Understanding Transformers

fzeng

03/01/24

Outline

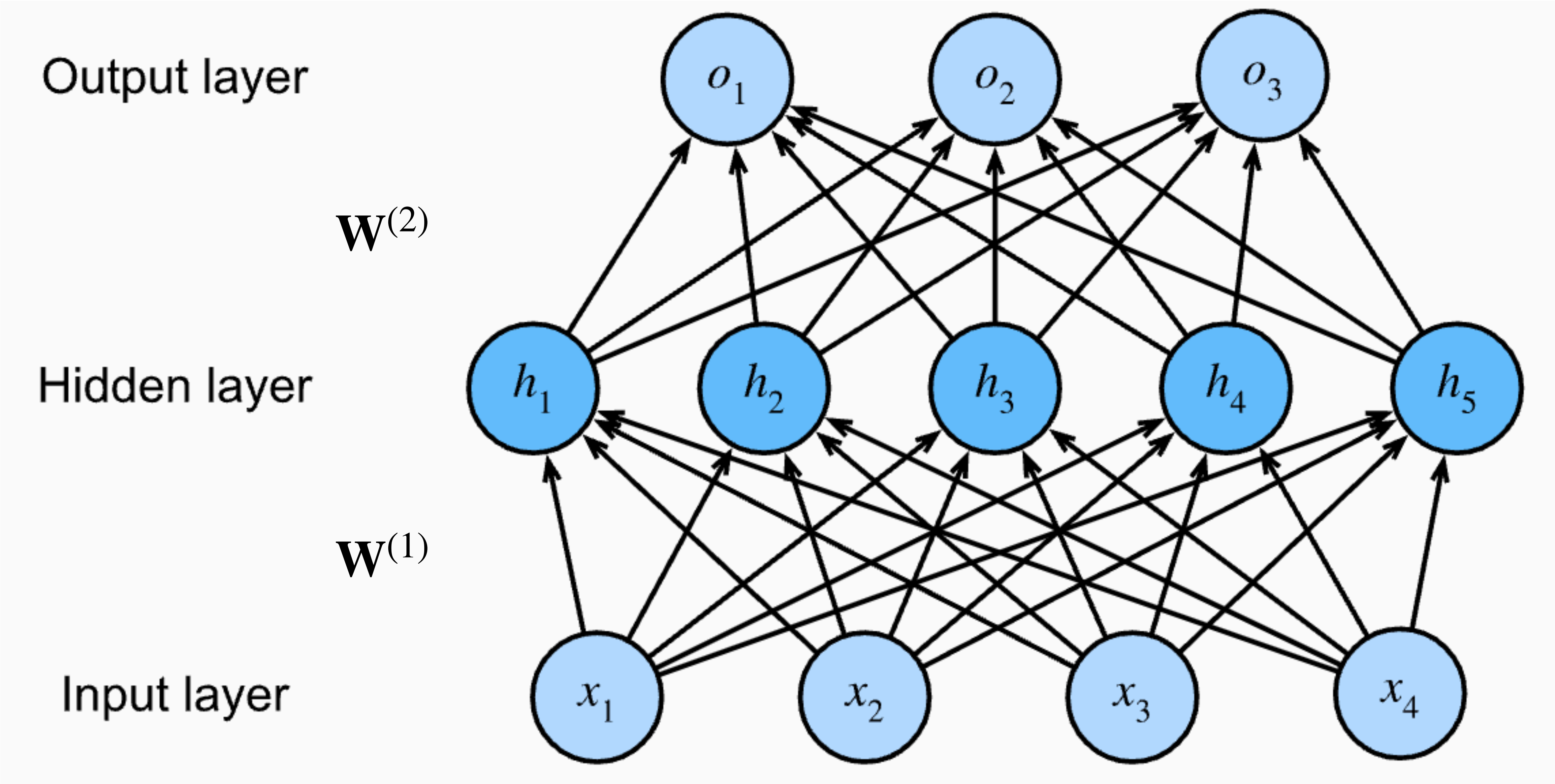
- Part 1 (today)
  - Neural Networks
  - Language Modeling
  - Sampling
  - Recurrent Neural Networks
  - Milestones in Language Modeling
- Part 2
  - Transformers
    - This has 9 sections under it don't worry

Neural Networks

Neural Network with 1 Hidden Layer

(aka Feedforward Neural Networks)

- Sample task: inputs $\mathbf{X} \in \mathbb{R}^4$, outputs $f(\mathbf{X}) \in \mathbb{R}^3$
- $\mathbf{W}^{(1)} \in \mathbb{R}^{4\times5}, \mathbf{W}^{(2)} \in \mathbb{R}^{5\times3}$: weight matrices
- $\mathbf{b}^{(1)} \in \mathbb{R}^5, \mathbf{b}^{(2)} \in \mathbb{R}^3$: bias vectors
- σ: activation function
- Forward pass:
  - Compute hidden layer with activations $\mathbf{H} = \sigma\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)$
  - Compute output layer for predictions $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$
- Neural network computes: $f(x) = \sigma\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$

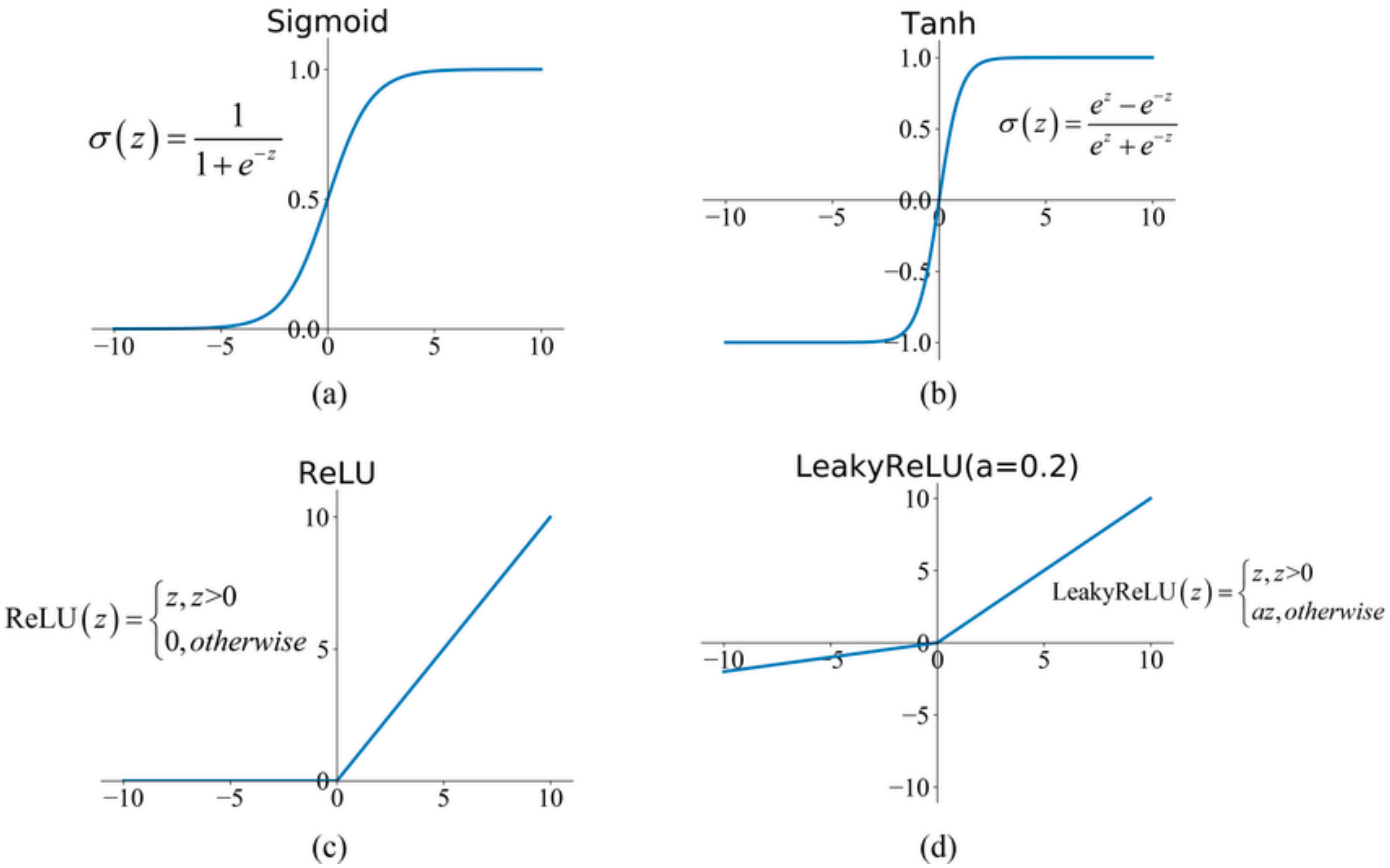


Activation Functions

- Idea: introduce non-linearity between the layers to make model more expressive
- Q: What if we didn't have activation functions? I.e

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

- Becomes linear regression!



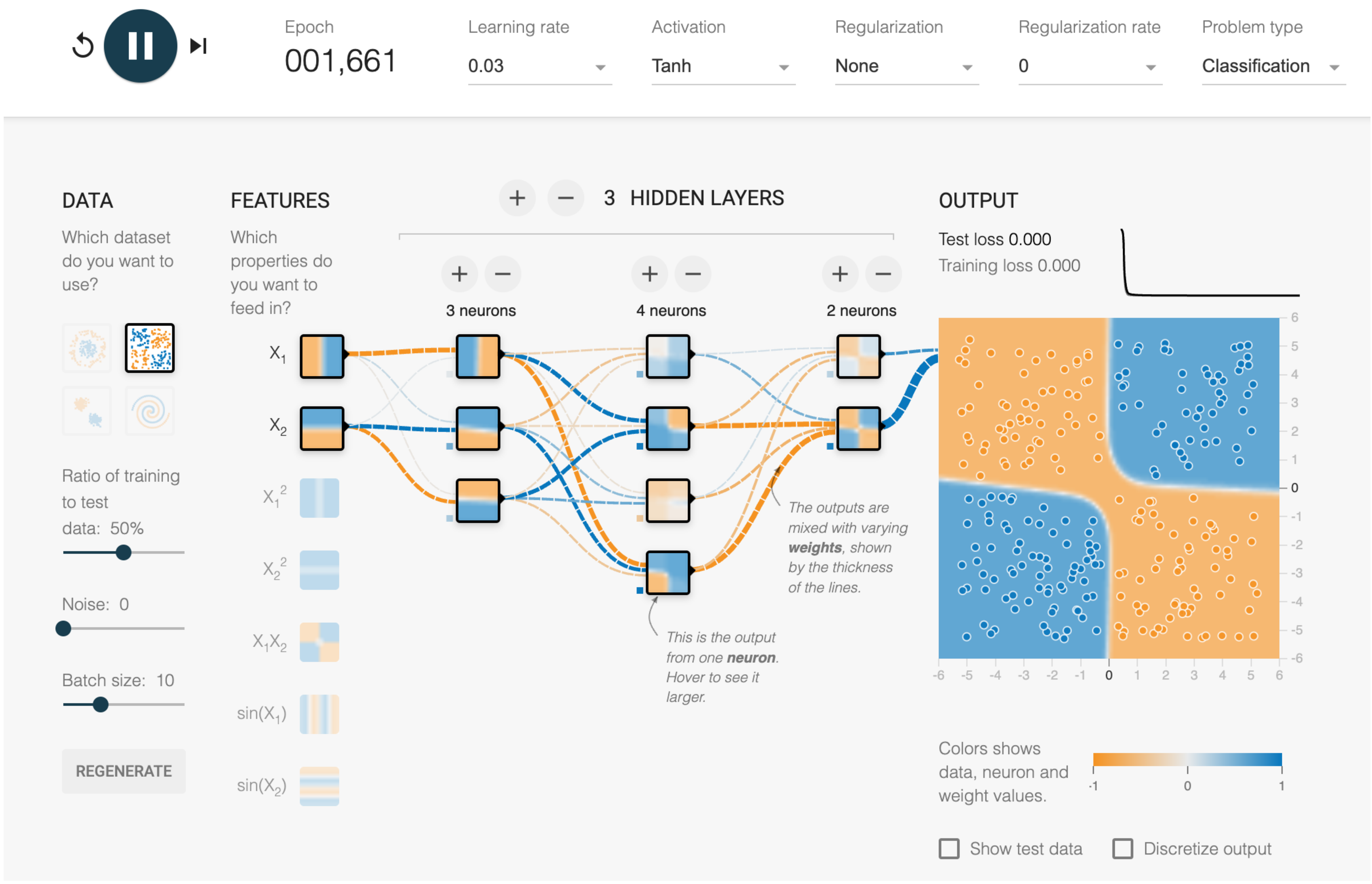
Common activation functions

Training: Loss Function

- The 7 Habits of Highly Effective People, Habit 2: Begin With the End in Mind
- What do you want your model to optimize for?
- Define objective/loss function $L(f(x), y)$
- Optimization algorithms typically minimize, so formulation should be lower loss => better model
- Common choices:
  - Regression: Mean square error (MSE)/L2 loss: $||f(x) - y||^2$
  - Classification: Cross entropy loss. Binary 0/1 labels version: $-(y \log f(x) + (1 - y)\log(1 - f(x)))$

Visualization

- https://playground.tensorflow.org/



Epoch 001,661

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

+ − 3 HIDDEN LAYERS

+ − 3 neurons

+ − 4 neurons

+ − 2 neurons

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

*This is the output from one **neuron**. Hover to see it larger.*

OUTPUT

Test loss 0.000

Training loss 0.000

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

Wait, this actually works?

- How can something this simple give rise to ChatGPT and other ML systems?
- Universal approximation theorem (Hornik et al. 1989, Cybenko 1989):

  A feedforward neural network with at least a single hidden layer, sufficient hidden units, and a linear output layer with any "squashing" activation function (i.e sigmoid activation) can approximate any function with any desired non-zero error*

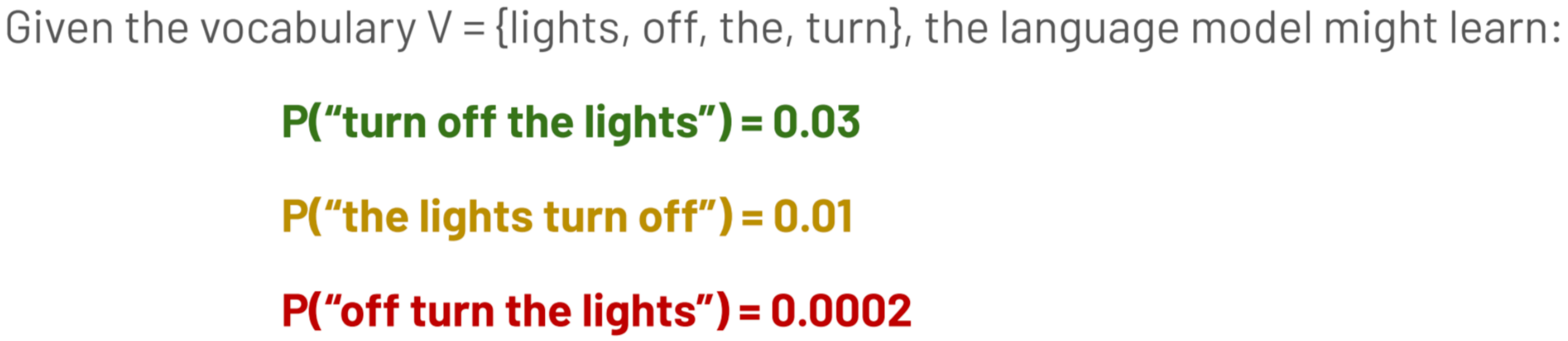
* to be precise: any Borel-measurable function

Language Modeling

Language Modeling

- Want a generative model for text
- Need to be able to sample from model *efficiently*
- Need to be able to learn the parameters of the model *efficiently*
- Suppose you have a string $\mathbf{x}$ of length L, could formulate probability as: $p(\mathbf{x}_{1..L})$

Language Modeling



Given the vocabulary V = {lights, off, the, turn}, the language model might learn:

P("turn off the lights") = 0.03

P("the lights turn off") = 0.01

P("off turn the lights") = 0.0002

Example from Simran Arora

- Problem of modeling this directly: difficult to model joint probabilities, may want to sample varying lengths

Autoregressive Language Modeling

- From the chain rule of probability:

$$p(\mathbf{x}_{1..L}) = p(\mathbf{x}_1)p(\mathbf{x}_2 \mid \mathbf{x}_1)\cdots p(\mathbf{x}_{\mathbf{L}} \mid \mathbf{x}_1 \cdots \mathbf{x}_{L-1})$$

$$= \prod_{i=1}^{L} p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})$$

```
P([“I”, “eat”, “the”, “apple”]) =

P(“apple” | [“I”, “eat”, “the”]) * P(“the” | [“I”, “eat”]) * P(“eat” | [“I”]) * P(“I”])
```

- Problem factorizes into next-token prediction!
  - Predict first token
  - Predict second token given first token
  - Predict third token given first two tokens
  - ...and so on

Autoregressive Language Modeling

- $p(\mathbf{x}_{1..L}) = \prod_{i=1}^{L} p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})$
- In practice people use log probabilities for numerical stability:

  $$\log p(\mathbf{x}_{1..L}) = \sum_{i=1}^{L} \log p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})$$

- To train language models: set loss function to minimize cross-entropy loss between predictions and ground-truth tokens

Perplexity

- A good language model should assign high probabilities to likely sequences and vice versa
- How to measure performance?
- Best case: evaluations for the task you care about
- What if you have no evals?
- Strawman approach: compute probability of sequences on test set
- Problem: shorter sequences have higher probability than longer sequences
- Want something independent of length!

Perplexity

- Take average of predicting the next token:

$$\frac{1}{L}\sum_{i=1}^{L}\log p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})$$

- We minimize in optimization:

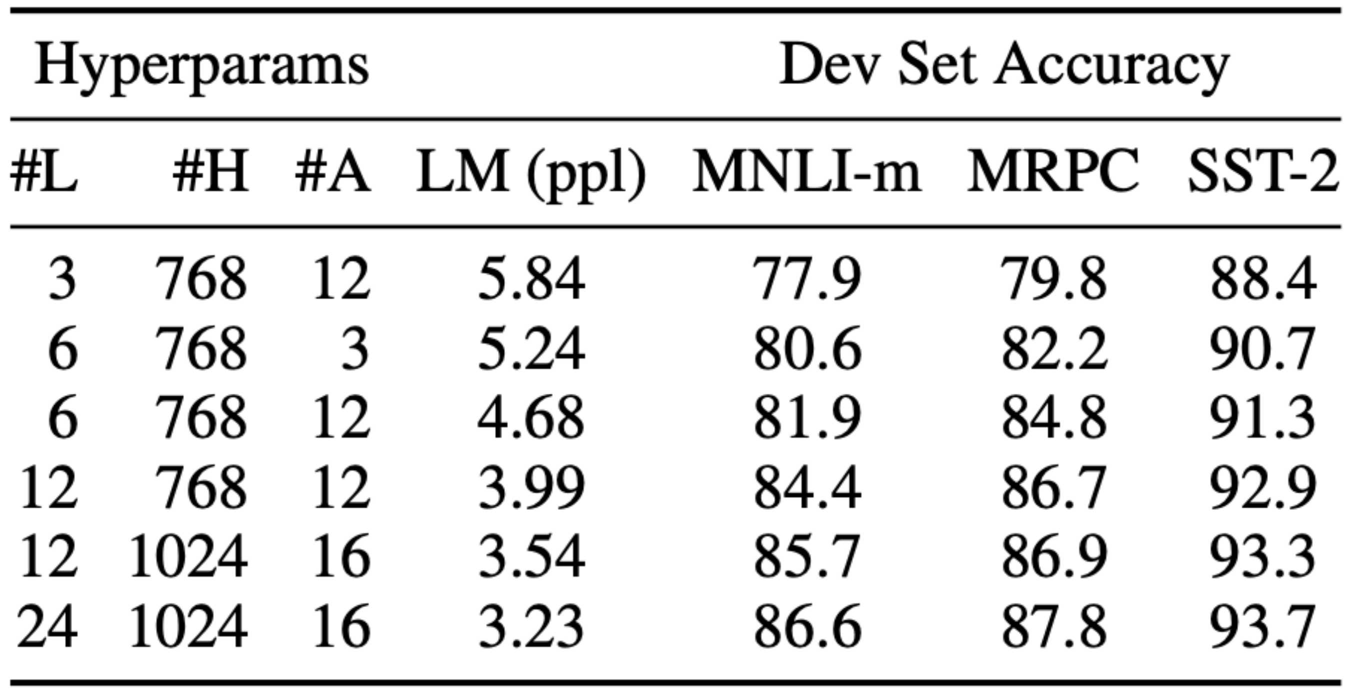
$$-\frac{1}{L}\sum_{i=1}^{L}\log p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})$$

- For historical reasons, people take the exponential, which is called perplexity:

$$\text{Perplexity}(\mathbf{x}) = \exp\left(-\frac{1}{L}\sum_{i=1}^{L}\log p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})\right)$$

Perplexity

- Perplexity$(\mathbf{x}) = \exp\left(-\frac{1}{L}\sum_{i=1}^{L}\log p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1})\right)$
- Can also be expressed as a geometric mean: Perplexity$(\mathbf{x}) = p(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_L)^{-\frac{1}{L}}$
- What if your model is perfect? Then $p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1}) = 1$ always, so perplexity is 1
- If your model always predicts next token wrongly, then $p(\mathbf{x}_i \mid \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1}) = 0$ always, and perplexity is infinity
- When comparing perplexity across models, ensure same test set is used!



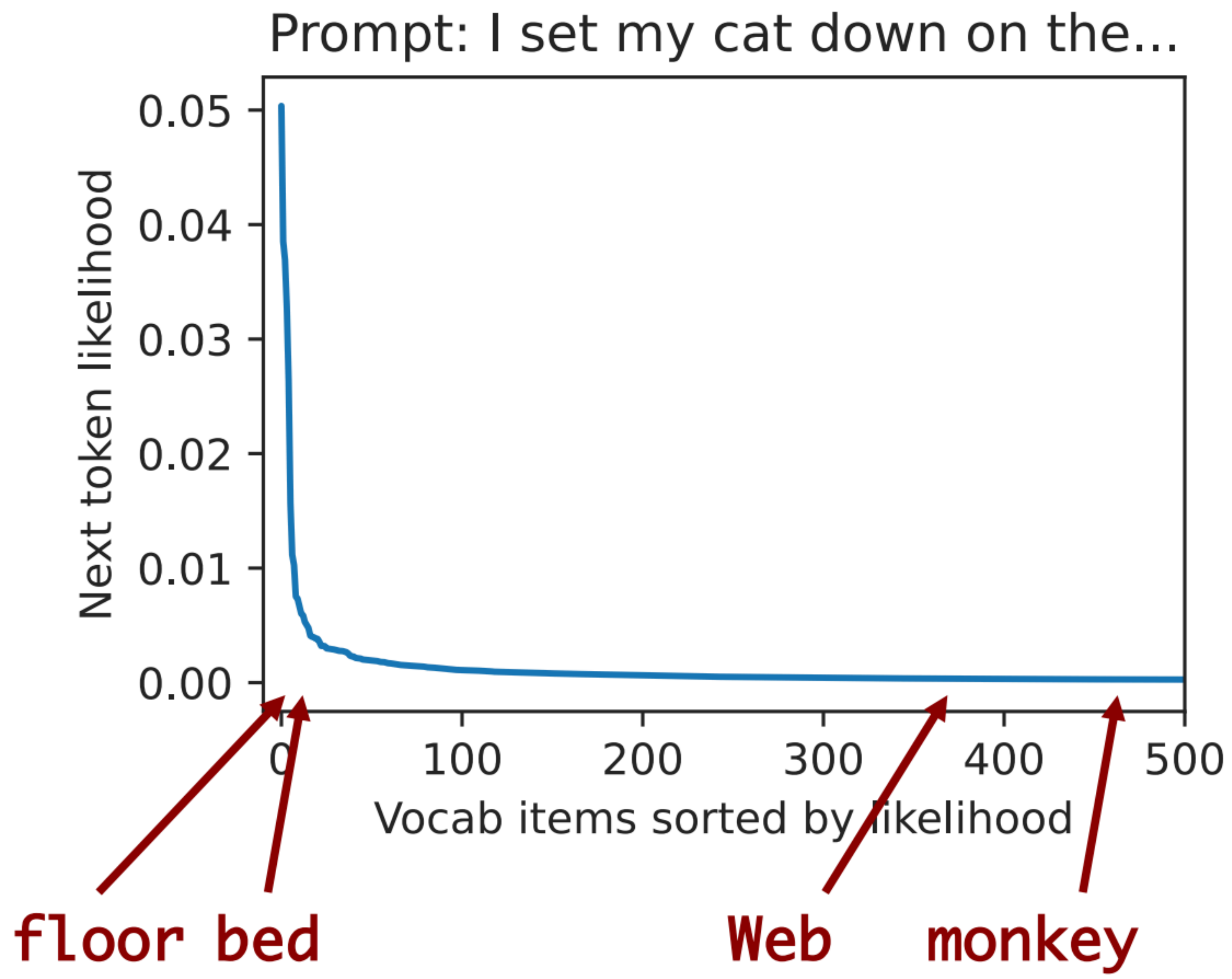
Hyperparams				Dev Set Accuracy		
#L	#H	#A	LM (ppl)	MNLI-m	MRPC	SST-2
3	768	12	5.84	77.9	79.8	88.4
6	768	3	5.24	80.6	82.2	90.7
6	768	12	4.68	81.9	84.8	91.3
12	768	12	3.99	84.4	86.7	92.9
12	1024	16	3.54	85.7	86.9	93.3
24	1024	16	3.23	86.6	87.8	93.7

Table 6: Ablation over BERT model size. #L = the number of layers; #H = hidden size; #A = number of attention heads. "LM (ppl)" is the masked LM perplexity of held-out training data.

Sampling

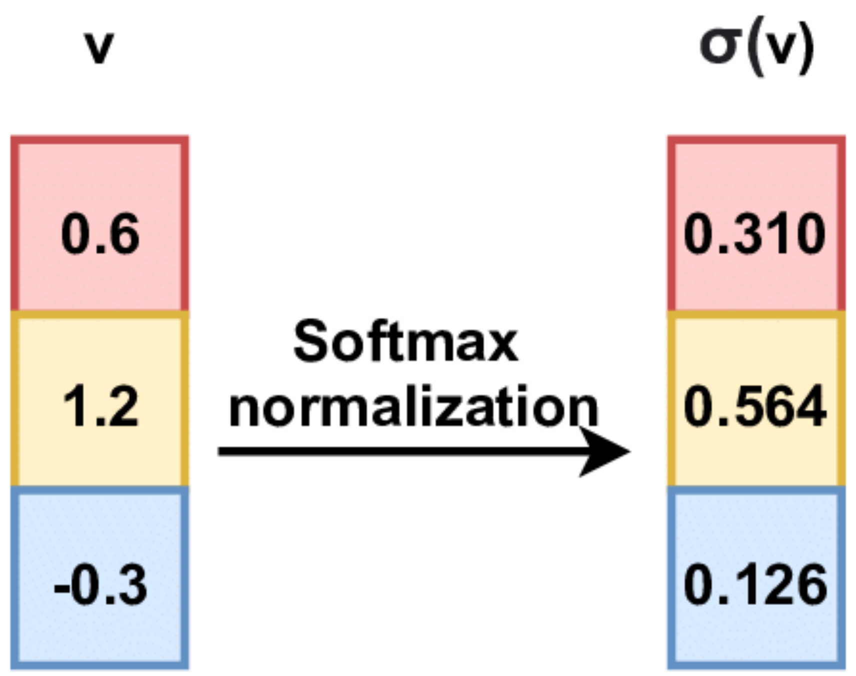
Sampling

- I trained my language model, I can just perform next token sampling from it now right?
- No: even though most tokens have low probability, cumulatively sampling a low probability token is very likely
- 29% chance of choosing token with probability < 0.01 on right



Sampling: Temperature

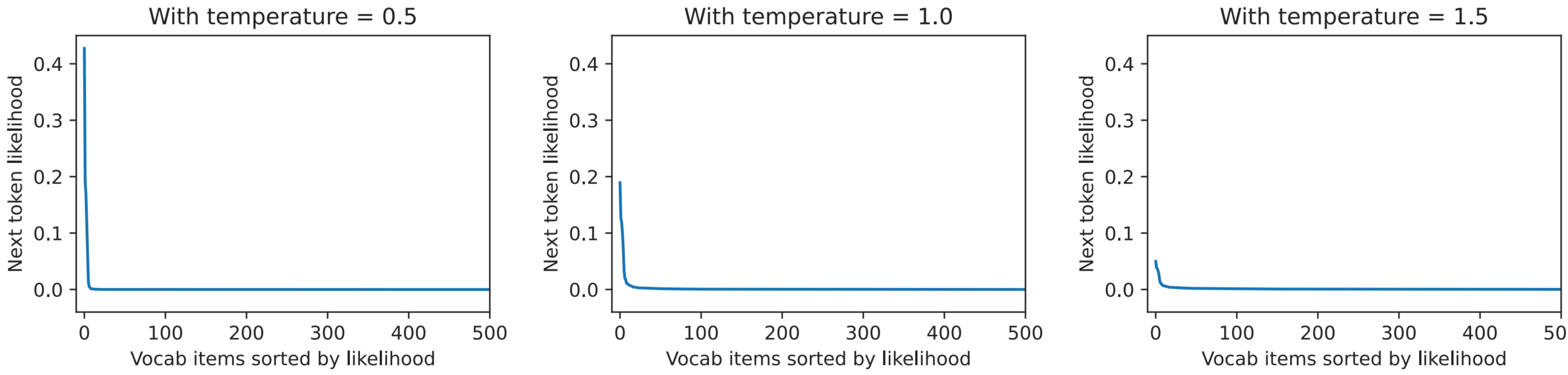
- Scale probabilities by temperature
- Final layer of model uses softmax to convert predictions to a probability distribution:
- $\sigma(z_i) = \dfrac{\exp(z_i)}{\sum_{j=0}^{N} \exp(z_j)}$
- Divide logits by temperature: $P(\mathbf{X}_t = i) = \dfrac{\exp(z_i/T)}{\sum_{j=0}^{N} \exp(z_j/T)}$



v

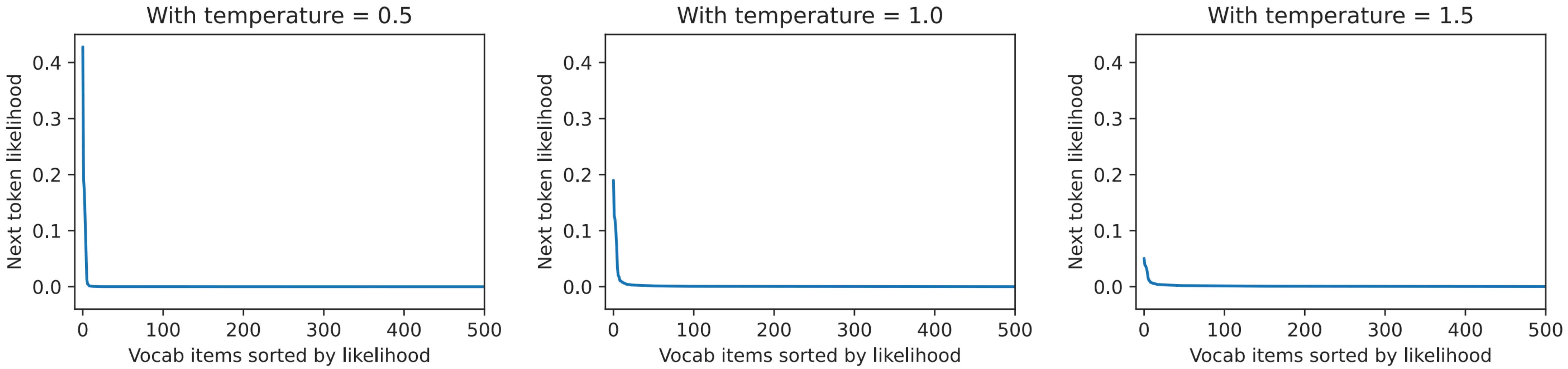
v	σ(v)
0.6	0.310
1.2	0.564
-0.3	0.126

Softmax normalization



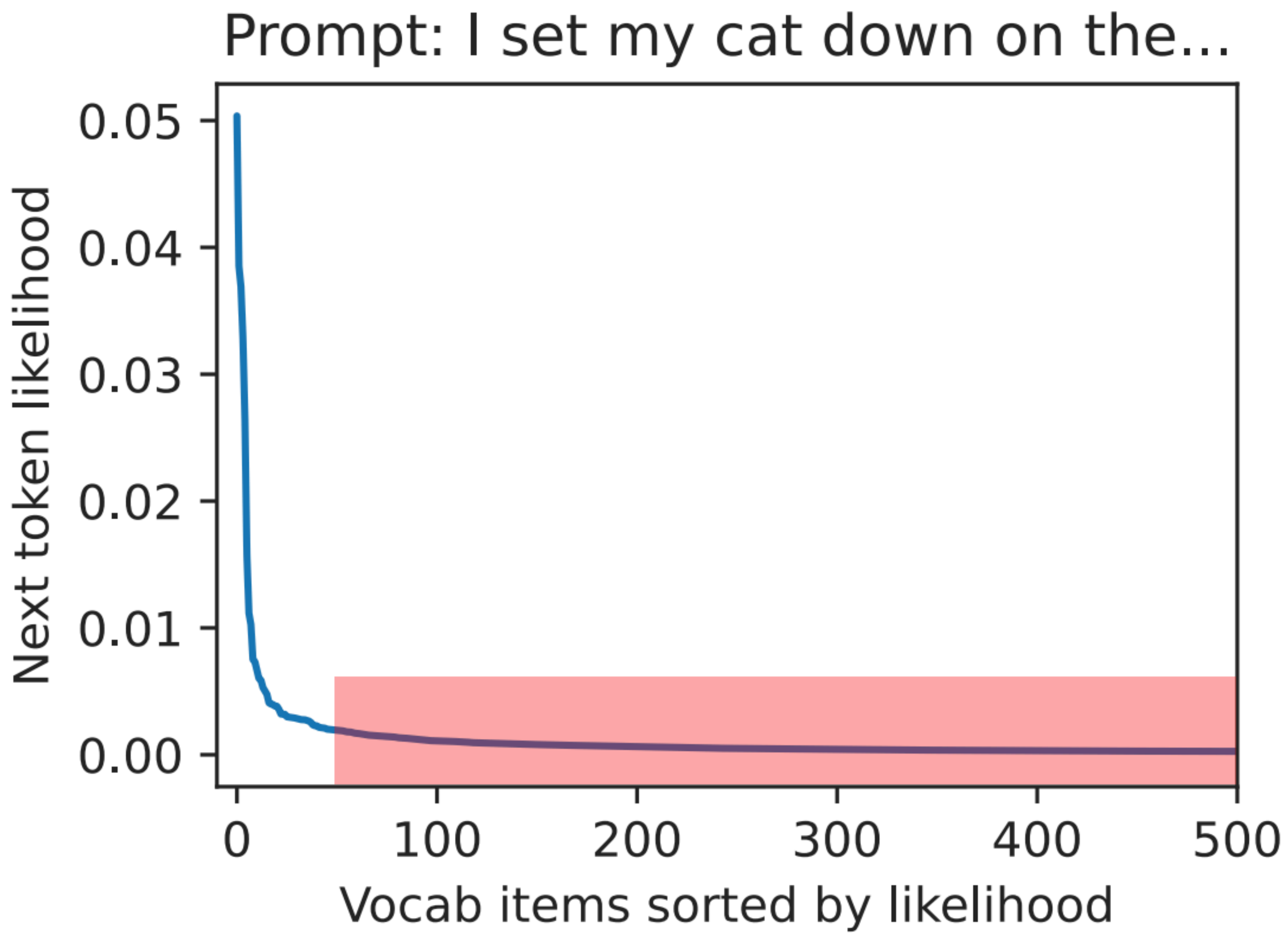
Sampling: Temperature

- $$P(\mathbf{X}_t = i) = \frac{\exp(z_i/T)}{\sum_{j=0}^{N} \exp(z_j/T)}$$
- Lower temperature, distribution is sharper => more conservative outputs
  - T=0: argmax
- Higher temperature, distribution is flatter => more diverse outputs



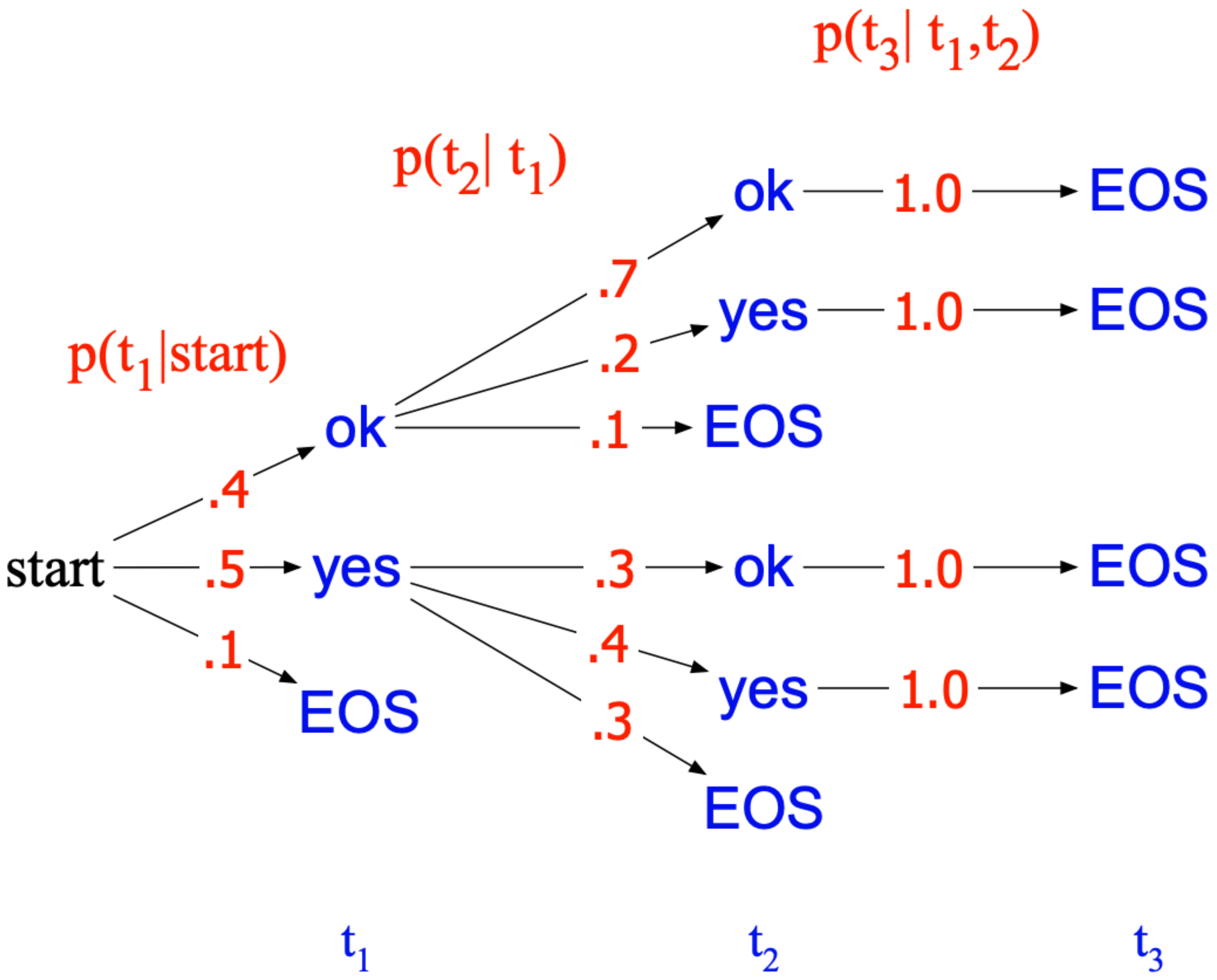
Top-k Sampling

- Another strategy: set probabilities of all other tokens except top-k to 0, then renormalize distribution
- Usually k set between 10 to 50



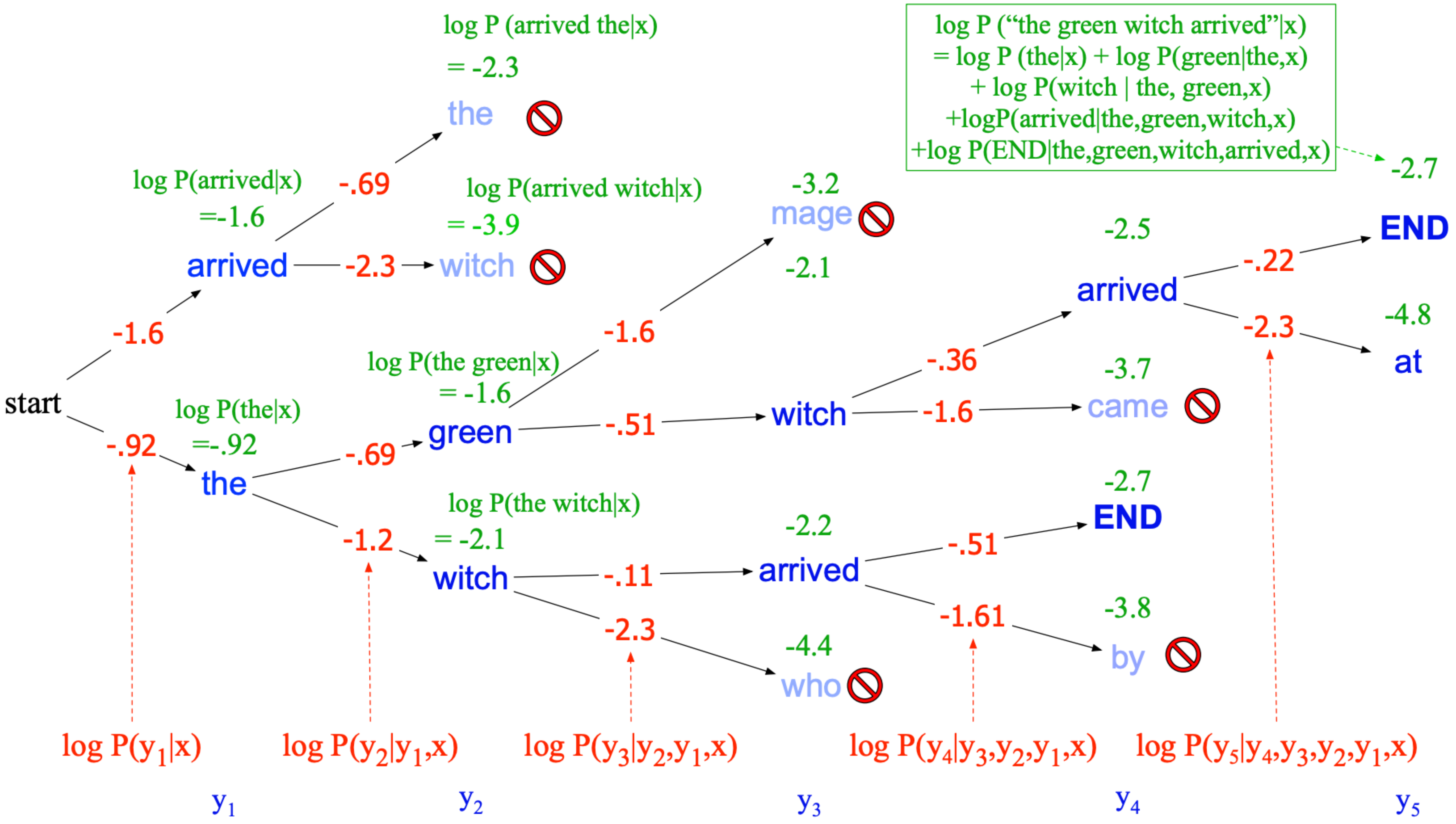
Beam Search

- Greedy (i.e argmax or T=0) search does not result in the overall most likely sequence
- Current best token may not be good in the long run
- Right: most likely sequence is “ok ok EOS” (.4 * .7 * 1 = 0.28), but greedy decoding gives “yes ok EOS” (.5 * .3 * 1 = 0.15)



Beam Search

- Instead: perform a search, keeping the top k most promising "beams" with some branching factor
- Right: $k = 2$



Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

- State-of-the-art architecture for sequence modeling prior to Transformers
- Difference from normal neural networks: input can be sequence of arbitrary length
- Idea: carry a hidden state that is updated while processing inputs to keep track of history

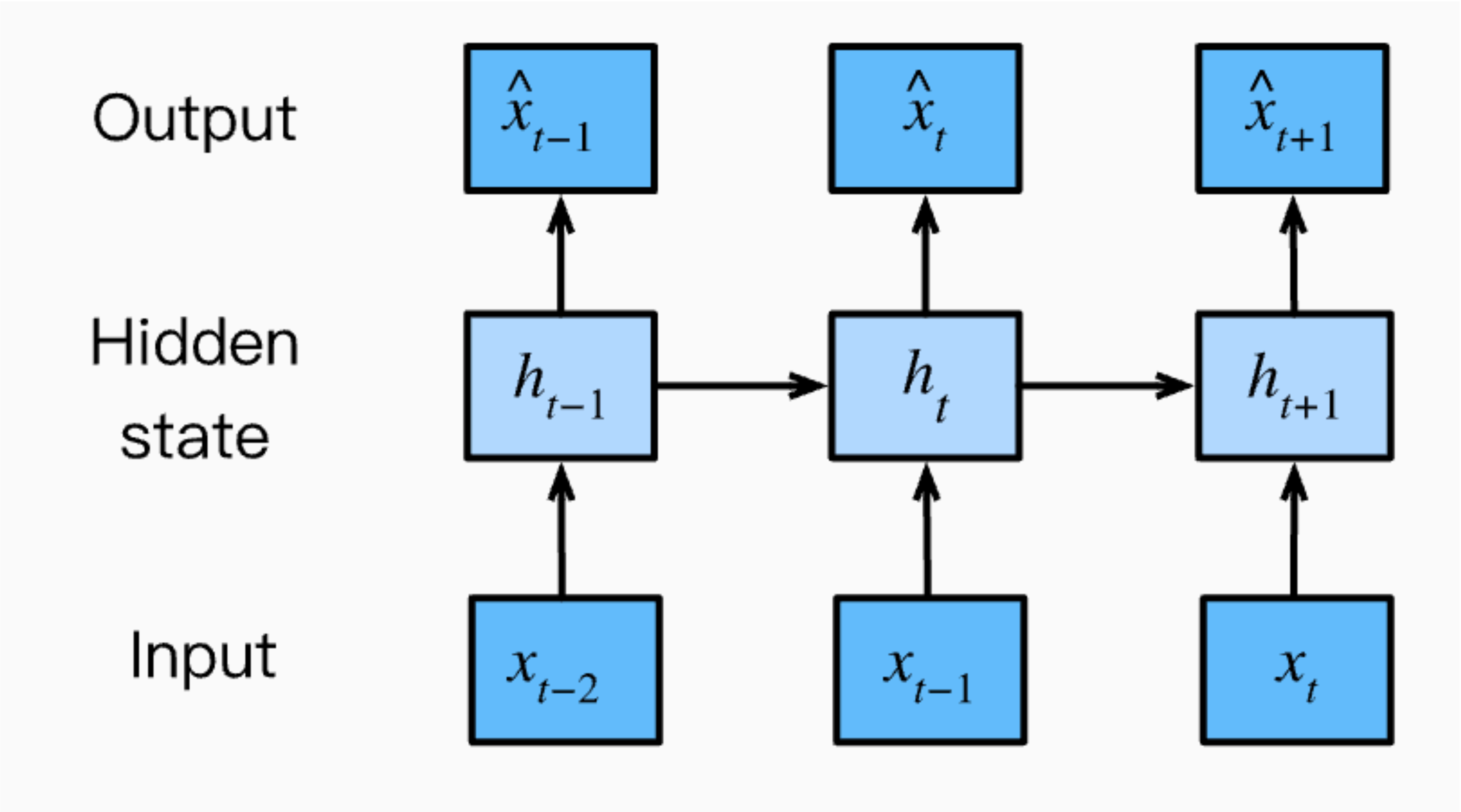


Figure from Dive into Deep Learning

Recurrent Neural Networks

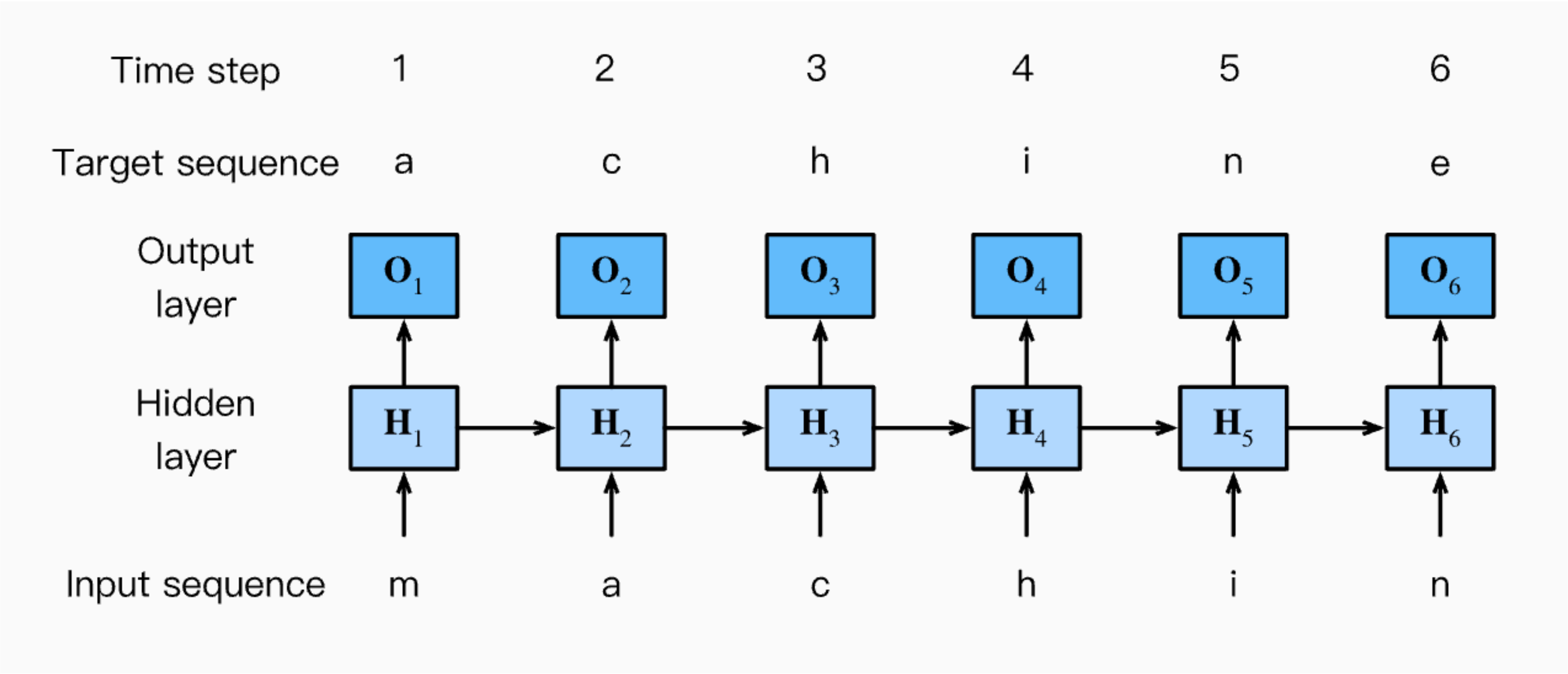
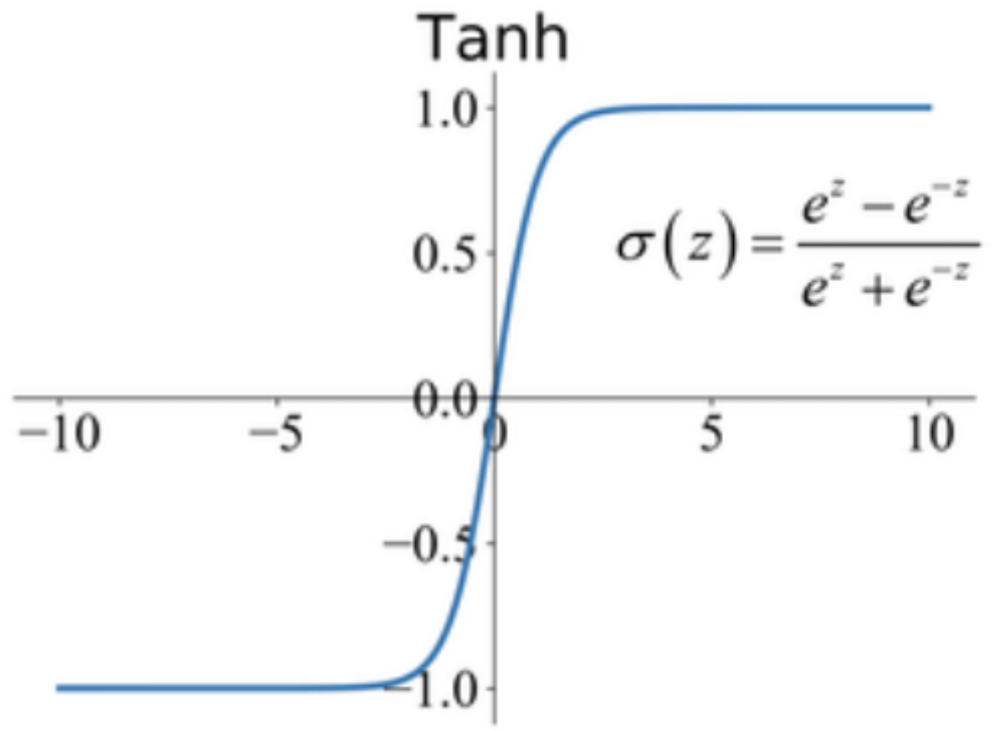


Figure from Dive into Deep Learning

Recurrent Neural Networks



- RNN update:

$$\boldsymbol{h}^{(t)} = \tanh\left(\boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}\right)$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{V}\boldsymbol{h}^{(t)}$$

- L: loss function
- y: ground-truth target
- Training: learn $\mathbf{U}, \mathbf{V}, \mathbf{W}$

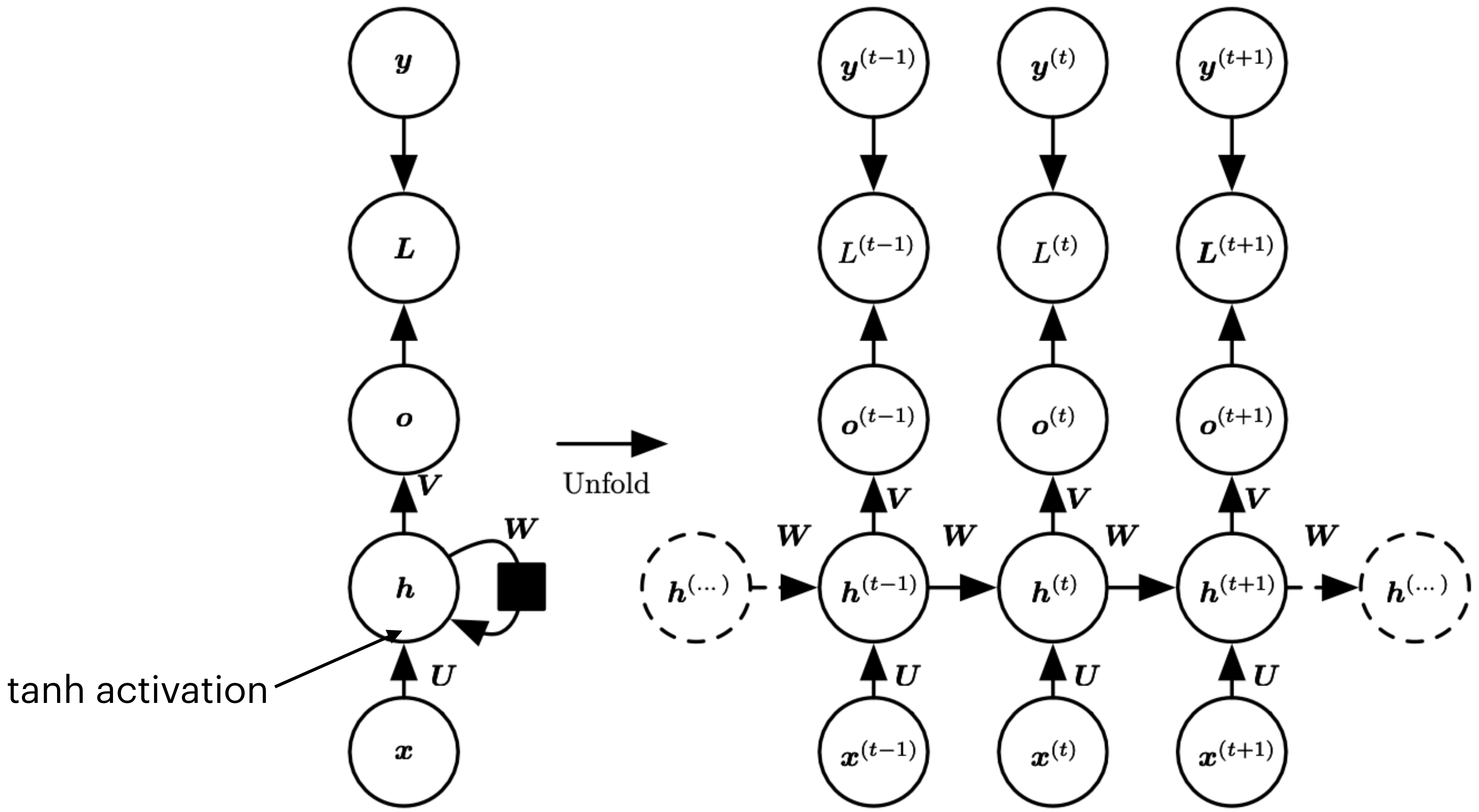


Diagram from Deep Learning (Goodfellow et al.)

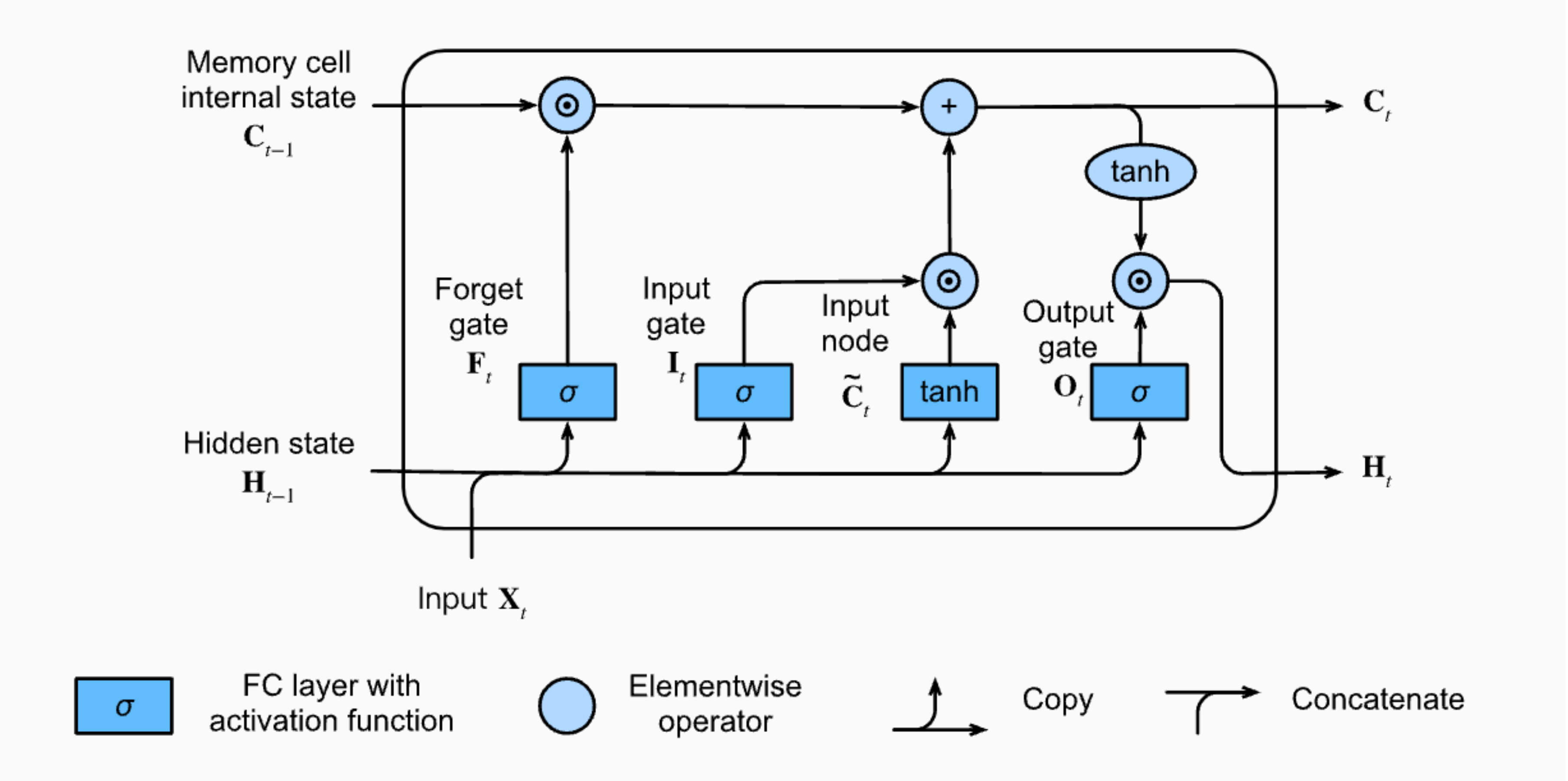
Limitations of RNNs

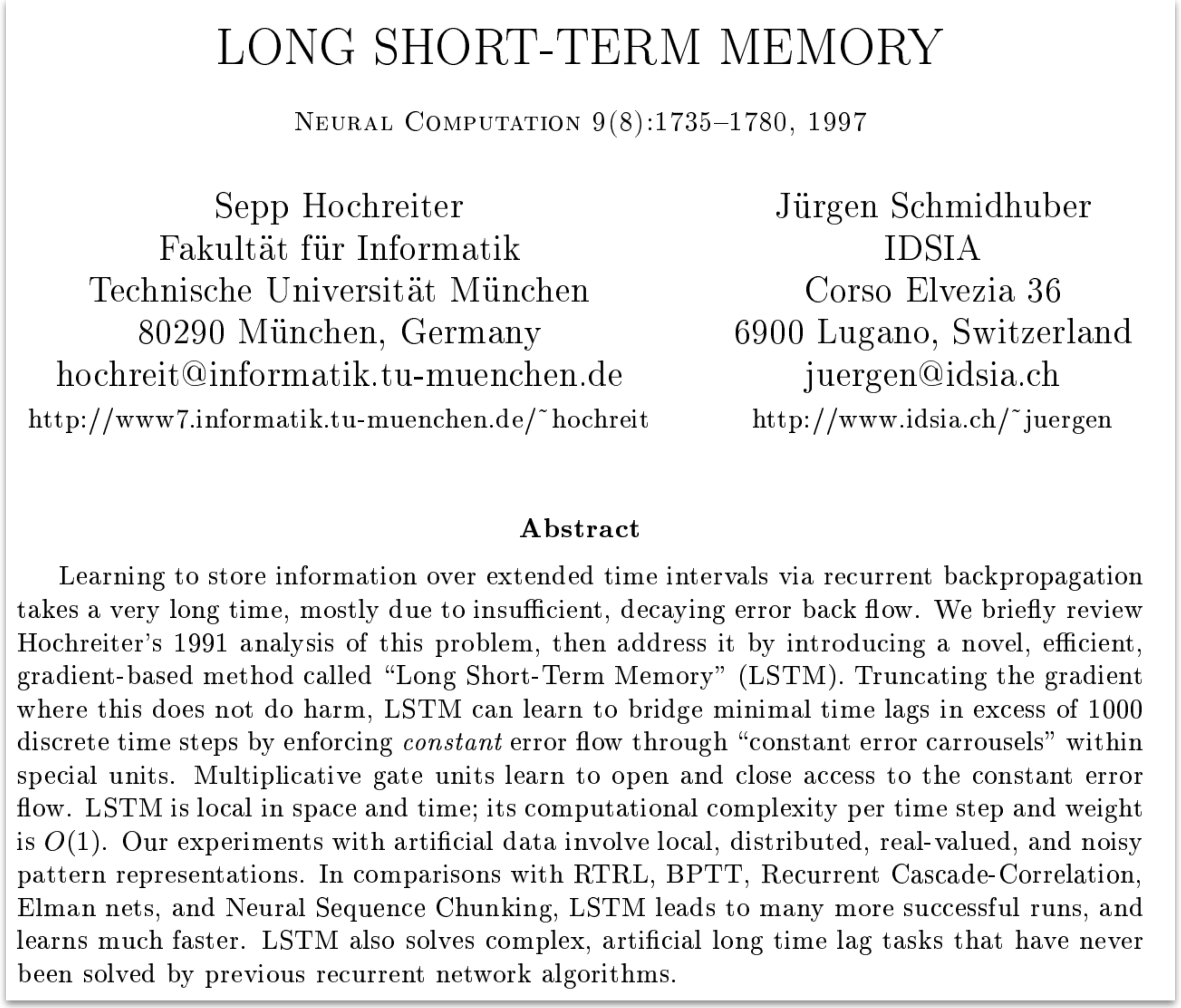
- Information bottleneck in the hidden state: difficult to model long-range dependencies for long sequences
- Vanishing gradient/exploding gradient problem: recurrent computation means that if gradients are <1 or >1 they are repeatedly multiplied against itself and goes to 0 or infinity respectively
- Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRU) tries to address previous limitations
- **But still slow to train as RNN computation is inherently sequential => can’t scale**

Milestones in Language Modeling

1997: LSTMs introduced

- Allowed RNNs to model long-range structure
- Achieved by adding a memory cell that can carry state over long sequences (if left unchanged)





LONG SHORT-TERM MEMORY

Neural Computation 9(8):1735–1780, 1997

Sepp Hochreiter
Fakultät für Informatik
Technische Universität München
80290 München, Germany
hochreit@informatik.tu-muenchen.de
http://www7.informatik.tu-muenchen.de/~hochreit

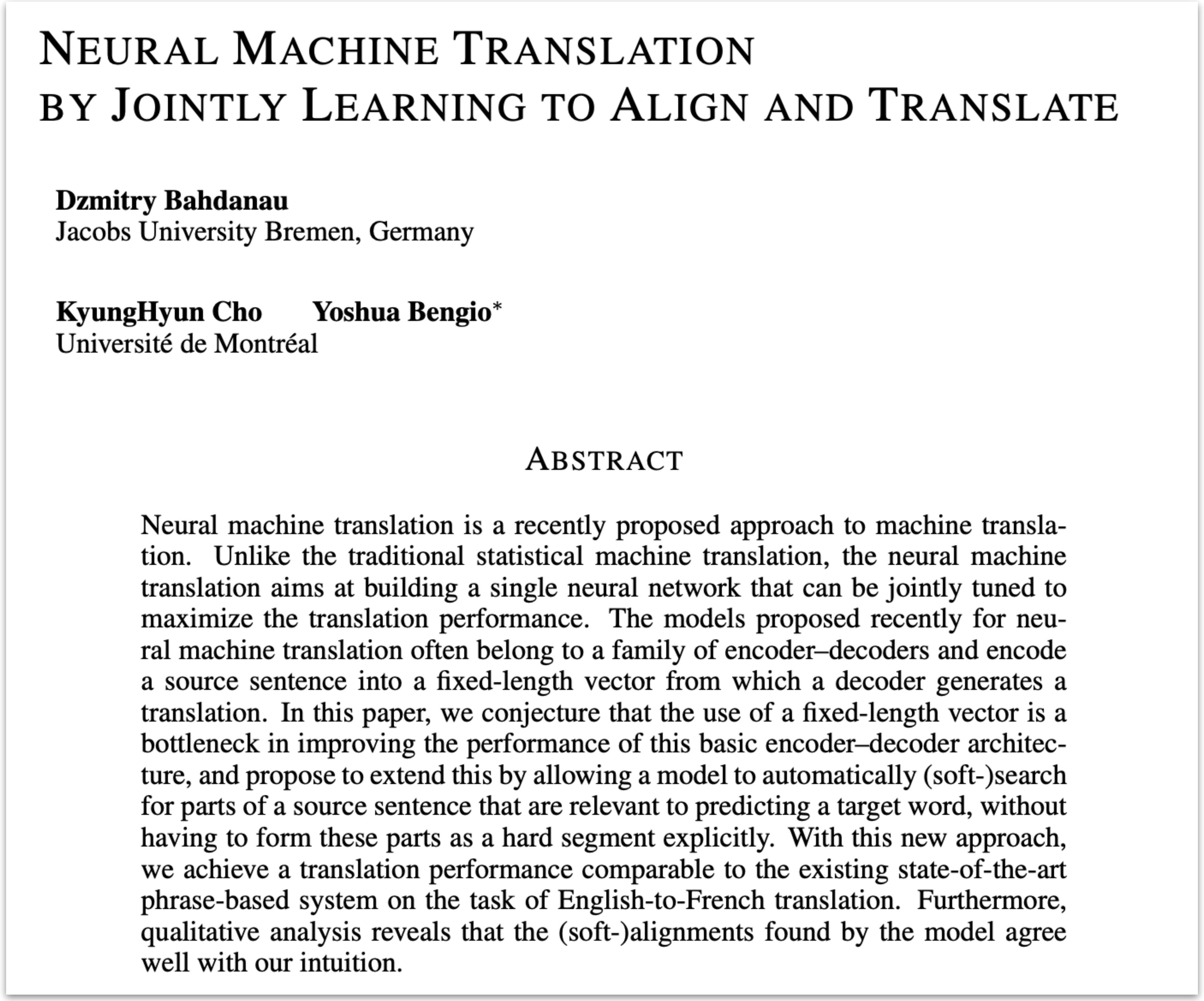
Jürgen Schmidhuber
IDSIA
Corso Elvezia 36
6900 Lugano, Switzerland
juergen@idsia.ch
http://www.idsia.ch/~juergen

Abstract

Learning to store information over extended time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow. We briefly review Hochreiter's 1991 analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called "Long Short-Term Memory" (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing *constant* error flow through "constant error carrousels" within special units. Multiplicative gate units learn to open and close access to the constant error flow. LSTM is local in space and time; its computational complexity per time step and weight is $O(1)$. Our experiments with artificial data involve local, distributed, real-valued, and noisy pattern representations. In comparisons with RTRL, BPTT, Recurrent Cascade-Correlation, Elman nets, and Neural Sequence Chunking, LSTM leads to many more successful runs, and learns much faster. LSTM also solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms.

2014: Attention Mechanism

- People were trying to use RNNs for encoder-decoder models (more on this session) for machine translation
- Introduced the attention mechanism to RNNs



NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder–decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

2017: The Transformer

- Dropped RNN portion
- Introduced self-attention, multi-head attention, and positional encoding
- Seminal paper that revolutionized NLP & gave rise to the era of LLMs
- Full focus of next session!

Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

Aidan N. Gomez[*] [†]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

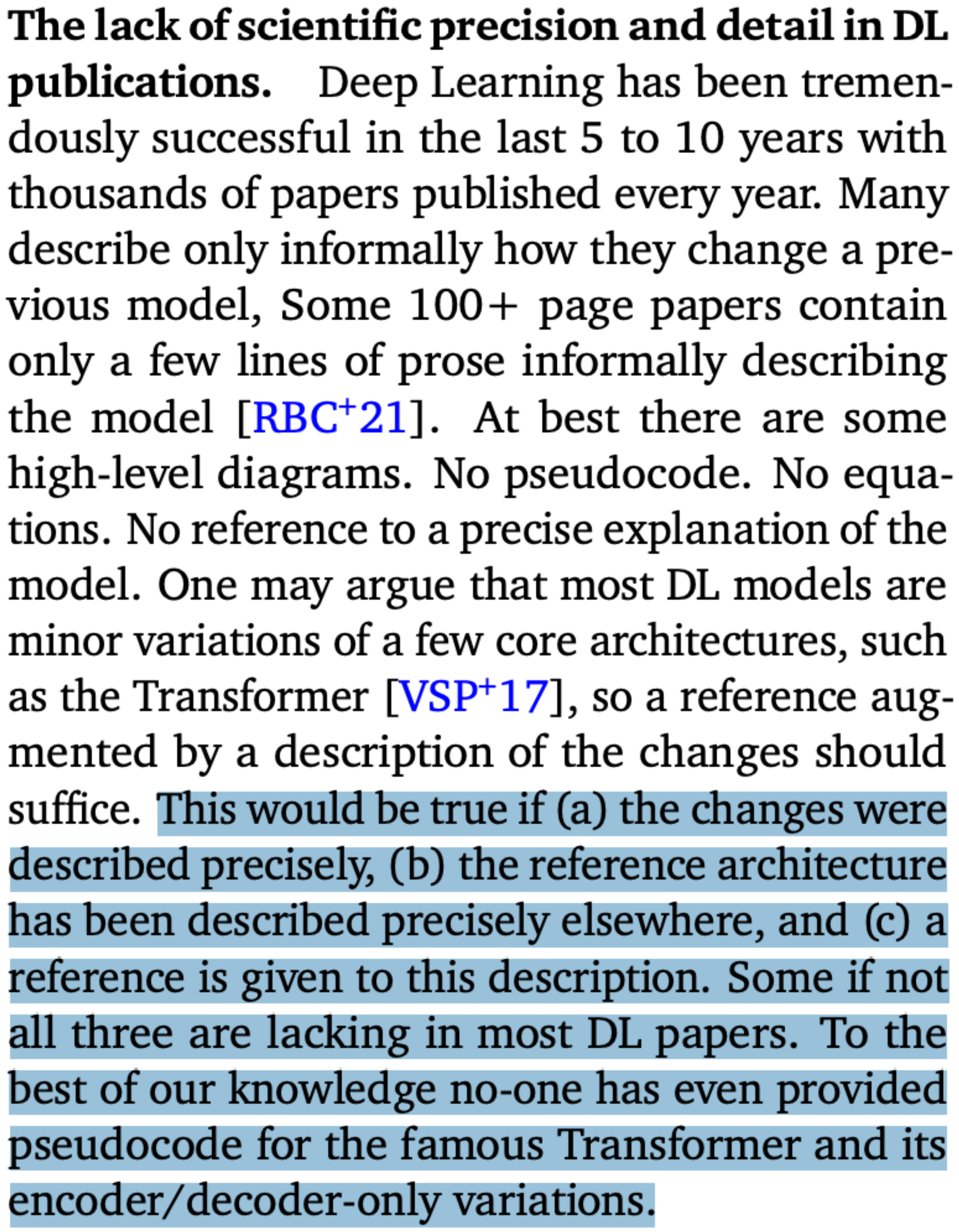
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

Transformers

Don’t Worry

- Even experts don’t fully understand how the Transformer works in detail due to ambiguity
- All the information might be overwhelming initially
    - Depends on a lot of small implementational details learnt from decades of DL research



The lack of scientific precision and detail in DL publications. Deep Learning has been tremendously successful in the last 5 to 10 years with thousands of papers published every year. Many describe only informally how they change a previous model, Some 100+ page papers contain only a few lines of prose informally describing the model [RBC+21]. At best there are some high-level diagrams. No pseudocode. No equations. No reference to a precise explanation of the model. One may argue that most DL models are minor variations of a few core architectures, such as the Transformer [VSP+17], so a reference augmented by a description of the changes should suffice. This would be true if (a) the changes were described precisely, (b) the reference architecture has been described precisely elsewhere, and (c) a reference is given to this description. Some if not all three are lacking in most DL papers. To the best of our knowledge no-one has even provided pseudocode for the famous Transformer and its encoder/decoder-only variations.

Formal Algorithms for Transformers, 2022

Transformer Outline

- **Encoder and decoders**
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

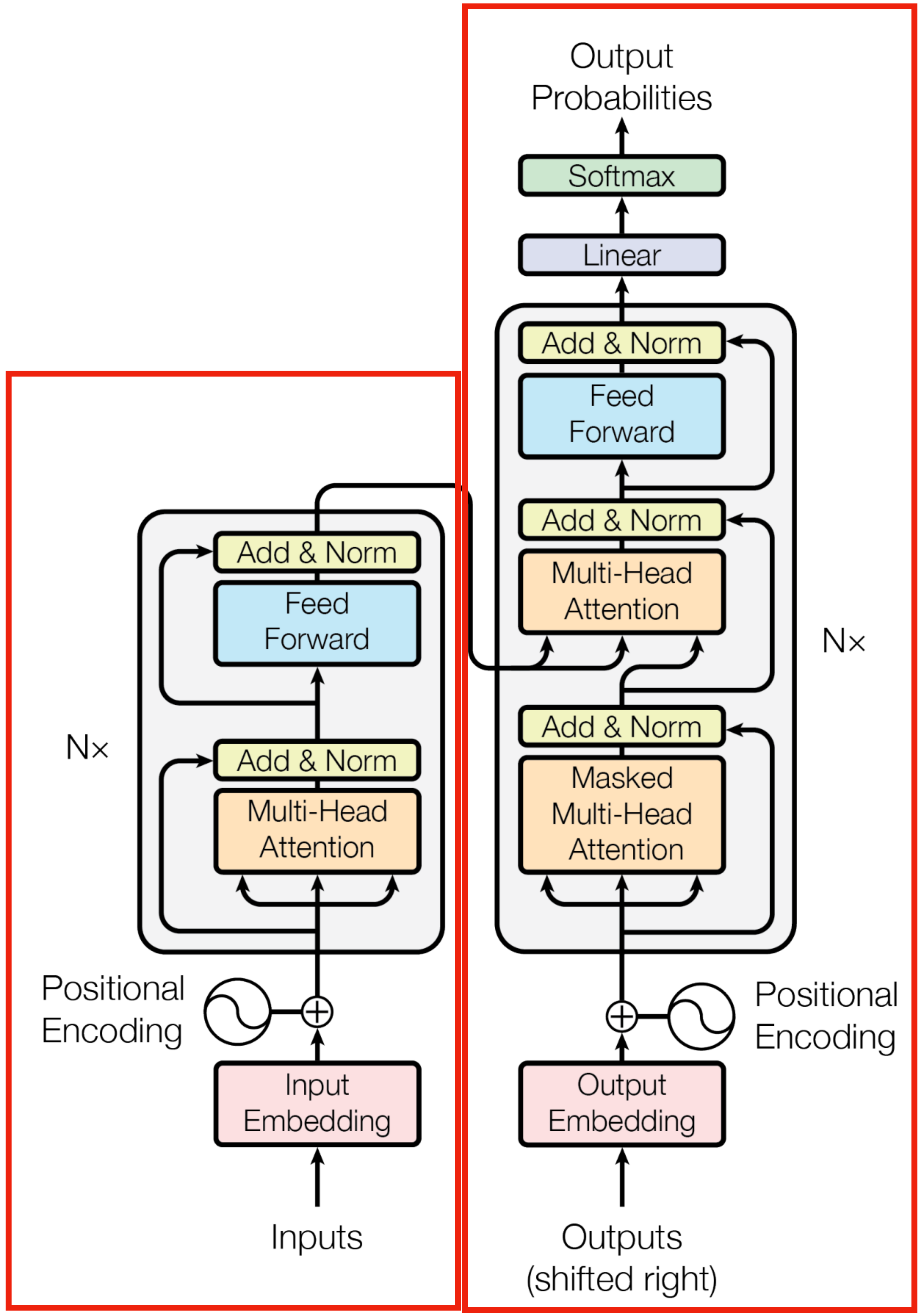


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- **Embeddings**
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

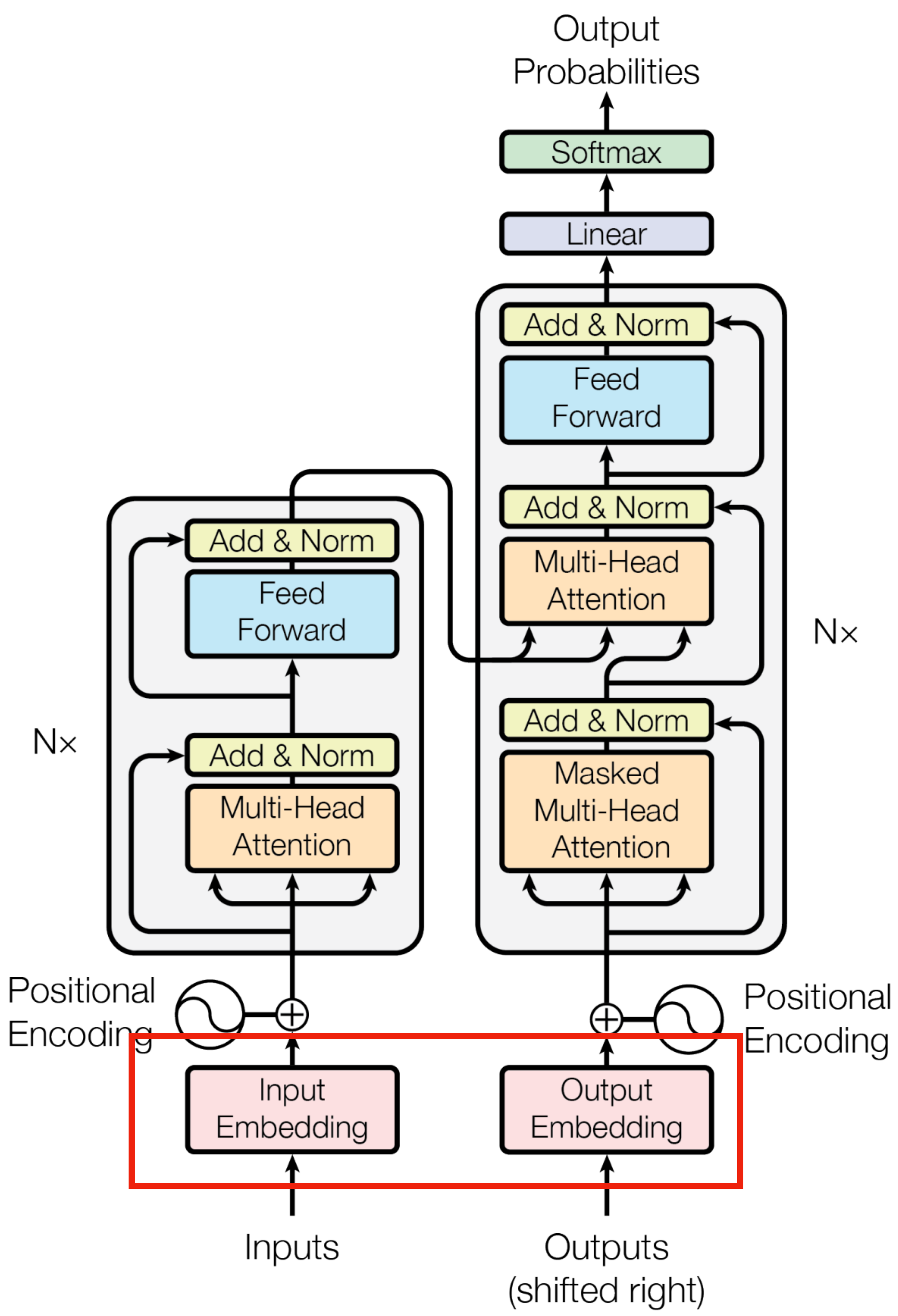


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- **Attention mechanism**
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

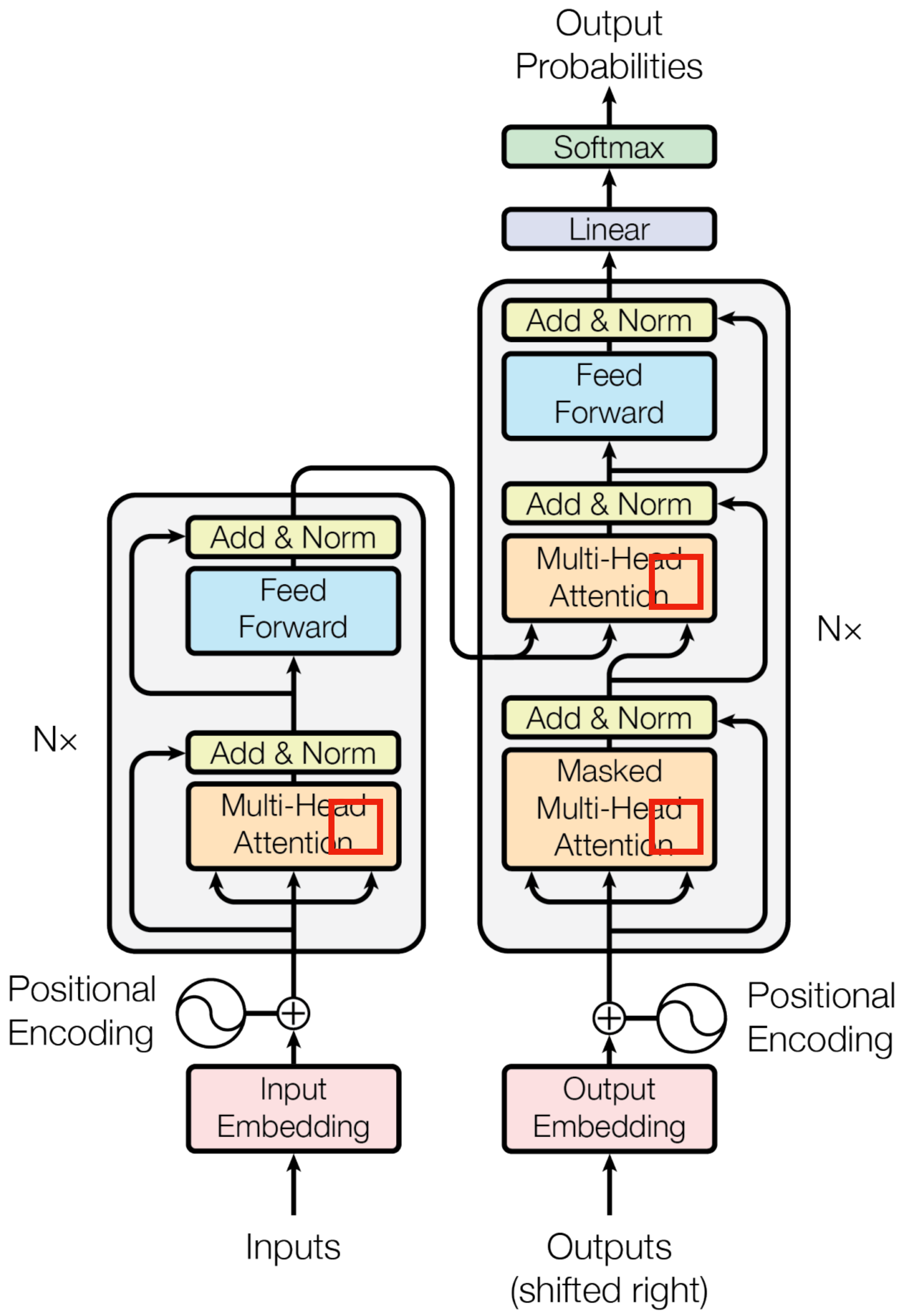


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- **Self-attention**
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

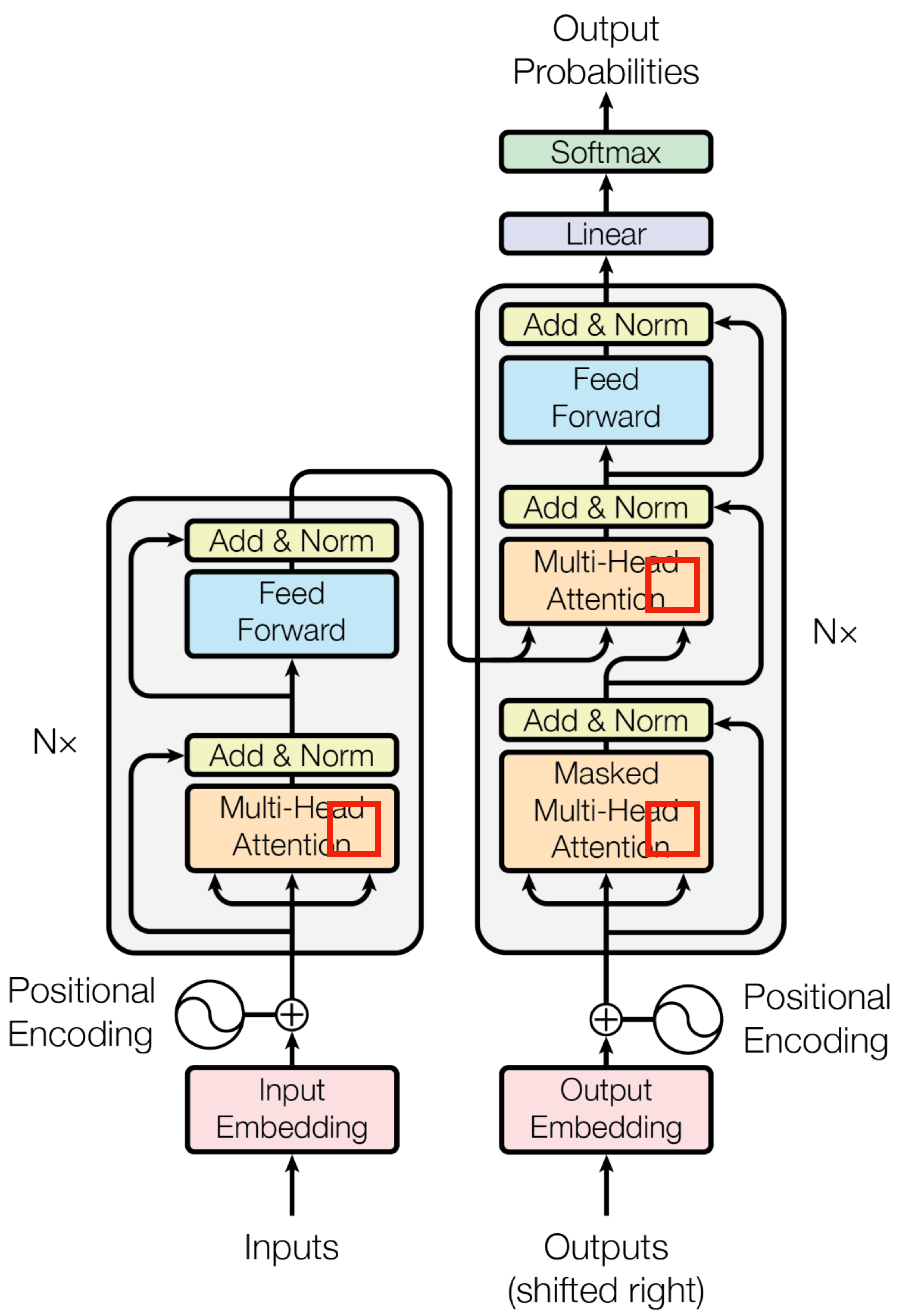


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- **Multi-head Attention**
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

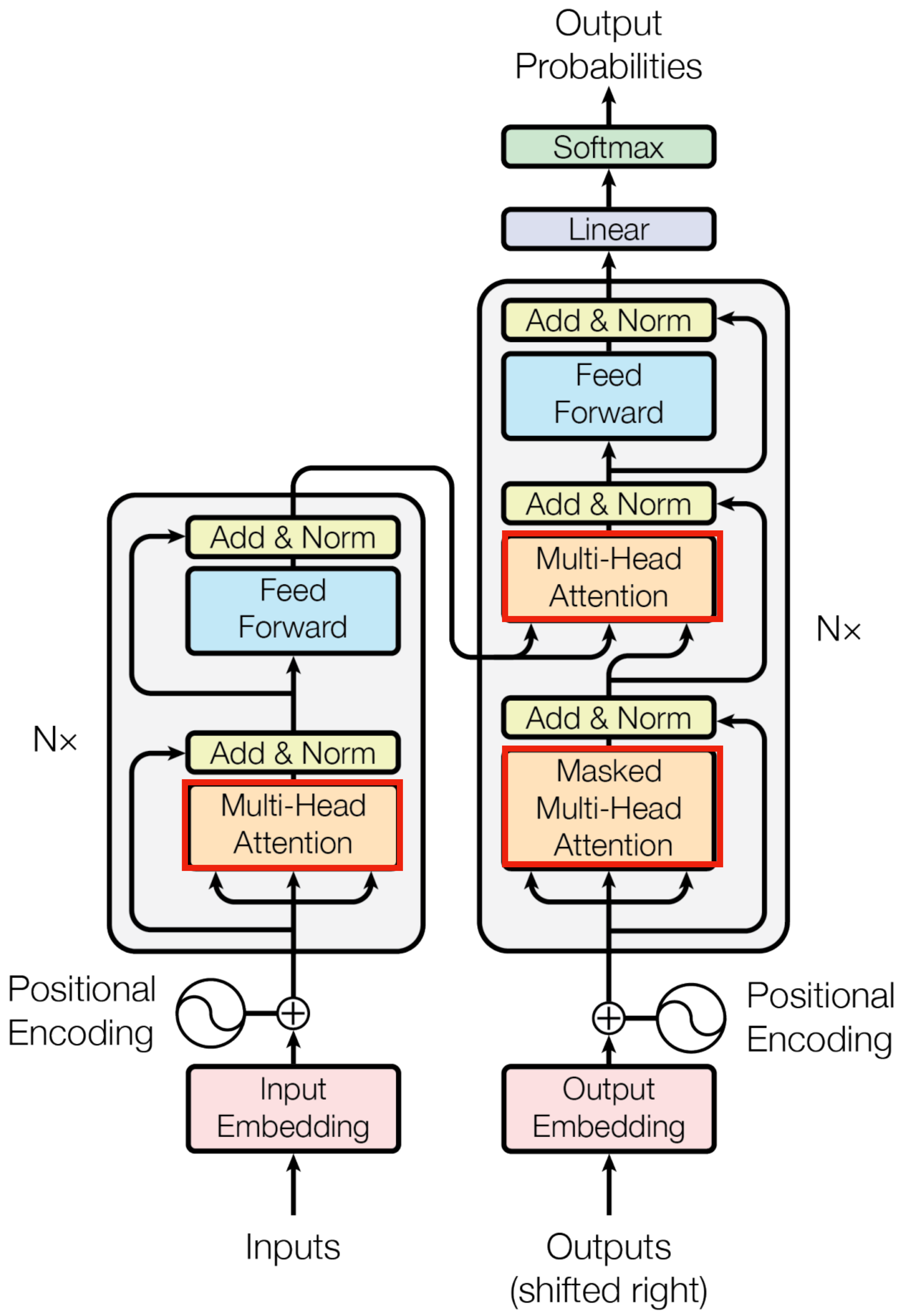


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- **Positional encoding**
- Residual connections
- Layer normalization
- Language Modeling Head

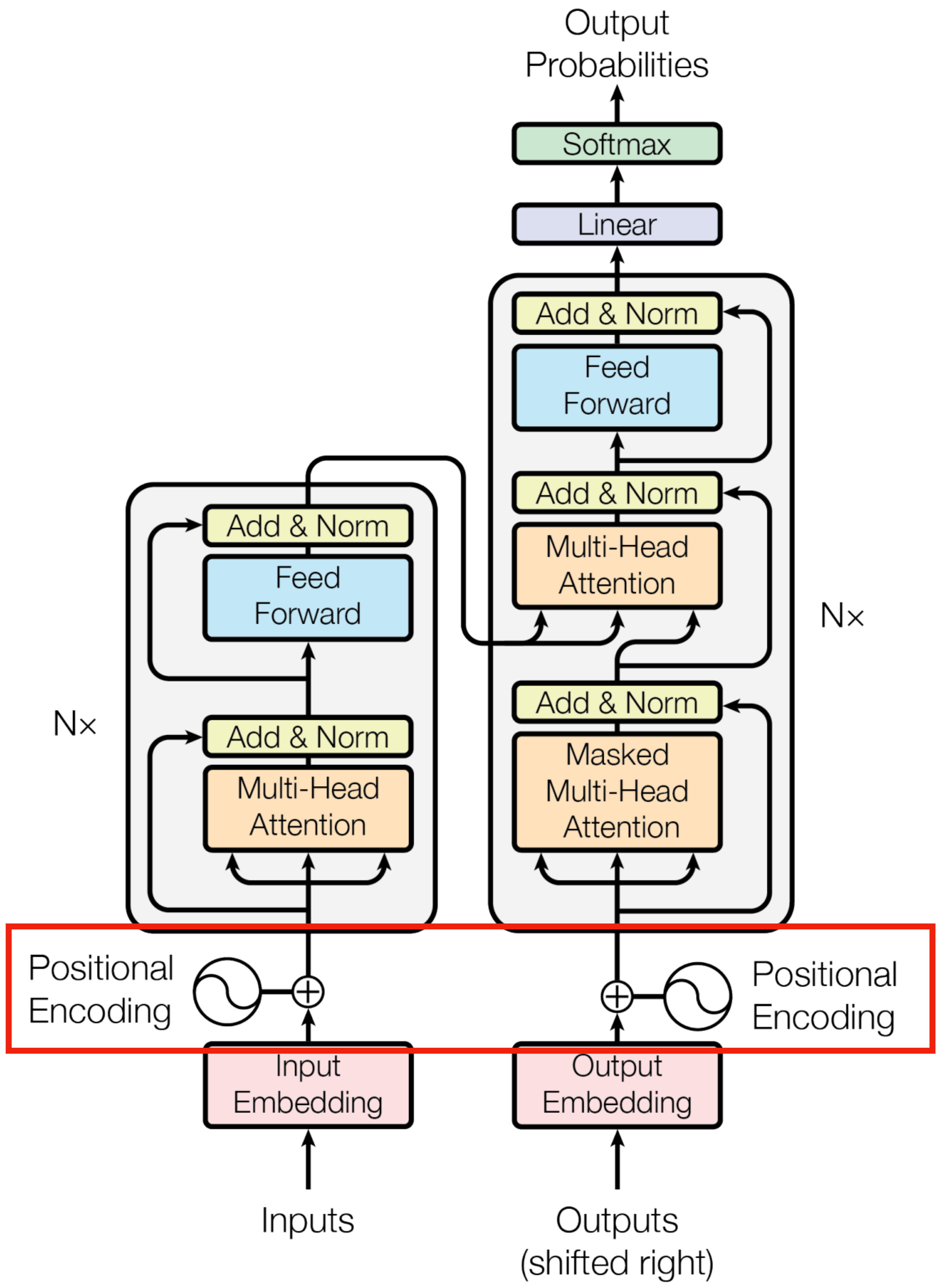


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- **Residual connections**
- Layer normalization
- Language Modeling Head

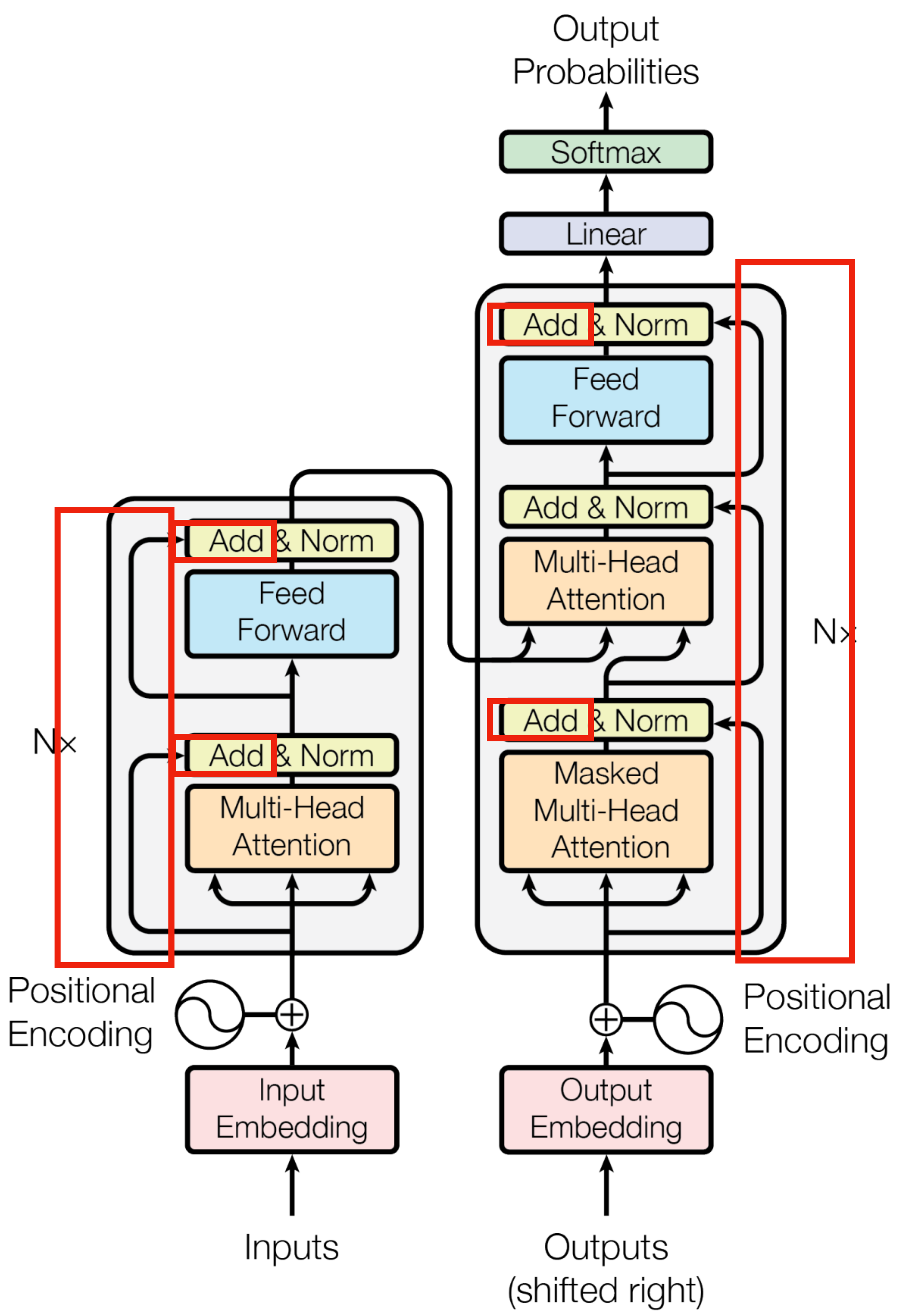


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- **Layer normalization**
- Language Modeling Head

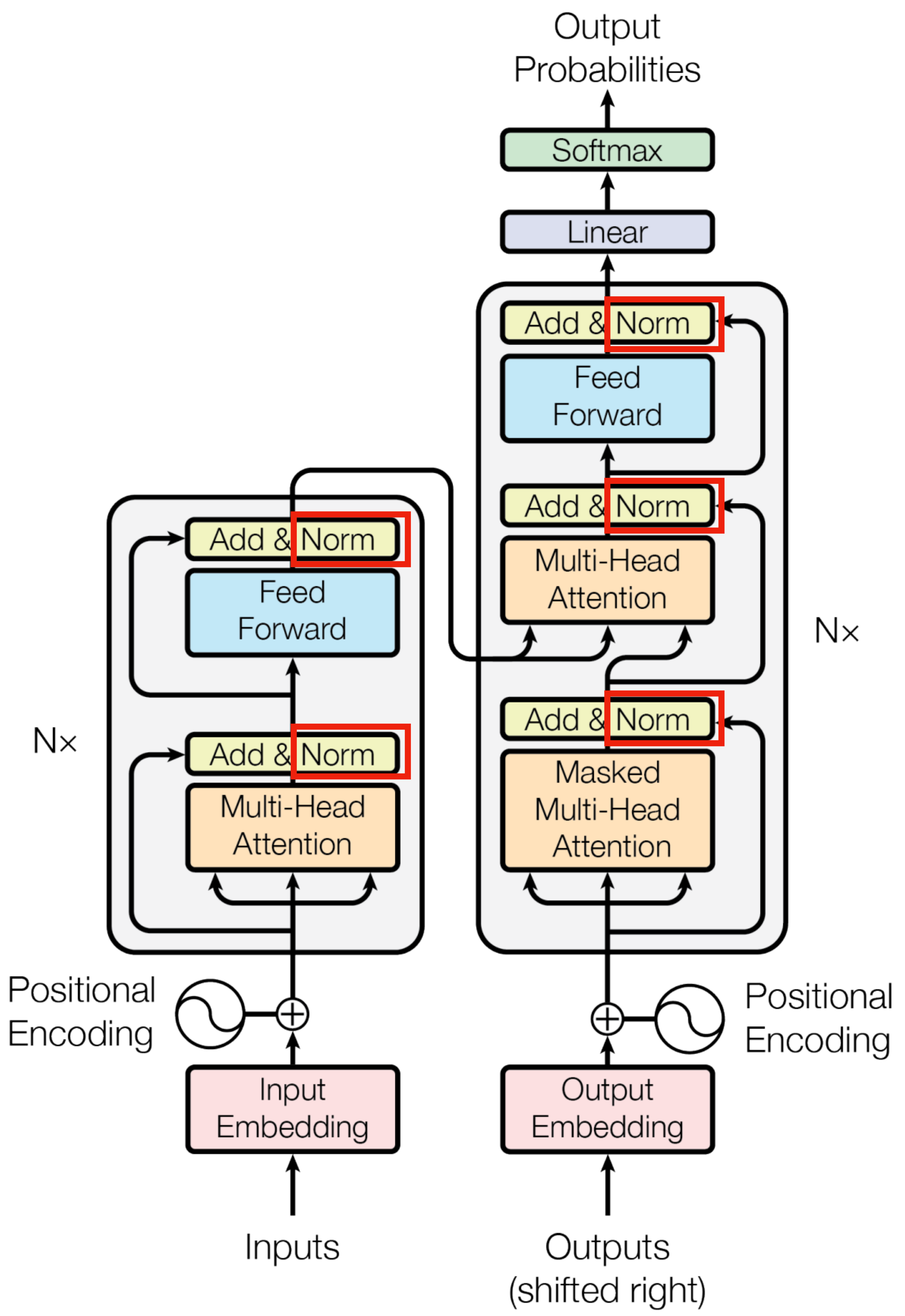


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- **Language Modeling Head**

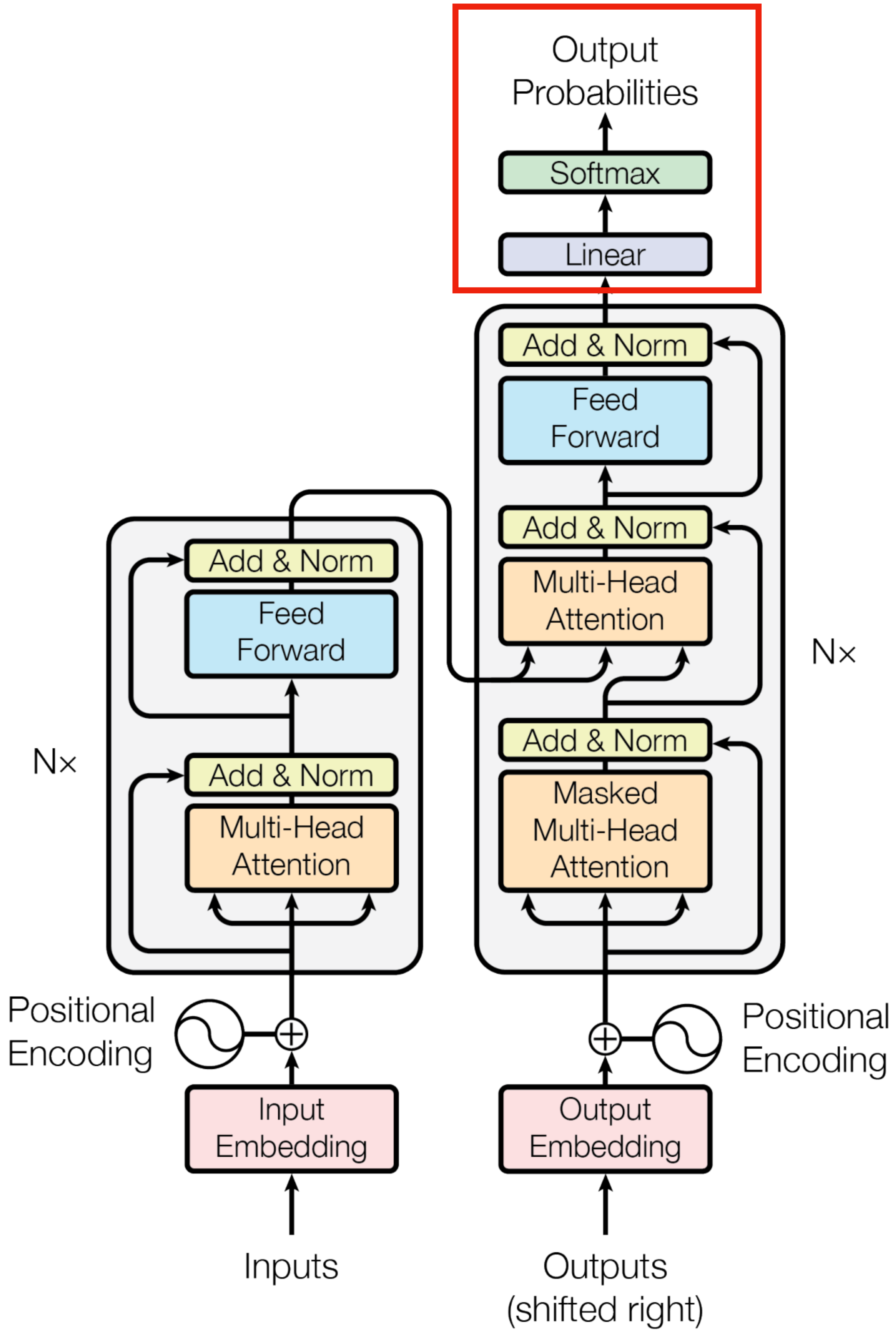
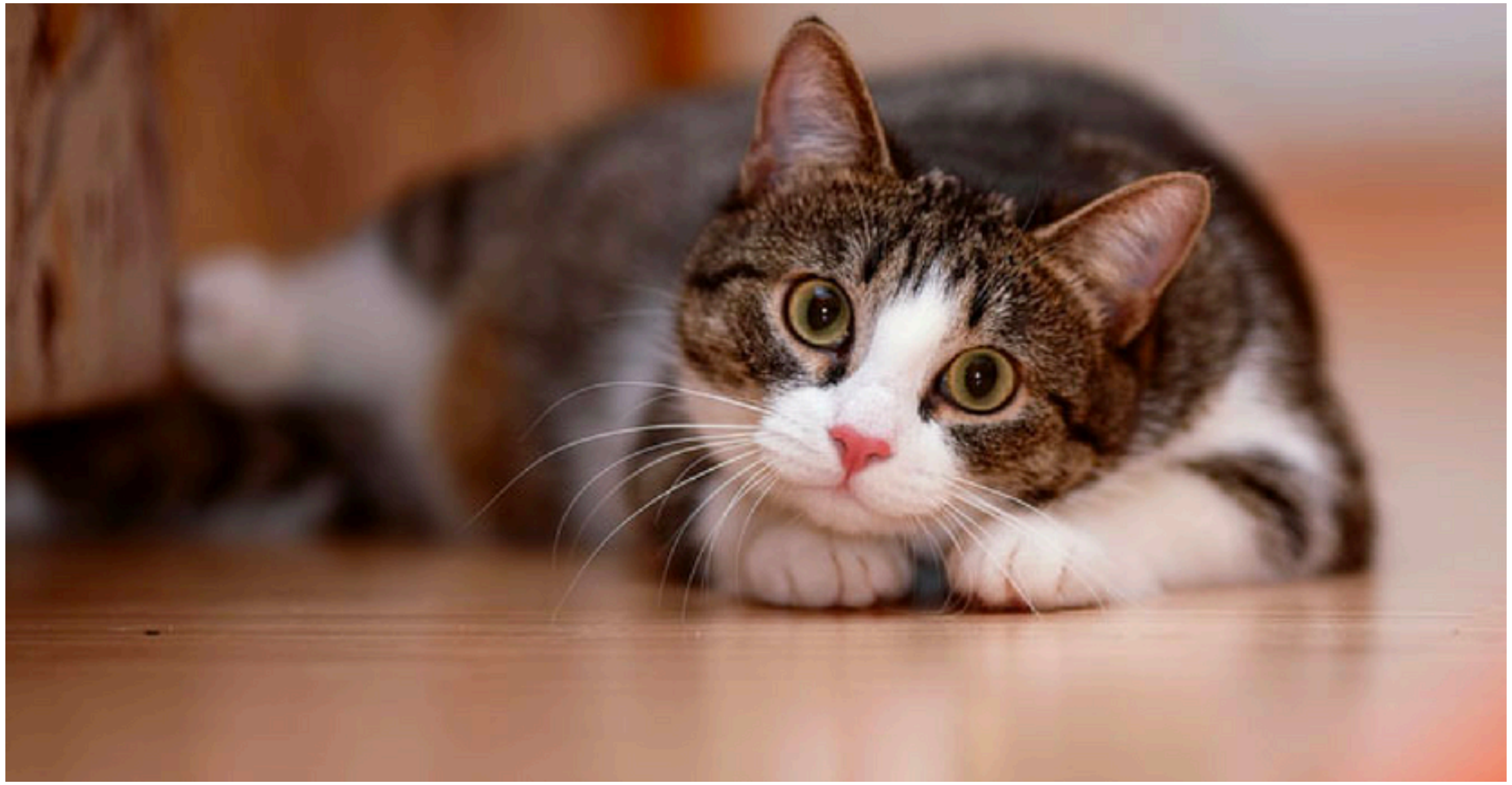


Figure 1: The Transformer - model architecture.

A kitten to *spurr* us on



Transformer Outline

- **Encoder and decoders**
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

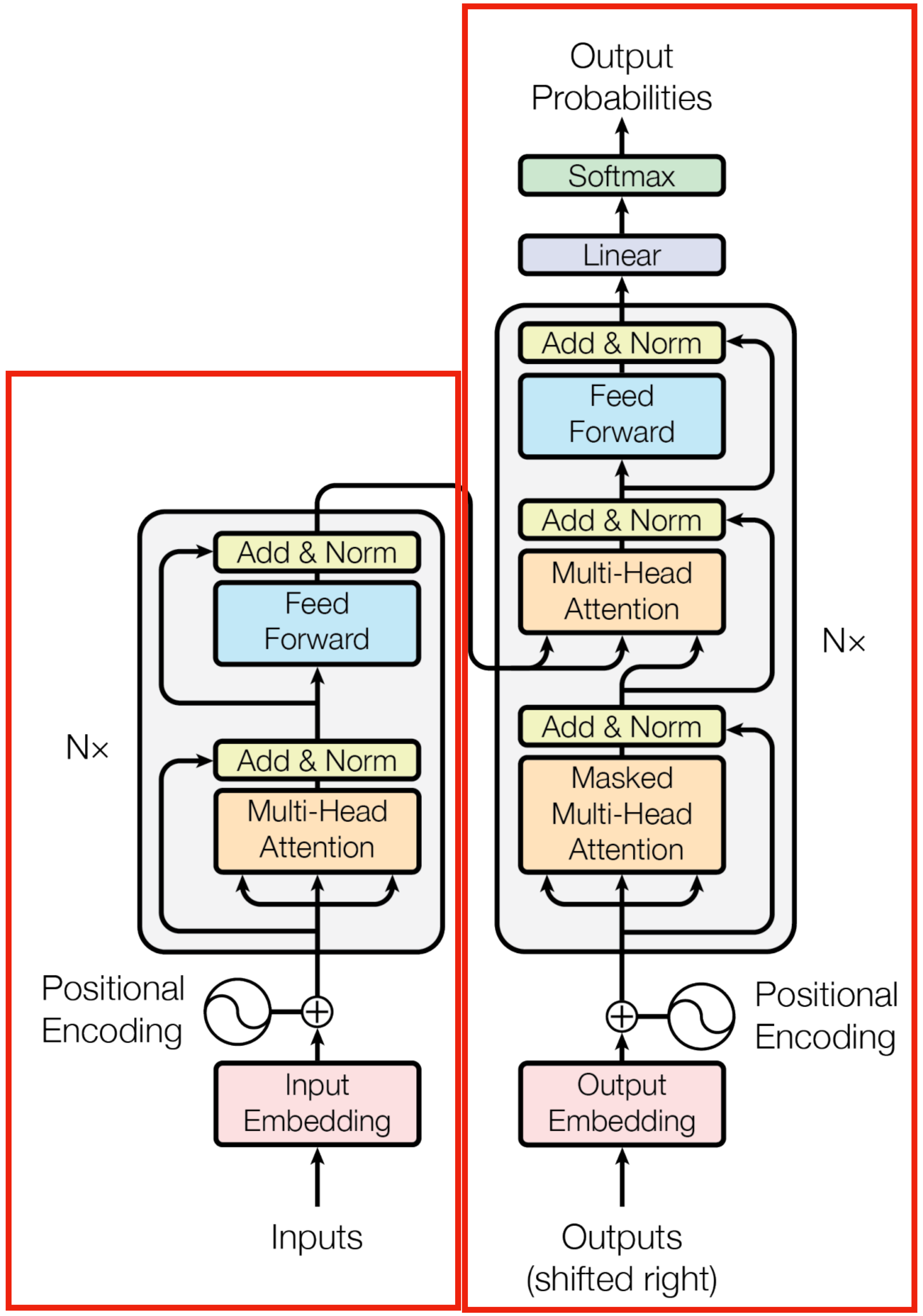
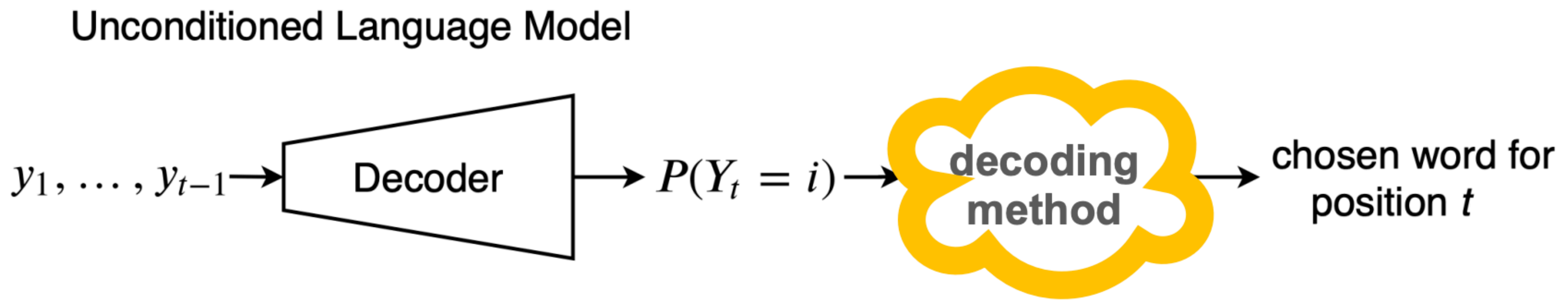


Figure 1: The Transformer - model architecture.

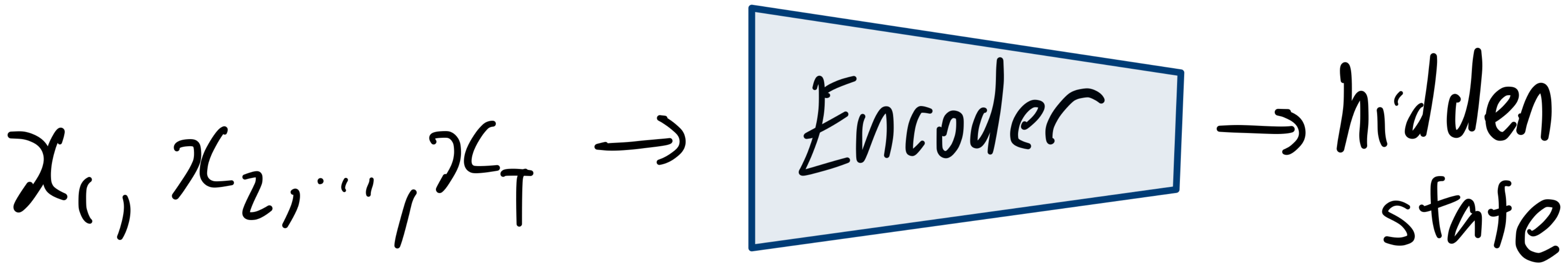
Encoder-only, Decoder-only, Encoder-Decoder Transformers

- Decoder-only:
  - Given previous outputs, generate next token
  - Good for text generation
  - GPT-2, GPT-3, LLaMA



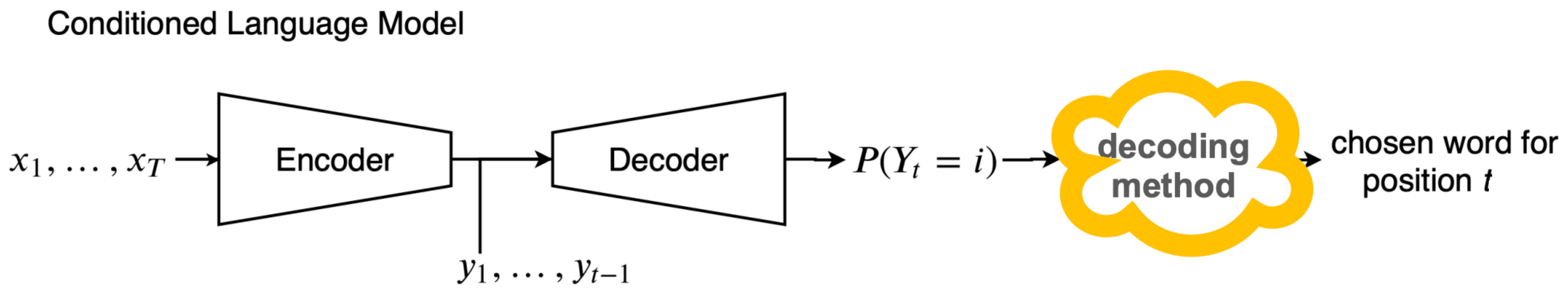
Encoder-only, Decoder-only, Encoder-Decoder Transformers

- Encoder-only:
  - Produces hidden state for use in downstream tasks
  - Text classification, sentiment analysis, named entity recognition
  - BERT ([CLS] token), DistilBERT, RoBERTa



Encoder-only, Decoder-only, Encoder-Decoder Transformers

- Encoder-Decoder:
  - Good for tasks requiring understanding input sequences, and then generating output sequence
  - Text translation, summarization
  - BART, T5
- Original attention paper uses encoder-decoder architecture



Encoders, Decoders, Encoder-Decoders

- Cute (but not very accurate) analogy:
- If you can understand a language, you have a trained encoder
- If you can speak the language, you have a trained decoder
- If you can hold a conversation in that language with another person, you are a trained encoder-decoder model

Transformer Outline

- Encoder and decoders
- **Embeddings**
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

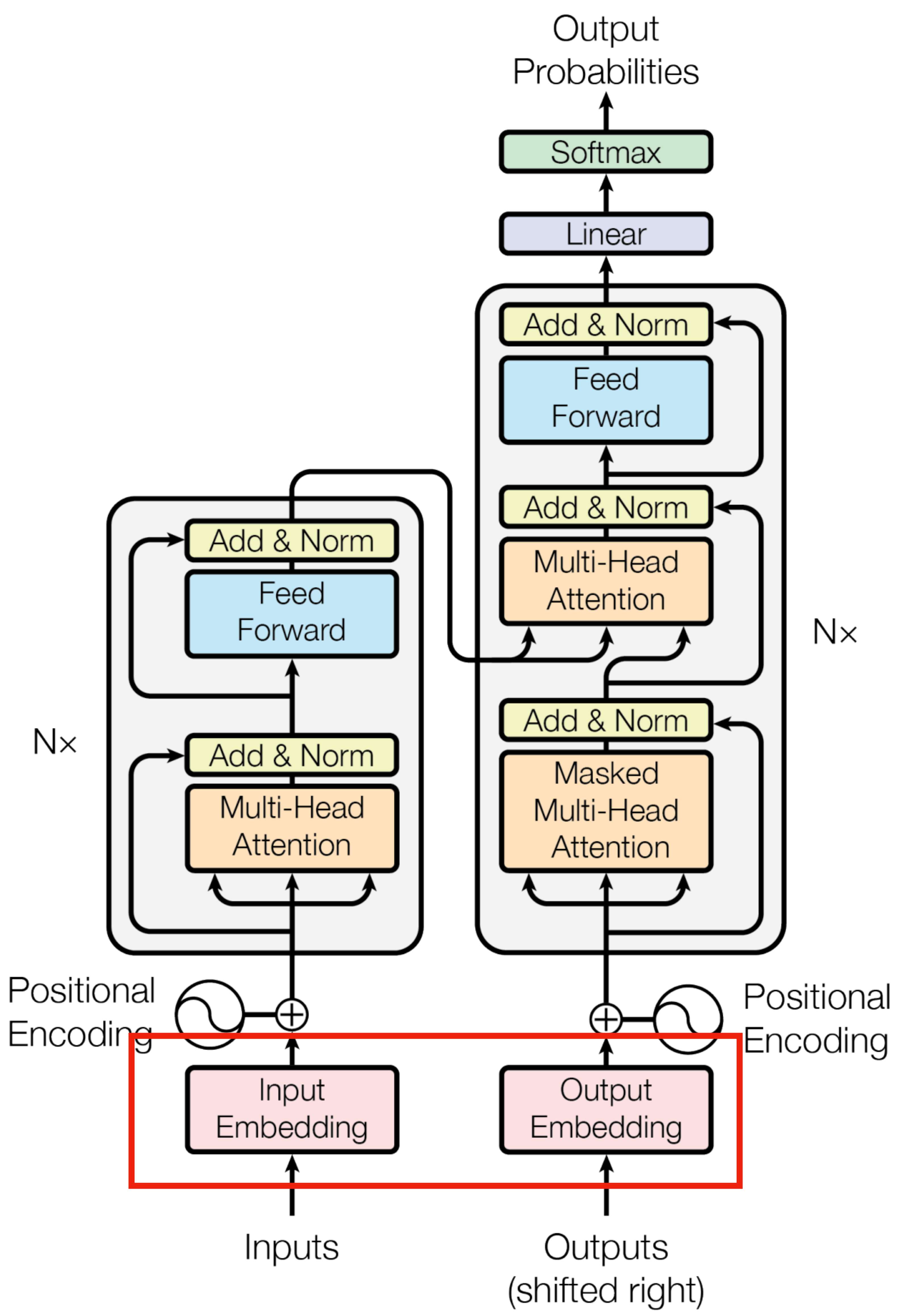
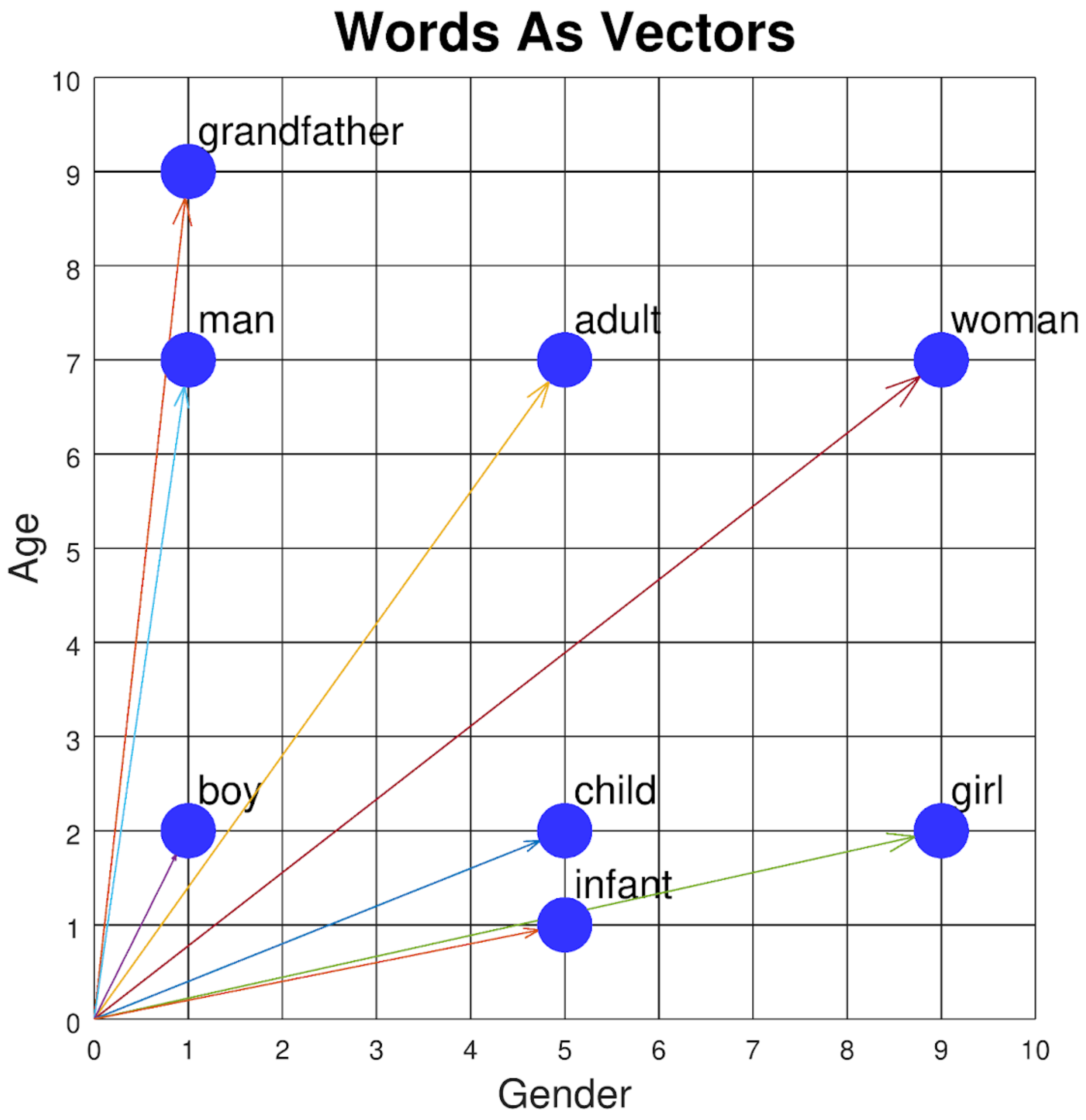


Figure 1: The Transformer - model architecture.

Word Embeddings

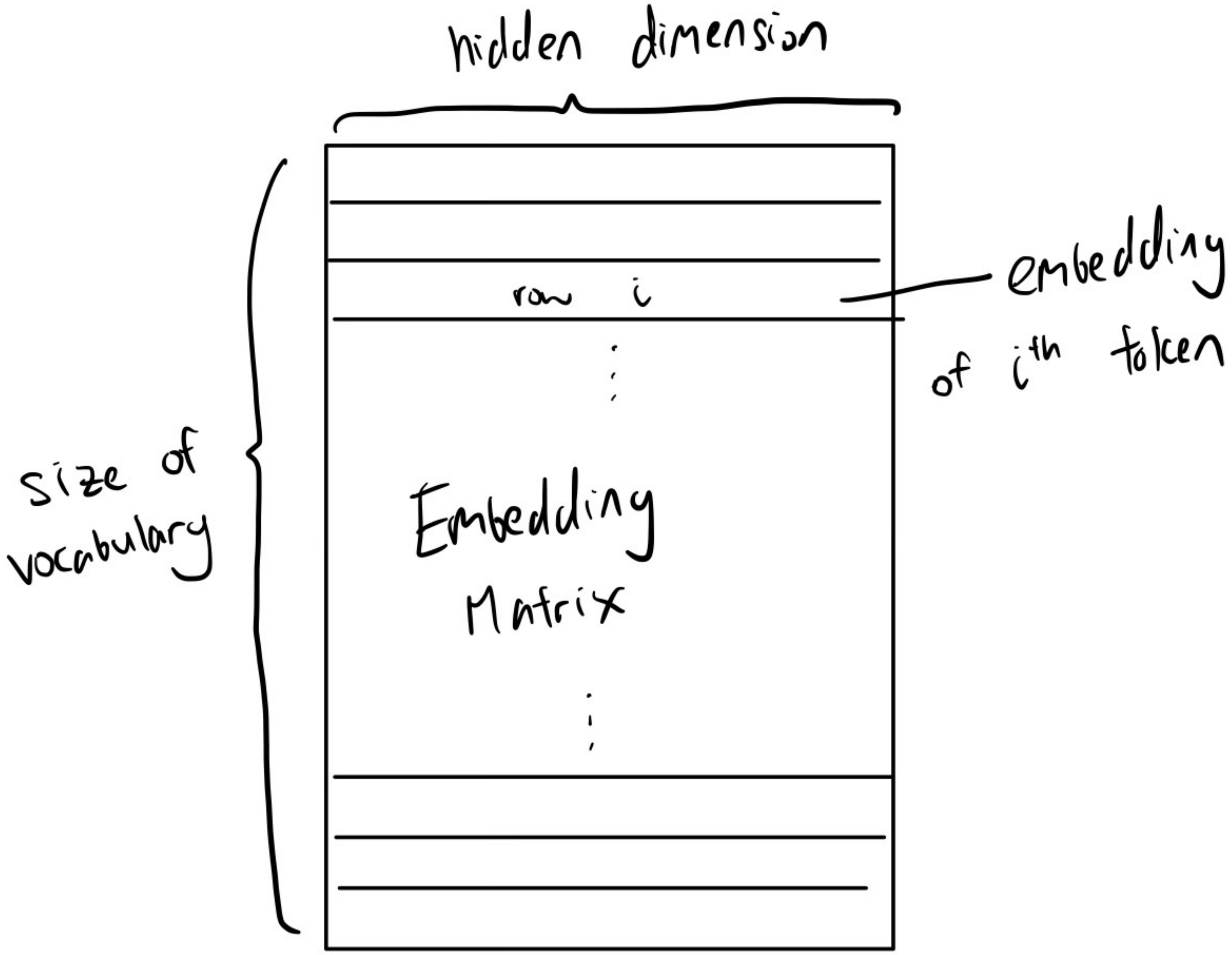
- Turn words into semantically meaningful vectors
- Benefits:
  - Semantically similar words closer together, different words further apart
  - Dimensionality reduction



Example of human attributes in 2D embeddings. Figure from Dave Touretzky

Word Embeddings

- In LLMs: embedding matrix is trained together with the rest of the model
- Input & output embeddings usually share same weights



Transformer Outline

- Encoder and decoders
- Embeddings
- **Attention mechanism**
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

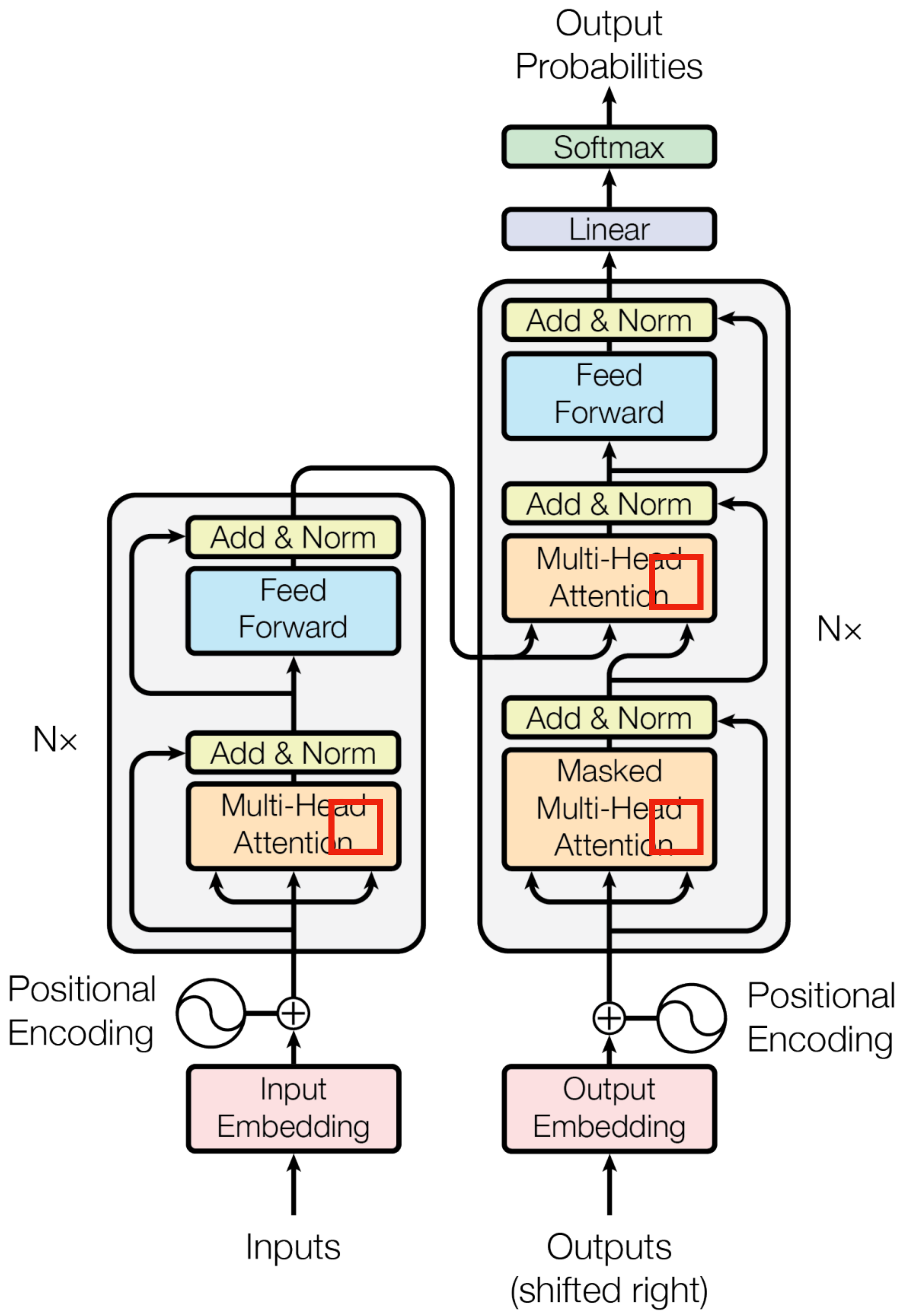


Figure 1: The Transformer - model architecture.

Attention

- Not all parts of the input equally important for task at hand
- E.g. image classification: background does not matter, helps to ignore spurious features
- Idea: provide more weight for more relevant features, fade out less relevant features
- Features now context-aware

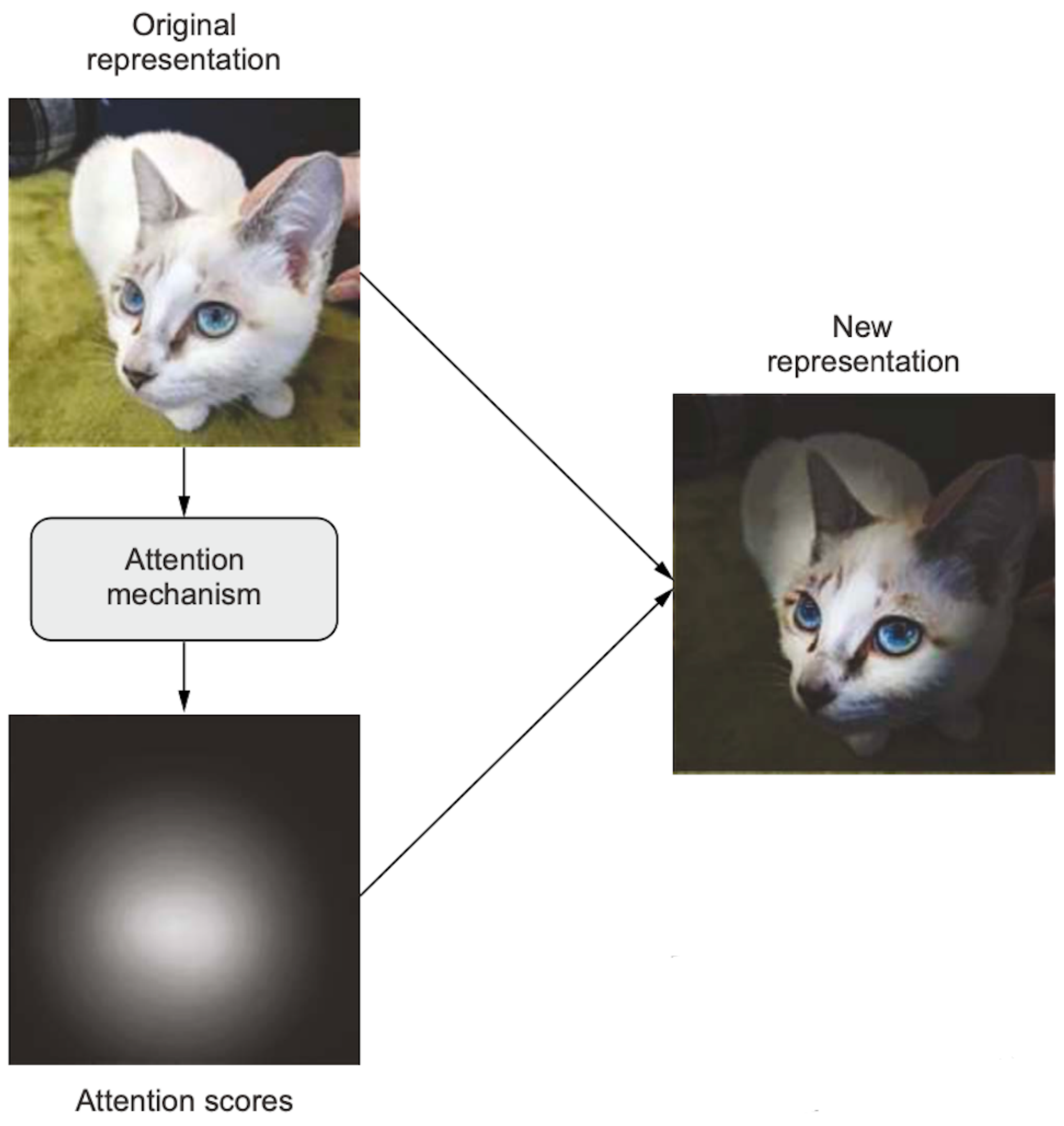


Image from Deep Learning with Python

Attention

- 3 components: query, key, values
- Terminology inspired by search engines
- Suppose you have a dataset of key-value pairs: (image tags, images)
- For a given query, how would you weigh your values to return the values blended by how important they are?
- Need some notion of similarity between the query and each of the keys!

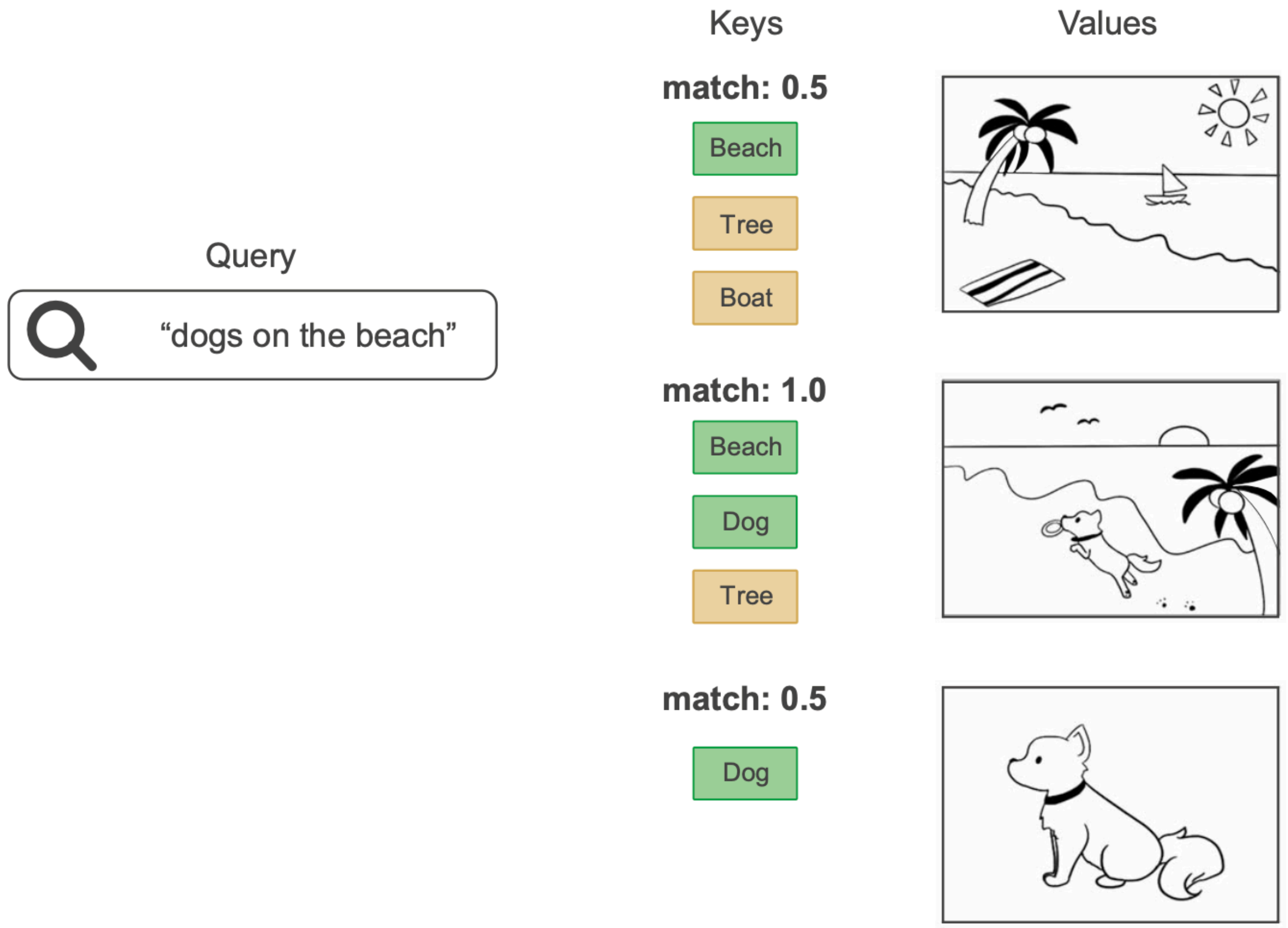


Image from Deep Learning with Python

Please Pay Attention

- We will derive the most famous equation in machine learning (Eq 1 in Attention Is All You Need):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

Attention

- Concretely: suppose we have m keys and values $\{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$
- Define attention on a query as:

$$\text{Attention}(\mathbf{q}) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$$

  for some weighing function $\alpha(\mathbf{q}, \mathbf{k}_i)$

- Idea: assigns different importance to each $\mathbf{v}_i$ depending on how similar $\mathbf{q}$ and $\mathbf{k}_i$ are!

Attention

- Attention$(\mathbf{q}) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$
- What is a good choice for $\alpha(\mathbf{q}, \mathbf{k}_i)$?
- Want non-negativity: $\alpha(\mathbf{q}, \mathbf{k}_i) > 0$
- Want normalization to 1: $\sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i) = 1$

Attention

- Suppose we have an arbitrary similarity function $a(\mathbf{q}, \mathbf{k}_i)$
- We can use it to construct α:
- Non-negativity: take exponentials, $\exp(a(\mathbf{q}, \mathbf{k}_i)) > 0$
- Normalization to 1: divide by sum of all values,

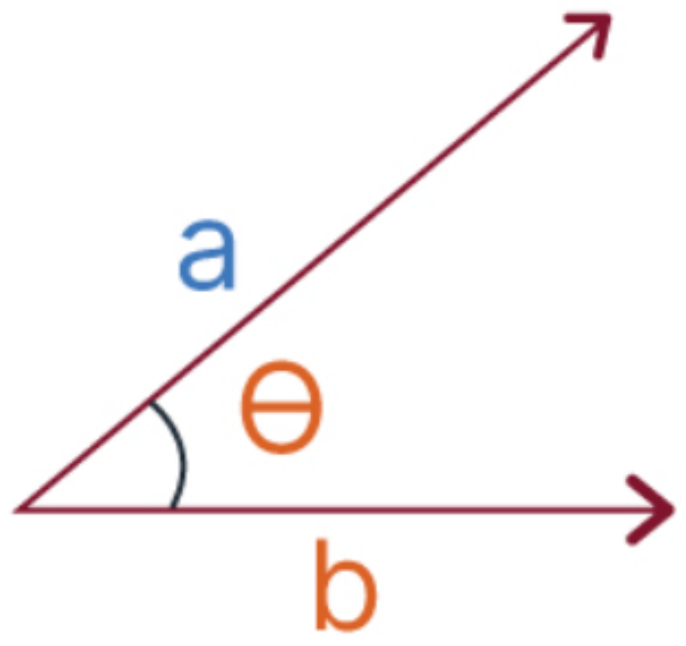
$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))} .$$

- Actually the above is just the softmax function:

$$\text{softmax}(\mathbf{x_i}) = \frac{\exp(\mathbf{x}_i)}{\sum_j \exp(\mathbf{x}_j)} .$$

Attention

- $\text{Attention}(\mathbf{q}) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$
- What is a good choice for $a(\mathbf{q}, \mathbf{k}_i)$?
- Dot product: distance metric that extends to arbitrary dimensions, measures “angle” between two vectors as notion of similarity
- So now we have dot product $\mathbf{q}^\top \mathbf{k}_i$



a . b = |a||b| cos θ

Attention

- Suppose $\mathbf{q}, \mathbf{k}_i$ are d-dimensional and drawn independently from standard normal distribution
- Dot product $\mathbf{q}^\top \mathbf{k}_i$ is now the sum of d products of two independent standard Gaussians
- If $X_i, Y_i \sim \mathcal{N}(0,1)$ i.i.d, then $E[X_iY_i] = 0,\ Var(X_iY_i) = 1$
- By linearity of expectations, $E\left[\sum_{i=1}^{d} X_iY_i\right] = 0$
- By linearity of variance, $Var\left(\sum_{i=1}^{d} X_iY_i\right) = d$
- High variance leads to instability especially since we have exponentials 😔

Attention

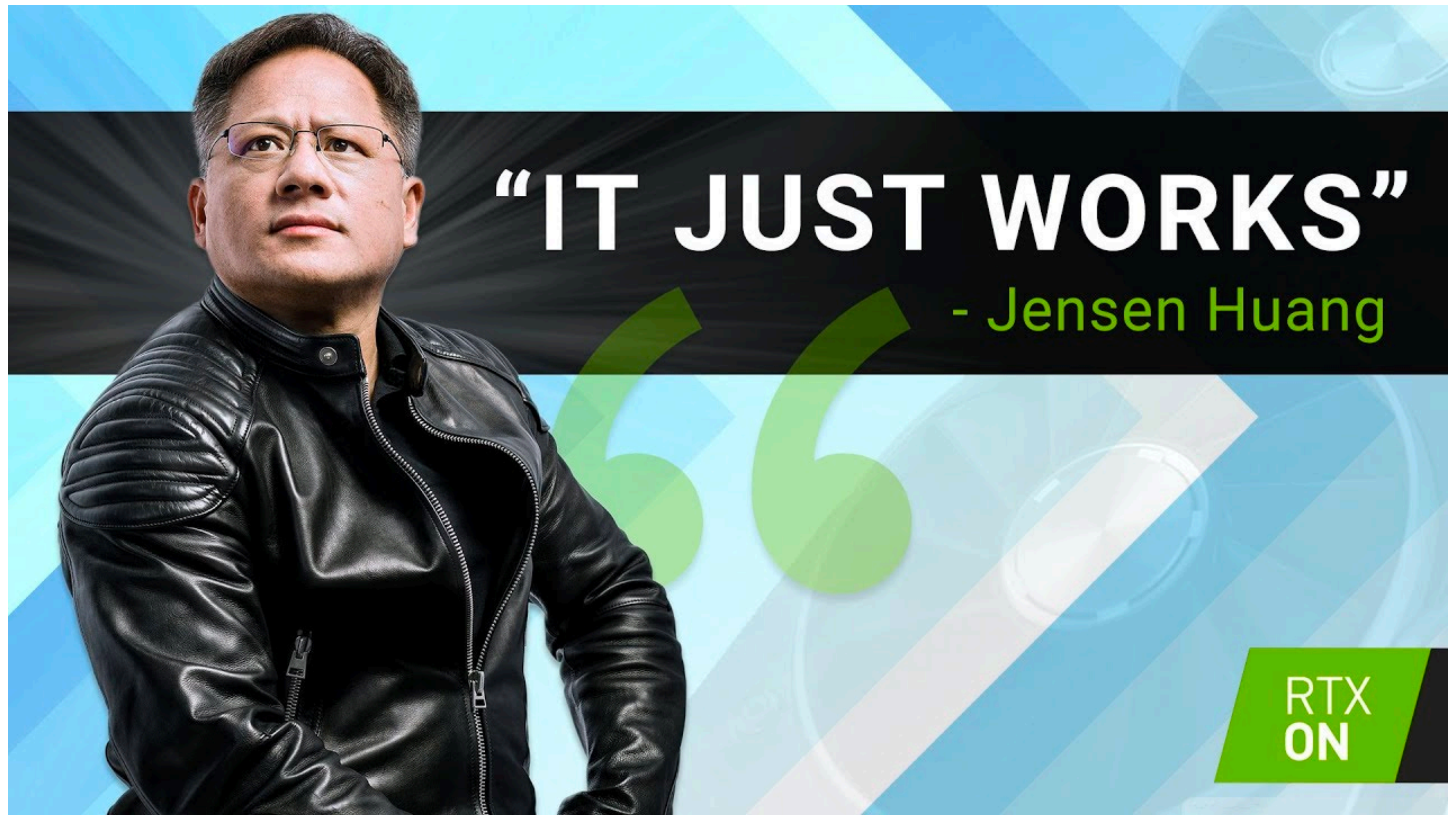
- Solution: scale by $1/\sqrt{d}$ to result in unit variance, since $Var(cX) = c^2 Var(X)$
- Putting everything together, we have scaled dot-product attention:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_j \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})} = \text{softmax}\left(\frac{\mathbf{q}^\top \mathbf{k}_i}{\sqrt{d}}\right)$$

- We are getting close 😊

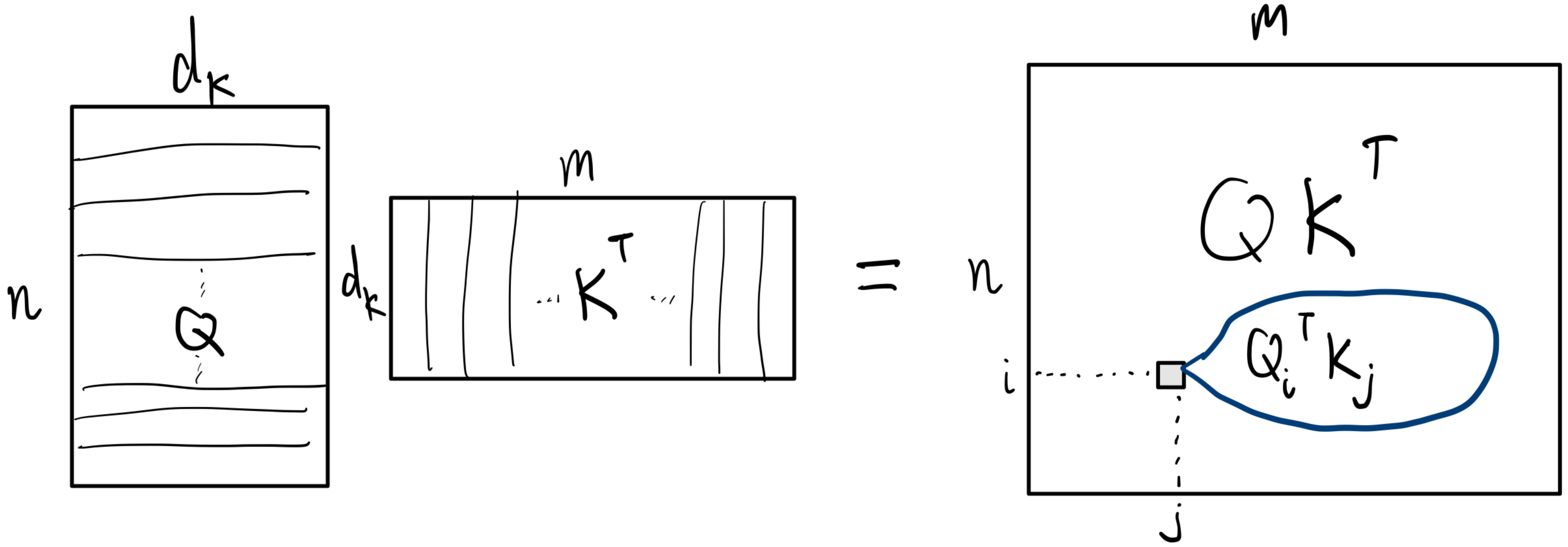
Batching

- Jensen Huang has blessed us with GPUs optimized for multiplying large matrices
- Instead of processing just an individual sample at a time, more efficient throughput-wise to *batch* multiple samples together



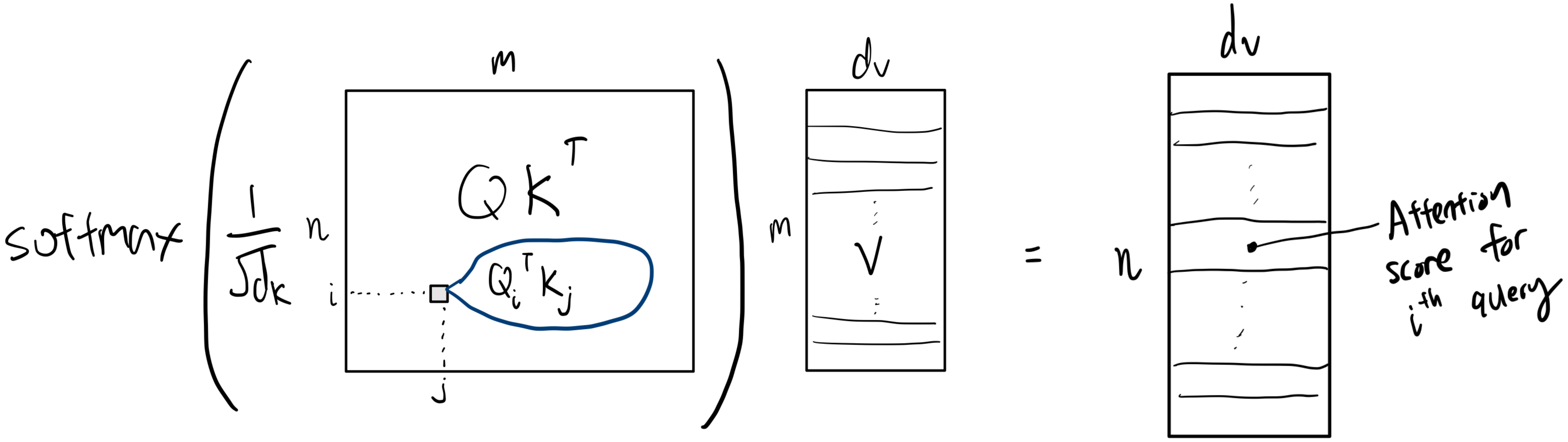
Batched Attention

- Suppose you have n queries and m keys and m values stacked together as matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ respectively
- Each key, query must have same dimension for dot product: d_k
- Each value has dimension d_v
- First compute $\mathbf{QK}^\top$



Batched Attention

- Next scale matrix entries, take softmax over each *row* in the matrix
- Multiply by $\mathbf{V}$, get batched attention:



Overall:

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

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- **Self-attention**
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

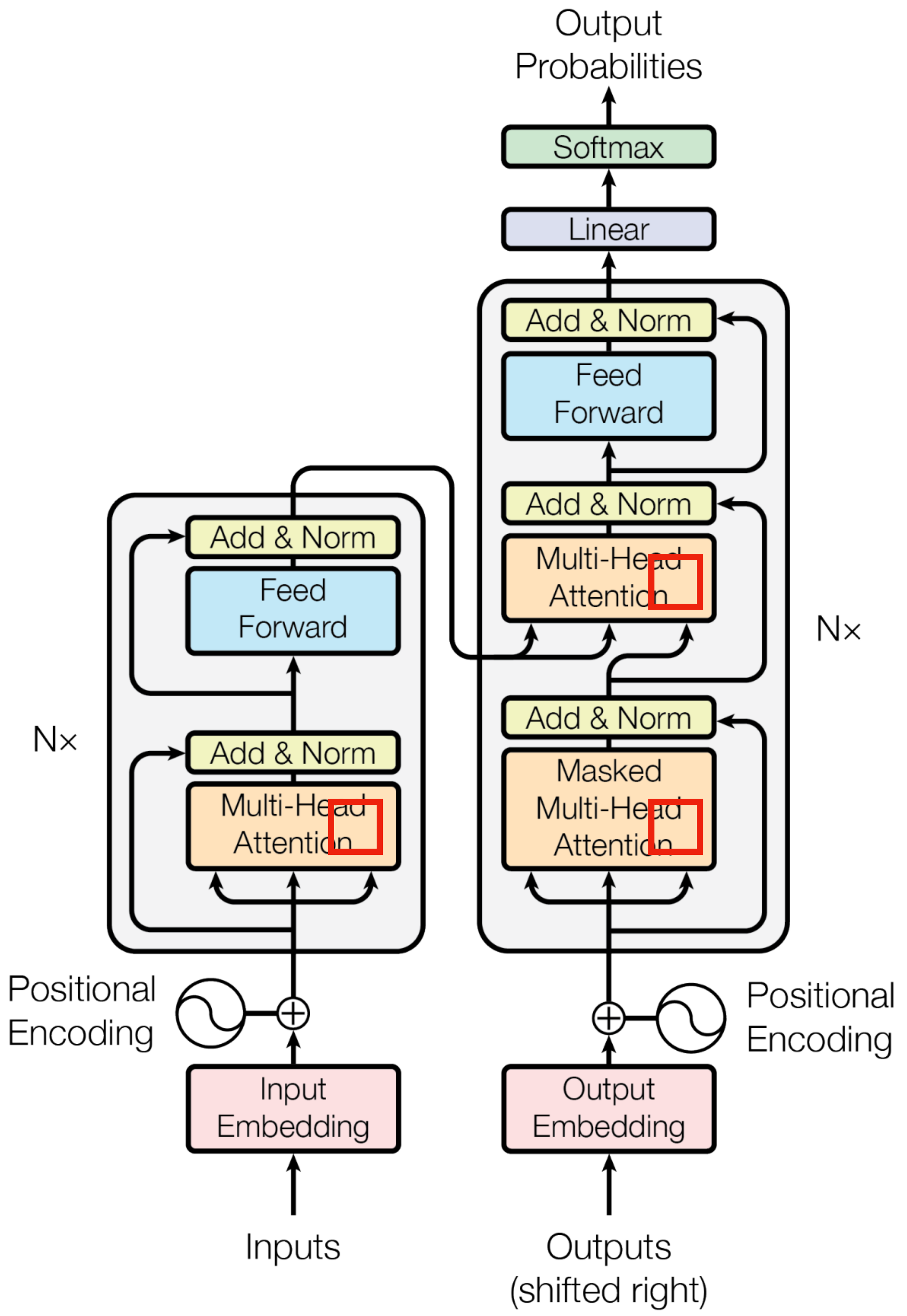


Figure 1: The Transformer - model architecture.

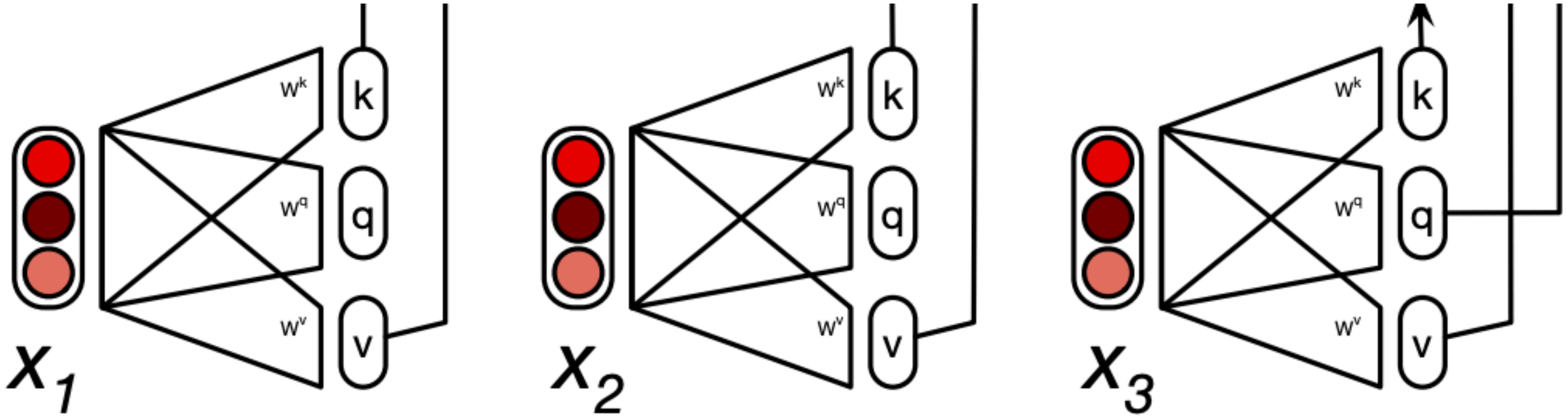
Self-Attention

- But how do we actually get our $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ for a given input to compute attention?

Self-Attention

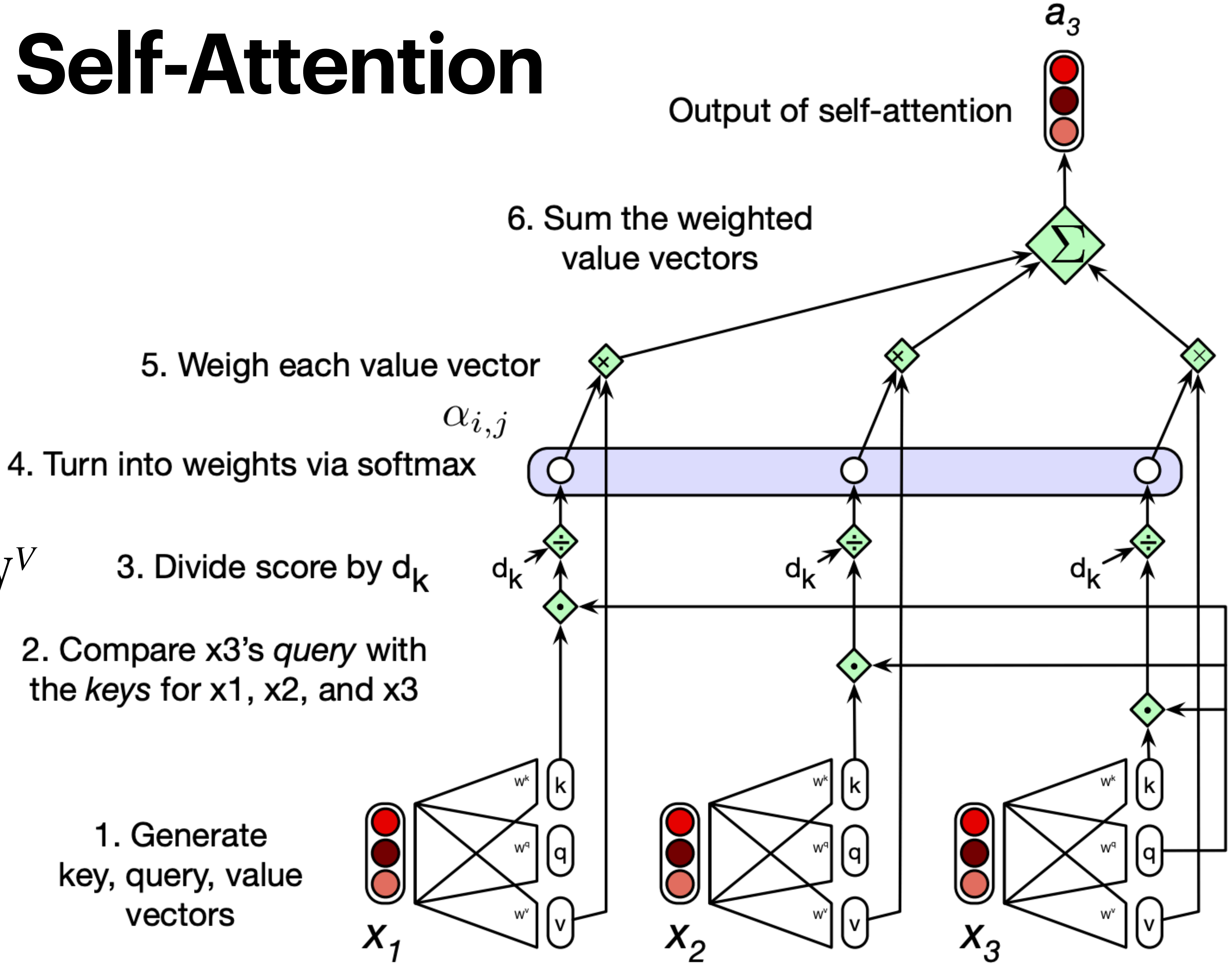
- But how do we actually get our $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ for a given input to compute attention?
- Given input vector x_i (corresponding to some token)
- Learnt weight matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$
- Project x_i by respective matrices for query, key, and value:
  $q_i = x_i\mathbf{W}^Q, k_i = x_i\mathbf{W}^K, v_i = x_i\mathbf{W}^V$

1. Generate key, query, value vectors



Self-Attention

- Parallelizing this computation with input matrix $\mathbf{X}$ instead, we recover $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \mathbf{K} = \mathbf{X}\mathbf{W}^K, \mathbf{V} = \mathbf{X}\mathbf{W}^V$
- Called self-attention, since query/key/values comes from same source


a_3
Output of self-attention
6. Sum the weighted value vectors
Σ
5. Weigh each value vector
$\alpha_{i,j}$
4. Turn into weights via softmax
3. Divide score by d_k
d_k
d_k
d_k
2. Compare x3's *query* with the *keys* for x1, x2, and x3
1. Generate key, query, value vectors
w^k
w^q
w^v
k
q
v
x_1
x_2
x_3

Masking Out Future Tokens

- Now that you understand how self-attention works, one thing might bother you…
- In computing $\mathbf{QK}^T$, we take all pairwise query-key comparisons, including between key values that follow the query value
- "Just Pay Attention To Future Tokens" is a cheat code
- Solution: mask comparisons between queries and future keys to $-\infty$ (so softmax gives 0)

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

q1•k1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
q2•k1	q2•k2	$-\infty$	$-\infty$	$-\infty$
q3•k1	q3•k2	q3•k3	$-\infty$	$-\infty$
q4•k1	q4•k2	q4•k3	q4•k4	$-\infty$
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- **Multi-head Attention**
- Positional encoding
- Residual connections
- Layer normalization
- Language Modeling Head

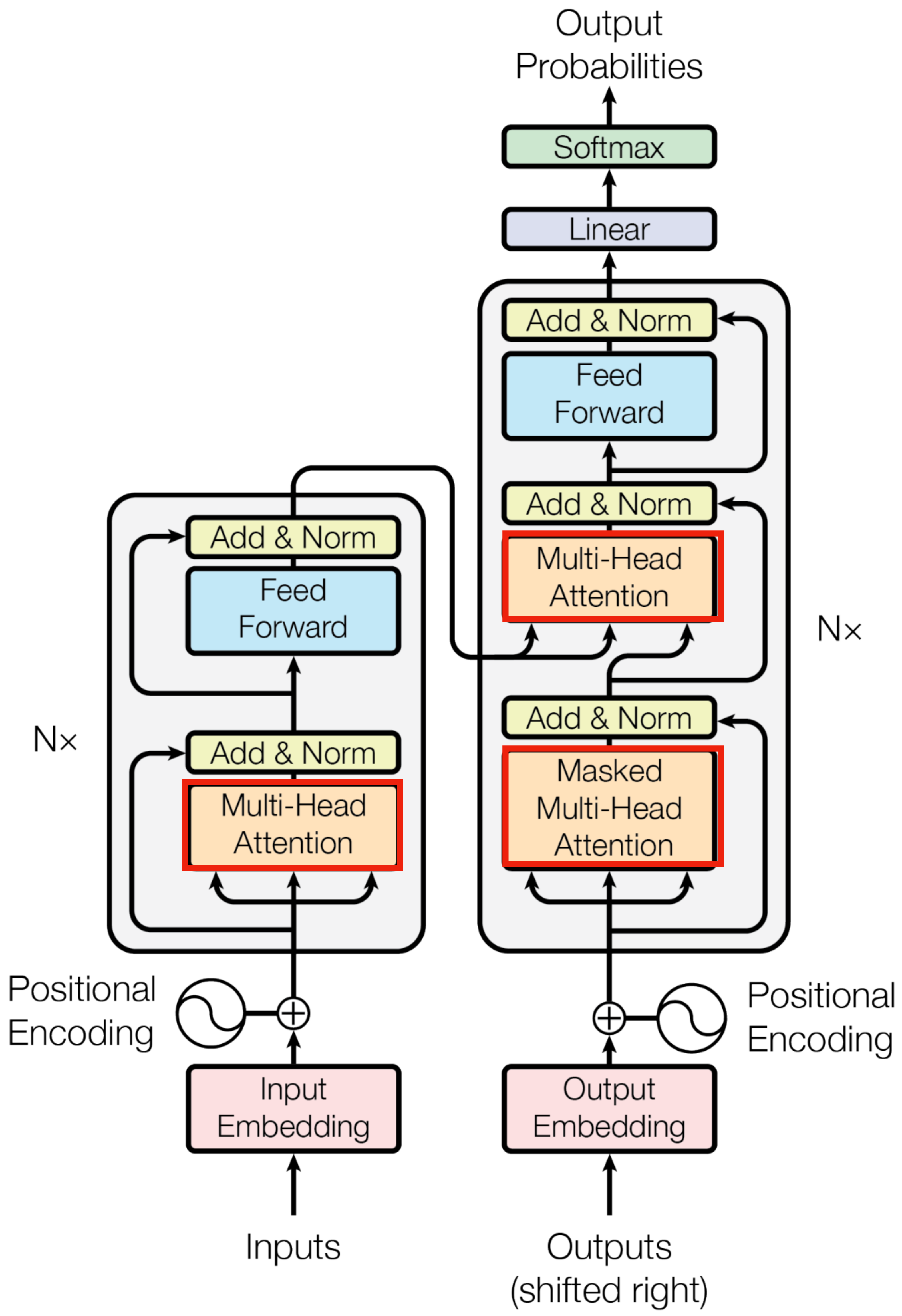
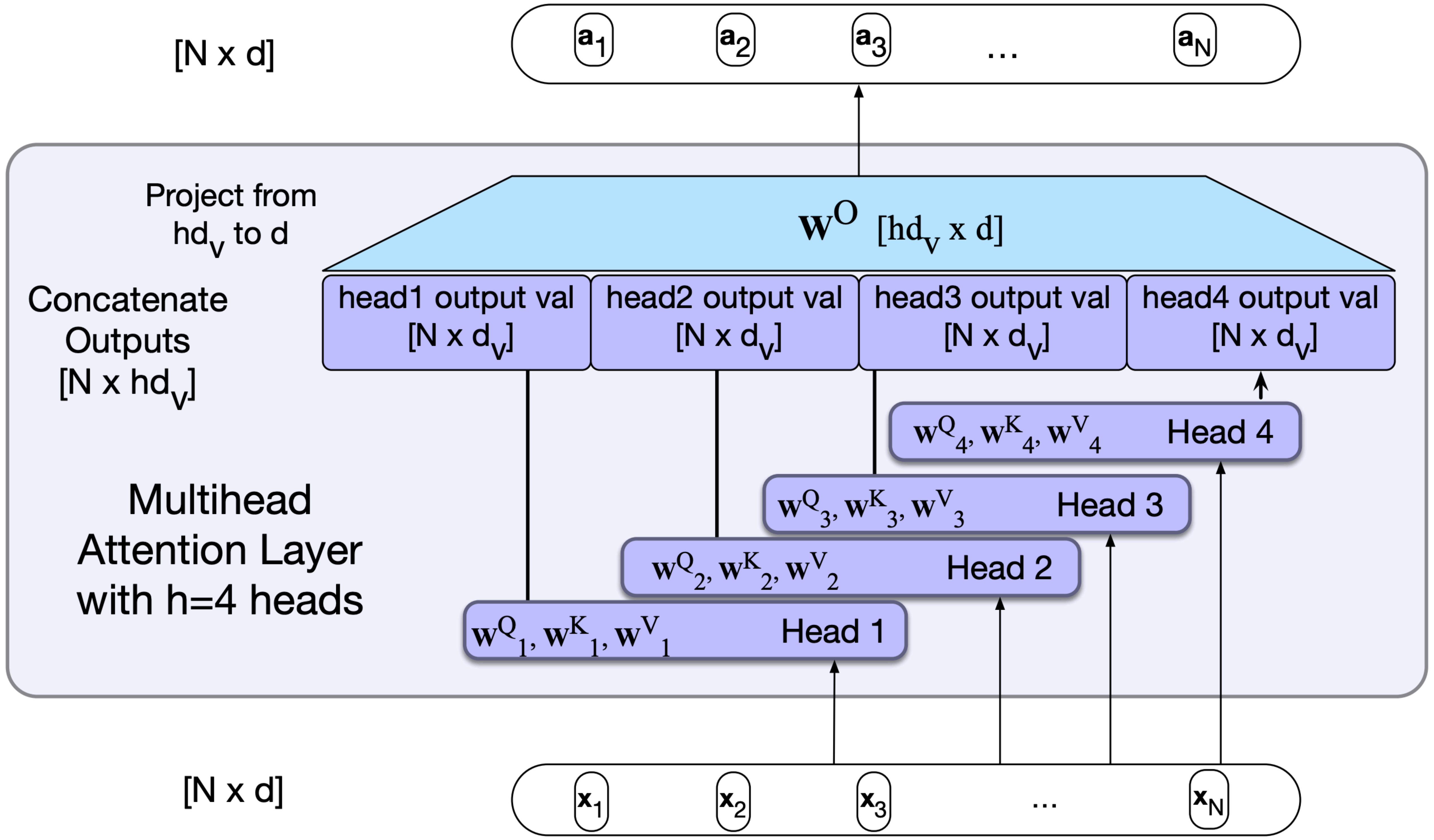


Figure 1: The Transformer - model architecture.

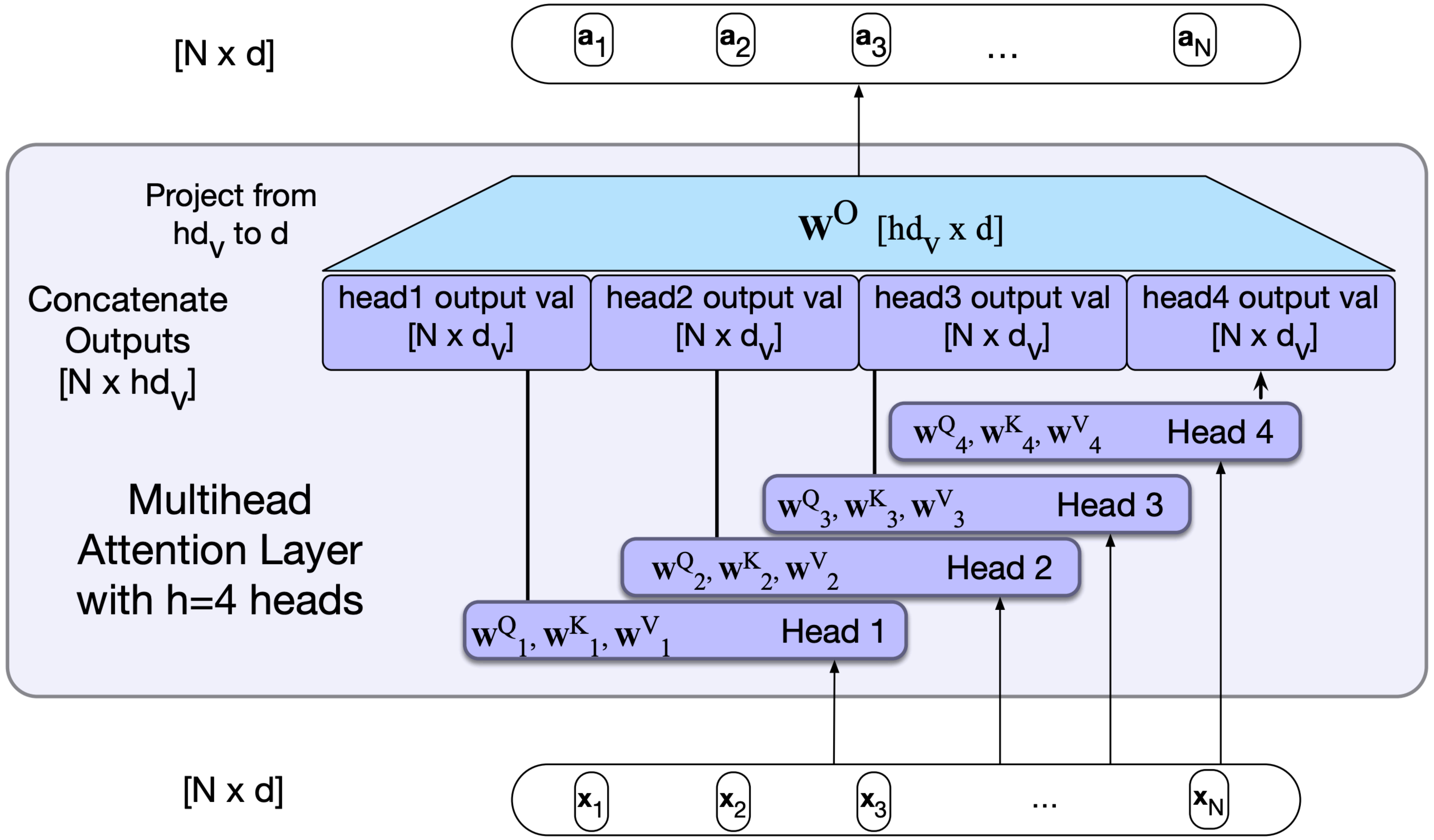
Multi-Head Attention

- Natural language can contain many distinct syntactic, semantic, and discourse relationships between words
- Hard for a single self-attention circuit to learn to capture all of these
- Instead, train multiple such circuits that operate in parallel (called multi-head attention)!


[N x d]
a1 a2 a3 … aN
Project from hdV to d
WO [hdV x d]
Concatenate Outputs [N x hdV]
head1 output val [N x dV]
head2 output val [N x dV]
head3 output val [N x dV]
head4 output val [N x dV]
WQ4, WK4, WV4 Head 4
WQ3, WK3, WV3 Head 3
WQ2, WK2, WV2 Head 2
WQ1, WK1, WV1 Head 1
Multihead Attention Layer with h=4 heads
[N x d]
x1 x2 x3 … xN

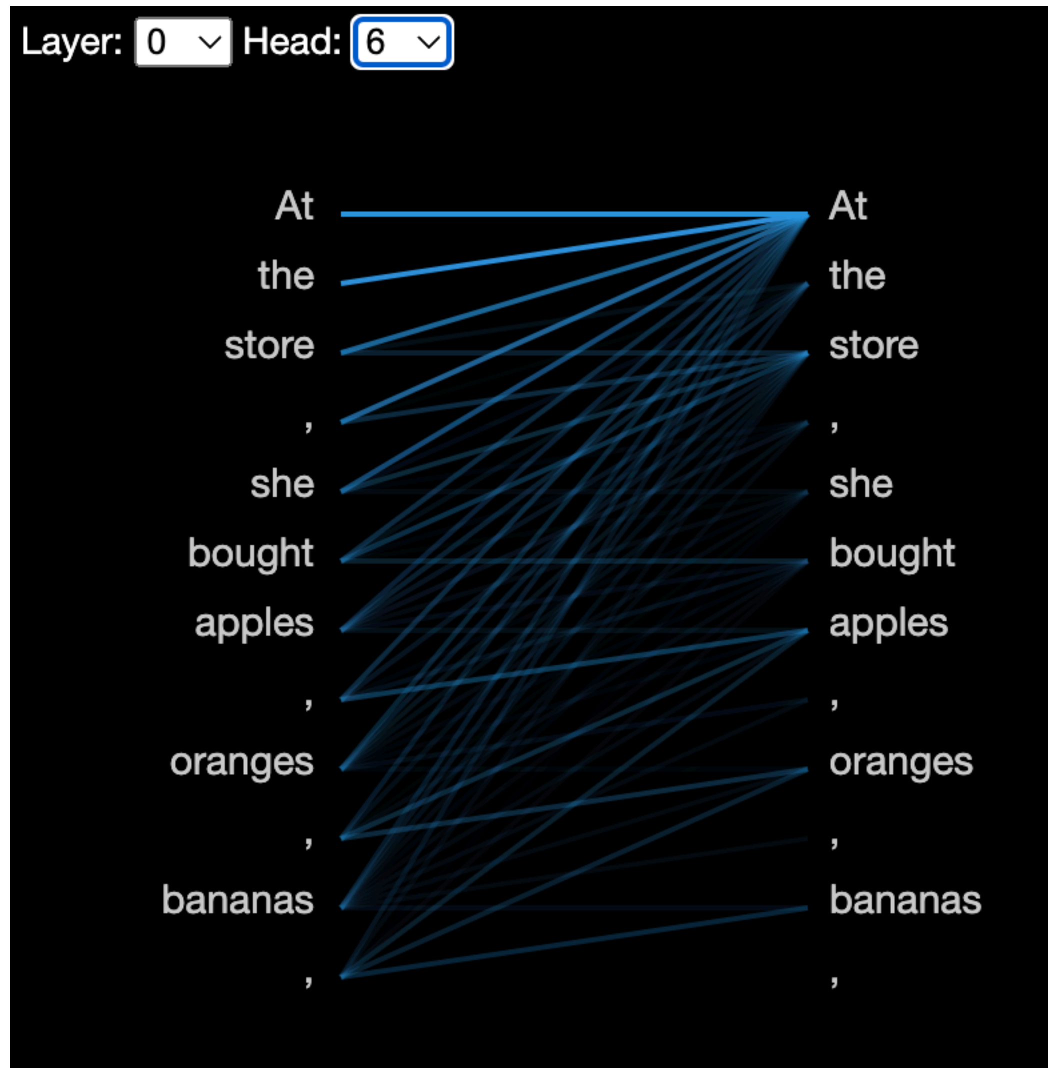
Multi-Head Attention

- If we have h heads, now we end up with h self-attention outputs
- But we want to preserve dimensions
- We concatenate all outputs, and project it back down to d dimensions with a learnt weight matrix $\mathbf{W}^O$

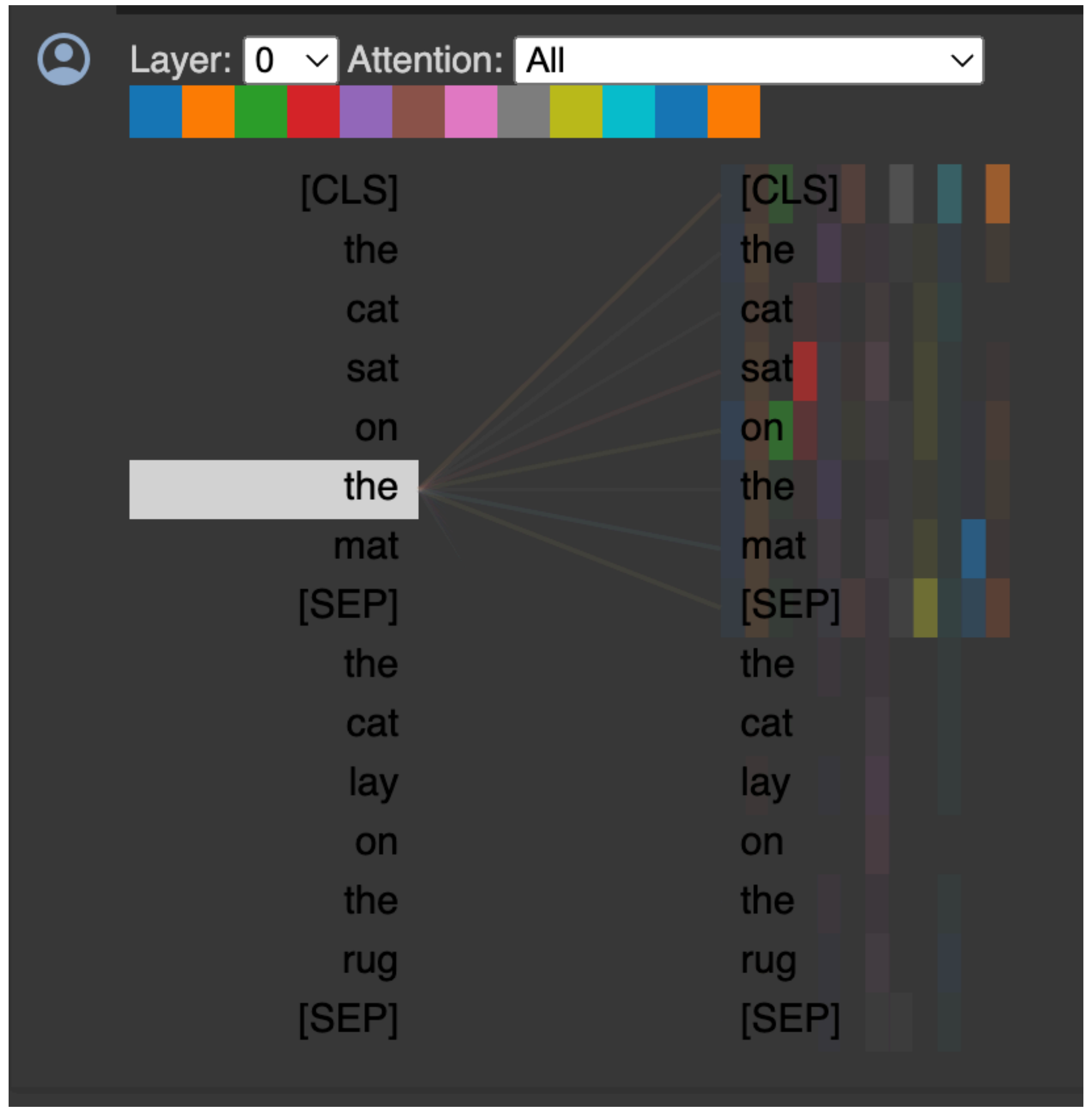


Multi-head Attention

- GPT-2: https://colab.research.google.com/drive/1s8XCCyxsKvNRWNzjWi5Nl8ZAYZ5YkLm_
- BERT (note this is bidirectional): https://colab.research.google.com/drive/1hXIQ77A4TYS4y3UthWF-Ci7V7vVUoxmQ?usp=sharing



Layer: 0 Head: 6



Layer: 0 Attention: All

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- **Positional encoding**
- Residual connections
- Layer normalization
- Language Modeling Head

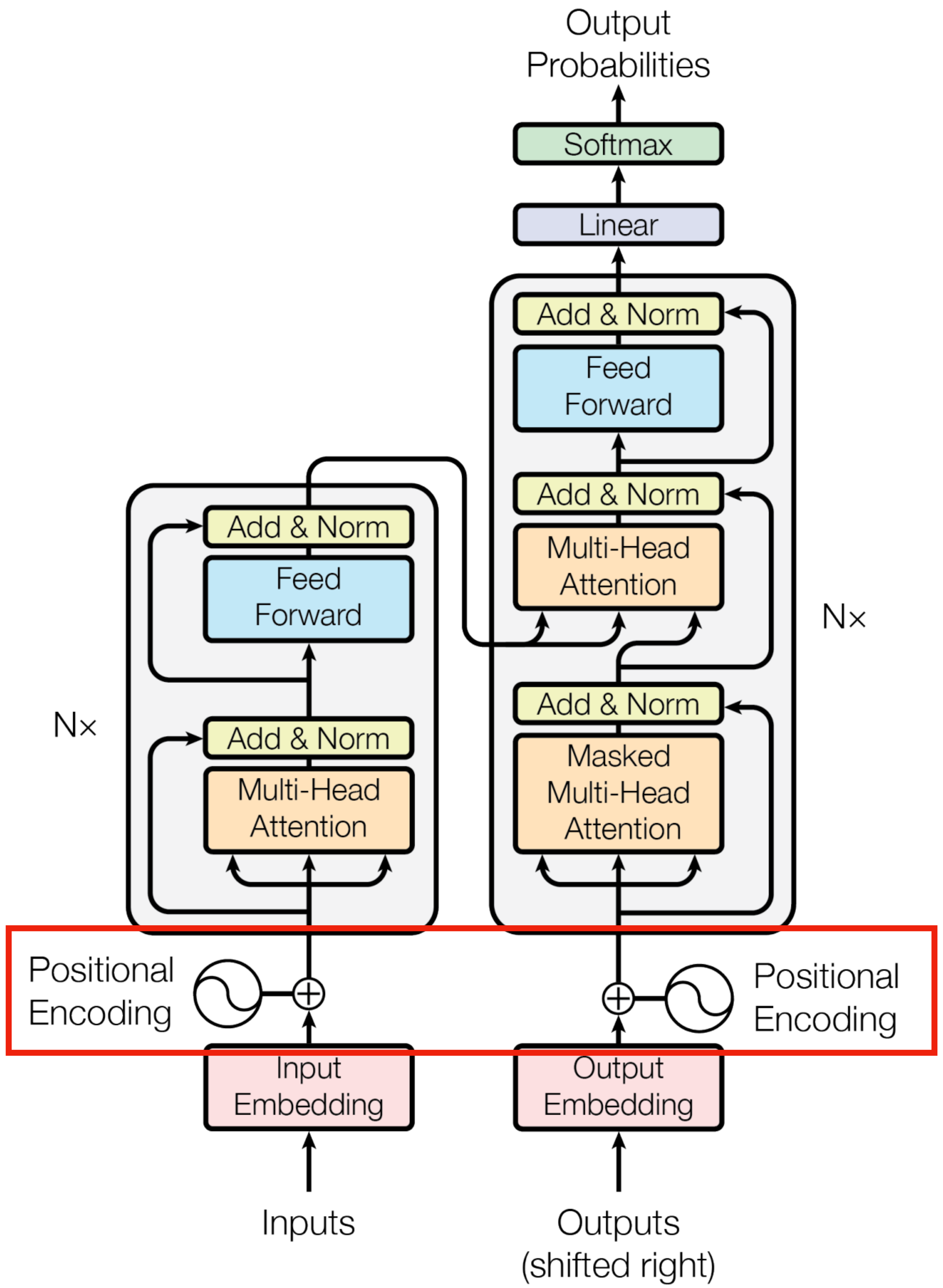
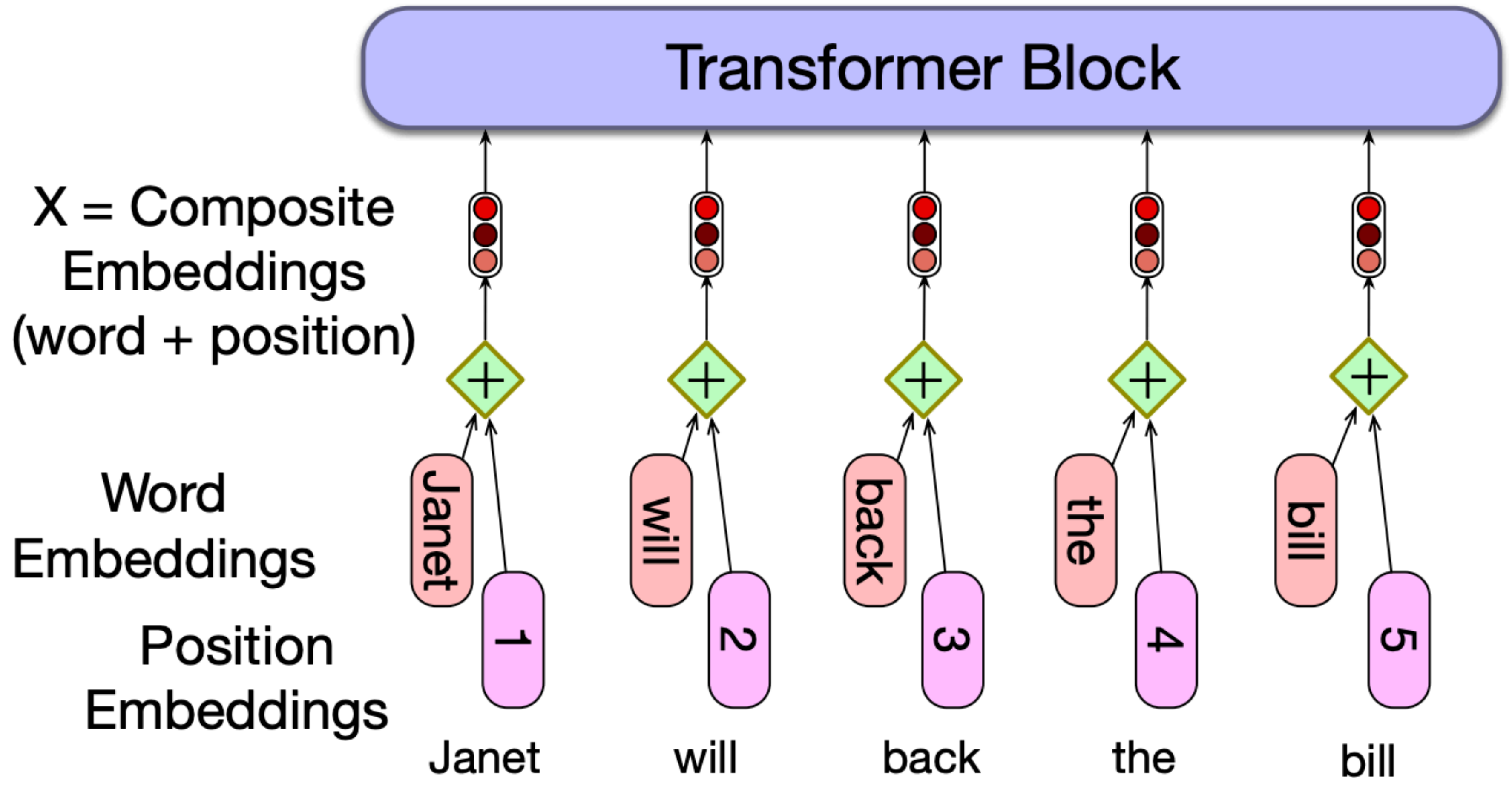


Figure 1: The Transformer - model architecture.

Positional Encoding

- Self-attention by itself is order-agnostic
- But ordering information is important in language!
- Idea: for each input token, add (not concatenate!) a vector denoting positional information to it
- Drawback to naive approach: short sequences much more common than long ones during training, so later embeddings may be poorly trained and fail to generalize



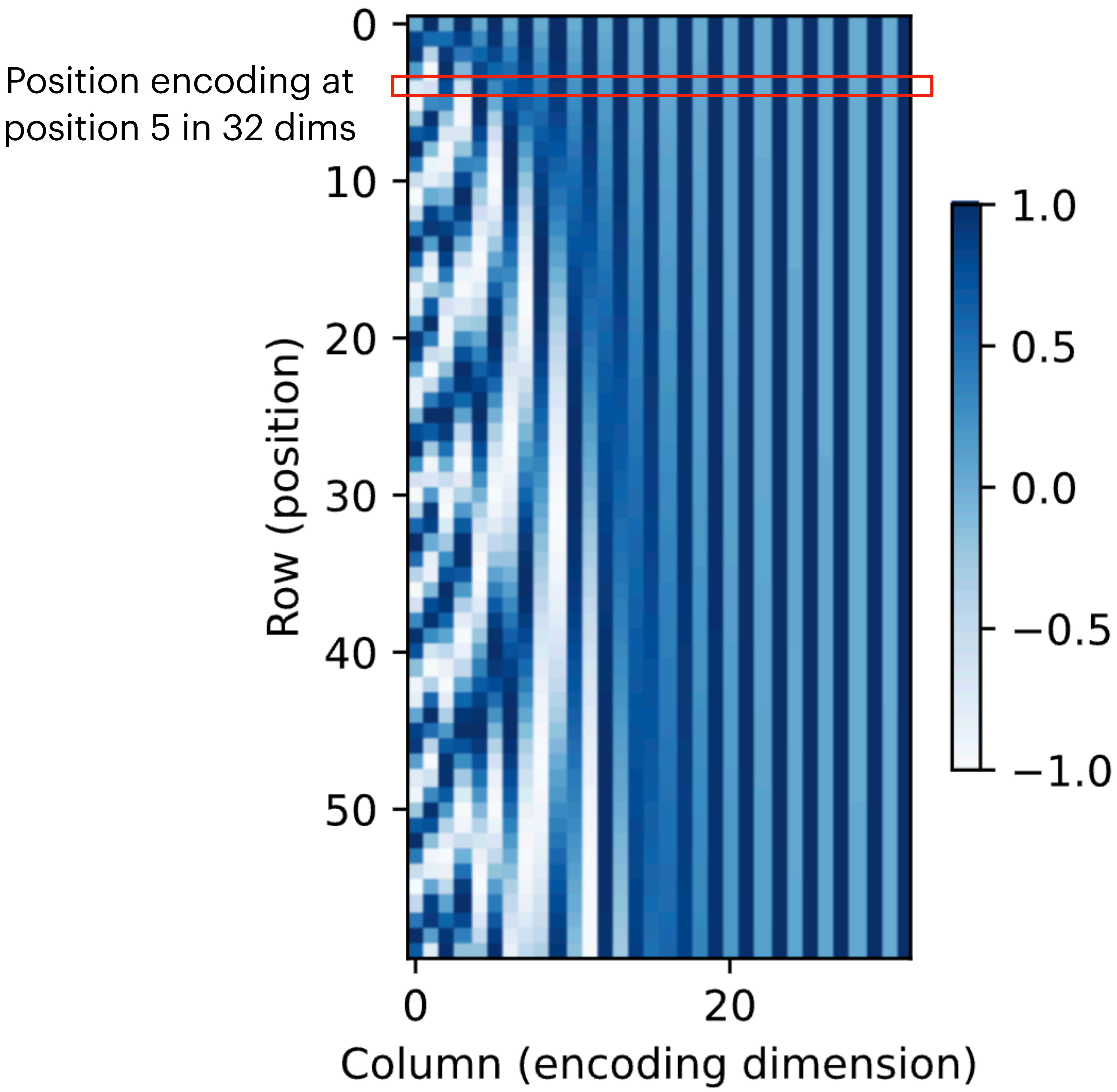
Naive approach of just using position itself as the position embedding

Positional Encoding (details unimportant)

- In original Transformer paper, authors used sinusoidal positional encoding
- Positional encoding for ith row and $2j$ or $2j+1$th column in d dimensions:

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right),$$

- $$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$
- Rationale
  - Exists an orthogonal rotation matrix that can map between positions by offsets
  - Could allow model to learn relationships between positions


Position encoding at position 5 in 32 dims

Positional Encoding

- Another common approach in the past: make positional encoding a learnable parameter during training
- Rotary Position Embeddings (RoPE) now the most widely-used positional encoding technique, which rotates the input directly instead of adding a rotation offset

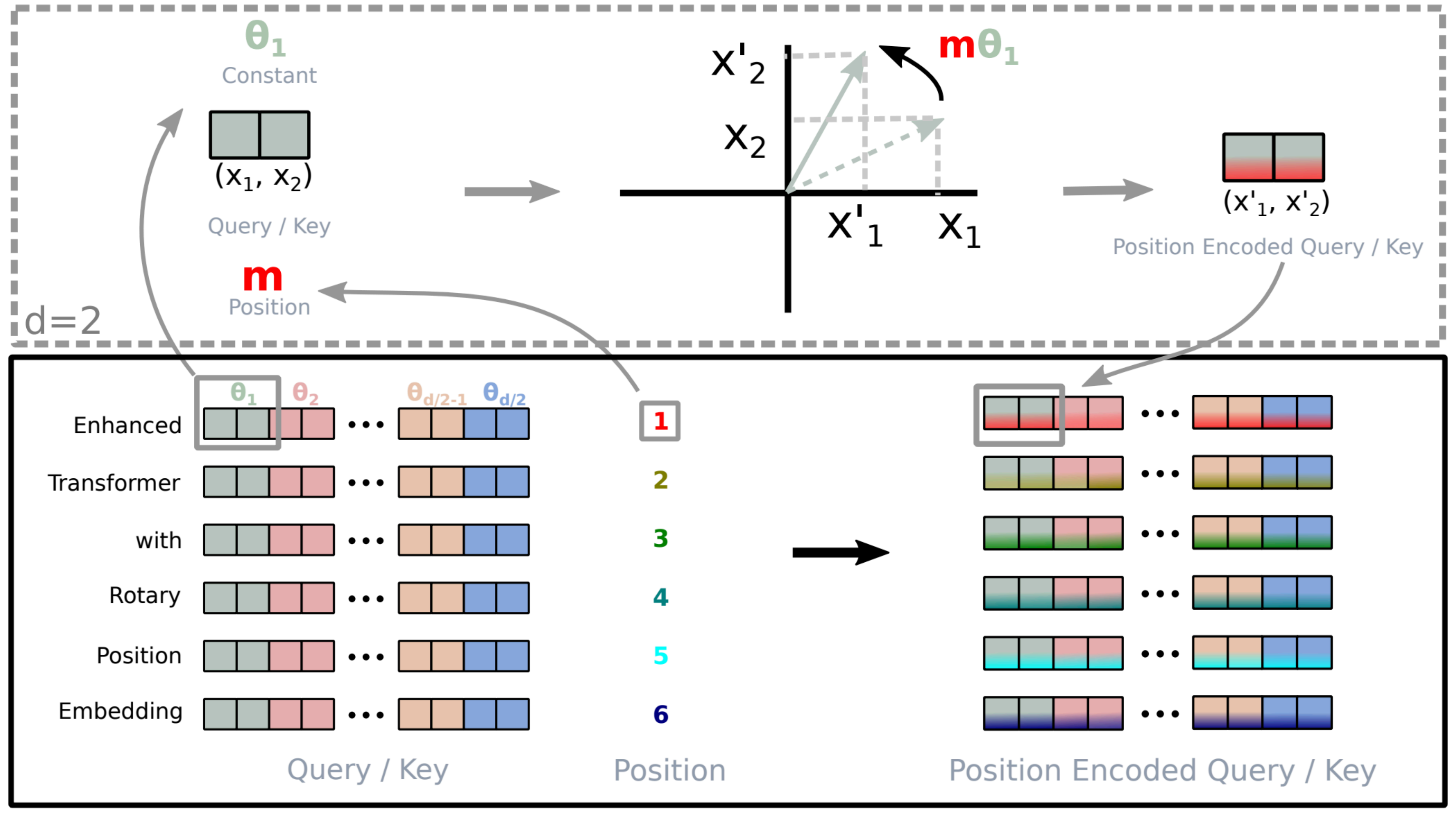


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- **Residual connections**
- Layer normalization
- Language Modeling Head

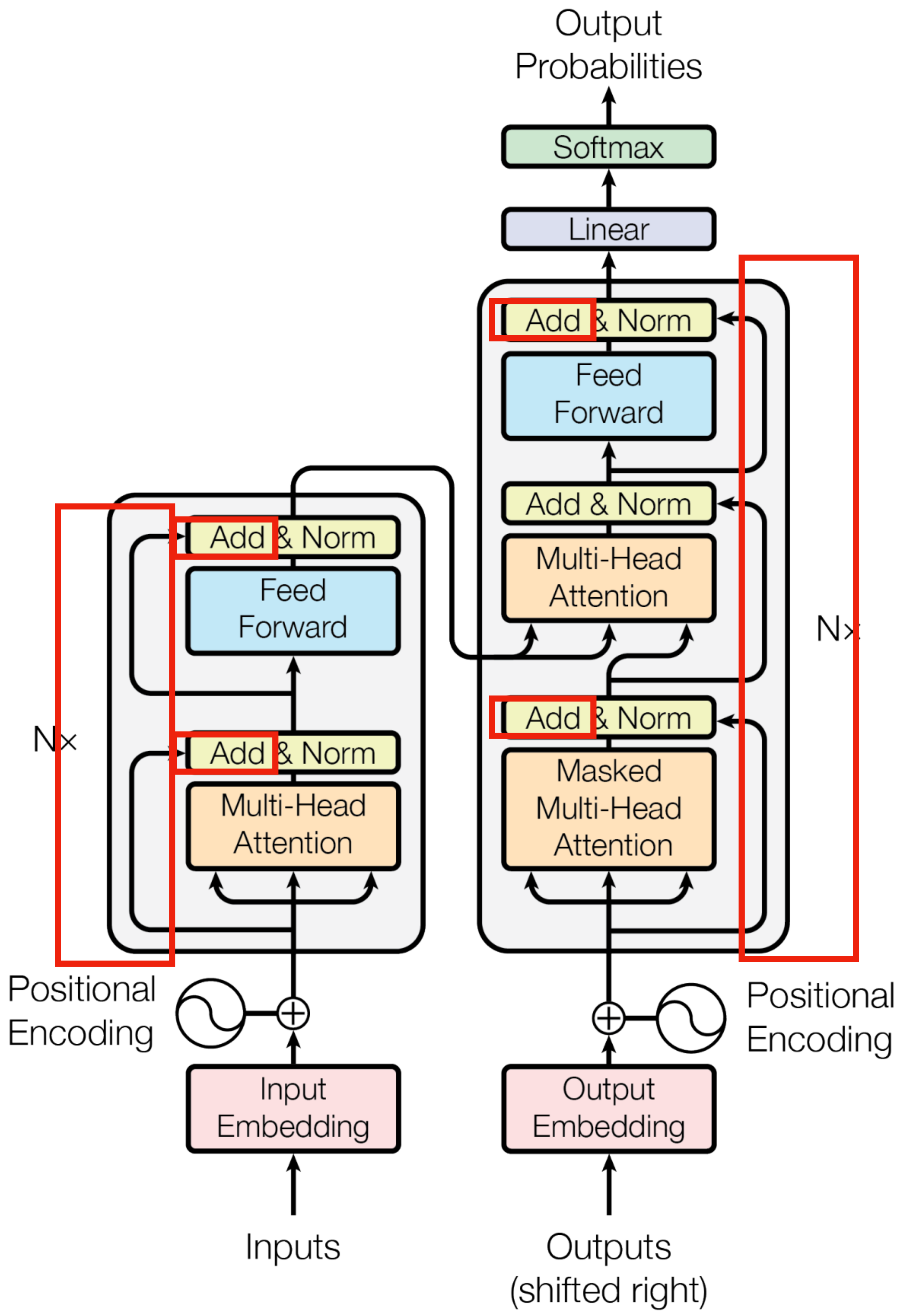
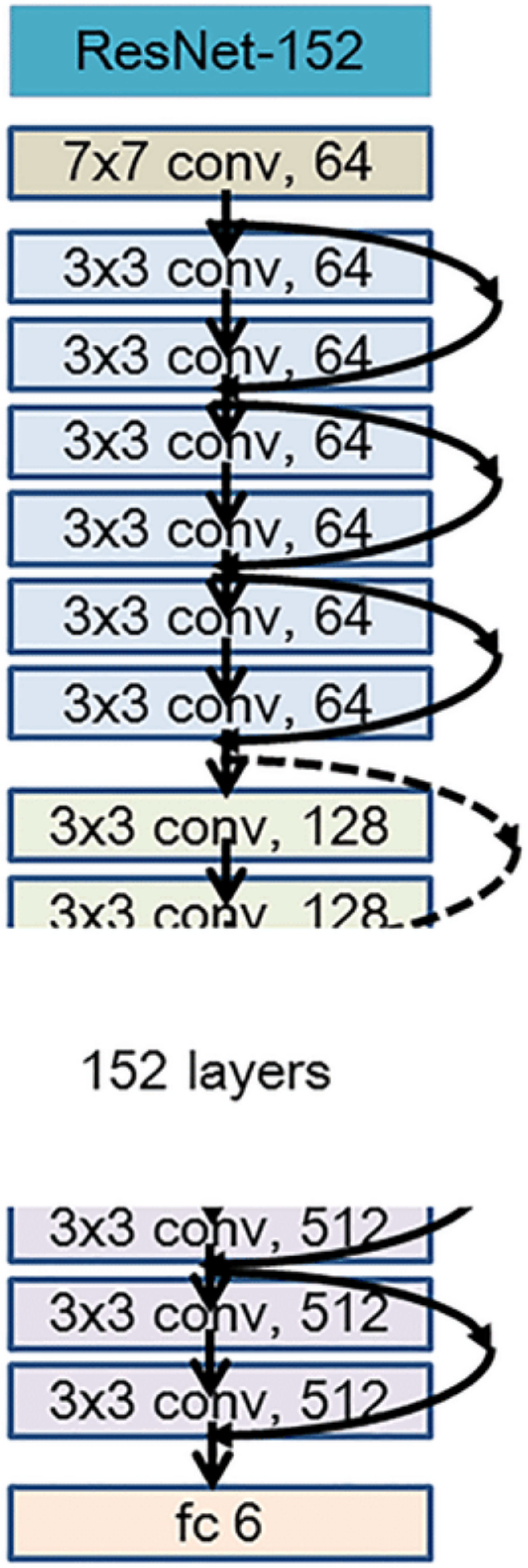


Figure 1: The Transformer - model architecture.

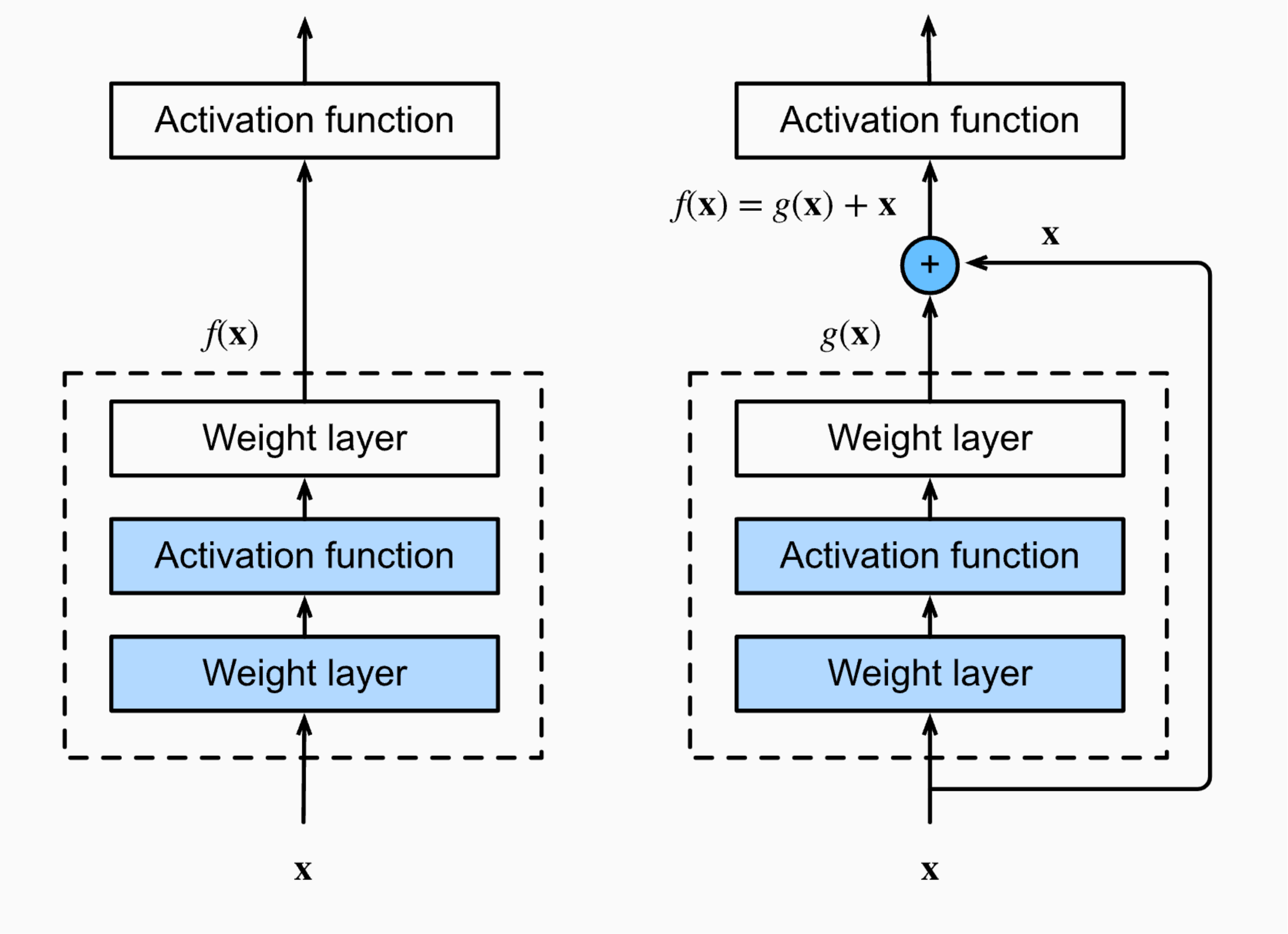
Residual Connections

- It used to be very hard to train deep architectures
- ResNet paper (200k citations, CVPR 2016 Best Paper Award) introduced the residual connection that allowed 152 layer CNN to be trained, 8x deeper than SOTA VGG networks
- Residual connections now standard in any deep neural network



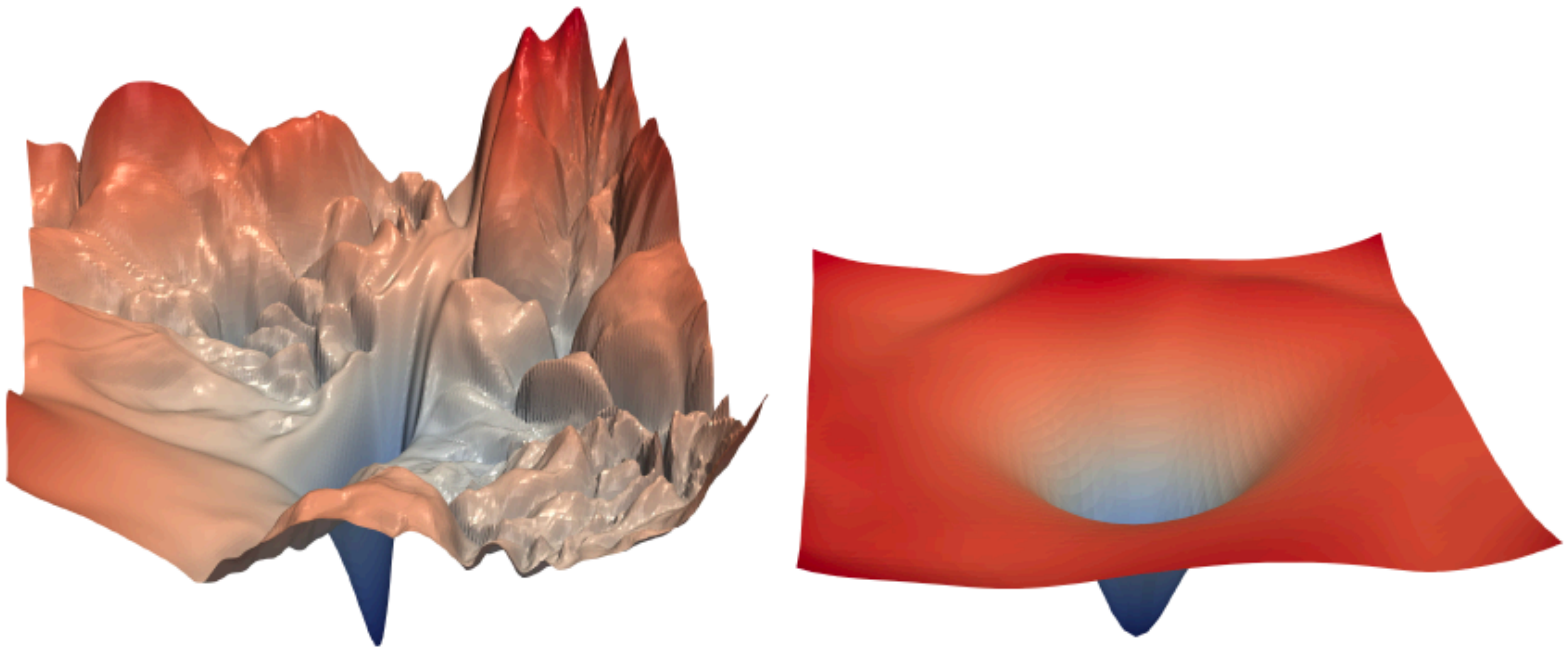
Residual Connections

- Intuition: for sequences of NN layers, it is hard to learn $f(x)$, but much easier to learn the residue $g(x) = f(x) - x$
- Each layer hence performs iterative refinement of representation from previous layer
- Residual connections hence allow for this reparameterization $f(x) = g(x) + x$



Residual Connections

- Also known as skip connections



(a) without skip connections

(b) with skip connections

Visualizing the Loss Landscape of Neural Nets

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- **Layer normalization**
- Language Modeling Head

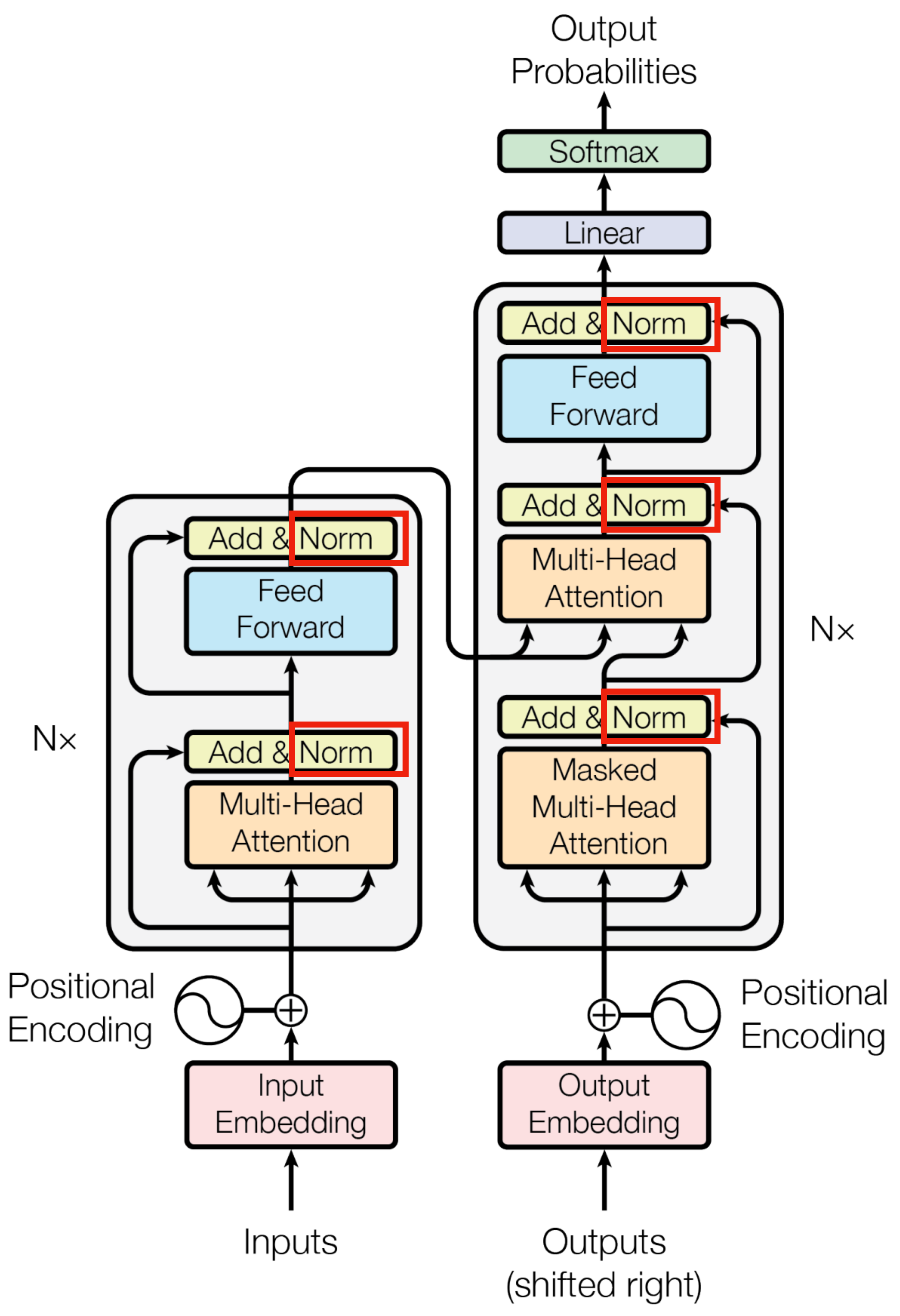


Figure 1: The Transformer - model architecture.

Layer Normalization

- Not to be confused with batch normalization
- Scales and shifts the input to keep values in a range
- Helps with training stability
- Suppose input $x \in \mathbb{R}^d$, then for learnable gain γ and offset β:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}\left(x_i - \mu\right)^2}$$

-

$$\text{LayerNorm(x)} = \gamma\frac{x - \mu}{\sigma} + \beta$$

Transformer Block

- We've now covered almost all the pieces
- Highlighted region is a Transformer block
- We repeat & stack both encoder and decoder Transformer blocks many times to learn deeper representations

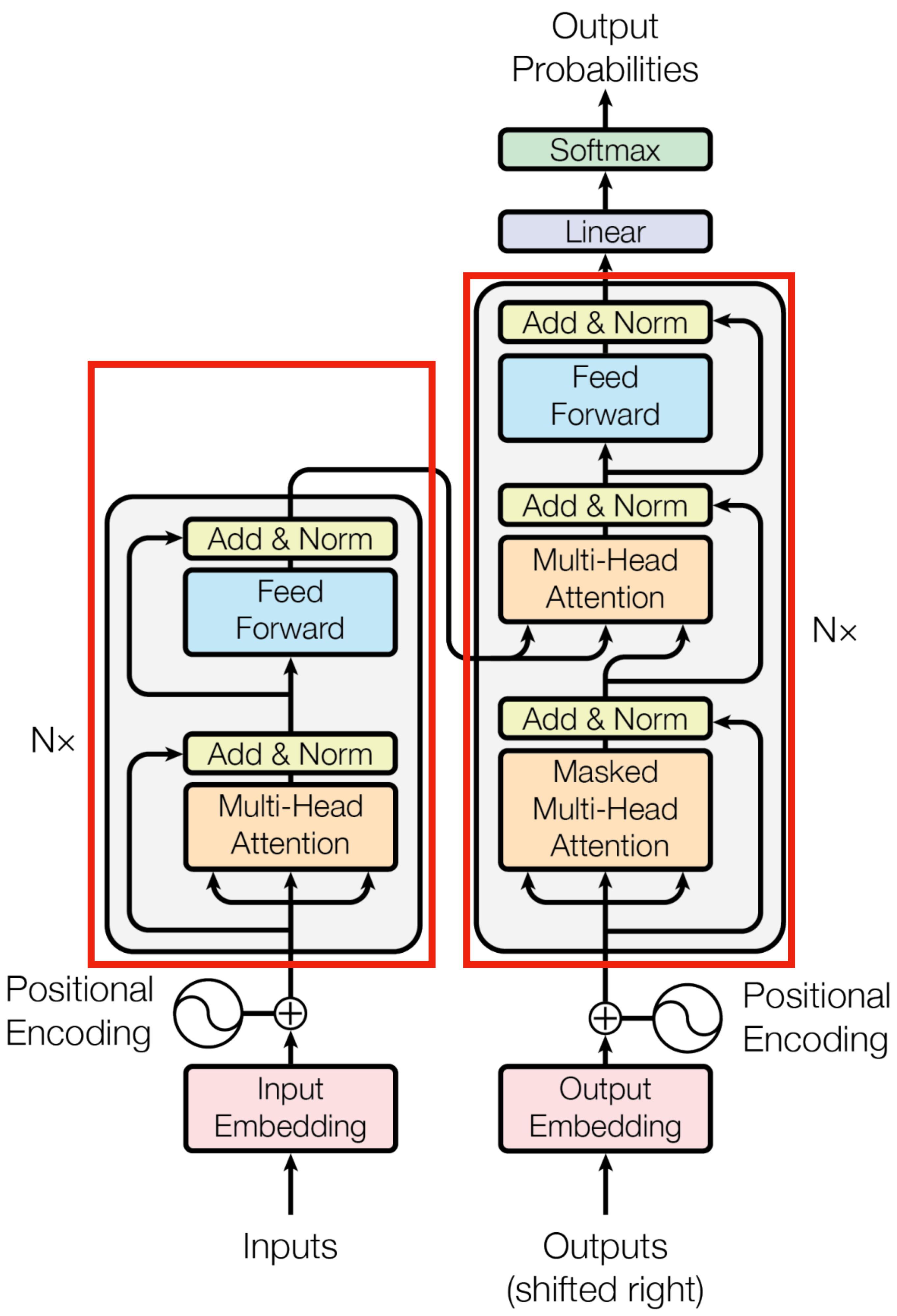


Figure 1: The Transformer - model architecture.

Transformer Outline

- Encoder and decoders
- Embeddings
- Attention mechanism
- Self-attention
- Multi-head Attention
- Positional encoding
- Residual connections
- Layer normalization
- **Language Modeling Head**

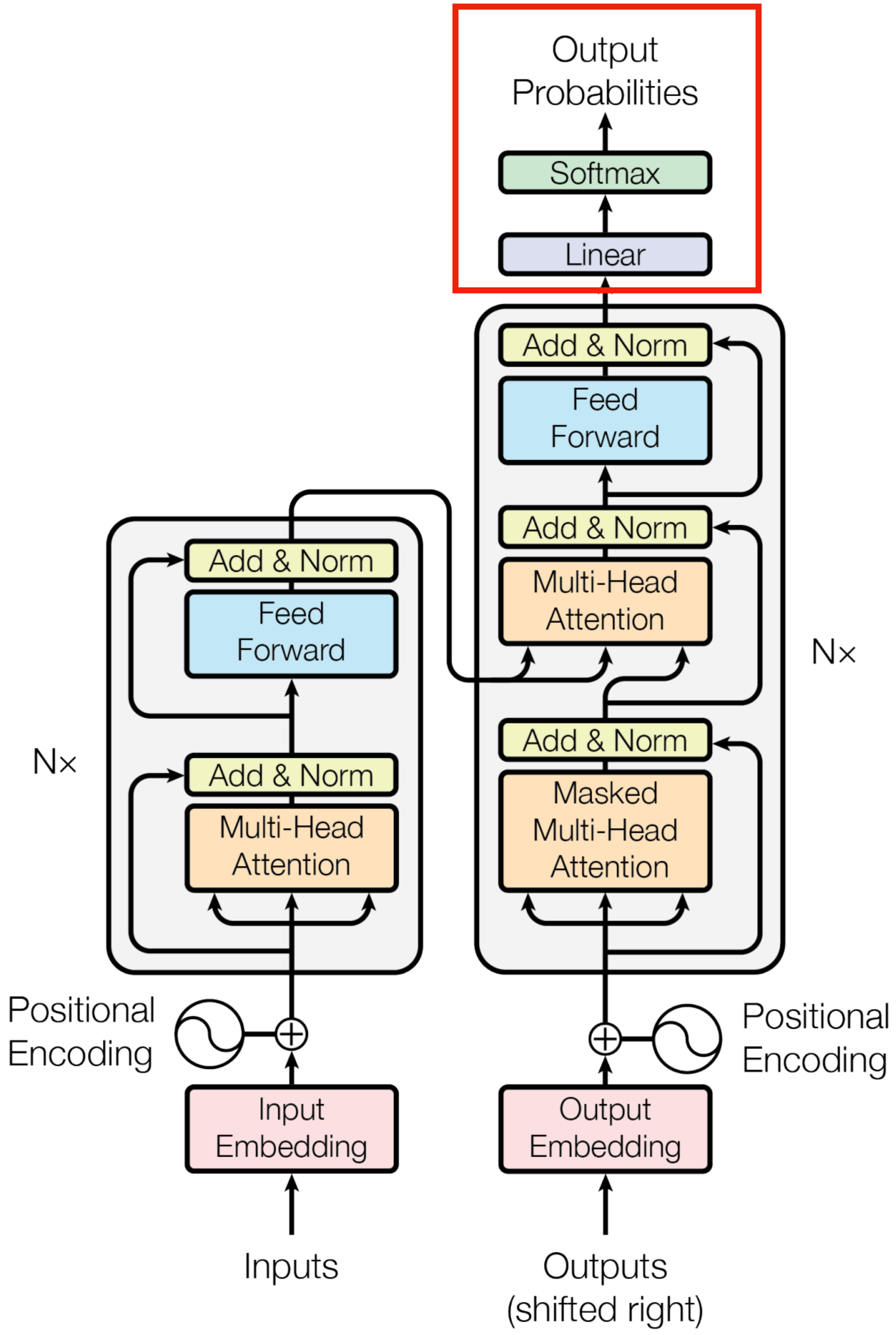
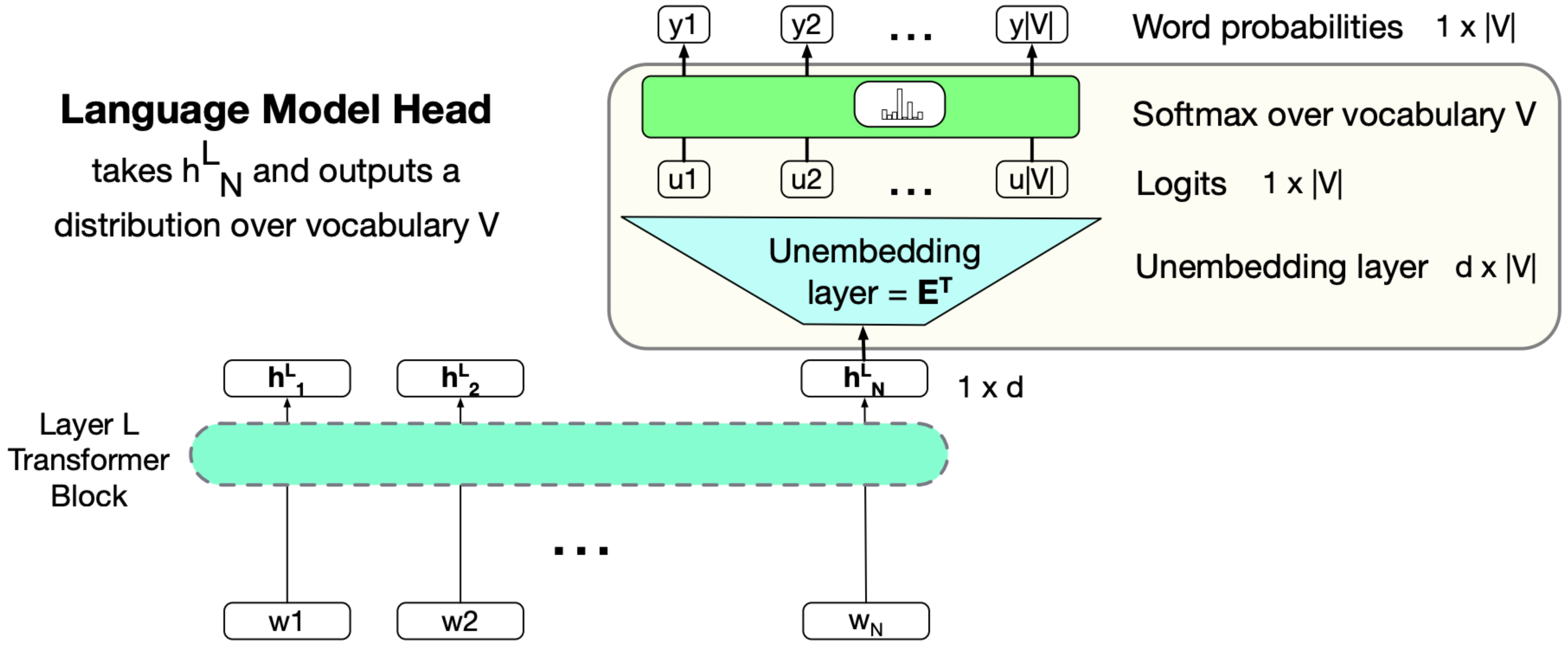


Figure 1: The Transformer - model architecture.

Language Modeling Head

- To convert outputs from last layer of Transformer block to probabilities over tokens
- Unembedding layer usually transpose of embedding matrix, hence performs reverse mapping
- Softmax normalizes outputs to follow a probability distribution



Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
y1
y2
…
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
…
u|V|
Logits 1 x |V|
Unembedding layer = $\mathbf{E}^T$
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
…
w1
w2
w_N

You did it!

- This was a long ride but I hope you enjoyed it
- You now understand (a big part of) how Transformers work!
- Architectures and specific techniques always evolving
- Important thing is to understand the problems and the spirit of the techniques
- 🥳🥳🎉🎉👏👏
- Further reading:
  - Neural Scaling Laws, Low Rank Approximation (LoRA), Mixture of Experts (MoE), Flash Attention, Quantization, Speculative Decoding, Mechanistic Interpretability, State Space Models, etc...

References

- Deep Learning, Goodfellow et al. 2016
- Dive into Deep Learning
- Speech and Language Processing (3rd ed. draft)
- CMU 11-667 Large Language Models Methods and Applications
- Stanford CS324 - Large Language Models
- Deep Learning with Python, François Chollet