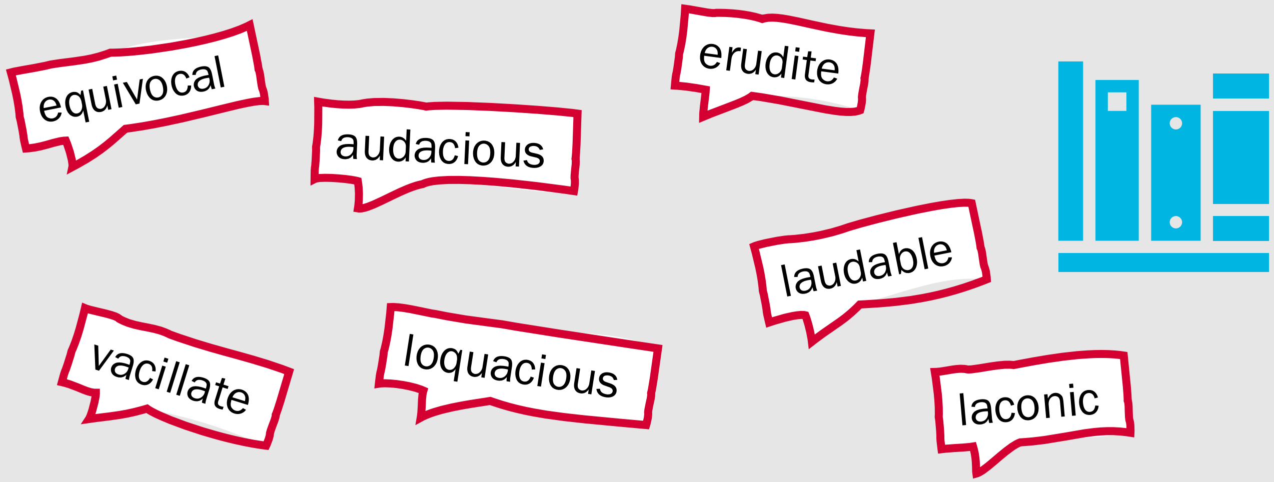
Transfer Learning with Pretrained Language Models and Large Language Models

Natalie Parde

UIC CS 521

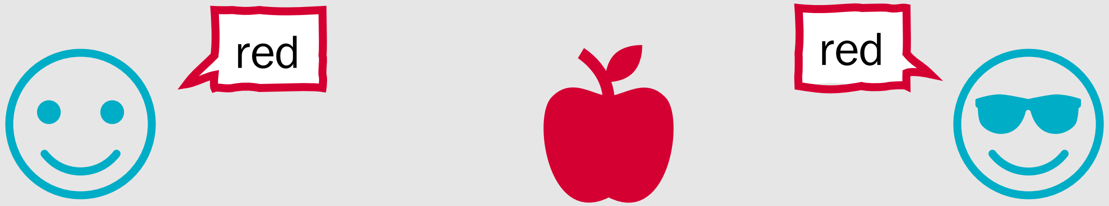
Language continually develops and evolves.

- Estimated vocabulary size of a young adult speaker of American English: ~30k-100k words
  - On average, 7-10 new words need to be learned per day through age 20!
- Early on in humans: Vocabulary is learned via spoken interactions with peers and caregivers
- Later: Vocabulary is mostly learned as a by-product of reading



Can computers learn language in the same way?

- Learning language through experience (e.g., through spoken interactions with peers in a situated environment) is an example of **grounded language learning**
  - Meaning is tied to an experiential (either implied or explicit) **common ground** between speakers



Recap: The distributional hypothesis states that we can learn language based solely on its context

- Word embedding techniques “learn” meaning using measures of the frequency with which words occur close to one another in large text corpora
- Recall:
  - **Word2Vec**
  - **GloVe**

What does this look like?

Start with an input *t*

super

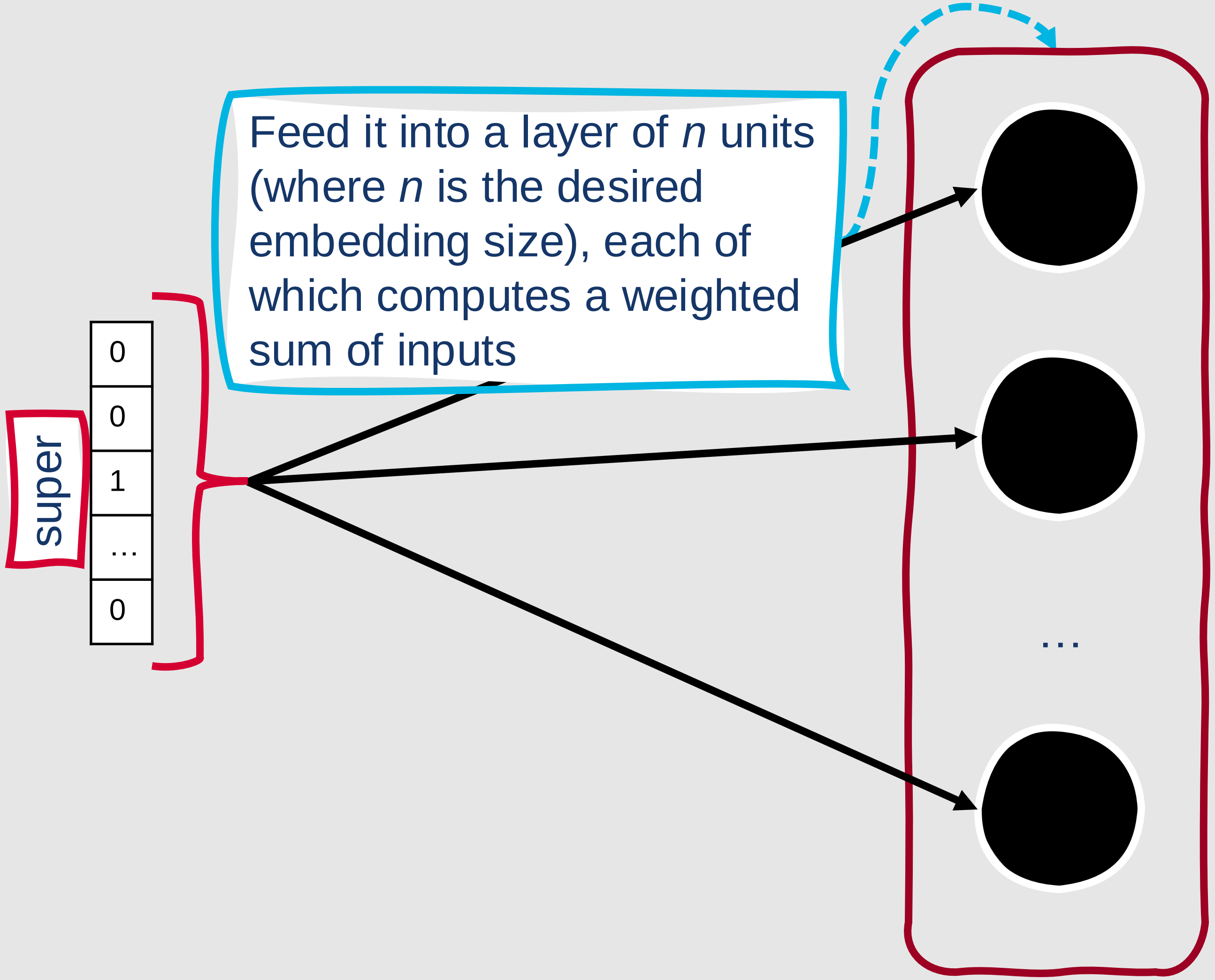
What does this look like?

Get the one-hot vector for *t*

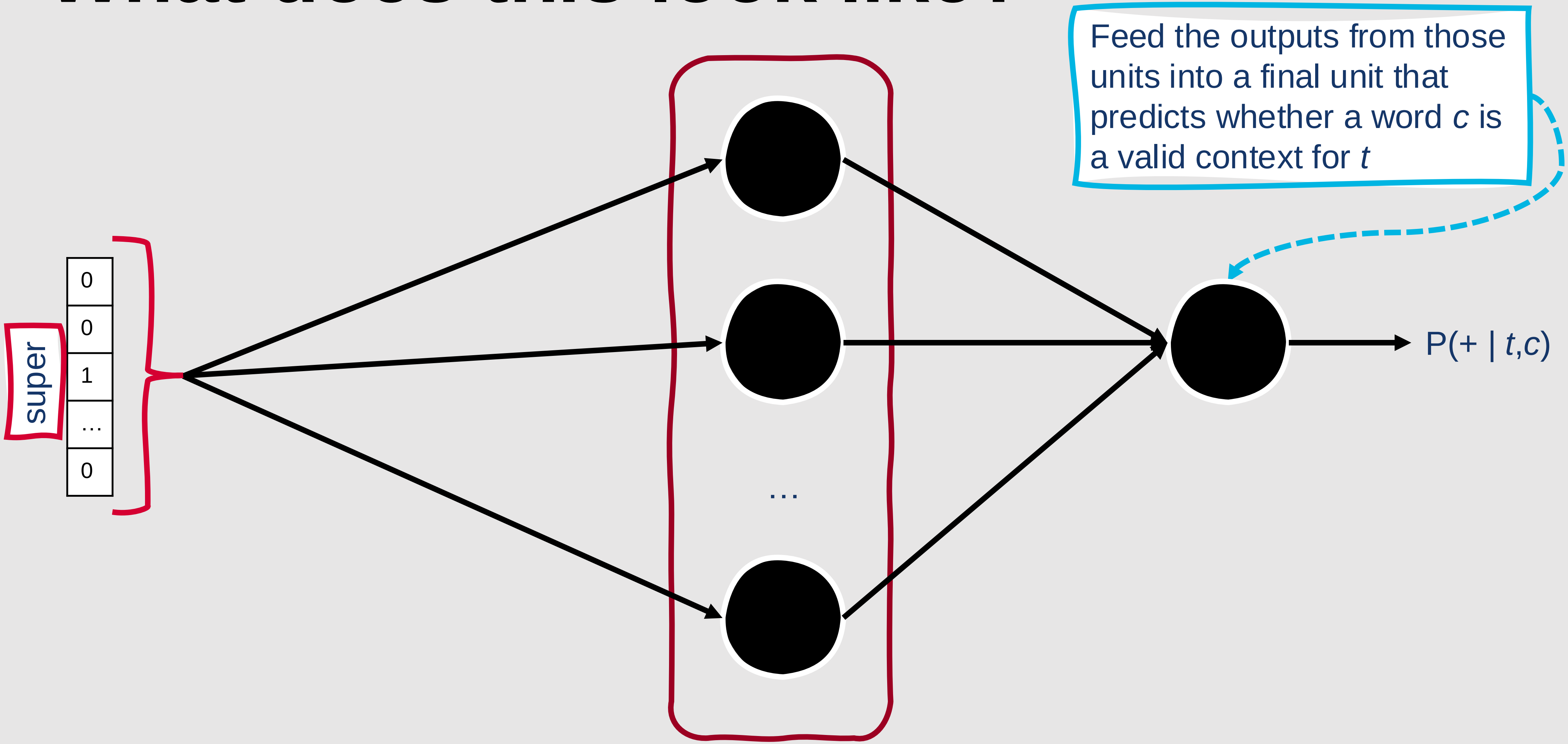
$$\begin{array}{cc} \text{super} & \begin{bmatrix} 0 \\ 0 \\ 1 \\ \dots \\ 0 \end{bmatrix} \end{array}$$

What does this look like?

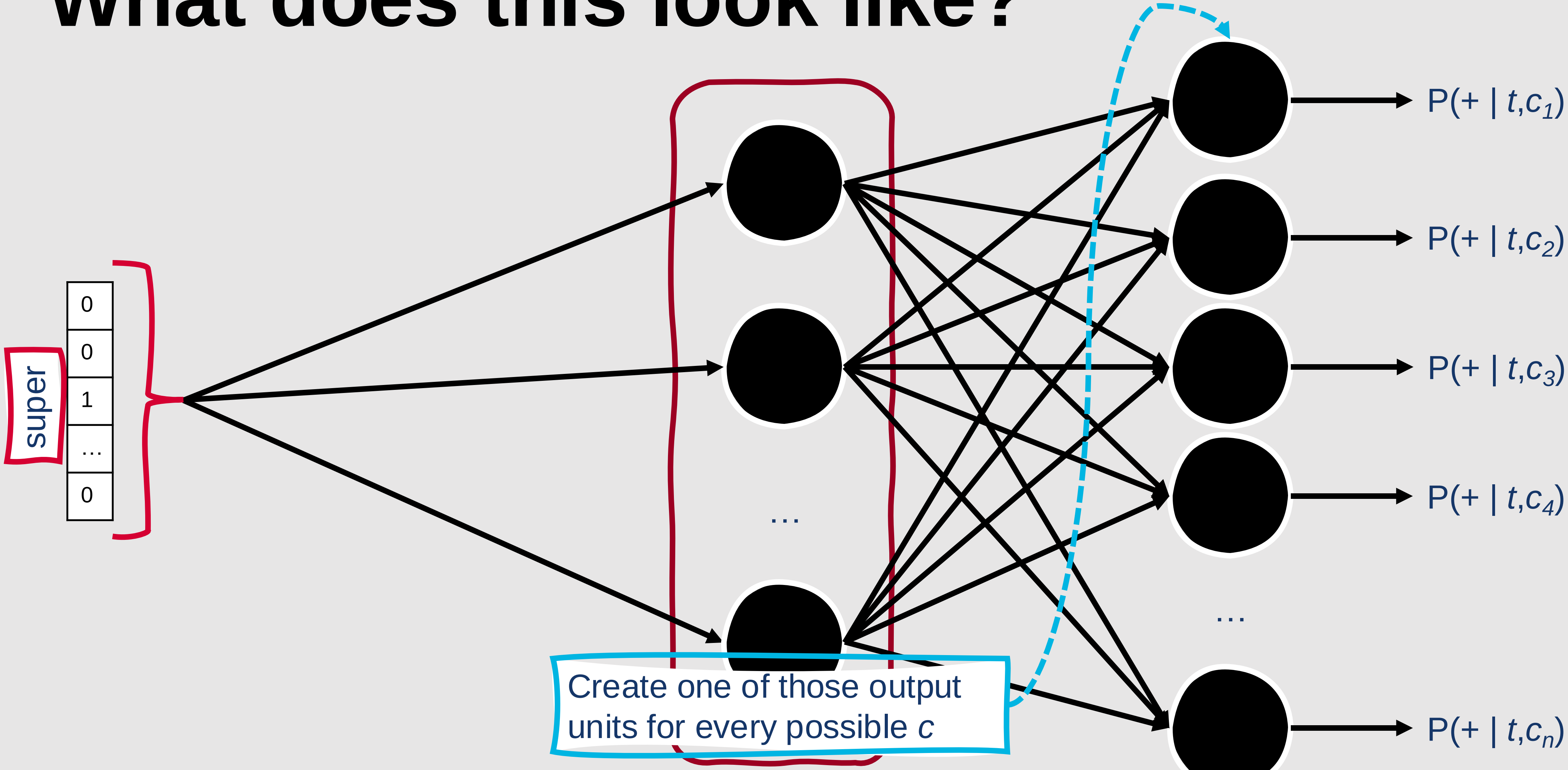
Feed it into a layer of *n* units (where *n* is the desired embedding size), each of which computes a weighted sum of inputs



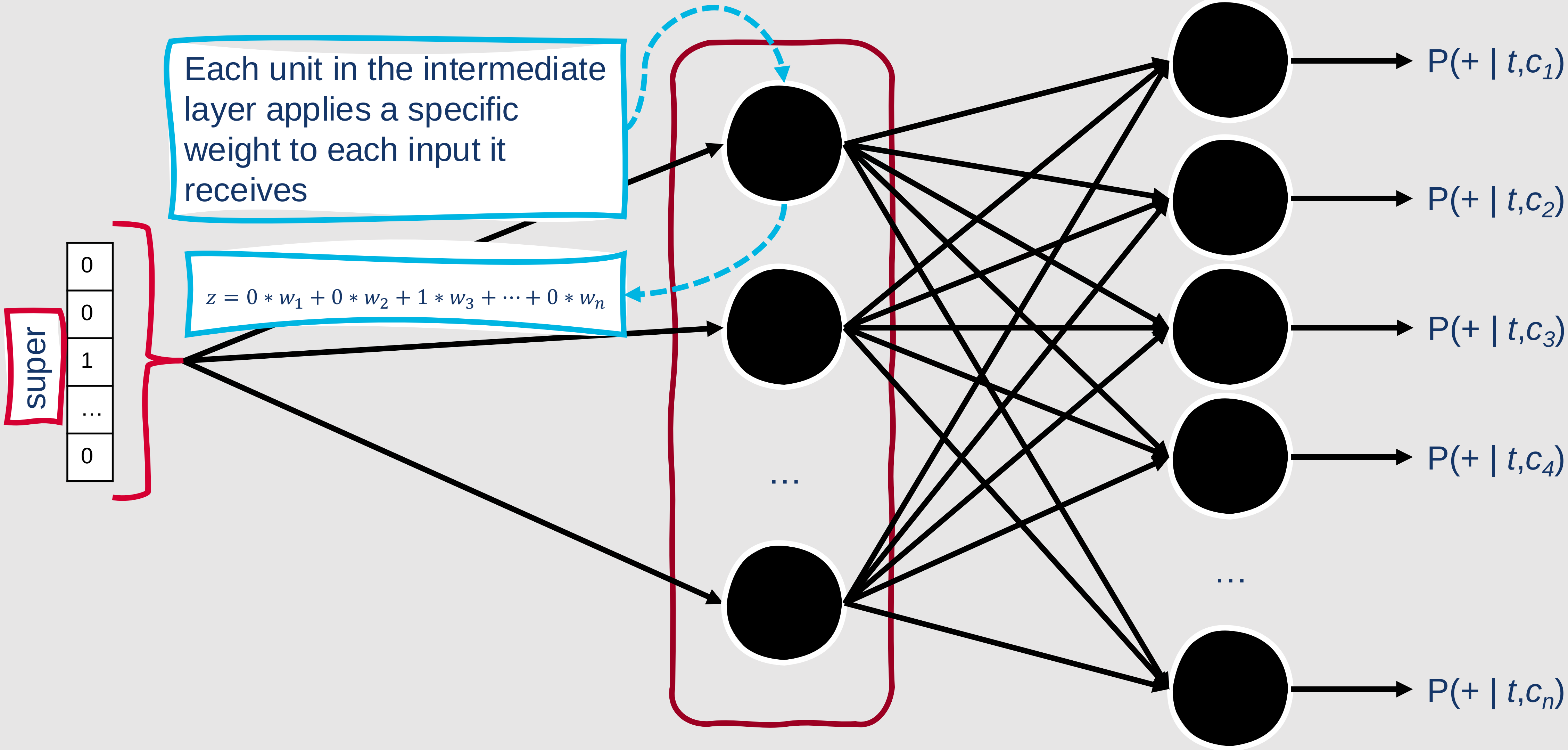
What does this look like?



What does this look like?

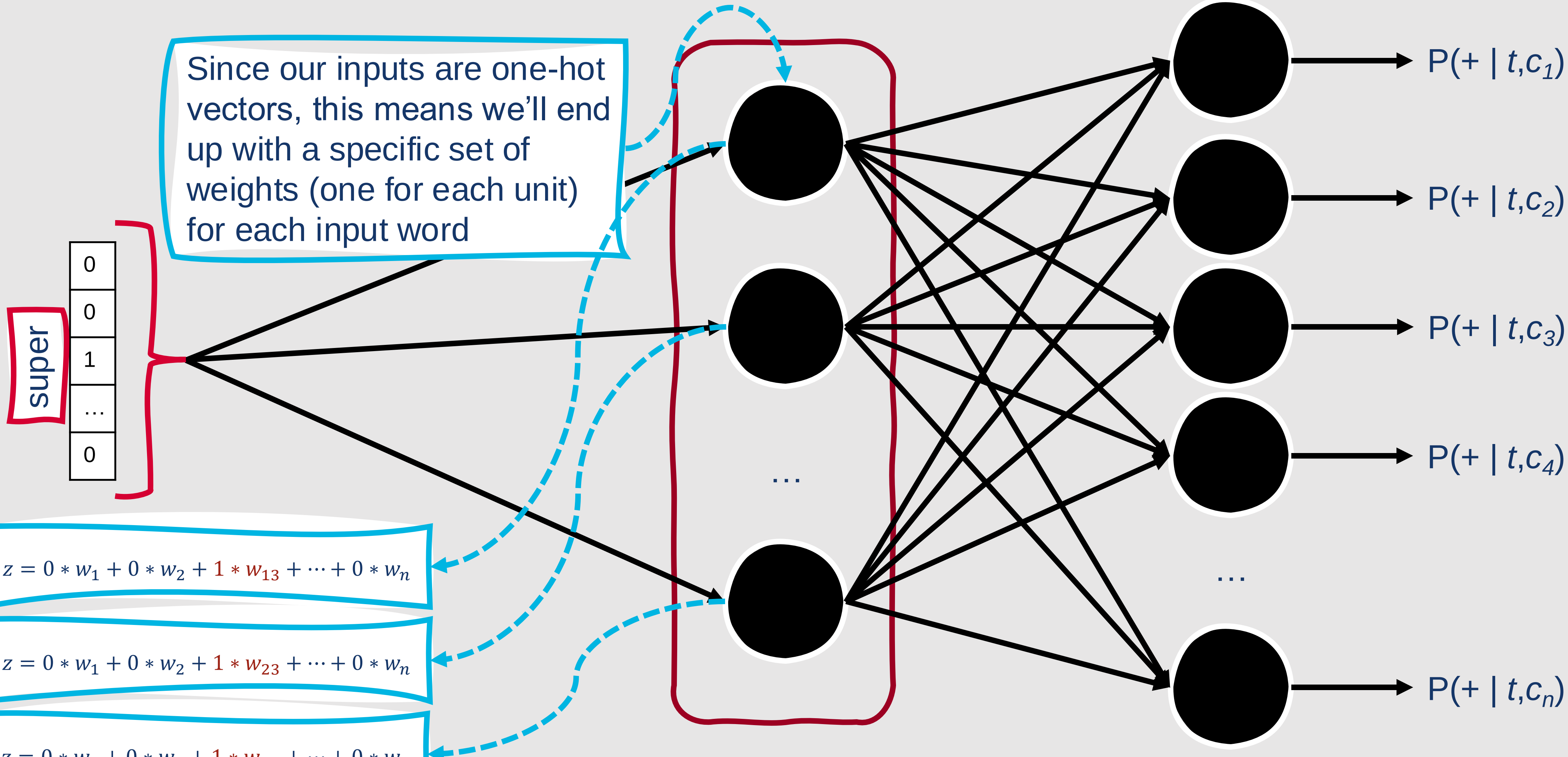


Behind the scenes….



Behind the scenes….

Since our inputs are one-hot vectors, this means we'll end up with a specific set of weights (one for each unit) for each input word

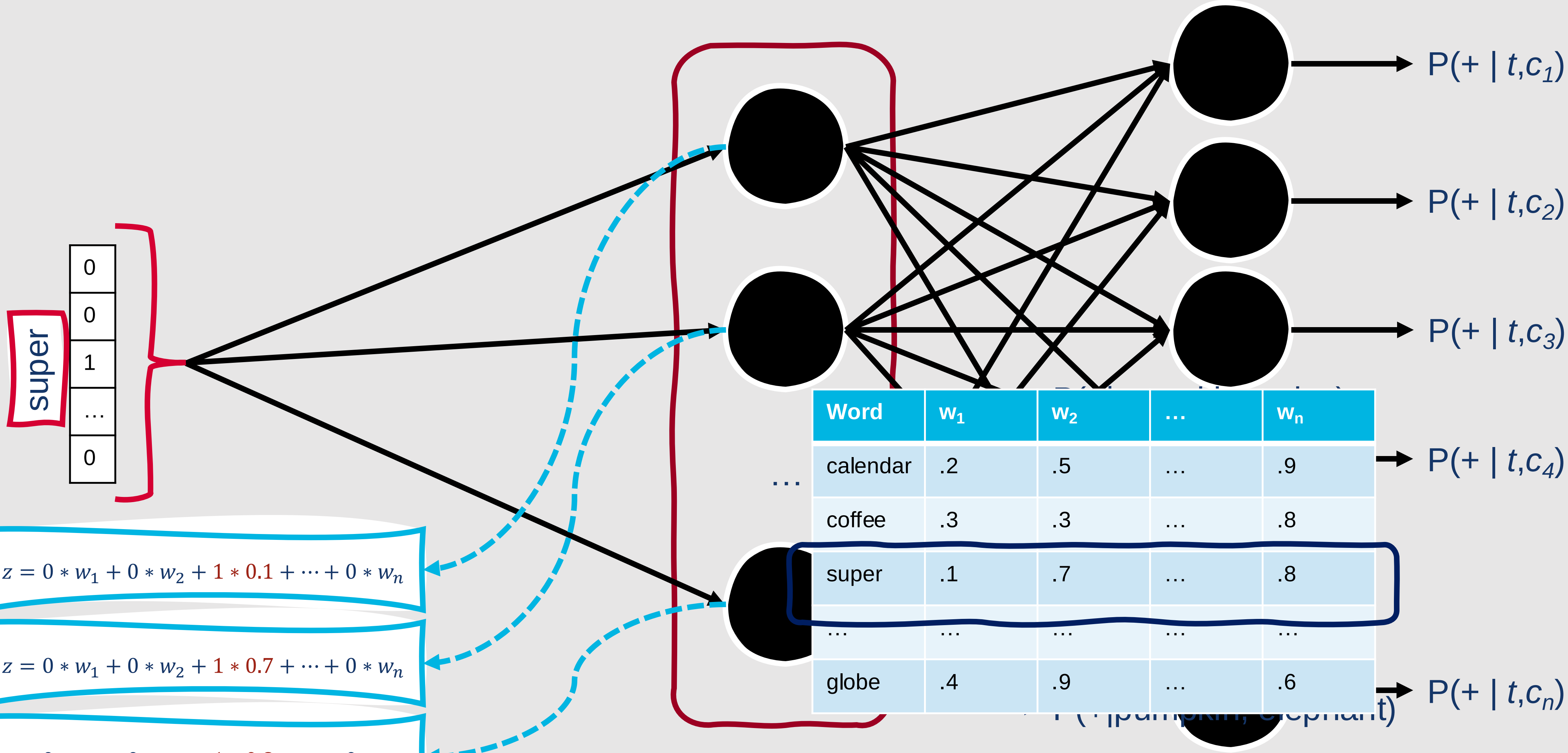


$z = 0 * w_1 + 0 * w_2 + 1 * w_{13} + \cdots + 0 * w_n$

$z = 0 * w_1 + 0 * w_2 + 1 * w_{23} + \cdots + 0 * w_n$

$z = 0 * w_1 + 0 * w_2 + 1 * w_{n3} + \cdots + 0 * w_n$

These are the weights we're interested in!



Word	w_1	w_2	…	w_n
calendar	.2	.5	…	.9
coffee	.3	.3	…	.8
super	.1	.7	…	.8
…	…	…	…	…
globe	.4	.9	…	.6

$z = 0 * w_1 + 0 * w_2 + 1 * 0.1 + \cdots + 0 * w_n$

$z = 0 * w_1 + 0 * w_2 + 1 * 0.7 + \cdots + 0 * w_n$

$z = 0 * w_1 + 0 * w_2 + 1 * 0.8 + \cdots + 0 * w_n$

GloVe

- While Word2Vec is a popular **predictive word embedding model**, researchers have also developed high-performing models that incorporate aspects of **count-based models**
- One example: Global Vectors for Word Representation **(GloVe)**
- Why is this useful?
  - Predictive models → black box
    - They work, but why?
  - GloVe models are easier to interpret
- GloVe models also encode the ratios of co-occurrence probabilities between different words …this makes these vectors useful for word analogy tasks

How does GloVe work?

	c_1	…	c_n
t_1	123	…	456
…	…	…	…
t_n	0	…	789

Build a huge word-context co-occurrence matrix

Define soft constraints for each word pair

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

Define a cost function

$$J = \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij})(w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

Minimize the cost function to learn ideal embedding values for w_i and w_j

0.4	0.7	1.2	4.3	0.9	6.7	1.3	0.5	0.7	5.3

Word2Vec and GloVe are both *static* word embeddings.

- A given word has the same embedding, regardless of its context
- Reasonable in many cases, but not always
  - What if a word has multiple senses?
  - What if a word starts appearing in new contexts?

Did you deposit that check at the **bank**?

0.4	0.2	0.5	0.7	0.1

A message in a bottle washed up on the **bank**.

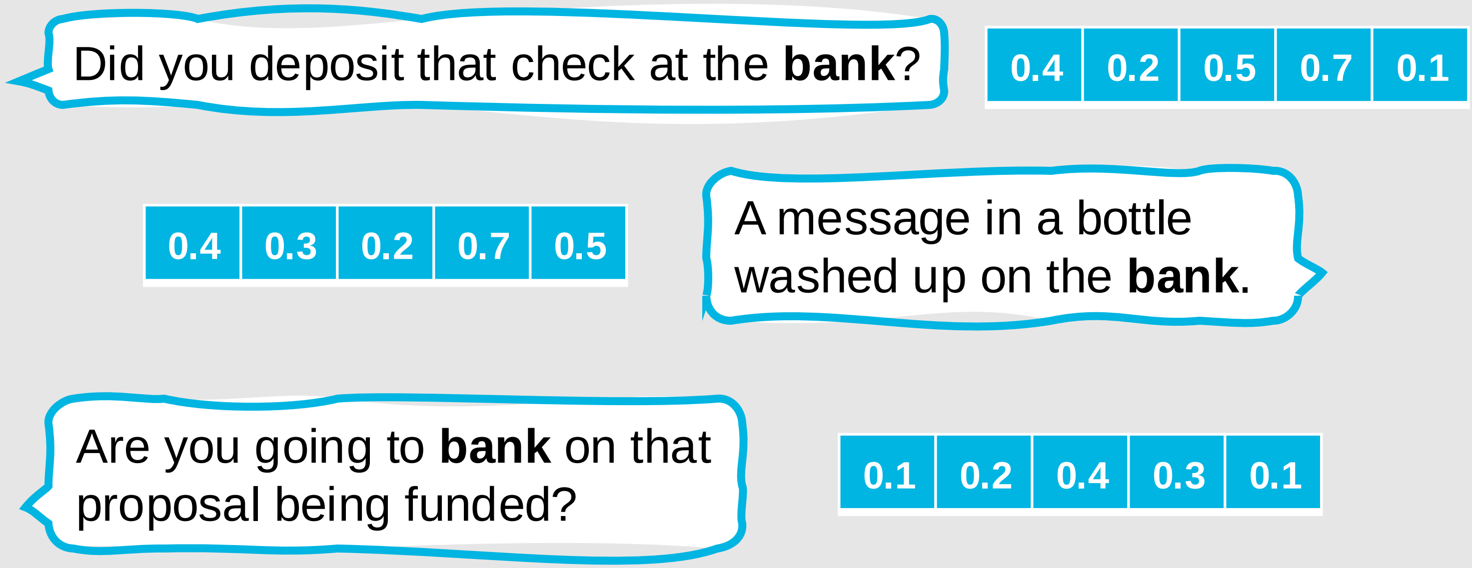
0.4	0.2	0.5	0.7	0.1

Are you going to **bank** on that proposal being funded?

0.4	0.2	0.5	0.7	0.1

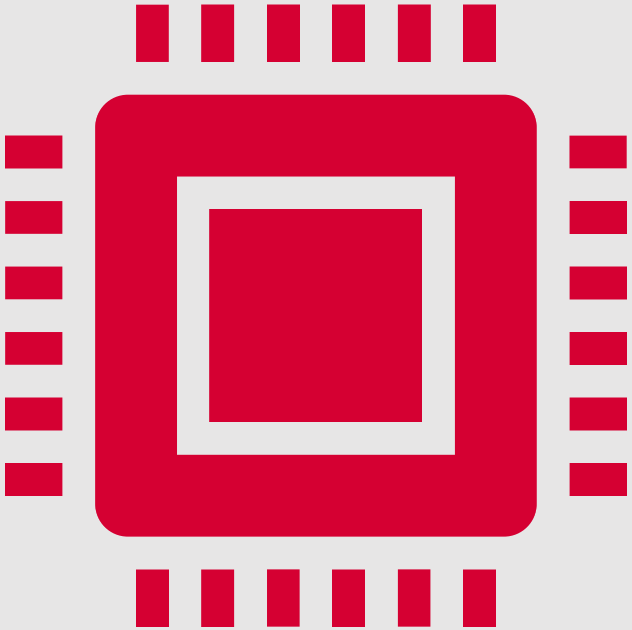
Contextual Word Embeddings

- Word representations that differ depending on the context in which the word appears
- Vocabulary words do *not* map to specific, predefined vectors
- We typically learn contextual word representations using **pretrained language models**



What base architecture should we use for pretrained language models?

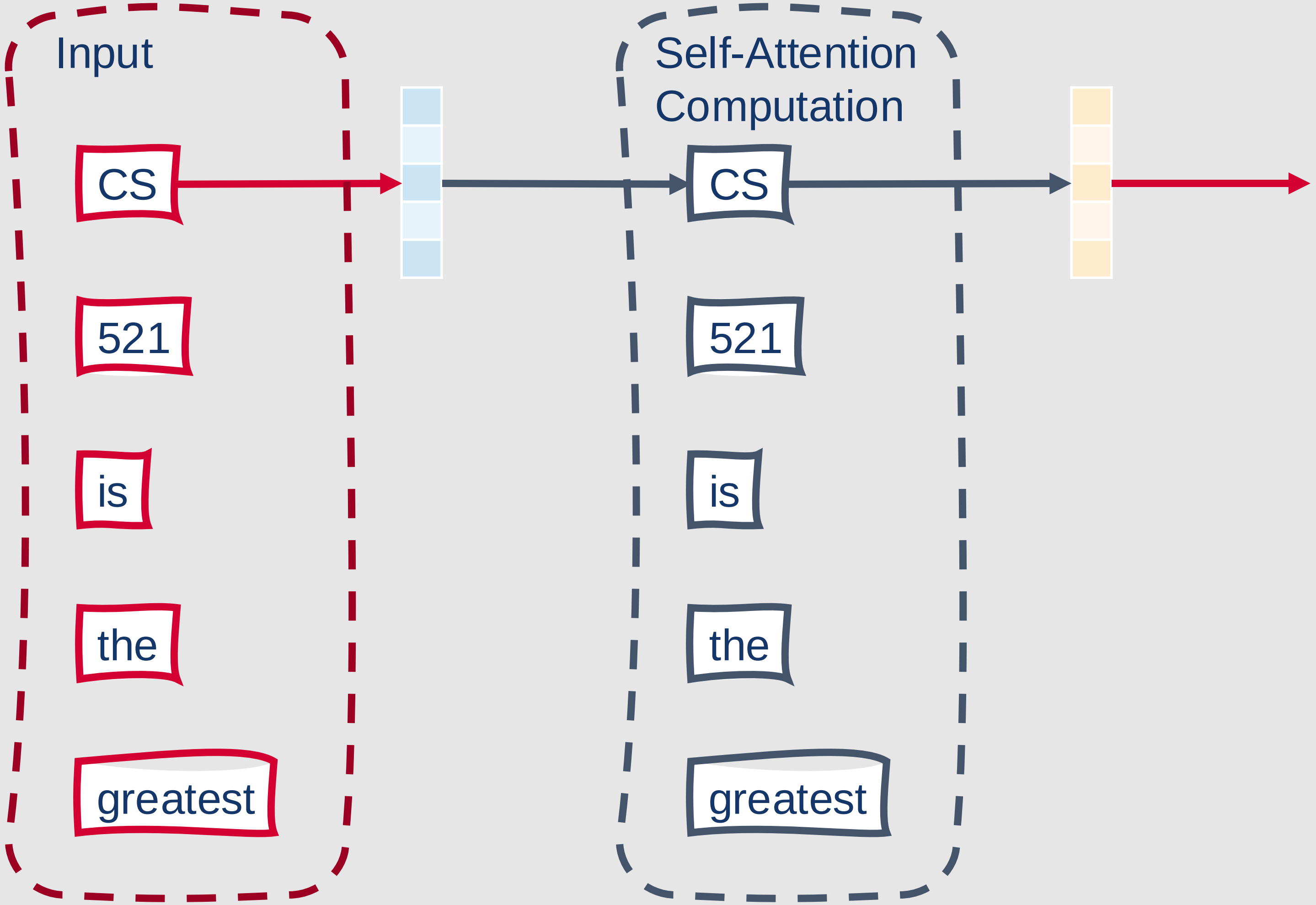
- Limitations of RNNs:
  - Processing long-distance dependencies through **many recurrences** can eventually lead to loss of valuable information
  - Recurrent models cannot productively leverage **parallel resources**



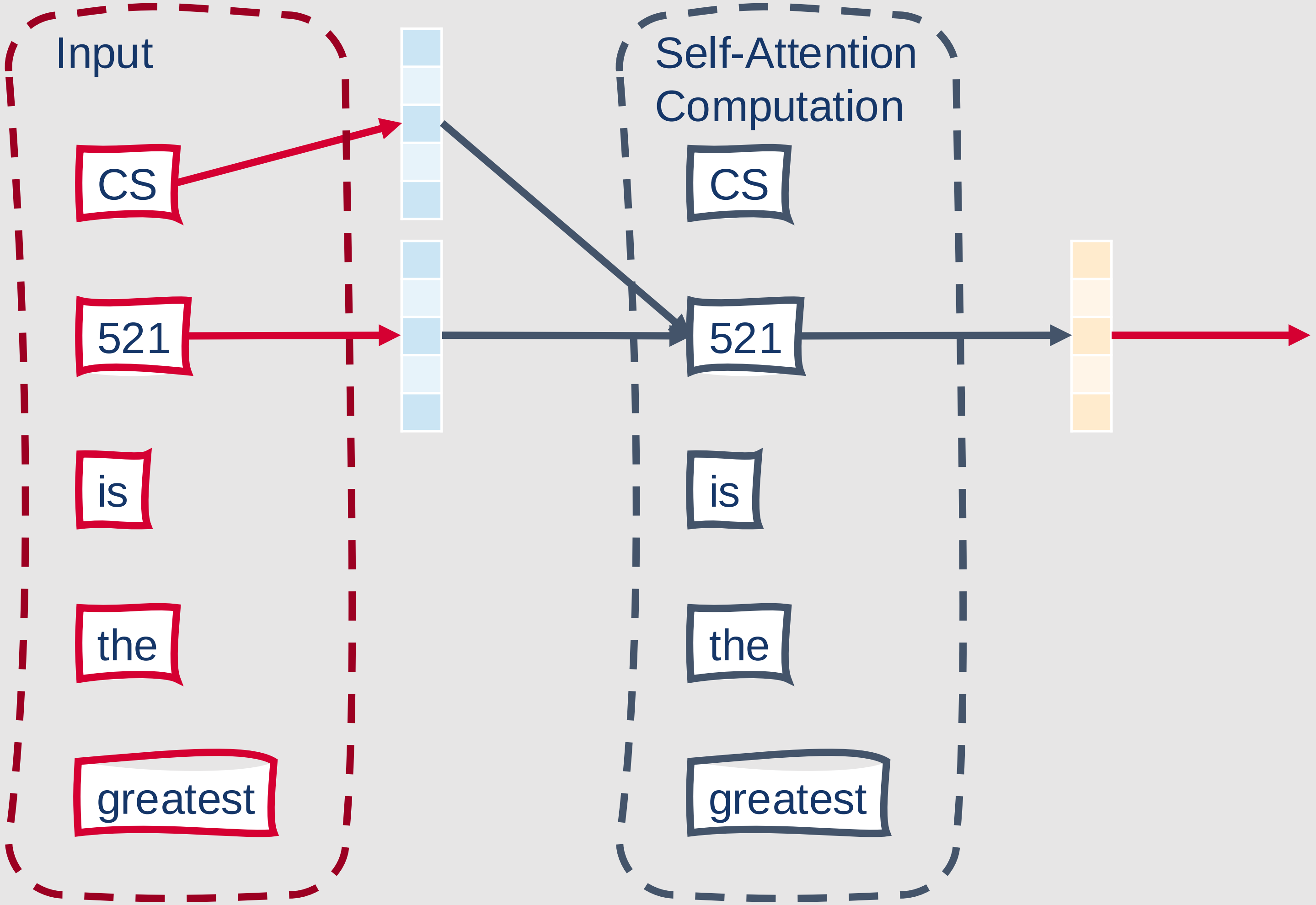
Transformers

- Entirely do away with recurrences
- Stacks of:
  - Linear layers
  - Feedforward layers
  - **Self-attention** layers
    - For a given element in a sequence, determines which other element(s) up to that point are most relevant to it
      - Each computation is independent of other computations → easy parallelization
      - Each computation only considers elements up to that point in the sequence → easy language modeling

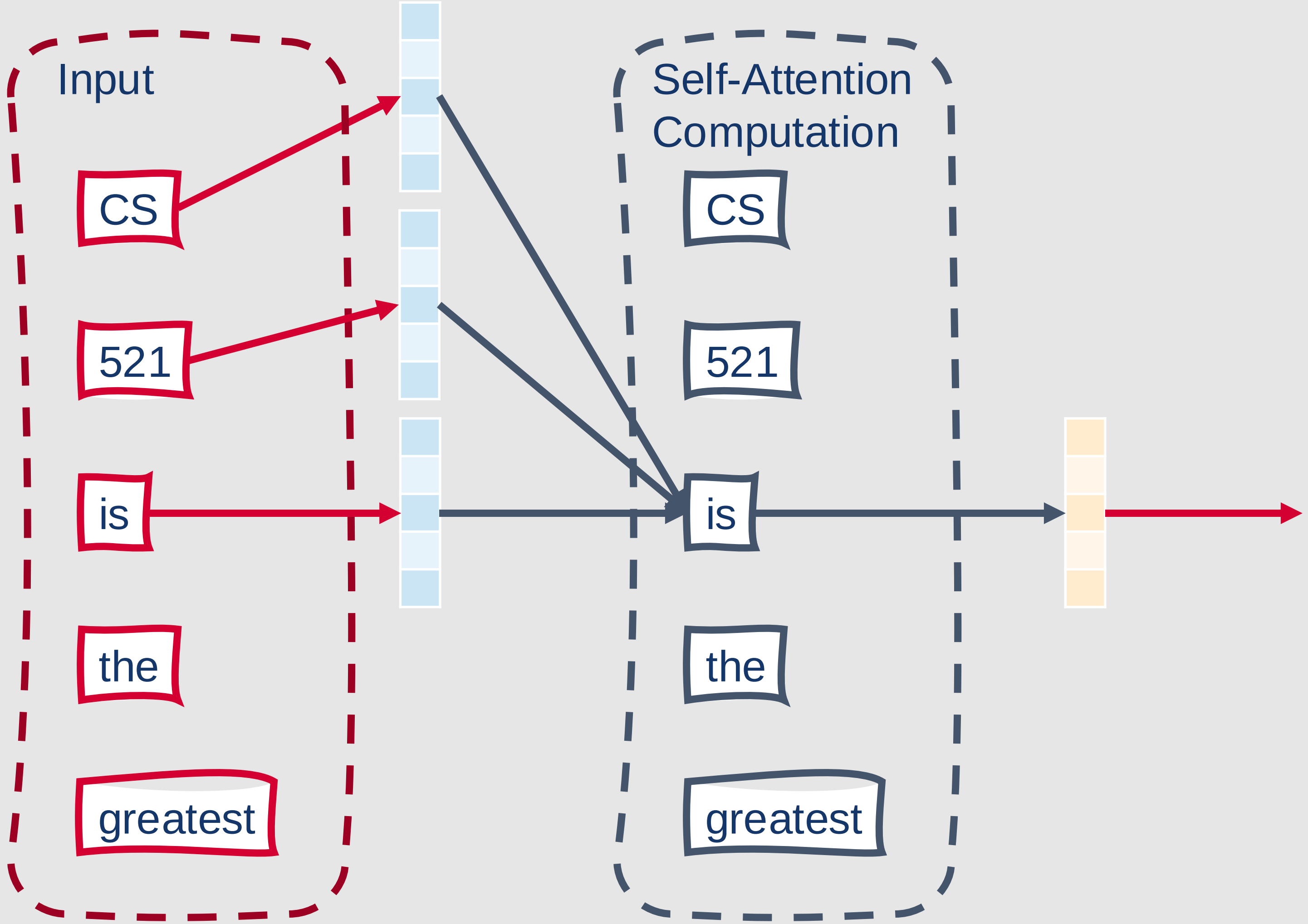
Self-Attention



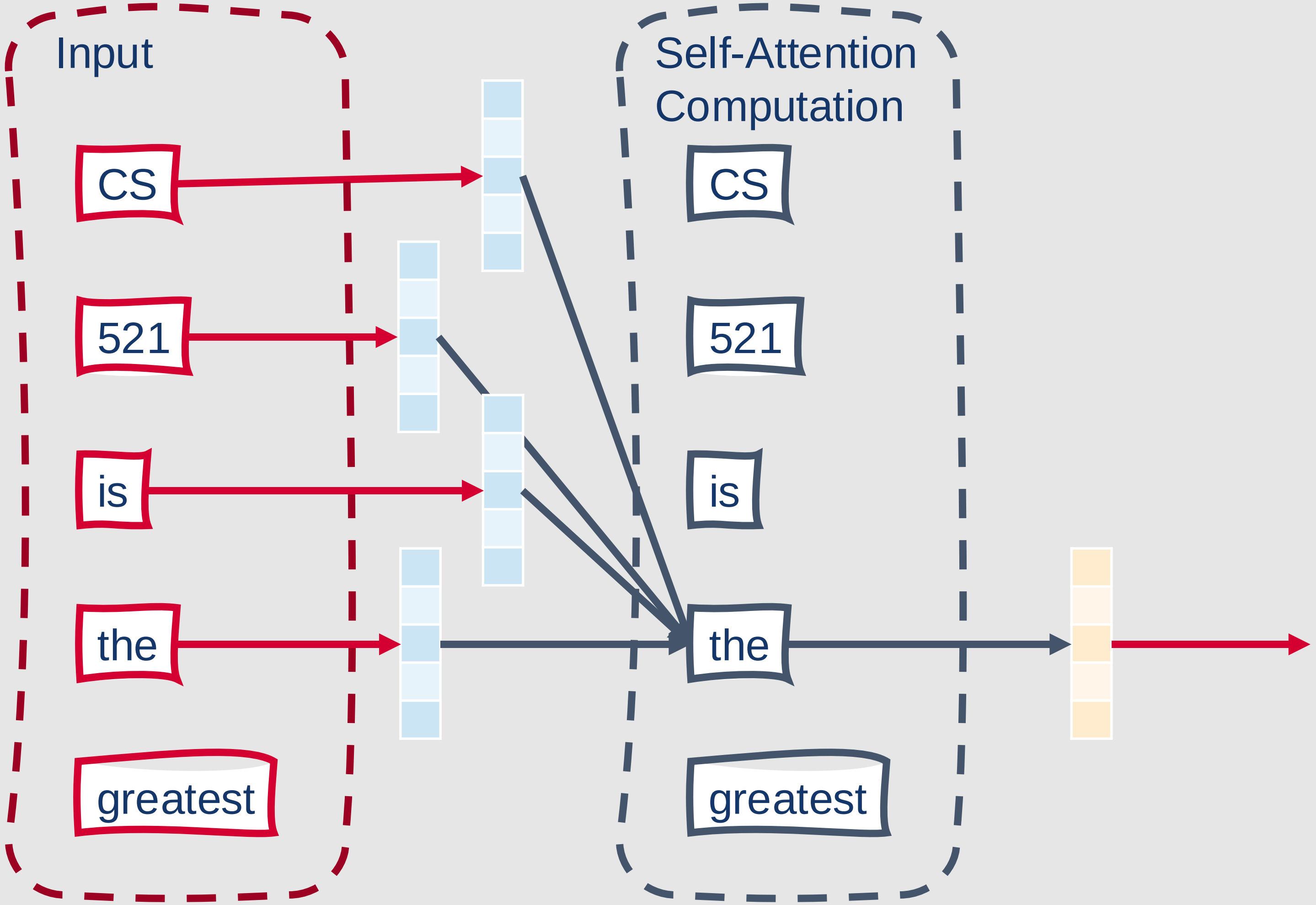
Self-Attention



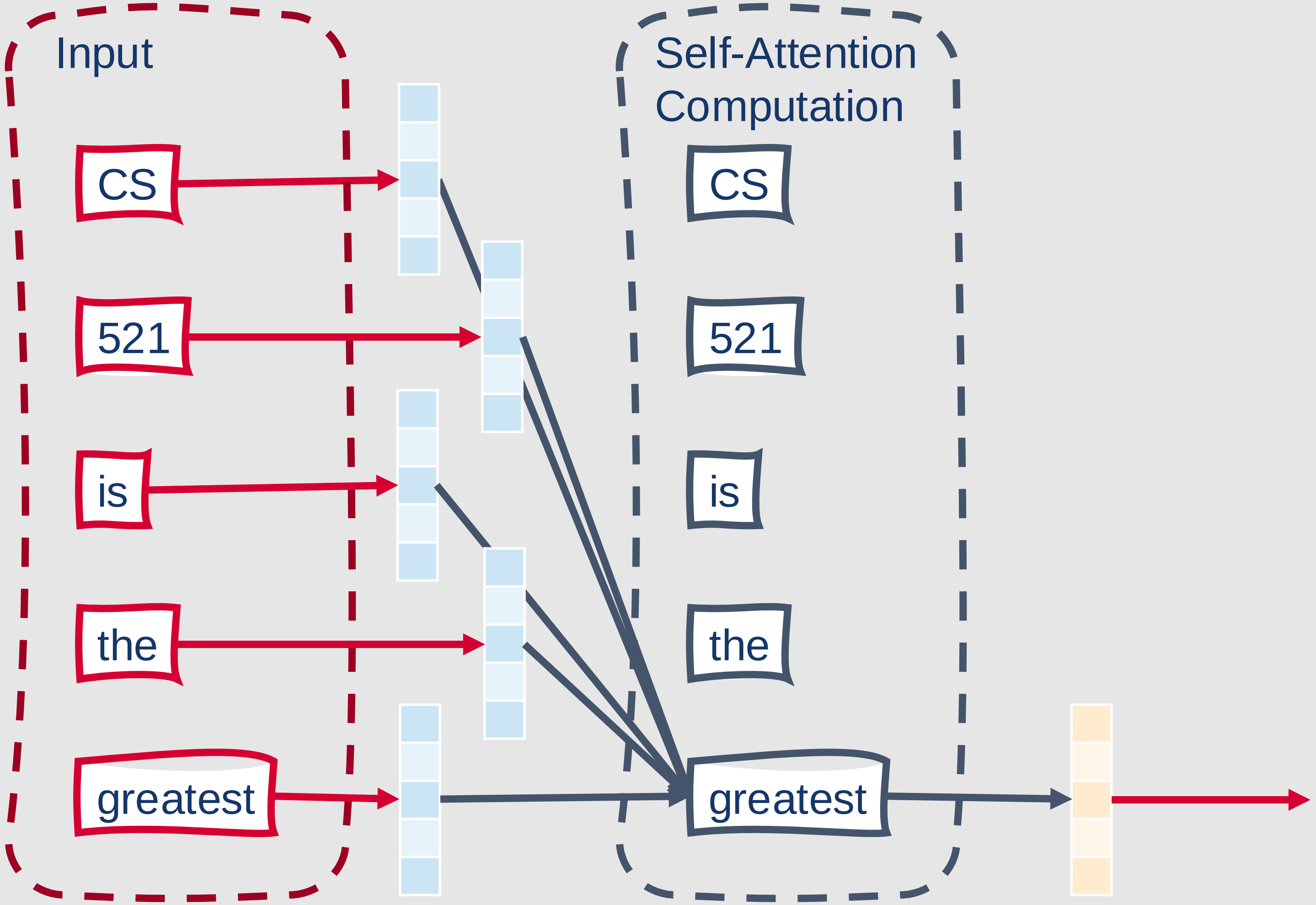
Self-Attention



Self-Attention



Self-Attention



Computing Self-Attention

- Take the dot product between a given input element x_i and each input element $(x_1, \ldots, x_i)$ up until that point
  - $\text{score}(x_i, x_j) = x_i \cdot x_j$
- Apply softmax normalization to create a vector of weights, α_i, indicating proportional relevance of each sequence element to the current focus of attention, x_i
  - $\alpha_{ij} = \text{softmax}\left(\text{score}(x_i, x_j)\right) \forall j \leq i = \frac{e^{score(x_i, x_j)}}{\sum_{k=1}^{i} e^{score(x_i, x_k)}} \forall j \leq i$
- Take the sum of inputs thus far weighted by α_i to produce an output y_i
  - $y_i = \sum_{j \leq i} \alpha_{ij} x_j$

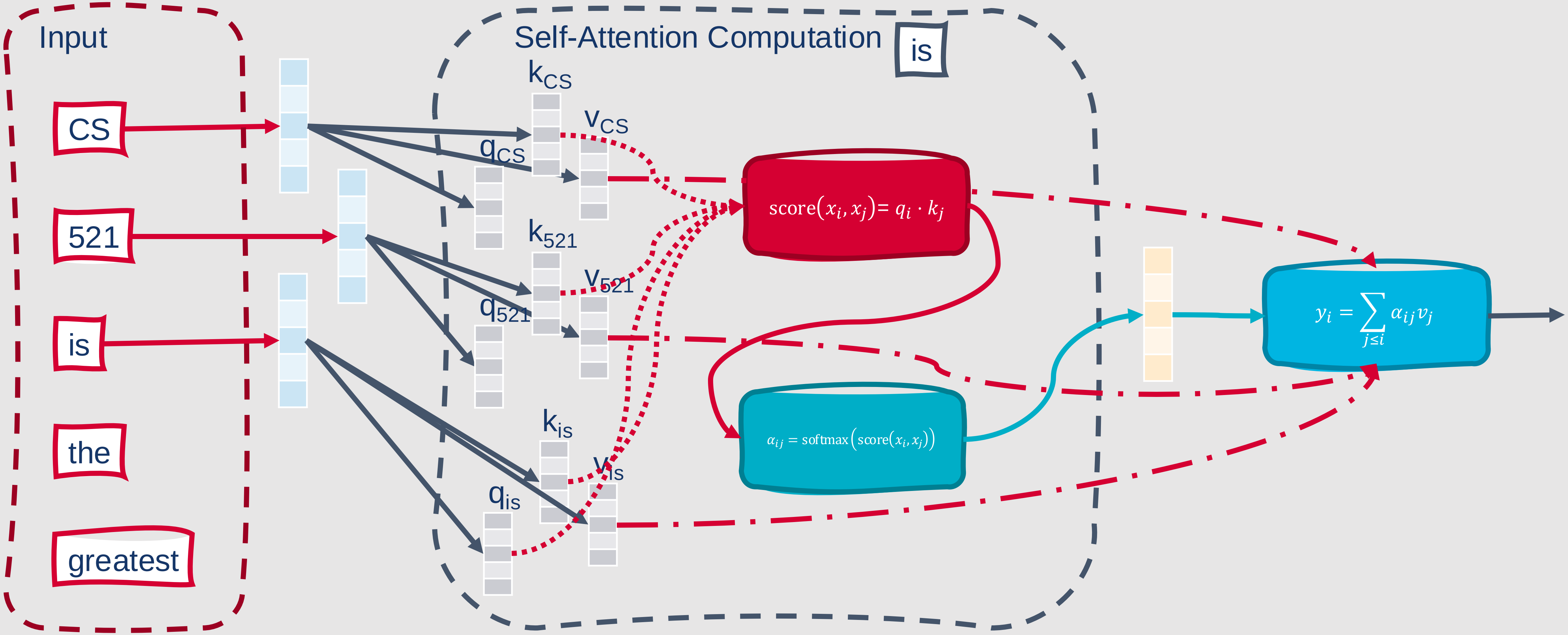
How do Transformers learn?

- Continually updating weight matrices applied to inputs
- Weight matrices are learned for each of three roles when computing self-attention:
  - **Query:** The focus of attention when it is being compared to inputs up until that point, W^Q
  - **Key:** An input that is being compared to the focus of attention, W^K
  - **Value:** A value being used to compute the output for the current focus of attention, W^V

Training Transformers

- Weight matrices are applied to inputs in the context of their respective roles
  - $q_i = W^Q x_i$
  - $k_i = W^K x_i$
  - $v_i = W^V x_i$
- Then, we can update our equations for computing self-attention so that these roles are reflected in them:
  - $\text{score}(x_i, x_j) = q_i \cdot k_j$
  - $\alpha_{ij} = \text{softmax}\left(\text{score}(x_i, x_j)\right) \forall j \leq i$
  - $y_i = \sum_{j \leq i} \alpha_{ij} v_j$

Self-Attention

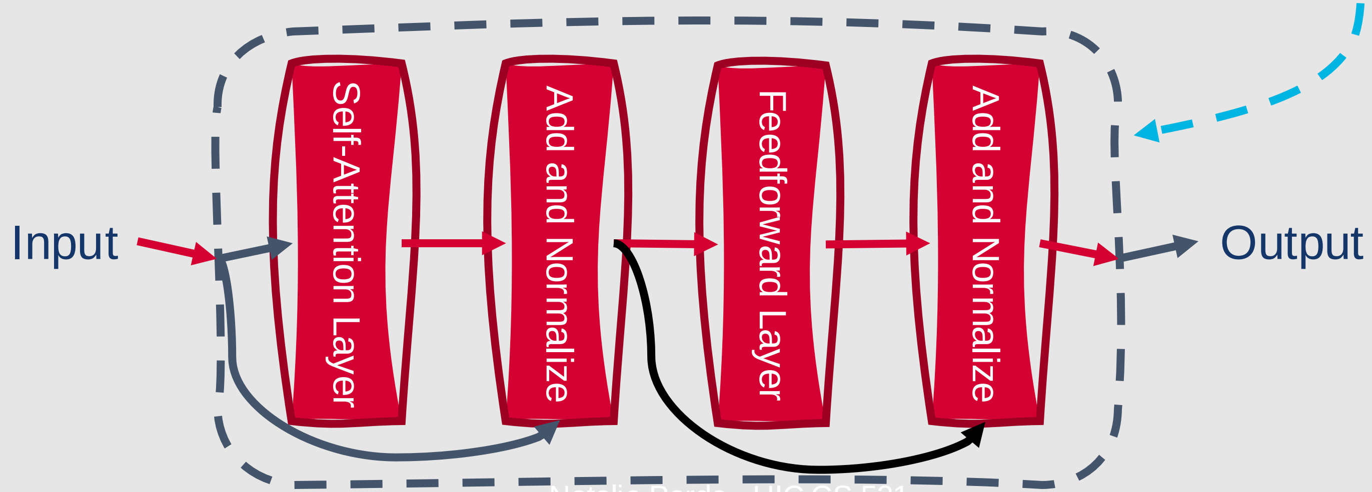


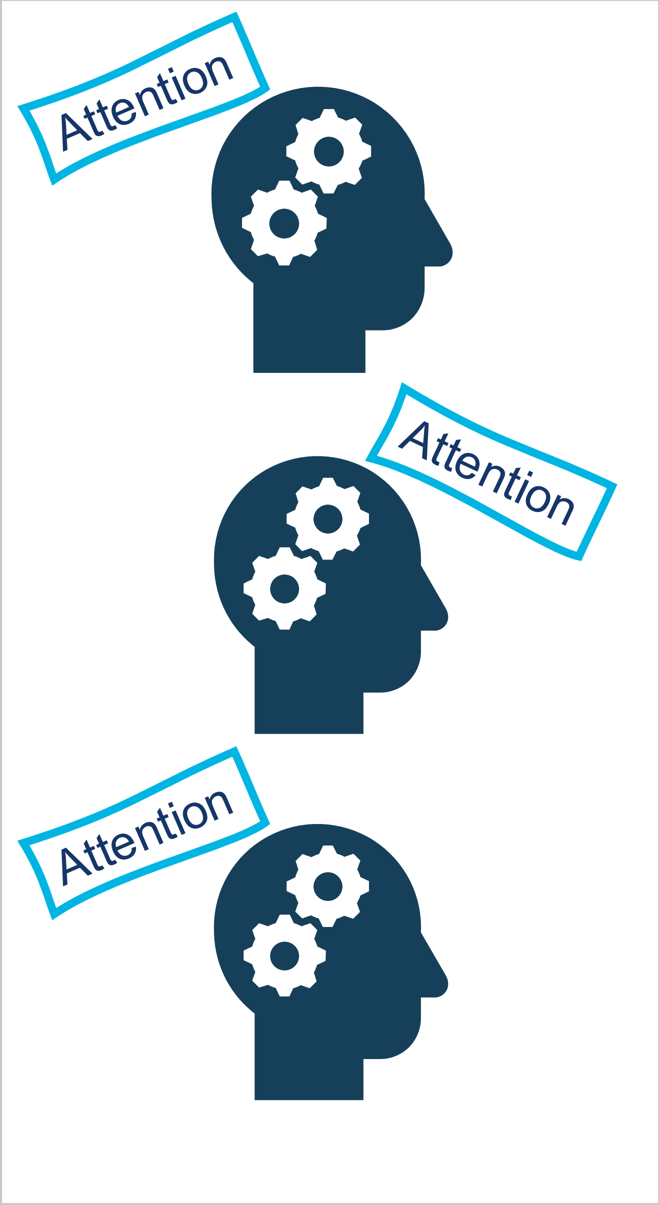
Practical Considerations

- Combining a dot product with an exponential (as in softmax) may lead to arbitrarily large values
- It is common to scale the scoring function based on the dimensionality of the key (and query) vectors, d_k
  - $\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$
- Each y_i is computed independently, so we can parallelize computations using matrix multiplication where X is a matrix containing all input embeddings
  - $Q = W^Q X$
  - $K = W^K X$
  - $V = W^V X$
  - $\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$
    - Make sure to avoid including knowledge of future words in autoregressive language modeling settings!

Transformer Blocks

- Self-attention is the central component of a **Transformer block**, which also includes:
  - Feedforward layers
  - Residual connections
  - Normalizing layers
- Transformer blocks can be stacked, just like RNN layers





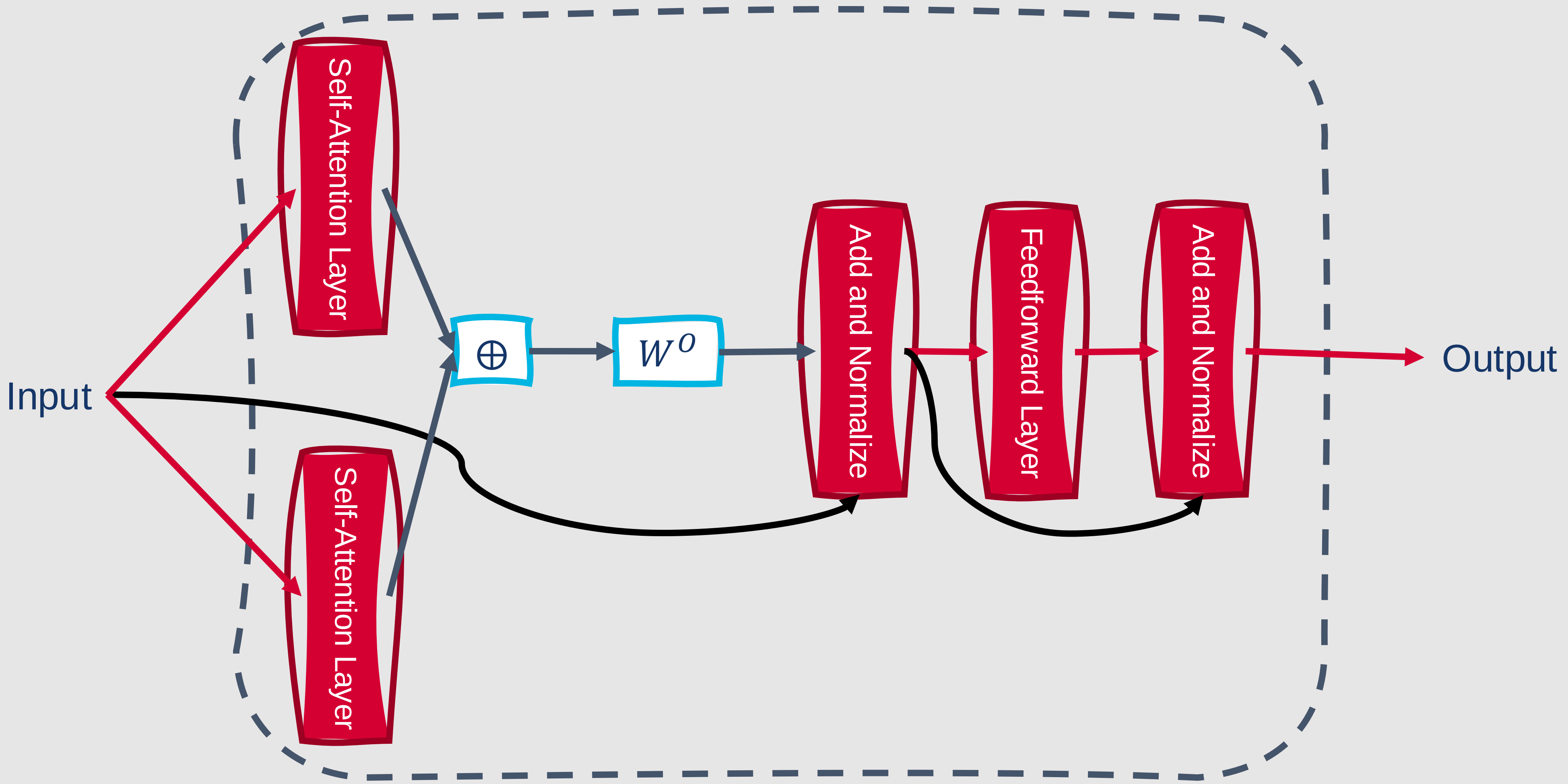
Multihead Attention

- Each self-attention layer represents a single **attention head**
- **Multihead attention** places multiple attention heads in parallel in the Transformer model
  - Since each attention head has its own set of weights, each one can learn different aspects of the relations between input elements at the same level of abstraction

Computing Multihead Attention

- Each head in the self-attention layer is parameterized with its own weights
  - $Q = W_i^Q X$
  - $K = W_i^K X$
  - $V = W_i^V X$
- The output of a multihead attention layer with n heads comprises n vectors of equal length
- These heads are concatenated and then reduced to the original input/output dimensionality
  - $\text{head}_i = \text{SelfAttention}(W_i^Q X, W_i^K X, W_i^V X)$
  - $\text{MultiheadAttention}(Q, K, V) = W^O(\text{head}_1 \oplus \text{head}_2 \oplus \ldots \oplus \text{head}_n)$

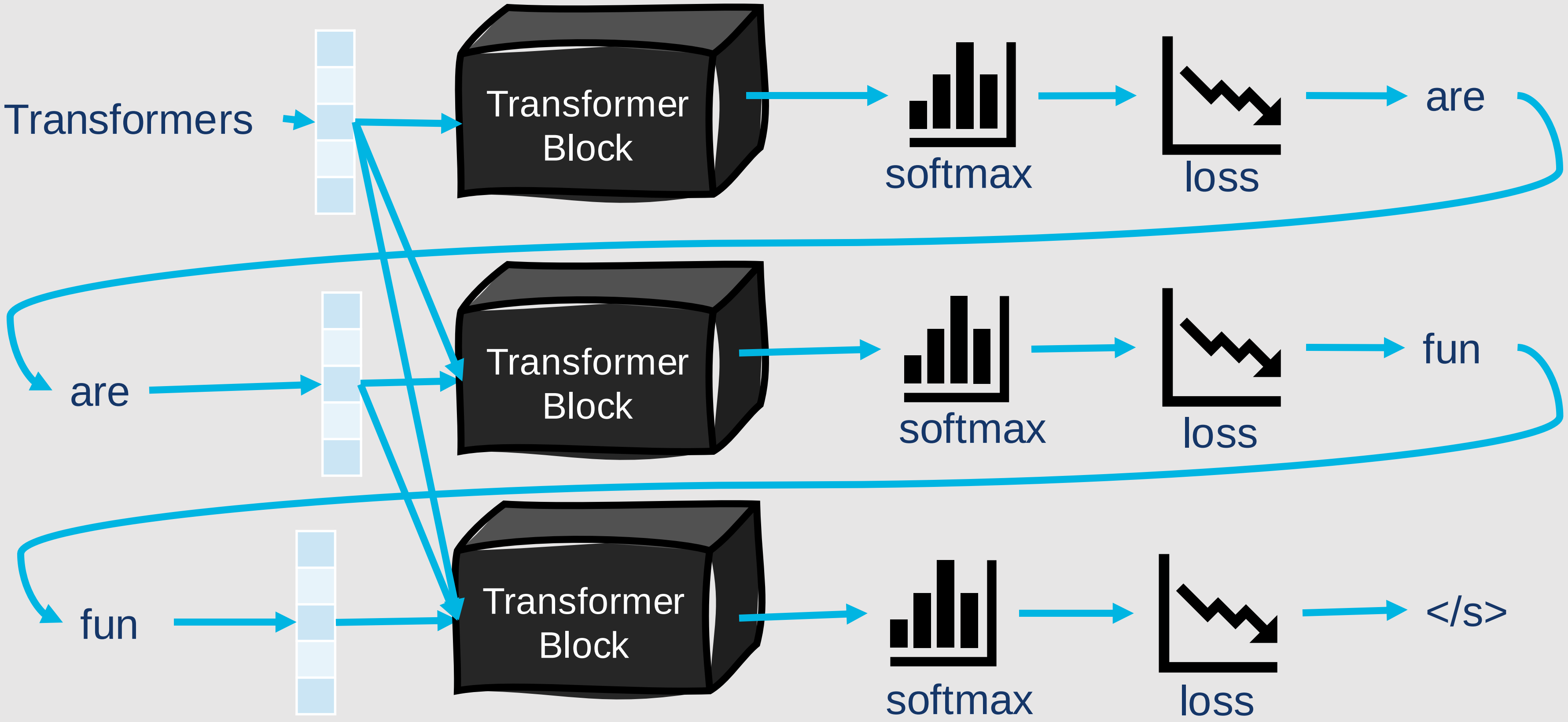
Multihead Attention



Positional Embeddings

- Since Transformers don't make use of recurrent connections, they instead employ separate **positional embeddings** to encode positionality
  - Randomly initialize an embedding for each input position
  - Update weights during the training process
  - Input embedding with positional information = word embedding + positional embedding
- Static functions mapping positions to vectors can be used as an alternative

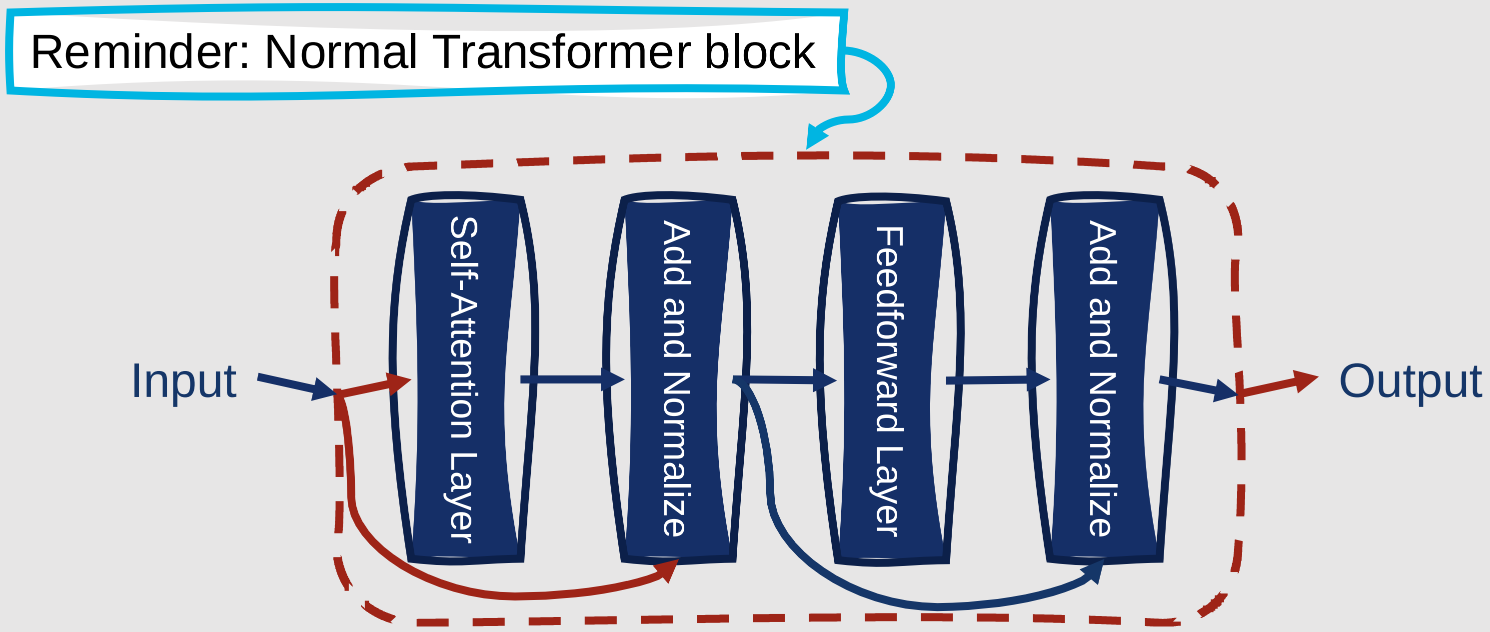
Transformers as Autoregressive Language Models



Encoder-Decoder Models with Transformers

- Similar to other encoder-decoder models
  - Encoder (Transformer model) maps sequential input to an output representation
  - Decoder (Transformer model) attends to the encoder representation and generates sequential output autoregressively
- However….
  - Transformer blocks in the decoder include an extra **cross-attention** layer

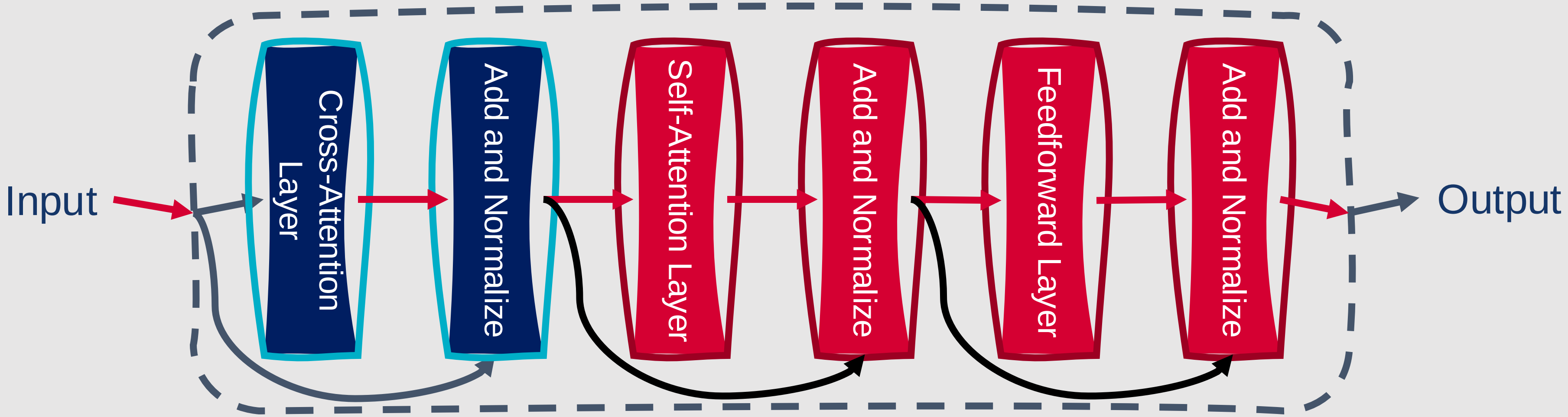
Cross-Attention



- Same form as **multiheaded self-attention** in a normal Transformer block, with one difference: queries come from the previous layer of the decoder as usual, but **keys and values come from the output of the encoder**
  - $\mathbf{Q} = \mathbf{W^Q H}^{dec[i-1]}$
  - $\mathbf{K} = \mathbf{W^Q H}^{enc}$
  - $\mathbf{V} = \mathbf{W^V H}^{enc}$

- $\mathrm{CrossAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V}$

Updated Decoder Transformer Block



Encoder-Decoder Models with Transformers

- Why is cross-attention useful?
  - Allows the decoder to attend to the entire encoder sequence
- Training Transformer-based encoder-decoders is similar to training RNN-based encoder-decoders
  - Use teacher forcing
  - Train autoregressively

Bidirectional Encoder Representations from Transformers (BERT)

- Popular method for building pretrained language models
- Many variations
  - DistilBERT
  - RoBERTa
  - SpanBERT
- Makes use of a **bidirectional Transformer encoder**

Prior to BERT:

- Statistical n-gram language models
- Feature-based classifiers
- Task-specific neural architectures

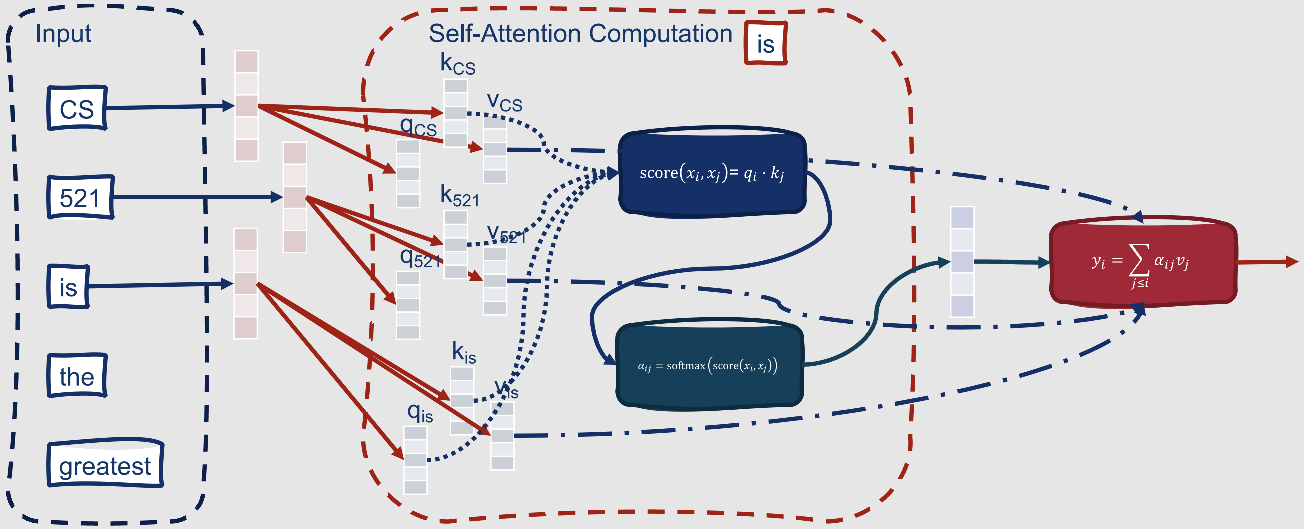
After BERT:

- Pretrained neural language models
- Task-specific fine-tuning

BERT was transformative to the NLP field!

Bidirectional Transformer Encoders

- We've already seen how causal Transformers work
  - Well-suited for language modeling problems since they prevent consideration of future context
- However, these models are inherently constrained
  - What about tasks for which future context is readily available?



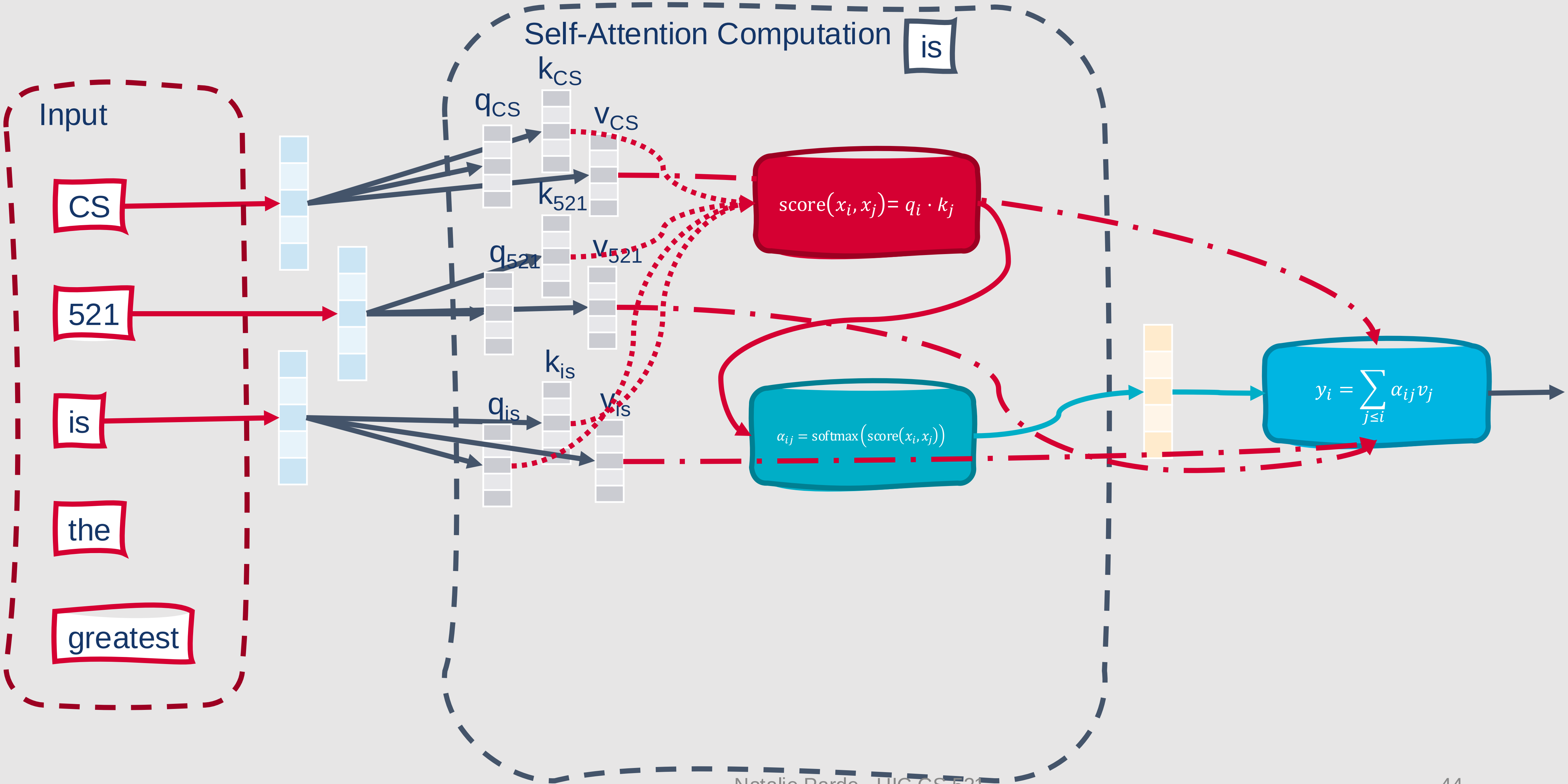
Many NLP tasks don’t need to restrict the model from viewing future context.

- Sequence classification
- (Sometimes) sequence labeling
- In general, most tasks that *aren’t* performed in real time

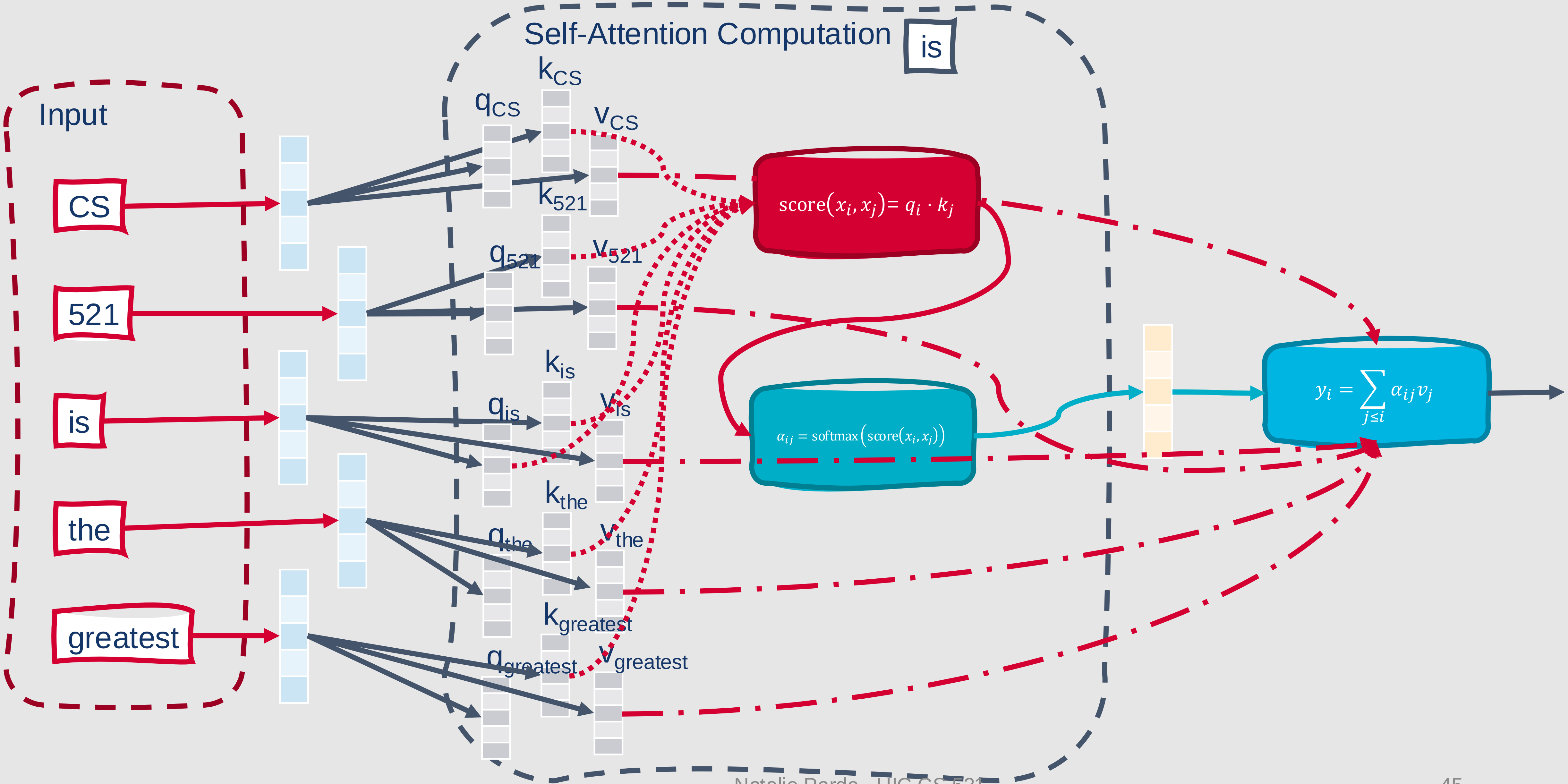
Transformers aren't innately constrained to processing from sequence beginning to end.

- With language modeling, self-attention computations are limited to current and prior context to avoid trivializing the problem
- Self-attention can be computed using the same equations we've already seen when allowing future context to be considered
- When that happens, the encoder produces sequences of output embeddings that are **contextualized** based on the entire input sequence

Bidirectional Self-Attention Layer



Bidirectional Self-Attention Layer



More formally….

- Step 1: Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
  - $\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i$
  - $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i$
  - $\mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i$

More formally….

- Step 2: Compute attention weights α by applying a softmax over the element-wise comparison scores between all possible query-key pairs in the full input sequence
  - $\text{score}_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$
  - $\alpha_{ij} = \frac{\exp(\text{score}_{ij})}{\sum_{k=1}^{n} \exp(\text{score}_{ik})}$

More formally….

- Step 3: Compute the output vector $\mathbf{h}_i$ as the attention-weighted sum of all of the input value vectors $\mathbf{v}$
  - $\mathbf{h}_{\boldsymbol{i}} = \sum_{j=1}^{n} \alpha_{ij} \mathbf{v}_j$

Visually….

$\mathbf{QK^T}$ matrix for a causal Transformer encoder

$q_1 \cdot k_1$	$q_1 \cdot k_2$	$q_1 \cdot k_3$	$q_1 \cdot k_4$	$q_1 \cdot k_5$
$q_2 \cdot k_1$	$q_2 \cdot k_2$	$q_2 \cdot k_3$	$q_2 \cdot k_4$	$q_2 \cdot k_5$
$q_3 \cdot k_1$	$q_3 \cdot k_2$	$q_3 \cdot k_3$	$q_3 \cdot k_4$	$q_3 \cdot k_5$
$q_4 \cdot k_1$	$q_4 \cdot k_2$	$q_4 \cdot k_3$	$q_4 \cdot k_4$	$q_4 \cdot k_5$
$q_5 \cdot k_1$	$q_5 \cdot k_2$	$q_5 \cdot k_3$	$q_5 \cdot k_4$	$q_5 \cdot k_5$

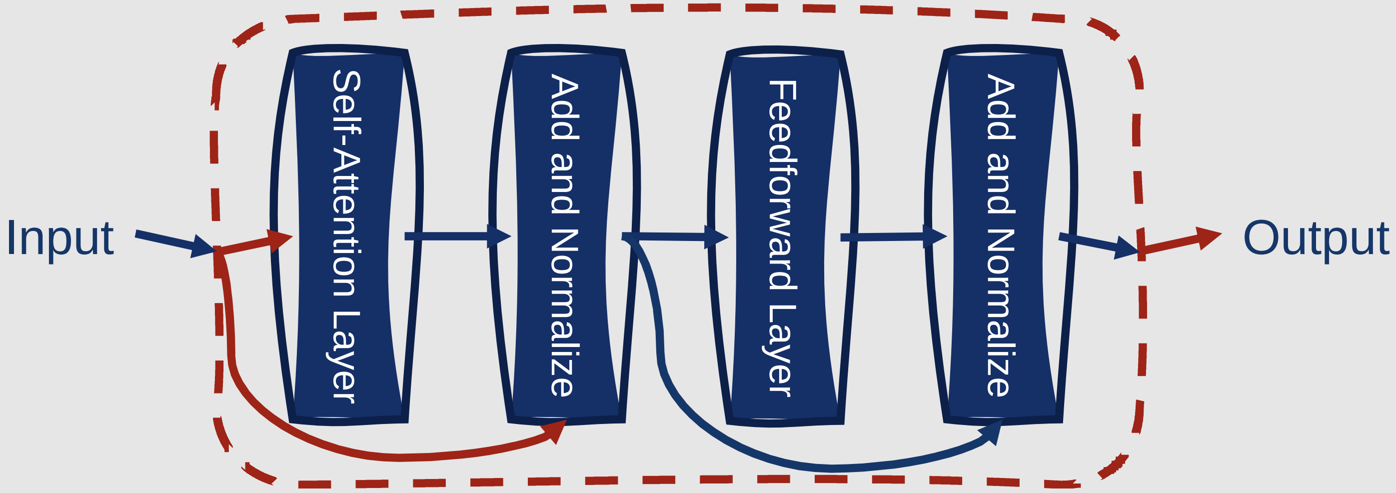
Visually….

$\mathbf{QK^T}$ matrix for a bidirectional Transformer encoder

$q_1 \cdot k_1$	$q_1 \cdot k_2$	$q_1 \cdot k_3$	$q_1 \cdot k_4$	$q_1 \cdot k_5$
$q_2 \cdot k_1$	$q_2 \cdot k_2$	$q_2 \cdot k_3$	$q_2 \cdot k_4$	$q_2 \cdot k_5$
$q_3 \cdot k_1$	$q_3 \cdot k_2$	$q_3 \cdot k_3$	$q_3 \cdot k_4$	$q_3 \cdot k_5$
$q_4 \cdot k_1$	$q_4 \cdot k_2$	$q_4 \cdot k_3$	$q_4 \cdot k_4$	$q_4 \cdot k_5$
$q_5 \cdot k_1$	$q_5 \cdot k_2$	$q_5 \cdot k_3$	$q_5 \cdot k_4$	$q_5 \cdot k_5$

Bidirectional Transformer Encoders

- All other elements remain the same as seen in causal Transformers!
  - Inputs are segmented using subword tokenization
  - Inputs are combined with positional embeddings
  - Transformer blocks include a self-attention layer and a feedforward layer, augmented with normalization layers and residual connections



BERT-Specific Architectural Details

Subword vocabulary of 30k tokens generated using the WordPiece algorithm

768-dimensional hidden layers

12 Transformer blocks

12 attention heads in each self-attention layer

In total, this comprises 100M trainable parameters!

Training a WordPiece Tokenizer

- Start with special tokens and an initial alphabet
- Split text in the training corpus at the character level, adding a prefix to all characters *inside* the word
  - language → l ##a ##n ##g ##u ##a ##g ##e
- Then:
  - Compute scores for each adjacent pair of tokens t_1 and t_2
    - $\text{score}(t_1, t_2) = \frac{\text{freq}(t_1 t_2)}{\text{freq}(t_1) \times \text{freq}(t_2)}$
  - Merge the highest-scoring pair of tokens and add the merged token to the vocabulary
  - Repeat until the desired vocabulary size is reached

WordPiece Tokenization

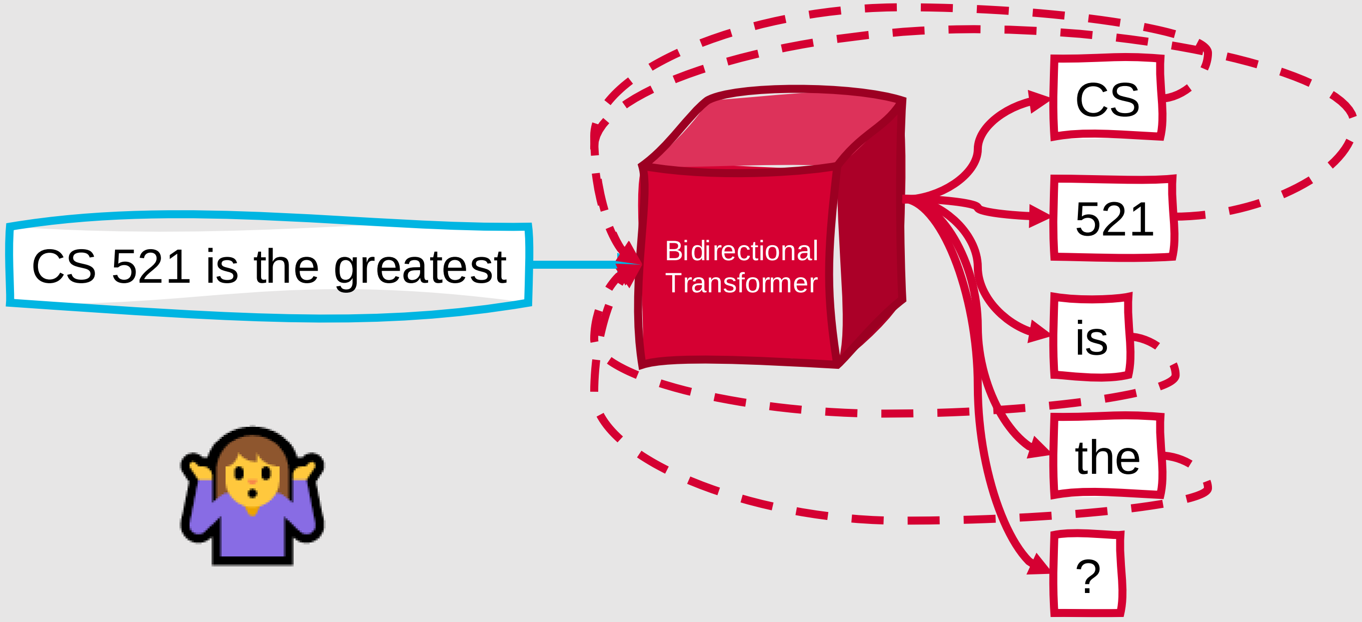
- Starting at the beginning of the text to tokenize, find the longest matching subword in the vocabulary
- Split on this subword
- Move forward to the first position after the split
- Repeat
  - If there are no matching subwords in the vocabulary, tokenize the text as [UNK]

Additional BERT Details

- Since subword tokenization is used, for some NLP tasks (e.g., named entity tagging) it is necessary to map subwords back to words
- BERT is costly to train (time and memory requirements grow quadratically with input length)
  - To increase efficiency, a fixed input length of 512 subword tokens is used---when working with longer texts, it's necessary to partition the text into different segments

Training Bidirectional Encoders

- With causal Transformer encoders, we employed autoregressive language modeling (next word prediction) as the training task
- With bidirectional Transformer encoders, this task becomes trivial …the answer is now directly available from the context!



A new task is needed for training bidirectional encoders….

After such a late _____ working on my project, it was ____ to wake up this morning!

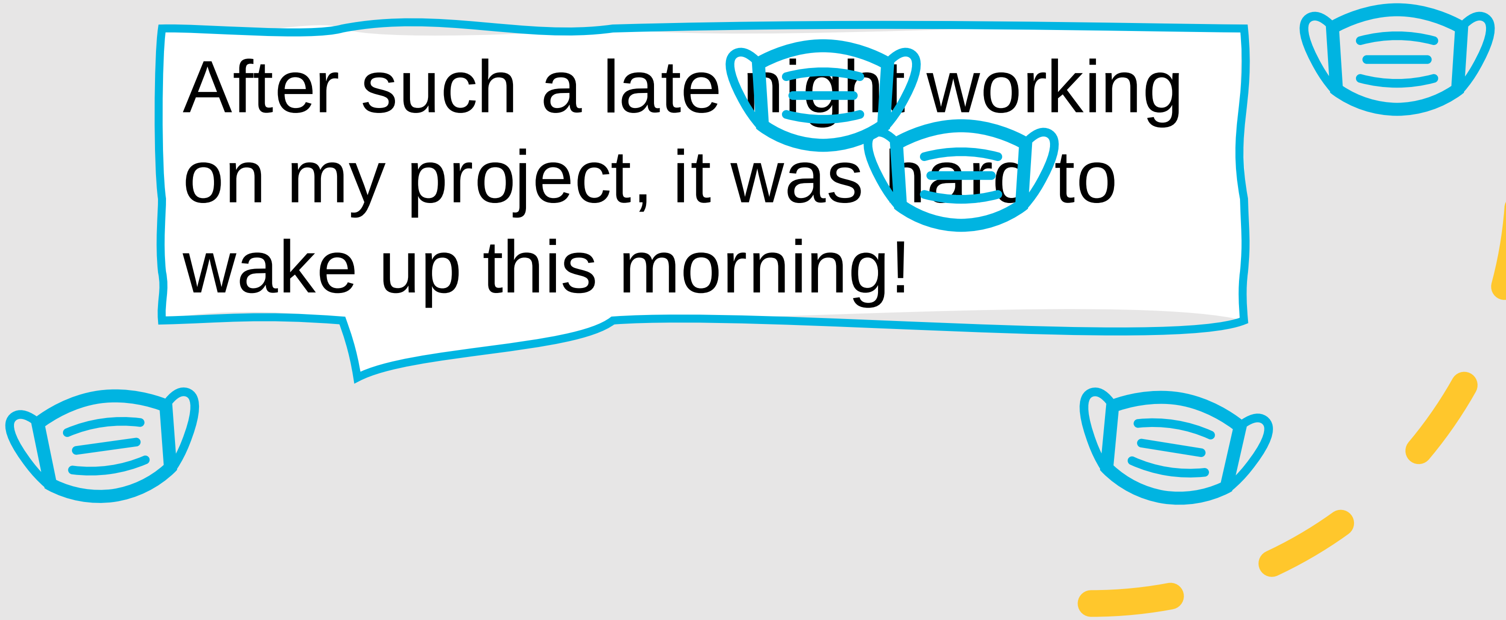
- **Cloze Task:** Instead of trying to predict the next word, learn how to predict the best word to fill in the blank
- How do we do this?
  - During training, **mask out** one or more elements from the input sequence
  - Generate a **probability distribution** over the vocabulary for each of the missing elements
  - Use the **cross-entropy loss** from these probabilities to drive the learning process

Cloze Task

- This task can be generalized to any method that:
    1. Corrupts the training input
    2. Asks the model to recover the original training input
- What are some ways to corrupt the training input?
    - Masks
    - Substitutions
    - Reorderings
    - Deletions
    - Extraneous insertions into the training text

Masking Words

- Original approach for corrupting input when training bidirectional Transformer encoders
- BERT uses a masking technique known as **masked language modeling** (MLM)



Masked Language Modeling

- Uses unannotated text from a large corpus
- Presents the models with sentences from the corpus
- For each sentence, a random sample of tokens is selected to be used in one of the following ways:
  - The token is replaced with a [MASK] token
  - The token is replaced with another randomly sampled token
  - The token is left unchanged

What is the intuition behind these corruptions?

- **[MASK] token:** The model learns to predict the masked words using *only* the available context ([MASK] isn't even in the training vocabulary!)
- **Random token:** The model learns to favor contextual cues more heavily than the word itself when encoding meaning
- **Same token:** The model learns to rely at least a little bit on the specific word in its specific contextual position

Masked Language Modeling

After such a late night working on my project, it was hard to wake up this morning!

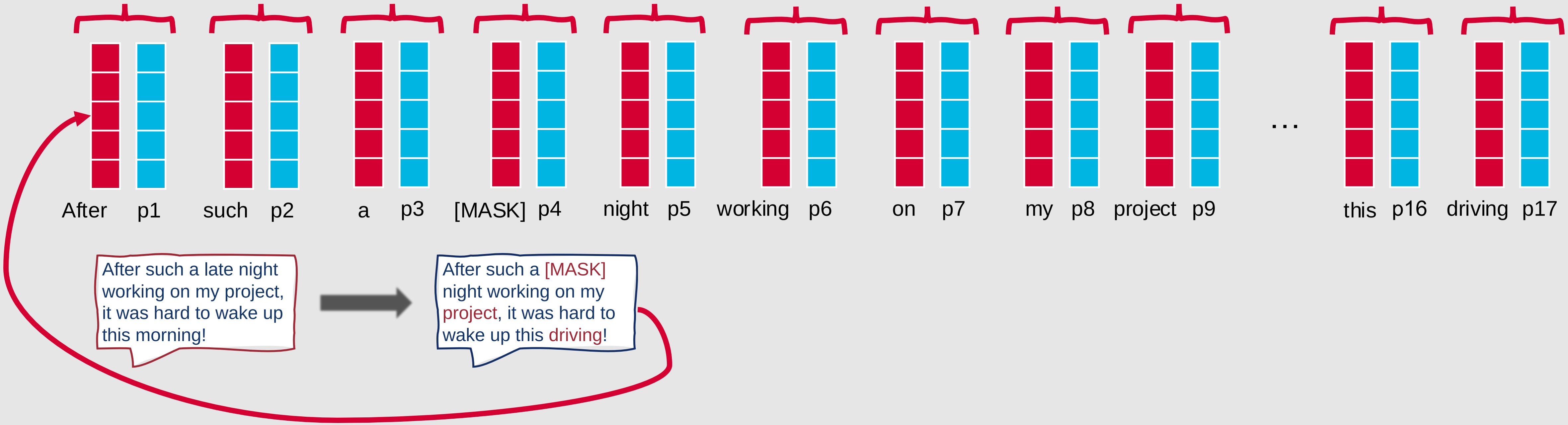
After such a [MASK] night working on my project, it was hard to wake up this driving!

Masked Language Modeling

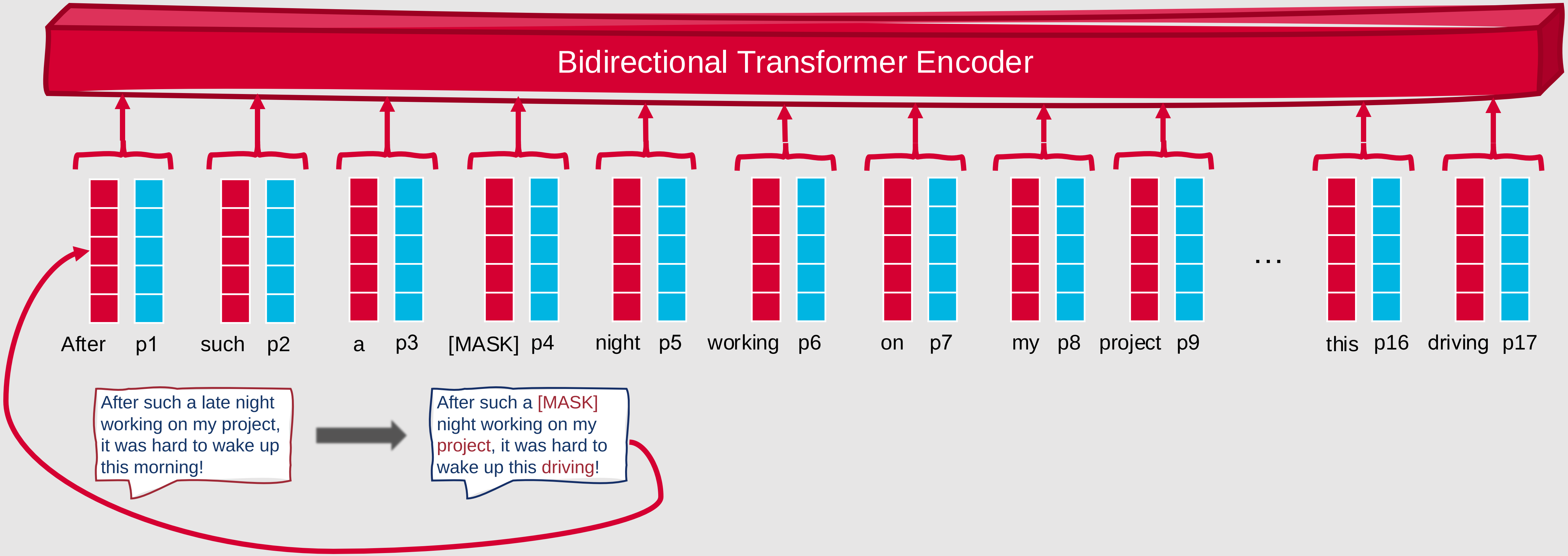
After such a late night working on my project, it was hard to wake up this morning!

After such a [MASK] night working on my project, it was hard to wake up this driving!

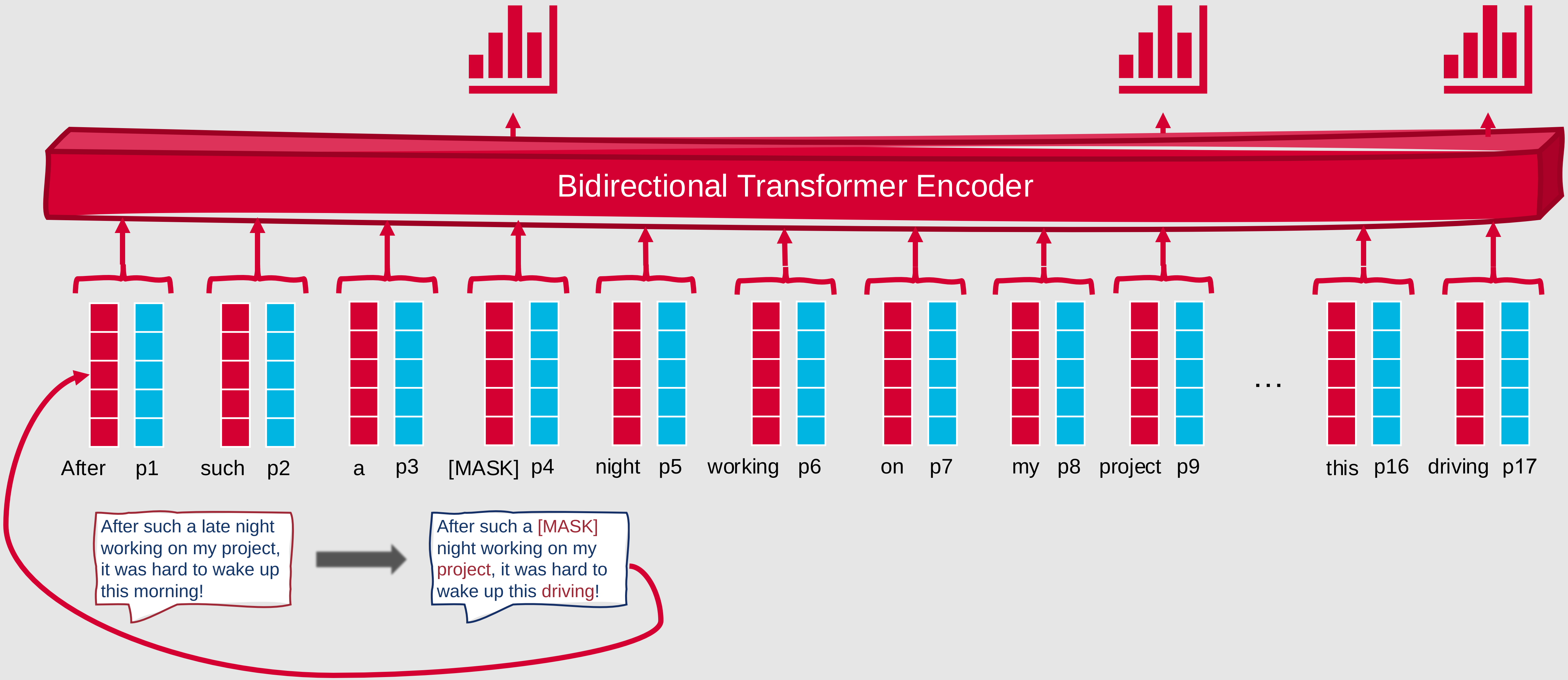
Masked Language Modeling



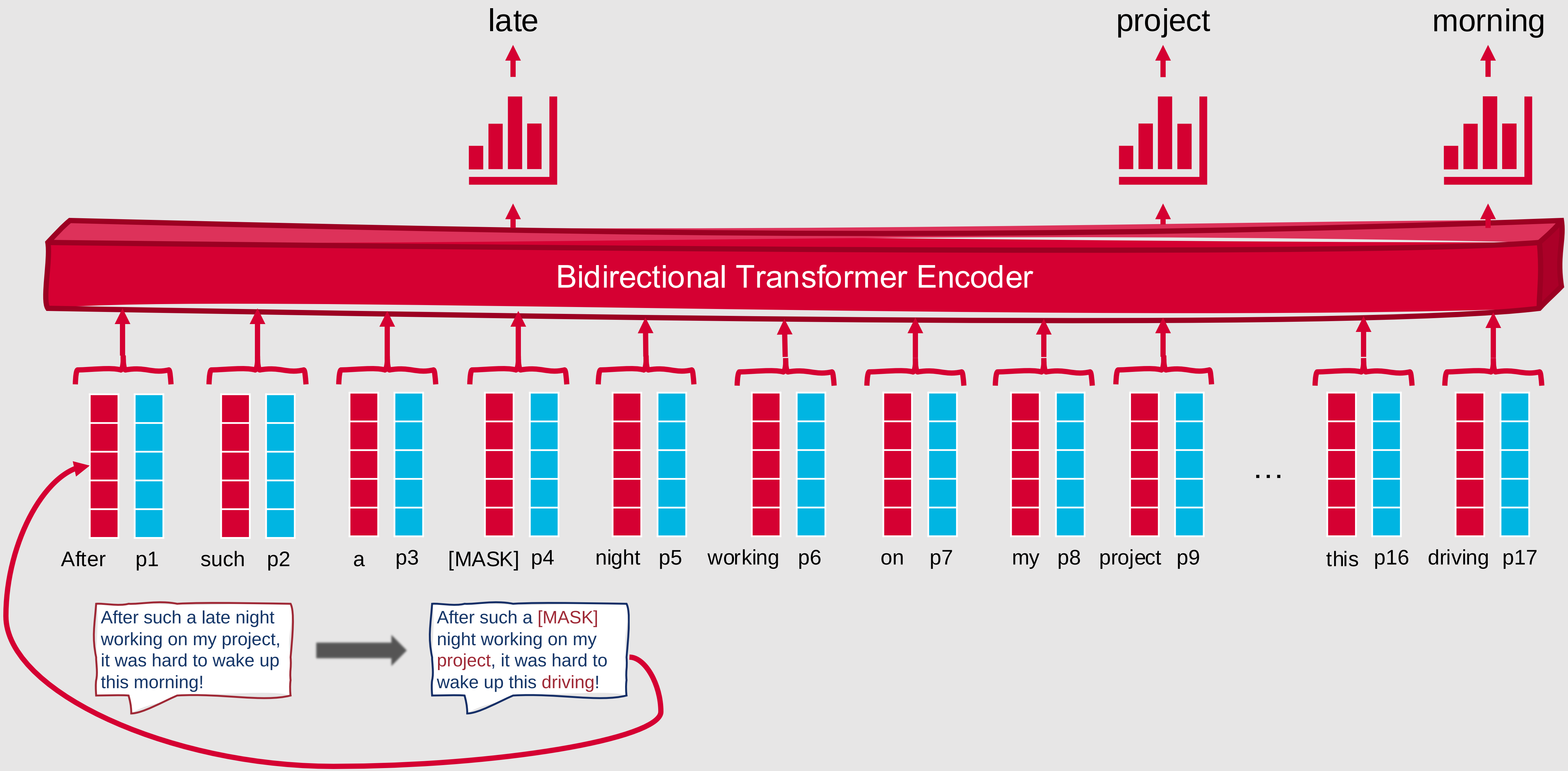
Masked Language Modeling



Masked Language Modeling



Masked Language Modeling



Masked Language Modeling

- Training objective:
  - Predict the original inputs for each of the sampled tokens using a bidirectional encoder
  - Make better predictions with each iteration based on cross-entropy loss
  - Gradients that form the basis for weight updates are based on average loss over the sampled learning tokens
- Although all tokens play a role in the self-attention layer, **only the sampled tokens are used for learning**

Masked Language Modeling in BERT

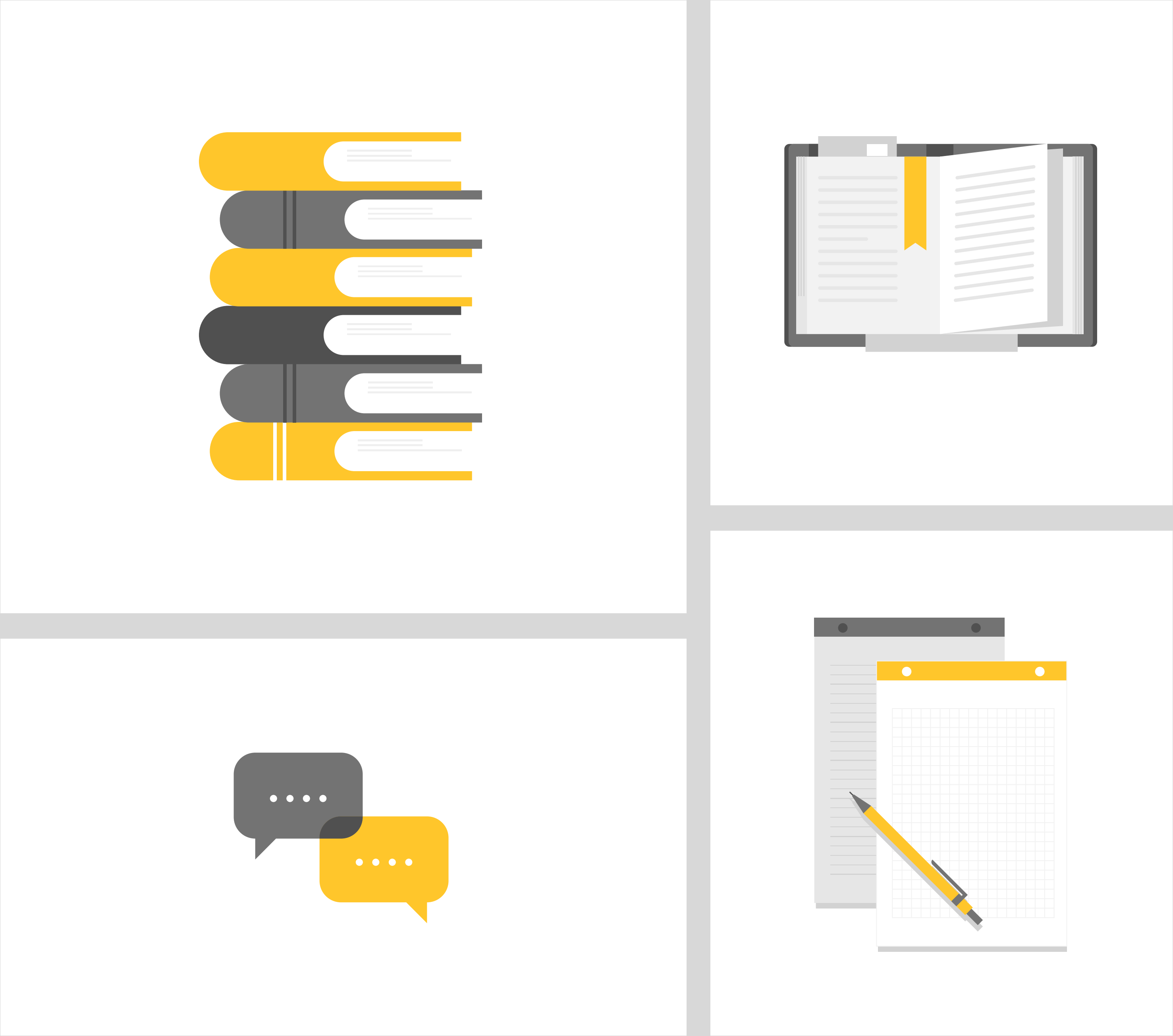
- Same process as shown, but uses subword tokens instead
- 15% of tokens in the training sequence are sampled
- Of these:
  - 80% are replaced with [MASK]
  - 10% are replaced with randomly selected tokens
  - 10% are left unchanged

Summary: Transformers and Masked Language Modeling

- **Contextual word embeddings** are typically generated using pretrained language models
- A popular sequence processing architecture for training modern language models is the **Transformer**
- **Bidirectional Transformer encoders** were used to create BERT, a transformative pretrained language model
- **Masked language modeling** is a learning objective for bidirectional Transformer encoders that forces the model to predict potentially masked or otherwise corrupted words, based on the surrounding context

What if the most useful language segment for our task isn't a single token?

- Lots of tasks have larger units of interest:
  - Question answering
  - Syntactic parsing
  - Coreference resolution
  - Semantic role labeling
- Solution: Apply a **span-oriented** masked learning objective



Masking Spans

- **Span:** A contiguous sequence of one or more words selected from a training sample, prior to subword tokenization
- How can we select spans for masking?
  1. Decide on a span length
     - In SpanBERT, this is sampled from a geometric distribution biased toward shorter spans, with an upper bound of 10
  2. Given this span length, sample a starting location

Masking Spans

- All sampling actions are performed at the span level
  - All tokens in the selected span are replaced with [MASK]
  - All tokens in the selected span are replaced with randomly sampled tokens
  - All tokens in the selected span are left as is
- After sampling actions are performed, the input is passed through the same Transformer architecture seen previously

$$\mathcal{L}(\text{football}) = \mathcal{L}_{\text{MLM}}(\text{football}) + \mathcal{L}_{\text{SBO}}(\text{football})$$
$$= -\log P(\text{football} \mid \mathbf{x}_7) - \log P(\text{football} \mid \mathbf{x}_4, \mathbf{x}_9, \mathbf{p}_3)$$

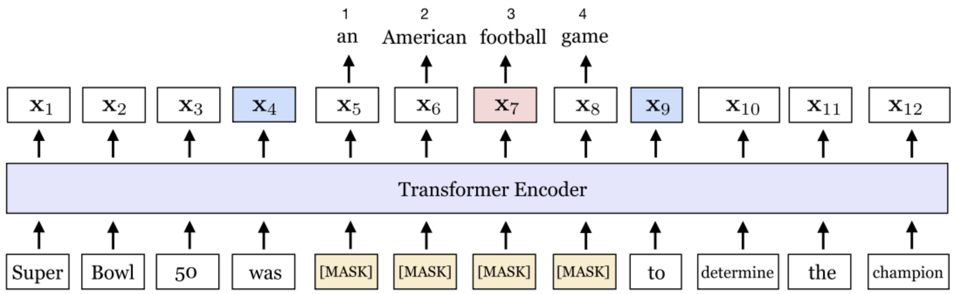


Figure 1: An illustration of SpanBERT training. The span *an American football game* is masked. The SBO uses the output representations of the boundary tokens, $\mathbf{x}_4$ and $\mathbf{x}_9$ (in blue), to predict each token in the masked span. The equation shows the MLM and SBO loss terms for predicting the token, *football* (in pink), which as marked by the position embedding $\mathbf{p}_3$, is the *third* token from x_4.

Masked Language Modeling in SpanBERT

- Analogous to “standard” BERT:
  - In 80% of spans, tokens are replaced with [MASK]
  - In 10% of spans, tokens are replaced with randomly sampled tokens
  - In 10% of spans, tokens are left unchanged
- Total token substitution is limited to 15% of the input

Masking Spans

After such a late night working on my project, it was hard to wake up this morning!

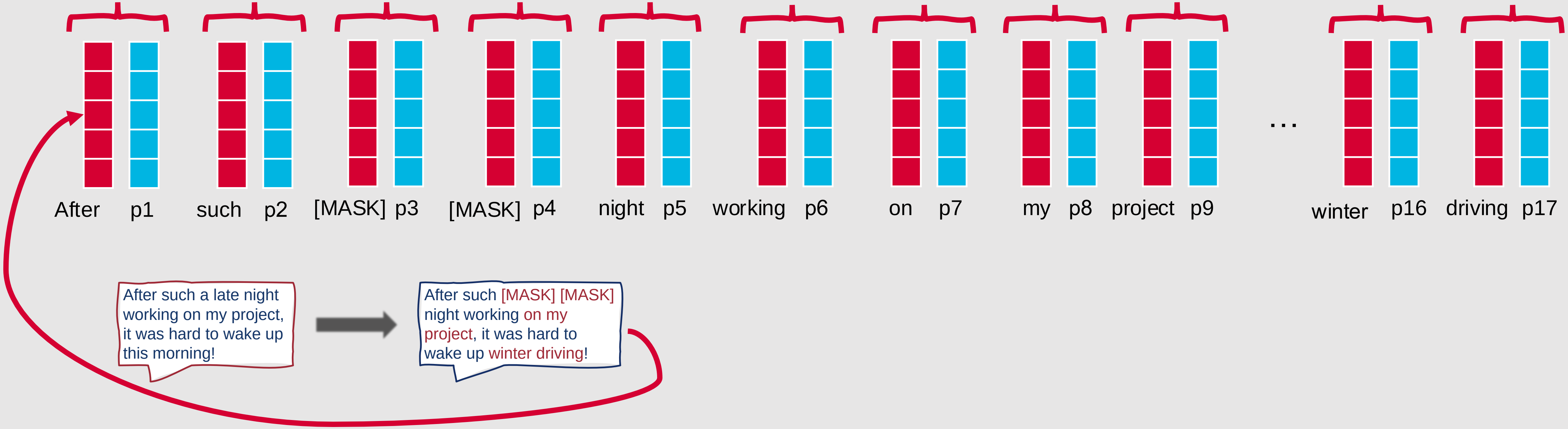
After such [MASK] [MASK] night working on my project, it was hard to wake up winter driving!

Span-Based Masked Language Modeling

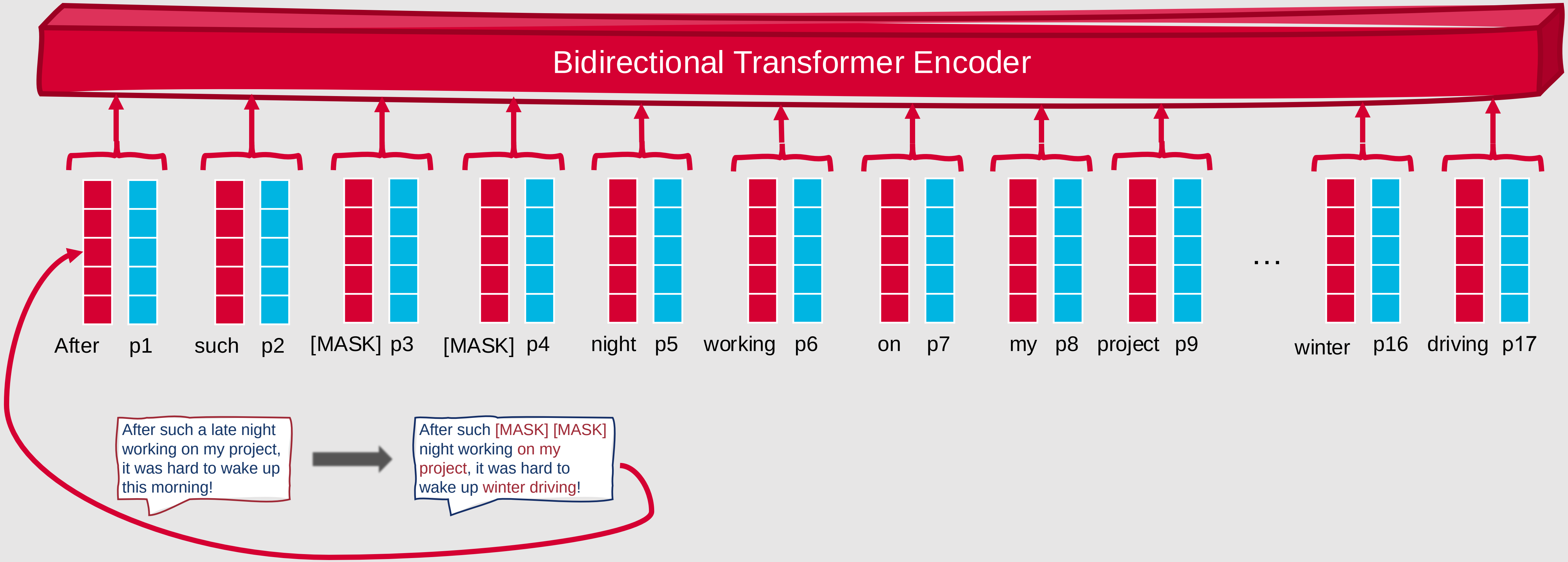
After such a late night working on my project, it was hard to wake up this morning!

After such [MASK] [MASK] night working on my project, it was hard to wake up winter driving!

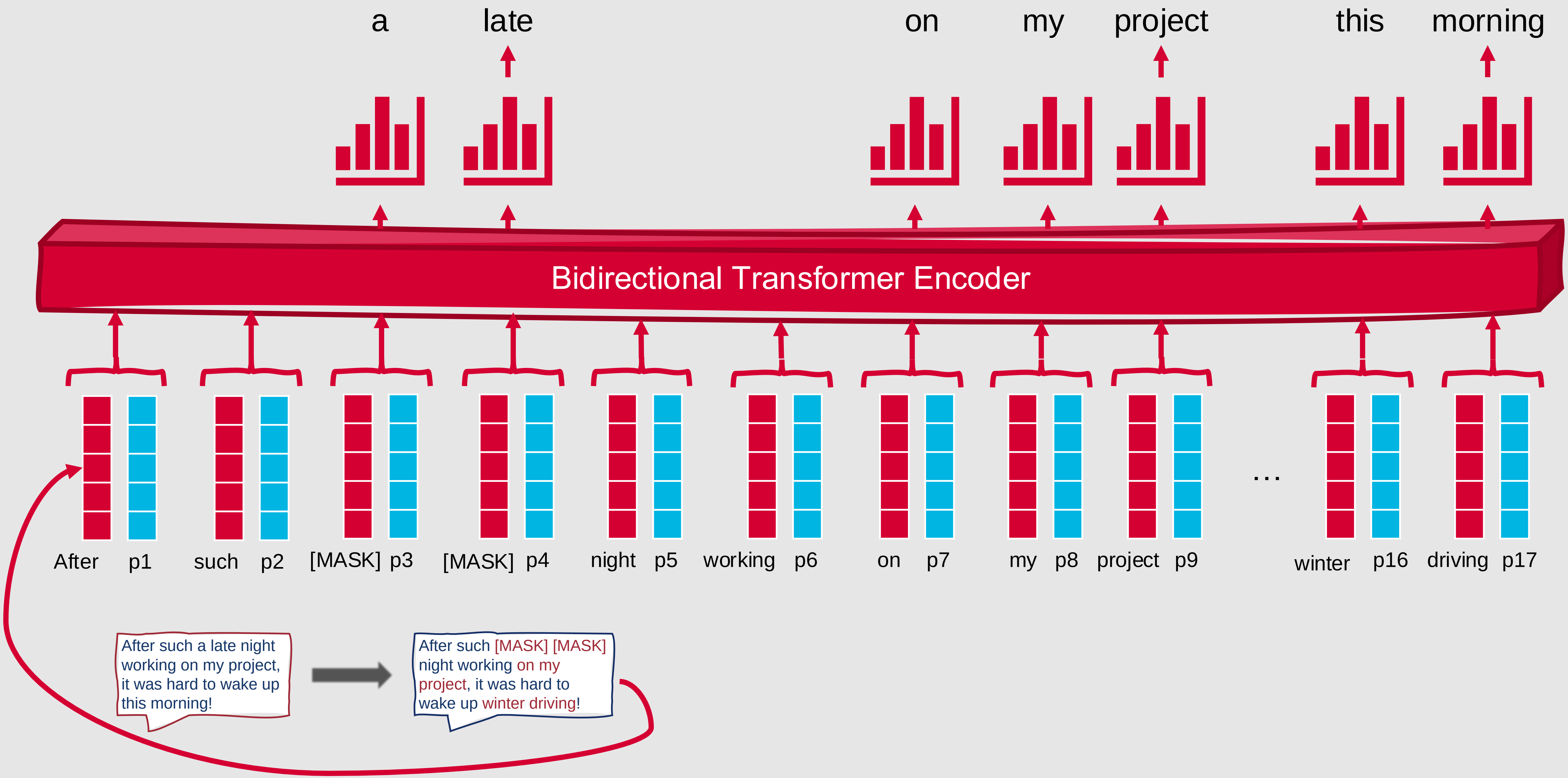
Span-Based Masked Language Modeling



Span-Based Masked Language Modeling



Span-Based Masked Language Modeling



What kind of information should be included in a span-level representation?

- Create span-level representations based on:
  - Tokens within the span
  - Span boundaries
- Boundary representations are usually derived from:
  - First and last words of the span
  - Words immediately before or after the span

Span Boundary Objective

- Augments the masked language modeling objective in SpanBERT, altering the loss function to account for the span boundary objective
  - $L(\mathbf{x}) = L_{MLM}(\mathbf{x}) + L_{SBO}(\mathbf{x})$
- Leverages the model's ability to predict words inside a span based on those just outside of it
  - $L_{SBO}(\mathbf{x}) = -\log P(\mathbf{x} | \mathbf{x}_{s-1}, \mathbf{x}_{e+1}, \mathbf{p}_{i-s+1})$

Word before the span

Word after the span

Positional embedding indicating which word in the span is being predicted

Bidirectional Transformer encoders can also help us learn another important piece of information!

- In many NLP tasks, it is crucial to learn the **relationship between pairs of sentences**
  - Detecting paraphrases
  - Determining entailment
  - Measuring discourse coherence

BERT also uses a *second* learning objective that helps us perform this task.

- What is this other learning objective?
  - **Next sentence prediction** (NSP)

Next Sentence Prediction

- Present the model with pairs of sentences
- Predict whether each pair is an *actual* pair of adjacent sentences, or a pair of unrelated sentences
  - In BERT, training pairs are evenly balanced across these two classes
- Base the loss on how well the model can distinguish actual pairs from unrelated pairs

After such a late night working on my project, it was hard to wake up this morning! I did though, because I had to give my project presentation.

After such a late night working on my project, it was hard to wake up this morning! A winter storm warning has been issued for your area.

How does NSP training work?

- Two new tokens are added to the input:
  - [CLS] is prepended to the input sentence pair
  - [SEP] is placed *between* the sentences and *after* the final token of the second sentence
- Embeddings representing each segment (first sentence and second sentence) are added to the word and positional embeddings

Additional Tokens

After such a late night working on my project, it was hard to wake up this morning! I did though, because I had to give my project presentation.

[CLS] After such a late night working on my project, it was hard to wake up this morning! [SEP] I did though, because I had to give my project presentation. [SEP]

Once we've made these adjustments….

```
    model = modeling.BertModel(
        config=bert_config,
        is_training=is_training,
        input_ids=input_ids,
        input_mask=input_mask,
        token_type_ids=segment_ids,
        use_one_hot_embeddings=use_one_hot_embeddings)

    (masked_lm_loss,
     masked_lm_example_loss, masked_lm_log_probs) = get_masked_lm_output(
         bert_config, model.get_sequence_output(), model.get_embedding_table(),
         masked_lm_positions, masked_lm_ids, masked_lm_weights)

    (next_sentence_loss, next_sentence_example_loss,
     next_sentence_log_probs) = get_next_sentence_output(
         bert_config, model.get_pooled_output(), next_sentence_labels)

    total_loss = masked_lm_loss + next_sentence_loss
```

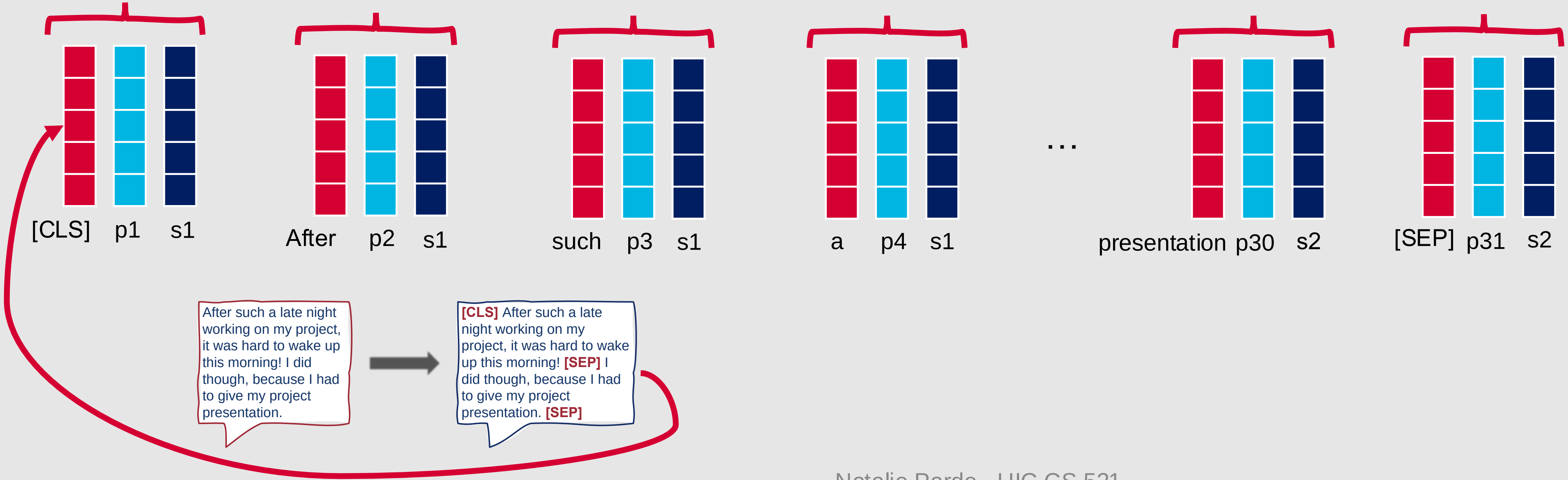
- The output vector associated with the [CLS] token represents the next sentence prediction
- Specifically, a learned set of classification weights $\mathbf{W}_{\mathbf{NSP}} \in \mathbb{R}^{2 \times d_h}$ is used to predict one of two classes from the raw [CLS] vector $\mathbf{h}_i$
    - $y_i = \text{softmax}(\mathbf{W}_{\mathbf{NSP}}\mathbf{h}_i)$
- A cross-entropy loss is used for the NSP loss
- In BERT, the final loss function is a linear combination of the NSP and MLM loss functions

Next Sentence Prediction

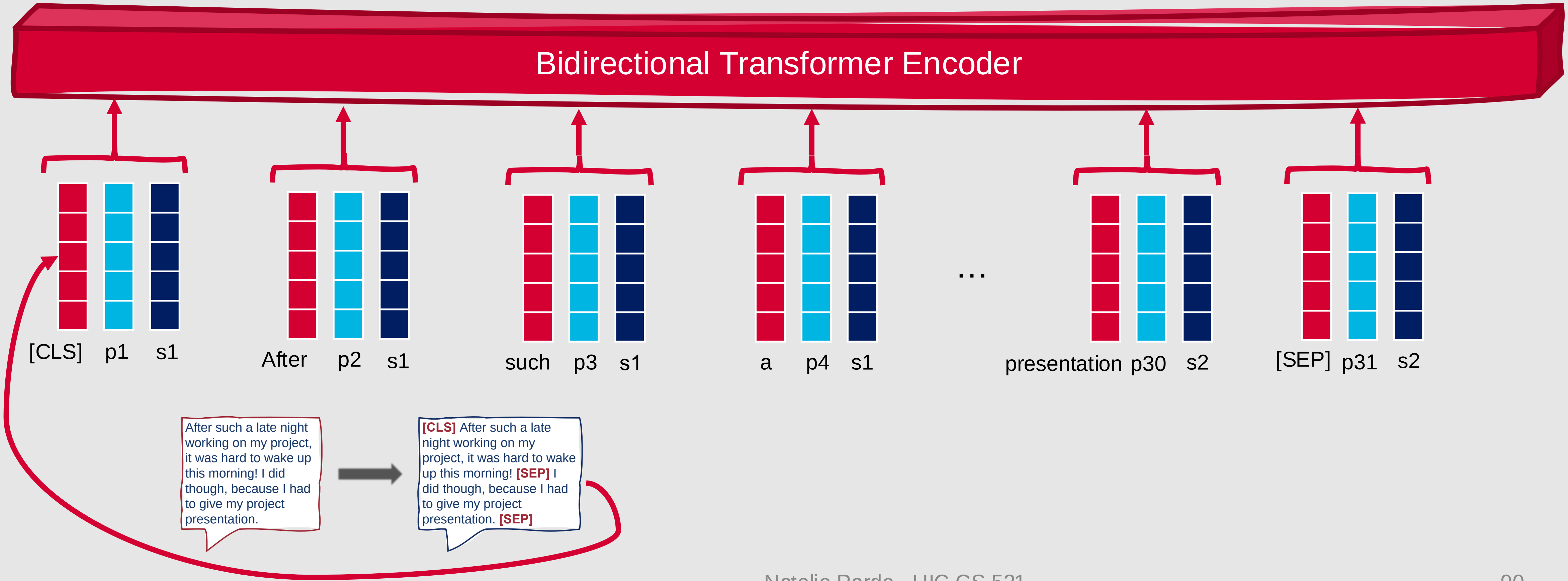
After such a late night working on my project, it was hard to wake up this morning! I did though, because I had to give my project presentation.

[CLS] After such a late night working on my project, it was hard to wake up this morning! **[SEP]** I did though, because I had to give my project presentation. **[SEP]**

Next Sentence Prediction

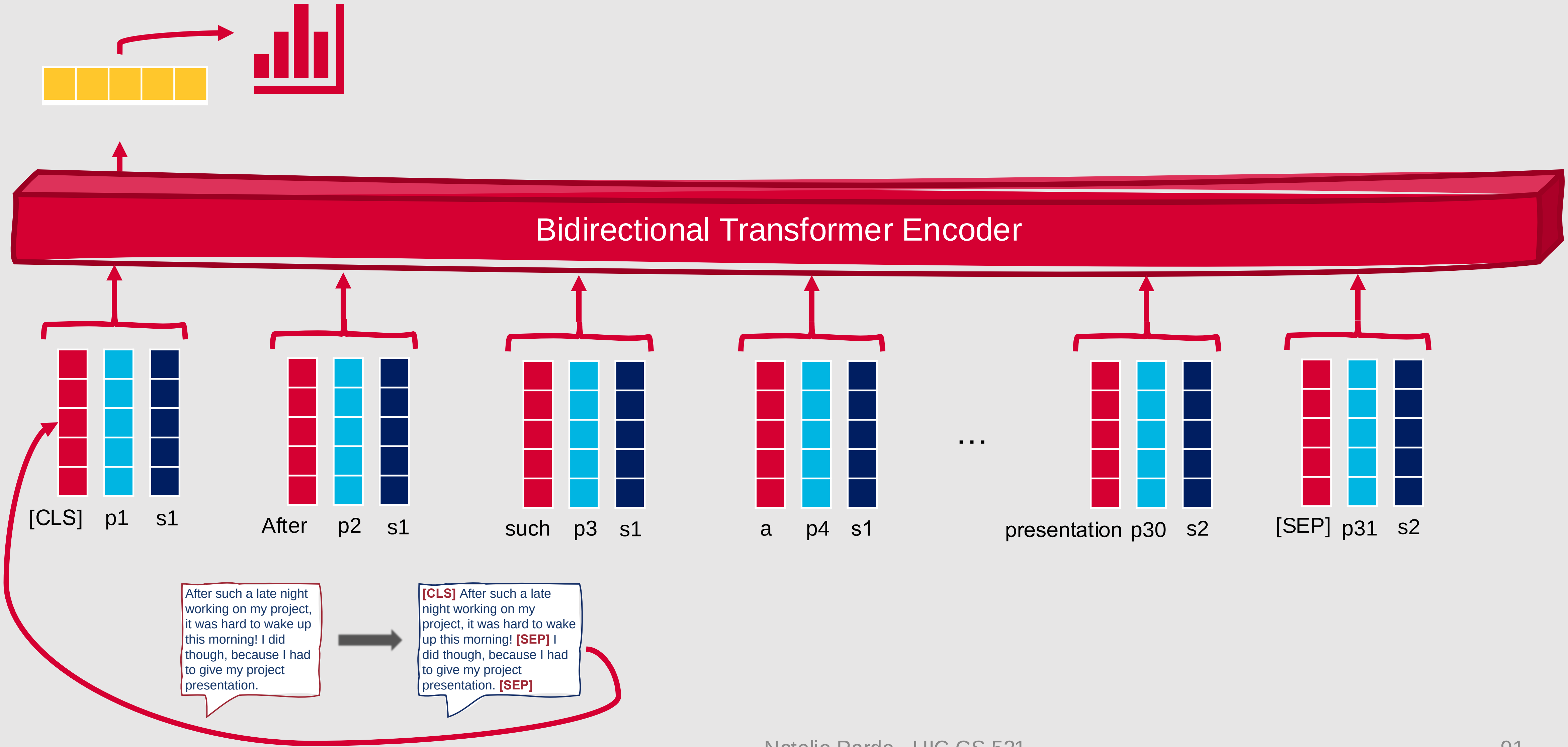


Next Sentence Prediction



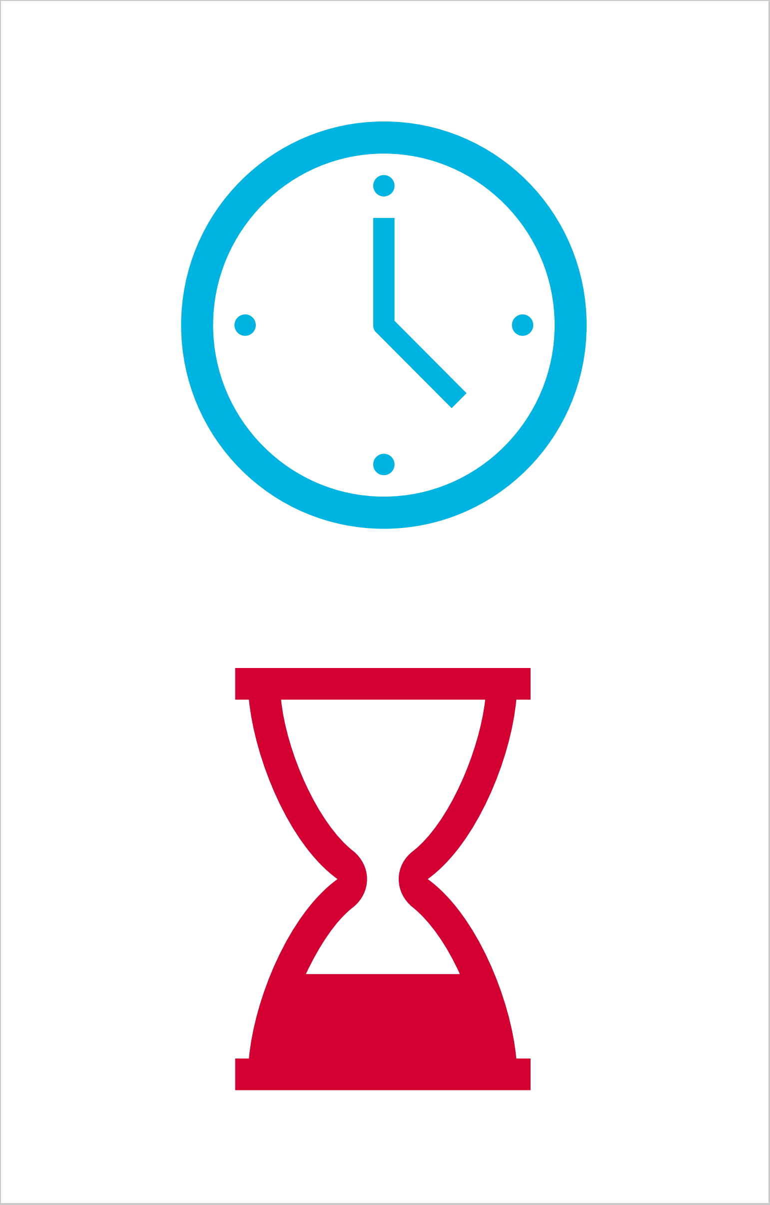
Next Sentence Prediction

Actually Adjacent



BERT-Specific Training Details

- Corpora:
  - Early Transformer-based language models (including BERT) used BooksCorpus (800M words) and English Wikipedia (2.5B words)
  - More recent state-of-the-art models learn from even larger corpora
- When training BERT, pairs of sentences were sampled such that their maximum combined length does not exceed 512 tokens
- Original BERT models converged after approximately 40 training iterations



Training models like BERT can be expensive and time-consuming….

- However, this pretraining process can result in models that can be used and reused for numerous tasks
    - Pretrained word embeddings and learned parameters to produce new contextual embeddings
    - Base models that can be fine-tuned for transfer learning purposes

Transfer Learning through Fine-Tuning

- Pretrained language models facilitate **generalization** across large text corpora
- This generalization makes it easier to incorporate these models effectively in downstream applications
- The process of learning an interface between a pretrained language model and a specific downstream task is called **fine-tuning**

Fine-Tuning

- Facilitates the creation of downstream applications on top of pretrained language models through the addition of a small set of application-specific parameters
- Labeled data from the downstream task domain is used to train these application-specific parameters
- In general, the pretrained language model is **frozen** or only minimally adjusted during this process

Many different applications have made use of fine-tuning!

- Sequence classification
- Sequence labeling
- Sentence-pair inference
- Span-based operations

Sequence Classification

- Models often represent an input sequence with a single representation
- For example:
  - Final hidden layer of an RNN model
  - [CLS] vector in a bidirectional Transformer model (e.g., BERT)
- This representation is sometimes referred to as a **sentence or document embedding**
- This representation serves as input to a **classifier head** for the downstream task

How do we fine-tune for sequence classification tasks?

- Learn a set of weights, $\mathbf{W_C} \in \mathbb{R}^{n \times d_h}$, to map the sequence representation to a set of scores over n possible classes
  - d_h is the dimensionality of the language model's hidden layers
- Requires supervised training data for the target task
- Learning process that optimizes $\mathbf{W_C}$ is driven by cross-entropy loss between the softmax output and the target task label

How do we classify test documents for sequence classification tasks?

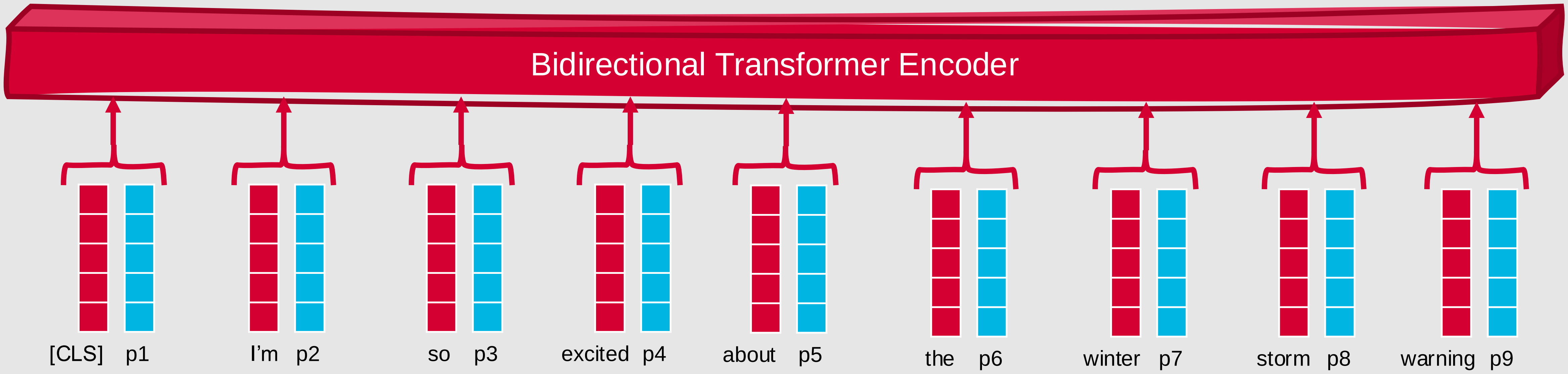
- Pass the input sample through the pretrained language model to generate an output representation $\mathbf{h}_{\mathbf{CLS}}$
- Multiply the output representation by the learned weights $\mathbf{W}_{\mathbf{C}}$
- Pass the resulting vector through a softmax:
  - $\mathbf{y} = \text{softmax}(\mathbf{W}_{\mathbf{C}}\mathbf{h}_{\mathbf{CLS}})$

Example: Sequence Classification

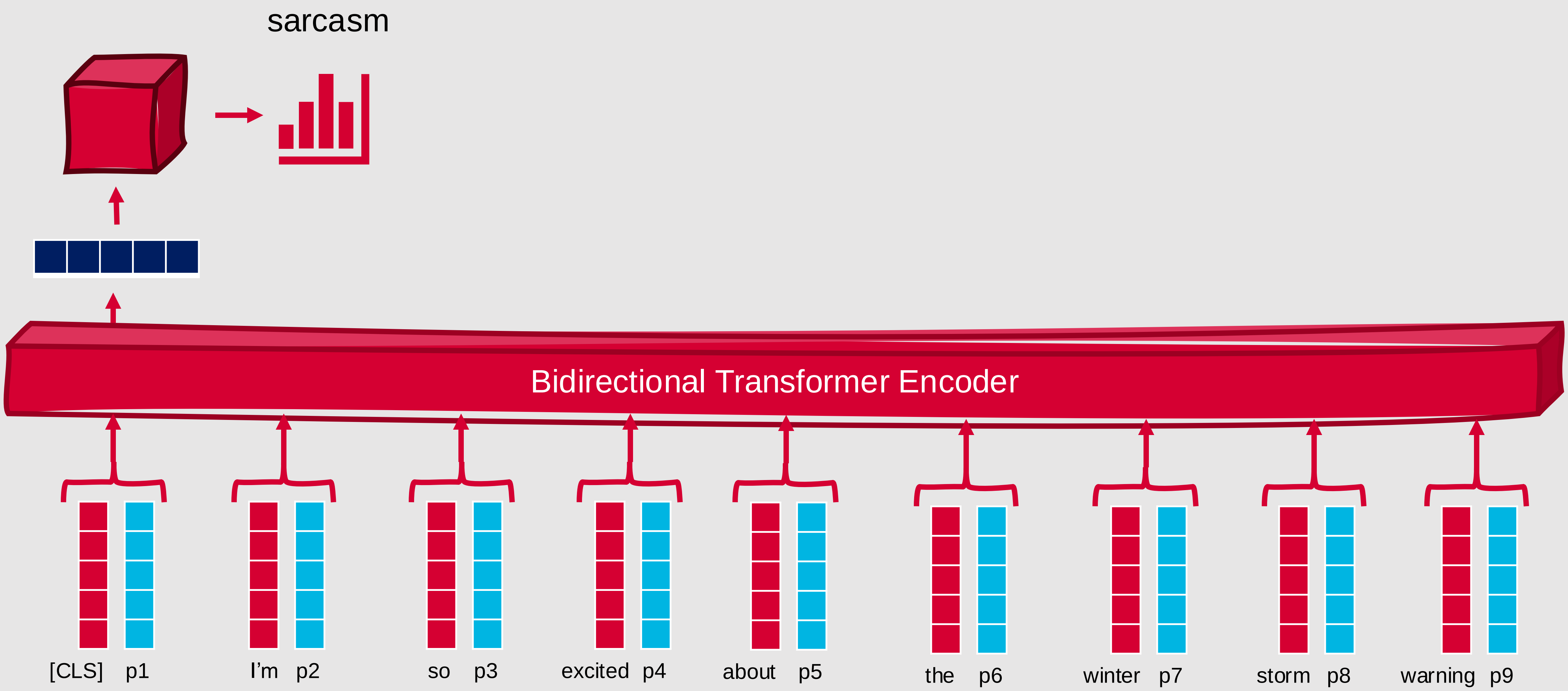
I’m so excited about the winter storm warning.

Example: Sequence Classification

Example: Sequence Classification



Example: Sequence Classification



What differs between this and earlier neural classifiers?

- If we want, we can use the computed loss to update not only the classifier weights, but also the weights for the pretrained language model itself
- However, substantial changes are rarely necessary!
  - Reasonable classification performance is often achieved with only minimal changes to the language model parameters
  - These changes are generally limited to updates over the final few layers of the model

Pair-Wise Sequence Classification

- Subcategory of sequence classification that focuses on classifying **pairs** of input sentences
- Useful for:
  - Logical entailment
  - Paraphrase detection
  - Discourse analysis

How does fine-tuning work for pair-wise sequence classification?

- Similar to pretraining with the **NSP objective**
  - Pairs of labeled sentences are presented to the model, separated by [SEP] and prepended with [CLS]
- During classification, the output [CLS] vector is multiplied by classification weights and passed through a softmax to generate label predictions

Example: Pair-Wise Sequence Classification (Entailment Task)

- Popular NLP task, also referred to as **natural language inference**
- Classify sentence pairs such that:
  - Sentence A **entails** Sentence B
  - Sentence A **contradicts** Sentence B
  - The relationship between Sentence A and Sentence B is **neutral**

Example: Pair-Wise Sequence Classification (Entailment Task)

It's a snow day! There is snow outside.

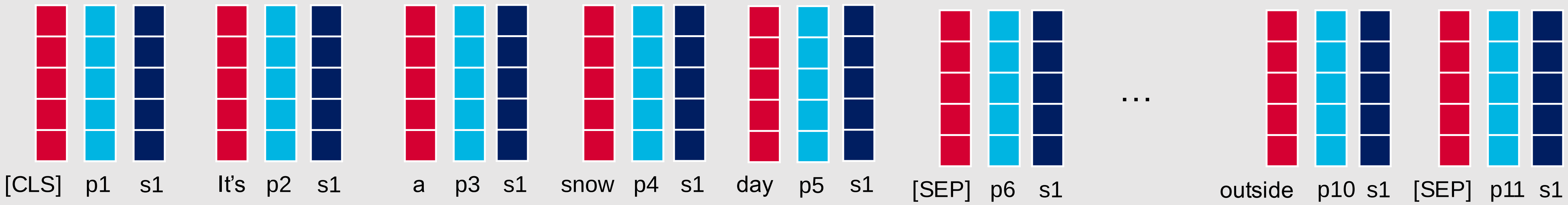
Example: Pair-Wise Sequence Classification (Entailment Task)

It's a snow day! There is snow outside.

→

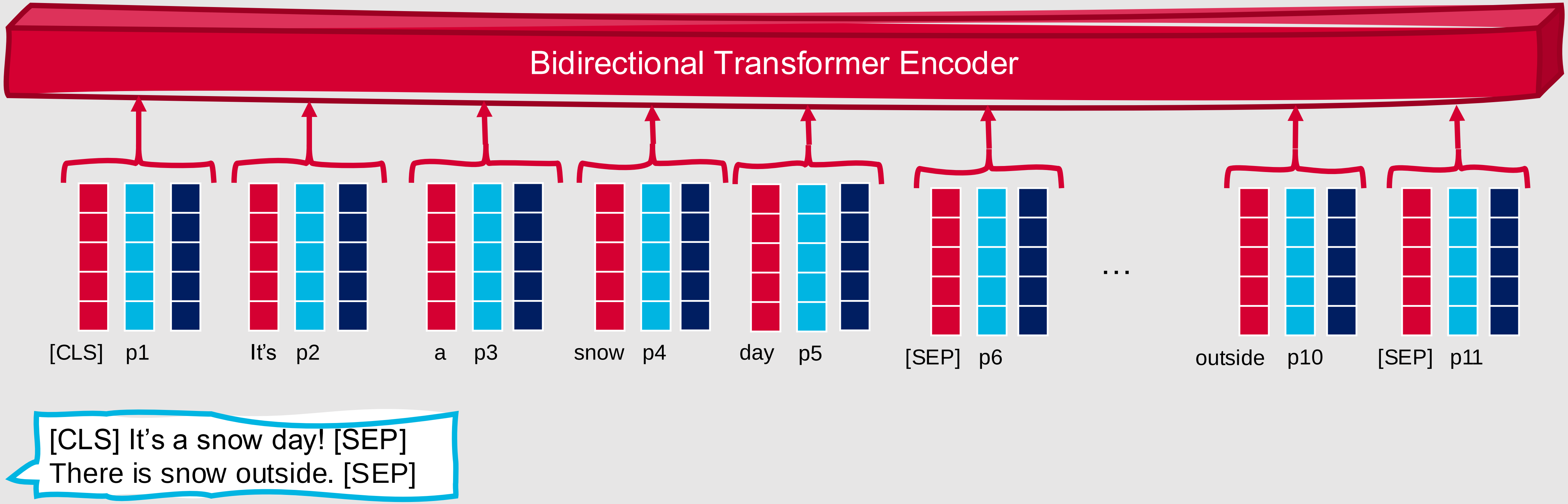
[CLS] It's a snow day! [SEP] There is snow outside. [SEP]

Example: Pair-Wise Sequence Classification (Entailment Task)

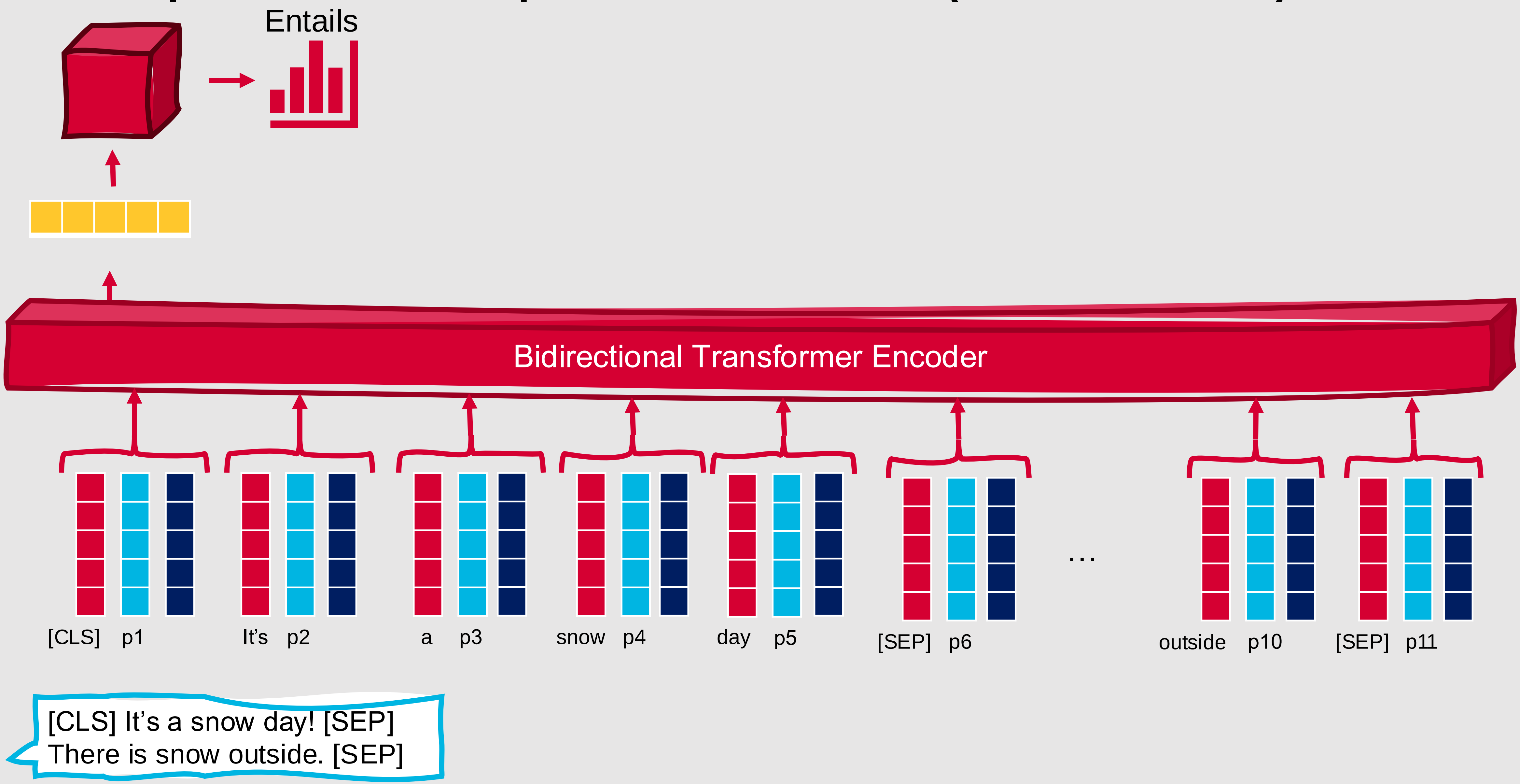


[CLS] It's a snow day! [SEP]
There is snow outside. [SEP]

Example: Pair-Wise Sequence Classification (Entailment Task)

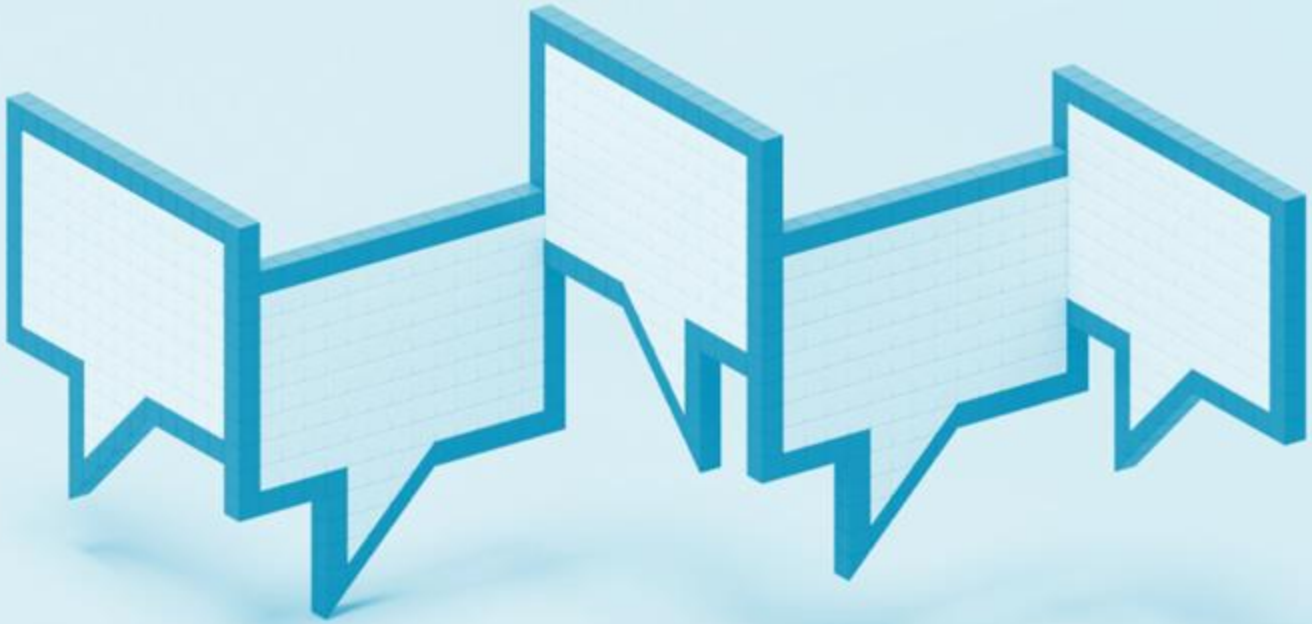


Example: Pair-Wise Sequence Classification (Entailment Task)



Sequence Labeling

- Similar to approach used for sequence classification
- However, the output vector for **each input token** is passed to a classification head that produces a softmax distribution over the possible classes
- The output tag sequence can be determined by a variety of methods
  - Common: **Greedy approach** accepting the argmax class for each token
    - $\mathbf{y}_i = \mathrm{softmax}(\mathbf{W}_K \mathbf{z}_i)$, where $k \in K$ is the set of tags for the task
    - $\mathbf{t}_i = \underset{k}{\mathrm{argmax}}(\mathbf{y}_i)$
  - Alternative: Distribution over labels can be passed to a **CRF layer**, allowing consideration of global tag-level transitions



Common Sequence Labeling Tasks

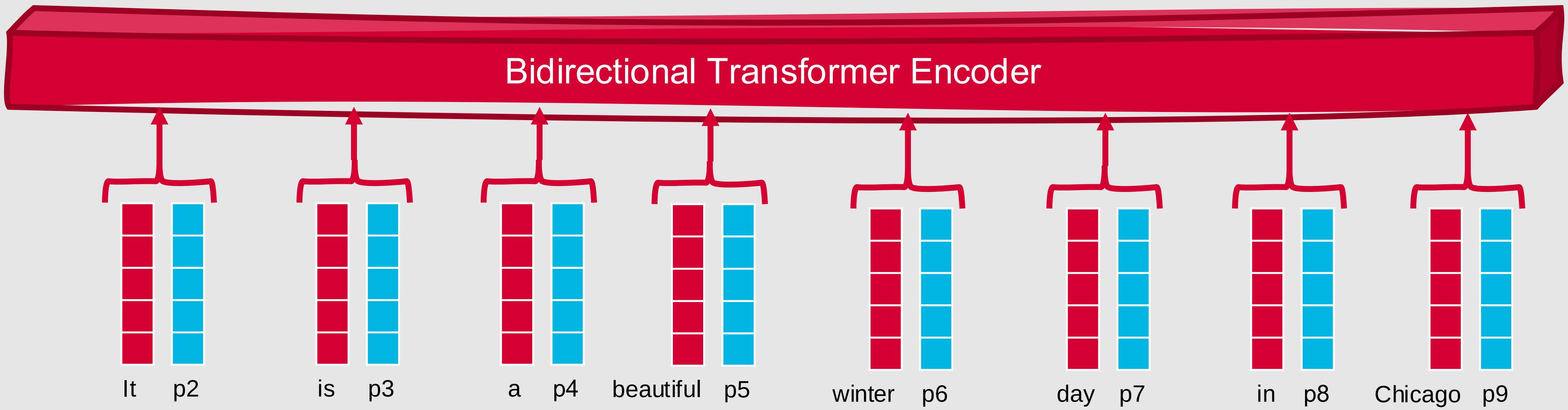
- Part-of-speech tagging
- Named entity recognition
- Shallow parsing

Example: Sequence Labeling

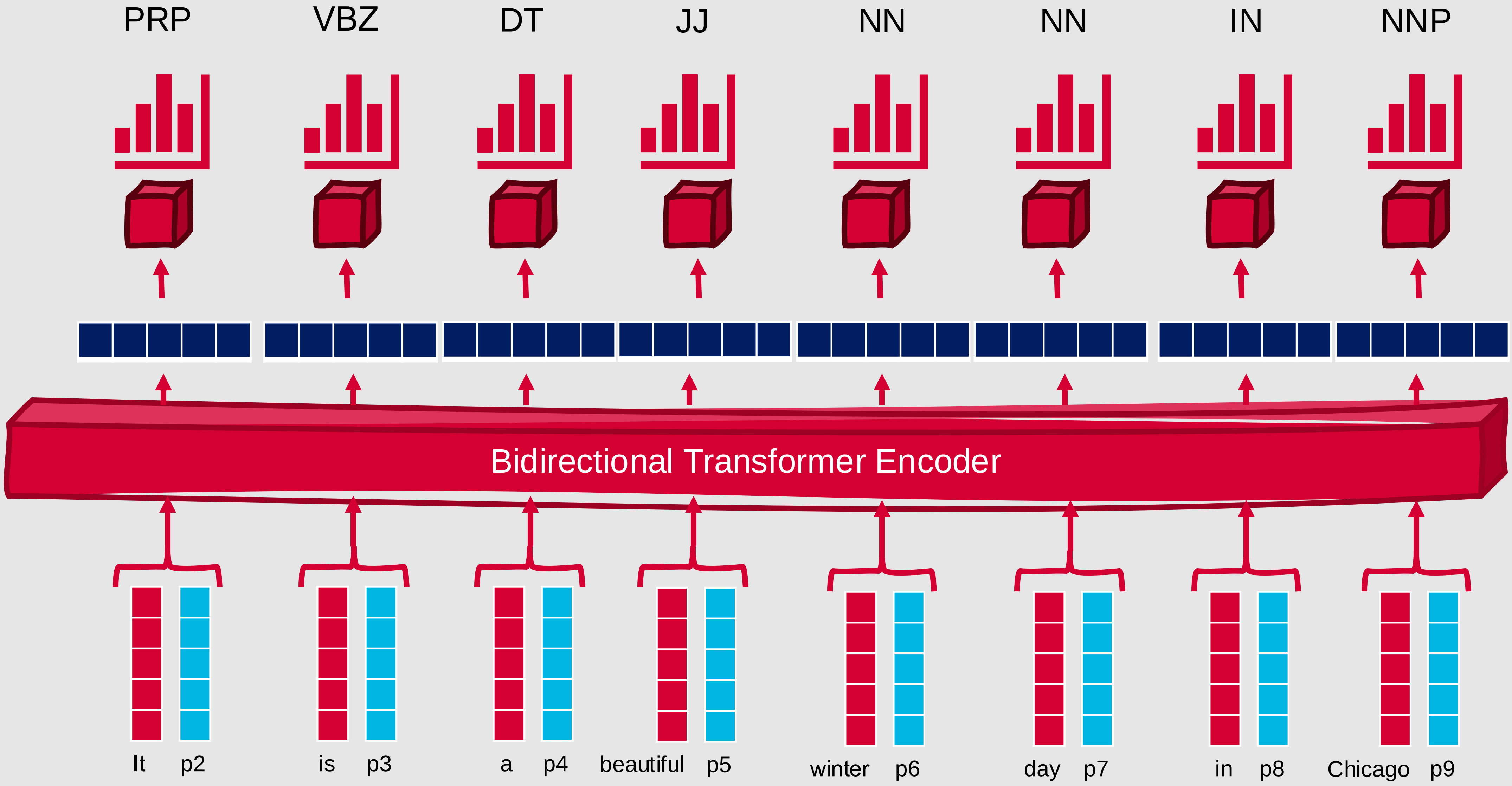
It is a beautiful winter day in Chicago.

Example: Sequence Labeling

Example: Sequence Labeling



Example: Sequence Labeling

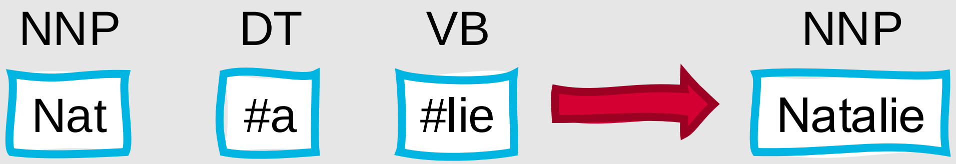


Complication with BERT (and related models)….

- Subword tokenization doesn’t play well with tasks requiring word-level labels
- How to address this?
  - During training, assign the gold standard label for a word to all its constituent subwords
  - During testing, recover word-level labels from subwords as part of the decoding process

Recovering Word-Level Labels

- Simplest approach:
  - For a given word, use the predicted label for its first subword as the label for the entire word
- More complex approaches consider the distribution of label probabilities across all subwords for a given word



Span-Based Sequence Labeling

- Carries attributes of both sequence classification and token-level sequence labeling
  - Goal: **Make decisions using representations of spans** of tokens
- Common Tasks:
  - Identify spans of interest
  - Classify spans
  - Determine relations among spans

Common Span-Based Sequence Labeling Applications

- Named entity recognition
- Question answering
- Syntactic parsing
- Semantic role labeling
- Coreference resolution

Span-Based Sequence Labeling

- Given an input sequence x comprising T tokens $(x_1, x_2, \ldots, x_T)$, a span is a contiguous sequence of tokens from x_i to x_j such that $1 \le i \le j \le T$
- This results in $\frac{T(T-1)}{2}$ possible spans
  - Most span-based models impose an application-specific length limit L
  - Legal spans are those where $(j - i) < L$
- Let the set of legal spans in x be represented as $S(x)$

How do we represent spans for span-based sequence labeling?

- Most span representations incorporate both:
  - **Span boundary representations**
  - **Summary representations** of span content
- These component representations are often concatenated with one another

Span Boundary Representations

- Simple approach: Just use the contextual embeddings of the start and end tokens of the span as the span boundary representations
  - However, internally this doesn't offer a way to distinguish *between* the start and end tokens
  - Words may carry different meaning at the beginning of a span than at the end!
- More complex approach: Use separate feedforward networks to learn representations for the beginning and end of the span
  - $\mathbf{s}_i = \mathrm{FFNN}_s(\mathbf{h}_i)$
  - $\mathbf{e}_j = \mathrm{FFNN}_e(\mathbf{h}_j)$

Summary Representations

- Simple approach: Just use the average of the output embeddings for words within the span as the summary representation
  - $\mathbf{g}_{ij} = \frac{1}{(j-i)+1} \sum_{k=i}^{j} \mathbf{h}_k$

- More complex approach: Place more representational emphasis on the head of the span
  - Can be done using syntactic parse information (if available) or a self-attention layer (if not)
  - $\mathbf{g}_{ij} = \text{SelfAttention}(\mathbf{h}_{i:j})$

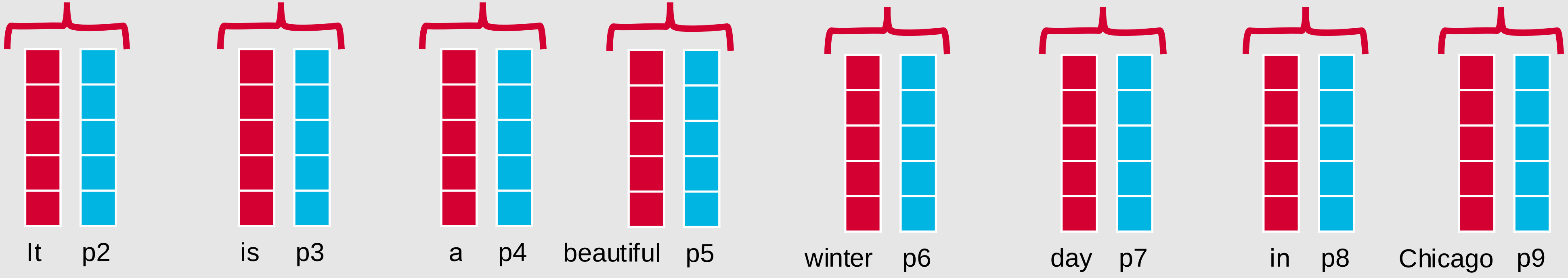
How does fine-tuning work in span-based sequence labeling?

- Learn the weights/parameters for:
  - Task classification head
  - Boundary representations
  - Summary representation
- Final classification output:
  - $\mathbf{span}_{ij} = [\mathbf{s}_i; \mathbf{e}_j; \mathbf{g}_{ij}]$
  - $\mathbf{y}_{ij} = \mathrm{softmax}(\mathrm{FFNN}(\mathbf{span}_{ij}))$

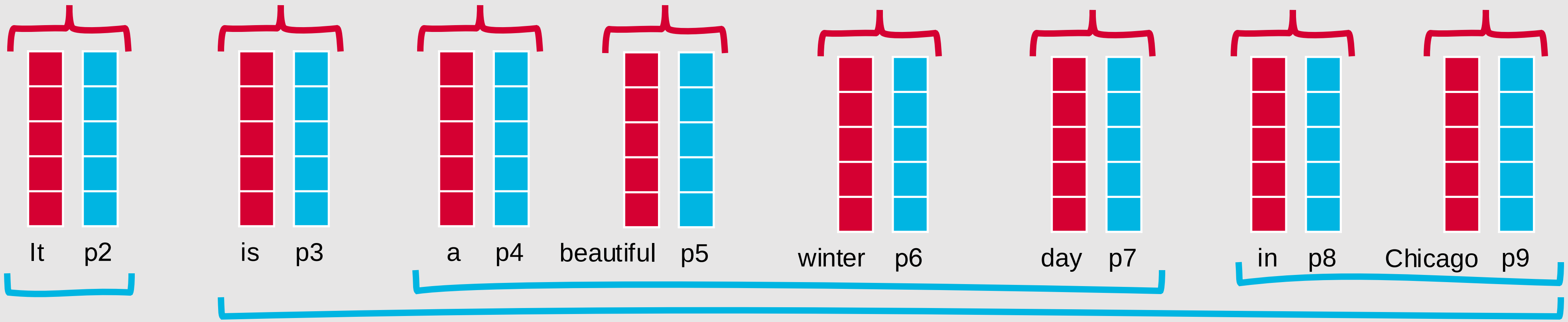
Example: Span-Based Sequence Labeling

It is a beautiful winter day in Chicago.

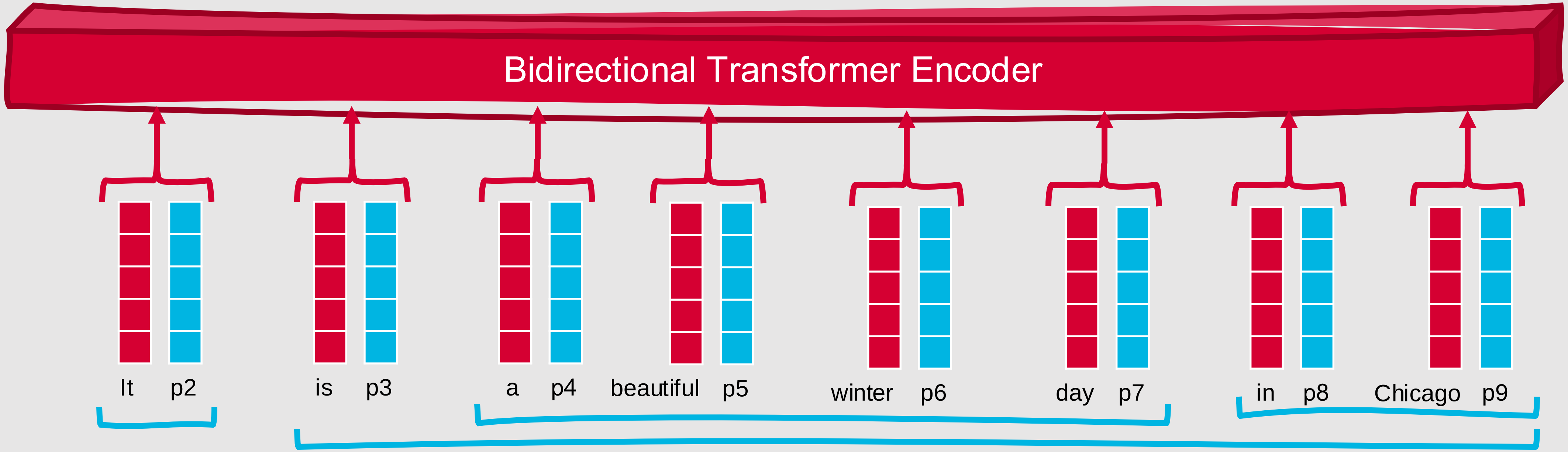
Example: Span-Based Sequence Labeling



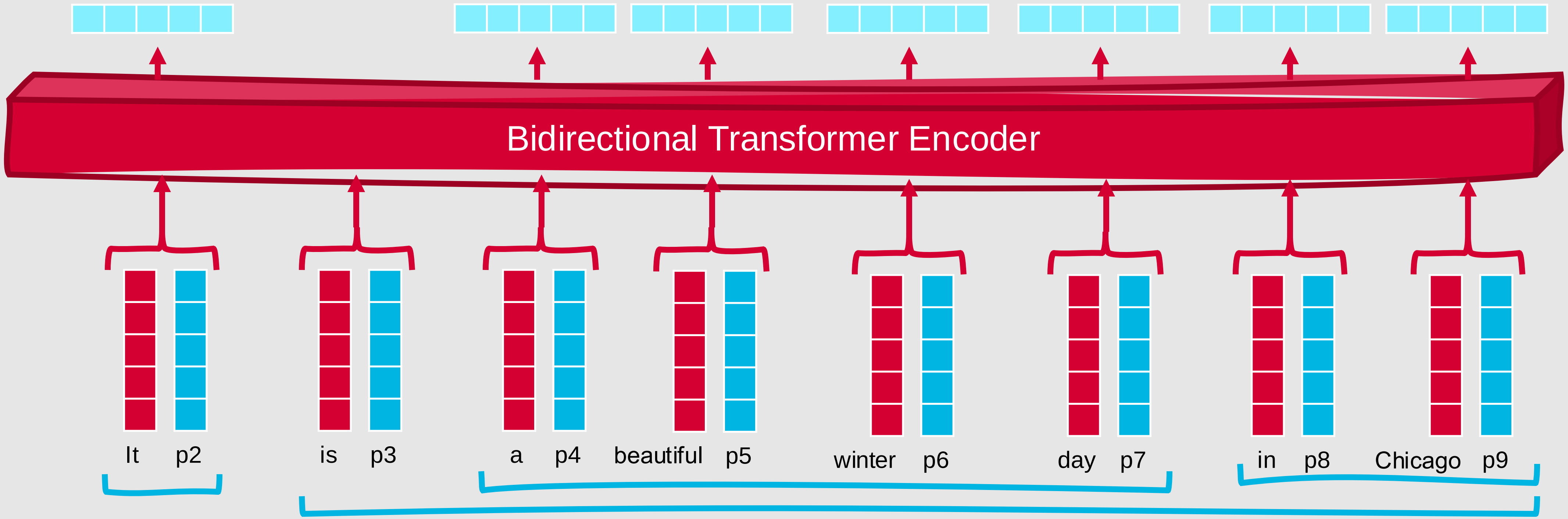
Example: Span-Based Sequence Labeling



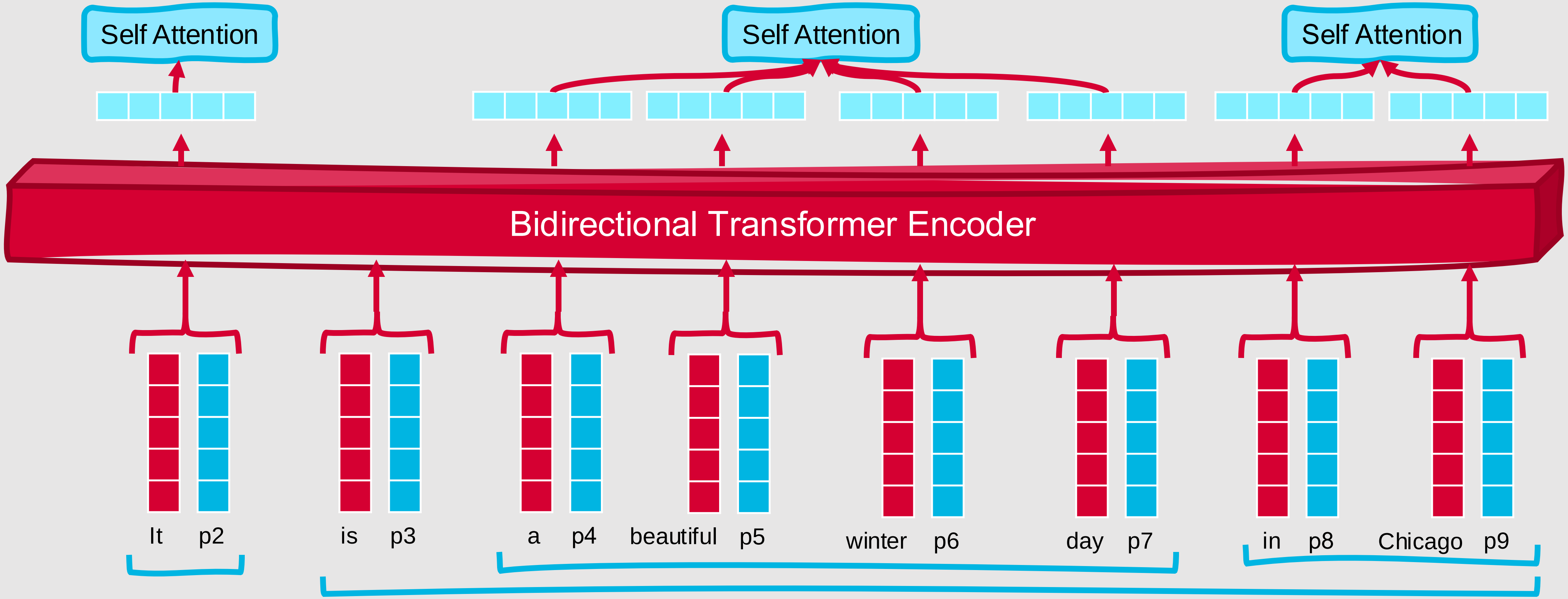
Example: Span-Based Sequence Labeling



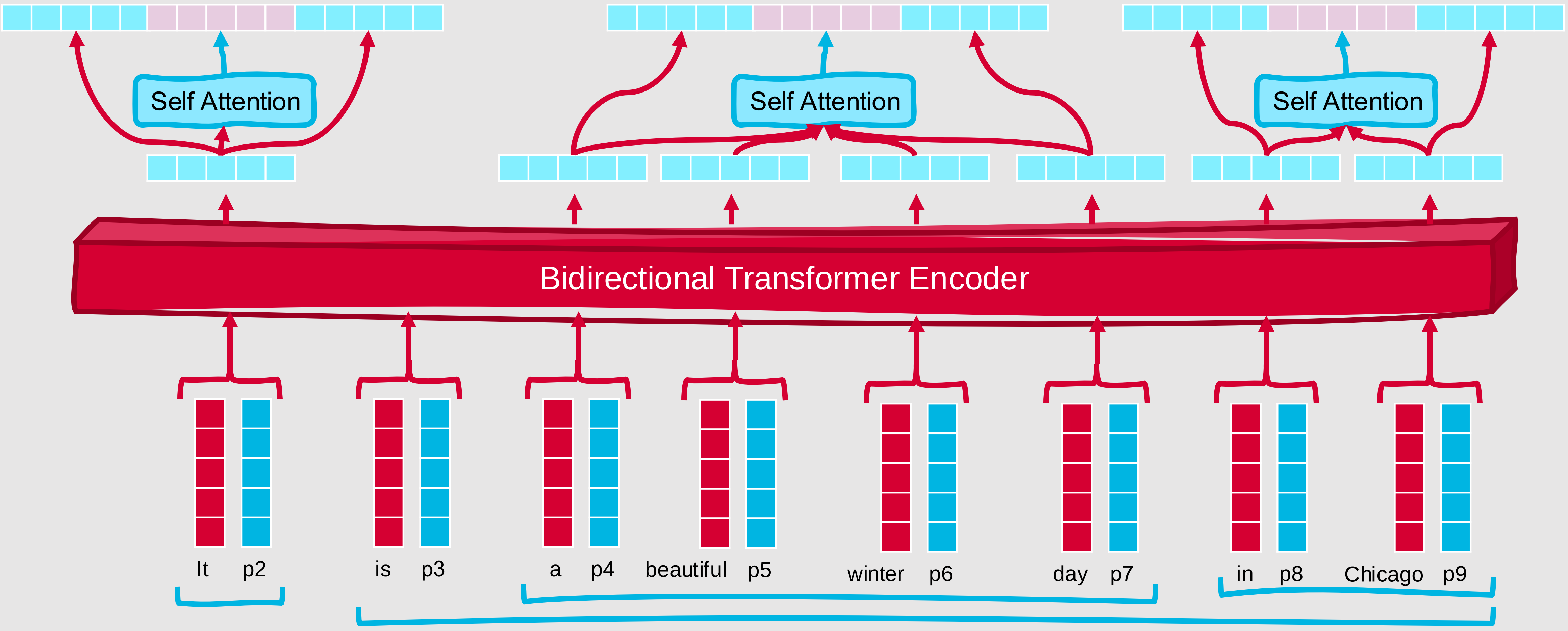
Example: Span-Based Sequence Labeling



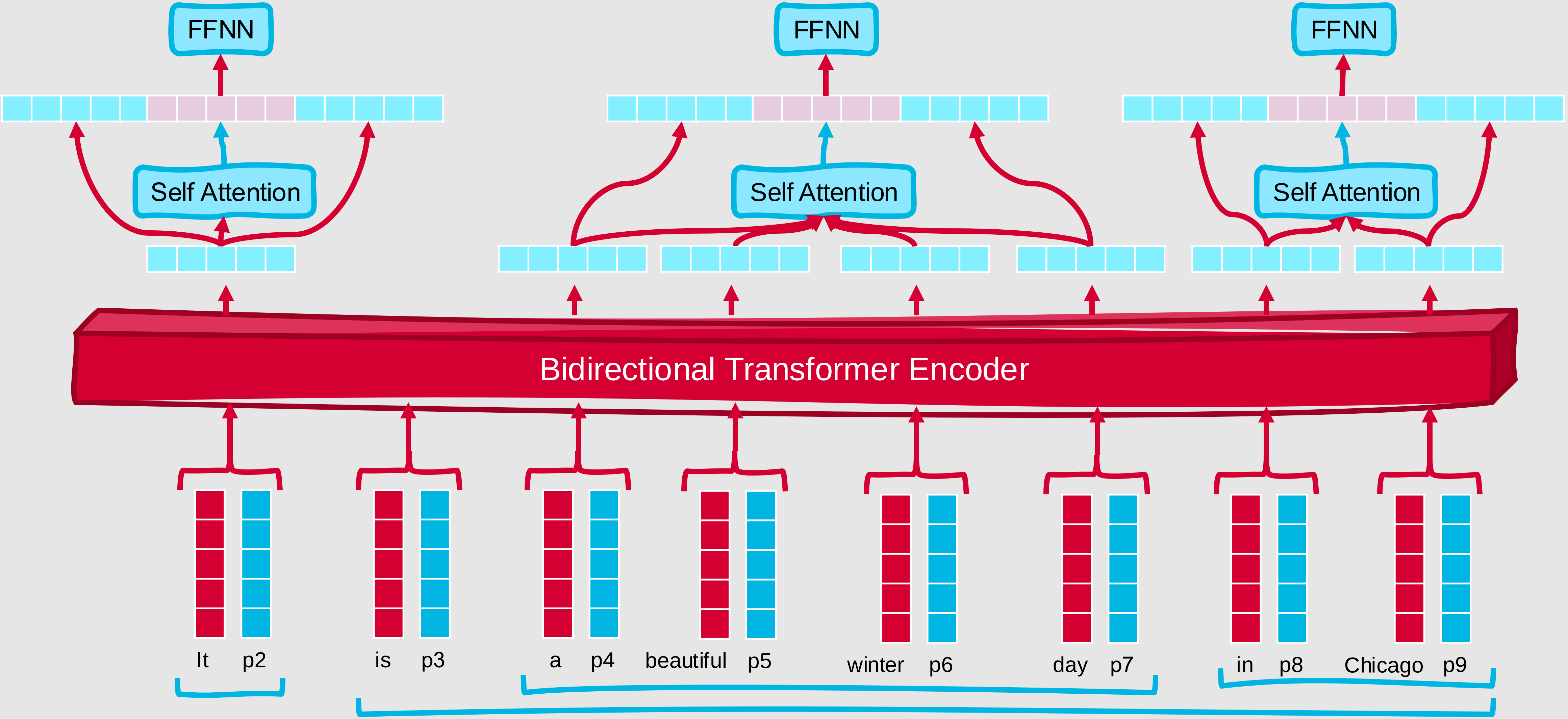
Example: Span-Based Sequence Labeling



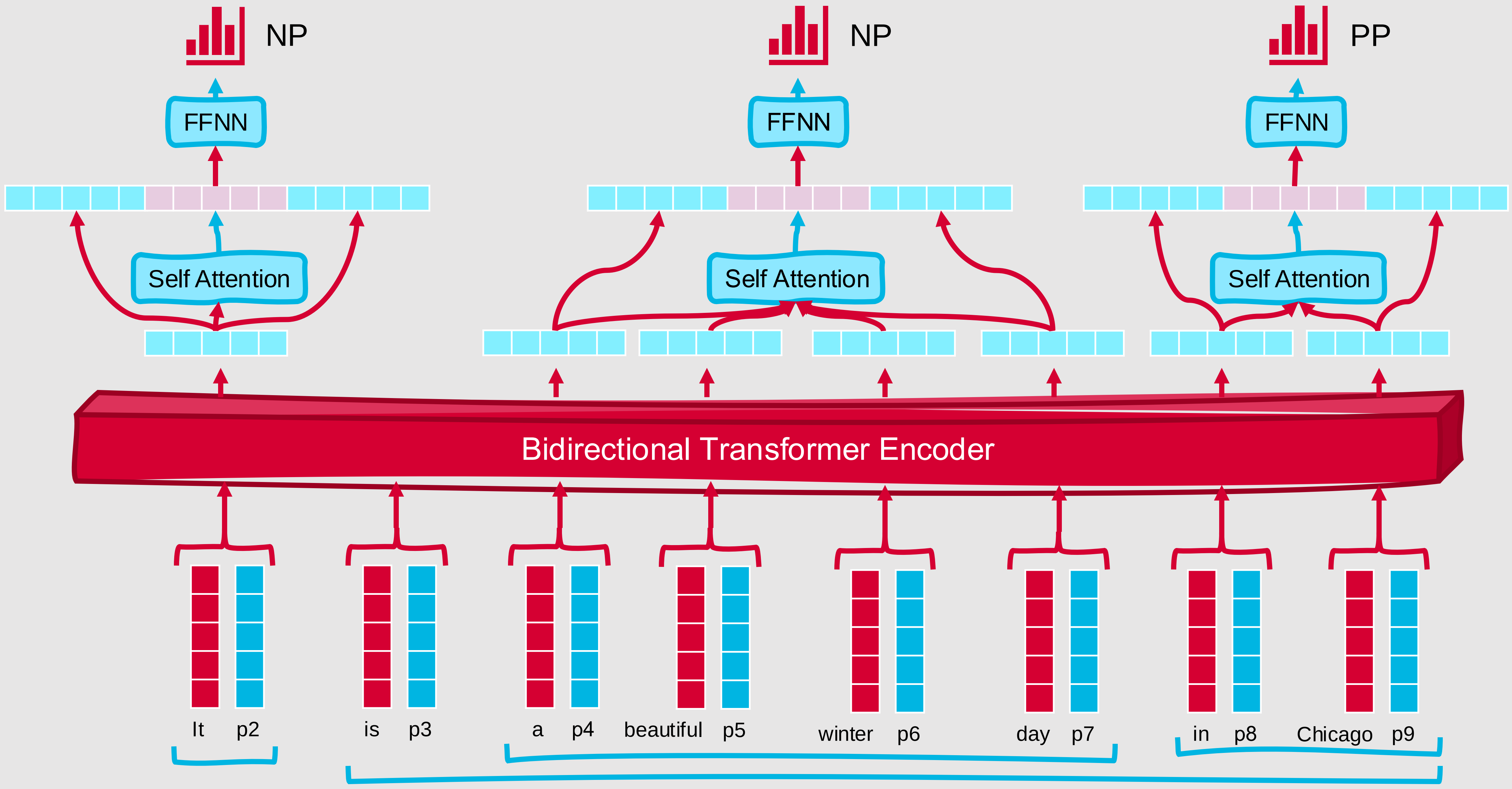
Example: Span-Based Sequence Labeling



Example: Span-Based Sequence Labeling



Example: Span-Based Sequence Labeling



Advantages of Span-Based Sequence Labeling

- Only require one label assignment per span
  - In comparison, BIO-based methods require labels for each constituent token
- Naturally accommodate hierarchical and/or overlapping labels
  - BIO-based methods assign a single label per token

We’ve learned a lot about transfer learning and pretrained language models …how can we implement them?

- 🤗
  - https://huggingface.co/docs/transformers/index
- TensorFlow
  - https://www.tensorflow.org/text/tutorials/classify_text_with_bert
- PyTorch
  - https://pytorch.org/hub/huggingface_pytorch-transformers/

Where do large language models (LLMs) fit in?

- What is “large”?
  - Not clearly defined, but generally speaking, anything “BERT-sized” (~110 million parameters) or larger
- Trained on massive quantities of text data to predict which word(s) should appear, given a context
- Can theoretically use any architecture that works for this setting, but in practice, modern LLMs are Transformer models

How are LLMs pretrained?

- Can be pretrained with numerous objectives
  - Masked language modeling
  - Next sentence prediction
  - Autoregressive generation
- Different pretraining objectives are useful for different purposes
  - Pretraining for masked language modeling may produce LLMs that are especially well-suited for classification
  - Pretraining for autoregressive generation may produce LLMs that are especially well-suited for longer-form generation tasks

What’s most popular right now?

- The most popular LLMs right now (e.g., GPT-X or LLaMa) are pretrained for autoregressive generation
  - Given the sequence of words that have been generated so far, decide which word should come next

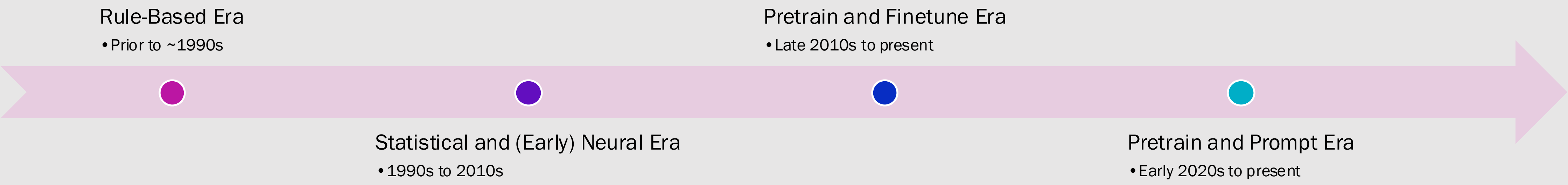
Generative	Pretrained	Transformer
G	P	T

Is this a step back?

- First came autoregressive generation, then came masked language modeling, then came …autoregressive generation again?
    - Autoregressive generation *without* instruction tuning is only useful for limited purposes (e.g., autocomplete)
    - Autoregressive generation + **instruction tuning** + **reinforcement learning with human feedback** (+ **better prefixes**) is a very recent development, and much more useful!

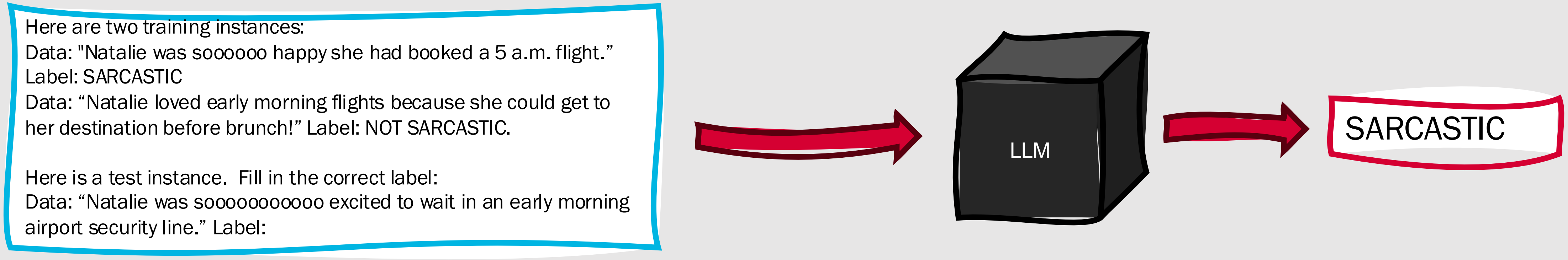
In fact, these recent developments have ushered in a new training paradigm.

- Why?
  - Fine-tuning pretrained models to perform new tasks works very well in many cases, but it still requires that you have a reasonably large supervised training set for the target task
  - In some cases, we only have a very tiny amount of training data (or none at all) for our target task!



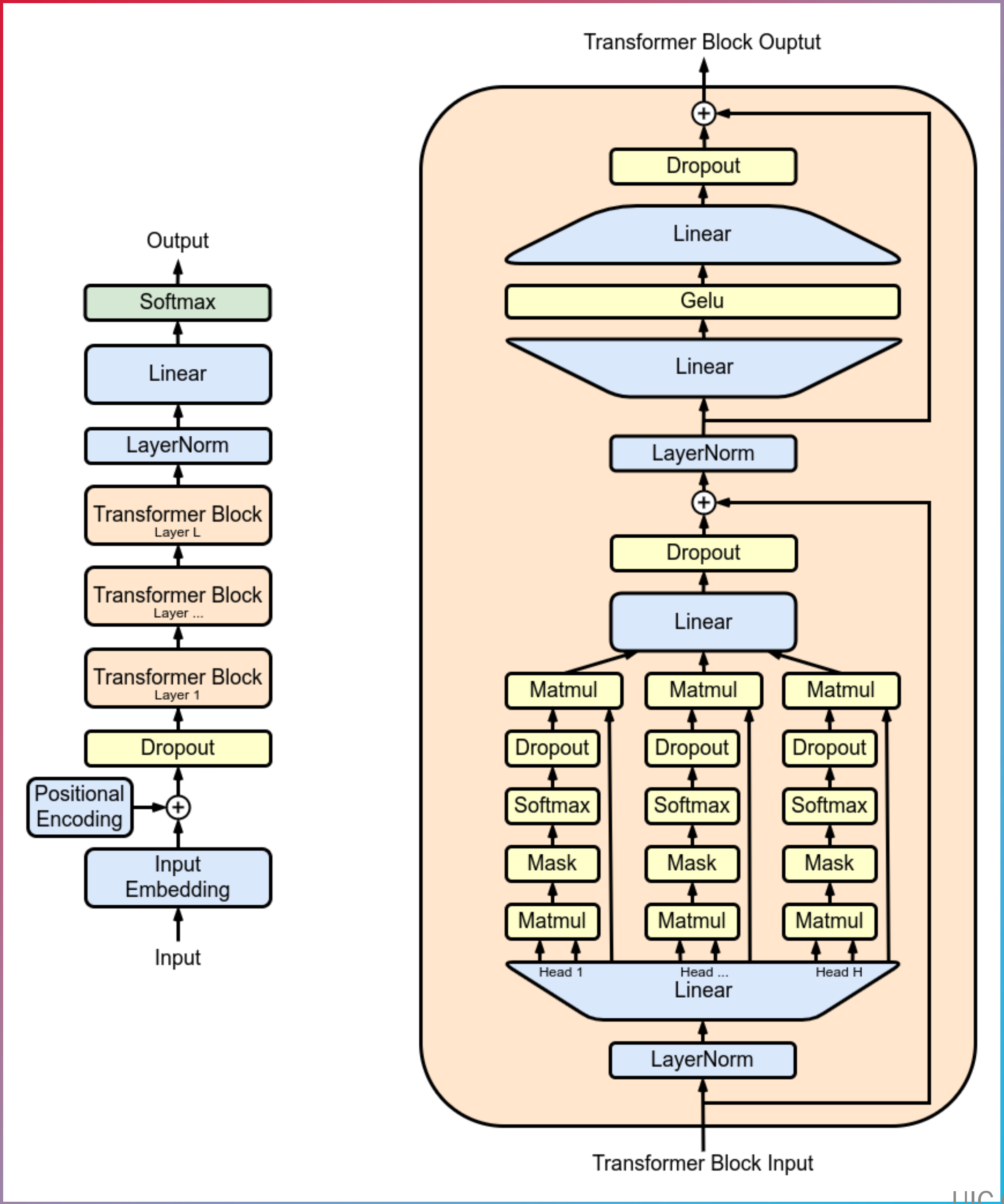
Introducing: Pretrain (and Optionally Fine-Tune) and Prompt

- Intuition:
  - If we take LLMs that have been pretrained on a wide variety of language data, we can prompt them to produce the correct labels or output for new tasks



This new paradigm has seen remarkably rapid uptake in the NLP community!

	# Full, Main Conference Papers with "Prompt" in Title
ACL 2022	22
EMNLP 2022	41
ACL 2023	36
EMNLP 2023	44
ACL 2024	38
EMNLP 2024	55



At the core of most recent work are **generative pretrained Transformers** (**GPTs**).

- Original GPT architecture was published in 2018: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
  - Transformer *decoder* model
  - 12 Transformer blocks
  - 12 attention heads per self-attention layer
  - Trained on BooksCorpus
    - 7000 books

Popular Large (Generative) Language Models

- Since the original GPT, these models have grown increasingly larger!
  - GPT-X
    - ~0.5+ trillion tokens of pretraining data (last reported for GPT-3; speculation for GPT-4 is a much higher number)
  - LLaMa 3
    - ~15 trillion tokens of pretraining data
- How much data *is* a trillion tokens?
  - ~15,000,000 books!

Open vs. Closed Models

GPT-4

With broad general knowledge and domain expertise, GPT-4 can follow complex instructions in natural language and solve difficult problems with accuracy.

Learn about GPT-4

Model	Input	Output
gpt-4	$0.03 / 1K tokens	$0.06 / 1K tokens
gpt-4-32k	$0.06 / 1K tokens	$0.12 / 1K tokens

GPT-3.5 Turbo

GPT-3.5 Turbo models are capable and cost-effective.

`gpt-3.5-turbo-0125` is the flagship model of this family, supports a 16K context window and is optimized for dialog.

`gpt-3.5-turbo-instruct` is an Instruct model and only supports a 4K context window.

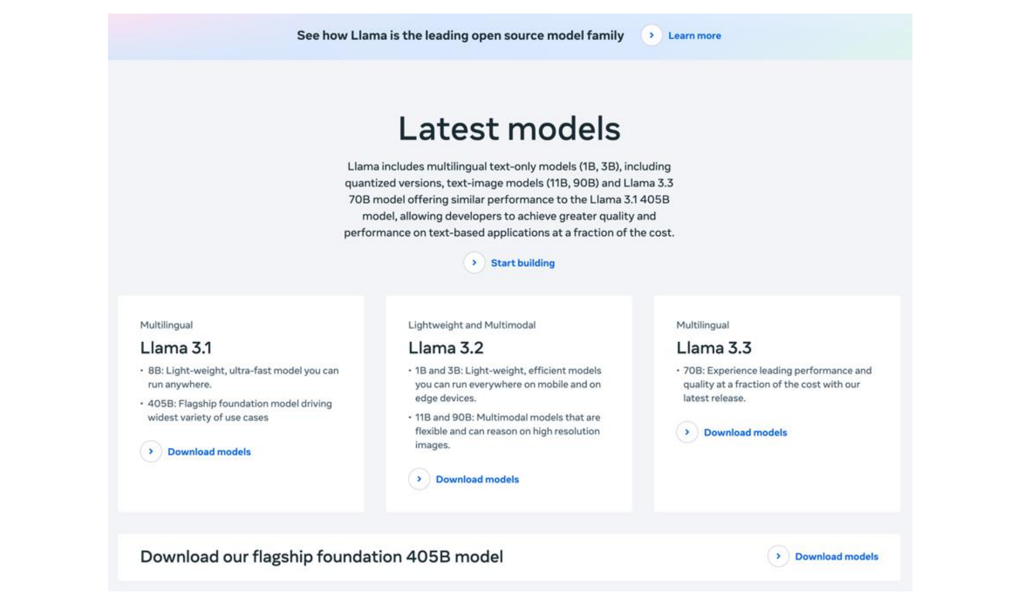
Learn about GPT-3.5 Turbo ↗

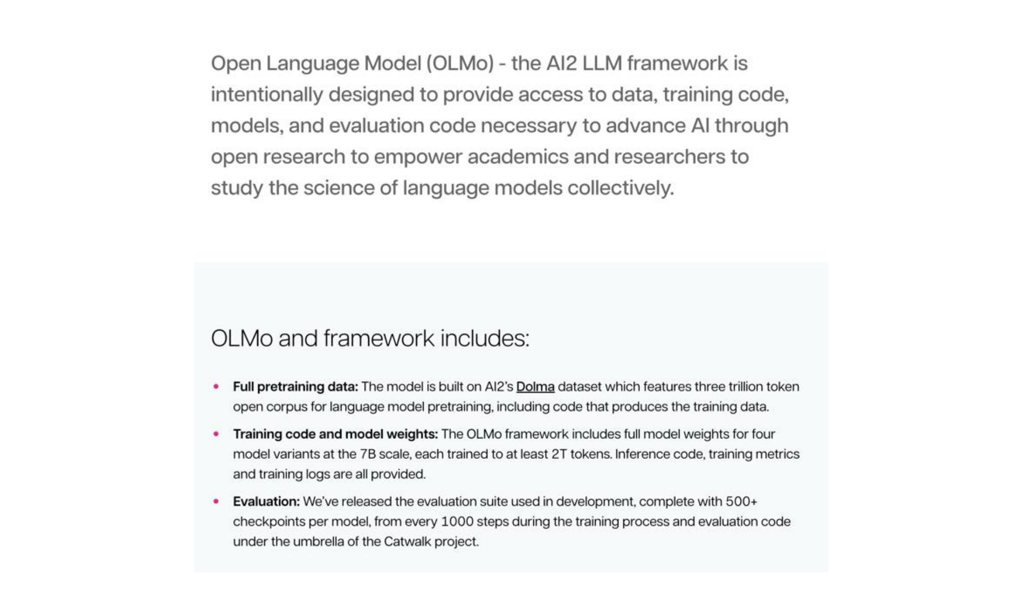
Model	Input	Output
gpt-3.5-turbo-0125	$0.0005 / 1K tokens	$0.0015 / 1K tokens
gpt-3.5-turbo-instruct	$0.0015 / 1K tokens	$0.0020 / 1K tokens

- Many popular high-performing LLMs are closed models
  - Full model cannot be modified or directly accessed by researchers
  - Details about training data and architecture may be scarce
  - Accessible via paid API
  - Example: GPT-4

Open vs. Closed Models

- However, very recent interest (and helpful efforts from community members!) have led to the public release of several open-source LLMs
  - Fully accessible and modifiable
  - Architecture is fully explorable
  - Free!
  - Examples:
    - Llama: https://www.llama.com/
    - OLMo: https://allenai.org/olmo





LLM Resources

- Open LLM Leaderboard: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard
- A Survey of Large Language Models
  - Paper: https://arxiv.org/abs/2303.18223
  - Repository: https://github.com/RUCAIBox/LLMSurvey
- Generative models on the Hugging Face model hub: https://huggingface.co/models?pipeline_tag=text-generation&sort=trending

Summary: Transfer Learning with Pretrained Language Models and Large Language Models

Bidirectional Transformer encoders learn representations by optimizing for two tasks:

- Masked language modeling
- **Next sentence prediction**

Pretrained language models can be **fine-tuned** for a variety of downstream tasks by adding classification heads to the end of the model

These tasks may include:

- Sequence classification
- Sequence labeling
- Span-based sequence labeling

Large language models are typically generative pretrained Transformer models with an autoregressive language modeling learning objective