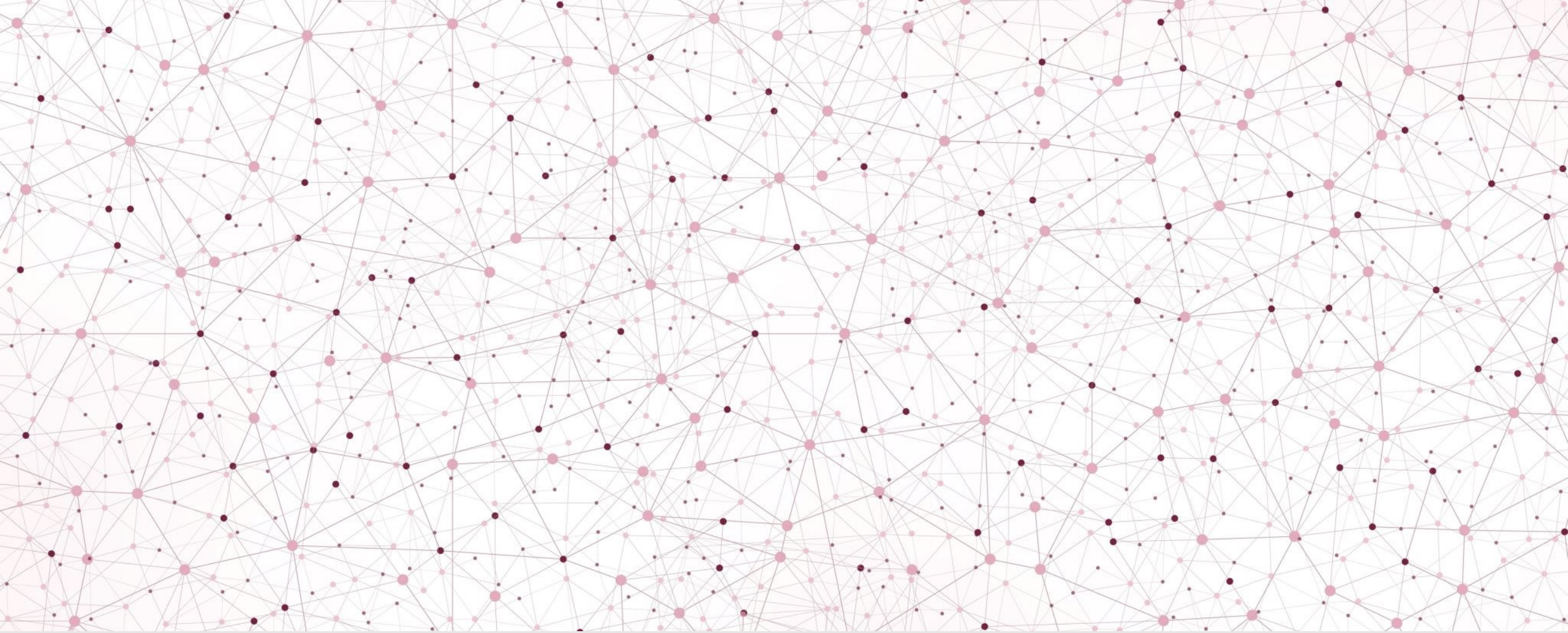
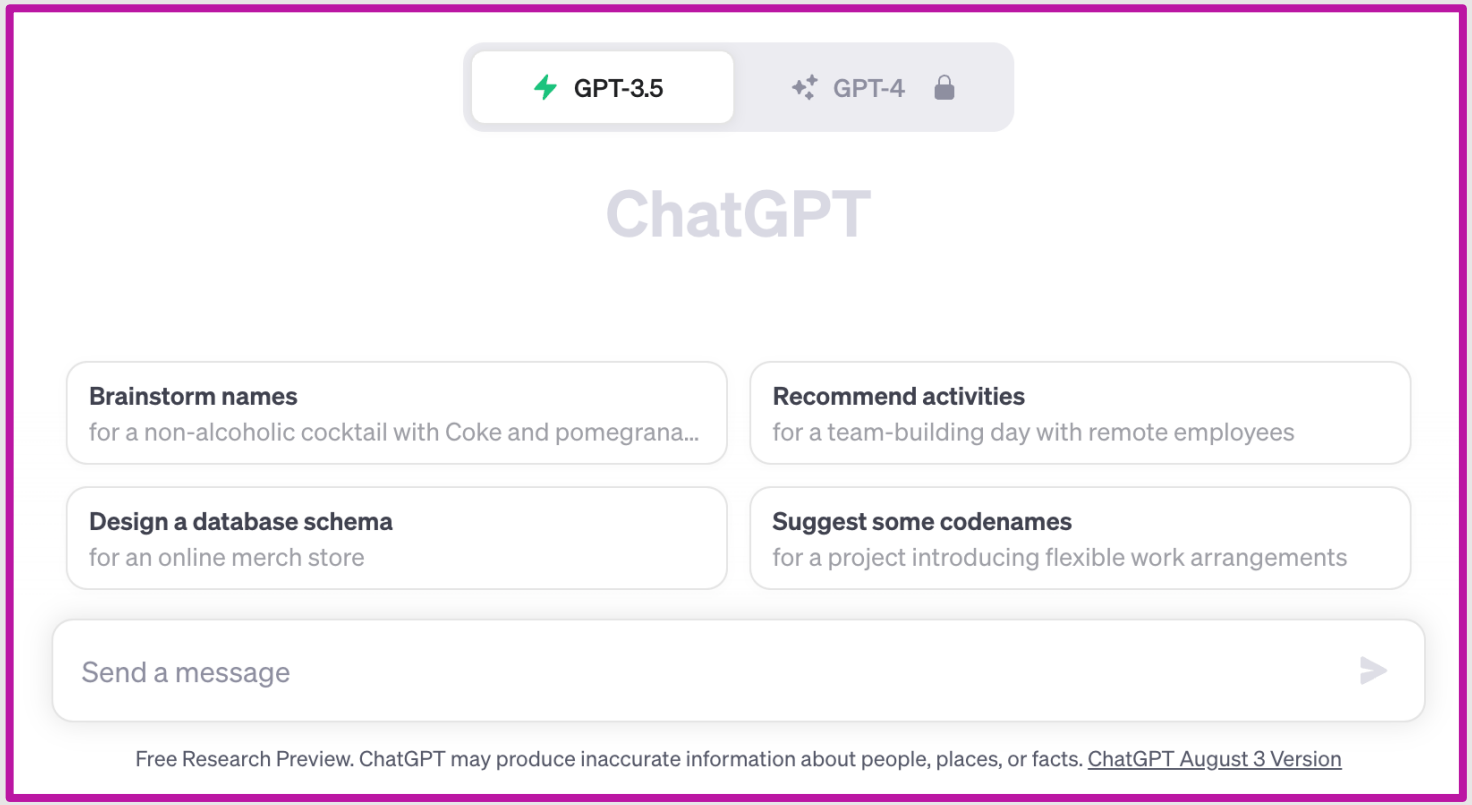


Overview of Deep Learning for NLP

Natalie Parde

UIC CS 421

Many modern NLP approaches are implemented using deep learning.



BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

Abstract

We introduce a new language representation model called **BERT**, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

1 Introduction

Language model pre-training has been shown to be effective for improving many natural language processing tasks (Dai and Le, 2015; Peters et al., 2018a; Radford et al., 2018; Howard and Ruder, 2018). These include sentence-level tasks such as natural language inference (Bowman et al., 2015; Williams et al., 2018) and paraphrasing (Dolan and Brockett, 2005), which aim to predict the relationships between sentences by analyzing them holistically, as well as token-level tasks such as named entity recognition and question answering, where models are required to produce fine-grained output at the token level (Tjong Kim Sang and De Meulder, 2003; Rajpurkar et al., 2016).

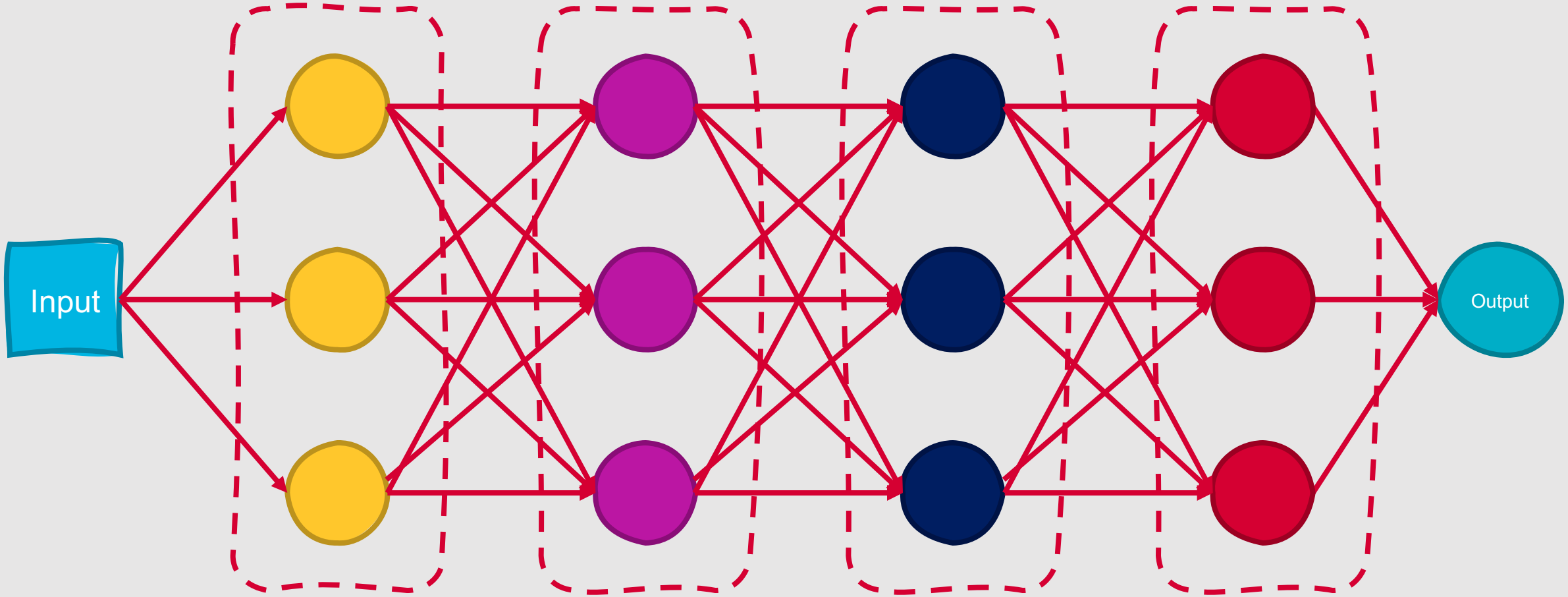
There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the power of the pre-trained representations, especially for the fine-tuning approaches. The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. For example, in OpenAI GPT, the authors use a left-to-right architecture, where every token can only attend to previous tokens in the self-attention layers of the Transformer (Vaswani et al., 2017). Such restrictions are sub-optimal for sentence-level tasks, and could be very harmful when applying fine-tuning based approaches to token-level tasks such as question answering, where it is crucial to incorporate context from both directions.

In this paper, we improve the fine-tuning based approaches by proposing BERT: **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. BERT alleviates the previously mentioned unidirectionality constraint by using a "masked language model" (MLM) pre-training objective, inspired by the Cloze task (Taylor, 1953). The masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked

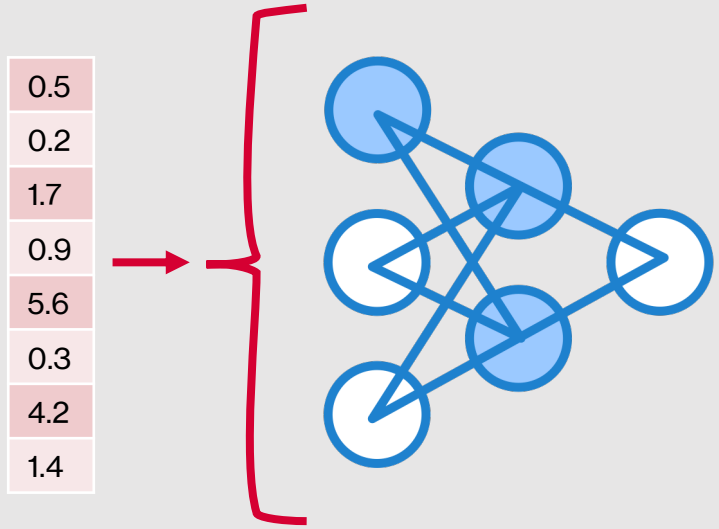
Proceedings of NAACL-HLT 2019, pages 4171–4186
Minneapolis, Minnesota, June 2 - June 7, 2019.

How does deep learning work?



Common Themes across Deep Learning Approaches

- Input is typically a **dense vector representation**
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes

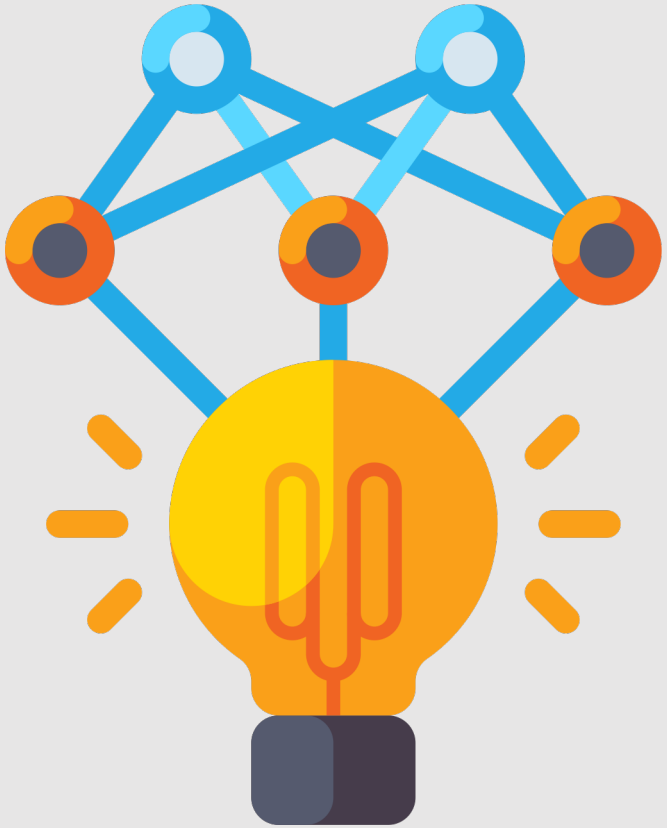


Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a **hyperparameter tuning process**
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data

Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a hyperparameter tuning process
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data
- Output is **task-dependent**
  - Can be a class label, a number, or a string of generated text



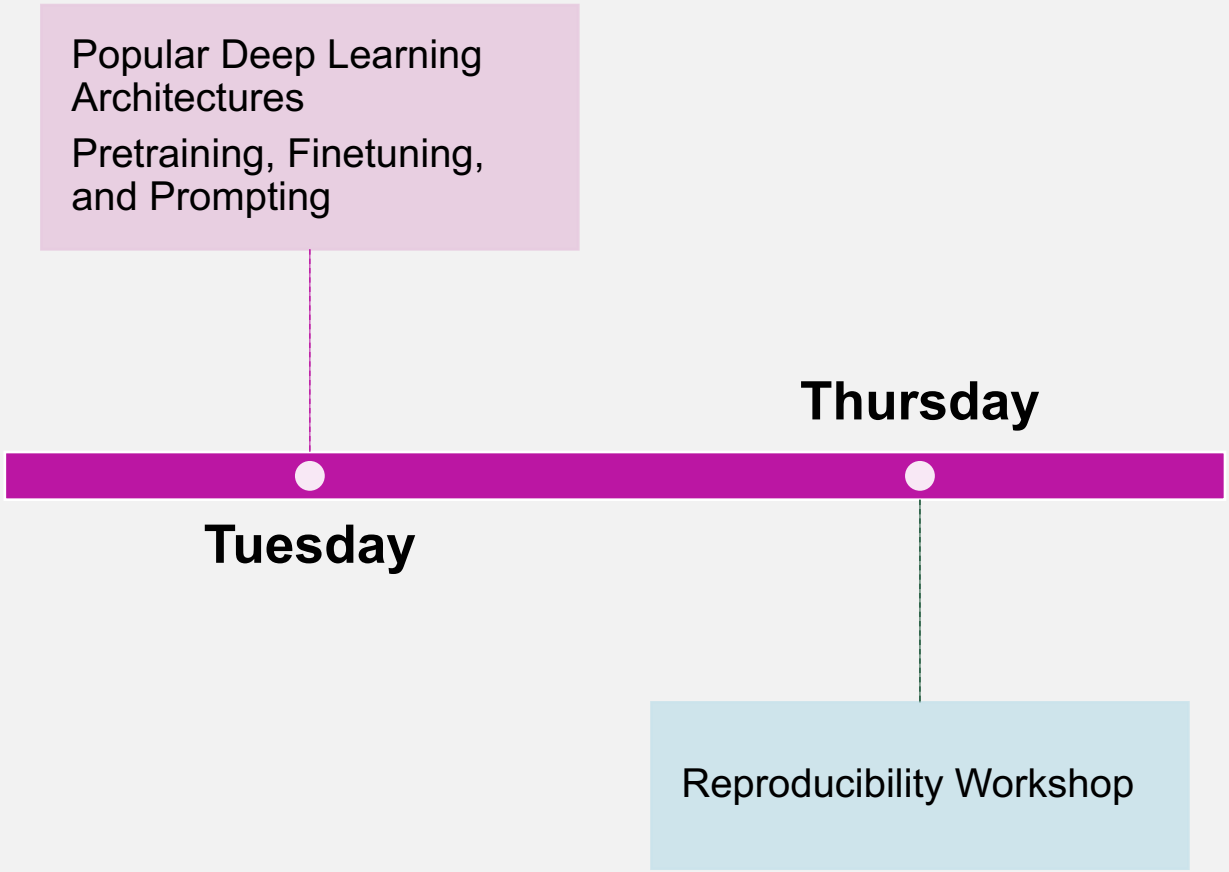
Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a hyperparameter tuning process
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data
- Output is task-dependent
  - Can be a class label, a number, or a string of generated text
- Training can be performed **end-to-end**
  - The model is trained to predict the target output directly, rather than through pipelined components

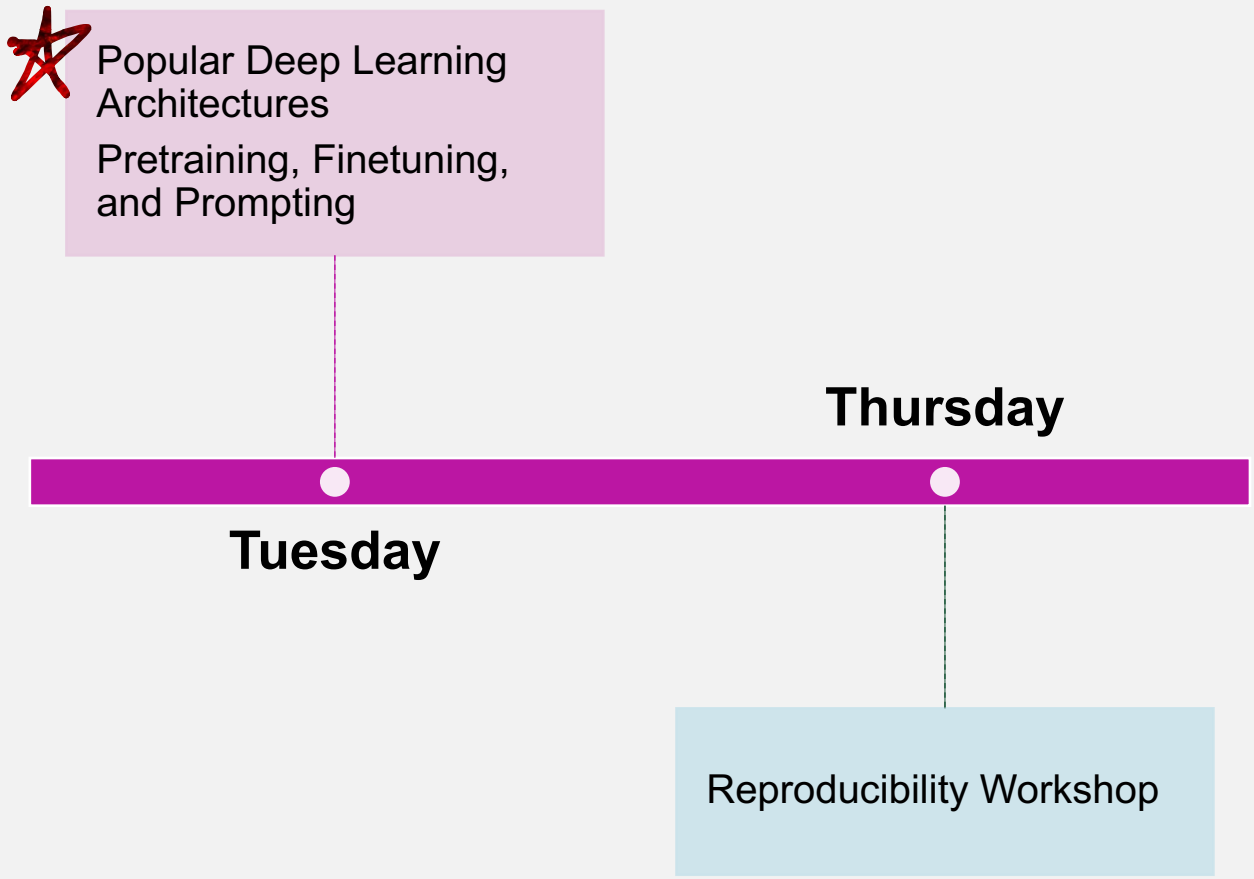
Despite these common themes, deep learning models are implemented in many different ways!

- They may vary in how they:
  - Handle prior context
  - Draw inferences from the data
  - Pass data between layers
- These variations make different kinds of deep learning models work better for different tasks

This Week's Topics

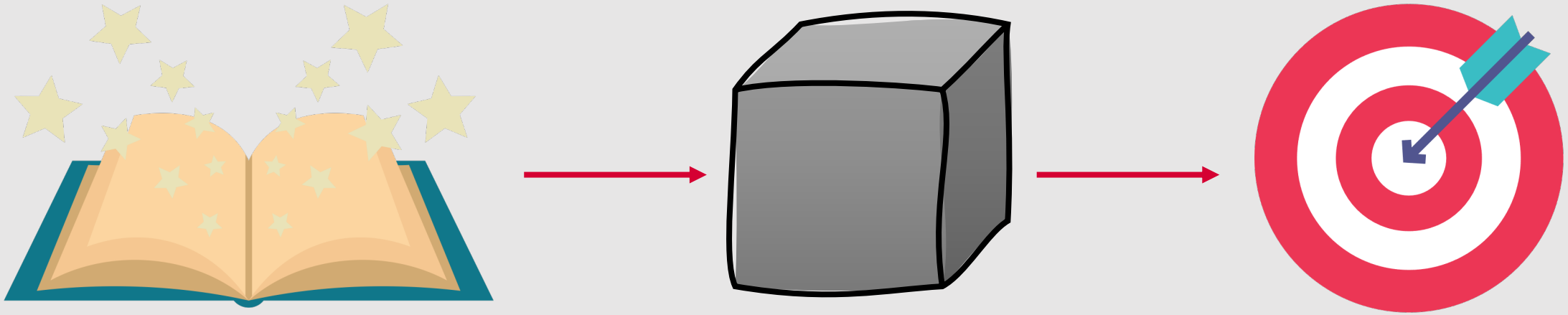


This Week's Topics



Popular Deep Learning Architectures in Contemporary NLP

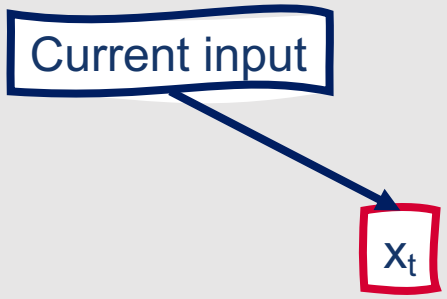
- Recurrent Neural Networks
- Convolutional Neural Networks
- Transformers



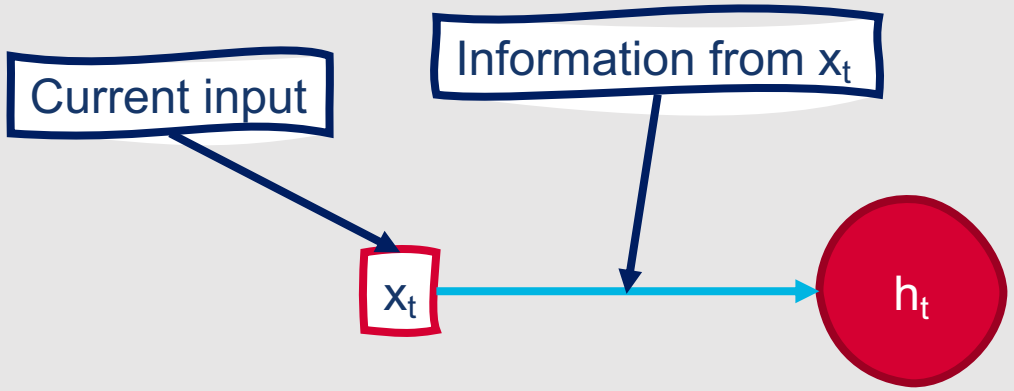
Recurrent Neural Networks (RNNs)

- General premise:
  - Deep learning models should be making decisions for sequential input based on decisions that have already been made at earlier points of the sequence
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for recurrent neural networks:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer **+ a vector of numbers representing the layer's output at the previous timestep**

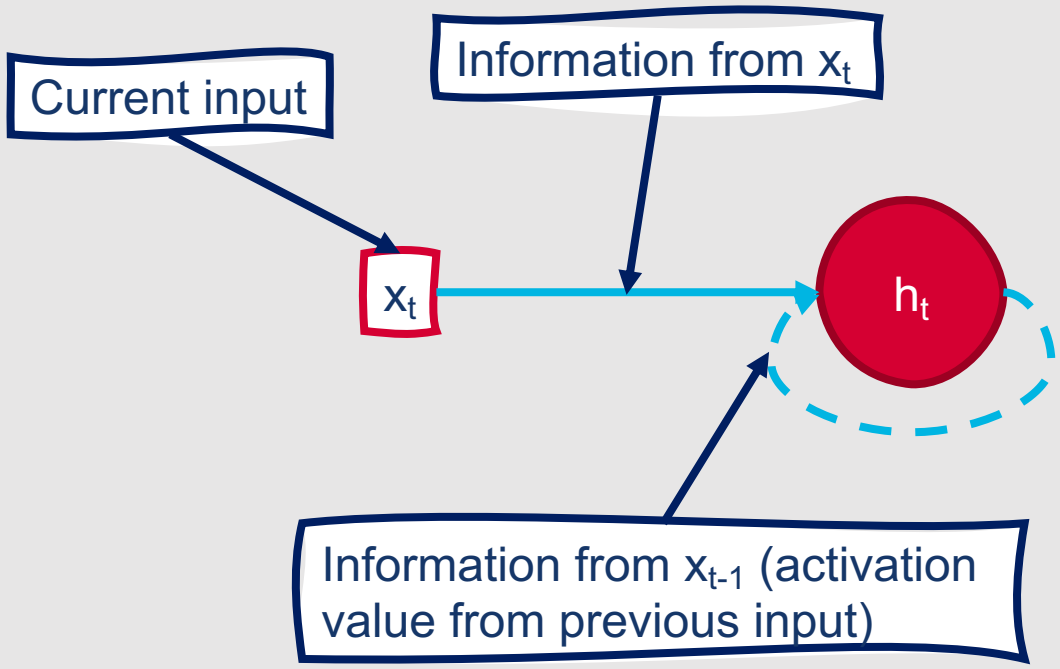
Structure of Single-Unit RNN Layer



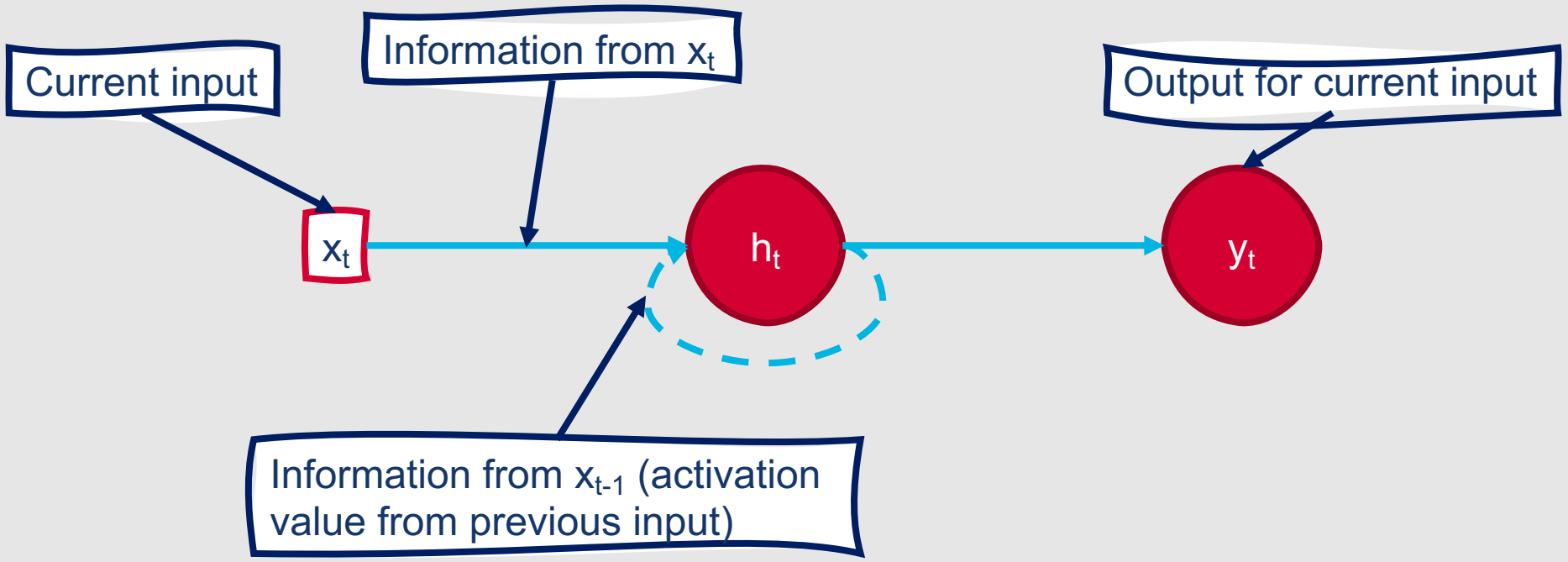
Structure of Single-Unit RNN Layer



Structure of Single-Unit RNN Layer



Structure of Single-Unit RNN Layer

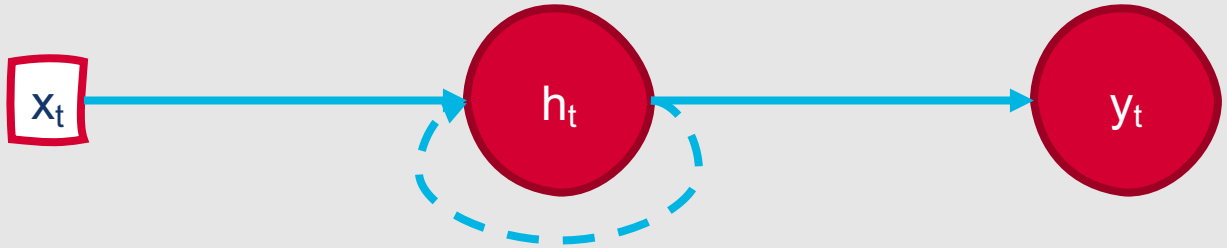


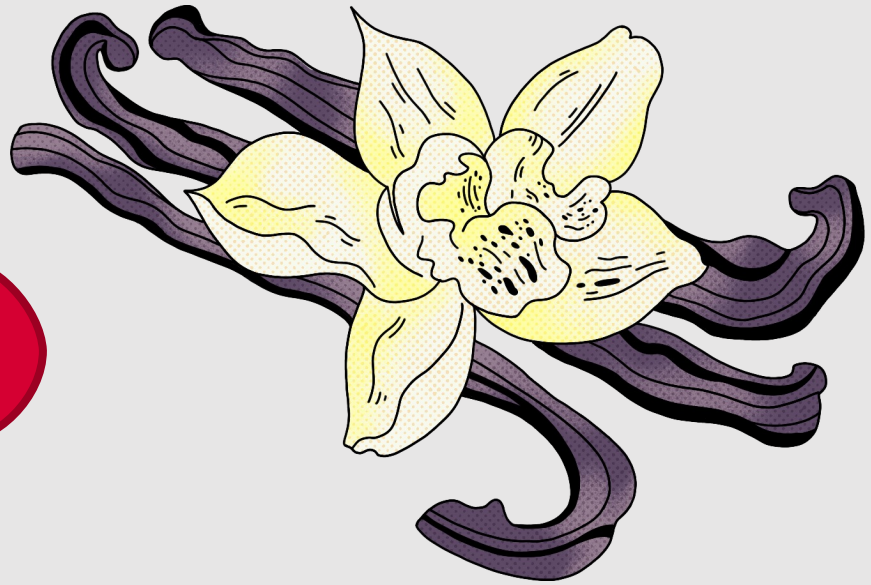
Why is this useful for NLP problems?

- Most data for NLP tasks is inherently sequential!
- Making use of sequences using feedforward neural networks requires:
  - Fixed-length context windows
  - Concatenated context vectors
- This limits the model's abilities, and prevents it from considering variable-length context

There are many popular variations of RNNs.

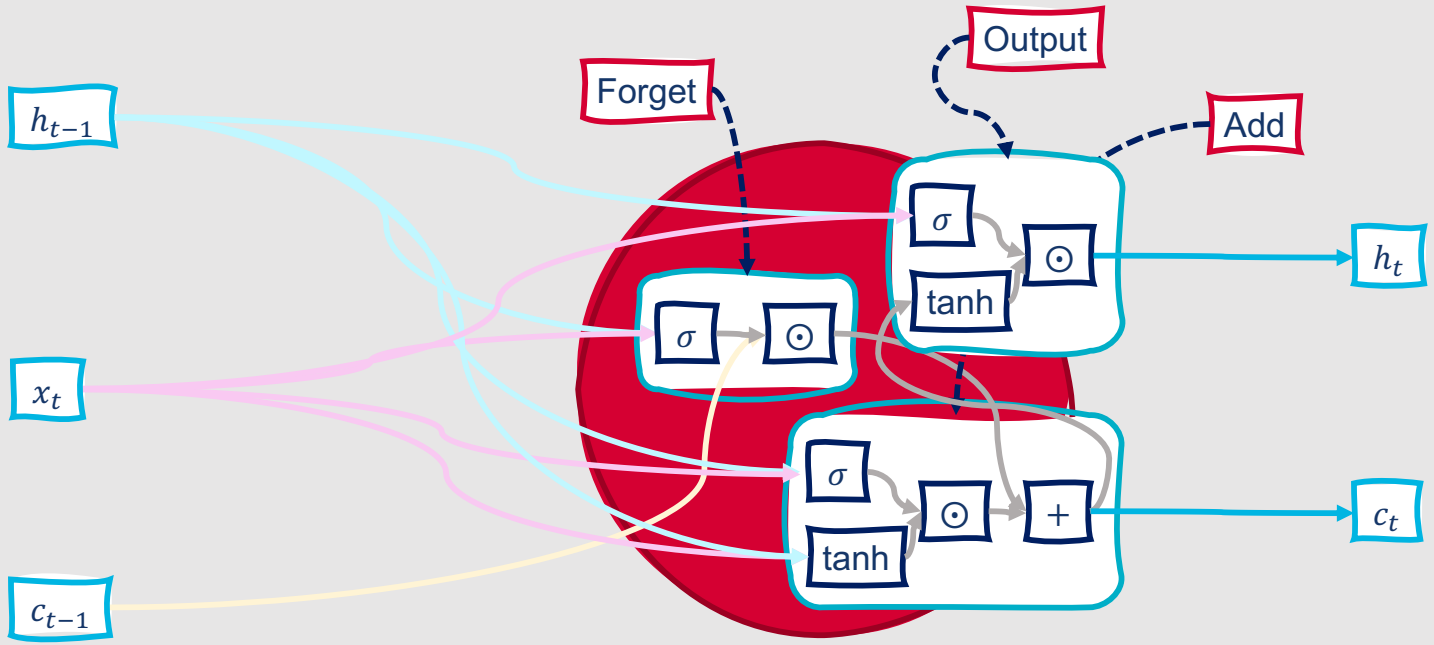
- "Standard" RNNs are often referred to informally as **vanilla RNNs**
- Some RNN architectures are modified to specifically improve the model's ability to consider long-term context
  - **Long short-term memory networks** (LSTMs)
  - **Gated recurrent units** (GRUs)





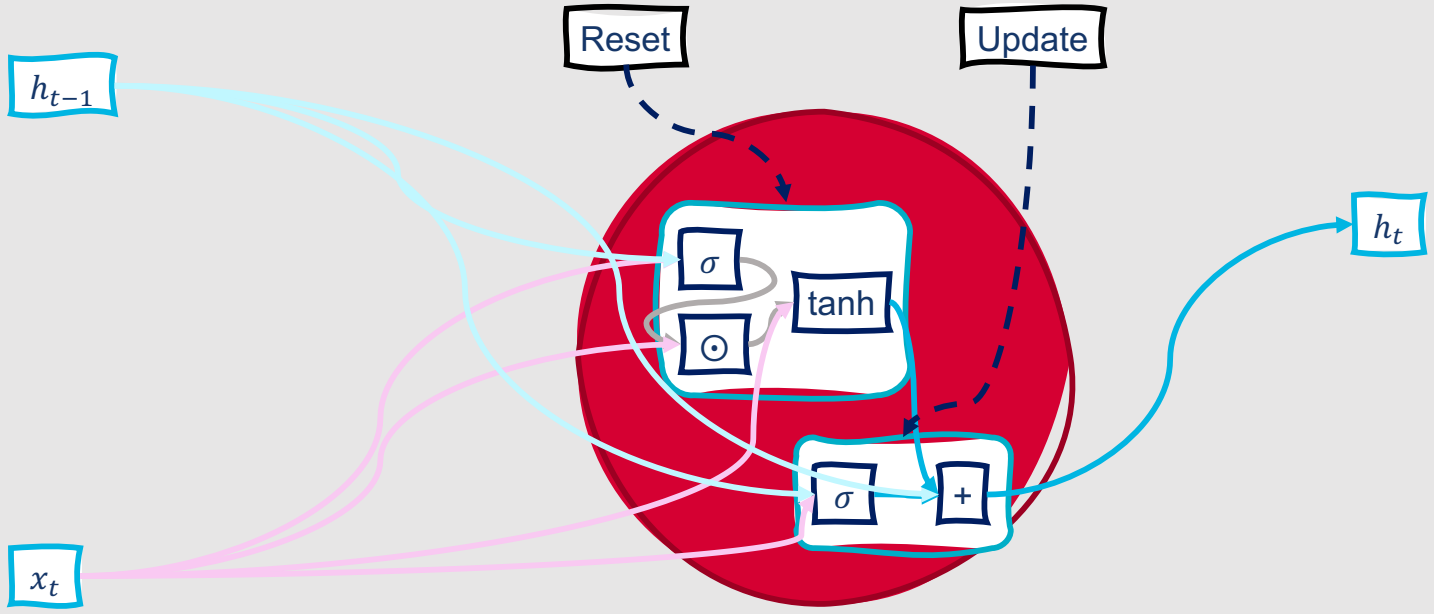
Long Short-Term Memory Networks (LSTMs)

- Specialized RNN units that incorporate gating mechanisms to remove information that is no longer needed from the context, and add information that is anticipated to be of use later
- Gating mechanisms include:
  - **Forget gate:** Should we erase this existing information from the context?
  - **Add gate:** Should we write this new information to the context?
  - **Output gate:** What information should be leveraged for the current hidden state?

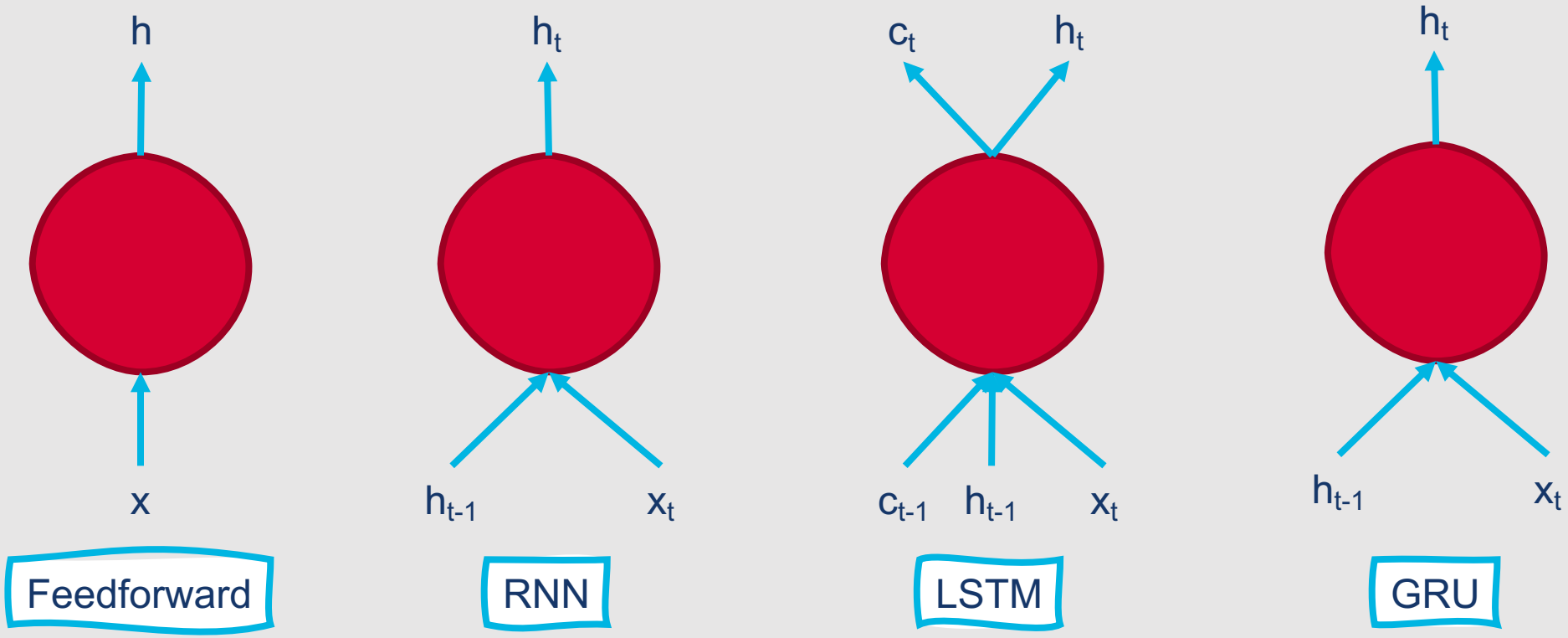


Gated Recurrent Units (GRUs)

- Also utilizes gating mechanisms to manage contexts, but uses a simpler architecture than LSTMs
- Only two gates:
  - **Reset gate:** Which elements of the previous hidden state are relevant to the current context?
  - **Update gate:** Which elements of the intermediate hidden state and of the previous hidden state need to be preserved for future use?



Overall, comparing inputs and outputs for some different types of neural units....



When to use LSTMs vs. GRUs?

Why use GRUs instead of LSTMs?

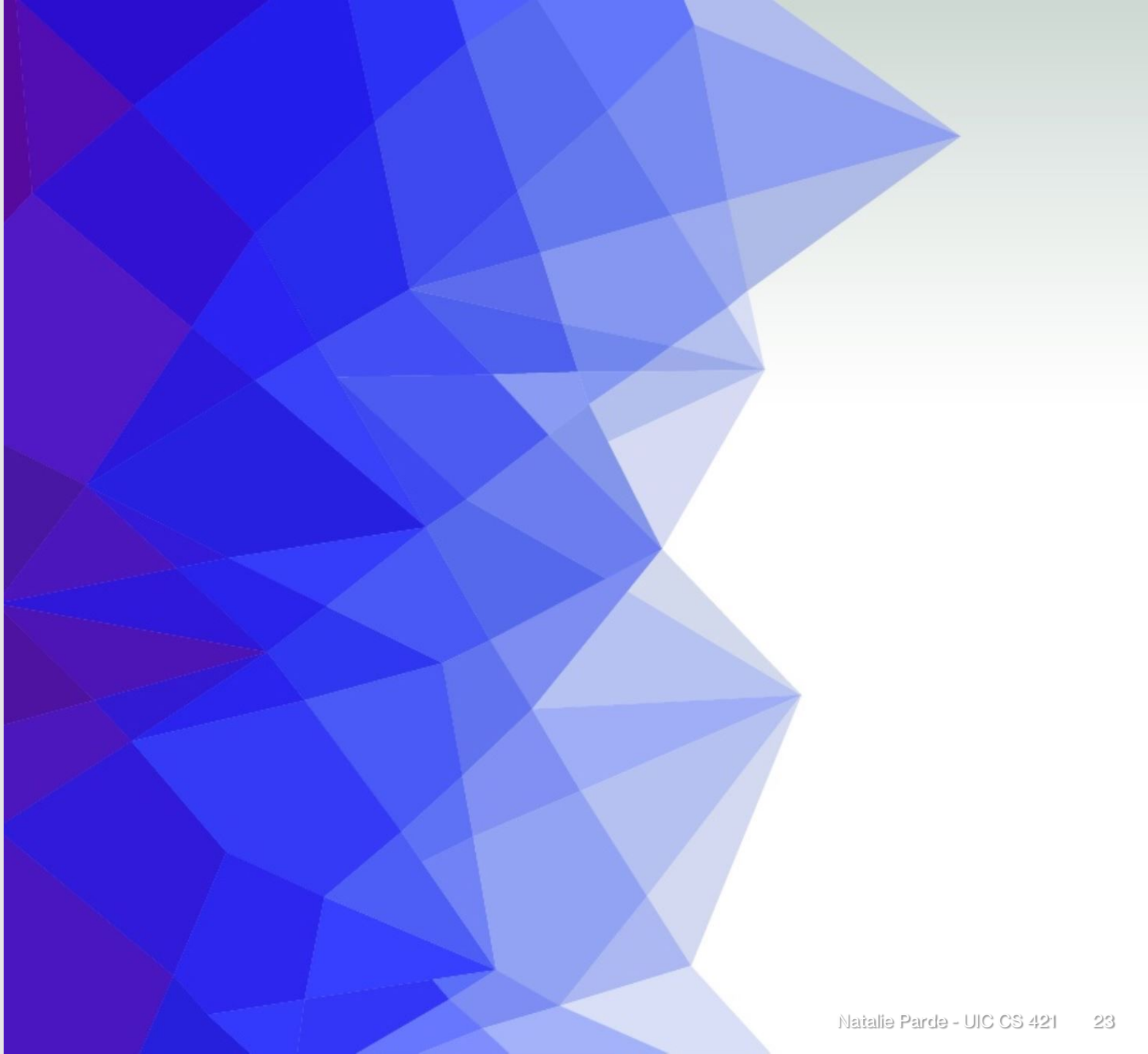
- **Computational efficiency:** Good for scenarios in which you need to train your model quickly and don't have access to high-performance computing resources

Why use LSTMs instead of GRUs?

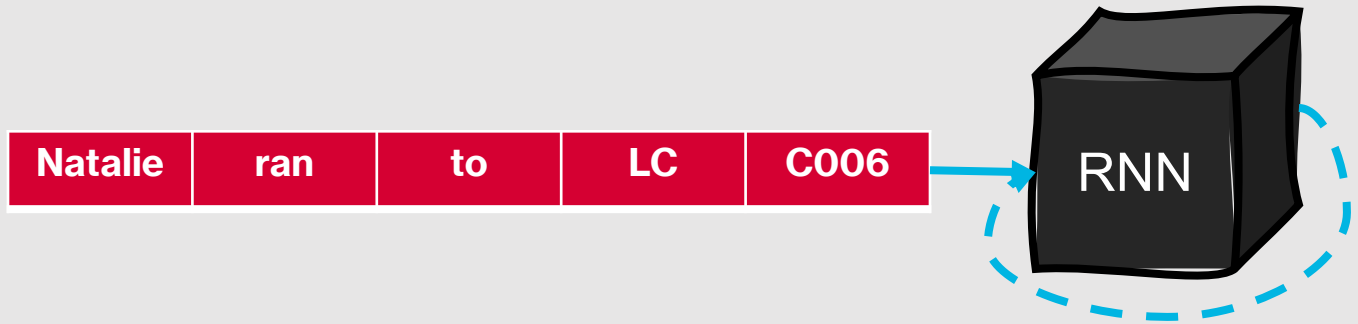
- **Performance:** LSTMs generally outperform GRUs at the same tasks

Bidirectional Models

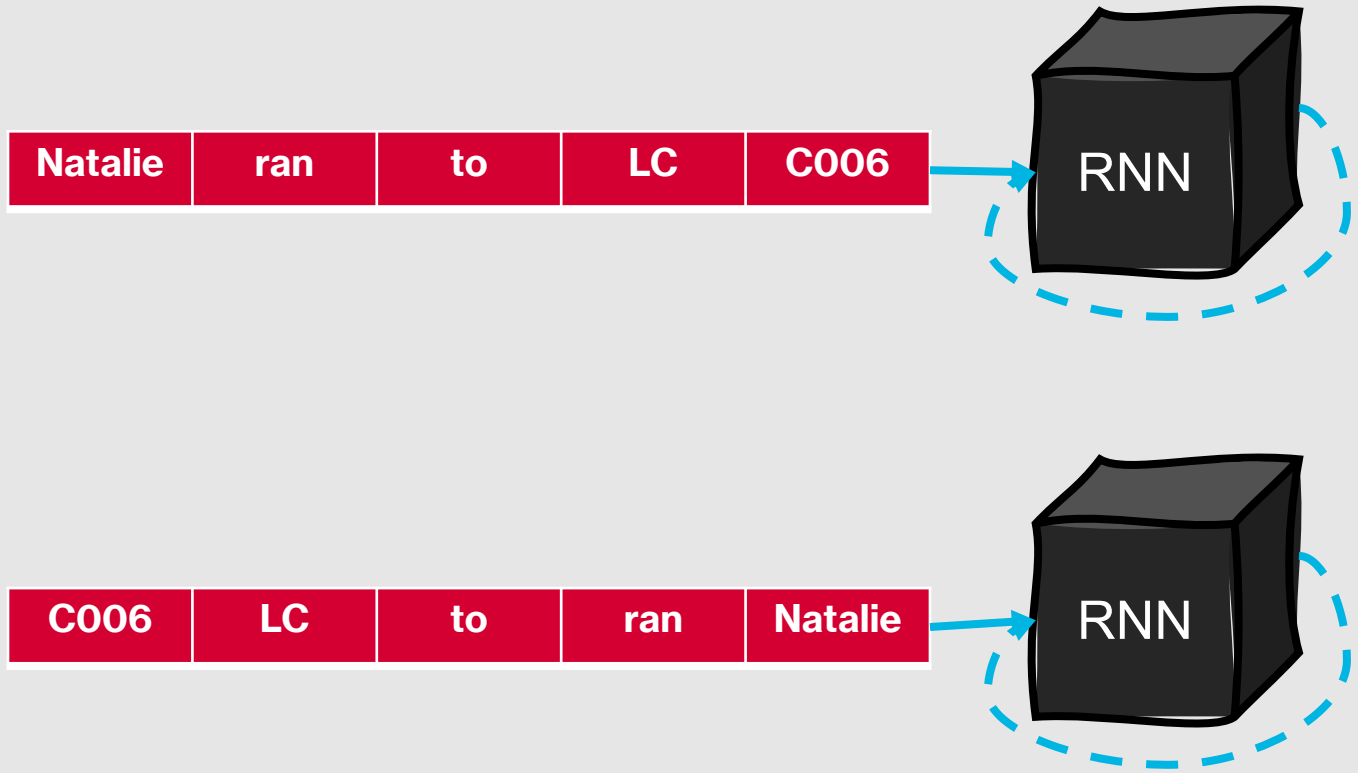
- All RNN units can be combined with one another in the same way that feedforward units can be combined
  - Layers of vanilla RNN units
  - Layers of LSTM units
  - Layers of GRU units
- These layers can also be combined to implement **bidirectional** architectures that process input both from beginning to end and from end to beginning



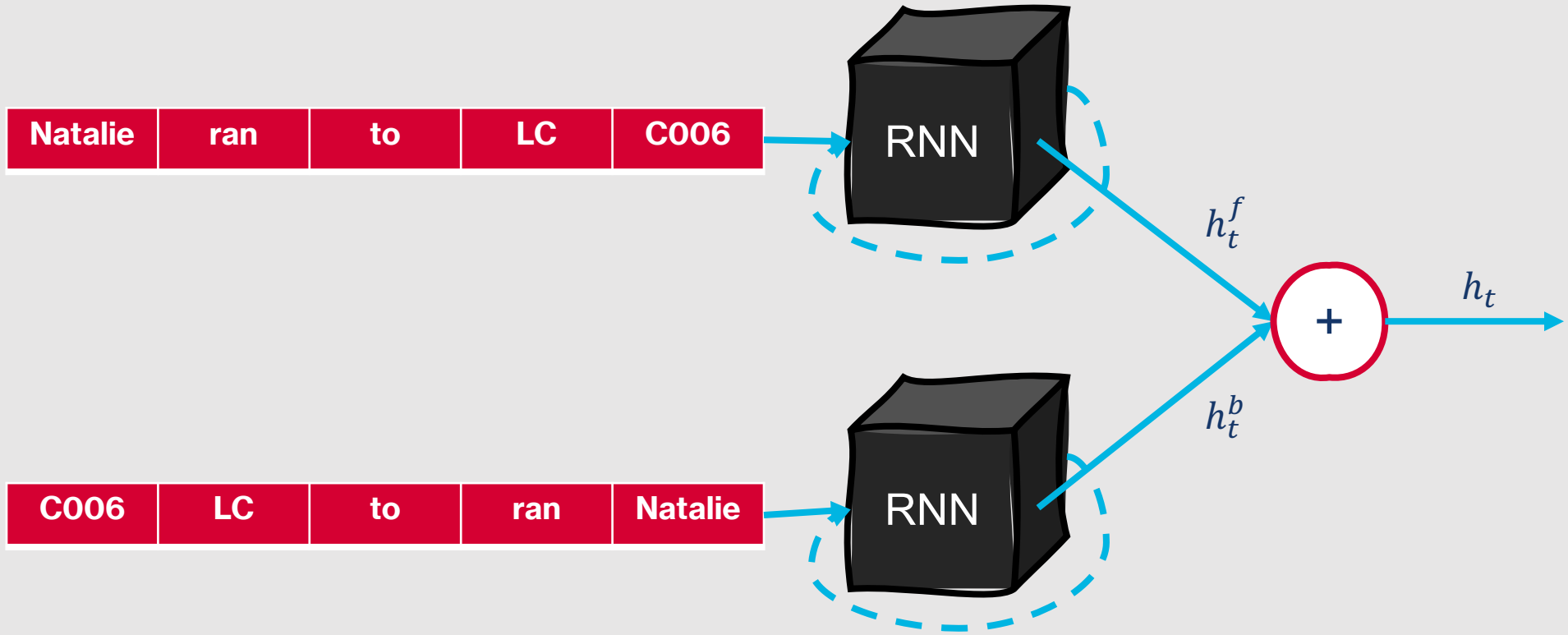
Bidirectional RNNs



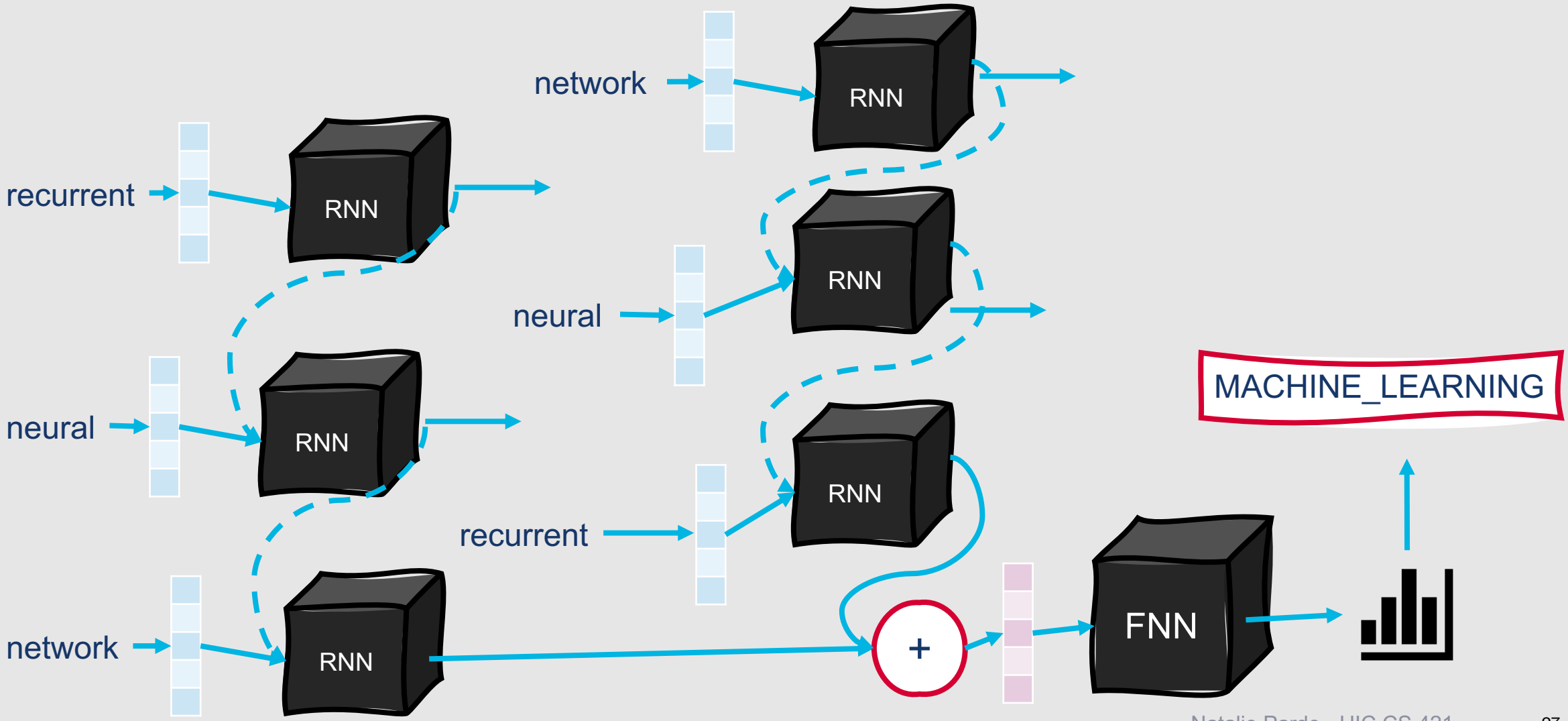
Bidirectional RNNs



Bidirectional RNNs



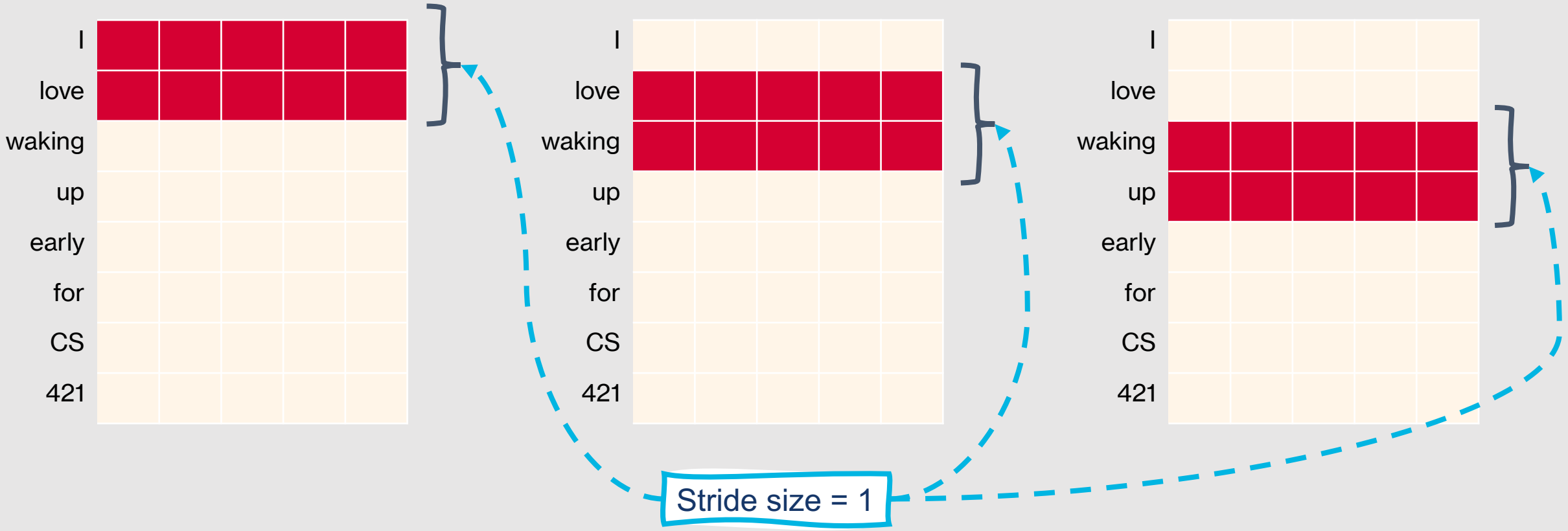
Sequence Classification with a Bidirectional RNN



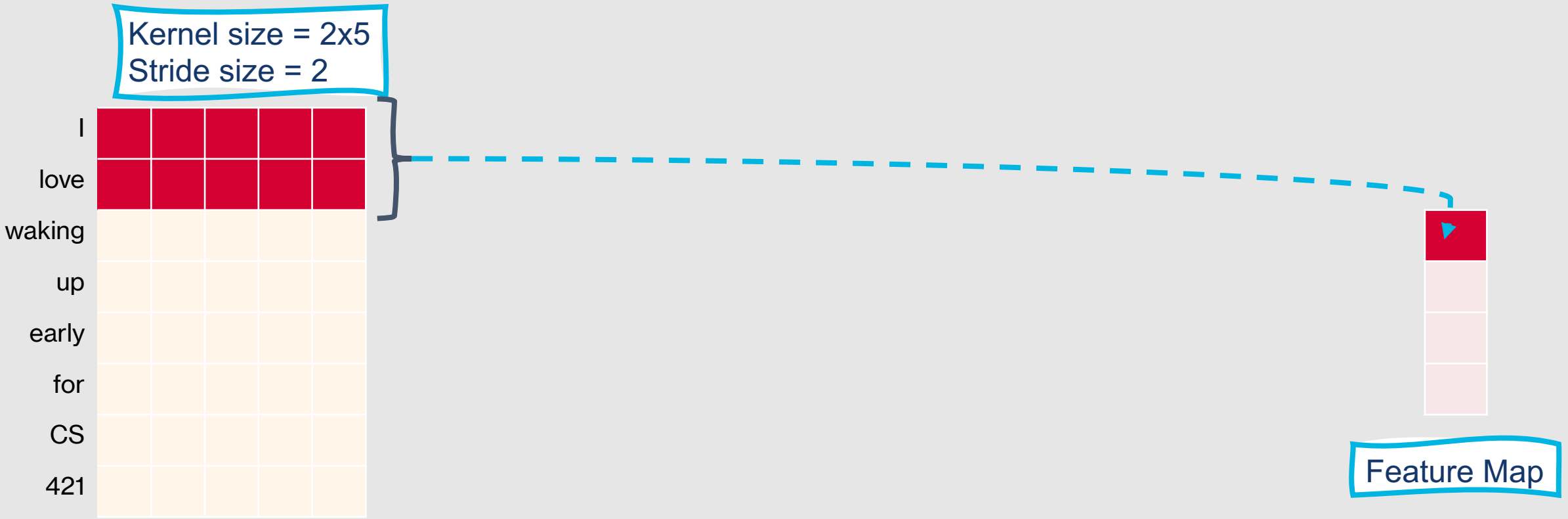
Convolutional Neural Networks (CNNs)

- General premise:
  - Deep learning models should be making decisions based on local regions of the context
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for convolutional neural networks:
  - Input to a layer is the output of **convolutional operations performed on subsets of the output** from the previous layer

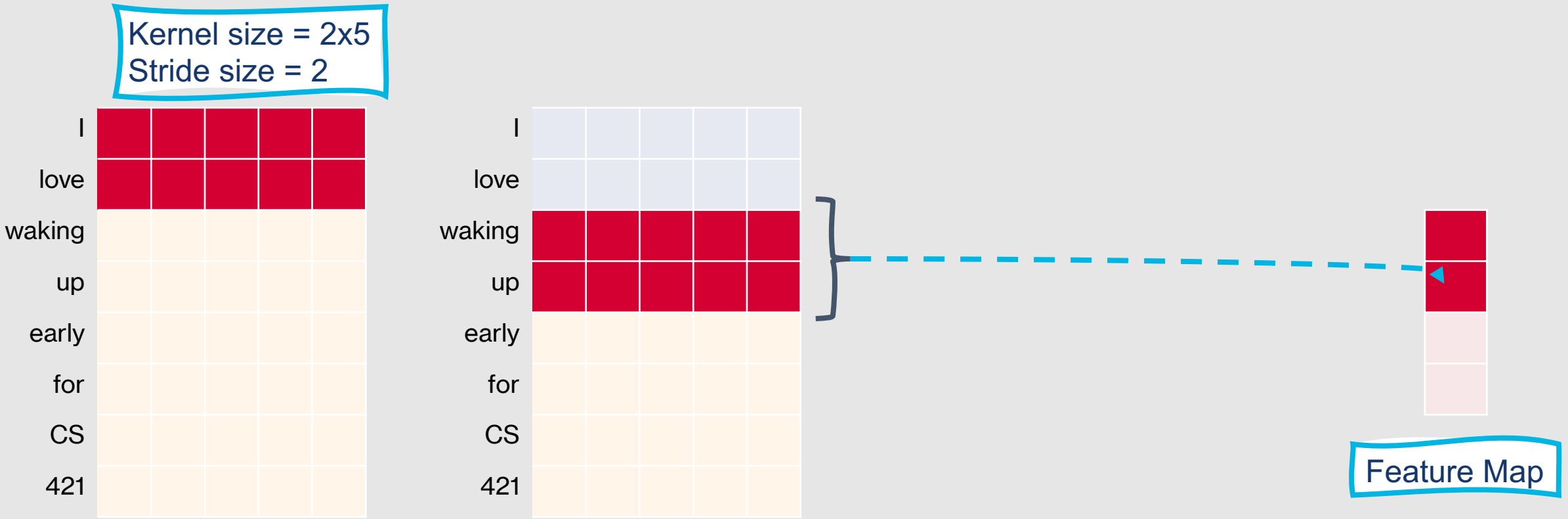
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



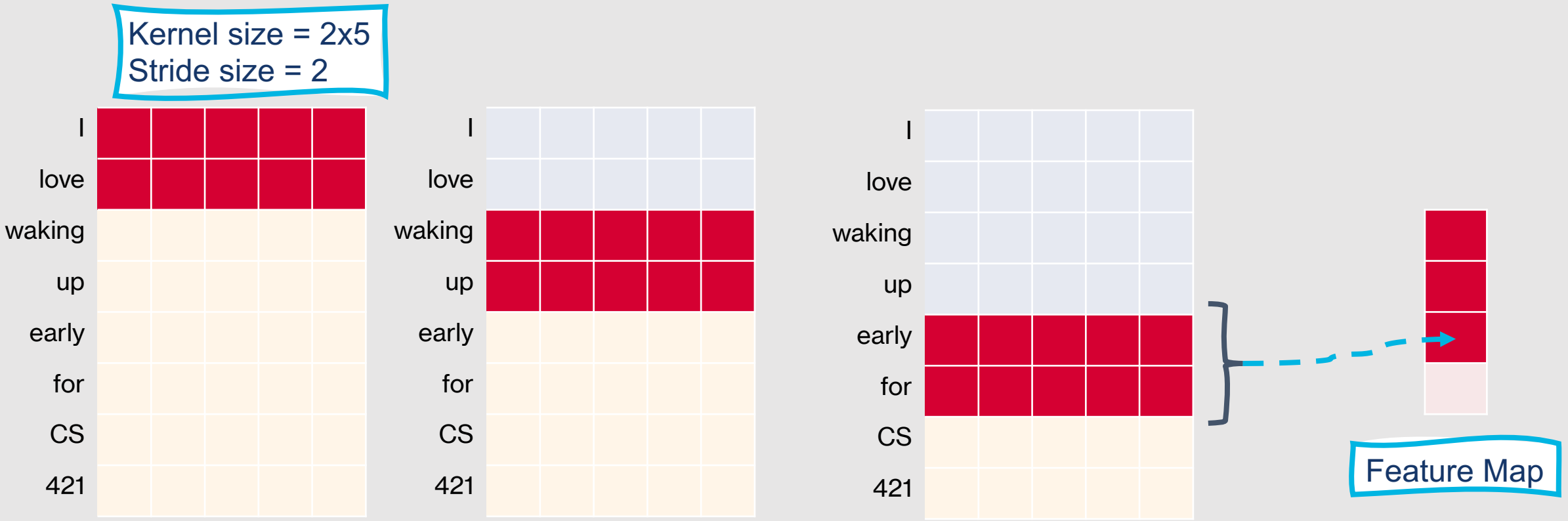
We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.



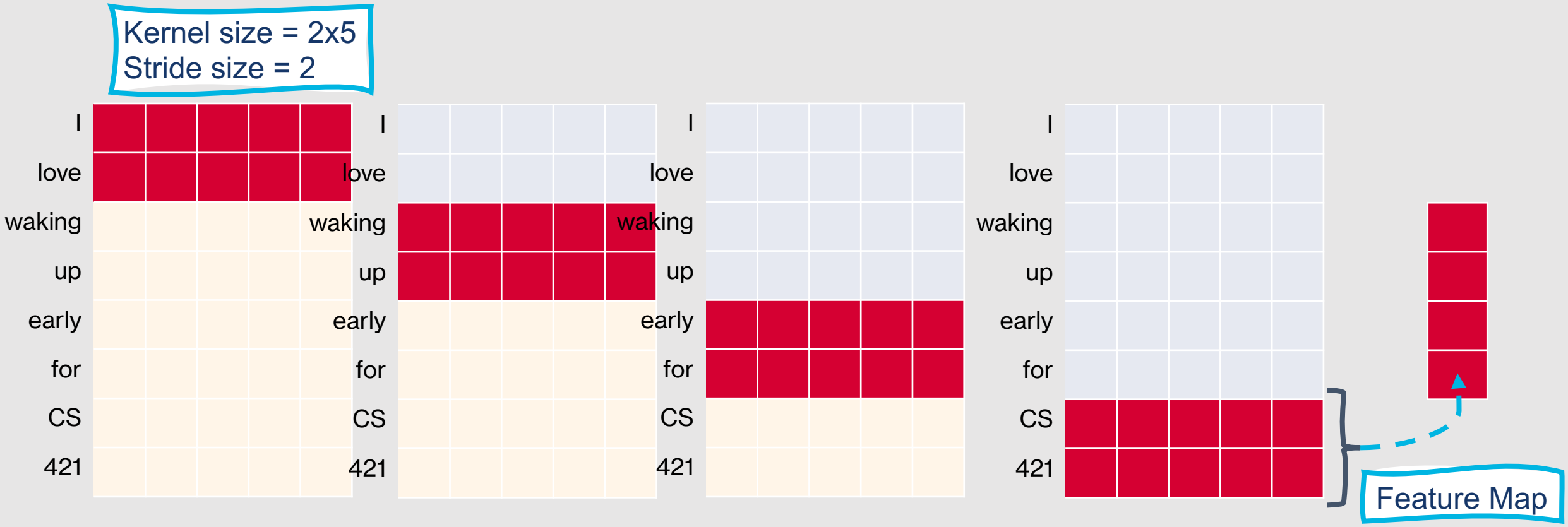
We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.



We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.

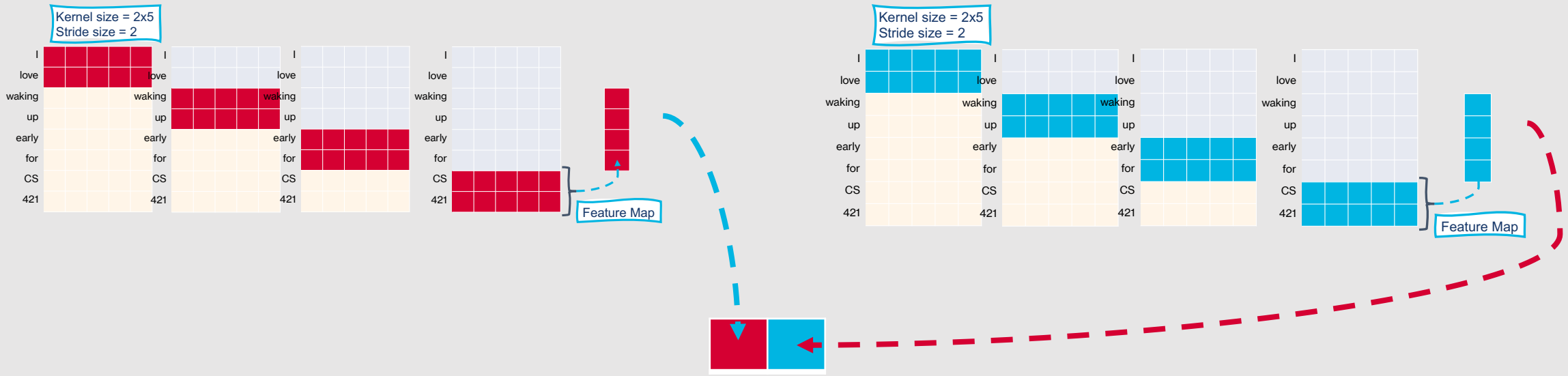


We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.

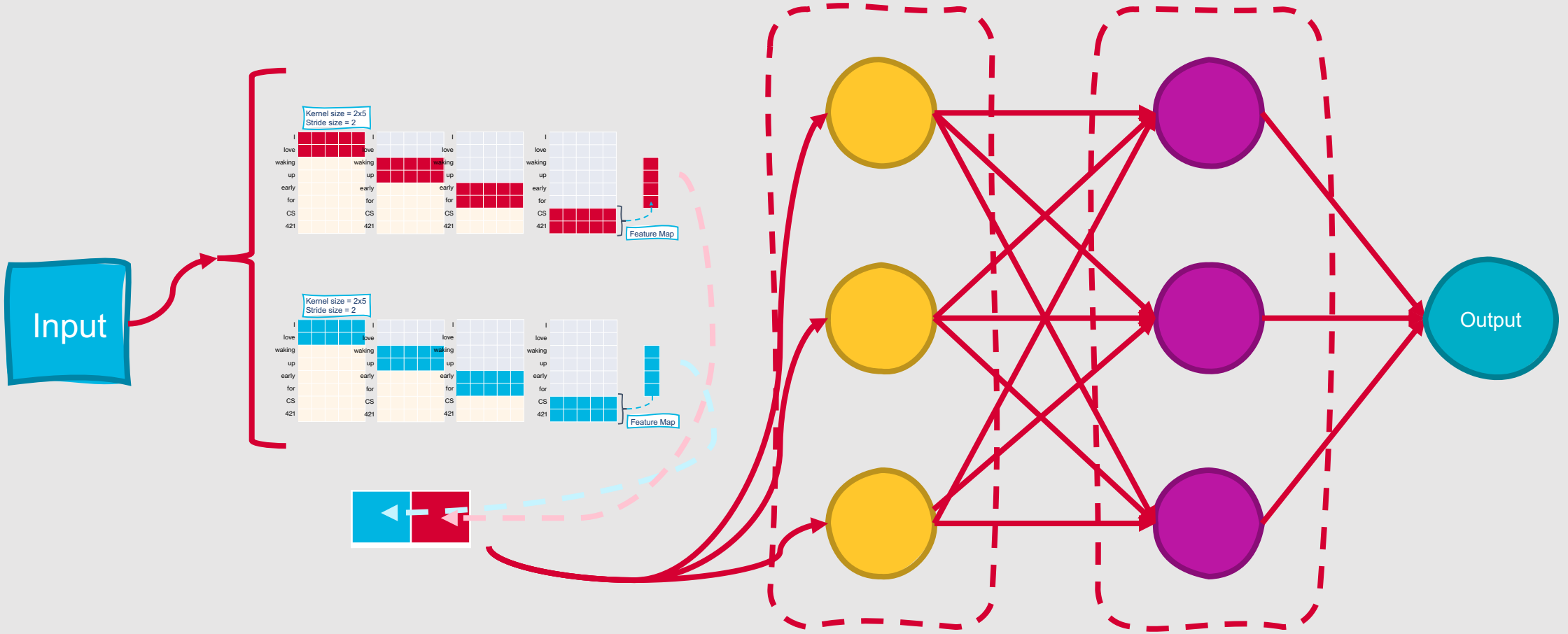


Typically, we learn multiple feature maps and then reduce the dimensionality of the learned feature maps by pooling (e.g., taking the average or maximum) subsets of their values.

- This is done to:
  - Further increase efficiency
  - Improve the model's invariance to small changes in the input



The output from pooling layers is typically then passed along as input to one or more feedforward layers.



Why use CNNs for an NLP task?

- Originally designed for image classification!
- However, offers unique advantages for NLP tasks:
  - Extracts meaningful local structures from input
  - Increases efficiency of the training process relative to feedforward neural networks

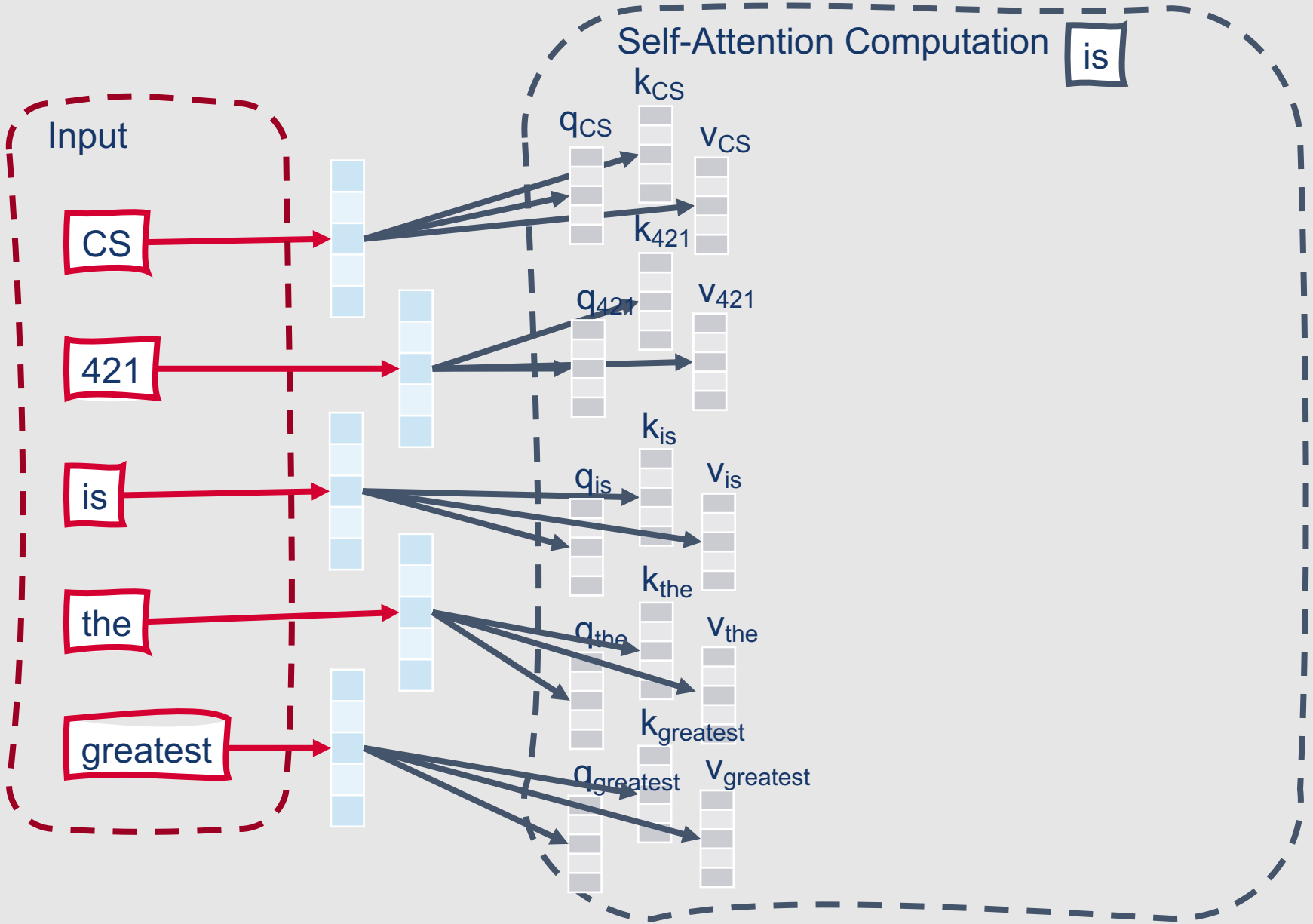
Transformers

- General premise:
  - Deep learning models don't need to wait to process items one after the other to incorporate sequential information
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for recurrent neural networks:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer + a vector of numbers representing the layer's output at the previous timestep
- Modification for Transformers:
  - Input to a feedforward layer is the output from a **self-attention layer** computed over the entire input sequence, indicating which words in the sequence are most important to one another

Self-Attention

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

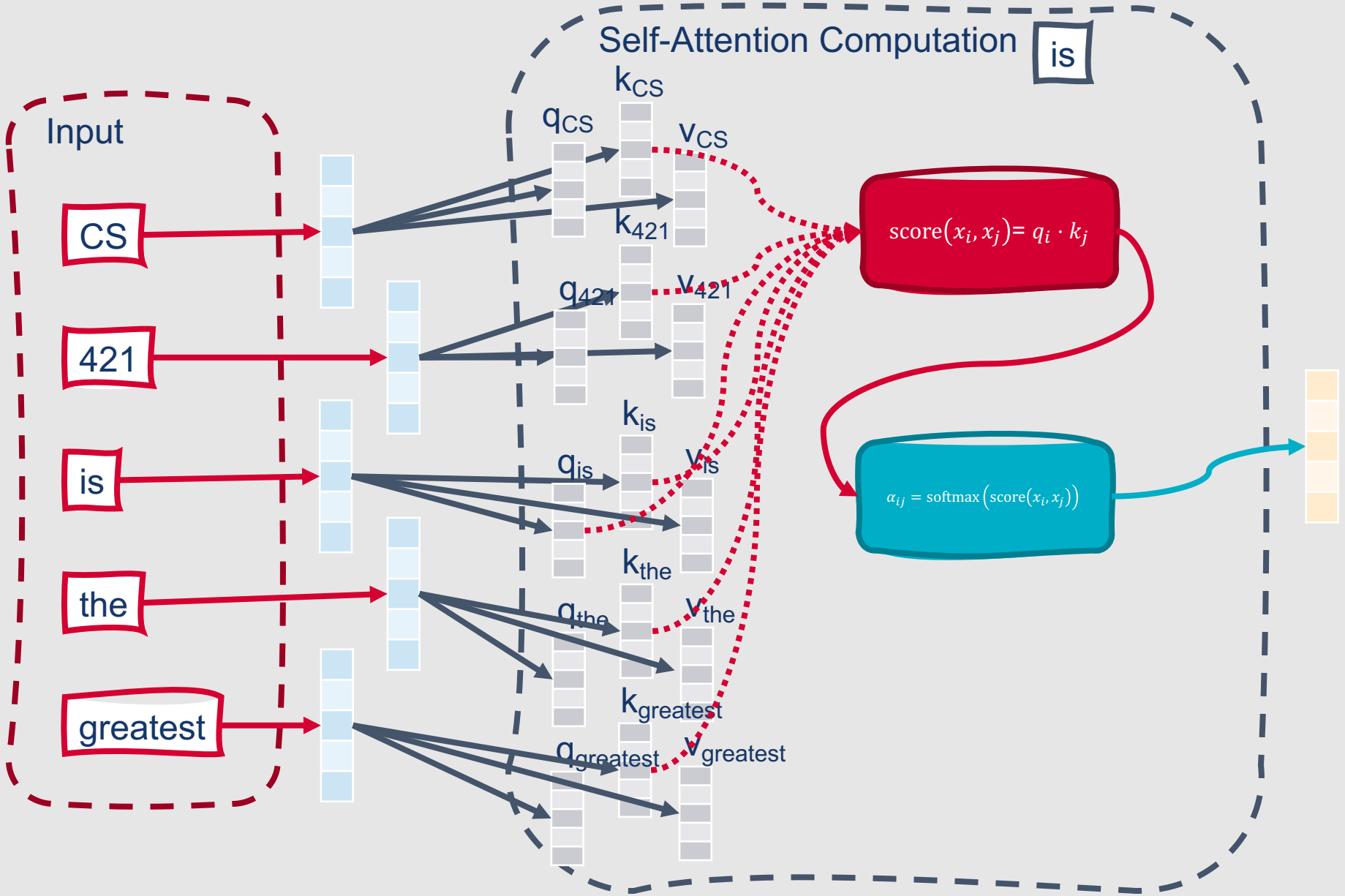
Bidirectional Self-Attention Layer



Self-Attention

1. Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
   - $\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i$
   - $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i$
   - $\mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i$
2. Compute attention weights α by applying a softmax activation over the element-wise comparison scores between all possible query-key pairs in the full input sequence
   - $\text{score}_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$
   - $\alpha_{ij} = \frac{\exp(\text{score}_{ij})}{\sum_{k=1}^{n} \exp(\text{score}_{ik})}$

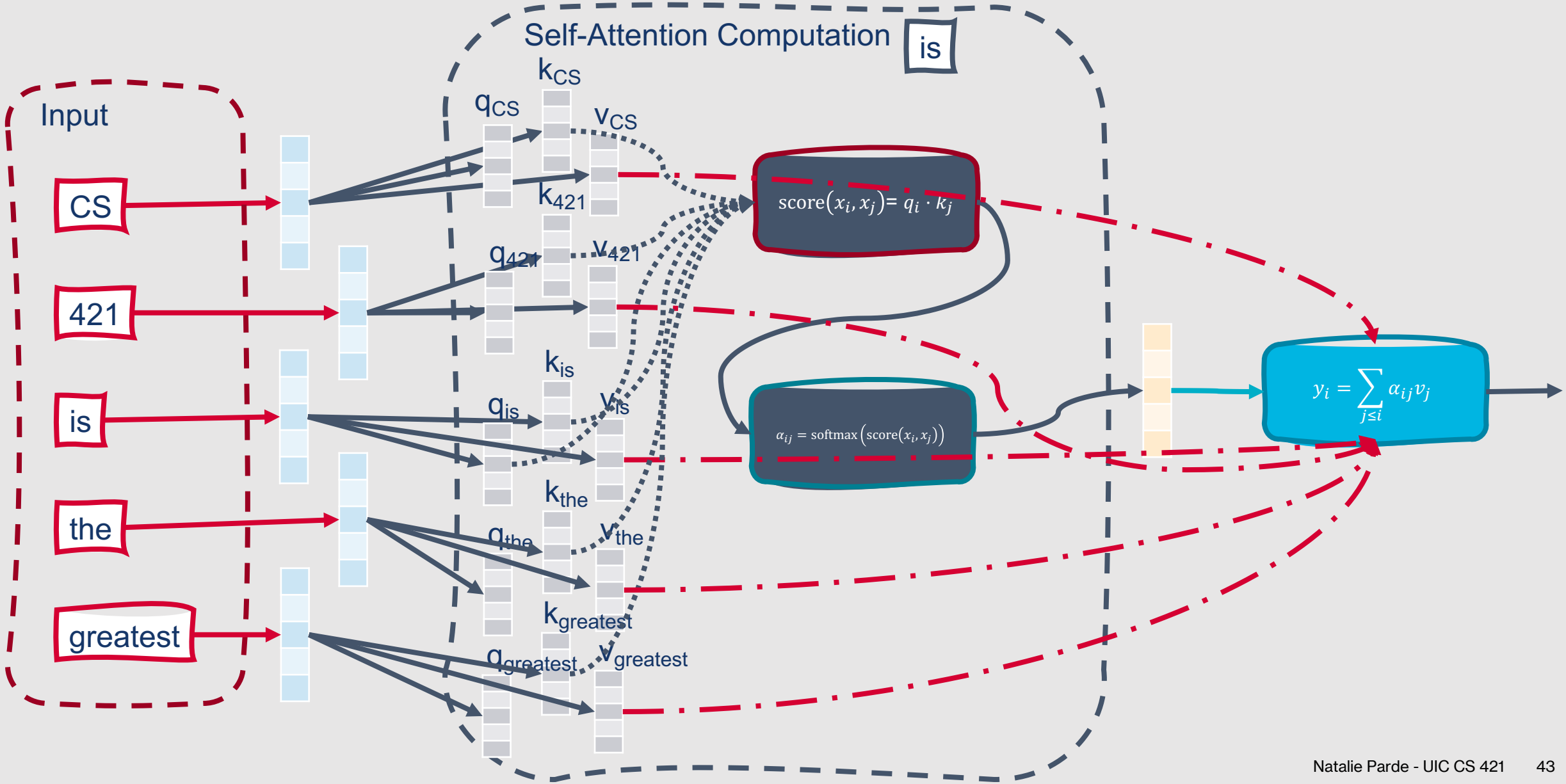
Bidirectional Self-Attention Layer



Self-Attention

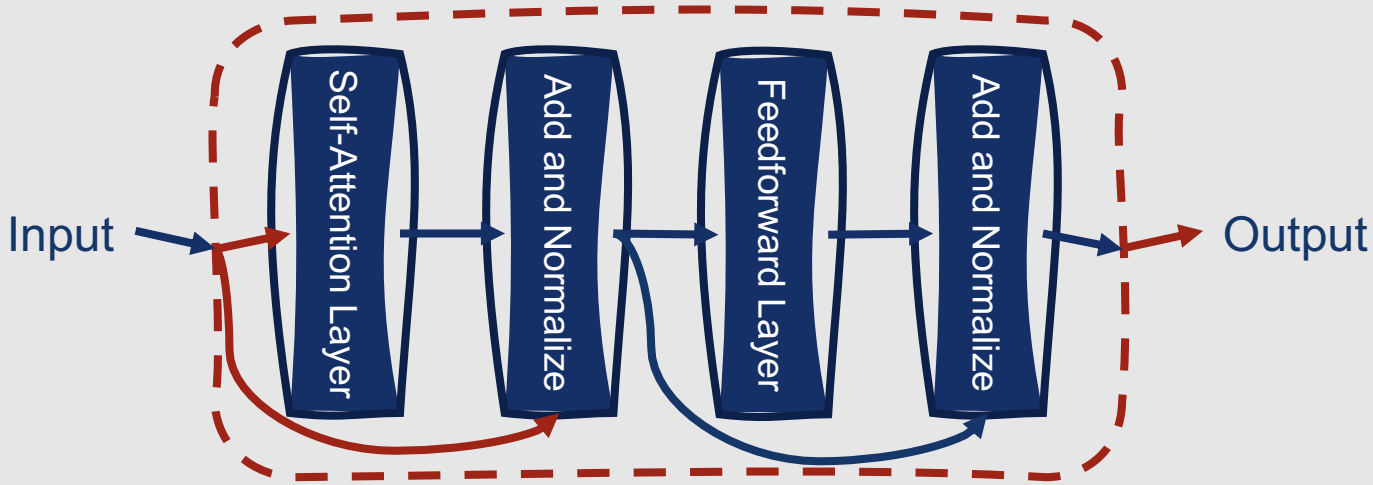
1. Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
   - $\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i$
   - $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i$
   - $\mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i$
2. Compute attention weights α by applying a softmax activation over the element-wise comparison scores between all possible query-key pairs in the full input sequence
   - $\text{score}_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$
   - $\alpha_{ij} = \frac{\exp(\text{score}_{ij})}{\sum_{k=1}^{n} \exp(\text{score}_{ik})}$
3. Compute the output vector $\mathbf{y}_i$ as the attention-weighted sum of the input value vectors $\mathbf{v}$
   - $\mathbf{y_i} = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j$

Bidirectional Self-Attention Layer

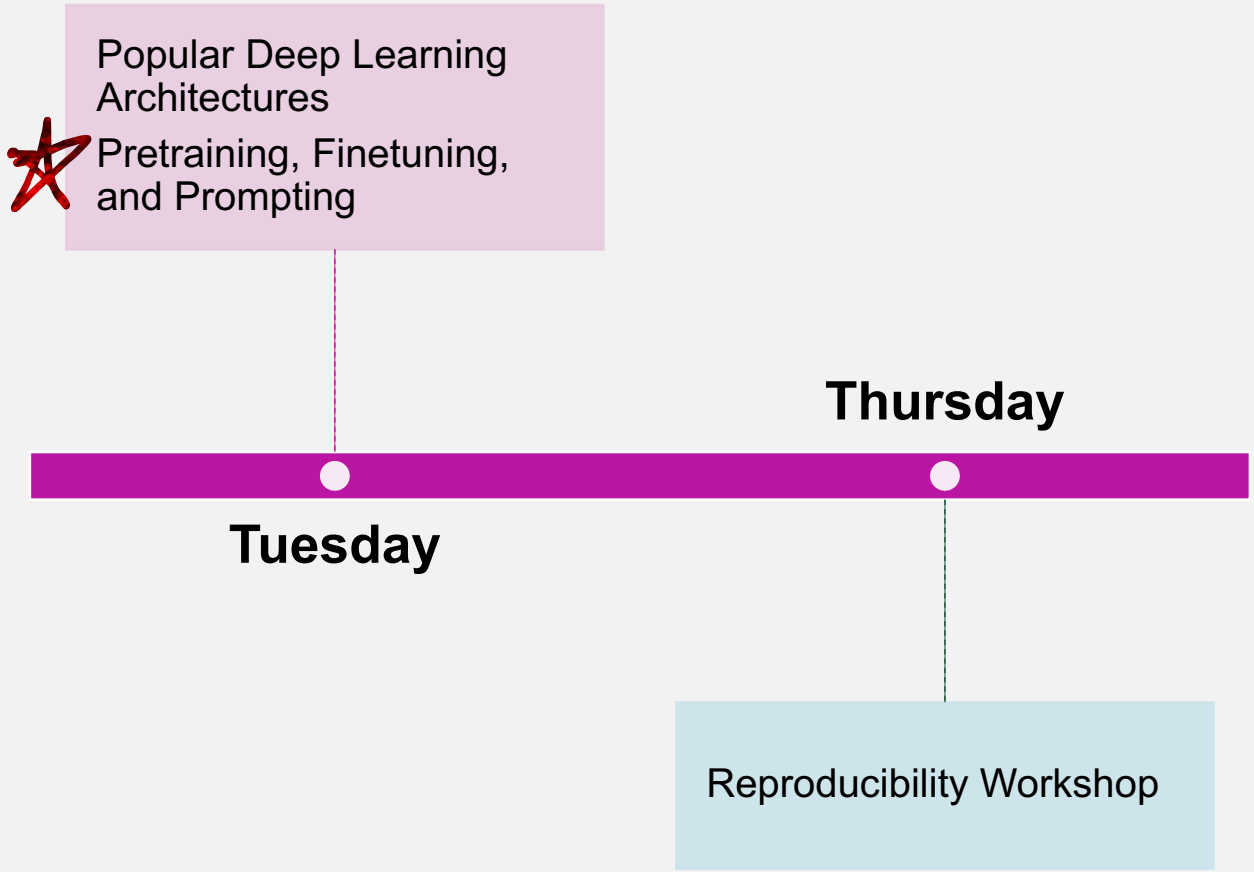


Transformer Blocks

- Transformers are implemented by stacking one or more blocks of the following layers:
  - Self-attention layer
  - Normalization layer
  - Feedforward layer
  - Another normalization layer
- Some of these layers have **residual** connections between them even though they do not immediately precede or proceed one another



This Week's Topics



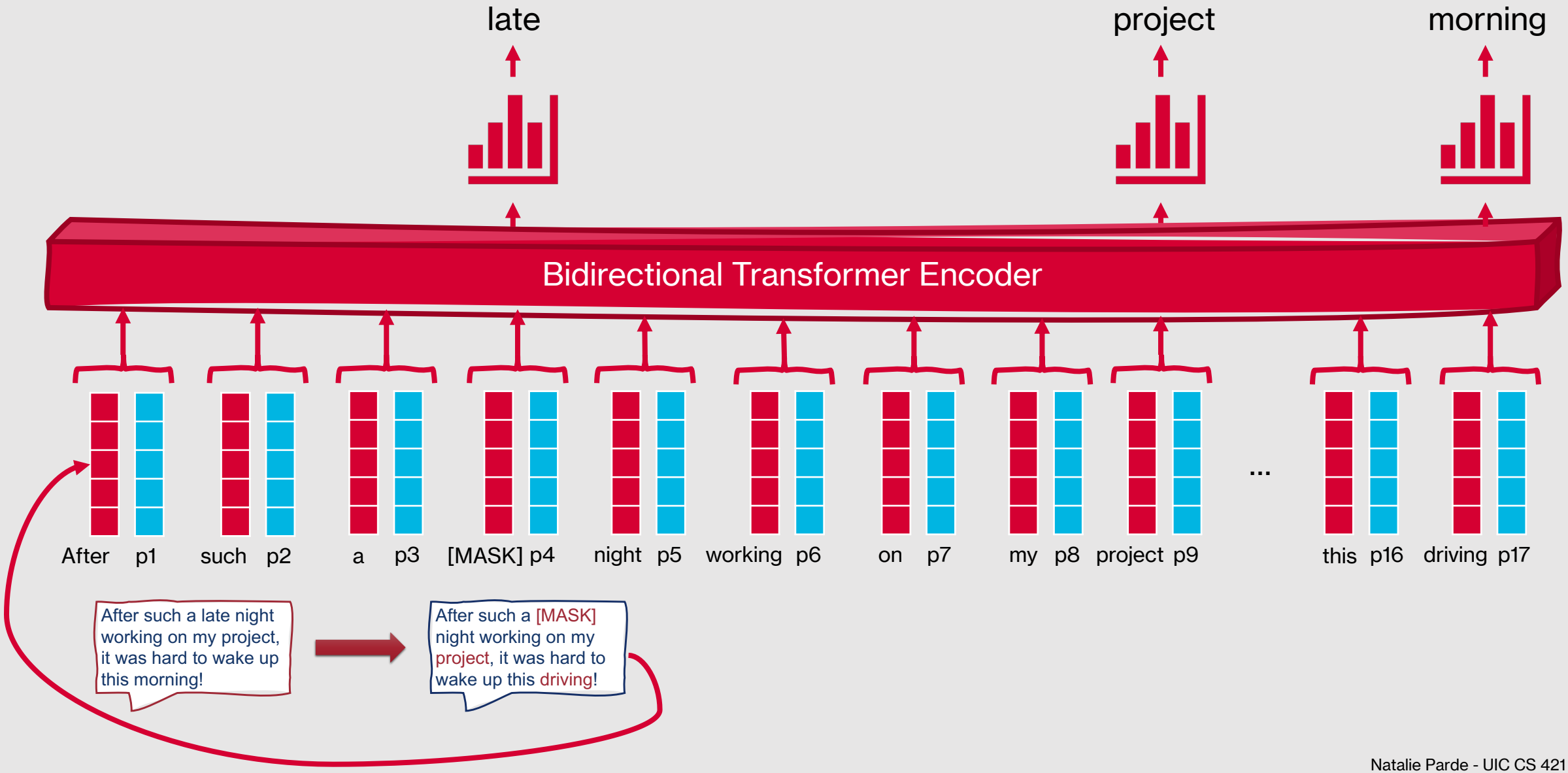
Bidirectional Encoder Representations from Transformers (BERT)

- The most popular Transformer-based architecture for NLP tasks
- Implemented using:
  - 12 Transformer blocks, each of which have 12 **attention heads** in each self-attention layer
  - 768-dimensional hidden layers
  - A subword vocabulary of 30,000 tokens
  - A fixed input length of 512 subword tokens
- Overall, this means that the model has 100,000,000 trainable parameters!

BERT is trained to perform two tasks.

- **Masked language modeling**
  - Randomly select a subset of tokens from the training input and:
    - Replace some of them with [MASK] tokens
    - Replace some of them with other randomly sampled tokens
    - Leave some of them unchanged
  - For each sampled token, try to predict what the correct token is

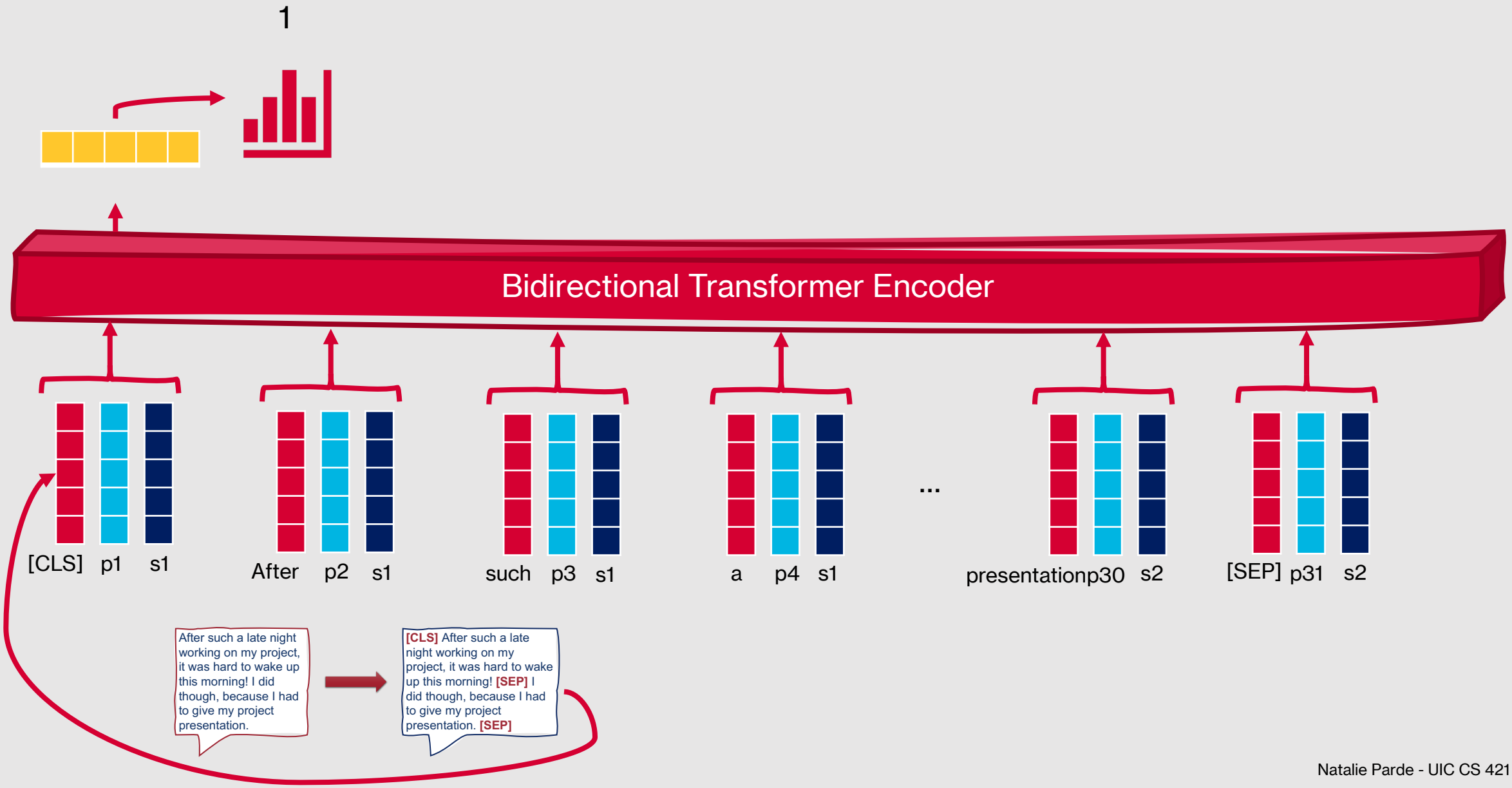
Masked Language Modeling



BERT is trained to perform two tasks.

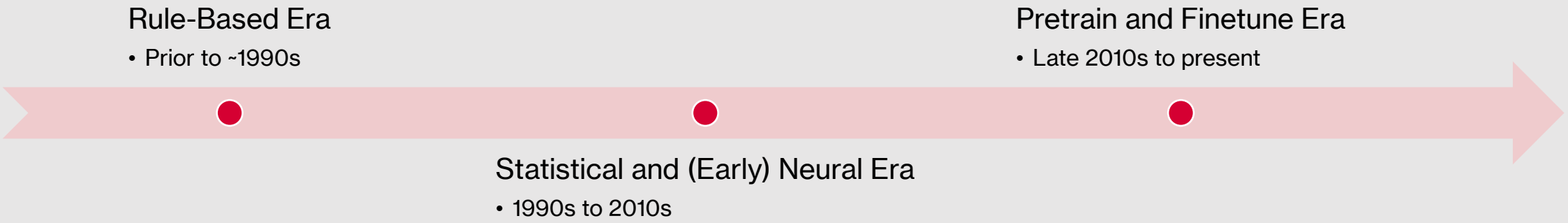
- Masked language modeling
  - Randomly select a subset of tokens from the training input and:
    - Replace some of them with [MASK] tokens
    - Replace some of them with other randomly sampled tokens
    - Leave some of them unchanged
  - For each sampled token, try to predict what the correct token is
- **Next sentence prediction**
  - Predict whether pairs of sentences are actually adjacent to one another in text
    - Prepend a [CLS] token to the pair of sentences
    - Separate the two sentences using a [SEP] token
    - Add segment embeddings to the model
  - Assign a label based on the representation learned for the [CLS] token

Next Sentence Prediction



The development of BERT was the catalyst for an important shift in contemporary NLP.

- Training BERT was very time-consuming and resource-intensive, but it produced a model that could be reused for many purposes
- Researchers began to consider task formulations in which they could finetune a pretrained model for a new purpose, rather than training a smaller model for that purpose from scratch



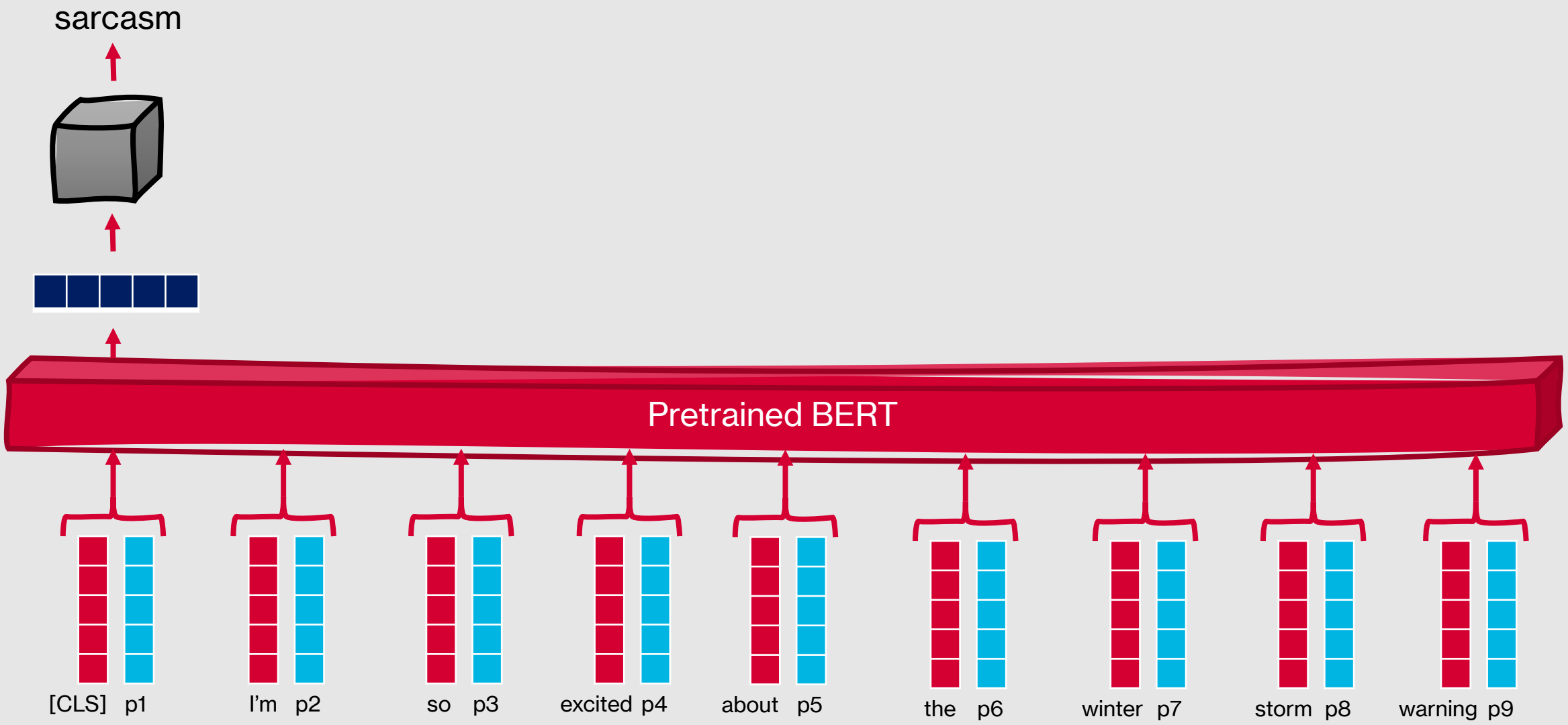
Pretrain and Finetune Paradigm

- Intuition:
  - If we take models that have been **pretrained** on massive datasets for other tasks, we can **finetune** them for our specific task while also taking advantage of the information that was learned during the pretraining process
- Popular pretrained model for this purpose: BERT

How does finetuning work?

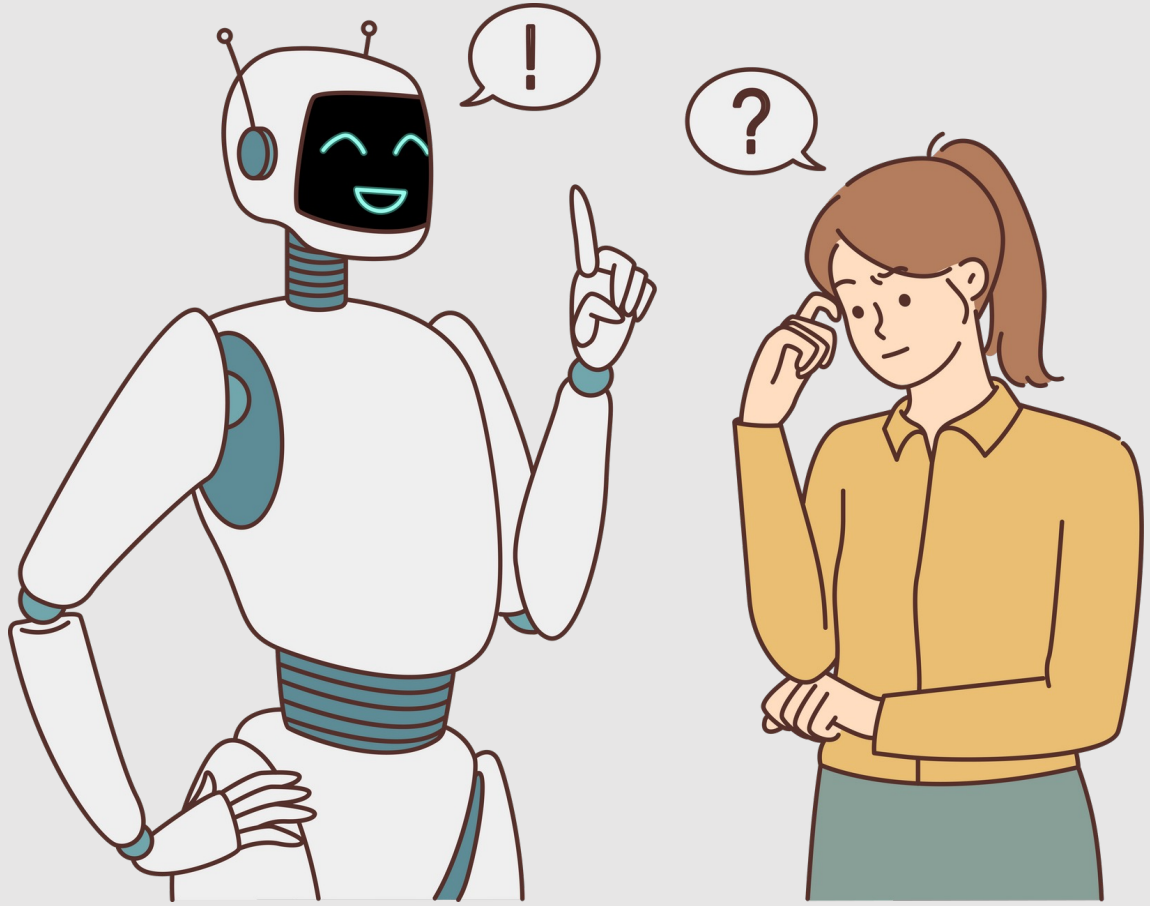
- Take a large model that has already been trained for some other task
- Add a **task head** to the model
  - Task-specific layer(s) that take the input representations from the pretrained model and produce your desired output
- Update the parameters for the task head while ignoring or only minimally adjusting the weights for the pretrained model
  - This will require that you have supervised training data for your target task

Example: Finetuned Sarcasm Detector



Why does this work?

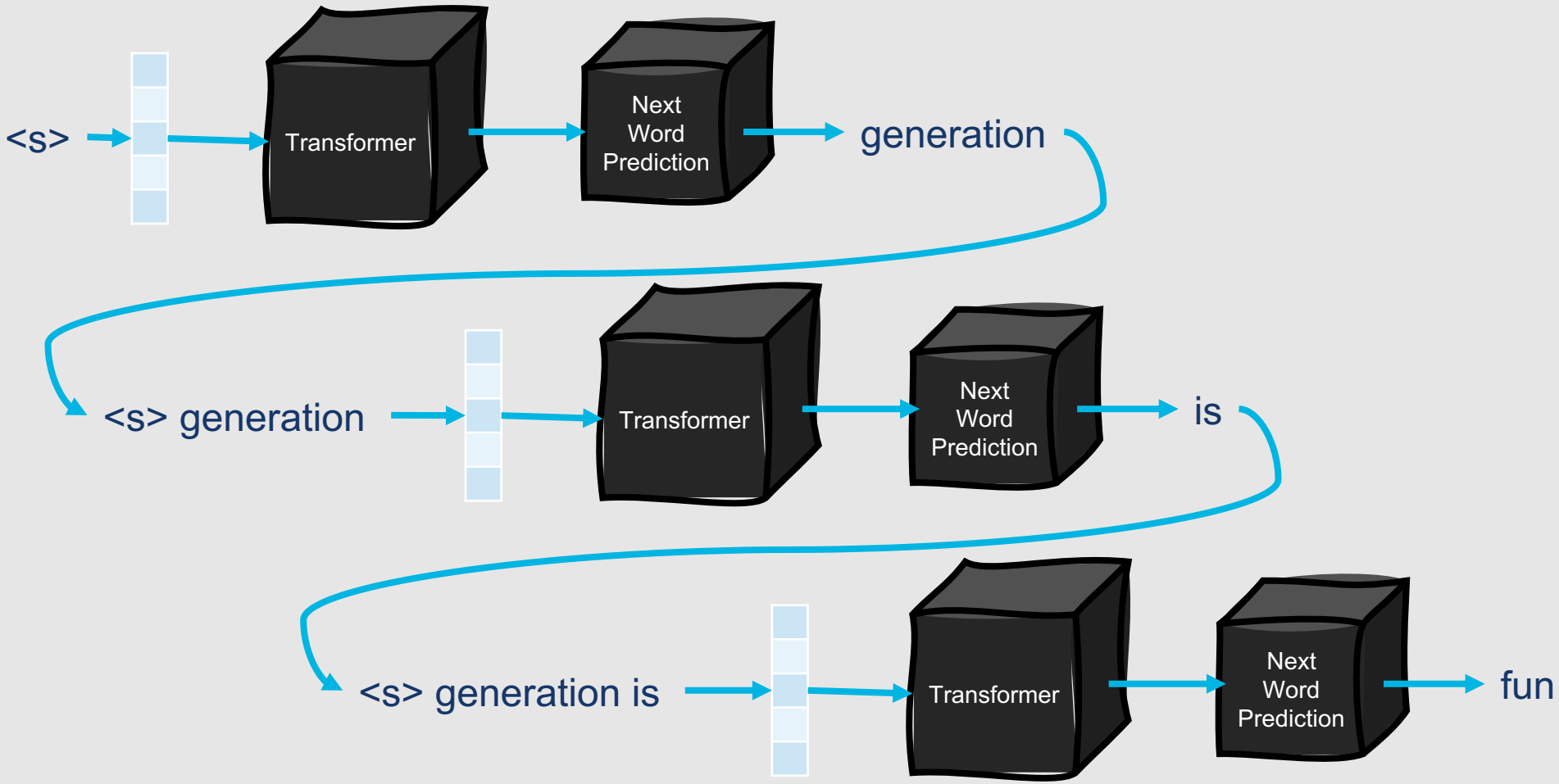
- Pretraining on large datasets allows language models to build high-quality representations facilitating **general language understanding**
- In many cases, this knowledge can be **reused** across many tasks
  - Sentences are likely to have similar structure across many language domains
  - Common sense knowledge is likely to transfer across problem settings
  - Semantic relationships often hold across tasks
- Specialized tasks often have much **less data available** than the tasks used to train large language models
- By finetuning an existing model to perform the specialized task, we can retain the useful general language information we've learned and use it to help us more **efficiently and effectively** solve our specialized task



Generative Transformers

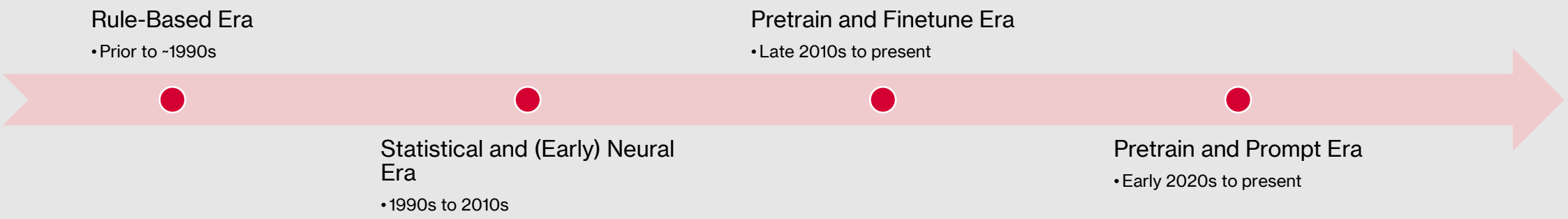
- Although we've so far examined Transformer models in settings where they are trained (and potentially later finetuned) to predict specific labels, they can also be trained for **autoregressive language modeling** purposes
  - Given the sequence of words that have been generated so far, decide which word should come next
- With autoregressive language modeling setups, we want to use causal (unidirectional) Transformers rather than bidirectional Transformers
  - Bidirectional Transformers trivialize the learning task too much
  - We want self-attention to only be computed based on what has already been generated

Autoregressive Generation



Recent advancements to generative Transformers have also ushered in another new training paradigm.

- 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

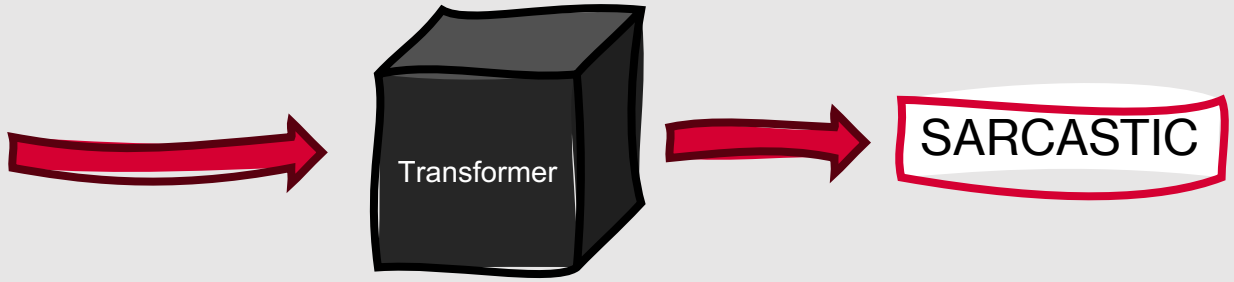


Pretrain and Prompt Paradigm

- Intuition:
  - If we take extremely large generative language models that have been pretrained on a wide variety of language data, we can **prompt** them to produce labels or output for new tasks
- Popular pretrained model for this purpose: GPT

Here are two training instances:
Data: "Natalie was soooooo happy she had booked a 5 a.m. flight." Label: SARCASTIC
Data: "Natalie loved early morning flights because she could get to her destination before brunch!" Label: NOT SARCASTIC.

Here is a test instance. Fill in the correct label:
Data: "Natalie was sooooooooooo excited to wait in an early morning airport security line." Label:



How does prompting work?

- Take a large model that has already been trained to perform generative language modeling
- Develop a set of **prompt templates** for your task
  - Prompt templates can be manually or automatically constructed
- Develop an approach for **answer engineering**
  - Build an answer space (set of possible answers that your model may generate) and map that answer space to your desired outputs
  - This can also be done manually or automatically using search techniques
- Format your input according to the relevant prompt template(s) and map the resulting language model output to your desired target output

Why is this useful?

- Successful approaches using the pretrain and prompt paradigm are able to perform **few-shot** or even **zero-shot** learning for the target task
  - Learning from few or no training examples
- This allows researchers to build models for tasks that were previously inaccessible due to extremely scarce resource availability
- Prompting also requires **limited or no parameter tuning** for the base language model, making it possible to develop classifiers more efficiently

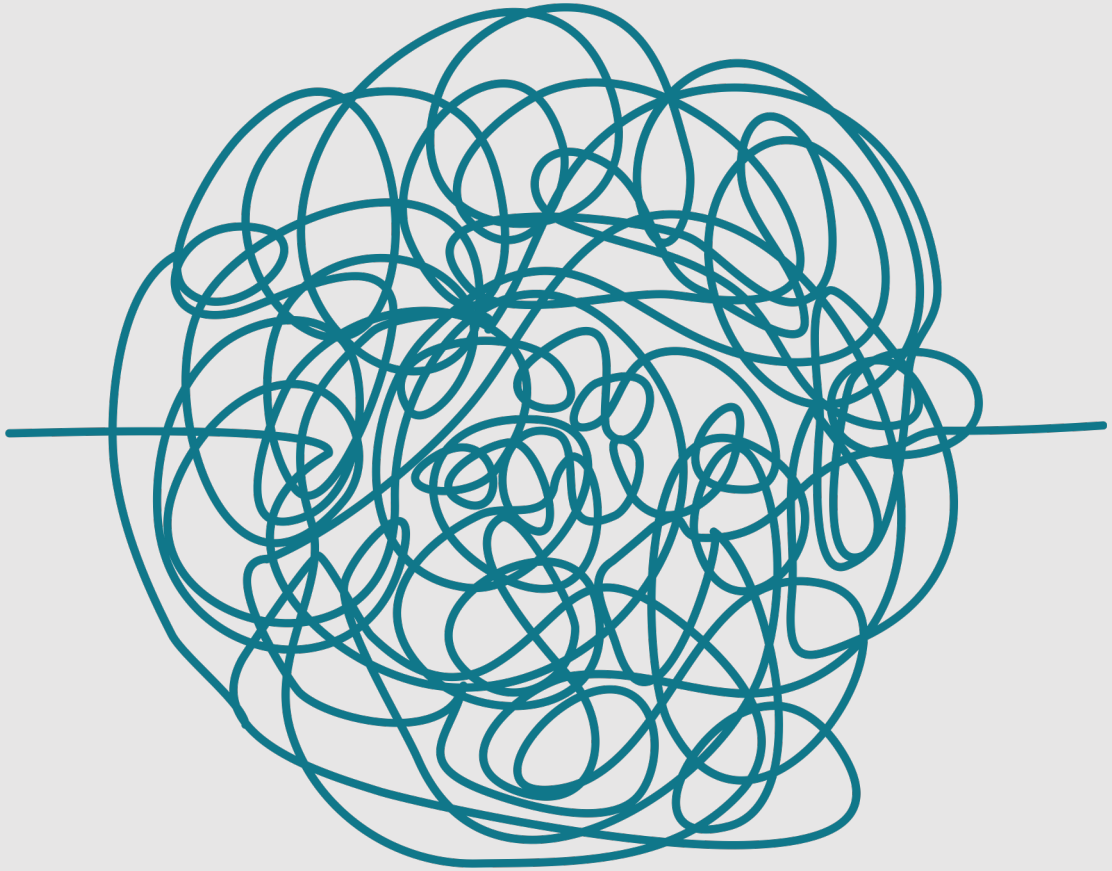
Which model should you use?

- Many approaches are now available for us to use when developing NLP systems, ranging from early rule-based techniques to very recent prompt-based methods
- In general, with each new modeling era of NLP we have sacrificed some degree of control and interpretability for increased performance

Rule-Based	Statistical	Neural End-to-End	Pretrain and Finetune	Pretrain and Prompt
• Complete control and interpretability, but very limited ability to generalize	• We have immediate access to feature values and weights and can generalize a bit more broadly, but we require supervised training data	• We no longer know our feature values or weights, but we can generalize more broadly and we know our exact inputs and outputs	• We are generalizing from a wealth of broad knowledge, although we only know specific data/task details pertaining to our target task	• We don't know exactly how or why our model is making its decisions, but we achieve strong performance and no longer require supervised training data

Remember, deep learning isn't necessarily the best solution in all scenarios!

- Less interpretable
  - Particularly important to consider when dealing with sensitive tasks (e.g., classifying health-related documents)
  - Prompt-based approaches may generate inaccurate output and present it confidently
- May overfit with very low-resource problems
- May overcomplicate the solution
  - In some cases, a naïve Bayes model may work just as well as a complex deep learning approach



Tools for Implementing Deep Learning Systems

- Pretrained Language Models
  - HuggingFace Model Hub: https://huggingface.co/models
- Deep Learning Frameworks
  - PyTorch: https://pytorch.org/
  - TensorFlow: https://www.tensorflow.org/
- Prompt Tuning Frameworks
  - OpenPrompt: https://github.com/thunlp/OpenPrompt

Summary: Overview of Deep Learning for NLP

- Many different forms of **deep learning** are popular in modern NLP
- **Recurrent neural networks** directly encode temporal context into the network's computational units
- **Convolutional neural networks** increase efficiency by performing operations over regions of input data
- **Transformers** calculate self-attention to encode temporal context for the full input in a single step
- In many cases, we can build task-specific classifiers by **fine-tuning** large pretrained models
- Recently, researchers have also started developing new approaches to **prompt** large pretrained models for relevant output
- Although modern deep learning approaches work very well, they may sacrifice control and interpretability for performance gains
- It is important to consider your research problem and data characteristics carefully when determining how you will implement your solution