9 Neural MT 3: Other Non-RNN Architectures

In this section, we describe another method of generating sentences that does not rely on RNNs.

9.1 Translation through Generalized Sequence Encoding

Before getting into specific methods for translation with non-RNN-based architectures, we'll first go over a more general way of thinking about the generation of output sequences, a method we'll call **sequence encoding**.

First, let us define a sequence encoder as a neural network that takes in a sequence of vectors $X = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{|X|}$, and converts it to another sequence of vectors $H = \boldsymbol{h}_1, \ldots, \boldsymbol{h}_{|H|}$. In most cases, we'll consider the case where the number of vectors in the input and output are the same, in other words $|X| = |H|$.

To tie this together with previously introduced concepts, we can view the RNN-based encoder for attention introduced in the previous chapter, as demonstrated in Equation 68 to Equation 70 as an example of a sequence encoder. Specifically, this uses bi-directional RNNs to convert the input into an output matrix $H^{(f)}$ representing the source sentence. Alternatively, we could also view the uni-directional RNNs used in language models of Equation 46 as another type of sequence encoder that only uses an encoder in a single direction.

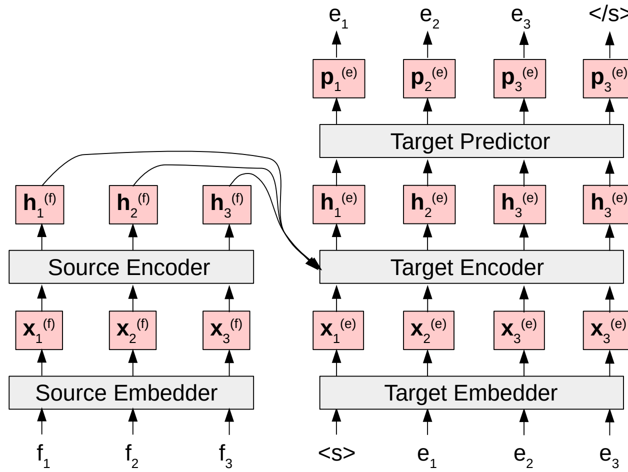

Figure 27: A general example of sequence encoding for sequence-to-sequence prediction.

Given this background, we can express just about any type of sequence-to-sequence model in the general framework shown in Figure 27. Here, we go from discrete symbols F and E in the source and target, to embeddings $X^{(f)}$ and $X^{(e)}$ respectively:

$$X^{(f)} = \text{src-embed}(F), \tag{81}$$

$$X^{(e)} = \text{trg-embed}(\langle \text{s} \rangle : E). \tag{82}$$

Notably, when encoding E, we pad it by adding a start-of-sentence symbol $\langle$s$\rangle$ at the beginning of the sentence. This is to ensure that words are only fed to the target encoder *after* they are predicted (as is obvious by looking at the time steps in the figure).

We then use a source encoding function to convert the source embeddings $X^{(f)}$ to the encoded vectors $H^{(f)}$

$$H^{(f)} = \text{src-encode}(X^{(f)}). \tag{83}$$

Next, we use a target encoder, which takes as input both the source encoded sequence, and target embeddings to generate encoded vectors for the target:

$$H^{(e)} = \text{trg-encode}(X^{(e)}, H^{(f)}). \tag{84}$$

Finally, we make predictions of the probability using a target predictor (such as a softmax function), to get the probabilities of the output to calculate loss functions or make predictions

$$P^{(e)} = \text{trg-predict}(H^{(e)}). \tag{85}$$

Here, when we make predictions we make sure to add the end-of-sentence symbol ⟨/s⟩ to ensure that we properly predict the end of the sentence, as was also necessary in previous chapters.

This framework encompasses the previous methods that we described. For example, the attentional sequence-to-sequence models introduced in the previous chapter define src-encode($\cdot$) as a bi-directional RNN, and trg-encode($\cdot$) as a uni-directional RNN with attention applied to reference $H^{(f)}$ at each time step. But notably it also opens up a wide array of other possibilities – we can define src-encode($\cdot$) and trg-encode($\cdot$) to be any function that we wish, with some conditions. One important condition is that the target predictor must not have access to words $e_{\geq t}$ when predicting e_t, as that would break down the basic assumption that we are predicting words one at a time from left-to-right, which is necessary to correctly calculate probabilities or make predictions (remember Equation 4). For example, we could not (naively) use a bi-directional RNN as trg-encode($\cdot$), as this would give the model access to information about e_t (through the backward RNN component) at training time. Because this information would not be available at test time when the model had to actually make predictions one-by-one from left to right, this would cause a training/test mismatch and cause the model to not function properly.

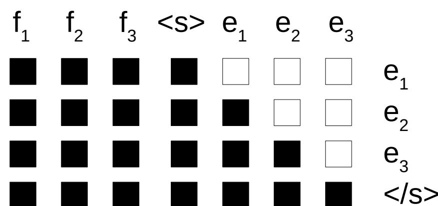

Figure 28: An example of masking for sequence-to-sequence prediction. Rows are inputs, columns are outputs, dark squares indicate used information, and light squares indicate unused information.

One way to explicitly satisfy this condition is by limiting the class of functions that we can use in trg-encode($\cdot$) to those that do not reference future information (e.g. by only using uni-directional RNNs, or other similar models that rely solely on past context). There is another way of doing so that is computationally convenient for some of the models we introduce below:

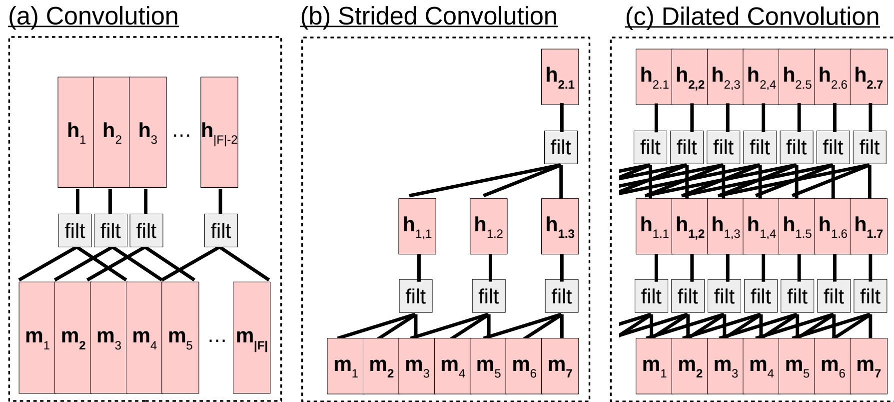

Figure 29: Several varieties of CNNs: vanilla, strided, and dilated.

masking. Remember back to Section 6.5, masking is used to cancel out parts of computation that we would like to have no effect on our final result. In this case, we would like to have the words $e_{\geq t}$ not have an effect on the calculation of e_t, so when calculating the probability of e_t, we can mask out the values of $e_{\geq t}$. A graphical example of this masking procedure can be found in Figure 28, where this mask can be applied to any layer in the target embedders or encoders to ensure that information from future words $e_{\geq t}$ doesn't leak back when making the prediction of e_t.

Now that we have a general framework for translation using sequence encoders, lets look at a couple varieties of neural network that we can use for this task.

9.2 Convolutional Neural Networks

Convolutional neural networks (CNNs; [5, 16, 11]), Figure 29(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 well as the processing of textual sequences. While there are many varieties of CNN-based models of text (e.g. [9, 12, 8]), here we will show an example from [10]. 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

$$\boldsymbol{h}_t = W \text{concat}(\boldsymbol{m}_t, \boldsymbol{m}_{t+1}, \ldots, \boldsymbol{m}_{t+w-1}) \tag{86}$$

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. This resulting matrix H can then be used as a generalized sequence encoder in the sequence-to-sequence models above.

CNNs have one major advantage over RNNs: the calculation of each $\boldsymbol{h}_t$ is independent of others. Remembering back to the introduction of RNNs, the value of $\boldsymbol{h}_t$ is dependent on the value of $\boldsymbol{h}_{t-1}$, which means that it is necessary to calculate the values in sequence, making the number of consecutive operations linear in the length of the sequence. On the other hand, for

CNNs $\boldsymbol{h}_t$ is not dependent on any other value of $\boldsymbol{h}$, and thus all values in H can be calculated *simultaneously, in parallel.* This is especially good news for highly parallelized computation hardware such as GPUs, which excel at this kind of parallel computation.

However, one major disadvantage of CNNs is that they can only capture limited context, up to the filter size. This is obviously a problem in sequence-to-sequence models, where having a global view of the entirety of content in the sentence is important. As a fix to this problem, it is common to use **strided convolutions**, convolutions that skip over positions to reduce the scale of the input. For example, in Figure 29(b), we can see two layers of CNN, where each layer has a filter width of 3, and stride of 2, indicating that it calculates filters of width 3, but skips over every other vector in each layer. By stacking together layers of strided convolutions, it becomes possible to have each vector in the final layer cover an exponentially increasing number of input vectors, expanding the effective amount of area that each vector represents.

Strided CNNs have a disadvantage however: unlike regular CNNs they reduce the size of the output compared to the input, and thus we no longer have a single output vector for every input vector. This may or may not be a major problem for the source encoder in Section 9.1, but it is certainly a problem for the target encoder, where we must make one prediction for each input word and thus need to have a vector for each output position. To fix this problem we can turn to a technique called **dilated convolutions**, which are strided convolutions shifted over each time step in the input [8], as shown in Figure 29(c). These combine the good features of standard CNNs (one vector per input) and strided CNNs (wider context window), and have been shown to be quite effective for both language and sequence-to-sequence modeling [8, 6].

Finally, it should be noted that CNNs are commonly used in sentence classification tasks, where they have generally allowed for superior performance to RNNs due to their ability to efficiently detect features of short word sequences in sentence text, which is often sufficient for high-accuracy classification [10]. To perform classification, we need a single vector that summarizes all the features in the sentence to feed into the final classifier. To get this vector, 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 downstream processing). Examples of pooling operations include:

Max: This takes the element-wise max of all of the vectors in H. Intuitively, this demonstrates whether a feature appeared anywhere in the sentence, and is the most common variety of pooling function.

Average: This takes the element-wise average of all the vectors in H, indicating how common the features is in the sentence.

k**-max:** This takes the k max elements of the vectors in H, counting up to k occurances of any particular feature [9].

Dynamic: Instead of performing an operation over the entire sentence, this instead performs an operation over k slices of the sentence (e.g. beginning, middle, and end if $k = 3$) [14].

9.3 Self Attention

Another method for encoding sequences of vectors that has recently been shown to be effective is **intra-attention** or **self attention** [4, 13, 15]. The concept behind this method is extremely simple: unlike standard attention, where a target vector is used to perform attention over source vectors, instead all vectors in an input X attend to *themselves*.

Specifically taking the example of [15], this is done by taking the original matrix X, and three parameter matrices W_Q, W_K and W_V, and calculating three transformations, the *query* matrix, the *key* matrix, and the *value* matrix:

$$Q = W_Q X, \tag{87}$$

$$K = W_K X, \tag{88}$$

$$V = W_V X. \tag{89}$$

Then, analogous to the standard attention in Equation 73 the columns in the query matrix are used to calculate attention over the columns in the key matrix:

$$a_{i,j} = \text{attn_score}(\boldsymbol{q}_i, \boldsymbol{k}_j). \tag{90}$$

We then take the softmax to normalize the values to each query vector to sum to one

$$\boldsymbol{\alpha}_i = \text{softmax}(\boldsymbol{a}_i), \tag{91}$$

and multiply the attention probabilities with the columns of the value matrix to get our final encoding for each time step:

$$\boldsymbol{h}_i = V\boldsymbol{\alpha}_i. \tag{92}$$

Compared to convolutions, self attention has several advantages and disadvantages. One advantage is that self attention is that it is even more efficient than convolution, and can be implemented almost entirely with highly optimized matrix multiplications. Another advantage is that any word in the sequence can easily reference any word in the input sequence, not limited by the fixed-size context provided by convolutions.

However, this advantage of being insensitive to context is also a disadvantage: the vanilla self-attentional model has no concept of whether words are nearby and far away, essentially making it a bag-of-words model with no concept of word ordering. In order to overcome this problem, it is common to use **positional embeddings** [6, 15]. Positional embeddings are essentially vectors that encode not the identity of the word itself, but rather the word's position in the sentence, so the positional embedding will be equivalent for all words e_1, all words e_2, etc. These positional embeddings can then be added to the standard word embedding

$$x'_t = x_t + \text{pos-embed}(t), \tag{93}$$

giving each word a concept of its position in the sentence. These embeddings can either be learned directly [6], or calculated using a pre-defined function that depends on the word's position [15], with the former allowing for more more flexibility, but the latter having fewer parameters and having a higher probability of generalizing to positions not seen in the training data.

9.4 Multi-head Attention

While attention provides a powerful method to summarize together information from a variable length sequence into a particular fixed-length vector, it is also restrictive – every row of H is combined together with the same weight specified by $\boldsymbol{\alpha}$. However, because the function of attention (or sequence encoders in general), is to combine together various parts of information scattered across the input sequence into a vector that is conducive to making final predictions, it also makes sense that we might want to take *different varieties of information from different places in the input sequence*. For example, let's take the sentence "If we are going to play, I'm going to have to buy a new racquet and balls." When choosing the translation of a word such as "play", we definitely will need information about the identity of the word "play" itself, but also may need information about syntactically-related words such as the verb's subject "we", or topical words indicating the topic of the sentence such as "racquet" or "balls". Thus, depending on the type of information we want to capture, we may need to use different $\boldsymbol{\alpha}$ to capture this information most effectively: sharp, peaky $\boldsymbol{\alpha}$ to focus on a single word such as "play", or flat distributed $\boldsymbol{\alpha}$ to focus on many topical words such as "play", "racquet", and "balls".

Multi-head attention is a method that operationalizes this intuition. The way the method works is by defining N attention "heads", which each are parameterized by different parameters (e.g. $W_{K,n}$, $W_{Q,n}$, and $W_{V,n}$ using the example from the previous section). These different heads are then used to calculate separate encoded vectors, which are finally concatenated together for use downstream. This concept has been used to calculate different attention heads to calculate context vectors or copy words in summarization models [1], or simply to calculate multiple attention heads to capture different features of the input sequence [15].

9.5 A Case Study: The Transformer Model

The most widely known example of a non-RNN based sequence-to-sequence model is the so-called "Transformer" model of [15]. This model is notable for both its computational efficiency and good translation results, and it and its variants are now widely used in translation systems. While giving a full treatment of this model is beyond the scope of this work, it essentially falls within the src-encode($\cdot$), trg-encode($\cdot$) paradigm described earlier in this chapter.

The source encoder of the Transformer works by first adding positional encodings, then running through multiple layers of:

1. multi-headed self attention with residual connections and layer normalization [2], which helps normalizes the value of each layer to be zero mean and unit variance.

2. a feed-forward layer with residual connections and layer normalization.

The target encoder is similar, adding positional encodings then running through multiple layers of:

1. multi-headed self attention over only the target, with residual connections and layer normalization.

2. multi-headed self attention over the source and target, with residual connections and layer normalization.

3. a feed-forward layer with residual connections and layer normalization.

Details of the model can be found in the original paper [15], and its corresponding official implementation.[30]

9.6 Further Reading

There are a number of interesting additional topics regarding non-RNN-based translation methods.

Efficient Decoding for Self-attention, CNNs: One problem with self-attentional and CNN-based models is that the speed of decoding decreases significantly as the number of elements in the output increases, as each output word must be run through several layers multiple times. To alleviate this problem, [17] propose **average attention networks**, which use averaging of the output embeddings over time steps. This makes it possible to remember which words have been previously generated in a more compact form, making calculation significantly cheaper.

Non-autoregressive Translation: To further remove the dependence on previously generated outputs, [7] propose **non-autoregressive** translation, which generates each of the output words independently. In order to do so, they need to choose in advance the word ordering and number of words that each word will generate, reminiscent of the the classical IBM models covered in Section 12. This allows for fully parallel decoding over time steps, which is efficient on hardware such as GPUs, but also makes it difficult to consider interactions between the output words, leading to decreased accuracy compared to standard auto-regressive models.

Combining Recurrent and Non-recurrent Architectures: While the original research papers proposing non-RNN architectures tended to eschew RNNs completely, [3] show that combining RNNs with non-RNN models can result in improved accuracy. In particular, RNNs were found to be efficient on the decoder side of the model, while non-RNN models were largely sufficient for the encoder side.

9.7 Exercise

For this chapter, it would be interesting to implement convolution or self-attention as an additional method for encoding the input in your exisiting translation code. Try to write in a modular way, perhaps following the example of Section 9.1, which would allow you to try different architectures.

As a challenge, you can attempt to implement the full transformer model in Section 9.5. Be aware that this is a challenge: there are many moving parts to this model, and it is notoriously fragile. After performing your initial implementation, it would be best to reference an existing implementation (ideally the official one) to make sure that you get comparable accuracy, and if you don't you can read their code to figure out where the differences are.

[30] `https://github.com/tensorflow/tensor2tensor`

References

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. *arXiv preprint arXiv:1602.03001*, 2016.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Mike Schuster, Noam Shazeer, Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 76–86. Association for Computational Linguistics, 2018.

[4] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 551–561, 2016.

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

[6] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1243–1252, 2017.

[7] Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. Non-autoregressive neural machine translation. 2018.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. 2017.

[14] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 801–809, 2011.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

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

[17] Biao Zhang, Deyi Xiong, and jinsong. Accelerating neural transformer via an average attention network. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1789–1798. Association for Computational Linguistics, 2018.