How Google Translate works (May 2018)

by

Chris Venour

1 Introduction

I was interested in seeing how Google Translate's state of the art translation system works. This “sequence to sequence” system, which is sometimes referred to as an encoder-decoder network, is described here: https://www.tensorflow.org/tutorials/seq2seq

Google has provided code which implements this model and I used that code to create and train two German to English translators: one that performs a greedy search when picking the English words of a translation and another which performs a beam width search.

Google's state of the art translation system (Google Translate) contains features such as:

- LSTM cells
- Gradient clipping
- Bi-directional forward propagation
- Attention mechanism
- Bucketing
- Teacher forcing

Each of these features is described in Section 3.

2 The architecture of the translators

Google's translation system has an encoder-decoder architecture. This architecture, shown in Figure 1, consists of two Recurrent Neural Networks (RNN): an encoder RNN (shown in blue in the figure) and a decoder RNN (shown in red).

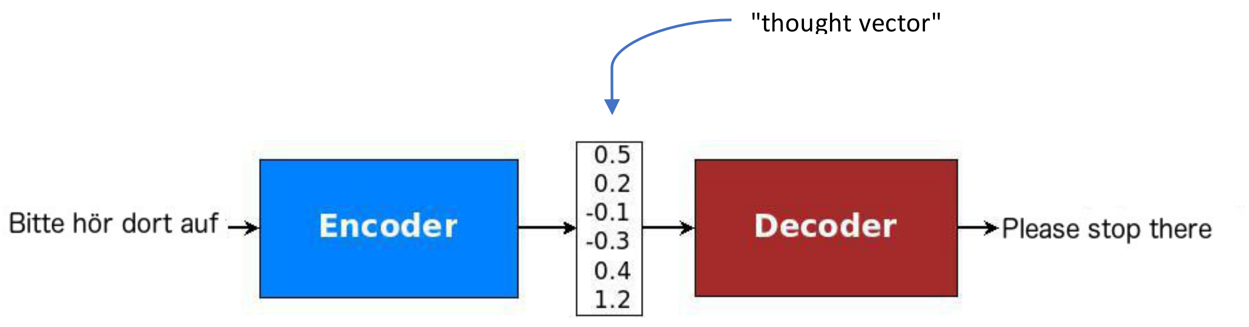

Figure 1: The translation systems use an encoder-decoder architecture. (Image from https://www.tensorflow.org/tutorials/seq2seq).

In Figure 1 the German sentence "Bitte hör dort auf" is fed into the encoder RNN and the encoder computes a vector that is a representation of the entire German sentence. This vector is sometimes referred to as a "thought vector" in the literature. The decoder network then takes the thought vector as input and uses it to produce an English translation of the German sentence.

2.1 The Encoder architecture

Let's take a closer look at how the Encoder RNN outputs the thought vector.

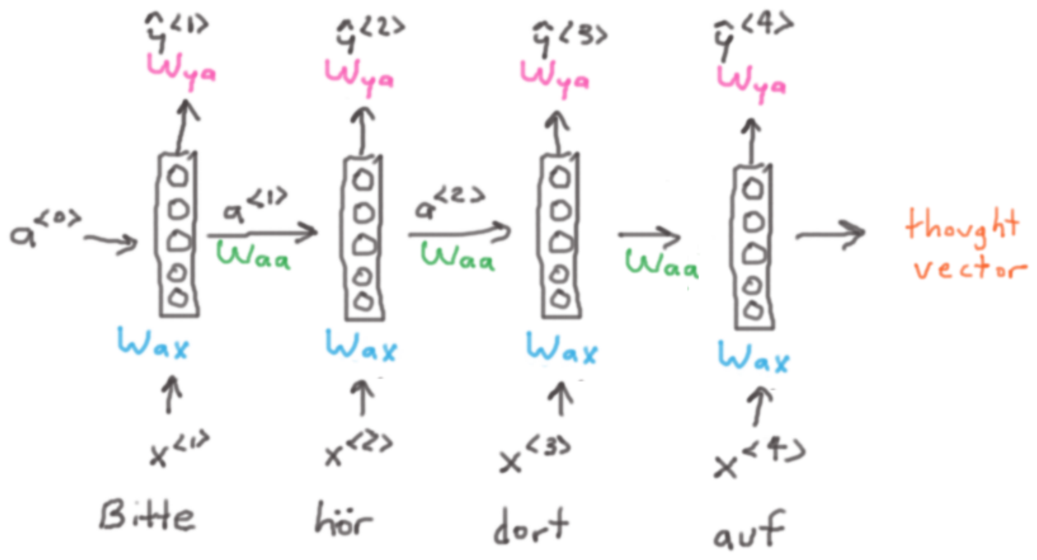

Figure 2: The RNN encoder

At each time step the next word in a sentence to be translated is passed into the encoder RNN. In this way, the RNN scans through the sentence from left to right. The RNN shown in Figure 2 has three different parameters, each of which is a matrix of values: W_{ax} , W_{aa}, and W_{ya}. The W_{ax} matrices, which appear in blue at each time step in Figure 2, are identical. Similarly, all the green W_{aa} matrix values are identical, as are the red W_{ya} matrices. The values in these three matrices are the principal things that are learned during the training of the encoder.

Let's go through an example of forward propagation in this encoder network. In this example, I'll show how forward propagation works in a vanilla RNN. The Google translator isn't a vanilla RNN and uses more complicated LSTM cells, however, for the purpose of explaining how forward propagation works I'll just look at the simpler case of forward propagation within an RNN that does not use LSTM.

Before forward propagation occurs, the first word in the sentence to be translated ("Bitte") is turned into a vector of numbers $x^{<1>}$ because networks can't handle raw text: data processed by any neural network, including RNNs, ultimately needs to be numerical.

Initially a word is assigned a vector of random values but during training, the values of this word vector and all the other word vectors of the German words in the training set are adjusted as the system learns to translate from German to English. The place in the network in which words are mapped to vectors is called the embedding layer and the weights of this embedding layer, along with the values of W_{ax} , W_{aa}, and W_{ya} (plus the bias terms in the equations below) are learned during the training of the RNN.

Step 1: The first step of forward propagation takes the embedding vector $x^{<1>}$ for the word "Bitte" and the value of the previous hidden state $a^{<0>}$to compute the value of $a^{<1>}$. Specifically, the value of $a^{<1>}$ is computed in the following way:

$$a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$$

$a^{<1>}$ is the information captured by the RNN at timestep 1 and b_a is the bias term associated with computing the $a^{<t>}$ values. There is no timestep 0 but a value for $a^{<0>}$ is required nonetheless in order to compute $a^{<1>}$ and so $a^{<0>}$ is simply a vector of zeroes. The function $g(\,)$ is an activation function such as ReLu, tanh or the now less commonly used sigmoid function. Activation functions such as this introduce non-linearity into a neural network. Without activation functions, a network would be too simple and would be unable to learn how to perform complicated tasks such as machine translation.

Step 2: A value for $\hat{y}^{<1>}$ is then computed in the following way:

$$\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y)$$

The RNN encoder shown in Figure 3 has two hidden layers and, as the figure shows, the $\hat{y}^{<t>}$ values computed in the first hidden layer are the means by which the first hidden layer communicates with the second hidden layer.

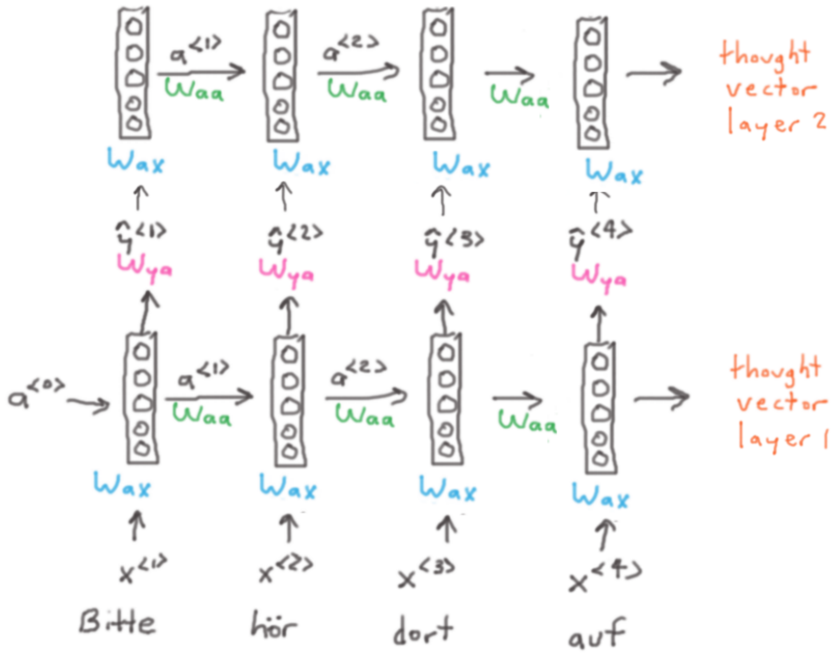

Figure 3: The Encoder RNN has two hidden layers.

The second layer doesn't compute $\hat{y}^{<t>}$ values because there isn't a third layer to pass this information to.

Steps 1-2 described above are then repeated for each word in the German sentence. The general equations for forward propagation - that is, the equations to compute $a^{<t>}$ and $\hat{y}^{<t>}$ at timestep t - are thus:

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

Once the last word vector of the German sentence is propagated through the network, the encoder RNN outputs a thought vector of the German sentence for each layer and these vectors are input to the two hidden layers of the decoder network.

2.2 The Decoder architecture

The decoder is almost identical to a conditional language model, an example of which is shown in Figure 4.

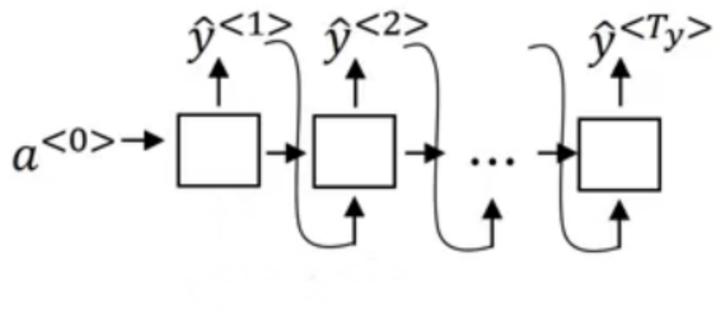

Figure 4: A language model.

Whereas a language model computes the probability of a sentence in a given language, $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \cdots, \hat{y}^{<t>})$, a decoder computes, in our case, the probability of an English translation given a German sentence: $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \cdots, \hat{y}^{<t>} \mid x^{<1>}, x^{<2>}, \cdots, x^{<n>})$, where the x terms are words in the German sentence and the $\hat{y}$ terms are words in the English sentence. And instead of receiving as input an $a^{<0>}$ vector that's initialized as a vector of all zeroes the way a language model does, the decoder network takes the encoder's thought vector as its initial input.

Like the encoder, the decoder used for this project has two hidden layers. The second hidden layer outputs $y^{<t>}$ values that are then turned into one-hot word vectors in a projection layer so that the words the RNN proposes as good translations of the German words can be mapped into actual English words.

To create the translator used in this project, the Encoder and Decoder are joined in the way shown in Figure 5. In the figure, the encoder network is shown in blue and appears on the left side of the figure and the decoder network appears in red.

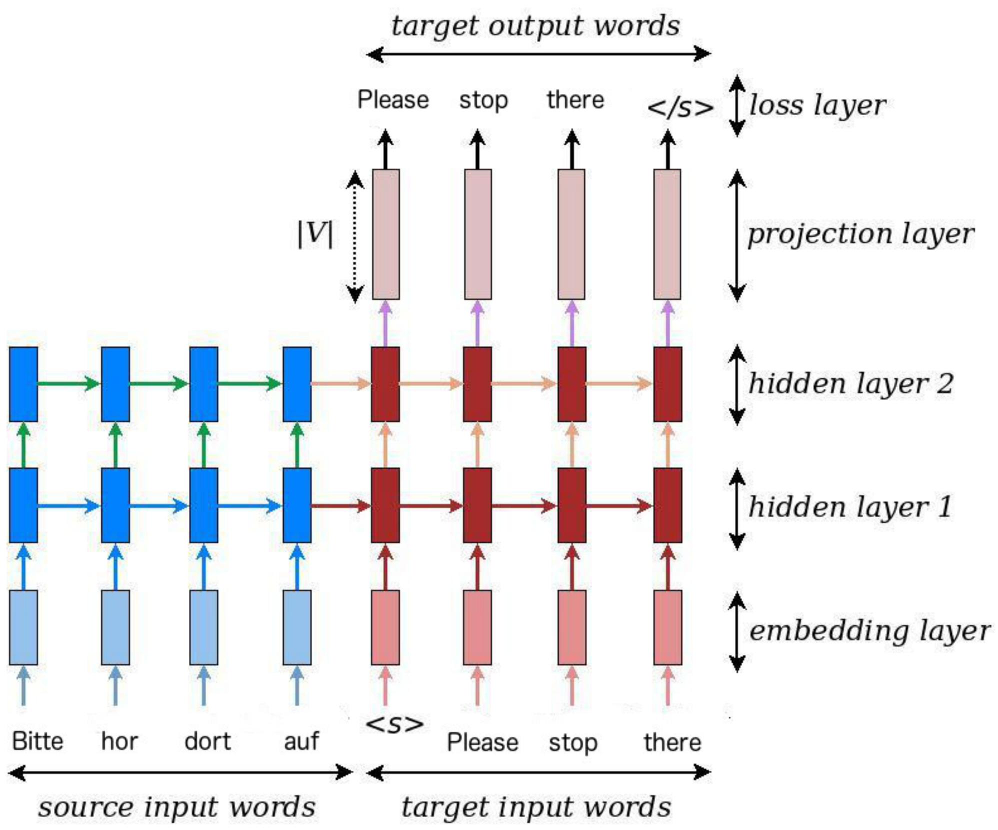

Figure 5: The translator used in this project has an embedding layer and two hidden layers. The encoder RNN appears in blue and the decoder RNN appears in red. (Image from https://www.tensorflow.org/tutorials/seq2seq).

3 Special features of the translators

Google Translate uses state of the art features which I'll briefly describe in this section.

3.1 LSTM cells

The basic or vanilla type of RNN described in Section 2.1 is unable to capture long range dependencies in a sentence. For example the following sentence has a long range dependency between the subject of the sentence and its verb which occurs at the end of the sentence:

> *The three passengers, shut up in the narrow compass of one lumbering old mail coach, have fallen asleep.*

The subject "The three passengers" and the verb phrase "have fallen asleep" have to agree in number: you couldn't say "The three passengers ... has fallen asleep" for example.

It is difficult, however, for a regular RNN to learn to remember the number of the subject of the sentence in order to generate the proper verb form that occurs later in the sentence. One of the reasons why the RNN can't remember things for long is the "vanishing gradient" problem.

During backpropagation, the network updates a weight w in the following way: $w \ = \ w - \propto \frac{dJ}{dw}$ where J is the cost function, $\propto$ is the learning rate and $\frac{dJ}{dw}$ is the derivative of the cost function with respect to the weight w at a given point. The vanishing gradient problem describes the situation when the $\frac{dJ}{dw}$ term becomes smaller - often exponentially smaller - as backpropagation progresses. This is a problem because if the $\frac{dJ}{dw}$ term is very small, the weights in the early layers of the network will be adjusted by only a miniscule amount and training will hardly progress.

When LSTM cells are used in place of just a regular RNN cell, however, the vanishing gradient problem is resolved and the RNN is able to keep track of long range dependencies in a sentence. These problems are resolved because an LSTM cell provides a memory cell along with update and forget gates. Using memory cells and update and forget gates, an RNN can learn to store information about an early part of a sentence that is required for properly handling a later part of the sentence, and to forget that information when it is no longer needed. A detailed description of the LSTM cell is outside the scope of this document, but the interested reader can find more information about this topic here: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

3.2 Gradient clipping

The $\frac{dJ}{dw}$ term in the equation above can instead **grow** exponentially during backpropagation and this problem is called the "exploding gradient" problem. This problem describes the situation when weights in the early layers of the network are drastically altered - so much so that the weights' optimal values are unlikely to be found because the weights are not being tuned finely enough during backpropagation. An operation known as "gradient clipping" addresses the exploding gradient problem. Gradient clipping simply reduces the size of the $\frac{dJ}{dw}$ vector, without altering the direction of this vector, to an allowed maximum (defined by the programmer).

3.3 Bi-directional forward propagation

Another state of the art feature of the Google translation system is bi-directional forward propagation.

Figure 6 shows a normal or uni-directional RNN layer into which words of a sentence are fed from left to right. As we have seen, the forward propagation formula for computing $\hat{y}^{<t>}$ is:

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

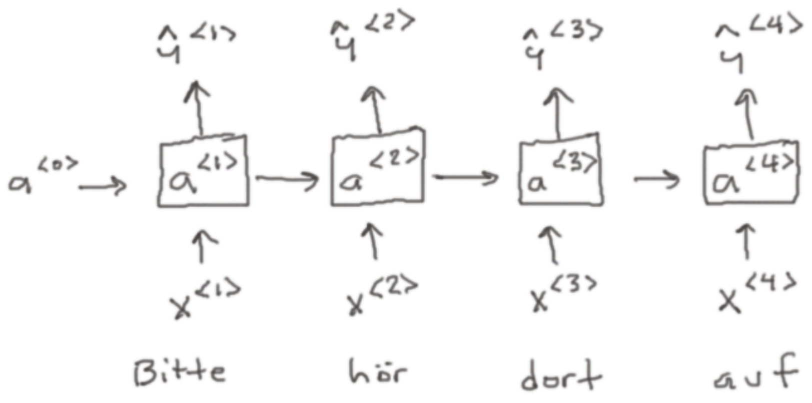

Figure 6: A uni-directional RNN.

In the past, researchers found that translation results often improved when the sentence to be translated was fed into an RNN in reverse order. A bi-directional RNN takes this empirical observation into account: it performs forward propagation on a sentence in both normal and reverse order. A bi-directional RNN adds a backward recurrent layer: the green cells $\overleftarrow{a}^{<t>}$ in Figure 7 represent a backward recurrent layer that has been added to the uni-directional RNN of Figure 6.

As Figure 7 shows, cells of the bi-directional RNN accept the words of a sentence in left to right order and the green cells accept the words of the sentence in reverse order. Note that Figure 7 shows forward propagation from the front of the sentence to the end and also forward propagation from the end of the sentence to the beginning - backpropagation is not what is being depicted here! Providing the RNN with information about words that precede a given word A and words that follow word A, helps the network choose a better translation for word A.

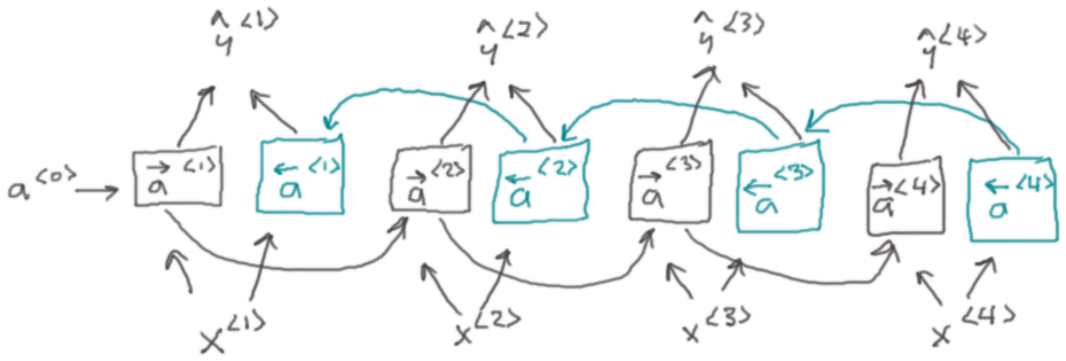

Figure 7: Forward propagation in a bi-directional RNN. Backpropagation is not being shown here, only forward prop.

In a bi-directional RNN, the forward propagation equation for computing $\hat{y}^{<t>}$ simply adds the following term, shown in green:

$$\hat{y}^{<t>} = g(W_{ya}[\vec{a}^{<t>} | \overleftarrow{a}^{<t>}] + b_y)$$

This term corresponds to the green cells depicted in Figure 7.

3.4 Attention mechanism

An attention mechanism improves the translators' accuracy and enables them to process long sentences. The attention mechanism helps in this regard by equipping the translators to learn to pay attention to just relevant words in the German sentence, rather than to all the words, when attempting to find a good translation of a given German word.

Let's say the encoder RNN is bi-directional and has a single hidden layer like the network shown in Figure 7. If we add a decoder RNN and an attention mechanism to this system, the resulting system is shown in Figure 8. Let's say the German sentence to translate is " Sogar jetzt ist die Axt an die Wurzel der Bäume gelegt." (which is translated into English as "Even now the axe is laid to the root of the tree"). In Figure 8, the system has chosen the first word of the translation to be the word "Even" and the orange lines represent the attention mechanism at work. Each word in the German sentence has an attention weight $\propto^{<1,t>}$ associated with it when deciding what the first word in the English translation should be. A different set of weights $\propto^{<2,t>}$are associated with choosing the second word of the English translation.

During training, the translator learns how much weight to give words in the German sentence when predicting an English word[1]. For example when searching for a good English translation of the first word in a German sentence, the translator learns to focus mostly on just the first few words of the German sentence (i.e. a local window of words) and to pay less attention to less relevant words that appear later in the German sentence.

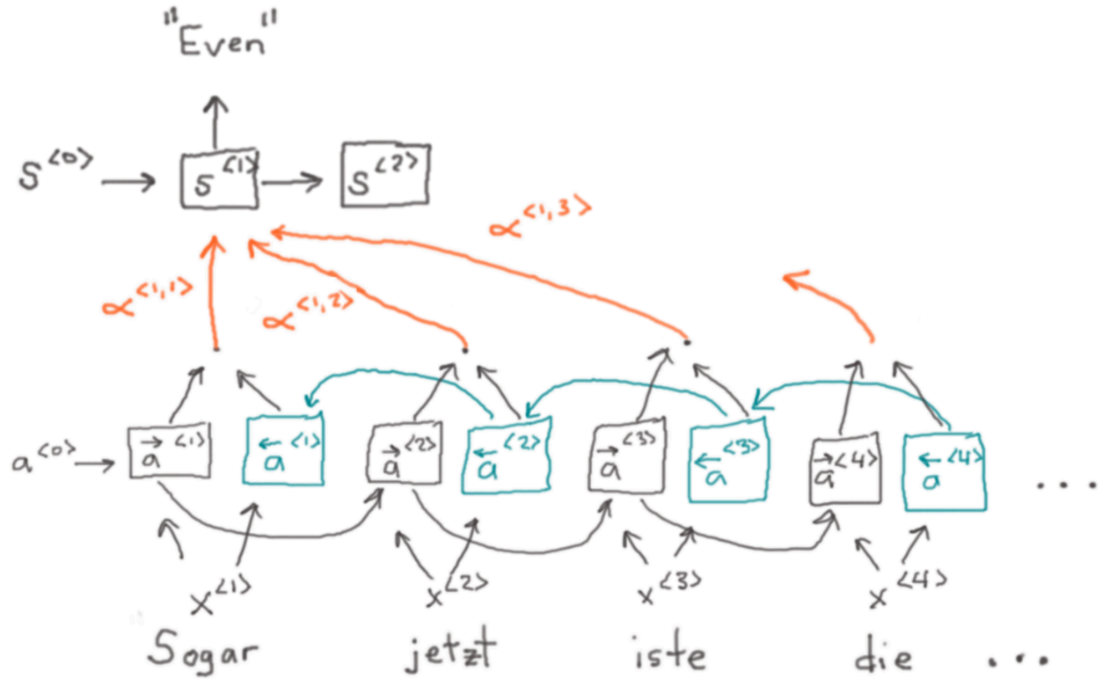

Figure 8: Bi-directional network which uses the Attention mechanism.

3. 5 Bucketing

Table 1 shows the longest sentence lengths (in terms of number of words) in the training, validation and test sets. Given these statistics, I defined the maximum sentence length allowed for a German or English sentence to be 50 tokens.

[1] The parameters in Figure 7 which determine the value of $\propto^{<1,t>}$ are $\vec{a}^{<t>}$, $\overleftarrow{a}^{<t>}$, and $s^{<0>}$. The parameters which determine the value of $\propto^{<2,t>}$ are $\vec{a}^{<t>}$, $\overleftarrow{a}^{<t>}$, and $s^{<1>}$ etc.

Table 1: Maximum sentence lengths in the datasets.

dataset	maximum sentence length (i.e. number of words
English-training	47
English-validation	48
English-test	48
German-training	47
German-validation	44
German-test	48

Instead of padding all sentences (with 0's) so that all the sentences have a length of 50 tokens, the translators used in the project do the following: when a variable length sentence is passed to a translator, the translator puts that sentence into a bucket which matches the sentence's size. For instance the translators that I implemented and describe later on in this document create the buckets shown in Table 2.

Table 2: Sentences in the training set were sorted into 5 different buckets and were padded accordingly.

bucket no.	number of tokens in a sentence
1	1-10
2	11-20
3	21-30
4	31-40
5	41-50

Sentences placed in bucket #1, for example, contain 1-10 tokens and are padded to a sentence length of 10 and sentences in bucket #2 are padded to a sentence length of 20 etc.

Each bucket is then fed into the translator as a mini-batch that is used for training the network. Bucketing reduces training time and the amount of memory space that would be required if all sentences were padded to the maximum token length of 50. Bucketing also improves the accuracy of the translators because short sentences are only padded to a sequence length of 10 rather than 50 tokens and translating a sequence of 10 tokens is easier than translating a sequence of 50 tokens. Figure 9 is a representation of how bucketing works.

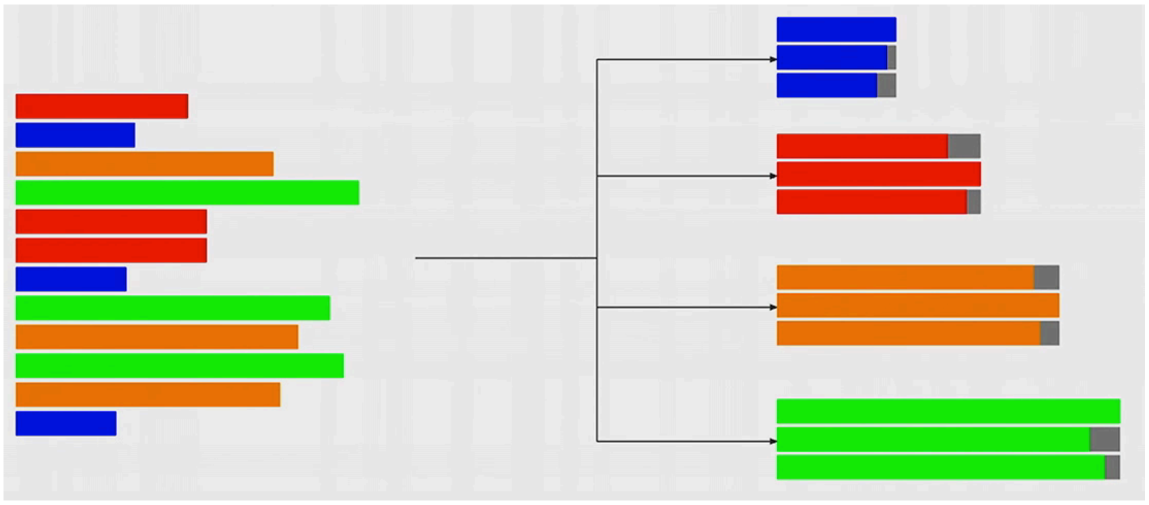

Figure 9: How bucketing works. Short (blue) sentences that appear in the data on the left are placed into the topmost bucket on the right hand side and are padded to a sentence length of 10 words. Red sentences are a bit longer than the short blue sentences and they are placed in a separate bucket - the second bucket from the top on the right hand side. These red sentences are padded to a sentence length of 20 words. etc. (Image from https://www.youtube.com/watch?v=RIR_-Xlbp7s&t=1033s)

3.6 Teacher forcing

"Teacher forcing" is the concept of using the real target output as the next input to the top layer of the decoder, instead of using the decoder's guess as the next input. As Figure 5 shows, the correct target words are run through the embedding and hidden layers of the decoder (the bottom three layers of the decoder network, shown in red in the figure) and replace the value predicted by the decoder at time t with the correct answer.

This is like a student receiving immediate feedback from a teacher that he chose the wrong word for a translation. This is useful because it prevents him from making further errors based on that error. Instead of keeping that poorly translated word around in the translation and impairing subsequent word choices/translations, the decoder network is told what the answer should have been. In other words, the correct word translation at time t rather than the decoder's possibly incorrect prediction is input into the t+1 node at the top of the red decoder network when it decides what the t+1 word translation should be.

4 Preparing data for training

Table 3 shows the sizes of the training, validation and test sets. The data I used to train the translators I implemented was already tokenized and all the letters of the words were turned into lowercase, so no further pre-processing was required.

Table 3: The number of datapoints in the training, validation and test sets.

dataset	size (i.e. number of sentence pairs)
training	179,643
validation	2433
test	1178

5 Results

The translator was trained for 12 epochs and this took 12 hours on a GPU I rented from Amazon Web Service (AWS).

Table 4 shows the human-produced translations of the first 20 German sentences in the test set. The first two columns of Table 5 show the output from the translator that performs a beam search with a beam width of 5 and the output from the greedy search translator on the 20 test set examples.

The BLEU scores achieved by the translator when it uses beam search and greedy search are discussed in Section 6.

Table 4: The human translations of the first 20 German sentences in the test set.

1. when i was in my 20s , i saw my very first psychotherapy client .
2. i was a ph.d. student in clinical psychology at berkeley .
3. she was a 26 - year - old woman named alex .
4. now alex walked into her first session wearing jeans and a big slouchy top , and she dropped onto the couch in my office and kicked off her flats and told me she was there to talk about guy problems .
5. now when i heard this , i was so relieved .
6. my classmate got an arsonist for her first client .
7. and i got a twentysomething who wanted to talk about boys .
8. this i thought i could handle .
9. but i did n't handle it .
10. with the funny stories that alex would bring to session , it was easy for me just to nod my head while we kicked the can down the road .
11. “ thirty 's the new 20 , ” alex would say , and as far as i could tell , she was right .
12. work happened later , marriage happened later , kids happened later , even death happened later .
13. twentysomethings like alex and i had nothing but time .
14. but before long , my supervisor pushed me to push alex about her love life .
15. i pushed back .
16. i said , “ sure , she 's dating down , she 's sleeping with a knucklehead , but it 's not like she 's going to marry the guy . ”
17. and then my supervisor said , “ not yet , but she might marry the next one .
18. besides , the best time to work on alex 's marriage is before she has one . ”
19. that 's what psychologists call an “ aha ! ” moment .

20. yes , people settle down later than they used to , but that did n't make alex 's 20s a developmental downtime .

Table 5: Output from the two translators on the first 20 German sentences in the test set.

Output from the translator that performs beam width search (bw = 5)	Output from the greedy search translator
1. when i was in my 20s <unk> i had my first psychotherapy patient <unk>	1. when i was in my 20s <unk> i had my first psychotherapy patient <unk>
2. i was a graduate student and studied medical psychology in berkeley <unk>	2. i was a student student <unk> studied clinical psychology <unk>
3. she was a 26 <unk> old woman called alex <unk>	3. she was a 26 <unk> year <unk> old woman named alex <unk>
4. when alex came into the first session <unk> she wore jeans and her one <unk> top <unk> she fell on the couch in my office <unk> their sandals in my office <unk> their eyes were going to talk about men 's men <unk>	4. when alex was the first session <unk> she was wearing jeans and one <unk> top <unk> she fell on the sofa in my office <unk> their sandals <unk> and told me <unk> she would be there to talk about men <unk>
5. and when i heard this <unk> i was relieved <unk>	5. and when i heard <unk> i was relieved <unk>
6. i mean <unk> got an unparalleled patient as a first patient <unk>	6. my <unk> was a veteran a veteran as a first patient <unk>
7. and i got a woman in the '20s who wanted to talk about boys <unk>	7. and i got a woman in the 20s who wanted to talk about boys <unk>
8. i 'll get rid of this <unk> i thought <unk>	8. i 'll get <unk> i thought <unk> i thought <unk>
9. but i did n't get it up <unk>	9. but i do n't get it <unk>
10. with the funny stories that alex with the session <unk> it was easy for me to zoom down with the head <unk> while we 're talking about problems <unk>	10. with the fun stories that brought alex into the meeting <unk> it was easy to me <unk> just with my head <unk> just as we are <unk>
11. “ 30 is the new 20 ” said <unk> alex and as far as i could appreciate that <unk>	11. “ 30 is the new 20 <unk> ” alex and as far as i could judge <unk> she was right <unk>
12. work later came later later later later later <unk> kids came later later <unk> even the death came later later <unk>	12. work came later later <unk> married came later <unk> children came later <unk>
13. people like alex and i had nothing than time <unk>	13. people in the 20s <unk> like alex <unk> and i had nothing <unk>
14. but soon i was troubled my mentor <unk> alex life in question <unk>	14. but soon <unk> i 'm pushing my mentor <unk> alex <unk>
15. i thought about it <unk>	15. i thought about it <unk>
16. i said <unk> “ yeah <unk> she meets men under your level <unk> she 's sleeping with men under their own level <unk> she will not marry him <unk> ”	16. i said <unk> “ yes <unk> she 's true with men under your level <unk> you sleep with one child <unk> but she will not marry him <unk> ”
17. and i said <unk> “ not yet <unk> but maybe you 'll get married the next thing <unk>	17. and there was my mentor <unk> “ not <unk> but maybe <unk> you 'll get your next <unk>
18. it 's also the best time to work before before she 's married <unk> ”	18. it 's also the best time to work <unk> when you get married <unk> ”
19. this is what psychologists call a math moment <unk>	19. this is <unk> what psychologists call a great moment <unk>
20. yeah <unk> people get down down and did n't alex and did that alex ” not to me <unk>	20. yes <unk> people get down <unk> but this did n't say <unk> alex <unk> did n't go <unk>

6 Questions

1. **Why does the greedy decoding algorithm make search errors even though it is possible to generate unbiased samples from the translation model?**

   Greedy search generates the most likely first word according to the conditional language model $P(\hat{y}^{<1>} \mid x)$ where x is the German sentence to be translated and $\hat{y}^{<1>}$is the first English word chosen for the translation. Greedy search then picks the 2nd word that seems most likely given x and $\hat{y}^{<1>}$ i.e. $P(\hat{y}^{<2>} \mid x, \hat{y}^{<1>})$.

   This is not a good strategy however and does not guarantee an optimal or even a very good translation. We want to maximize $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \cdots, \hat{y}^{<t>} \mid x^{<1>}, x^{<2>}, \cdots, x^{<n>})$ and this means sometimes rejecting short term rewards for longer term rewards - i.e. rejecting a word that yields the highest probability at time t because in the long run this choice does not lead to a higher overall probability score which a less probable word chosen at time t would have led to.

   The computer scientist Andrew Ng gives the following illustration[2]. Let's say a French to English translation system is given the following sentence to translate: "Jane visite l'Afrique en Septembre". The greedy search is more likely to pick the translation "Jane is **going to be visiting** Africa in September" rather than the less wordy and superiour translation "Jane is **visiting** Africa in September". This is because, when choosing the third word, the greedy search is more likely to select the word 'going' rather than the word 'visiting' because the former word is more common than the latter and would likely yield a higher probability score. This choice, which in the short term seems best, leads to an inferiour translation in the long run.

   Beam search, however, will nearly always produce better results than greedy search because beam search entertains multiple paths/translations down a tree of possibilities. The testing results, which are described in the answer to question #2, support this assertion: when the translator used beam search with a beam width of 5 (i.e. the 5 best greedy choices/paths were considered in the tree as the translation progressed), it achieved significantly higher BLEU scores than when greedy search was used.

[2] The illustration by Andrew Ng is from https://www.coursera.org/learn/nlp-sequence-models/lecture/v2pRn/picking-the-most-likely-sentence

2. **Sample 20 translations of the test set. What is the min/max/mean/median/stddev of the BLEU score? How do these compare to the BLEU score obtained by the greedy decoder?**

   BLEU scoring was performed on the first 20 translations of the test set using the multi-bleu.pl program. However, this program appears to have failed to give proper BLEU statistics for 13/20 of the sentences output by the translators. These sentences received a BLEU score of 0, not because all of them were terrible translations but because the perl program appears to have failed in some way. For example in all the cases when a BLEU score of 0 was output, the following warning message was output by the multi-bleu program: "Use of uninitialized value in division".

   Instead of spending hours investigating why this perl program may or may not have been failing, I performed an additional BLEU score test using the Python module NLTK. All the score results - from the multi-bleu perl program and the NLTK bleu method appear in Tables 6 - 9.

   Both BLEU scoring methods show that beam-width search, with a beam-width of 5, produced significantly better results than greedy searching.

Table 6: Greedy search multi-BLEU scores

multi-BLEU scores (greedy search)	statistics
20.33, 0, 21.23, 7.5, 0, 0, 24.09, 0,0,0,18.26, 0, 11.21, 0,0,0,0,13.10,0,0	min: 0.0 max: 24.09 mean: 5.786 median: 0.0 std-dev: 8.54911012913

Table 7: Greedy search NLTK BLEU scores

NLTK BLEU scores (greedy search)	statistics
68.52, 56.65, 65.90, 56.81, 62.36, 27.16, 68.16, 28.10, 46.85, 53.04, 62.59, 34.68, 49.089, 22.03, 35.93, 35.42, 35.00, 44.58, 59.97, 31.87	min: 22.03 max: 68.52 mean: 47.24 median: 47.97 std-dev: 14.67

Table 8: Beam width = 5 search multi-BLEU scores

multi-BLEU scores (beam width=5)	statistics
41.61, 0, 29.38, 14.04, 32.47, 0, 49.20, 0, 36.56, 32.86, 27.64, 0, 53.73, 0, 0, 10.28, 0, 24.67, 0, 0	min: 0.0 max: 53.73 mean: 17.62 median: 12.16 std-dev: 18.49

Table 9: Beam width = 5 search NLTK BLEU scores

NLTK BLEU scores (beam-width=5)	statistics
68.52, 55.96, 61.36, 51.42, 66.41, 29.37, 67.10, 35.52, 54.13, 62.63, 51.34, 36.99, 62.39, 28.19, 35.93, 38.30, 45.50, 58.84, 65.84, 36.01	min: 28.1929527089 max: 68.5212231981 mean: 50.5869710146 median: 52.7714306983 std-dev: 13.3352804297

3. Do you expect word embeddings for English and German words that mean the same to be "close" in terms of cosine similarity? Why or why not?

The translators I implemented contain two RNNs each: an encoder RNN and a decoder RNN. These two RNNs are linked together to accomplish an ultimate final goal: taking a German sentence and outputting an English translation. However, in pursuing this final goal, the two RNNs have more immediate goals that are distinct. The Encoder RNN consumes the German sentence and outputs thought vectors which are passed to the Decoder RNN. The Decoder RNN processes the correct English translation of the sentence (because it performs "teacher forcing") and uses that processing, along with the thought vectors, to predict the next English words.

Each of these RNNs has its own distinct embedding layer and each is pursuing a distinct immediate goal. The Encoder RNN aims to produce the most useful thought vectors (i.e. representations of the German sentence as a whole) and the Decoder RNN aims to predict the correct English word prediction using those thought vectors. Because the Encoder and Decoder have different immediate goals, it is unlikely that the embeddings for the German words and the embeddings for their English translations will have a high cosine similarity.