## The “Deep Learning for NLP” Lecture Roadmap

# Lecture 9: Large Language Models (1/2)

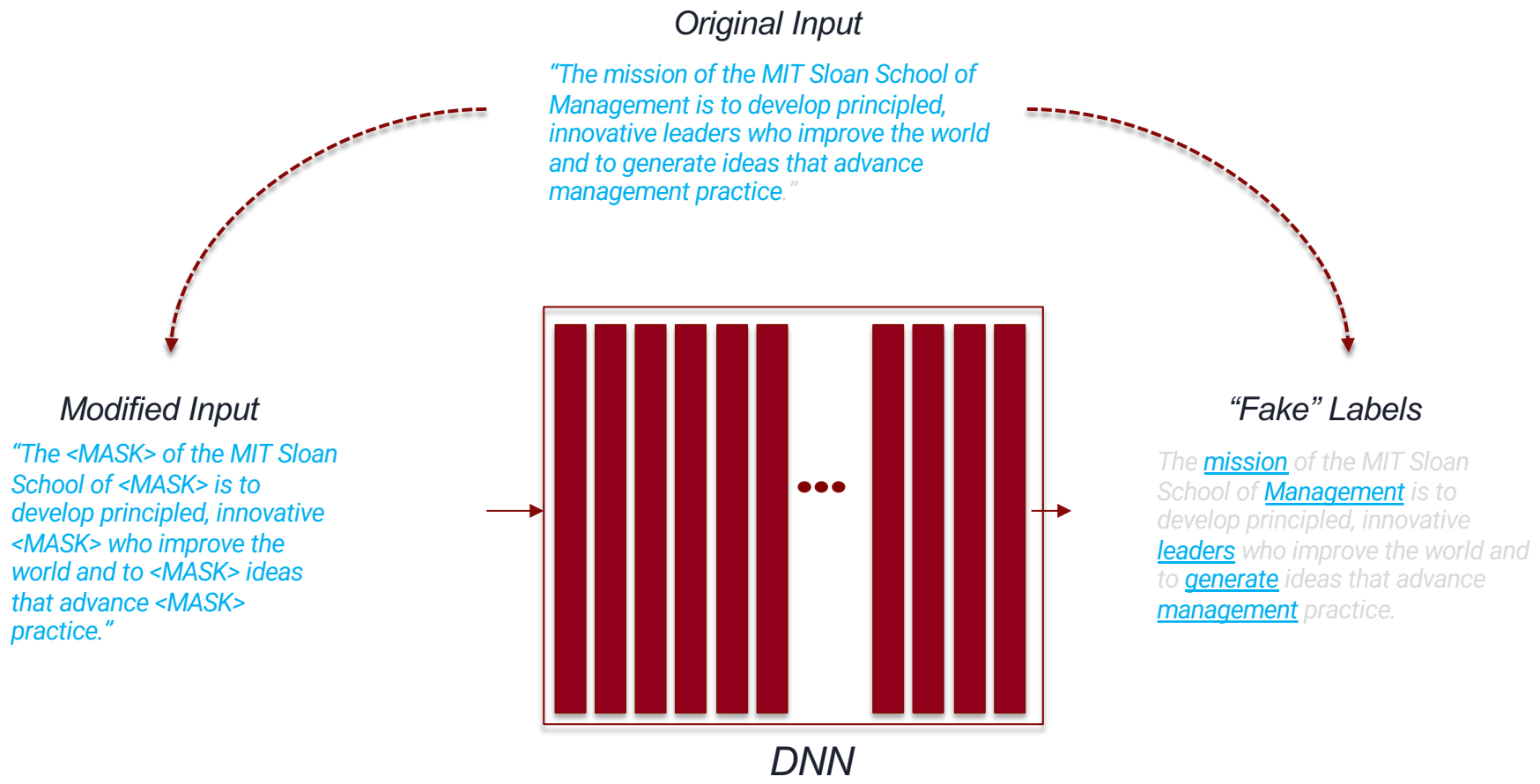
~~Lecture 5: Text Vectorization and the Bag-of-Words Model~~
~~Lecture 6: Embeddings~~
~~Lecture 7: Transformers – (1/2)~~
~~Lecture 8: Transformers – (2/2)~~

MIT MANAGEMENT SLOAN SCHOOL

15.S04: Hands-on Deep Learning
Spring 2024

Farias, Ramakrishnan

# [Review] Masking*: A Self-Supervised Learning technique for text data



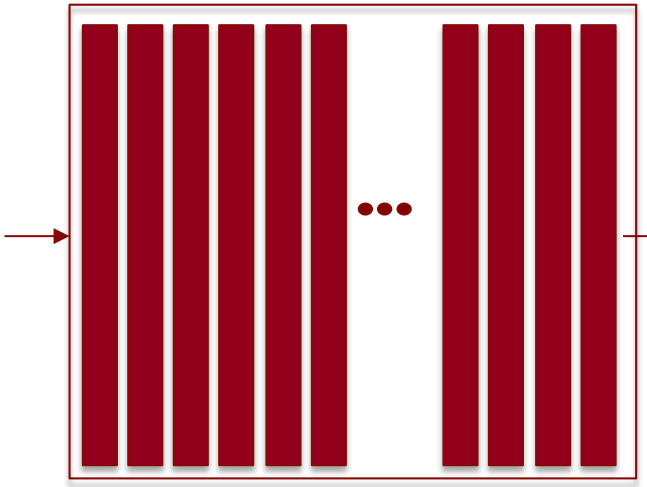

*Sometimes referred to as "Masked Language Modeling"

# Another Self-Supervised Learning for text inputs: *Next word prediction**

*Original Input*

*"The mission of the MIT Sloan School of Management is to develop principled, innovative leaders who improve the world and to generate ideas that advance management practice."*

| Modified Input | ”Fake” Label |
|---|---|
| The | mission |
| The mission | of |
| The mission of | the |
| The mission of the | MIT |
| The mission of the MIT | Sloan |
| The mission of the MIT Sloan | School |
| … | |
| The mission of the MIT Sloan … management | practice |



*DNN*

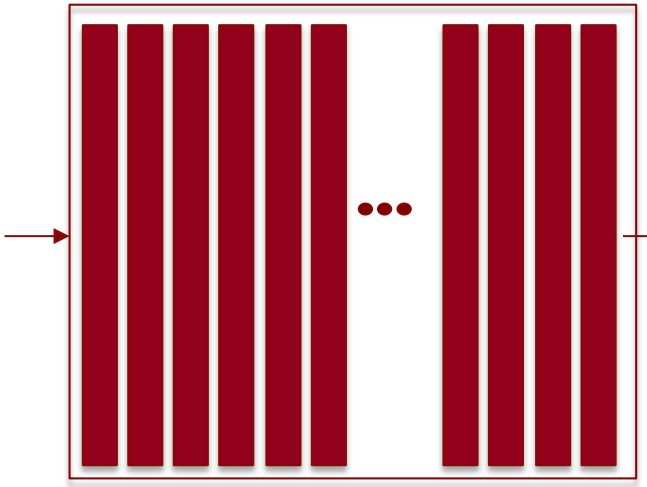
*Sometimes referred to as “Causal Language Modeling”

# Let's try to use the Transformer Encoder to implement Next Word Prediction

*Original Input*

*"The mission of the MIT Sloan School of Management is to develop principled, innovative leaders who improve the world and to generate ideas that advance management practice."*

| Modified Input |
| --- |
| The |
| The mission |
| The mission of |
| The mission of the |
| The mission of the MIT |
| The mission of the MIT Sloan |
| … |
| The mission of the MIT Sloan … management |



*DNN*

| "Fake" Label |
| --- |
| mission |
| of |
| the |
| MIT |
| Sloan |
| School |
| |
| practice |

# First, each sentence gets transformed into an input-output pair

| *Input* | *Output* |
| --- | --- |
| the | cat |
| cat | sat |
| sat | on |
| on | the |
| the | mat |

# The raw input gets tokenized* and converted into embeddings

| *Input* | *Positional input embeddings* | *Output* |
| --- | --- | --- |
| the |  | cat |
| cat |  | sat |
| sat |  | on |
| on |  | the |
| the |  | mat |

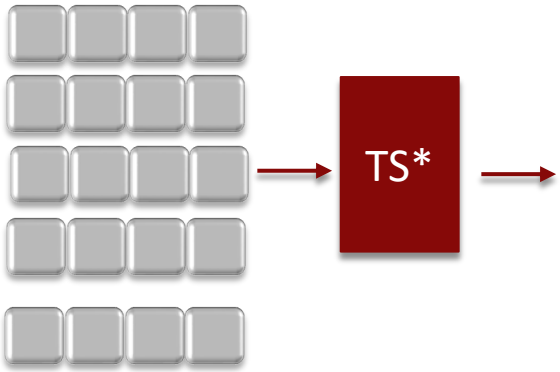
*not shown

# The embeddings are passed through several transformer blocks

*Input*

the
cat
sat
on
the

*Positional input embeddings*

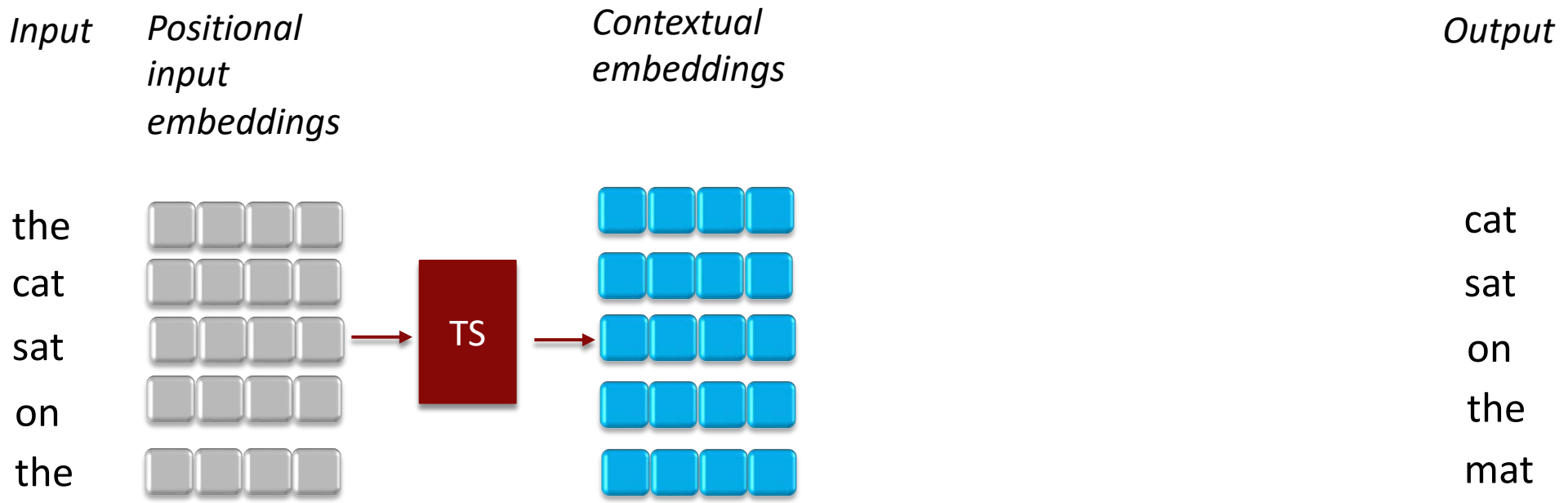



*Output*

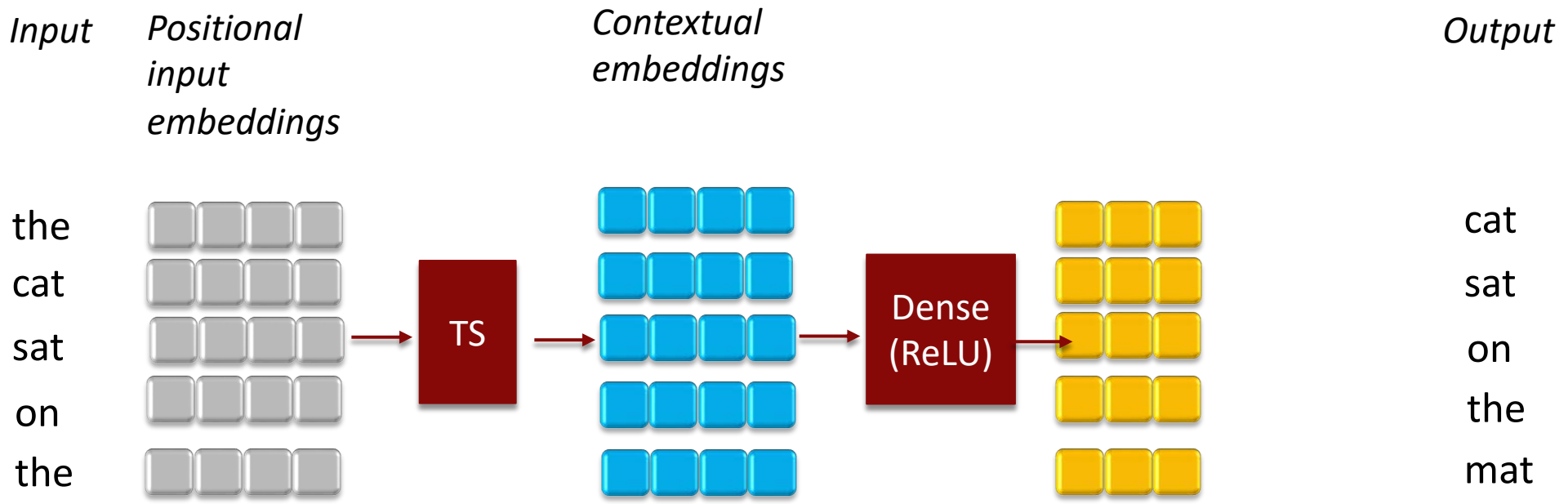
cat
sat
on
the
mat

*Transformer stack

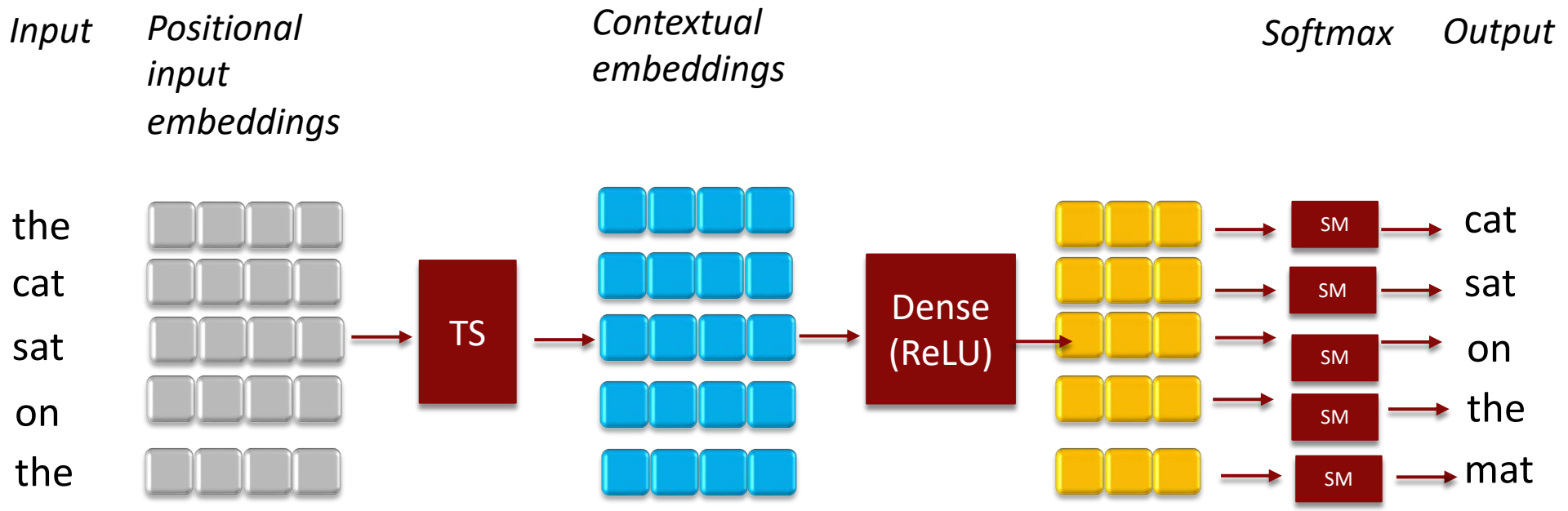
# The output of the transformer stack are contextual embeddings (as we have learned earlier)



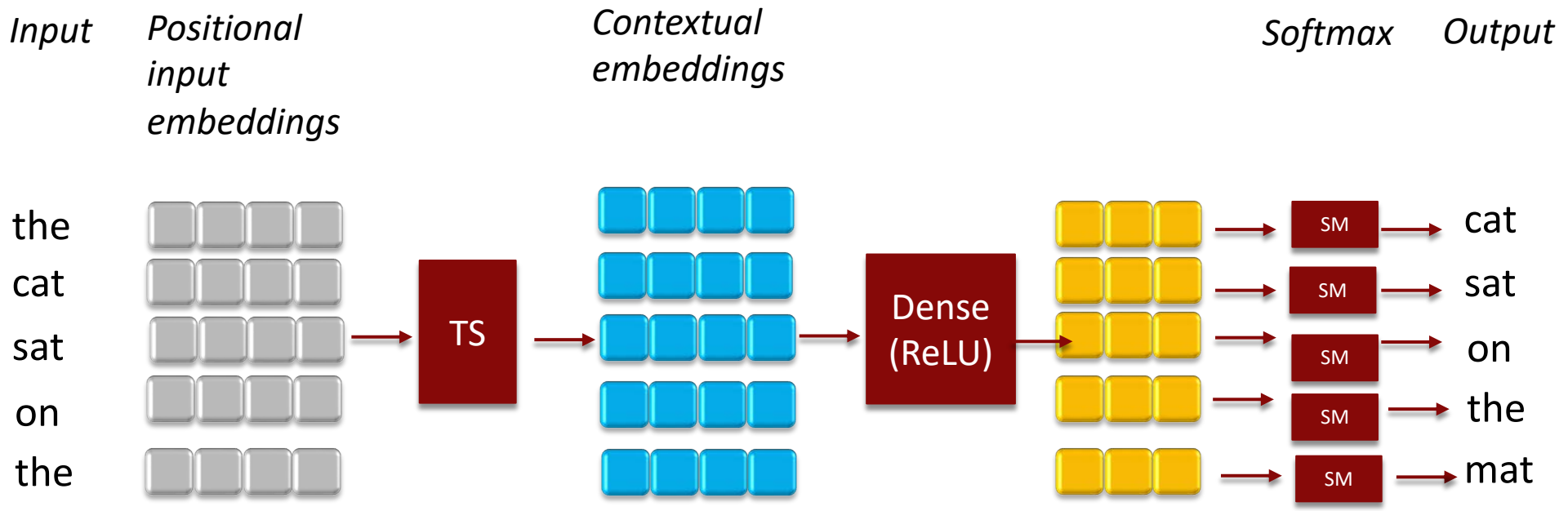
# The contextual embeddings are passed through one or more dense-RELU layers



# Each embedding coming out of the final dense layer is passed through its own softmax layer (with # of softmax categories = vocab size)

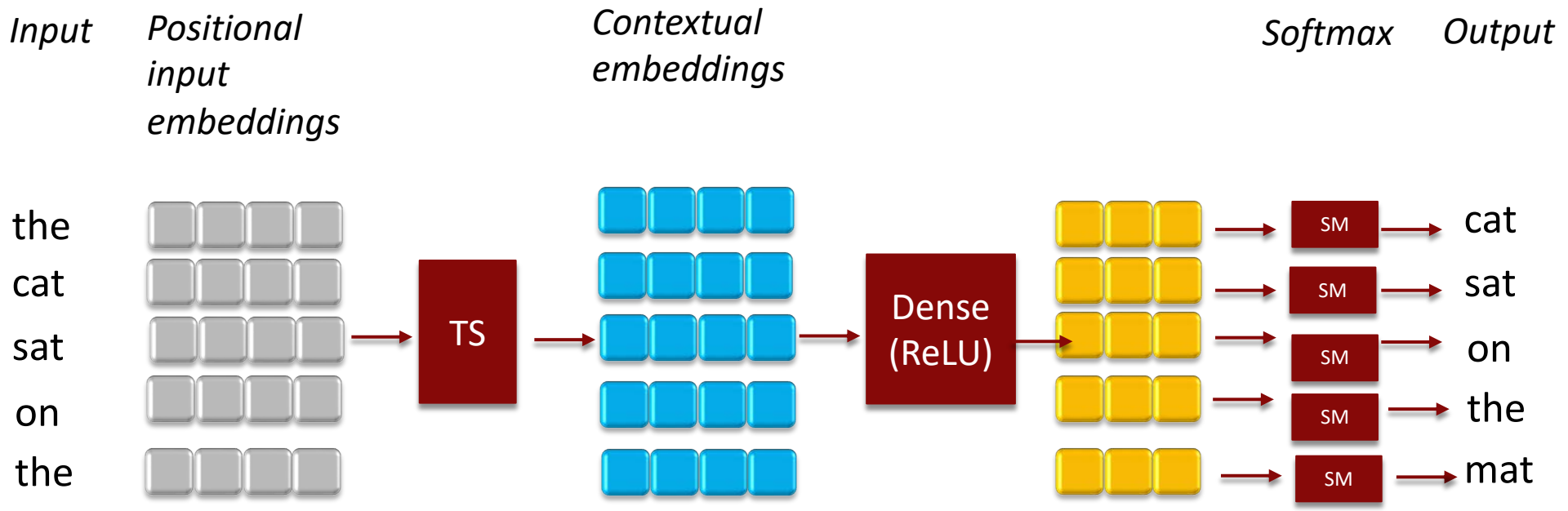


# We use the categorical cross entropy loss function for training

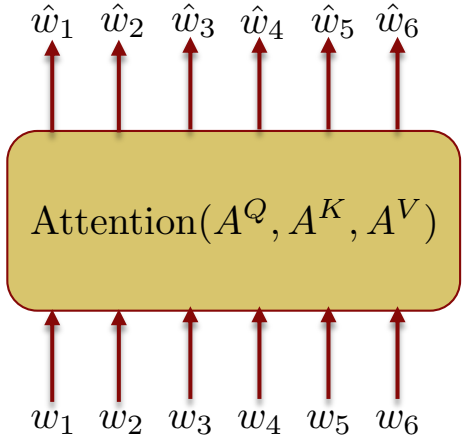



-(1/7) [log(prob(“the”)) + log(prob(“cat”)) + … + log(prob(“<E>”))]

# Notice any problems with this setup?

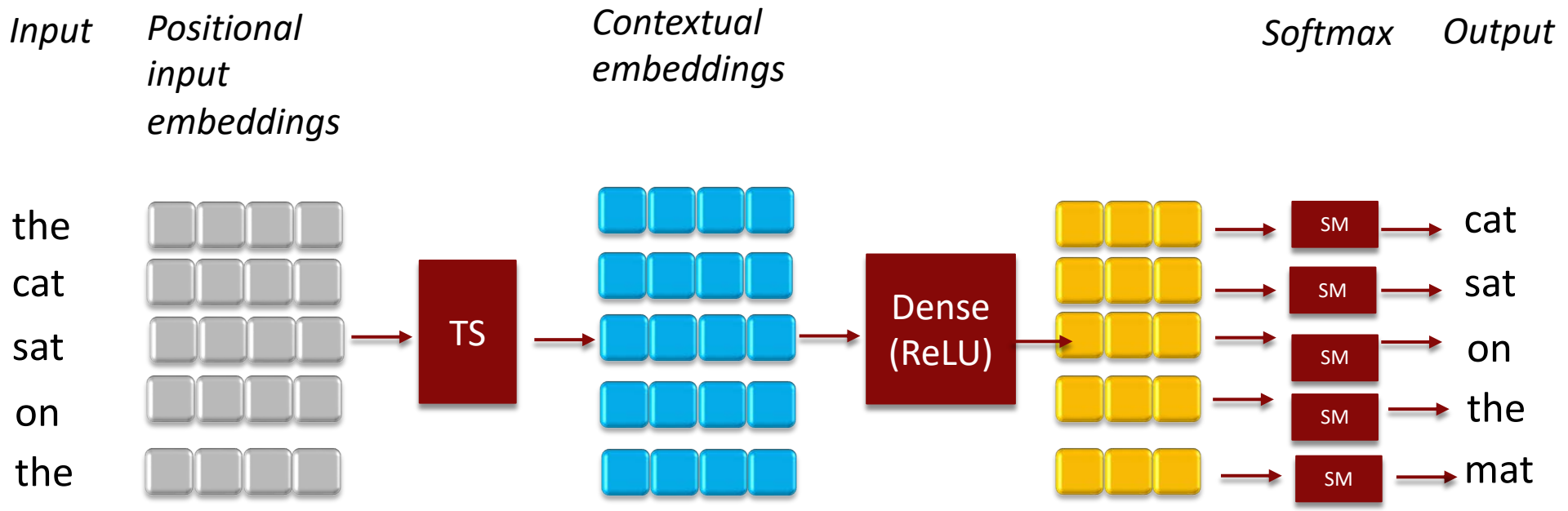


# Clue: Recall how self-attention works

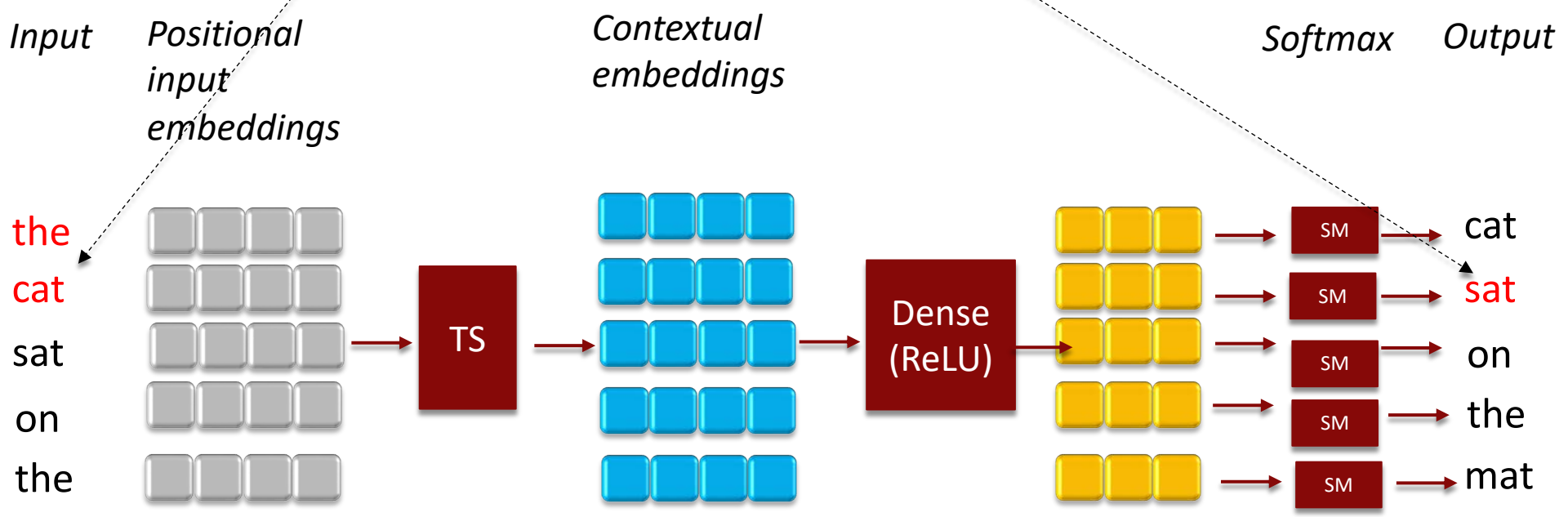



The contextual embedding for a word is based on ALL the words in the sentence

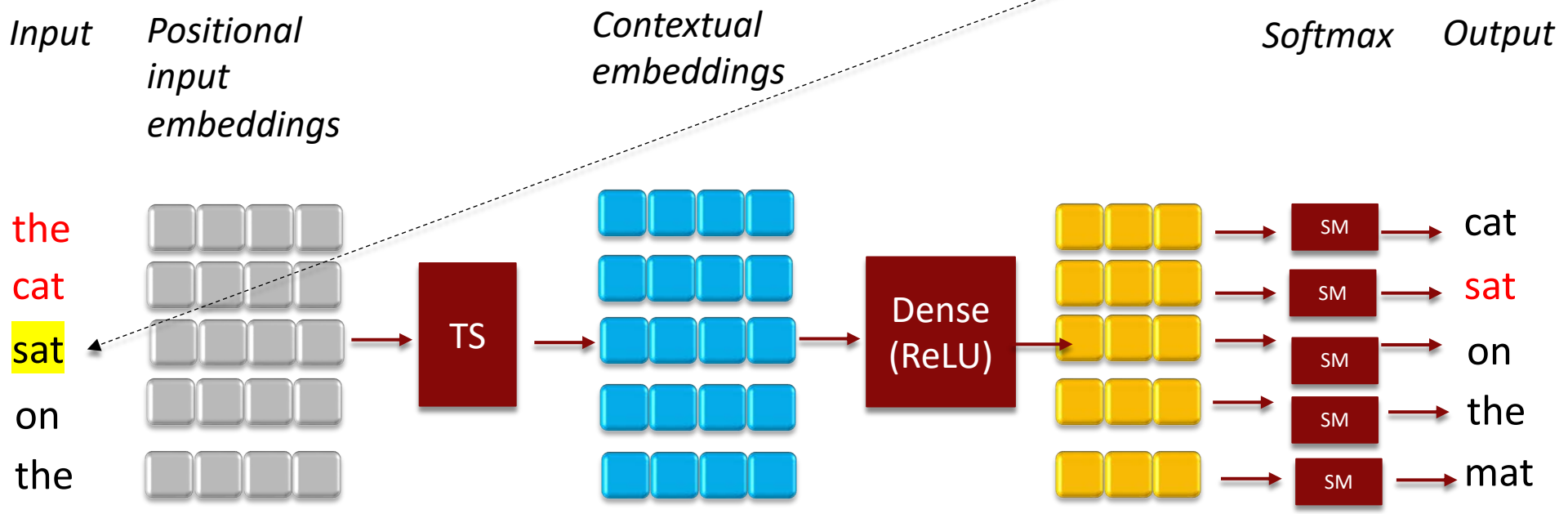
# The problem: To predict the next word, the architecture below can simply copy it from the input since it can "see" the whole sentence!



# The problem: To predict sat, the network should only use "the cat"

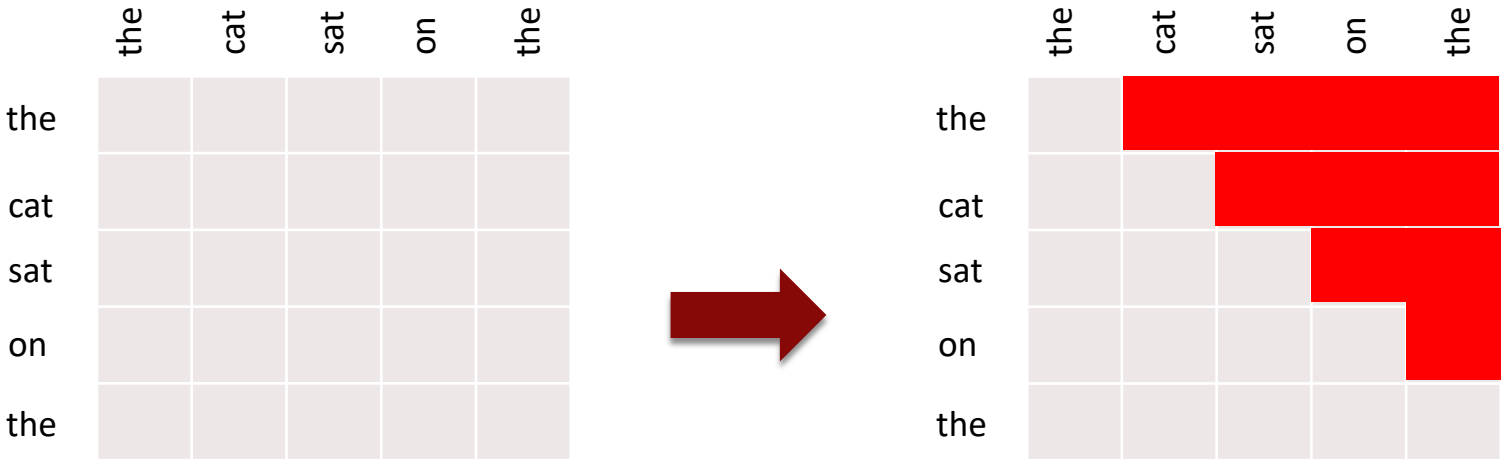


# The problem: To predict sat, the network should only use "the cat", but because Self-Attention can see sat, it will trivially predict the next word to be sat



The solution: When calculating the contextual embedding of a word, give zero weight to “future” words

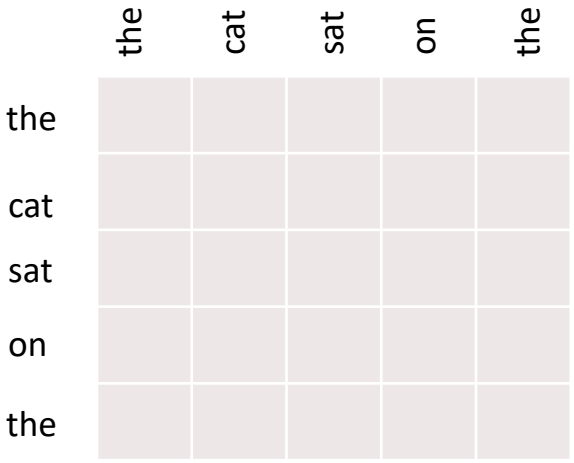
# We can easily modify the Self-Attention layer to do this



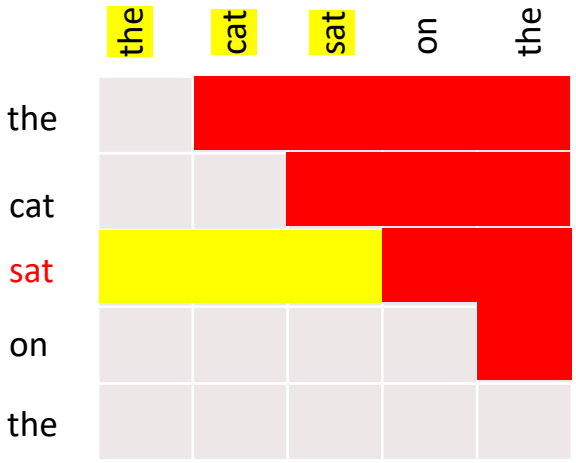

When calculating attention weights …

… we zero out the weights for the "future" words (and renormalize so the weights in each row still add up to 1.0)

# For example: When calculating the contextual embedding for sat, only the embeddings for "the", "cat" and "sat" will be weighted-averaged

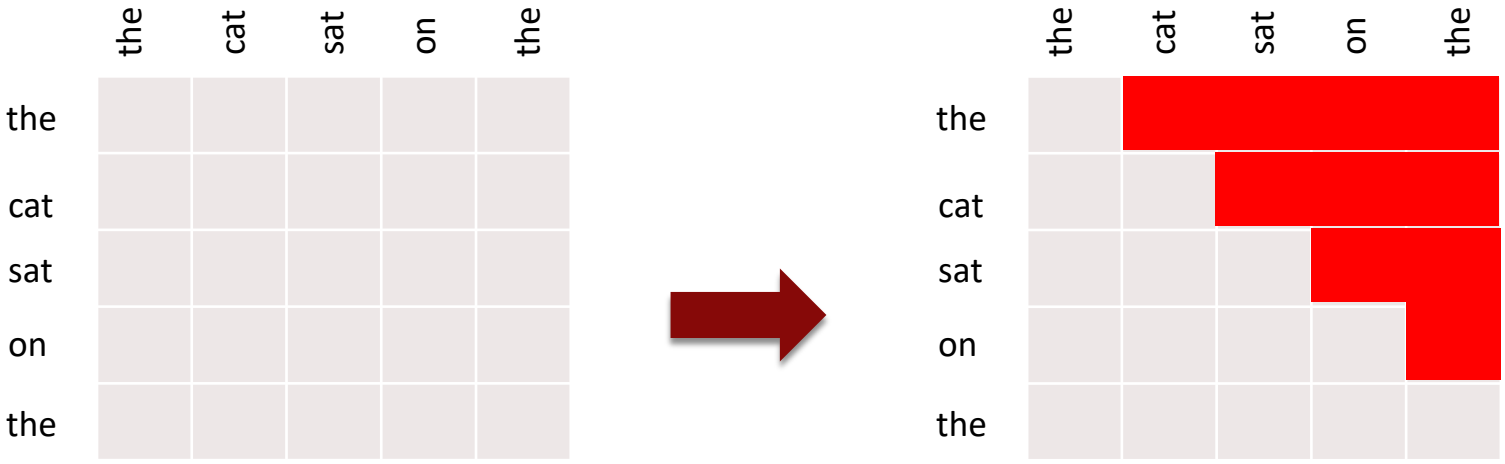



When calculating attention weights …




… we zero out the weights for the "future" words (and renormalize so the weights in each row still add up to 1.0)

# This modification to Self-Attention is called Causal Self-Attention*

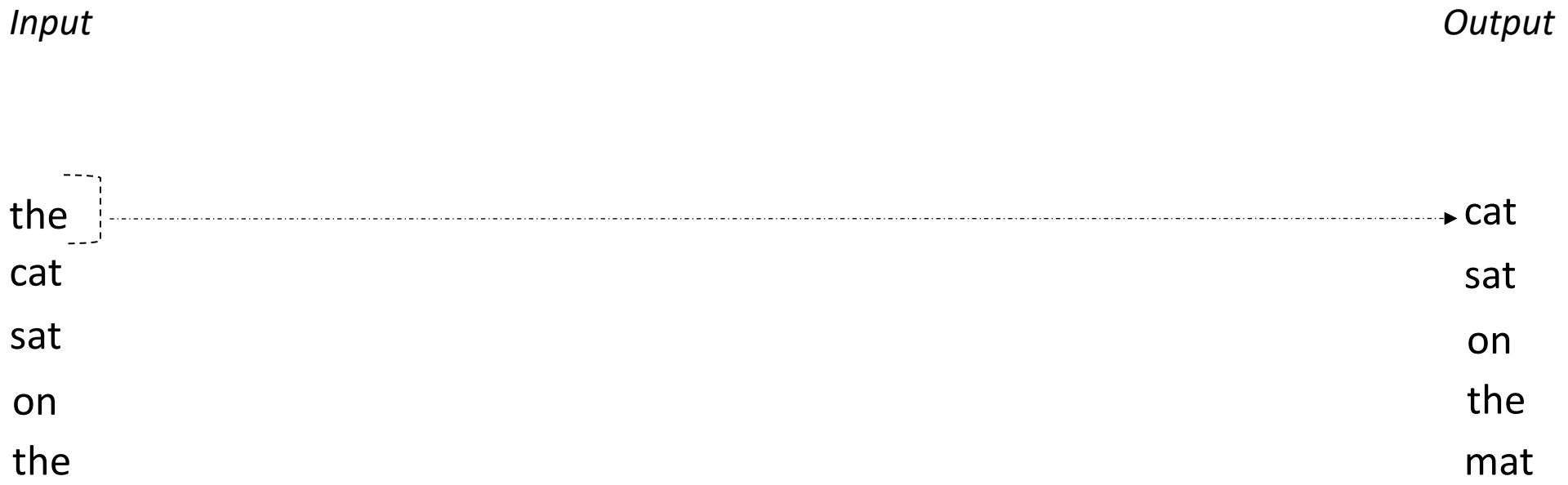



When calculating attention weights …

… we zero out the weights for the "future" words (and renormalize so the weights in each row still add up to 1.0)

*Also called "Masked Self-Attention" since future words are being 'masked' from the calculation

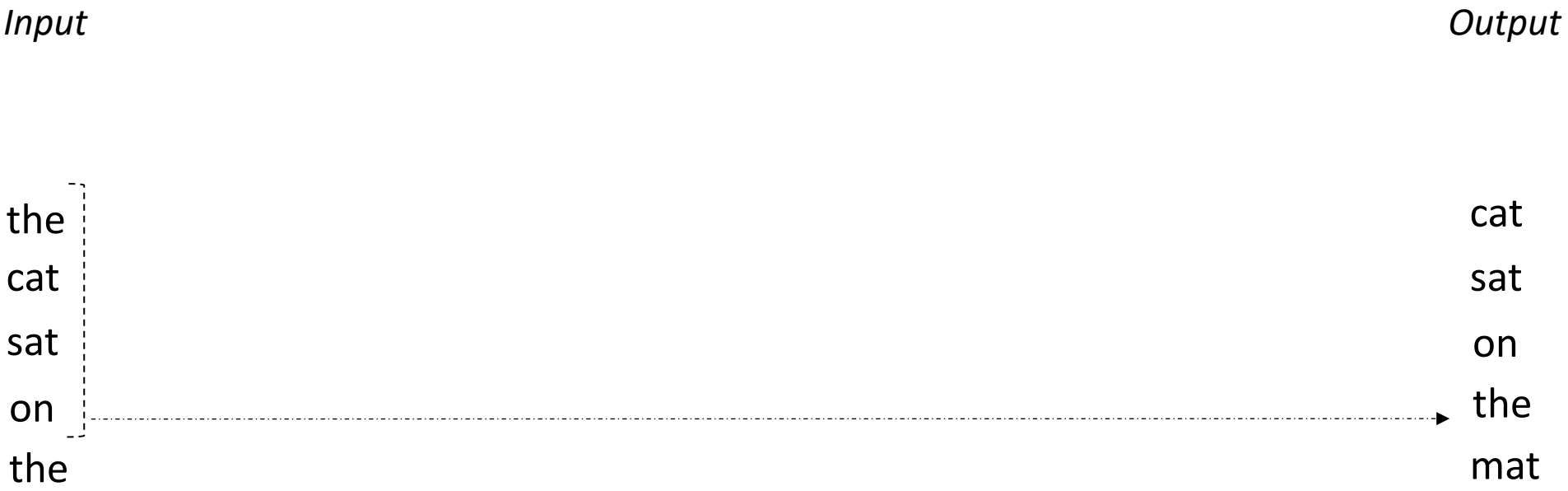
# This ensures that the prediction for a word will only depend on “past” words



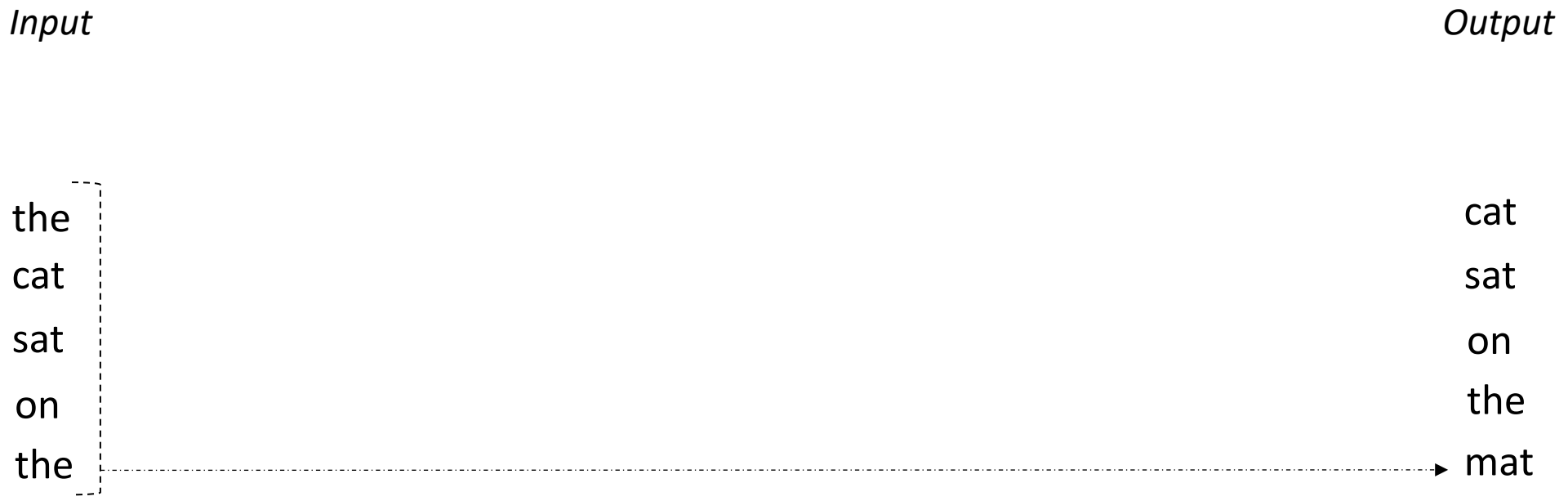
# This ensures that the prediction for a word will only depend on “past” words

# This ensures that the prediction for a word will only depend on “past” words

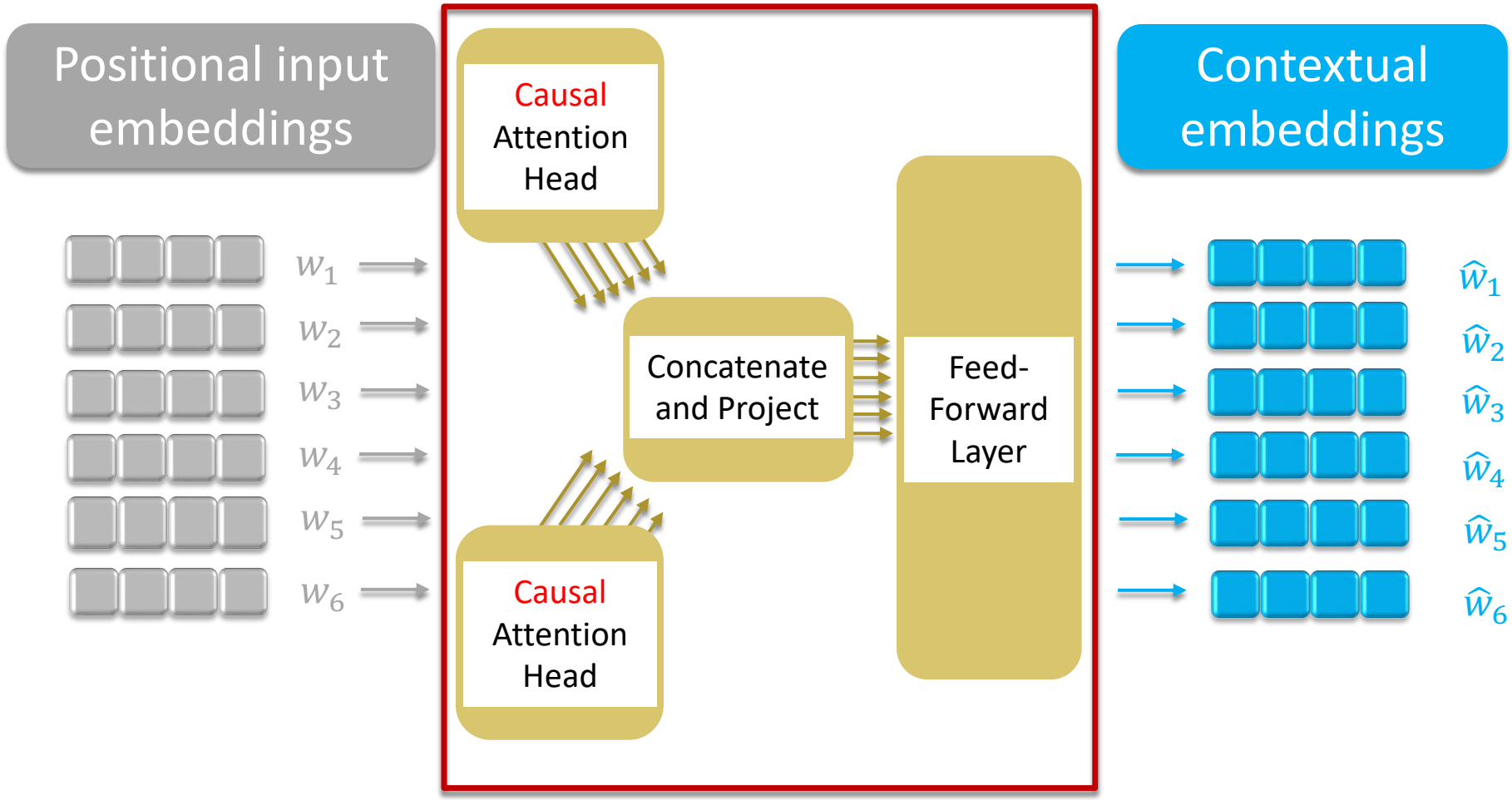
# This ensures that the prediction for a word will only depend on “past” words



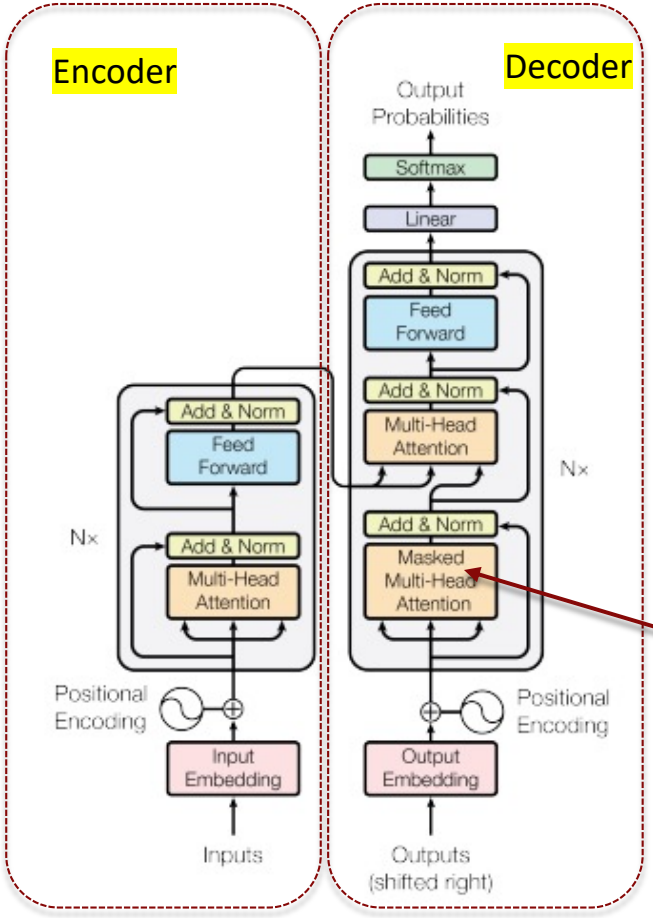
# This ensures that the prediction for a word will only depend on “past” words



# By replacing Self-Attention with Causal Self-Attention in the Transformer block, we get a Transformer Causal Encoder



# The original Transformer paper has what's called an encoder-decoder architecture




This architecture was designed to solve sequence-to-sequence problems like Machine Translation.

We won't cover sequence-to-sequence due to time constraints.

But note that the Decoder uses "Masked (aka Causal) Multi-Head Attention". As a result …

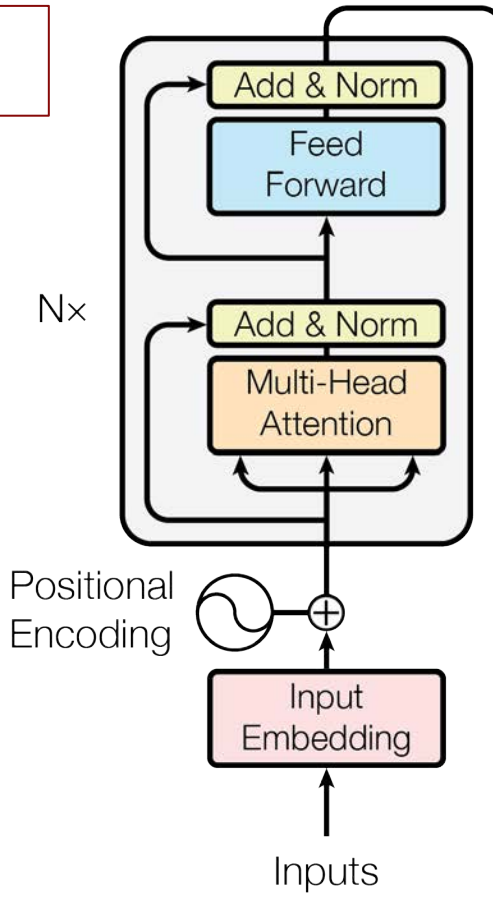
https://arxiv.org/abs/1706.03762

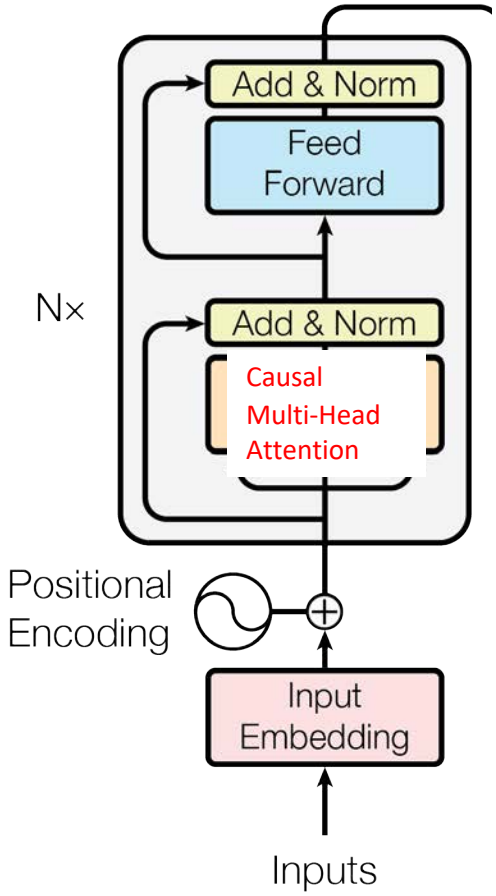
The Transformer Causal Encoder is also referred to as the Transformer Decoder.

It is usually clear from context which version of the Decoder is being referred to.

# Summary

Transformer Encoder

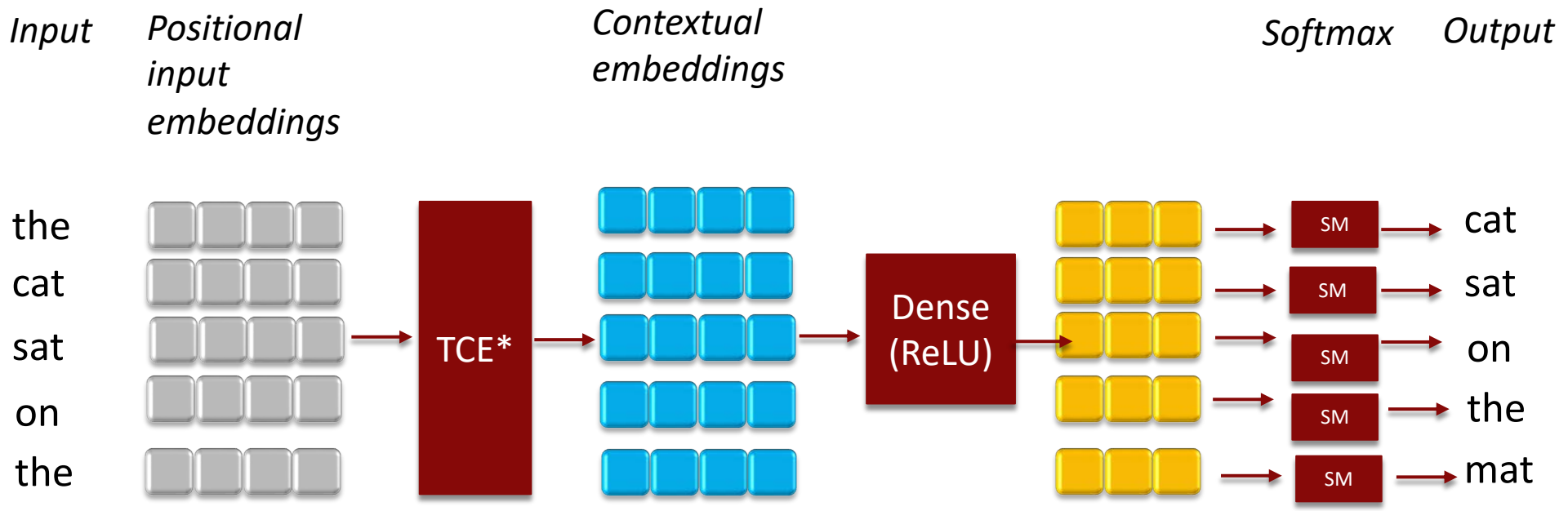



Transformer Causal Encoder/ Transformer Decoder

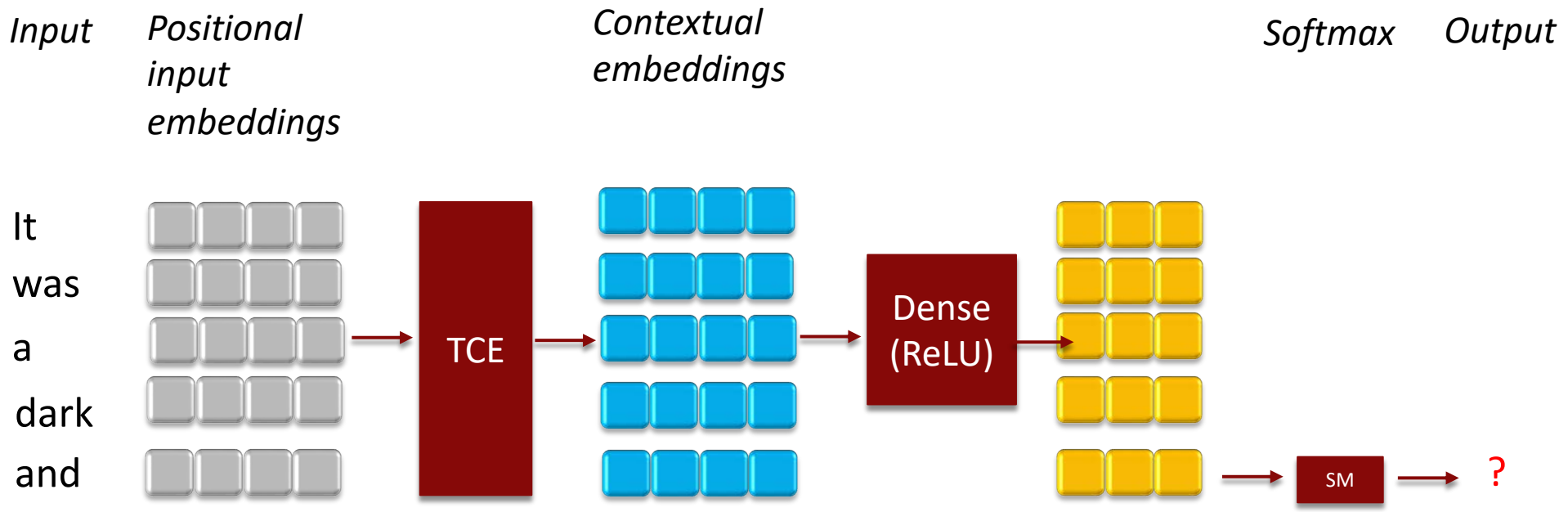
https://arxiv.org/abs/1706.03762

# Using Transformer Causal Encoders, we can train models for next word prediction




*Transformer Causal Encoder stack

# After training such a model, given any input sequence, the softmax for the last token …



# … is a probability distribution over the vocabulary

| Next Word* | Probability |
|---|---|
| aardvark | 0.0003 |
| … | |
| rainy | 0.3 |
| … | |
| stormy | 0.6 |
| … | |
| zebra | 0.00009 |

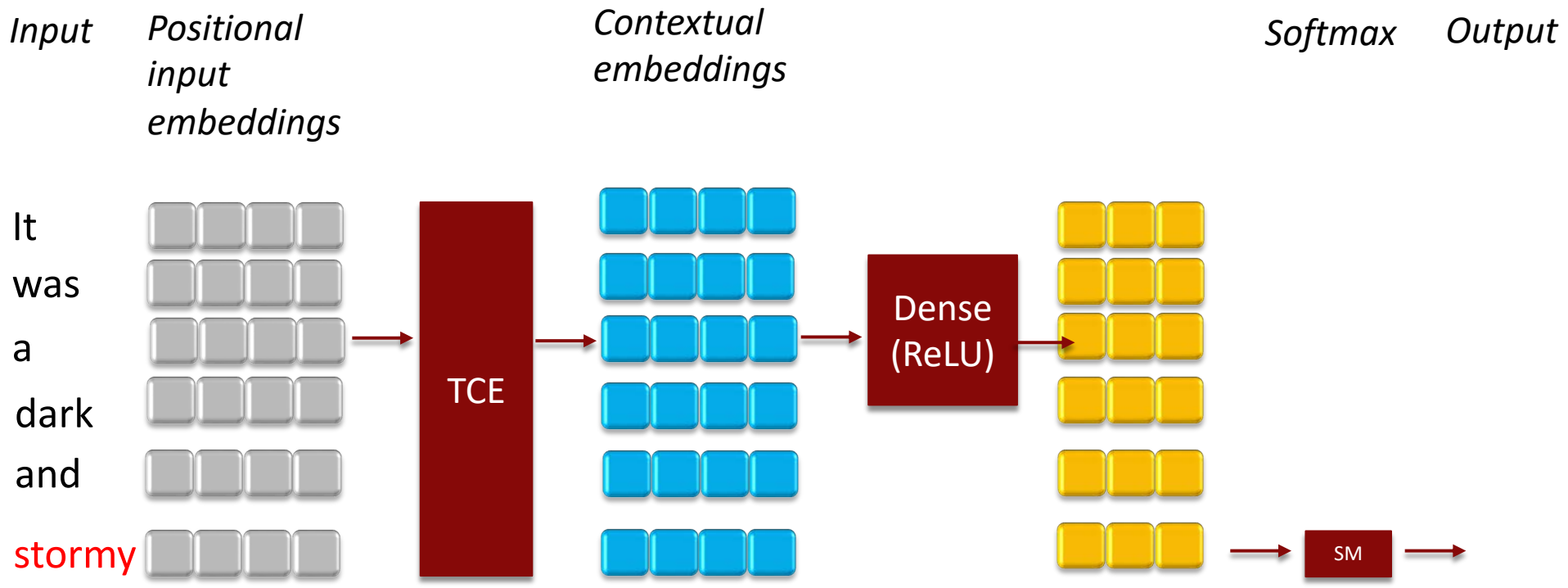
*Using ‘word’ instead of ‘token’ for simplicity

# We choose* a token from this table …

| Next Word* | Probability |
|---|---|
| aardvark | 0.0003 |
| … | |
| rainy | 0.3 |
| … | |
| stormy | 0.6 |
| … | |
| zebra | 0.00009 |

*Details on how to do this coming shortly

# … append it to the input and run it through the model …

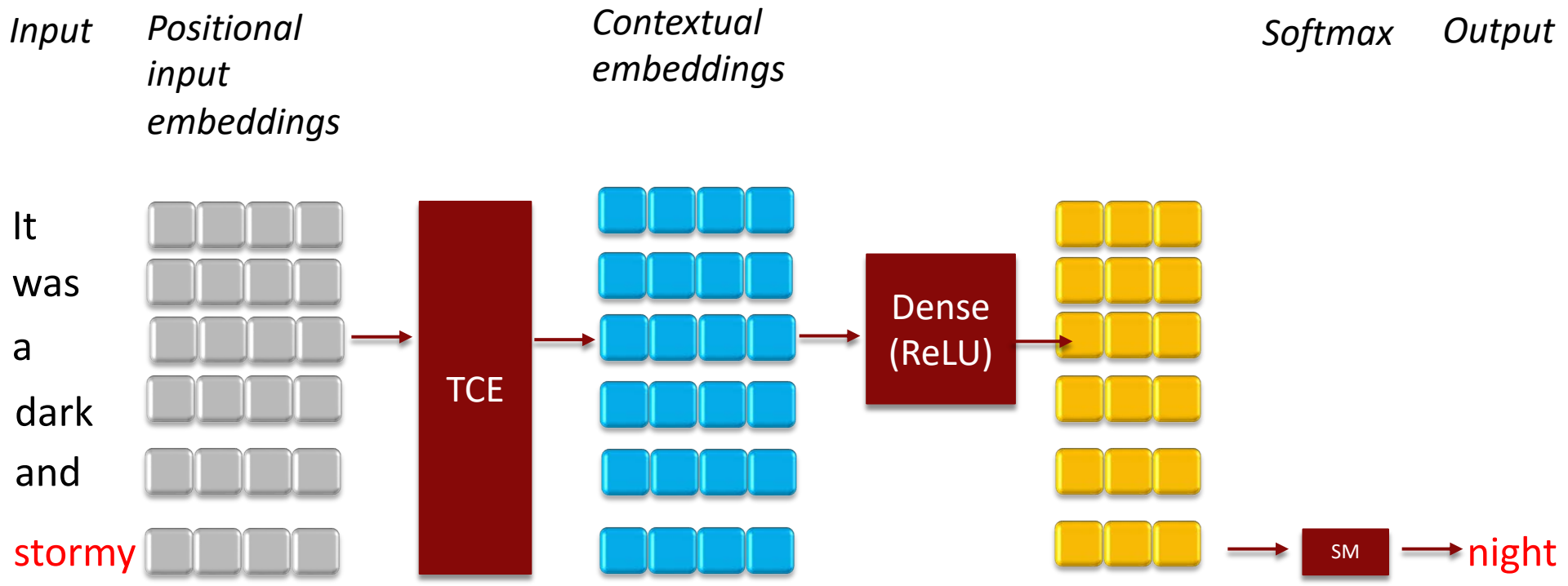


# … generate a new softmax and choose the next word …

| Next Word* | Probability |
|---|---|
| aardvark | 0.000008 |
| … | |
| day | 0.03 |
| … | |
| night | 0.95 |
| … | |
| zebra | 0.00000003 |

*Using ‘word’ instead of ‘token’ for simplicity

# … and repeat as needed.



# In other words, we know how to do Sequence Generation!

## Sequence classification

"I loved the movie" → Transformer Encoder → Positive

## Sequence labeling

"fly from Boston to Denver" → Transformer Encoder →

| Token | Label |
|---|---|
| fly | O |
| from | O |
| Boston | B-fromloc.city_name |
| to | O |
| Denver | B-toloc.city_name |

## ☑ Sequence generation

"I loved the movie" → Transformer Causal Encoder →

"I loved the movie, especially the cinematography and the background score"

Such sequence generation models trained on text sequences are called *Autoregressive Language Models* or *Causal Language Models*

# GPT-3 is an autoregressive LLM!

- Architecture: 96 Transformer layers and 96 heads in each multi-head attention layer!
- Training data for next word prediction: ~30 billion sentences from the Internet and books

| Dataset | Quantity (tokens) |
|---|---|
| Common Crawl (filtered) | 410 billion |
| WebText2 | 19 billion |
| Books1 | 12 billion |
| Books2 | 55 billion |
| Wikipedia | 3 billion |

http://arxiv.org/abs/2005.14165

# Sequence generation via LLMs like GPT-3 covers a wide array of use-cases

- *Text generation*: convert a text prompt into a paragraph that completes the prompt.
- *Code generation*: convert a text prompt into code
- *Code documentation*: augment code with comments
- *Text summarization*: convert a long document to a shorter version that retains the most important information.
- *Question-answering*: convert an input question into its answer.
- *Chatbots*: convert a dialogue prompt into a reply to the prompt or convert the history of a conversation into the next reply in the conversation.

# Sidebar: If you would like to implement a causal LLM from scratch ☺ …

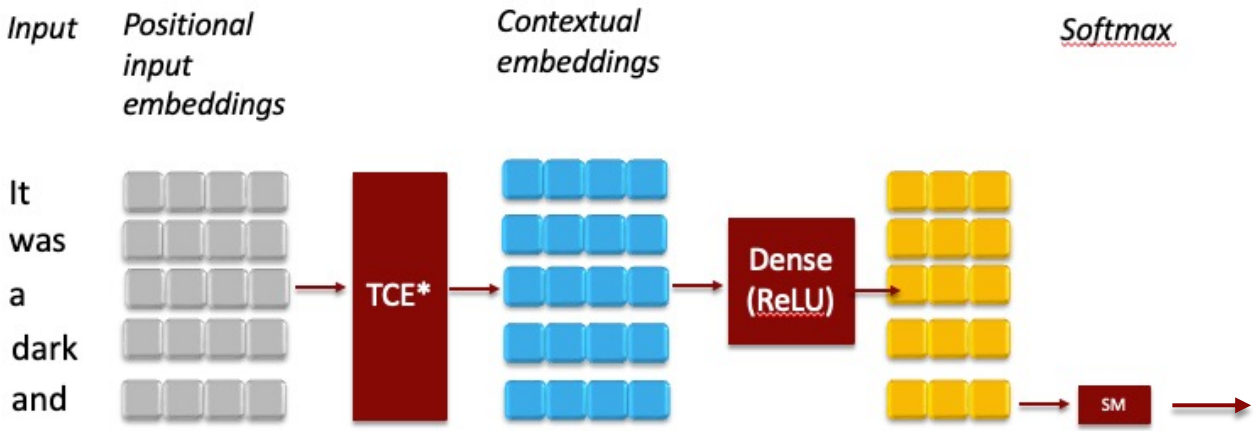

https://jaykmody.com/blog/gpt-from-scratch/




https://www.youtube.com/watch?v=kCc8FmEb1nY

# Decoding/Sampling Strategies

# The process of choosing a token from the probability distribution generated by the softmax is called decoding




| Next Word | Probability |
|---|---|
| aardvark | 0.0003 |
| … | |
| rainy | 0.3 |
| … | |
| stormy | 0.6 |
| … | |
| zebra | 0.00009 |

# Every time a token needs to be chosen, there are two simple options

- Choose the highest probability token. This is called Greedy Decoding*.
- Randomly sample a token (in proportion to its probability)

| Next Word | Probability |
|---|---|
| aardvark | 0.0003 |
| … | |
| rainy | 0.3 |
| … | |
| stormy | 0.6 |
| … | |
| zebra | 0.00009 |

* Not necessarily optimal. Beam Search can help. See https://www.borealisai.com/research-blogs/tutorial-6-neural-natural-language-generation-decoding-algorithms/ for more

# The decoding method should be matched to the type of response that's needed

- Greedy Decoding is appropriate when
  - Factual accuracy of response is important
  - Deterministic outputs are needed

- Random sampling is better suited when
  - Diversity and "creativity" of generated responses is important
  - Stochastic outputs are OK

# Random sampling has shortcomings

- Our expectation is that the probability distribution from the softmax has a “short head” of high-probability “good” tokens and a “long tail” of low-probability “bad” tokens
- We would like random sampling to sample from the ‘head’ and not the ‘tail’
- While the probability of choosing any individual token in the tail is small, the probability of sampling some token from the tail can still be high
- If the LLM happens to sample a token from the tail, it may not be able to “recover” from its mistake and may go off the rails

# Example: If the most probable token - “invited” - is chosen by random sampling

…

Prompt

Students at the MIT Sloan School of Management are

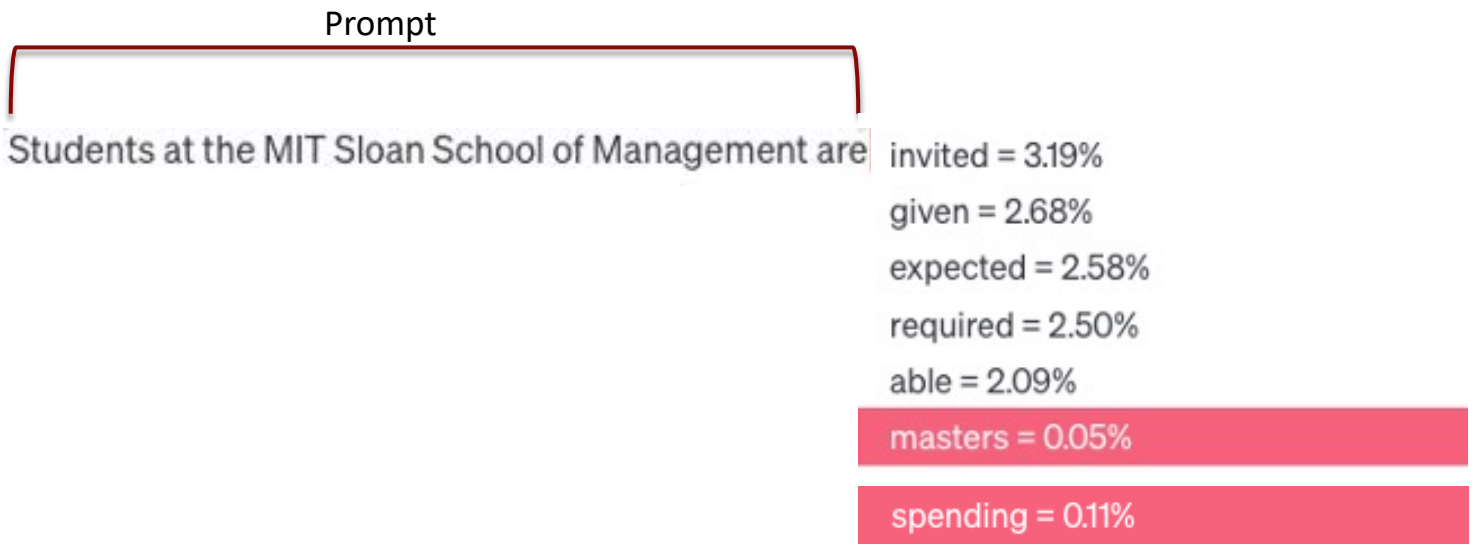
| |
|---|
| invited = 3.18% |
| given = 2.69% |
| expected = 2.59% |
| required = 2.50% |
| able = 2.09% |

# … the response seems reasonable.

Prompt

Students at the MIT Sloan School of Management are invited to submit their original white papers to the annual MIT

# However, if a low-probability token happens to be chosen …



Prompt

Students at the MIT Sloan School of Management are masters of chaos. They routinely blow past deadlines, fracture

Prompt

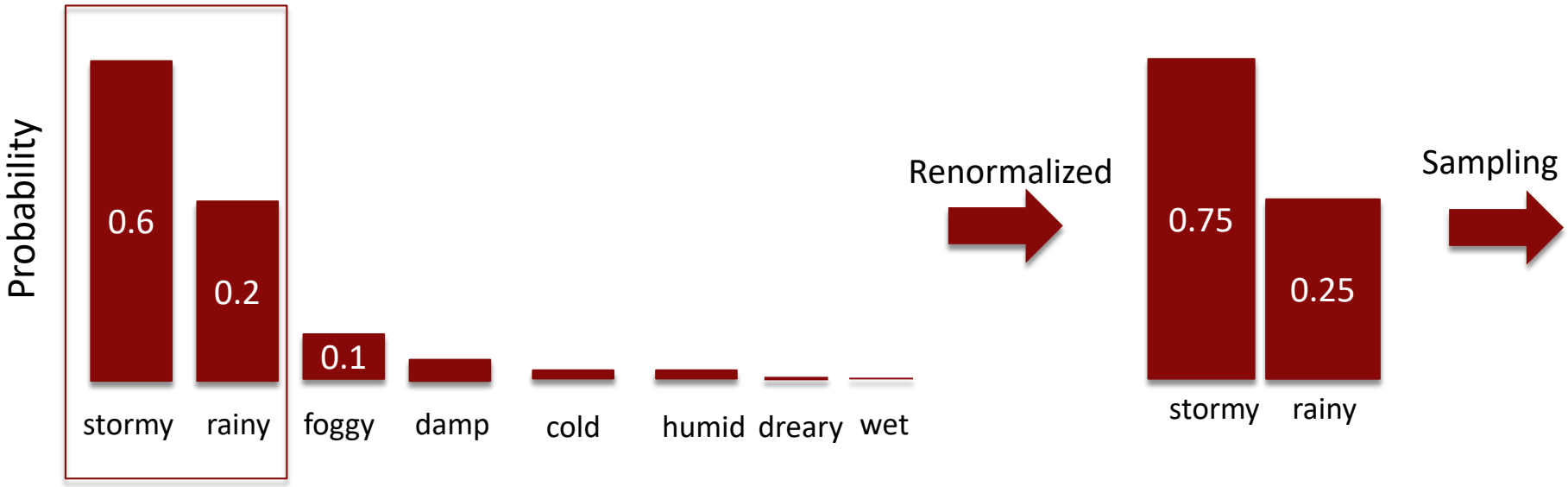
Students at the MIT Sloan School of Management are spending the semester learning life skills — through knitting socks.

# We can “tune” random sampling by first modifying the distribution and then sampling

- *Top-K Sampling*: Consider only the *K* most probable words. Renormalize their probabilities so they add up to 1. Do random sampling from this set.

# Top-K Sampling with K=2

Prompt: “It was a dark and ____”



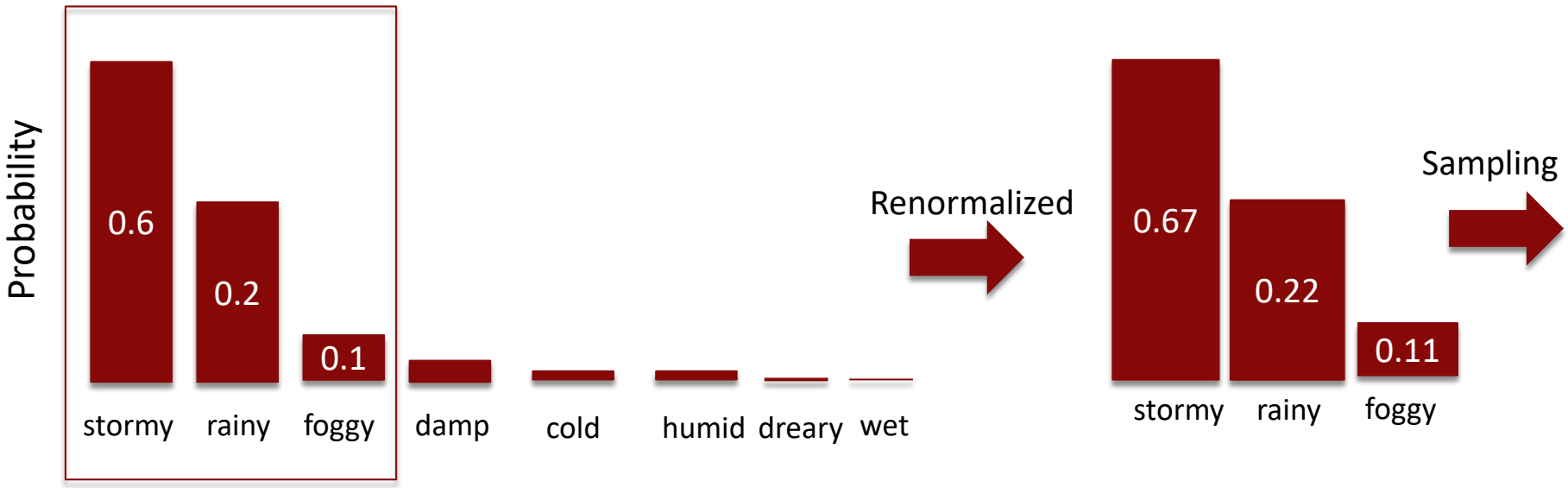
# We can "tune" random sampling by first modifying the distribution and then sampling

- *Top-K Sampling*: Consider only the *K* most probable words. Renormalize their probabilities so they add up to 1. Do random sampling from this set.

- *Top-P (aka Nucleus) Sampling*: Consider the minimum set of words whose total probability exceeds *P.* Renormalize their probabilities so they add up to 1. Do random sampling from this set.

# Top-P Sampling with P=0.9

Prompt: “It was a dark and ____”



# We can “tune” random sampling by first modifying the distribution and then sampling

- *Top-K Sampling*: Consider only the *K* most probable words. Renormalize their probabilities so they add up to 1. Do random sampling from this set.

- *Top-P (aka Nucleus) Sampling*: Consider the minimum set of words whose total probability exceeds *P.* Renormalize their probabilities so they add up to 1. Do random sampling from this set.

- *Temperature*: By adjusting this parameter, we can make the ”head” of the distribution less or more important

# Temperature

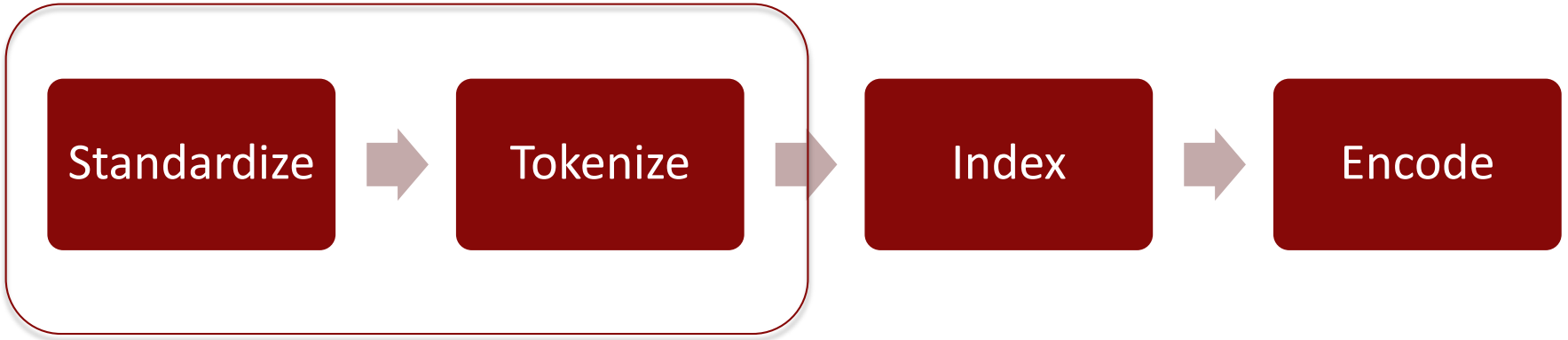
iPad

# Quick Demo – OpenAI Playground

https://platform.openai.com/playground?mode=complete

# LLM Tokenization: Byte Pair Encoding

# Any disadvantages to what we described earlier?



# Better Tokenization Schemes

- Modern generative models use tokenization schemes that try to address these disadvantages
- The GPT family uses Byte Pair Encoding (BPE). BERT uses WordPiece
- BPE finds a good "middle ground" between characters and complete words. Ending token vocabulary will be
  - Characters
  - Full words that occur frequently enough to be worth adding
  - Sub-words (i.e., word fragments) that occur frequently enough to be worth adding

# How BPE works

Key Intuition: Start with each character as a token, and "merge" tokens that most frequently occur next to each other. Don't merge across whitespace. Stop when the size of your vocabulary reaches a user-defined limit (we will assume 12 in the example below*)

| | |
|---|---|
| *Training corpus* | The cat sat on the mat |
| *After Standardization* | the cat sat on the mat |
| *Starting corpus* | [t,h,e,_,c,a,t,_,s,a,t,_o,n,_t,h,e_m,a,t] |
| *Starting vocabulary* | [t] [h] [e] [c] [a] [s] [o] [n] [m] |

*GPT-2/3 and GPT-3.5/4 appear to have a vocab size of ~50,000 and ~100,000 respectively. Source.

# How BPE works*

Key Intuition: Start with each character as a token, and "merge" tokens that most frequently occur next to each other. Don't merge across whitespace. Stop when the size of your vocabulary reaches a user-defined limit (we will assume 12 in the example below)

| | |
|---|---|
| *Starting vocabulary* | [t] [h] [e] [c] [a] [s] [o] [n] [m] |
| *Starting corpus* | [t,h,e,_,c,a,t,_,s,a,t,_o,n,_t,h,e_m,a,t] |
| *Frequency of adj tokens* | [t,h] – 2 [h,e] – 2 [c,a] – 1 [a,t] – 3 [s,a] – 1 [o,n] – 1 [m,a] – 1 |
| *Vocabulary after 1st merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at] |
| *Corpus after 1st merge* | t,h,e,_,c,at,_,s,at,_,o,n,_,t,h,e_,m,at |



*For simplicity, we are ignoring whitespace in the example

# How BPE works

Key Intuition: Start with each character as a token, and "merge" tokens that most frequently occur next to each other. Don't merge across whitespace. Stop when the size of your vocabulary reaches a user-defined limit (we will assume 12 in the example below)

| | |
|---|---|
| *Starting vocabulary* | [t] [h] [e] [c] [a] [s] [o] [n] [m] |
| *Starting corpus* | [t,h,e,_,c,a,t,_,s,a,t,_,o,n,_,t,h,e,_,m,a,t] |
| *Frequency of adj tokens* | [t,h] – 2 [h,e] – 2 [c,a] – 1 [a,t] – 3 [s,a] – 1 [o,n] – 1 [m,a] – 1 |
| *Vocabulary after 1st merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at] |
| *Corpus after 1st merge* | t,h,e,_,c,at,_,s,at,_,o,n,_,t,h,e_,m,at |
| *Frequency of adj tokens* | [t,h] – 2 [h,e] – 2 [c,at] – 1 [s,at] – 1 [o,n] – 1 [m,at] – 1 |
| *Vocabulary after 2nd merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at][th] |
| *Corpus after 2nd merge* | th,e,_,c,at,_,s,at,_,o,n,_,th,e,_,m,at |

# How BPE works

Key Intuition: Start with each character as a token, and "merge" tokens that most frequently occur next to each other. Don't merge across whitespace. Stop when the size of your vocabulary reaches a user-defined limit (we will assume 12 in the example below)

| | |
|---|---|
| *Starting vocabulary* | [t] [h] [e] [c] [a] [s] [o] [n] [m] |
| *Starting corpus* | [t,h,e,_,c,a,t,_,s,a,t,_,o,n,_,t,h,e,_,m,a,t] |
| *Frequency of adj tokens* | [t,h] – 2 [h,e] – 2 [c,a] – 1 [a,t] – 3 [s,a] – 1 [o,n] – 1 [m,a] – 1 |
| *Vocabulary after 1st merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at] |
| *Corpus after 1st merge* | t,h,e,_,c,at,_,s,at,_,o,n,_,t,h,e_,m,at |
| *Frequency of adj tokens* | [t,h] – 2 [h,e] – 2 [c,at] – 1 [s,at] – 1 [o,n] – 1 [m,at] – 1 |
| *Vocabulary after 2nd merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at][th] |
| *Corpus after 2nd merge* | th,e,_,c,at,_,s,at,_,o,n,_,th,e,_,m,at |
| *Frequency of adj tokens* | [th,e] – 2 [c,at] – 1 [s,at] – 1 [o,n] – 1 [m,at] – 1 |
| *Vocabulary after 3rd merge* | [t] [h] [e] [c] [a] [s] [o] [n] [m] [at][th][the] |
| *Corpus after 3rd merge* | the,_,c,at,_,s,at,_,o,n,_,the,_,m,at |

# How BPE works

The merges happened in this order:

- a,t => at
- t,h => th
- th,e => the

When a new piece of text arrives, the BPE tokenization will apply the merges in the same order.

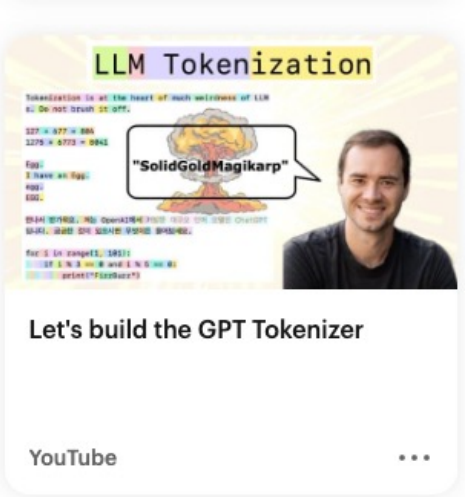
Example: [t,h,e,_,r,a,t]

- [t,h,e,_,r,at]
- [th,e,_,r,at]
- [the,_,r,at]

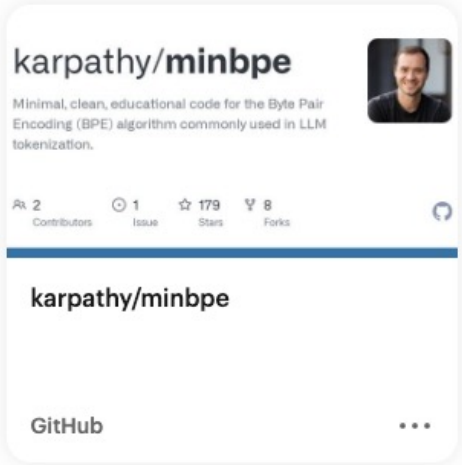
# Quick Demo – Token Visualizer

https://observablehq.com/@simonw/gpt-tokenizer

# Sidebar: If you want to see how to code BPE from scratch …




Link




Link

# 15.773 Hands-on Deep Learning

Spring 2024