COMP 554 / CSDS 553 Advanced NLP

Faizad Ullah

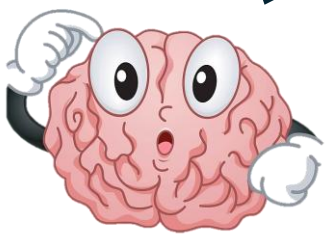
Language

- Language is an inherently temporal phenomenon.

- Spoken language is a sequence of acoustic events over time, and we comprehend and produce both spoken and written language as a sequential input stream.

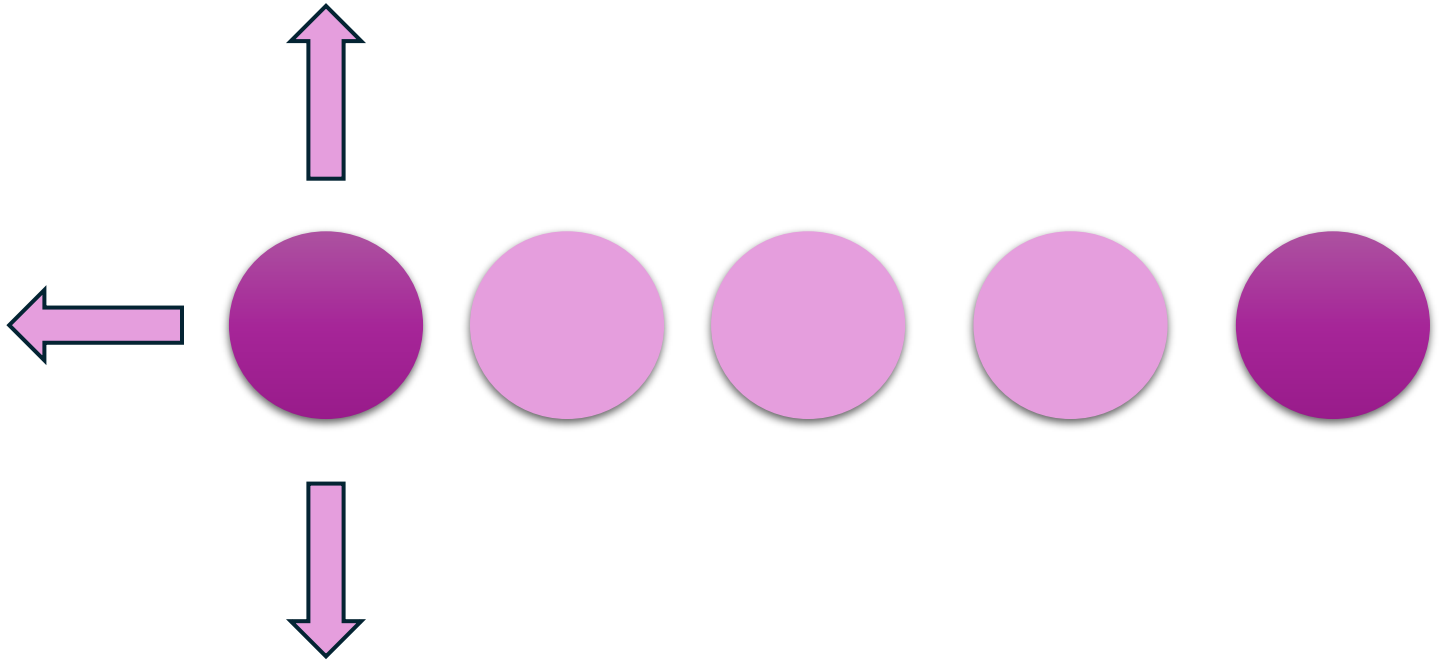
- The temporal nature of language is reflected in the metaphors we use; we talk of the flow of conversations, news feeds, and twitter streams, all of which emphasize that language is a sequence that unfolds in time.

Most of the machine learning approaches we've studied so far, don't have this temporal nature – they assume simultaneous access to all aspects of their input.



Sequential Data and Context

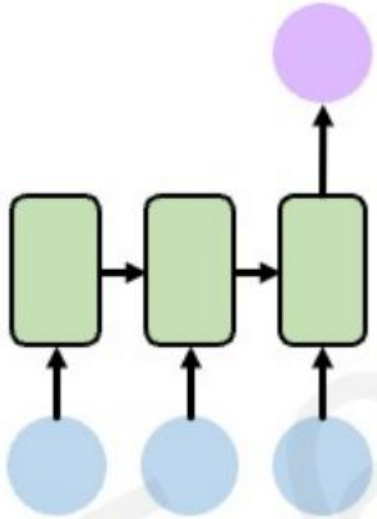
Test is an example of sequential data



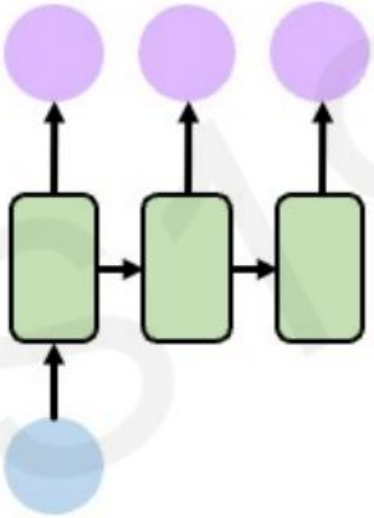
Sequence Modeling Applications

One to One
Binary Classification

"Will I pass this class?"
Student → Pass?

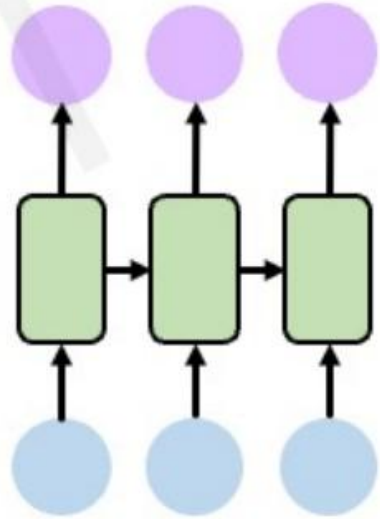


Many to One
Sentiment Classification



One to Many
Image Captioning

"A baseball player throws a ball."

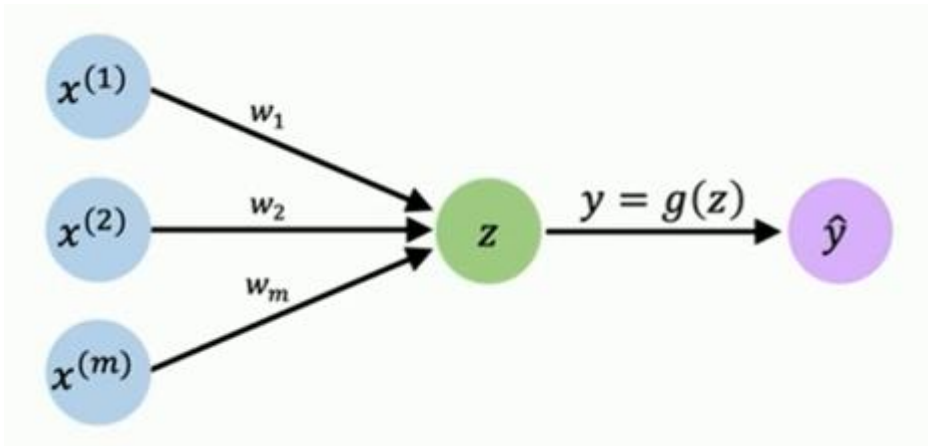


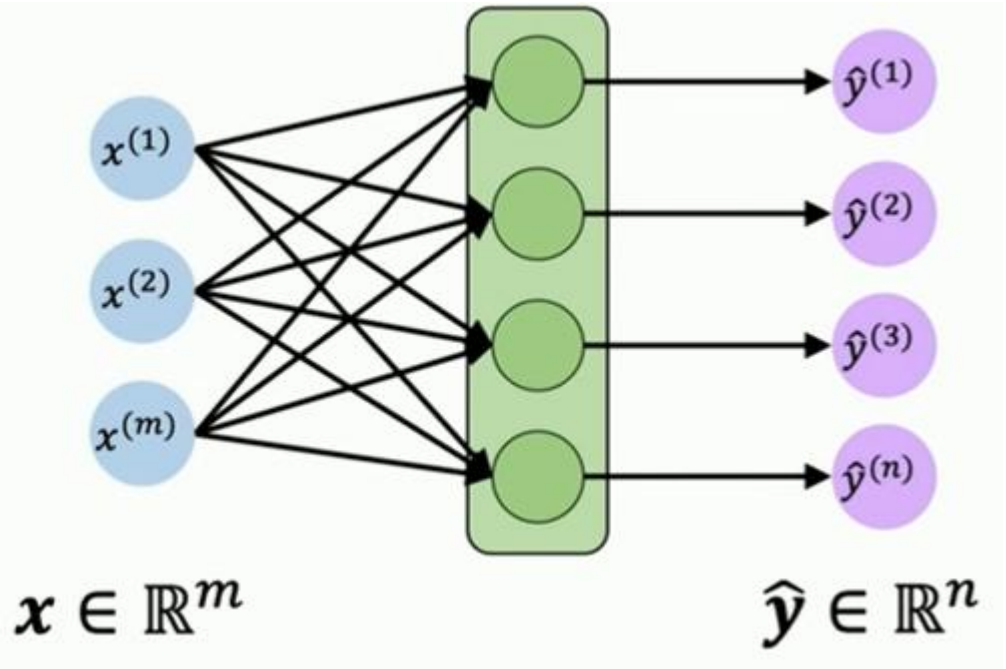
Many to Many
Machine Translation

Examples

- Image Classification:
  - one to one
- Sentiment Classification:
  - many to one
- Image Captioning:
  - one to many
- Video Captioning:
  - many to many
- Machine Translation:
  - many to many
- Image Generation Models (like Stable Diffusion):
  - many to one
- Language Models (like GPT-4):
  - many to many

Perceptron





Simple Neural Networks are Fixed-Length input/output devices, This severely limits the ability to account for context

Problems

- Fixed length input/output models
- No explicit notion of word order (no temporal dimension)
- Sentences are represented the same way in whatever order
  - “The food was good, not bad at all”
  - “The food was bad, not good at all”
- “Thinking from scratch” at every time step
- Understanding of a sequence should evolve as we read more words
- Not sharing their parameters “across time”
- Not adaptive to the dynamic nature of meaning in sequences

Sequential Data

- Words in sentences
- Characters in words
- Sentences in discourse
- …

Long-Distance Dependencies

- **He** does not have very much confidence in **himself**
- **She** does not have very much confidence in **herself**

- Selectional Preferences
  - The **reign** has lasted as long as the life of the **queen**
  - The **rain** has lasted as long as the life of the **clouds**

Long-Distance Dependencies

- What is the referent of “it”?

- The trophy would not fit in the brown suitcase because it was too big.

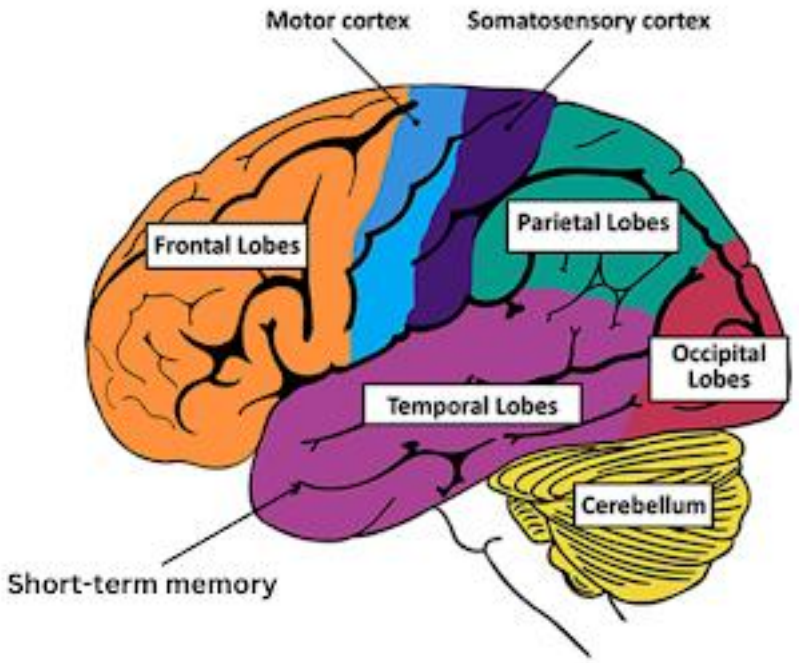
?

- The trophy would not fit in the brown suitcase because it was too small.

?

Recurrent Neural Networks

Intuition


Motor cortex
Somatosensory cortex
Frontal Lobes
Parietal Lobes
Occipital Lobes
Temporal Lobes
Cerebellum
Short-term memory

Intuition

- Recurrent Neural Networks are inspired by a part of the brain called the Temporal Lobe which resides in short-term memory.
- Short-term memory is the part of the brain that helps us to remember what happened around us recently, It stores information temporarily.
- Contributes a lot to our ability to handle languages, understand patterns, and understand what is going around us recently.
- RNNs allow the network to capture short-term dependencies in the given dataset.
- Useful for handling sequential temporary information that can be used for different tasks like Language Modelling, Speech Recognition, Machine Translation, and more.

Recurrent Neural Networks

- The recurrent network offers a new way to represent the prior context, in its recurrent connections.

- It allows the model's decision to depend on information from hundreds of words in the past.

- A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

Recurrent Neural Networks

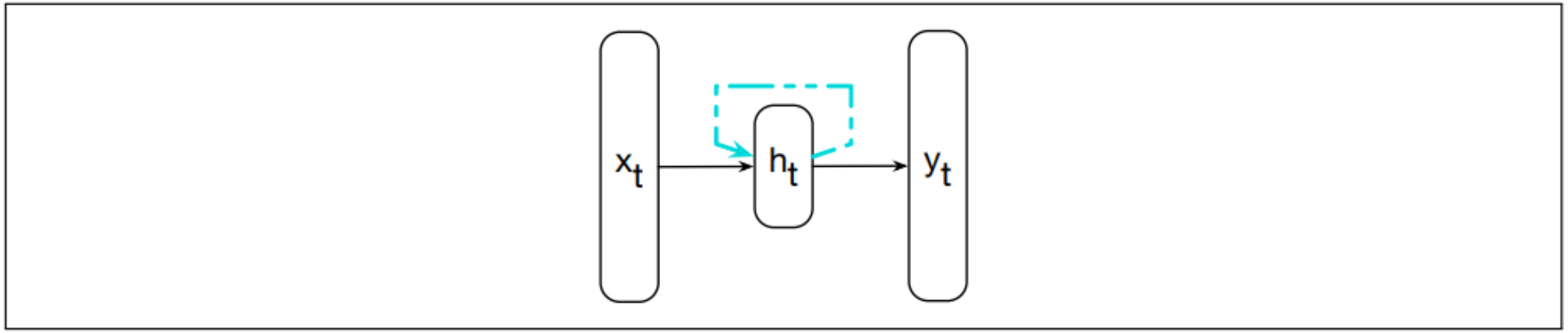
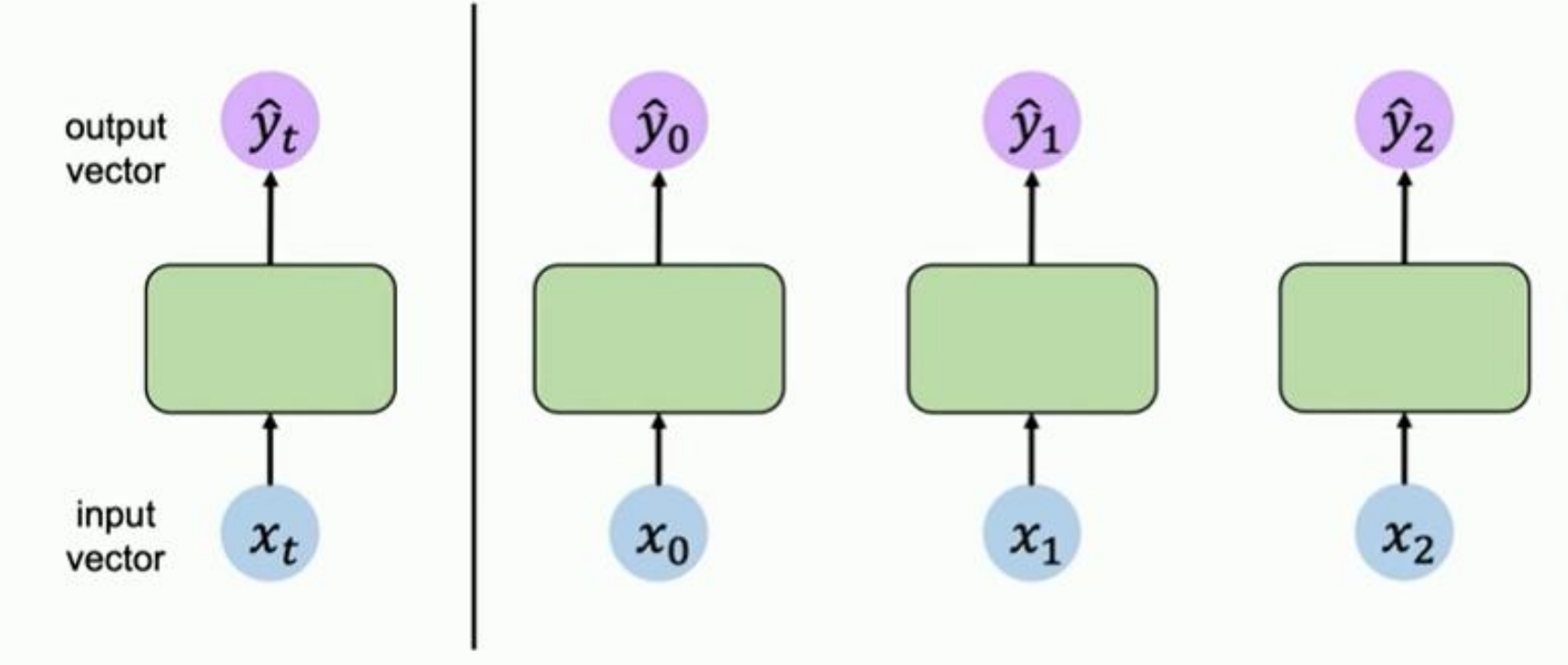
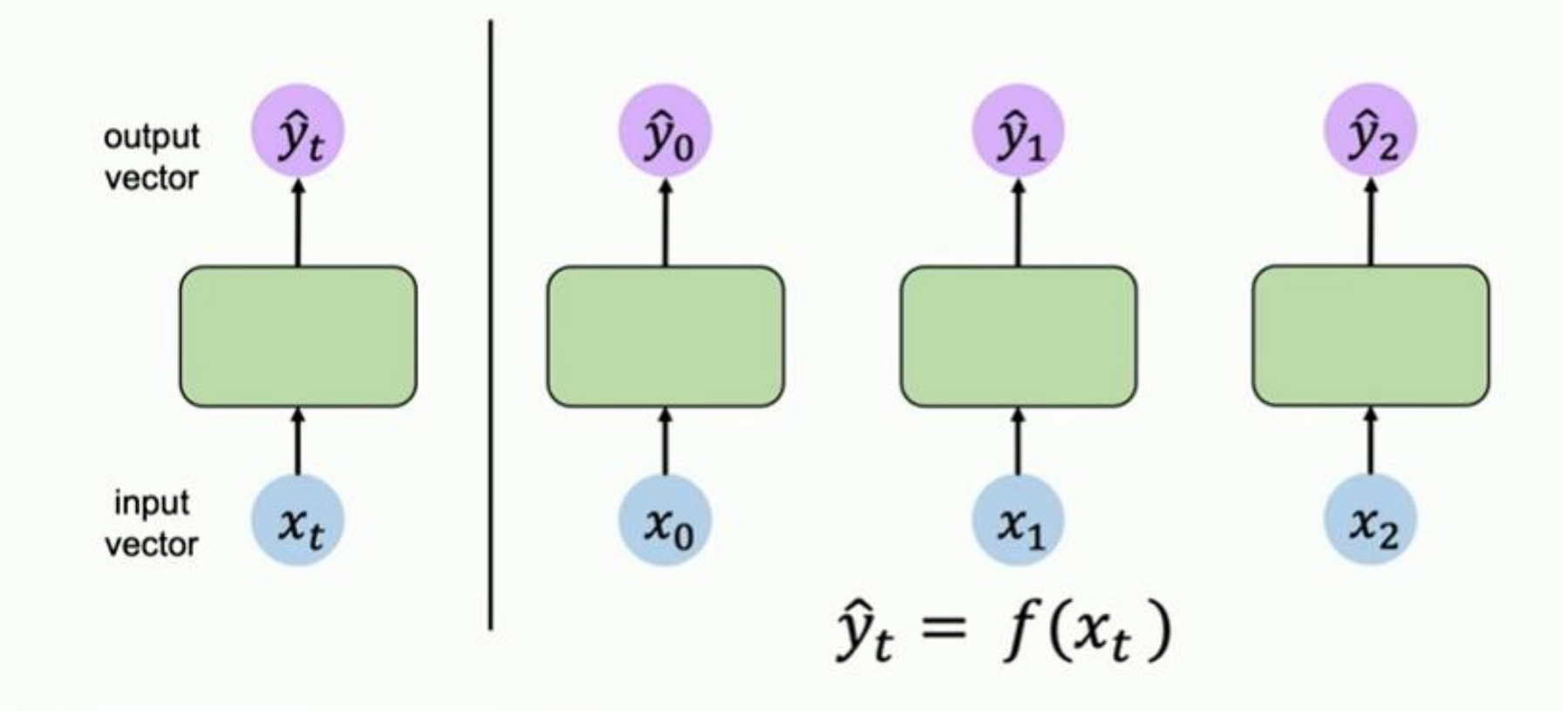


Figure 8.1 Simple recurrent neural network after Elman (1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

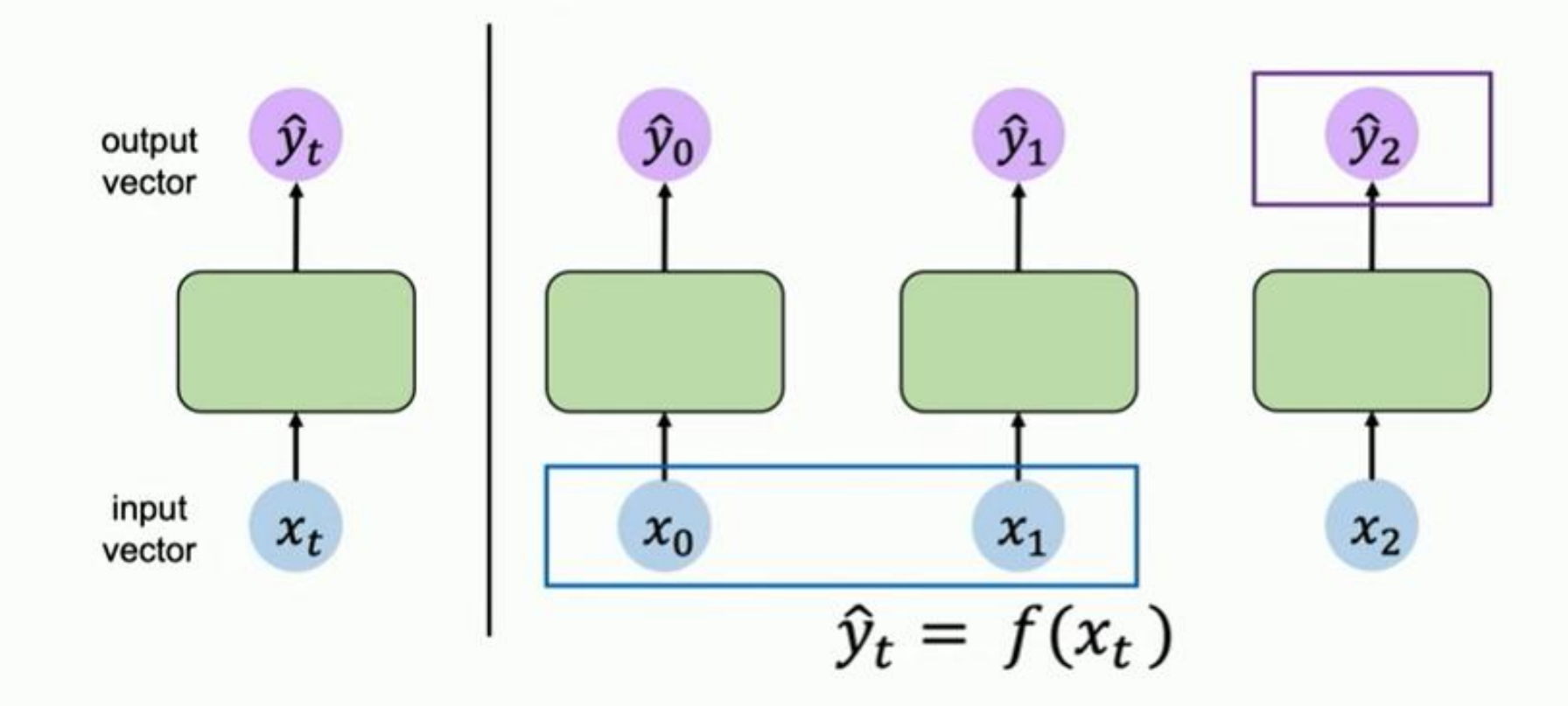
Introducing Time in Neural Networks



Introducing Time in Neural Networks



Introducing Time in Neural Networks



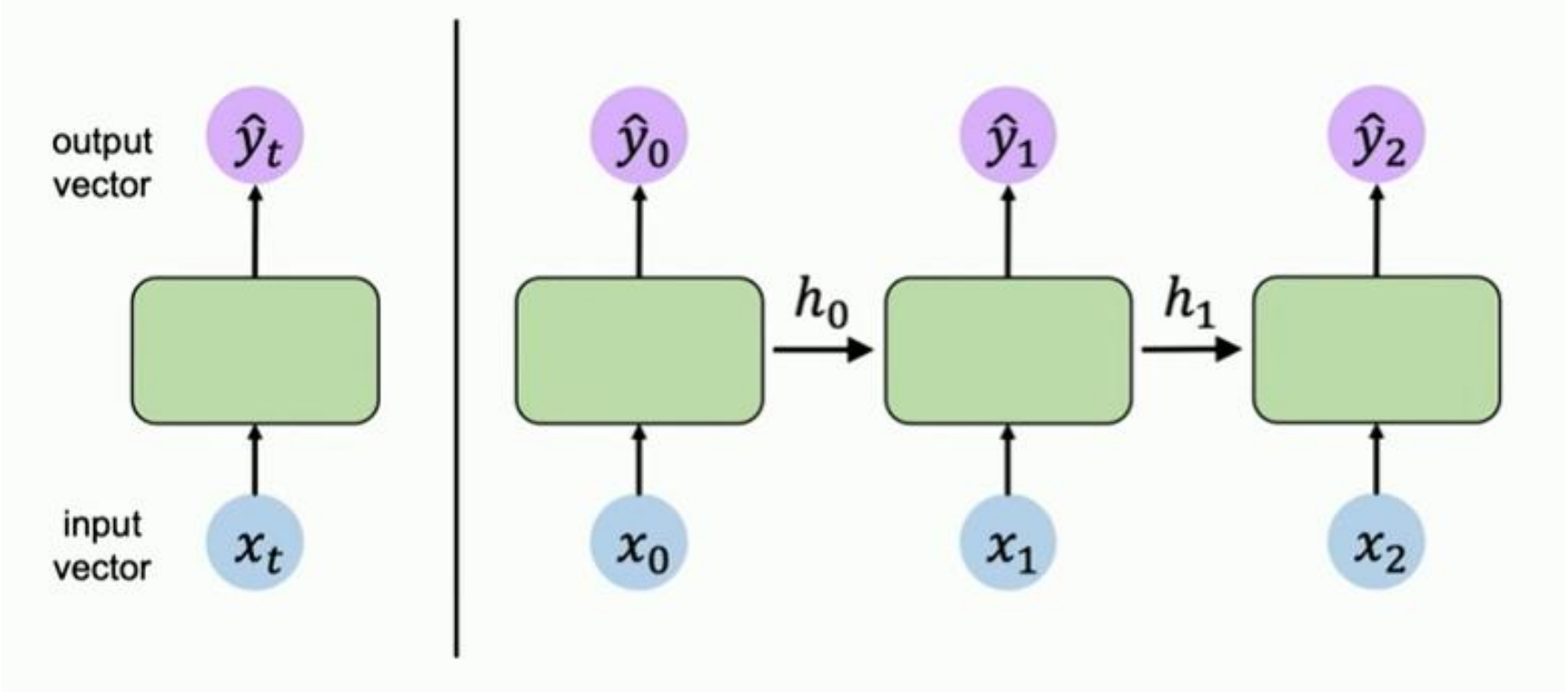
Recurrent Neural Networks

- As with ordinary feedforward networks, an input vector x_t, is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units.
- This hidden layer is then used to calculate a corresponding output, y_t.
- In a departure from our earlier window-based approach, sequences are processed by presenting one item at a time to the network.
- We'll use subscripts to represent time, thus x_t will mean the input vector x at time t.
- The key difference from a feedforward network lies in the recurrent link.
- This link augments the input to the computation at the hidden layer with the value of the hidden layer from the preceding point in time.

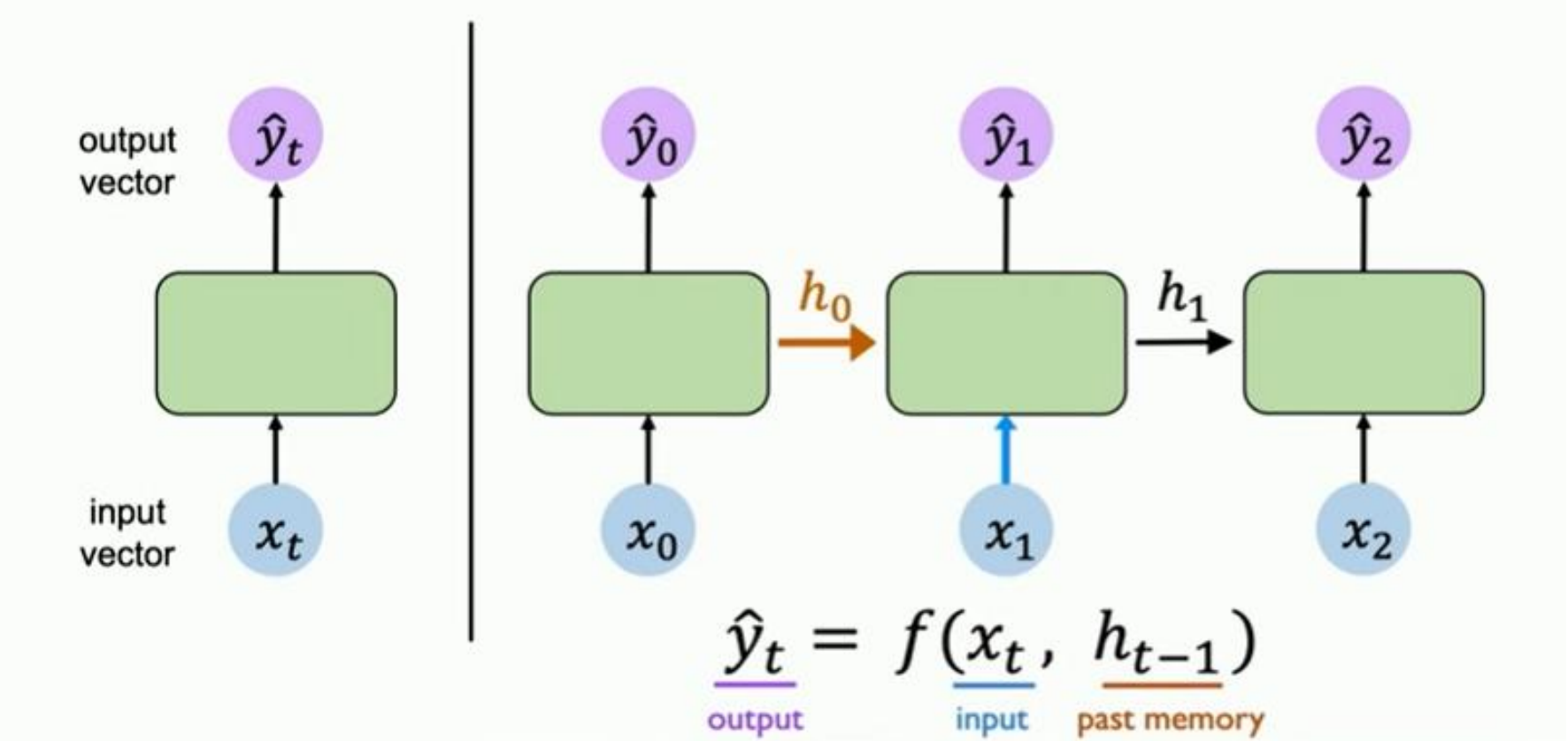
Recurrent Neural Networks

- The hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time.
- The most significant change lies in the new set of weights, U, that connect the hidden layer from the previous time step to the current hidden layer.
- These weights determine how the network makes use of past context in calculating the output for the current input.
- As with the other weights in the network, these connections are trained via backpropagation.

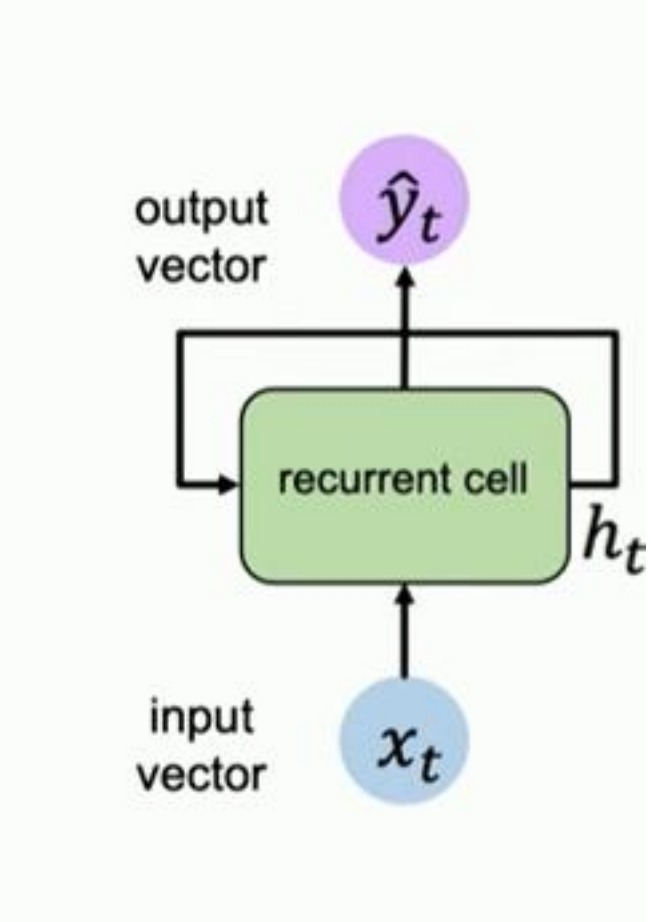
Hidden States



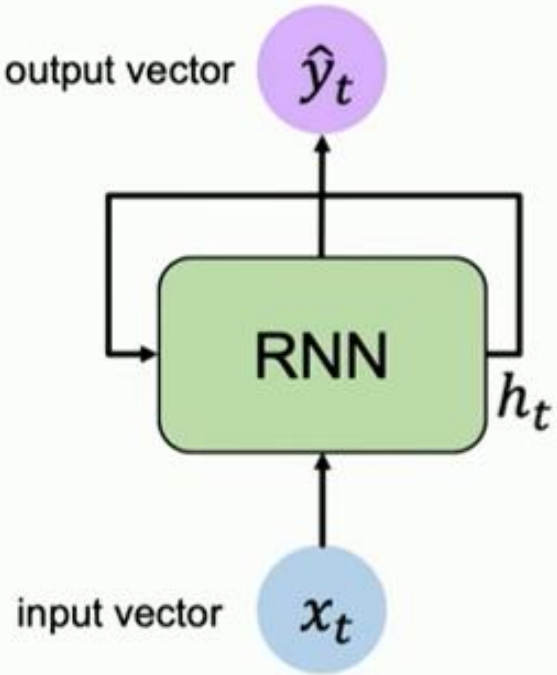
Hidden States



Recurrence



Recurrence



Apply a **recurrence relation** at every time step to process a sequence:

$$h_t = f_W(x_t, h_{t-1})$$

cell state

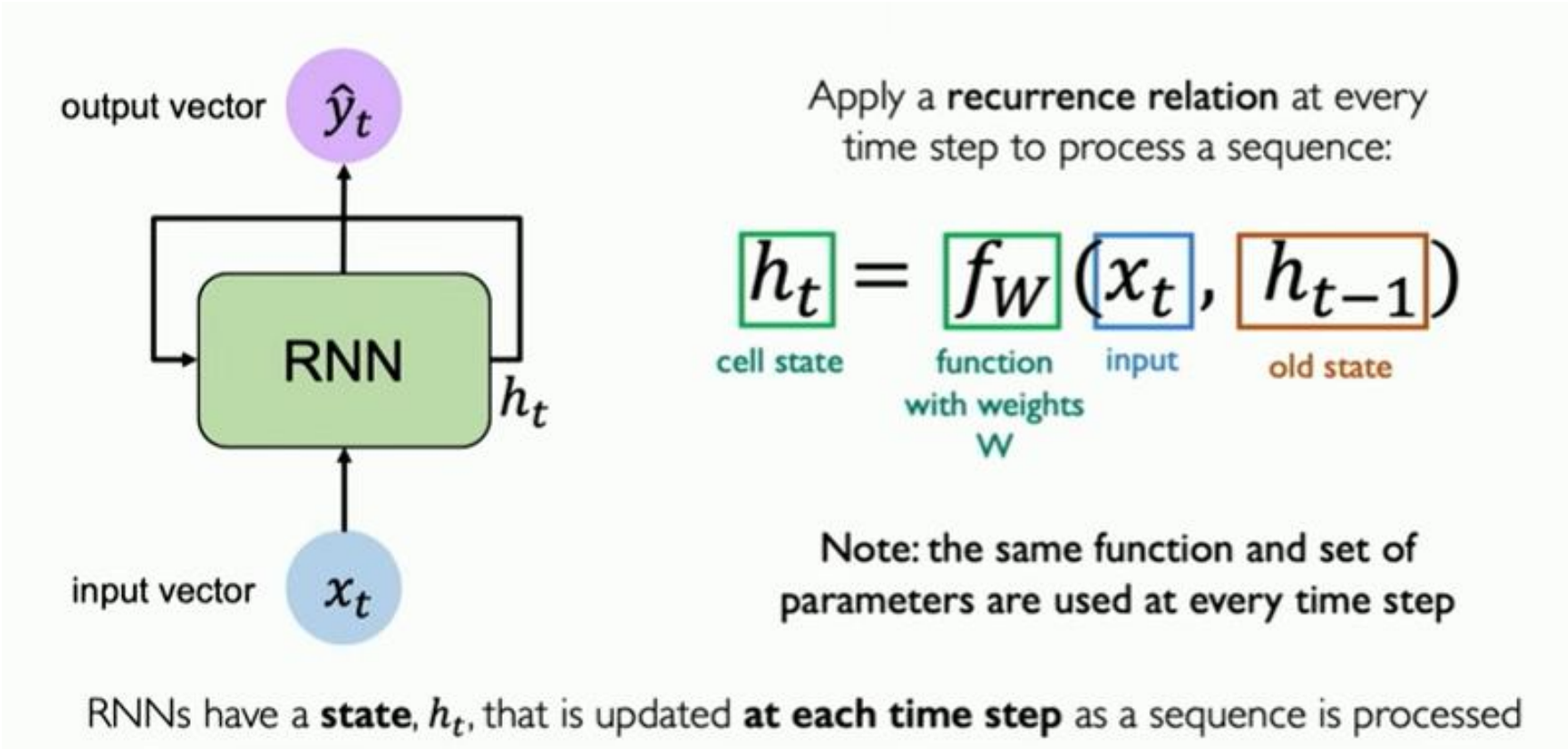
function with weights W

input

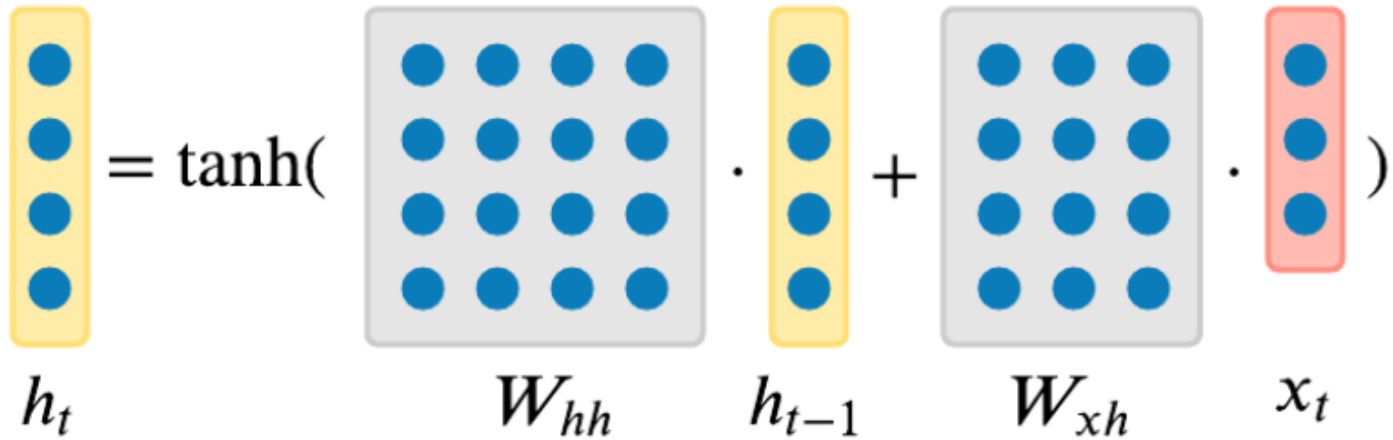
old state

RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

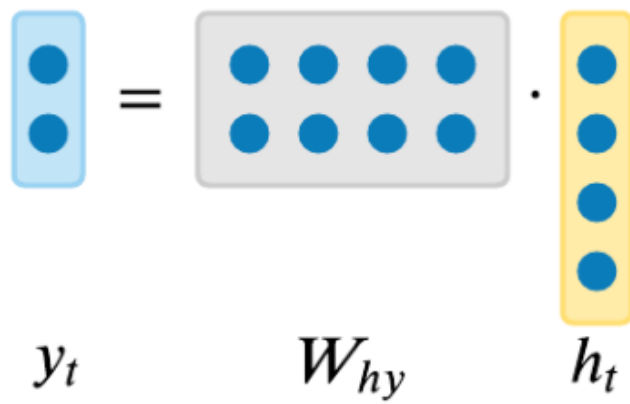
Recurrence


output vector
$\hat{y}_t$
RNN
h_t
input vector
x_t
Apply a **recurrence relation** at every time step to process a sequence:
$h_t = f_W(x_t, h_{t-1})$
cell state
function with weights W
input
old state
Note: the same function and set of parameters are used at every time step
RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



$$y_t = W_{hy}h_t$$



Inference in RNNs

- Let's refer to the input, hidden and output layer dimensions as d_{in}, d_h, and d_{out} respectively.
- Given this, our three parameter matrices are:

$$\mathbf{W} \in \mathbb{R}^{d_h \times d_{in}}$$

$$\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$$

$$\mathbf{V} \in \mathbb{R}^{d_{out} \times d_h}$$

Recurrent Neural Networks

- As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks.

- There are 3 sets of weights to update:
  - W: input to hidden layer
  - U: previous hidden layer to current hidden layer
  - V: hidden layer to output layer

Recurrent Neural Networks

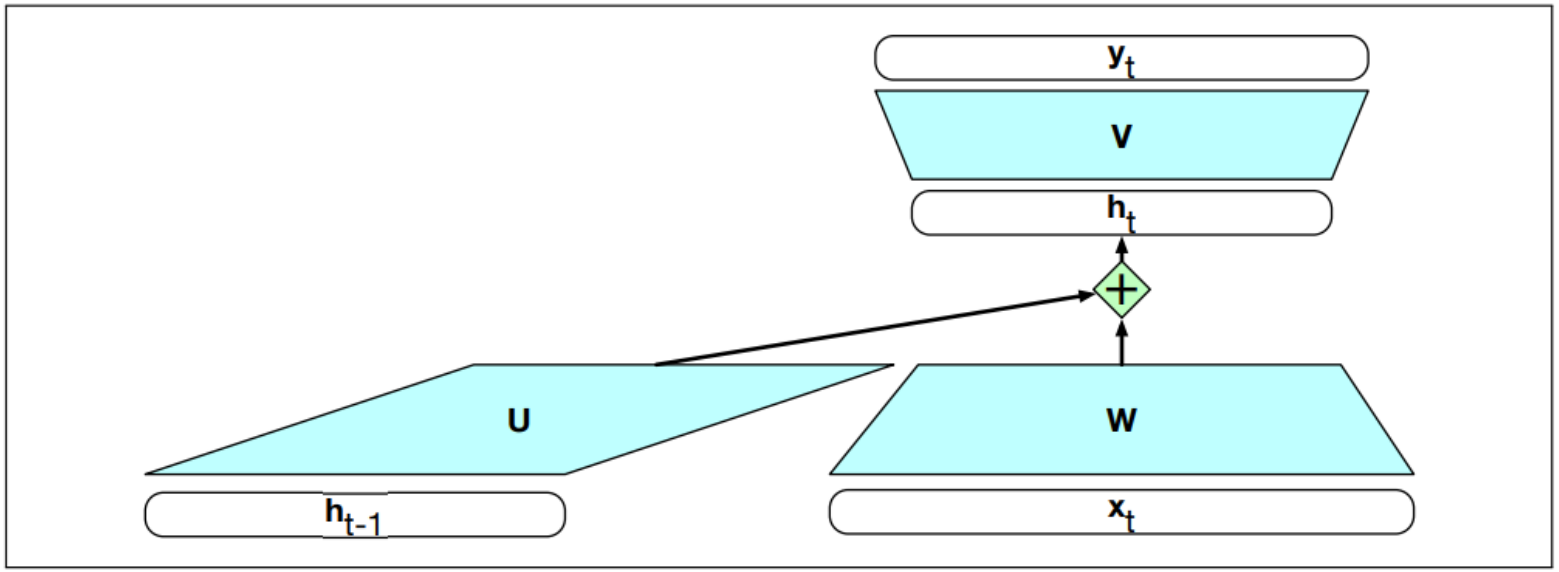


Figure 8.2 Simple recurrent neural network illustrated as a feedforward network. The hidden layer $\mathbf{h}_{t-1}$ from the prior time step is multiplied by weight matrix **U** and then added to the feedforward component from the current time step.

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$
$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

Inference in RNNs

- Forward inference maps a sequence of inputs to outputs, similar to feedforward networks.
- To compute y_t for x_t , we need the activation value for the hidden layer h_t.
- Calculate h_t by combining:
  - Current input x_t multiplied by weight matrix W
  - Previous hidden state h_{t-1} multiplied by weight matrix U
- Sum these and apply an activation function g to get h_t.
- Use h_t to compute the output y_t.

Inference in RNNs

- The fact that the computation at time t requires the value of the hidden layer from time t −1 mandates an incremental inference algorithm that proceeds from the start of the sequence to the end as illustrated in Fig. 8.3.

function FORWARDRNN(**x**, *network*) **returns** output sequence **y**

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(**x**) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Figure 8.3 Forward inference in a simple recurrent network. The matrices **U**, **V** and **W** are shared across time, while new values for **h** and **y** are calculated with each time step.

Inference in RNNs

The sequential nature of simple recurrent networks can also be seen by unrolling the network in time as is shown in Fig. 8.4.

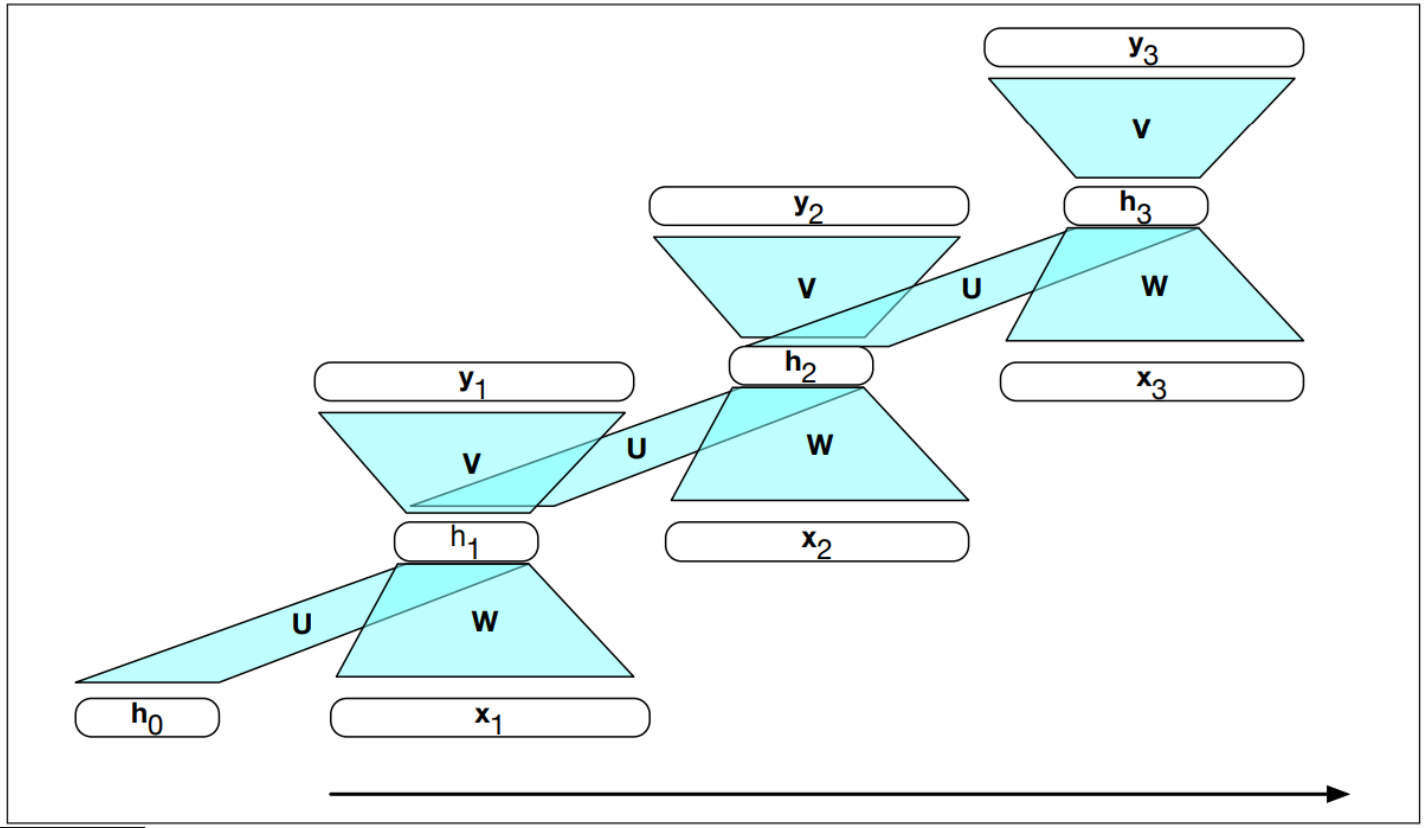
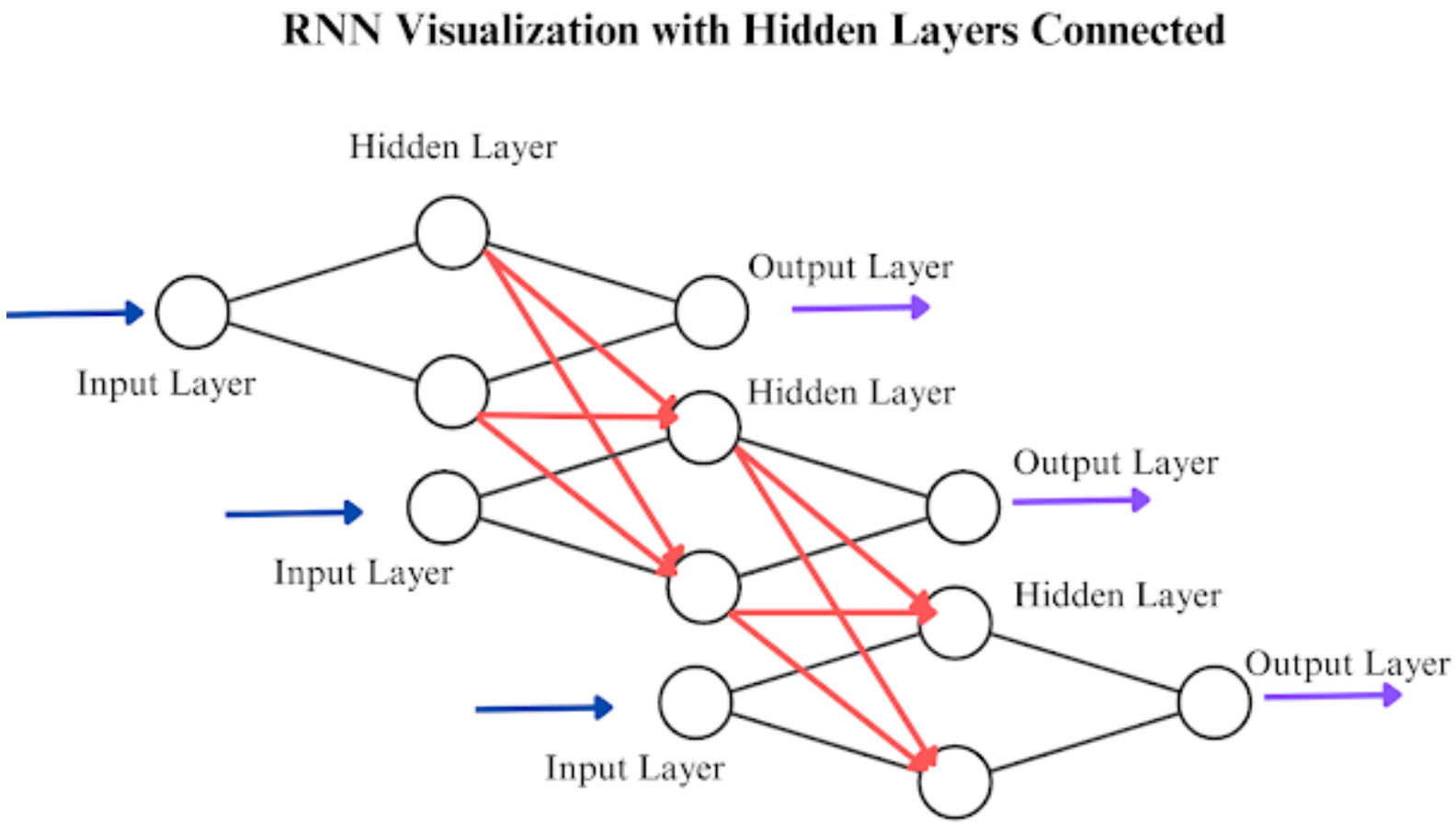


Figure 8.4 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared across all time steps.

Recurrent Neural Networks



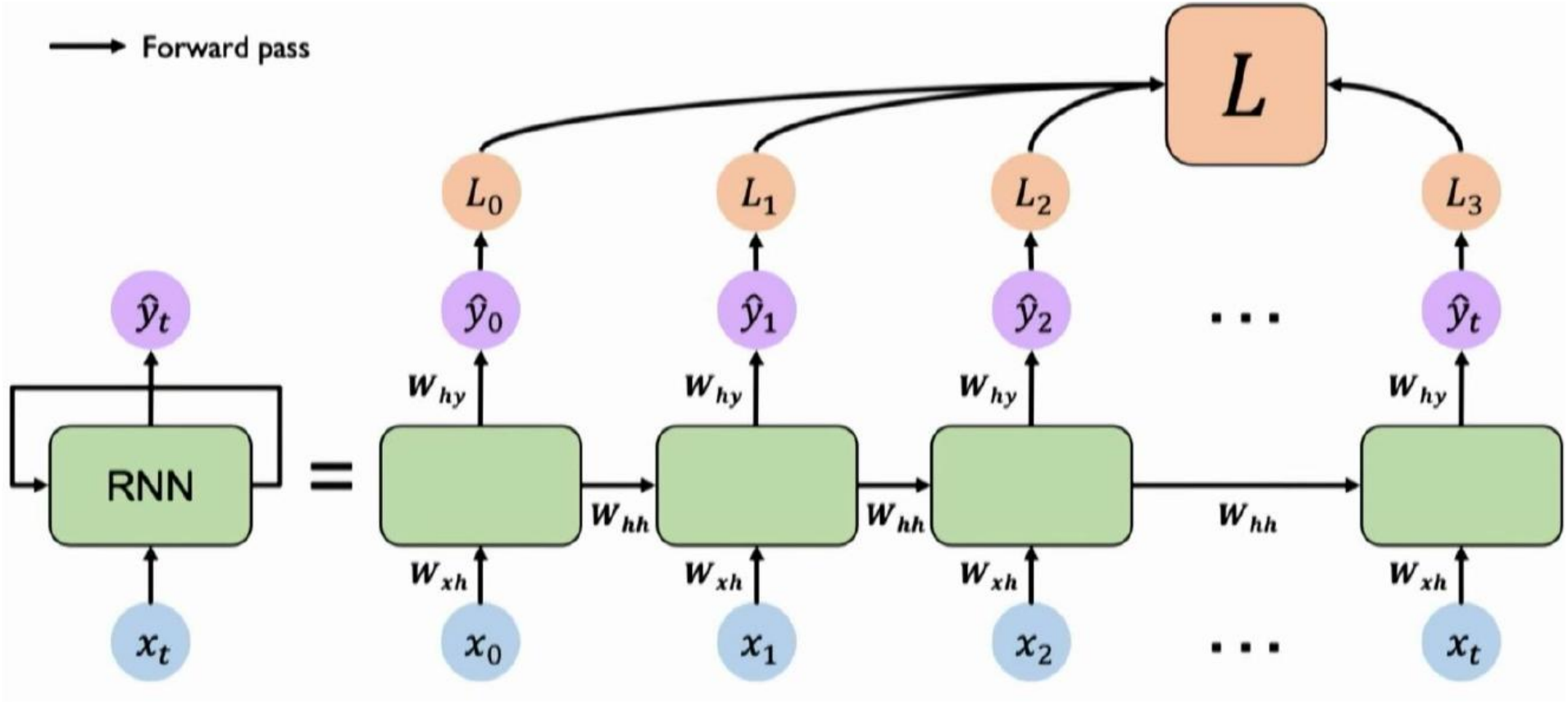
Backpropagation Through Time

- Fig. 8.4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks.
- First, to compute the loss function for the output at time t we need the hidden layer from time t − 1.
- Second, the hidden layer at time t influences both the output at time t and the hidden layer at time t +1 (and hence the output and loss at t +1).
- It follows from this that to assess the error accruing to ht, we'll need to know its influence on both the current output as well as the ones that follow.

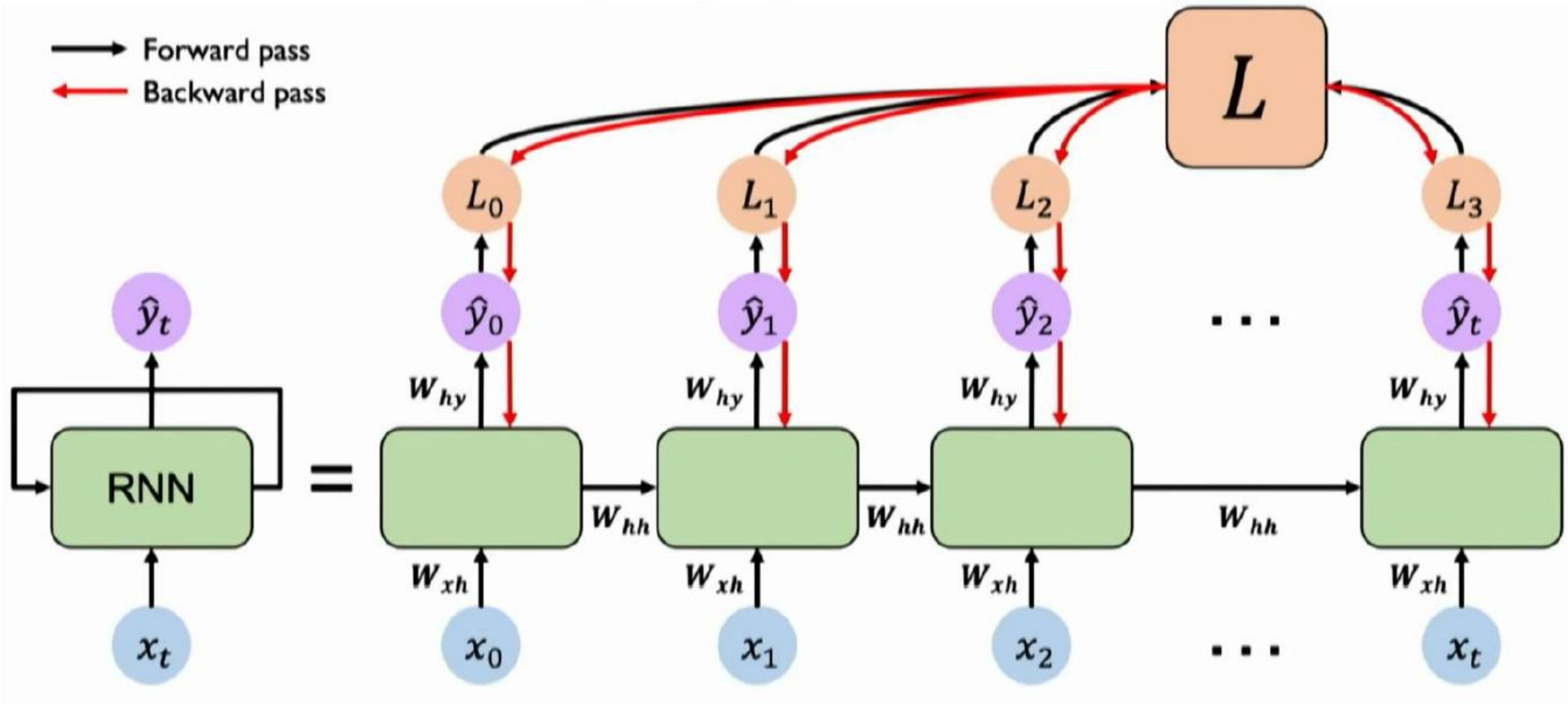
Backpropagation Through Time

- Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs.
- In the first pass, we perform forward inference, computing h_t, y_t, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step.
- In the second phase, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time.
- This general approach is commonly referred to as backpropagation through time.

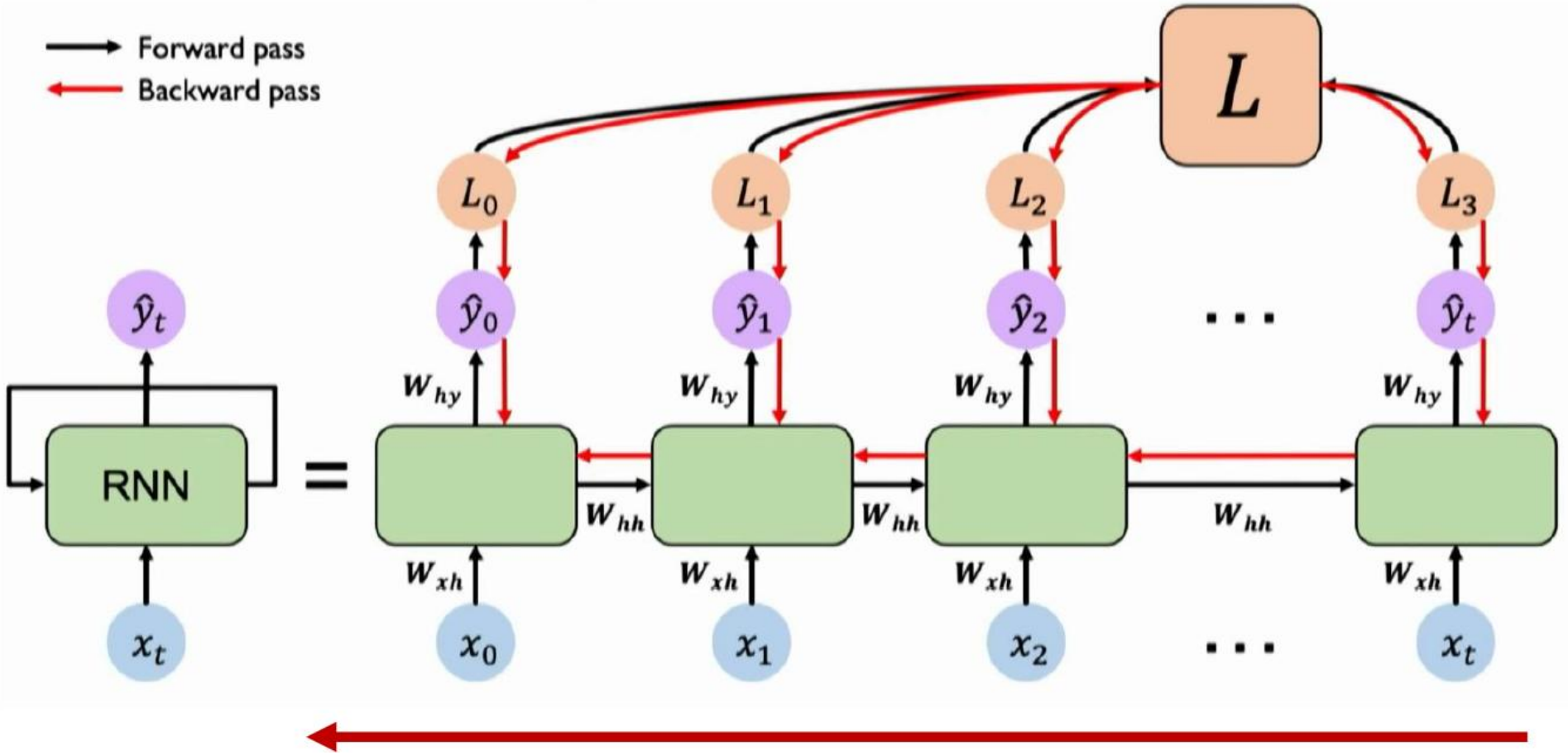
Backpropagation Through Time



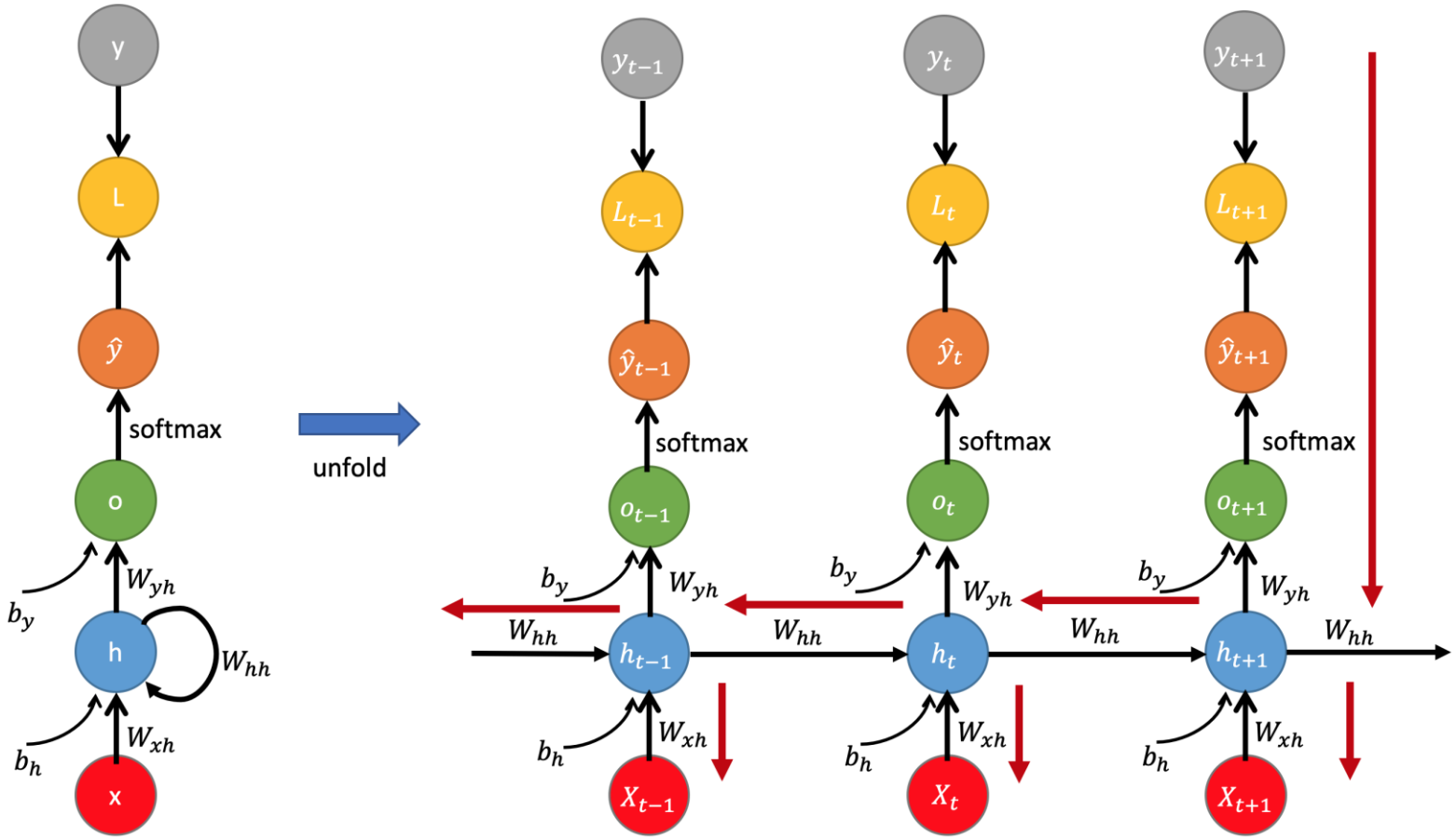
Backpropagation Through Time



Backpropagation Through Time



Backpropagation Through Time



https://mmuratarat.github.io/2019-02-07/bptt-of-rnn

RNNs as Language Models

RNNs as Language Models

- Let's see how to apply RNNs to the language modeling task. Recall from Chapter 3 that language models predict the next word in a sequence given some preceding context.
- For example, if the preceding context is "Thanks for all the" and we want to know how likely the next word is "fish" we would compute:
- P(fish|Thanks for all the).
- The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the n−1 prior words.
- The context is thus of size n−1.
- For the feedforward language models of Chapter 7, the context is the window size.

RNNs as Language Models

- RNN language models process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state.
- RNNs thus don’t have the limited context problem that n-gram models have, or the fixed context that feedforward language models have, since the hidden state can in principle represent information about all of the preceding words all the way back to the beginning of the sequence.

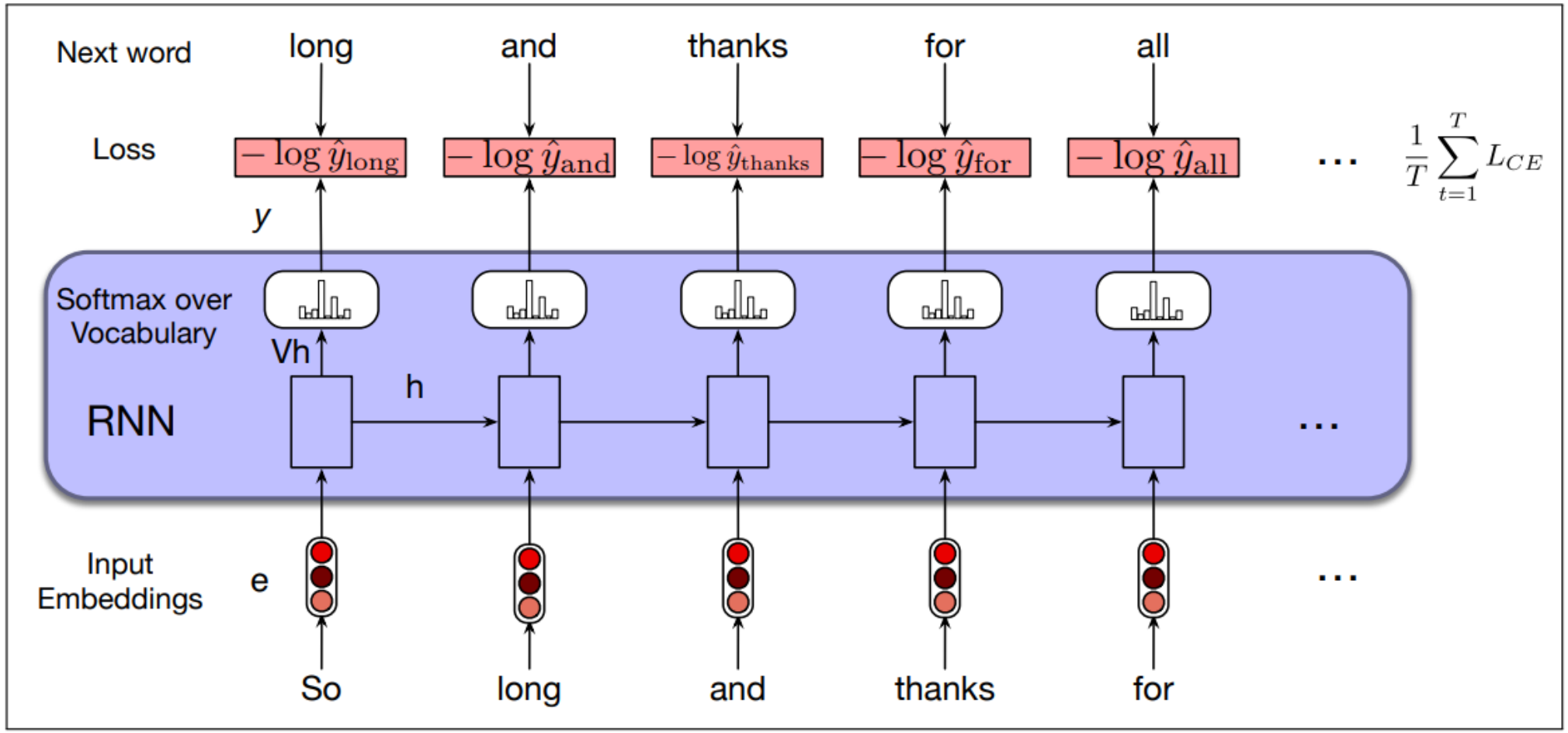


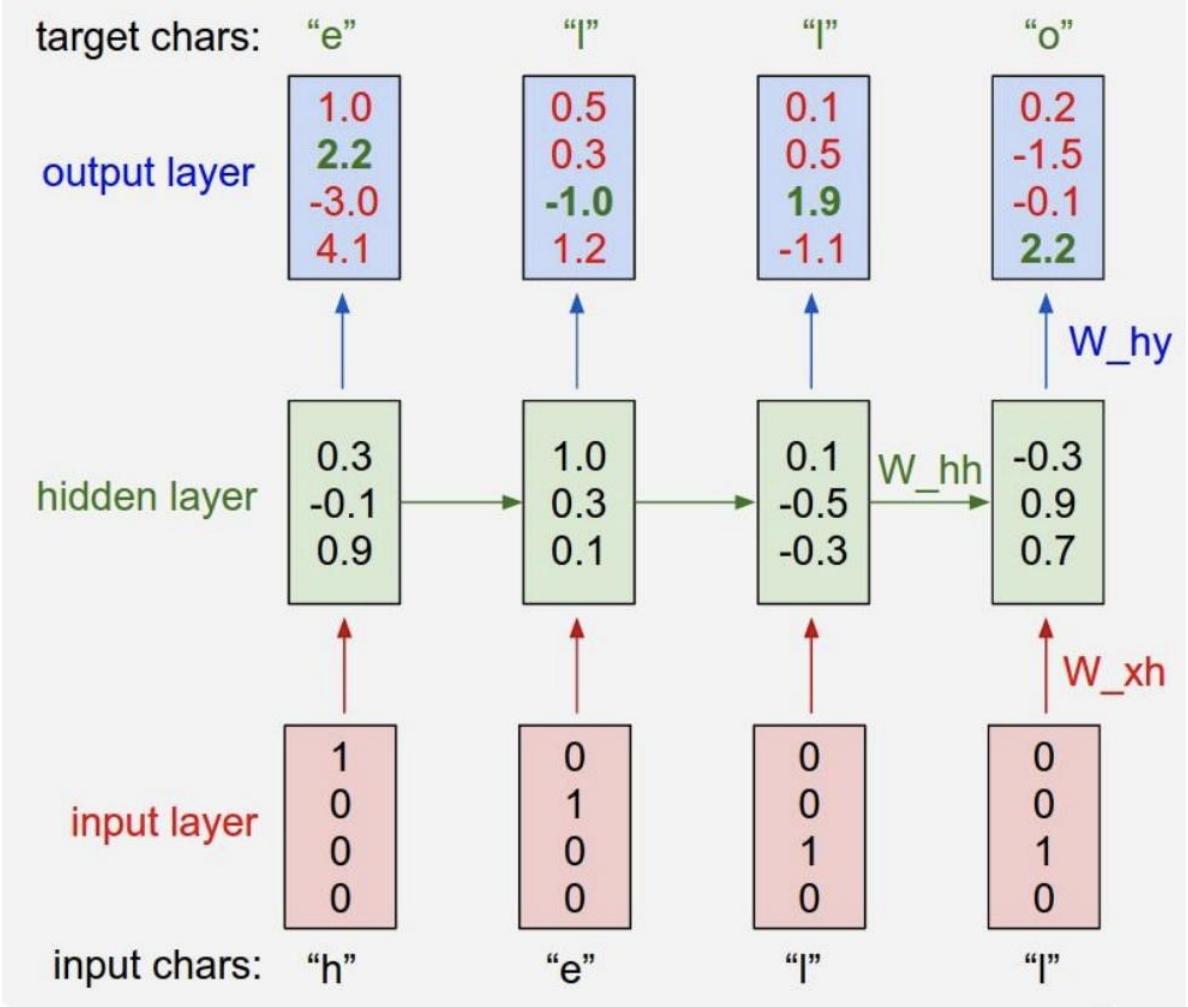
Figure 8.6 Training RNNs as language models.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

RNN as Character-level Language Model

- One of the simplest ways in which we can use an RNN is in the case of a character-level language model since it's intuitive to understand.
- Suppose, in a very simple example, we have the training sequence of just one string "hello", and we have a vocabulary
  - $V \in \{"h", "e", "l", "o"\}$
- We are going to try to get an RNN to learn to predict the next character in the sequence on this training data.

RNN as Character-level Language Model



This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called teacher forcing.

RNN as Character-level Language Model

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \text{"h"} \quad \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \text{"e"} \quad \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \text{"l"} \quad \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \text{"o"}$$

$$\begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + W_{xh} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}) \quad (1)$$

$$\begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}) \quad (2)$$

$$\begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}) \quad (3)$$

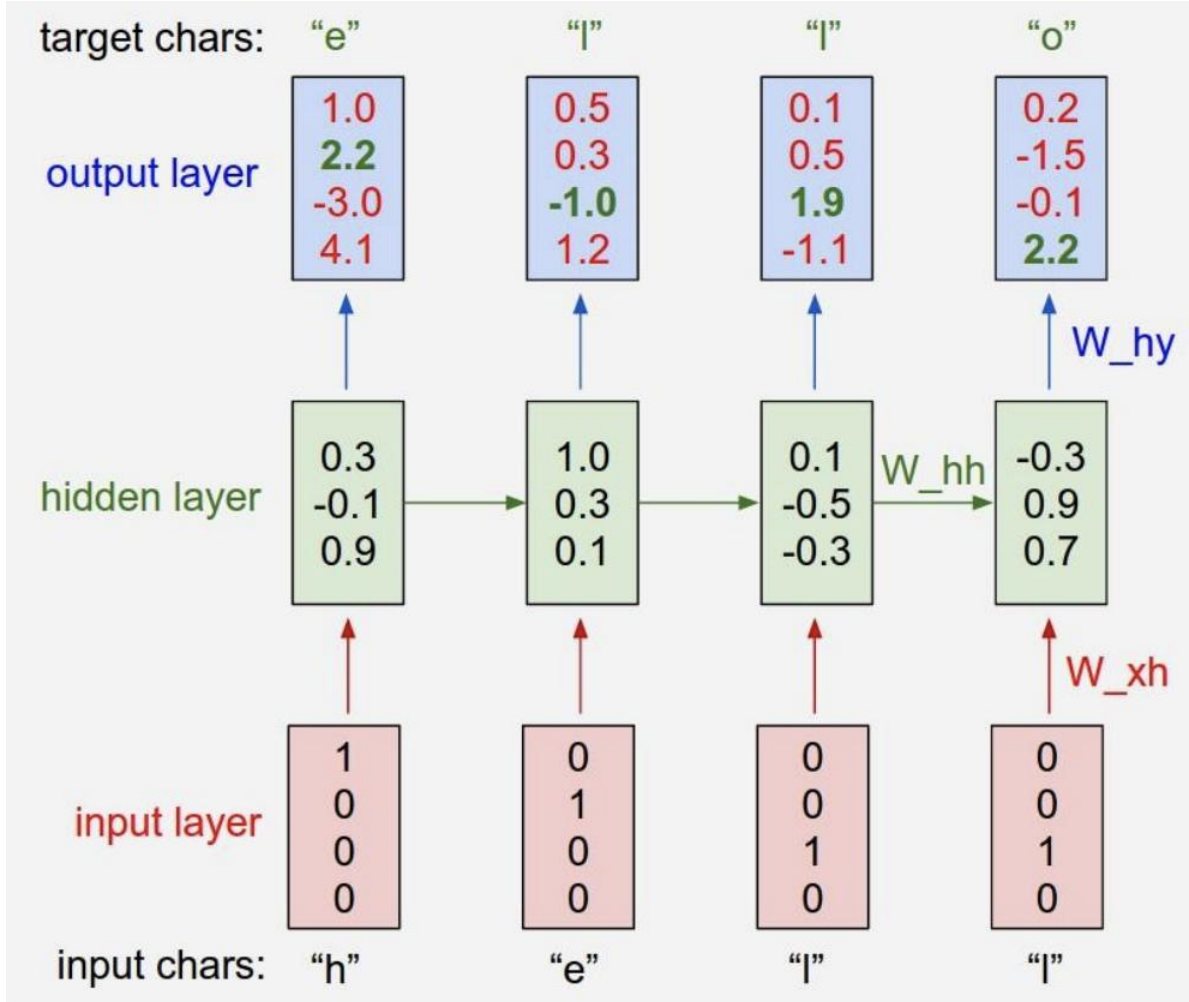
$$\begin{bmatrix} -0.3 \\ 0.9 \\ 0.7 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}) \quad (4)$$

$$\begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + W_{xh} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}) \quad (1)$$

$$\begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}) \quad (2)$$

$$\begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}) \quad (3)$$

$$\begin{bmatrix} -0.3 \\ 0.9 \\ 0.7 \end{bmatrix} = f_W(W_{hh} \begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}) \quad (4)$$



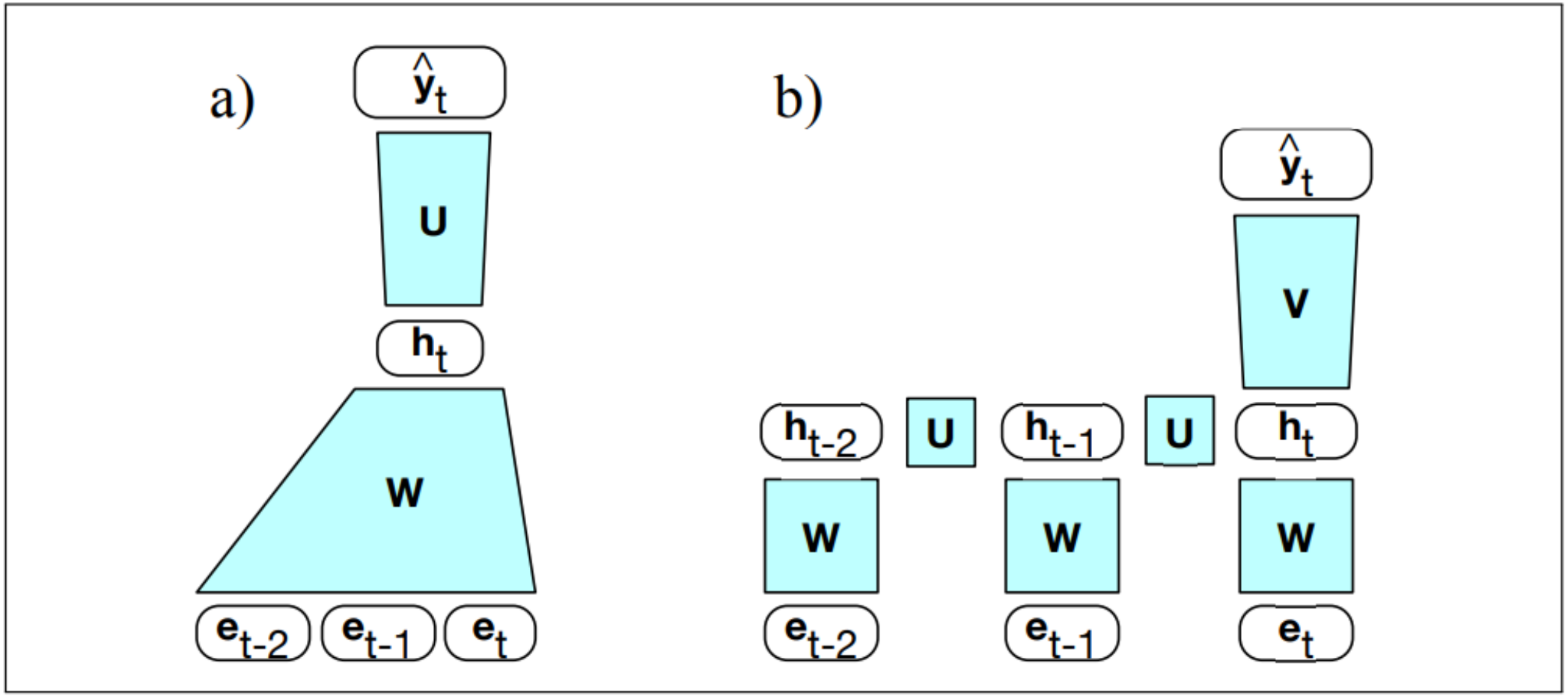
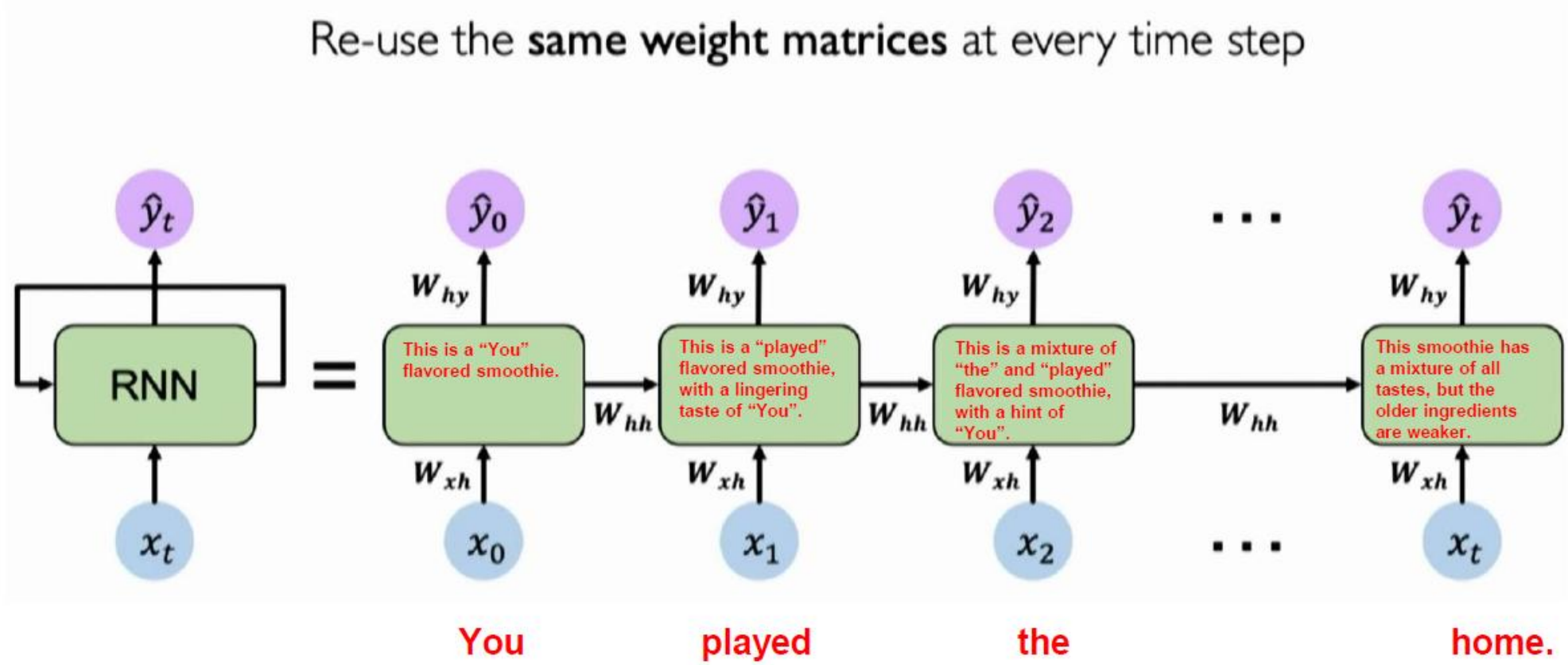
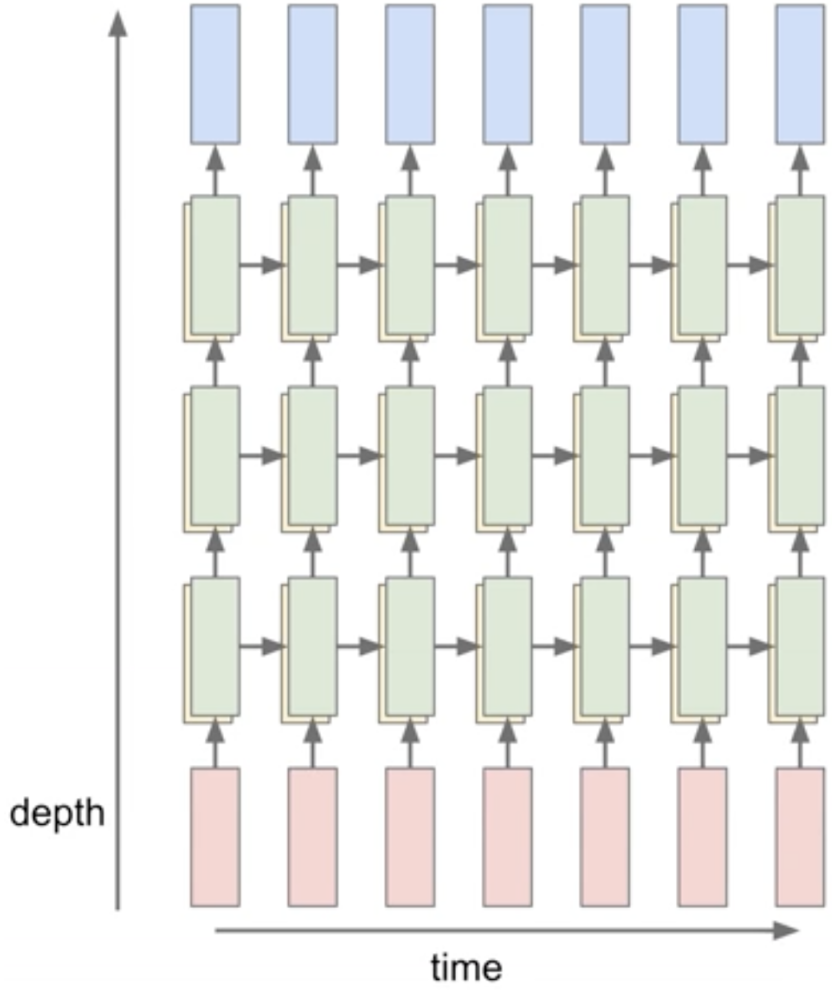


Figure 8.5 Simplified sketch of two LM architectures moving through a text, showing a schematic context of three tokens: (a) a feedforward neural language model which has a fixed context input to the weight matrix **W**, (b) an RNN language model, in which the hidden state $\mathbf{h}_{t-1}$ summarizes the prior context.

RNN as Character-level Language Model



Multilayer RNNs



RNNs for other NLP tasks

Part-of-Speech Tagging

Sequence Labeling

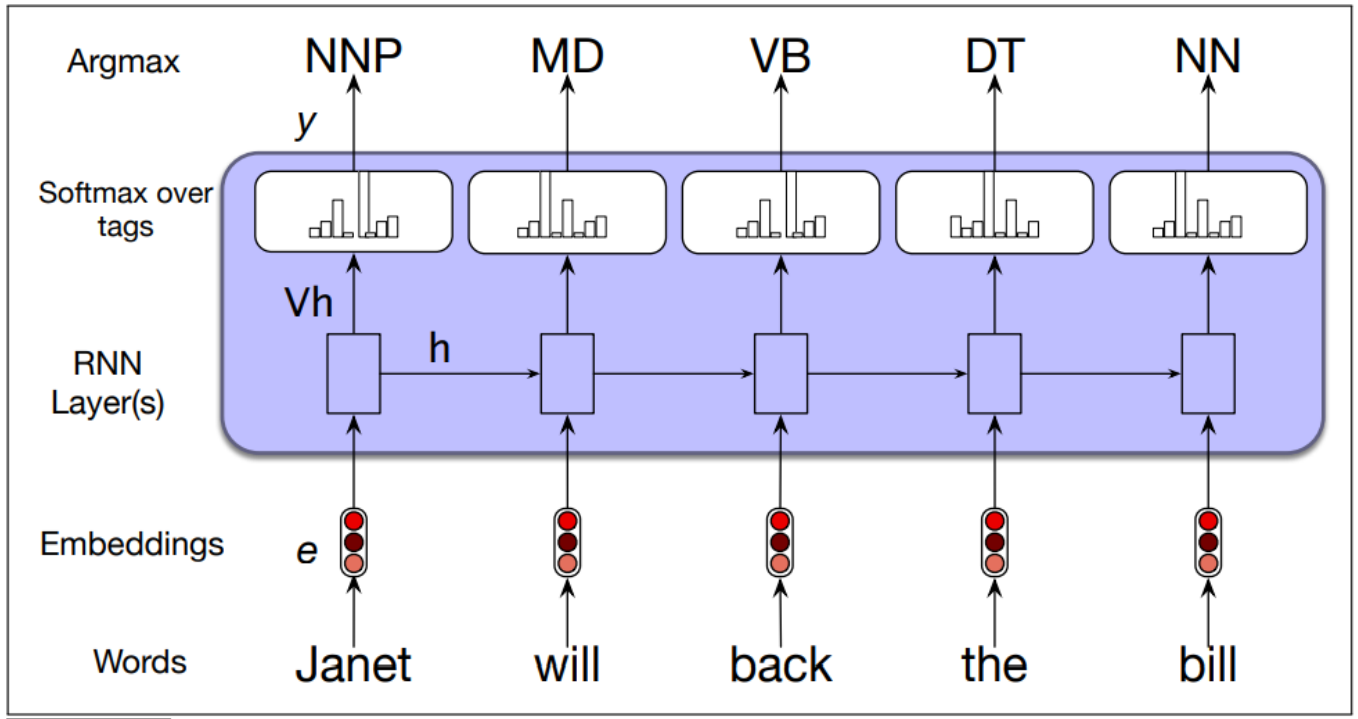


Figure 8.7 Part-of-speech tagging as sequence labeling with a simple RNN. The goal of part-of-speech (POS) tagging is to assign a grammatical label to each word in a sentence, drawn from a predefined set of tags. (The tags for this sentence include NNP (proper noun), MD (modal verb) and others; we'll give a complete description of the task of part-of-speech tagging in Chapter 17.) Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

RNNs for Sequence Classification

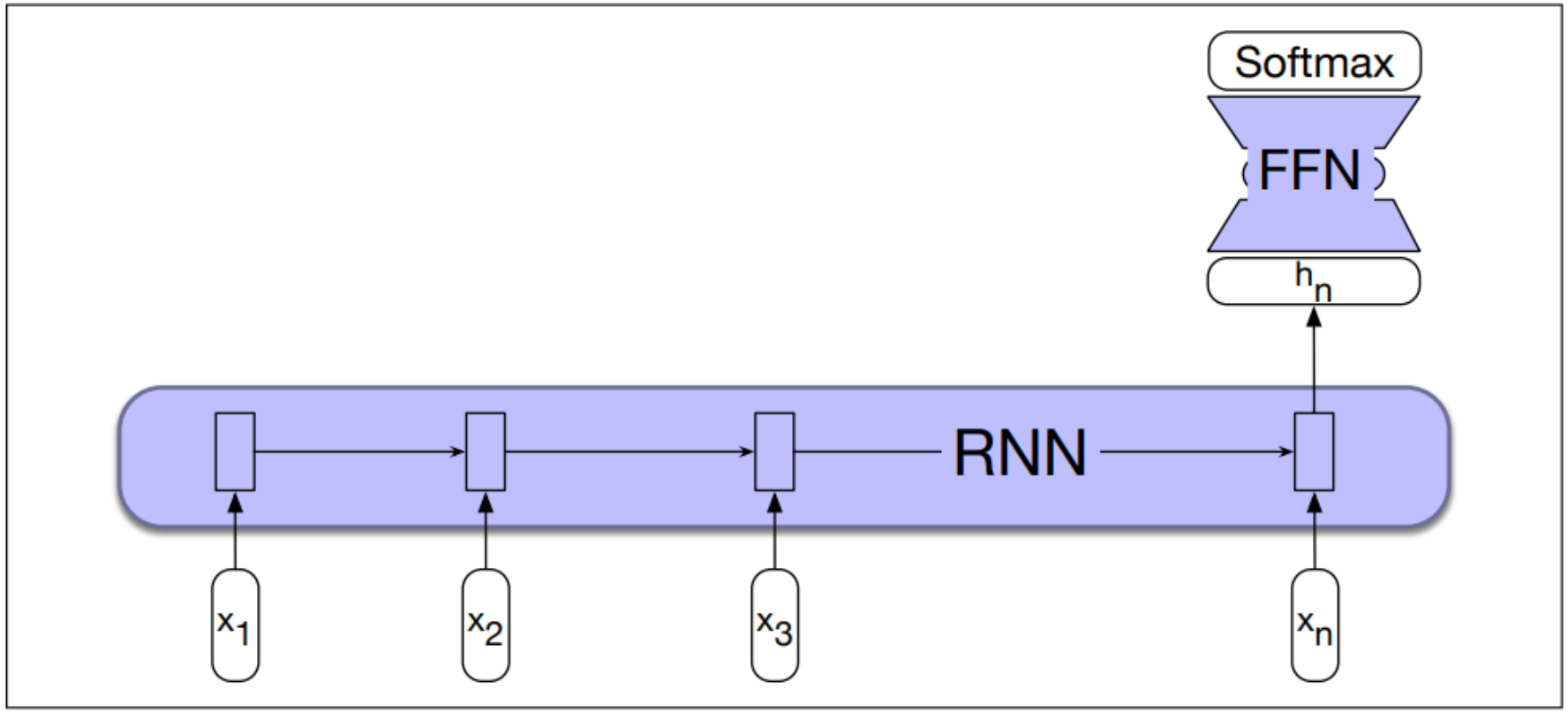


Figure 8.8 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Generation with RNN-Based Language Models

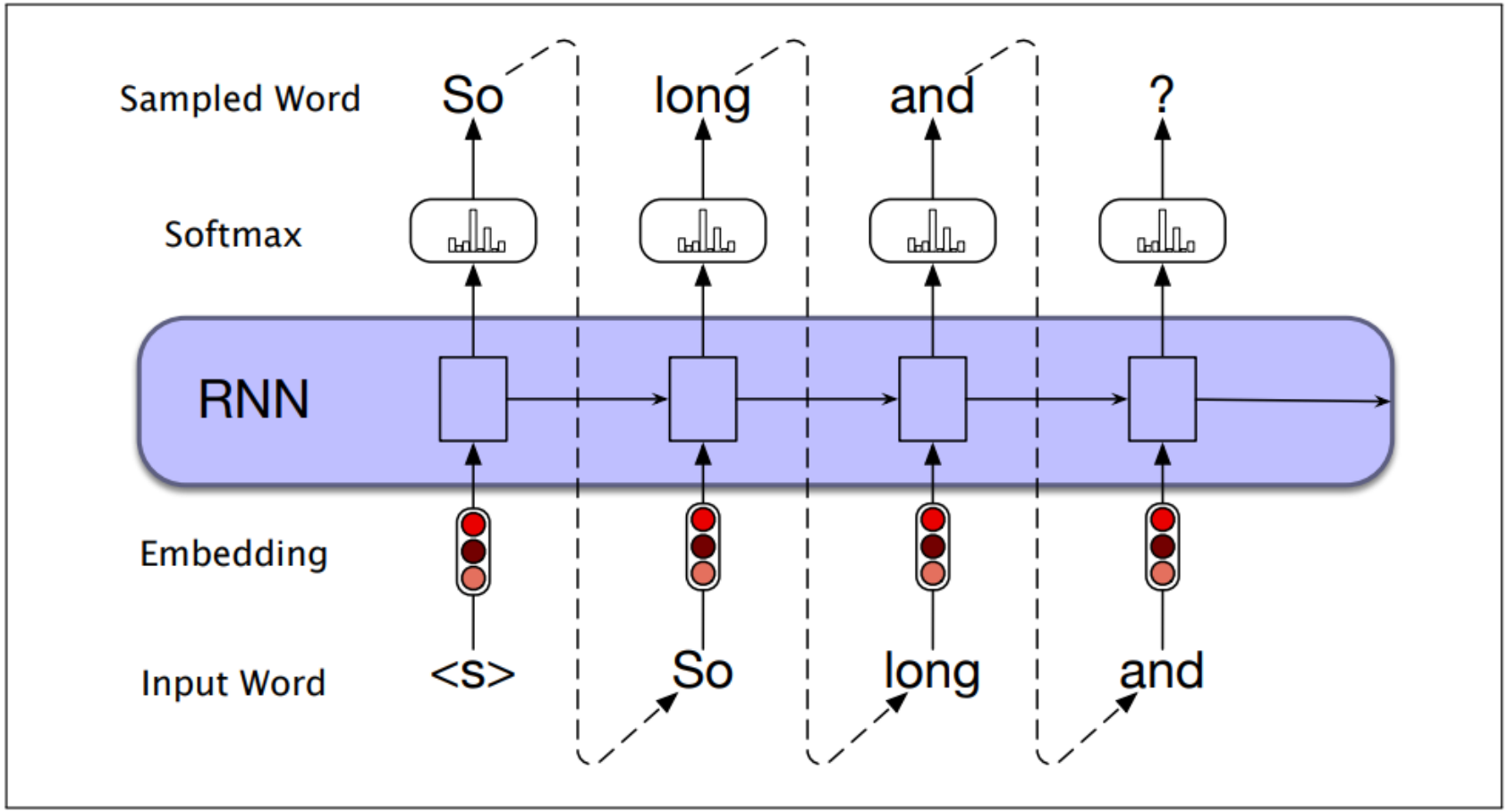


Figure 8.9 Autoregressive generation with an RNN-based neural language model.

Stacked and Bidirectional RNN architectures

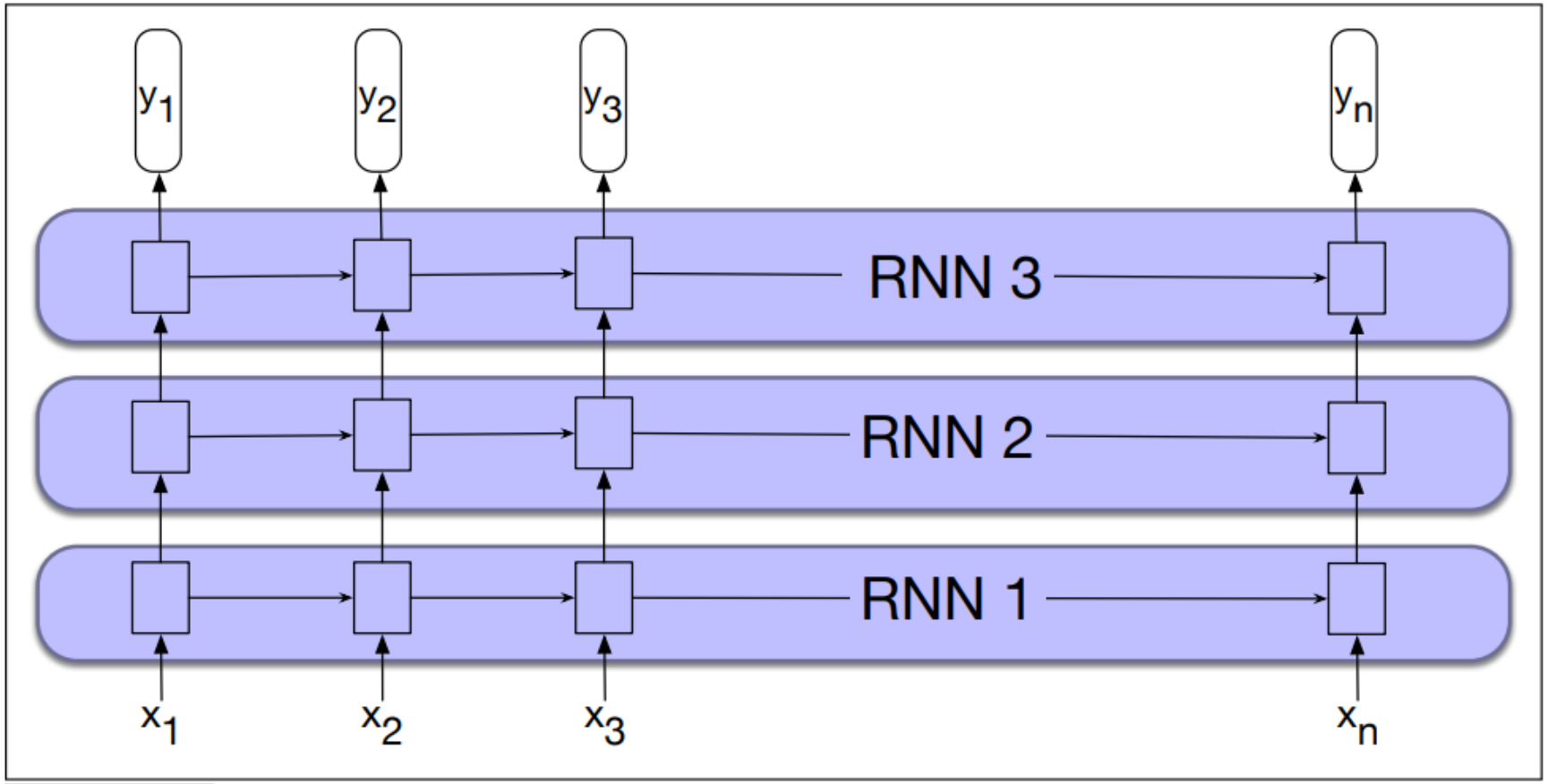


Figure 8.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

Stacked and Bidirectional RNN architectures

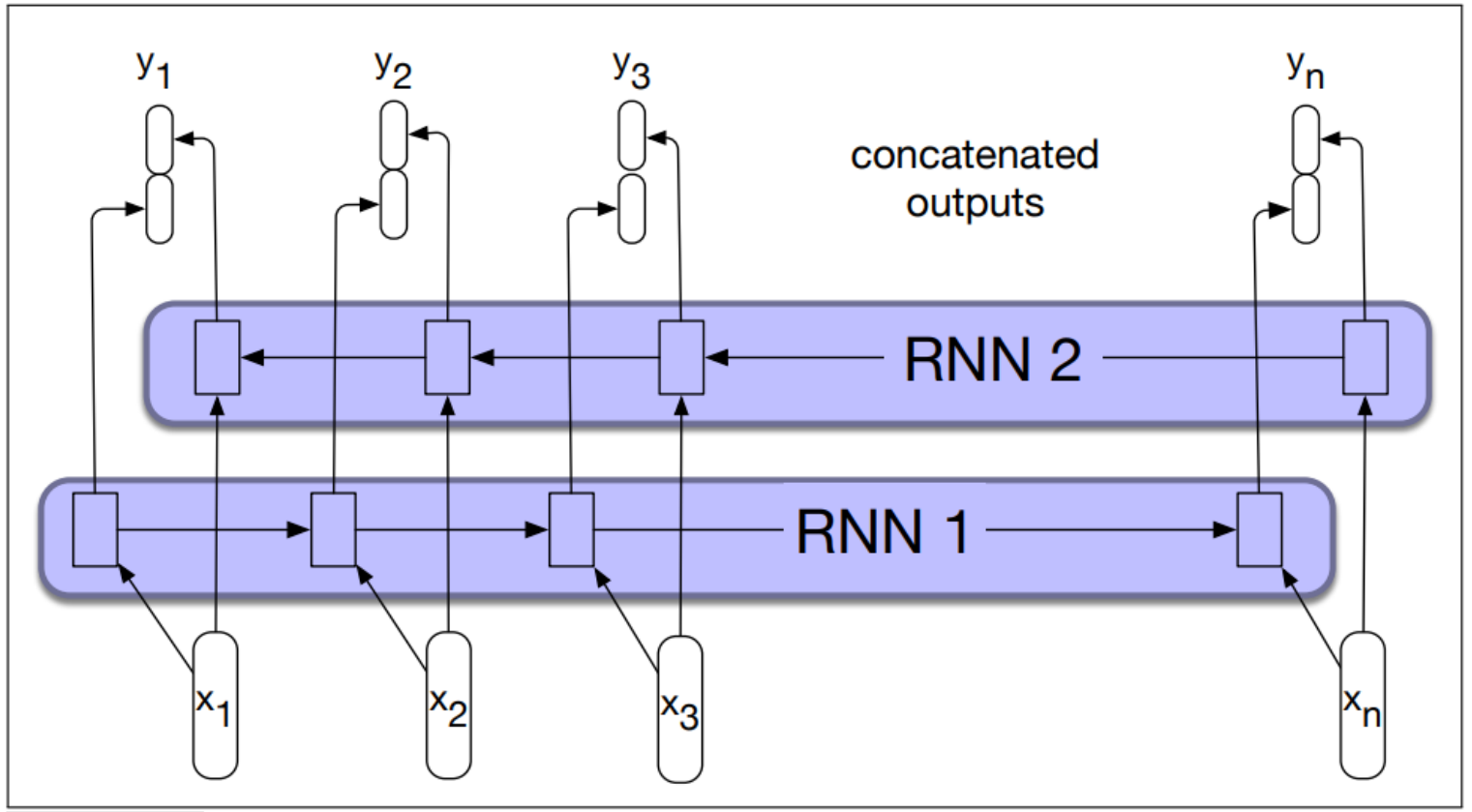


Figure 8.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Stacked and Bidirectional RNN architectures

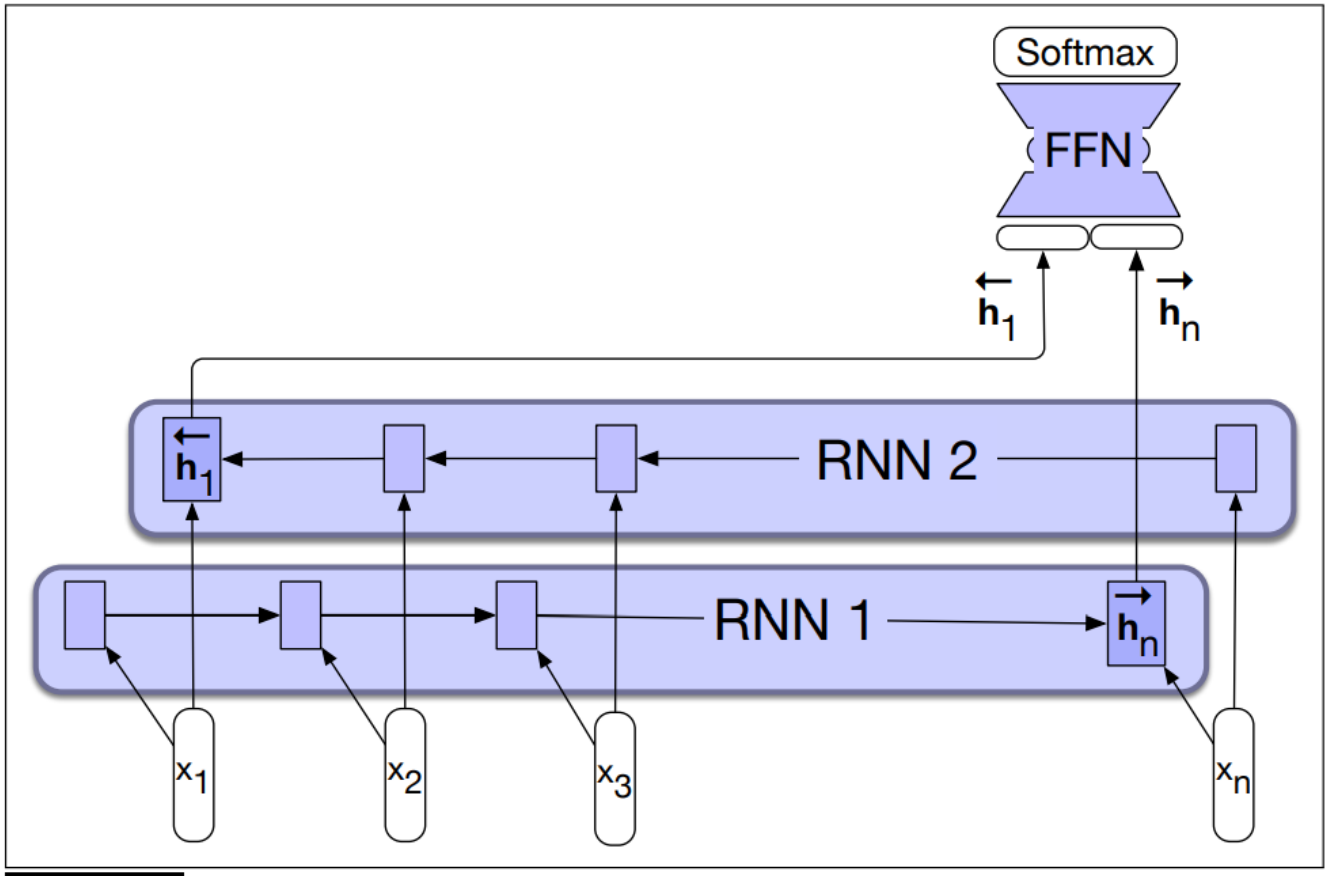
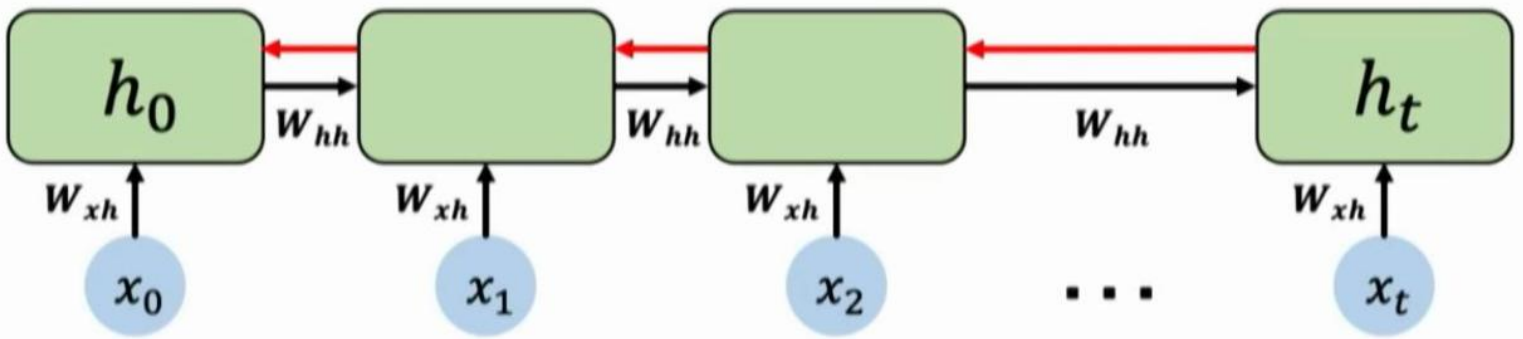


Figure 8.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

RNN Limitations

Exploding Gradients

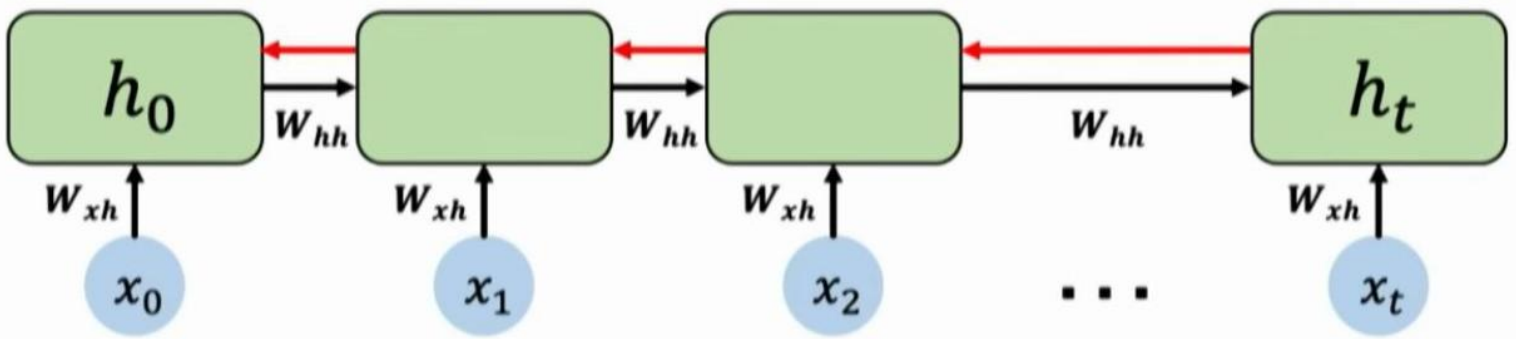


Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

Many values > 1:
exploding gradients

Gradient clipping to
scale big gradients

Vanishing Gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

Many values < 1:
vanishing gradients

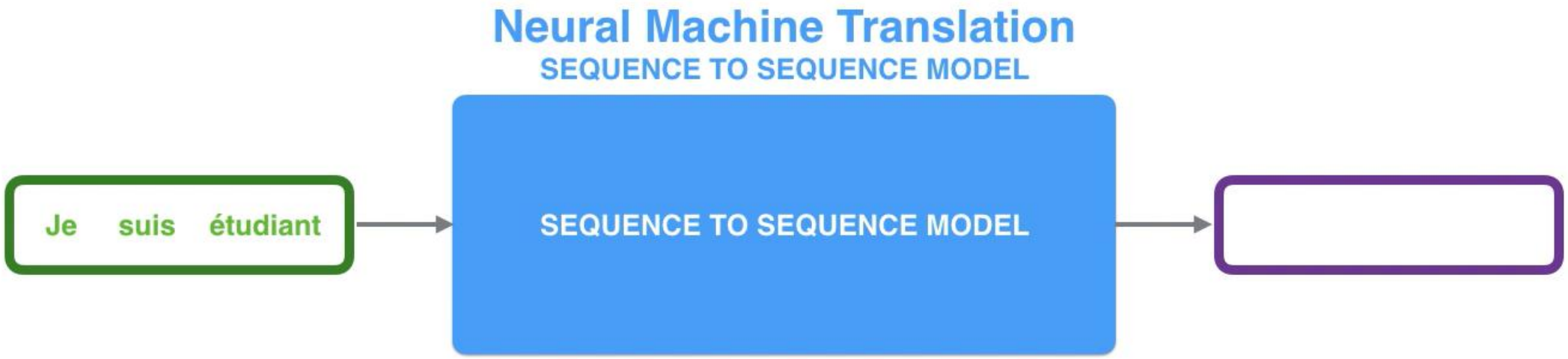
1. Activation function
2. Weight initialization
3. Network architecture

Sequence-to-Sequence Models

A sequence-to-sequence model is a model that takes a sequence of items (words, letters, features of an images…etc) and outputs another sequence of items. A trained model would work like this:

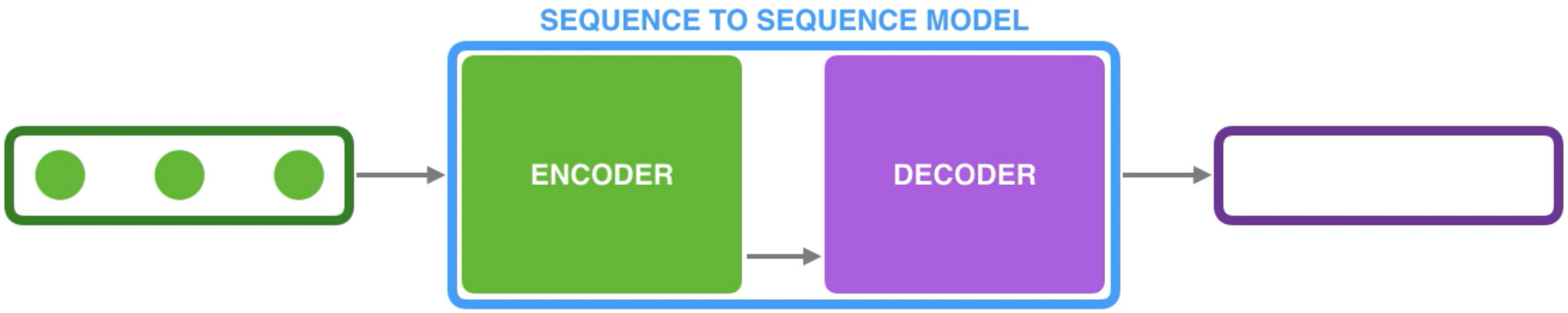
Sequence-to-Sequence Models

In neural machine translation, a sequence is a series of words, processed one after another. The output is, likewise, a series of words:

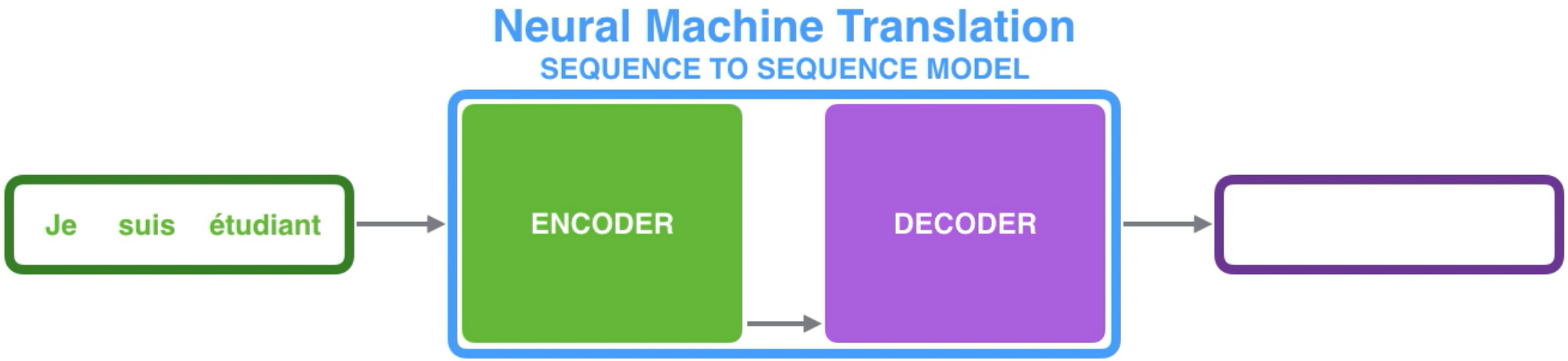


Sequence-to-Sequence Models

The encoder processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sequence item by item.

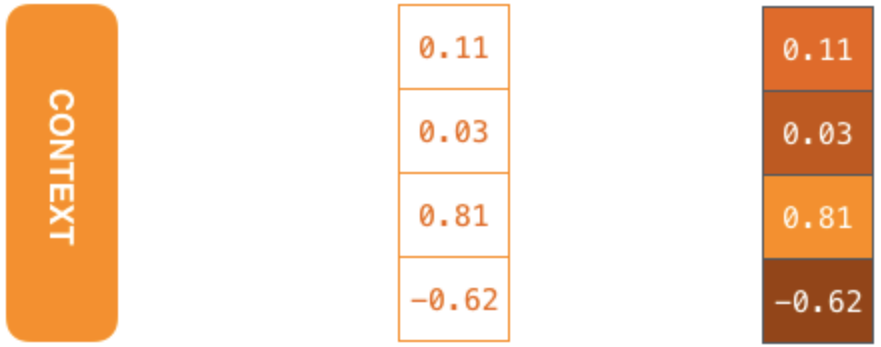


Sequence-to-Sequence Models



Sequence-to-Sequence Models

The context is a vector (an array of numbers, basically) in the case of machine translation.
The encoder and decoder tend to both be recurrent neural networks.



The context is a vector of floats. Later in this post we will visualize vectors in color by assigning brighter colors to the cells with higher values.

Long Short-Term Memory (LSTM)

NN, RNN and LSTM

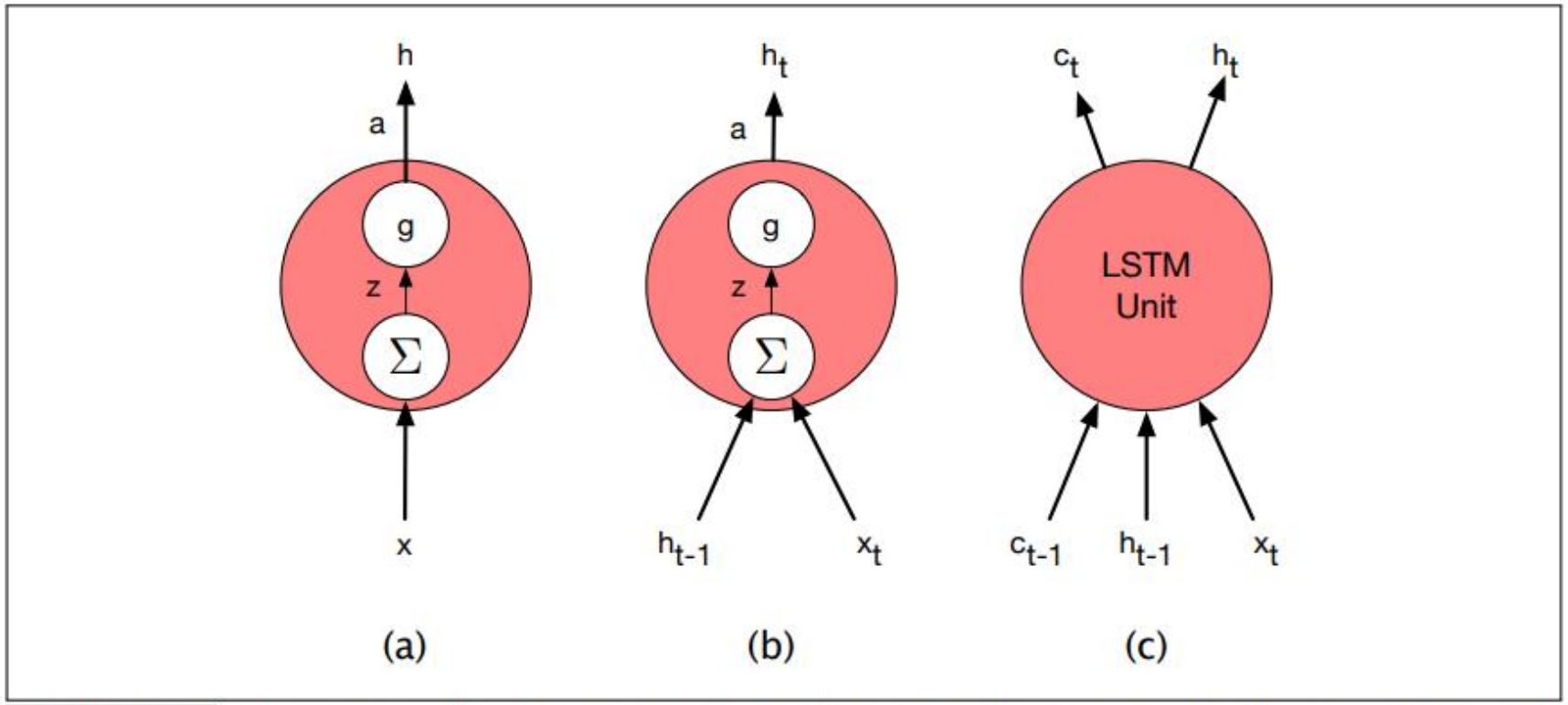


Figure 8.14 Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

A Long short-term memory (LSTM) is a type of Recurrent Neural Network specially designed to prevent the neural network output for a given input from either decaying or exploding as it cycles through the feedback loops.

Long Short-Term Memory (LSTM)

- `The flights the airline was canceling were full.`

- Assigning a high probability to was following airline is straightforward since airline provides a strong local context for the singular agreement.
- However, assigning an appropriate probability to were is quite difficult, not only because the plural flights is quite distant, but also because the singular noun airline is closer in the intervening context.
- Ideally, a network should be able to retain the distant information about plural flights until it is needed, while still processing the intermediate parts of the sequence correctly

Long Short-Term Memory (LSTM)

- LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making.

Long Short-Term Memory (LSTM)

- The gates in an LSTM share a common design pattern; each consists of a feedforward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated.
- The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1.
- Combining this with a pointwise multiplication has an effect similar to that of a binary mask.
- Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

LSTM

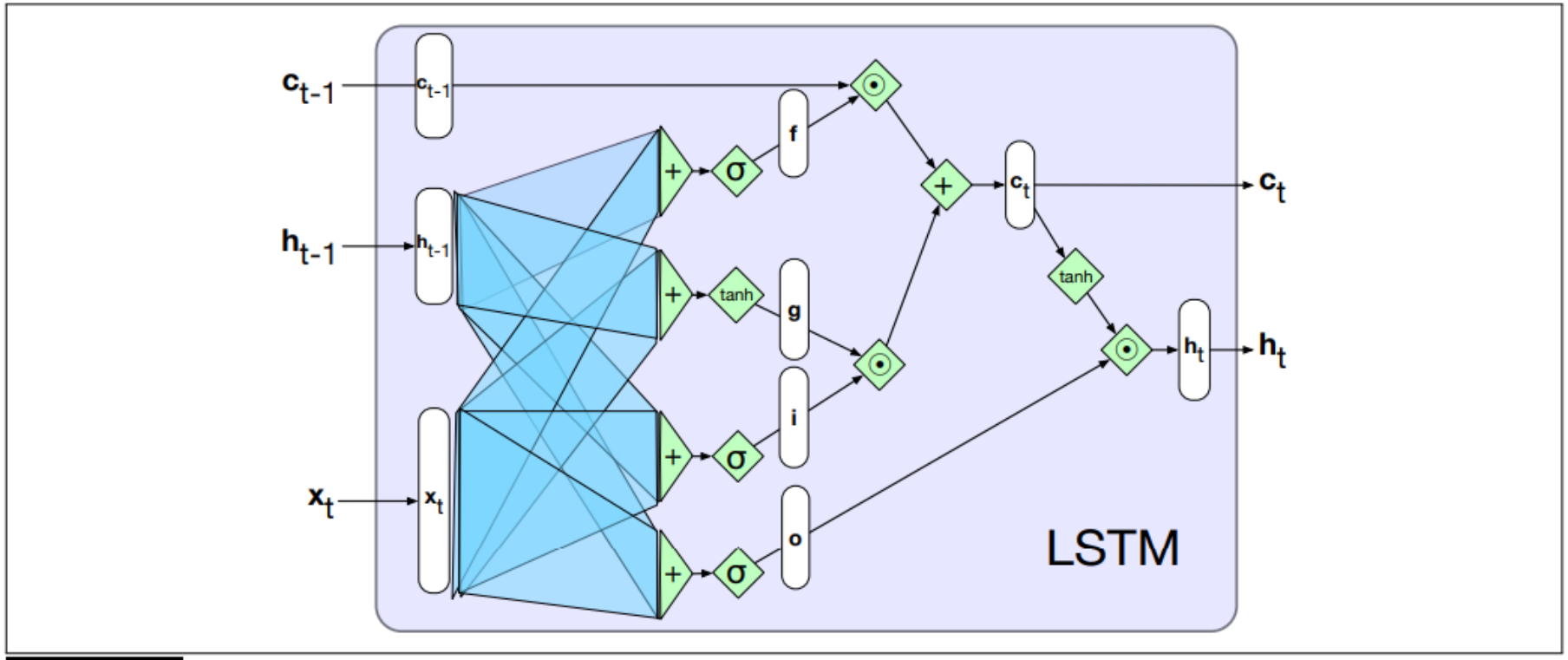
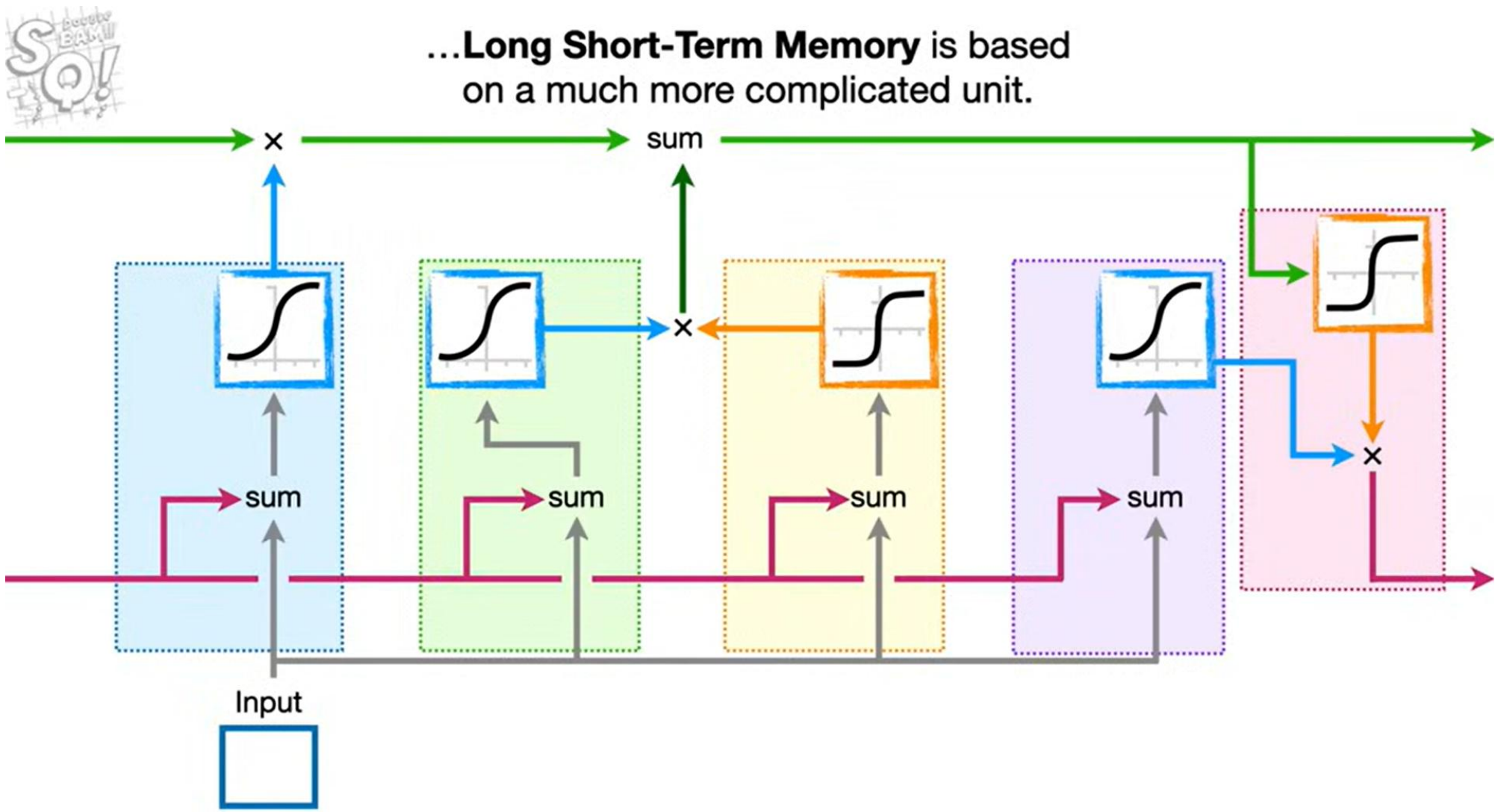
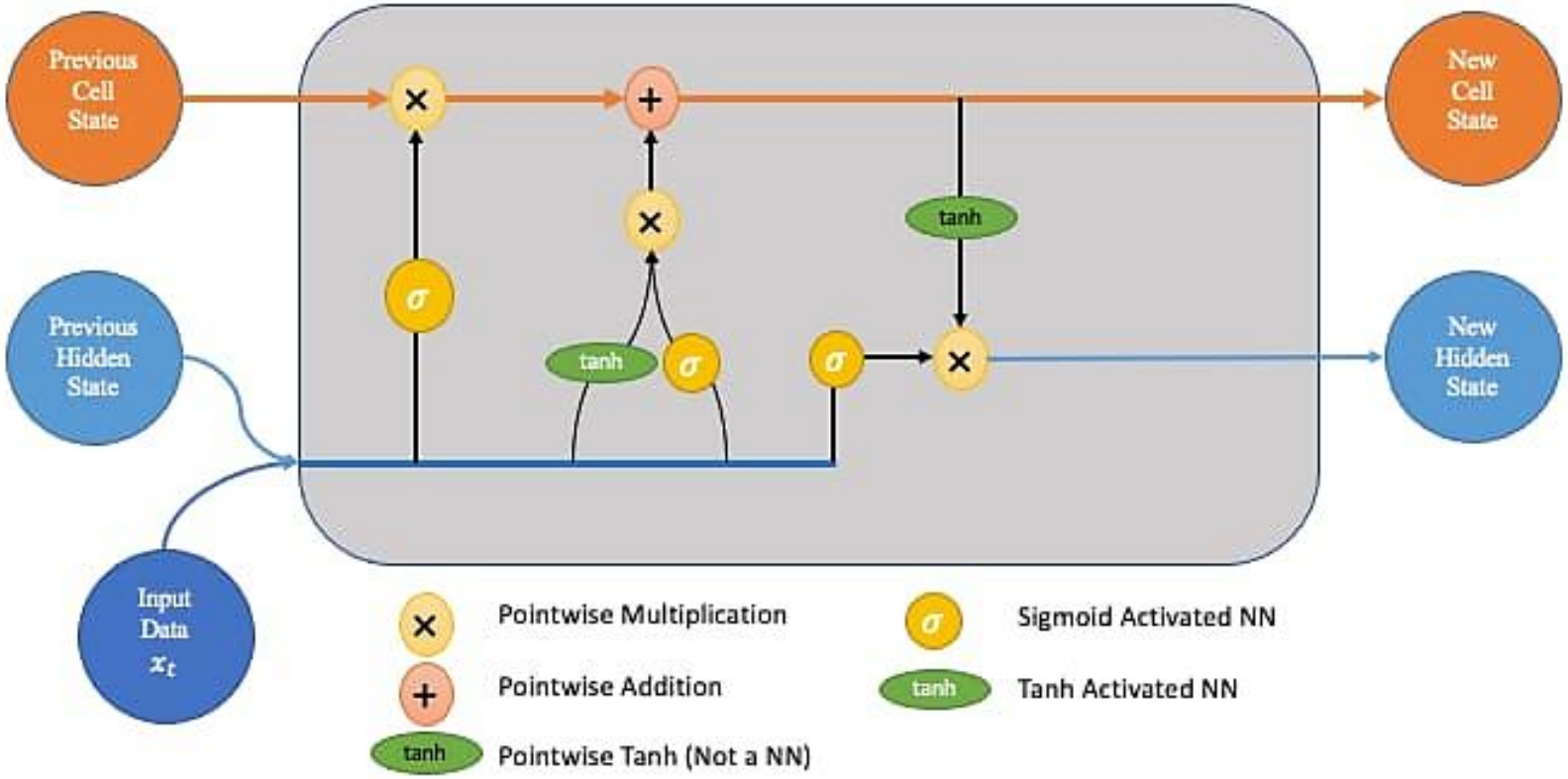


Figure 8.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.


…Long Short-Term Memory is based
on a much more complicated unit.
x
sum
sum
sum
sum
x
x
Input

LSTM


Previous Cell State
New Cell State
Previous Hidden State
New Hidden State
Input Data x_t
tanh
tanh
Pointwise Multiplication
Pointwise Addition
Pointwise Tanh (Not a NN)
Sigmoid Activated NN
Tanh Activated NN

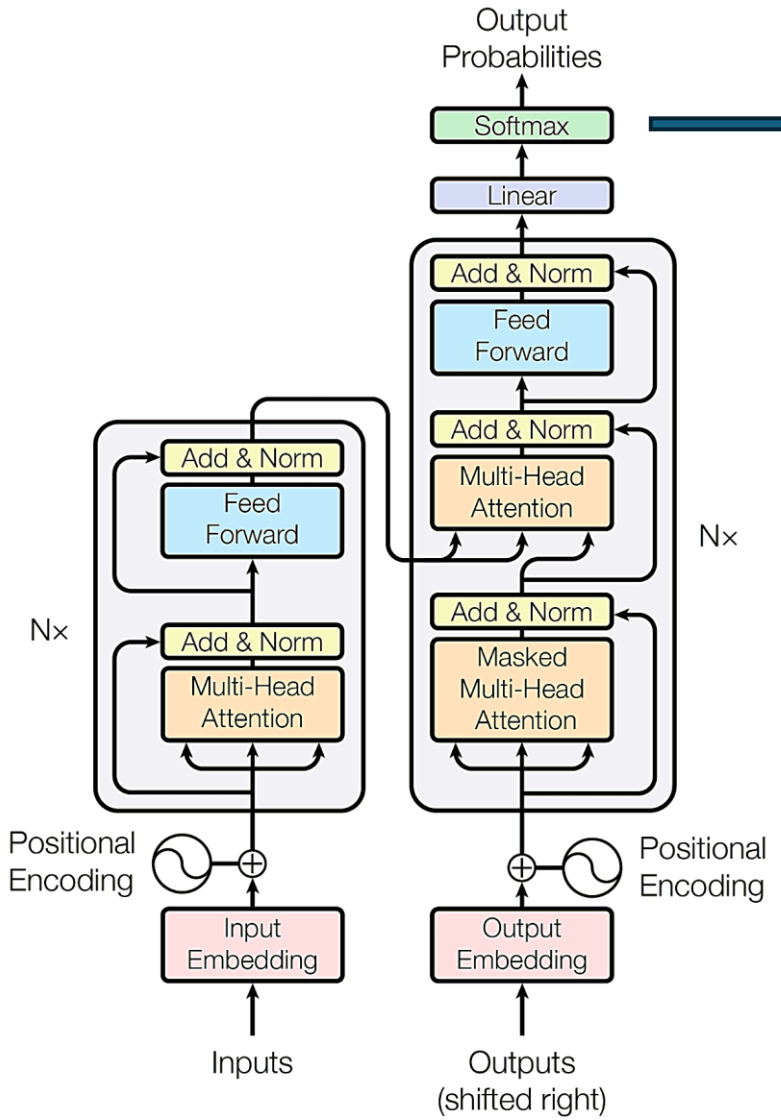
LSTM Gates

- LSTM (Long Short-Term Memory) networks utilize three types of gates to manage the flow of information: the forget gate, the input gate, and the output gate.
- These gates, implemented with sigmoid functions, regulate what information is stored, discarded, and outputted by the LSTM cell, addressing the vanishing gradient problem common in standard RNNs.

LSTM Gates

- Forget Gate:This gate decides what information from the previous cell state should be discarded. It takes the previous hidden state and the current input as input, and uses a sigmoid function to produce values between 0 and 1. A value close to 0 indicates forgetting, while a value close to 1 indicates remembering.
- Input Gate:This gate determines what new information from the current input should be added to the cell state. It has two parts: a sigmoid layer that decides which values to update, and a tanh layer that creates a vector of new candidate values.
- Output Gate:This gate controls what information from the cell state is outputted as the hidden state. It uses a sigmoid function to determine which parts of the cell state are relevant, and then multiplies this with the tanh of the cell state to produce the output.

Softmax


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Softmax

Softmax

- For a vector z of dimensionality K, the softmax is defined as:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \leq i \leq K$$

- The softmax of an input vector $z = [z_1, z_2, \ldots, z_K]$ is thus a vector itself:

$$\text{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right]$$

Softmax

$$\operatorname{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right]$$

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

The resulting (rounded) softmax(z) is:

$$[0.05, 0.09, 0.01, 0.1, 0.74, 0.01]$$

Write a short joke about a dog.

Why did the dog sit in the shade?

Because he didn't want to be a hot dog! 🌭🐶

Write a short joke about a dog.

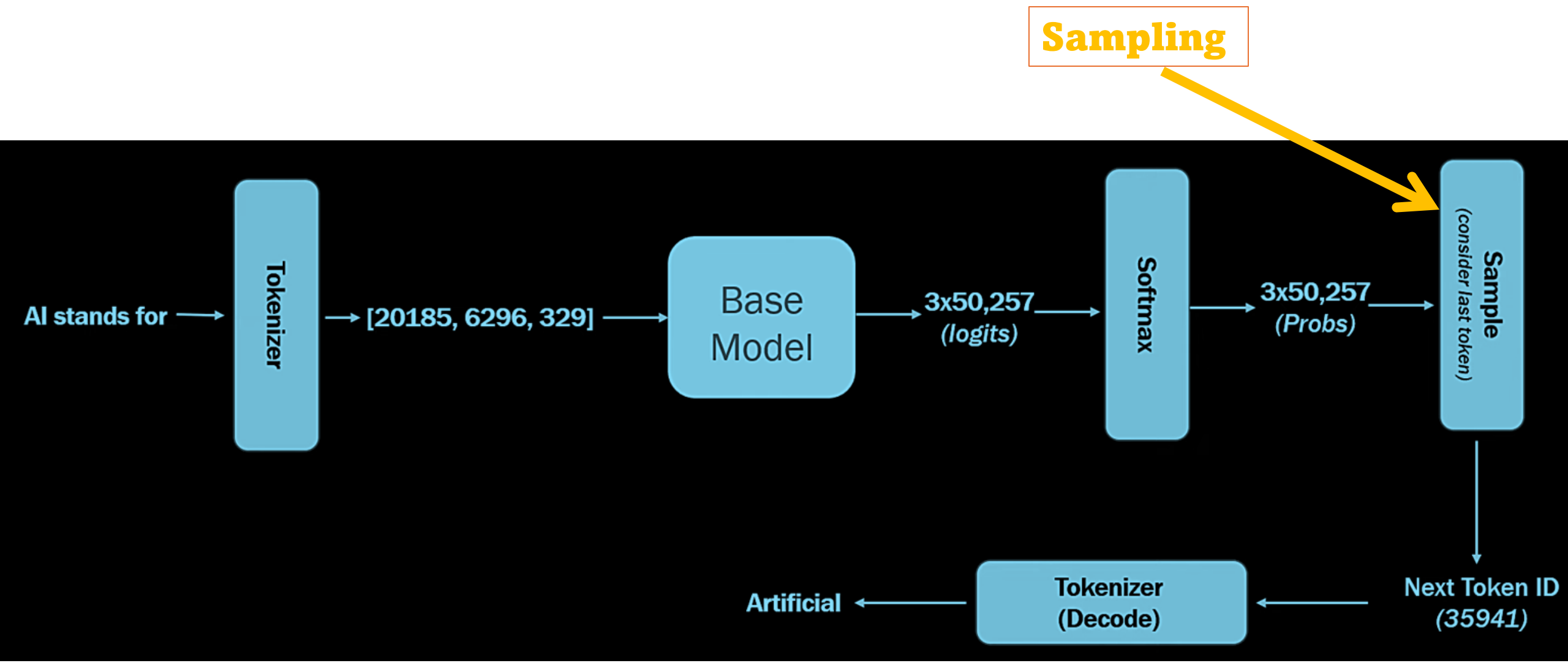
Why don't dogs make good dancers?

Because they have two left feet! 🐾

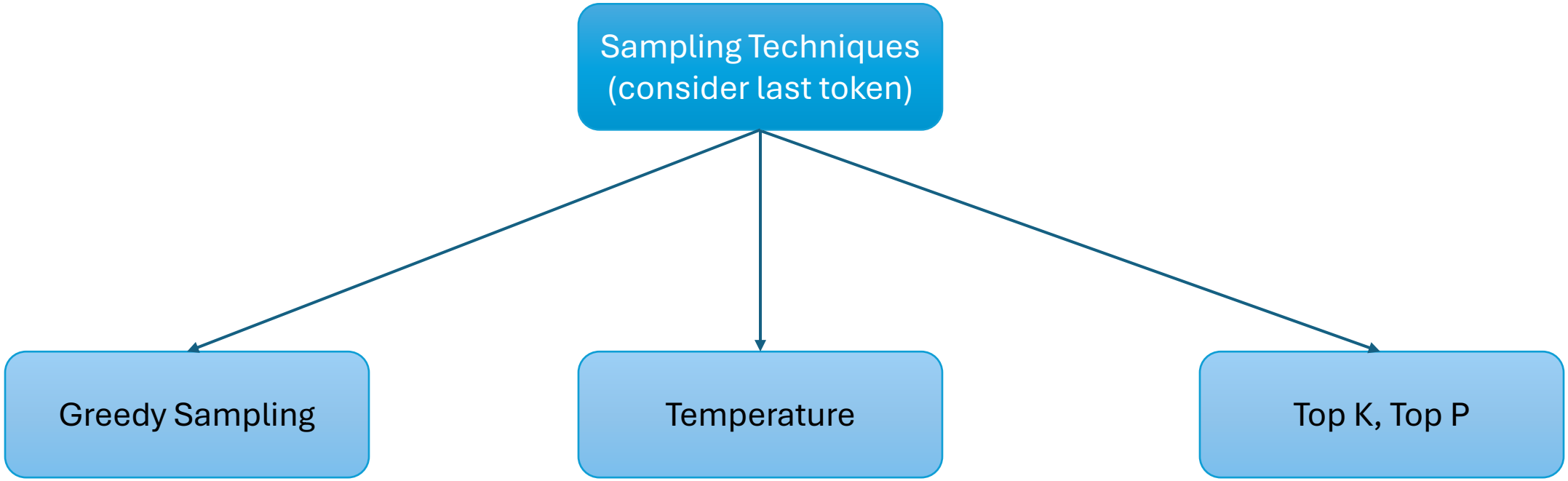
Write a short joke about a dog.

What do you call a dog magician?

A labracadabrador! 🐕✨

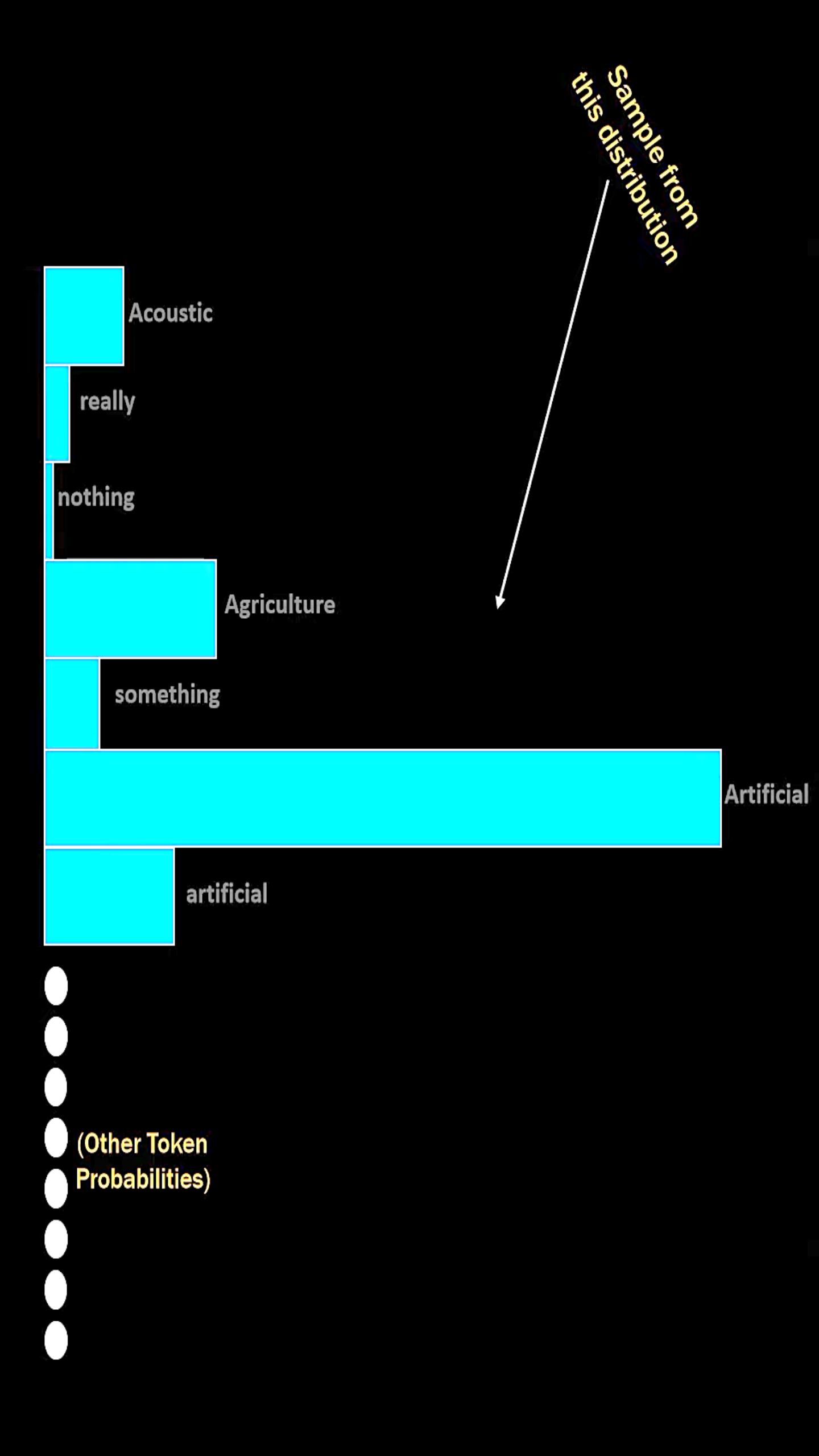

Sampling
AI stands for
Tokenizer
[20185, 6296, 329]
Base Model
3x50,257
(logits)
Softmax
3x50,257
(Probs)
Sample
(consider last token)
Next Token ID
(35941)
Tokenizer
(Decode)
Artificial

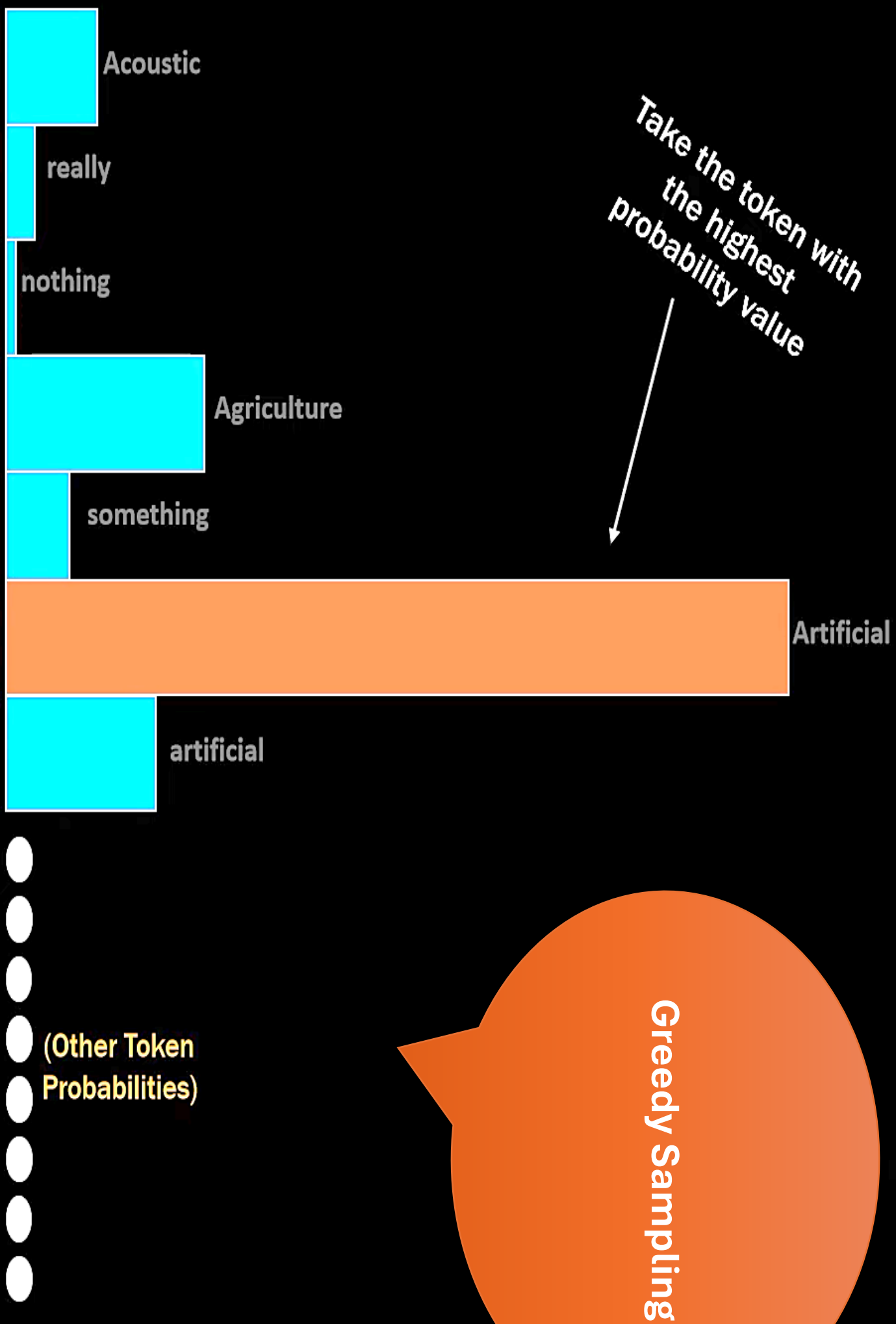
Sampling Strategies



AI stands for →  → ?

Greedy Sampling


Sample from this distribution
Acoustic
really
nothing
Agriculture
something
Artificial
artificial
(Other Token Probabilities)


Acoustic
really
nothing
Agriculture
something
Artificial
artificial
Take the token with the highest probability value
(Other Token Probabilities)
Greedy Sampling

Greedy Sampling

- Pick the token with the highest probability.
- Response is always same given the same context.
- Not creative responses.

Temperature Scaling

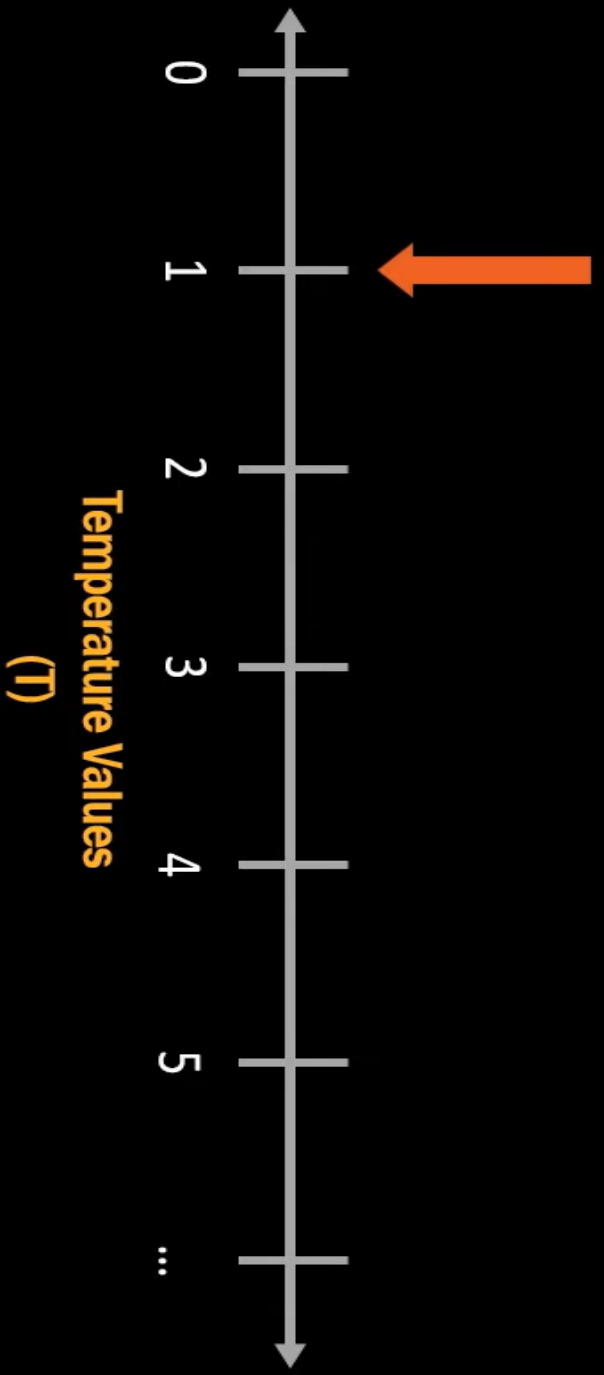
Raw Logits [2 3 1 4]

Scale by some Temperature value (T)

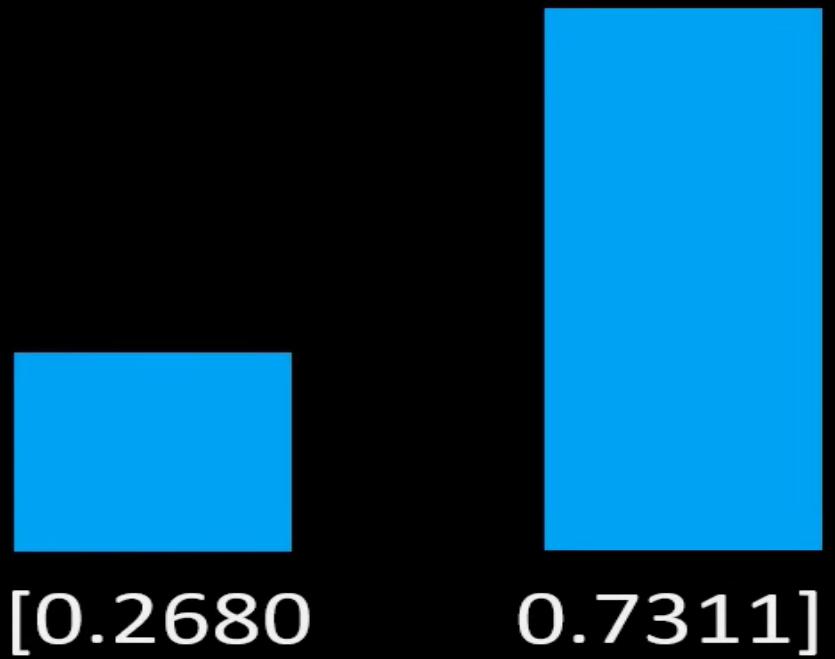
[2/T 3/T 1/T 4 / T]

Apply Softmax on this

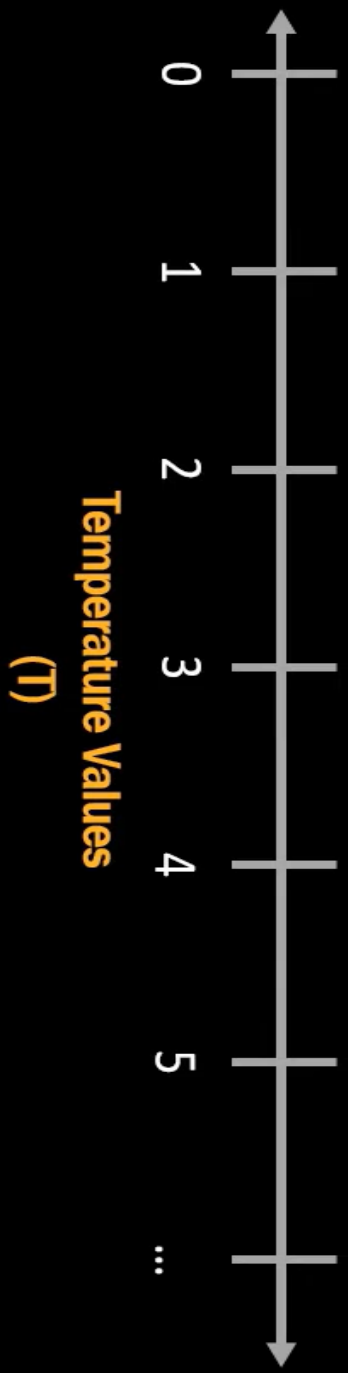
$$softmax = \frac{e^{xi/T}}{\Sigma_j e^{xj/T}}$$



Raw Logits [1 2]



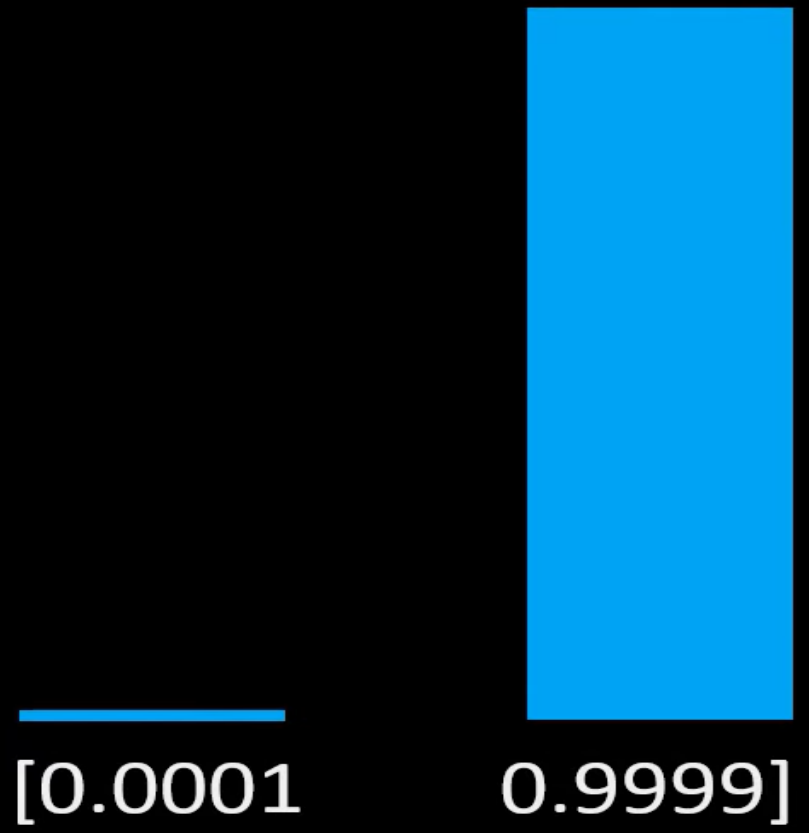
Probability Distribution

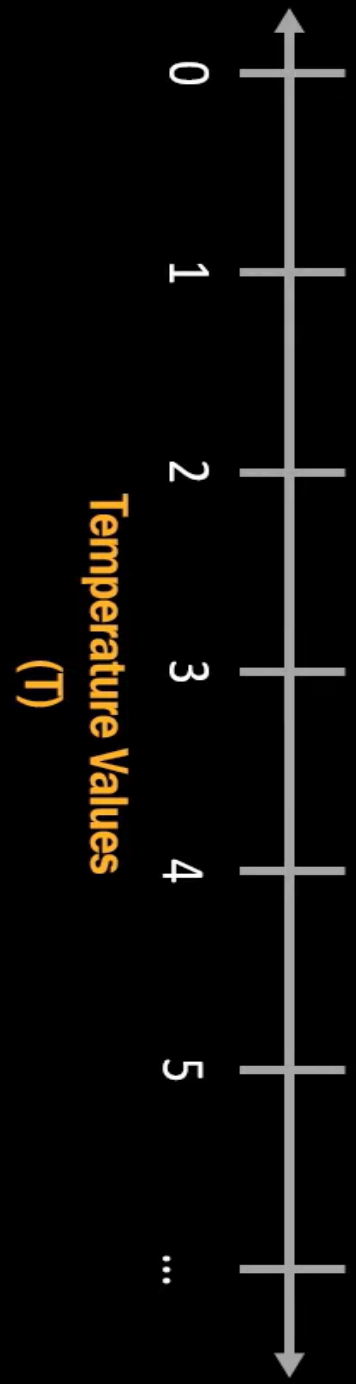


T = 0.1

Raw Logits [1 2]

Scaled Logits [1/0.1 2/0.1]

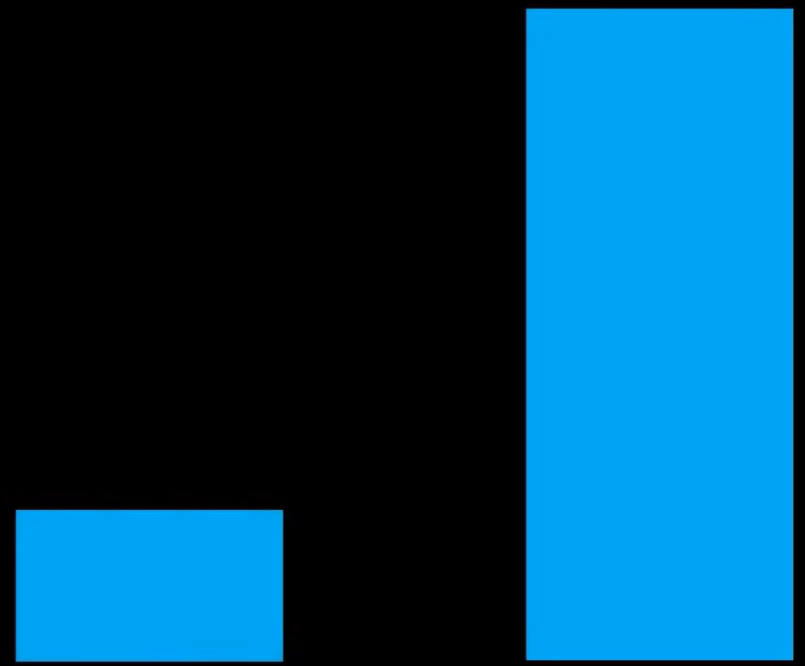




T = 0.5

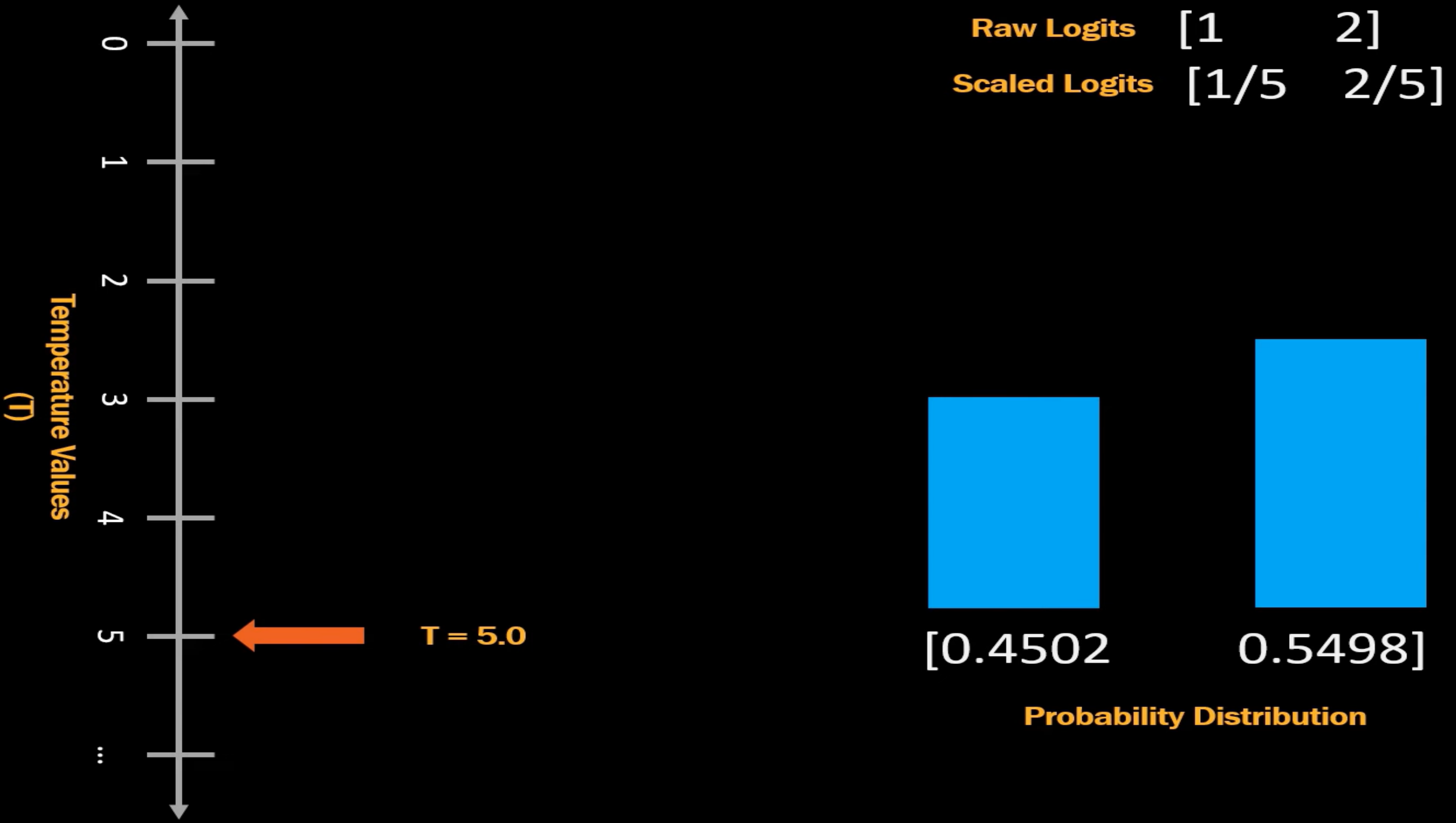
Raw Logits [1 2]

Scaled Logits [1/0.5 2/0.5]



[0.1192 0.8808]

Probability Distribution


Raw Logits [1 2]
Scaled Logits [1/5 2/5]
Temperature Values (T)
0
1
2
3
4
5
...
T = 5.0
[0.4502 0.5498]
Probability Distribution

0
1
2
3
4
5
...
Temperature Values
(T)
Probability Distribution

Temperature Scaling

- Low temperature: More confident and less random
- High temperature: Less confident and high random (because all tokens has almost equal probability)
- It is crucial to choose appropriate temperature value to get the most suitable response.

Top K Sampling

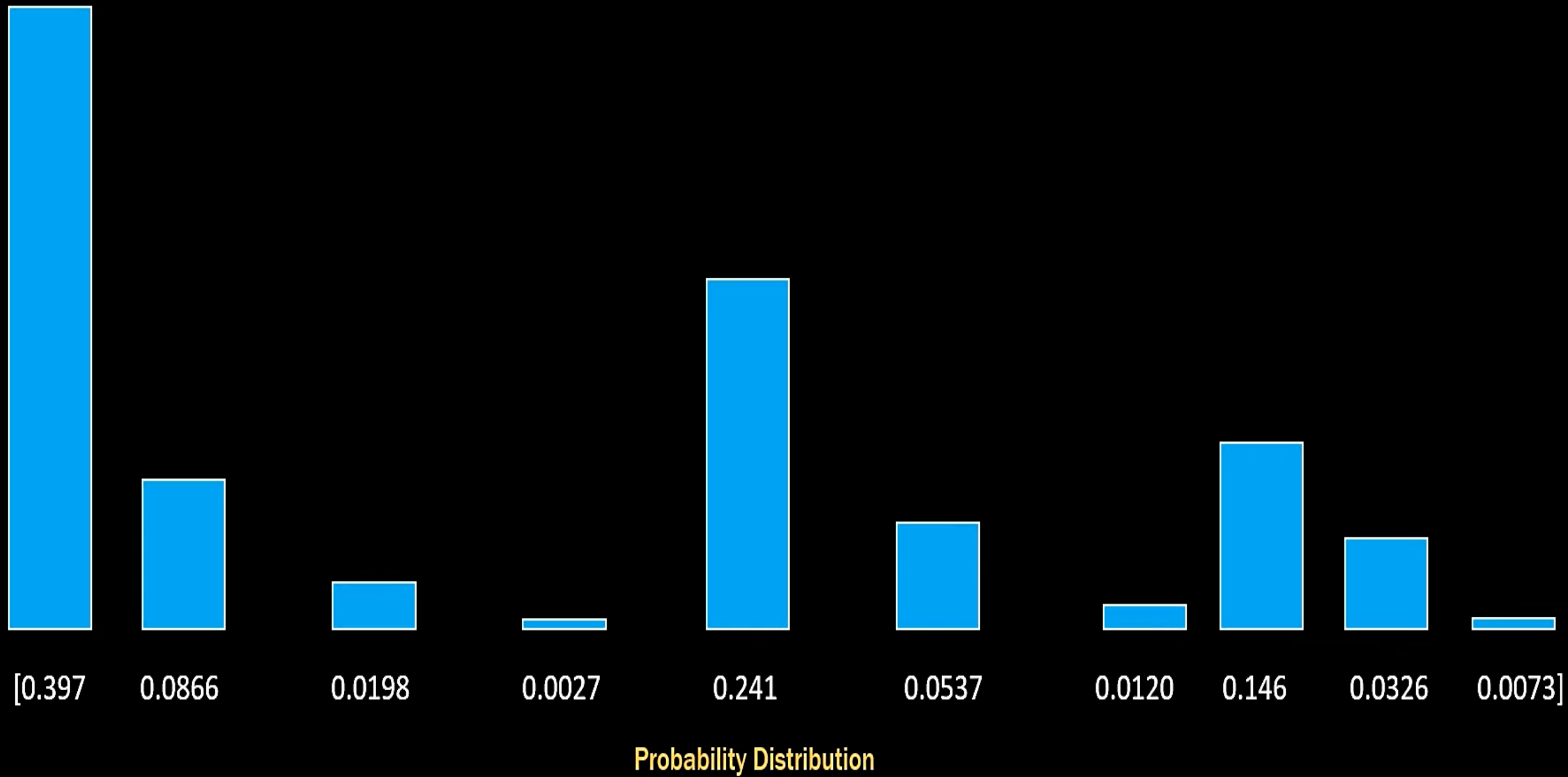
Raw Logits of 10 Tokens

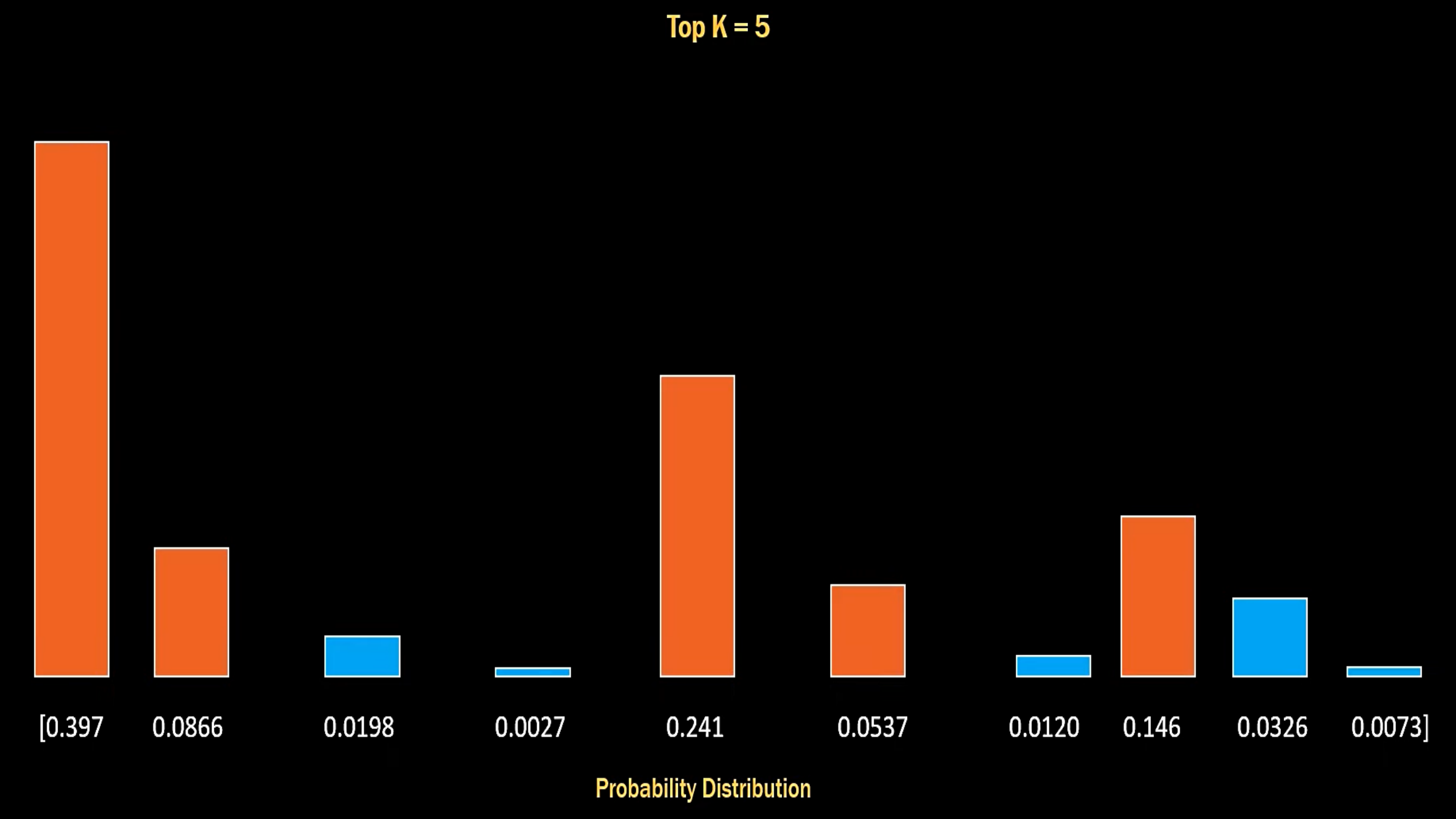
[3.0 1.5 0.0 -2.0 2.5 1.0 -0.5 2.0 0.5 -1.0]

$$softmax = \frac{e^{xi}}{\sum_j e^{xj}}$$

[0.397 0.0866 0.0198 0.0027 0.241 0.0537 0.0120 0.146 0.0326 0.0073]

Probability Distribution


[0.397
0.0866
0.0198
0.0027
0.241
0.0537
0.0120
0.146
0.0326
0.0073]
Probability Distribution


Top K = 5
[0.397
0.0866
0.0198
0.0027
0.241
0.0537
0.0120
0.146
0.0326
0.0073]
Probability Distribution


Top K = 5
3.0
Token A
42.85%
1.5
Token B
9.56%
2.5
Token E
25.99%
1.0
Token F
5.80%
2.0
Token G
15.78%
Logits

Top K Sampling

- Limits the sampling distribution
- Only keeps the probable token so more accurate responses
- Can use in combination with temperature for controlling randomness

Top P Sampling (Nucleus Sampling)

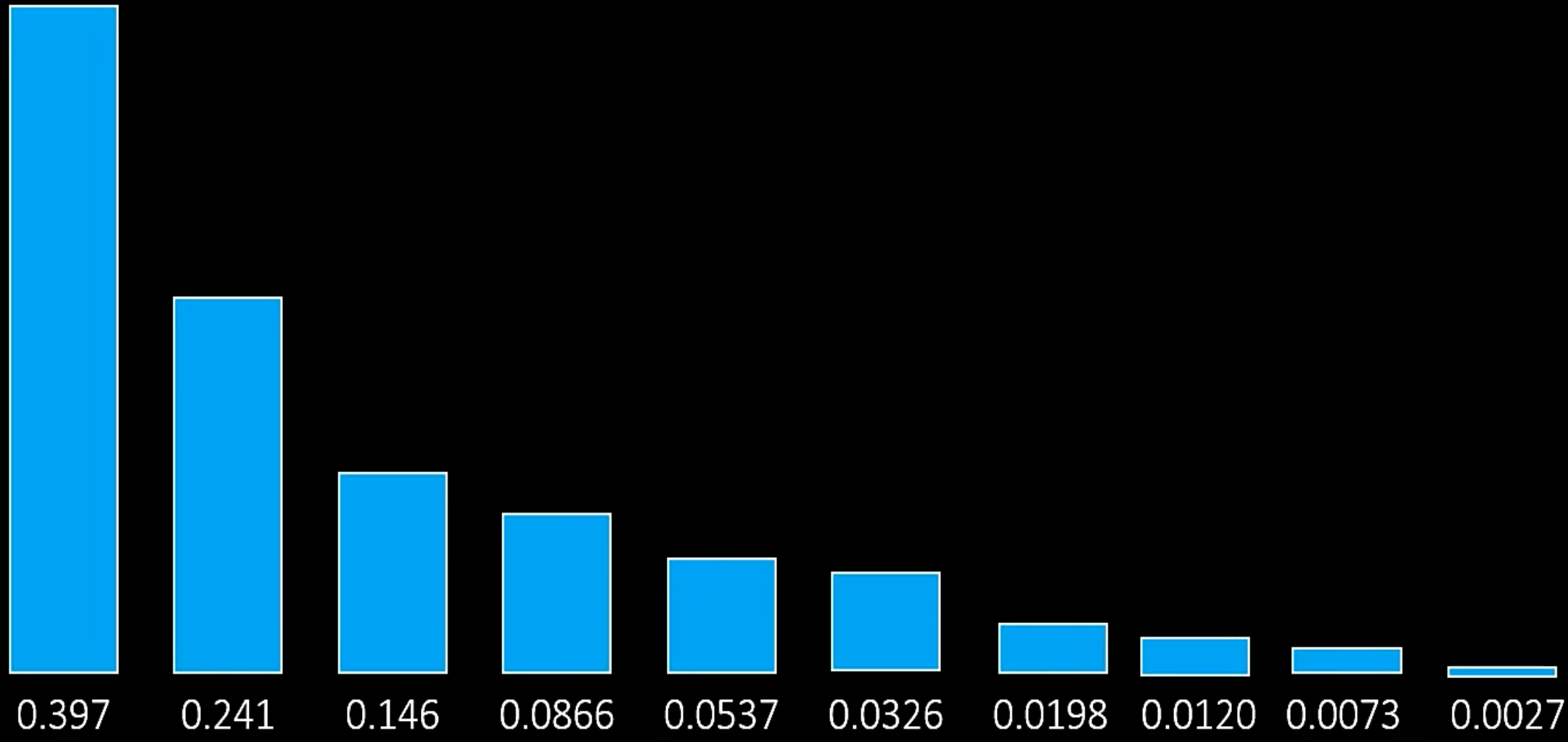
Raw Logits of 10 Tokens

[3.0 1.5 0.0 -2.0 2.5 1.0 -0.5 2.0 0.5 -1.0]

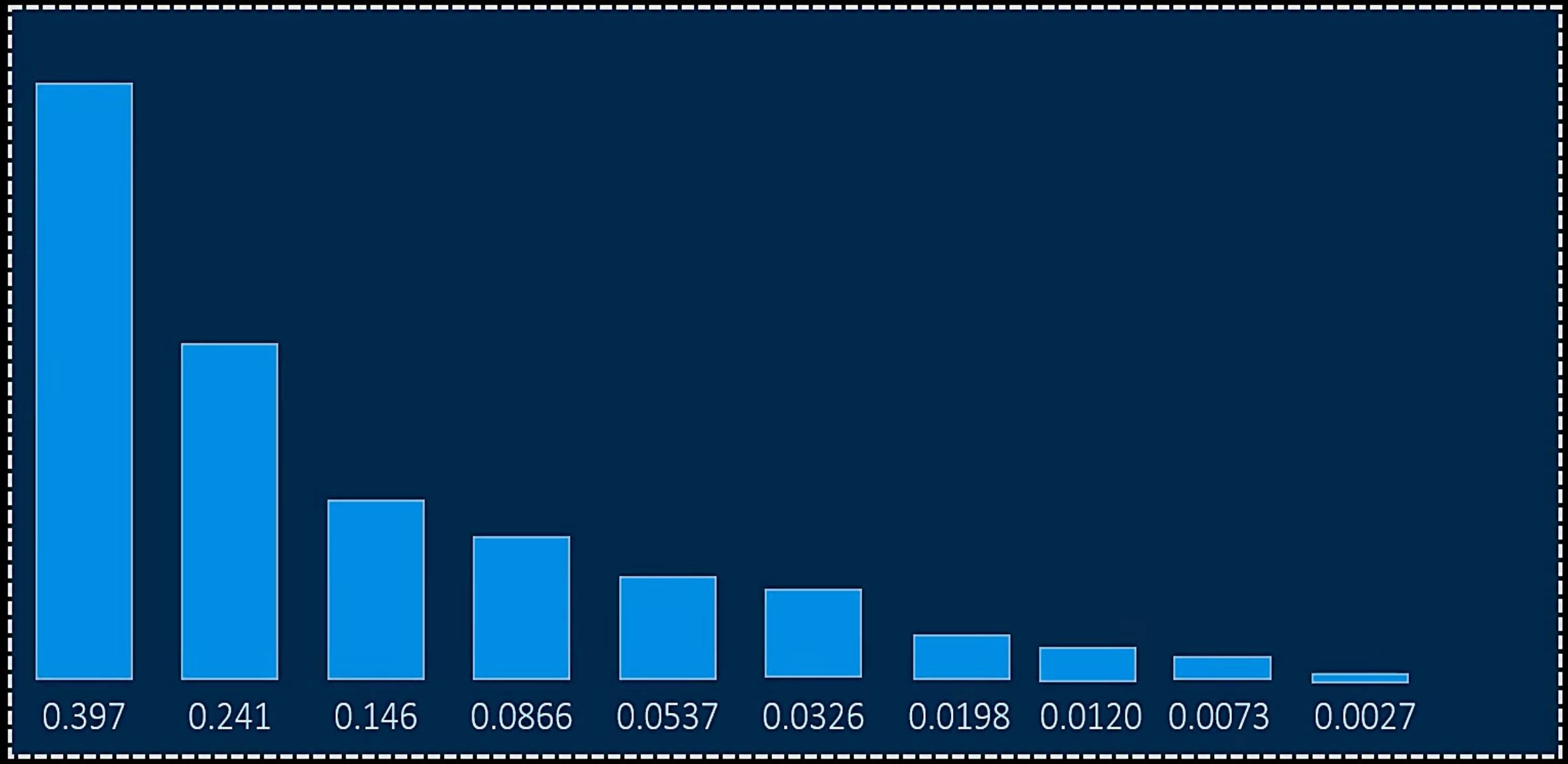
$$softmax = \frac{e^{xi}}{\sum_j e^{xj}}$$

[0.397 0.0866 0.0198 0.0027 0.241 0.0537 0.0120 0.146 0.0326 0.0073]

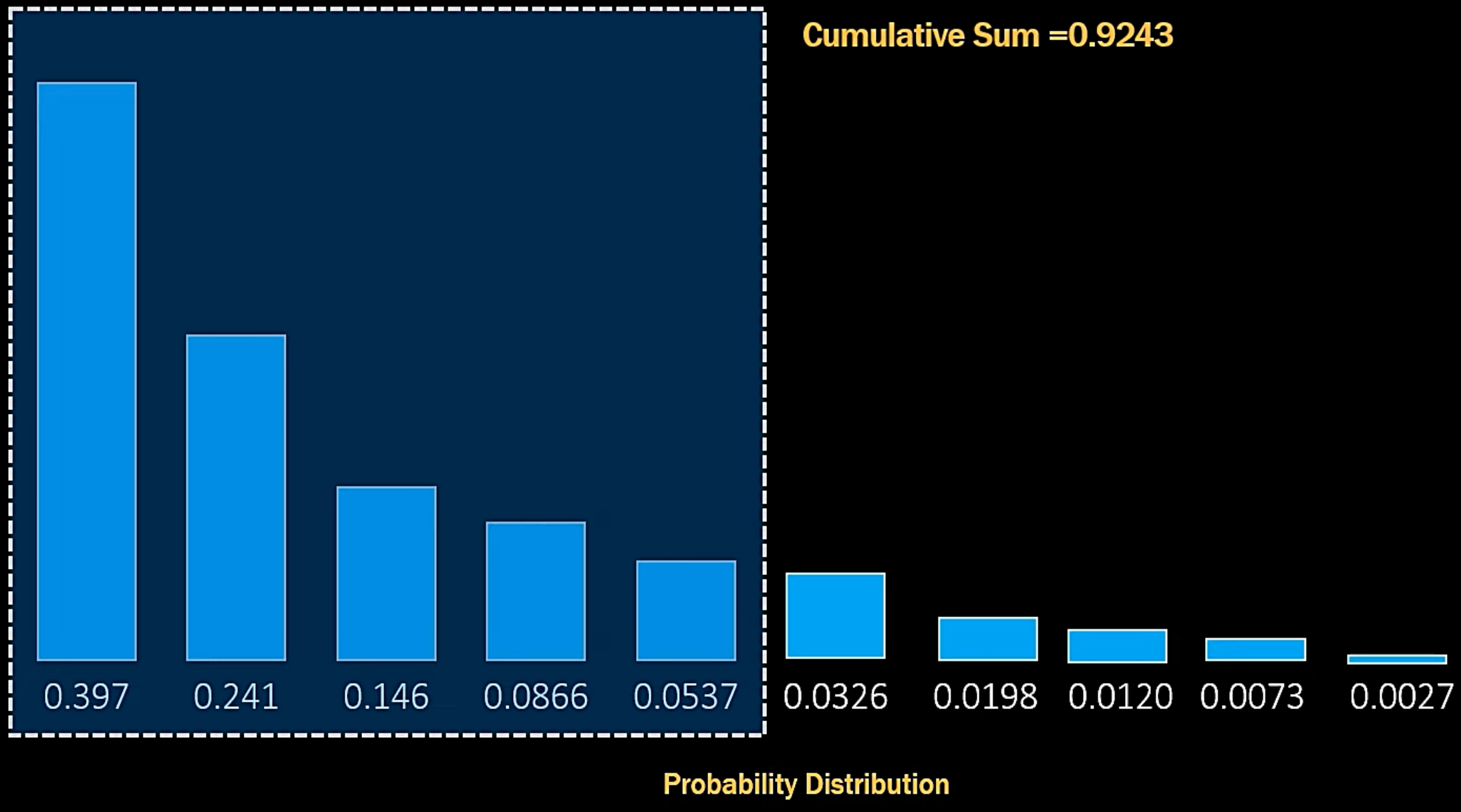
Probability Distribution

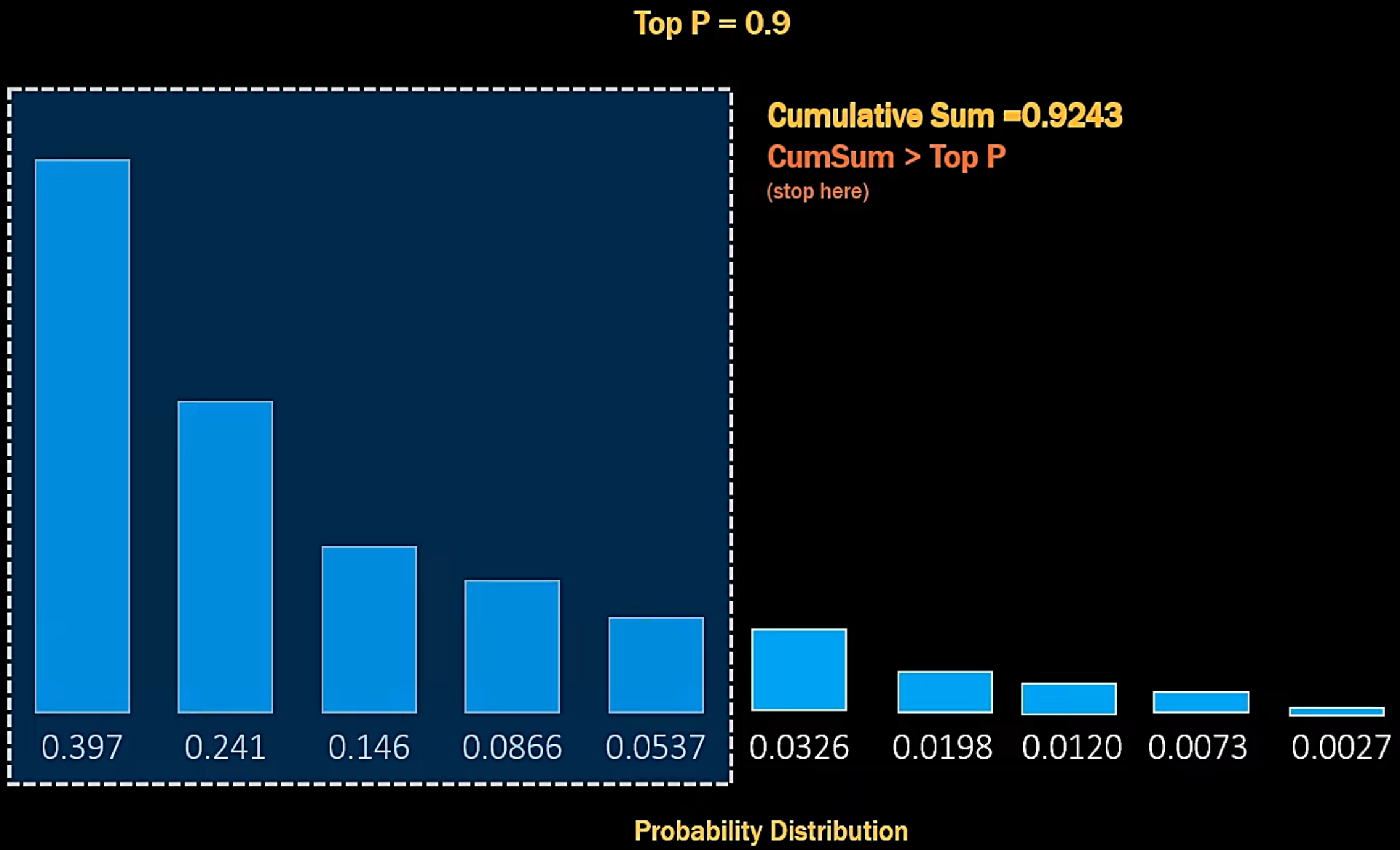

0.397
0.241
0.146
0.0866
0.0537
0.0326
0.0198
0.0120
0.0073
0.0027
Probability Distribution

Top P = 1.0 (By Default)



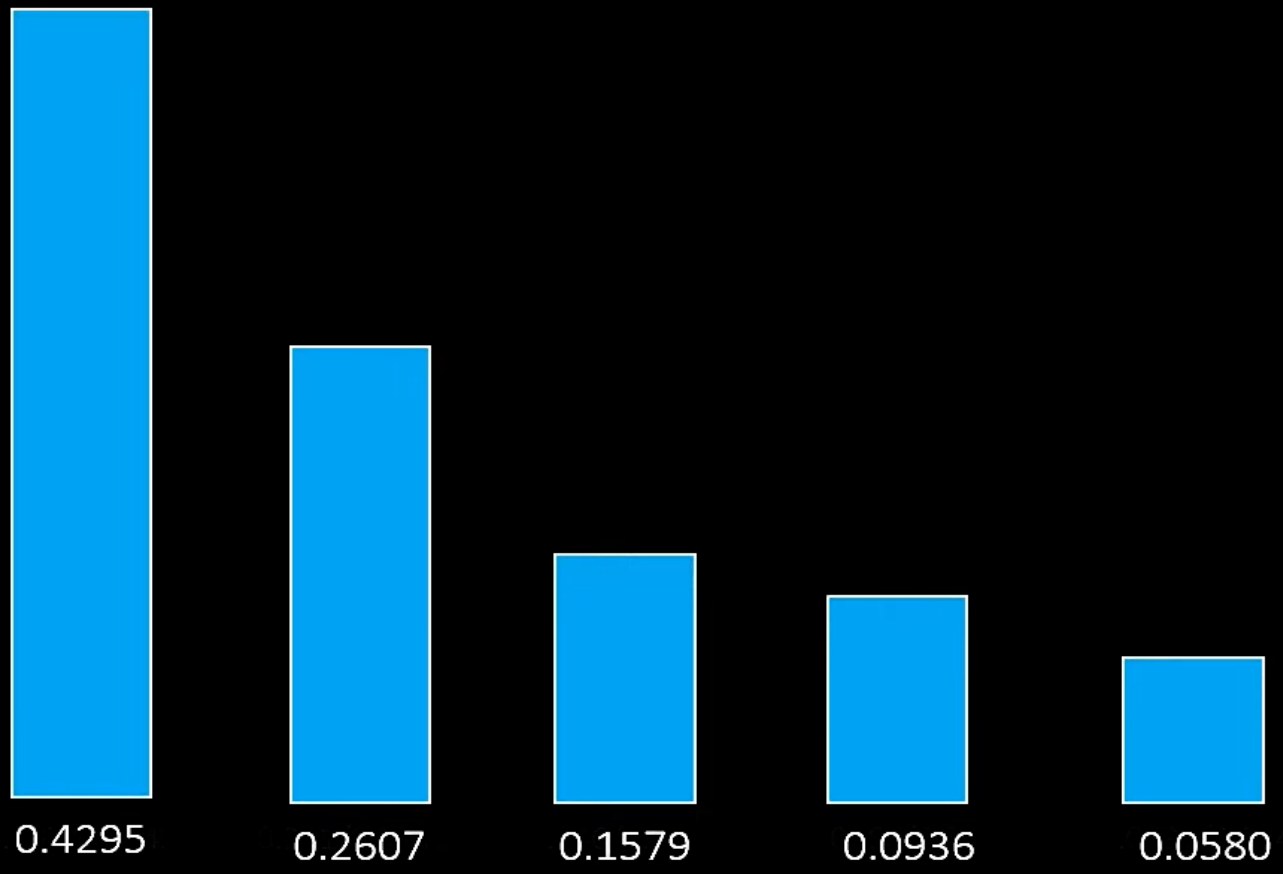
Probability Distribution


Top P = 0.9
Cumulative Sum =0.9243
0.397
0.241
0.146
0.0866
0.0537
0.0326
0.0198
0.0120
0.0073
0.0027
Probability Distribution


Top P = 0.9
Cumulative Sum =0.9243
CumSum > Top P
(stop here)
0.397
0.241
0.146
0.0866
0.0537
0.0326
0.0198
0.0120
0.0073
0.0027
Probability Distribution

Top P = 0.9

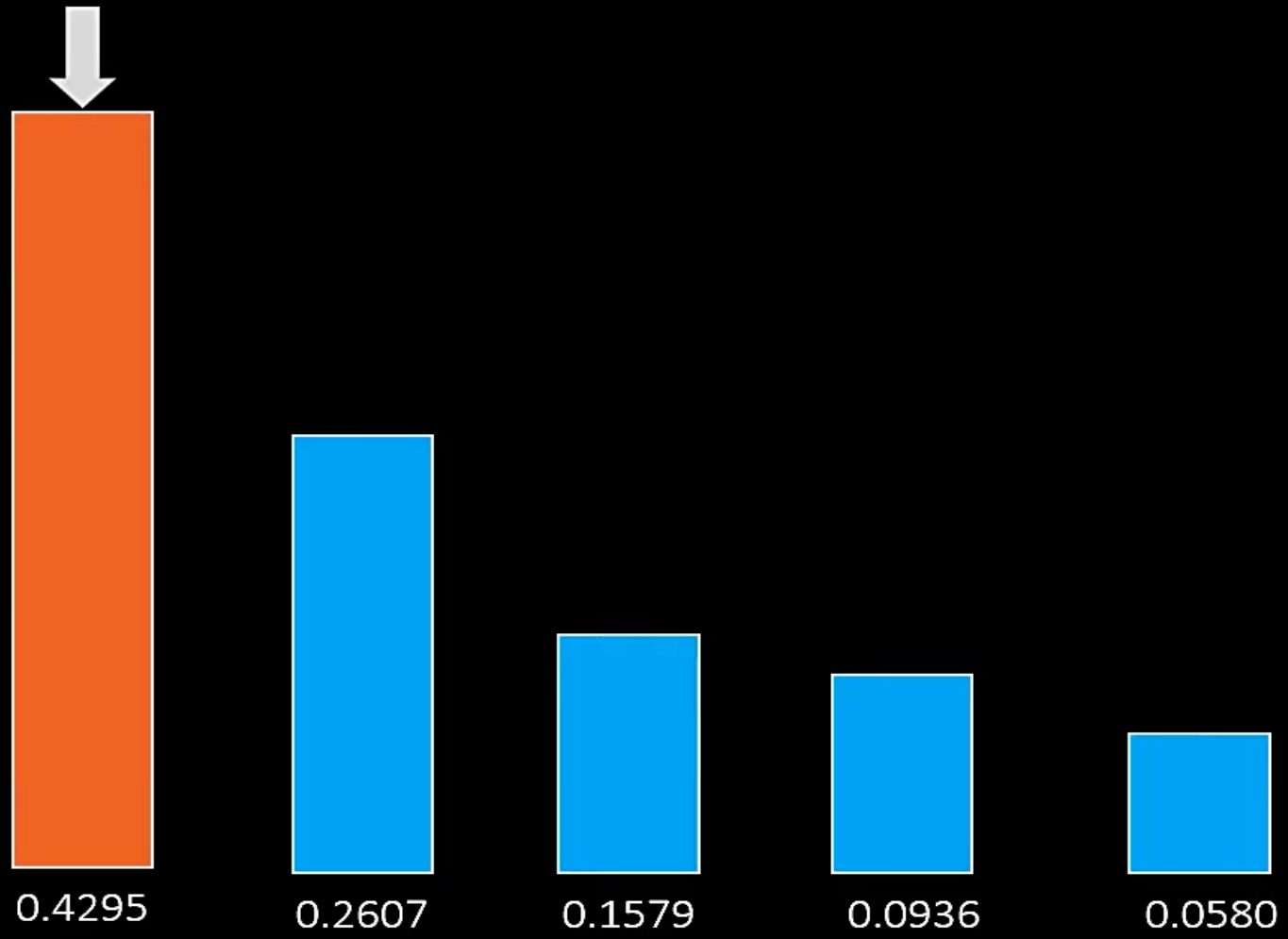
Cumulative Sum =0.9243



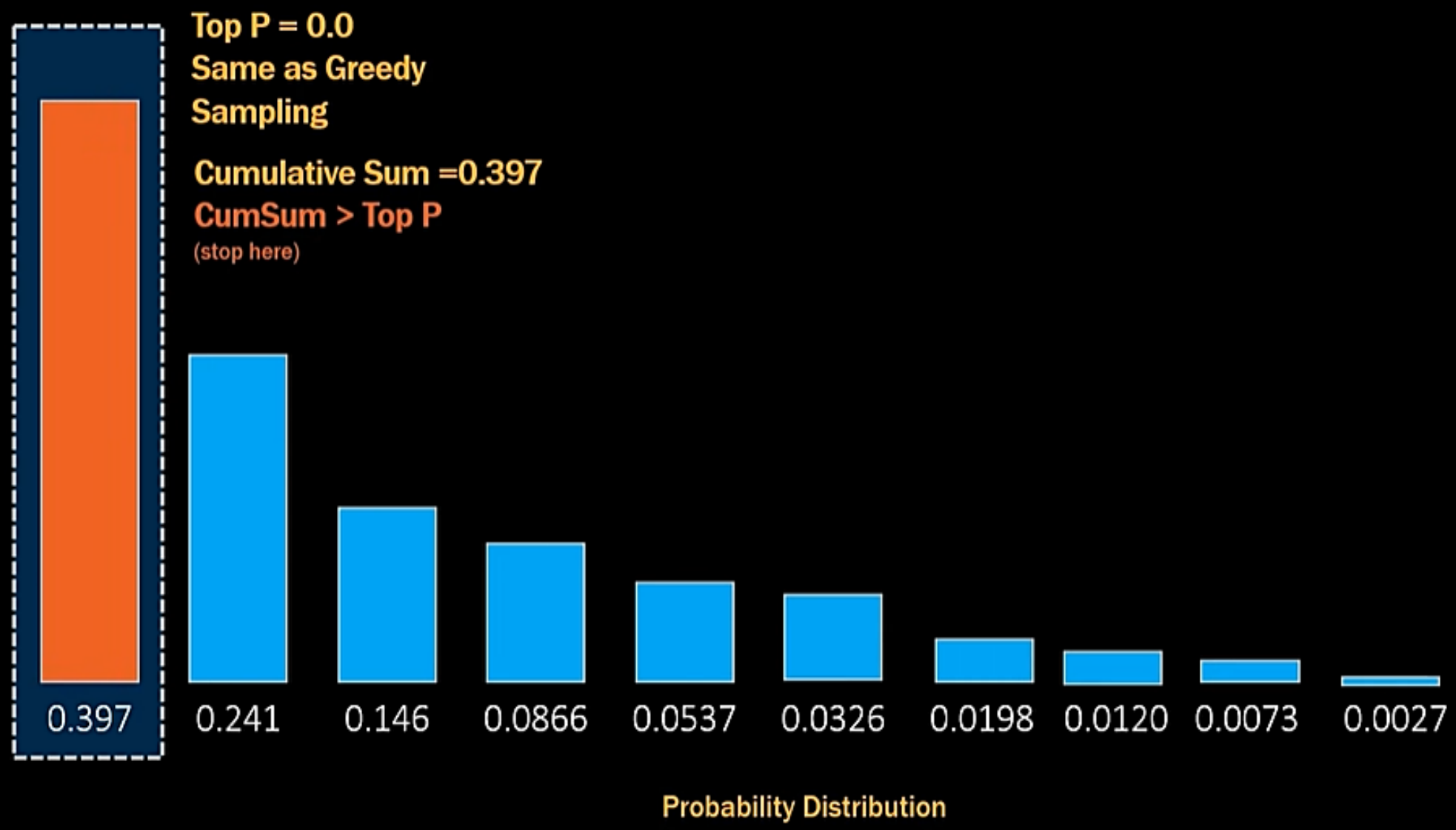
New Distribution
(Pi/CumSum)

Top P = 0.9

Cumulative Sum =0.9243

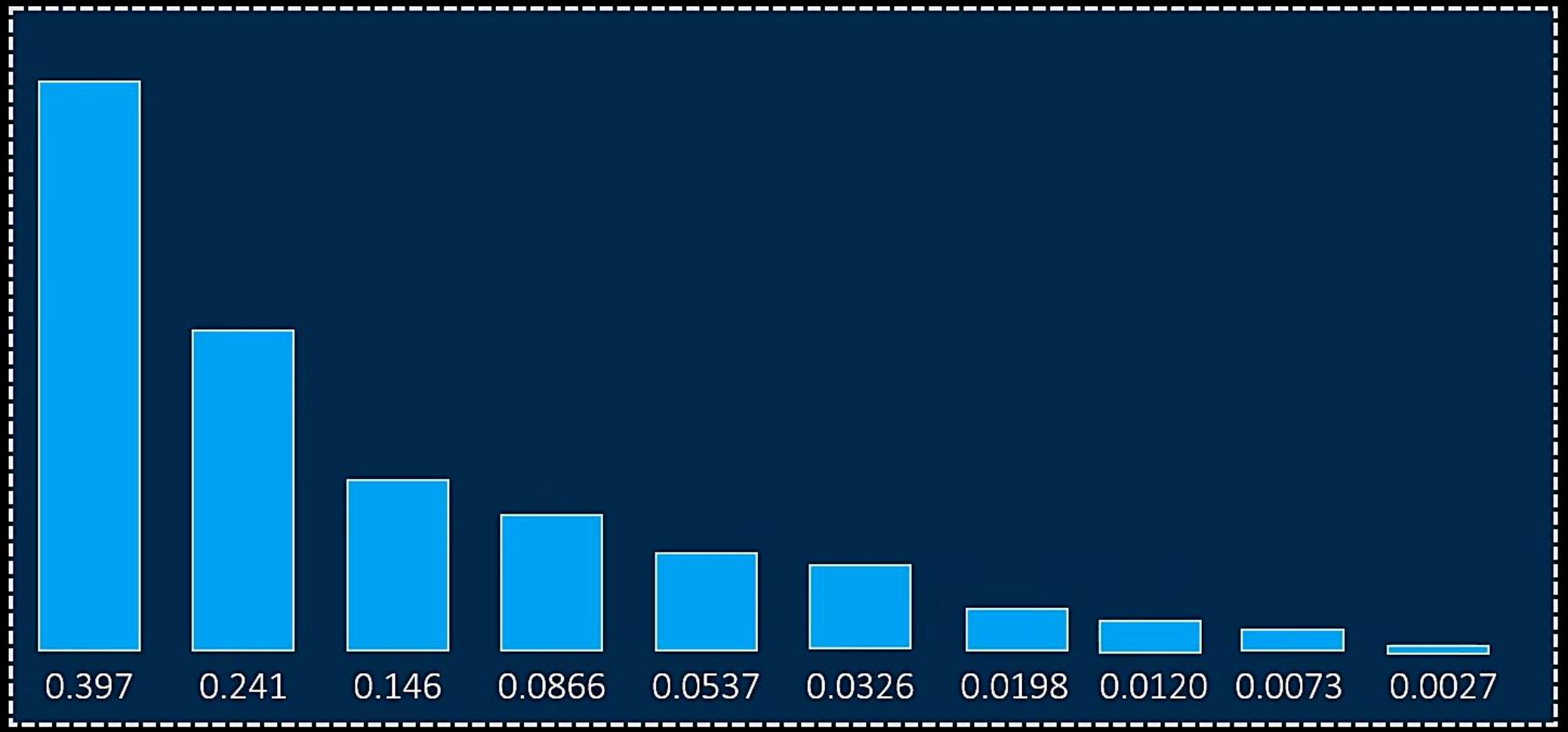


New Distribution
(Pi/CumSum)


Top P = 0.0
Same as Greedy
Sampling
Cumulative Sum =0.397
CumSum > Top P
(stop here)
0.397
0.241
0.146
0.0866
0.0537
0.0326
0.0198
0.0120
0.0073
0.0027
Probability Distribution

Top P = 1.0
Same as Random Sampling From all tokens

Cumulative Sum = 1.0
Includes all the tokens

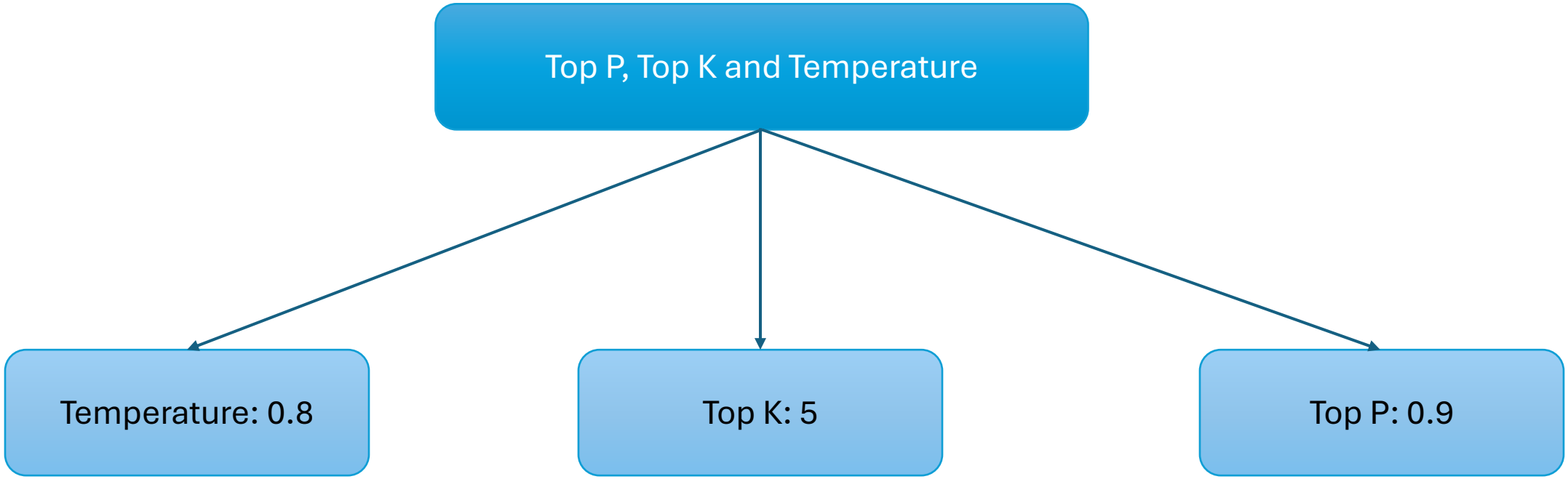


Probability Distribution

Top P Sampling

- Limits the sampling distribution to top P probability
- Dynamically limits the total token to be sampled
- Can use in combination with temperature for controlling randomness

Top P, Top K and Temperature

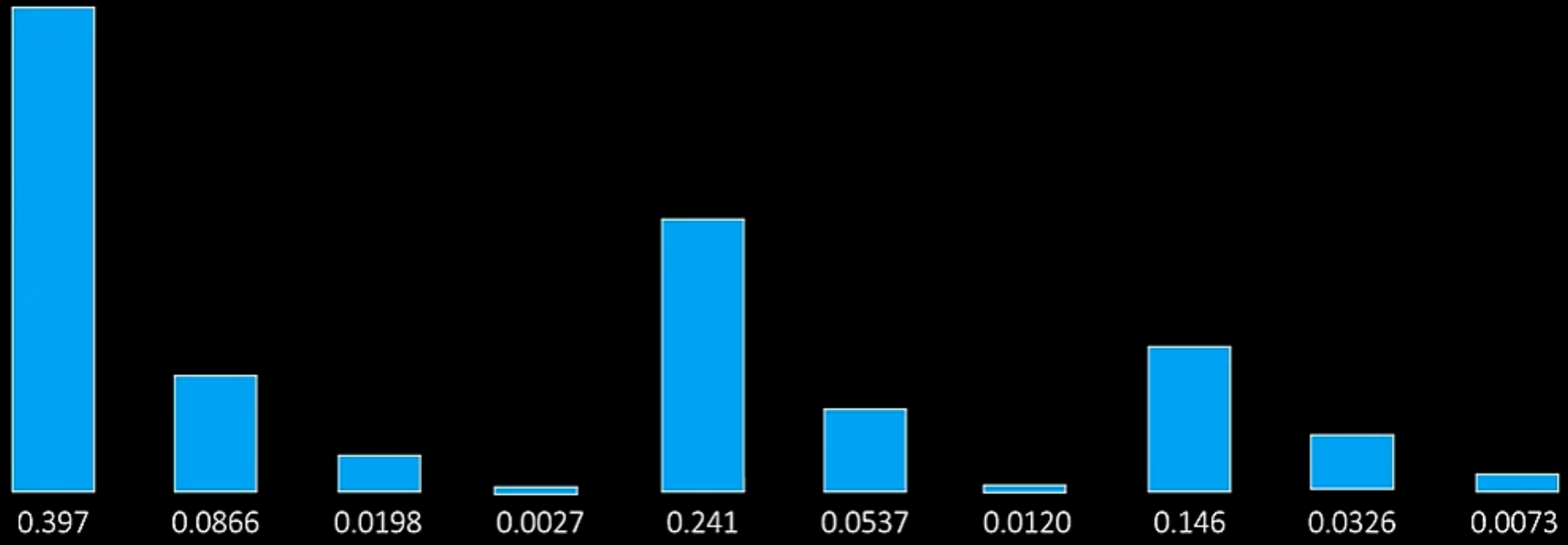


Raw Logits of 10 Tokens

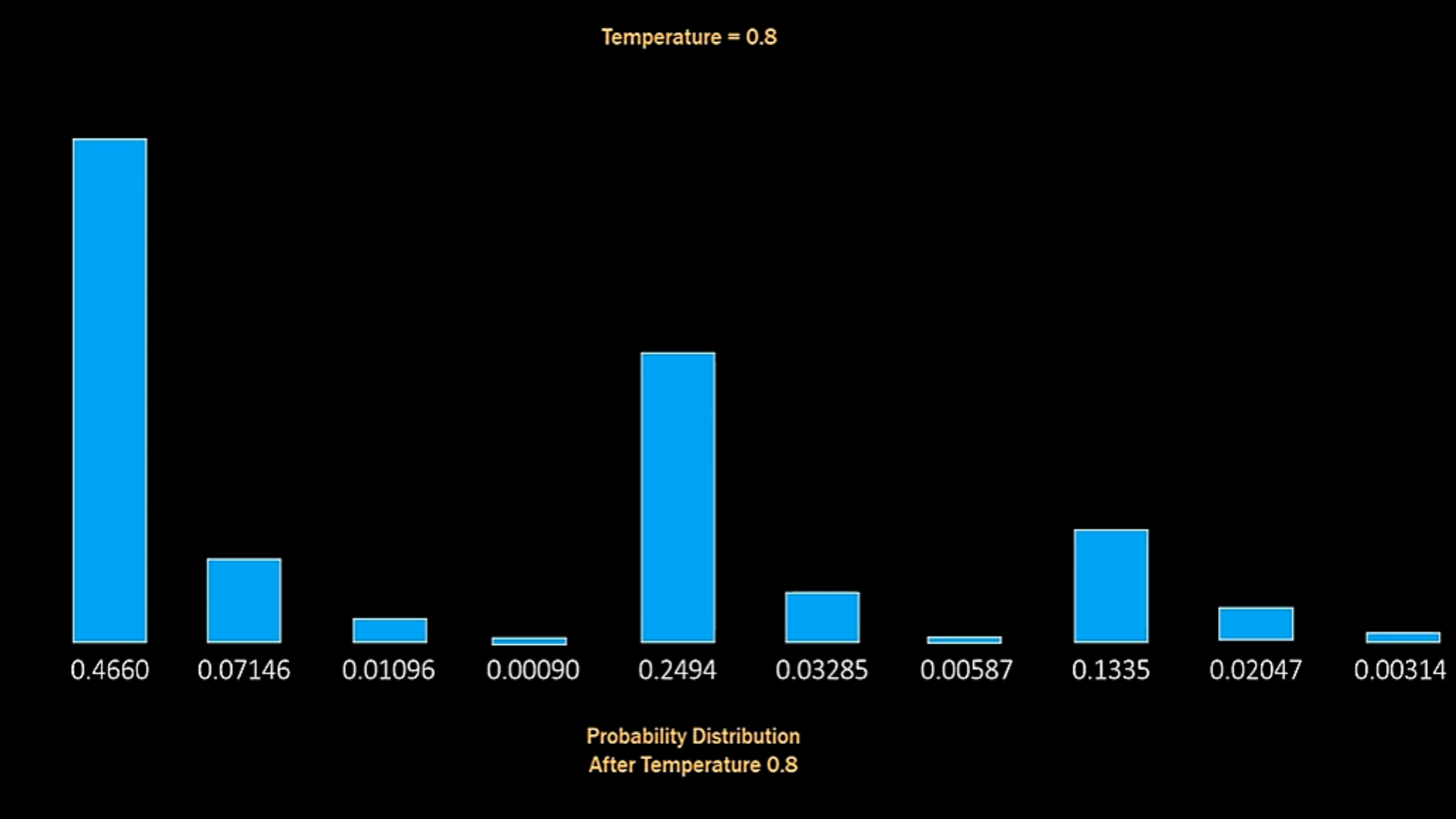
[3.0 1.5 0.0 -2.0 2.5 1.0 -0.5 2.0 0.5 -1.0]

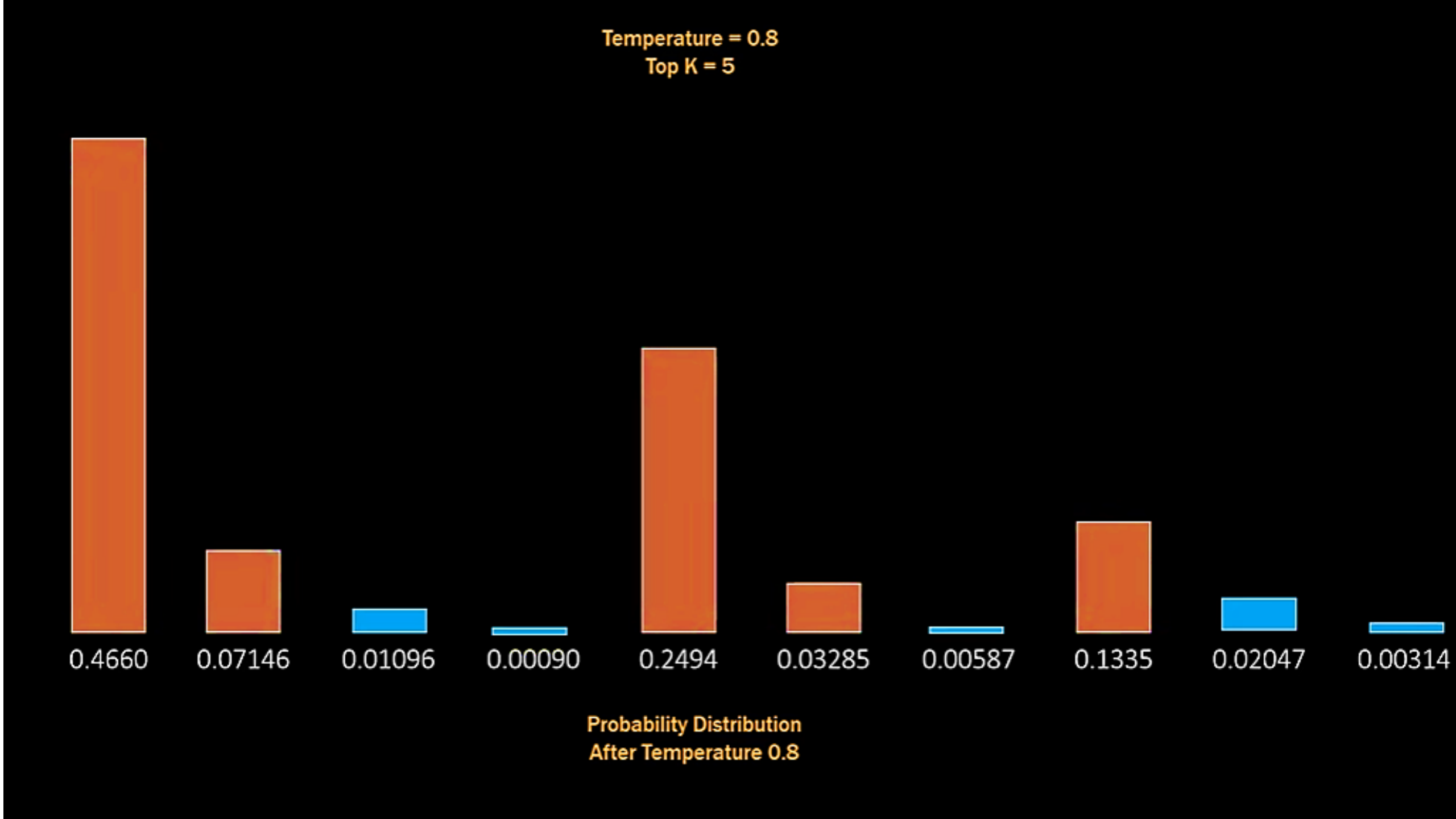
Scale by Temperature
(T = 0.8)

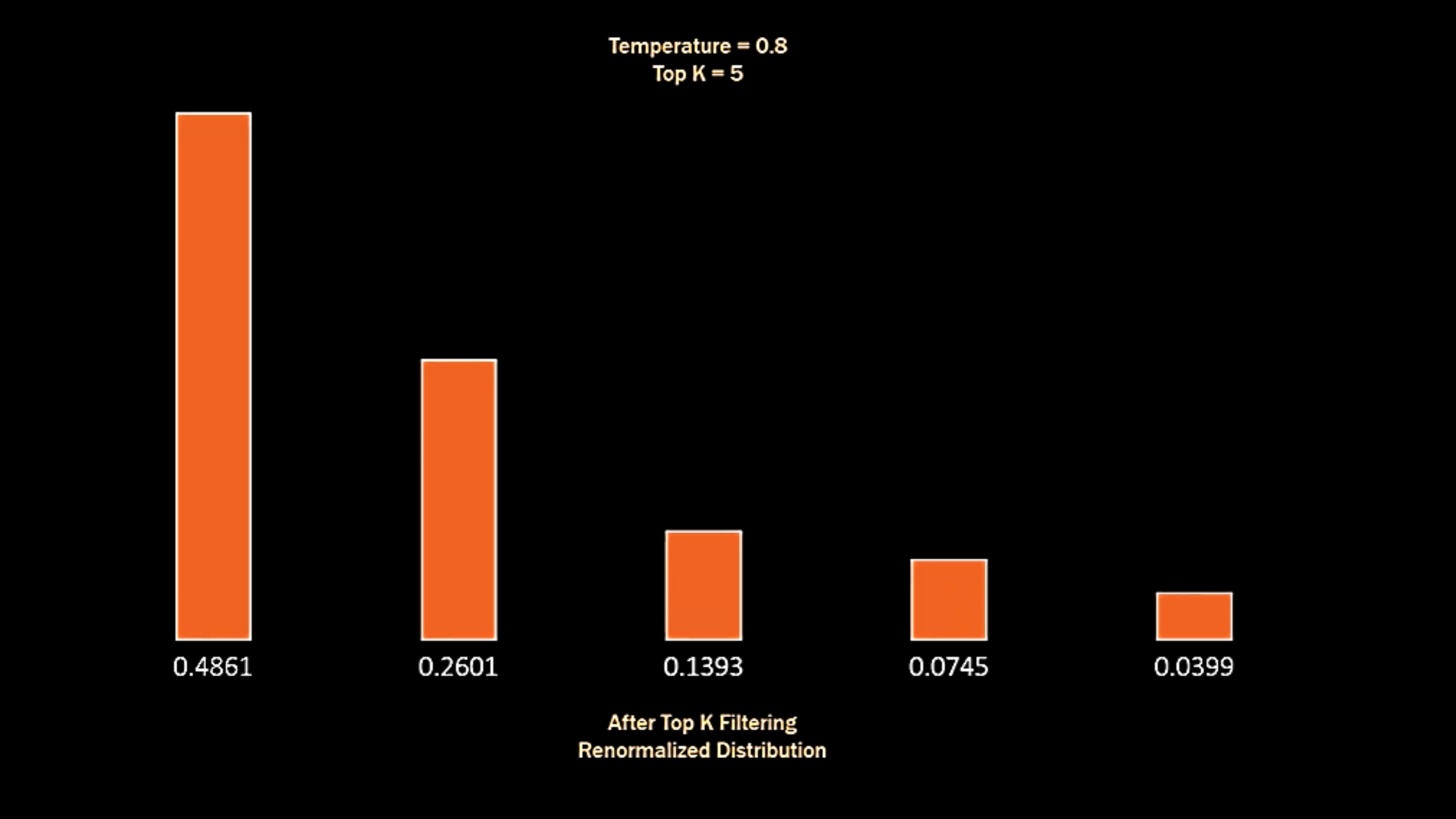
[3.0/0.8 1.5/0.8 0.0/0.8 -2.0/0.8 2.5/0.8 1.0/0.8 -0.5/0.8
2.0/0.8 0.5/0.8 -1.0/0.8]



Probability Distribution

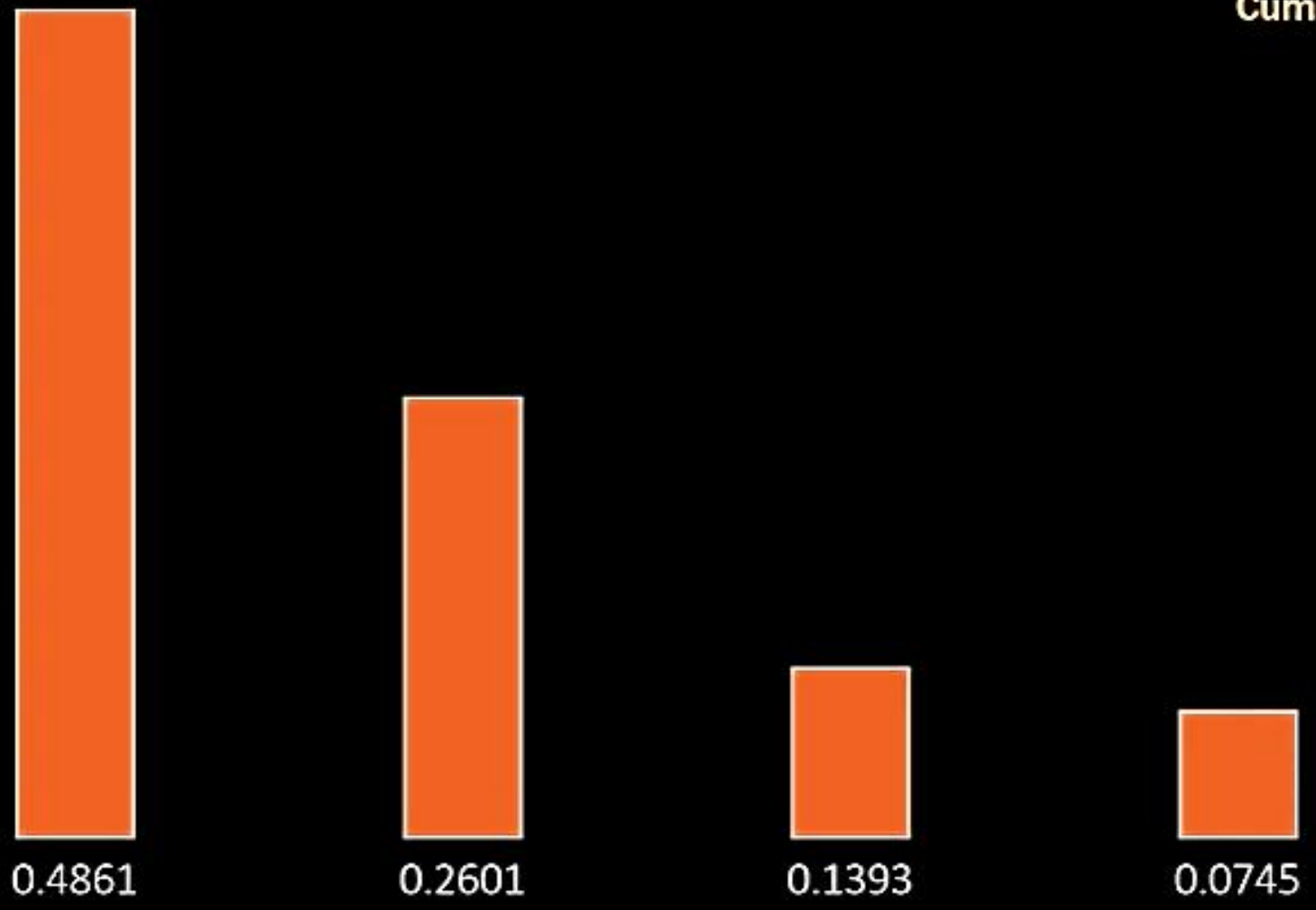

Temperature = 0.8
0.4660
0.07146
0.01096
0.00090
0.2494
0.03285
0.00587
0.1335
0.02047
0.00314
Probability Distribution
After Temperature 0.8


Temperature = 0.8
Top K = 5
0.4660
0.07146
0.01096
0.00090
0.2494
0.03285
0.00587
0.1335
0.02047
0.00314
Probability Distribution
After Temperature 0.8

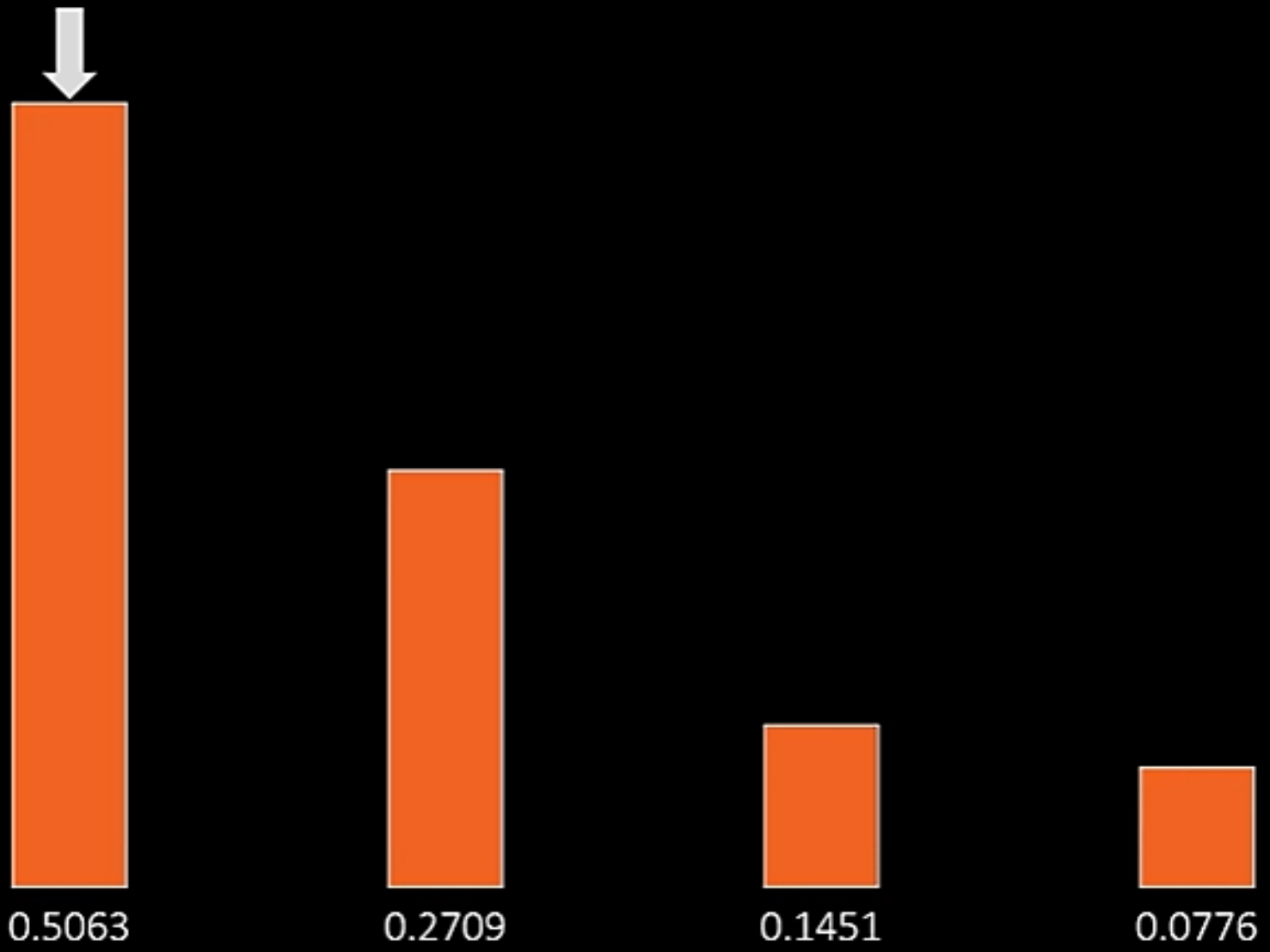

Temperature = 0.8
Top K = 5
0.4861
0.2601
0.1393
0.0745
0.0399
After Top K Filtering
Renormalized Distribution

Temperature = 0.8
Top K = 5
Top P = 0.9

Cumulative Sum = 0.96

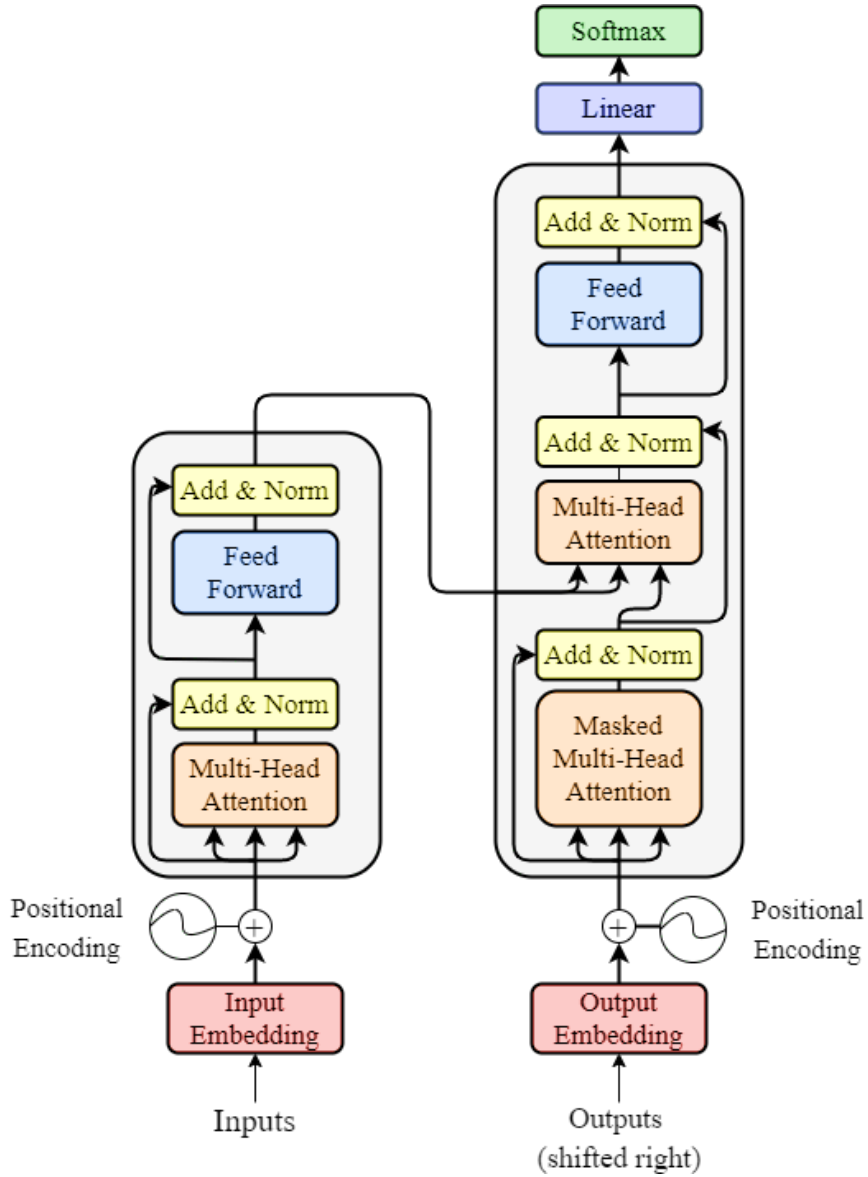


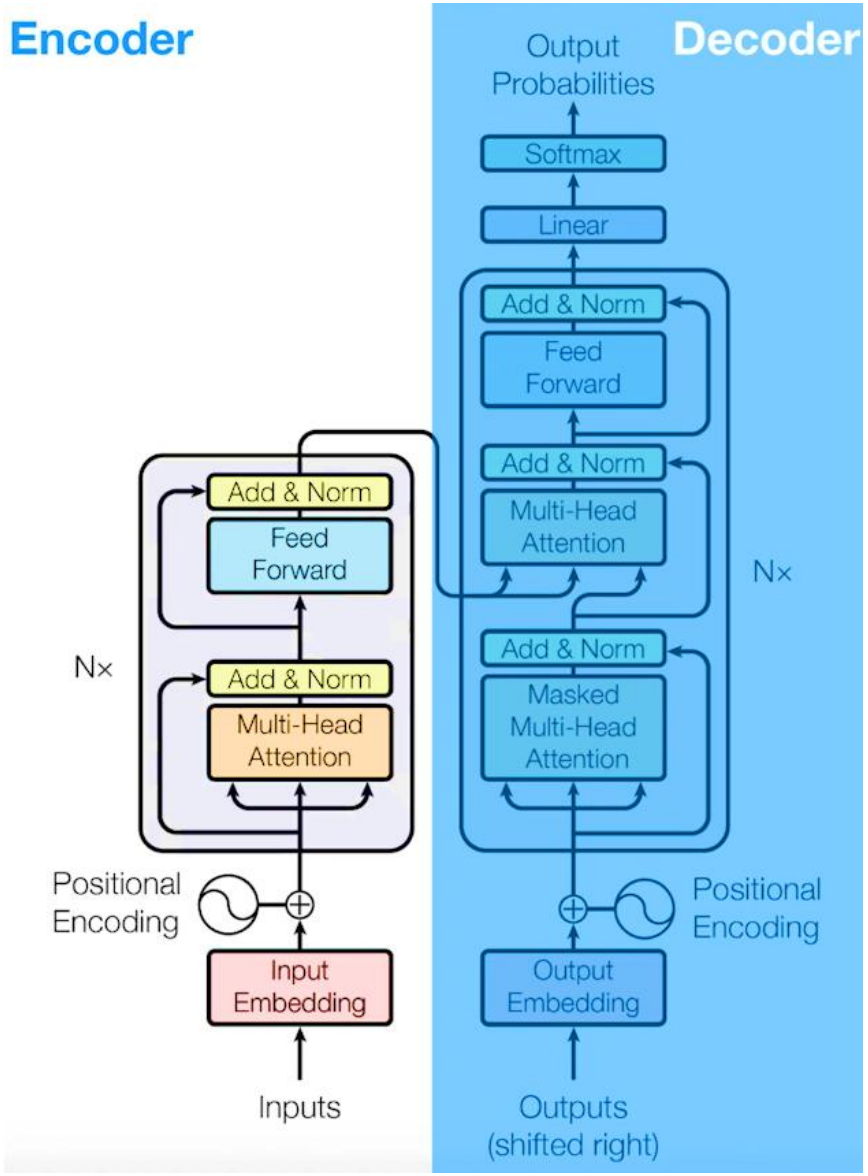
After Top P Filtering



Final Tokens to Sample

Transformer

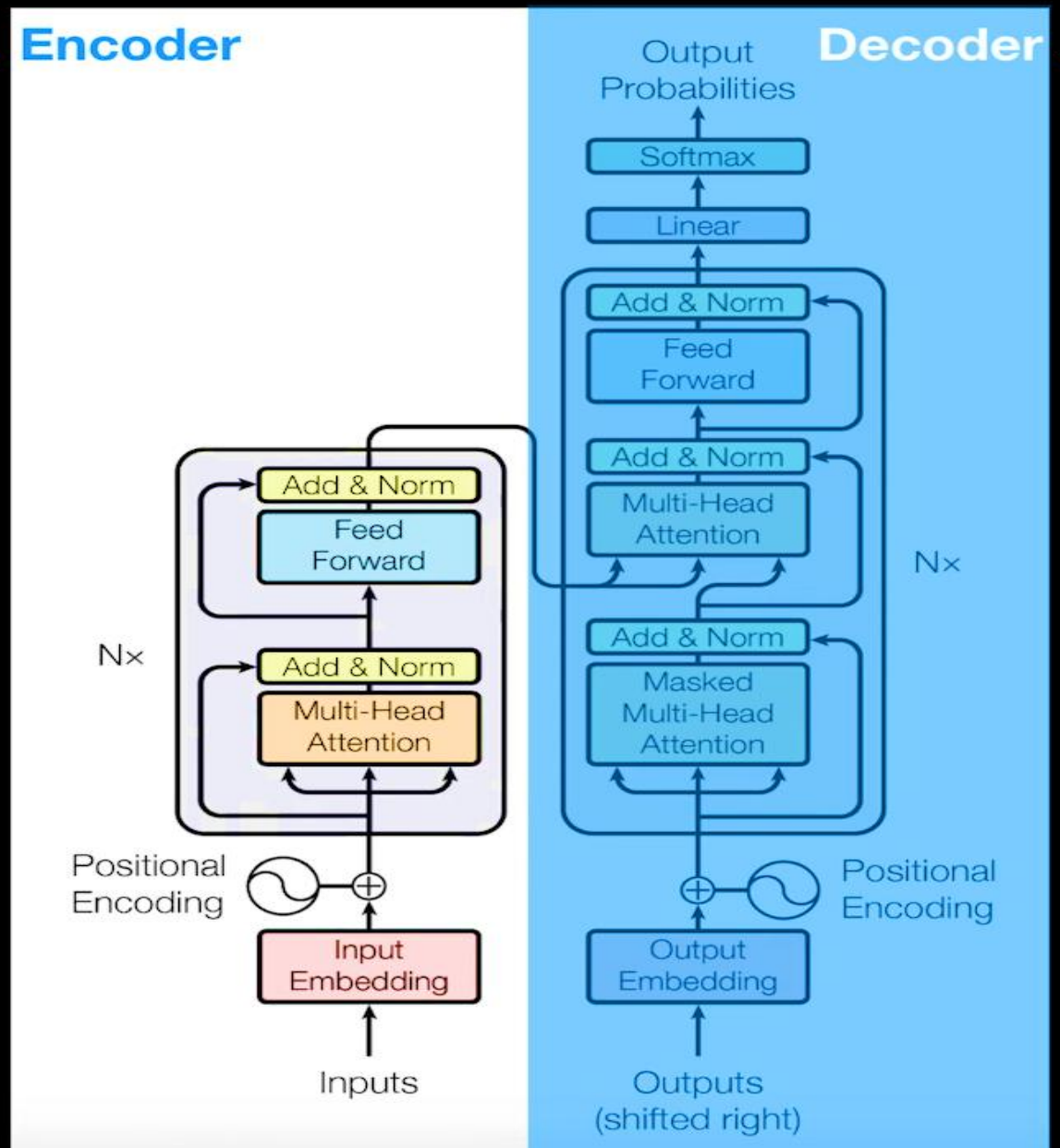

Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

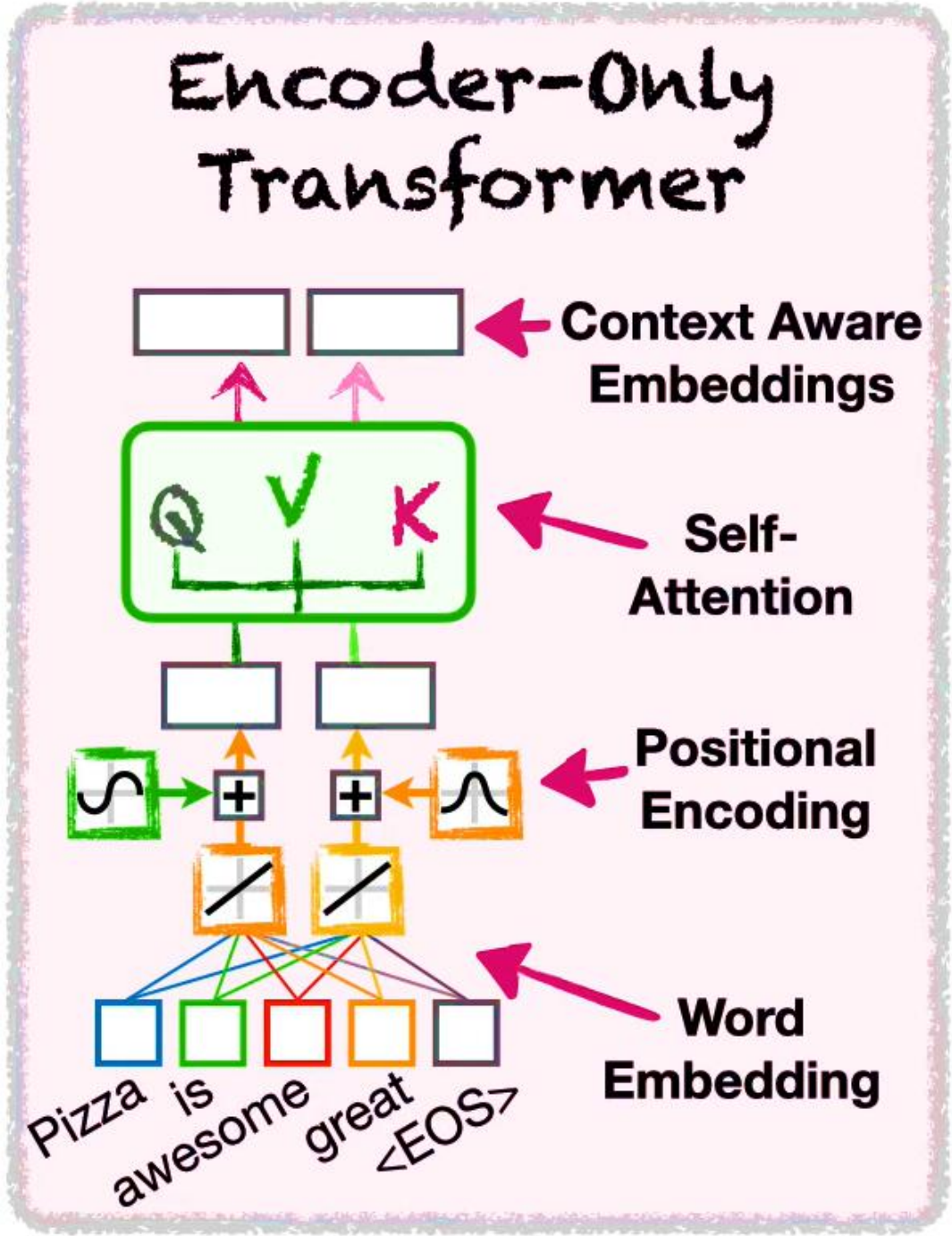

Encoder
Decoder
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Transformer

BERT uses Encoder part only

GPT uses Decoder part only




Encoder-Only Transformer
Context Aware Embeddings
Q
V
K
Self-Attention
Positional Encoding
Word Embedding
Pizza
is
awesome
great
<EOS>

zero shot learning

zero shot learning

zero shot learning

zero shot learning

One shot learning

One shot learning

Few shot learning

Few shot learning

Few shot learning