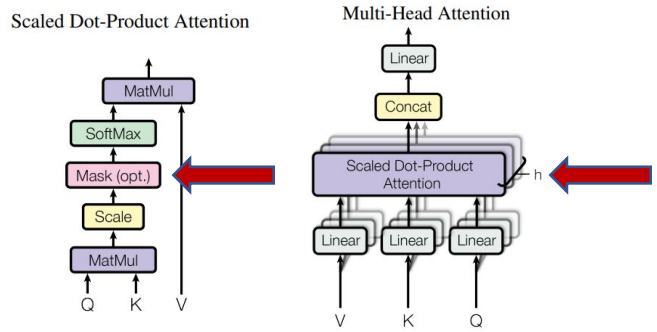


Figure 1: The Transformer - model architecture.

Decoder Attention
Masked

Masking



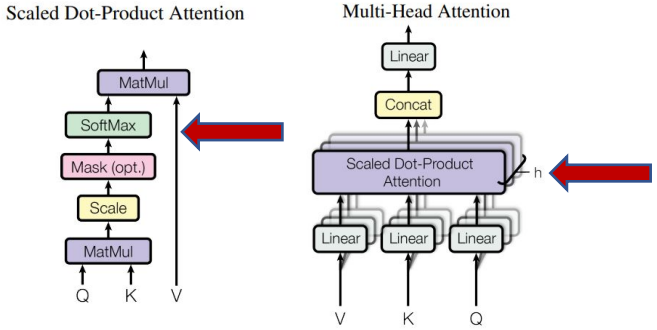
Masked Attention (4, 4)

	I	like	to	eat
I	14.42	0	0	0
like	20.16	28.53	0	0
to	18.65	26.57	24.92	0
eat	21.83	31.22	29.32	34.54

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

```
wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
```

Softmax



Masked Attention (4, 4)

First get explonential of each element and add them up

	I	like	to	eat	SUM
I	1827800	1	1	1	1827803
like	567261070	2449586694989	1	1	2450153956061
to	125826465	345618259437	66316918440	1	412061004343
eat	3039308147	36359847277715	5389806117804	1003656459556760	1045409152260420

Masked Attention (4, 4)

Divide each row by the sum of the exponents

	I	like	to	eat
I	1.00	0.00	0.00	0.00
like	0.00	1.00	0.00	0.00
to	0.00	0.84	0.16	0.00
eat	0.00	0.03	0.01	0.96

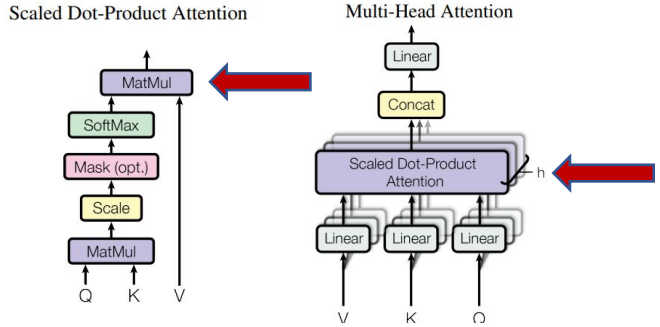
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Softmax Function

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

```
wei = F.softmax(wei, dim=-1)
```

Multiply by Values Matrix



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

Masked Attention (4, 4)

	I	like	to	eat
I	1.00	0.00	0.00	0.00
like	0.00	1.00	0.00	0.00
to	0.00	0.84	0.16	0.00
eat	0.00	0.03	0.01	0.96

@

Values Matrix (4, 4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.08	3.39	4.06	3.44
eat	4.18	3.14	3.55	4.12

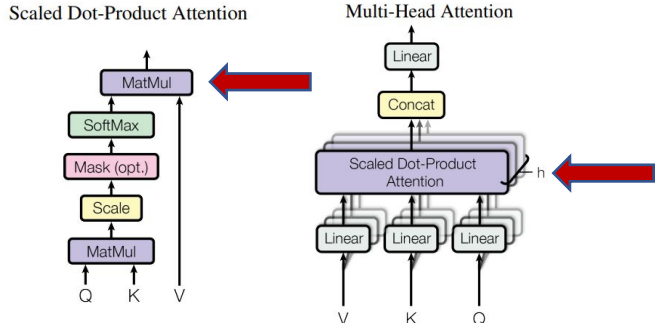
=

Weighted Values (4, 4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.02	3.31	4.51	3.73
eat	4.13	3.15	3.59	4.11

```
out = wei @ v
```

Multiply by Values Matrix



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

Masked Attention (4, 4)

	I	like	to	eat
I	1.00	0.00	0.00	0.00
like	0.00	1.00	0.00	0.00
to	0.00	0.84	0.16	0.00
eat	0.00	0.03	0.01	0.96

@

Values Matrix (4, 4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.08	3.39	4.06	3.44
eat	4.18	3.14	3.55	4.12

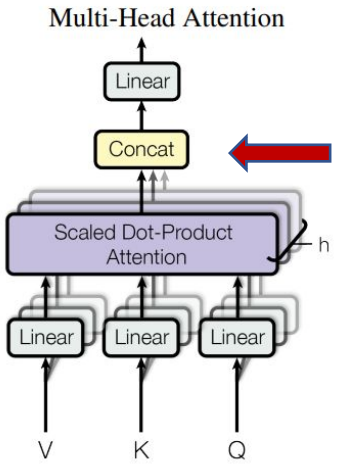
Weighted Values (4, 4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.02	3.31	4.51	3.73
eat	4.13	3.15	3.59	4.11

0X1.73 + .03 X3.01 + 0.01X3.08 + .96X4.18 = 4.13

```
out = wei @ v
```

Concatenate Multiple Attention Heads



Head 1: Weighted Values (4, 4)

	d1	d2	d3	d4
I	1.73	1.96	2.34	2.26
like	3.01	3.30	4.60	3.79
to	3.02	3.31	4.51	3.73
eat	4.13	3.15	3.59	4.11

Head 2: Weighted Values (4, 4)

	d1	d2	d3	d4
I	0.28	1.22	0.64	1.65
like	1.50	1.93	2.15	3.16
to	1.31	3.07	2.56	2.06
eat	1.74	1.32	1.49	0.47

Head 3: Weighted Values (4, 4)

	d1	d2	d3	d4
I	1.63	0.82	1.93	0.57
like	2.27	2.51	2.63	0.57
to	0.82	1.41	1.21	0.89
eat	2.48	2.53	2.85	0.64

Head 4: Weighted Values (4, 4)

	d1	d2	d3	d4
I	1.11	0.63	0.67	1.75
like	1.25	0.52	3.75	1.06
to	3.05	2.33	1.14	1.12
eat	0.77	2.04	2.68	1.74

Multi-headed Attention: Concatenated Weighted Values (4, 16)

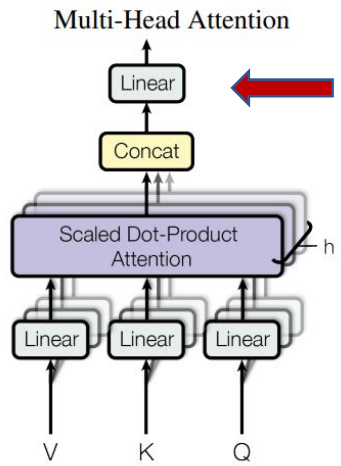
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16
I	1.73	1.96	2.34	2.26	1.56	1.93	2.12	0.28	1.28	1.83	0.93	1.03	0.33	1.80	0.47	0.25
like	3.01	3.30	4.60	3.79	0.18	0.03	1.14	2.66	1.32	0.59	1.69	3.23	2.74	3.09	2.22	1.39
to	3.02	3.31	4.51	3.73	0.55	1.05	2.56	2.26	1.13	1.97	2.86	2.56	0.93	1.86	0.02	2.55
eat	4.13	3.15	3.59	4.11	0.91	1.27	1.63	2.47	0.76	2.15	2.50	1.18	0.26	1.67	3.15	3.87

```
# Run multiple attention heads in parallel
# Head() is a class that runs the self-attention mechanism
num_heads = 4
heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])

# Concatenate the outputs into one big matrix
out = torch.cat([h(x) for h in self.heads], dim=-1)
```

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Linear Layer on Attention Heads



Multi-headed Attention: Concatenated Weighted Values (4, 16)

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16
I	1.73	1.96	2.34	2.26	1.56	1.93	2.12	0.28	1.28	1.83	0.93	1.03	0.33	1.80	0.47	0.25
like	3.01	3.30	4.60	3.79	0.18	0.03	1.14	2.66	1.32	0.59	1.69	3.23	2.74	3.09	2.22	1.39
to	3.02	3.31	4.51	3.73	0.55	1.05	2.56	2.26	1.13	1.97	2.86	2.56	0.93	1.86	0.02	2.55
eat	4.13	3.15	3.59	4.11	0.91	1.27	1.63	2.47	0.76	2.15	2.50	1.18	0.26	1.67	3.15	3.87

@

Linear Layer Weights (16, 8)

	d1	d2	d3	d4	d5	d6	d7	d8
1	0.015	0.535	0.668	0.347	0.743	0.64	0.593	0.052
2	0.955	0.108	0.532	0.731	0.321	0.011	0.175	0.083
3	0.711	0.497	0.906	0.378	0.13	0.168	0.663	0.729
4	0.127	0.958	0.425	0.878	0.195	0.836	0.523	0.719
5	0.287	0.702	0.9	0.531	0.555	0.737	0.09	0.702
6	0.008	0.245	0.319	0.528	0.074	0.789	0.997	0.603
7	0.699	0.548	0.888	0.263	0.369	0.386	0.873	0.93
8	0.787	0.675	0.767	0.268	0.948	0.203	0.938	0.467
9	0.729	0.873	0.553	0.787	0.386	0.56	0.458	0.311
10	0.709	0.01	0.263	0.224	0.498	0.14	0.742	0.838
11	0.75	0.05	0.104	0.843	0.177	0.381	0.845	0.539
12	0.711	0.828	0.862	0.602	0.929	0.164	0.696	0.082
13	0.06	0.71	0.622	0.709	0.286	0.433	0.545	0.76
14	0.439	0.742	0.699	0.472	0.085	0.131	0.929	0.628
15	0.511	0.4	0.627	0.853	0.464	0.293	0.772	0.753
16	0.989	0.253	0.964	0.964	0.431	0.781	0.086	0.224

=

Multi-headed Attention (4, 8)

	1	2	3	4	5	6	7	8
1	10.98	11.24	13.58	11.94	7.82	9.172	13.97	12.26
2	18.85	19.83	23.21	20.81	14.45	12.36	21.83	17.03
3	20.22	17.33	22.3	19.88	13.93	13.6	21.43	17.09
4	20.48	17.22	23.3	22.23	15.14	15.67	21.96	18.09

```
# Initialize linear layer and dropout (not shown)
proj = nn.Linear(n_heads*head_size, n_embd)
dropout = nn.Dropout(dropout)
# Concatenate
out = torch.cat([h(x) for h in self.heads], dim=-1)
# Apply linear layer
out = self.dropout(self.proj(out))
```

Addition

Multi-headed Attention (4, 8)

	1	2	3	4	5	6	7	8
1	10.98	11.24	13.58	11.94	7.82	9.172	13.97	12.26
2	18.85	19.83	23.21	20.81	14.45	12.36	21.83	17.03
3	20.22	17.33	22.3	19.88	13.93	13.6	21.43	17.09
4	20.48	17.22	23.3	22.23	15.14	15.67	21.96	18.09

Token Embeddings (4,8) + Position Embeddings (4,8) =

Token	Position	d1	d2	d3	d4	d5	d6	d7	d8
I	0	0.68	0.54	0.06	0.65	0.23	1.45	0.76	0.84
like	1	0.43	1.22	1.13	1.49	1.04	1.09	0.65	0.62
to	2	1.20	0.66	1.08	1.50	0.54	0.91	0.94	0.16
eat	3	1.80	0.92	0.84	0.14	0.72	0.94	1.93	0.76

Multi-headed Attention (4, 8) + Token Embeddings (4, 8) + Position Embeddings (4, 8)

	1	2	3	4	5	6	7	8
1	11.66	11.78	13.64	12.59	8.05	10.62	14.73	13.10
2	19.28	21.05	24.34	22.30	15.49	13.45	22.48	17.65
3	21.42	17.99	23.38	21.38	14.47	14.51	22.37	17.25
4	22.28	18.14	24.14	22.37	15.86	16.61	23.89	18.85

```
# Initalize multi-headed attention and linear layer
sa = MultiHeadAttention(n_head, head_size)
# Add token+position embeddings to the output of the
multi-headed attention
x = x + self.sa(self.ln1(x))
```

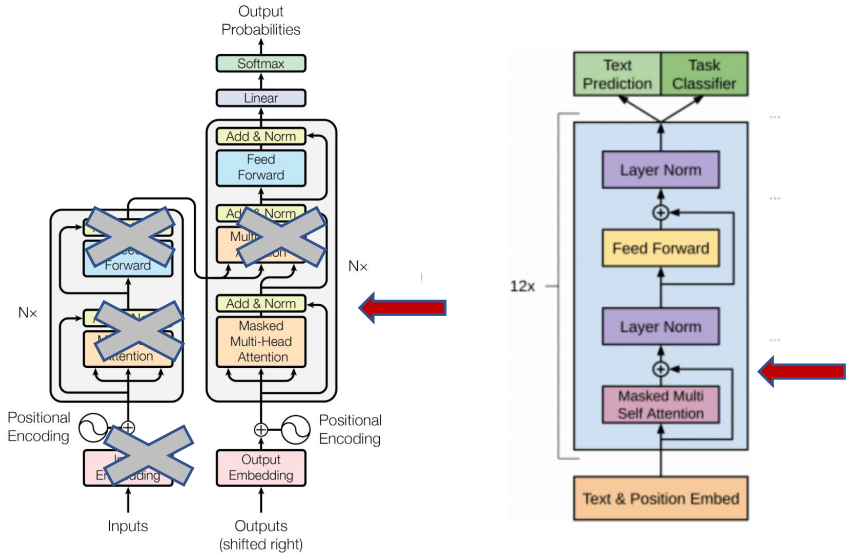


Figure 1: The Transformer - model architecture.

Decoder-Only Architecture used by GPT-2.

Layer Normalization

Operates on the columns of a single row

Multi-headed Attention (4, 8) + Token Embeddings (4, 8) + Position Embeddings (4, 8)

	1	2	3	4	5	6	7	8
1	11.66	11.78	13.64	12.59	8.05	10.62	14.73	13.10
2	19.28	21.05	24.34	22.30	15.49	13.45	22.48	17.65
3	21.42	17.99	23.38	21.38	14.47	14.51	22.37	17.25
4	22.28	18.14	24.14	22.37	15.86	16.61	23.89	18.85

Technically it is “standardization”

$$x_{std} = \frac{x - \mu}{\sigma}$$

PyTorch Implementation

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Paper: Layer Normalization (2016)

These are optional learnable weights if elementwise_affine=True

This is an error term

```
# Initialize the layer norm
self.ln1 = nn.LayerNorm(n_embd)
# Add token+position embeddings
x = x + self.sa(self.ln1(x))
```

Layer Normalization

- Leaving out learnable weights for simplicity

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

ϵ = 0.001

Multi-headed Attention (4, 8) + Token Embeddings (4, 8) + Position Embeddings (4, 8)

	1	2	3	4	5	6	7	8	Mean	Std
1	11.66	11.78	13.64	12.59	8.05	10.62	14.73	13.10	12.02	2.05
2	19.28	21.05	24.34	22.30	15.49	13.45	22.48	17.65	19.51	3.76
3	21.42	17.99	23.38	21.38	14.47	14.51	22.37	17.25	19.10	3.52
4	22.28	18.14	24.14	22.37	15.86	16.61	23.89	18.85	20.27	3.29

Layer Normalized - Multi-headed Attention (4, 8) + Token Embeddings (4, 8) + Position Embeddings (4, 8)

	1	2	3	4	5	6	7	8
1	-0.25	-0.17	1.13	0.40	-2.78	-0.98	1.89	0.75
2	-0.12	0.80	2.50	1.44	-2.07	-3.12	1.54	-0.96
3	1.24	-0.59	2.28	1.22	-2.47	-2.44	1.74	-0.98
4	1.11	-1.17	2.13	1.16	-2.43	-2.02	2.00	-0.78

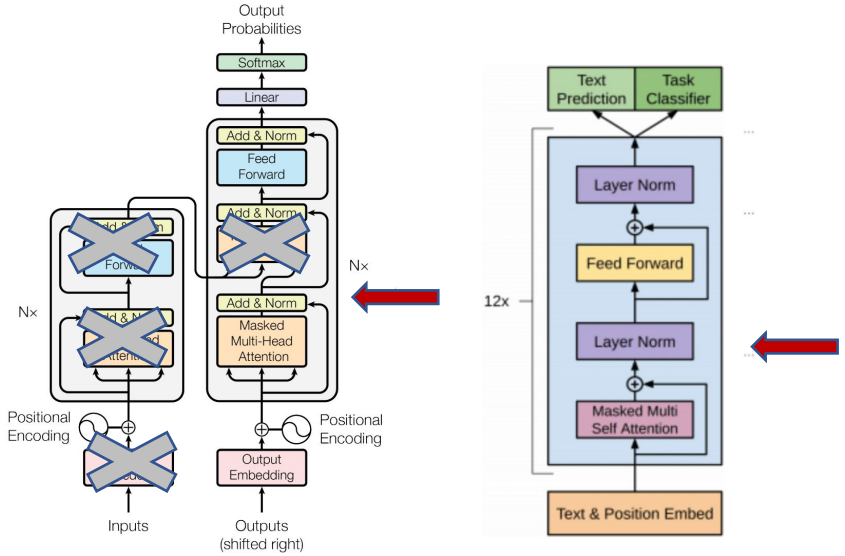


Figure 1: The Transformer - model architecture.

Decoder-Only Architecture used by GPT-2.

```
# Initialize the layer norm
self.ln1 = nn.LayerNorm(n_embd)
# Add token+position embeddings
x = x + self.sa(self.ln1(x))
```

Feed Forward Network (FFN)

```
#Initialize feed forward network
net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )
#Run feed forward network
net(x)
```

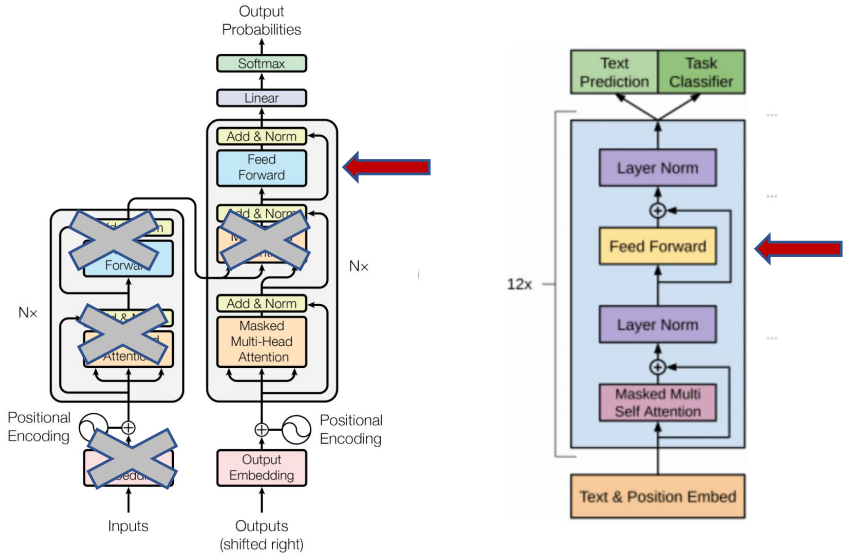


Figure 1: The Transformer - model architecture.

Decoder-Only Architecture used by GPT-2.

FFN: Linear Layer

Multi-headed Attention (4, 8) + Token Embeddings (4, 8) + Position Embeddings (4, 8)

	1	2	3	4	5	6	7	8
1	11.66	11.78	13.64	12.59	8.05	10.62	14.73	13.10
2	19.28	21.05	24.34	22.30	15.49	13.45	22.48	17.65
3	21.42	17.99	23.38	21.38	14.47	14.51	22.37	17.25
4	22.28	18.14	24.14	22.37	15.86	16.61	23.89	18.85

@

Weights (8, 32)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0.6	0.6	0.5	0.6	0.5	0.5	0.1	0.9	0.4	0.8	0.5	0.6	0.4	0.1	1.0	0.1	0.3	0.2	0.7	0.8	0.3	0.1	0.9	0.6	0.4	0.8	0.6	0.5	0.7	0.3	0.7	0.6
2	0.6	0.4	1.0	1.0	0.6	0.4	0.6	0.3	0.3	0.0	1.0	0.4	0.9	0.6	0.7	0.9	0.6	0.3	0.6	0.2	0.2	0.5	0.8	0.7	0.3	0.5	0.2	0.2	0.9	0.7	0.4	0.9
3	0.2	0.8	1.0	1.0	0.8	0.6	0.8	0.2	0.8	0.2	0.7	0.9	0.3	0.9	0.2	0.2	0.4	0.9	0.8	0.8	0.8	0.3	0.6	0.7	0.4	0.6	0.0	0.6	0.1	1.0	0.0	0.6
4	0.9	0.1	0.9	0.3	0.0	0.8	0.4	0.4	0.4	0.3	0.7	0.2	0.8	0.4	0.3	0.8	1.0	0.4	1.0	0.9	0.0	0.6	0.7	0.5	0.1	0.1	0.4	0.2	0.9	0.2	0.2	0.2
5	0.3	0.3	0.6	0.8	0.2	0.9	0.8	0.8	0.8	0.7	0.2	0.4	0.9	0.4	0.2	0.1	0.8	0.4	0.6	0.3	1.0	0.8	0.8	0.8	0.3	0.5	0.5	0.3	0.1	0.5	0.3	0.6
6	0.3	0.7	0.9	0.7	0.8	0.5	0.1	0.4	0.3	0.7	0.8	0.3	0.4	0.1	0.5	0.8	0.5	0.9	0.7	0.3	0.6	0.5	0.6	0.1	1.0	0.4	0.4	0.8	0.3	0.1	0.9	0.2
7	0.4	0.2	0.2	0.7	0.9	0.8	0.7	0.8	0.6	0.8	0.3	0.4	0.6	0.3	0.4	0.3	0.0	1.0	0.6	0.0	0.2	0.9	0.4	0.8	0.6	1.0	0.2	0.2	0.7	0.9	0.0	0.1
8	0.3	0.2	0.6	0.2	0.4	0.7	0.1	0.2	0.1	0.9	0.9	0.5	0.4	0.8	0.3	0.5	1.0	0.0	0.4	0.1	0.8	0.9	0.6	0.2	0.6	0.7	0.9	0.4	1.0	0.9	0.1	0.4

Output of First Lineary Layer (4, 32)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0.3	-0.3	-0.4	-0.8	1.3	-0.1	0.2	-0.9	-0.5	-0.3	0.8	0.4	-1.1	1.1	-0.1	0.3	-1.3	1.0	0.0	0.1	-1.4	0.0	-1.0	0.0	0.2	1.0	-0.6	-0.3	1.7	1.9	-1.6	-0.7
2	1.1	-0.3	0.3	0.4	0.6	0.1	2.2	-0.7	0.9	-2.5	0.3	1.1	-0.1	2.0	-0.4	-0.2	-1.4	1.0	0.8	1.6	-2.1	-0.6	-0.3	2.0	-2.1	0.4	-1.9	-1.1	1.3	2.4	-2.7	0.4
3	1.0	0.2	-0.4	-0.1	0.8	0.1	1.1	0.1	0.8	-1.1	-0.3	1.1	-1.1	0.9	0.1	-1.1	-2.1	1.2	0.9	2.2	-2.1	-1.1	-0.4	1.4	-1.5	0.8	-1.3	-0.5	1.1	1.6	-1.9	-0.2
4	0.8	0.1	-0.6	-0.3	0.8	0.2	0.8	0.2	0.7	-0.5	-0.4	0.8	-1.2	0.6	0.0	-1.0	-1.9	1.3	0.7	1.7	-1.6	-0.8	-0.6	1.0	-0.9	0.8	-1.0	-0.2	0.7	1.3	-1.6	-0.6

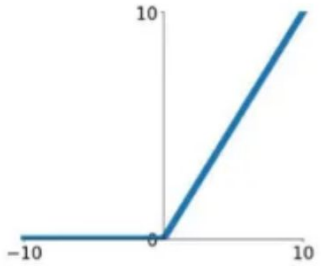
```
nn.Linear(n_embd, 4 * n_embd)
```

FFN: ReLU

- Rectified Linear Units

ReLU

$$\max(0, x)$$



Output after Relu (4, 32)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0.3	0.0	0.0	0.0	1.3	0.0	0.2	0.0	0.0	0.0	0.8	0.4	0.0	1.1	0.0	0.3	0.0	1.0	0.0	0.1	0.0	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.7	1.9	0.0	0.0
2	1.1	0.0	0.3	0.4	0.6	0.1	2.2	0.0	0.9	0.0	0.3	1.1	0.0	2.0	0.0	0.0	0.0	1.0	0.8	1.6	0.0	0.0	0.0	2.0	0.0	0.4	0.0	0.0	1.3	2.4	0.0	0.4
3	1.0	0.2	0.0	0.0	0.8	0.1	1.1	0.1	0.8	0.0	0.0	1.1	0.0	0.9	0.1	0.0	0.0	1.2	0.9	2.2	0.0	0.0	0.0	1.4	0.0	0.8	0.0	0.0	1.1	1.6	0.0	0.0
4	0.8	0.1	0.0	0.0	0.8	0.2	0.8	0.2	0.7	0.0	0.0	0.8	0.0	0.6	0.0	0.0	0.0	1.3	0.7	1.7	0.0	0.0	0.0	1.0	0.0	0.8	0.0	0.0	0.7	1.3	0.0	0.0

```
nn.ReLU()
```

FFN: Linear Layer 2

Output after Relu (4, 32)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0.3	0.0	0.0	0.0	1.3	0.0	0.2	0.0	0.0	0.0	0.8	0.4	0.0	1.1	0.0	0.3	0.0	1.0	0.0	0.1	0.0	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.7	1.9	0.0	0.0
2	1.1	0.0	0.3	0.4	0.6	0.1	2.2	0.0	0.9	0.0	0.3	1.1	0.0	2.0	0.0	0.0	0.0	1.0	0.8	1.6	0.0	0.0	0.0	2.0	0.0	0.4	0.0	0.0	1.3	2.4	0.0	0.4
3	1.0	0.2	0.0	0.0	0.8	0.1	1.1	0.1	0.8	0.0	0.0	1.1	0.0	0.9	0.1	0.0	0.0	1.2	0.9	2.2	0.0	0.0	0.0	1.4	0.0	0.8	0.0	0.0	1.1	1.6	0.0	0.0
4	0.8	0.1	0.0	0.0	0.8	0.2	0.8	0.2	0.7	0.0	0.0	0.8	0.0	0.6	0.0	0.0	0.0	1.3	0.7	1.7	0.0	0.0	0.0	1.0	0.0	0.8	0.0	0.0	0.7	1.3	0.0	0.0

@

Weights(32, 8)

	1	2	3	4	5	6	7	8
1	0.6	0.8	0.7	0.6	0.3	1	0.6	0.3
2	0.7	0.2	0.7	0.1	0.4	0.7	0.1	0.1
3	0.2	0.2	0.9	0.1	0.1	0.3	0.1	0.1
4	0.7	0.6	0	0.9	0.4	0	0	0.2
5	0	0	0.7	0.1	0.8	0.2	0.9	0.6
6	0.6	0.7	0.7	0.6	0	0.6	0.1	0.9
7	0	0.3	0.6	0	0.3	0.2	0.4	0.3
8	0	0.2	0.8	0.2	0.2	0.3	0.4	0.3
9	0.6	0.1	0.5	0.6	0.4	0.8	0.7	0.6
10	0.2	0.8	0.2	0.2	0.9	0.3	0.3	0.1
11	0.1	0.3	0.8	0.5	0.9	0	0.4	0.3
12	0.6	0.1	0	0.6	0.6	0.6	0.9	0.1
13	0.4	0.3	0	0.1	0.3	0.7	0.6	0.5
14	0.4	0.7	0.2	0.4	0.7	0.5	0.2	0.3
15	0.4	0.2	0.3	0.9	1	0.6	0.7	0.7
16	0.3	0.1	0	0.1	0.8	0.7	0.9	0.2
17	0.2	0.9	0.2	0.3	0.3	0.8	0.5	0.3
18	0.2	0.1	0.1	0.5	0.2	0.1	0.5	0.1
19	0.3	0.4	0	0.4	0.7	0.1	0.6	0.8
20	0.1	0.8	1	0.2	0.9	0.9	1	0.7
21	0.9	0.1	0.7	0.7	0.4	0	0.5	0.8
22	0.7	0.7	0.9	0.7	0.3	0.3	0	0.1
23	0.8	0.6	0.3	0.1	0.7	0.4	0.3	0.8
24	0.2	0.1	0.4	0.4	0.3	0.5	0.3	0.4
25	0.6	0.8	0.5	0.9	0.2	0.4	0.3	0.8
26	0.2	0.6	0.3	0.5	0.8	0.2	0.9	0.2
27	0.8	0.7	0.6	0.8	0.6	0.1	0.2	0.9
28	0.9	0.8	0.9	0.9	0.5	0.5	0.8	0.8
29	0.9	0.9	0.1	0.5	0.2	0	0.8	0.7
30	0.7	0.8	0.6	0	0.8	0.2	0.1	0.2
31	0.8	0.8	0.8	0.7	0.7	0.5	0.1	0.9
32	0.8	0.7	0.1	0.5	0.1	0.3	0.4	0.8

=

Output of Second Linear Layer (4, 8)

	1	2	3	4	5	6	7	8
1	4.5	5.31	3.85	3.58	5.93	2.37	5.69	3.85
2	7.1	8.69	8.04	6.38	9.58	7.21	8.83	7.51
3	5.48	6.96	6.98	5.5	8.58	6.64	9.03	6.56
4	4.28	5.49	5.67	4.41	6.84	5.17	7.4	5.27

```
nn.Linear(4 * n_embd, n_embd)
```

Dropout

- Some neurons are set randomly set to zero during each training pass
- Dropout does not occur during inference

Output of Second Linear Layer (4, 8)

	1	2	3	4	5	6	7	8
1	4.5	5.31	3.85	3.58	5.93	2.37	5.69	3.85
2	7.1	8.69	8.04	6.38	9.58	7.21	8.83	7.51
3	5.48	6.96	6.98	5.5	8.58	6.64	9.03	6.56
4	4.28	5.49	5.67	4.41	6.84	5.17	7.4	5.27

Output after Dropout

0	1	2	3	4	5	6	7	8
1	4.5	5.31	3.85	3.58	5.93	2.37	5.69	3.85
2	7.1	0	8.04	6.38	9.58	7.21	0	7.51
3	5.48	6.96	6.98	5.5	8.58	6.64	9.03	6.56
4	4.28	5.49	5.67	4.41	0	5.17	7.4	5.27

```
nn.Dropout(dropout)
```

Feed Forward Network (FFN)

```
#Initialize feed forward network
net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )
#Run feed forward network
net(x)
```

Output after Dropout

0	1	2	3	4	5	6	7	8
1	4.5	5.31	3.85	3.58	5.93	2.37	5.69	3.85
2	7.1	0	8.04	6.38	9.58	7.21	0	7.51
3	5.48	6.96	6.98	5.5	8.58	6.64	9.03	6.56
4	4.28	5.49	5.67	4.41	0	5.17	7.4	5.27

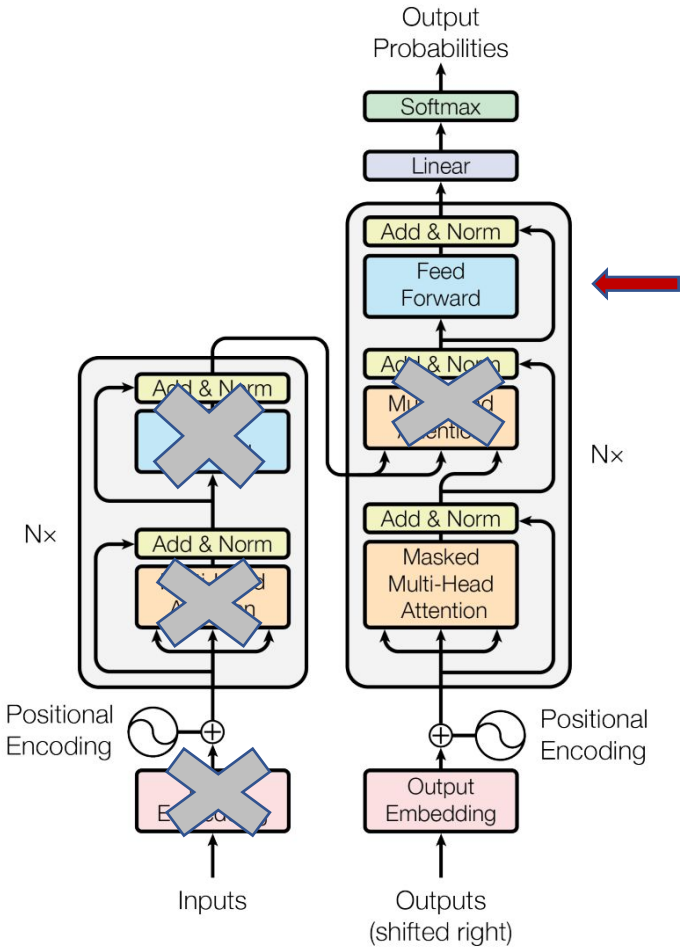


Figure 1: The Transformer - model architecture.

Linear Layer

Output after Final Layer Normalization (4, 8)

	0	1	2	3	4	5	6	7	8
1	0.11	0.84	-0.5	-0.7	1.41	-1.8	1.18	-0.5	
2	0.72	-3	1.21	0.34	2.02	0.77	-3	0.93	
3	-1.3	-0	0.02	-1.3	1.43	-0.3	1.82	-0.4	
4	-0.3	0.57	0.7	-0.2	-3.4	0.34	1.97	0.41	

@

Weights (8, 12)

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.19	0.91	0.16	0.65	0.52	0.31	0.83	0.96	0.09	0.93	0.08	0.76
2	0.22	0.77	0.9	0.14	0.83	0.85	0.62	0.11	0.32	0.6	0	0.05
3	0.69	0.06	0.51	0.94	0.13	0.67	0.6	0.37	0.41	0.75	0.69	0.85
4	0.79	0.16	0.88	0.95	0.32	0.34	0.61	0.19	0.68	0.91	0.03	0.56
5	0.19	0.2	0.51	0.35	0.28	0.47	0.49	0.2	0.96	0.5	1	0.72
6	0.78	0.31	0.63	0.56	0.44	0.76	0.54	0.75	0.63	0.71	0.65	0.72
7	0.19	0.76	0.03	0.26	0	0.31	0.43	0.92	0.32	0.87	0.49	0.19
8	0.76	0.38	0.98	0.45	0.34	0.41	0.24	0.25	0.61	0.83	0.55	0.77

Logits (4, 12)

	I	like	to	eat	soup	am	a	good	cook	.	is	yummy
1	-2	1.02	-1	-1.4	-0.1	-0.4	-0	-0.3	-0.14	-0.391	0.17	-1.147
2	1.7	-2.8	0.66	2.29	-0.6	-0.4	-0	-0.7	1.856	-0.176	2.45	3.779
3	-1.2	0.02	-1.1	-1.4	-0.9	0.03	-0.6	0.13	0.552	-0.6	1.79	-0.807
4	0.68	1.22	-0.5	-0	-0.4	0.18	-0.2	1.45	-1.91	0.9294	-1.6	-1.302

```
self.lm_head = nn.Linear(n_embd, vocab_size)
logits = self.lm_head(x)
```

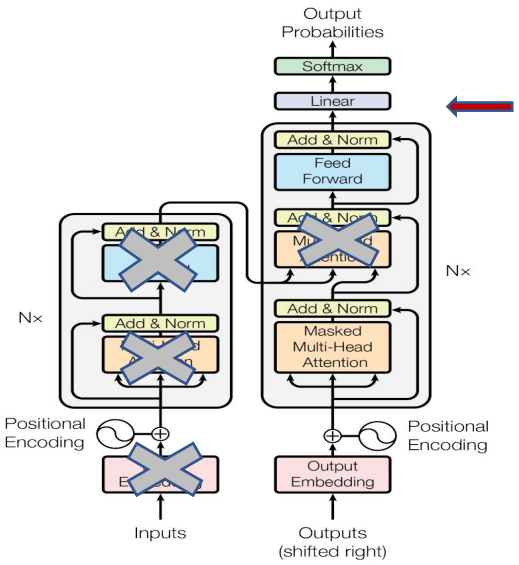


Figure 1: The Transformer - model architecture.

Softmax to Get Probabilities

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Logits (4, 12)

	I	like	to	eat	soup	am	a	good	cook	.	is	yummy
1	-2	1.02	-1	-1.4	-0.1	-0.4	-0	-0.3	-0.14	-0.391	0.17	-1.147
2	1.7	-2.8	0.66	2.29	-0.6	-0.4	-0	-0.7	1.856	-0.176	2.45	3.779
3	-1.2	0.02	-1.1	-1.4	-0.9	0.03	-0.6	0.13	0.552	-0.6	1.79	-0.807
4	0.68	1.22	-0.5	-0	-0.4	0.18	-0.2	1.45	-1.91	0.9294	-1.6	-1.302

Probabilities (4, 12)

	I	like	to	eat	soup	am	a	good	cook	.	is	yummy	SUM
I	0.01	**0.28**	0.04	0.02	0.09	0.07	0.1	0.08	0.088	0.0685	0.12	0.032	**1**
like	0.07	0	0.02	0.12	0.01	0.01	0.01	0.01	0.077	0.0101	0.14	**0.529**	**1**
to	0.02	0.07	0.02	0.02	0.03	0.08	0.04	0.08	**0.126**	0.0399	0.44	0.032	**1**
eat	0.11	0.2	0.04	0.06	0.04	0.07	0.05	**0.25**	0.009	0.1476	0.01	0.016	**1**

```
# apply softmax to get probabilities
probs = F.softmax(logits, dim=-1)
```

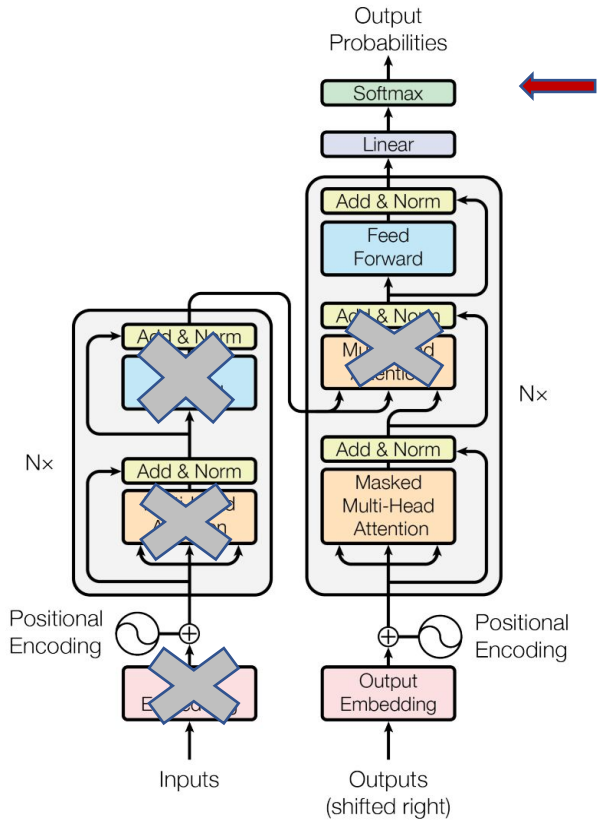


Figure 1: The Transformer - model architecture.

Predict What Comes Next!

Probabilities (4, 12)

	I	like	to	eat	soup	am	a	good	cook	.	is	yummy	SUM
I	0.01	**0.28**	0.04	0.02	0.09	0.07	0.1	0.08	0.088	0.0685	0.12	0.032	**1**
like	0.07	0	0.02	0.12	0.01	0.01	0.01	0.01	0.077	0.0101	0.14	**0.529**	**1**
to	0.02	0.07	0.02	0.02	0.03	0.08	0.04	0.08	**0.126**	0.0399	0.44	0.032	**1**
eat	0.11	0.2	0.04	0.06	0.04	0.07	0.05	**0.25**	0.009	0.1476	0.01	0.016	**1**

```
# Sample from a distribution
idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
```

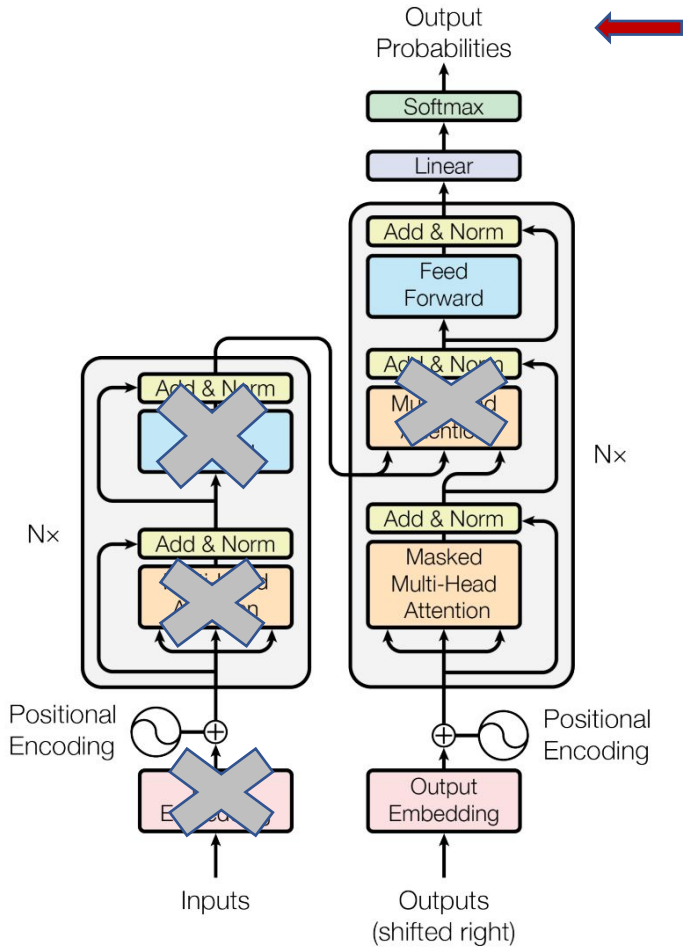


Figure 1: The Transformer - model architecture.

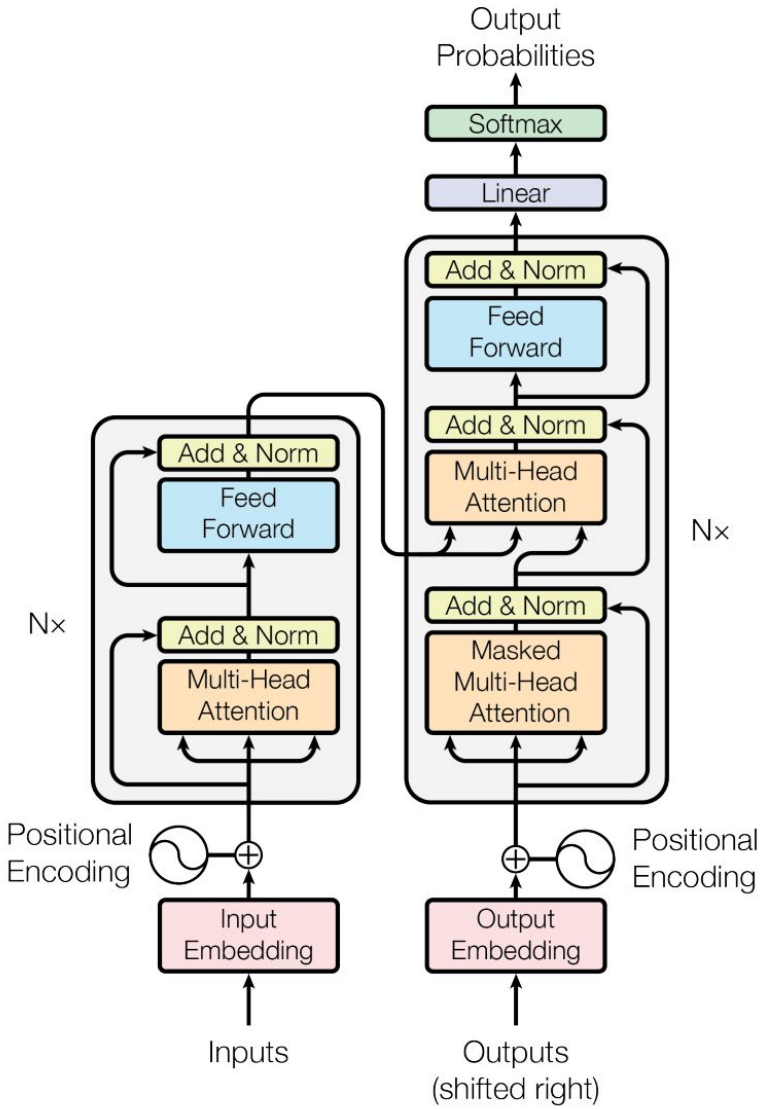


Figure 1: The Transformer - model architecture.

Encoders vs Decoders

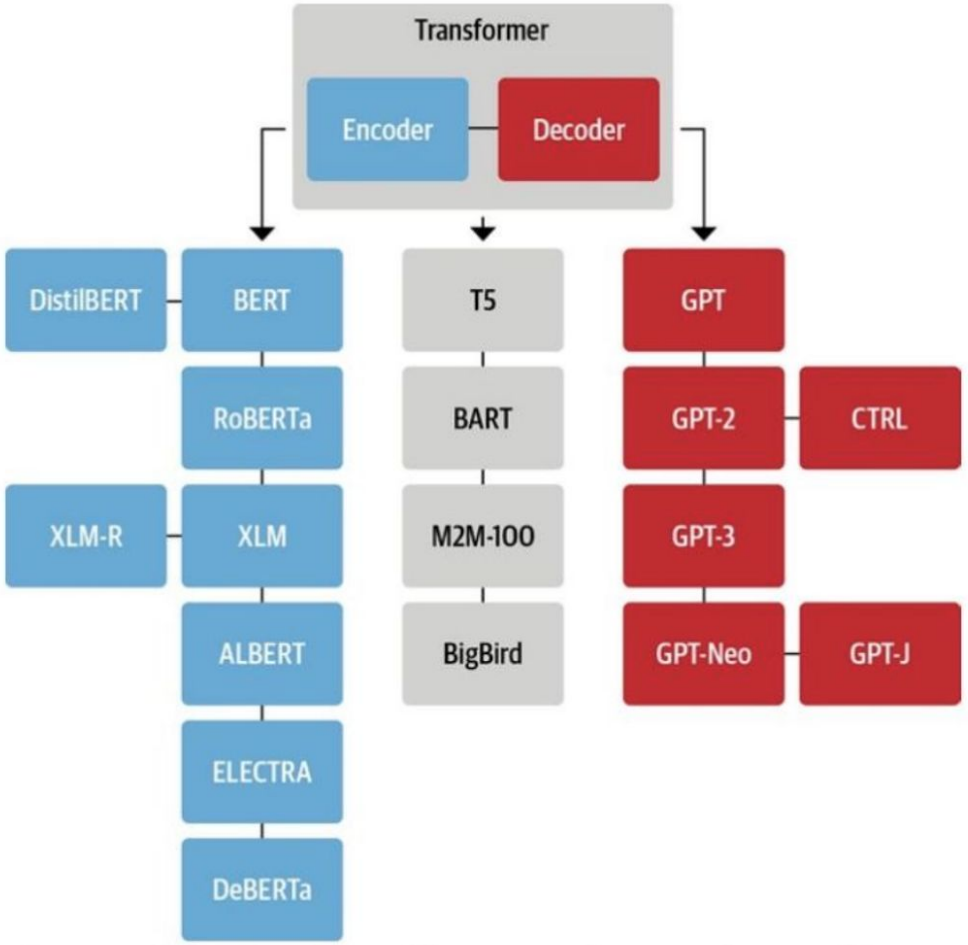


Figure 3-8. An overview of some of the most prominent transformer architectures

Language Models are REALLY Big

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

https://arxiv.org/pdf/2005.14165.pdf

Resources

- Andrej Kaparthy's YouTube video:
  - Let's build GPT: from scratch, in code, spelled out.
- Andrej Kaparthy's Colab Notebook:
  - Building a GPT
- The famous paper with the Transformer Architecture:
  - Attention is All You Need
- GPT Papers
  - Language Models are Few Shot Learners (GPT-3)
  - Language Models are Unsupervised Multitask Learners (GPT-2)
- YouTube video explaining Self-Attention:
  - Intuition Behind Self Attention in Transformer Networks
- Helpful blog post with matrix diagrams:
  - Step-by-Step Illustrated Explanations for Transformer
- Stanford lecture on word vectors:
  - Stanford CS224N: NLP with Deep Learning | Winter 2021 | Lecture 1 - Intro & Word Vectors