Lecture 1
Introduction to Machine Learning

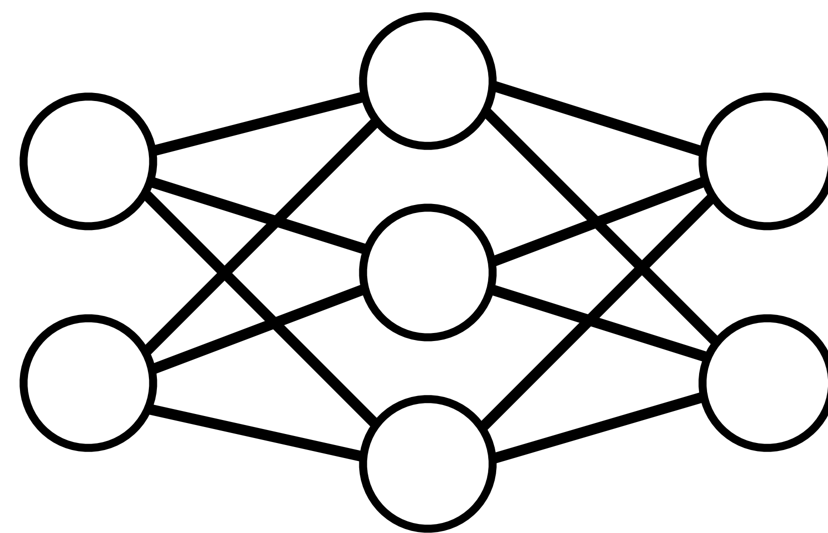
FABIAN RUEHLE (NORTHEASTERN UNIVERSITY & IAIFI)

ML in Maths and Physics 2023
University of Oxford

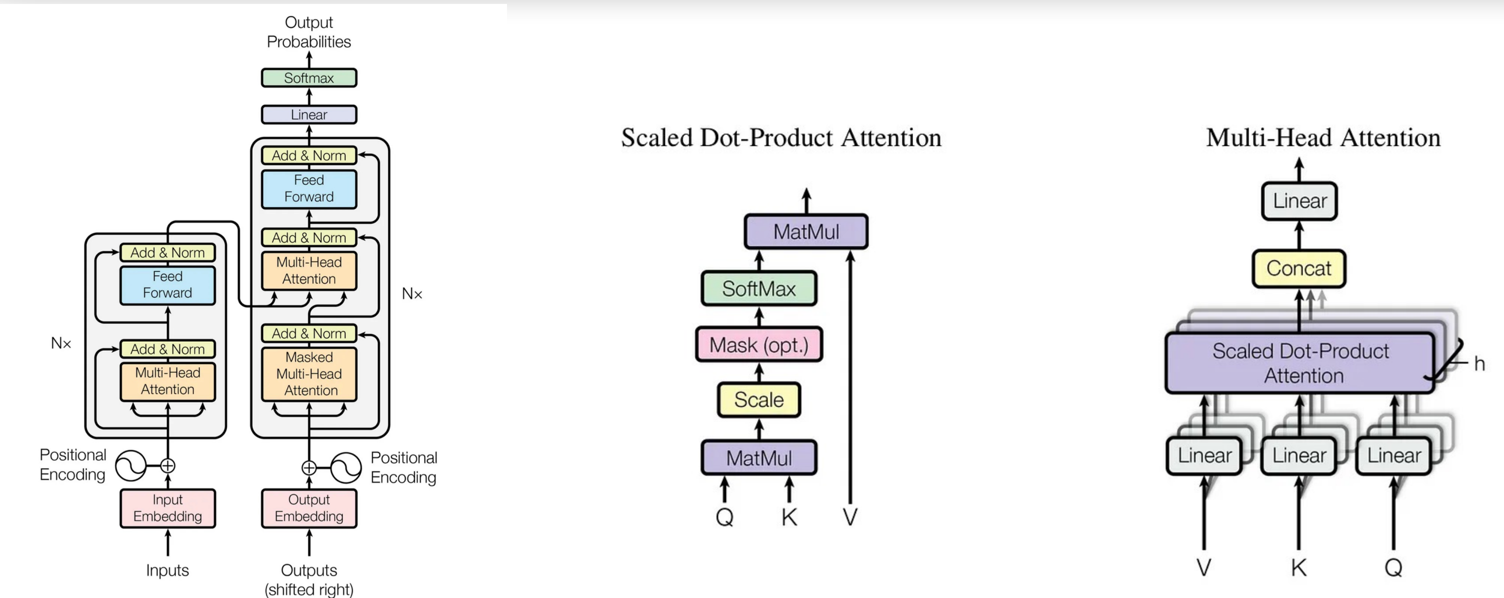
17-21 July 2023

What you will learn in the next 90 min

1. Introduction to NNs

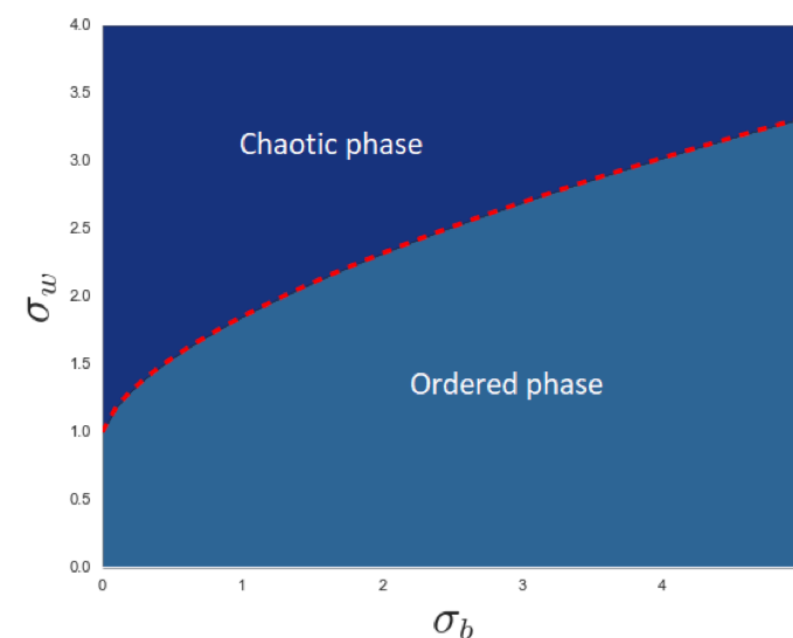


3. The transformer architecture

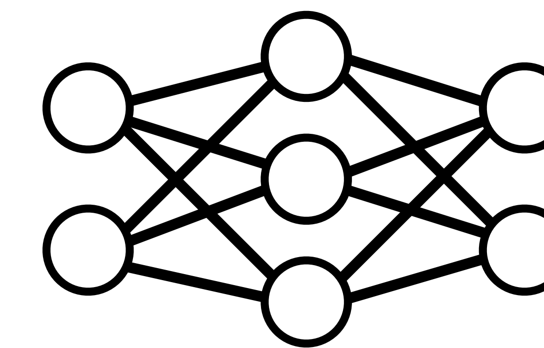


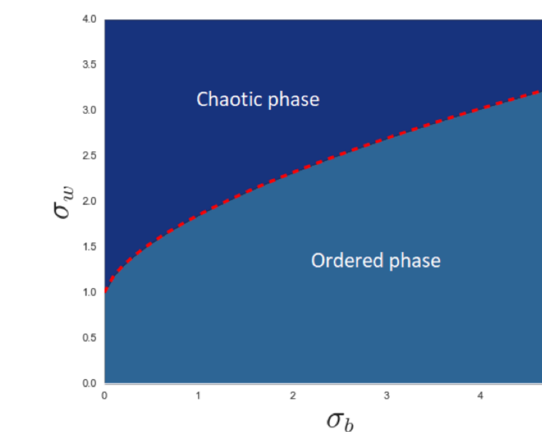
2. Training with backpropagation

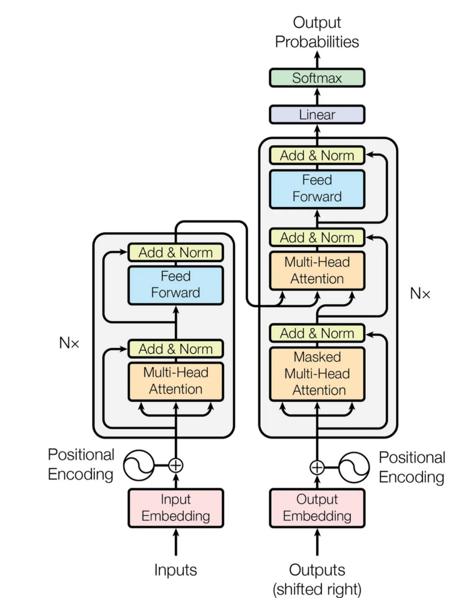
$$\theta^{(i)} \to \theta^{(i)} - \alpha \frac{\partial L}{\partial \theta^{(i)}}$$

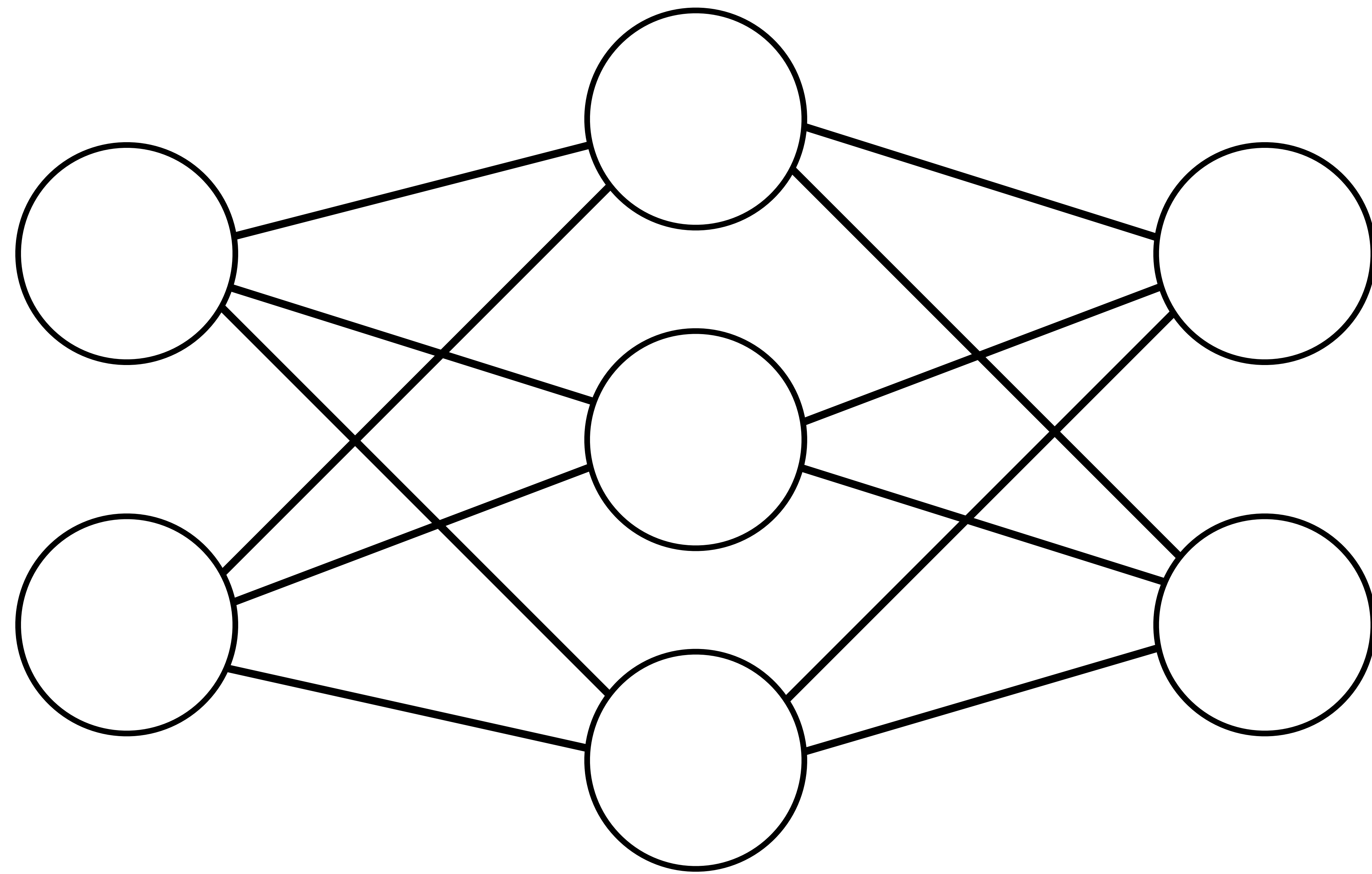


4. Recap





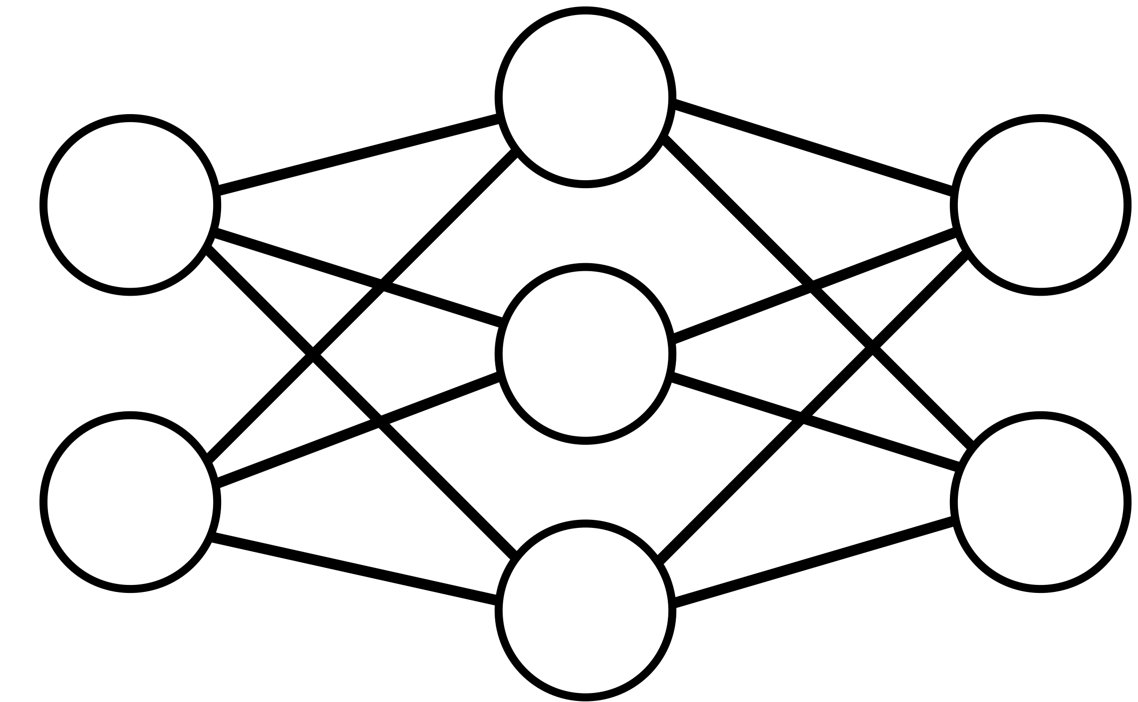




Intro to Neural Networks

Neural Networks

- NN is a family of maps $f_\theta : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$ parameterized by parameters θ
- The map f_θ is a concatenation of
  - affine transformations $z_j \to W^i_j z_i + b_j$
  - element-wise non-linear functions $z_j \to \sigma(z_j)$



$$\begin{pmatrix} z_0 \\ z_1 \end{pmatrix} \mapsto \begin{pmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} \end{pmatrix} \cdot \begin{pmatrix} \tanh(w^0_{1,1} z_1 + w^0_{1,2} z_2 + b^0_1) \\ \tanh(w^0_{2,1} z_1 + w^0_{2,2} z_2 + b^0_2) \\ \tanh(w^0_{3,1} z_1 + w^0_{3,2} z_2 + b^0_3) \end{pmatrix} + \begin{pmatrix} b^1_1 \\ b^1_2 \end{pmatrix}$$

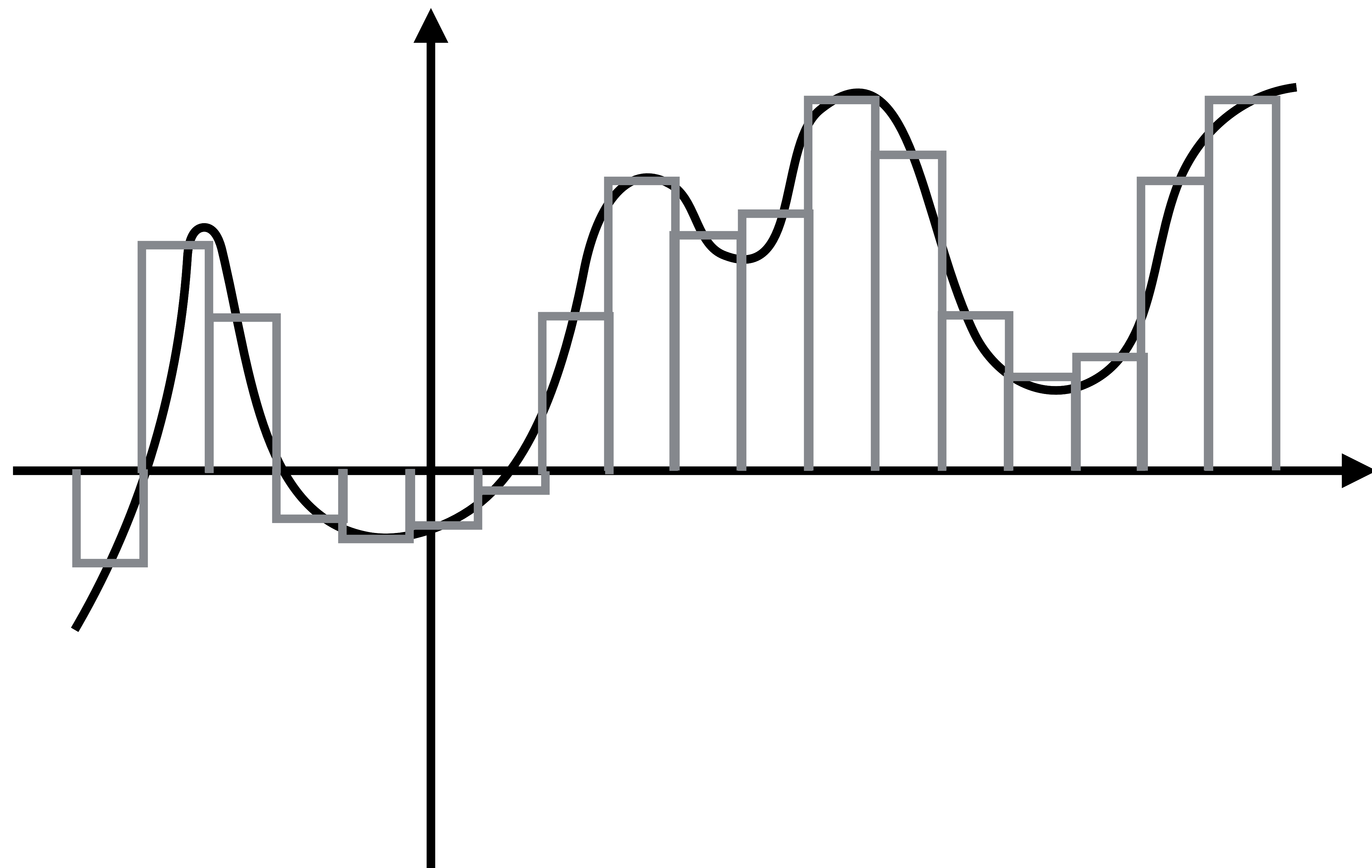
- Compare with
  - Taylor Expansion $f(z) = \sum_i w_i (z - b_i)^i$ with $w_i = \partial^n f(x)|_{x=b}$
  - Fourier Expansion $f(z) = \sum_i w^1_i \cos(w^0_i z + b_i)$

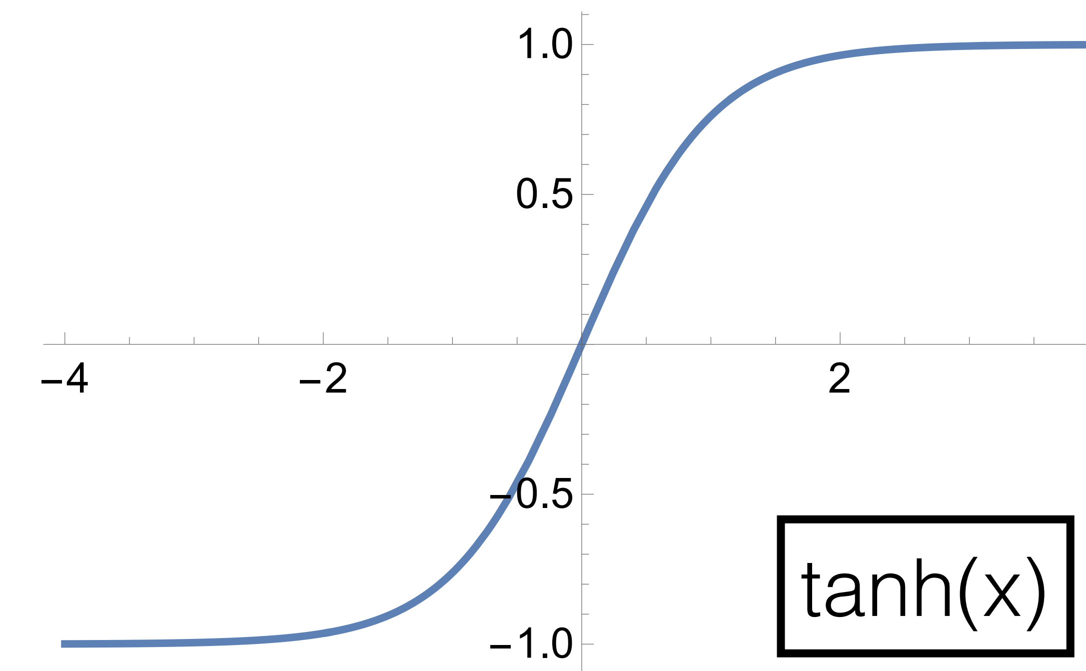
Neural Networks as function approximations

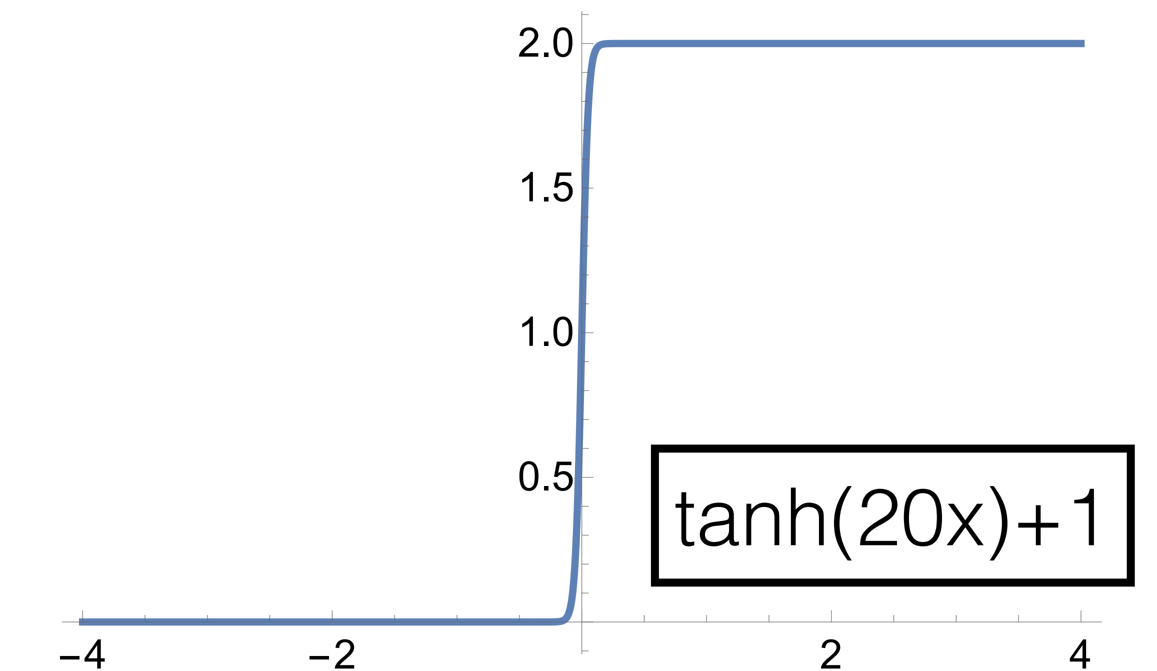
- So why would one ever do this?

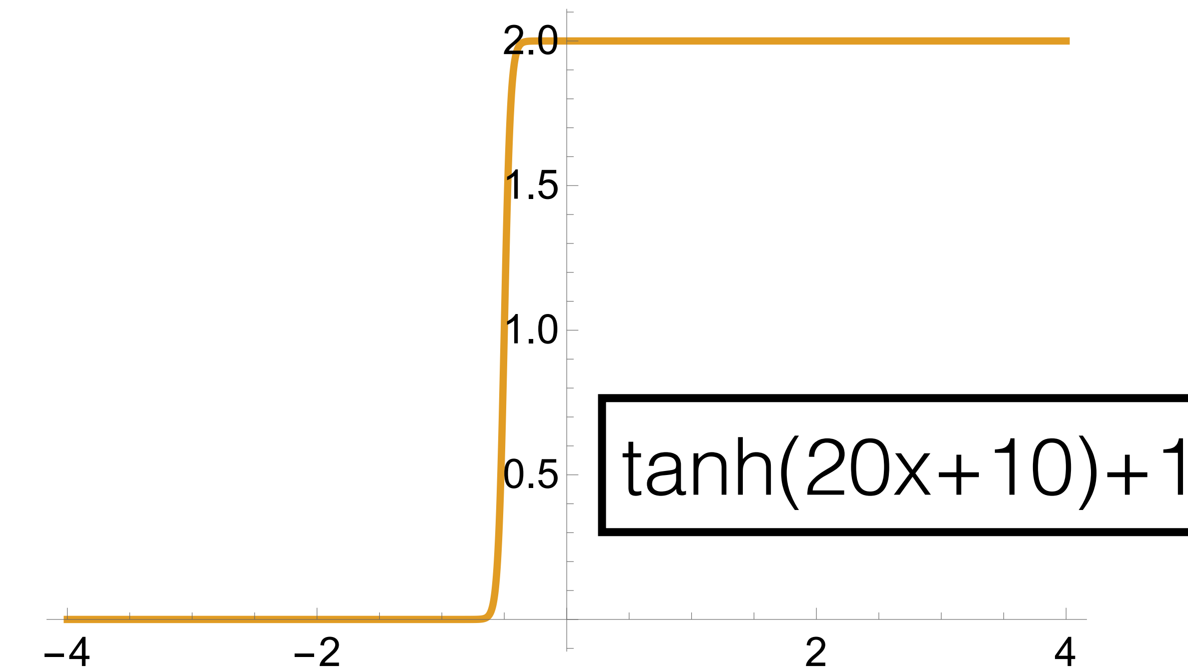
Theorem: NNs are universal function approximators

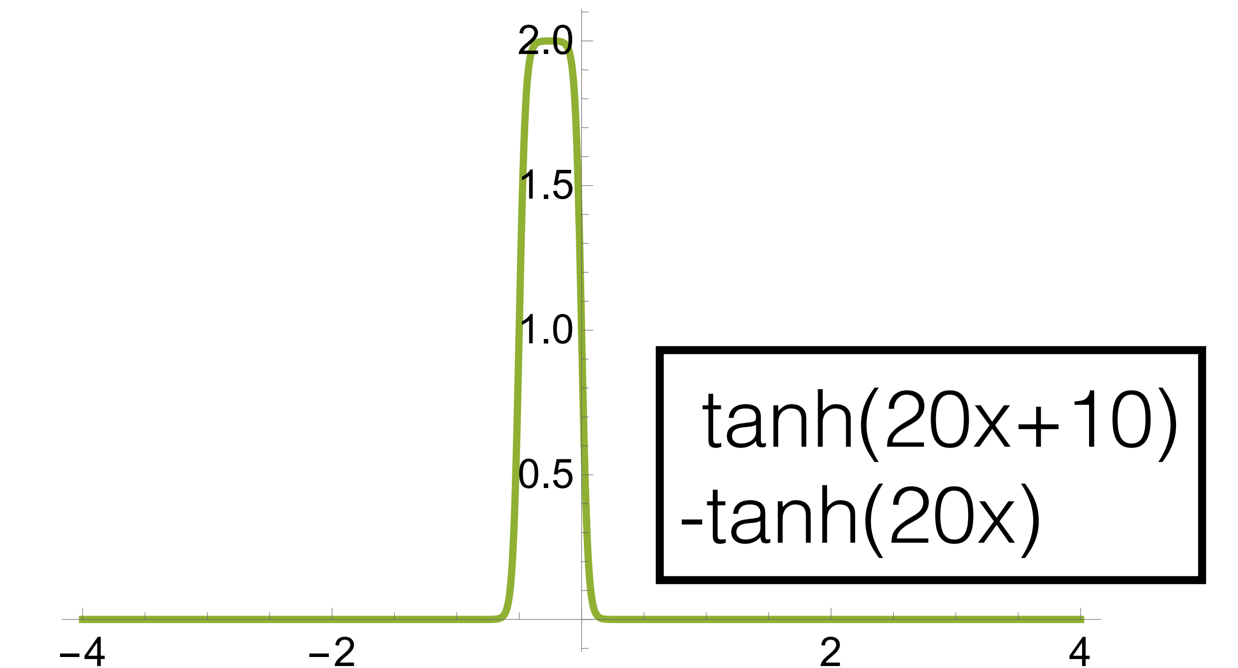
[Cybenko `89; Hornik `91; Leshno et.al. `93; Pinkus `99]







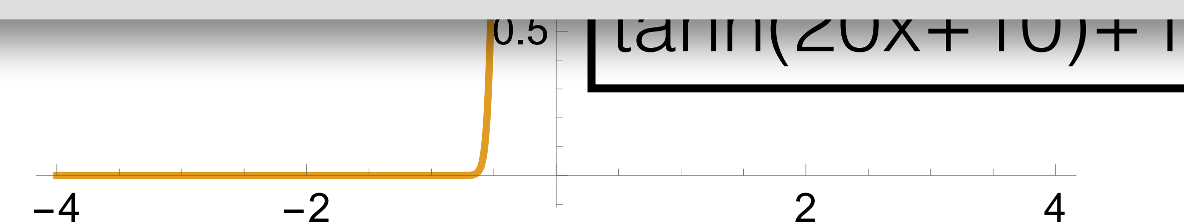


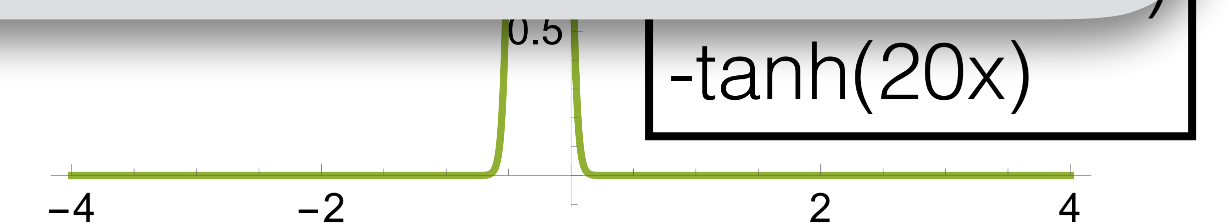


Neural Networks as function approximations

- So why would one ever do this?

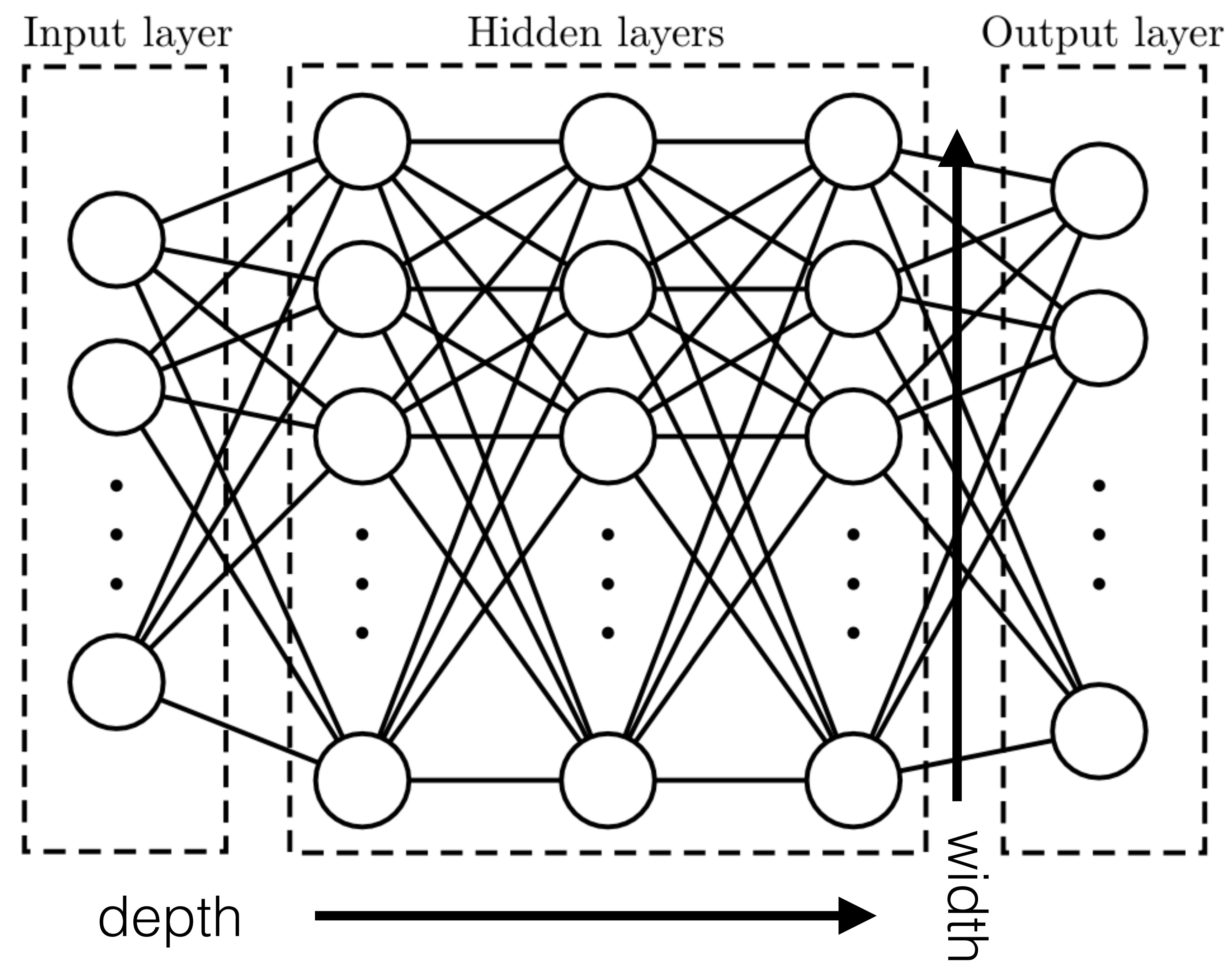
DISCLAIMER:
This is ***NOT***
what a NN does



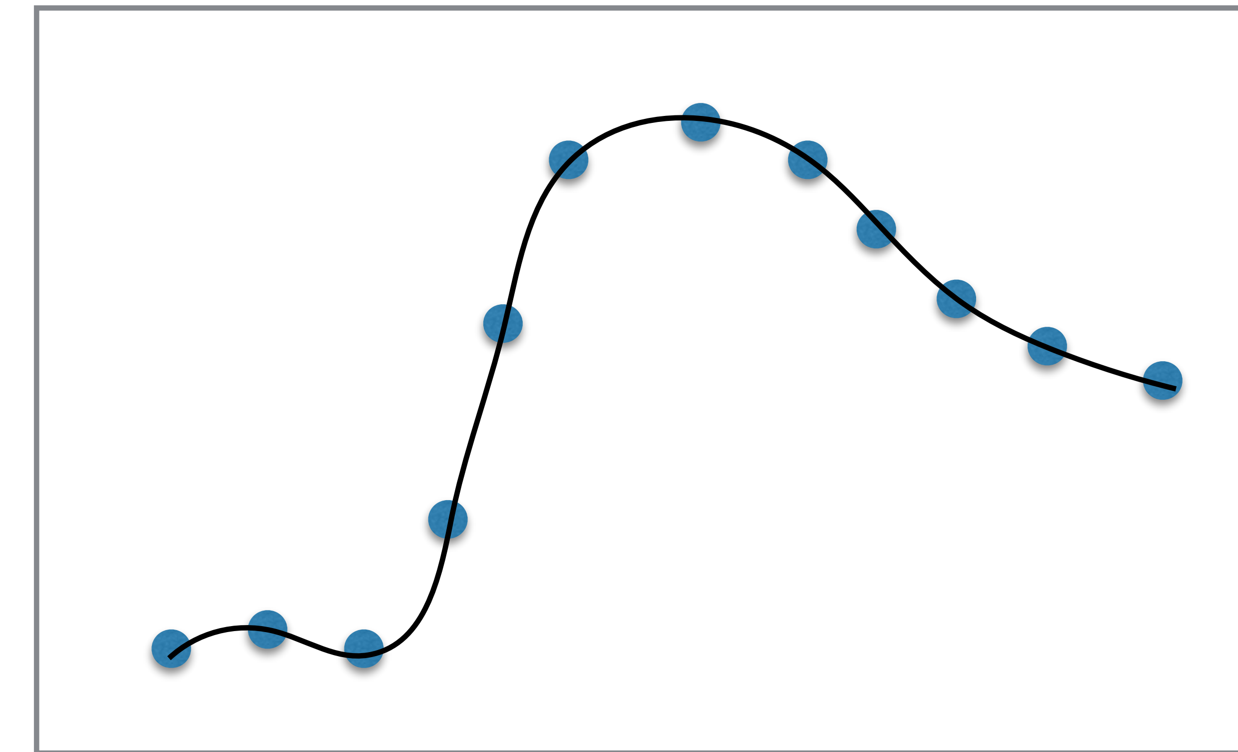


Neural Networks as function approximations

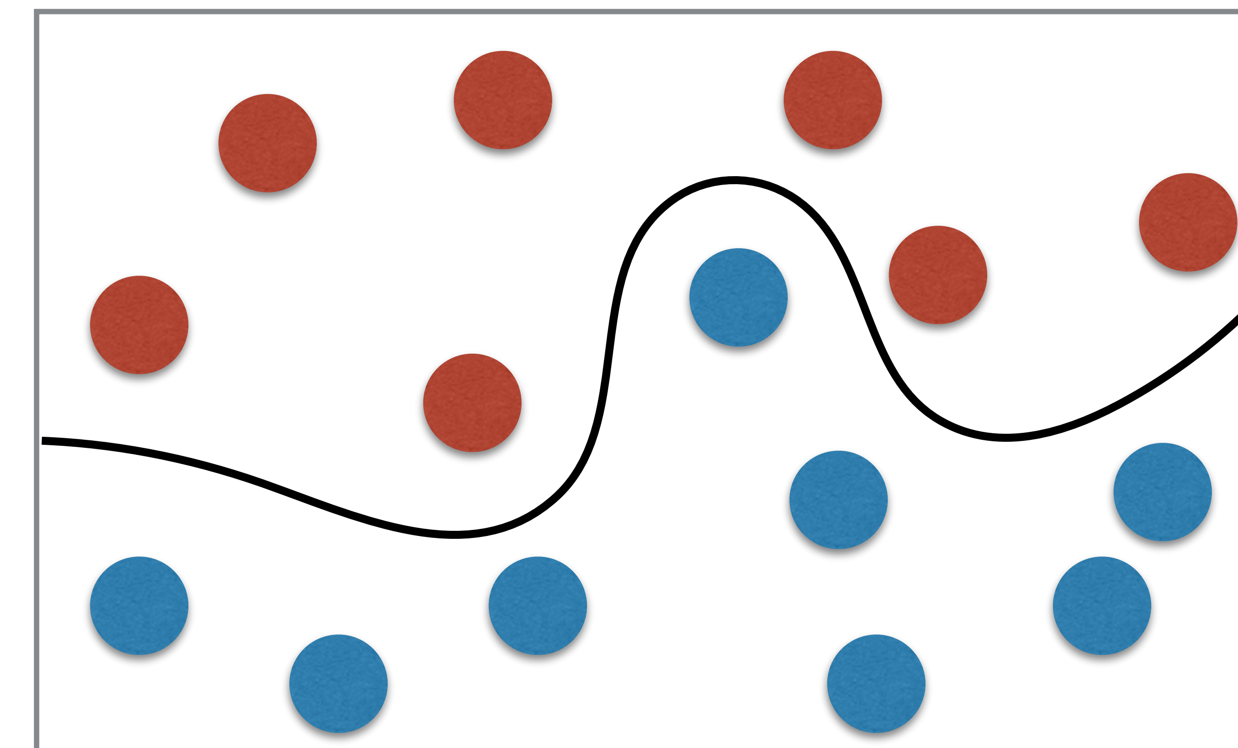
- In practice: build finite width, finite depth NN

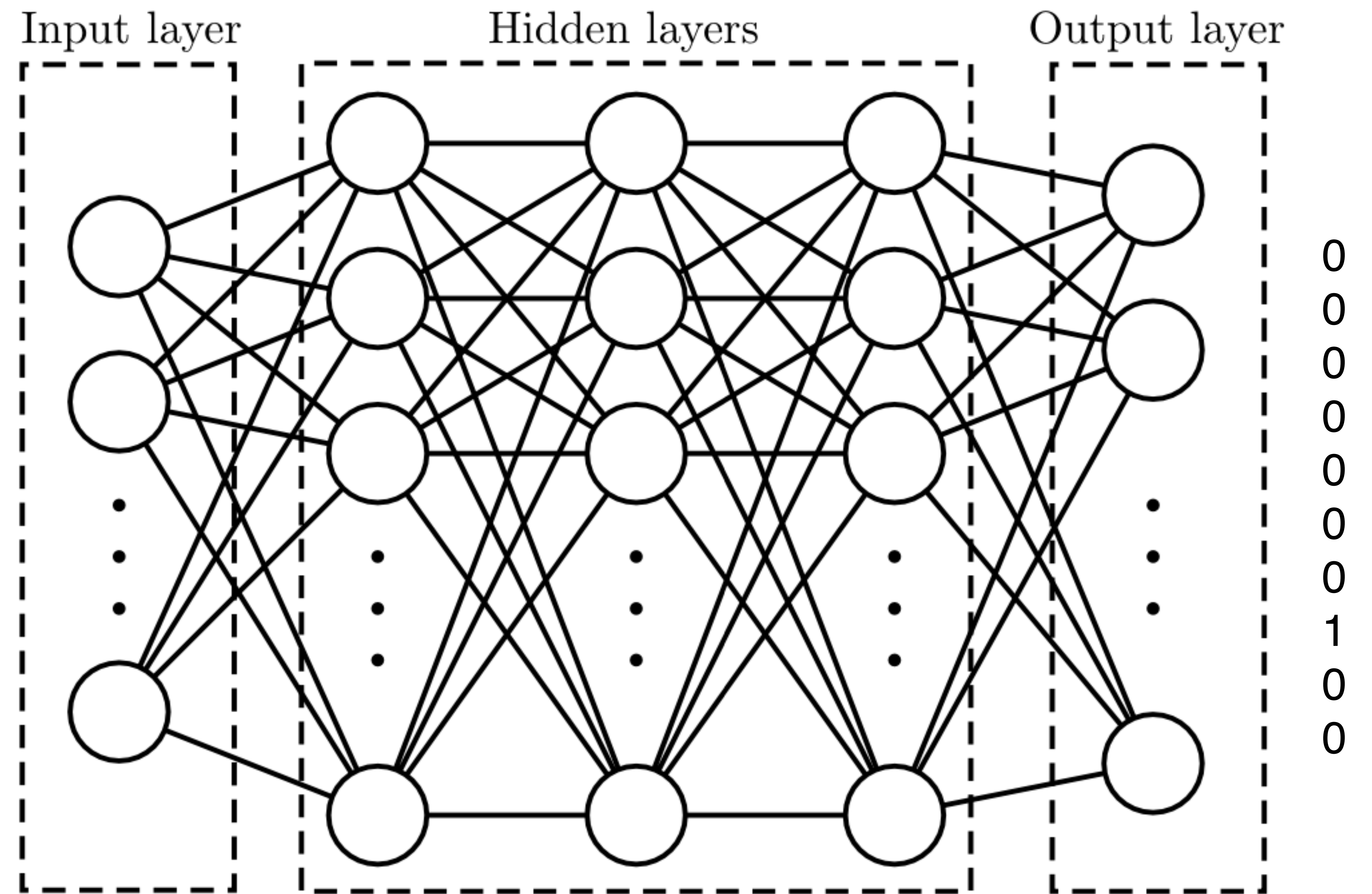


- Learning functions for regression



- Learning functions as decision boundaries for classification



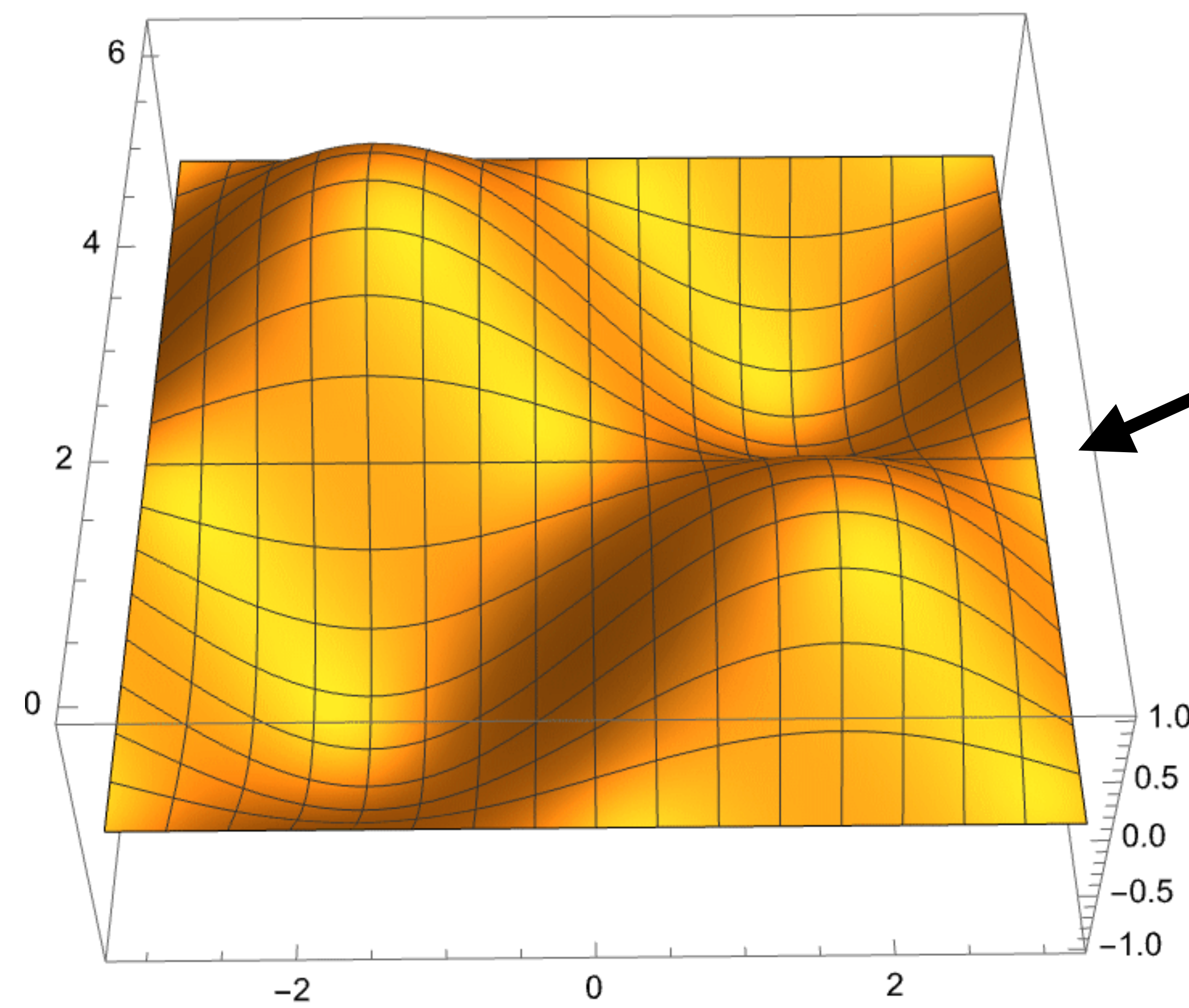


Training with backpropagation

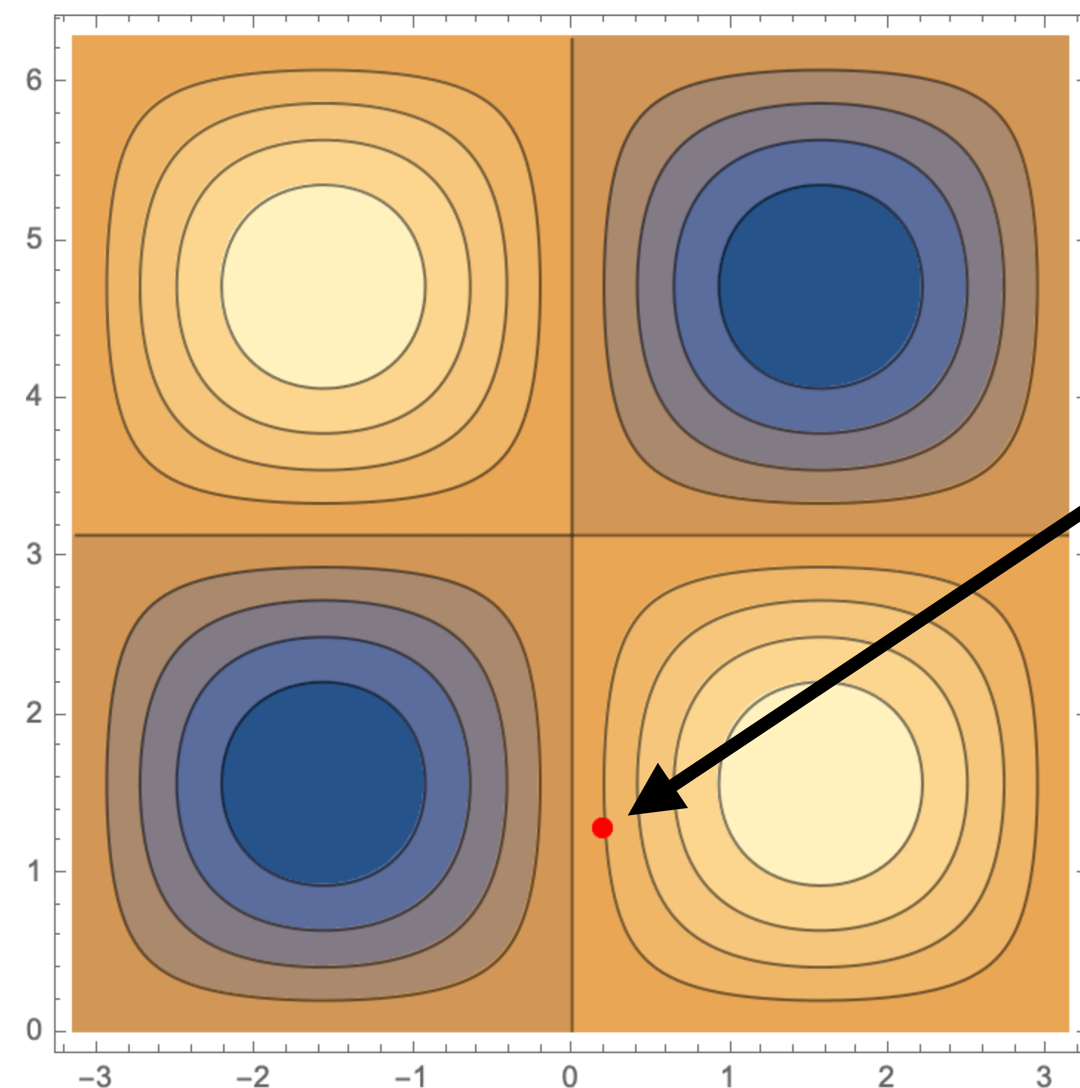
Neural Networks

- In supervised ML, you have a notion of what you want the NN output to be
- This is encoded in a **loss function** L, which gives the distance between the NN output $f_\theta(\vec{x})$ and the desired output $\vec{y}(\vec{x})$
- Given this distance or loss, you want to update the NN parameters θ s.t. $f_\theta(\vec{x}) \simeq \vec{y}(\vec{x})$
- Typically, the loss function is chosen such that $L(f_\theta(\vec{y}), \vec{y}(\vec{x})) \geq 0$ with equality iff $f_\theta(\vec{y}) = \vec{y}(\vec{x})$
- This means we need to find minima in (a million- to billion-dimensional) parameter space that generalizes well beyond training data
- Use “cheapest” (linear) optimizer: gradient descent

Visualization - Gradient descent

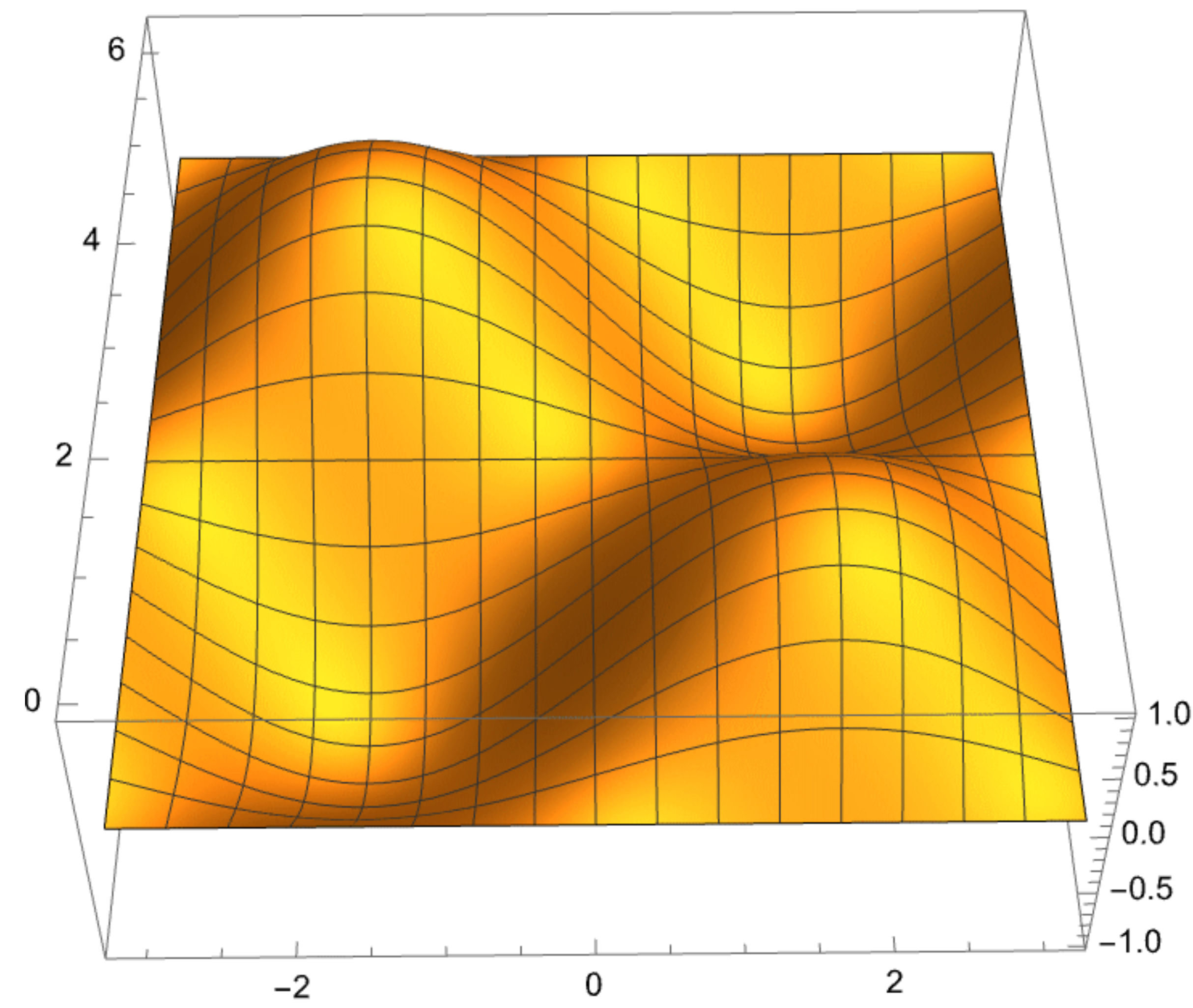


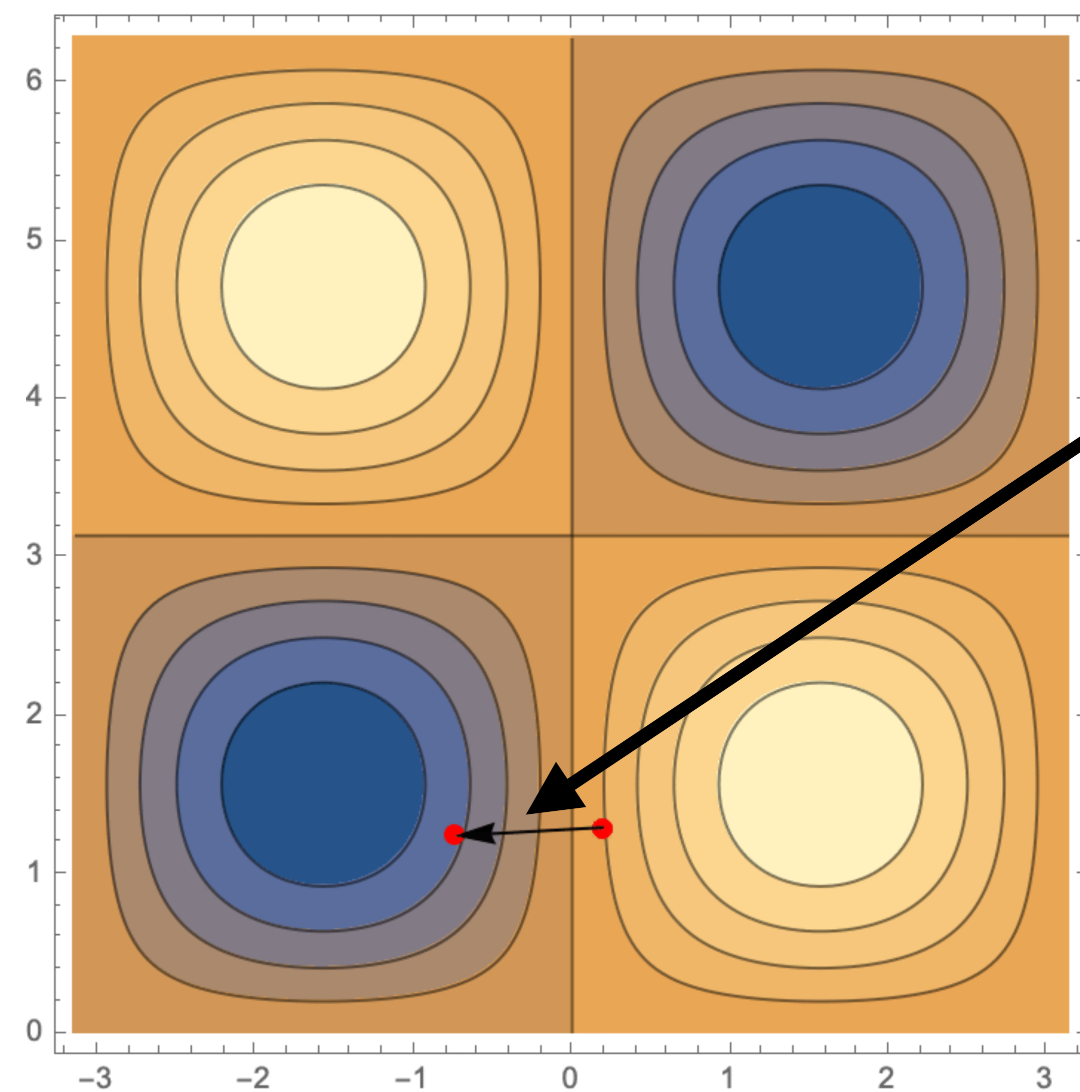
Loss landscape as a function of the NN parameters, and thus as a functional (function of a function) of the NN



Randomly initialized NN
= Random function
= Random point in loss landscape

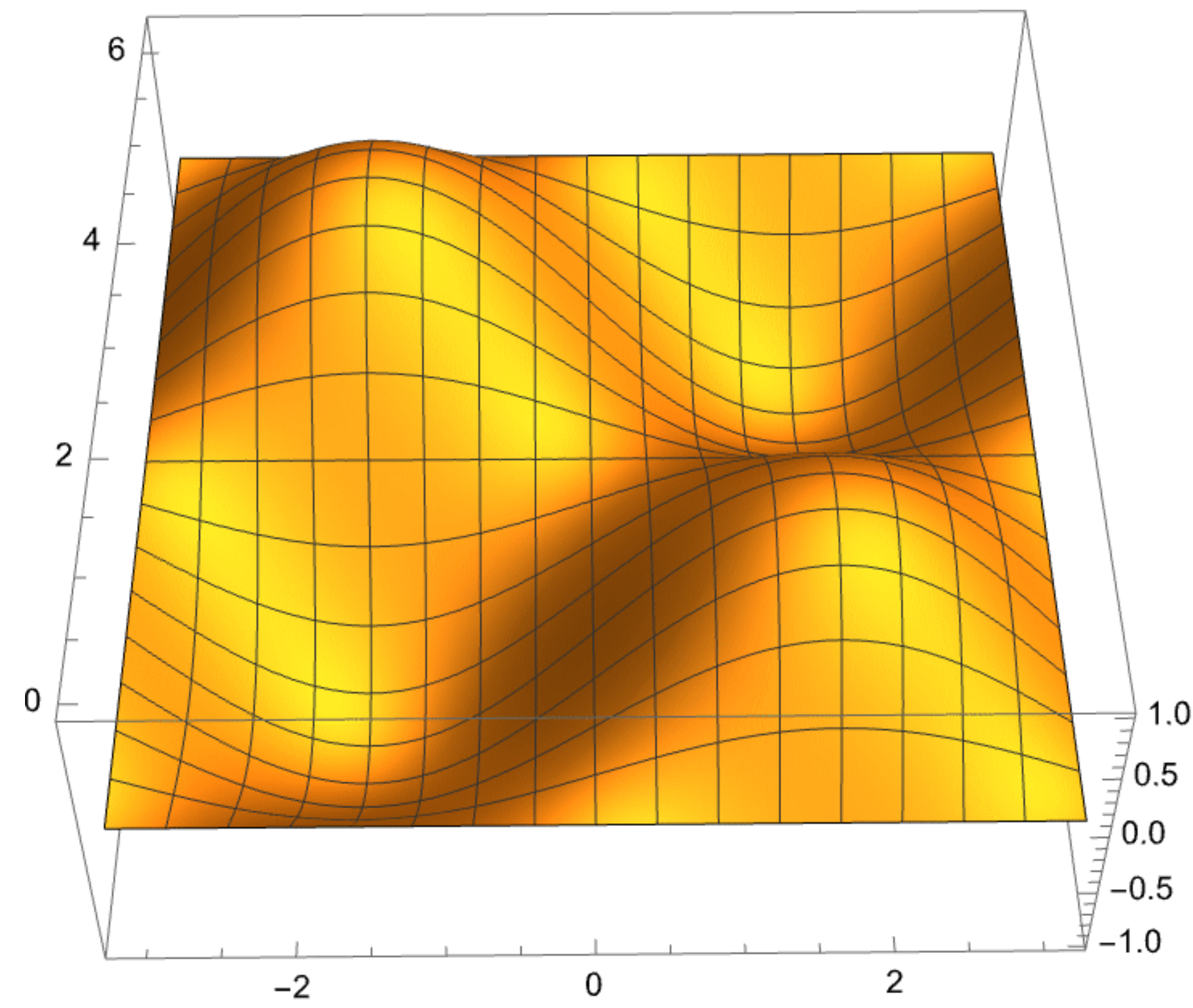
Visualization - Gradient descent

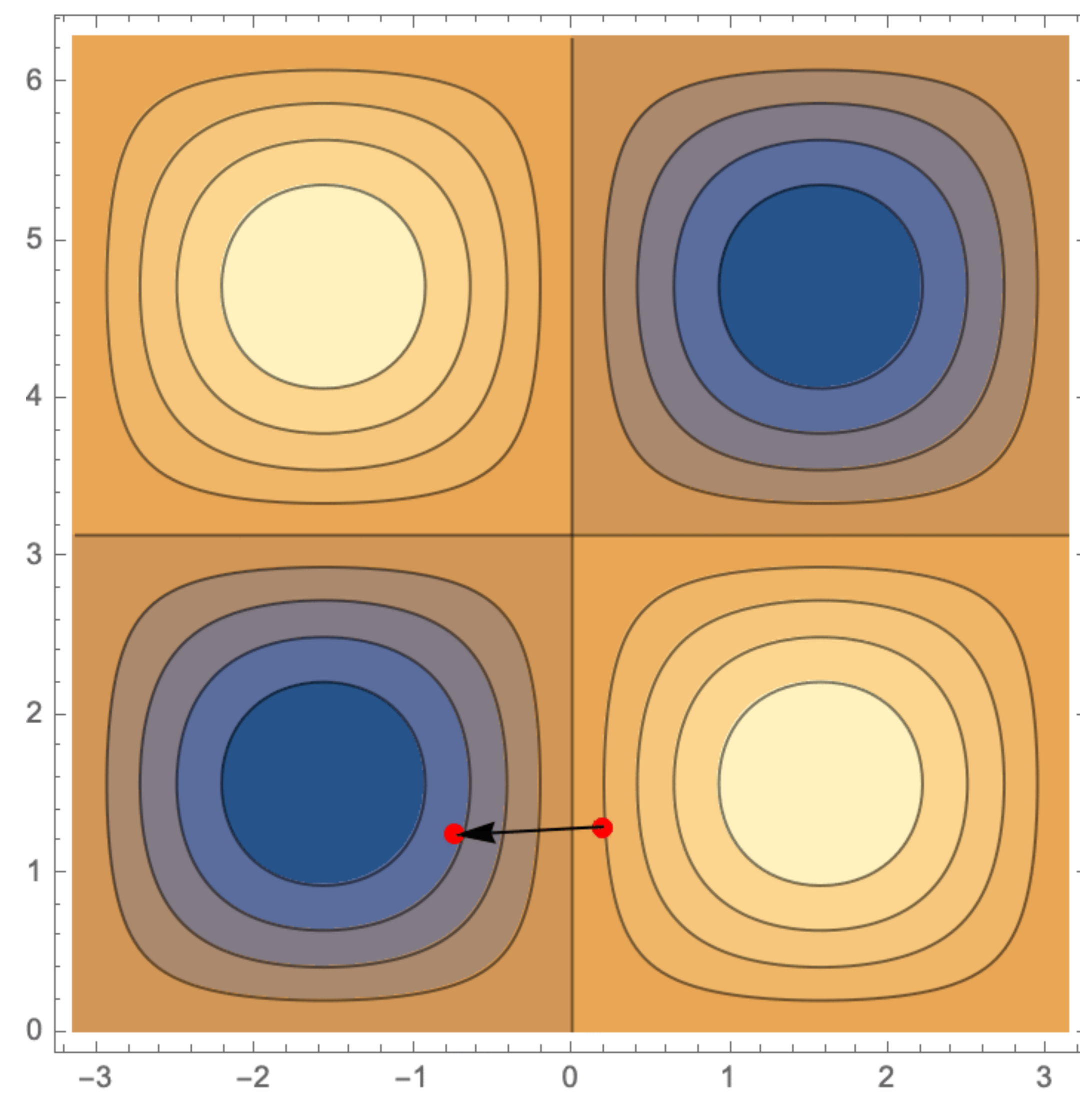




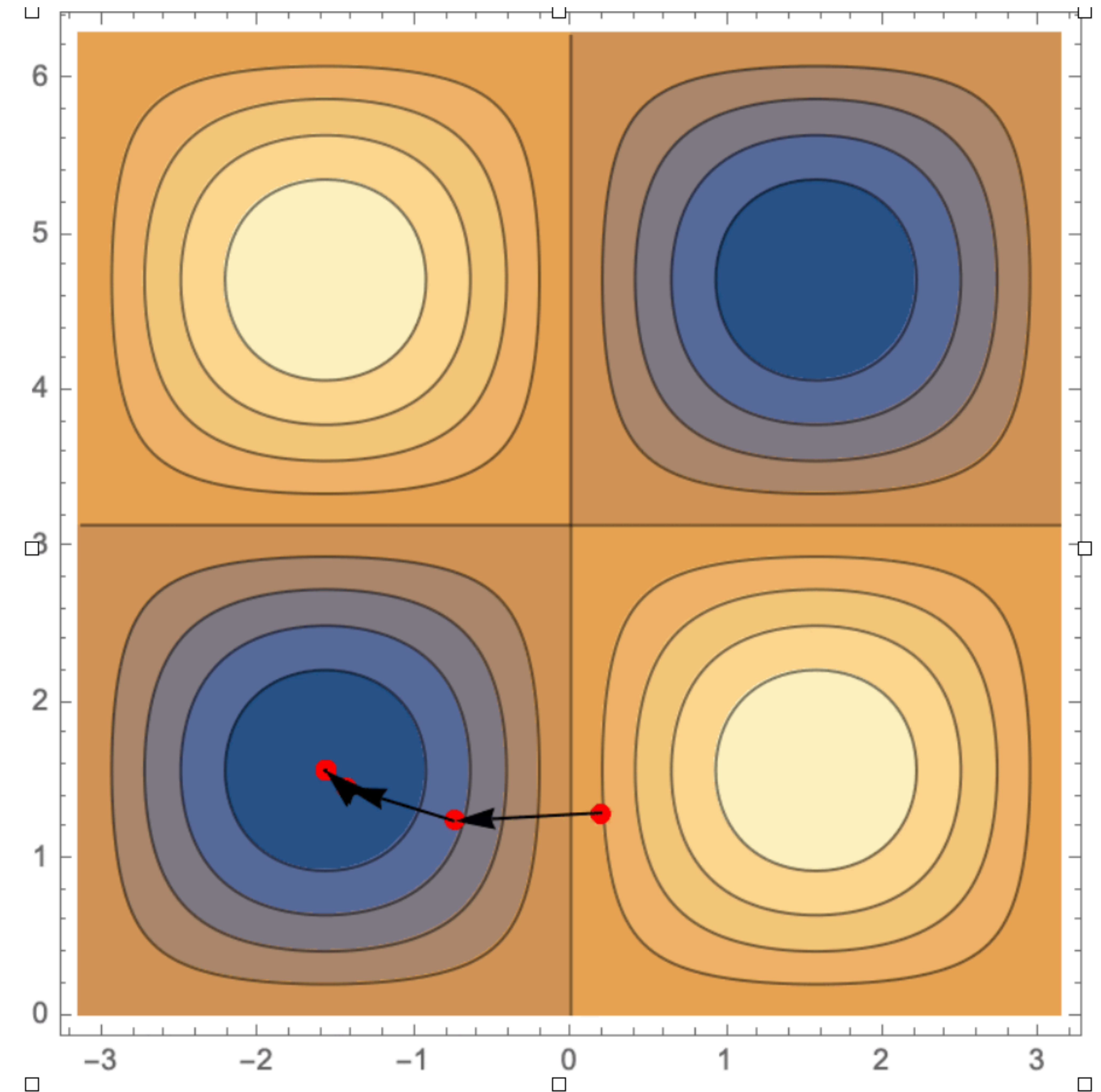
step size = learning rate

Visualization - Gradient descent

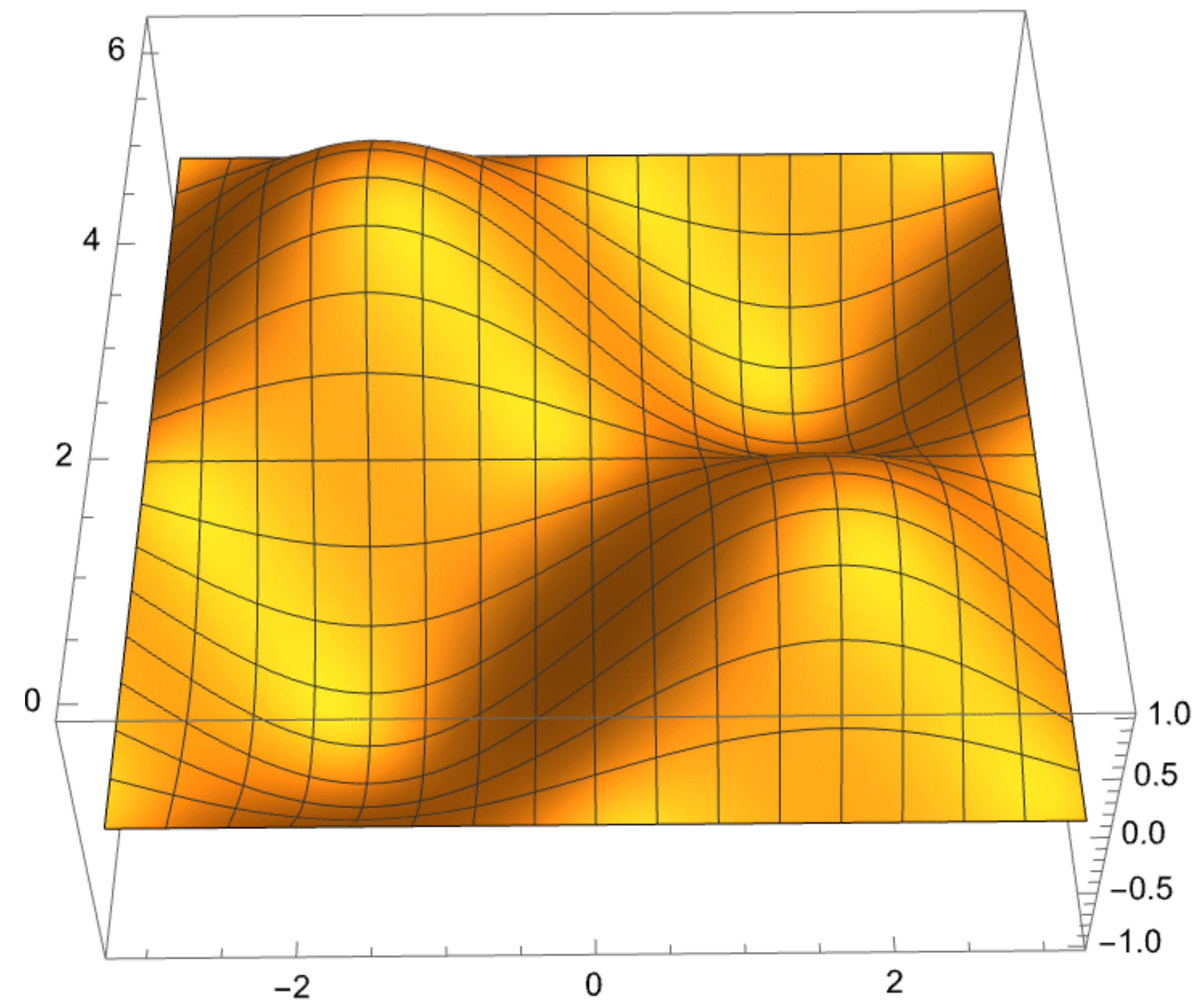


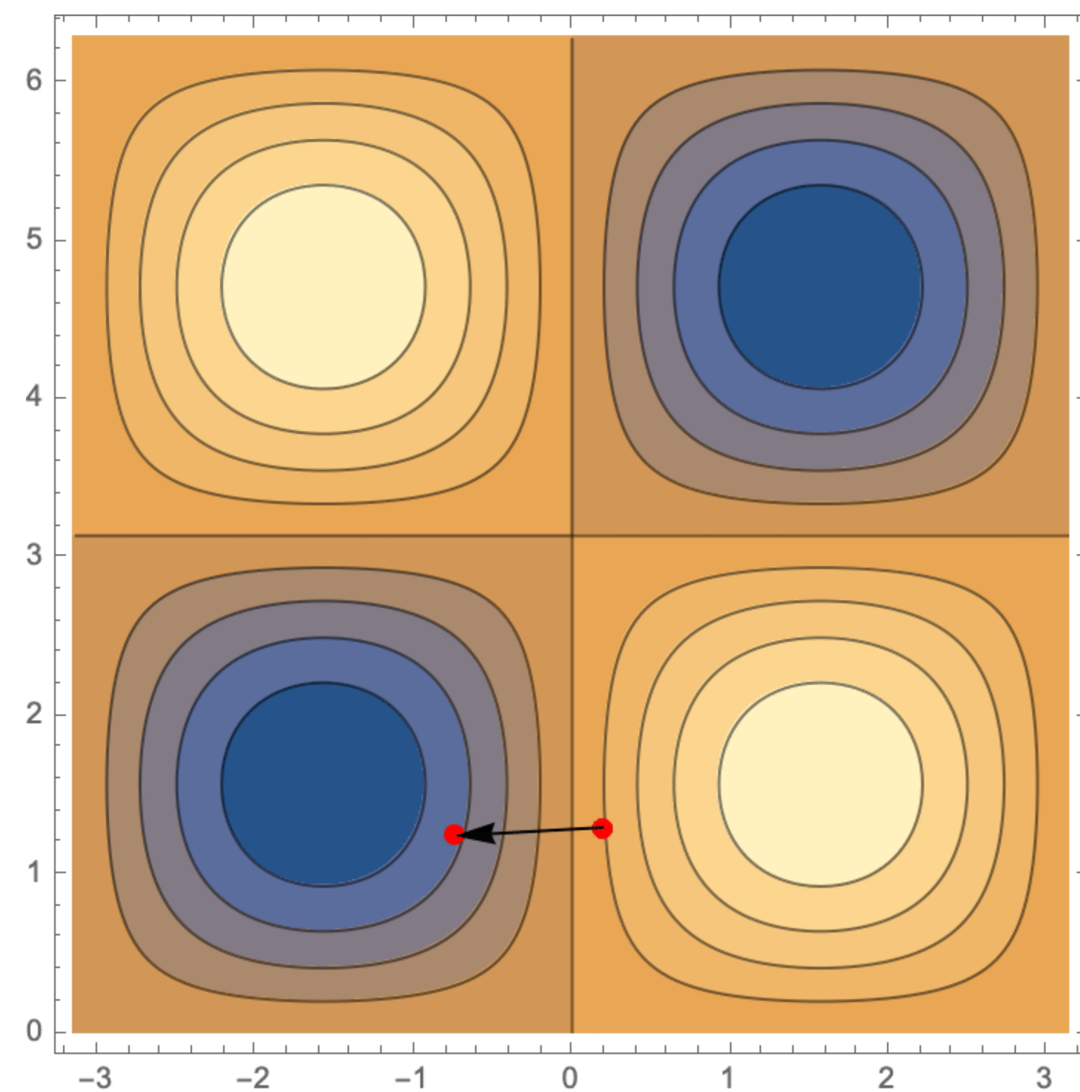


good learning rate

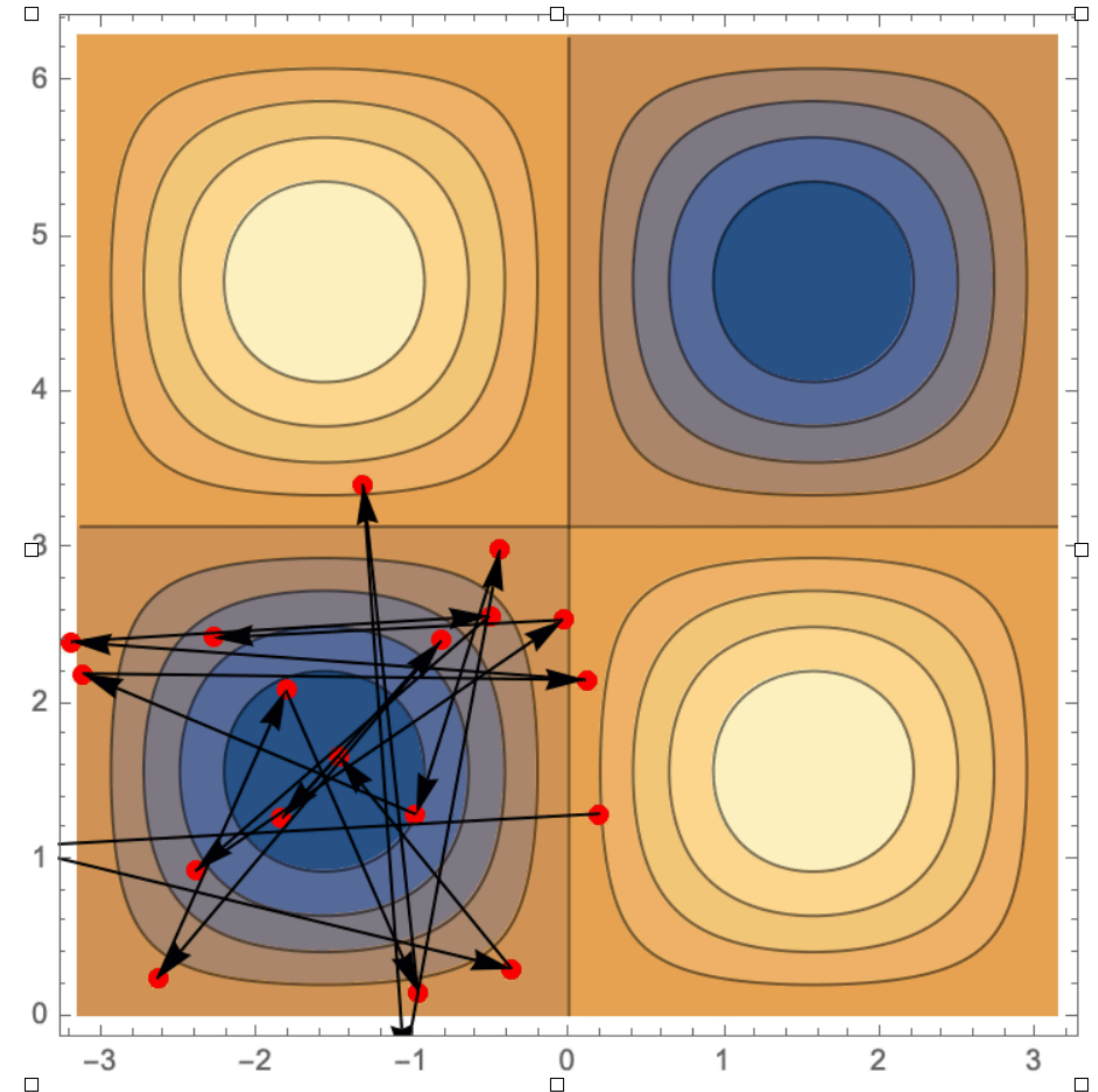


Visualization - Gradient descent

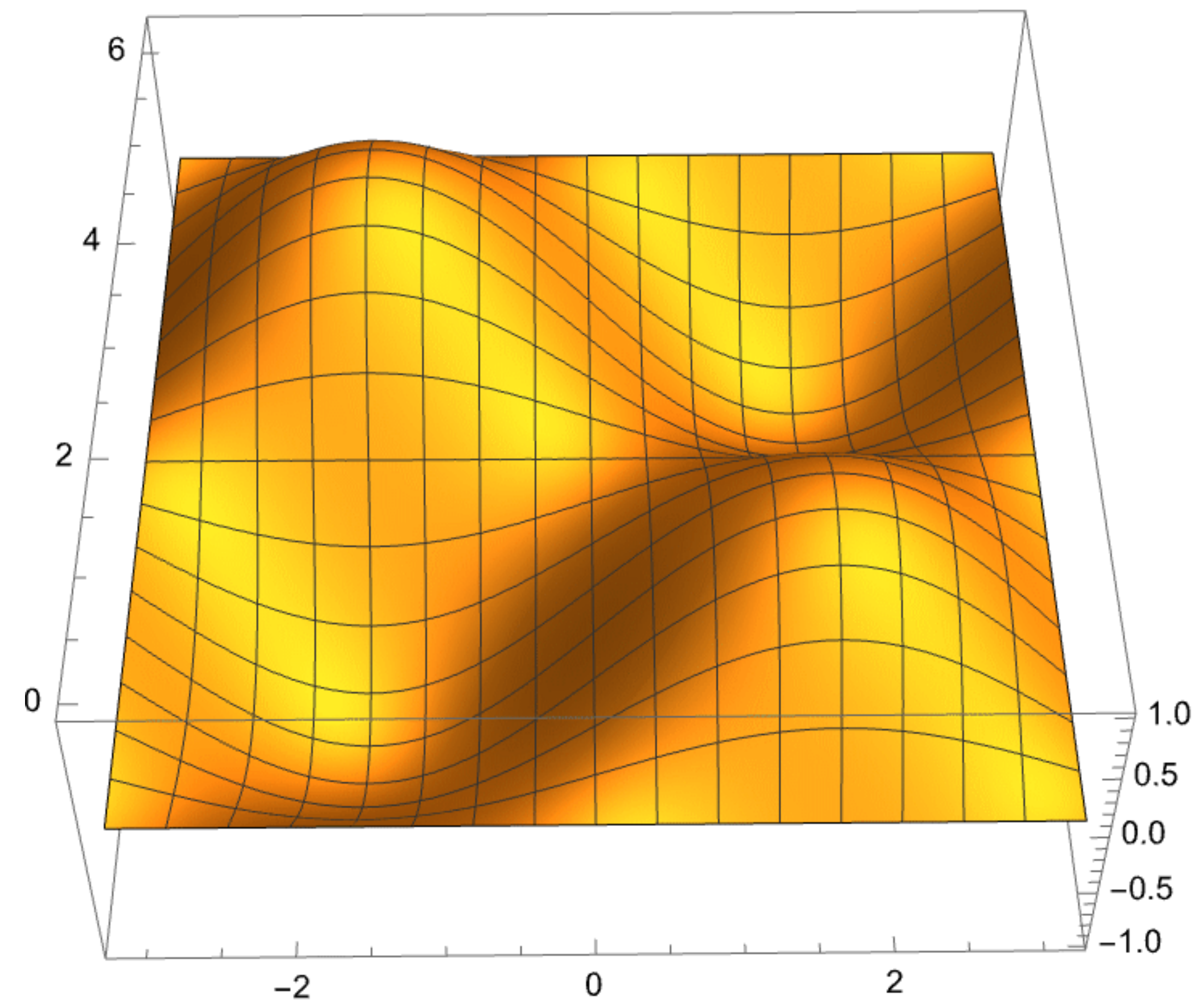


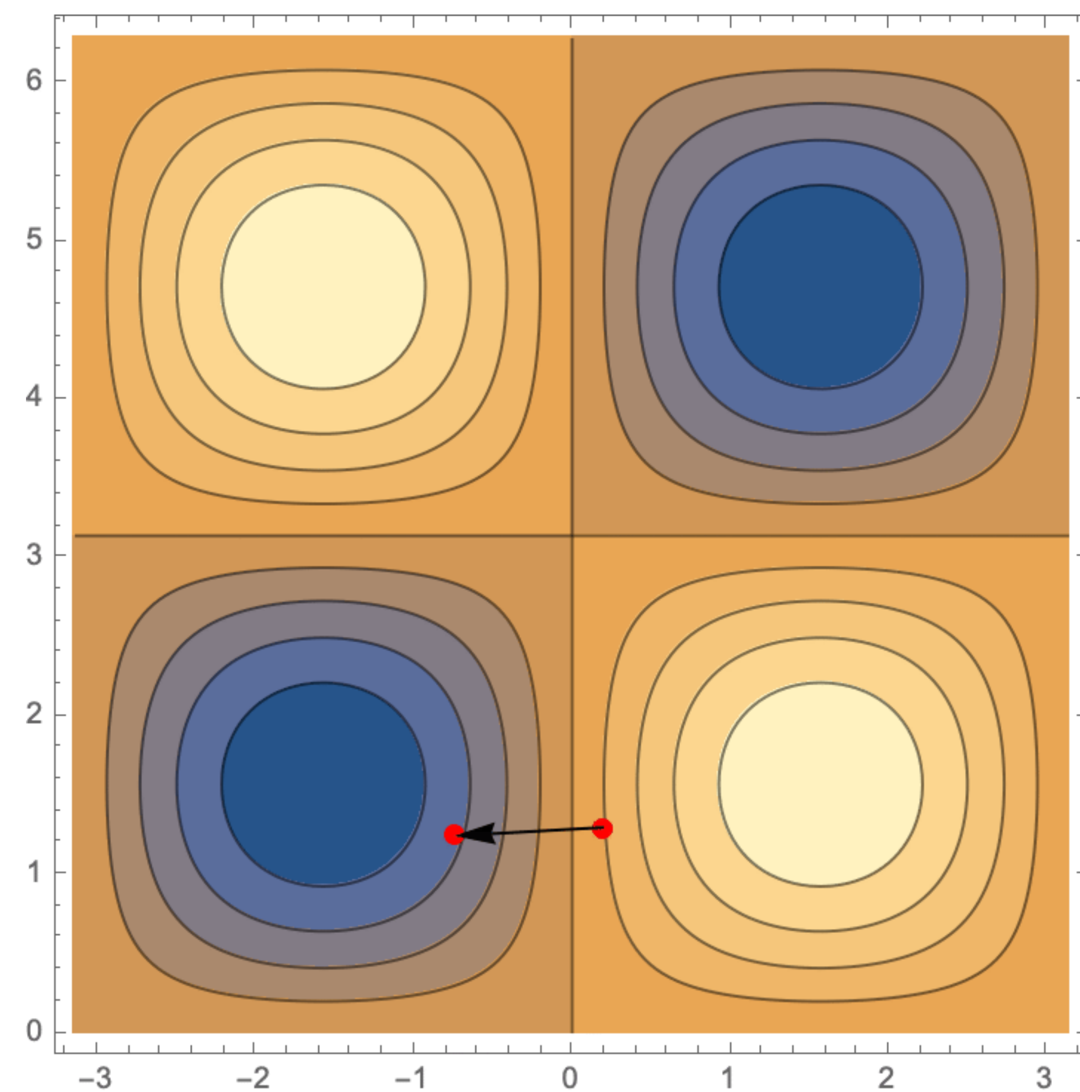


learning rate too big

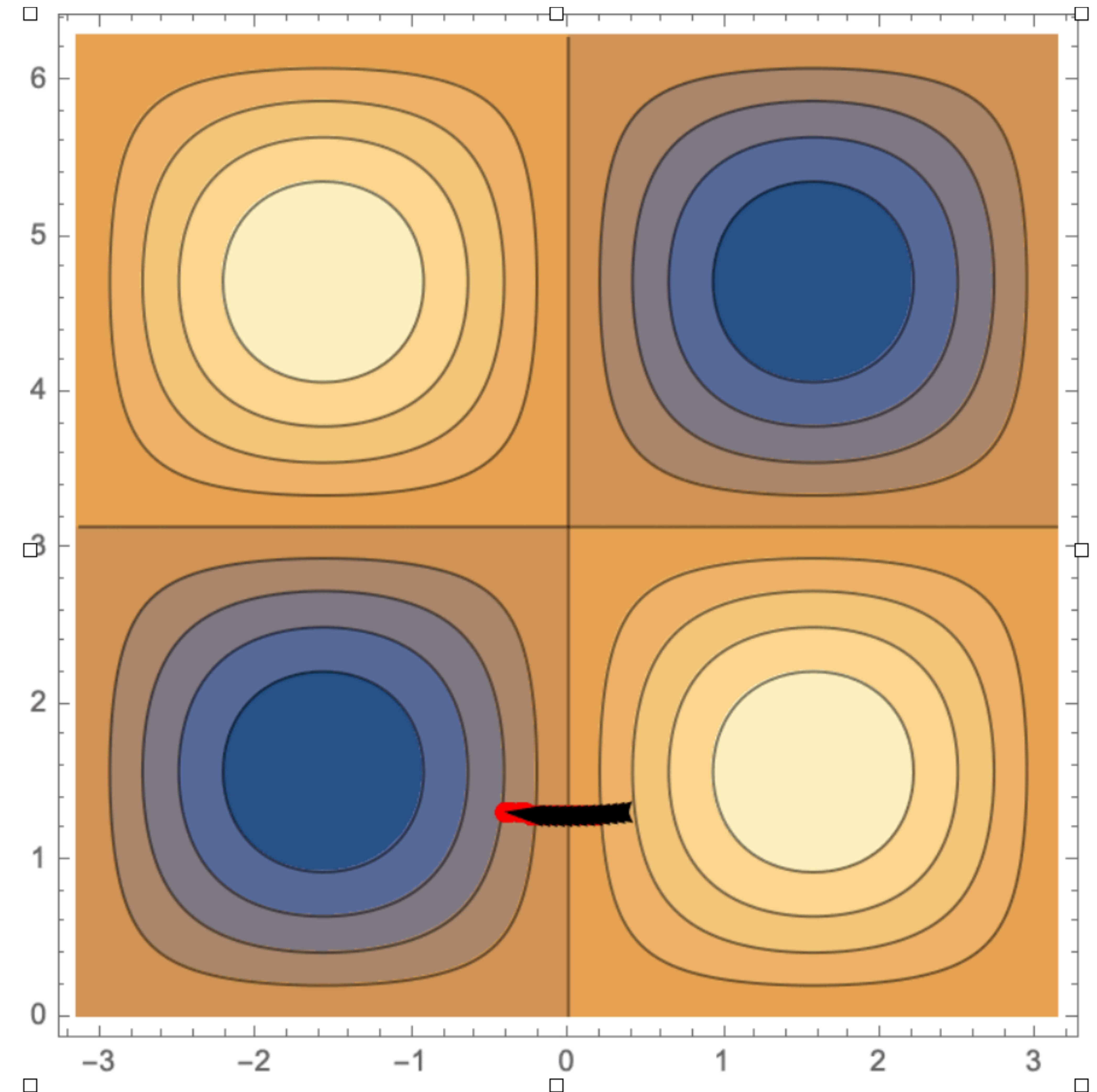


Visualization - Gradient descent





learning rate too small



Gradient Descent - Loss functions

- **For regression:**
  - MSE (Mean squared error): $$L_{\mathrm{MSE}} = \frac{1}{N}\sum_{i=1}^{N}\left[y(x_i) - f_\theta(x_i)\right]^2$$
  - MAE (Mean absolute error): $$L_{\mathrm{MAE}} = \frac{1}{N}\sum_{i=1}^{N}\left|y(x_i) - f_\theta(x_i)\right|$$
  - MAPE (Mean absolute percentage error): $$L_{\mathrm{MAPE}} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y(x_i) - f_\theta(x_i)}{y(x_i)}\right|$$
- **For classification:**
  - Binary cross-entropy: $$L_{\mathrm{BXE}} = \frac{1}{N}\sum_{i=1}^{N} y(x_i)\log(f\theta(x_i)) - (1 - y(x_i))\log(1 - f\theta(x_i))$$

Gradient Descent - Details

- NN = composition of maps $\Rightarrow$ use chain rule for derivatives
- At each layer of the NN, we want the gradient for each point in the dataset
- We introduce the following notation:

$$\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)}) \quad \text{with} \quad z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}$$

post-activation of layer $i+1$

layer index

Pre-activation of layer $i+1$

μ^{th} neuron

activation function

Jupyter Notebook Presentation

NN approximating a function

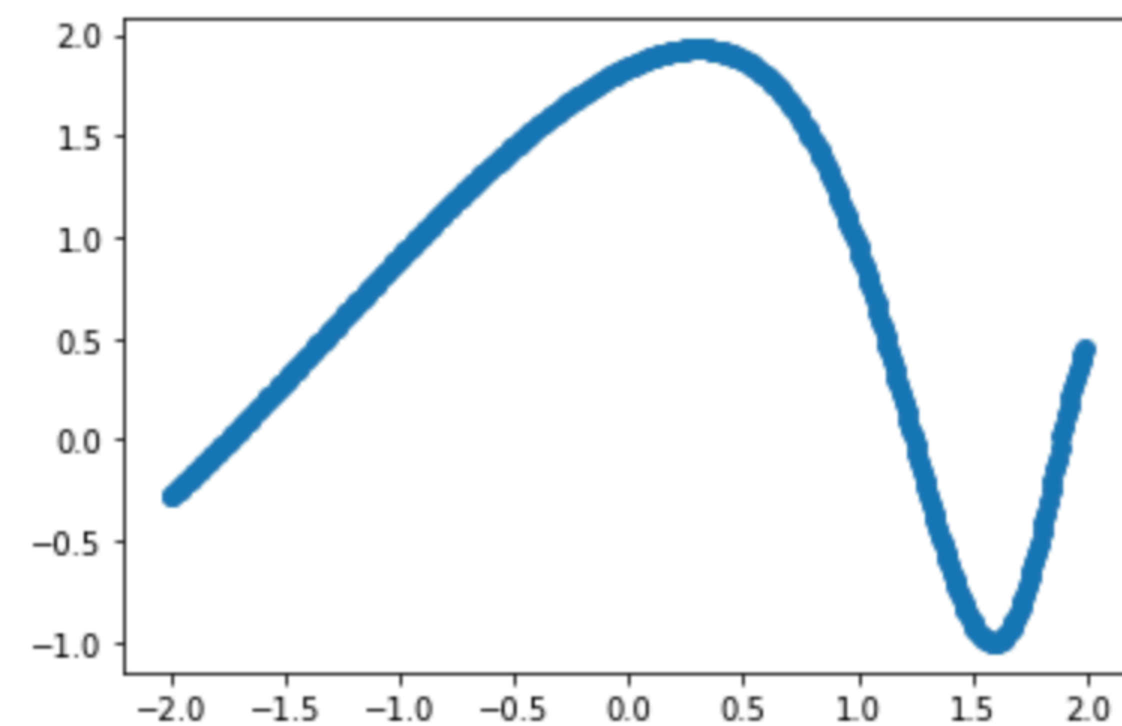
Create dataset:

$x = [-2, 2]\,, \quad y = sin(e^x) + cos(x)$

In [2]:
```
f.cast(np.arange(-2, 2, .01), dtype=tf.float32)
f.sin(tf.exp(x)) + tf.cos(x)
```

Plot the dataset

In [3]:
```
igure(1)
catter(x, y);
```



Setup a NN to learn this function

In [4]:
```
tf.keras.Sequential()
ut layer
d(tfk.Input(shape=(1)))
```

Gradient Descent - Details

- NN = composition of maps ⇒ use chain rule for derivatives
- At each layer of the NN, we want the gradient for each point in the dataset
- We introduce the following notation: $\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)})$ with $z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}$
- Then:
  - ✦ $$\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial \theta^{(n)}}$$

$$\frac{\partial z_\mu^{(n)}}{\partial \theta_a^{(n)}} = \begin{cases} \dfrac{\partial z_\mu^{(n)}}{\partial w_{\mu\nu}^{(n)}} = \ell_\nu^{(n-1)} & \text{for } \theta_a^{(n)} = w_{\mu\nu}^{(n)} \\[2ex] \dfrac{\partial z_\mu^{(n)}}{\partial b_\nu^{(n)}} = \delta_{\mu\nu} & \text{for } \theta_a^{(n)} = b_\nu^{(n)} \end{cases}$$

known from forward pass

$$\frac{\partial L_{\mathrm{MSE}}}{\partial z^{(n)}} = -\frac{2}{N}\left[y - a^{(n)}(z^{(n)})\right] a'^{(n)}(z^{(n)})$$

Gradient Descent - Details

- NN = composition of maps ⇒ use chain rule for derivatives
- At each layer of the NN, we want the gradient for each point in the dataset
- We introduce the following notation: $\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)})$ with $z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}$
- Then:
  - ✦ $$\frac{\partial L}{\partial \theta^{(n)}} = \boxed{\frac{\partial L}{\partial z^{(n)}}} \frac{\partial z^{(n)}}{\partial \theta^{(n)}}$$
  - ✦ $$\frac{\partial L}{\partial \theta^{(n-1)}} = \boxed{\frac{\partial L}{\partial z^{(n)}}} \frac{\partial z^{(n)}}{\partial z^{(n-1)}} \frac{\partial z^{(n-1)}}{\partial \theta^{(n-1)}}$$

$$\frac{\partial z^{(n)}}{\partial z^{(n-1)}}: \quad \left(\sum_{\mu=1}^{N_{n-1}} w_{\mu\nu}^{(n-1)} \frac{\partial L}{\partial z_\mu^{(n)}} \right) a'^{(n-1)}(z_\nu^{(n-1)})$$

$$\frac{\partial z_\mu^{(n-1)}}{\partial \theta_a^{(n-1)}} = \begin{cases} \dfrac{\partial z_\mu^{(n-1)}}{\partial w_{\mu\nu}^{(n-1)}} = \ell_\nu^{(n-2)} & \text{for} \quad \theta_a = w_{\mu\nu}^{(n)} \\[2ex] \dfrac{\partial z_\mu^{(n-1)}}{\partial b_\nu^{(n-1)}} = \delta_{\mu\nu} & \text{for} \quad \theta_a = b_\mu^{(n)} \end{cases}$$

known from forward pass

Gradient Descent - Details

- NN = composition of maps ⇒ use chain rule for derivatives
- At each layer of the NN, we want the gradient for each point in the dataset
- We introduce the following notation: $\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)})$ with $z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}$
- Then:
  - $$\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial \theta^{(n)}}$$
  - $$\frac{\partial L}{\partial \theta^{(n-1)}} = \boxed{\frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial z^{(n-1)}}} \frac{\partial z^{(n-1)}}{\partial \theta^{(n-1)}}$$
  - $$\frac{\partial L}{\partial \theta^{(n-2)}} = \boxed{\frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial z^{(n-1)}}} \frac{\partial z^{(n-1)}}{\partial z^{(n-2)}} \frac{\partial z^{(n-2)}}{\partial \theta^{(n-2)}}$$

Gradient Descent - Details

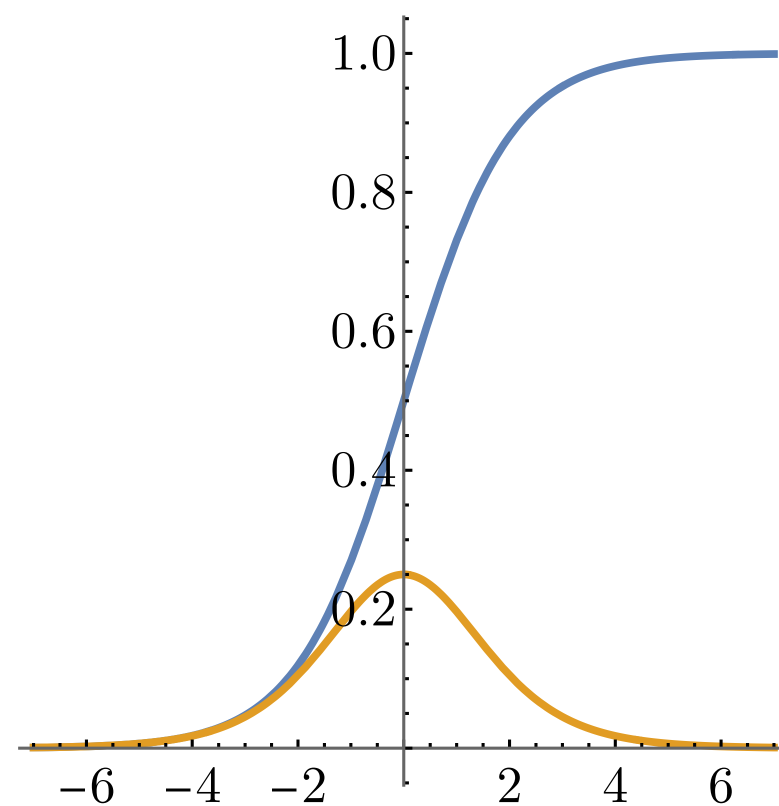
- NN = composition of maps ⇒ use chain rule for derivatives
- At each layer of the NN, we want the gradient for each point in the dataset
- We introduce the following notation: $\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)})$ with $z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}$
- Then:
  - $$\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial \theta^{(n)}}$$
  - $$\frac{\partial L}{\partial \theta^{(n-1)}} = \boxed{\frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial z^{(n-1)}}} \frac{\partial z^{(n-1)}}{\partial \theta^{(n-1)}}$$
  - $$\frac{\partial L}{\partial \theta^{(n-2)}} = \boxed{\frac{\partial L}{\partial z^{(n)}} \frac{\partial z^{(n)}}{\partial z^{(n-1)}}} \frac{\partial z^{(n-1)}}{\partial z^{(n-2)}} \frac{\partial z^{(n-2)}}{\partial \theta^{(n-2)}}$$

GD parameter update

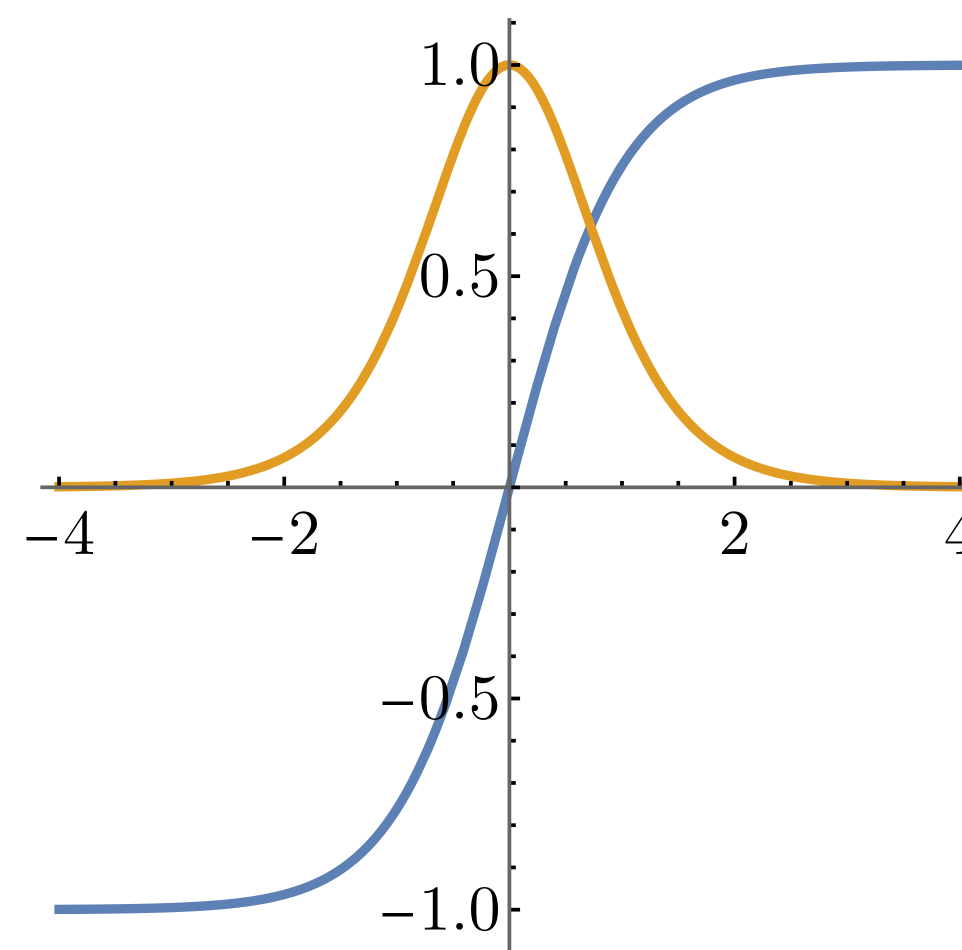
$$\theta^{(i)} \to \theta^{(i)} - \alpha \frac{\partial L}{\partial \theta^{(i)}}$$

Gradient Descent - Consequences

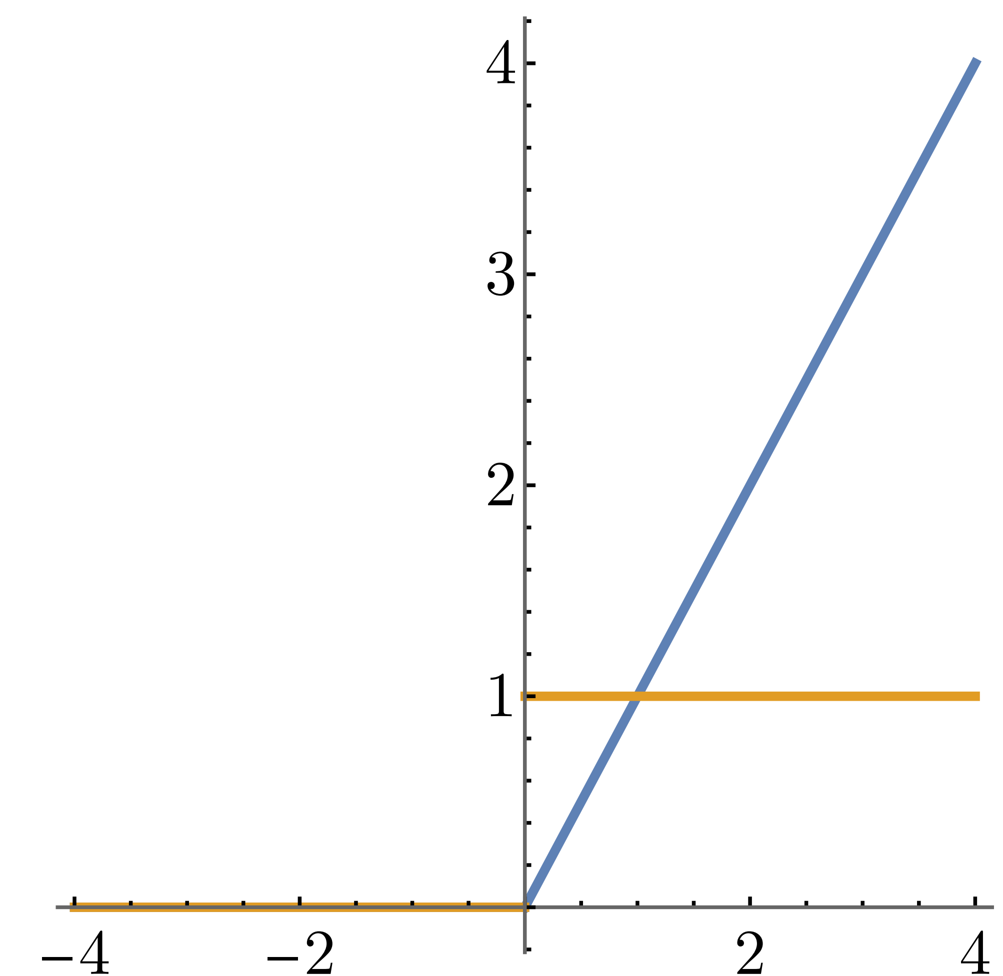
- Most of what we need in backward pass has been computed in forward pass or can be trivially computed
- …but we need $a'^{(i)}(z^{(i-1)})$
- Common activation functions have $a'^{(i)} = f(a^{(i)})$, so reuse from fwd pass



$$a(x) = \sigma(x) = 1/(1 + e^{-x})$$
$$a'(x) = a(x)[1 - a(x)]$$



$$a(x) = \tanh(x)$$
$$a'(x) = 1 - [a(x)]^2$$



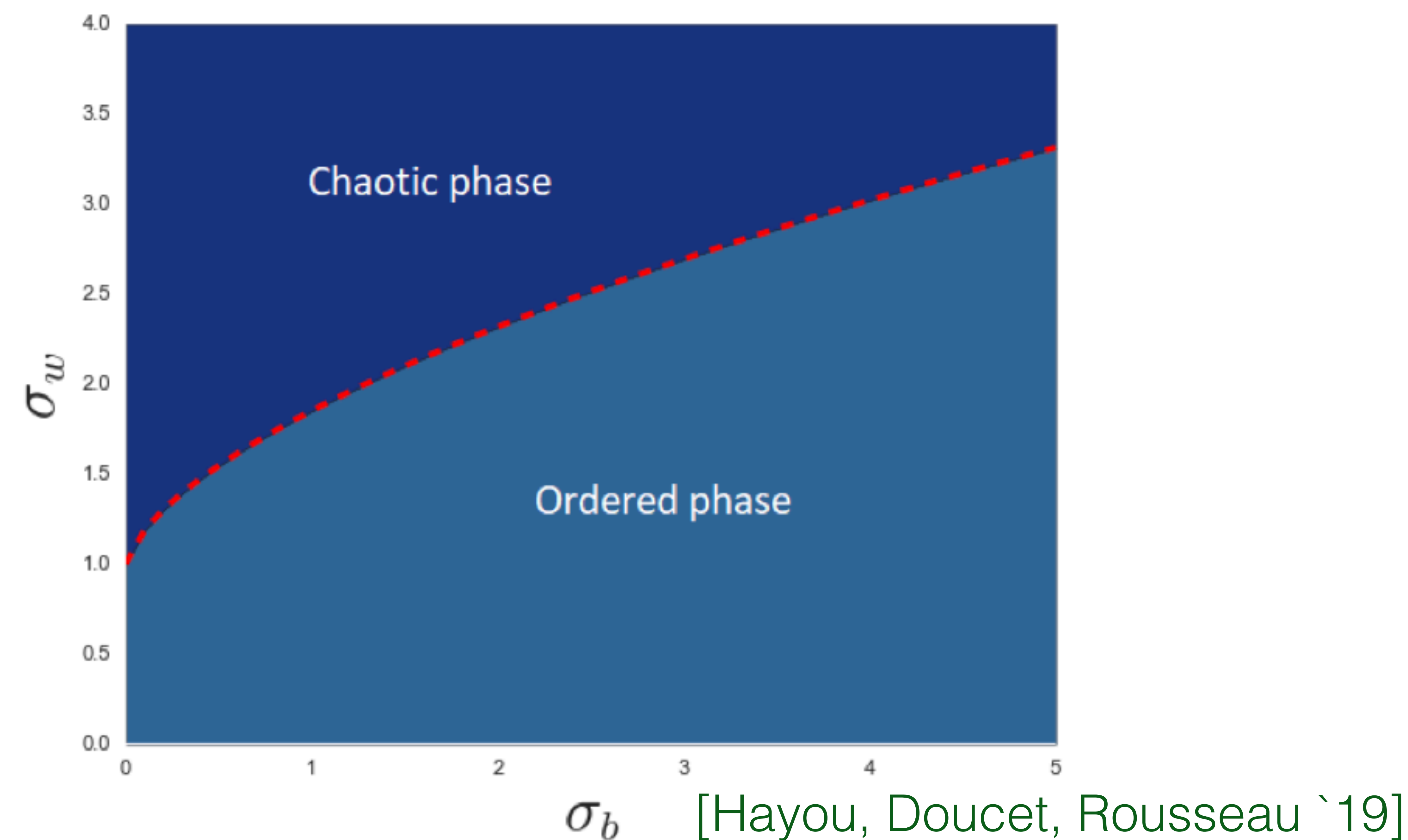
$$a(x) = \mathrm{ReLU}(x) = \max(0, x)$$
$$a'(x) = \theta(x)$$

Gradient Descent - Consequences

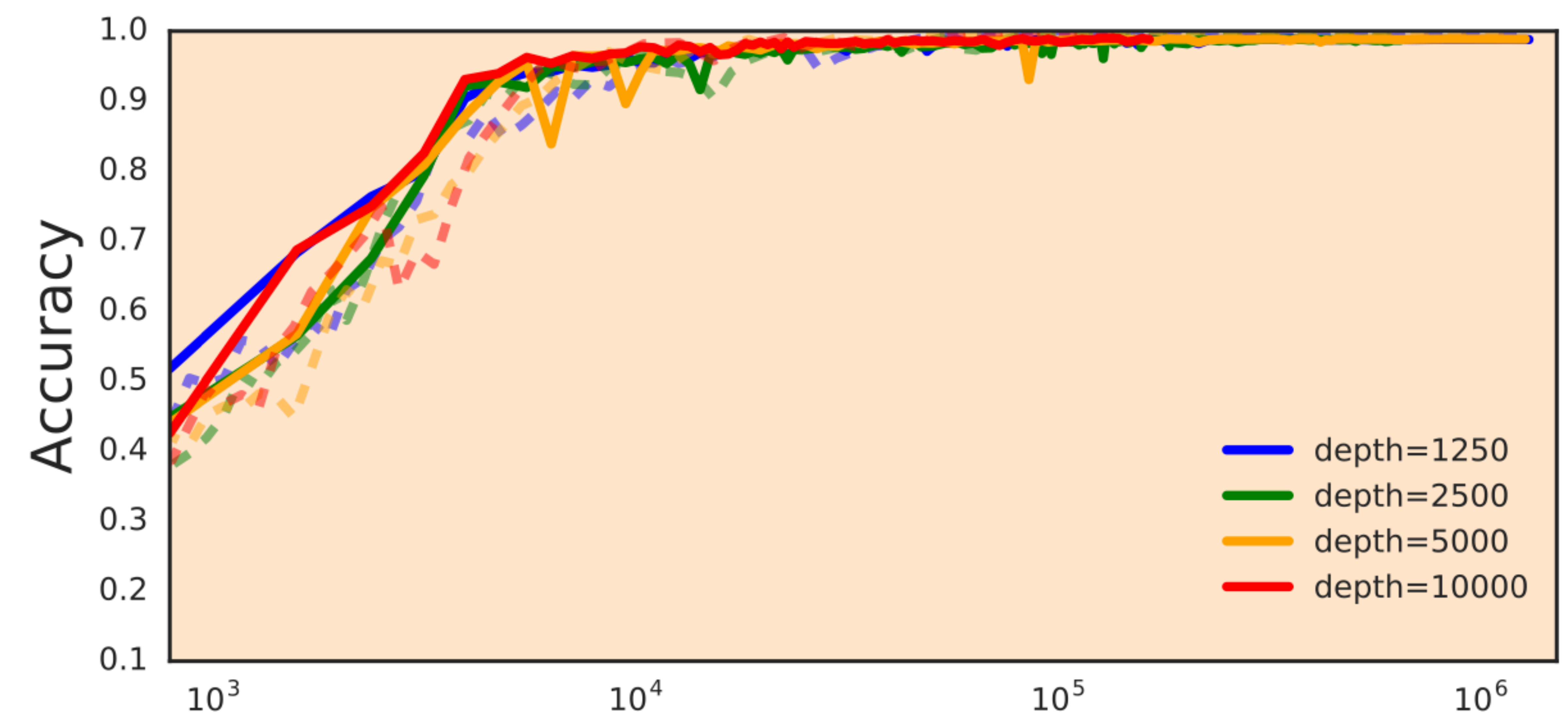
- NN are typically used in big data applications
- Can parallelize over input-output-pairs
  - O(10) cores on CPUs
  - O(10k) cores on GPUs
- But not all input-output-pairs can fit into memory simultaneously
- Batch data into mini-batches and perform updates after each mini-batch (typically batch size is O(10)-O(100))
- This leads to stochastic GD (since we are not optimizing on entire loss landscape but only on a subset computed from each batch)
- This can in fact help overcome local minima and saddles, but hinders parallelization

Gradient Descent - Consequences

- Gradient at layer i proportional to gradient at layer $i+1$
- Get vanishing / exploding gradients at earlier layers
- Counter: Edge of chaos initialization, gradient/weight clipping, batch norm, skip connections (UNET, ResNET), …



[Hayou, Doucet, Rousseau `19]



[Xiao et.al. `18]

Gradient Descent - Consequences

- Gradient at layer i proportional to gradient at layer $i+1$
- Dying ReLU problem: $\mathrm{ReLU}(x) = \mathrm{ReLU}'(x) = 0$ for $x \leq 0$
- ReLU can die and not activate other layers; updates are zero as well, so this behavior won't change
- Combat with leaky ReLU: $\text{leaky ReLU} = \begin{cases} x & \text{if } x > 0 \\ .01\, x & \text{if } x \leq 0 \end{cases}$
- To a lesser extent the same can happen to sigmoid:

$$\sigma(x) \approx \sigma'(x) \approx 0 \quad \text{for} \quad x \ll 1$$

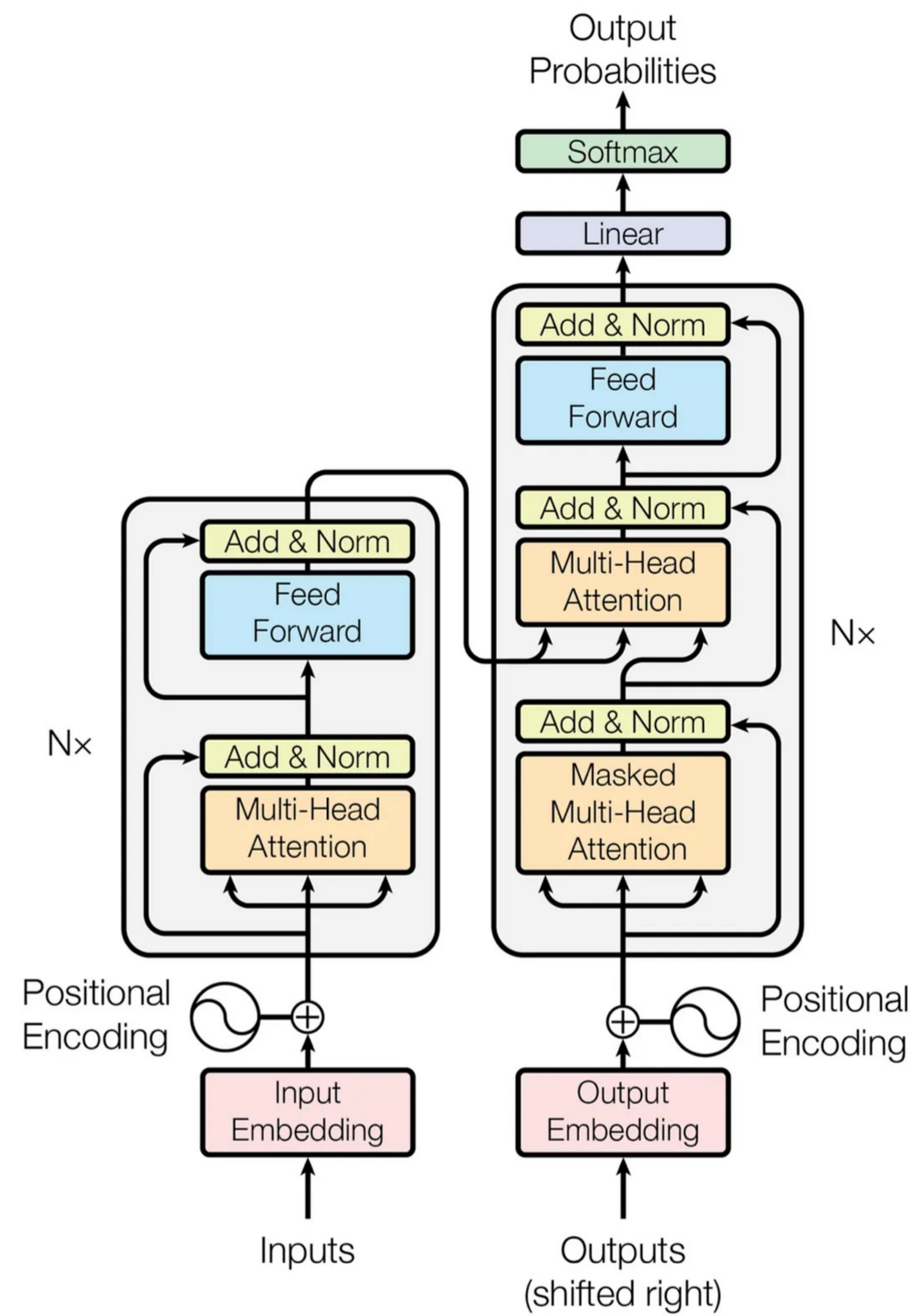
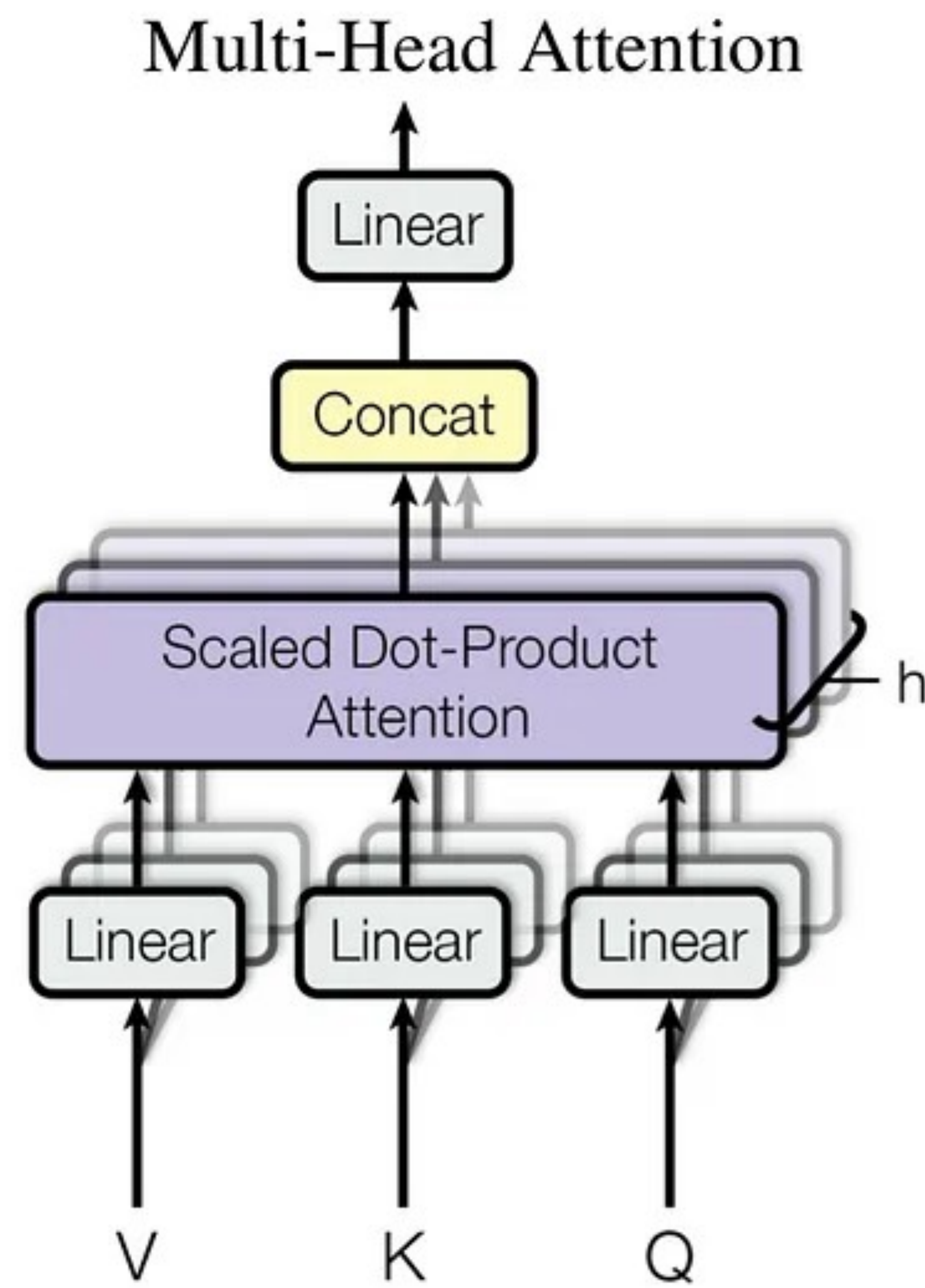
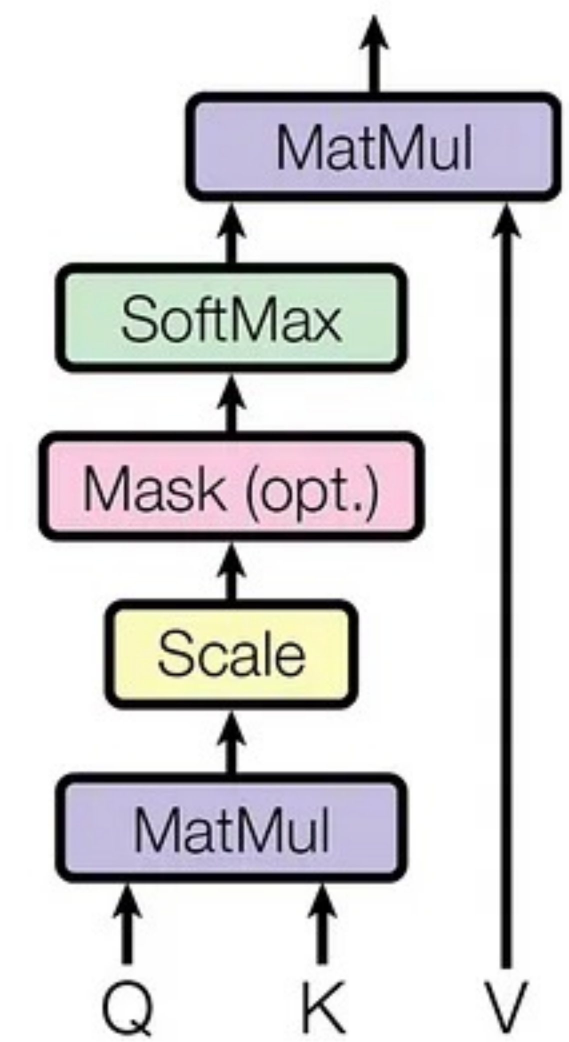


Figure 1: The Transformer - model architecture.





[Vaswani et.al. `17]

The transformer architecture

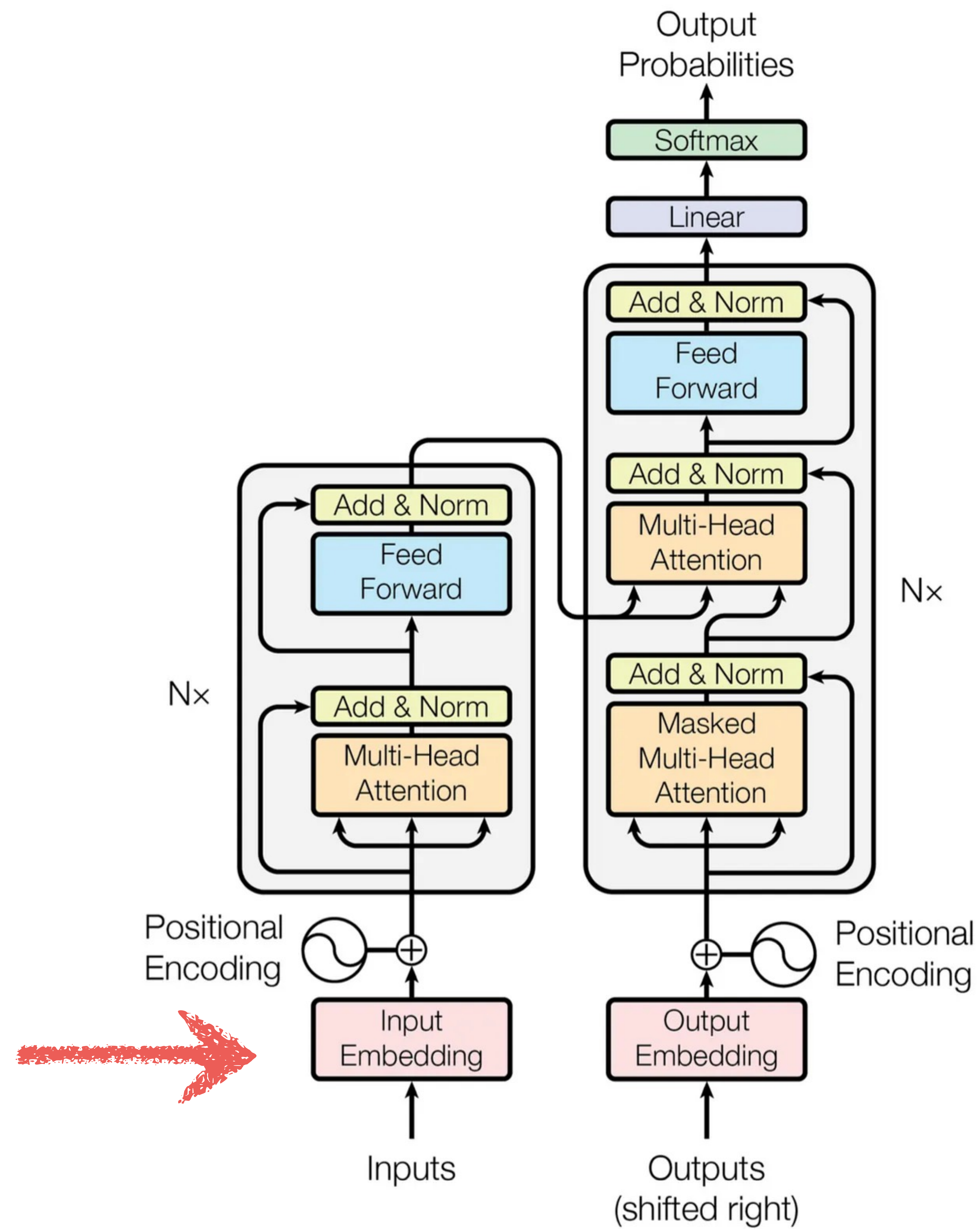


Figure 1: The Transformer - model architecture.

NLP and input embedding

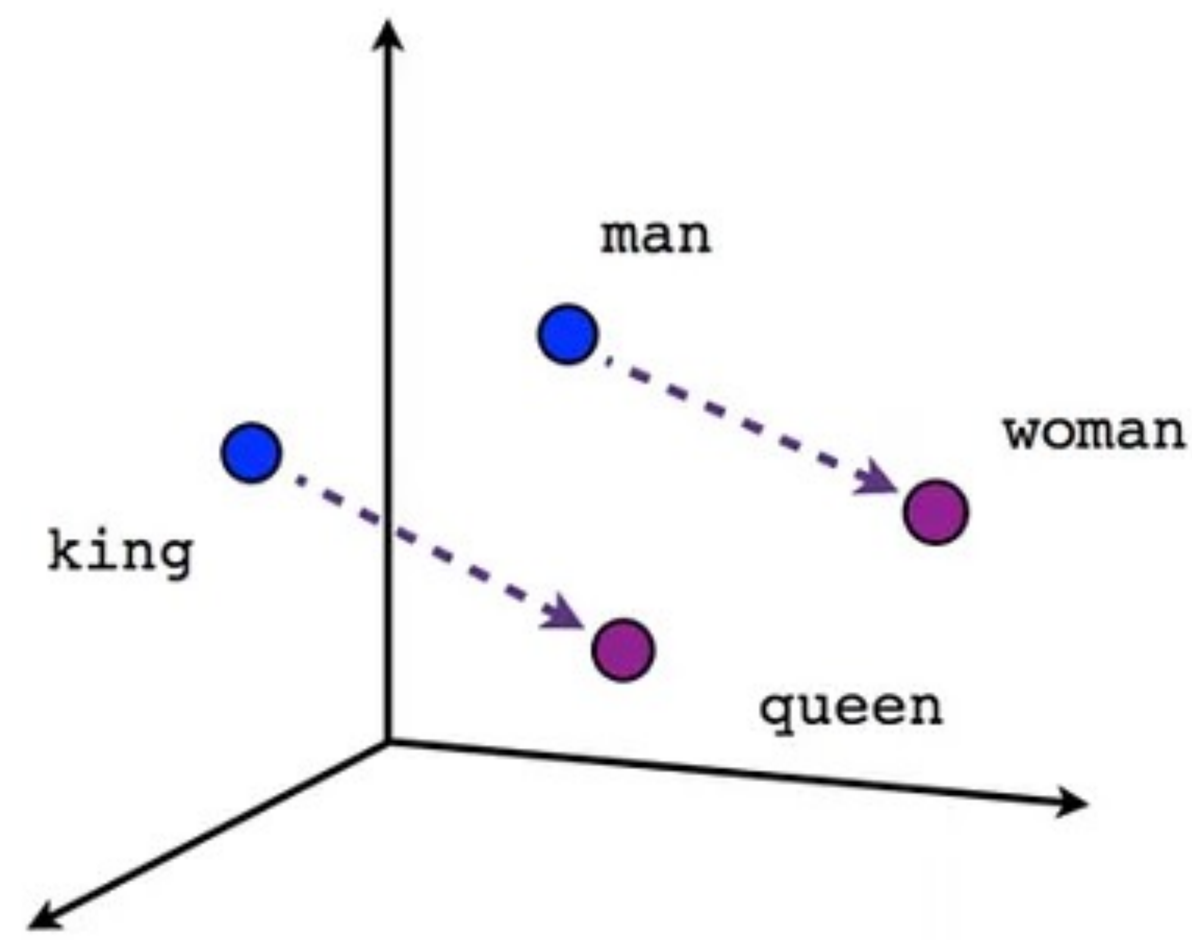
- The goal of NLP is to produce a sequence of outputs from a sequence of inputs
- In order to feed words/characters of natural languages to a NN, we need to represent each word/character as a vector in $\mathbb{R}^{n_{\text{in}}}$
- A simple way of doing this is as follows:
  - Tokenize the words/alphabet:
    - ✦ Take a dictionary (with 10k unique English words, say) and enumerate each word from 1 to 10k.
    - ✦ assign to each word a 10'000-dim vector, which has a 1 at position I and zeros everywhere else

      a $\boxed{1\,0\,0}\cdots\boxed{0\,0}$ Aardvark $\boxed{0\,1\,0}\cdots\boxed{0\,0}$ … Zyzzyva $\boxed{0\,0\,0}\cdots\boxed{0\,1}$
  - Multiply this 10k-dim vector with a 512 x 10k matrix to produce a vector in $\mathbb{R}^{512}$

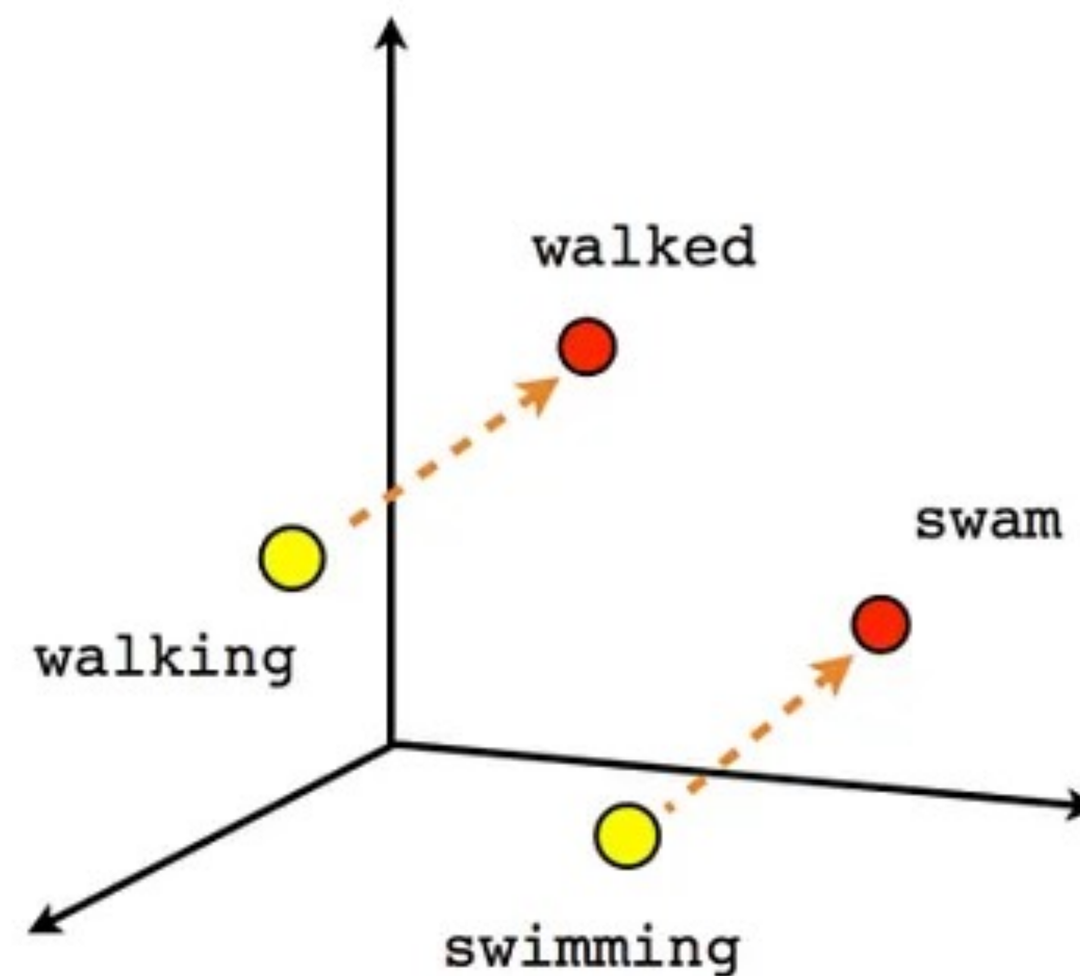
$$\begin{pmatrix} e_{1,1} & \cdots & e_{10000,1} \\ \vdots & \vdots & \vdots \\ e_{1,512} & \cdots & e_{10000,512} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} e_{1,1} \\ \vdots \\ e_{1,512} \end{pmatrix} \qquad \begin{pmatrix} e_{1,1} & \cdots & e_{10000,1} \\ \vdots & \vdots & \vdots \\ e_{1,512} & \cdots & e_{10000,512} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} e_{10000,1} \\ \vdots \\ e_{10000,512} \end{pmatrix}$$

  - Learning the entries of this matrix allows to learn "useful" embeddings

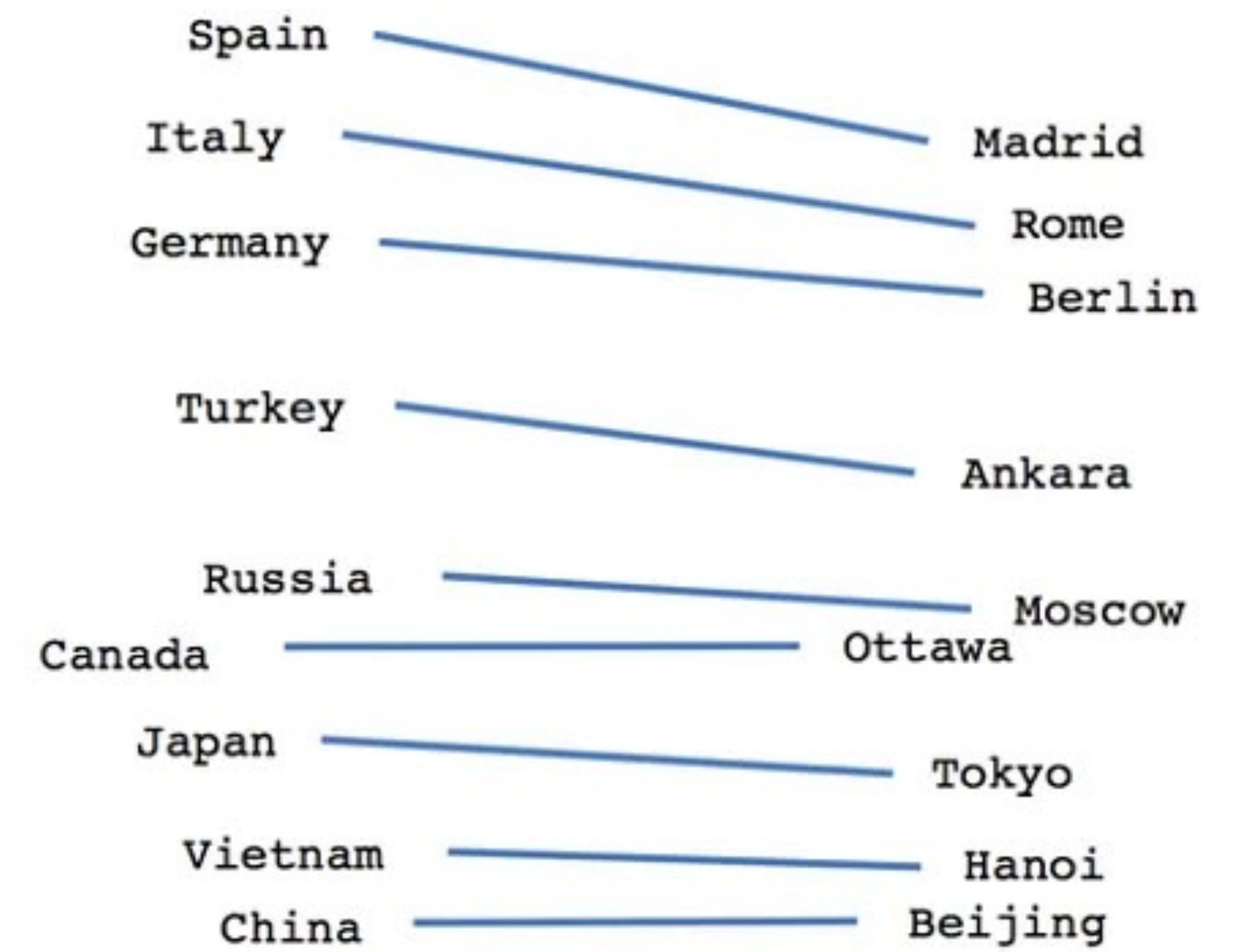
Examples of learned embeddings



Male-Female



Verb tense



Country-Capital

[Image Credit: Tensorflow]

Learns semantics

"queen - woman + man = king"

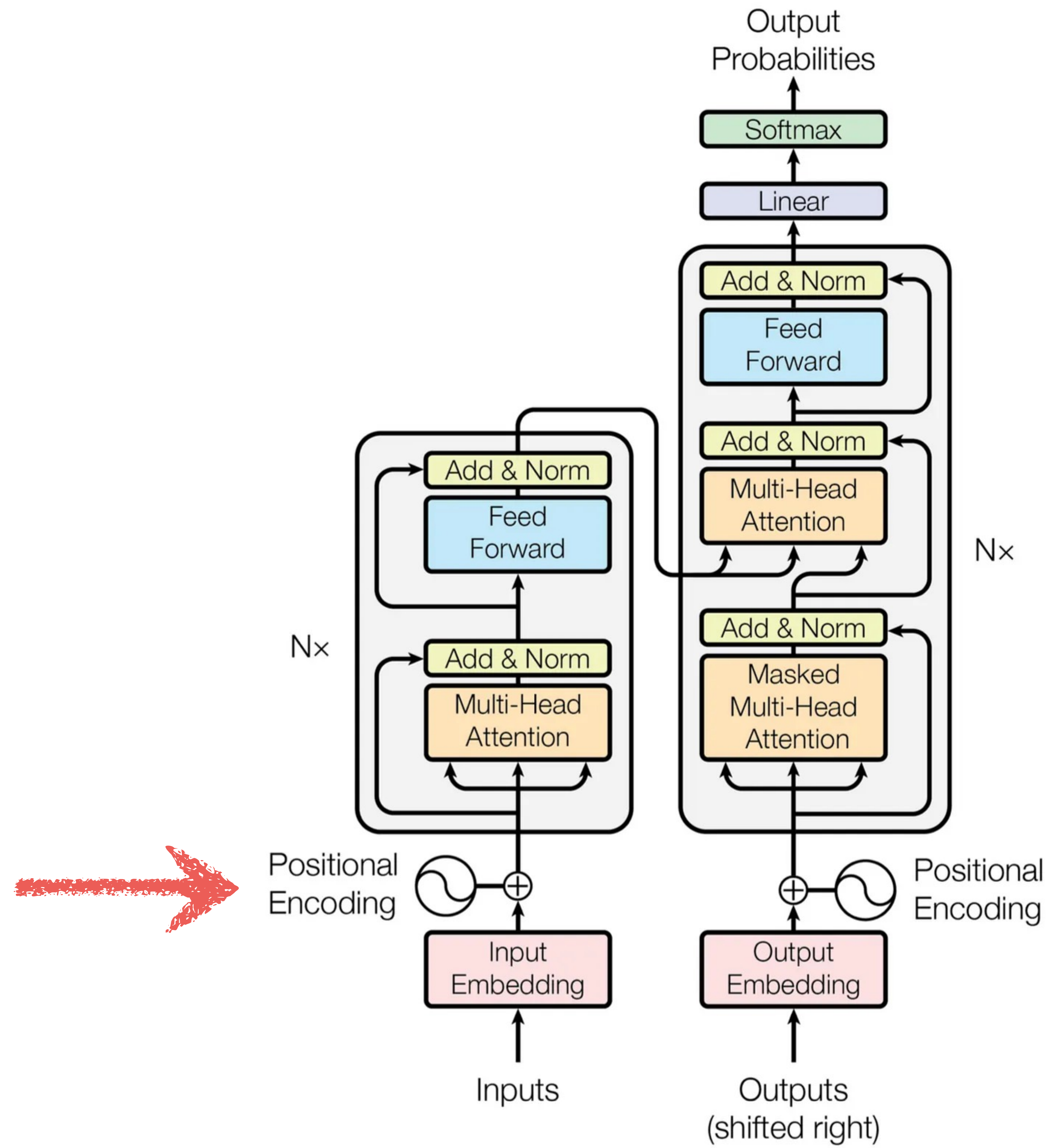


Figure 1: The Transformer - model architecture.

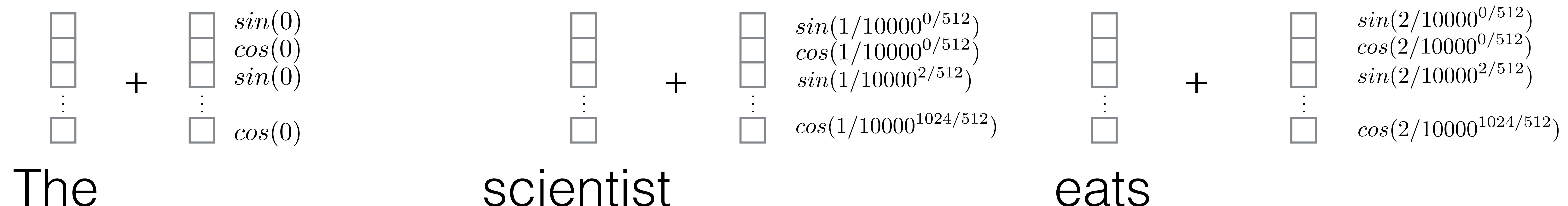
Positional embedding

- A fully connected layer has no notion of position: each node is connected to all other nodes
- But position of words in a sentence is very important:

  The scientist eats the chicken ≠ The chicken eats the scientist

- Add a (time-modulated) signal to each embedded word to signal its position in the sentence. The NN will learn to use it

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



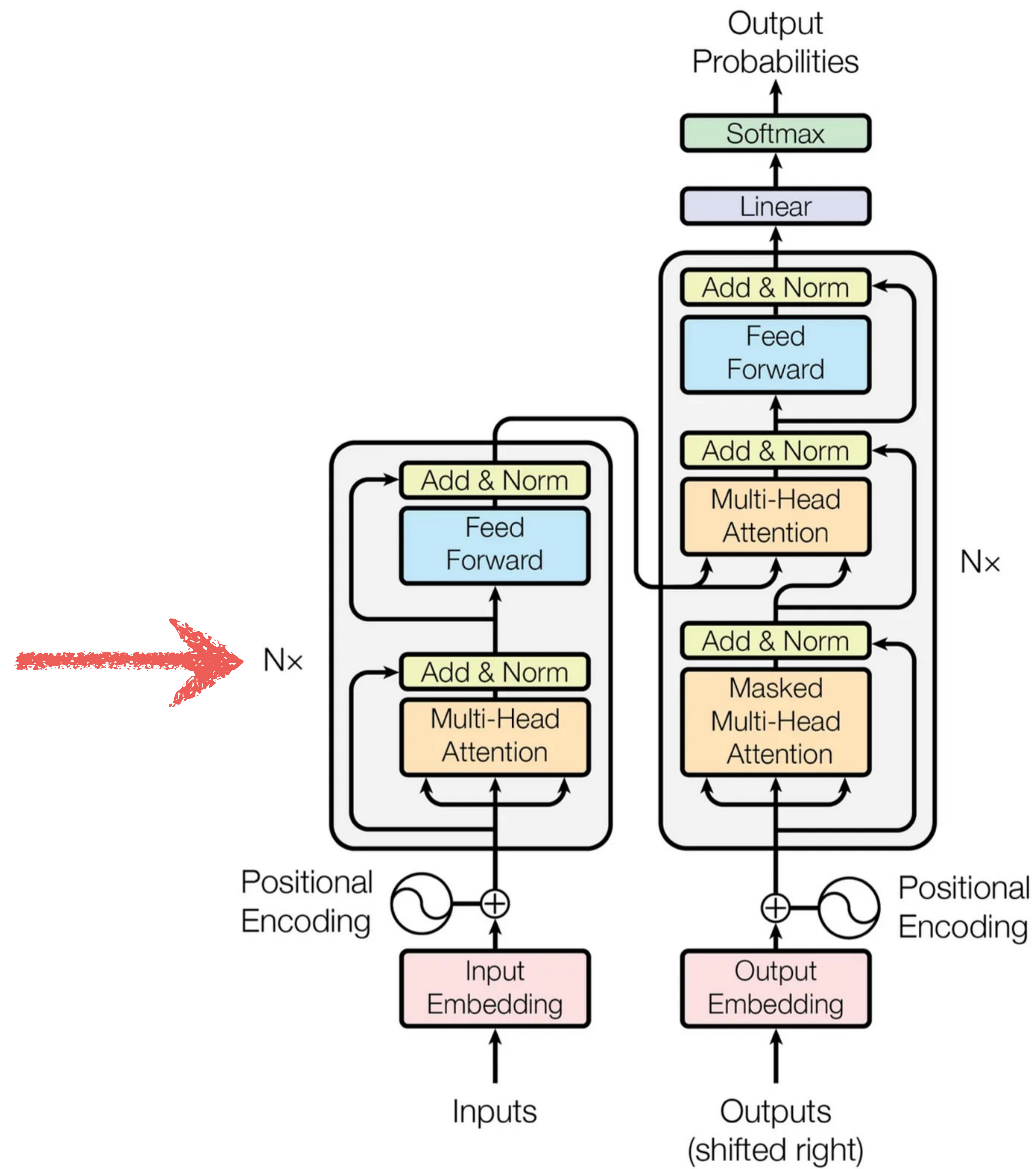
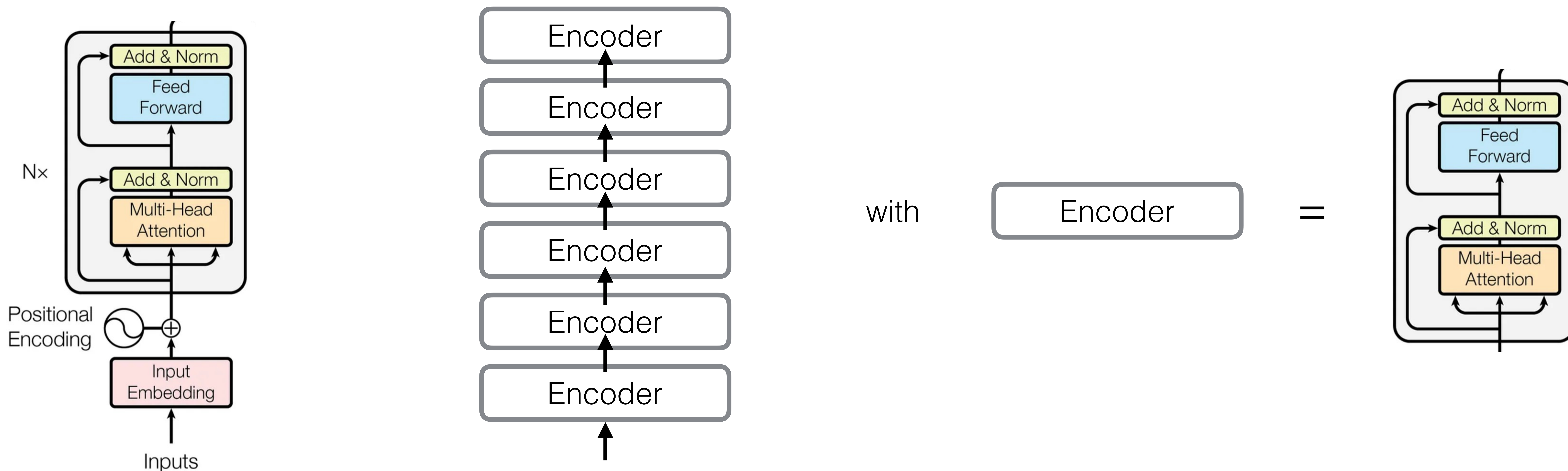


Figure 1: The Transformer - model architecture.

Encoding

- Now we have an encoded sequence of words with temporal information
- Just as in a deep NN, we now keep encoding this sequence over and over, feeding the output of the last encoder to the input of the next
- Repeat (6 times in the paper) until the NN has learned a good encoding



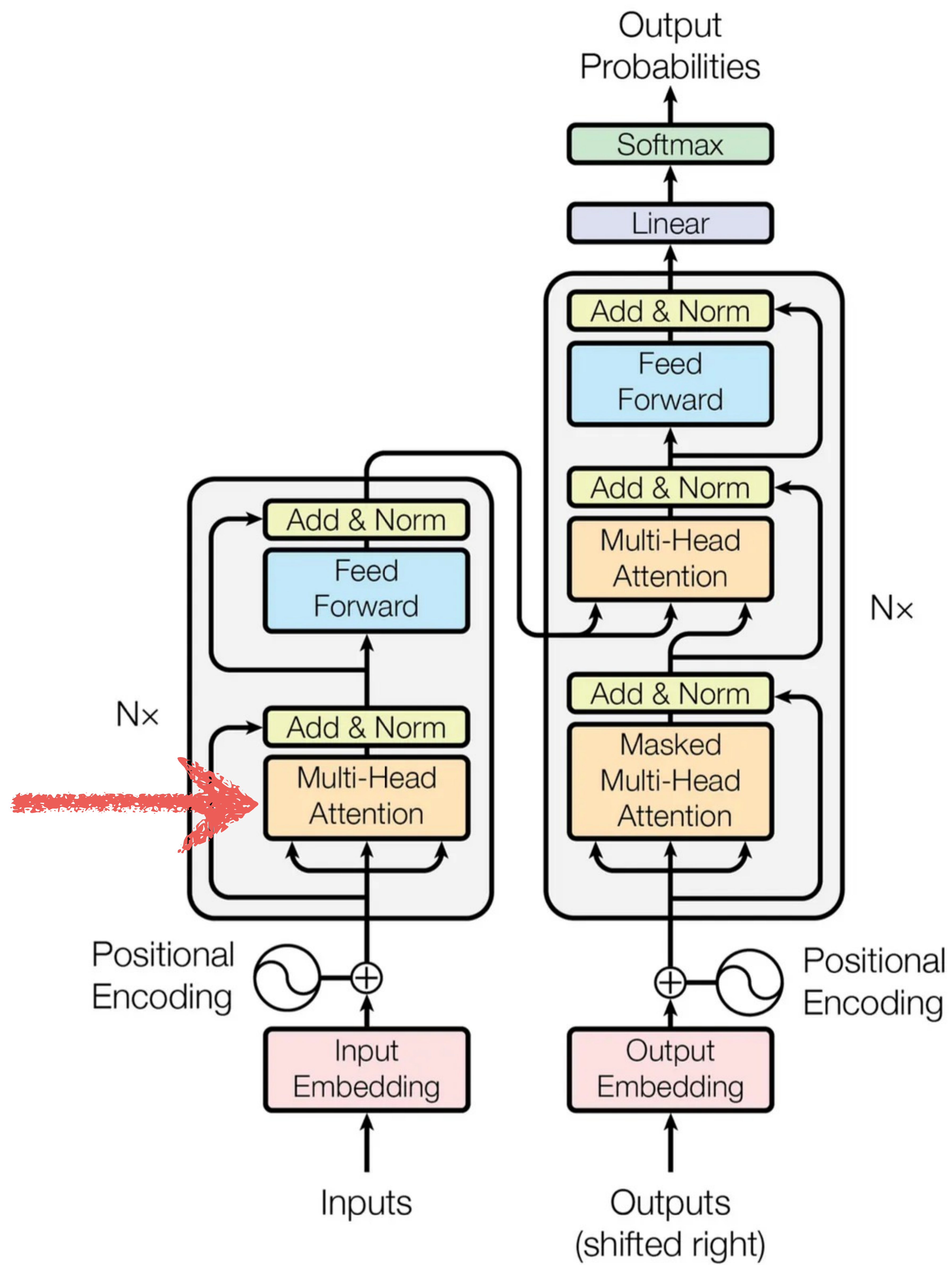
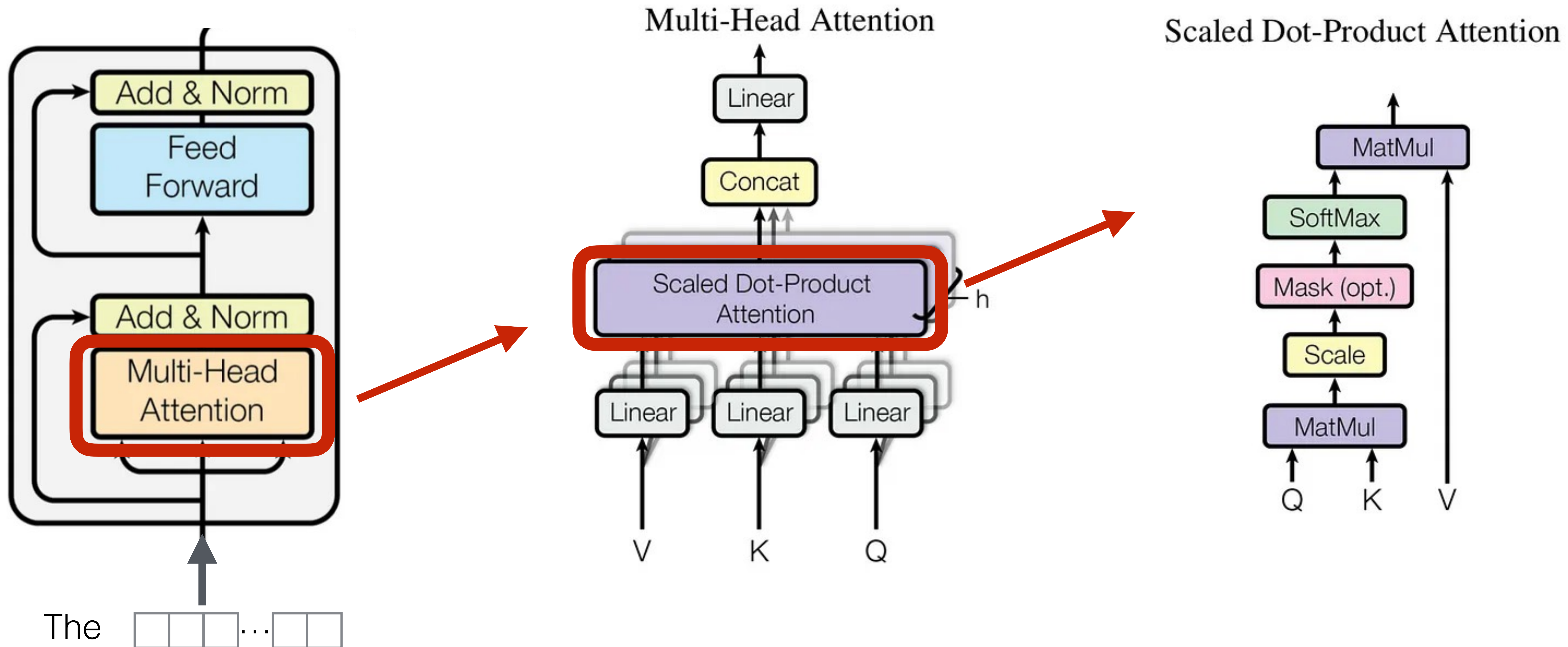


Figure 1: The Transformer - model architecture.

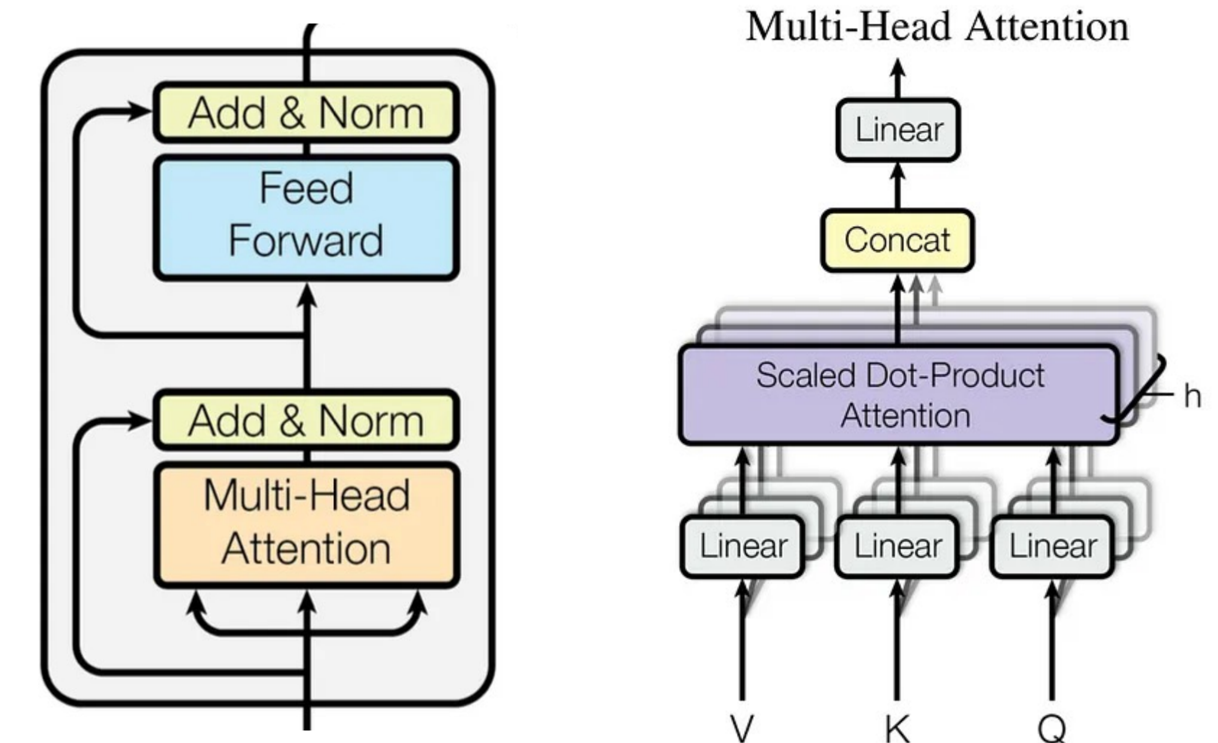
Self-attention

- So each encoder layer consists of attention heads and a vanilla feed-forward layer (and normalizations and skip connections for better training performance)



Self-attention I

- As illustrated, we create 3 copies of each input vector, denoted by V(value), K(ey), Q(uery)
- This is done for all input vector (i.e. each encoded word) in the sequence
- Then, each copy V, K,Q goes into a feed-forward layer (i.e., matrix multiplication, where the entries of the matrix are learnable parameters)



$$\begin{pmatrix} v_{1,1} & \cdots & v_{512,1} \\ \vdots & \vdots & \vdots \\ v_{1,64} & \cdots & v_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{The}} = \begin{pmatrix} v_1(\text{“The”}) \\ \vdots \\ v_{64}(\text{“The”}) \end{pmatrix}$$

$$\begin{pmatrix} v_{1,1} & \cdots & v_{512,1} \\ \vdots & \vdots & \vdots \\ v_{1,64} & \cdots & v_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{scientist}} = \begin{pmatrix} v_1(\text{“scientist”}) \\ \vdots \\ v_{64}(\text{“scientist”}) \end{pmatrix}$$

$$\begin{pmatrix} v_{1,1} & \cdots & v_{512,1} \\ \vdots & \vdots & \vdots \\ v_{1,64} & \cdots & v_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{eats}} = \begin{pmatrix} v_1(\text{“eats”}) \\ \vdots \\ v_{64}(\text{“eats”}) \end{pmatrix}$$

$$\begin{pmatrix} k_{1,1} & \cdots & k_{512,1} \\ \vdots & \vdots & \vdots \\ k_{1,64} & \cdots & k_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{The}} = \begin{pmatrix} k_1(\text{“The”}) \\ \vdots \\ k_{64}(\text{“The”}) \end{pmatrix}$$

$$\begin{pmatrix} k_{1,1} & \cdots & k_{512,1} \\ \vdots & \vdots & \vdots \\ k_{1,64} & \cdots & k_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{scientist}} = \begin{pmatrix} k_1(\text{“scientist”}) \\ \vdots \\ k_{64}(\text{“scientist”}) \end{pmatrix}$$

$$\begin{pmatrix} k_{1,1} & \cdots & k_{512,1} \\ \vdots & \vdots & \vdots \\ k_{1,64} & \cdots & k_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{eats}} = \begin{pmatrix} k_1(\text{“eats”}) \\ \vdots \\ k_{64}(\text{“eats”}) \end{pmatrix}$$

$$\begin{pmatrix} q_{1,1} & \cdots & q_{512,1} \\ \vdots & \vdots & \vdots \\ q_{1,64} & \cdots & q_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{The}} = \begin{pmatrix} q_1(\text{“The”}) \\ \vdots \\ q_{64}(\text{“The”}) \end{pmatrix}$$

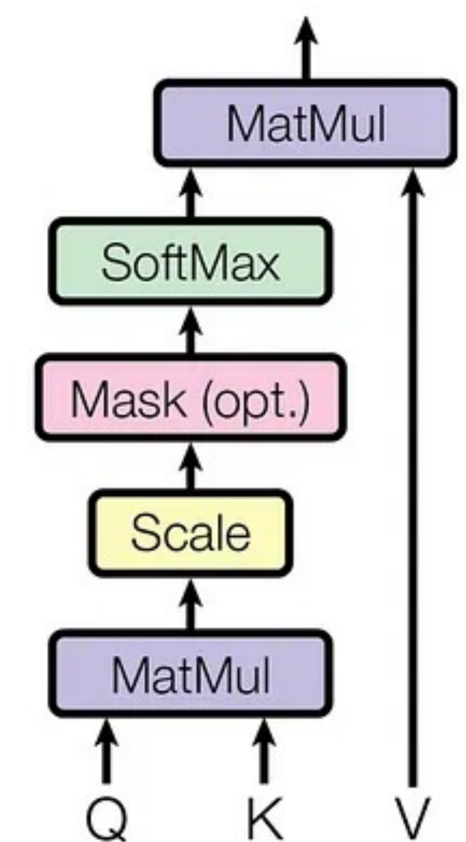
$$\begin{pmatrix} q_{1,1} & \cdots & q_{512,1} \\ \vdots & \vdots & \vdots \\ q_{1,64} & \cdots & q_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{scientist}} = \begin{pmatrix} q_1(\text{“scientist”}) \\ \vdots \\ q_{64}(\text{“scientist”}) \end{pmatrix}$$

$$\begin{pmatrix} q_{1,1} & \cdots & q_{512,1} \\ \vdots & \vdots & \vdots \\ q_{1,64} & \cdots & q_{512,64} \end{pmatrix} \underbrace{\begin{pmatrix}\square \\ \square \\ \square \\ \square\end{pmatrix}}_{\text{eats}} = \begin{pmatrix} q_1(\text{“eats”}) \\ \vdots \\ q_{64}(\text{“eats”}) \end{pmatrix}$$

Self-attention II

- The set of 64-dim vectors $(q(\text{word}_i), k(\text{word}_i), v(\text{word}_i))$ are then used to compute a score between the current word and all other words in the input sentence.
- The higher the score the more relevant the other word is for understanding the current word
- This score (or attention) is computed as follows:
  - Take the dot product (think "overlap" of the encoded words) between the query and the key for all words in the input sentence
  - Scale it down by $\sqrt{64}$ to avoid too big gradients, and optionally mask future inputs such that you can only pay attention to what has been seen already (for decoder)
  - Take the softmax to convert the "overlaps" to probabilities
  - The resulting score tells you how relevant other words in the sentence are for understanding the current word
  - Multiply the value of all words with this score and sum them all up to get a (superposition of) words which one should pay attention to for understanding the first word. Rinse and repeat

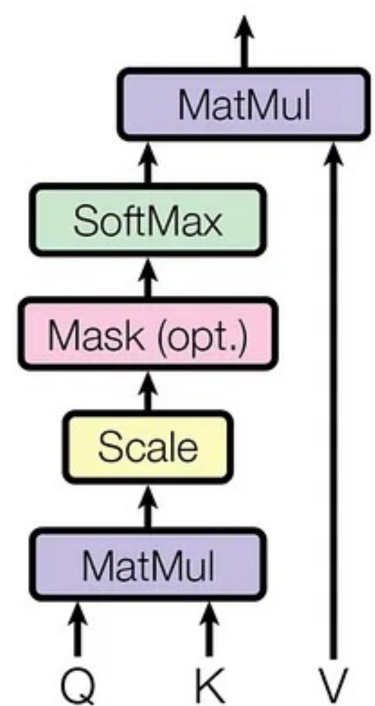
Scaled Dot-Product Attention



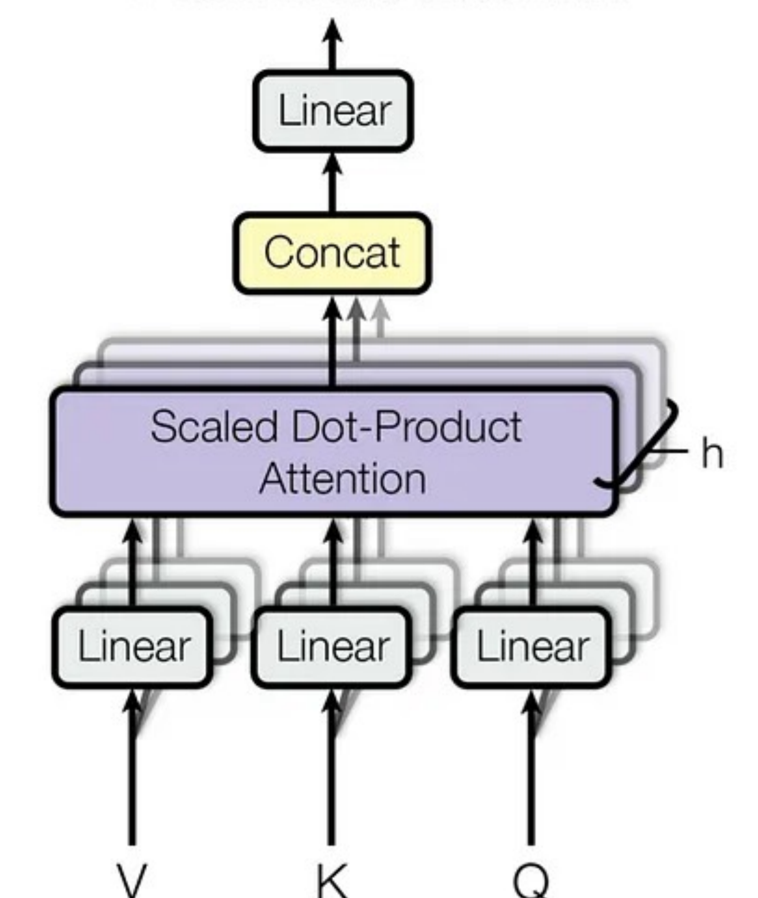
Self-attention III

- In matrix notation, the scaled dot-product attention head is

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



- Repeat this attention step in parallel for 8 different (learnable) matrices Q,K,V ("multi-heads")
- The resulting 64-dimensional vectors from each head are put into one large vector of dimension $64 \times 8 = 512$
- This vector is fed into a fully-connected layer (with output dimension 512, so that it can fit into the next encoding layer with the same architecture



Self-attention example

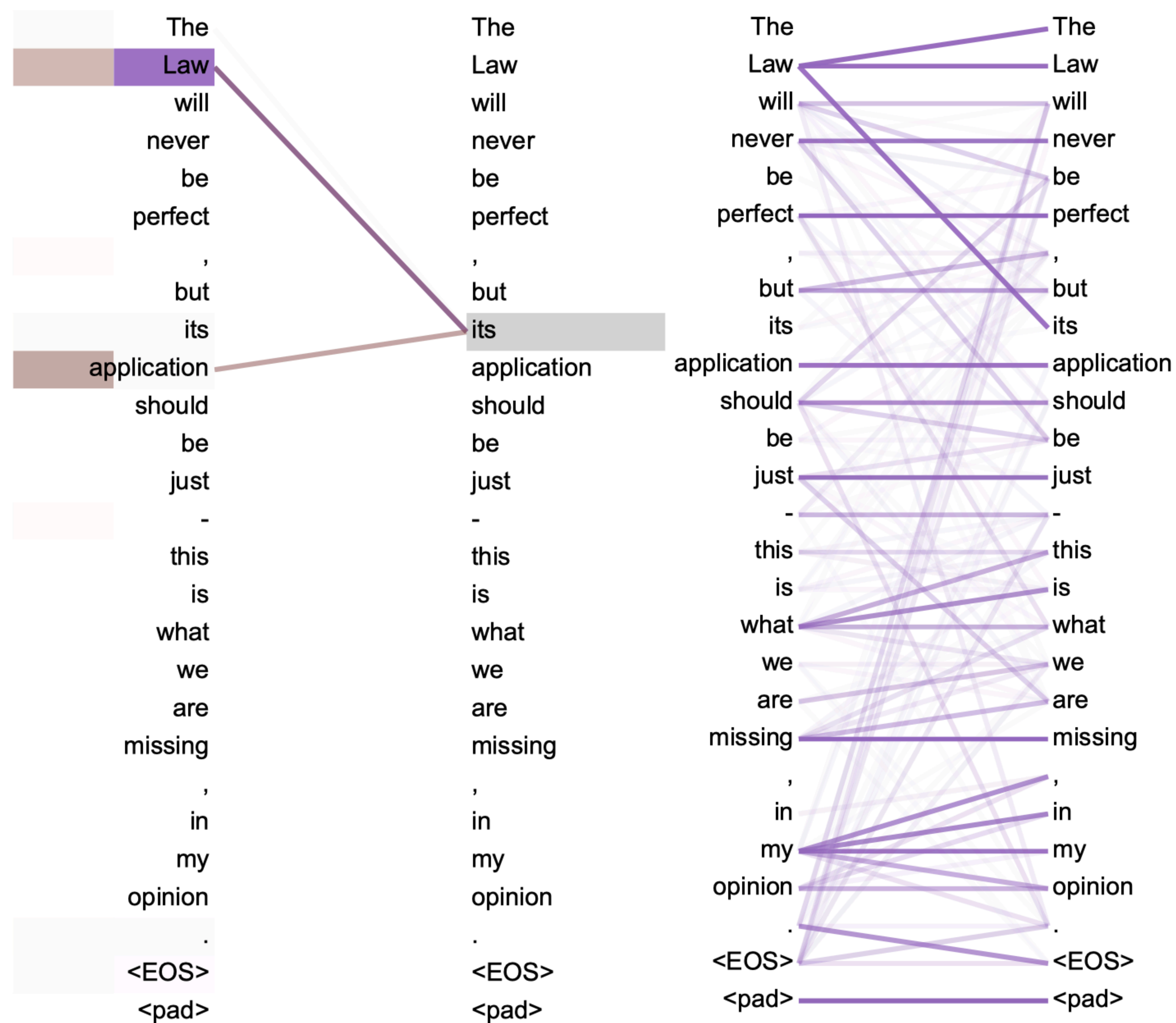
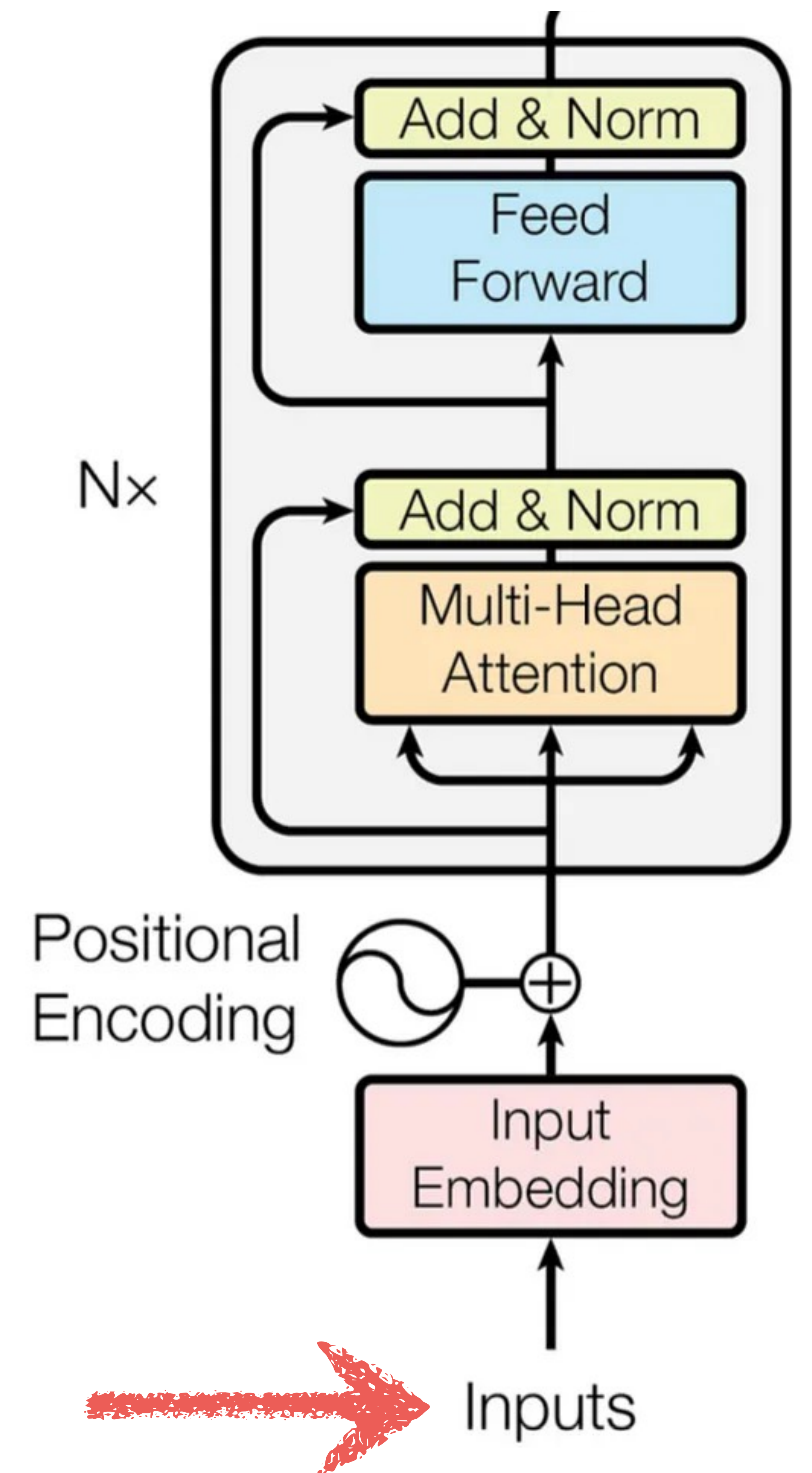


Figure 4: Two attention heads, also in layer 5 of 6, apparently involved in anaphora resolution. **Left:** Isolated attentions from just the word 'its' for attention heads 5 and 6. Note that the attentions are very sharp for this word. **Right:** Full attentions for head 5.

Summary of encoder



[The, scientist, eats, the, chicken, eos]

Summary of encoder

$$\begin{pmatrix} e_{1,1} & \cdots & e_{10000,1} \\ \vdots & \vdots & \vdots \\ e_{1,512} & \cdots & e_{10000,512} \end{pmatrix} \begin{bmatrix} \square \\ \square \\ \square \\ \vdots \\ \square \end{bmatrix} = \begin{pmatrix} e_1(\text{“The”}) \\ \vdots \\ e_{512}(\text{“The”}) \end{pmatrix}$$

The

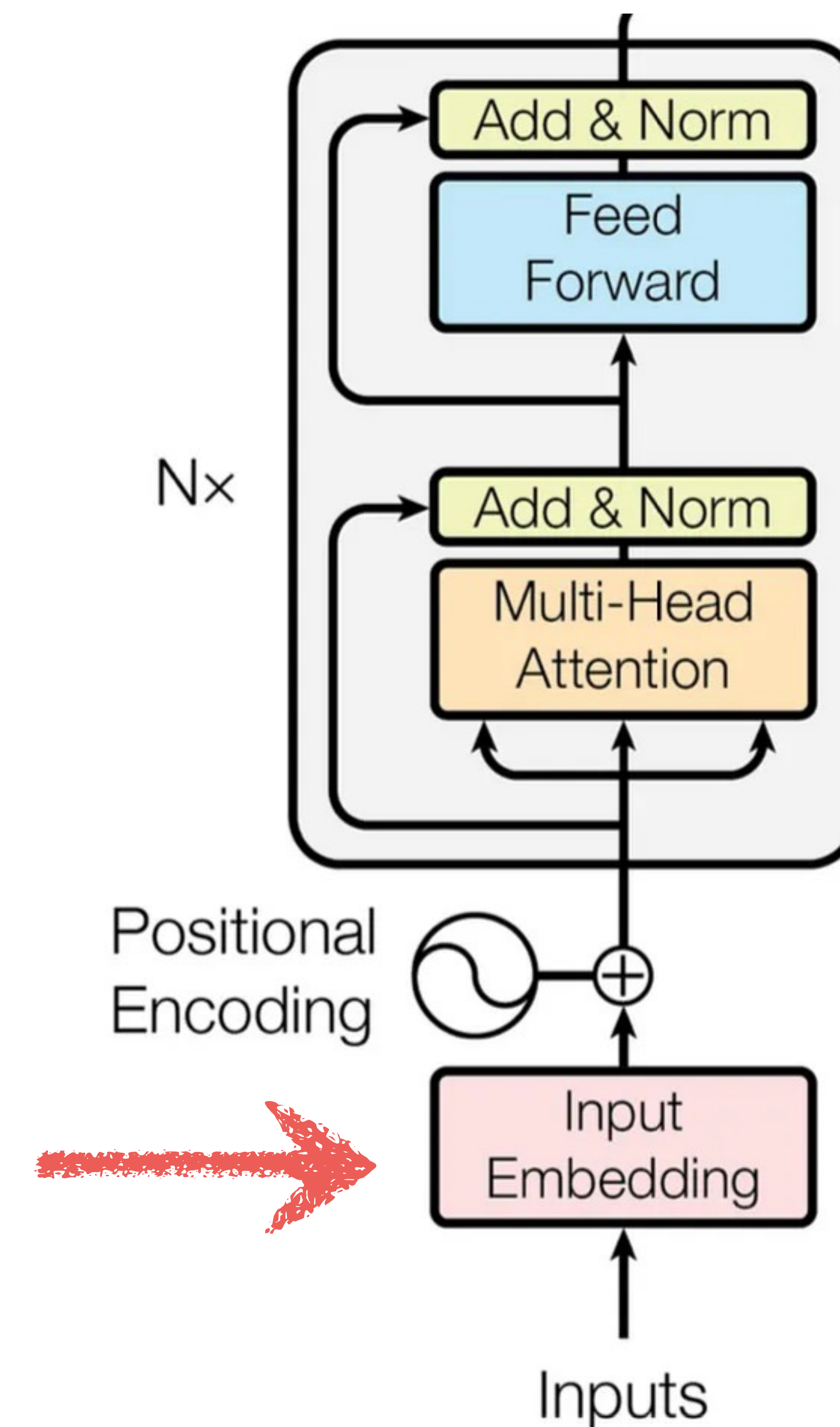
$$\begin{pmatrix} e_{1,1} & \cdots & e_{10000,1} \\ \vdots & \vdots & \vdots \\ e_{1,512} & \cdots & e_{10000,512} \end{pmatrix} \begin{bmatrix} \square \\ \square \\ \square \\ \vdots \\ \square \end{bmatrix} = \begin{pmatrix} e_1(\text{“scientist”}) \\ \vdots \\ e_{512}(\text{“scientist”}) \end{pmatrix}$$

scientist

$$\begin{pmatrix} e_{1,1} & \cdots & e_{10000,1} \\ \vdots & \vdots & \vdots \\ e_{1,512} & \cdots & e_{10000,512} \end{pmatrix} \begin{bmatrix} \square \\ \square \\ \square \\ \vdots \\ \square \end{bmatrix} = \begin{pmatrix} e_1(\text{“eats”}) \\ \vdots \\ e_{512}(\text{“eats”}) \end{pmatrix}$$

eats

$\vdots$

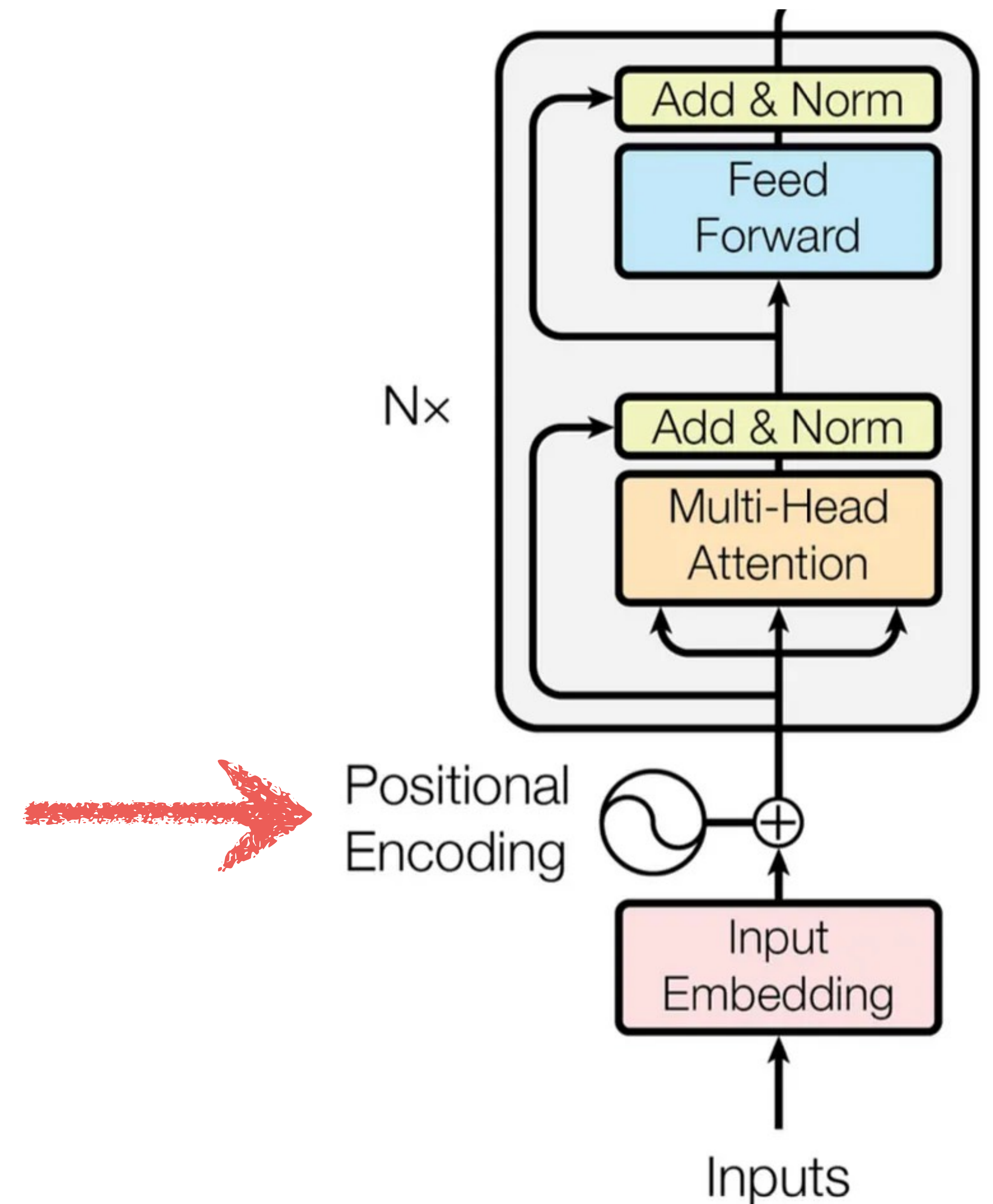


Summary of encoder

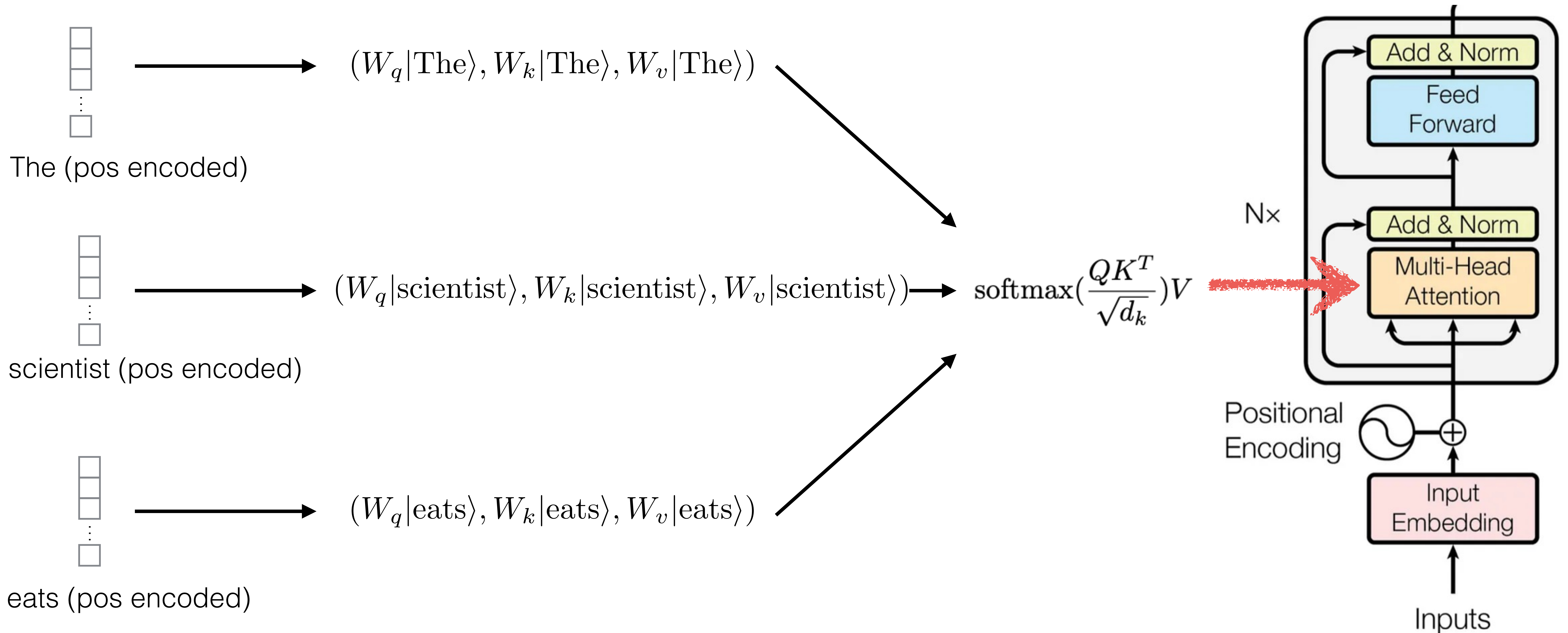
$$\begin{pmatrix} e_1(\text{"The"}) \\ \vdots \\ e_{512}(\text{"The"}) \end{pmatrix} + \begin{matrix} sin(0) \\ cos(0) \\ sin(0) \\ \vdots \\ cos(0) \end{matrix} = |\text{The}\rangle$$

$$\begin{pmatrix} e_1(\text{"scientist"}) \\ \vdots \\ e_{512}(\text{"scientist"}) \end{pmatrix} + \begin{matrix} sin(1/10000^{0/512}) \\ cos(1/10000^{0/512}) \\ sin(1/10000^{2/512}) \\ \vdots \\ cos(1/10000^{1024/512}) \end{matrix} = |\text{scientist}\rangle$$

$$\begin{pmatrix} e_1(\text{"eats"}) \\ \vdots \\ e_{512}(\text{"eats"}) \end{pmatrix} + \begin{matrix} sin(2/10000^{0/512}) \\ cos(2/10000^{0/512}) \\ sin(2/10000^{2/512}) \\ \vdots \\ cos(2/10000^{1024/512}) \end{matrix} = |\text{eats}\rangle$$



Summary of encoder



Summary of encoder

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

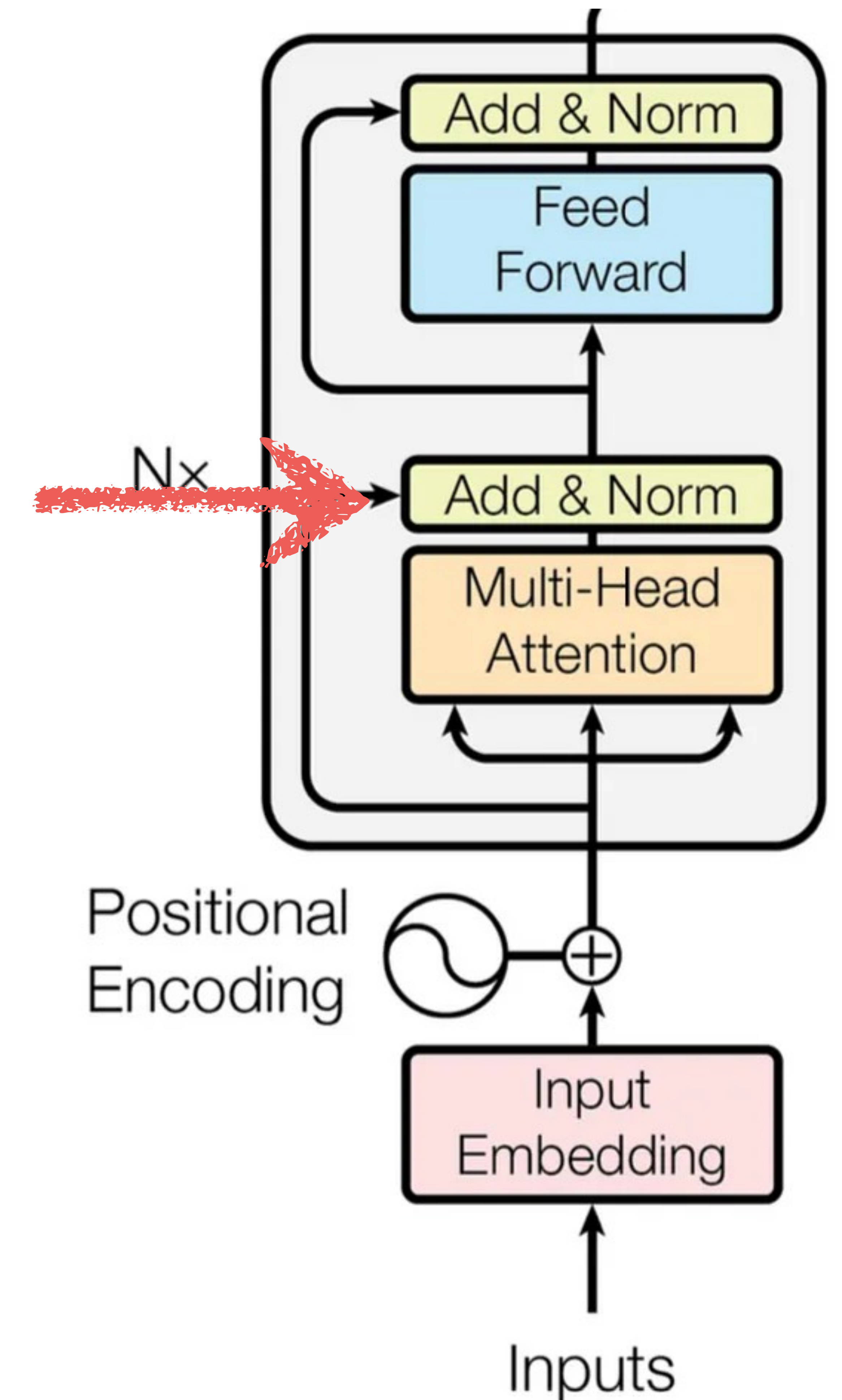
$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

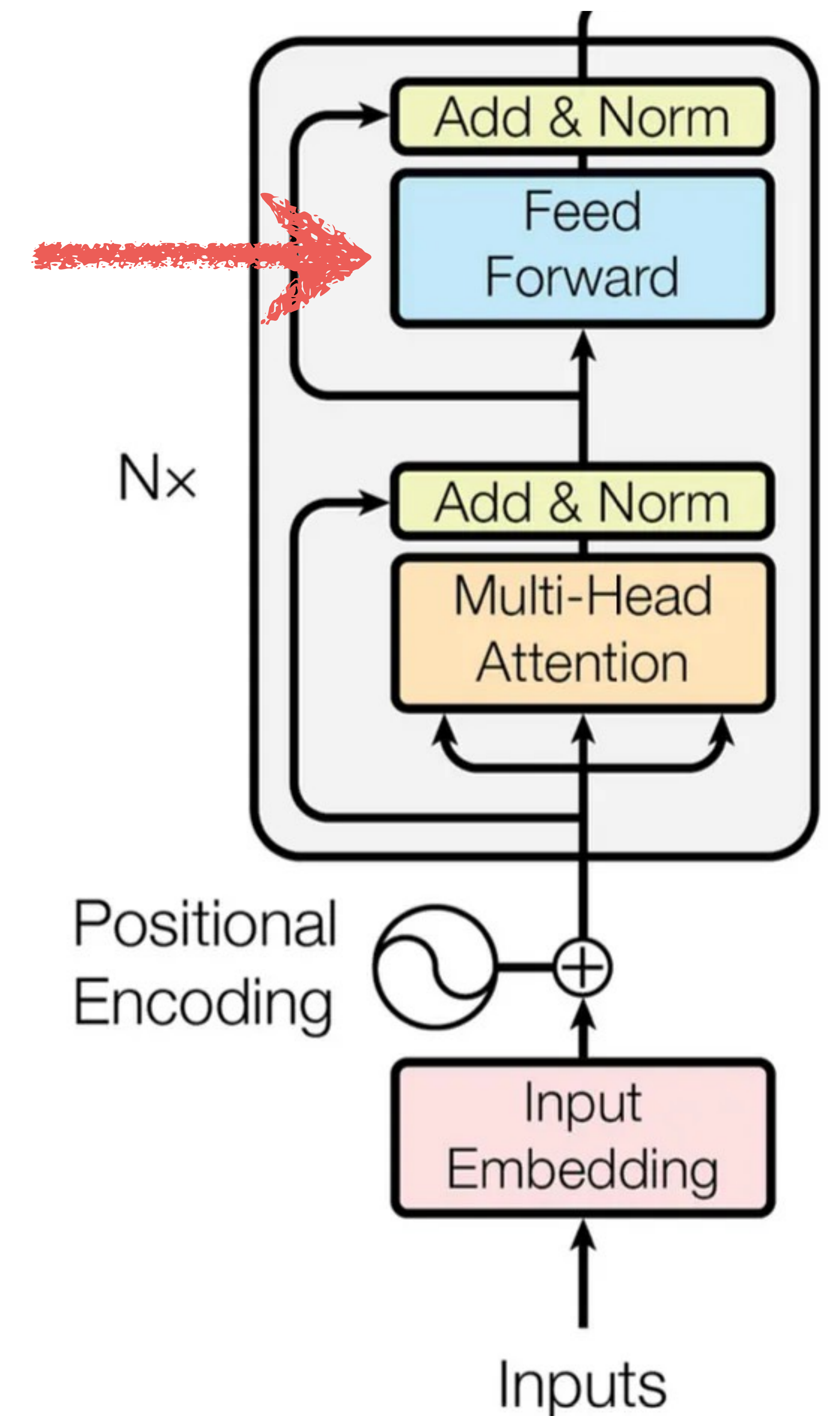
$$\text{Layer_norm}(\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V + |\text{The}\rangle)$$

$$\bigoplus \rightarrow |\text{mha}(\text{"The"})\rangle$$



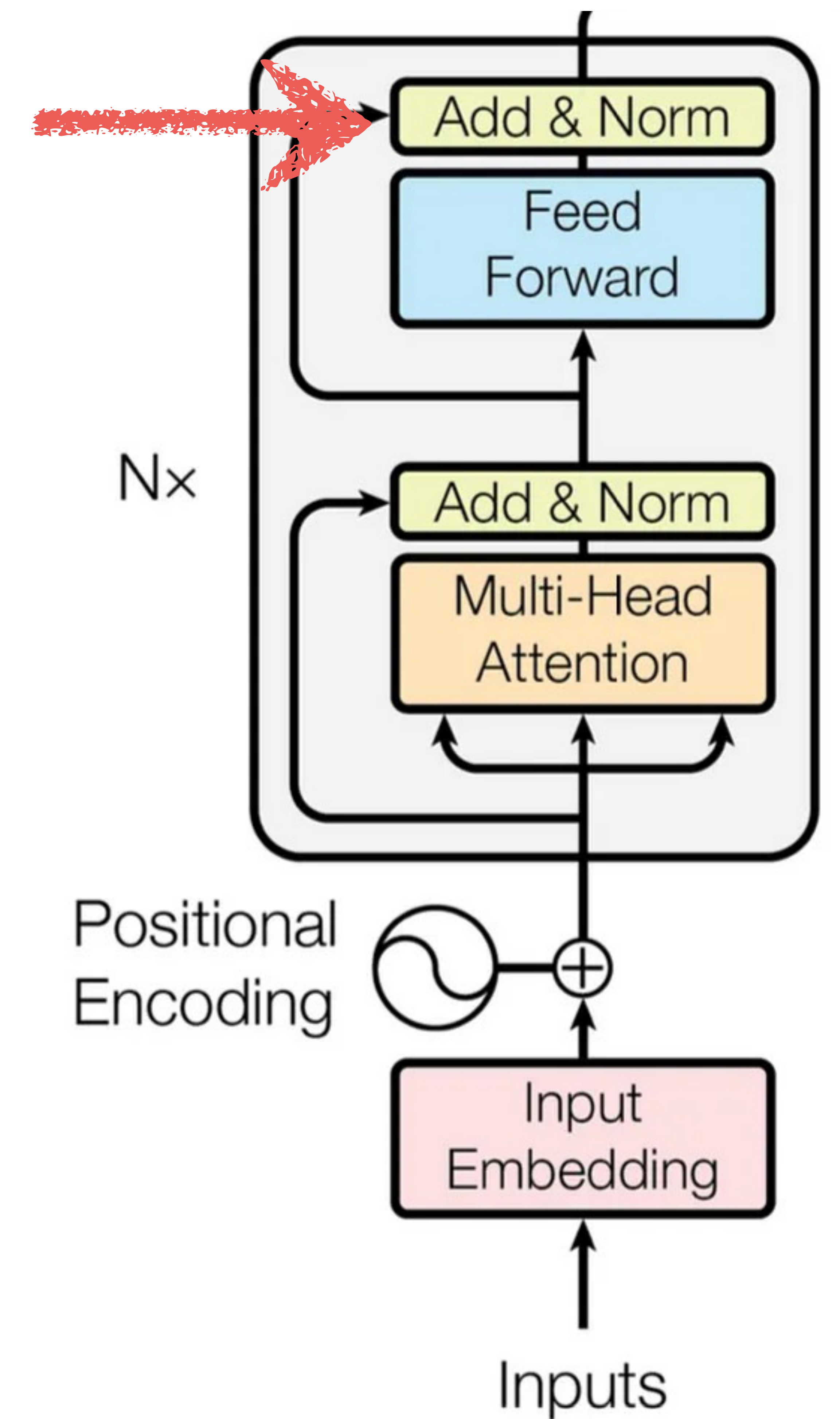
Summary of encoder

$$W_2\,\text{ReLU}(W_1|\text{mha}(\text{"The"})\rangle + b_1) \;+\; b_2 \;=\; |\text{tmha}(\text{"The"})\rangle$$



Summary of encoder

Layer_norm($|\text{tmha}(\text{"The"})\rangle$ + $|\text{mha}(\text{"The"})\rangle$)



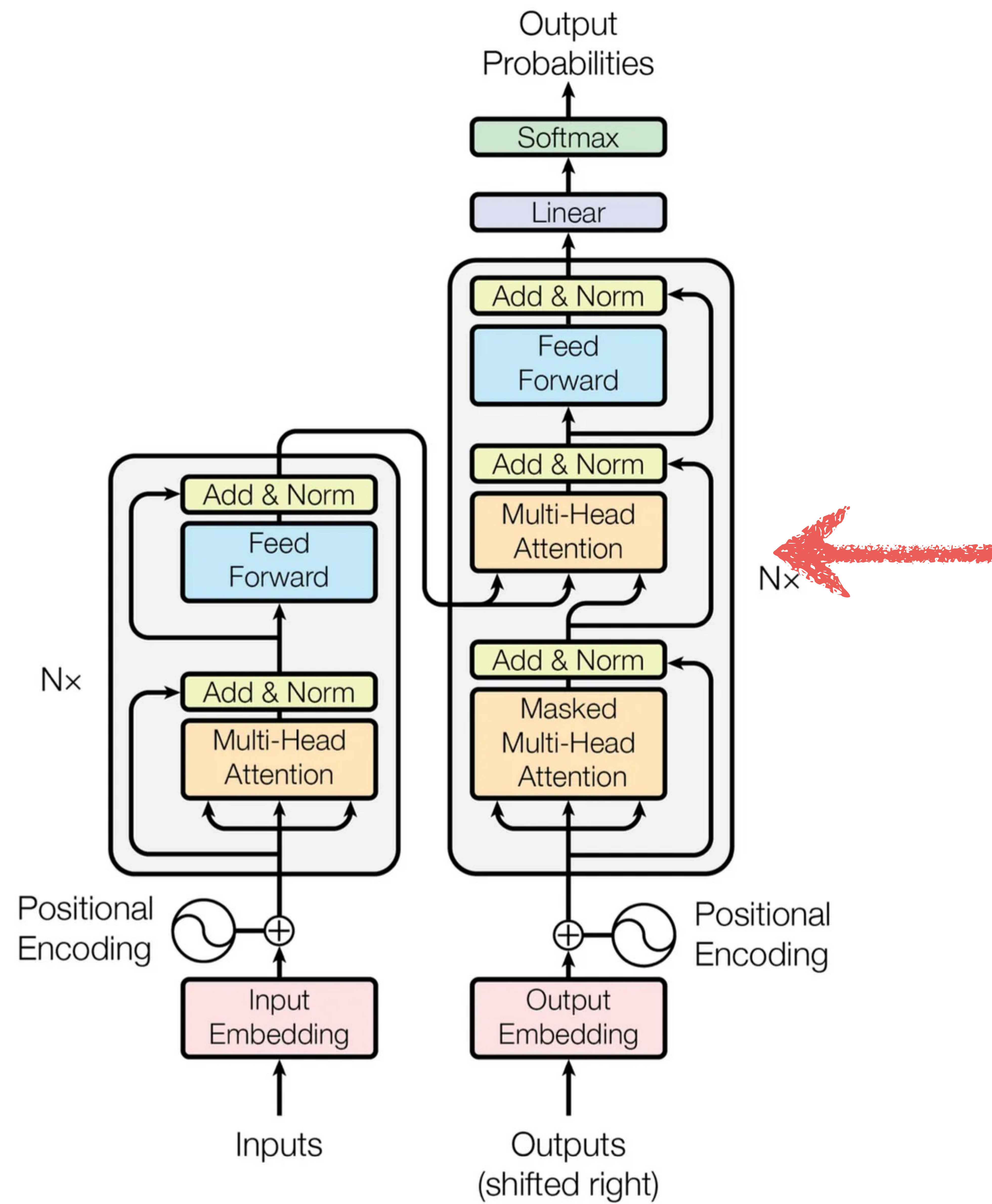


Figure 1: The Transformer - model architecture.

Decoder

- In a sequence to sequence task like translation, the decoder needs its own output and the (encoded) input it should translate
- The first multi-head attention computes an attention score+embedding from its own previous outputs

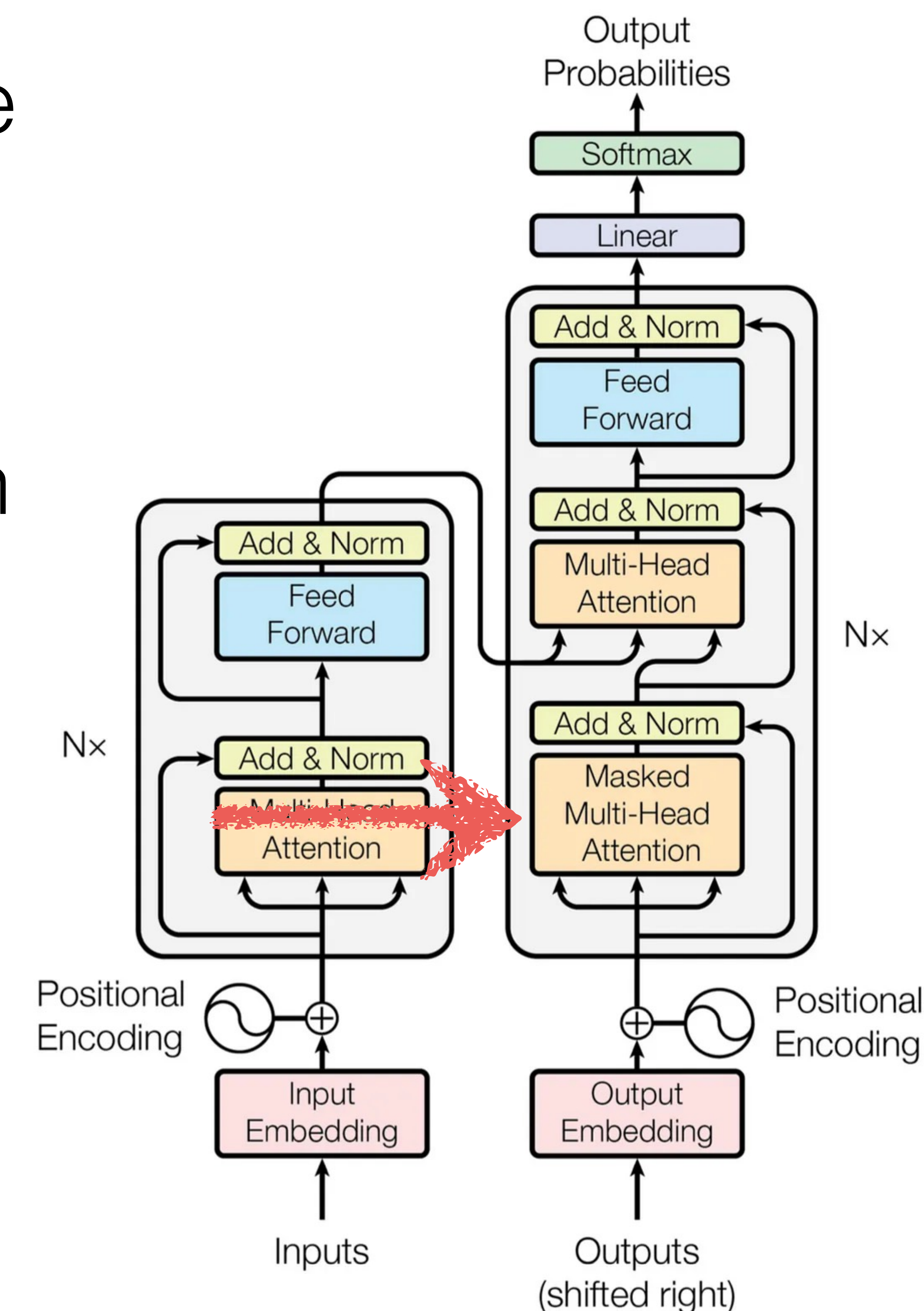


Figure 1: The Transformer - model architecture.

Decoder

- In a sequence to sequence task like translation, the decoder needs its own output and the (encoded) input it should translate
- The first multi-head attention computes an attention score+embedding from its own previous outputs
- The second multi-head attention takes this as the query and uses the output of the next word from the encoder as key and value
- Based on this, it computes another attention score, which is then used again in a FFNN

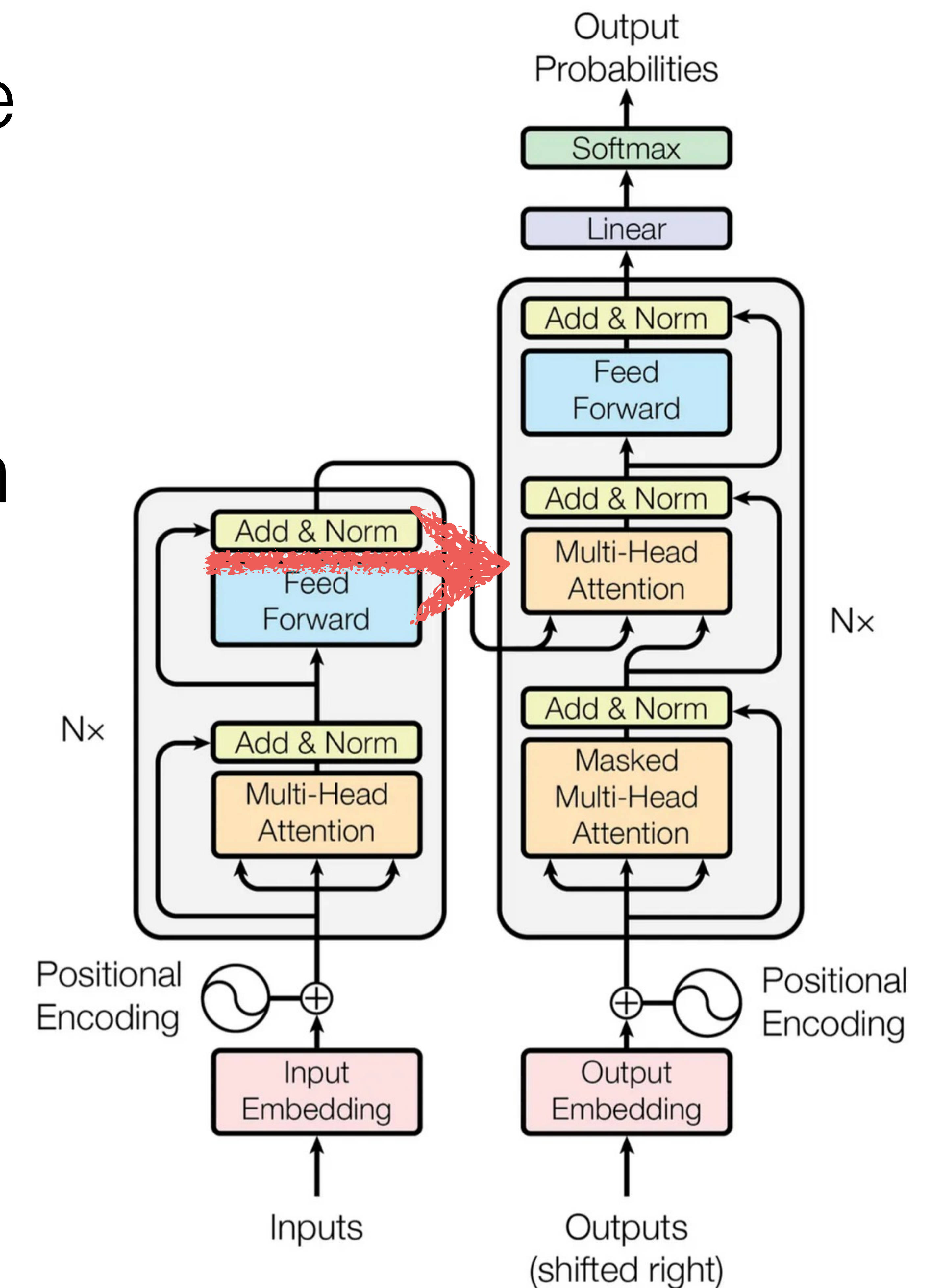


Figure 1: The Transformer - model architecture.

Decoder

- After repeating this decoding step 6 times, the 512dim output is put into a linear layer of dimension $10k \times 512$

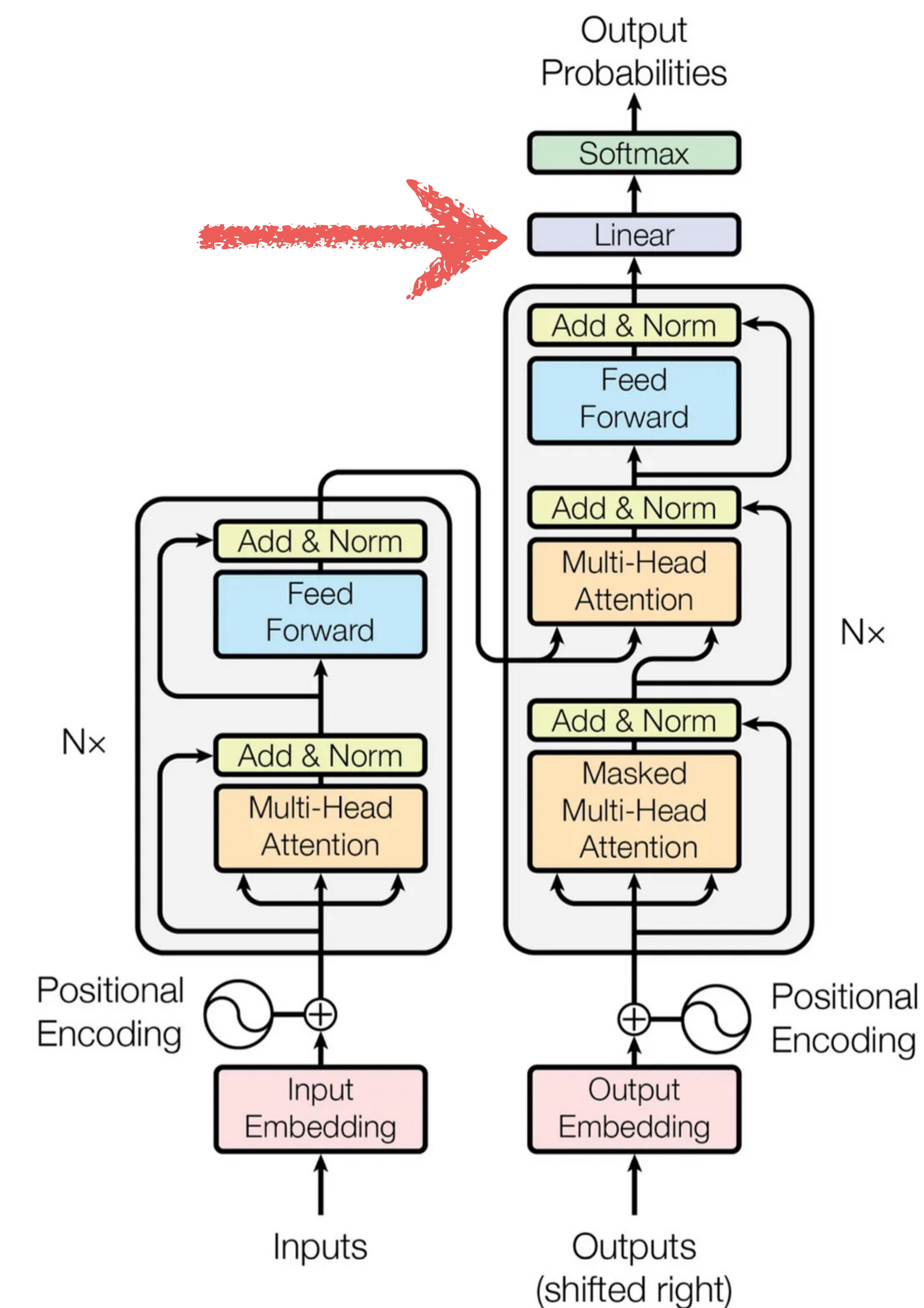


Figure 1: The Transformer - model architecture.

Decoder

- After repeating this decoding step 6 times, the 512dim output is put into a linear layer of dimension $10k \times 512$
- Then we take the softmax to obtain a $10k$-dim vector of probabilities which word to use next

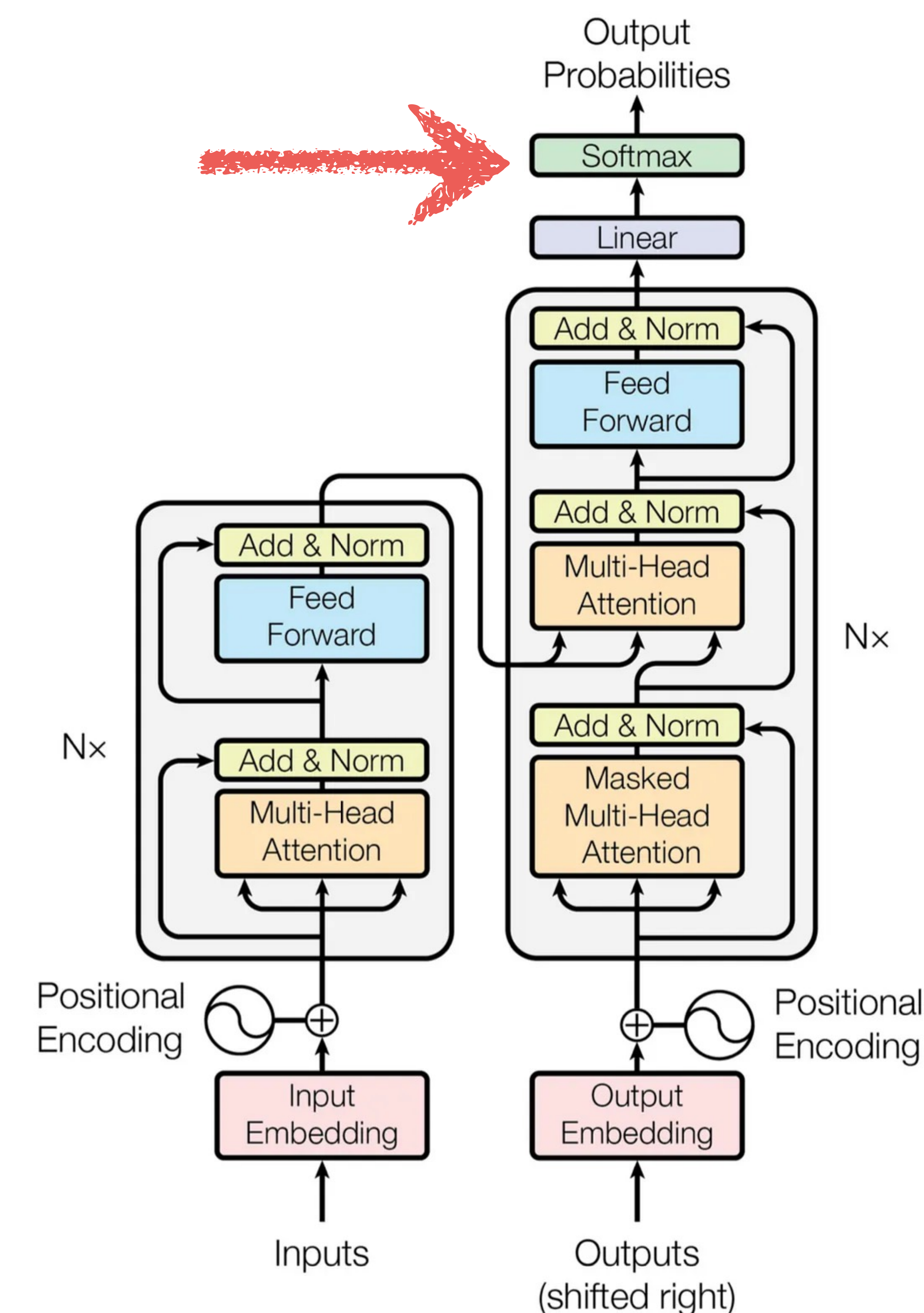


Figure 1: The Transformer - model architecture.

Decoder

- After repeating this decoding step 6 times, the 512dim output is put into a linear layer of dimension $10k \times 512$
- Then we take the softmax to obtain a $10k$-dim vector of probabilities which word to use next
- In greedy decoding, the word with the highest probability is chosen from the $10k$ dictionary entries and added to the output

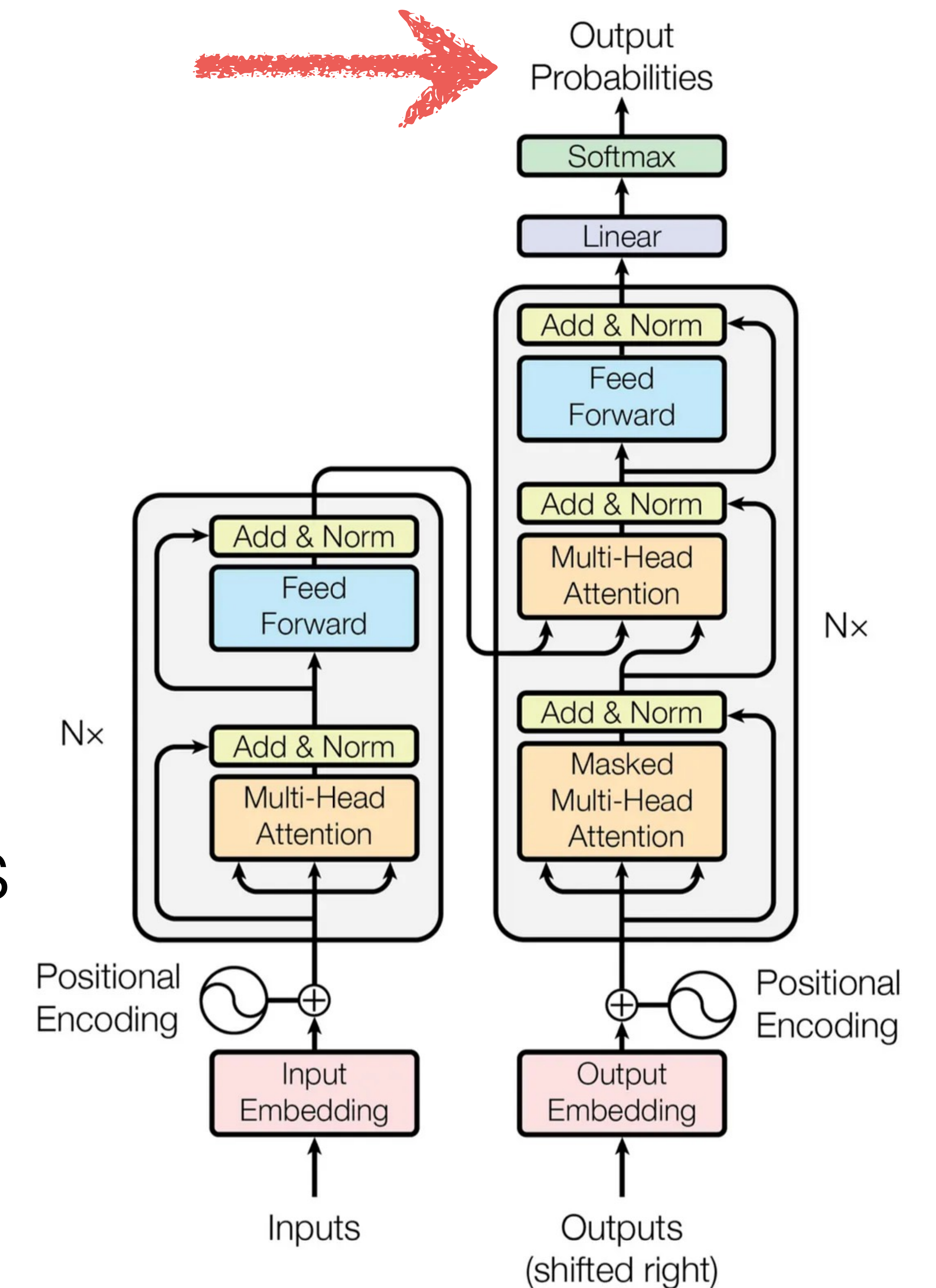


Figure 1: The Transformer - model architecture.

Decoder

- After repeating this decoding step 6 times, the 512dim output is put into a linear layer of dimension $10k \times 512$
- Then we take the softmax to obtain a $10k$-dim vector of probabilities which word to use next
- In greedy decoding, the word with the highest probability is chosen from the $10k$ dictionary entries and added to the output
- Then the decoding step is repeated with the new output that contains this new word, and the next word to translate from the encoder

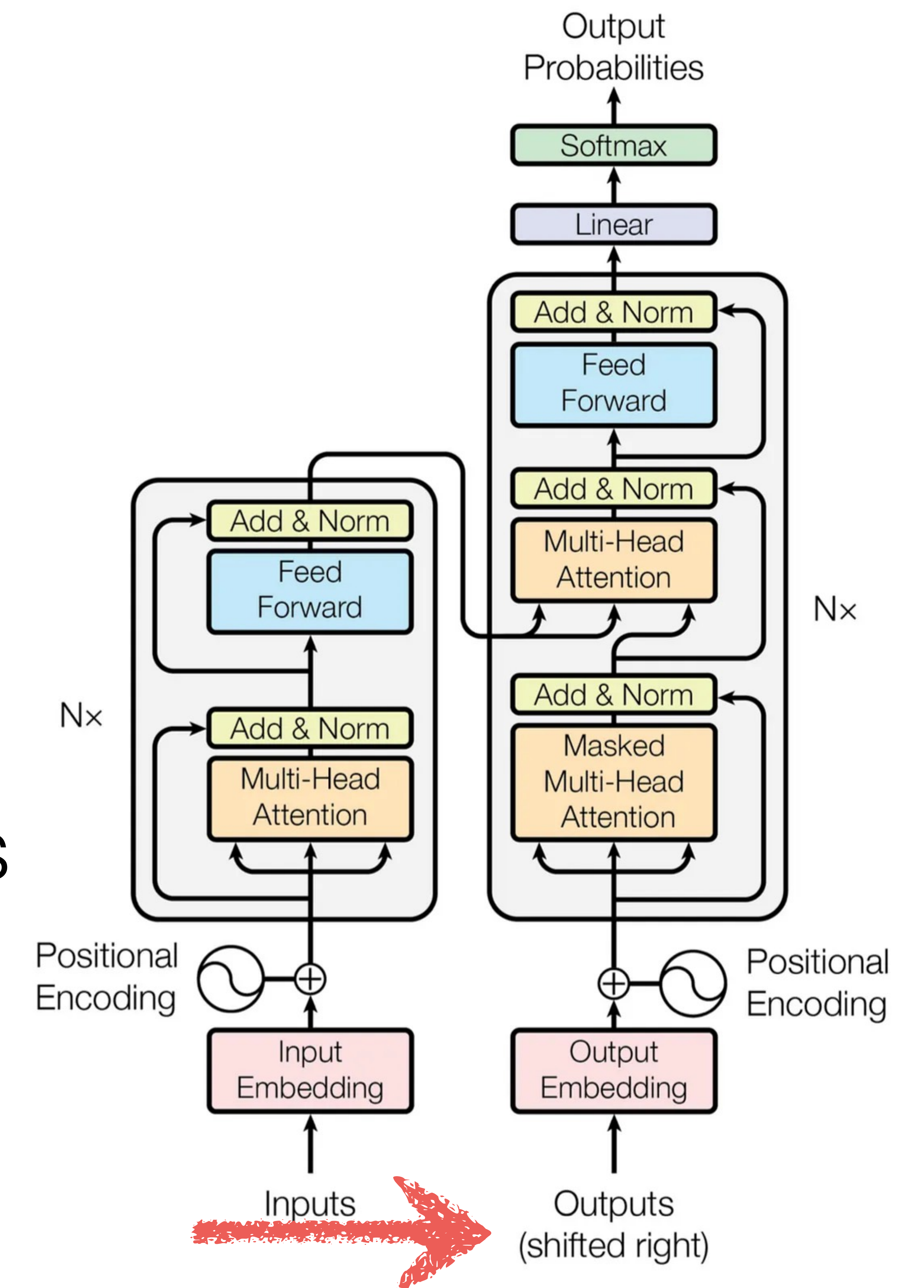
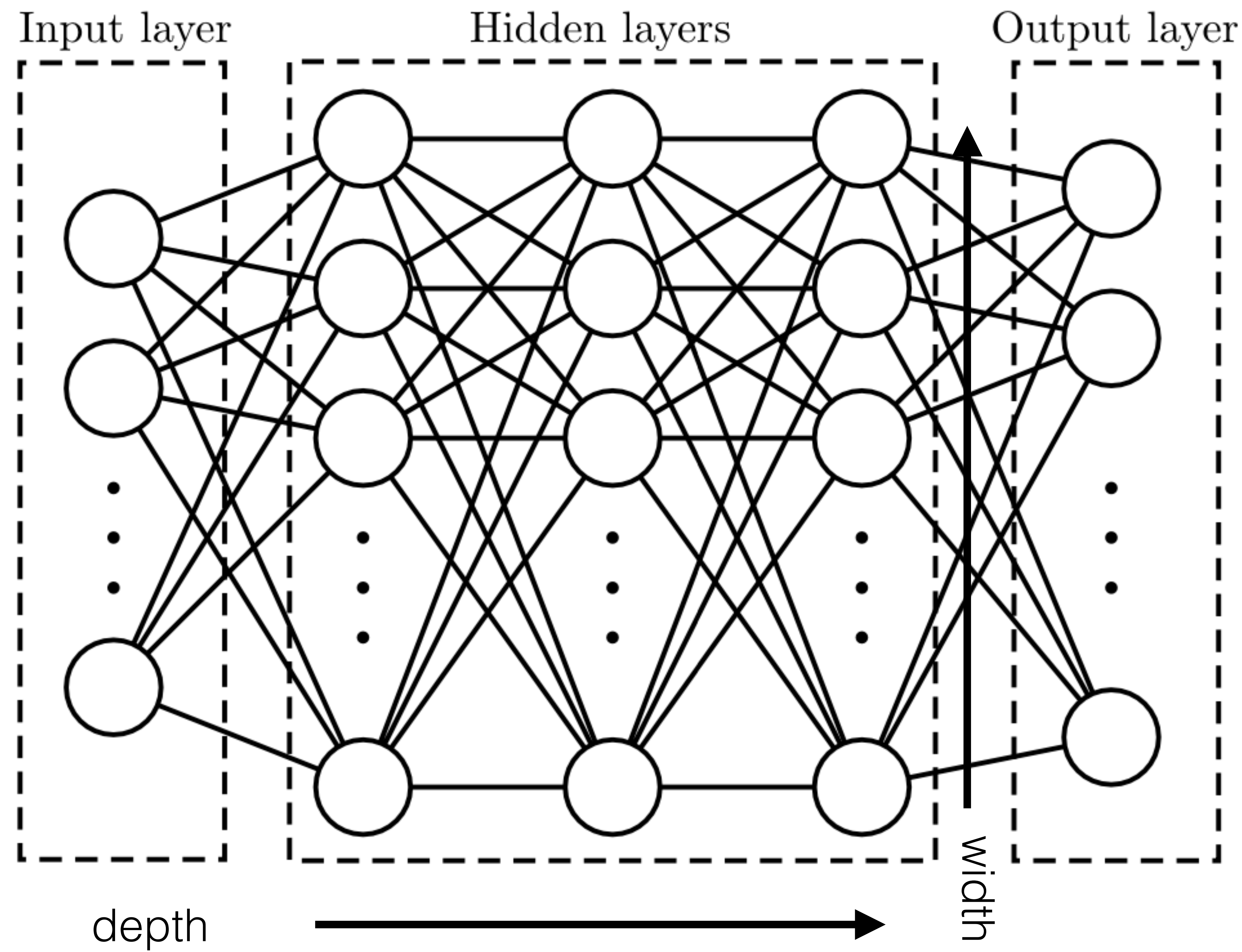
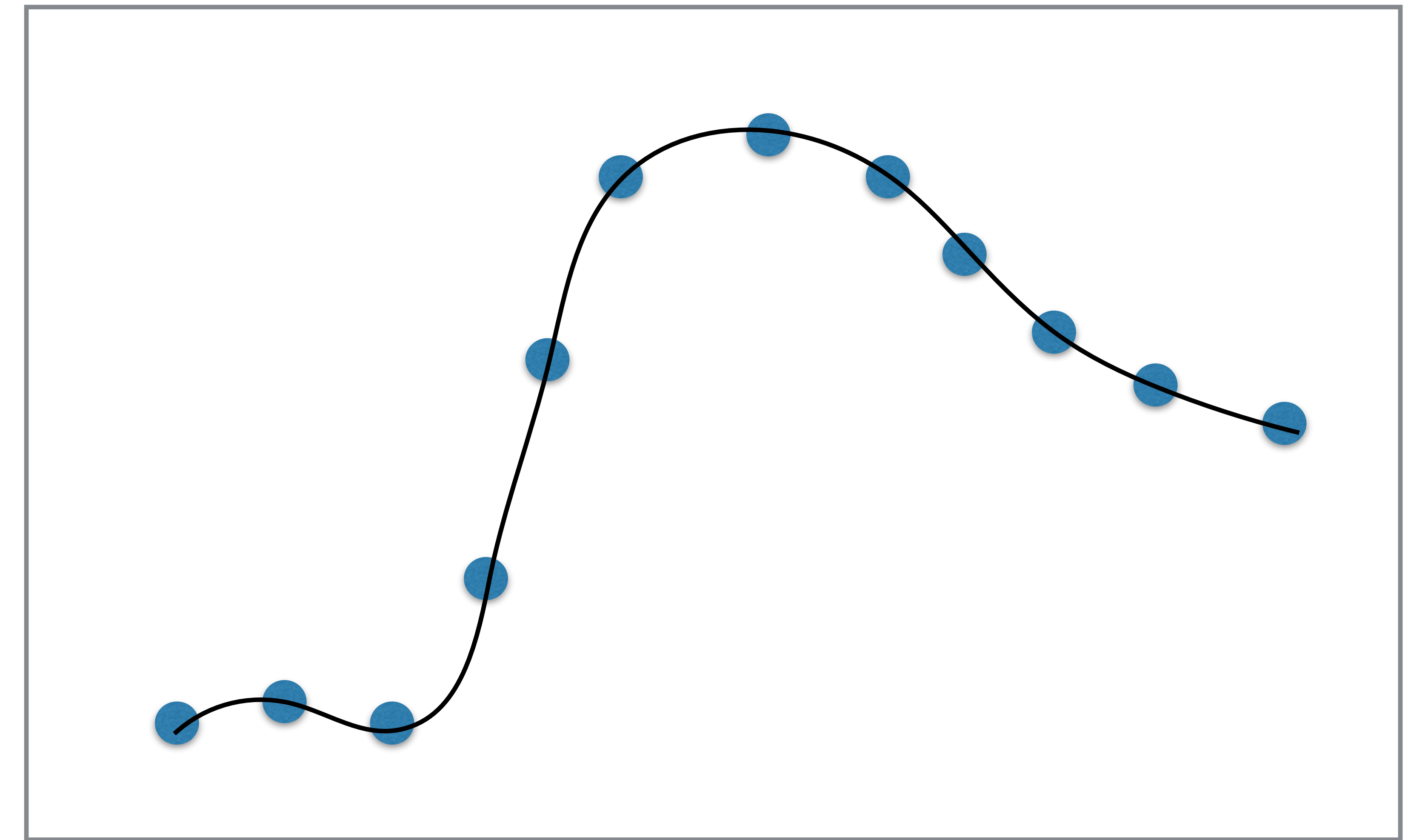
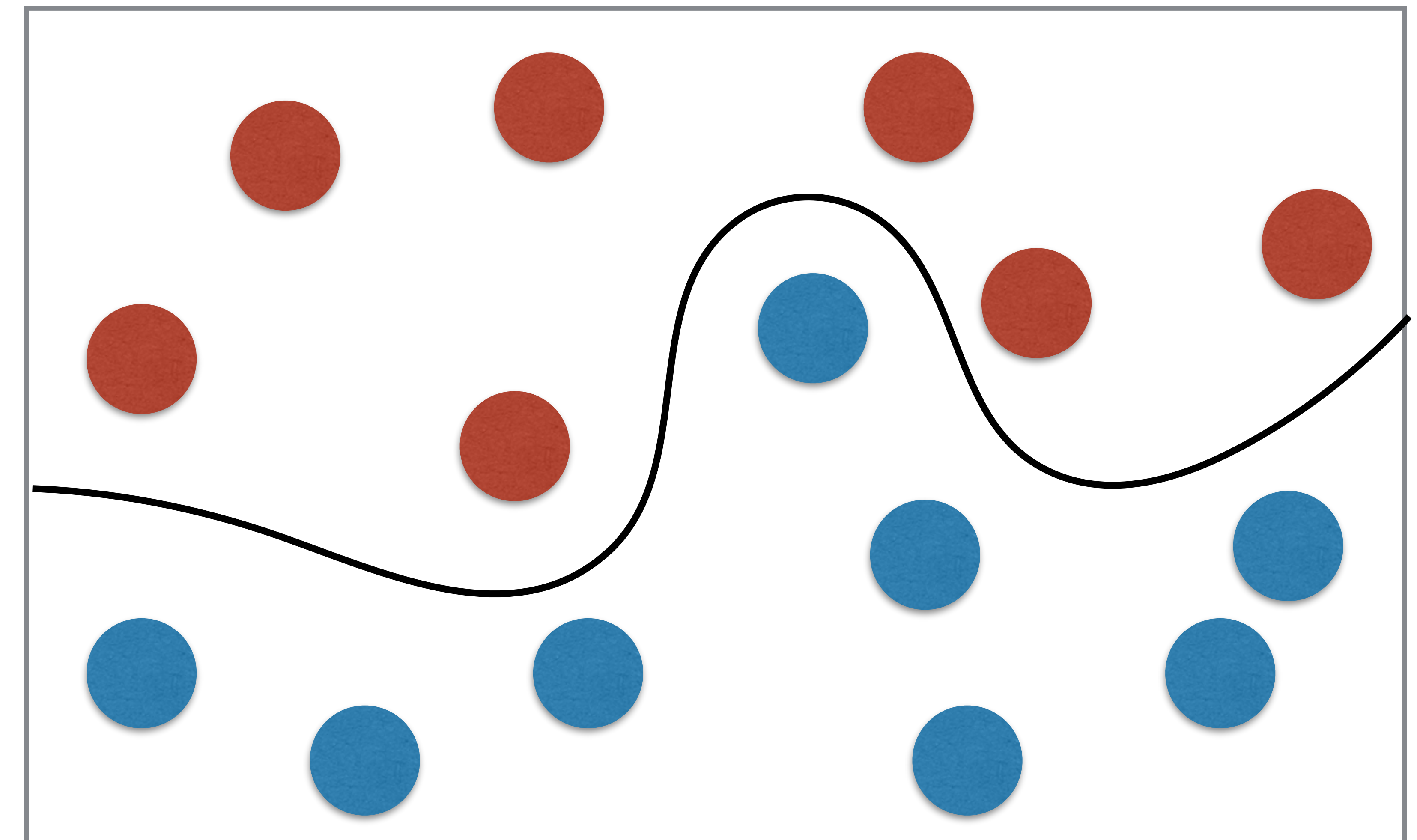


Figure 1: The Transformer - model architecture.

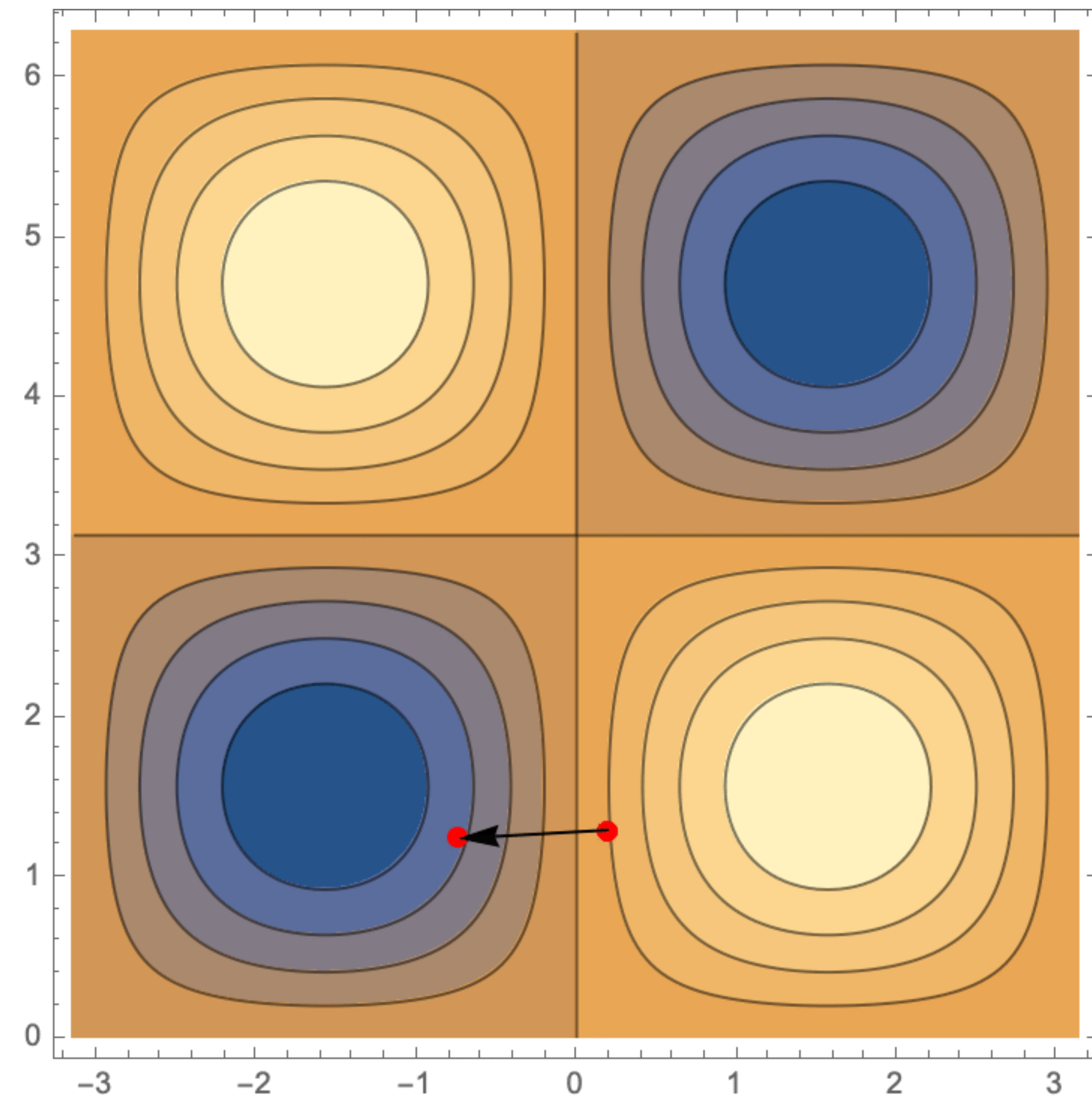
Recap - NN for regression and classification







Recap - Training with GD



GD parameter update

$$\theta^{(i)} \rightarrow \theta^{(i)} - \alpha \frac{\partial L}{\partial \theta^{(i)}}$$

Chaotic phase

Ordered phase

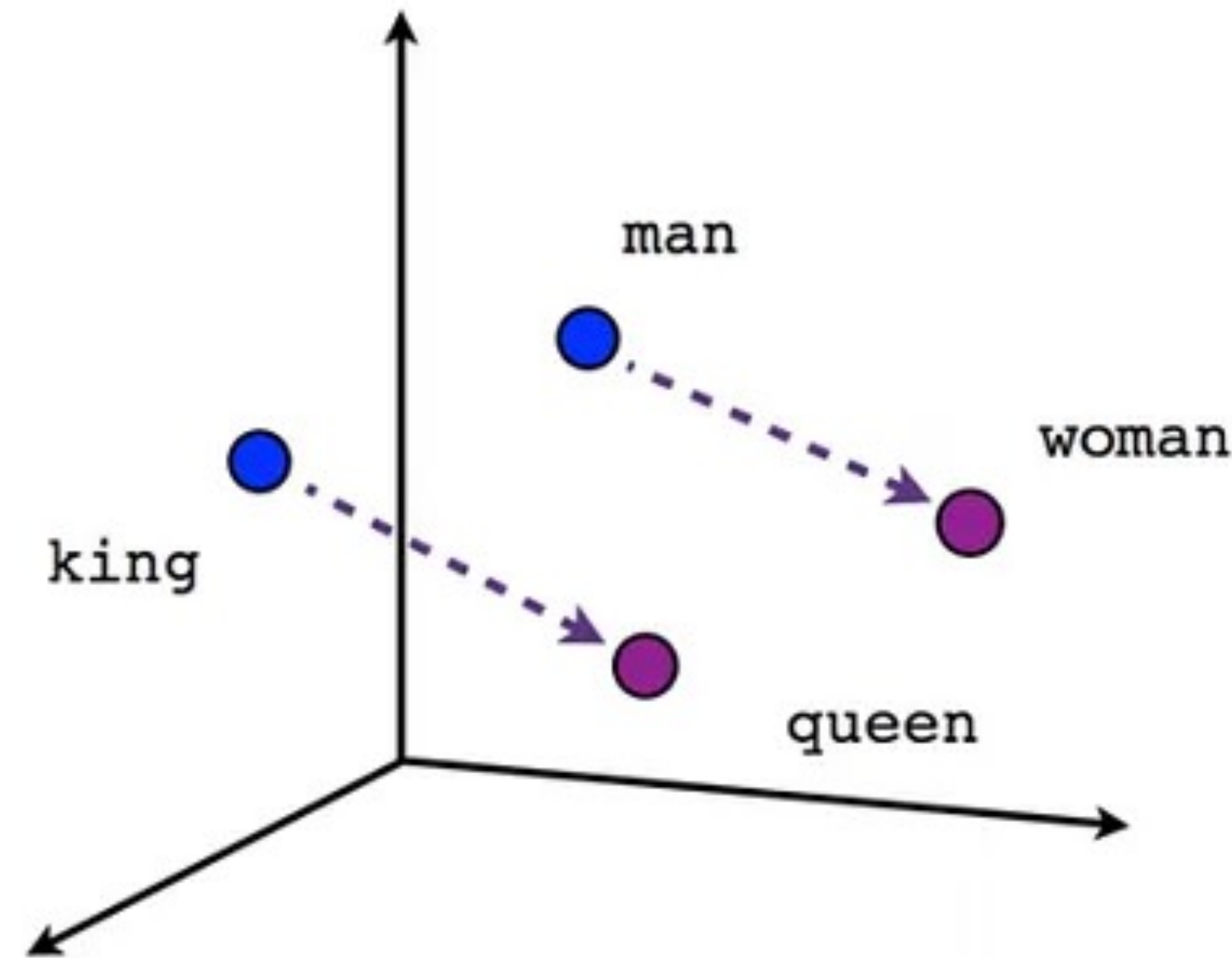
σ_w

σ_b

4.0 3.5 3.0 2.5 2.0 1.5 1.0 0.5 0.0

0 1 2 3 4 5

Recap - Transformer



Male-Female

Learns semantics

"queen - woman + man = king"

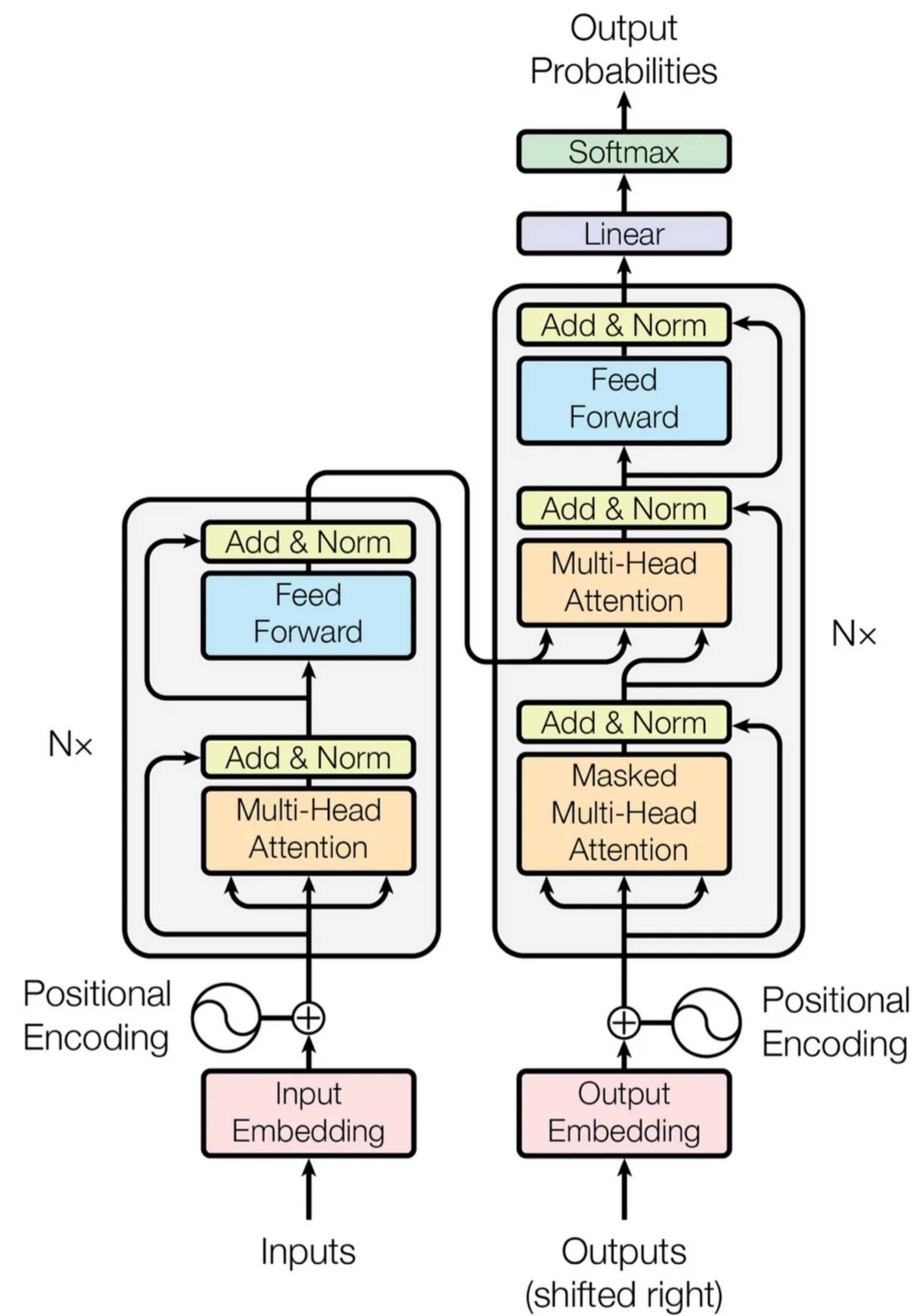
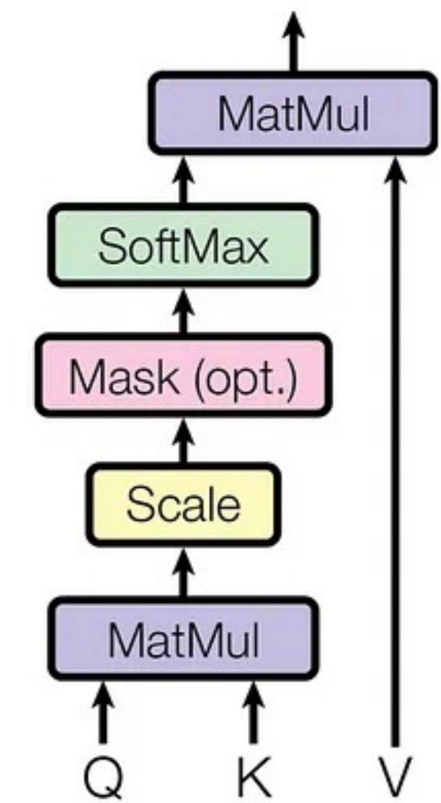
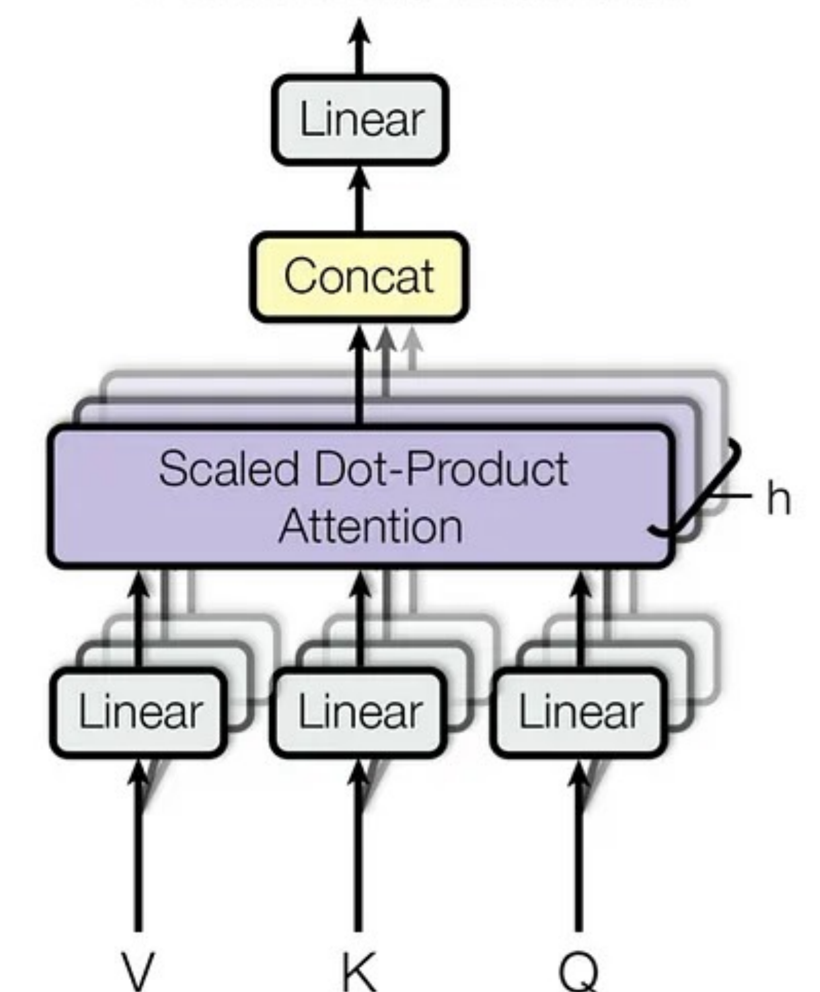
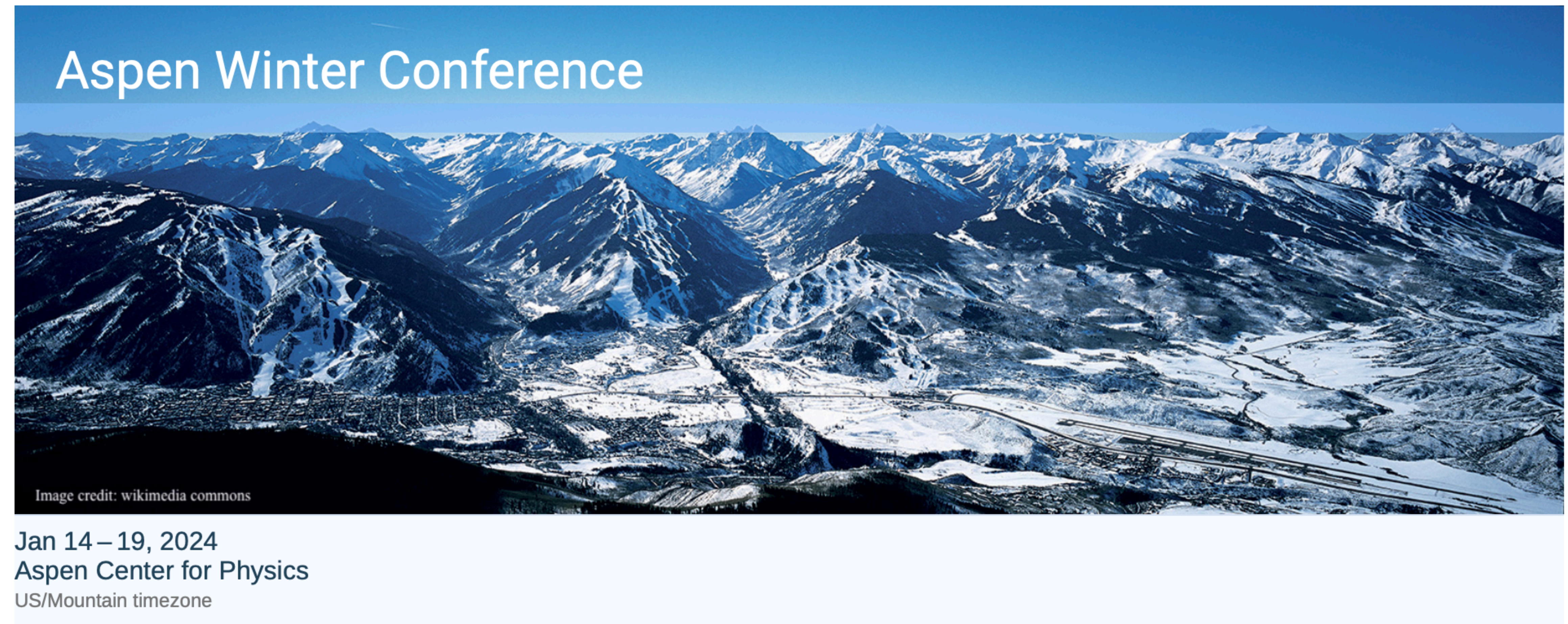


Figure 1: The Transformer - model architecture.





... want to learn more?



Overview

Timetable

Registration and Accommodation

Travel and Visas

Local information, Transportation, Skiing

Block Award

Poster

Fields, Strings, and Deep Learning

Progress in deep learning has traditionally involved experimental data, but in recent years it has impacted our understanding of formal structures arising in theoretical high energy physics and pure mathematics, via both theoretical and applied deep learning. This conference will bring together high energy theorists, mathematicians, and computer scientists across a broad variety of topics at the interface of these fields. Featured topics include the interface of neural network theory with quantum field theory, lattice field theory, conformal field theory, and the renormalization group; theoretical physics for AI, including equivariant, diffusion, and other generative models; ML for pure mathematics, including knot theory and special holonomy metrics, and deep learning for applications in string theory and holography.

ML Meetings @ Caltech

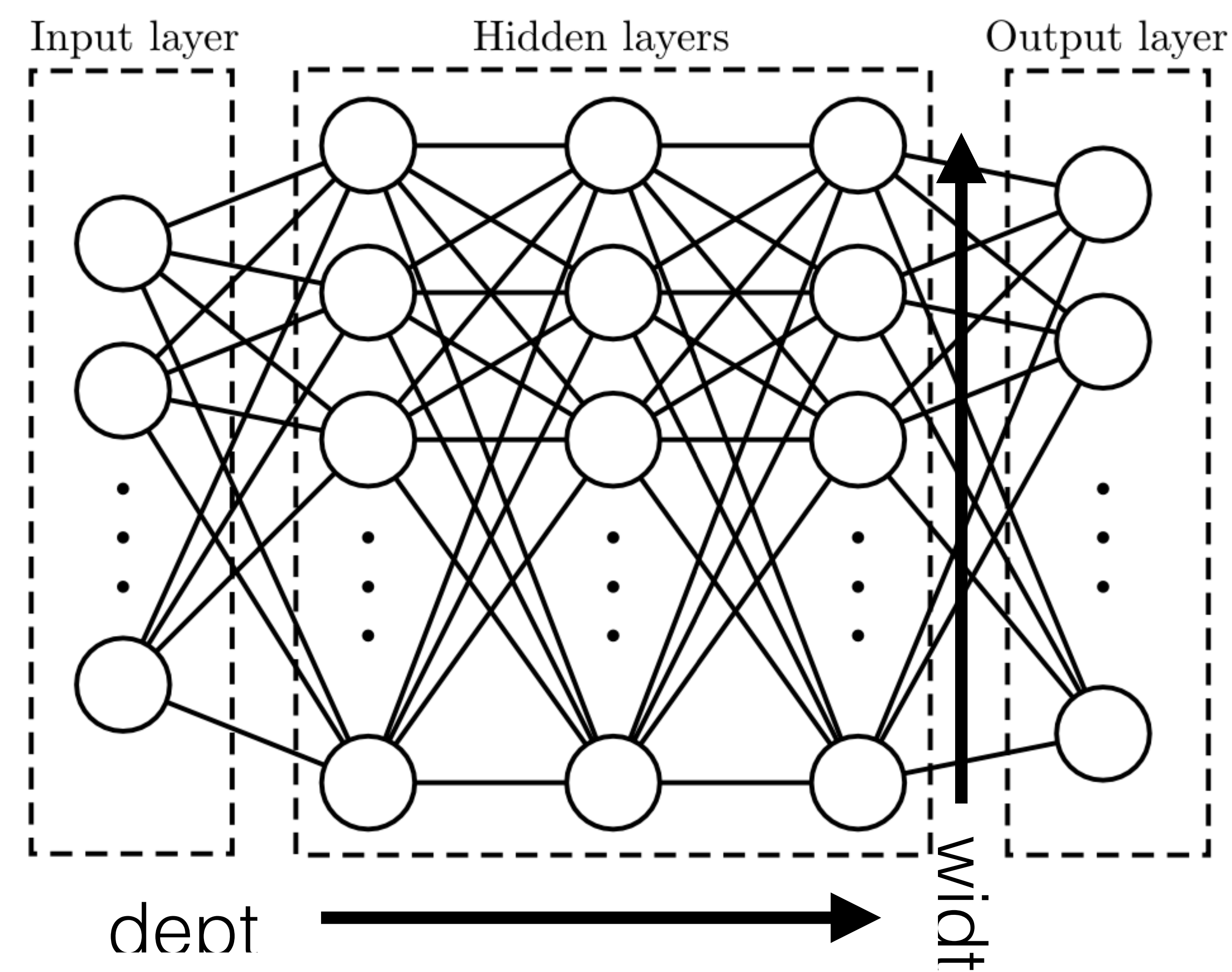
- Dec 10-12: Mathematics and ML 2023
- Dec 13-15: string_data 2023

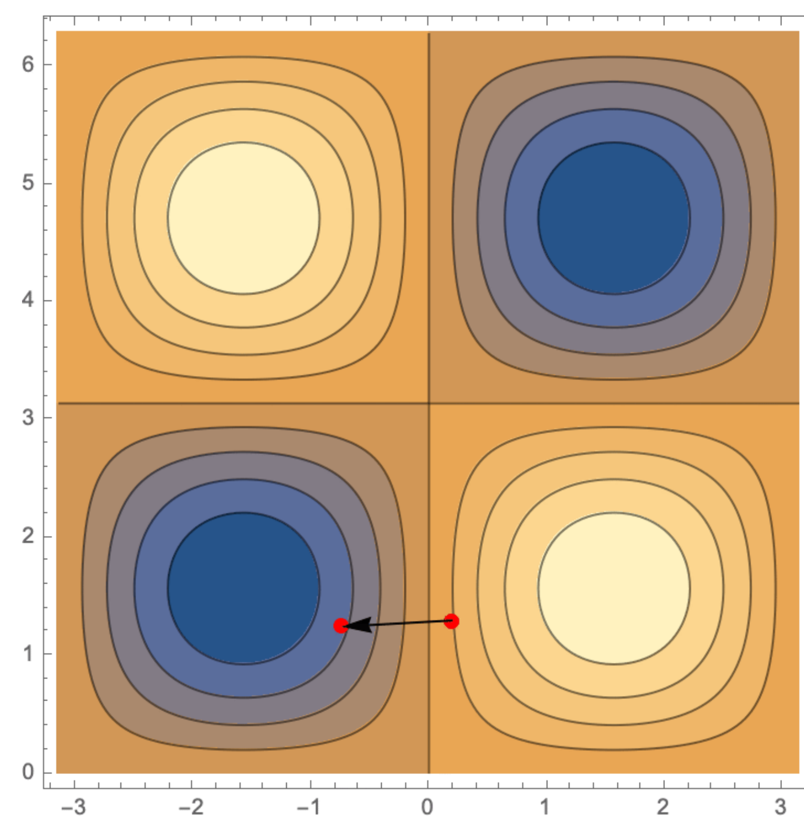
Aspen Winter Conference

- Jan 14-19: Fields, Strings, and Deep Learning

Application deadline: Aug 31

Thank you - Questions?





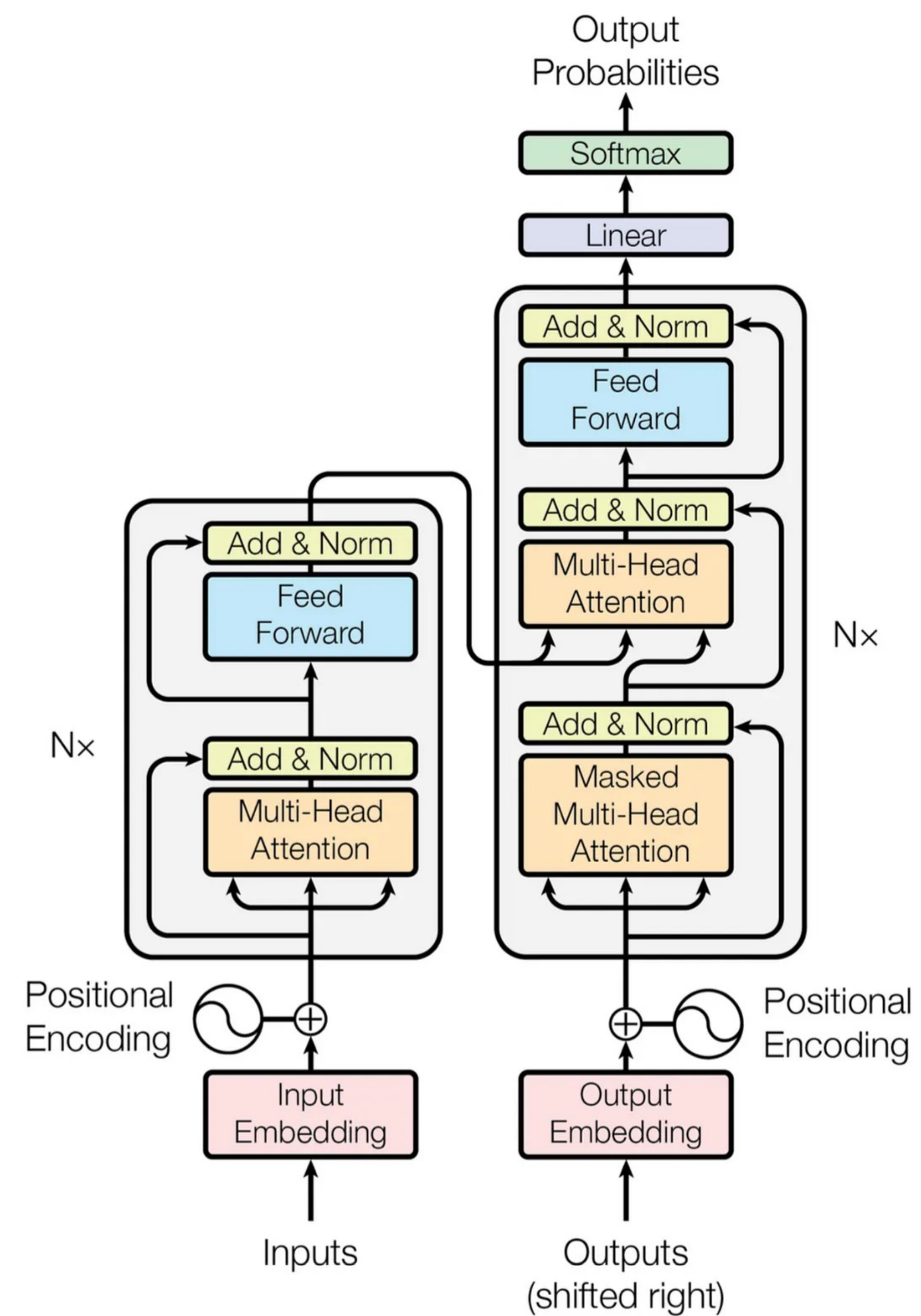
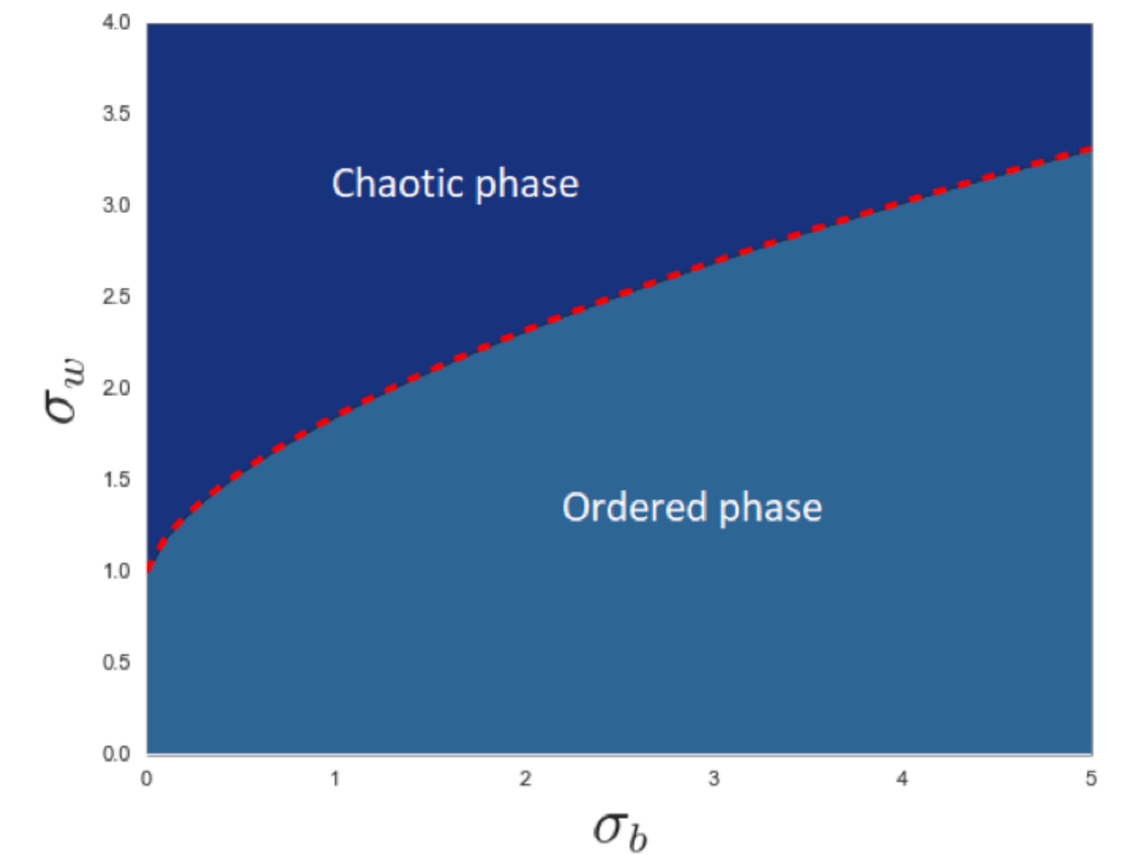
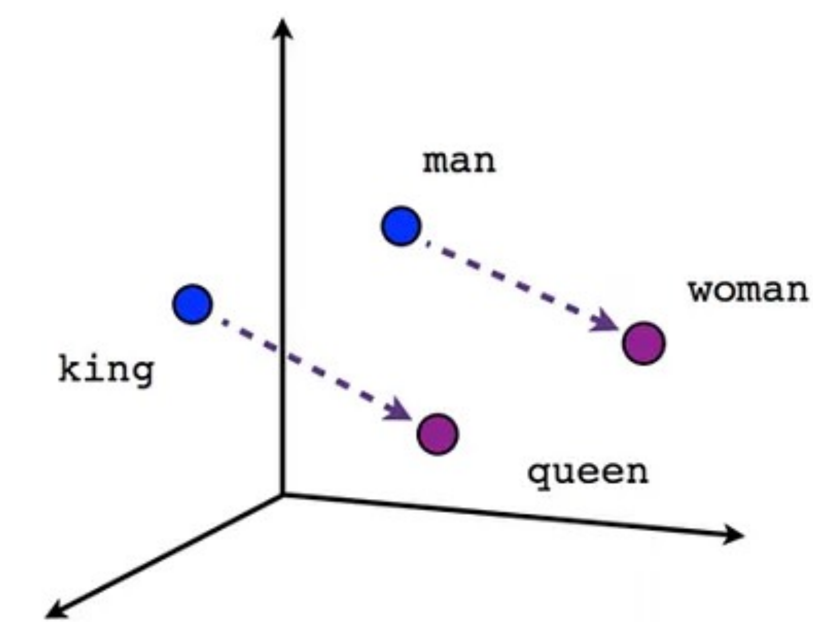


Figure 1: The Transformer - model architecture.





Male-Female