CSCI 6521

Advanced Machine Learning I

Chapter 07: NLP with RNNs and Attention

Md Tamjidul Hoque

NLP with RNNs and Attention

Here, we want to build a machine that can read and write natural language:

- A common approach for natural language tasks is to use recurrent neural networks (RNNs).
- First, we will build a *character RNN*, trained to predict the next character in a sentence.
  - This will allow us to generate some original text, and in the process, we will see how to build a TensorFlow Dataset on a very long sequence.
  - We will first use a *stateless RNN,* which learns on random portions of text at each iteration, without any information on the rest of the text,
  - then we will build a *stateful RNN,* which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns.

… NLP with RNNs and Attention

- Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie),
  - this time treating sentences as sequences of words, rather than characters.
  - Then we will show how RNNs can be used to build an Encoder–Decoder architecture capable of performing neural machine translation (NMT).
    - For this, we will use the seq2seq API provided by the TensorFlow Addons project.
- Later, we will look at **attention** mechanisms.
  - As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step.
    - First, we will see how to boost the performance of an RNN-based Encoder–Decoder architecture using attention,
    - then we will drop RNNs altogether and look at a very successful attention only architecture called the **Transformer**.
    - Finally, we will look at some of the most important advances in NLP, including incredibly powerful language models such as **GPT-2** and **BERT**, both based on Transformers.

Generating Shakespearean Text Using a Character RNN

Here, we build a simple and fun model that can write like Shakespeare (well, sort of).

- A *char-RNN* can be used to generate novel text, one character at a time.
- Here is a small sample of the text generated by a Char-RNN model after it was trained on all of Shakespeare's work:

> PANDARUS:
>
> Alas, I think he shall be come approached and the day
>
> When little srain would be attain'd into being never fed,
>
> And who is but a chain and subjects of his death,
>
> I should not sleep.

- Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence.
- Next, we build a char-RNN, step by step, starting with the creation of the dataset.

… Generating Shakespearean Text Using a Character RNN

- Here, first, we will create the Training Dataset (**see exercise**).
- Second, we need to split the dataset into –
  - a training set,
  - a validation set, and
  - a test set.
- Also, we can not just shuffle all the characters in the text, so how do you split a sequential dataset?
  - It is very important to avoid any overlap between the training set, the validation set, and the test set.
  - For example, we can take the first 90% of the text for the training set, then the next 5% for the validation set, and the final 5% for the test set.
  - It would also be a good idea to leave a gap between these sets to avoid the risk of a paragraph overlapping over two sets.

… Generating Shakespearean Text Using a Character RNN

- When dealing with time series, we would in general split across time. For example,
  - we might take the years 2000 to 2012 for the training set,
  - the years 2013 to 2015 for the validation set, and
  - the years 2016 to 2018 for the test set.
- However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example,
  - if you have data about the financial health of 10,000 companies from 2000 to 2018, you might be able to split this data across the different companies.
  - It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and
  - if you have correlated companies across the training set and the test set will not be as useful, as its measure of the generalization error will be optimistically biased.

… Generating Shakespearean Text Using a Character RNN

- So, it is often safer to split across time—
  - but this implicitly assumes that the patterns the RNN can learn in the past (in the training set) will still exist in the future.
- For many time series this assumption is reasonable. E.g.,
  - chemical reactions should be fine, since the laws of chemistry don't change every day,
- but for many others it is not. E.g.,
  - financial markets are notoriously not stationary* since patterns disappear as soon as traders spot them and start exploiting them.
- To make sure the time series is indeed sufficiently stationary, we can plot the model's errors on the validation set across time:
  - if the model performs much better on the first part of the validation set than on the last part, then the time series may not be stationary enough, and
  - we might be better off training the model on a shorter time span.

… Generating Shakespearean Text Using a Character RNN

- Lesson:
  - In short, splitting a time series into a training set, a validation set, and a test set is not a trivial task, and
  - how it's done, will depend strongly on the task at hand.

- Chopping the Sequential Dataset into Multiple Windows:
  - The training set now consists of a single sequence of over a million characters, so we can not just train the neural network directly on it:
    - the RNN would be equivalent to a deep net with over a million layers, and we would have a single (very long) instance to train it.
    - Instead, we will use the dataset's window() method to convert this long sequence of characters into many smaller windows of text.
    - Every instance in the dataset will be a fairly short substring of the whole text, and
    - the RNN will be unrolled only over the length of these substrings.
    - This is called *truncated backpropagation through time*.
    - **See exercise**.

Stateful RNN

- Until now, we have used only stateless RNNs:
  - at each training iteration the model starts with a hidden state full of zeros,
  - then it updates this state at each time step, and
  - after the last time step, it throws it away, as it is not needed anymore.
- What if we told the RNN to preserve this final state after processing one training batch and use it as the initial state for the next training batch?
  - This way the model can learn long-term patterns despite only backpropagating through short sequences.
  - This is called a *stateful RNN*.
- Let us see how to build one (**see exercise**).

Tokenize Text

Spaces are not always the best way to tokenize text:

- Think of “San Francisco” or “#ILoveDeepLearning.”

Fortunately, there are better options:

- See paper #1 for an unsupervised learning technique to tokenize and detokenize text at the subword level in a language-independent way, treating spaces like other characters.
- With this approach, even if your model encounters a word, it has never seen before, it can still reasonably guess what it means.
- For example, it may never have seen the word “smartest” during training, but perhaps it learned the word “smart,” and it also learned that the suffix “est” means “the most,” so it can infer the meaning of “smartest.”
- Google’s *SentencePiece* project provides an open-source implementation, described in a paper #2.

… Tokenize Text

- Paper #1 explores creating subword encodings using byte pair encoding.
- TensorFlow team released the TF.Text library in June 2019, which implements various tokenization strategies, including ***WordPiece*** (see paper #2) (a variant of byte pair encoding).
- **Masking** (**see exercise**):
  - **Masking** is a way to tell sequence-processing layers that certain timesteps in an input are missing, and thus should be skipped when processing the data.
- **Reusing Pretrained Embeddings** (**see exercise**)

Encoder-Decoder or Seq2Seq is Multipurpose

- Many NLP tasks can be solved using seq2seq, such as:
  - Neural Machine Translation (e.g., English text to France text),
  - Code Generation (e.g., Natural language to C++ code, see ex1, ex2),
  - Chatbots and Dialogue Systems (e.g., Conversational Agents),
  - Summarization (e.g., Long text to short text), etc.

An Encoder–Decoder Network for Neural Machine Translation

- Let us look at a simple neural machine translation model (see paper #1) that will translate English sentences to French (see Figure 3).

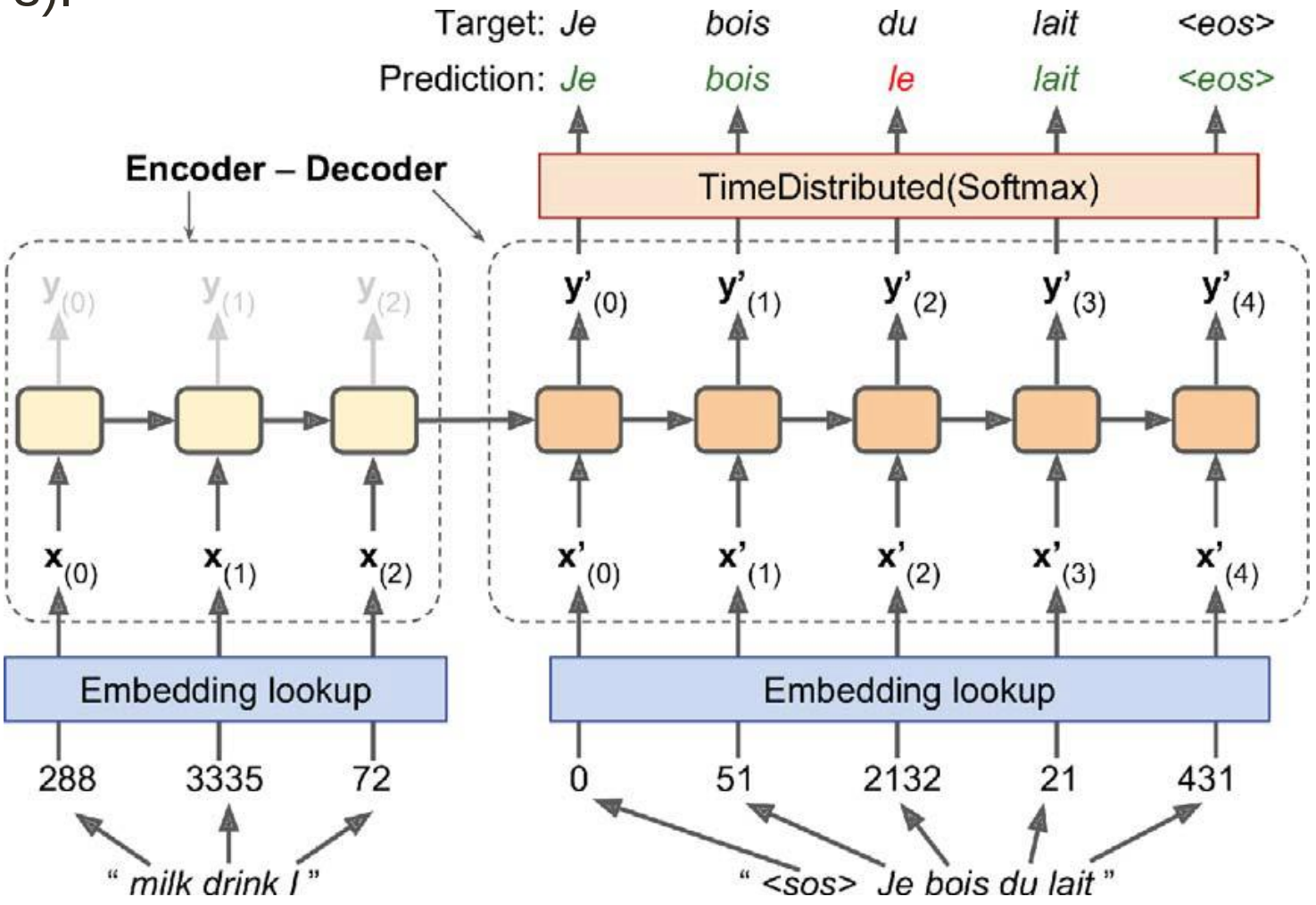

Figure 3: A simple machine translation model.

… An Encoder–Decoder Network for Neural Machine Translation

- In short, the English sentences are fed to the encoder, and the decoder outputs the French translations.
- Note that the French translations are also used as inputs to the decoder but shifted back by one step.
- In other words, the decoder is given as input the word that it *should* have output at the previous step (regardless of what it actually output).
- For the very first word, it is given the start-of-sequence (SOS) token.
- The decoder is expected to end the sentence with an end-of-sequence (EOS) token.

… An Encoder–Decoder Network for Neural Machine Translation

- Note that the English sentences are reversed before they are fed to the encoder.
- For example, “I drink milk” is reversed to “milk drink I.”
- This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that’s generally the first thing that the decoder needs to translate.

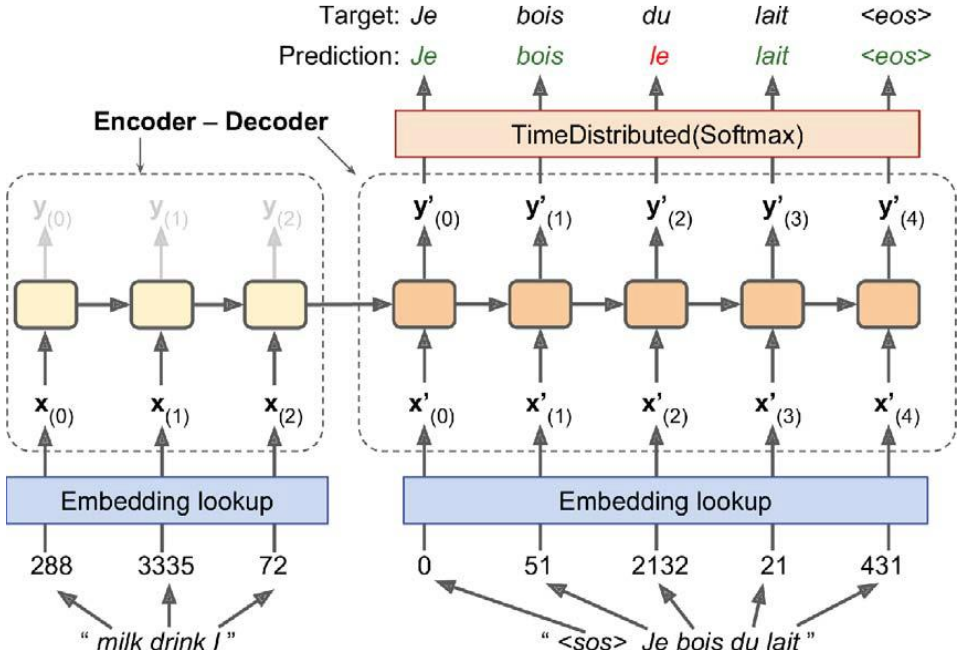

Figure 3: A simple machine translation model.

… An Encoder-Decoder Network for Neural Machine Translation

- Each word is initially represented by its ID (e.g., 288 for the word "milk").

- Next, an embedding layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.

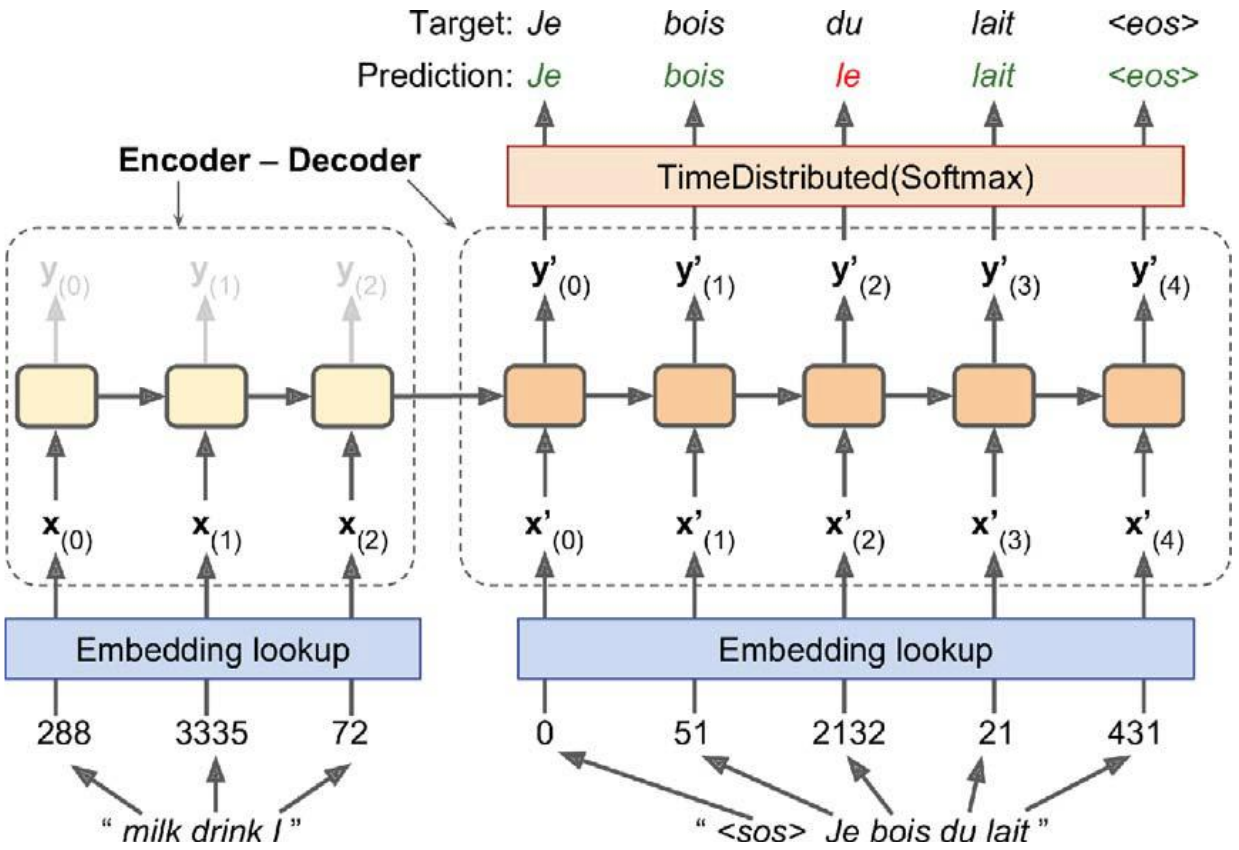

Figure 3: A simple machine translation model.

… An Encoder–Decoder Network for Neural Machine Translation

- At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities.

- For example, at the first step the word “Je” may have a probability of 20%, “Tu” may have a probability of 1%, and so on.

- The word with the highest probability is output. This is very much like a regular classification task, so you can train the model using the "sparse_categorical_crossentropy" loss, much like we did in the Char-RNN model.

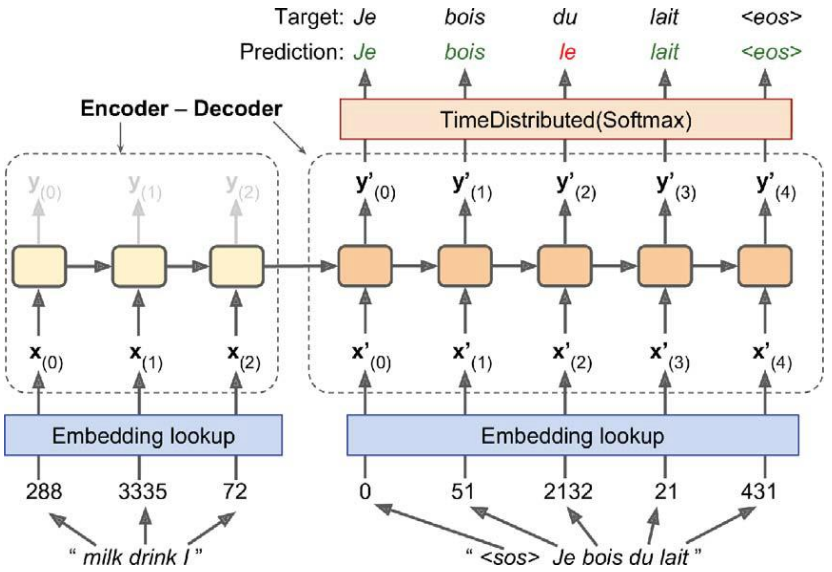

Figure 3: A simple machine translation model.

… An Encoder-Decoder Network for Neural Machine Translation

- Note that at inference time (after training), we will not have the target sentence to feed to the decoder.
- Instead, simply feed the decoder the word that it output at the previous step, as shown in Figure 4 (this will require an embedding lookup that is not shown in the diagram).

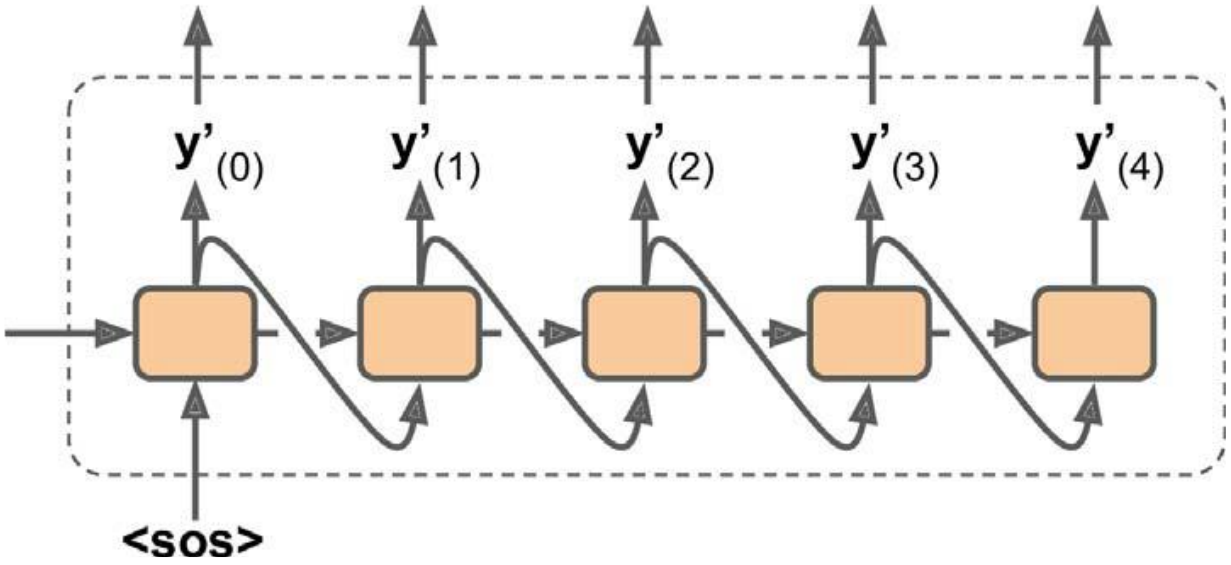

Figure 4: Feeding the previous output word as input at inference time.

… An Encoder–Decoder Network for Neural Machine Translation

Still, there are a few more details to handle if we implement this model:

So far, we have assumed that all input sequences (to the encoder and to the decoder) have a constant length. But obviously sentence lengths vary.

- Since regular tensors have fixed shapes, they can only contain sentences of the same length.
- We can use masking to handle this.
- However, if the sentences have very different lengths, we can't just crop them like we did for sentiment analysis (because we want full translations, not cropped translations).
- Instead, group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on), using padding for the shorter sequences to ensure all sentences in a bucket have the same length (check out the `tf.data.experimental.bucket_by_sequence_length()` function for this).
- For example, "I drink milk" becomes "<pad> <pad> <pad> milk drink I."

… An Encoder–Decoder Network for Neural Machine Translation

We want to ignore any output past the EOS token, so these tokens should not contribute to the loss (they must be masked out).

- For example, if the model outputs “Je bois du lait <eos> oui,” the loss for the last word should be ignored.

When the output vocabulary is large (which is the case here), outputting a probability for each and every possible word would be terribly slow.

- If the target vocabulary contains, say, 50,000 French words, then the decoder would output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive.
- To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words,
- then compute an approximation of the loss based only on these logits.
- This sampled softmax technique was introduced in paper #1.
- In TensorFlow we can use the `tf.nn.sampled_softmax_loss()` function for this during training and use the normal softmax function at inference time (sampled softmax cannot be used at inference time because it requires knowing the target) [**see exercise**].

Bidirectional RNNs

- At each time step, a regular recurrent layer only looks at past and present inputs before generating its output.
- In other words, it is “causal,” meaning it cannot look into the future.
- This type of RNN makes sense when forecasting time series, but for many NLP tasks, such as Neural Machine Translation, it is often preferable to look ahead at the next words before encoding a given word.
- For example, consider the phrases
  - “the Queen of the United Kingdom,”
  - “the queen of hearts,” and
  - “the queen bee”:
  - to properly encode the word “queen,” we need to look ahead.
- To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left.

… Bidirectional RNNs

- Then simply combine their outputs at each time step, typically by concatenating them. This is called a ***bidirectional recurrent layer*** (see Figure 5). [**see exercise (**see: Bidirectional Recurrent Layers**)**]

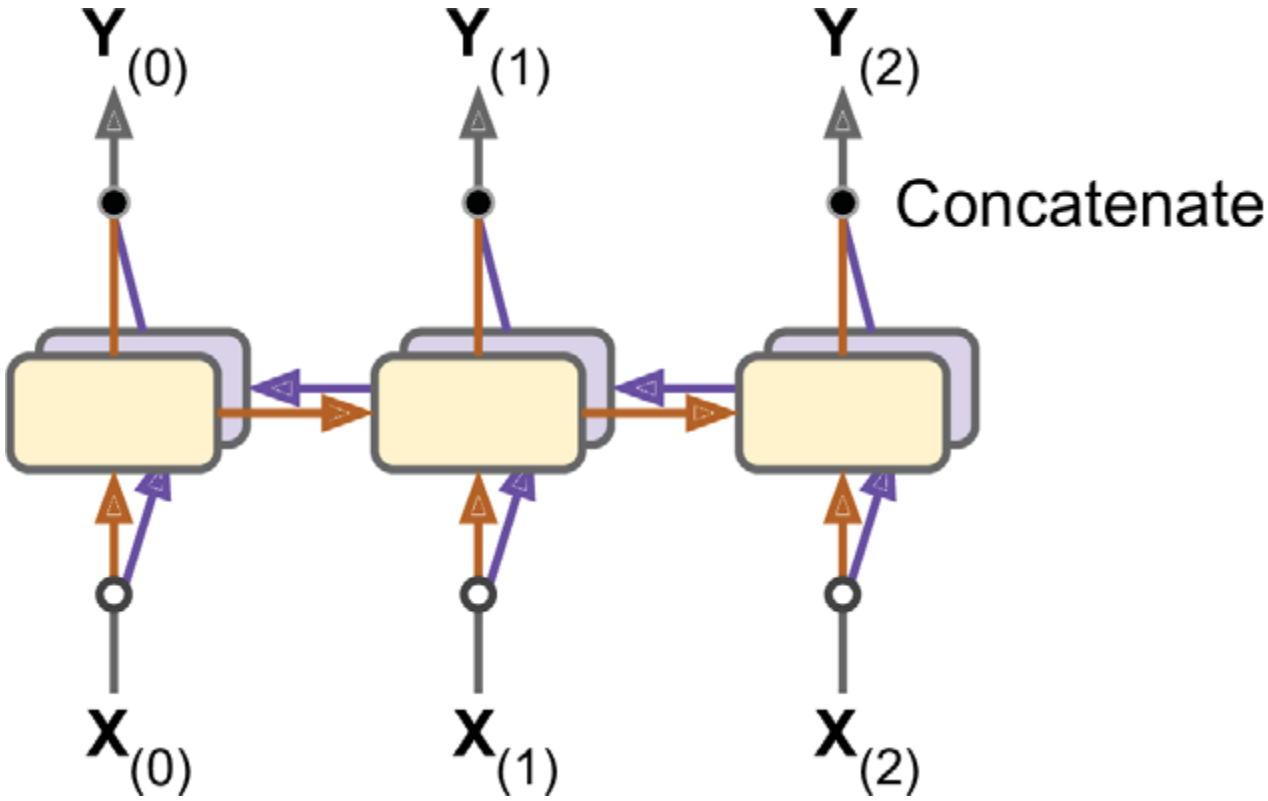

Figure 5: A bidirectional recurrent layer.

Beam Search

- Suppose we train an Encoder–Decoder model, and
- use it to translate the French sentence “Comment vas-tu?” to English.
- We are hoping that it will output the proper translation (“How are you?”), but unfortunately it outputs “How will you?”
- Looking at the training set, we notice many sentences such as “Comment vas-tu jouer?” which translates to “How will you play?”
- So it wasn’t absurd for the model to output “How will” after seeing “Comment vas.”
- Unfortunately, in this case it was a mistake, and the model could not go back and fix it, so it tried to complete the sentence as best it could.
- By greedily outputting the most likely word at every step, it ended up with a suboptimal translation.

… Beam Search

- How can we give the model a chance to go back and fix mistakes it made earlier?
- One of the most common solutions is ***beam search***:
  - it keeps track of a short list of the ***k*** most promising sentences (say, the top three), and
  - at each decoder step it tries to extend them by one word, keeping only the k most likely sentences.
  - The parameter ***k*** is called the ***beam width***.
- For example, suppose you use the model to translate the sentence "Comment vas-tu?" using beam search with a beam width of 3.
- At the first decoder step, the model will output an estimated probability for each possible word.
- Suppose the top three words are "How" (75% estimated probability), "What" (3%), and "You" (1%). That's our short list so far.

… Beam Search

- Next, we create three copies of our model and use them to find the next word for each sentence.
- Each model will output one estimated probability per word in the vocabulary.
- The first model will try to find the next word in the sentence "How," and perhaps it will output a probability of 36% for the word "will," 32% for the word "are," 16% for the word "do," and so on.
- Note that these are actually conditional probabilities, given that the sentence starts with "How." The second model will try to complete the sentence "What";
- it might output a conditional probability of 50% for the word "are," and so on.
- Assuming the vocabulary has 10,000 words, each model will output 10,000 probabilities.

… Beam Search

- Next, we compute the probabilities of each of the 30,000 two-word sentences that these models considered (3 × 10,000).
- We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes.
- For example, the estimated probability of the sentence "How" was 75%, while the estimated conditional probability of the word "will" (given that the first word is "How") was 36%, so the estimated probability of the sentence "How will" is 75% × 36% = 27%.
- After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. Perhaps they all start with the word "How": "How will" (27%), "How are" (24%), and "How do" (12%).
- Right now, the sentence "How will" is winning, but "How are" has not been eliminated.

… Beam Search

Then we repeat the same process:

- we use three models to predict the next word in each of these three sentences, and
- we compute the probabilities of all 30,000 three word sentences we considered.
- Perhaps the top three are now “How are you” (10%), “How do you” (8%), and “How will you” (2%).
- At the next step we may get “How do you do” (7%), “How are you <eos>” (6%), and “How are you doing” (3%).
- Notice that “How will” was eliminated, and we now have three perfectly reasonable translations.
- We boosted our Encoder–Decoder model’s performance without any extra training, simply by using it more wisely.

… Beam Search

- You can implement beam search fairly easily using TensorFlow Addons:

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

- We first create a BeamSearchDecoder, which wraps all the decoder clones (in this case 10 clones).
- Then we create one copy of the encoder's final state for each decoder clone, and
- we pass these states to the decoder, along with the start and end tokens.

… Beam Search

- With all this, we can get good translations for fairly short sentences (especially if we use pretrained word embeddings).
- Unfortunately, this model will be really bad at translating long sentences.
- Once again, the problem comes from the limited short-term memory of RNNs.
- **Attention** mechanisms are the game-changing innovation that addressed this problem.

Attention Mechanisms

- Consider the path from the word “milk” to its translation “lait” in Figure 3: it is quite long!
- This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

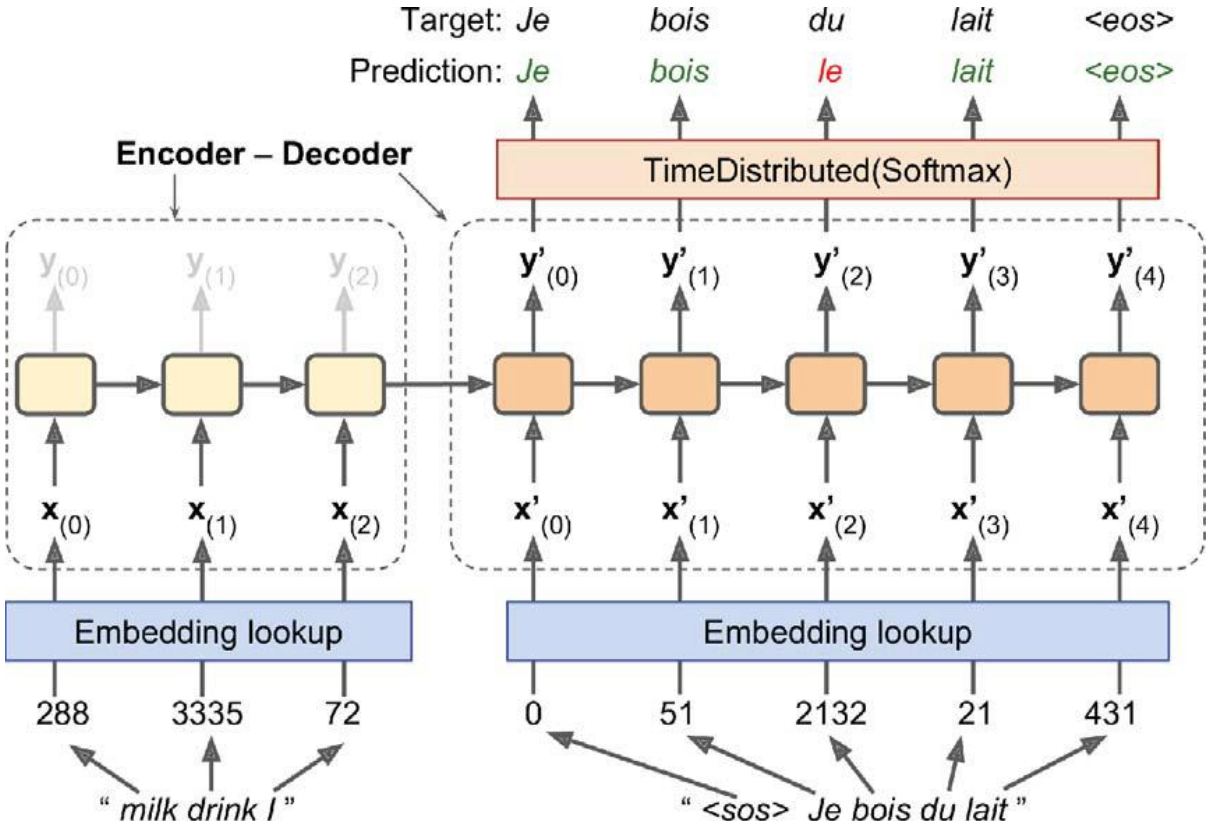

Figure 3: A simple machine translation model.

… Attention Mechanisms

- The groundbreaking **paper#1** introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step.

- For example, at the time step where the decoder needs to output the word “lait,” it will focus its attention on the word “milk.”

- This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact.

- **Attention mechanisms** revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words)†.

… Attention Mechanisms

- Figure 6 shows (slightly simplified version of) this model's architecture.

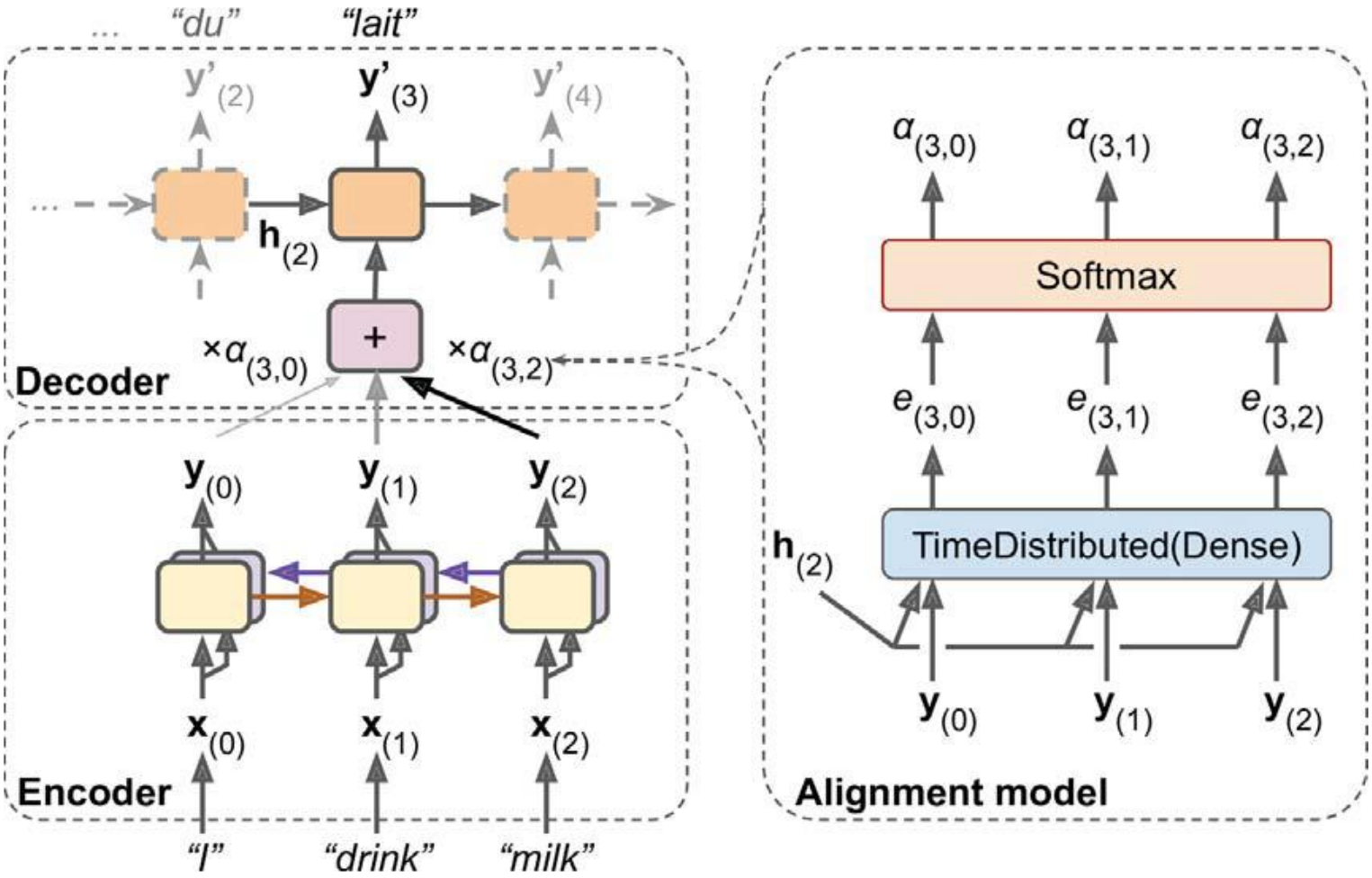

Figure 6: Neural machine translation using an Encoder–Decoder network with an attention model

… Attention Mechanisms

- On the left of Figure 6, we have the encoder and the decoder.
- Instead of just sending the encoder's final hidden state to the decoder (which is still done, although it is not shown in the figure), we now send all of its outputs to the decoder.
- At each time step, the decoder's memory cell computes a weighted sum of all these encoder outputs:
  - this determines which words it will focus on at this step.
- The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step.
- For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to word number 2 ("milk") than to the other two words, at least at this time step.

… Attention Mechanisms

- The rest of the decoder works just like earlier:
  - at each time step the memory cell receives the inputs we just discussed,
  - plus, the hidden state from the previous time step, and
  - finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

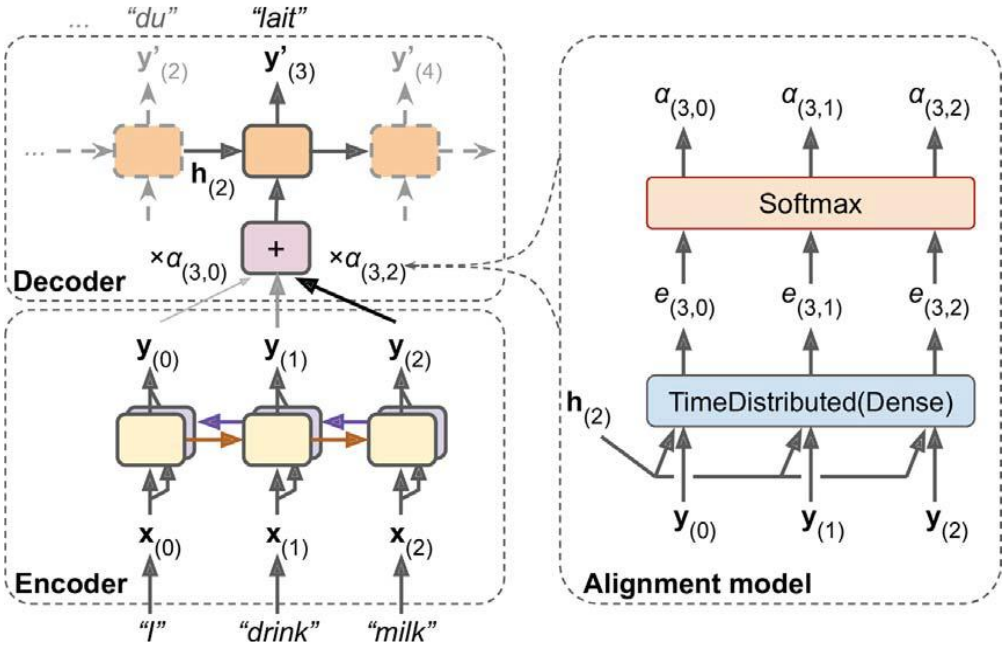

Figure 6: Neural machine translation using an Encoder–Decoder network with an attention model

… Attention Mechanisms

- But where do these $\alpha_{(t,i)}$ weights come from?
- It's actually pretty simple:
  - they are generated by a type of small neural network called an alignment model (or an attention layer), which is trained jointly with the rest of the Encoder–Decoder model.
  - This alignment model is illustrated on the righthand side of Figure 6.

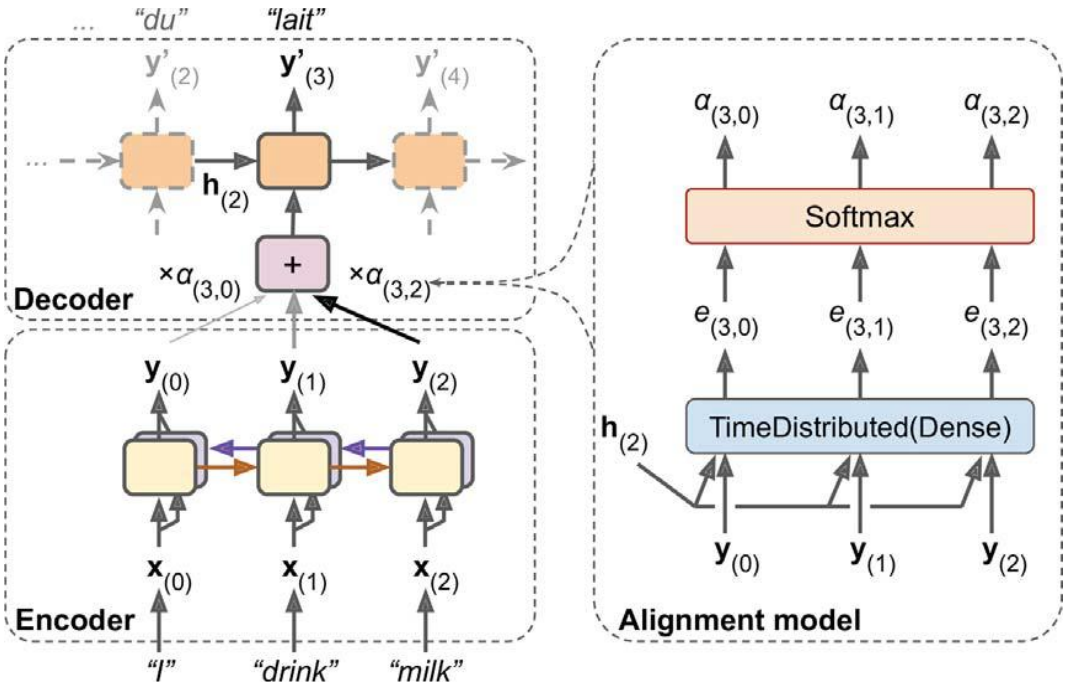

Figure 6: Neural machine translation using an Encoder–Decoder network with an attention model

… Attention Mechanisms

- It starts with a **time-distributed Dense layer**† with a single neuron, which receives as input: all the encoder outputs, concatenated with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$).
- This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$):
  - this score measures how well each output is aligned with the decoder's previous hidden state.
- Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$).
- All the weights for a given decoder time step add up to 1 (since the softmax layer is not time-distributed).
- This particular attention mechanism is called ***Bahdanau attention*** (named after the paper's first author).
- Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called ***concatenative attention*** (or ***additive attention***).

… Attention Mechanisms

- **Complexity**: If the input sentence is ***n*** words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights.

- Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism was proposed in **paper#1**.

- Because the goal of the attention mechanism is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the dot product of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it much faster.

- For this to be possible, both vectors must have the same dimensionality. This is called ***Luong attention*** (again, after the paper's first author), or sometimes ***multiplicative attention***.

… Attention Mechanisms

- The dot product gives a score, and
- all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention.
- Another simplification they proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$) rather than $\mathbf{h}_{(t-1)}$),
- then to use the output of the attention mechanism (noted $\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions (rather than using it to compute the encoder's current hidden state).
- They also proposed a variant of the dot product mechanism where the encoder outputs first go through a linear transformation (i.e., a time-distributed Dense layer without a bias term) before the dot products are computed.
- This is called the "general" dot product approach.

… Attention Mechanisms

- They compared both dot product approaches to the concatenative attention mechanism (adding a rescaling parameter vector **v**), and
- they observed that the dot product variants performed better than concatenative
- For this reason, concatenative attention is much less used now.
- The equations for these three attention mechanisms are summarized in Equation 1:

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

$$\text{with } \alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

$$\text{and } e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^{\top} \mathbf{y}_{(i)} & \textit{dot} \\ \mathbf{h}_{(t)}^{\top} \mathbf{W} \mathbf{y}_{(i)} & \textit{general} \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & \textit{concat} \end{cases}$$

… Attention Mechanisms

- Here is how you can add Luong attention to an Encoder–Decoder model using TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

- We simply wrap the decoder cell in an AttentionWrapper, and we provide the desired attention mechanism (Luong attention in this example).

Visual Attention

- Attention mechanisms are now used for a variety of purposes.
- One of their first applications beyond NMT was in generating image captions using visual attention (see paper#1) (see Fig):
  - a convolutional neural network first processes the image and
  - outputs some feature maps,
  - then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time.
  - At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image.

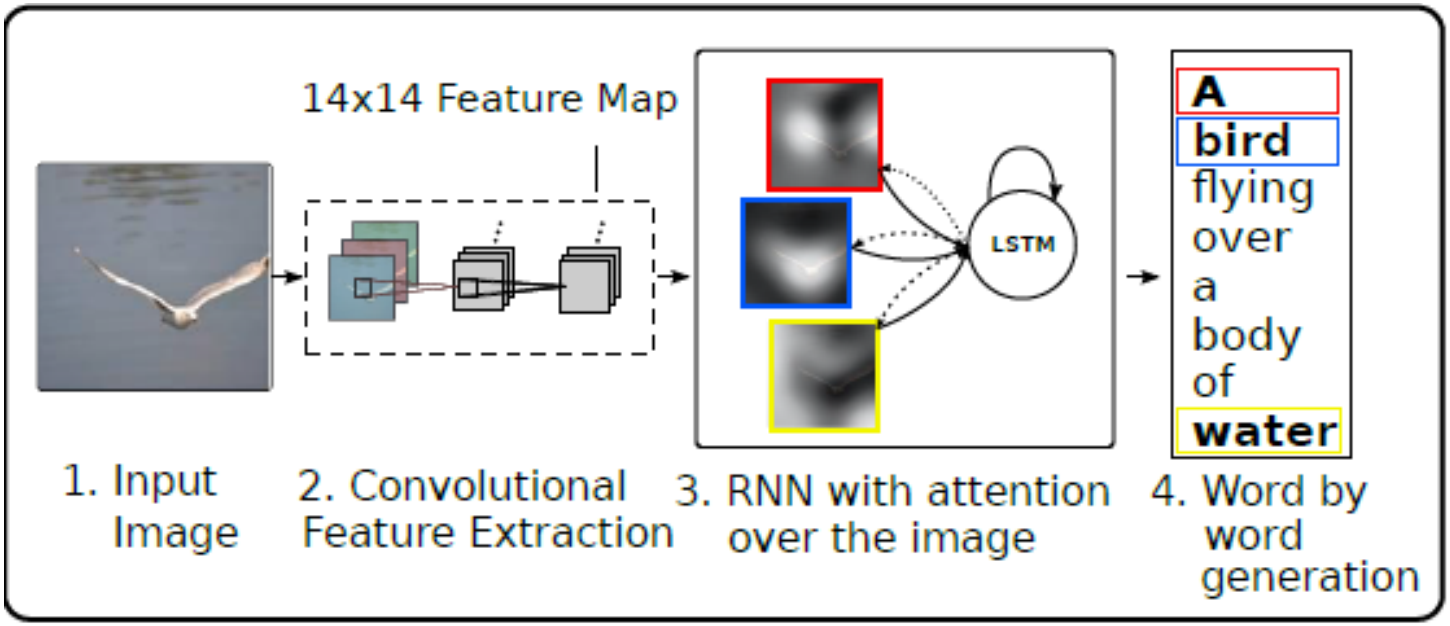

Figure: Our model learns a words/image alignment.

… Visual Attention

- For example, in Figure 7, the model generated the caption “A woman is throwing a frisbee in a park,” and
- we can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”:
  - clearly, most of its attention was focused on the frisbee.

Figure 7: Visual attention: an input image (left) and the model’s focus before producing the word “frisbee” (right)

Explainability

- One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*.
- It can be especially useful when the model makes a mistake:
  - for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow,"
  - then we can go back and check what the model focused on when it output the word "wolf."
  - We may find that it was paying attention not only to the dog, but also to the snow,
  - hinting at a possible explanation:
    - perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there's a lot of snow around.

… Explainability

- We can then fix this by training the model with more images of wolves without snow, and dogs with snow.
- This example comes from a great **paper(#1)** that uses a different approach to explainability:
  - learning an interpretable model locally around a classifier's prediction.

- In some applications, explainability is not just a tool to debug a model;
  - it can be a legal requirement (think of a system deciding whether or not it should grant you a loan).

Attention Is All You Need: The Transformer Architecture

- In a groundbreaking **paper (#1)**, Google team created an architecture called the ***Transformer***,
- which significantly improved the state of the art in NMT without using any recurrent or convolutional layers†,
- just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces).
- As an extra bonus, this architecture was also –
  - much faster to train and
  - easier to parallelize,
- so, they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

… Attention Is All You Need: The Transformer Architecture

The Transformer architecture is represented in Figure 8.

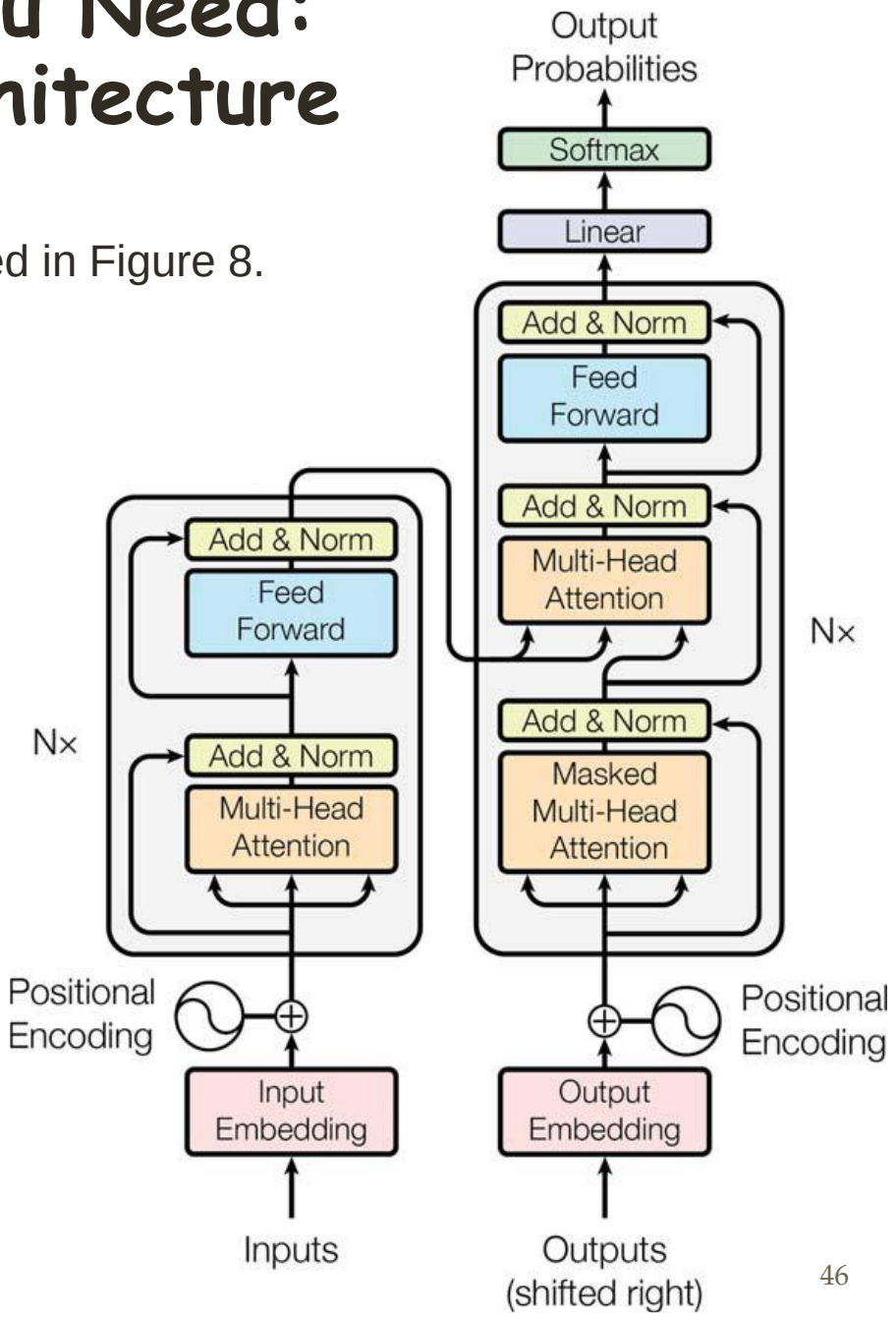

Figure 8: The Transformer architecture.

… Attention Is All You Need: The Transformer Architecture

- In Fig. 8, the left-hand part is the encoder.
- Just like earlier, it takes as input a batch of sentences represented as sequences of word IDs (the input shape is [batch size, max input sentence length]), and
- it encodes each word into a 512-dimensional representation (so the encoder's output shape is [batch size, max input sentence length, 512]).
- Note that the top part of the encoder is stacked N times (in the paper, N = 6).

… Attention Is All You Need: The Transformer Architecture

- In Fig. 8, the right-hand part is the decoder.
- During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning).
- It also receives the outputs of the encoder (i.e., the arrows coming from the left side).
- Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels.
- Just like earlier, the decoder outputs a probability for each possible next word, at each time step (its output shape is [batch size, max output sentence length, vocabulary length]).

… Attention Is All You Need: The Transformer Architecture

- During inference, the decoder cannot be fed targets,
- so we feed it the previously output words (starting with a start-of-sequence token).
- So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).

… Attention Is All You Need: The Transformer Architecture

The Transformer architecture is represented in Figure 8.

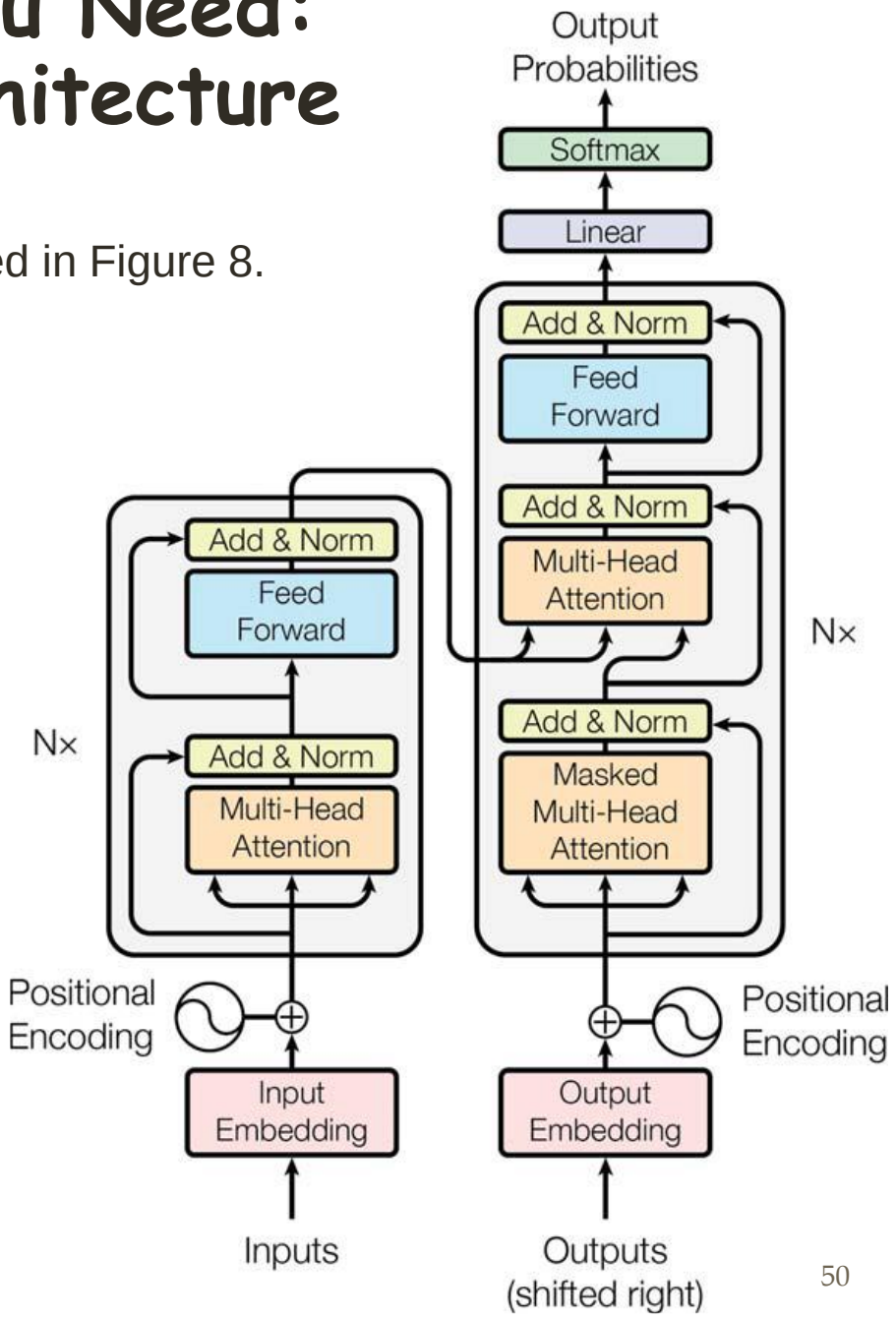

Figure 8: The Transformer architecture.

… Attention Is All You Need: The Transformer Architecture

- Looking more closely, we can see that we are already familiar with most components:
  - there are two embedding layers, 5 × N skip connections, each of them followed by a layer normalization layer, 2 × N "Feed Forward" modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and
  - finally, the output layer is a dense layer using the softmax activation function.
  - All of these layers are timedistributed, so each word is treated independently of all the others.
  - But how can we translate a sentence by only looking at one word at a time?
  - Well, that's where the new components come in:
    - The encoder's *Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones.

… Attention Is All You Need: The Transformer Architecture

- For example, the output of this layer for the word “Queen” in the sentence “They welcomed the Queen of the United Kingdom” will depend on all the words in the sentence,
- but it will probably pay more attention to the words “United” and “Kingdom” than to the words “They” or “welcomed.”
- This attention mechanism is called *self-attention* (the sentence is paying attention to itself).
- The decoder’s *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it.
- Finally, the decoder’s upper *Multi-Head Attention* layer is where the decoder pays attention to the words in the input sentence.
- For example, the decoder will probably pay close attention to the word “Queen” in the input sentence when it is about to output this word’s translation.

… Attention Is All You Need: The Transformer Architecture

- The *positional embeddings* are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence.

- The n^{th} positional embedding is added to the word embedding of the n^{th} word in each sentence.

- This gives the model access to each word's position, which is needed because the *Multi-Head Attention* layers do not consider the order or the position of the words; they only look at their relationships.

- Since all the other layers are time-distributed, they have no way of knowing the position of each word (either relative or absolute).

- Obviously, the relative and absolute word positions are important, so we need to give this information to the Transformer somehow, and *positional embeddings* are a good way to do this.

The Transformer Architecture: Positional Embeddings

- A positional embedding is a dense vector that encodes the position of a word within a sentence:
- the **i**th positional embedding is simply added to the word embedding of the ith word in the sentence.
- These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the sine and cosine functions of different frequencies.
- The positional embedding matrix **P** is defined in **Equation 2**: *Sine/cosine positional embeddings -*

$$P_{p,2i} = \sin\left(p/10000^{2i/d}\right)$$
$$P_{p,2i+1} = \cos\left(p/10000^{2i/d}\right)$$

The Transformer Architecture: Positional Embeddings

- The positional embedding matrix **P** is (defined in Eqn. 2) and represented at the bottom of Figure 9 (transposed),

- where $P_{p,i}$ is the i^{th} component of the embedding for the word located at the p^{th} position in the sentence.

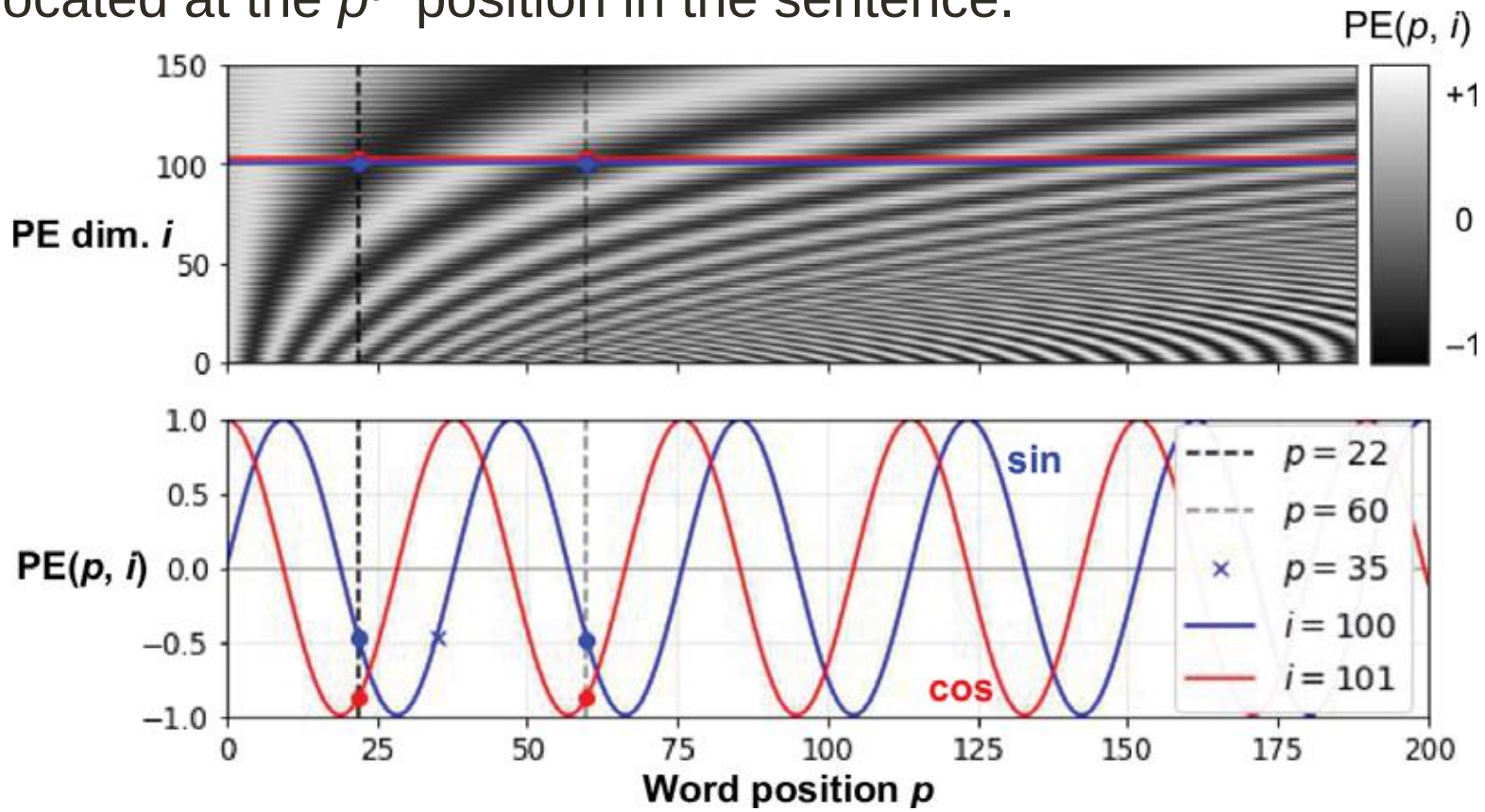

Figure 9: Sine/cosine positional embedding matrix (transposed, top) with a focus on two values of i (bottom).

The Transformer Architecture: Positional Embeddings

- This solution gives the same performance as learned positional embeddings do,
- but it can extend to arbitrarily long sentences, which is why it's favored.
- After the positional embeddings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional embedding for each position
  - e.g., the positional embedding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the bottom left of Figure 9, and
  - you can see that it is unique to that position.
- Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well.

The Transformer Architecture: Positional Embeddings

- For example, words located 38 words apart (e.g., at positions p = 22 and p = 60) always have the same positional embedding values in the embedding dimensions i = 100 and i = 101, as we can see in Figure 9.

- This explains why we need both the sine and the cosine for each frequency:
  - if we only used the sine (the blue wave at i = 100), the model would not be able to distinguish positions p = 22 and p = 35 (marked by a cross).

- [**See exercise**].

Multi-Head Attention

- Multi-Head Attention layer works based on the Scaled Dot-Product Attention layer.
- Now, let us suppose the encoder analyzed the input sentence “They played chess,” and it managed to understand that the word
  - “They” is the subject and
  - the word “played” is the verb, so it encoded this information in the representations of these words.
- Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next.
- For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it’s as if the encoder created a dictionary {“subject”: “They”, “verb”: “played”, …} and
- the decoder wanted to look up the value that corresponds to the key “verb.”

… Multi-Head Attention

- However, the model does not have discrete tokens to represent the keys (like “subject” or “verb”);
- rather it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the query) will not perfectly match any key in the dictionary.
- The solution is to compute a similarity measure between the query and each key in the dictionary, and
- then use the softmax function to convert these similarity scores to weights that add up to 1.
- If the key that represents the verb is by far the most similar to the query, then that key’s weight will be close to 1.

… Multi-Head Attention

- Then the model can compute a weighted sum of the corresponding values, so if the weight of the “verb” key is close to 1,
- then the weighted sum will be very close to the representation of the word “played.”
- In short, we can think of this whole process as a differentiable dictionary lookup.
- The similarity measure used by the Transformer is just the dot product, like in Luong attention.
- In fact, the equation is the same as for Luong attention, except for a scaling factor.
- The equation is shown in Equation 3, in a vectorized form.

… Multi-Head Attention

Equation 3: *Scaled Dot-Product Attention* –

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{keys}}}\right)\mathbf{V}$$

- where, **Q** is a matrix containing one row per query. Its shape is [n_{queries}, d_{keys}], where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- **K** is a matrix containing one row per key. Its shape is [n_{keys}, d_{keys}], where n_{keys} is the number of keys and values.
- **V** is a matrix containing one row per value. Its shape is [n_{keys}, d_{values}], where d_{values} is the number of each value.
- The shape of $\mathbf{Q}\,\mathbf{K}^T$ is [n_{queries}, n_{keys}]:
  - it contains one similarity score for each query/key pair.
  - The output of the softmax function has the same shape, but all rows sum up to 1.
  - The final output has a shape of [n_{queries}, d_{values}]:
    - there is one row per query, where each row represents the query result (a weighted sum of the values).

… Multi-Head Attention

- The scaling factor scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax.
- This is useful in the Masked Multi-Head Attention layer.

- In the encoder, this equation is applied to every input sentence in the batch, with **Q**, **K**, and **V** all equal to the list of words in the input sentence (so each word in the sentence will be compared to every word in the same sentence, including itself).
- Similarly, in the decoder's masked attention layer, the equation will be applied to every target sentence in the batch, with **Q**, **K**, and **V** all equal to the list of words in the target sentence,
- but this time using a mask to prevent any word from comparing itself to words located after it (at inference time the decoder will only have access to the words it already output, not to future words, so during training we must mask out future output tokens).

… Attention Is All You Need: The Transformer Architecture

The Transformer architecture is represented in Figure 8.

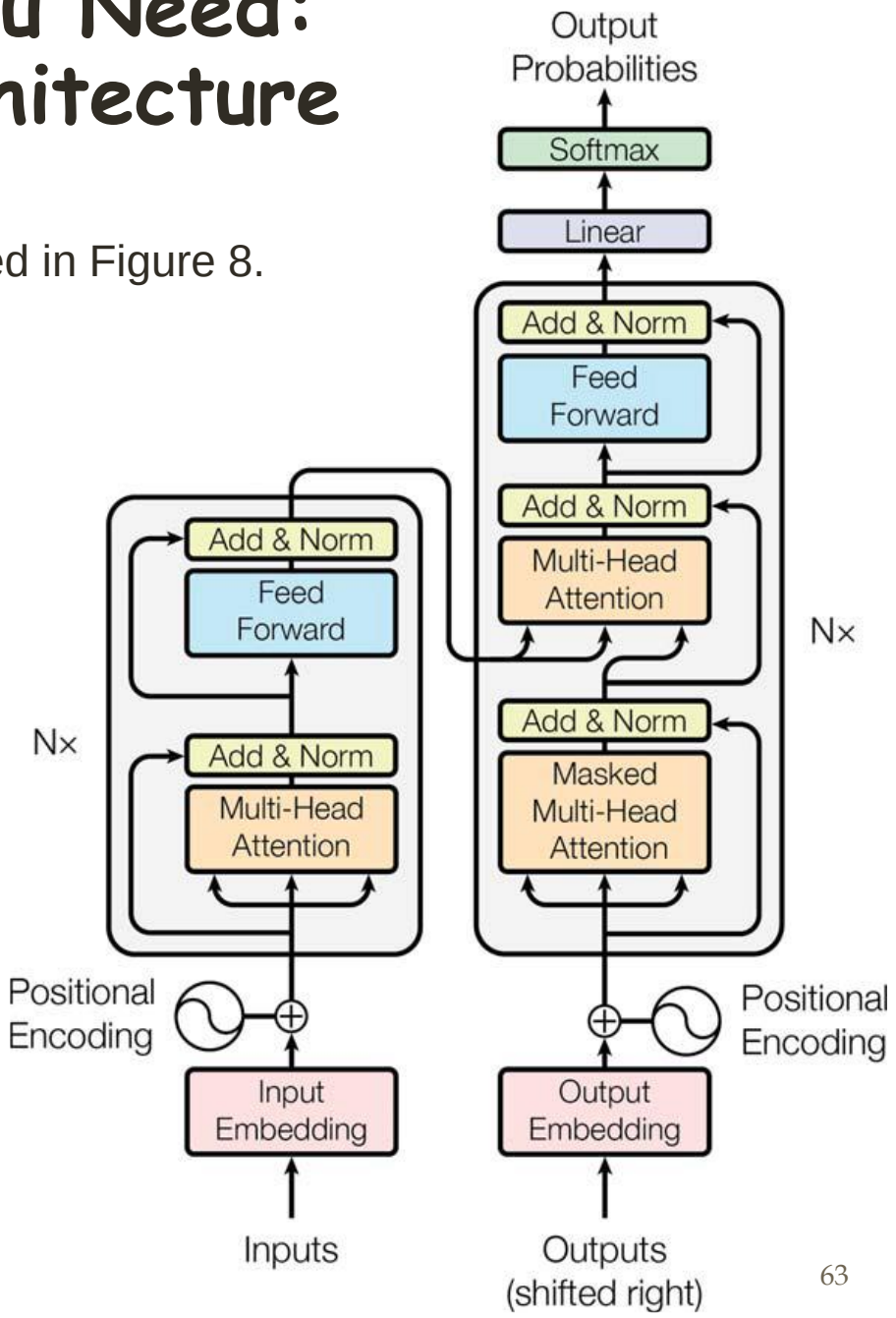

Figure 8: The Transformer architecture.

… Multi-Head Attention

- In the upper attention layer of the decoder, the keys **K** and values **V** are simply the list of word encodings produced by the encoder, and

- the queries **Q** are the list of word encodings produced by the decoder.

- The keras.layers.Attention layer implements Scaled Dot-Product Attention, efficiently applying Equation 3 to multiple sentences in a batch.

- Its inputs are just like **Q**, **K**, and **V**, except with an extra batch dimension (the first dimension).

… Multi-Head Attention

- In TensorFlow, if A and B are tensors with more than two dimensions —
- say, of shape [2, 3, 4, 5] and [2, 3, 5, 6] respectively—
- then tf.matmul(A, B) will treat these tensors as 2 × 3 arrays where each cell contains a matrix, and
- it will multiply the corresponding matrices:
  - the matrix at the i^{th} row and j^{th} column in A will be multiplied by the matrix at the i^{th} row and j^{th} column in B.
  - Since the product of a 4 × 5 matrix with a 5 × 6 matrix is a 4 × 6 matrix, tf.matmul(A, B) will return an array of shape [2, 3, 4, 6].

… Multi-Head Attention

- Figure 10 shows the architecture of a Multi-Head Attention layer

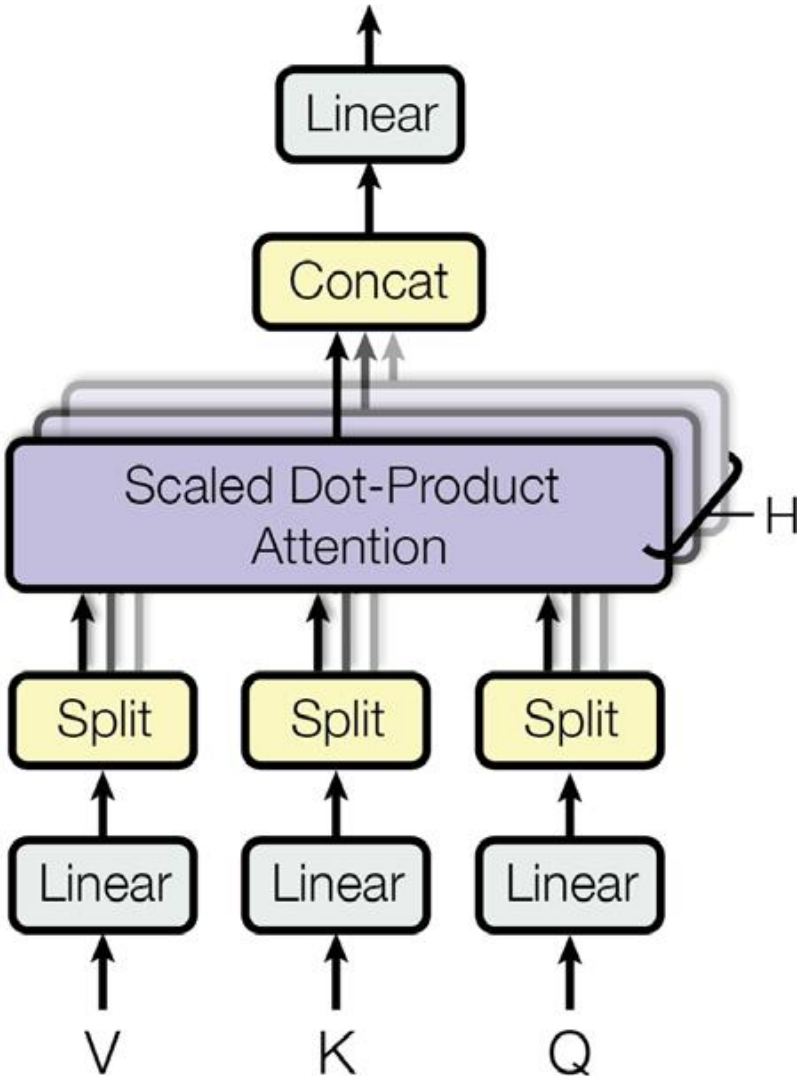

Figure 10: Multi-Head Attention layer architecture.

… Multi-Head Attention

- As we can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the
  - values,
  - keys, and
  - queries (i.e., a timedistributed Dense layer with no activation function).
- All the outputs are simply concatenated, and they go through a final linear transformation (again, timedistributed).

But why? What is the intuition behind this architecture?

- Well, consider the word “played” we discussed earlier (in the sentence “They played chess”).
- The encoder was smart enough to encode the fact that it is a verb.

… Multi-Head Attention

- But the word representation also includes its position in the text, due to the positional encodings, and
- it probably includes many other features that are useful for its translation, such as the fact that it is in the past tense.
- In short, the word representation encodes many different characteristics of the word.
- If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot.
- This is why the Multi-Head Attention layer applies multiple different linear transformations of the values, keys, and queries:
  - this allows the model to apply many different projections of the word representation into different subspaces,
  - each focusing on a subset of the word's characteristics.

… Multi-Head Attention

- Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a verb,
- another linear layer will extract just the fact that it is past tense, and so on.

- Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.
- For the TensorFlow's Transformer class, see this
- For the MultiHeadAttention see this

Recent Innovations in Language Models

The year 2018 has been called the "ImageNet moment for NLP": progress was astounding –

- The ELMo (Embeddings from Language Models) paper (#1) introduced (ELMo):
  - these are contextualized word embeddings learned from the internal states of a deep bidirectional language model.
  - For example, the word "queen" will not have the same embedding in "Queen of the United Kingdom" and in "queen bee."
- The ULMFiT paper (#2) demonstrated the effectiveness of unsupervised pretraining for NLP tasks:
  - the authors trained an LSTM language model using self-supervised learning (i.e., generating the labels automatically from the data) on a huge text corpus,
  - then they fine-tuned it on various tasks.

… Recent Innovations in Language Models

- Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases).
- Moreover, they showed that by fine-tuning the pretrained model on just 100 labeled examples, they could achieve the same performance as a model trained from scratch on 10,000 examples.

- The GPT paper (#1) by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining,
  - but this time using a Transformer-like architecture.
  - The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self-supervised learning.
  - Then they fine-tuned it on various language tasks, using only minor adaptations for each task.

… Recent Innovations in Language Models

- The tasks were quite diverse:
  - they included text classification, entailment (whether sentence A entails sentence B):
    - For example, the sentence “Jane had a lot of fun at her friend’s birthday party” entails “Jane enjoyed the party,” but it is contradicted by “Everyone hated the party” and it is unrelated to “The Earth is flat.”
  - Similarity, e.g., “Nice weather today” is very similar to “It is sunny”, and
  - question answering, e.g., given a few paragraphs of text giving some context, the model must answer some multiple-choice questions.
- GPT-2 paper (**#1**) proposed a very similar architecture, but larger still (with over 1.5 billion parameters!) and
- they showed that it could achieve good performance on many tasks without any fine-tuning.
- This is called zero-shot learning (ZSL).
- A smaller version of the GPT-2 model (with “just” 117 million parameters) is available at https://github.com/openai/gpt-2, along with its pretrained weights.

… Recent Innovations in Language Models

- The BERT paper (**#1**) demonstrates the effectiveness of self-supervised pretraining on a large corpus,
- using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer's encoder).
- The model is naturally bidirectional; hence the B in BERT (Bidirectional Encoder Representations from Transformers).
- Most importantly, the authors proposed two pretraining tasks that explain most of the model's strength:
  - (1) Masked language model (MLM)
  - (2) Next sentence prediction (NSP)

… Recent Innovations in Language Models

Masked language model (MLM):

- Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words.
- For example, if the original sentence is
  - "She had fun at the birthday party,"
- then the model may be given the sentence
  - "She <mask> fun at the <mask> party" and it must predict the words "had" and "birthday" (the other outputs will be ignored).

… Recent Innovations in Language Models

Next sentence prediction (NSP):

- The model is trained to predict whether two sentences are consecutive or not.
- For example, it should predict that "The dog sleeps" and "It snores loudly" are consecutive sentences,
- while "The dog sleeps" and "The Earth orbits the Sun" are not consecutive.
- This is a challenging task, and it significantly improves the performance of the model when it is fine-tuned on tasks such as question answering or entailment.

… Recent Innovations in Language Models

Summary:

- The main innovations in 2018 and 2019 have been
  - better subword tokenization,
  - shifting from LSTMs to Transformers, and
  - pretraining universal language models using self-supervised learning,
    - then fine-tuning them with very few architectural changes (or none at all).
  - Things are moving fast; no one can say what architectures will prevail next year.

- Today, it's clearly Transformers, but tomorrow it might be CNNs:
  - see **paper #1**, where the researchers use masked 2D convolutional layers for sequence-to-sequence tasks.

- Or it might even be RNNs, if they make a surprise comeback:
  - see **paper #2** that shows that by making neurons independent of each other in a given RNN layer, it is possible to train much deeper RNNs capable of learning much longer sequences.

END