Encoder-Decoder Models and Attention

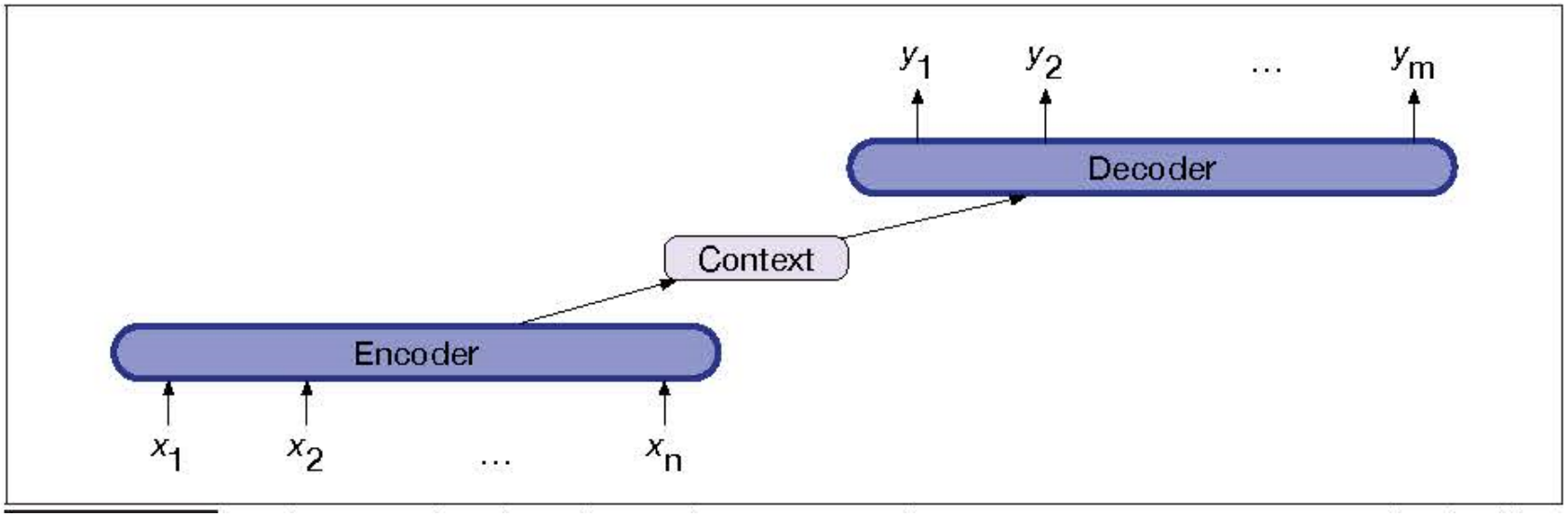
Reference:
- D. Jurafsky and J. Martin, “Speech and Language Processing”

Motivation

- Encoder-decoder networks, or sequence-to-sequence models, are models capable of generating contextually appropriate, arbitrary length, output sequences.
- Have been applied to machine translation, summarization, question answering, dialogue.
- The key idea underlying these networks is the use of an **encoder** network that takes an input sequence.
- Creates a contextualized representation of it, often called the **context**.
- This representation is then passed to a **decoder** which generates a task specific output sequence.

Encoder-Decoder Networks

Fig. 1



Basic architecture for an abstract encoder-decoder network. The **context** is a function of the vector of contextualized input representations and may be used by the decoder in a variety of ways.

Motivation

Encoder-decoder networks consist of three components:

1. An **encoder** that accepts an input sequence, x_1^n, and generates a corresponding sequence of contextualized representations, h_1^n. LSTMs, convolutional networks, and Transformers can all be employed as encoders.
2. A **context vector**, c, which is a function of h_1^n, and conveys the essence of the input to the decoder.
3. A **decoder**, which accepts c as input and generates an arbitrary length sequence of hidden states , h_1^m, from which a corresponding sequence of output states , y_1^m can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

Neural Language Models and Generation Revisited

- Let's begin by describing an encoder-decoder network based on a pair of RNNs.
- Recall the conditional RNN language model for computing $p(y)$, the probability of a sequence y. Like any language model, we can break down the probability as follows:

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1, y_2) \dots p(y_m|y_1, \dots, y_{m-1})$$

- At a particular time t, we pass the prefix of $t - 1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix.
- We then use the final hidden state of the prefix as our starting point to generate the next token.

Neural Language Models and Generation Revisited

- More formally, if g is an activation function like tanh or ReLU, a function of the input at time t and the hidden state at time $t-1$ and f is a softmax over the set of possible vocabulary items, then at time t the output y_t and hidden state h_t are computed as:

$$h_t = g(h_{t-1}, x_t)$$

$$y_t = f(h_t)$$

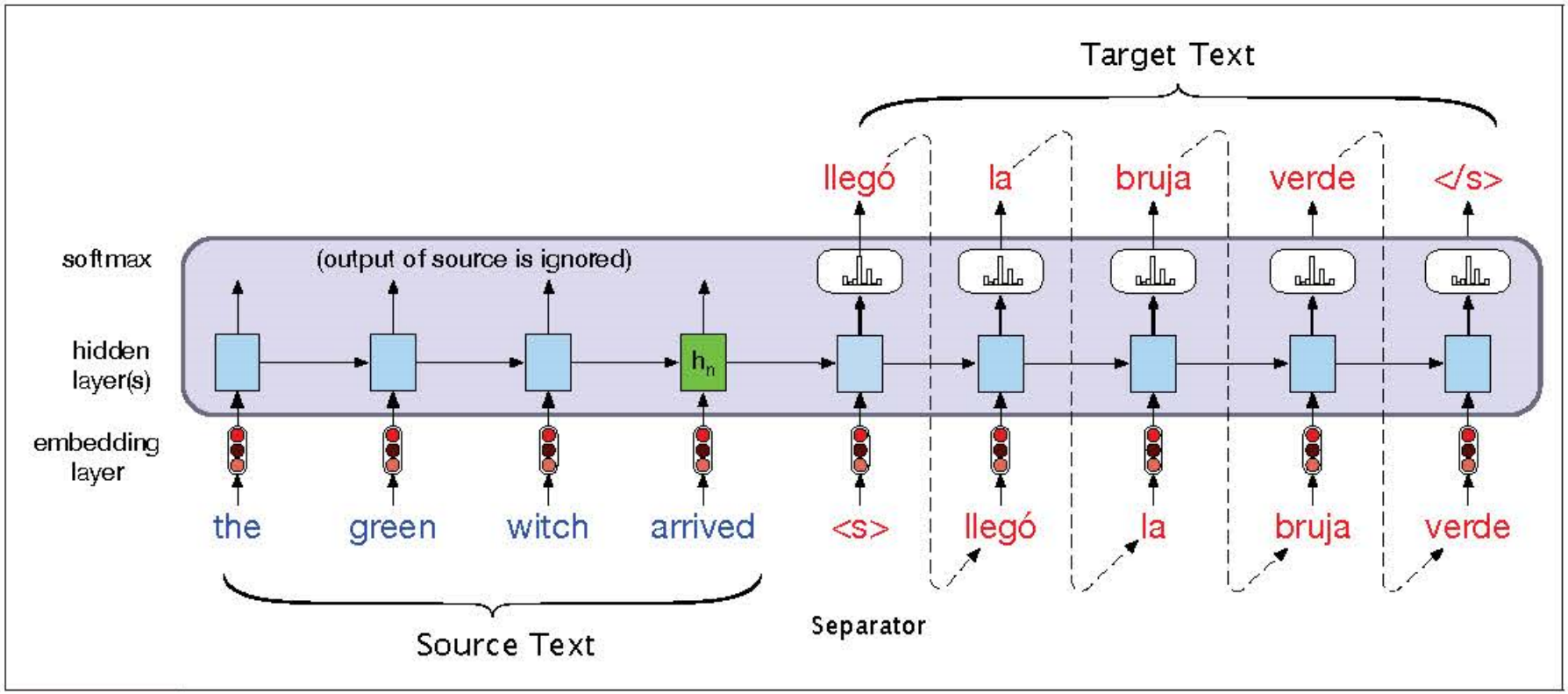
Neural Language Models and Generation Revisited

- We only have to make one slight change to turn this language model with autoregressive generation into a translation model that can translate from a source text target in one language to a target text:
  - Add a sentence separation marker at the end of the source text, and then simply concatenate the target text.
- If we call the source text x and the target text y, we are computing the probability p(yjx) as follows:

$$p(y|x)$$
$$= p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x) \dots p(y_m|y_1,\dots,y_{m-1},x)$$

Neural Language Models and Generation Revisited

Fig. 2



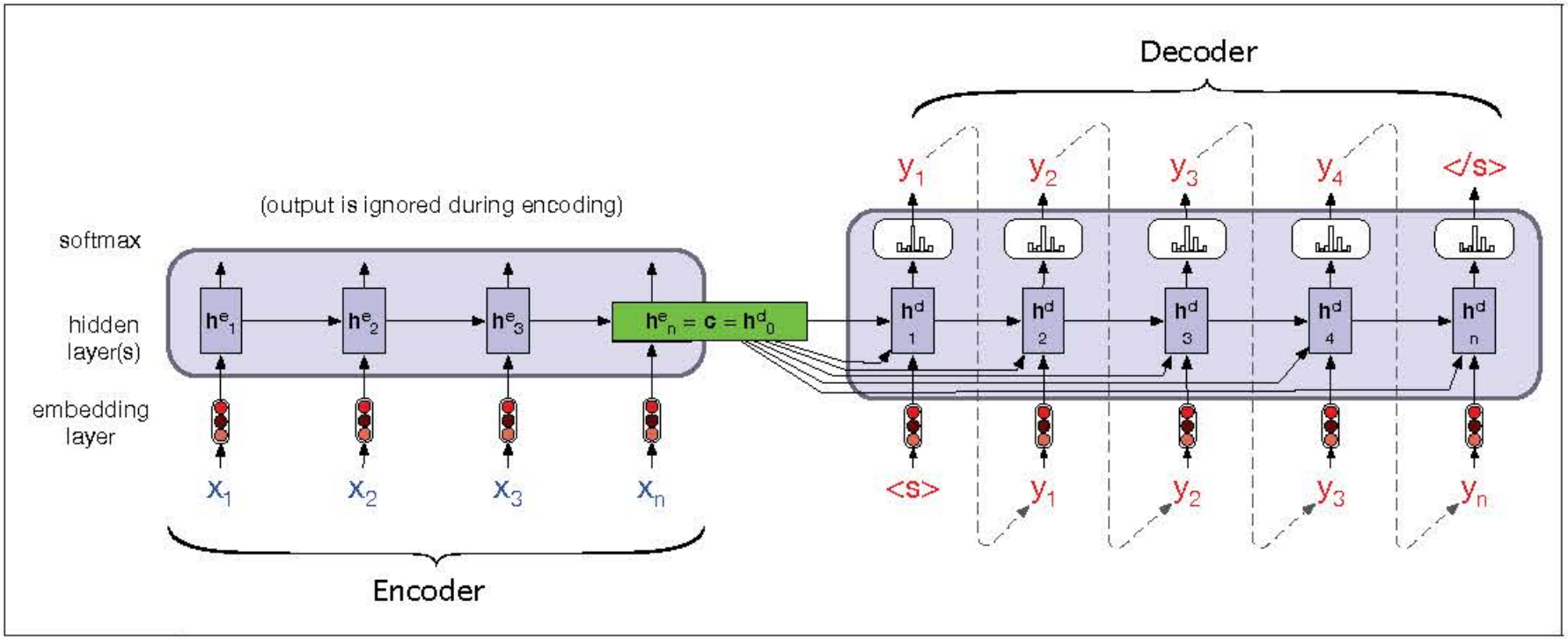
Basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Neural Language Models and Generation Revisited

- To translate a source text, we run it through the network performing forward inference to generate hidden states until we get to the end of the source.
- Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker.
- Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

Neural Language Models and Generation Revisited

Fig. 3



A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, h_n^e, serves as the context for the decoder in its role as h_0^d in the decoder RNN.

Neural Language Models and Generation Revisited

- Let's formalize and generalize this model a bit as shown in the previous diagram (To help keep things straight, we'll use the superscripts e and d where needed to distinguish the hidden states of the encoder and the decoder.)
- The elements of the network on the left process the input sequence x and comprise the encoder.
  - While our simplified figure shows only a single network layer for the encoder, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation.
  - A widely used encoder design makes use of stacked biLSTMs where the hidden states from top layers from the forward and backward passes are concatenated to provide the contextualized representations for each time step.

Neural Language Models and Generation Revisited

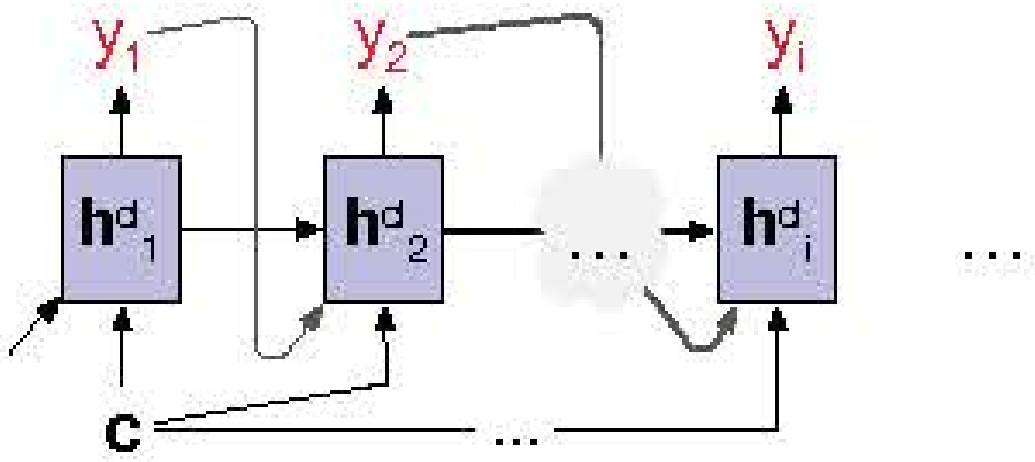
- The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder, h_n^e.
- This representation, also called c for **context**, is then passed to the decoder.
- The decoder network on the right takes this state and uses it to initialize the first hidden state of the decoder. That is, the first decoder RNN cell uses c as its prior hidden state h_0^d.
- The decoder autoregressively generates a sequence of outputs, an element at a time, until an end-of-sequence marker is generated.
- Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.

Encoder-Decoder Networks
Decoder

- One weakness of this approach is that the influence of the context vector c, will wane as the output sequence is generated.
- A solution is to make the context vector c available at each step in the decoding process by adding it as a parameter to the computation of the current hidden state.

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$

Fig. 4



Encoder-Decoder Networks
Decoder

- Recall that g is a stand-in for some flavor of RNN and $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step:

$$c = h_n^e$$
$$h_0^d = c$$
$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$
$$z_t = f(h_t^d)$$
$$y_t = \text{softmax}(z_t)$$

- We compute the most likely output at each time step by taking the argmax over the softmax output:

$$\hat{y}_t = \text{argmax}_{w \in V} P(w | x, y_1, \cdots, y_{t-1})$$

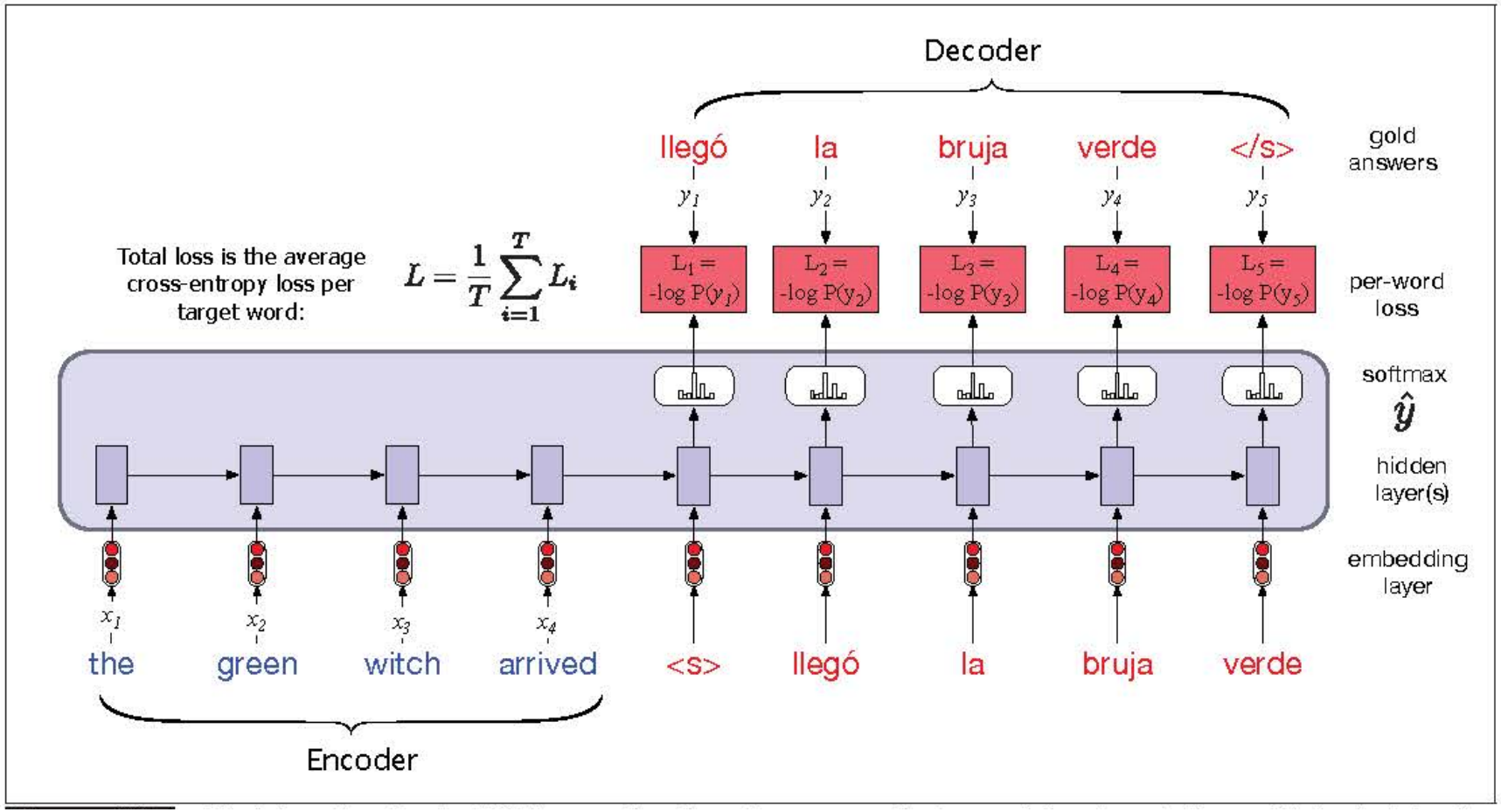
Encoder-Decoder Networks

Training

- Encoder-decoder architectures are trained end-to-end, just as with the RNN language models.
- Each training example is a tuple of paired strings, a source and a target.
- Concatenated with a separator token, these source-target pairs can now serve as training data.
  - For machine translation, the training data typically consists of sets of sentences and their translations.
- Once we have a training set, the training itself proceeds as with any RNN-based language model.
- The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in the next diagram.

Neural Language Models and Generation Revisited

Fig. 5



Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence.

Encoder-Decoder Networks

Training

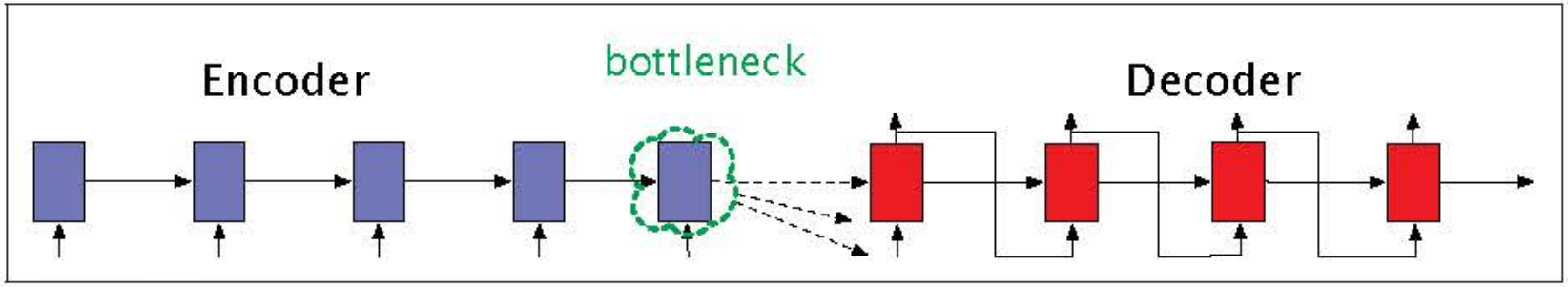
- Note the differences between training and inference with respect to the outputs at each time step.
- The decoder during inference uses its own estimated output $\hat{y}_t$ as the input for the next time step x_{t+1}. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating teacher forcing more tokens.
- In the decoder, teacher forcing means that we force the system to use the gold target token from training as the next input x_{t+1}, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$.

Attention

- The simplicity of the encoder-decoder model is its clean separation of the encoder — which builds a representation of the source text — from the decoder, which uses this context to generate a target text.
- This context vector is h_n, the hidden state of the last (nth) time step of the source text.
- This final hidden state is thus acting as a **bottleneck**:
  - It must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector.
- Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.

Attention

Fig. 6



Requiring the context c to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

- The **attention mechanism** is a way of mechanism allowing the decoder to get information from *all* the hidden states of the encoder, not just the last hidden state.
- In the attention mechanism, as in the vanilla encoder-decoder model, the context vector c is a single vector that is a function of the hidden states of the encoder,

$$c = f(h_1^e, \ldots, h_n^e)$$

Attention

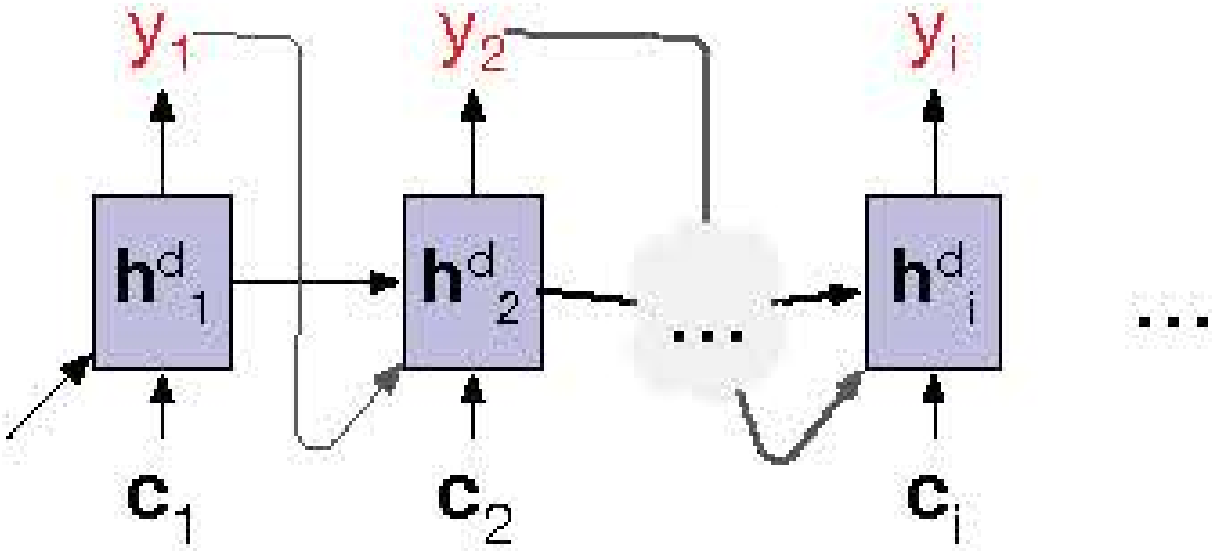
- Because the number of hidden states varies with the size of the input, we can't use the entire tensor of encoder hidden state vectors directly as the context for the decoder.
- The idea of attention is instead to create the single fixed-length vector c by taking a weighted sum of all the encoder hidden states.
- The weights focus on ('attend to') a particular part of the source text that is relevant for the token the decoder is currently producing.
- Attention thus replaces the static context vector with one that is dynamically derived from the encoder hidden states, different for each token in decoding.

Attention

- This context vector, c_i, is generated anew with each decoding step i and takes all of the encoder hidden states into account in its derivation.
- We then make this context available during decoding by conditioning the computation of the current decoder hidden state on it (along with the prior hidden state and the previous output generated by the decoder).

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

Fig. 7



Attention

- The first step in computing c_i is to compute how much to focus on each encoder state, how relevant each encoder state is to the decoder state captured in h_{i-1}^d.
- We capture relevance by computing – at each i during decoding – a score(h_{i-1}^d, h_j^e) for each encoder state j.
- The score **dot product attention** implements relevance as similarity: measuring how similar the decoder hidden state is to an encoder hidden state by computing the dot product:

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

- The result of the dot product is a scalar that reflects the degree of similarity between the two vectors.
- The vector of these scores across over all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

Attention

- To make use of these scores, we'll normalize them with a softmax to create a vector of weights, α_{ij}, that tells us the proportional relevance of each encoder hidden state j to the current decoder state h_{i-1}^{d}.

$$\alpha_{ij} = \text{softmax}(\text{score}(h_{i-1}^{d}, h_{j}^{e}) \; \forall j \in e)$$

$$= \frac{\exp(\text{score}(h_{i-1}^{d}, h_{j}^{e}))}{\sum_{k} \exp(\text{score}(h_{i-1}^{d}, h_{k}^{e}))}$$

Attention

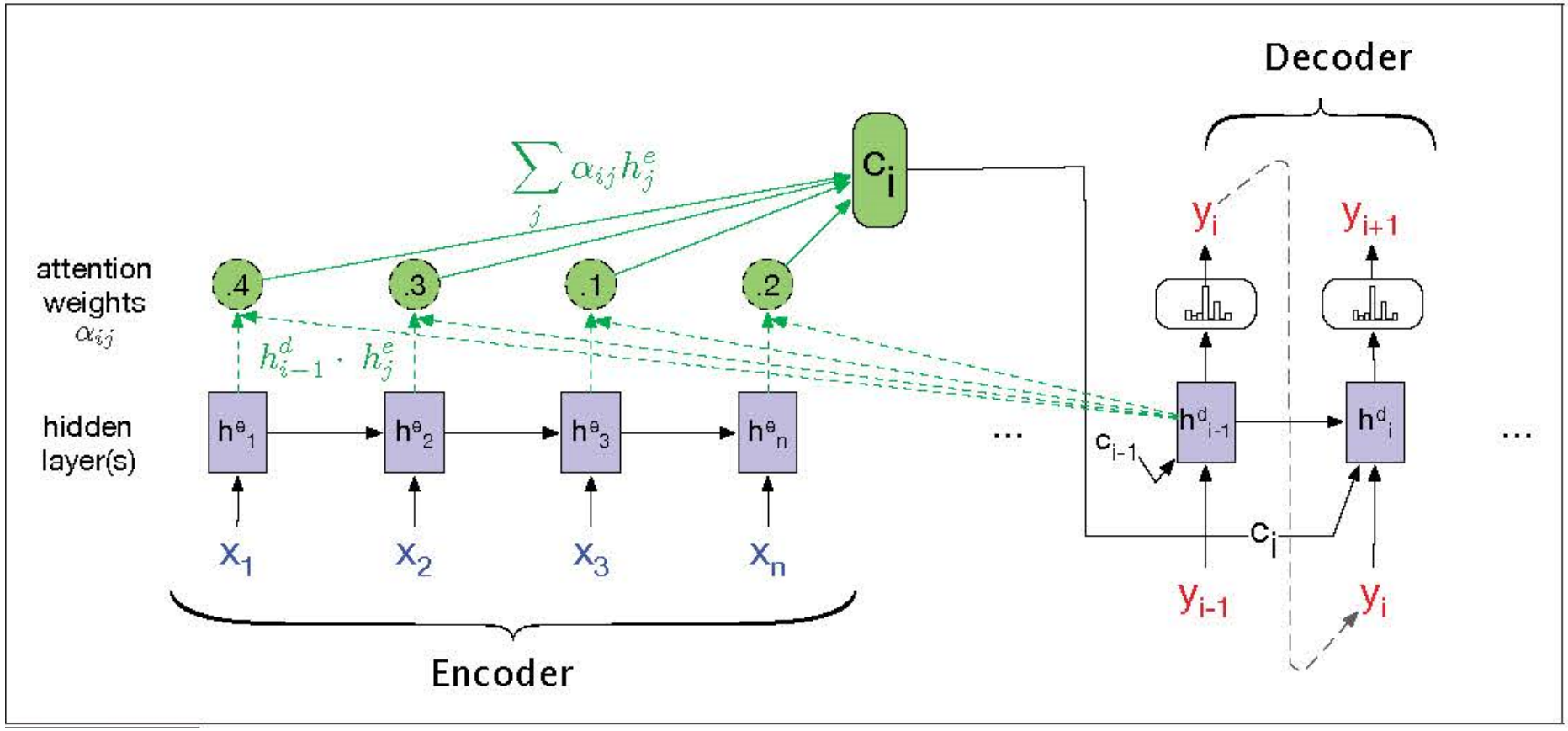
- Finally, given the distribution in α, we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$c_i = \sum_j \alpha_{ij} h_j^e$$

- With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically update to reflect the needs of the decoder at each step of decoding.

Attention

Fig. 8



A sketch of the encoder-decoder network with attention, focusing on the computation of c_i. The context value c_i is one of the inputs to the computation of h_i^d. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state h_{i-1}^d.

Attention

- Instead of simple dot product attention, we can get a more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights, W_s.

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d W_s h_j^e$$

- The weights W_s, which are then trained during the normal end-to-end training, give the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application.
- This bilinear model also allows the encoder and decoder to use different dimensional vectors.

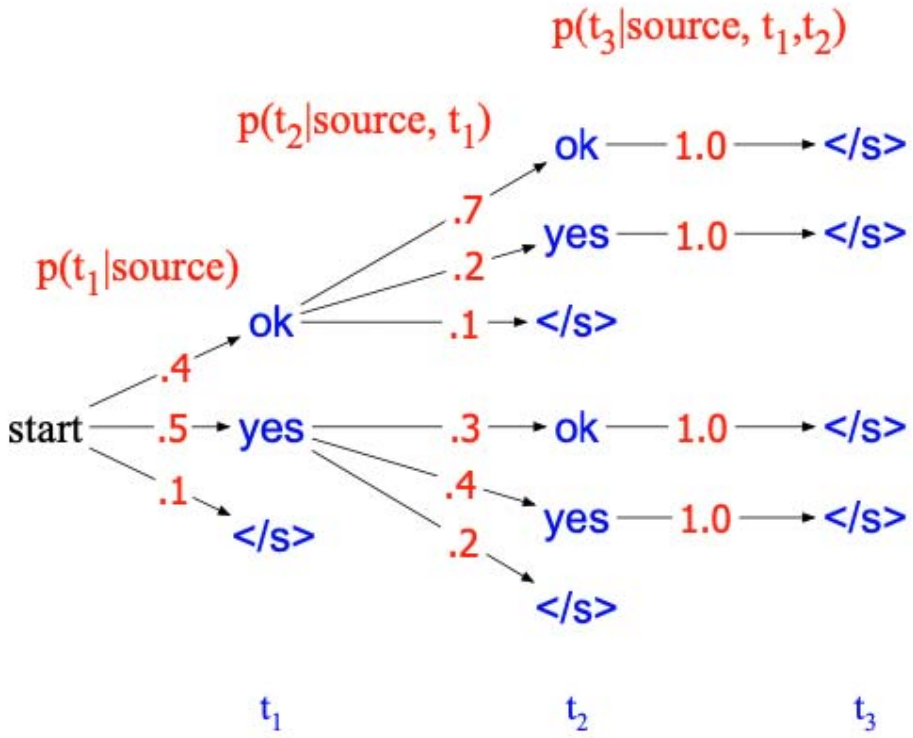
Encoder-Decoder Networks

Beam Search

- Choosing the single most probable token to generate at each step is called greedy decoding; resulting in local optimal
- Consider a graphical representation of the choices the decoder makes in searching for the best output, in which we view the decoding problem as a heuristic state-space search and systematically explore the space of possible outputs.
- In such a **search tree**, the branches are the actions, in this case the action of generating a token, and the nodes are the states, in this case the state of having generated a particular prefix.
- Fig. 4 demonstrates the problem, using a made-up example.

Encoder-Decoder Networks
Beam Search

Fig. 4



A search tree for generating the target string $T = t_1, t_2, \ldots$ from the vocabulary V = {yes, ok, <s>}, given the source string, showing the probability of generating each token from that state. Greedy search would choose *yes* at the first time step followed by *yes*, instead of the globally most probable sequence *ok ok*.

Encoder-Decoder Networks

Beam Search

- Decoding in sequence generation problems (e.g. machine translation) generally uses a method called **beam search**.
- In beam search, instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step.
- This fixed-size memory footprint k is called the **beam width**, on the metaphor of a flashlight beam that can be parameterized to be wider or narrower.

Encoder-Decoder Networks
Beam Search

- At the first step of decoding, we compute a softmax over the entire vocabulary, assigning a probability to each word.
- We then select the k-best options from this softmax output.
- These initial k outputs are the search frontier and these k initial words are called **hypotheses**.
- A hypothesis is an output sequence, a translation-so-far, together with its probability.

Encoder-Decoder Networks

Beam Search

- At subsequent steps, each of the k best hypotheses is extended incrementally by being passed to distinct decoders, which each generate a softmax over the entire vocabulary to extend the hypothesis to every possible next token.
- Each of these k$*$V hypotheses is scored by $\mathrm{P}(y_i|\mathrm{x}, y_{<i})$: the product of the probability of current word choice multiplied by the probability of the path that led to it.
- We then prune the k$*$V hypotheses down to the k best hypotheses, so there are never more than k hypotheses at the frontier of the search, and never more than k decoders.

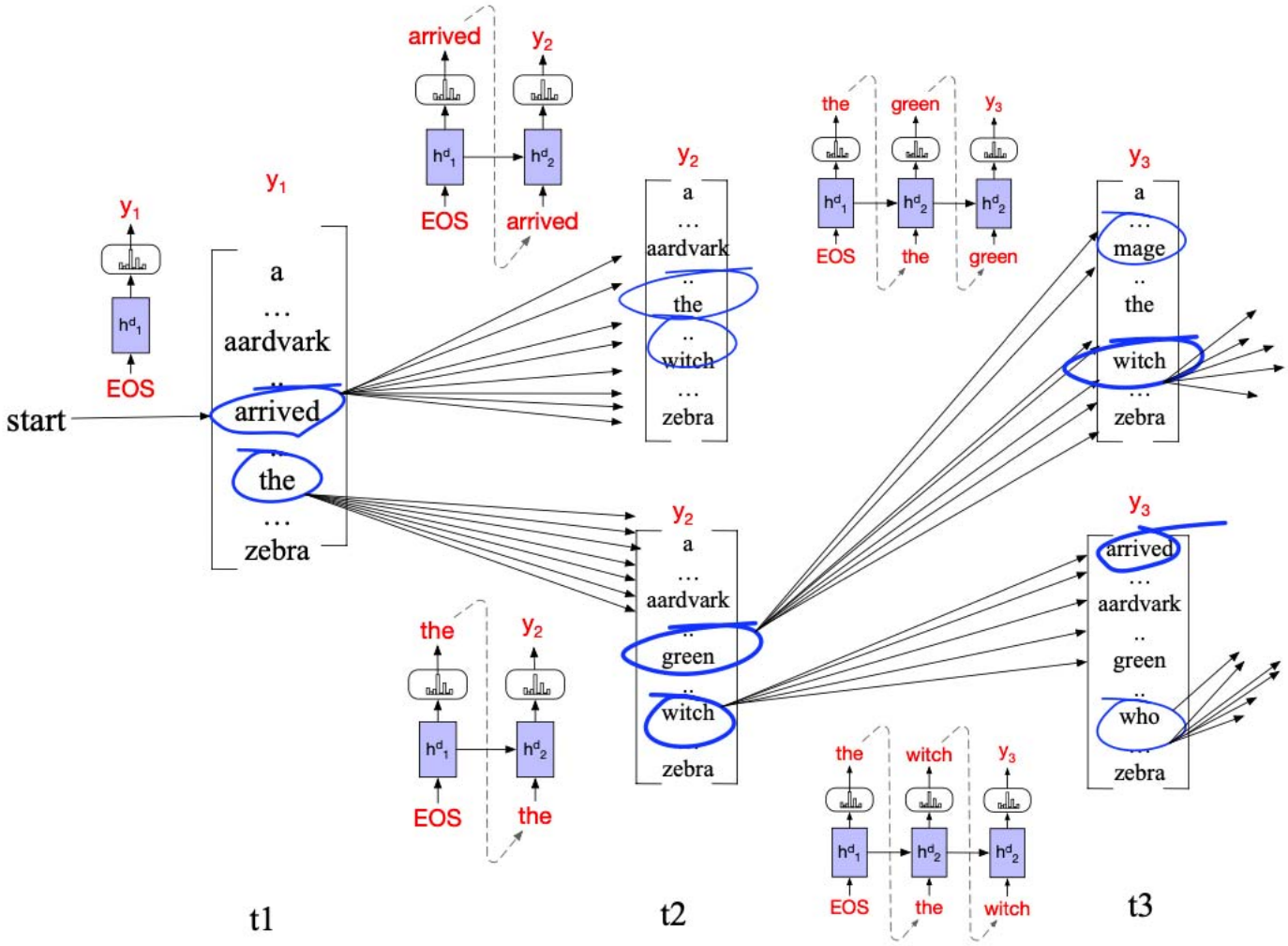
Encoder-Decoder Networks

Beam Search

- This process continues until a <\s> is generated indicating that a complete candidate output has been found.
- At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one.
- The search continues until the beam has been reduced to 0. The result will be k hypotheses.
- Fig. 5 illustrates this process with a beam width of 2.

Encoder-Decoder Networks
Beam Search

Fig. 5



Beam search decoding with a beam width of k = 2.

Encoder-Decoder Networks

Beam Search

- At each time step, we choose the k best hypotheses, compute the V possible extensions of each hypothesis, score the resulting k $*$ V possible hypotheses and choose the best k to continue.
- At time 1, the frontier is filled with the best 2 options from the initial state of the decoder: *arrived* and *the*.
- We then extend each of those, compute the probability of all the hypotheses so far (*arrived the, arrived aardvark, the green, the witch*) and compute the best 2 (in this case *the green* and *the witch*) to be the search frontier to extend on the next step
- On the arcs we show the decoders that we run to score the extension words (although for simplicity we haven't shown the context value c_i that is input at each step).

Encoder-Decoder Networks

Beam Search

- Let's see how the scoring works in detail, scoring each node by its log probability.
- Recall that we can use the chain rule of probability to break down $p(y|x)$ into the product of the probability of each word given its prior context, which we can turn into a sum of logs (for an output string of length t):

$$score(y) = logP(y|x)$$

$$= \log\big(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)\dots P(y_t|y_1,\dots,y_{t-1},x)\big)$$

$$= \sum_{i=1}^{t} \log P(y_i|y_1,\dots,y_{i-1},x)$$

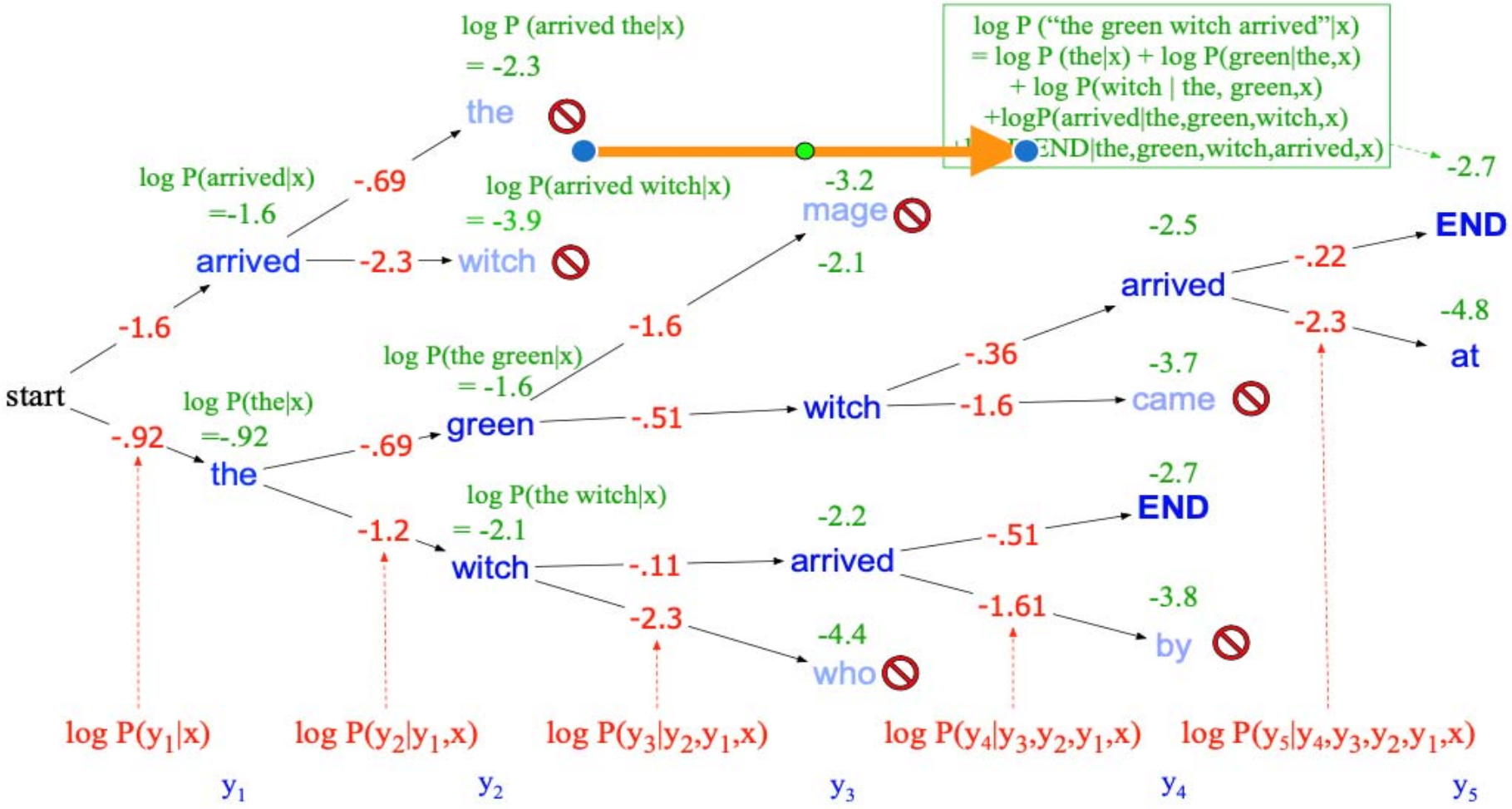
Encoder-Decoder Networks

Beam Search

- At each step, to compute the probability of a partial translation, we simply add the log probability of the prefix translation so far to the log probability of generating the next token.
- Fig. 6 shows the scoring for the example sentence using some simple made-up probabilities.
- Log probabilities are negative or 0, and the max of two log probabilities is the one that is greater (closer to 0).

Encoder-Decoder Networks
Beam Search

Fig. 6



Scoring for beam search decoding with a beam width of k = 2.

Encoder-Decoder Networks

Beam Search

- One problem arises from the fact that the completed hypotheses may have different lengths.
- Because models generally assign lower probabilities to longer strings, a naive algorithm would also choose shorter strings for y.
- This was not an issue during the earlier steps of decoding; due to the breadth-first nature of beam search all the hypotheses being compared had the same length.
- The usual solution to this is to apply some form of length normalization to each of the hypotheses, for example simply dividing the negative log probability by the number of words:

$$score(y) = -logP(y|x) = \frac{1}{T}\sum_{i=1}^{t} -logP(y_i|y_1, \ldots, y_{i-1}, x)$$

Encoder-Decoder Networks

Context

Fig. 6

Beam search decoding.

```
function BEAMDECODE(c, beam_width) returns best paths

  y_0, h_0 ← 0
  path ← ()
  complete_paths ← ()
  state ← (c, y_0, h_0, path)          ;initial state
  frontier ← ⟨state⟩                   ;initial frontier

  while frontier contains incomplete paths and beamwidth > 0
    extended_frontier ← ⟨⟩
    for each state ∈ frontier do
      y ← DECODE(state)
      for each word i ∈ Vocabulary do
        successor ← NEWSTATE(state, i, y_i)
        new_agenda ← ADDTOBEAM(successor, extended_frontier, beam_width)

    for each state in extended_frontier do
      if state is complete do
        complete_paths ← APPEND(complete_paths, state)
        extended_frontier ← REMOVE(extended_frontier, state)
        beam_width ← beam_width - 1
    frontier ← extended_frontier

  return completed_paths

function NEWSTATE(state, word, word_prob) returns new state

function ADDTOBEAM(state, frontier, width) returns updated frontier

  if LENGTH(frontier) < width then
    frontier ← INSERT(state, frontier)
  else if SCORE(state) > SCORE(WORSTOF(frontier))
    frontier ← REMOVE(WORSTOF(frontier))
    frontier ← INSERT(state, frontier)
  return frontier
```

Encoder-Decoder Networks

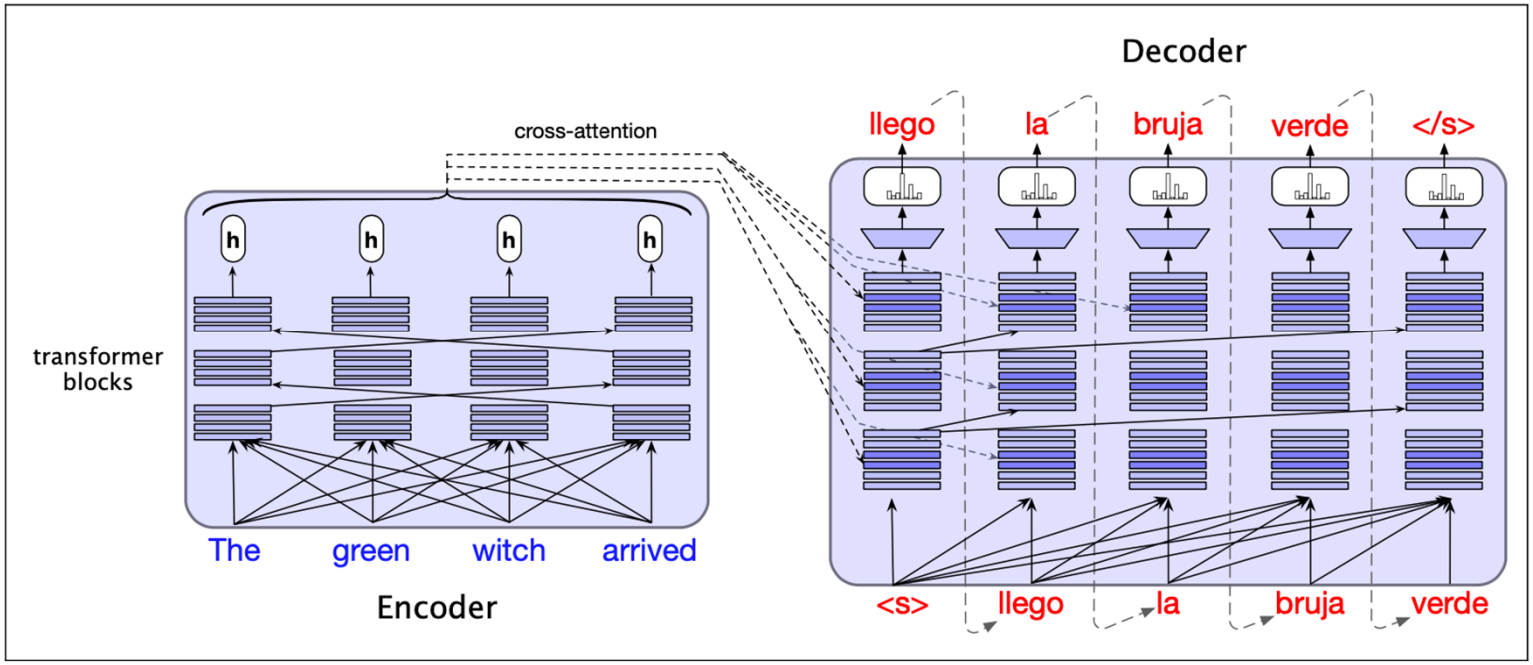
Encoder-Decoder with Transformers

- The encoder-decoder architecture can also be implemented using transformers (rather than RNN/LSTMs) as the component modules.
- It consists of an encoder that takes the source language input words $X = x_1, \cdots, x_T$ and maps them to an output representation $H^{enc} = h_1, \cdots, h_T$; usually via $N = 6$ stacked encoder blocks.
- The decoder, just like the encoder-decoder RNN, is essentially a conditional language model that attends to the encoder representation and generates the target words one by one, at each timestep conditioning on the source sentence and the previously generated target language words.

Encoder-Decoder Networks

Encoder-Decoder with Transformers

Fig. 1



The encoder-decoder architecture using transformer components. The encoder uses the transformer blocks, while the decoder uses a more powerful block with an extra cross-attention layer.

Encoder-Decoder Networks

Encoder-Decoder with Transformers

- In order to attend to the source language, the decoder transformer block includes an extra layer with a special kind of attention, **cross-attention** (also sometimes called **encoder-decoder attention** or **source attention**)
- Recall that the transformer block consists of a self-attention layer that attends to the input from the previous layer, followed by layer norm, a feed forward layer, and another layer norm.
- Cross-attention has the same form as the multi-headed self-attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the encoder.

Encoder-Decoder Networks

Encoder-Decoder with Transformers

- The final output of the encoder $H^{enc} = h_1, \cdots, h_T$ is multiplied by the cross-attention layer's key weights W^K and value weights W^V, but the output from the prior decoder layer $H^{dec[i-1]}$ is multiplied by the cross-attention layer's query weights W^Q :

$$Q = W^Q H^{dec[i-1]}; K = W^K H^{enc}; V = W^V H^{enc}$$

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

Encoder-Decoder Networks

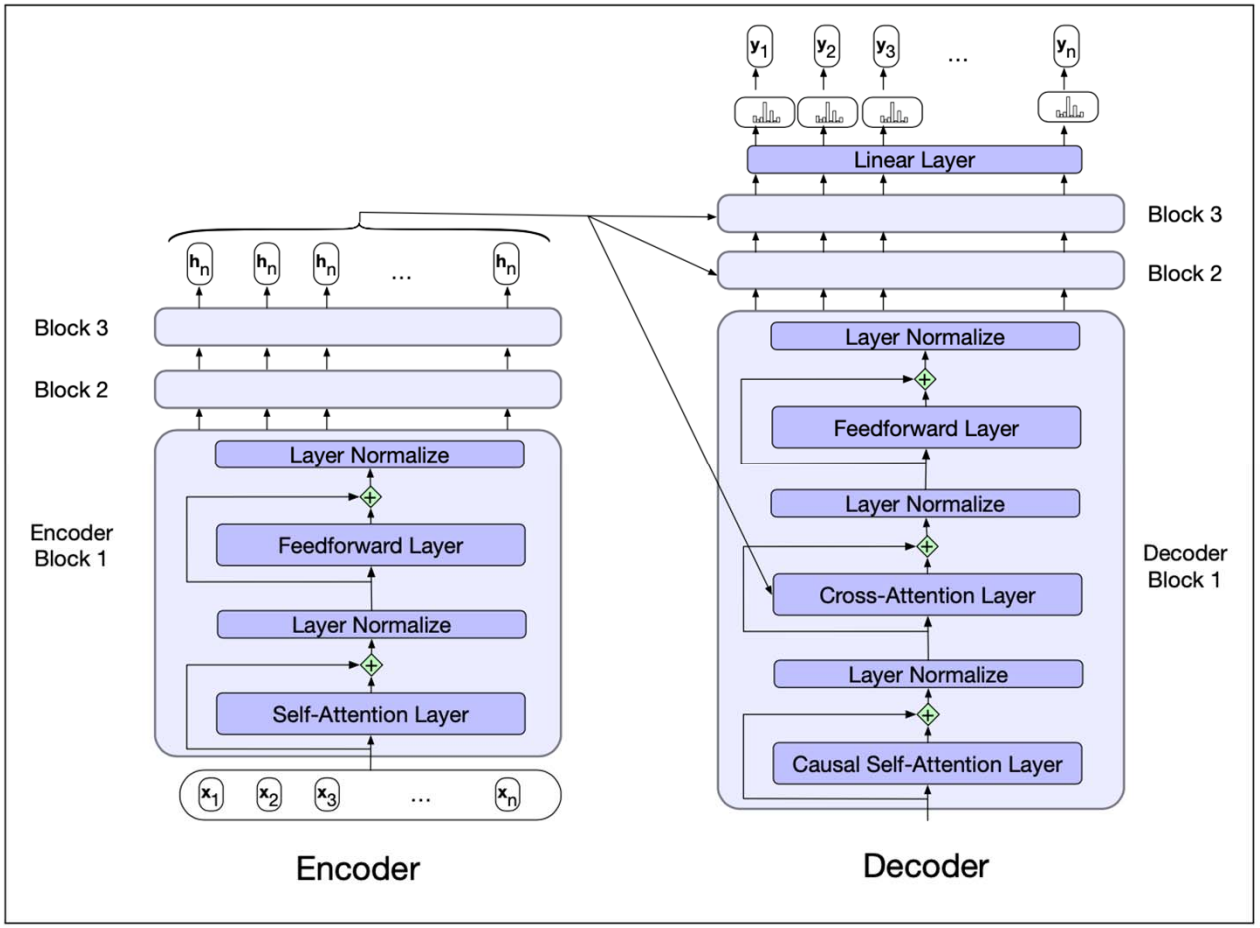
Encoder-Decoder with Transformers

- The cross attention allows the decoder to attend to each of the source language words as projected into the the entire encoder final output representations. The other attention layer in each decoder block, the self-attention layer, is the same causal (left-to-right) self-attention that we saw in Chapter 9. The self-attention in the encoder, however, is allowed to look ahead at the entire source language text.
- In training, just as for RNN encoder-decoders, we use teacher forcing, and train autoregressively, at each time step predicting the next token in the target language, using cross-entropy loss.

Encoder-Decoder Networks

Encoder-Decoder with Transformers

Fig. 2



The transformer block for the encoder and the decoder. Each decoder block has an extra cross-attention layer, which uses the output of the final encoder layer $H^{enc} = h_1, \cdots, h_T$ to produce its key and value vectors.