Natural Language Understanding, Generation and Machine Translation - Week 3 - Encoder-Decoder Models, Attention and Transformers

Antonio León Villares

February 2023

Contents

Based on:

- *Sections 7, 8 and 9 of Neubig's "Neural Machine Translation and Sequence-tosequence Models: A Tutorial*
- *Attention is All You Need by Vaswami et al.*
- *Transformers from Scratch, a blog by Peter Bloem*
- *Visual Guide to Transformers, a video by Hedu AI*
- *Illustrated Guide to Transformers Neural Network: A step by step explanation, by The A.I. Hacker - Michael Phi (this explains encoder-decoder transformer architectures)*

1 RNNs for Translation

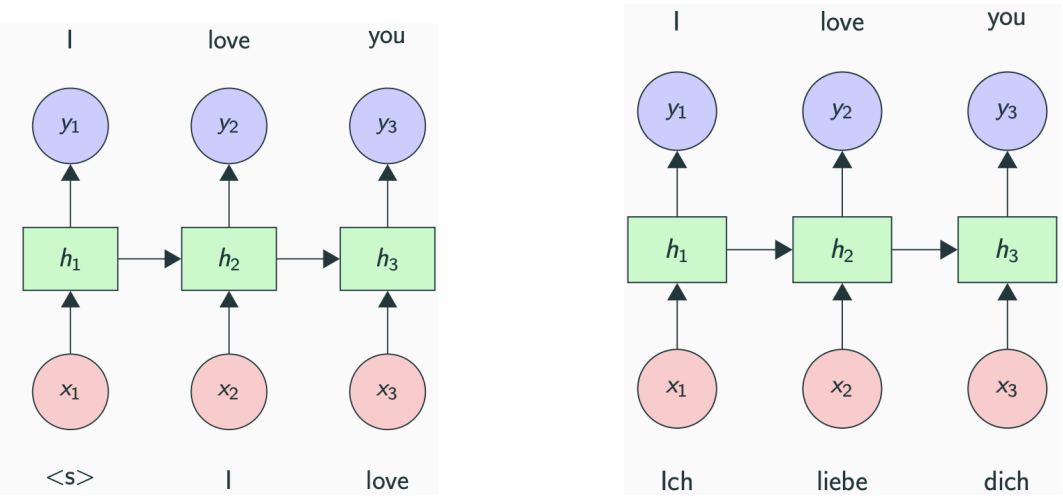
- **What is the machine translation task?**
  - given a **source** language string $\underline{x}$, producing a **target** language string $\underline{y}$
  - overall, we are seeking to learn a **function**:

$$P(\underline{y} \mid \underline{x})$$

  - this doesn't necessarily have to involve languages (i.e English to French), but can be **part of speech tagging** or some other **sequence-to-sequence task**
- **What are the benefits of using RNNs for translation?**
  - the main issues with using n-grams are:
    - only uses a **fixed-length** context
    - requires **independence assumption**
    - without **smoothing**, 0 probabilities are possible
  - **neural models** are **universal function approximators**, so technically these constraints won't affect our task
  - RNNs are espcially well-suited, since they can take inputs of arbitrary length

1.1 Outputs in Target Language

- **What is the most straightforward way to adapt RNNs for translation?**
  - as input, take the words in the **source language**
  - use the **output** of the RNN cells as the translation in the **target language**



(a) A standard RNN language model.

(b) An RNN for translating German to English.

Figure 1: By using RNN cells, we can directly output a translation.

- **In practice, why is this approach not appropriate for the translation task?**
  - **Different Number of Words**:

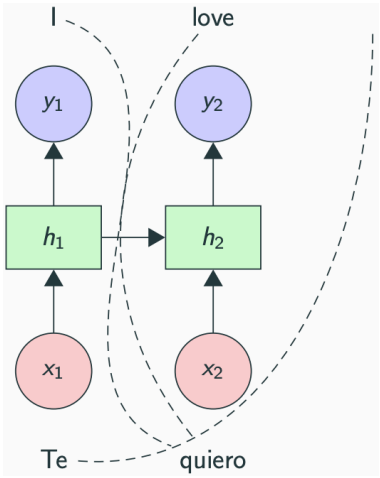


Figure 2: In Spanish, "I love you" gets translated to "Te quiero". An RNN would need to output dummy tokens to account for the different in number of words. This will be non-trivial if the target language requires **more** words (i.e going from English to Spanish). Moreover, this difference in word number often implies difference in conveyed meaning (for instance, "Te" doesn't mean "I", it refers to "you" -or more accurately, "to you")

  - **Different Word Order/Syntax**:

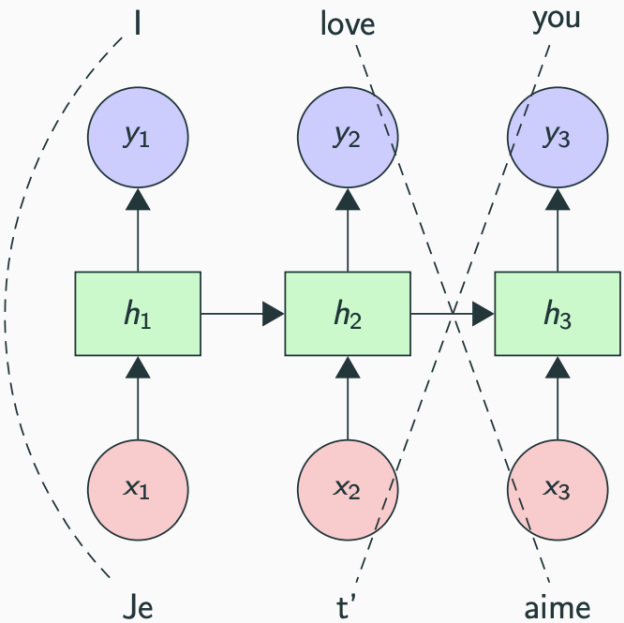


Figure 3: In French, "I love you" gets translated to "Je t'aime". An RNN would not only need to output a different word for translation (i.e map "love → t'"), but it would also need to know what words will appear in the future (i.e knowing that "you" will go after "love", so as to output "t'" when it sees "love").

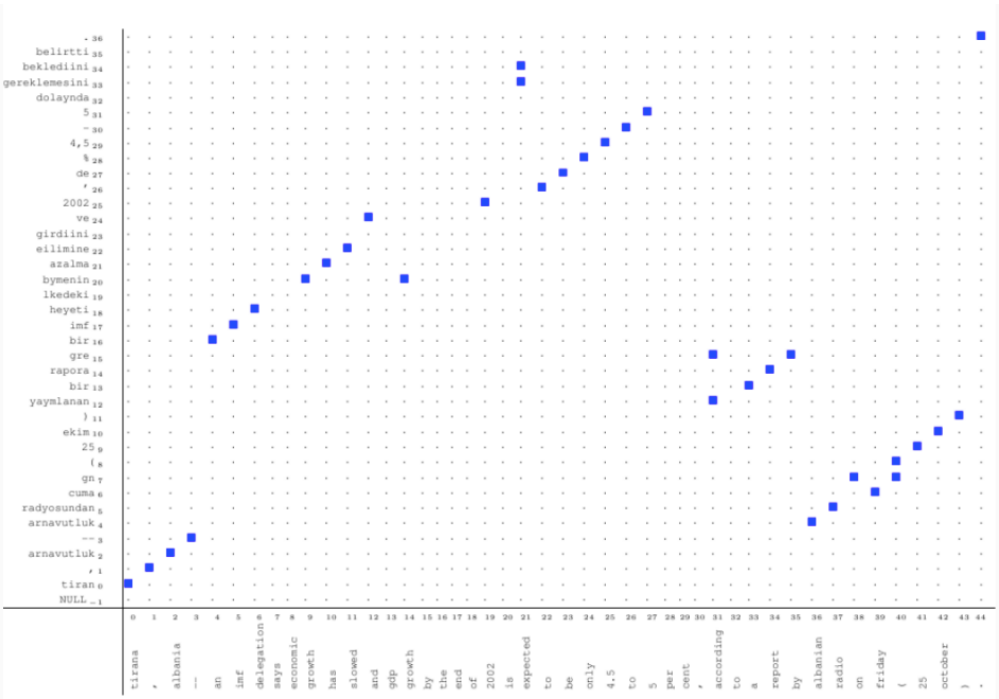
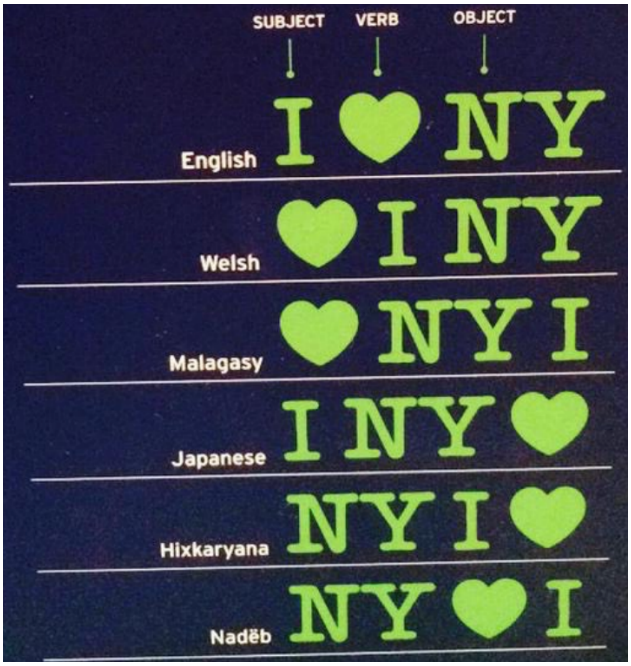
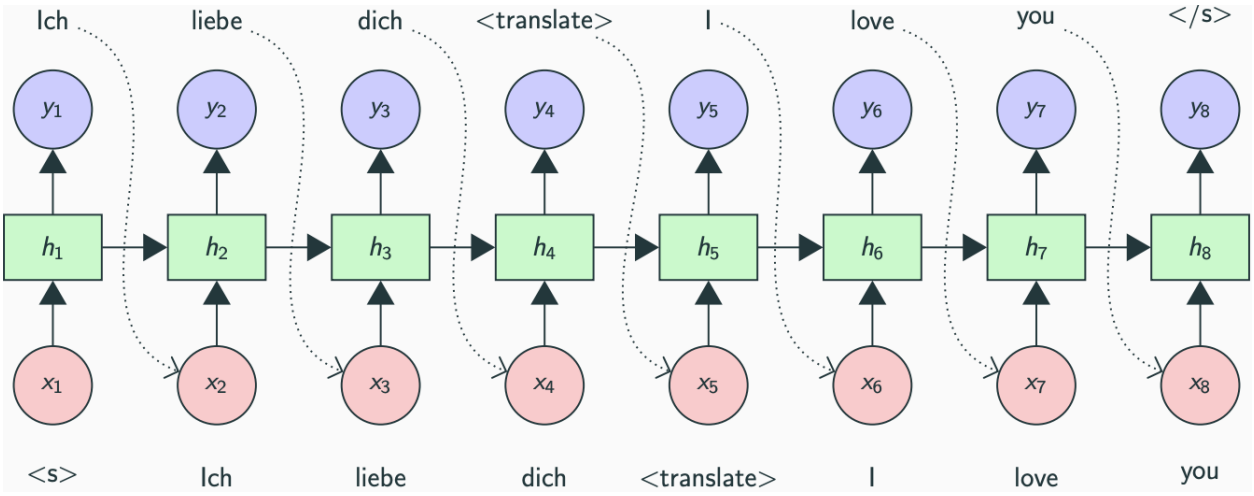


Figure 4: Translation chart, with Turkish on the y-axis, and English on the x-axis. blue squares represent which words in Turkish correspond to which words in English. If word order were to be preserved, we should see a straight diagonal line.



1.2 String Completion

- **How can the translation task be modelled as a string completion problem?**
  - instead of **outputting** the translation, just generate long strings
  - the **source language** goes in, and once the model observes a `<translate>` token, it should generate a corresponding string in the **target language**



- **What issues arise from using a string completion architecture?**
  - **Language Modelling**: we are essentially learning a large language model, but for 2 different languages:
    - we don't really care about a LM for the **source** (we already assume it to be grammatical)

* the 2 languages probably have extremely different **vocabulary** (we'd have to understand different words in different languages) and **morphosyntactic structure** (gauging this with a simple RNN cell for **both** languages is hard)

– **Long-Range Dependencies**: we might want to translate sentences which are 20 words long, but when the translation begins, the hidden representation will be mainly conditioned by the last few words of the source language

2 The Encoder-Decoder Approach to Translation

2.1 Encoder-Decoder Architecture

- **What is the structure of an encoder-decoder for MT?**

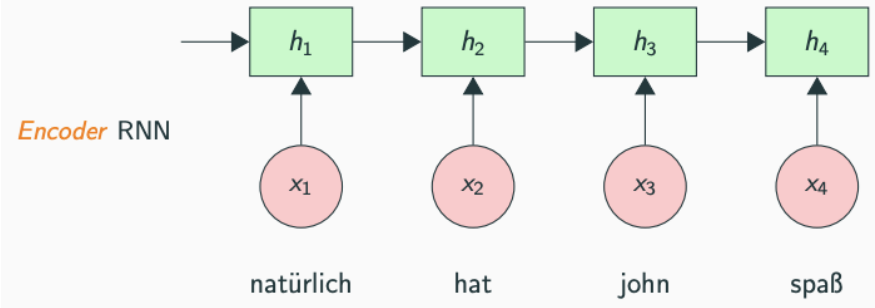


Figure 5: The **encoder** constructs a hidden representation for the input sentence, h_4.

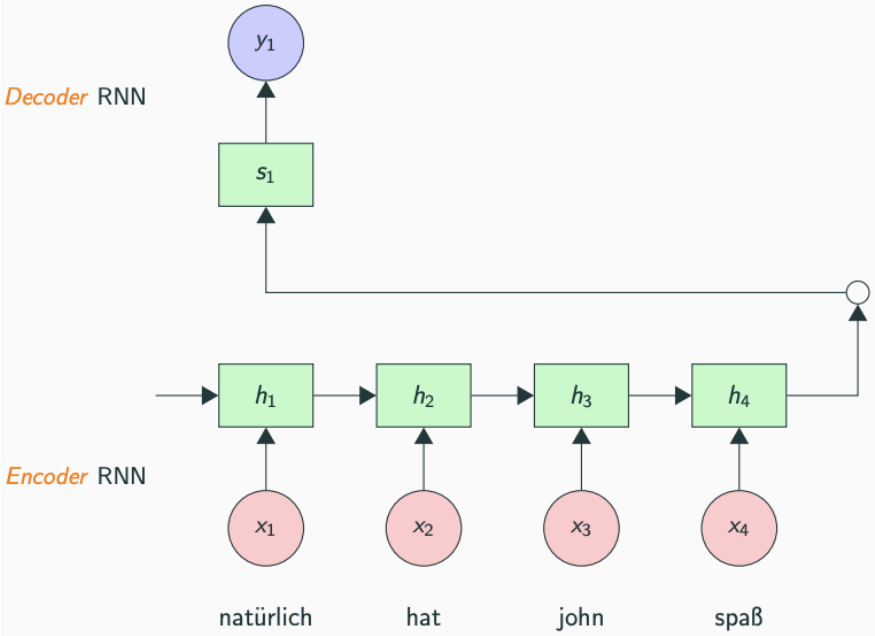


Figure 6: h_4 is passed to the decoder, providing it with an **initial state**, with which to compute the hidden state s_1. s_1 and h_4 are concatenated, and used to create the output y_1 (a probability distribution over **target language** words).

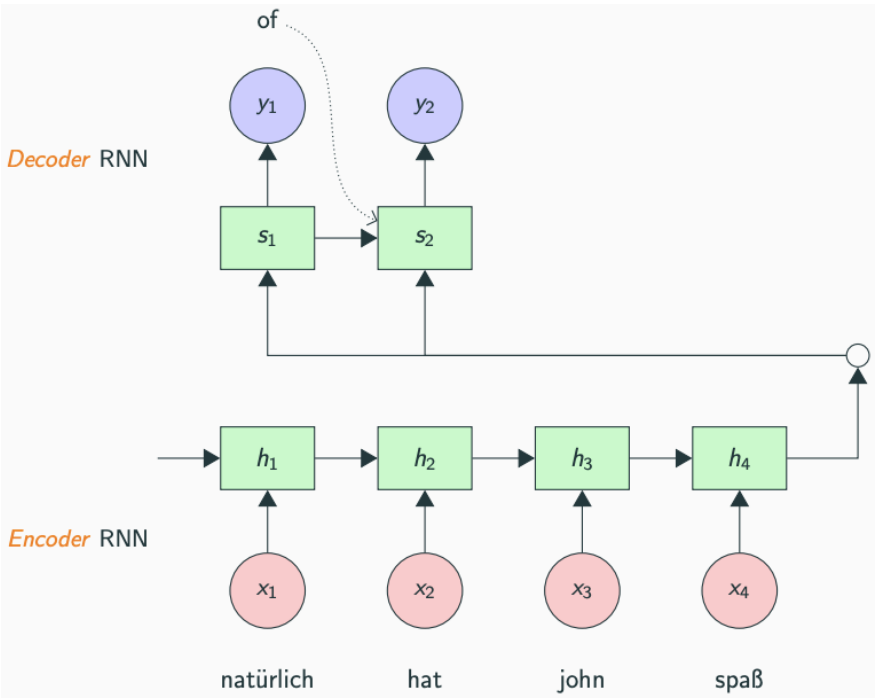


Figure 7: The output y_1 corresponds to "of". We then pass "of" and the **hidden state** s_1 to the next cell.

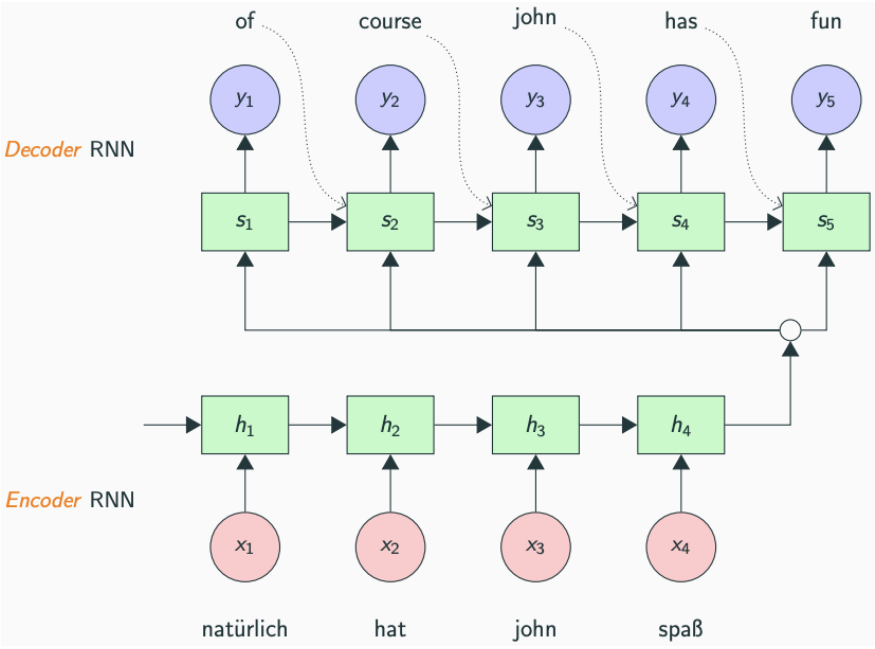


Figure 8: We can repeat this for each cell in the network, which uses the **previous** hidden state and output prediction to generate the current hidden state, and use this hidden state alongside h_4 to generate the current output.

- **How can we mathematically formalise the encoder-decoder architecture for translation?**

- whilst the encoder/decoder don't have to be RNNs (they don't even have to have the same architecture), it is conceptually easier to understand
- let RNN_{enc} denote an RNN cell in the **encoder**, and RNN_{dec} denote an RNN cell in the **decoder**. Moreover, let the **input sentence** contain N words.
- for the **encoder**, we compute the ith **hidden state** $\underline{h}_i$ via:

$$\underline{h}_i = \begin{cases} RNN_{enc}(\underline{x}_i, \underline{h}_{i-1}), & i \in [1, N] \\ \underline{0}, & i = 0 \end{cases}$$

  where:
  - the first word is w_1
  - $\underline{x}_i$ represents the **embedding** for the ith word, w_i
- for the **decoder**, we compute the ith **hidden state** $\underline{s}_i$ via:

$$\underline{s}_i = \begin{cases} RNN_{dec}(\underline{y}_{i-1}, \underline{s}_{i-1}), & t \geq 1 \\ \underline{h}_N, & t = 0 \end{cases}$$

  where $\underline{y}_i$ is the **embedding** for the ith word outputted by the decoder
- to compute the **output**, we learn weights $W, \underline{b}$:

$$P(\hat{w}_i \mid \hat{w}_{1:i-1}, w_{1:N}) = P(\underline{y}_i) = softmax(W concat(\underline{s}_i, \underline{h}_N) + \underline{b})$$

  where $concat(\underline{s}_i, \underline{h}_N)$ is the vector obtained by concatenating $\underline{s}_i, \underline{h}_N$)

- **How is the output of the translation model generated?**
  - to generated $\underline{y}_i$, we can follow one of:
    1. **Random Sampling**: randomly sample from the distribution $P(\underline{y}_i)$. This might be useful if we want the same input to have different outputs (i.e chatbots)
    2. **Greedy 1-Best Search**: simply select $argmax(\underline{y}_i)$ (select the word with the highest probability). This doesn't guarantee that the generated translation is the most probable.
    3. **Beam Search**: like greedy search, but selecting the n most likely words.
  - care must be taken, since neural models will tend to prefer **shorter sentences** (if we think about multiplying probabilities, the more words we add, the lower the probability of a sentence)
  - as such, we might try to choose sentences which maximise the **average log probability per word**

2.2 Issues with the Encoder-Decoder Paradigm

- **Why does the encoder hidden representation constitute a bottleneck?**
  - $\underline{h}_N$ must encode **all** the information contained in the **source language** input, **independently** of the length of the sentence
  - this induces a **recency bias**: even with LSTMs, RNNs will have a representation bias for the most recent words
  - thus, the **starting** words in the **target language** will rely on **longer dependencies**

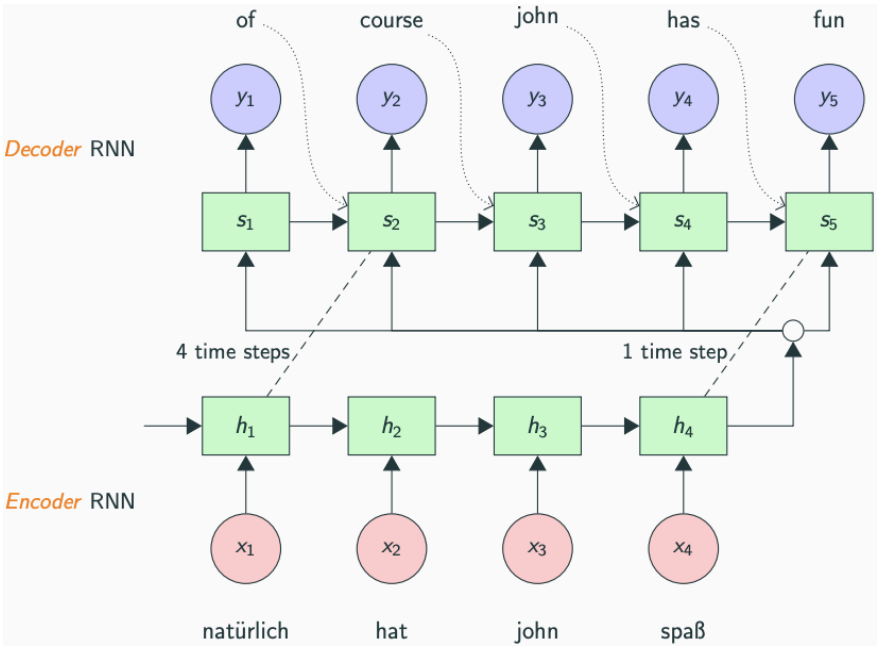


Figure 9: The hidden state for the first word h_1 gets lost in the final hidden state h_4. Thus, we are losing potentially useful information for decoding s_1.

  - solving this problem with brute force (i.e deeper networks with more parameters) would be wasteful: it requires excessive **memory** and **training time**

- **Why does reversing the order in which the input sentence is "read" not help?**
  - we can alternatively process from **end** to **start**, resulting in a representation that better gauges the first few words
  - in some senses this can be useful: words in certain languages might appear in the same order (i.e words at the beginning of a French sentence should appear towards the beginning of the corresponding English sentence)
  - however:
    * we will now lose information about the **end** of the **source sentence**
    * the assumption that words will appear in similar positions in the **source** and **target** languages doesn't always hold

- **How can a bidirectional encoder mitigate the loss of information in the hidden representation bottleneck?**
  - we can obtain both **forward** and **backward** hidden representations (passing the encoder from start to end, and from end to start)
  - the resulting hidden representations can then be **concatenated**, to create a **bidirectional hidden representation**
  - however, this still causes problems if the sentence is **long**: information about the middle words will be lost in both **forward** and **backward** representations

- **Can the decoder be biridectional?**
  - no, since the **decoder** generates the output
  - we'd have no **right context** to pass on to the next state

- **Can we somehow just directly pass on each of the hidden representations $\underline{h}_1, \ldots, \underline{h}_N$ to the decoder?**
  - the main issue with this approach is that **we don't know the input length**
  - thus, we can't really **concatenate** all the hidden layers (we'd be passing variable length vectors)
  - an alternative would be **adding** or taking a **mean** - but this again would lose information
  - for example, "natürlich" translates to "of course"; as such, it shouldn't care about words like "john", "has" or "fun"
  - we'd need to find a way of assigning **weights** to each of the hidden representations

2.3 Attention

- **What is attention?**
  - intuitively, when we look at **text** or an **image**, depending on the **task**, we put our **focus** on different parts

  - **attention** in our **encoder-decoder** architecture allows us to assign **weights** to each of the **hidden representations** $\underline{h}_1, \ldots, \underline{h}_N$
  - thus, when predicting an **output** in the **target** representation, we no longer rely on the **last** hidden representation $\underline{h}_N$; instead, we can use a **weighted sum** to create a **context vector**:

$$\underline{c}_i = \sum_{j=1}^{N} \alpha_{ij} \underline{h}_j$$

  - we then use the **context vector** to make output predictions:

$$P(\underline{y}_i) = softmax(W[\underline{s}_i; \underline{c}_i] + \underline{b})$$

– $\underline{c}_i$ will have the same dimension as the hidden representations, independently of the input length
– intuitively, this allows us to focus on the **source** words which will be relevant for the **target translation**

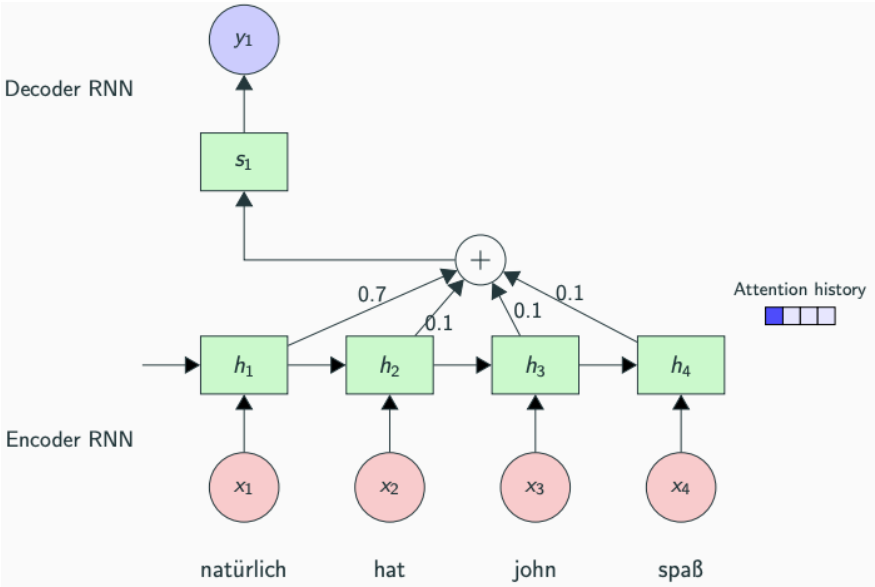


Figure 10: We can see that "natürlich" is the most heavily weighted for the first output, which makes sense, as it translates to "of course", and it will be the start of the target sentence.

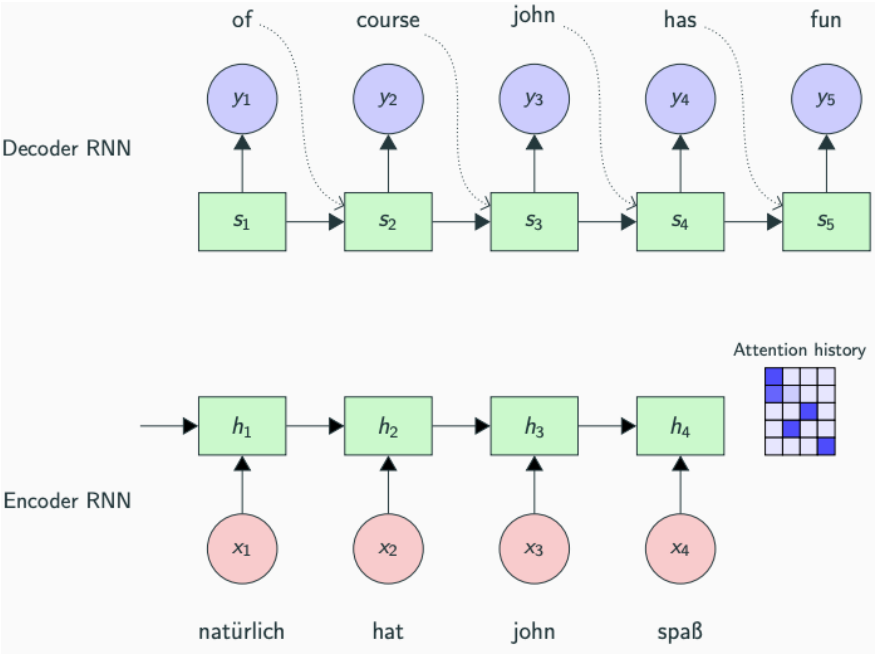


Figure 11: By using attention, we can see that we obtain "alignments" (not technically). Looking at the **attention history**: "natürlich" corresponds to "of course"; "hat" corresponds to "has"; "john" corresponds to "john"; and "spaß" corresponds to "fun".

- **What is a simple way of computing attention?**

- we can compute:

$$a_{ij} = \underline{s}_i \cdot \underline{h}_j \implies \underline{\alpha}_i = softmax(\underline{a}_i)$$

- this gives us an **attention vector**, which is a **distribution** over **hidden representations** in the **encoder**
- by taking the **dot product**, we are using a **similarity measure**: hidden representation $\underline{h}_j$ are favoured when they are similar to $\underline{s}_i$
- for example, we expect that "john"'s hidden representation in the **encoder** and **decoder** to be pretty similar

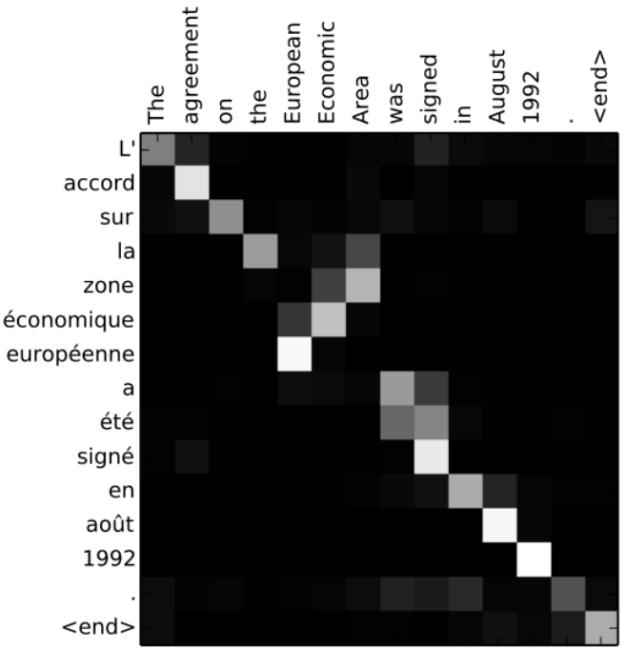


Figure 12: Words in French and English, alongside their corresponding attention score (lighter indicates higher weighting).

- **How can we learn an attention vector?**
  - the **similarity** argument above only works to a certain extent: for example, if the **encoder** and **decoder** have different architectures, their **representations** will likely be extremely different for the same word
  - we need to learn **weights**, so that **attention** can be **fine-tuned**:
    1. **Bilinear Functions**: we can learn a **parameter matrix**, which allows our **encoder-decoder** architectures to be independent (i.e different dimension for representation vector):

$$a_{ij} = \underline{s}_i \cdot (V\underline{h}_j)$$

    2. **FFNs**: we can use a **feed-forward neural network** to learn **attention scores**:

$$a_{ij} = FFN(\underline{s}_i, \underline{h}_j)$$

  - after computing the scores, we pass $\underline{a}_i$ through **softmax** to obtain our **distribution**
  - naturally, this additional **expressivity** comes at the **cost** of having to learn **additional parameters**

- **How does using a bidirectional encoder benefit attention?**
  - in practice, $\underline{h}_i$ doesn't represent w_i, but rather w_i **in the context of** $w_{1:i-1}$
  - however, we are losing information about $w_{i+1:N}$
  - if we use a **bidirectional encoder**, $\underline{h}_i$ contains both the **forward** and **backward** representations, so attention might be better at understanding **contexts**
- **How can attention be improved (beyond learning further weights)?**
  - **Intuitive Priors**: we can improve accuracy of attention by selecting sensible **priors**, based on:
    - **Position Bias**: languages with similar word order should be aligned similarly, so encourage this behaviour for attention.
    - **Markov Condition**: assume that if 2 target words are contiguous, source words should also be contiguous. This discourages large jumps in translation, and encourages local behaviour.
    - **Fertility**: certain words tend to be translated into multiple words ("cats" $\rightarrow$ "les chats"). We can penalise when particular words aren't attended too much (or when they are attended too much), to avoid repeating the same behaviour.
    - **Bilingual Symmetry**: words aligned when translating English to French should be aligned when translating French to English. We can enforce this, by ensuring that the alignment matrices are similar in both directions.
  - **Hard Attention**: instead of **soft attention** (i.e probabilities), use **binary** decisions, to focus on particular contexts
  - **Supervised Attention**: with hand annotated data for alignments, we can train attention models to predict these alignments.

> *Notice, we sometimes call $\underline{s}_i$ the **query**, and the hidden representation $\underline{h}_1, \ldots, \underline{h}_N$ the **keys**.*
> *The idea is that we try to find the **query** which best matches the **key**.*

3 The Transformer Architecture

3.1 Self-Attention

- **What is self-attention?**
  - we have used **attention** to determine which **hidden representations** in the **encoder** are useful for the **decoder**
  - **self-attention** is a **modification** of **attention**, which operates entirely as part of the **encoder**
  - it allows us to learn a **contextualised embedding** for $\underline{x}_i$, by attending to each of the other inputs $\underline{x}_j$
  - these **self-attended** embeddings are powerful enough, that **encoder-decoder** architectures are no longer necessary
  - this is useful: RNNs, being recursive, might be **inefficient**
- **Why is self-attention useful for NLP?**
  - it can gauge different word meanings, based on **context** (i.e "bank" can refer to a financial institution or the land which is next to a river) and incorporate it to known embeddings
  - it can model **dependencies** between words in a sentence (i.e it can enforce subject-verb agreement, gender, etc...)

3.1.1 Basic Self-Attention

- **How can we understand self-attention mathematically?**
  - **self-attention** is nothing but a **linear combination** (weighted sum) of **input vectors**
  - given an **input sequence** of N vectors $\underline{x}_{1:N}$, we can obtain an **output sequence** via:

$$\underline{y}_i = \sum_{j=1}^{N} w_{ij} \underline{x}_j$$

  - a simple way of computing the **weights** is by using a **dot product** of the **inputs**:

$$z_{ij} = \underline{x}_i \cdot \underline{x}_j \implies \underline{w}_i = softmax(\underline{z}_i)$$

  - if 2 inputs are similar (**similar embeddings**, which typically implies **appearing in similar contexts**), then they'll be weighted highly according to the **dot product**

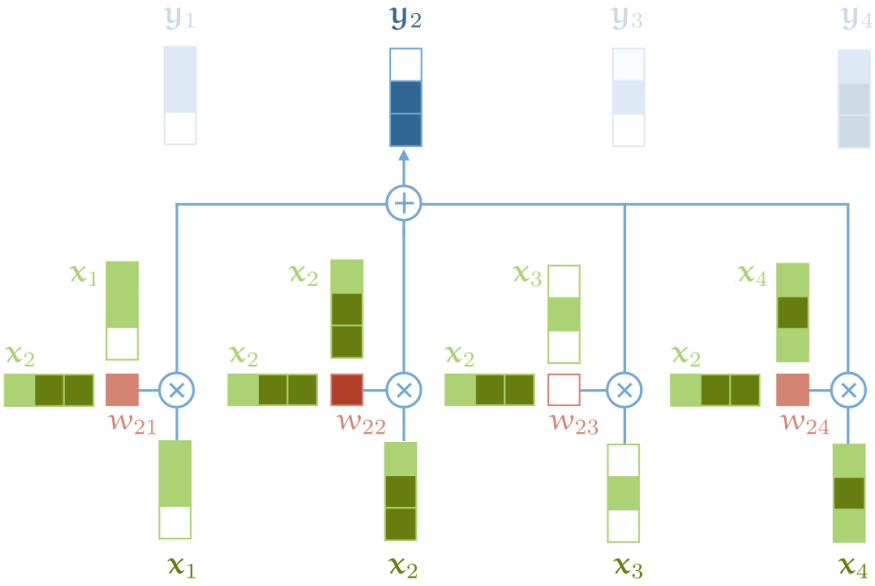


Figure 13: The self attention mechanism. For each input $\underline{x}_i$, we take a **weighted sum** of the context inputs to generate the output $\underline{y}_i$. In red, we have the **attention weights**.

- **What issues does this naive implementation of self-attention have?**
  1. **Lack of Parameters**: what self-attention focuses on is entirely dependent on the **embeddings** learnt; without **trainable parameters**, the attention mechanism won't understand important features
  2. **Input Ordering**: in NLP, **sequence ordering** is imperative for understanding. However, self-attention treats the inputs as a **set**: 2 input sequences with the same input vectors will generate the exact same **weights**, independently of **input order** (this is known as **permutation equivariance**)
- **How can we compute the attention weights efficiently?**
  - say we have N embeddings, such that $\underline{x}_i \in \mathbb{R}^K$
  - we can create an **embedding matrix** $X \in \mathbb{R}^{N \times K}$, with $\underline{x}_i^T$ as the ith row of X

- the matrix:

$$Z = XX^T \in \mathbb{R}^{N \times N}$$

is a symmetric matrix, such that:

$$Z_{ij} = \underline{x}_i \cdot \underline{x}_j = z_{ij}$$

- hence, the ith row/column $\underline{z}_i$ contains the self-attention weights for input $\underline{x}_i$ (before applying softmax)
- if we then want to compute the **output**:

$$Y = softmax(Z)X$$

where $\underline{y}_i^T$ is the ith row of Y

3.1.2 Enhancing Self-Attention: Query, Key and Value Vectors

- **How can we introduce parameters into the self-attention structure?**
  - notice, an individual input vector $\underline{x}_i \in \mathbb{R}^d$ is used three times within the basic **self-attention** mechanism:
    1. **Query Vector**: when computing $w_{i,j} = \underline{x}_i \cdot \underline{x}_j$, the weights for its own output $\underline{y}_i$
    2. **Key Vector**: when computing $w_{j,i} = \underline{x}_j \cdot \underline{x}_i$, the weight of $\underline{x}_i$ for some other output $\underline{y}_j$
    3. **Value Vector**: as a vector in the linear combinations for each output:

$$\underline{y}_j = \sum_{i=1}^{N} w_{j,i} \underline{x}_i$$

  - to make attention more **flexible**, we can learn 3 parameter matrices:
    1. a **query matrix**, $W_q \in \mathbb{R}^{d_k \times d}$
    2. a **key matrix**, $W_k \in \mathbb{R}^{d_k \times d}$
    3. a **value matrix**, $W_v \in \mathbb{R}^{d_v \times d}$
  - this allows us to learn multiple representation for the same inputs, depending on their role
  - for each output, we can think of the **key** vectors as trying to match the **query** vectors, in order to maximise the representation of the **value** vector
- **Mathematically, how do the query, key and value matrices affect the self-attention mechanism?**
  - with the **query matrix**, we can learn a **query vector representation**:

$$\underline{q}_i = W_q \underline{x}_i$$

which is the representation used to extract the attention weights for $\underline{x}_i$ when computing $\underline{y}_i$

  - with the **key matrix**, we can learn a **key vector representation**:

$$\underline{k}_i = W_k \underline{x}_i$$

which is the representation used to extract the attention weights for $\underline{x}_i$ when computing $\underline{y}_j$

  - with the **value matrix**, we can learn a **value vector representation**:

$$\underline{v}_i = W_v \underline{x}_i$$

which is the representation used by $\underline{x}_i$ when computing outputs

– putting all this together:

$$z_{ij} = \underline{q}_i \cdot \underline{k}_j \implies \underline{w}_i = softmax(\underline{z}_i)$$

and the output is:

$$\underline{y}_i = \sum_{j=1}^{N} w_{ij} \underline{v}_j$$

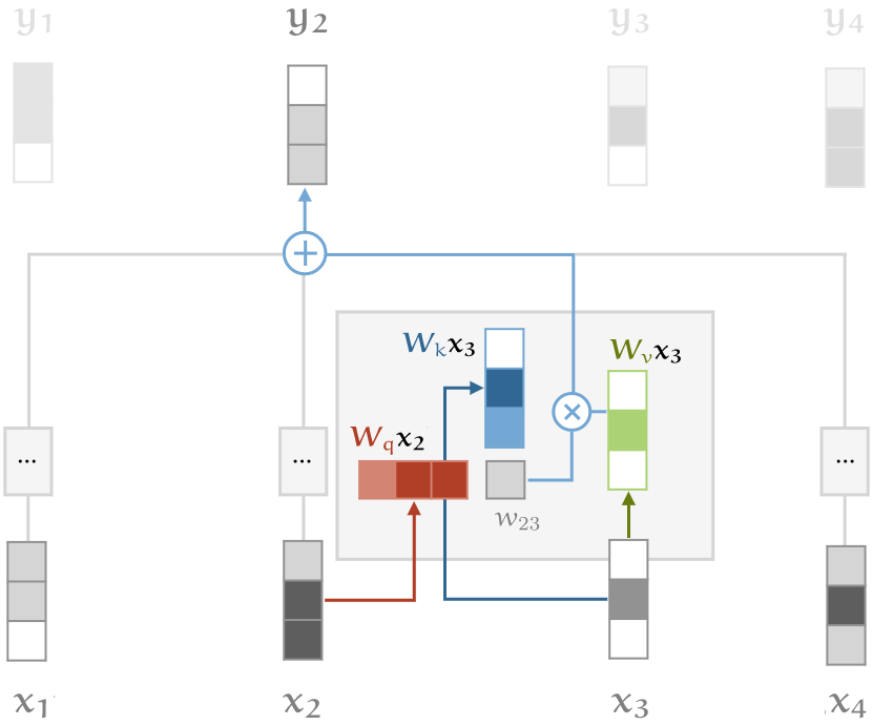


Figure 14: The enhanced self-attention mechanism. In red is the query vector (converting $\underline{x}_2$ into $\underline{q}_2$, for the weights of $\underline{y}_2$); in blue is the key vector (converting $\underline{x}_3$ into $\underline{q}_3$, for the weights of $\underline{y}_2$); and in green is the value vector (converting $\underline{x}_3$ into $\underline{v}_3$, for the weighted sum of $\underline{y}_2$)

- **Why should attention weights be scaled?**
  - much like the **sigmoid**, the **softmax** function can **saturate**: it is sensitive to **large** inputs (this forces all probabilities to be nearly 0)
  - as such, the **attention weights** should be scaled:

$$z_{ij} = \frac{\underline{q}_i \cdot \underline{k}_j}{\sqrt{d_k}}$$

    where d_k is the dimension of the query/key vectors
  - the idea behind this is that the weighted sum is a **dot product**, and these tend to increase with the vector dimension
  - by scaling, we make sure that the inputs to the softmax don't get too large

> *To understand why a factor of $\sqrt{d_k}$ was chosen, consider a vector with c in all its entries. The **magnitude** of such a vector will be $c\sqrt{d_k}$.*

3.1.3 Multi-Head Attention

- **What is multi-head attention?**
  - **self-attention** allows us to **contextualise** embeddings, and helps keep track of **dependencies** between the words in an input
  - however, all this information is compressed into a **single output** $\underline{y}_i$
  - with **multi-head attention**, we can make our model focus on the different parts of an input
  - for example, if we had 3 heads:
    - one head could focus on the subject of a verb
    - another head could focus on the object of a verb
    - the last head could attend to the referents of pronouns

> *For example, consider the phrase:*
>
> *"Mary gave roses to Susan"*
>
> *With **standard** self-attention, we know that x_{mary} and x_{susan} will influence y_{gave} by different amounts (based on their similarity to x_{gave}), but not in different ways.*
> *With **multi-head attention**, we can potentially encode who **gave** the roses and who **received** the roses within y_{gave}.*

Figure 15: An intuitive way of understanding **multi-head attention** is through images. When we look at a picture, we don't immediately focus on the whole picture: we might initially look at a person, then its background, the sides, etc ...

- **How is multi-head self-attention implemented?**
  - if we want to use r **attention heads**, we just need to learn $3r$ parameter matrices:

$$\forall h \in [1, r], \quad W_q^h, W_k^h, W_v^h$$

  - for each **vector input** $\underline{x}_i \in \mathbb{R}^d$, each attention head produces an output $\underline{y}_i^h$ (so each input generates a matrix output $Y_i \in \mathbb{R}^{d \times r}$)
  - the r outputs can then be **concatenated** into a large vector (in $\mathbb{R}^{rd_v}$), which can then be passed through a **linear layer** to recover an **output vector** $\underline{y}_i$:

$$\underline{x}_i \mapsto \{\underline{y}_i^h\}_{h \in [1,r]} \mapsto concat\{\underline{y}_i^h\} \mapsto \underline{y}_i$$

- explicitly, for the ith head let Q_i, K_i, V_i denote the matrices containing the **queries**, **keys** and **values** (so for example the jth row of Q_i is the result of applying W_q^i on the jth input $\underline{x}_j$). Then, **self-attention** for the ith head is computed as:

$$\texttt{self-attention}_i = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

- the **multihead attention** is obtained via:

$$[\texttt{self-attention}_1, \ldots, \texttt{self-attention}_h] W^O$$

where W^O is the output matrix

- it is important to note that in all these computations, **bias** terms could be included (i.e when computing **queries**, **keys**, **values**, and when computing the multihead attention output)

- **What 2 forms of multihead attention are there?**

1. **Narrow Self-Attention**: given an input $\underline{x} \in \mathbb{R}^d$, our attention matrices will be:

$$W_q^h \in \mathbb{R}^{d_k \times \frac{d}{r}}, W_k^h \in \mathbb{R}^{d_k \times \frac{d}{r}}, W_v^h \in \mathbb{R}^{d_v \times \frac{d}{r}}$$

where each W_q^h, W_k^h, W_v^h focuses on a section of $\underline{x}_i$ with d/r elements (so when we concatenate them, we obtain back a vector of dimension d)

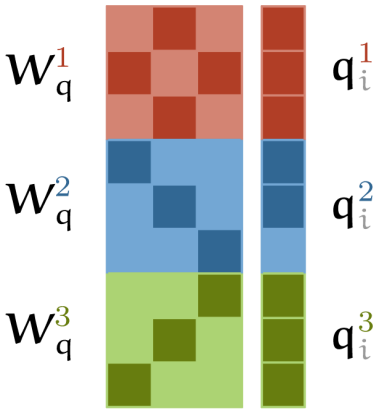


Figure 16: With 3 attention heads acting on a 9 dimensional vector, each of the 3 query matrices focuses on a separate third of the vector. The results then get concatenated back to generate the final query vector.

2. **Wide Self-Attention**: given an input $\underline{x} \in \mathbb{R}^d$, our attention matrices will be:

$$W_q^h \in \mathbb{R}^{d_k \times d}, W_k^h \in \mathbb{R}^{d_k \times d}, W_v^h \in \mathbb{R}^{d_v \times d}$$

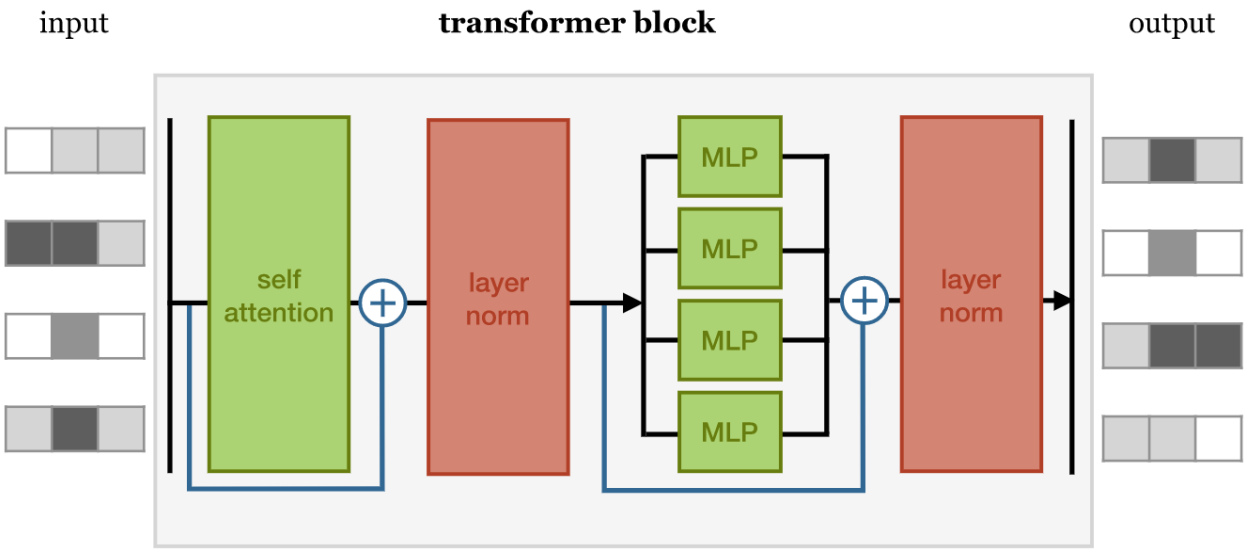
each of the r matrices focus on the **whole input**. Due to the **random weight initalisation**, the different heads should (theoretically) learn to **attend** different parts of the input

- **Which of the 2 multihead architectures is mostly used?**

  - in practice, **wide self-attention** is the most used, since it has more expressive power
  - however, **narrow self-attention** is less expensive to train (we are dealing with smaller matrices), and is useful in the sense that, practically, every word shouldn't be attending to every other word

3.2 Transformers

- **How is multihead self-attention used in transformer blocks?**
  - **transformer blocks** use **multihead self-attention** to create models which are much more **efficient** than traditional RNN architectures
  - a **transformer block** has the following structure:
    1. a **multihead self-attention** layer
    2. a **residual** layer (this adds the input matrix to the output matrix of the self-attention layer)
    3. a layer normalisation layer
    4. a **MLP**, which is applied to each input individually (i.e apply the same MLP to each row vector)
    5. another **residual** layer
    6. a final **layer normalisation**



  - the structure itself isn't too important: all that matters is to pair **self-attention** with the **non-linearities** of the MLP, alongside the benefits to training of **residual** (prevent overly large/small gradients) and **normalisation** layers (faster training)

- **How can input ordering be enforced in the inputs?**
  - up to now, we have simply let $\underline{x}_i$ be the **embedding** of the ith input
  - however, alongside **self-attention**, this produces a **position equivariant model**
  - two common solutions exist:
    1. **Position Embeddings**: create **embeddings**, which represent a particular **position** within the input. This works well, and is easy to implement, but during training we'd need to see sentences of every possible length, to make sure that all position embeddings are trained on.
    2. **Position Encodings**: instead of **learning** position vectors, we pick some function:

$$f : \mathbb{N} \to \mathbb{R}^k$$

       which maps positions within a sentence to a vector. For example, in Attention is All You Need, the authors use:

$$f(pos, i) = \begin{cases} \sin\left(\frac{pos}{10000^{i/k}}\right), & \text{i is } \textbf{even} \\ \cos\left(\frac{pos}{10000^{(i-1)/k}}\right), & \text{i is } \textbf{odd} \end{cases}$$

where:

* $pos \in \mathbb{N}$ denotes the **position** of the input vector within the input sequence
* $i \in [1, k]$ is the index of the component in the input vector

If f is well-chosen, the model should be able to deal with sequences of unseen length, although we don't know how well this will be, and picking a good encoding is a complicated hyperparameter.

- once we have a vector which encodes position (be it through a **position embedding** or a **position encoding**), we add them to the **input vectors**

- **How do transformers differ from RNNs for sequence-to-sequence tasks?**
  - a RNN used **RNN cells** to recursively process an **input sequence** of **arbitrary length**
  - a **transformer** is a neural architecture which uses a set of **transformer blocks** to process sequential input **all at once**
  - however, unlike with RNNs, the input sequence must be **fixed** in length
  - for example, we must enforce that we only process sentences with **at most** 50 words
  - if we have a **longer** sentence, we truncate it; if we have a **shorter** sentence, we add **padding**
  - however, in general **transformers** are more **powerful** and **efficient** than RNNs (since they don't rely on recurrence)

- **How do encoder and decoder transformers differ?**
  - in **encoder** transformers, the **multihead self-attention** operates over the whole **input sequence**
  - in the decoder, there are 2 multihead self-attentions:
    * the first **multihead self-attention** operates over the **outputs** which have been generated until now. A **self-attention** mask is applied, which prevents the self-attention mechanisms to attend to future outputs (more is mentioned below).
    * the second **multihead self-attention** combines the representations from the **encoder**, and the representations generated by the decoder in the previous head. Specifically, it uses the **encoder representations** as **queries** and **keys**, whereas the **output representations** as **values**.

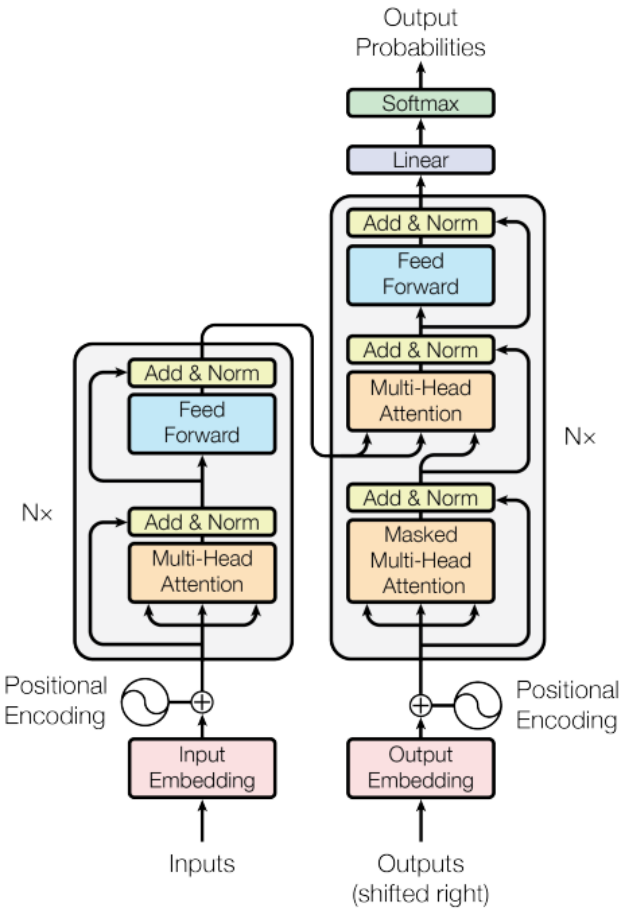


Figure 17: The **encoder transformer** processess all of the inputs using self-attention. The decoder first processes the outputs which have been generated until now, and then combines these with the encoder representations. After a linear layer, softmax is applied, to obtain the output probabilities.

3.2.1 Transformers for Movie Predictions

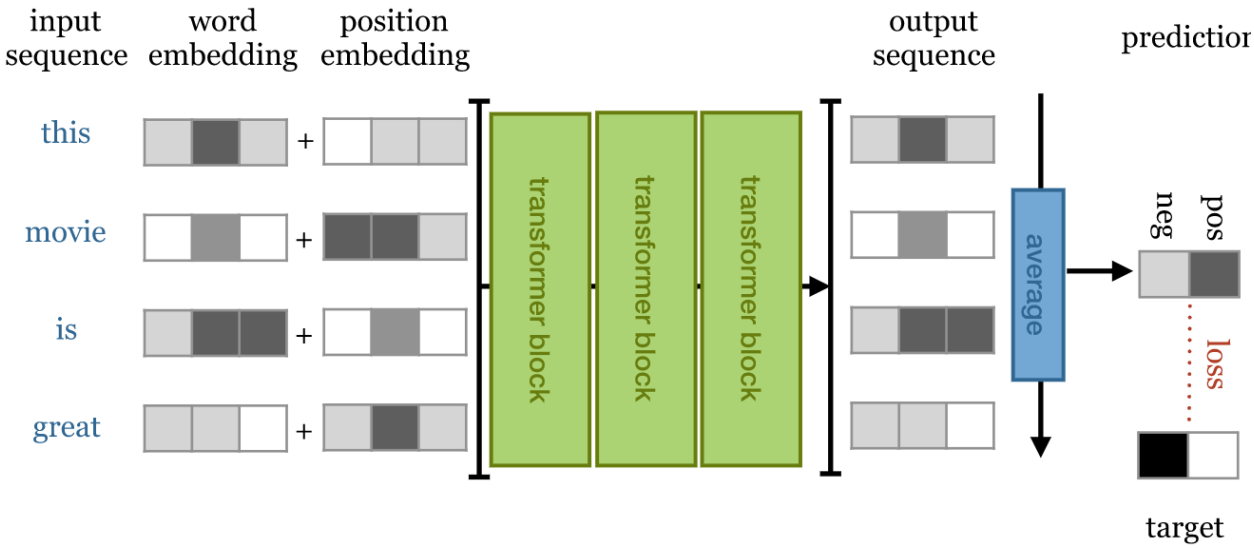


Figure 18: We can use **transformers** to **classify** movie reviews as either positive or negative. To each word embedding we add its position embedding to generate the input sequence. In the last layer, we apply global average pooling to convert the **output sequence** into a single vector.

3.2.2 Transformers for Text Generation

- **What is an autoregressive model?**
  - a model which predicts **future events** from **past ones**
  - for example, **text generation** is **autoregressive**: given $w_{1:t}$, we want to predict w_{t+1}
- **Can transformers be used for autoregressive modelling?**
  - not immediately
  - RNNs could be used directly for language generation, because the tth cell could only "see", the $t-1$ previous cells
  - however, **transformers** can see the whole input, so predicting the next element of the input sequence would be trivial
- **How can transformers be adapted for text generation?**
  - we need to make it so that the **attention weights** are 0 for all words beyond w_t
  - to do this we can use a **mask**:

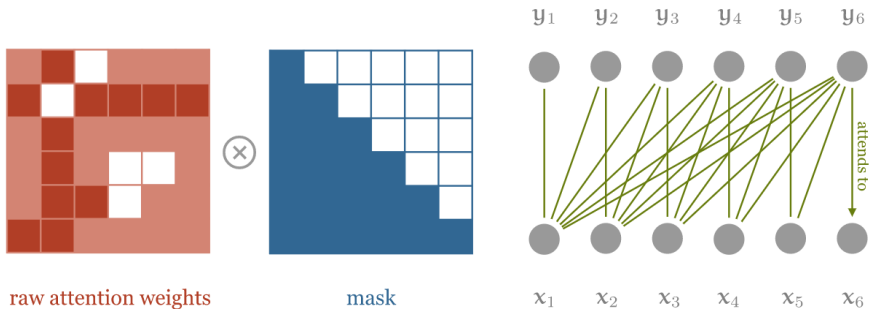
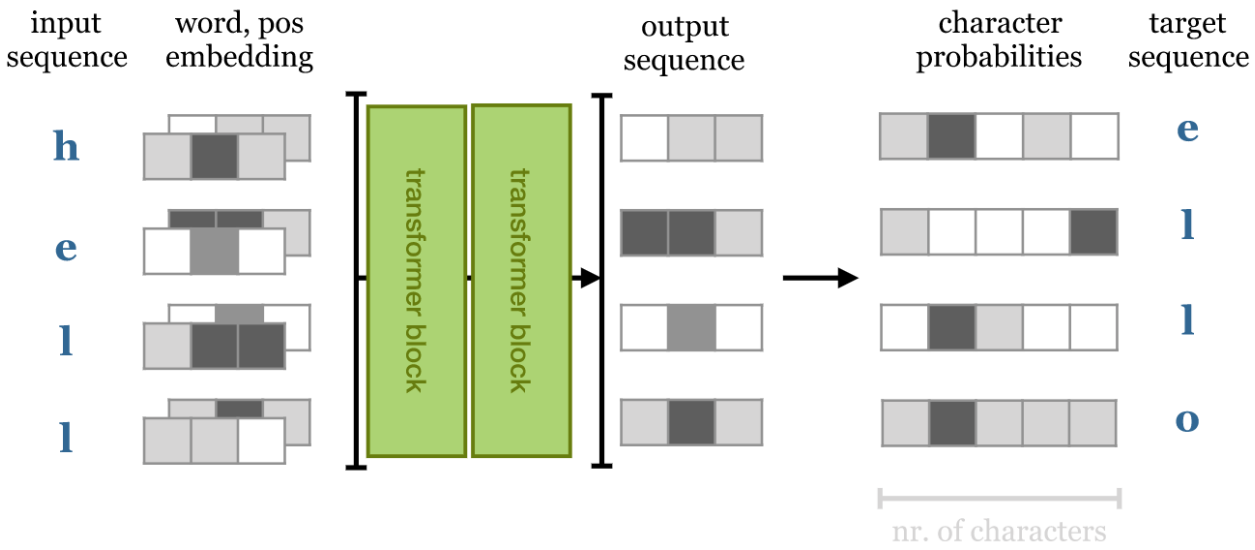


Figure 19: The raw attention weights above the diagonal get multiplied by $-\infty$. When applying softmax, this will then convert them into 0 weights. This makes it so that attention only applies to preceding elements in the sequence.

– once we have the masked attention weights, we can just use a standard **transformer**:



4 Challenges in Neural Machine Translation

- **Large Vocabularies**: neural MT typically struggle with **large vocabularies**: translation is difficult if there are many rare words. One way to ammend this is to use **characters**/**subwords**, or use **translation dictionaries** to handle low-frequency phenomena
- **Optimisation**: we have focused on finding a **target sentence** which is most likely given the **source**; in practice, we care more about the **accuracy** of the generated sentence.
- **Multi-Lingual Learning**: it can be beneficial to train a MT model to translate between many different languages (not just 2). Alternatively, a pre-trained model in 2 languages can be used as a model for 2 different languages.

*Overall, sequence-to-sequence models have many applications beyond **MT**:*

- *dialogue systems*
- *text summarisation*

- *speech recognition*
- *speech synthesis*
- *image captioning*
- *image generation*