7 Neural MT 1: Neural Encoder-Decoder Models

From Section 3 to Section 6, we focused on the language modeling problem of calculating the probability $P(E)$ of a sequence E. In this section, we return to the statistical machine translation problem (mentioned in Section 2) of modeling the probability $P(E|F)$ of the output E given the input F.

7.1 Encoder-decoder Models

The first model that we will cover is called an **encoder-decoder** model [3, 7, 9, 19]. The basic idea of the model is relatively simple: we have an RNN language model, but before starting calculation of the probabilities of E, we first calculate the initial state of the language model using another RNN over the source sentence F. The name "encoder-decoder" comes from the idea that the first neural network running over F "encodes" its information as a vector of real-valued numbers (the hidden state), then the second neural network used to predict E "decodes" this information into the target sentence.

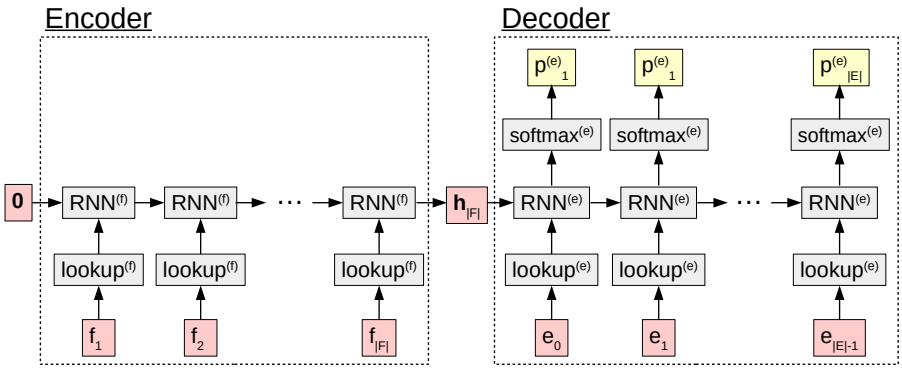

Figure 21: A computation graph of the encoder-decoder model.

If the encoder is expressed as $\text{RNN}^{(f)}(\cdot)$, the decoder is expressed as $\text{RNN}^{(e)}(\cdot)$, and we have a softmax that takes $\text{RNN}^{(e)}$'s hidden state at time step t and turns it into a probability, then our model is expressed as follows (also shown in Figure 21):

$$\begin{aligned}
\boldsymbol{m}_t^{(f)} &= M_{\cdot,f_t}^{(f)} \\
\boldsymbol{h}_t^{(f)} &= \begin{cases} \text{RNN}^{(f)}(\boldsymbol{m}_t^{(f)}, \boldsymbol{h}_{t-1}^{(f)}) & t \geq 1, \\ \mathbf{0} & \text{otherwise.} \end{cases} \\
\boldsymbol{m}_t^{(e)} &= M_{\cdot,e_{t-1}}^{(e)} \\
\boldsymbol{h}_t^{(e)} &= \begin{cases} \text{RNN}^{(e)}(\boldsymbol{m}_t^{(e)}, \boldsymbol{h}_{t-1}^{(e)}) & t \geq 1, \\ \boldsymbol{h}_{|F|}^{(f)} & \text{otherwise.} \end{cases} \\
\boldsymbol{p}_t^{(e)} &= \text{softmax}(W_{hs}\boldsymbol{h}_t^{(e)} + b_s)
\end{aligned} \tag{62}$$

In the first two lines, we look up the embedding $\boldsymbol{m}_t^{(f)}$ and calculate the encoder hidden state $\boldsymbol{h}_t^{(f)}$ for the tth word in the source sequence F. We start with an empty vector $\boldsymbol{h}_0^{(f)} = \mathbf{0}$, and

by $\boldsymbol{h}^{(f)}_{|F|}$, the encoder has seen all the words in the source sentence. Thus, this hidden state should theoretically be able to encode all of the information in the source sentence.

In the decoder phase, we predict the probability of word e_t at each time step. First, we similarly look up $\boldsymbol{m}^{(e)}_t$, but this time use the previous word e_{t-1}, as we must condition the probability of e_t on the previous word, not on itself. Then, we run the decoder to calculate $\boldsymbol{h}^{(e)}_t$. This is very similar to the encoder step, with the important difference that $\boldsymbol{h}^{(e)}_0$ is set to the final state of the encoder $\boldsymbol{h}^{(f)}_{|F|}$, allowing us to condition on F. Finally, we calculate the probability $\boldsymbol{p}^{(e)}_t$ by using a softmax on the hidden state $\boldsymbol{h}^{(e)}_t$.

While this model is quite simple (only 5 lines of equations), it gives us a straightforward and powerful way to model $P(E|F)$. In fact, [19] have shown that a model that follows this basic pattern is able to perform translation with similar accuracy to heavily engineered systems specialized to the machine translation task (although it requires a few tricks over the simple encoder-decoder that we'll discuss in later sections: beam search (Section 7.2), a different encoder (Section 7.3), and ensembling (Section 18)).

7.2 Generating Output

However, at this point, we have only mentioned how to create a probability model $P(E|F)$ and haven't yet covered how to actually generate translations from it, which we will now cover in the next section. In general, when we generate output we can do so according to several criteria:

Random Sampling: Randomly select an output E from the probability distribution $P(E|F)$. This is usually denoted $\hat{E} \sim P(E|F)$.

1-best Search: Find the E that maximizes $P(E|F)$, denoted $\hat{E} = \operatorname{argmax}_E P(E|F)$.

n-best Search: Find the n outputs with the highest probabilities according to $P(E|F)$.

Which of these methods we will choose will depend on our application, so we will discuss some use cases.

7.2.1 Random Sampling and Greedy Search

First, **random sampling** is useful in cases where we may want to get a variety of outputs for a particular input. One example of a situation where this is useful would be in a sequence-to-sequence model for a dialog system, where we would prefer the system to not always give the same response to a particular user input to prevent monotony. Luckily, in models like the encoder-decoder above, it is simple to exactly generate samples from the distribution $P(E|F)$ using a method called **ancestral sampling**. Ancestral sampling works by sampling variable values one at a time, gradually conditioning on more context, so at time step t, we will sample a word from the distribution $P(e_t|\hat{e}_1^{t-1})$. In the encoder-decoder model, this means we simply have to calculate $\boldsymbol{p}_t$ according to the previously sampled inputs, leading to the simple generation algorithm in Algorithm 3.

One thing to note is that sometimes we also want to know the probability of the sentence that we sampled. For example, given a sentence $\hat{E}$ generated by the model, we might want to

know how certain the model is in its prediction. During the sampling process, we can calculate $P(\hat{E}|F) = \prod_t^{|\hat{E}|} P(\hat{e}_t|F, \hat{E}_1^{t-1})$ incrementally by stepping along and multiplying together the probabilities of each sampled word. However, as we remember from the discussion of probability vs. log probability in Section 3.3, using probabilities as-is can result in very small numbers that cause numerical precision problems on computers. Thus, when calculating the full-sentence probability it is more common to instead add together log probabilities for each word, which avoids this problem.

Algorithm 3 Generating random samples from a neural encoder-decoder

```
1: procedure SAMPLE
2:     for t from 1 to |F| do
3:         Calculate m_t^(f) and h_t^(f)
4:     end for
5:     Set ê_0 ="⟨s⟩" and t ← 0
6:     while ê_t ≠"⟨/s⟩" do
7:         t ← t + 1
8:         Calculate m_t^(e), h_t^(e), and p_t^(e) from ê_{t-1}
9:         Sample ê_t according to p_t^(e)
10:    end while
11: end procedure
```

Next, let's consider the problem of generating a 1-best result. This variety of generation is useful in machine translation, and most other applications where we simply want to output the translation that the model thought was best. The simplest way of doing so is **greedy search**, in which we simply calculate $\boldsymbol{p}_t$ at every time step, select the word that gives us the highest probability, and use it as the next word in our sequence. In other words, this algorithm is exactly the same as Algorithm 3, with the exception that on Line 9, instead of sampling $\hat{e}_t$ randomly according to $\boldsymbol{p}_t^{(e)}$, we instead choose the max: $\hat{e}_t = \operatorname{argmax}_i p_{t,i}^{(e)}$.

Interestingly, while ancestral sampling exactly samples outputs from the distribution according to $P(E|F)$, greedy search is not guaranteed to find the translation with the highest probability. An example of a case in which this is true can be found in the graph in Figure 22, which is an example of search graph with a vocabulary of $\{\text{a}, \text{b}, \langle/\text{s}\rangle\}$.[25] As an exercise, I encourage readers to find the true 1-best (or n-best) sentence according to the probability $P(E|F)$ and the probability of the sentence found according to greedy search and confirm that these are different.

7.2.2 Beam Search

One way to solve this problem is through the use of **beam search**. Beam search is similar to greedy search, but instead of considering only the one best hypothesis, we consider b best hypotheses at each time step, where b is the "width" of the beam. An example of beam search where $b = 2$ is shown in Figure 23 (note that we are using log probabilities here because they are more conducive to comparing hypotheses over the entire sentence, as mentioned before).

[25] In reality, we will never have a probability of exactly $P(e_t = \langle/\text{s}\rangle|F, e_1^{t-1}) = 1.0$, but for illustrative purposes, we show this here.

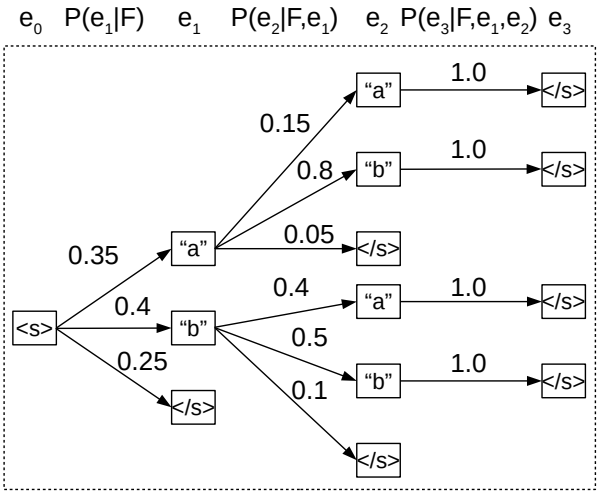

Figure 22: A search graph where greedy search fails.

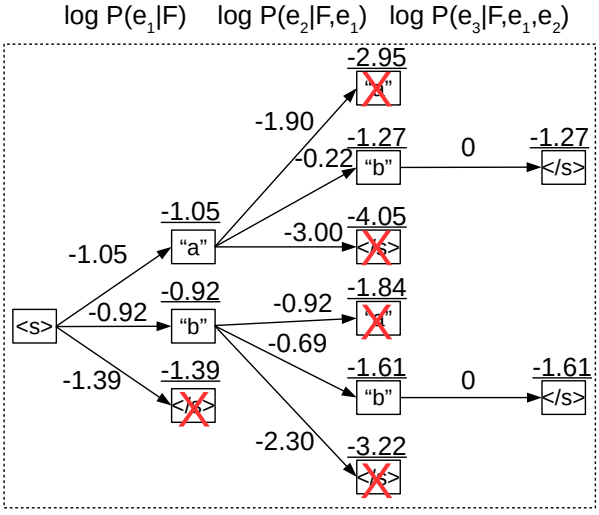

Figure 23: An example of beam search with $b = 2$. Numbers next to arrows are log probabilities for a single word $\log P(e_t|F, e_1^{t-1})$, while numbers above nodes are log probabilities for the entire hypothesis up until this point.

In the first time step, we expand hypotheses for e_1 corresponding to all of the three words in the vocabulary, then keep the top two ("b" and "a") and delete the remaining one ("⟨/s⟩"). In the second time step, we expand hypotheses for e_2 corresponding to the continuation of the first hypotheses for all words in the vocabulary, temporarily creating $b*|V|$ active hypotheses. These active hypotheses are also pruned down to the b active hypotheses ("a b" and "b b"). This process of calculating scores for $b*|V|$ continuations of active hypotheses, then pruning back down to the top b, is continued until the end of the sentence.

One thing to be careful about when generating sentences using models, such as neural machine translation, where $P(E|F) = \prod_t^{|E|} P(e_t|F, e_1^{t-1})$ is that they tend to prefer shorter sentences. This is because every time we add another word, we multiply in another probability, reducing the probability of the whole sentence. As we increase the beam size, the search algorithm gets better at finding these short sentences, and as a result, beam search with a larger beam size often has a significant **length bias** towards these shorter sentences.

There have been several attempts to fix this length bias problem. For example, it is possible to put a prior probability on the length of the sentence given the length of the previous sentence $P(|E||F|)$, and multiply this with the standard sentence probability $P(E|F)$ at decoding time [6]:

$$\hat{E} = \operatorname*{argmax}_E \log P(|E||F|) + \log P(E|F). \tag{63}$$

This prior probability can be estimated from data, and [6] simply estimate this using a multinomial distribution learned on the training data:

$$P(|E||F|) = \frac{c(|E|, |F|)}{c(|F|)}. \tag{64}$$

A more heuristic but still widely used approach normalizes the log probability by the length of the target sentence, effectively searching for the sentence that has the highest average log probability per word [2]:

$$\hat{E} = \operatorname*{argmax}_E \log P(E|F)/|E|. \tag{65}$$

7.3 Other Ways of Encoding Sequences

In Section 7.1, we described a model that works by encoding sequences linearly, one word at a time from left to right. However, this may not be the most natural or effective way to turn the sentence F into a vector $\boldsymbol{h}$. In this section, we'll discuss a number of different ways to perform encoding that have been reported to be effective in the literature.

Reverse and Bidirectional Encoders: First, [19] have proposed a **reverse encoder**. In this method, we simply run a standard linear encoder over F, but instead of doing so from left to right, we do so from right to left.

$$\overleftarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overleftarrow{\text{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overleftarrow{\boldsymbol{h}}_{t+1}^{(f)}) & t \leq |F|, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{66}$$

The motivation behind this method is that for pairs of languages with similar ordering (such as English-French, which the authors experimented on), the words at the beginning of F will generally correspond to words at the beginning of E. Assuming the extreme case that words with identical indices correspond to each-other (e.g. f_1 corresponds to e_1, f_2 to e_2, etc.), the

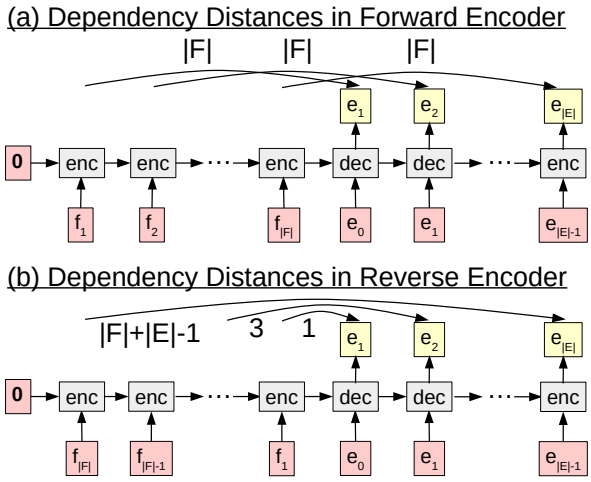

Figure 24: The distances between words with the same index in the forward and reverse decoders.

distance between corresponding words in the linear encoding and decoding will be $|F|$, as shown in Figure 24(a). Remembering the vanishing gradient problem from Section 6.3, this means that the RNN has to propagate the information across $|F|$ time steps before making a prediction, a difficult feat. At the beginning of training, even RNN variants such as LSTMs have trouble, as they have to essentially "guess" what part of the information encoded in their hidden state is being used without any prior bias.

Reversing the encoder helps solve this problem by reducing the length of dependencies for a subset of the words in the sentence, specifically the ones at the beginning of the sentences. As shown in Figure 24(b), the length of the dependency for f_1 and e_1 is 1, and subsequent pairs of f_t and e_t have a distance of $2t-1$. During learning, the model can "latch on" to these short-distance dependencies and use them as a way to bootstrap the model training, after which it becomes possible to gradually learn the longer dependencies for the words at the end of the sentence. In [19], this proved critical to learn effective models in the encoder-decoder framework.

However, this approach of reversing the encoder relies on the strong assumption that the order of words in the input and output sequences are very similar, or at least that the words at the beginning of sentences are the same. This is true for languages like English and French, which share the same "subject-verb-object (SVO)" word ordering, but may not be true for more typologically distinct languages. One type of encoder that is slightly more robust to these differences is the **bi-directional encoder** [1]. In this method, we use two different encoders: one traveling forward and one traveling backward over the input sentence

$$\overrightarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overrightarrow{\text{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overrightarrow{\boldsymbol{h}}_{t+-}^{(f)}) & t \geq 1, \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{67}$$

$$\overleftarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overleftarrow{\text{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overleftarrow{\boldsymbol{h}}_{t+1}^{(f)}) & t \leq |F|, \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{68}$$

which are then combined into the initial vector $\boldsymbol{h}_0^{(e)}$ for the decoder RNN. This combination

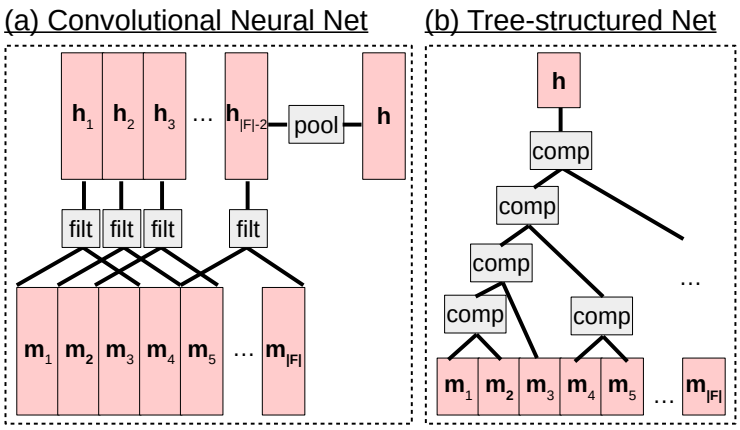

Figure 25: Examples of convolutional and tree-structured networks.

can be done by simply concatenating the two final vectors $\overrightarrow{\boldsymbol{h}}_{|F|}$ and $\overleftarrow{\boldsymbol{h}}_1$. However, this also requires that the size of the vectors for the decoder RNN be exactly equal to the combined size of the two encoder RNNs. As a more flexible alternative, we can add an additional parameterized hidden layer between the encoder and decoder states, which allows us to convert the bidirectional encoder states into an appropriately-sized state for the decoder:

$$\boldsymbol{h}_0^{(e)} = \tanh(W_{\overrightarrow{f}e}\overrightarrow{\boldsymbol{h}}_{|F|} + W_{\overleftarrow{f}e}\overleftarrow{\boldsymbol{h}}_1 + \boldsymbol{b}_e). \tag{69}$$

Convolutional Neural Networks: In addition, there are also methods for decoding that move beyond a simple linear view of the input sentence. For example, **convolutional neural networks** (CNNs; [8, 13], Figure 25(a)) are a variety of neural net that combines together information from spatially or temporally local segments. They are most widely applied to image processing but have also been used for speech processing (as "time-delay neural networks" [22]), as well as the processing of textual sequences. While there are many varieties of CNN-based models of text (e.g. [11, 14, 10]), here we will show an example from [12]. This model has n **filters** with a width w that are passed incrementally over w-word segments of the input. Specifically, given an embedding matrix M of width $|F|$, we generate a hidden layer matrix H of width $|F| - w + 1$, where each column of the matrix is equal to

$$\boldsymbol{h}_t = W\text{concat}(\boldsymbol{m}_t, \boldsymbol{m}_{t+1}, \ldots, \boldsymbol{m}_{t+w-1}) \tag{70}$$

where $W \in \mathbb{R}^{n \times w|\boldsymbol{m}|}$ is a matrix where the ith row represents the parameters of filter i that will be multiplied by the embeddings of w consecutive words. If $w = 3$, we can interpret this as $\boldsymbol{h}_1$ extracting a vector of features for f_1^3, $\boldsymbol{h}_2$ as extracting a vector of features for f_2^4, etc. until the end of the sentence.

Finally, we perform a **pooling** operation that converts this matrix H (which varies in width according to the sentence length) into a single vector $\boldsymbol{h}$ (which is fixed-size and can thus be used in down-stream processing). Examples of pooling operations include average, max, and k-max [11].

Compared to RNNs and their variants, CNNs have several advantages and disadvantages:

- On the positive side, CNNs provide a relatively simple way to detect features of short word sequences in sentence text and accumulate them across the entire sentence.

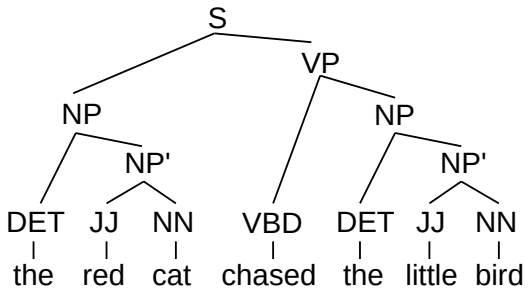

Figure 26: An example of a syntax tree for a sentence showing the sentence structure and phrase types (DET="determiner", JJ="adjective", NN="noun", VBD="past tense verb", NP="noun phrase", NP'="part of a noun phrase", VP="verb phrase", S="sentence").

- Also on the positive side, CNNs do not suffer as heavily from the vanishing gradient problem, as they do not need to propagate gradients across multiple time steps.
- On the negative side, CNNs are not quite as expressive and are a less natural way of expressing complicated patterns that move beyond their filter width.

In general, CNNs have been found to be quite effective for text classification, where it is more important to pick out the most indicative features of the text and there is less of an emphasis on getting an overall view of the content [12]. There have also been some positive results reported using specific varieties of CNNs for sequence-to-sequence modeling [10].

Tree-structured Networks: Finally, one other popular form of encoder that is widely used in a number of tasks are **tree-structured networks** ([16, 18], Figure 25(b)). The basic idea behind these networks is that the way to combine the information from each particular word is guided by some sort of structure, usually the syntactic structure of the sentence, an example of which is shown in Figure 26. The reason why this is intuitively useful is because each syntactic phrase usually also corresponds to a coherent semantic unit. Thus, performing the calculation and manipulation of vectors over these coherent units will be more appropriate compared to using random substrings of words, like those used by CNNs.

For example, let's say we have the phrase "the red cat chased the little bird" as shown in the figure. In this case, following a syntactic tree would ensure that we calculate vectors for coherent units that correspond to a grammatical phrase such as "chased" and "the little bird", and combine these phrases together one by one to obtain the meaning of larger coherent phrase such as "chased the little bird". By doing so, we can take advantage of the fact that language is **compositional**, with the meaning of a more complex phrase resulting from regular combinations and transformation of smaller constituent phrases [20]. By taking this linguistically motivated and intuitive view of the sentence, we hope will help the neural networks learn more generalizable functions from limited training data.

Perhaps the most simple type of tree-structured network is the **recursive neural network** proposed by [18]. This network has very strong parallels to standard RNNs, but instead of calculating the hidden state $\boldsymbol{h}_t$ at time t from the previous hidden state $\boldsymbol{h}_{t-1}$ as follows:

$$\boldsymbol{h}_t = \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h), \tag{71}$$

we instead calculate the hidden state of the parent node $\boldsymbol{h}_p$ from the hidden states of the left and right children, $\boldsymbol{h}_l$ and $\boldsymbol{h}_r$ respectively:

$$\boldsymbol{h}_p = \tanh(W_{xp}\boldsymbol{x}_t + W_{lp}\boldsymbol{h}_l + W_{rp}\boldsymbol{h}_r + \boldsymbol{b}_p). \tag{72}$$

Thus, the representation for each node in the tree can be calculated in a bottom-up fashion.

Like standard RNNs, these recursive networks suffer from the vanishing gradient problem. To fix this problem there is an adaptation of LSTMs to tree-structured networks, fittingly called **tree LSTMs** [21], which fixes this vanishing gradient problem. There are also a wide variety of other kinds of tree-structured composition functions that interested readers can explore [17, 4, 5]. Also of interest is the study by [15], which examines the various tasks in which tree structures are necessary or unnecessary for NLP.

7.4 Exercise

In the exercise for this chapter, we will create an encoder-decoder translation model and make it possible to generate translations.

Writing the program will entail:

- Extend your RNN language model code to first read in a source sentence to calculate the initial hidden state.
- On the training set, write code to calculate the loss function and perform training.
- On the development set, generate translations using greedy search. Look at them to see if they look good or not. We will discuss how to evaluate them automatically next time.

Potential improvements to the model include: Implementing beam search. Implementing an alternative encoder.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[2] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of the Workshop on Syntax and Structure in Statistical Translation*, pages 103–111, 2014.

[3] Lonnie Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3(4):345–366, 1991.

[4] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 334–343, 2015.

[5] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 199–209, 2016.

[6] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 823–833, 2016.

[7] Mikel L Forcada and Ramón P Ñeco. Recursive hetero-associative memories for translation. In *International Work-Conference on Artificial Neural Networks*, pages 453–462. Springer, 1997.

[8] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[9] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1700–1709, 2013.

[10] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[11] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 655–665, 2014.

[12] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Molding cnns for text: non-linear, non-consecutive convolutions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1565–1575, 2015.

[15] Jiwei Li, Thang Luong, Dan Jurafsky, and Eduard Hovy. When are tree structures necessary for deep learning of representations? pages 2304–2314, 2015.

[16] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1):77–105, 1990.

[17] Richard Socher, John Bauer, Christopher D. Manning, and Ng Andrew Y. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 455–465, 2013.

[18] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. pages 129–136, 2011.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

[20] Zoltán Gendler Szabó. Compositionality. *Stanford encyclopedia of philosophy*, 2010.

[21] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

[22] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. 37(3):328–339, 1989.