Word embeddings

IN4080

Natural Language Processing

Yves Scherrer

Vector semantics

How do we get vectors that represent the meaning of words?

- Word-document (or term-document) matrices based on co-occurrence counts
  - Count weighting with tf-idf
- Word-context matrices based on co-occurrence counts
  - Dimensionality reduction (SVD, LSA)
    - Makes the vectors short and dense
- An alternative approach: word2vec
    - Creates short and dense vectors directly

Vector semantics

Why short and dense vectors?

- Easier to use as feature vectors in machine learning (fewer weights to tune)
- Generalize better than explicit counts
- Capture semantic relations (synonymy, …) better

Word embeddings: Another name for short and dense word vectors

Word embeddings

- Count-based methods:
  - Count how often each word w occurs near "apricot"
- Alternative: **predict rather than count**
  - How likely is word w to show up near "apricot"?
- We don't actually care about this task, but we take the hidden representations of the network as the word embeddings
- This is an instance of **self-supervision**:
  - Words that occur near "apricot" in the corpus can act as likely answers
  - Words that never occur near "apricot" in the corpus can act as unlikely answers
  - No need for human annotation

Skip-gram with negative sampling (SGNS)

- One of two options proposed by word2vec
- Train a model for a binary classification task:
  - Input: one-hot vectors of two words w_t and w_c
  - Output: probability that w_c occurs in the context of w_t
- We need two types of training examples:
  - Word pairs with high probability (positive examples)
  - Word pairs with low probability (negative examples)
- Train a binary classifier on this data
  - We use a type of logistic regression classifier

Approach

Predict if candidate word w_c is a neighbor of w_t:

1. Treat the target word w_t and a neighboring context word w_c as **positive examples**
2. Randomly sample other words in the lexicon to get **negative examples**
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the learned weights as the embeddings
   - We're not so much interested in the classification performance itself

Positive and negative examples

... lemon, a tablespoon of apricot jam, a pinch of ...

c1 c2 t c3 c4

- Positive examples:
  - Directly extracted from the text
- Negative examples:
  - Any pair of words that never occurs as positive example

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Skip-gram classifier

Goal: train a classifier that:

- Given a candidate pair $\langle w_t, w_c \rangle$
  - $\langle \text{apricot}, \text{jam} \rangle$
  - $\langle \text{apricot}, \text{aardvark} \rangle$
- Assigns it a probability:
  - $P(+|w_t, w_c)$
  - $P(-|w_t, w_c) = 1 - P(+|w_t, w_c)$
- We expect the probability to be high if w_t and w_c are similar
  - If w_t and w_c are encoded in vectors, we can use cosine similarity…

Computing context probabilities

- Given two vectors $\boldsymbol{w_t}$ and $\boldsymbol{w_c}$, we can compute their similarity using the dot product: $\boldsymbol{w_t} \cdot \boldsymbol{w_c}$
  - Note: cosine similarity is just a normalized dot product, and we don't need this type of normalization here.
- The dot product gives us any number. How can we convert it into a probability?
  - Sigmoid function (binary logistic regression):

$$P(+|\boldsymbol{w_t}, \boldsymbol{w_c}) = \sigma(\boldsymbol{w_t} \cdot \boldsymbol{w_c}) = \frac{1}{1 + e^{-(\boldsymbol{w_t} \cdot \boldsymbol{w_c})}}$$

Computing context probabilities

- This is for one context word:

$$P(+|\boldsymbol{w_t}, \boldsymbol{w_c}) = \sigma(\boldsymbol{w_t} \cdot \boldsymbol{w_c}) = \frac{1}{1 + e^{-(\boldsymbol{w_t} \cdot \boldsymbol{w_c})}}$$

- But we have many of them! We assume independence and just multiply them:

$$P(+|\boldsymbol{w_t}, \boldsymbol{w_{c1}}, \ldots, \boldsymbol{w_{cn}}) = \prod_{i=1}^{n} \sigma(\boldsymbol{w_t} \cdot \boldsymbol{w_{ci}})$$

But we don't have these vectors!
Wasn't the whole point to create them???

Summary

- We want to train a probabilistic classifier that estimates the probability that a target word w_t occurs in the context window containing words $w_{c1} \ldots w_{cn}$
- This probability is estimated from the similarity of $\boldsymbol{w_t}$ (embedding vector) and $\boldsymbol{w_{ci}}$ (embedding vector).
- To compute this, we “just” need embedding vectors for all the words.

Training data

... lemon, a tablespoon of apricot jam, a pinch of ...

c1 c2 t c3 c4

- For each positive example, we create k negative examples, sampling by frequency
  - Use any random word that isn't w_t

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

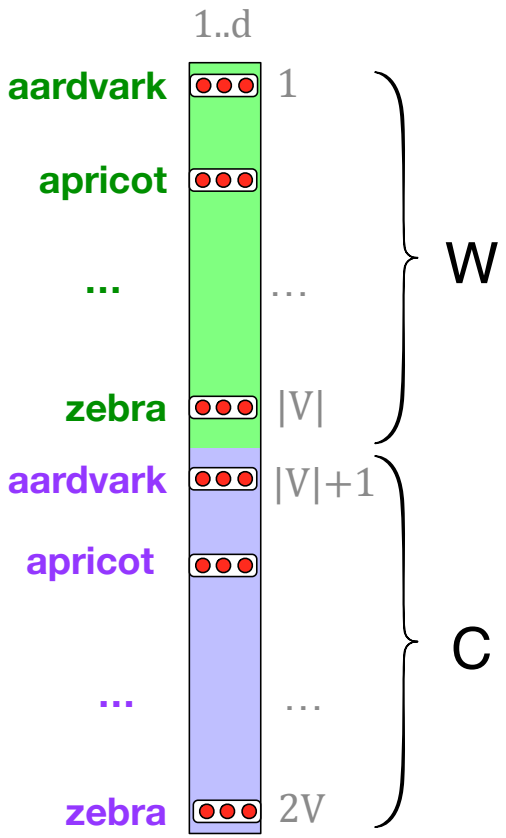
w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Training the classifier

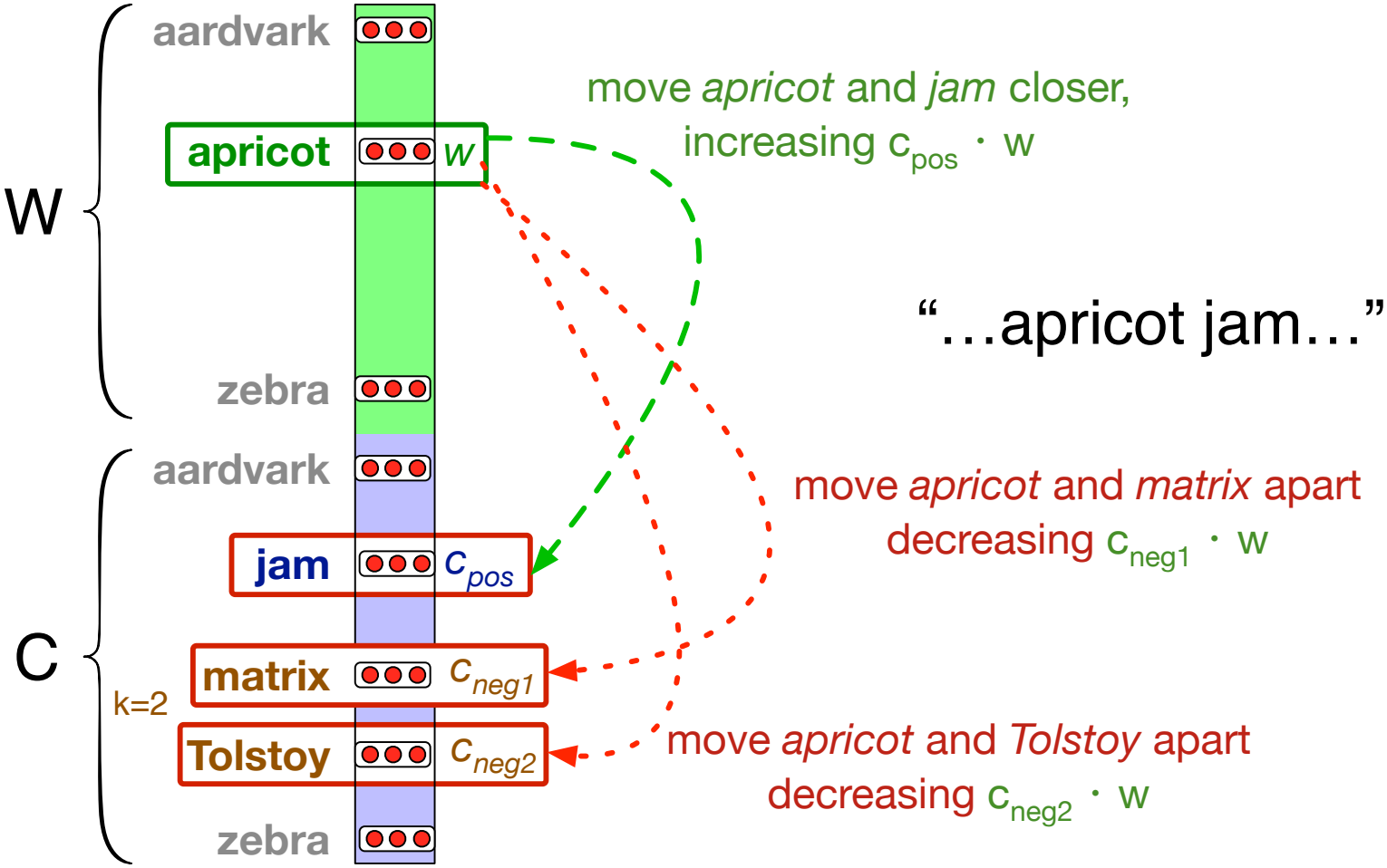
- Given a set of positive and negative training instances, and an initial set of embedding vectors
  - Initialize all embedding vectors randomly
- Adjust those word vectors such that we:
  - Maximize the similarity of the $\langle w_t, w_{c+} \rangle$ pairs drawn from the positive data
  - Minimize the similarity of the $\langle w_t, w_{c-} \rangle$ pairs drawn from the negative data
- When finished, throw away the classifier code and keep the embeddings ☺

Training procedure

- Randomly initialize two vectors for each word of the dataset
  - One for its use as target word ($\boldsymbol{W}$)
  - One for its use as context word ($\boldsymbol{C}$)
- Pick an example from the training data (positive or negative)
- Predict probability
- Compute loss and update vectors based on gradient
- Pick next example



Training word embeddings



Word embeddings

SGNS learns **two sets of embeddings**:

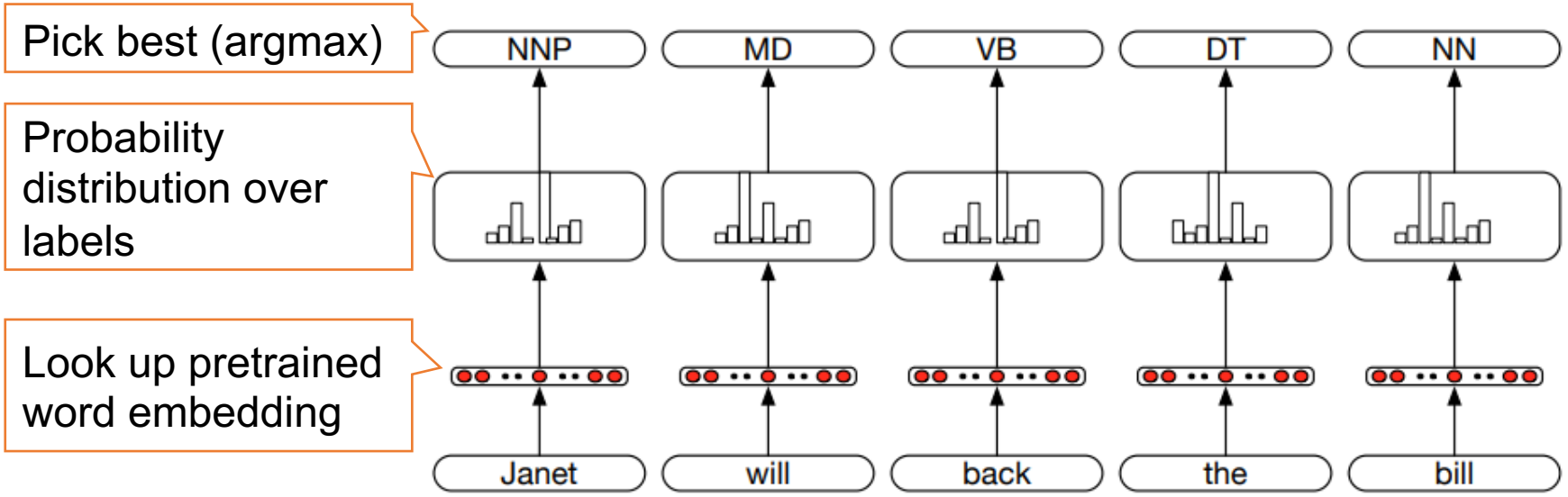
- Embeddings for the target words w_t
- Embeddings for the context words w_c

- It is common to just add them together, so word t is represented by the vector $\boldsymbol{w_t + w_c}$
  - That's what is commonly referred to by **word2vec**

Use cases for word embeddings:

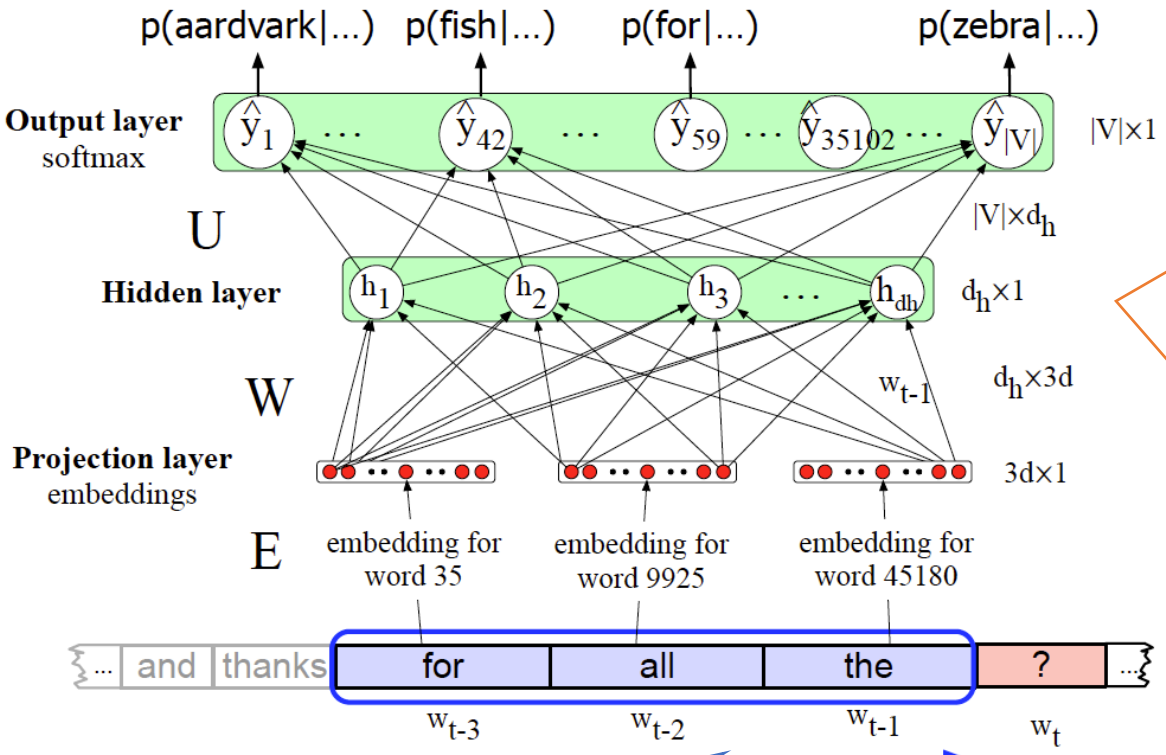
- Sequence labelling
- Language modelling
- Document classification

Sequence labeling with embeddings




```
said 0.0552 -0.0181 -0.0561 0.0086 -0.0203 -0.1505 -0.0029 0.0232 -0.0645 -0.2647
more -0.0588 -0.0566 -0.0397 0.0094 0.0356 0.0346 0.0027 -0.0144 0.0238 -0.0855
page -0.0702 0.0289 0.0413 0.0366 0.1205 -0.1255 -0.0043 0.0970 0.1616 -0.0868
he 0.0468 -0.0502 0.0332 -0.1099 -0.0402 -0.0179 -0.0137 -0.0941 0.0077 0.0397
your -0.1001 -0.0906 -0.0496 -0.0332 0.0295 -0.1788 0.0294 -0.1279 0.0420 0.1440
will -0.0368 0.0231 -0.1271 -0.1208 -0.0330 -0.0648 -0.0633 -0.0936 0.0096 -0.0760
its -0.0665 -0.0693 0.0478 -0.0544 -0.0050 0.0355 0.0339 0.0049 0.0325 -0.1453
so -0.0154 -0.0020 -0.0725 -0.0519 -0.0176 -0.0549 -0.0281 0.0256 0.0891 0.0902
were 0.0995 -0.0703 -0.1248 0.0915 -0.0324 -0.0358 0.0974 0.0636 -0.0521 0.0291
had 0.1112 -0.1055 0.0189 -0.0002 0.0389 -0.0208 0.0428 0.0665 -0.0183 -0.0010
also -0.0764 0.0451 -0.0883 0.0223 -0.2671 0.0404 0.0121 -0.0349 0.0515 -0.1087
```

Neural language models

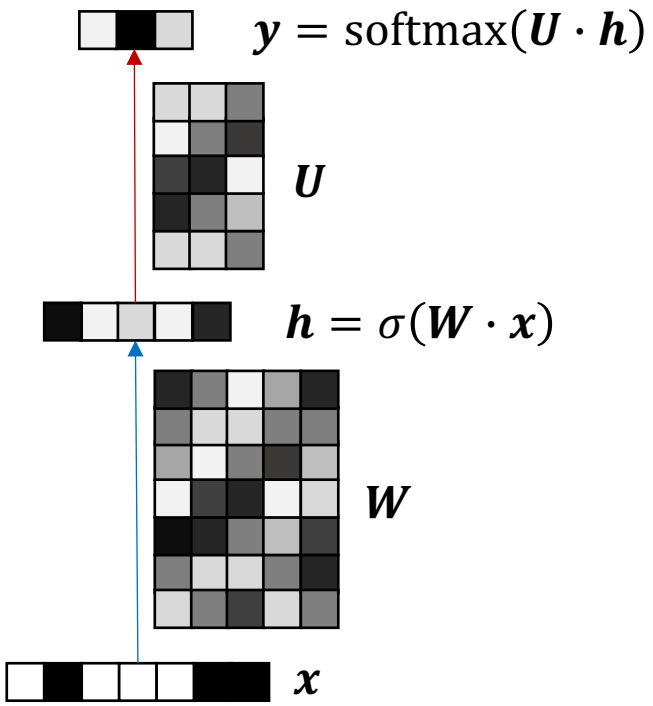


Sliding window (of length 3)

This model has serious drawbacks:

- Not efficient, need to run the same embeddings several times through the network
- Context is limited to window size

Neural document classification



The input vector needs to represent an entire document, not just one word. How can we use word embeddings here?

Neural document classification

We need document embeddings!

- **Option 1:** concatenate word embeddings
  - Problem: documents of different lengths get vectors of different lengths
  - Possible solution: padding (set values at unused positions to zero)
- **Option 2: pooling**
  - Take the mean, sum or element-wise maximum of all the word embeddings
  - Word order doesn't matter (like bag-of-words)
- More complicated options exist too…

Resources

- FastText:
  - https://fasttext.cc/docs/en/pretrained-vectors.html
  - https://fasttext.cc/docs/en/crawl-vectors.html
- Word2vec and GloVe models through Gensim:
  - https://github.com/piskvorky/gensim-data
- NLPL vector repository:
  - https://vectors.nlpl.eu/repository/

Static and contextualized word embeddings

- Word embeddings produced by LSA, SGNS etc. are **static**
  - The vectors are tied to the word type
  - They are the same for all occurrences of the word
  - They represent the average of all contexts in which the word occurs
- Other approaches like BERT produce **contextualized** word embeddings
  - The vectors are tied to the word token
  - They are different for each occurrence of a word
  - They represent the word in the specific context (sentence) in which it occurs

The Transformer

IN4080

Natural Language Processing

Yves Scherrer

Types of Transformer models

Sequence = document / sentence

- Sequence encoders with self-attention
  - BERT
  - Contextualized word embeddings, document classification, sequence labelling
- Sequence decoders with self-attention
  - GPT
  - Language modelling, text generation
- Encoder-decoder model with cross-attention
  - The original Transformer
  - Machine translation

Types of Transformer models

- Sequence encoders with self-attention
  - BERT
  - Contextualized word embeddings, document classification, sequence labelling
- Sequence decoders with self-attention
  - GPT
  - Language modelling, text generation
- Encoder-decoder model with cross-attention
  - The original Transformer
  - Machine translation

Encoder models

For SGNS, we used one target word and one context word to predict their similarity

- The model doesn't know about the sentence in which the target word is used
- The model doesn't distinguish between right and left contexts, and between close and far contexts
- Similarity prediction is practical because it enables a self-supervised setup (no data annotation needed)

Masked language modelling

Let us use another setup:

- Take a sentence, replace 12% of tokens by a blank
- In addition, replace 1.5% of tokens by another randomly chosen token
- Train a model to “fill the blanks”
- 1 sentence = 1 training instance
- Self-supervised

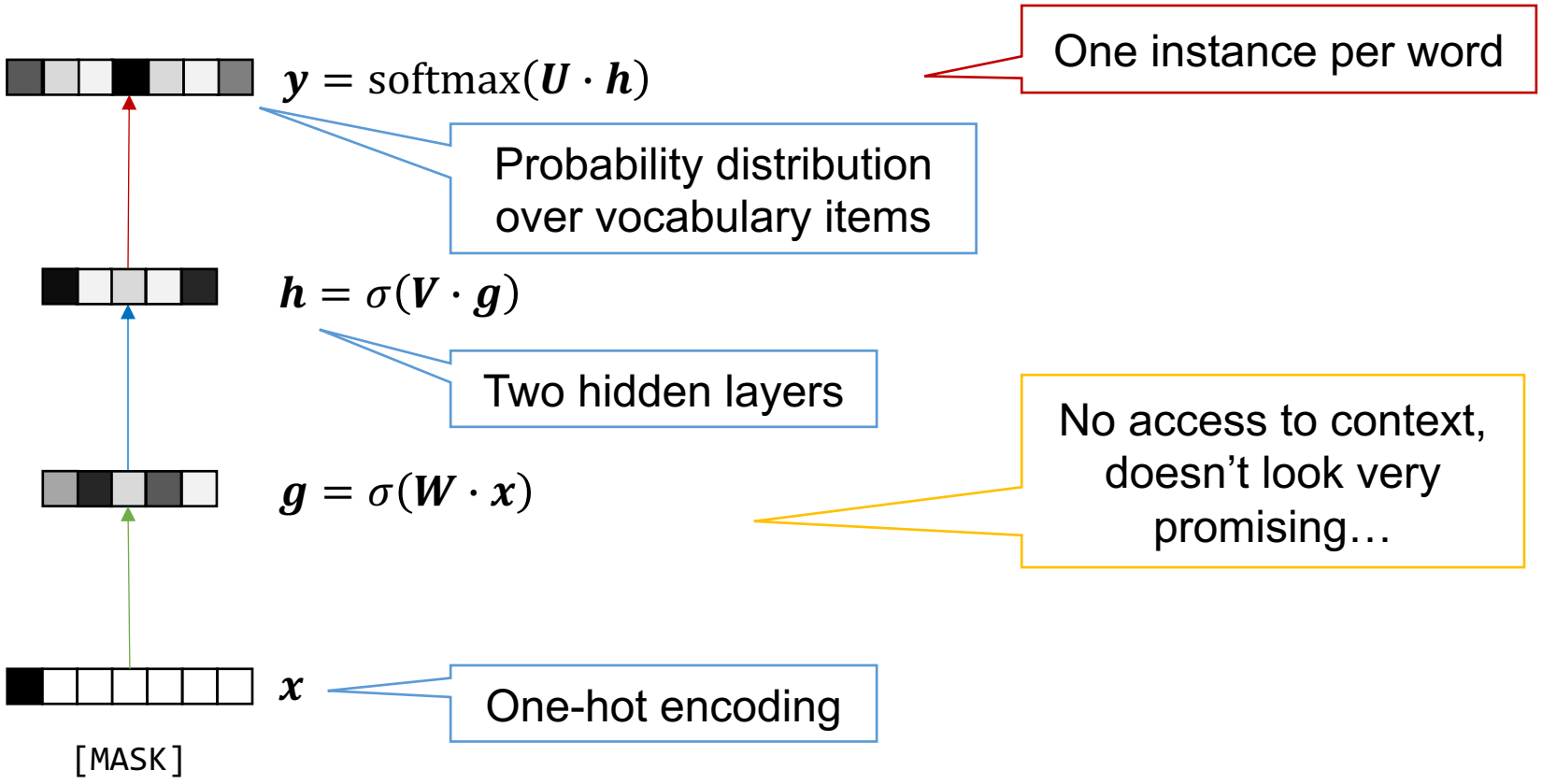
Example:



```
So long   and thanks for all the     fish
So [MASK] and [MASK] for all apricot fish
```

What kind of neural network?

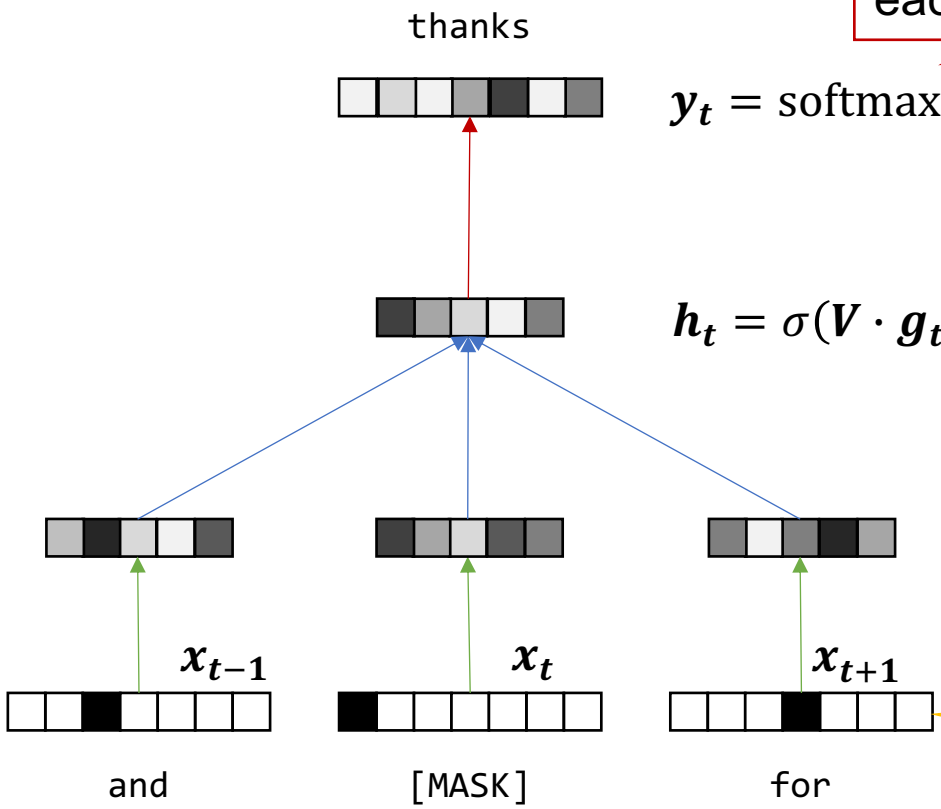
A simple feed-forward network:



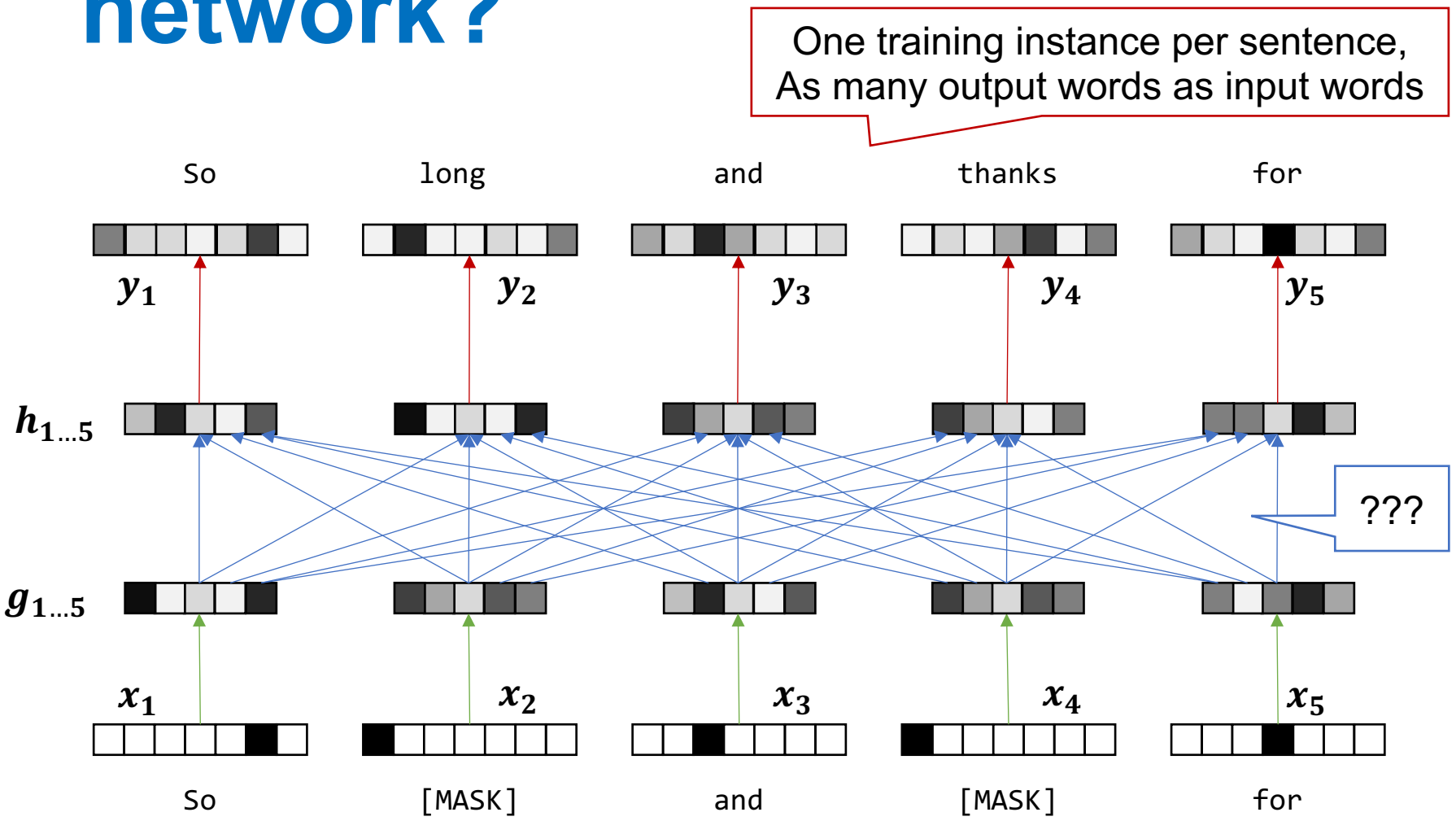
What kind of neural network?

Include context words:

One instance per center word, each word is present in 3 instances



What kind of neural network?



Self-attention

How to compute the $\boldsymbol{h}$ layer?

- Simple average:
  $$\boldsymbol{h_t} = \sigma\left(\frac{1}{n} \cdot \boldsymbol{V} \cdot \boldsymbol{g_1} + \cdots + \frac{1}{n} \cdot \boldsymbol{V} \cdot \boldsymbol{g_t} + \cdots + \frac{1}{n} \cdot \boldsymbol{V} \cdot \boldsymbol{g_n}\right)$$
- Some words are more important than others. Let's use a weighted average:
  $\boldsymbol{h_t} = \sigma\left(\alpha_{t,1} \cdot \boldsymbol{V} \cdot \boldsymbol{g_1} + \cdots + \alpha_{t,t} \cdot \boldsymbol{V} \cdot \boldsymbol{g_t} + \cdots + \alpha_{t,n} \cdot \boldsymbol{V} \cdot \boldsymbol{g_n}\right)$
- How do we know the α values?
  - Maybe depending on the distance to t? Maybe not?
  - Anyway, a neural network should be able to learn these values automatically…
- Idea: depending on similarity: $\alpha_{t,c} = \boldsymbol{g_t} \cdot \boldsymbol{g_c}$
  - Recall: the dot product is a simple measure of similarity

Self-attention

How to compute the $\boldsymbol{h}$ layer?

- The more similar the vectors $\boldsymbol{g_c}$ and $\boldsymbol{g_t}$ are, the more important $\boldsymbol{g_c}$ is for computing $\boldsymbol{h_t}$:

$$\alpha_{t,c} = \boldsymbol{g_t} \cdot \boldsymbol{g_c}$$

- For a weighted average, all αs should sum up to 1. Let's use softmax:

$$\alpha_{t,c} = \text{softmax}(\boldsymbol{g_t} \cdot \boldsymbol{g_c})$$

We could also have used sigmoid here, as for SGNS.

- Putting everything together:

$$\boldsymbol{h_t} = \sigma\left(\sum_{i=1}^{n} \alpha_{t,i} \cdot \boldsymbol{V} \cdot \boldsymbol{g_i}\right)$$

The "real" thing is still more complicated…

- That's a simple type of **self-attention**.

Self-attention, continued

- The $\boldsymbol{g}$ vectors now occur in 3 places and roles:
  - As target word for computing α: $\mathrm{softmax}(\boldsymbol{g_t} \cdot \boldsymbol{g_c})$
    - We call this role **query**
  - As context word for computing α: $\mathrm{softmax}(\boldsymbol{g_t} \cdot \boldsymbol{g_c})$
    - We call this role **key**
  - As a factor of the final product: $\alpha_{t,i} \cdot \boldsymbol{V} \cdot \boldsymbol{g_i}$
    - We call this role **value**
- The 3 roles are different, and the $\boldsymbol{g}$ vectors are not equally well suited for all of them.
  - Let's create 3 different vectors tailored to the different roles!

Self-attention, continued

- Let's create 3 different vectors:
  - Query vector: $\boldsymbol{q_i} = \boldsymbol{W^Q} \cdot \boldsymbol{g_i}$
  - Key vector: $\boldsymbol{k_i} = \boldsymbol{W^K} \cdot \boldsymbol{g_i}$
  - Value vector: $\boldsymbol{v_i} = \boldsymbol{V} \cdot \boldsymbol{g_i}$ (we already have this one)
- What about our weight values α?

$$\alpha_{t,c} = \text{softmax}(\boldsymbol{q_t} \cdot \boldsymbol{k_c})$$

- It turns out that the dot product needs to be scaled before passing it to the softmax:

$$\alpha_{t,c} = \text{softmax}\left(\frac{\boldsymbol{q_t} \cdot \boldsymbol{k_c}}{\sqrt{d_k}}\right)$$

d_k refers to the dimensionality of $\boldsymbol{k}$ (and also of $\boldsymbol{q}$)

Self-attention, continued

Putting things together again:

$$\boldsymbol{h_t} = \sigma\left(\sum_{i=1}^{n} \underbrace{\text{softmax}\left(\frac{\boldsymbol{q_t} \cdot \boldsymbol{k_i}}{\sqrt{d_k}}\right)}_{\alpha_{t,i}} \cdot \underbrace{\boldsymbol{v_i}}_{\boldsymbol{V} \cdot \boldsymbol{g_i}}\right)$$

α is a square matrix that shows how important a context word is at a given position:

q1•k1	q1•k2	q1•k3	q1•k4	q1•k5
q2•k1	q2•k2	q2•k3	q2•k4	q2•k5
q3•k1	q3•k2	q3•k3	q3•k4	q3•k5
q4•k1	q4•k2	q4•k3	q4•k4	q4•k5
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

Multi-head attention

- We may want to have several α matrices to represent different types of attention:
  - One for syntactic relatedness
  - One for semantic relatedness
  - One for coreference, etc.

- Let's start over, with an extra index h (head):
  - Query vector: $\boldsymbol{q_i^h = W^{Q,h} \cdot g_i}$
  - Key vector: $\boldsymbol{k_i^h = W^{K,h} \cdot g_i}$
  - Value vector: $\boldsymbol{v_i^h = V^h \cdot g_i}$

Multi-head attention

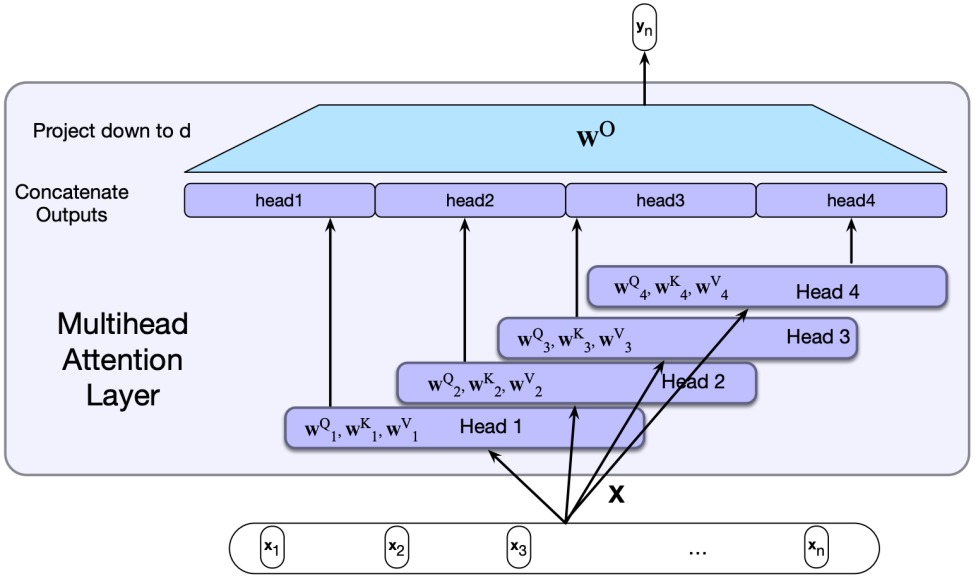
We have removed the sigmoid here

- Compute the vector for one head:

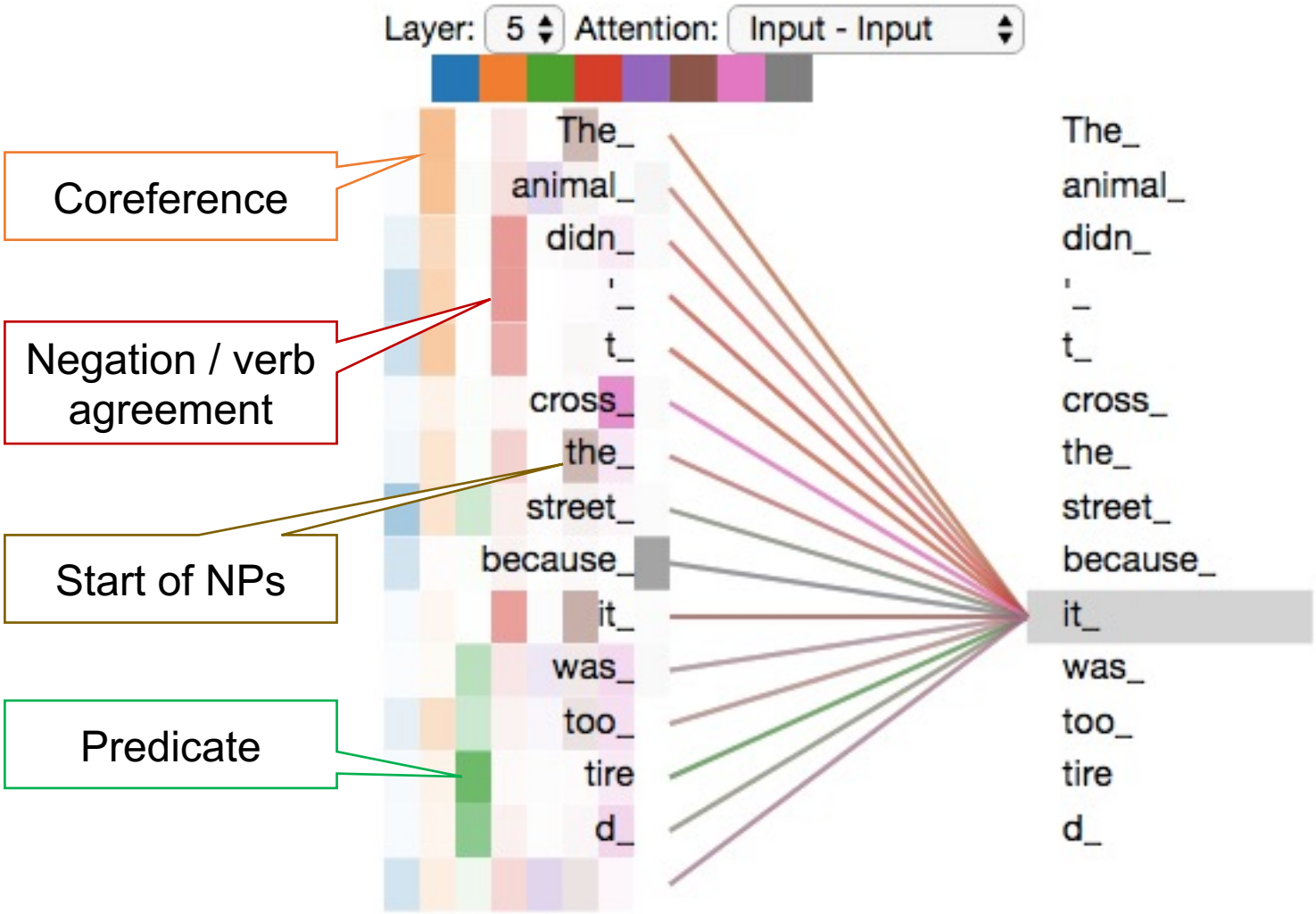
$$\mathbf{head}_t^h = \sum_{i=1}^{n} \text{softmax}\left(\frac{\boldsymbol{q}_t^h \cdot \boldsymbol{k}_i^h}{\sqrt{d_k}}\right) \cdot \boldsymbol{v}_i^h$$

- Then concatenate all head vectors and project them:

$$\boldsymbol{h}_t = \left(\mathbf{head}_t^1 \oplus \cdots \oplus \mathbf{head}_t^m\right) \cdot \boldsymbol{W}^O$$



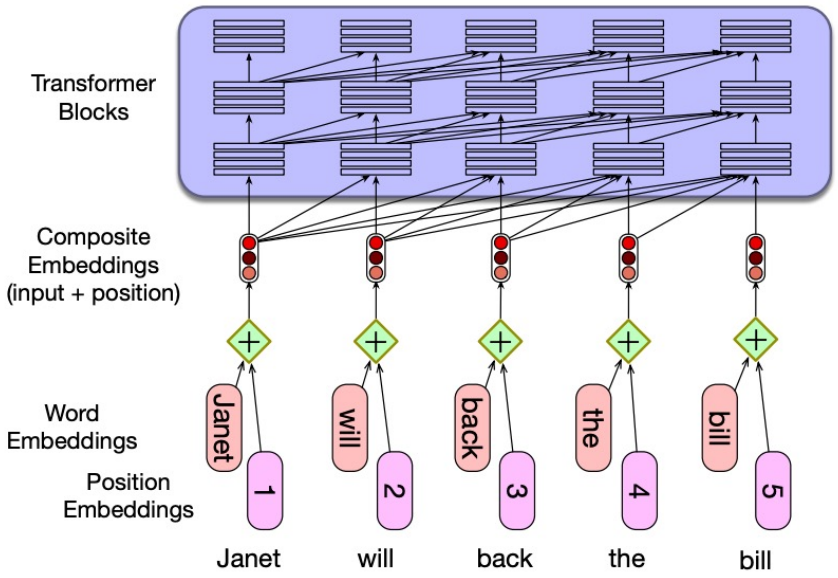
Multi-head attention



http://jalammar.github.io/illustrated-transformer/

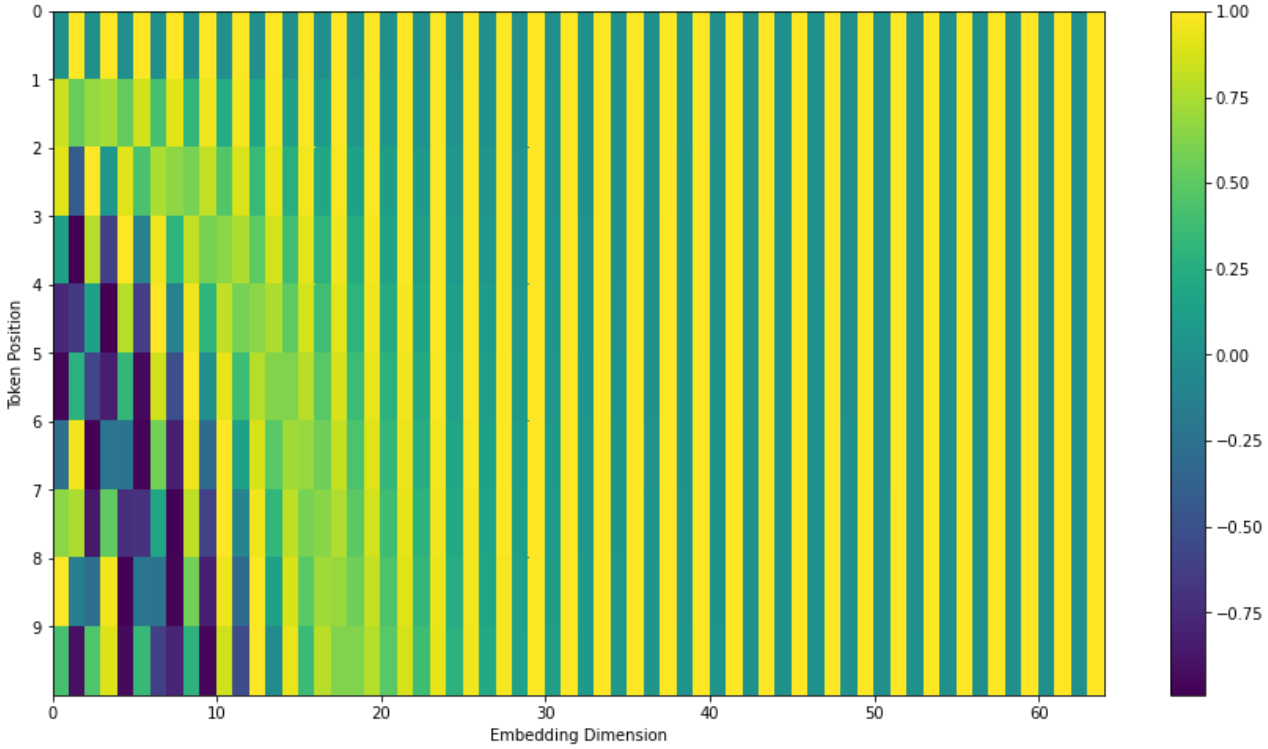
Position embeddings

- Self-attention does not have any notion of word ordering
  - As of now, the attention weights are only chosen based on semantic similarity of the words
  - But we probably should take simple proximity into account…
- Simple solution:
  - Concatenate semantic word embedding with absolute position embedding



Position embeddings

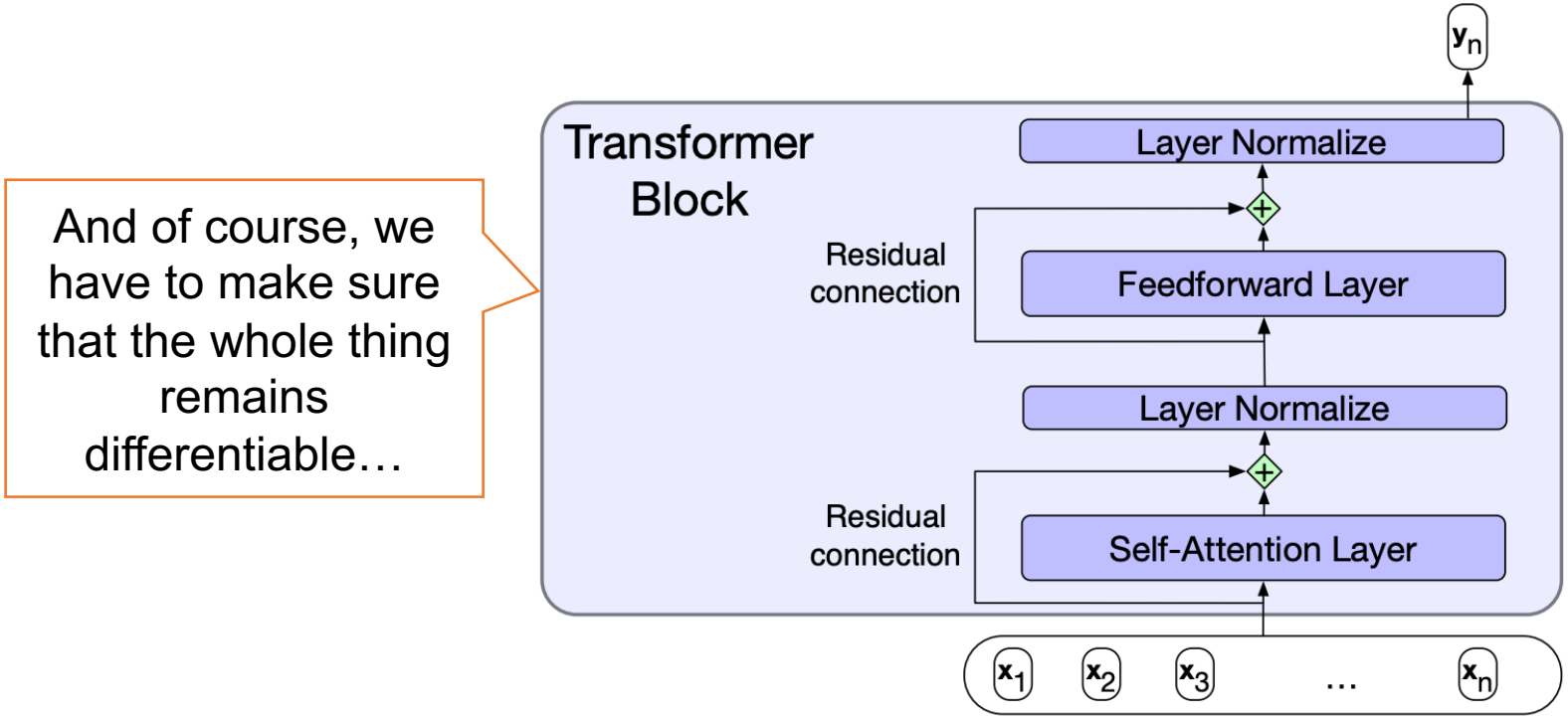
- Absolute numbers are not very efficient
- The “real thing” uses several overlaid sine functions:



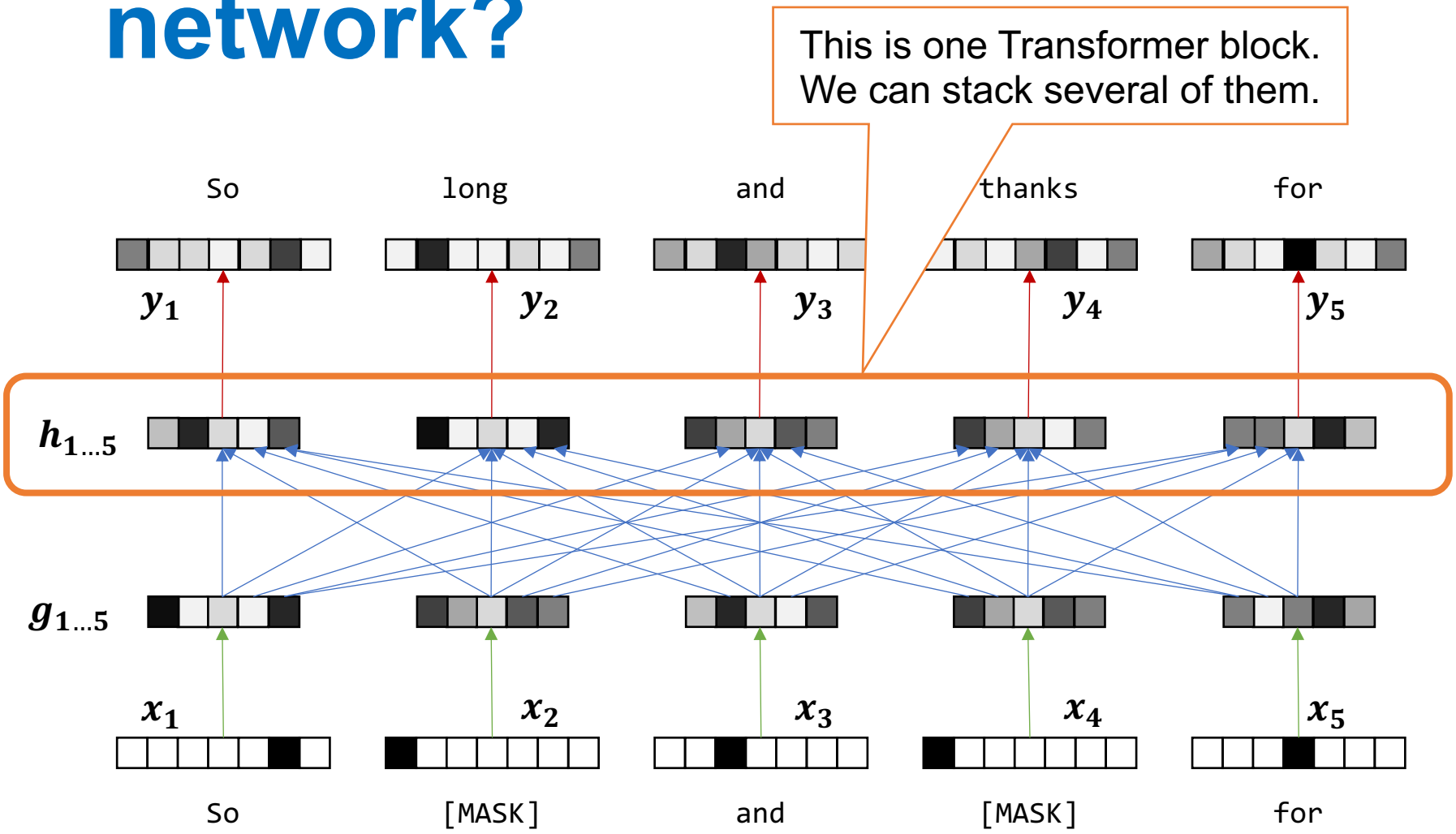
Transformer blocks

Multi-head self-attention isn't quite sufficient

We need a few extra things and package everything up into so-called Transformer blocks:



What kind of neural network?

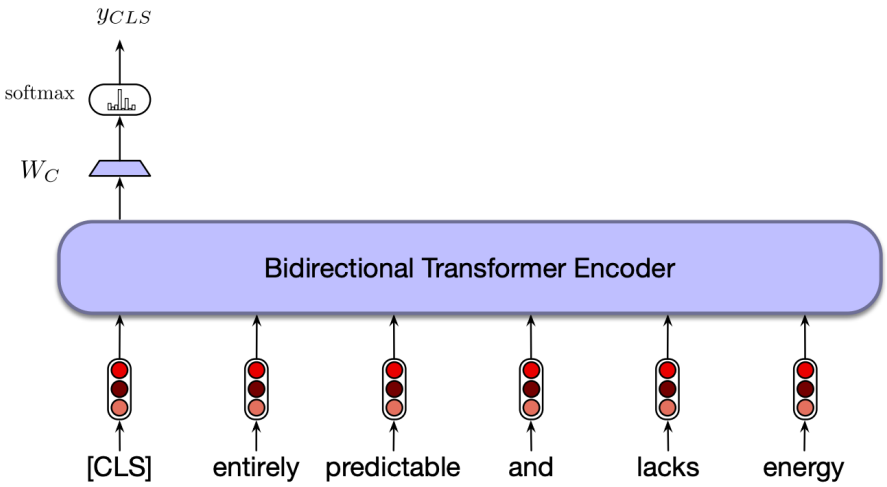


What can we do with sequence encoders?

- Get contextualized word embeddings
  - Train a model on the MLM task
  - Pass a sentence through the network and extract the h vector of each word as its embedding
  - Alternative: average the vectors of several layers

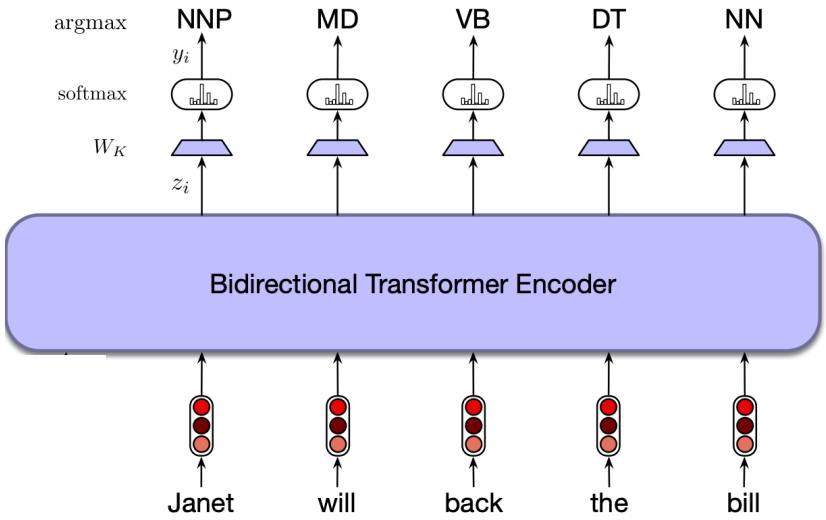
What can we do with sequence encoders?

- Text classification (one label per sentence)
  - Train a model on the MLM task, adding a `[CLS]` token in front of every sentence
  - Throw away the output layer, create a new one
  - **Fine-tune** the model to predict the label at the `[CLS]` position



What can we do with sequence encoders?

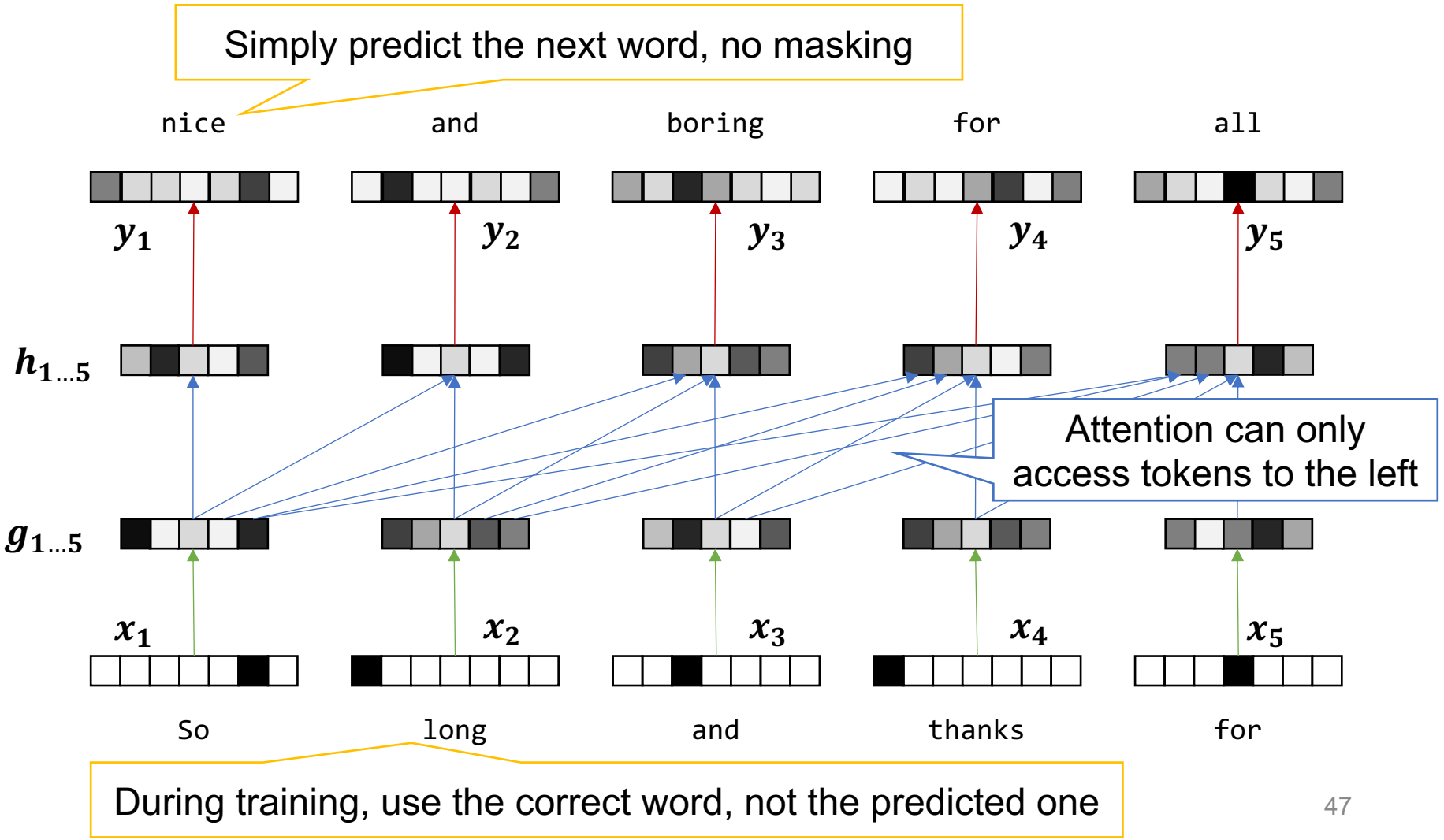
- Sequence labeling (e.g. POS tagging)
  - Train a model on the MLM task
  - Throw away the output layer, create a new one
  - **Fine-tune** the model on POS-annotated data



Types of Transformer models

- Sequence encoders with self-attention
  - BERT
  - Contextualized word embeddings, document classification, sequence labeling
- Sequence decoders with self-attention
  - GPT
  - Language modeling, text generation
- Encoder-decoder model with cross-attention
  - The original Transformer
  - Machine translation

Language modeling with self-attention



Language modeling with self-attention

Probabilistic n-gram language model:

$$P(w_1, \ldots, w_n) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdot \cdots \cdot P(w_n | w_1, \ldots, w_{n-1})$$

- For n-gram models, we could not condition on all preceding tokens
- Thanks to attention, we can do that now
- But typically, far-away tokens get less attention

Language modeling with self-attention

For simplicity, let's assume a single head:

$$\alpha_{t,c} = \text{softmax}\left(\begin{cases} -\infty, & c > t \\ \dfrac{\boldsymbol{q_t} \cdot \boldsymbol{k_c}}{\sqrt{d_k}}, & c \leq t \end{cases}\right)$$

q1•k1	−∞	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞	−∞
q4•k1	q4•k2	q4•k3	q4•k4	−∞
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

$$\boldsymbol{h_t} = \sigma\left(\sum_{i=1}^{n} \alpha_{t,i} \cdot \boldsymbol{v_t}\right)$$

Language modeling with self-attention

- In terms of model architecture, that's about all you need to know in order to build your own GPT model ☺
- Of course, you'll still need the right amount (and quality) of training data…

Types of Transformer models

- Sequence encoders with self-attention
  - BERT
  - Contextualized word embeddings, document classification, sequence labeling
- Sequence decoders with self-attention
  - GPT
  - Language modeling, text generation
- Encoder-decoder model with cross-attention
  - The original Transformer
  - Machine translation

Machine translation

Some (fairly obvious) facts about MT:

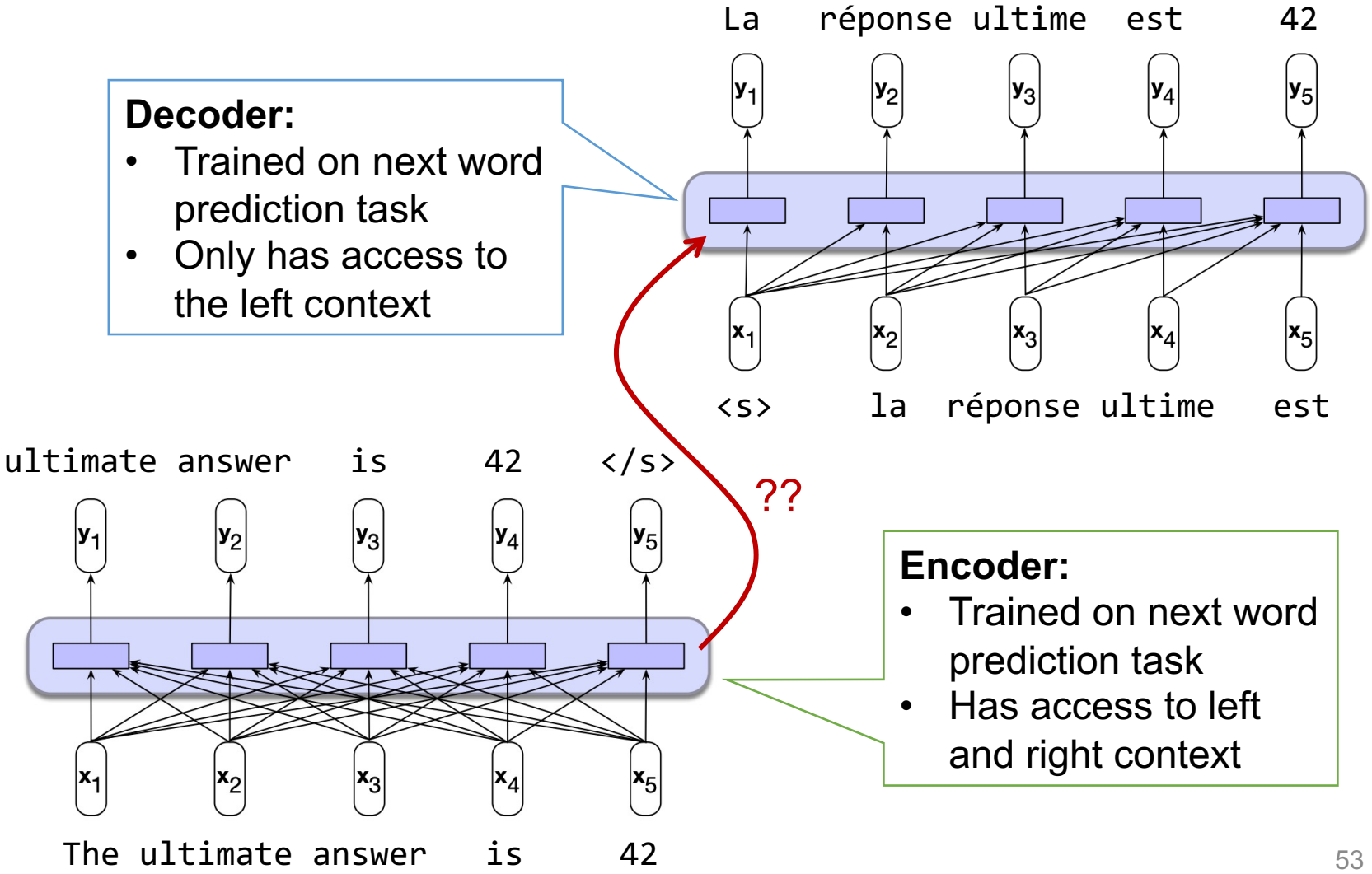
- Input: a sentence in the source language
- Output: a sentence in the target language
  - Hopefully grammatical
  - Hopefully with the same meaning
- Source and target sentence most often do not have the same number of words nor the same word order

Main idea:

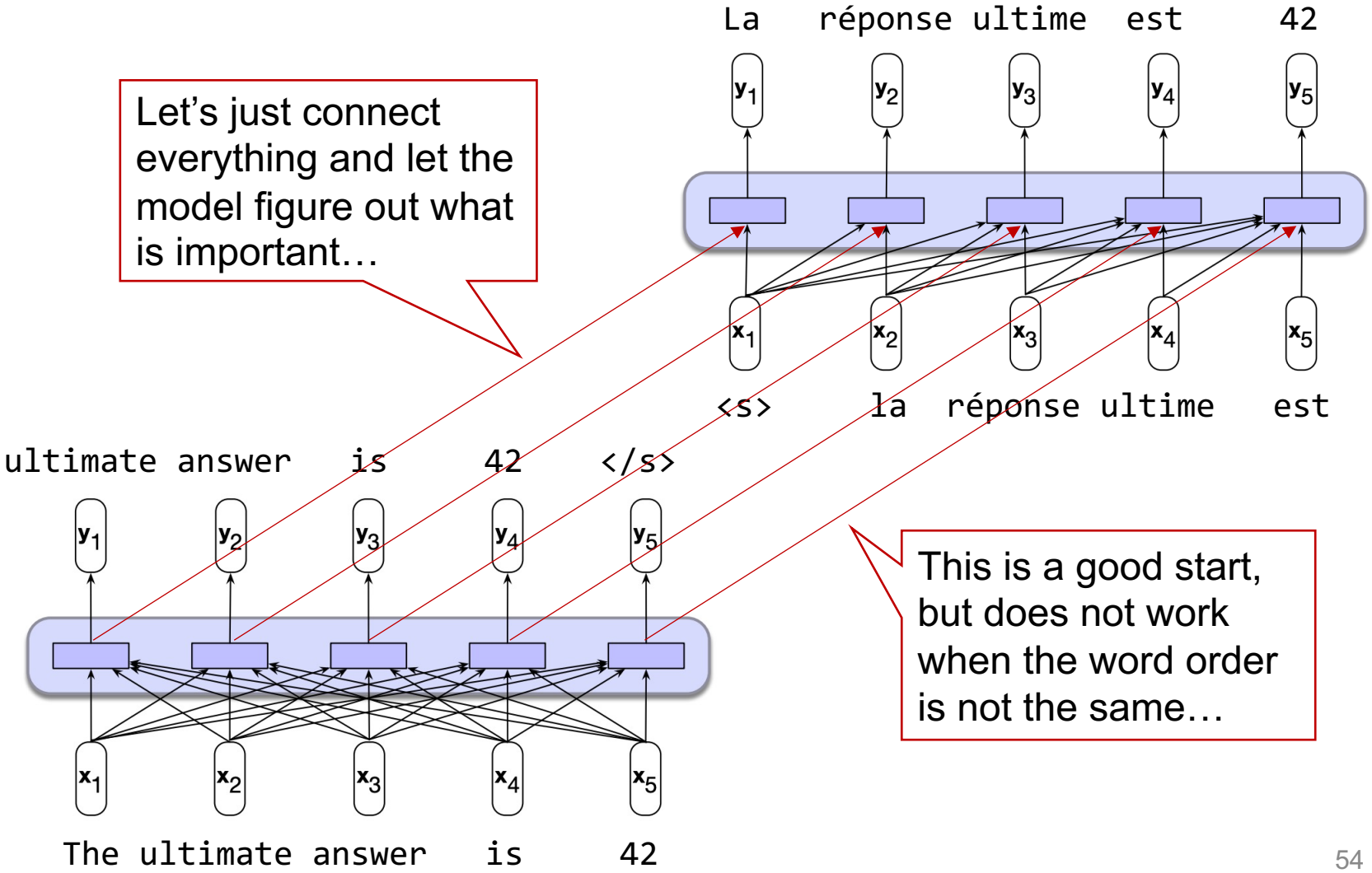
- Train an encoder for the source language
- Train a decoder for the target language
- Connect the two

Everything can be trained together thanks to backpropagation

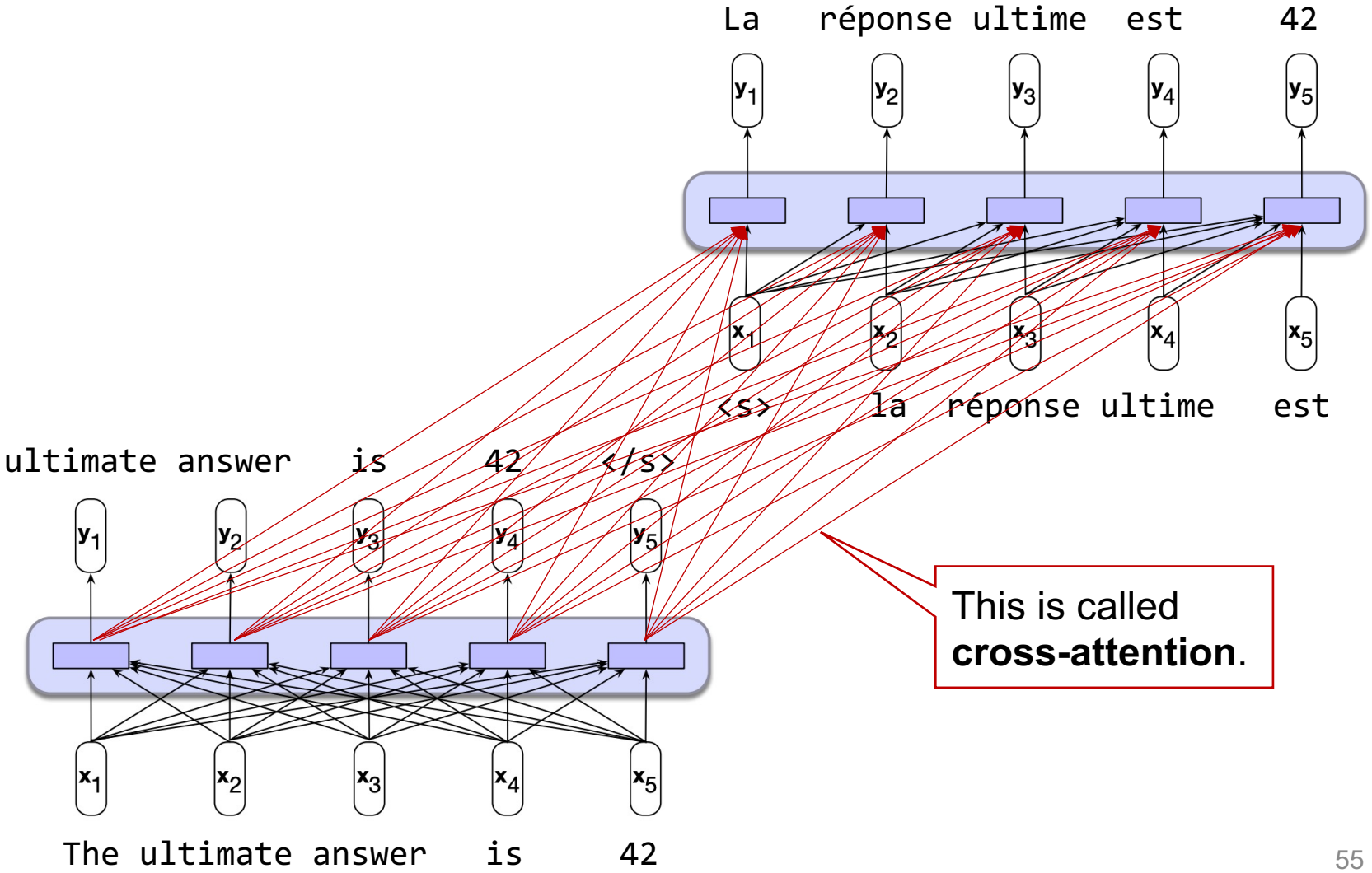
Connecting encoders and decoders



Connecting encoders and decoders



Connecting encoders and decoders



Cross-attention

- Dot-product attention:

$$\alpha_{i,j} = \text{softmax}\left(\boldsymbol{h_j^{enc}} \cdot \boldsymbol{h_{i-1}^{dec}}\right)$$

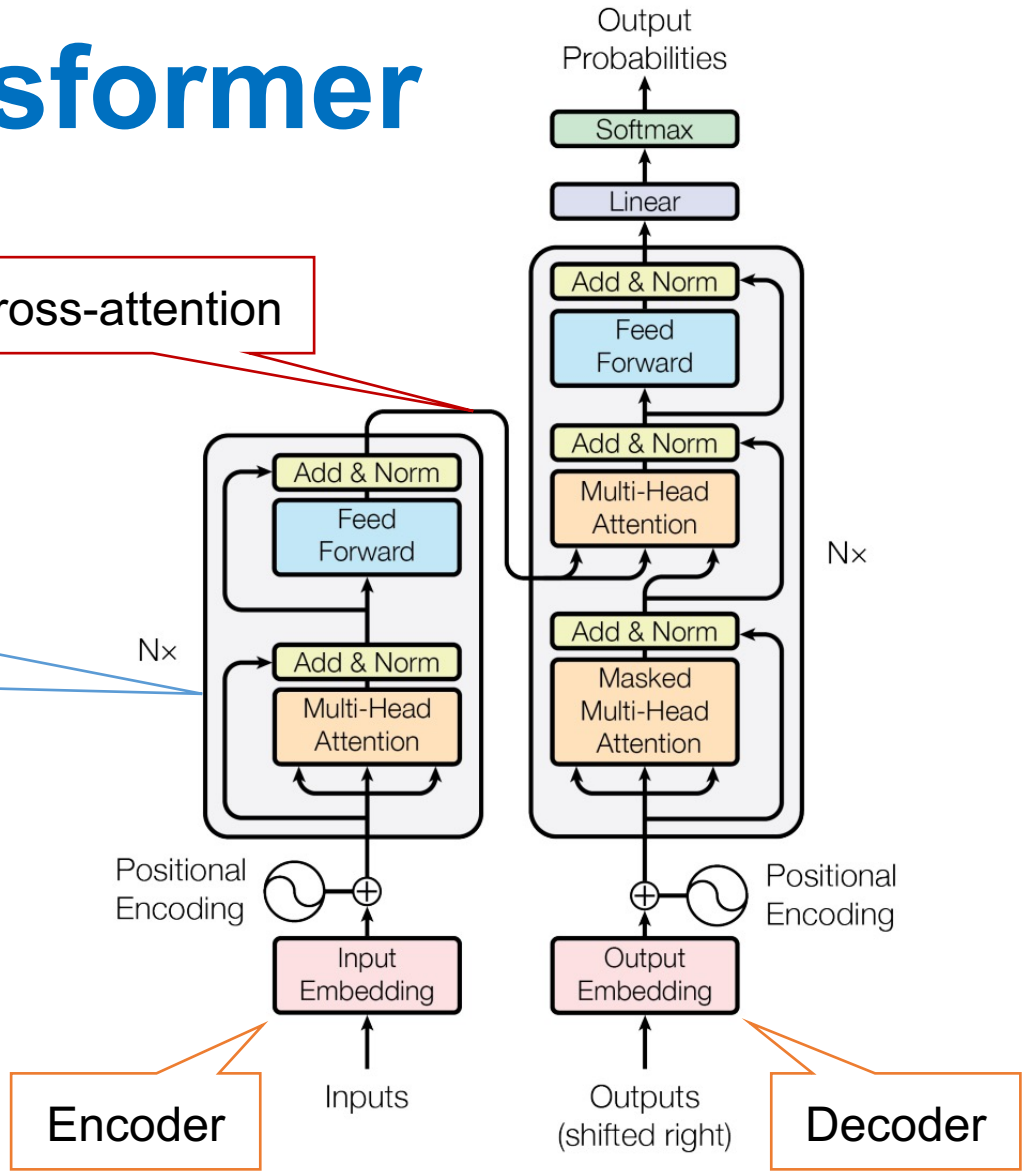
  - Determines the importance of the jth word of the encoder for the ith word of the encoder.
  - There are other attention types, and one can again use multiple heads.
- Context vector:

$$\boldsymbol{c_i} = \sum_j \alpha_{i,j} \cdot \boldsymbol{h_j^{enc}}$$

  - This vector is then combined with the vector produced by the decoder self-attention.

The full Transformer

Historically, this is what everything started with…



https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Readings

- Jurafsky & Martin, section 5.5
  - Word embeddings, word2vec
- Jurafsky & Martin, chapter 8
  - Transformers
- Jurafsky & Martin, chapter 11
  - Masked language models (i.e., sequence encoders with self-attention / BERT)