Recurrent Neural Networks (RNNs)

Swapnoneel Kayal

22 June-28 June 2021

1 Recurrent Neural Networks (RNNs)

1.1 Why do we need Sequence Models?

So far, we have only worked with data where we do not need to bother with whether something occurs earlier or later in the data. But we cannot deal with all learning tasks this way; for example, making a transcription for a voice recording involves the time dimension, as the words need to be arranged in the order in which they were spoken. Similarly, in the analysis of a DNA sequence, it is important to observe the order in which nucleotides occur in the sequence. For such cases, we need a new type of neural network architecture; the previous ordinary networks and CNNs cannot deal with sequence data appropriately. This is why we need sequence models.

1.2 Notation

A new kind of notation is used for sequence models.

1. We use $x^{(i)<t>}$ to denote the t^{th} element of the input sequence belonging to the i^{th} data instance. (The letter t is used to create a notion of time.)
2. Similarly, we use $y^{(i)<t>}$ to denote the t^{th} element of the output sequence belonging to the i^{th} data instance.
3. We use $T_x^{(i)}$ to denote the length of the input sequence belonging to the i^{th} data instance.
4. We use $T_y^{(i)}$ to denote the length of the output sequence belonging to the i^{th} data instance.

1.3 Basic RNN Architecture

A typical recurrent neural network looks like this:

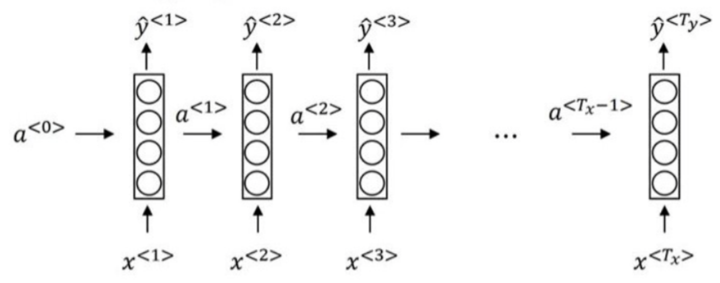


Figure 1: A recurrent neural network (RNN).

This is the unrolled form of the RNN; there is a more compact way of representation, but this one is easier to understand. Accompanying the RNN diagram, we have the following equations:

$$
\begin{aligned}
a^{<0>} &= 0 \\
a^{<t>} &= g\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \\
\hat{y}^{<t>} &= g\left(W_{ya}a^{<t>} + b_y\right)
\end{aligned}
$$

The notation can be simplified as follows:

$$
\begin{aligned}
a^{<t>} &= g\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right) \\
\hat{y}^{<t>} &= g\left(W_y a^{<t>} + b_y\right)
\end{aligned}
$$

Here, W_a is the matrix obtained by horizontally stacking together the weight matrices W_{aa} and W_{ax}, $\left[a^{<t-1>}, x^{<t>}\right]$ is the matrix obtained by vertically stacking together the matrices $a^{<t-1>}$ and $x^{<t>}$, and W_y is the same as W_{ya}.

The loss function for any data instance is as follows:

$$
L(\hat{y}, y) = \sum_{t=1}^{T_y} L(\hat{y}^{<t>}, y^{<t>})
$$

where:

$$
L(\hat{y}^{<t>}, y^{<t>}) = -\left(y^{<t>}\log\left(\hat{y}^{<t>}\right) + (1 - y^{<t>})\log\left(1 - \hat{y}^{<t>}\right)\right)
$$

As always, we use backpropagation with gradient descent to improve the parameters step by step. The parameters that are trained are W_a and W_y.

1.4 Types of RNNs

Different types of RNNs, on the basis of number of inputs and number of outputs, are:

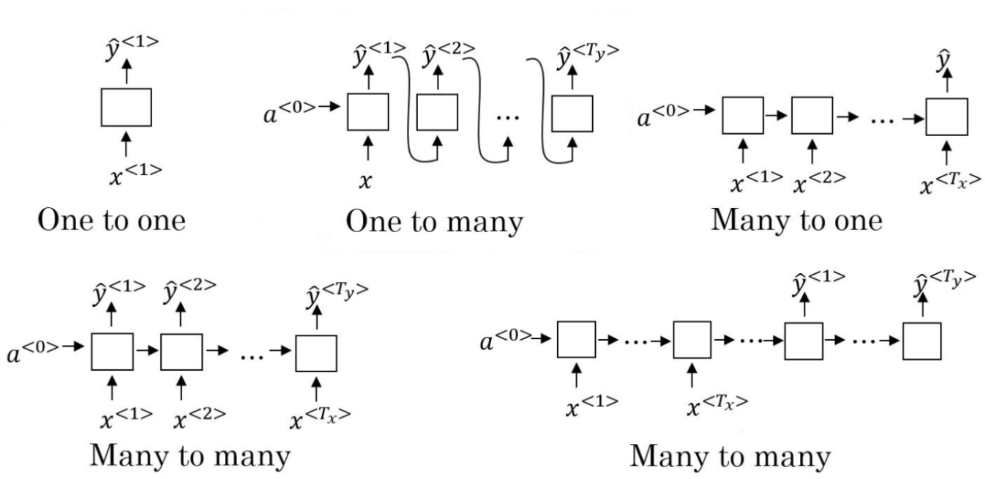


Figure 2: Types of recurrent neural networks.

A one-to-one RNN is essentially equivalent to a logistic regression unit.

A one-to-many RNN has just one input, and each output after the first is generated by using the previous output as the next input, as shown in figure. An example where this is used is in image captioning, where we input a single image and generate an appropriate sentence using sequence output.

A many-to-one RNN has a sequence input and a single output after the last input. An example where this is used is sentiment classification, where we input one or more sentences as sequence input and we output whether the emotions being conveyed are positive or negative.

A many-to-many RNN may be of two different types. The first one in the diagram is used when $T_x = T_y$, i.e., each element of the input sequence has a corresponding element in the output sequence. For example, this is used when, given a sentence, we want to identify the names in it. We would output 1 for name words, and 0 for other words. The second one in the diagram is used in cases where there is no element-to-element correspondence between the input and the output. So, all the output is generated only after the entire input sequence has been processed. An example where this is used is in machine translation, such as from English to French.

1.5 Language Models

A language model is used to compare probabilities of two sentences. This basically means, if someone speaks any sentence randomly, what is the likelihood that he/she speaks a particular sentence? We calculate the probabilities of both sentences and compare. This is useful in speech recognition and transcription systems, to compare the likelihoods of several sentences that are possible transcriptions. (This enables systems to choose between homophones, like pair and pear.)

To calculate the probability of a sentence being spoken randomly, we first find a way to represent sentences as sequences for RNNs. We thus create a dictionary of all possible words that we can come across, using one index for each word. Then, we represent each word using a one-hot encoding, where the element which corresponds to word's index in the dictionary has value 1 and all others have 0. Note that we should also give a place in the dictionary to 'end of sentence' (EOS) and to unknown words (UNK).

To create a language model, we use a many-to-many RNN as shown here:

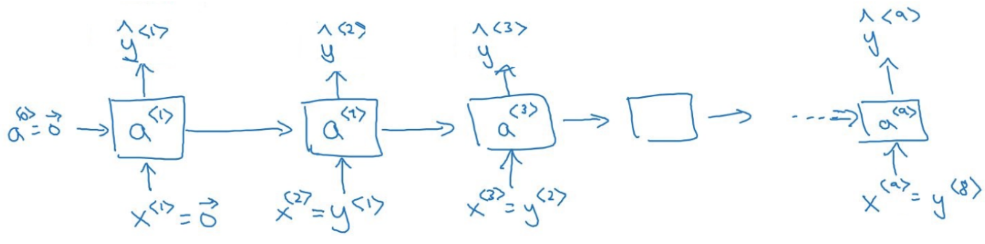


Figure 3: A language model architecture.

In a language model, the first input element is a zero. So as our first output, we generate a vector containing the probabilities of every possible word for occurring as the first word of a sentence. For our second input, we give the actual first word of the sentence, and output the probabilities of every possible word for occurring as the second word of a sentence that begins with the real first word. Thus, we have that $\hat{y}^{<t>}$ is the vector of probabilities of words occurring at the t^{th} position of the sentence given the previous $t-1$ words of the sentence. For example, consider the sentence "I want some pear salad." In this case, $\hat{y}^{<4>}$ is the vector of probabilities for the next word in the sentence given the first three words are "I want some". When we reach the end of the sentence, we calculate the probability of the sentence as:

$$P(y^{<1>}, \ldots, y^{<T_y>}) = P(y^{<1>}) \cdot P(y^{<2>}|y^{<1>}) \cdot P(y^{<3>}|y^{<1>}, y^{<2>}) \cdots P(y^{<T_y>}|y^{<1>}, \ldots, y^{<T_y-1>})$$

A language model can be trained on a large text corpus using the same loss function as discussed above in general RNNs.

1.6 Novel Sequence Sampling

Sampling novel sequences involves getting random meaningful sequence outputs from a language model. For example, we can use this technique to generate random sentences. The language model is first trained (as already discussed). Then, the value of $x^{<1>}$ is passed in as 0. As the first word, we select any word randomly, but we use the probabilities output in $y^{<1>}$ for the word selection. We then pass in the selected first word as the second input, and this process goes on until the end. This generates a random sentence.

Note that, if we select the highest probability words from the output vectors, instead of choosing words randomly with the output probabilities as we did, we would always end up with the same output sentence. So that doesn't work out.

1.7 Gated Recurrent Units (GRUs)

Just like other neural networks, RNNs also suffer from the problems of vanishing and exploding gradients. While exploding gradients can be cured through gradient clipping (any gradient values too large in magnitude are clipped to a maximum limit), it requires a little more effort to solve the vanishing gradient problem. Using GRUs instead of ordinary RNN units is one way.

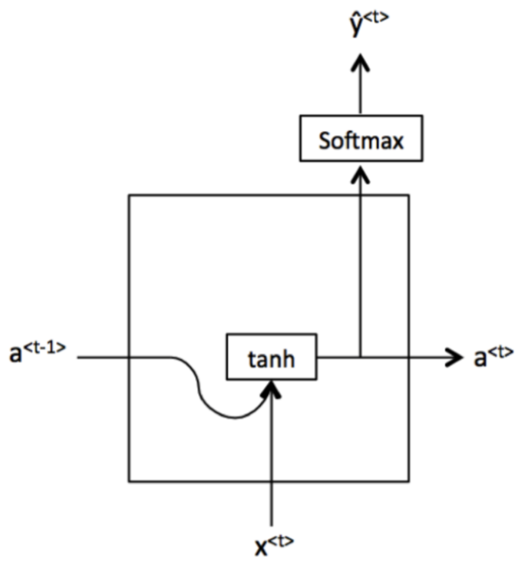


Figure 4: An RNN unit.

We can represent an ordinary RNN unit as shown above. (We've considered the tanh activation function here.) In a similar way, a gated recurrent unit can be represented as follows:

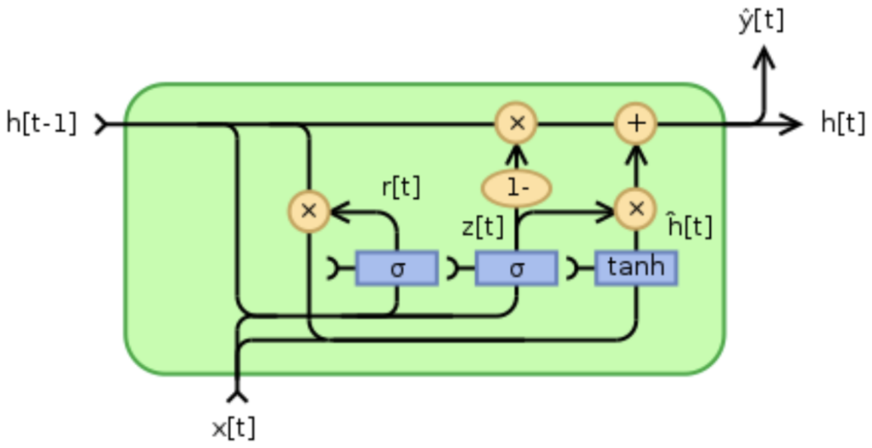


Figure 5: A gated recurrent unit.

Here, we have replaced $a^{<t>}$ with $h^{<t>}$. The equations in play here are:

$$\begin{aligned}
r^{<t>} &= \sigma\left(W_r\left[h^{<t-1>}, x^{<t>}\right] + b_r\right)\\
z^{<t>} &= \sigma\left(W_z\left[h^{<t-1>}, x^{<t>}\right] + b_z\right)\\
\hat{h}^{<t>} &= \tanh\left(W_h\left[r^{<t>} * h^{<t-1>}, x^{<t>}\right] + b_h\right)\\
h^{<t>} &= z^{<t>} * \hat{h}^{<t>} + \left(1 - z^{<t>}\right) * h^{<t-1>}
\end{aligned}$$

This way, we can use a weighted sum of the previous activation and the result of the current layer to calculate the current layer activation, and this helps avoid the vanishing gradients problem.

1.8 Long Short Term Memory (LSTM)

LSTM units are another way to deal with sequence learning tasks. These are more widely used than GRUs or ordinary RNNs. An LSTM unit is represented as follows:

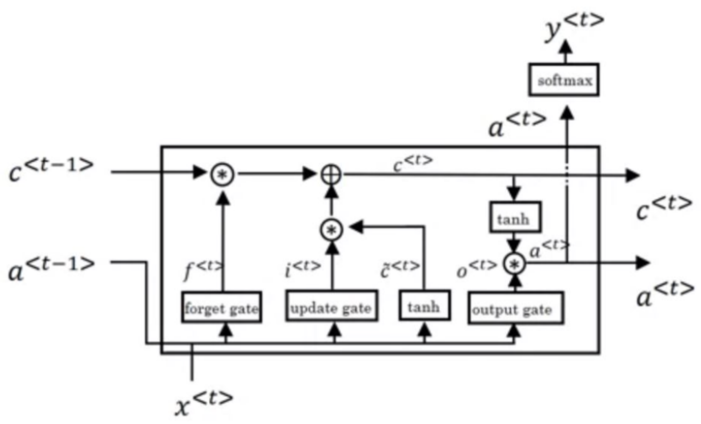


Figure 6: An LSTM unit.

In LSTM units, we keep two kinds of values, $a^{<t>}$ and $c^{<t>}$. The math is as follows:

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right)\\
\Gamma_u &= \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right)\\
\Gamma_f &= \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right)\\
\Gamma_o &= \sigma\left(W_o\left[a^{<t-1>}, x^{<t>}\right] + b_o\right)\\
c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}\\
a^{<t>} &= \Gamma_o * \tanh\left(c^{<t>}\right)
\end{aligned}$$

LSTM units have been found to deal very effectively with sequence tasks, and vanishing gradients are encountered much less frequently when using LSTMs.

1.9 Bidirectional RNNs

Often, in sequence learning tasks, it is useful to consider the entire input sequence when producing any outputs. In case this is true, and $T_x = T_y$, we prefer using bidirectional RNNs over the second kind of many-to-many RNNs (see figure 19.2). In bidirectional RNNs, we have the sequence running in both directions, as shown:

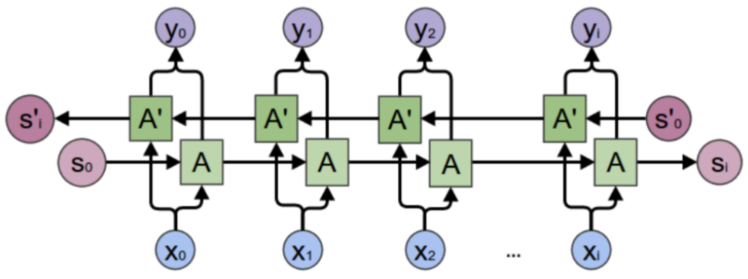


Figure 7: A bidirectional recurrent neural network.

2 Word Representations

2.1 Word Embeddings

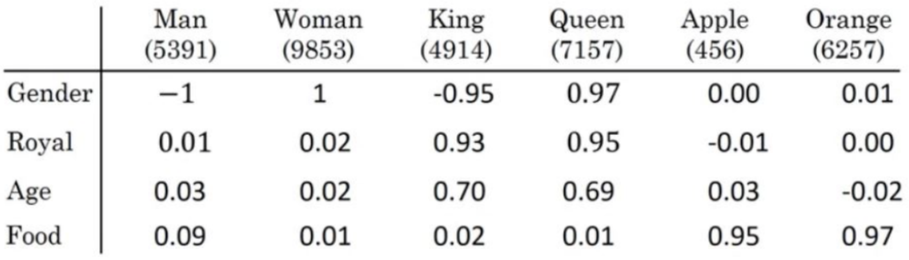
So far, we have used simple one-hot encoded word representations, based on a dictionary. This is not good enough; for example, suppose the RNN has learned that the word ‘juice’ is a very probable next word after the words “I want a glass of apple”. It would be very good if the model recognized apple and orange as similar things (both fruits), so that it would predict ‘juice’ after the words “I want a glass of orange” as well. To achieve this, we use word embeddings.

Word embeddings are featurized word representations, i.e., the i^{th} element of the embedding for each word represents the relation of the word to some feature. For example, if the feature was ‘fruit’, both apple and orange would have high values. In practice, these features may not represent something humanly understandable (like ‘fruit’), since they are learned by the model.

If we use word embeddings, then the features are learned so that the vectors representing similar words occur close to each other in the multidimensional space of features. Similar word embeddings of words helps the model recognize similar words, and thus work better. Word embeddings can be learnt on a large text corpus (or we could just use already available word embeddings), and then used (like transfer learning) for the task at hand.

2.2 Analogies in Word Embeddings

Word embeddings also contain analogies, which turn out to be useful for certain tasks. For example, through its feature vectors, a word embedding can solve questions like: man $\to$ woman as king $\to$?, for which the answer is obviously queen. For example, consider this (oversimplified) word embedding:



	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	−1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

Figure 8: A word embedding.

We analyze how we can solve the same problem: find X if man $\to$ woman as king $\to$ X. If this analogy holds, then we have: $e_{\text{man}} - e_{\text{woman}} \approx e_{\text{king}} - e_{\text{X}}$. So, we simply calculate the vector $e_{\text{king}} - e_{\text{man}} + e_{\text{woman}}$, and we find the word whose embedding vector is closest to it. This word is our answer. Thus, the answer is:

$$\text{X} = \arg\max_{w} \text{sim}\left(e_w, e_{\text{king}} - e_{\text{man}} + e_{\text{woman}}\right)$$

where sim is a similarity function. The cosine similarity function is often used:

$$\text{sim}(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$$

2.3 Word Embedding Matrix

The embedding matrix E is the matrix consisting of embeddings of all the words in our dictionary. Suppose our vocabulary size is m, and we have learnt word embeddings with n features. Then the embedding matrix has shape $n \times m$, with each column representing the word embedding of a single word from our vocabulary.

Let o_k be the one-hot encoding of the k^{th} word in our dictionary. Then, the word embedding for the k^{th} word would be $E \cdot o_k$. Computationally, though, a word embedding is selected by slicing through the embedding matrix, since matrix multiplication has higher computational costs.

2.4 Word2Vec Model

The Word2Vec model is one method of learning word embeddings. Suppose we have a dictionary of m words, and the embeddings are trained to have n features. To train our embeddings, we first select a word at random from our text corpus, and call it the context word c. Then, from some window of a few words (maybe 4-5) surrounding the context word, we select another word at random and call it the target word t.

To make the context and target words for each such selection have similar word embeddings, we actually work on a different supervised learning problem; given the context word, we try predicting the target word. For this purpose, we feed the context word embedding $e_c = E \cdot o_c$ into a softmax unit. Each word in the dictionary is given a corresponding parameter vector θ, and the calculation done in the softmax unit is:

$$P(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{m} e^{\theta_j^T e_c}}$$

This way, the softmax unit uses the parameter vector θ of every possible target word to calculate its own probability of being the target word. The loss function used is:

$$L(\hat{y}, y) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

where $\hat{y}$ is the vector of probabilities calculated by the softmax unit, and y is the one-hot encoded vector for the actual target word. Using gradient descent with this loss function, we can train pretty good word embeddings.

2.5 Negative Sampling

The Word2Vec technique has very high computational cost, because of the softmax unit. To solve this problem, negative sampling is used. Every time we pick a context word and target word, we also pick a number of negative samples; we actually just pick any k random words in our dictionary, since randomly selected words are still very likely to be unrelated to the context word in case of a good-enough vocabulary.

As before, we calculate e_c for the context word, but instead of a softmax unit, we now use a binary classifier for every possible target word, which uses the sigmoid activation function, and outputs $\sigma(\theta_t^T e_c)$. We train the classifier corresponding to the chosen target word with an expected output of 1, and then we train the classifiers corresponding to the k random (assumed negative) samples with expected outputs of 0. We simply ignore the other classifiers (for words other than our $k+1$ chosen words) at this step, and we move on to choose another context word. This way, computational costs are reduced significantly.

2.6 GloVe Model

The global vectors for word representation (GloVe) model is another method of training word embeddings. In this method, we first calculate X_{ij}, the number of times j occurs in the context of i in our text corpus, for all i and j. Then, the optimization problem is as follows:

$$\min_{\theta,b,e} \sum_{i=1}^{m} \sum_{j=1}^{m} f(X_{ij}) \left(\theta_i^T e_j + b_i + b'_j - \log X_{ij}\right)^2$$

It is easy to see that θ and e become totally symmetric in the GloVe model. Thus, the final embedding for any word w is taken as $\frac{e_w + \theta_w}{2}$. Also, $f(X_{ij})$ is kept as 0 if $X_{ij} = 0$, to zero out the undefined value of $\log(0)$, and it is allowed to have other values for other cases.

2.7 Debiasing Word Embeddings

Being trained on a random text corpus, a word embedding may often suffer from stereotypes. An analogy such as man $\rightarrow$ scientist as woman $\rightarrow$? may produce an answer like 'homemaker'. Obviously, we would want to remove stereotypes related to gender, race, ethnicity, sexual orientation, etc. from word embeddings.

Let us formulate a solution to the gender bias problem. We first find out the approximate direction of gender bias in our word embeddings, by taking the average value of several vectors like $e_{\text{he}} - e_{\text{she}}$, $e_{\text{boy}} - e_{\text{girl}}$, $e_{\text{man}} - e_{\text{woman}}$, $e_{\text{him}} - e_{\text{her}}$, etc. We then find the $n-1$ dimensional hyperplane perpendicular to this direction, which represents the non-gender-bias direction.

Next, we project the vector of every non-definitional word (scientist, homemaker, doctor, etc., which should be gender independent) onto the non-gender-bias hyperplane, to get rid of gender biases. But we are not done yet; it may so happen that 'he' is still closer to 'scientist' than 'she', and 'she' is closer to homemaker than 'he'. To solve this, we also need to equalize the distances of each definitional word (such as 'he' and 'she') on opposite sides of the non-gender-bias hyperplane.

Let g be the vector in the direction of gender bias. Then, for debiasing a non-definitional word, we replace its word embedding e with:

$$e_{\text{debiased}} = e - \frac{e \cdot g}{\|g\|_2^2} * g$$

The equalization of pairs of non-definitional words is more complex:

$$\begin{aligned}
\mu &= \frac{e_{w1} + e_{w2}}{2} \\
\mu_B &= \frac{\mu \cdot g}{\|g\|_2^2} * g \\
\mu_\perp &= \mu - \mu_B \\
e_{w1B} &= \frac{e_{w1} \cdot g}{\|g\|_2^2} * g \\
e_{w2B} &= \frac{e_{w2} \cdot g}{\|g\|_2^2} * g \\
e_{w1B}^{\text{corrected}} &= \sqrt{|1 - \|\mu_\perp\|_2^2|} * \frac{e_{w1B} - \mu_B}{\|(e_{w1} - \mu_\perp) - \mu_B\|} \\
e_{w2B}^{\text{corrected}} &= \sqrt{|1 - \|\mu_\perp\|_2^2|} * \frac{e_{w2B} - \mu_B}{\|(e_{w2} - \mu_\perp) - \mu_B\|} \\
e_1 &= e_{w1B}^{\text{corrected}} + \mu_\perp \\
e_2 &= e_{w2B}^{\text{corrected}} + \mu_\perp
\end{aligned}$$

3 Many-to-Many RNNs

3.1 Machine Translation Model

The following kind of many-to-many RNN model is used for machine translation:

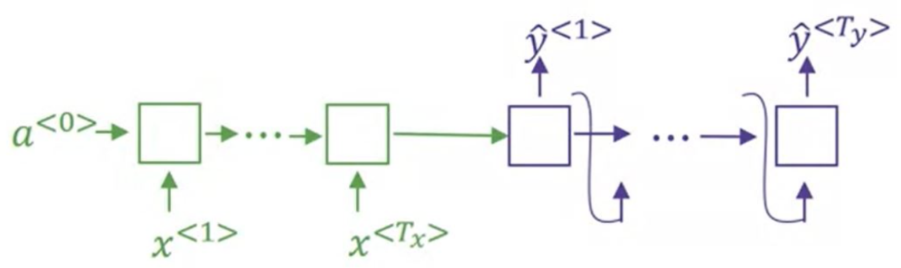


Figure 9: A many-to-many model for machine translation.

This model can be thought of as a conditional language modelling network; it is similar to a language model, but now with some input taken before the sentence generation begins. The sentence generated here is chosen based on $P(y^{<1>}, \ldots, y^{<T_y>} | x^{<1>}, \ldots, x^{<T_x>})$. As before, we can use a greedy algorithm, just moving on generating a new word at each step with probabilities decided by the words already chosen.

However, there is one drawback to using greedy algorithms; we cannot go back to make changes if we later realize that another beginning to the sentence would have been better. This means the greedy algorithm may not end up with the most probable sentence. To tackle this problem, we use beam search.

3.2 Beam Search

Beam search is similar the greedy algorithm mentioned above; the only difference is that beam search tries to keep several options open, just in case a sentence other than the one produced by the greedy approach turns out to be better.

In beam search, we have a beam width B, which is the number of options we keep open at each step of next word prediction. The model used is still the one shown in figure 21.1. For the first word prediction, we generate the probability corresponding to every word in our dictionary. Out of these m words, we then pick the B most probable words, and remember them. For the second word prediction, we feed in each of the B words one by one to the next input of the RNN, each time generating probabilities of occurring as the second word, corresponding to all possible words in the dictionary. Out of these $B \cdot m$ options, we then select the B most probable word couples (first and second word pairs), and remember these for the third step. In this way, we keep moving forward, and at the end, we pick the most probable sentence in our beam. The following diagram shows beam search with $B = 3$:

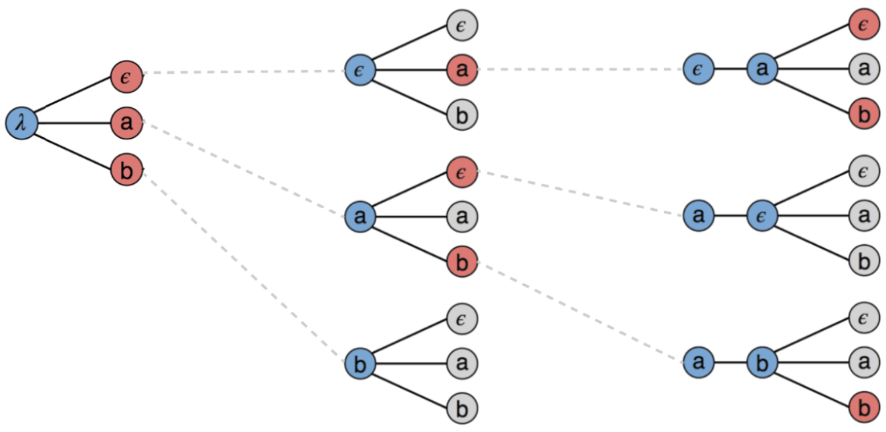


Figure 10: Beam search with a beam width of 3.

Note that the machine translation model is distinctly divided into the RNN component and the beam search component, making error analysis easier. If the algorithm outputs a translation that is not good enough, then we can attribute the mistake to one of the two components by comparing the computed probabilities of the two translations. If the algorithmic translation is not good enough, and yet it had higher probability according to the model, the RNN is at fault. If the probability of the algorithm's translation is less than that of the ideal translation, we can conclude that beam search is at fault.

3.3 Sentence Length Normalization

In these machine translation models, we are calculating:

$$P(y^{<1>},\dots,y^{<T_y>}|x^{<1>},\dots,x^{<T_x>}) = P(y^{<1>}|x)\cdot P(y^{<2>}|x,y^{<1>})\cdots P(y^{<T_y>}|x,y^{<1>},\dots,y^{<T_y-1>})$$

But this gives an undesirable probability advantage to shorter output sequences. To solve this problem, we use sentence normalization. Instead of solving:

$$\arg\max_y \prod_{t=1}^{T_y} P\left(y^{<t>}|x,y^{<1>},\dots,y^{<t-1>}\right)$$

we solve the following optimization problem:

$$\arg\max_y \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P\left(y^{<t>}|x,y^{<1>},\dots,y^{<t-1>}\right)$$

where α is some hyperparameter between 0 and 1.

3.4 Bleu Score

The bilingual evaluation understudy (Bleu) score is a way to automatically evaluate machine translations. Given a machine translation $\hat{y}$ and a number of human translations, we calculate:

$$\text{Bleu score on n-grams} = p_n = \frac{\sum_{\text{n-grams}\in\hat{y}} \text{count}_{\text{clipped}}(\text{n-gram})}{\sum_{\text{n-grams}\in\hat{y}} \text{count}(\text{n-gram})}$$

where count(n-gram) represents the number of times that n-gram appears in the machine translation $\hat{y}$, and $\text{count}_{\text{clipped}}$(n-gram) represents the maximum number of times that n-gram appears in any one of the human translations.

Considering $p_1, p_2, \ldots, p_k$, the combined Bleu score is:

$$\text{BP} \cdot \exp\left(\frac{1}{k}\sum_{n=1}^{k} p_n\right)$$

where BP is called the brevity penalty, and takes a value of 1 if the machine translation is longer than the human translations, and a value of $\exp\left(1 - \frac{\text{human translation length}}{\text{machine translation length}}\right)$ otherwise.

3.5 Attention Models

Suppose we need to translate a long paragraph from one language to another. Clearly, the many-to-many RNN discussed so far would not be very effective for this task, as it would have to memorize a lot of things all at once before it can start producing output. This is why we need attention models.

An attention model looks like this:

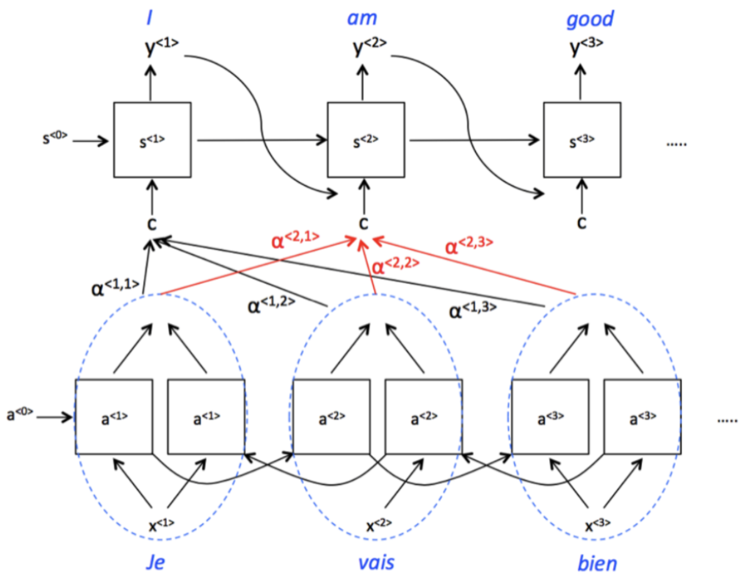


Figure 11: An attention model.

The input sentence is fed into a bidirectional RNN, to encode into feature vectors. Then, the output sentence is produced by another RNN whose inputs are the encoded feature vectors, with each RNN unit having some attention weights corresponding to every encoded feature vector. The parameter $\alpha^{<t,t'>}$ is the attention that $y^{<t>}$ gives to the feature vector produced from $a^{<t'>}$. The attention weights for a particular output add up to one, i.e., $\sum_{t'} \alpha^{<t,t'>} = 1$.

The attention weights are computed as follows:

$$\alpha^{<t,t'>} = \frac{\exp\left(e^{<t,t'>}\right)}{\sum_{t'=1}^{T_x} \exp\left(e^{<t,t'>}\right)}$$

This step ensures that the sum of all attention weights for a particular output is 1. The values of $e^{<t,t'>}$ are obtained as output from a simple neural network, having inputs $s^{<t-1>}$ and $a^{<t'>}$.

This algorithm produces very good results for translating long sentences, but it takes quadratic time to execute, which may be too heavy for some applications.