Part IV
Probability models for sequences

or: why ChatGPT is just another probability model trained mostly by likelihood maximization

QUESTION. What does GPT stand for?

Generative Pre-trained Transformer

Generative: prob. model

Pre-trained: parameters learnt by mle

[Lecture notes for this part are available on the course website]

A piece of text is a sequence of tokens from a finite alphabet.

ChatGPT-4o uses an alphabet of ≈200k tokens.

The following is a classic Chinese poem from the Tang dynasty, translated into English.

The dawn light strikes the head of my bed
I see leaves

TEXT TOKEN IDS

[464, 1708, 318, 257, 6833, 3999, 21247, 422, 262, 18816, 30968, 11, 14251, 656, 3594, 13, 198, 198, 464, 17577, 1657, 8956, 262, 1182, 286, 616, 3996, 198, 40, 766, 5667, 220]

TEXT TOKEN IDS

GPT tokenizer: https://platform.openai.com/tokenizer

A piece of text is a sequence of tokens from a finite alphabet.

How might we generate a random piece of text that looks like English?

- Write a piece of text of length ℓ as $\underline{x} = x_0 x_1 x_2 \cdots x_\ell$
- We want code to generate a random text $\underline{X}$

```
def X():
  ???
```

- Our code should have learnable parameters, call them θ
- We'll collect a corpus of documents $\{\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots, \underline{x}^{(n)}\}$ and tune θ to make our code produce outputs similar to these documents

```
def X(θ):
  ???
```

- This is a probability model, and the random variable $\underline{X}$ has a likelihood function $\Pr_{\underline{X}}(\underline{x}; \theta)$
- We can fit the model by likelihood maximization: $\max_{\theta} \sum_i \log \Pr_{\underline{X}}\left(\underline{x}^{(i)}; \theta\right)$

A piece of text is a sequence of tokens from a finite alphabet.

How might we generate a random piece of text that looks like English?

Why might this be interesting?

Text completion:
we give it initial text $\underline{x} = x_0 x_1 \cdots x_m$ and it completes the text.

```
To make the perfect
pasta sauce, first ␣
```

- *This is how we talked with LLMs prior to ChatGPT.*
- *Current LLMs are still based on the same sort of probability models, fine-tuned to work better in chats.*

In probability language, text completion is just sampling from a conditional distribution:

$$(\underline{X} \mid X_0 = x_0, \dots, X_m = x_m)$$

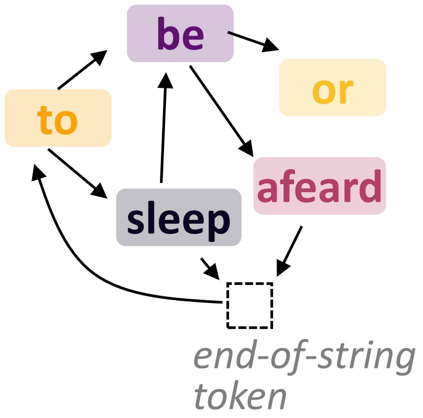
A piece of text is a sequence of tokens from a finite alphabet.

How might we generate a random piece of text that looks like English?

Why might this be interesting?

So, what might this code look like?

- `def X(θ): return ???`
- We also want its likelihood $\Pr_{\underline{X}}(\underline{x};\theta)$ for training



Markov model

Based on a graph of token-to-token transitions.

`“to foreign princes lie in your blessing god who shall have the prince of rome □”`

Probability model: generate $\underline{X}$ by starting at □ and jumping from token to token until we hit □ again.

$$\square \to X_1 \to X_2 \to \cdots \to X_L \to \square$$

Choose the jumps according to a transition probability matrix $\theta \in \mathbb{R}^{W \times W}$:

$\theta \geq 0$, row sums $= 1$

$$\mathbb{P}(X_{n+1} = v | X_n = u) = \theta_{u,v}$$

The likelihood function is easy:

$$\Pr_{\underline{X}}(x_1 x_2 \cdots x_\ell; \theta) = \theta_{\square, x_1} \theta_{x_1, x_2} \cdots \theta_{x_{\ell-1}, x_\ell} \theta_{x_\ell, \square}$$

Andrei Markov (1856–1922)

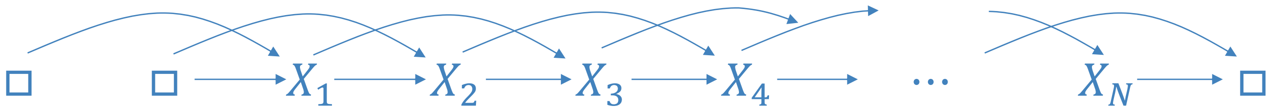
be contented **to be** what they
who is **to be** executed this
in him **to be** truly touched
took occasion **to be** quickly woo'd

Markov's trigram model

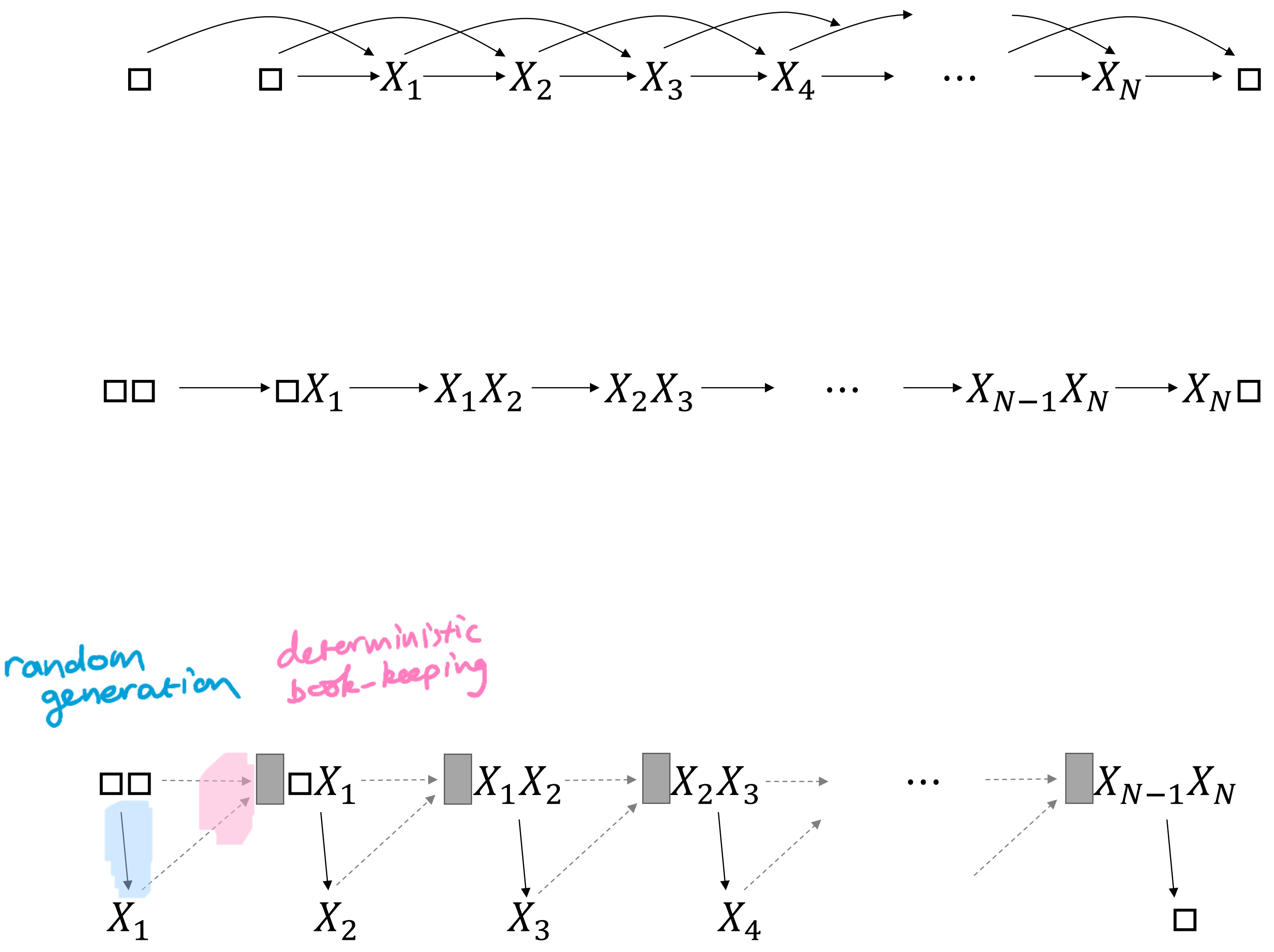
```
"to be wind-shaken we will be glad to receive at
once for the example of thousands □"
```

Probability model: Generate $\underline{X}$ by starting with □□ and repeatedly generating the next word based on the preceding **two**, until we produce □.

$$\Pr_{\underline{X}}(x_1 x_2 \cdots x_n) = \Pr(x_1|□□)\Pr(x_2|□x_1)\Pr(x_3|x_1x_2) \times \cdots \times \Pr(x_n|x_{n-2}x_{n-1})\Pr(□|x_{n-1}x_n)$$

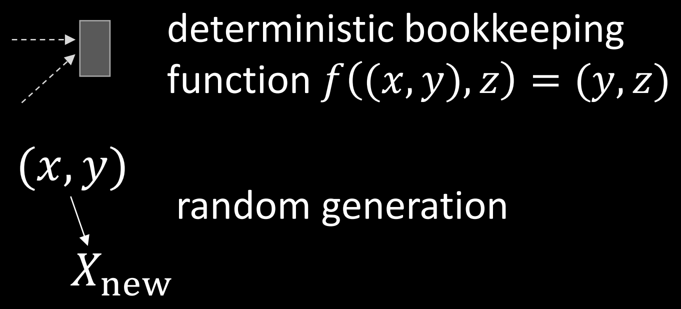


Different ways to write the trigram model:

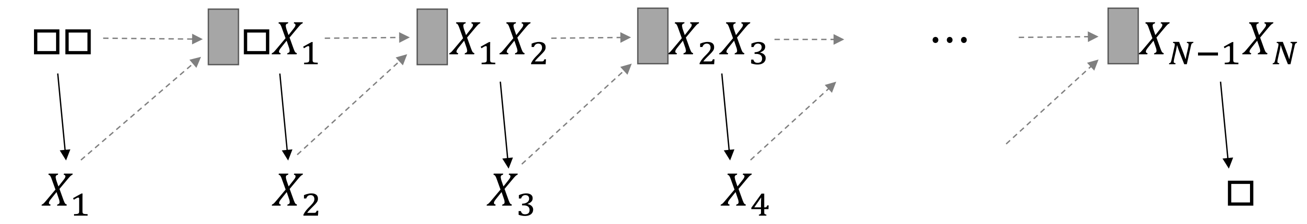


A *Markov Chain* is a sequence in which each item is generated based only on the preceding item.

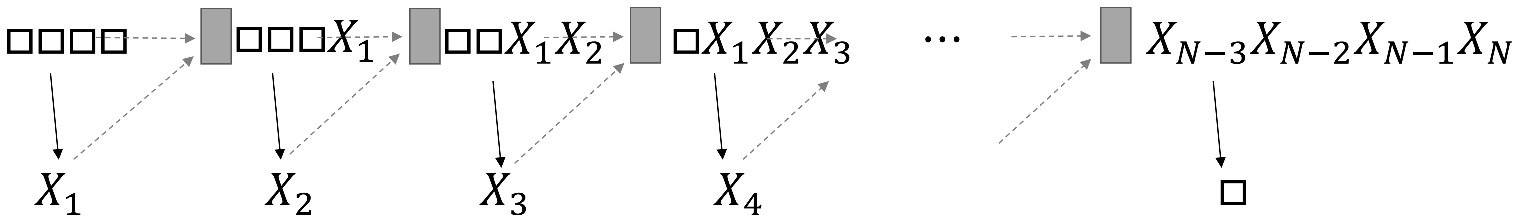
The trigram model is a Markov chain, whose items are word-pairs.



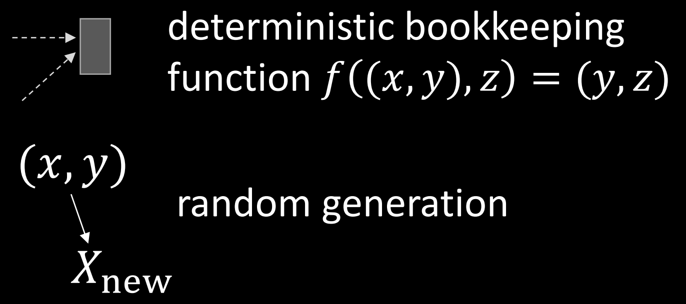
Can we get a better model by using more history?



Trigram character-by-character model trained on Shakespeare:
"on youghtlee for vingiond do my not whow'd no crehout withal deepher forand a but thave a doses?"



5-gram character-by-character model trained on Shakespeare:
"once is pleasurely. though the the with them with comes in hand. good. give and she story tongue."

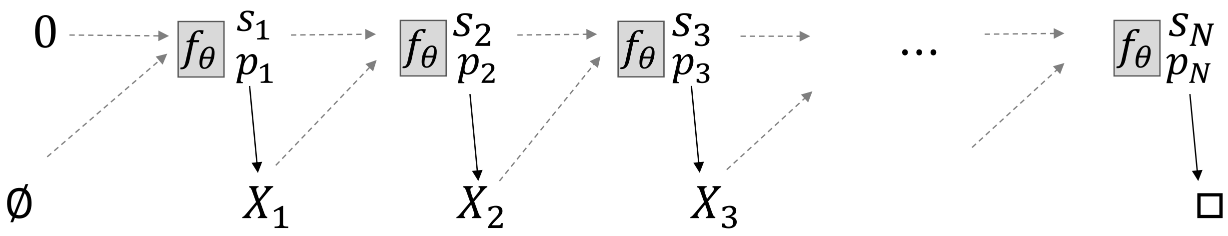


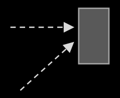
QUESTION. What are the advantages and disadvantages of a long history window?

QUESTION. Can we do better than using a fixed history window?

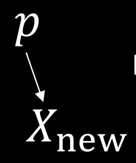
Recurrent Neural Network (RNN)

Let's use a neural network to learn an appropriate history digest.
This is more flexible than choosing a fixed history window.





learnable function
$$f_\theta(s, x) = (p, s_{\text{new}})$$



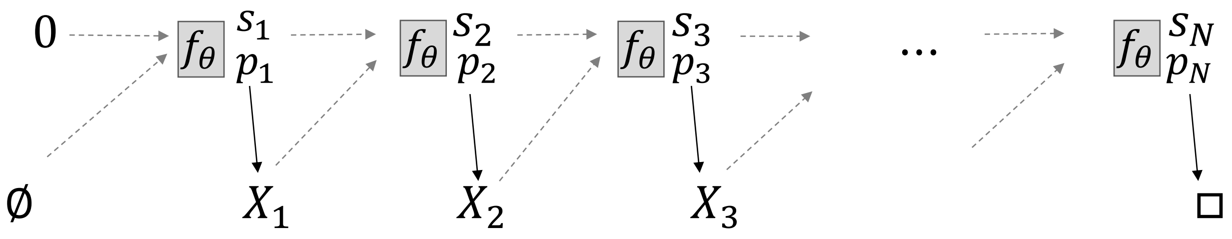
random generation
$$X_{\text{new}} \sim \text{Cat}(p)$$
i.e. $\mathbb{P}(X_{\text{new}} = x) = [p]_x$

RNN character-by-character model trained on Shakespeare
[due to Andrej Karpathy]:

```
"PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep."
```

Recurrent Neural Network (RNN)

Let's use a neural network to learn an appropriate history digest. This is more flexible than choosing a fixed history window.



Random variable notation:

$X_i \sim \text{Cat}(p_i)$ i.e. $\mathbb{P}(X_i = x) = [p_i]_x$

$(s_{i+1}, p_{i+1}) = f_\theta(s_i, X_i)$

Chain rule for probability:

$\mathbb{P}(A, B, C) = \mathbb{P}(A)\, \mathbb{P}(B|A)\, \mathbb{P}(C|A, B)$

For training, we need a formula for the likelihood $\Pr_{\underline{X}}(\underline{x}; \theta)$:

$$\Pr_{\underline{X}}(x_1, \dots, x_n) = \Pr_{X_1}(x_1)\, \Pr_{X_2}(x_2|x_1) \cdots \times \Pr_{X_n}(x_n|x_1 \cdots x_{n-1})\, \Pr_{X_{n+1}}(\square|x_1 \cdots x_n)$$

by the chain rule for probability

$$= [p_1]_{x_1}\, [p_2]_{x_2} \times \cdots \times [p_n]_{x_n} [p_{n+1}]_\square$$

where each p_i is a function of $x_1 \cdots x_{i-1}$

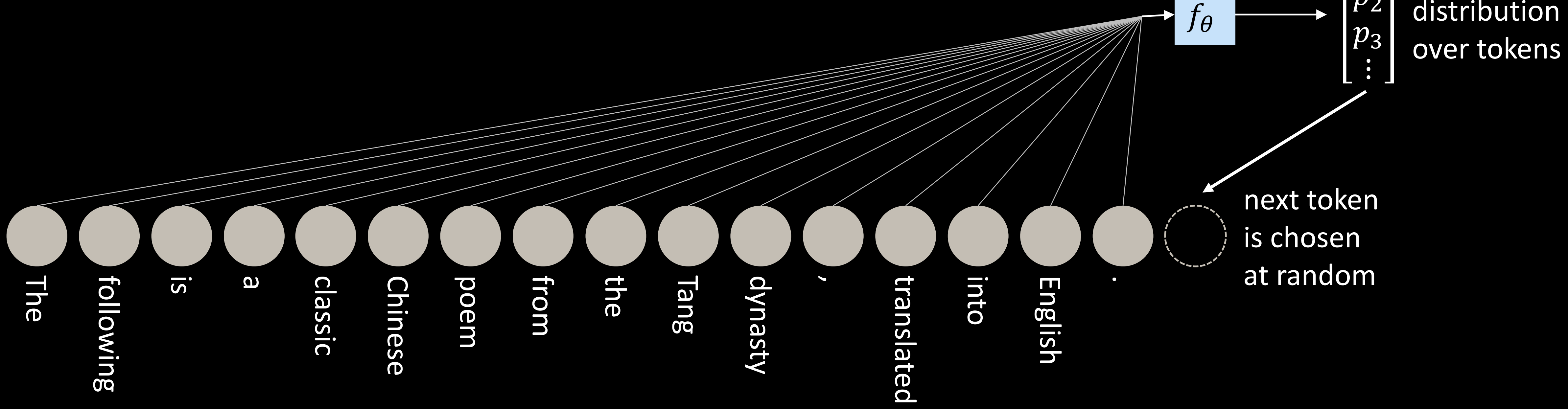
```
def loglik(xstr):
    res = 0
    s,x = 0,□
    for x_next in xstr + "□":
        s,p = f_θ(s,x)
        res += log(p[x_next])
        x = x_next
    return res
```

No one has managed to make RNNs produce coherent text longer than ≈20 tokens

Perhaps we'd do better with a model where the next token is allowed to depend on the entire sequence so far.

What would a suitable f_θ look like?

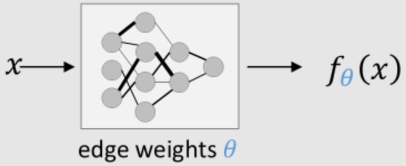
- It has to accept an input sequence of any length
- It has to be differentiable with respect to θ, so that we can run numerical optimization on $\Pr(\underline{x};\theta)$
- … so `while` loops and recursion are out!



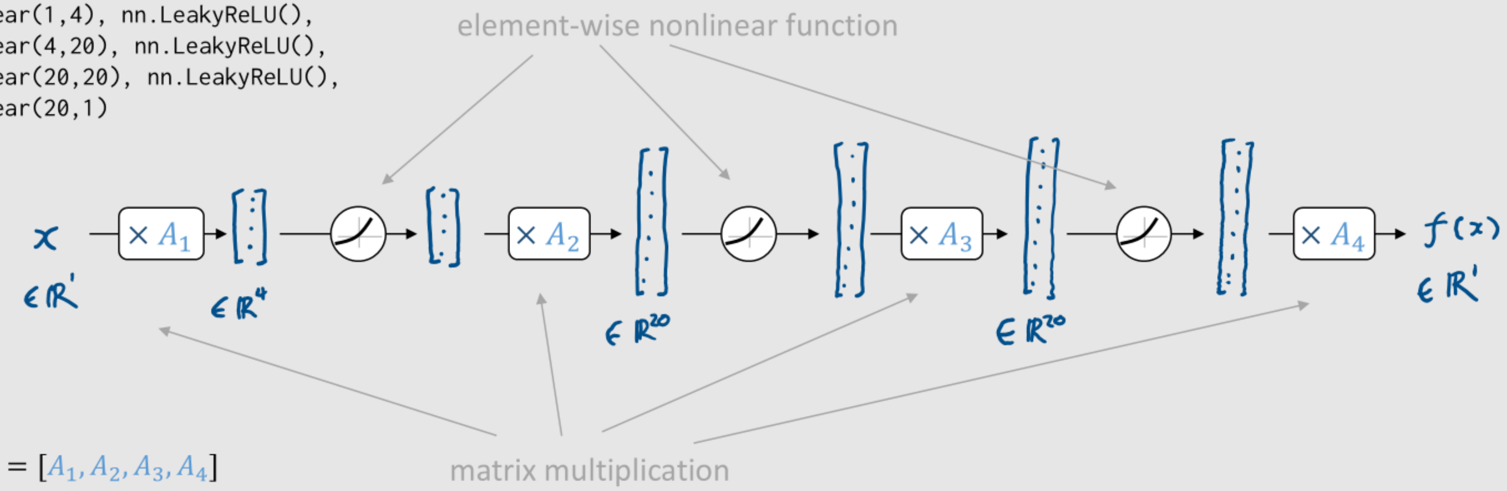
`The following is a classic Chinese poem from the Tang dynasty, translated into English.`

LECTURE 4

it's easy to work with functions made out of matrix multiplication and element-wise non-linear maps.

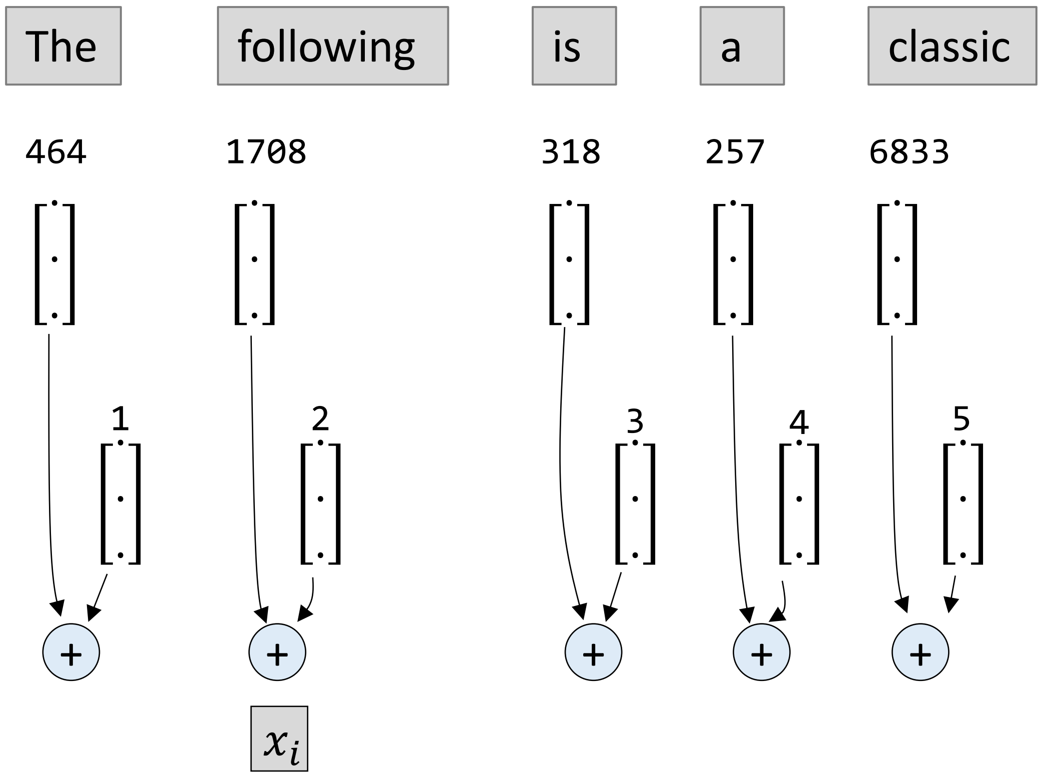



```
f = nn.Sequential(
      nn.Linear(1,4), nn.LeakyReLU(),
      nn.Linear(4,20), nn.LeakyReLU(),
      nn.Linear(20,20), nn.LeakyReLU(),
      nn.Linear(20,1)
      )
```



parameters $\theta = [A_1, A_2, A_3, A_4]$

The Transformer architecture is a cleverly designed f function.



W = vocab size

Split the text into tokens $t_i \in \{1, \dots, W\}$

d is the working dimensionality, e.g. d=200

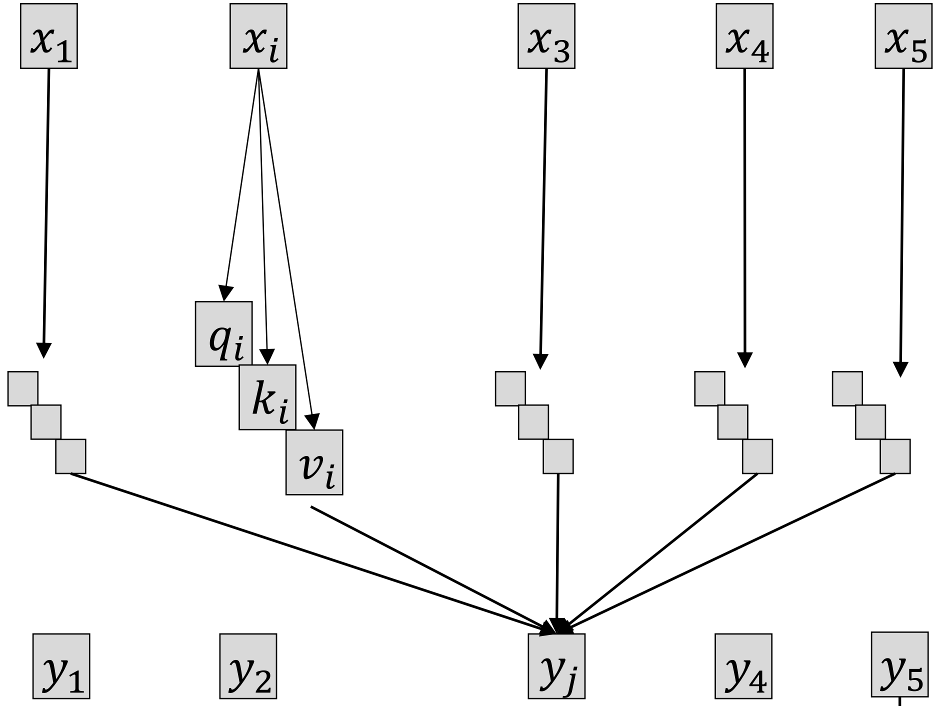
Turn each token into a vector $e_i \in \mathbb{R}^d$ by looking up an embedding matrix $E \in \mathbb{R}^{W \times d}$

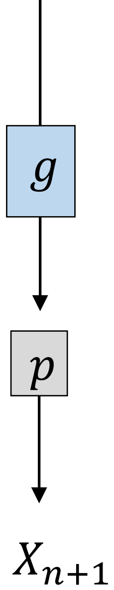
E is a matrix to be learnt in training

For each position $i \in \{1, \dots, n\}$ create a position-embedding vector $t_i \in \mathbb{R}^d$ $\begin{bmatrix} \sin(i) \\ \cos(i) \\ \sin(i/2) \\ \cos(i/2) \\ \vdots \end{bmatrix}$

This allows the attention mechanism to say e.g. "give me the item 3 time-steps back"

Let $x_i = e_i + t_i \in \mathbb{R}^d$





For each position $i \in \{1, \dots, n\}$,
let $q_i = Qx_i$, let $k_i = Kx_i$, let $v_i = Vx_i$
$\in \mathbb{R}^e$ $\in \mathbb{R}^e$ $\in \mathbb{R}^d$

Q,K,V are matrices to be learnt in training

q_i = query k_i = key v_i = value

The queries and keys say how much attention each position should pay to each other position. The values are some internal representation of "relevant content", like the state variable s in the RNN.

For each position $j \in \{1, \dots, n\}$ we'll produce an output vector $y_j \in \mathbb{R}^d$, as follows:

s ji = attention to pay to position I when producing output j

1. let $s_{ji} = q_j \cdot k_i$
2. let $a_{j*} = \mathrm{softmax}\left(s_{j*}/\sqrt{e}\right)$
3. let $y_j = \Sigma_i a_{ji} v_i$

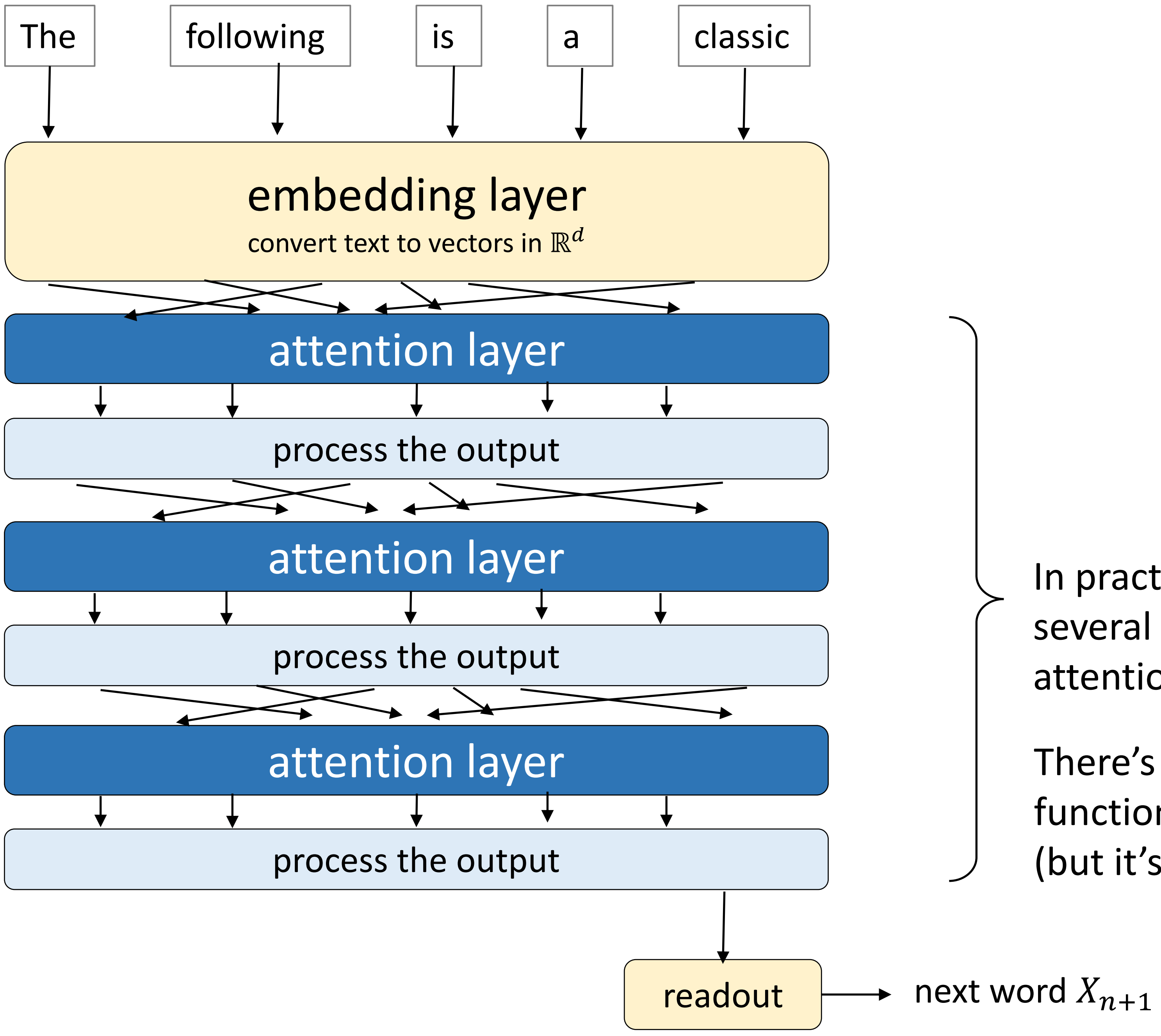
convert the attention scores into a vector aj that sums to I

ATTENTION MECHANISM

- Consider a dictionary $\{k_i{:}v_i$ for i in range(n)$\}$.
- To produce the output in position j, I'll look up the value for key q_j in this dictionary.
- But there are never exact matches, and anyway exact matching isn't a nice optimizable function. Instead use a "fuzzy lookup" based on how well q_j matches k_i.

Convert the final value y_n into a distribution over tokens $p \in \mathbb{R}^W$ using some neural network $p = g(y_n; \theta)$

Generate the next token by $X_{n+1} \sim \mathrm{Cat}(p)$



In practice, it's useful to use several passes of the attention mechanism.

There's still an explicit likelihood function, and it's easy to code it (but it's messy to write out an explicit formula).

The history of random sequence models