Chapter 5

Neural Translation Models

We are finally prepared to look at actual translation models. We have already done most of the work, however, since the most commonly used architecture for neural machine translation is a straightforward extension of neural language models with one refinement, an alignment model.

5.1 Encoder-Decoder Approach

Our first stab at a neural translation model is a straightforward extension of the language model. Recall the idea of a recurrent neural network to model language as a sequential process. Given all previous words, such a model predicts the next word. When we reach the end of the sentence, we now proceed to predict the translation of the sentence, one word at a time.

See Figure 5.1 for an illustration. To train such a model, we simply concatenate the input and output sentences and use the same method as to train a language model. For decoding, we feed in the input sentence, and then go through the predictions of the model until it predicts an end of sentence token.

How does such a network work? Once processing reaches the end of the input sentence (having predicted the end of sentence marker </s>), the hidden state encodes its meaning. In other words, the vector holding the values of the nodes of this final hidden layer is the **input sentence embedding**. This is the **encoder** phase of the model. Then this hidden state is used to produce the translation in the **decoder** phase.

Clearly, we are asking a lot from the hidden state in the recurrent neural network here. During encoder phase, it needs to incorporate all information about the input sentence. It cannot forget the first words towards the end of the sentence. During the decoder phase, not only does it need to have enough information to predict each next word, there also needs to be some accounting for what part of the input sentence has been already translated, and what still needs to be covered.

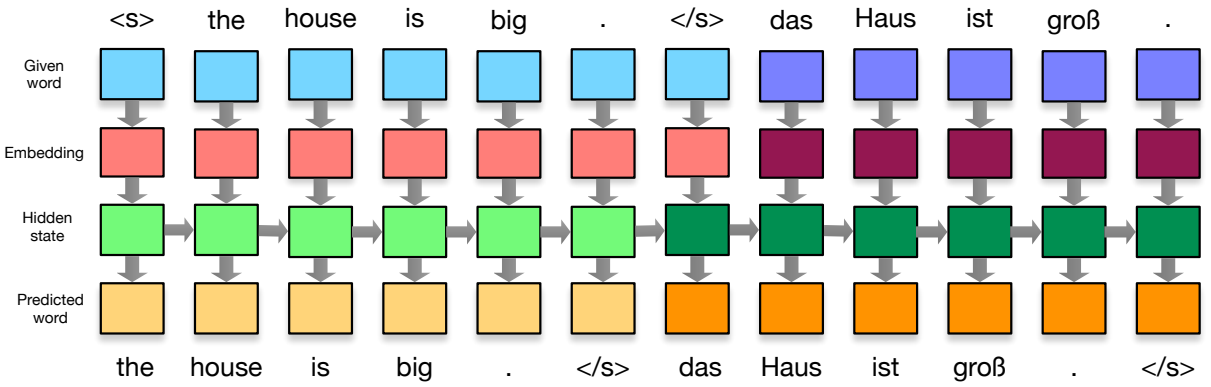

Figure 5.1: Sequence-to-sequence encoder-decoder model: Extending the language model, we concatenate the English input sentence *the house is big* with the German output sentence *das Haus ist groß*. The first dark green box (after processing the end-of-sentence token *</s>*) contains the embedding of the entire input sentence .

In practice, the proposed models works reasonable well for short sentences (up to, say, 10–15 words), but fails for long sentences. Some minor refinements to this model have been proposed, such using the sentence embedding state as input to all hidden states of the decoder phase of the model. This makes the decoder structurally different from the encoder and reduces some of the load from the hidden state during decoding, since it does not need to remember anymore the input. Another idea is to reverse the order of the output sentence, so that the last words of the input sentences are close to the last words of the output sentence.

However, in the following section, we will embark on a more significant improvement of the model, by explicitly modelling alignment of output words to input words.

5.2 Adding an Alignment Model

At the time of writing, the state of the art in neural machine translation is a sequence-to-sequence encoder-decoder model with attention. That is a mouthful, but it is essentially the model we just described in the previous section, with an explicit alignment mechanism. In the deep learning world, this alignment is called **attention**, we are using the words *alignment* and *attention* interchangeably here.

Since the attention mechanism does add a bit of complexity to the model, we are now slowly building up to it, by first taking a look at the encoder, then the decoder, and finally the attention mechanism.

5.2.1 Encoder

The task of the encoder is to provide a representation of the input sentence. The input sentence is a sequence of words, for which we first consult the embedding matrix. Then, as in the basic language model described previously, we process these words with a recurrent neural network.

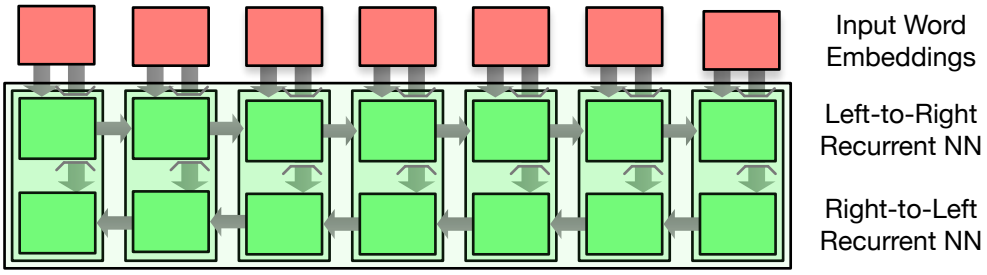

Figure 5.2: Neural machine translation model, part 1: input encoder. It consists of two recurrent neural networks, running right to left and left to right (bidirectional recurrent neural network). The encoder states are the combination of the two hidden states of the recurrent neural networks.

This results in hidden states that encode each word with its left context, i.e., all the preceding words. To also get the right context, we also build a recurrent neural network that runs right-to-left, or more precisely, from the end of the sentence to the beginning.

Figure 5.2 illustrates the model. Having two recurrent neural networks running in two directions is called a **bidirectional recurrent neural network**. Mathematically, the encoder consists of the embedding lookup for each input word x_j, and the mapping that steps through the hidden states $\overleftarrow{h_j}$ and $\overrightarrow{h_j}$

$$\overleftarrow{h_j} = f(\overleftarrow{h_{j+1}}, \bar{E}\, x_j) \tag{5.1}$$
$$\overrightarrow{h_j} = f(\overrightarrow{h_{j-1}}, \bar{E}\, x_j) \tag{5.2}$$

In the equation above, we used a generic function f for a cell in the recurrent neural network. This function may be a typical feed-forward neural network layer — such as $f(x) = \tanh(Ax + b)$ — or the more complex gated recurrent units (GRUs) or long short term memory cells (LSTMs). The original paper proposing this approached used GRUs, but lately LSTMs have become more popular.

Note that we could train these models by adding a step that predicts the next word in the sequence, but we are actually training it in the context of the full machine translation model. Limiting the description to the decoder, its output is a sequence of word representations that concatenate the two hidden states $(\overleftarrow{h_j}, \overrightarrow{h_j})$.

5.2.2 Decoder

The decoder is also a recurrent neural network. It takes some representation of the input context (more on that in the next section on the attention mechanism) and the previous hidden state and output word prediction, and generates a new hidden decoder state and a new output word prediction. See Figure 5.3 for an illustration.

Mathematically, we start with the recurrent neural network that maintains a sequence of hidden states s_i which are computed from the previous hidden state s_{i-1}, the embedding of the previous output word Ey_{i-1}, and the input context c_i (which we still have to define).

$$s_i = f(s_{i-1},\ Ey_{i-1}, c_i) \tag{5.3}$$

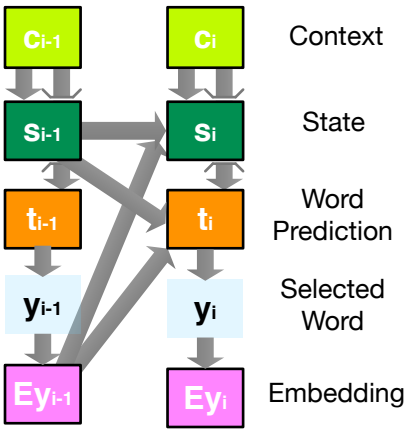

Figure 5.3: Neural machine translation model, part 2: output decoder. Given the context from the input sentence, and the embedding of the previously selected word, new decoder states and word predictions are computed.

Again, there are several choices for the function f that combines these inputs to generate the next hidden state: linear transforms with activation function, GRUs, LSTMs, etc. Typically, the choice here matches the encoder. So, if we use LSTMs for the encoder, then we also use LSTMs for the decoder.

From the hidden state. we now predict the output word. This prediction takes the form of a probability distribution over the entire output vocabulary. If we have a vocabulary of, say, 50,000 words, then the prediction is a 50,000 dimensional vector, each element corresponding to the probability predicted for one word in the vocabulary.

The prediction vector t_i is conditioned on the decoder hidden state s_{i-1} and, again, the embedding of the previous output word Ey_{i-1} and the input context c_i.

$$t_i = \text{softmax}\big(W(Us_{i-1} + VEy_{i-1} + Cc_i)\big) \tag{5.4}$$

Note that we repeat the conditioning on Ey_{i-1} since we use the hidden state s_{i-1} and not s_1. This separates the encoder state progression from s_{i-1} to s_i from the prediction of the output word t_i.

The softmax is used to convert the raw vector into a probability distribution, where the sum of all values is 1. Typically, the highest value in the vector indicates the output word token y_i. Its word embedding Ey_{i-1} informs the next time step of the recurrent neural network.

During training, the correct output word y_i is known, so training proceeds with that word. The training objective is to give as much probability mass as possible to the correct output word. The cost function that drives training is hence the negative log of the probability given to the correct word translation.

$$\text{cost} = -\log t_i[y_i] \tag{5.5}$$

Ideally, we want to give the correct word the probability 1, which would mean a negative log probability of 0, but typically it is a lower probability, hence a higher cost. Note that the cost function is tied to individual words, the overall sentence cost is the sum of all word costs.

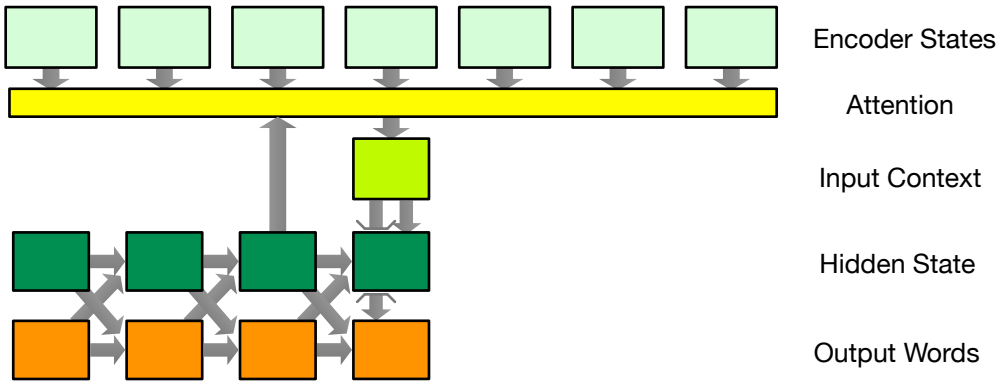

Figure 5.4: Neural machine translation model, part 3: attention model. Associations are computed between the last hidden state of the decoder and the word representations (encoder states). These associations are used to compute a weighted sum of encoder states.

During inference on a new test sentence, we typically chose the word y_i with the highest value in t_i use its embedding Ey_i for the next steps. But we will also explore beam search strategies where the next likely words are selected as y_i, creating a different conditioning context for the next words. More on that later.

5.2.3 Attention Mechanism

We currently have two loose ends. The decoder gave us a sequence of word representations $h_j = (\overleftarrow{h_j}, \overrightarrow{h_j})$ and the decoder expects a context c_i at each step i. We now describe the attention mechanism that ties these ends together.

The attention mechamism is hard to visualize using our typical neural network graphs, but Figure 5.4 gives at least an idea what the input and output relations are. The attention mechanism is informed by all input word representations $(\overleftarrow{h_j}, \overrightarrow{h_j})$ and the previous hidden state of the decoder s_{i-1}, and it produces a context state c_i.

The motivation is that we want to compute an association between the decoder state (which contains information where we are in the output sentence production) and each input word. Based on how strong this association is, or in other words how relevant each particular input word is to produce the next output word, we want to weight the impact of its word representation.

Mathematically, we first compute this association with a feedforward layer (using weight vectors w^a, u^a and bias value b^a)

$$a(s_{i-1}, h_j) = w^{aT} s_{i-1} + u^{aT} h_j + b^a \tag{5.6}$$

The output of this computation is a scalar value, indicating how important input word j is to produce output word i.

We normalize this attention value, so that the attention values across all input words j add up to one, using the softmax.

$$\alpha_{ij} = \frac{\exp(a(s_{i-1}, h_j))}{\sum_k \exp(a(s_{i-1}, h_k))} \tag{5.7}$$

Now we use the normalized attention value to weigh the contribution of the input word representation h_j to the context vector c_i and we are done.

$$c_i = \sum_j \alpha_{ij} h_j \tag{5.8}$$

Simply adding up word representation vectors (weighted or not) may at first seem an odd and simplistic thing to do. But it is very common practice in deep learning for natural language processing. Researchers have no qualms about using sentence embeddings that are simply the sum of word embeddings and other such schemes.

5.3 Training

With the complete model in hand, we can now take a closer look at training. One challenge is that the number of steps in the decoder and the number of steps in the encoder varies with each training example. Sentence pairs consist of sentences of different length, so we cannot have the same computation graph for each training example but instead have to dynamically create the computation graph for each of them. This technique is called **unrolling** the recurrent neural networks, and we already discussed it with regard to language models (recall Section 4.4).

The fully unrolled computation graph for a short sentence pair is shown in Figure 5.5. Note a couple of things. The error computed from this one sentence pair is the sum of the errors computed for each word. When proceeding to the next word prediction, we use the correct word as conditioning context for the decoder hidden state and the word prediction. Hence, the training objective is based on the probability mass given to the correct word, given a perfect context. There have been some attempts to use different training objectives, such as the BLEU score, but they have not yet been shown to be superior.

Practical training of neural machine translation models requires GPUs which are well suited to the high degree of parallelism inherent in these deep learning models (just think of the many matrix multiplications). To increase parallelism even more, we process several sentence pairs (say, 100) at once. This implies that we increase the dimensionality of all the state tensors.

To given an example. We represent each input word in specific sentence pair with a vector h_j. Since we already have a sequence of input words, these are lined up in a matrix. When we process a batch of sentence pairs, we again line up these matrices into a 3-dimensional tensor.

Similarly, to give another example, the decoder hidden state s_i is a vector for each output word. Since we process a batch of sentences, we line up their hidden states into a matrix. Note that in this case it is not helpful to line up the states for all the output words, since the states are computed sequentially.

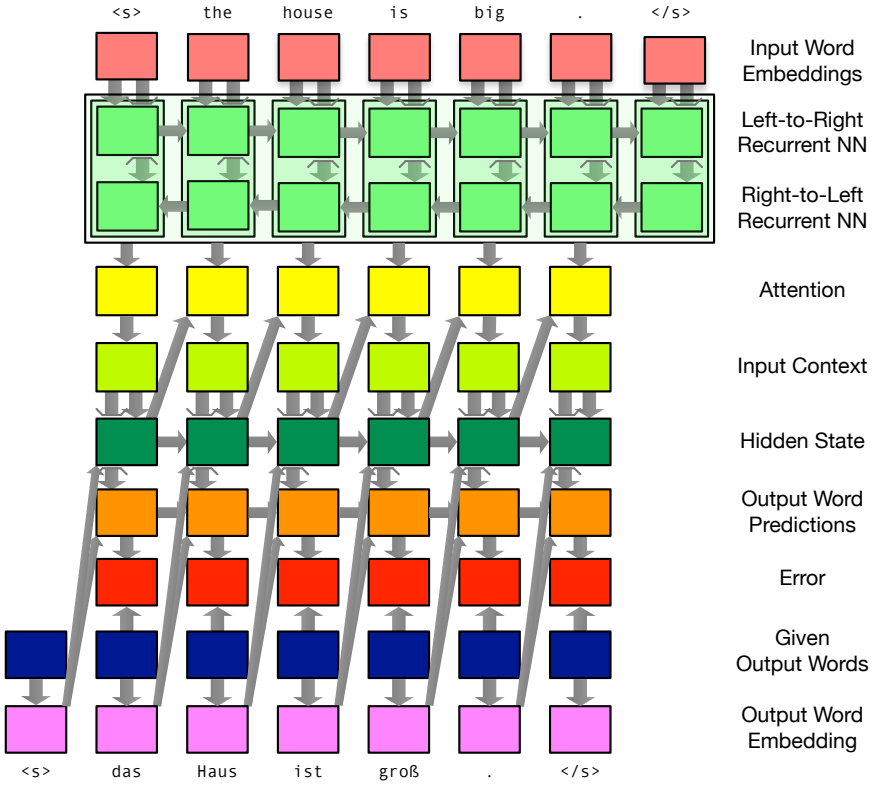

Figure 5.5: Fully unrolled computation graph for training example with 7 input tokens *<s> the house is big </s>* and 6 output tokens *das Haus is groß</s>*. The cost function (error) is computed for each output word individually, and summed up across the sentence. When walking through the deocder states, the correct previous output words are used as conditioning context.

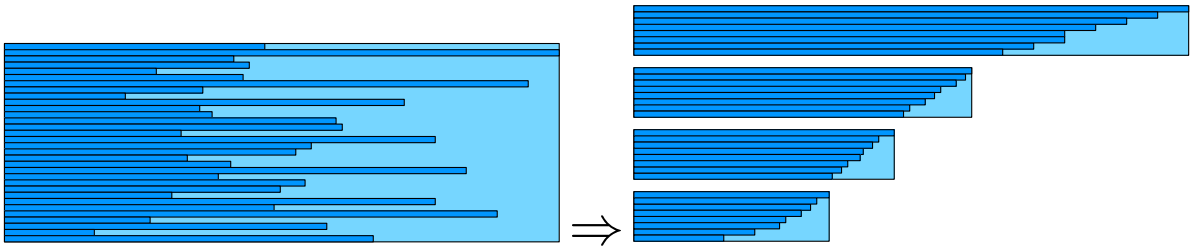

Figure 5.6: To make better use of parallelism in GPUs, we process a batch of training examples (sentence pairs) at a time. Converting a batch of training examples into a set of mini batches that have similar length. This wastes less computation on filler words (light blue).

Recall the first computation of the attention mechanism

$$a(s_{i-1}, h_j) = W^a s_{i-1} + U^a h_j + b^a \tag{5.9}$$

We can pass this computation to the GPU with a matrix of encoder states s_{i-1} and a 3-dimensional tensor of input encodings h_j, resulting in a matrix of attention values (one dimension for the sentence pairs, one dimension for the input words). Due to the massive re-use of values in W^a, U^a, and b^a as well as the inherent parallelism of this computation, GPUs can show their true power.

You may feel that we just created a glaring contradiction. First, we argued that we have to process one training example at a time, since sentence pairs typically have different length, and hence computation graphs have different size. Then, we argued for batching, say, 100 sentence pairs together to better exploit parallelism. These are indeed conflicting goals.

See Figure 5.6. When batching training examples together, we have to consider the maximum sizes for input and output sentences in a batch and unroll the computation graph to these maximum sizes. For shorter sentences, we fill the remaining gaps with non-words and keep track of where the valid data is with a **mask**. This means, for instance, that we have to ensure that no attention is given to words beyond the length of the input sentence, and no errors and gradient updates are computed from output words beyond the length of the output sentence.

To avoid wasted computations on gaps, a nice trick is to sort the sentence pairs in the batch by length and break it up into **mini-batches** of similar length.[1]

To summarize, training consists of the following steps

- Shuffle the training corpus (to avoid undue biases due to temporal or topical order)
- Break up the corpus into maxi-batches
- Break up each maxi-batch into mini-batches

[1]There is a bit of confusion of the technical terms here. Sometimes, the entire training corpus is called a *batch*, as used in the contrast between *batch* updating and *online* updating. In that context, smaller batches with a subset of the are called *mini-batches* (recall Section 2.6.7 on page 26). Here, we use the term *batch* (or *maxi-batch*) for such a subset, and *mini-batch* for a subset of the subset.

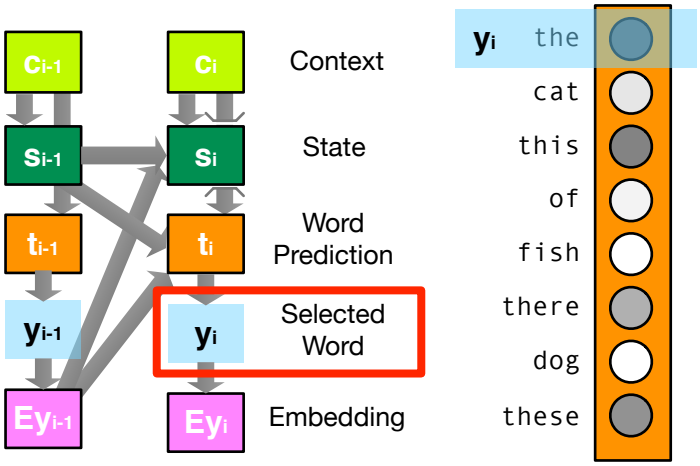

Figure 5.7: Elementary decoding step: The model predicts a word prediction probability distribution. We select the most likely word (*the*). Its embedding is part of the conditioning context for the next word prediction (and decoder state).

- Process each mini-batch, gather gradients
- Apply all gradients for a maxi-batch to update the parameters

Typically, training neural machine translation models takes about 5–15 epochs (passes through entire training corpus). A common stopping criteria is to check progress of the model on a validation set (that is not part of the training data) and halt when the error on the validation set does not improve. Training longer would not lead to any further improvements and may even degrade performance due to overfitting.

5.4 Beam Search

Translating with neural translation models proceeds one step at a time. At each step, we predict one output word. In our model, we first compute a probability distribution over all words. We then pick the most likely word and move to the next prediction step. Since the model is conditioned on the previous output word (recall Equation 5.3), we use its word embedding in the conditioning context for the next step.

See Figure 5.7 for an illustration. At each time step, we obtain a probability distribution over words. In practice, this distribution is most often quite spiked, only few words — or maybe even just one word — amass almost all of the probability. In the example, the word *the* received the highest probability, so we pick it as the output word.

A real example of how a neural machine translation model translates a German sentence into English is shown in Figure 5.8. The model tends to give most, if not almost all, probability mass to the top choice, but the sentence translation also indicates word choice ambiguity, such as *believe* (68.4%) vs. *think* (28.6%) or *different* (41.5%) vs. *various* (22.7%). There is also ambiguity

Input Sentence

ich glaube aber auch , er ist clever genug um seine Aussagen vage genug zu halten , so dass sie auf verschiedene Art und Weise interpretiert werden können .

Output Word Predictions

Best		Alternatives
but	(42.1%)	*however* (25.3%), *I* (20.4%), *yet* (1.9%), *and* (0.8%), *nor* (0.8%), ...
I	(80.4%)	*also* (6.0%), *,* (4.7%), *it* (1.2%), *in* (0.7%), *nor* (0.5%), *he* (0.4%), ...
also	(85.2%)	*think* (4.2%), *do* (3.1%), *believe* (2.9%), *,* (0.8%), *too* (0.5%), ...
believe	(68.4%)	*think* (28.6%), *feel* (1.6%), *do* (0.8%), ...
he	(90.4%)	*that* (6.7%), *it* (2.2%), *him* (0.2%), ...
is	(74.7%)	*'s* (24.4%), *has* (0.3%), *was* (0.1%), ...
clever	(99.1%)	*smart* (0.6%), ...
enough	(99.9%)	
to	(95.5%)	*about* (1.2%), *for* (1.1%), *in* (1.0%), *of* (0.3%), *around* (0.1%), ...
keep	(69.8%)	*maintain* (4.5%), *hold* (4.4%), *be* (4.2%), *have* (1.1%), *make* (1.0%), ...
his	(86.2%)	*its* (2.1%), *statements* (1.5%), *what* (1.0%), *out* (0.6%), *the* (0.6%), ...
statements	(91.9%)	*testimony* (1.5%), *messages* (0.7%), *comments* (0.6%), ...
vague	(96.2%)	*v@@* (1.2%), *in* (0.6%), *ambiguous* (0.3%), ...
enough	(98.9%)	*and* (0.2%), ...
so	(51.1%)	*,* (44.3%), *to* (1.2%), *in* (0.6%), *and* (0.5%), *just* (0.2%), *that* (0.2%), ...
they	(55.2%)	*that* (35.3%), *it* (2.5%), *can* (1.6%), *you* (0.8%), *we* (0.4%), *to* (0.3%), ...
can	(93.2%)	*may* (2.7%), *could* (1.6%), *are* (0.8%), *will* (0.6%), *might* (0.5%), ...
be	(98.4%)	*have* (0.3%), *interpret* (0.2%), *get* (0.2%), ...
interpreted	(99.1%)	*interpre@@* (0.1%), *constru@@* (0.1%), ...
in	(96.5%)	*on* (0.9%), *differently* (0.5%), *as* (0.3%), *to* (0.2%), *for* (0.2%), *by* (0.1%), ...
different	(41.5%)	*a* (25.2%), *various* (22.7%), *several* (3.6%), *ways* (2.4%), *some* (1.7%), ...
ways	(99.3%)	*way* (0.2%), *manner* (0.2%), ...
.	(99.2%)	*</s>* (0.2%), *,* (0.1%), ...
</s>	(100.0%)	

Figure 5.8: Word predictions of the neural machine translation model. Frequently, most of the probability mass is given to the top choice, but semantically related words may rank high, e.g., *believe* (68.4%) vs. *think* (28.6%). The subword units *interpre@@* are explain in Section 6.2 on page 69.

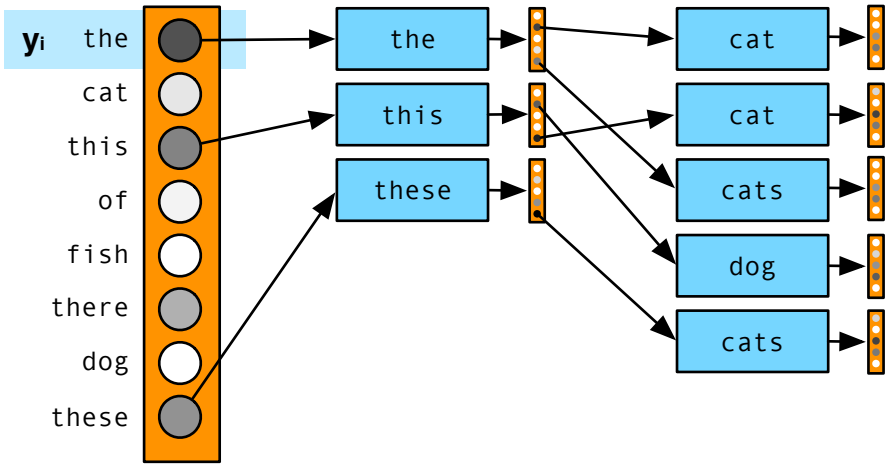

Figure 5.9: Beam search in neural machine translation. After committing to a short list of specific output words (the **beam**), new word predictions are made for each. These differ since the committed output word is part of the conditioning context to make predictions.

about grammatical structure, such as if the sentence should start with the discourse connective *but* (42.1%) or the subject *I* (20.4%).

This process suggests that we perform 1-best greedy search. This makes us vulnerable to the so-called **garden-path problem**. Sometimes we follow a sequence of words and realize too late that we made a mistake early on. In that case, the best sequence consists of less probable words initially which are redeemed by subsequent words in the context of the full output. Consider the case of having to produce an idiomatic phrase that is non-compositional. The first words of these phrases may be really odd word choices by themselves (e.g., *piece of cake* for *easy*). Only once the full phrase is formed, their choice is redeemed.

Note that we are faced with the same problem in traditional statistical machine translation models — arguable even more so there since we rely on sparser contexts when making predictions for the next words. Decoding algorithms for these models keep a list of the n-best candidate **hypotheses**, expand them and keep the n-best expanded hypotheses. We can do the same for neural translation models.

When predicting the first word of the output sentence, we keep a **beam** of the top n most likely word choices. They are scored by their probability. Then, we use each of these words in the beam in the conditioning context for the next word. Due to this conditioning, we make different word predictions for each. We now multiply the score for the partial translation (at this point just the probability for the first word), and the probabilities from its word predictions. We select the highest scoring word pairs for the next beam. See Figure 5.9 for an illustration.

This process continues. At each time step, we accumulate word translation probabilities, giving us scores for each hypothesis. A sentence translation is complete, when the end of sentence token is produced. At this point, we remove the completed hypothesis from the beam and reduce beam size by 1. Search terminates, when no hypotheses are left in the beam.

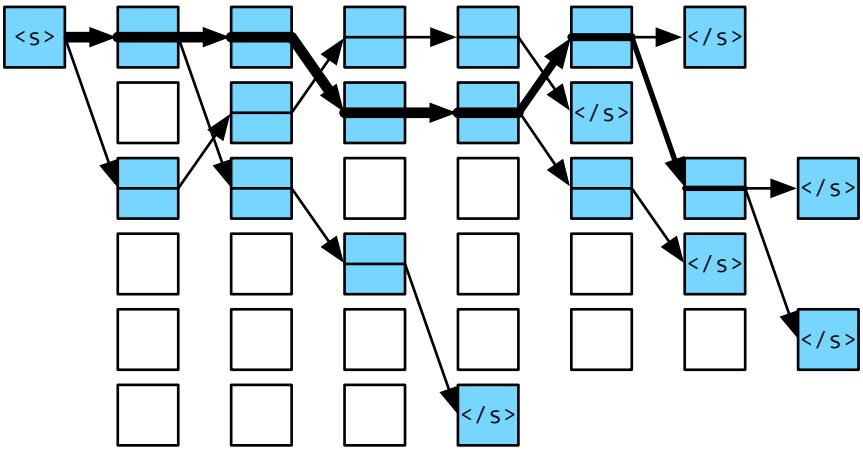

Figure 5.10: Search graph for beam search decoding in neural translation models. At each time step, the $n = 6$ best partial translations (called hypotheses) are selected. An output sentence is complete when the end of sentence token </s> is predicted. We reduce the beam after that and terminate when n full sentence translations are completed. Following the back-pointers from the end of sentence tokens allows us to read them off. Empty boxes represent hypotheses that are not part of any complete path.

Search produces a graph of hypotheses, as shown in Figure 5.10. It starts with the start of sentence symbol <s> and its paths terminate with the end of sentence symbol </s>. Given the compete graph, the resulting translations can be obtained by following the back-pointers. The complete hypothesis (i.e., one that ended with a </s> symbol) with the highest score points to the best translation.

When choosing among the best paths, we score each with the product of its word prediction probabilities. In practice, we get better results when we normalize the score by the output length of a translation, i.e., divide by the number of words. We carry out this normalization after search is completed. During search, all translations in a beam have the same length, so the normalization would make no difference.

Note that in traditional statistical machine translation, we were able to combine hypotheses if they share the same conditioning context for future feature functions. This not possible anymore for recurrent neural networks since we condition on the entire output word sequence from the beginning. As a consequence, the search graph is generally less diverse than search graphs in statistical machine translation models. It is really just a search tree where the number of complete paths is the same as the size of the beam.

Further Readings The attention model has its roots in a sequence-to-sequence model. Cho et al. (2014) use recurrent neural networks for the approach. Sutskever et al. (2014) use a LSTM (long short-term memory) network and reverse the order of the source sentence before decoding.

The seminal work by Bahdanau et al. (2015) adds an alignment model (so called “attention mechanism") to link generated output words to source words, which includes conditioning on the hidden state that produced the preceding target word. Source words are represented by the two hidden states of recurrent neural networks that process the source sentence left-to-right and right-to-left. Luong et al.

(2015b) propose variants to the attention mechanism (which they call “global" attention model) and also a hard-constraint attention model (“local" attention model) which is restricted to a Gaussian distribution around a specific input word.

To explicitly model the trade-off between source context (the input words) and target context (the already produced target words), Tu et al. (2016a) introduce an interpolation weight (called “context gate") that scales the impact of the (a) source context state and (b) the previous hidden state and the last word when predicting the next hidden state in the decoder.

Tu et al. (2017) augment the attention model with a reconstruction step. The generated output is translated back into the input language and the training objective is extended to not only include the likelihood of the target sentence but also the likelihood to the reconstructed input sentence.