2016-10-27

Fundamentals of Machine Learning for Neural Machine Translation

John Kelleher
Technological University Dublin, john.d.kelleher@tudublin.ie

Fundamentals of Machine Learning for Neural Machine Translation

Dr. John D. Kelleher
ADAPT Centre for Digital Content Technology
Dublin Institute of Technology, Ireland

1 Introduction

This paper[1] presents a short introduction to neural networks and how they are used for machine translation and concludes with some discussion on the current research challenges being addressed by neural machine translation (NMT) research. The primary goal of this paper is to give a no-tears introduction to NMT to readers that do not have a computer science or mathematical background. The secondary goal is to provide the reader with a deep enough understanding of NMT that they can appreciate the strengths of weaknesses of the technology. The paper starts with a brief introduction to standard FEED-FORWARD NEURAL NETWORKS (what they are, how they work, and how they are trained), this is followed by an introduction to WORD-EMBEDDINGS (vector representations of words) and then we introduce RECURRENT NEURAL NETWORKS. Once these fundamentals have been introduced we then focus in on the components of a standard neural-machine translation architecture, namely: ENCODER NETWORKS, DECODER LANGUAGE MODELS, and the ENCODER-DECODER architecture.

2 Basic Building Blocks: Neurons

Neural networks are from a field of research called machine learning. Machine learning is fundamentally about learning functions from data. So the first thing we need to know is what a function is:

A FUNCTION maps a set of input (numbers) to an output (number)

[1] In 2016 I was invited by the *European Commission Directorate-General for Translation* to present an tutorial on neural-machine translation at the *Translating Europe Forum 2016: Focusing on Translation Technologies* held in Brussels on the 27^{th} and 28^{th} October 2016. This paper is based on that tutorial. A video of the tutorial is available at: `https://webcast.ec.europa.eu/translating-europe-forum-2016-jenk-1`, the tutorial starts 2 hours into the video (timestamp $2:00:15$) and runs for just over 15 minutes.

For example, the function SUM will map the inputs 2, 5 and 4 to the number 11:

$$sum(2, 5, 4) \rightarrow 11$$

The fundamental function we use when we are building a neural network is call a WEIGHTED SUM function. This function takes in a sequences of numbers as input and multiples each number by a weight and then sums the results of these multiplications together.

$$\begin{aligned}\text{WEIGHTEDSUM}(&\underbrace{[n_1, n_2, \ldots, n_m]}_{\text{Input Numbers}}, \underbrace{[w_1, w_2, \ldots, w_m]}_{Weights}) \\ &= (n_1 \times w_1) + (n_2 \times w_2) + \cdots + (n_m \times w_m)\end{aligned}$$

For example, if we had a WEIGHTED SUM function that had the predefined weights -3 and 1 and we passed it the numbers 3 and 9 as input then the WEIGHTED SUM function would output the value 0:

$$\begin{aligned}&\text{WEIGHTEDSUM}([3, 9], [-3, 1]) \\ &\qquad = (3 \times -3) + (9 \times 1) \\ &\qquad = -9 + 9 \\ &\qquad = 0\end{aligned}$$

When we are learning a WEIGHTED SUM function from data we are actually learning the weights that we apply to the inputs prior to the sum.

When we are making a neural network we generally take the output of the WEIGHTED SUM function an pass it through another function which we call an ACTIVATION function. An ACTIVATION function takes the output of our WEIGHTED SUM function and applies another mapping to it. For technical reasons that I won't go into in this paper we generally want our ACTIVATION function to provide a non-linear mapping. We could use any non-linear function as our ACTIVATION function. For example, a frequently used ACTIVATION function is the LOGISTIC function (see Figure 1). The LOGISTIC function maps any number between $+\infty$ and $-\infty$ to the range 0 to 1. Figure 1 below illustrates the mapping the LOGISTIC function would apply to the input values in the range -10 to $+10$. Notice that the LOGISTIC function maps the input value 0 to the output value of 0.5.

So, if we use a LOGISTIC function as our non-linear mapping then our ACTIVATION function is defined as the output of a WEIGHTED SUM function passed through the LOGISTIC function:

$$\begin{aligned}&\text{ACTIVATION} = \\ &\text{LOGISTIC}(\text{WEIGHTEDSUM}((\underbrace{[n_1, n_2, \ldots, n_m]}_{\text{Input Numbers}}, \underbrace{[w_1, w_2, \ldots, w_m]}_{Weights}))\end{aligned}$$

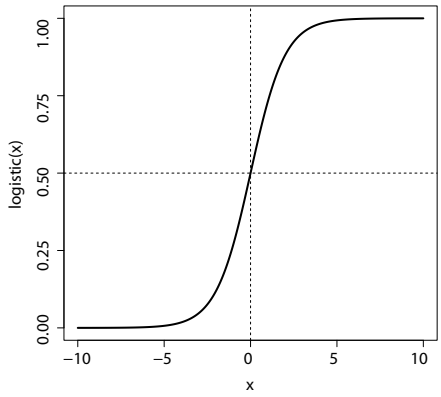

Figure 1: A Graph of the Logistic Function Mapping from input x to output $logistic(x)$

The following example shows how we can take the output of a WEIGHTED SUM and pass it through a logistic function:

$$\begin{aligned}
&\text{LOGISTIC}(\text{WEIGHTEDSUM}([3,9],[-3,1])) \\
&\qquad = \text{LOGISTIC}((3 \times -3) + (9 \times 1)) \\
&\qquad = \text{LOGISTIC}(-9 + 9) \\
&\qquad = \text{LOGISTIC}(0) \\
&\qquad = 0.5
\end{aligned}$$

The simple list of operations that we have just described defines the fundamental building block of a neural network: the NEURON.

$$\begin{aligned}
&\text{NEURON} = \\
&\text{ACTIVATION}(\text{WEIGHTEDSUM}((\underbrace{[n_1, n_2, \ldots, n_m]}_{\text{Input Numbers}}, \underbrace{[w_1, w_2, \ldots, w_m]}_{Weights}))
\end{aligned}$$

3 What is a Neural Network?

We can create a neural network by simply connecting together lots of neurons. If we use a circle to represent a neuron, squares to represent locations in memory where we store data without transforming it, and arrows to represent the flow of information between neurons we can then draw a feed forward neural network as shown in Figure 2. The interesting thing to note in this figure is that the output from one neuron is often the input to another neuron. Remember, the arrows indicate the flow of information between neurons, if there is an arrow from one neuron to another neuron then the output of the first neuron is passed as input

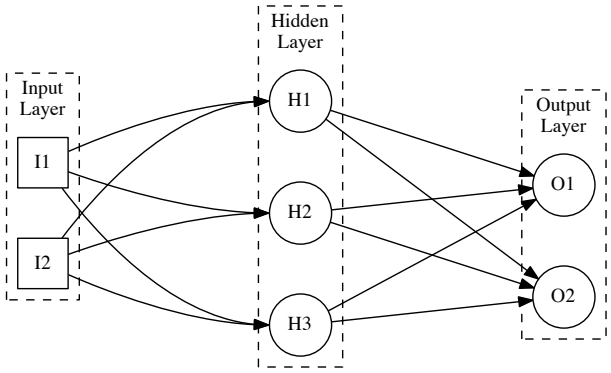

Figure 2: A feed-forward neural network

to the second neuron. Notice, also, that in our feed forward network there are some cells that are inbetween the input and output cells. These cells are hidden from view and are called the HIDDEN UNITS. We will discuss these cells in more detail later when we are explaining RECURRENT NEURAL NETWORKS.

It is probably worth emphasising that even when we create a neural network each neuron in the network (circle) is still doing a very simply set of operations:

1. multiply each input by a weight,
2. add together the results of the multiplications
3. then push this result through our non-linear ACTIVATION function

4 Where do the weights come from?

The fundamental function in neural network is the WEIGHTED SUM function. So it is important to understand how the weights used in the WEIGHTED SUM function are represented in a neural network and where these weights come from. In a neural network the weight applied to each input in a neuron is determined by the edge the input comes into the neuron on. So each edge in the network has a weight associated with it, see Figure 3.

When we are training a neural network from data we are searching for the best set of weights for the network. We train a neural network by iteratively updating the weights in the network. We start by randomly assigning weights to each edge. We then show the network examples of inputs and expected outputs. Each time we show the network an example we compare the output of the network with the expected output. This comparison gives us a measure of

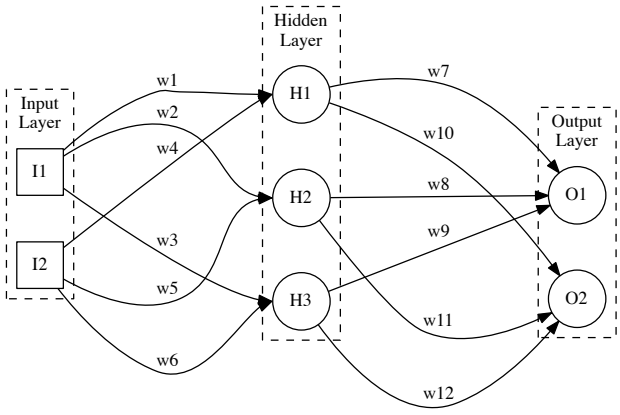

Figure 3: Illustration of a feed-forward neural network showing the weights associated with the edges in the network

the error of the network on that example. Using the measure of error and an algorithm called Backpropogation we then update the weights in the network so that the next time the network is shown the input for this example the output of the network will be closer to the expected ouput (i.e., the networks error will be reduced). We keep showing the network examples and updating the weights until the network is working the way we want it to.

5 Word Embeddings

One problem with using neural networks for language processing is that we need to convert language into a numeric format. There are lots of different ways we could do this but the standard way of doing this at the moment is to use a Word Embedding representation. The basic idea is that each word is represented by a vector of numbers that embeds (or positions) the word in a multi-dimensional space. For example, assuming we are using a 4 dimensional space for our embeddings[2] then we might define the following word embeddings

[2]Note that normally we would use a much higher dimensional spaces for embeddeings; for example, 50, 100 or 200 dimensions.

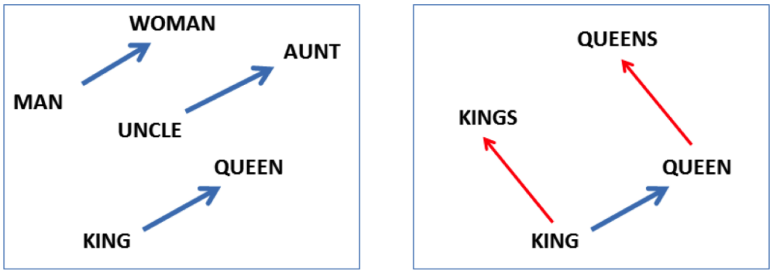

Figure 4: Illustration showing how vector offsets between word-embedding vectors can encode semantic relationships between words. These figures are taken from [6].

for the words *king*, *man*, *woman*, and *queen*:

$$\begin{aligned} king &=< 55, -10, 176, 27 > \\ man &=< 10, 79, 150, 83 > \\ woman &=< 15, 74, 159, 106 > \\ queen &=< 60, -15, 185, 50 > \end{aligned}$$

Looking at these embeddings you might be wondering what is the meaning of these numbers. The first thing to be aware of is that the absolute values of these numbers don't mean anything. What is important here is the relative position of the words relative to each other. When we are using a word embedding different directions in the multi-dimensional space encode different semantic relationships between words. Figure 4 illustrate how we can use different directions to encode semantics relationships between words: the left panel shows vector offsets for three word pairs illustrating the gender relation and the right panel shows a different semantic relationship, in this case the singular/plural relation for two words pairs.

In high-dimensional space, multiple (semantic) relations can be embedded for a single word. We do not define these word embeddings manually. Instead, we use specialized neural networks to learn these word vectors from corpora. I won't explain these neural networks in this paper, but see [2] and [5] for more information on this topic. However, once we have learnt our word embeddings we can use these embeddings to represent words as vectors of numbers and we can now train neural networks to process language. In the rest of this paper when we are referring to a word you can consider that the word is presented to the neural network as a vector of numbers.

6 Recurrent Neural Networks

We can make different types of neural networks by changing the topology of the network. A particular type of neural network that is useful for processing SEQUENTIAL data (such as, language) is a RECURRENT NEURAL NETWORK (RNN). Using an RNN we process our sequential data one input at a time. In an RNN the outputs of some of the neurons for one input are feed back into the network as part the next input. To create a recurrent neural network we augment our neural network with a memory buffer, as shown in Figure 5. Note that we generally create an RNN model by extending a feed-forward neural network that has just one hidden layer.

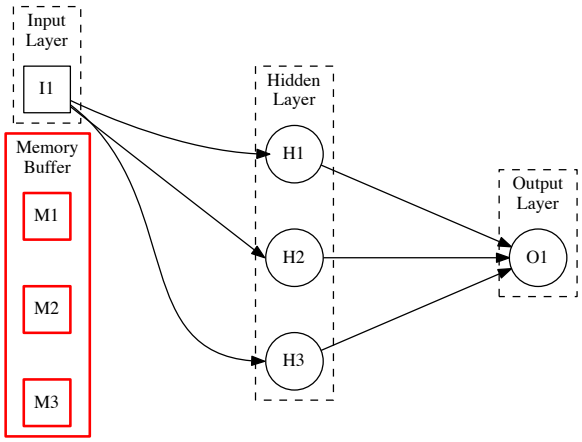

Figure 5: Adding a memory buffer to a feed-forward neural network with one hidden layer

Each time we present an input to the network the output from the hidden units for that input are stored in the memory buffer overwriting whatever was in the memory, see Figure 6. At the next time step the data stored in the buffer is merged with the input for that time step, see Figure 7. So as we move through the sequence we have a constant cycle of storing the state of the network and using that state at the next time step, see Figure 8.

In order to keep the rest of the graphics in the paper legible I won't draw all the separate neurons and connections in the remaining network illustrations. Instead I will just represent each layer of neurons as a rounded box and show the flow of information between layers using arrows. Also so as to save space I will refer to the input layer as x_t, the hidden layer as h_t, the output layer as y_t, and the memory layer as h_{t-1}. The image on the left of Figure 9 illustrates the use of rounded boxes to represent layers of neurons and the flow of information

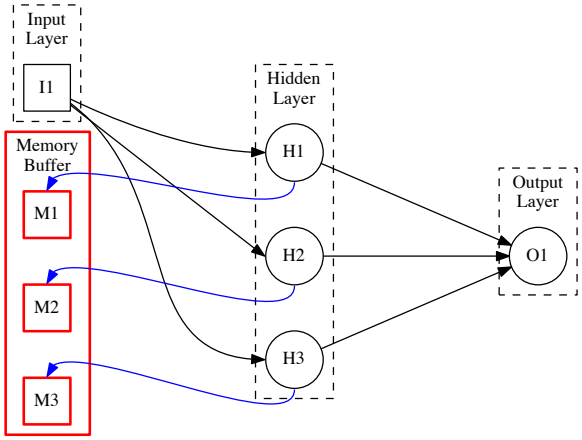

Figure 6: Writing the activation of the hidden layer to the memory buffer

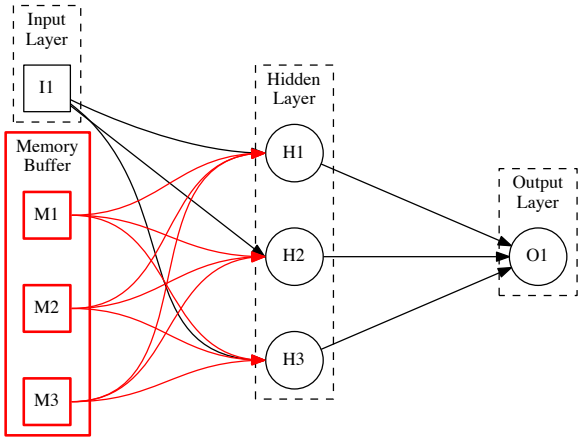

Figure 7: Merging the memory buffer with the next input.

through an RNN using this representation and the image on the right of Figure 9 shows the same network using the shorter naming convention.

Using this shorthand notation we can illustrate the flow of information through an RNN as it processes a sequence of inputs, see Figure 10. An inter-

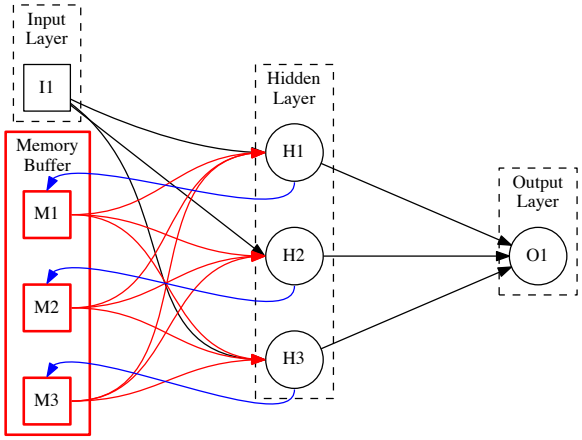

Figure 8: The cycle of writing to memory and merging with the next input as the network processes a sequence.

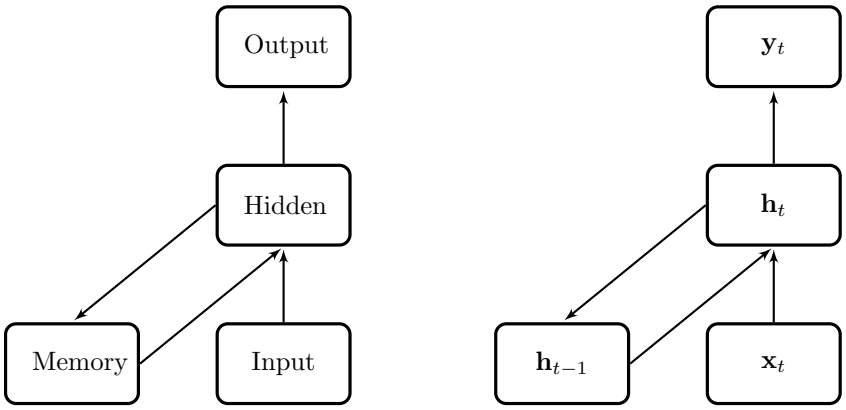

Figure 9: Recurrent Neural Network

esting thing to note here is that there is a path connecting each h (the hidden layer for each input) to all the previous hs. So the hidden layer in an RNN at each point in time is dependent on its past. In other words, the network has a memory so that when the network is making a decision at time step t it can remember what it has seen previously. This allows the model to model data that depends on previous data - such as sequences. And, this is the reason why an RNN is useful for language processing: having a memory of the previous words that have been seen in a sequence (sentence) is useful for processing language.

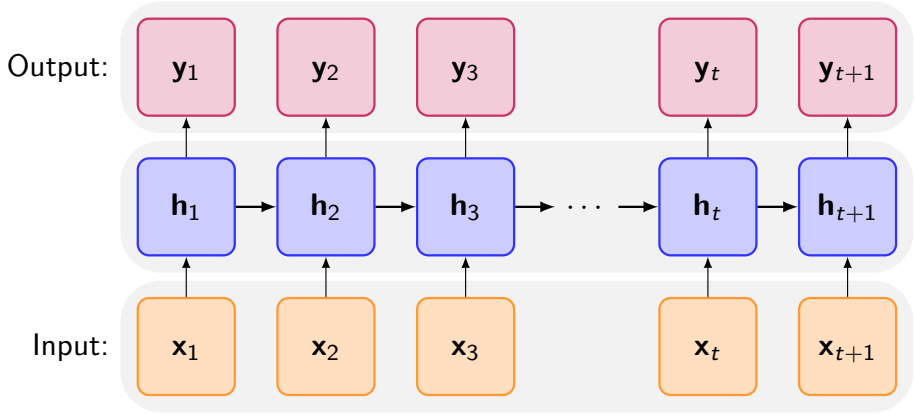

Figure 10: An RNN Unrolled Through Time

We can use RNNs to process language in a number of different ways and in the following sections I am going to introduce two ways of using them: RNN Encoders and RNN Language Models.

7 Encoders

Similar to the way we can learn vector representations of words, we can use an RNN to learn vector representations of sequences of words. To do this we first learn a set of word embeddings (vector representations). These word embeddings then remained fixed for the rest of the encoding. Then to generate an encoding for a sequence of words we input each word in the sequence in turn into an RNN network (using the word embedding representations of the words as our input representation to the network) and then we use the state of the hidden layer of the RNN after we have input the last word in the sequence as a representation for the sequence. Using an RNN in this was is known as ENCODING. Figure 11 illustrates using an RNN encoder to generate an encoding for a sequence of words.

8 Decoders (Language Models)

A language model is a computational model that can take a sequence of words as input and return a probability distribution over a vocabulary that defines the probability of each of the words in the vocabulary being the next word in the sequence. We can train and RNN language model by training the model to predict the next word in a sequence. Figure 12 illustrates how information flows through an RNN language model as it processes a sequence of words and attempts to predict the next word in the sequence after each input. Note in this

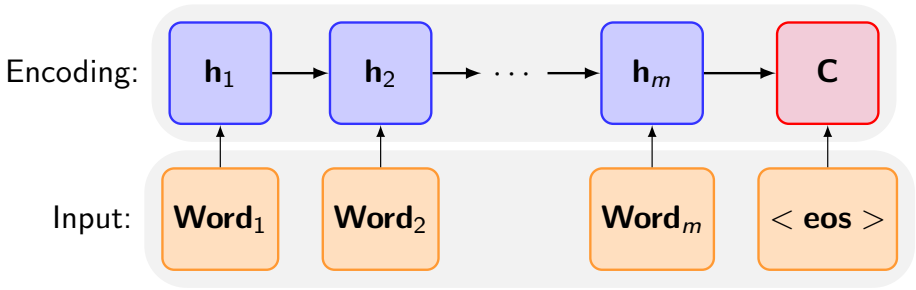

Figure 11: Using an RNN to Generate an Encoding of a Word Sequence: the symbol $< eos >$ is a special symbol used to mark the end of a sequence, the box labelled **C** holds the embedding for the word sequence.

image that the * marks indicate the next word as predicted by the system. All going well $*Word_2 = Word_2$ but if the system makes a mistake this will not be the case.

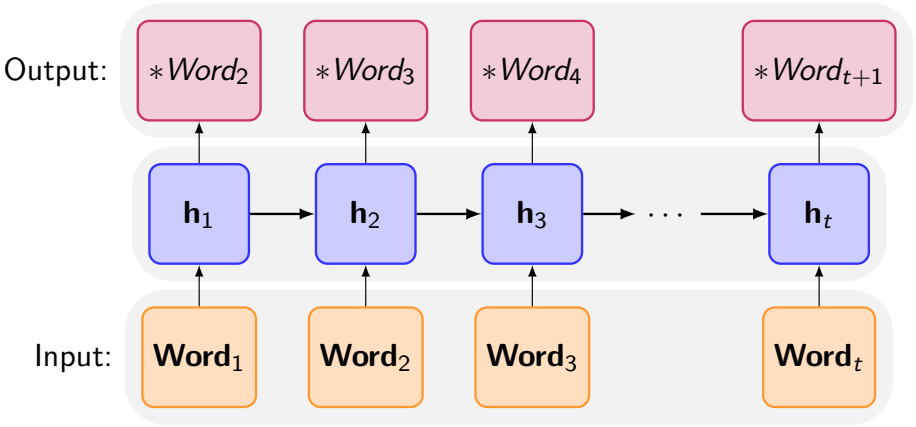

Figure 12: RNN Language Model Unrolled Through Time

When we have trained a language model we can get it to hallucinate language by giving it an initial word and then inputing the word that the language model predicts as the most likely next word as the next word into the model etc. Figure 13 shows how we can use an RNN language model to generate (hallucinate) text by feeding the words the language model predicts back into the model. If the language model is initialised with the output of an ENCODER (i.e., if the language model is initialised with a vector representation of a sequence of words) then we call the RNN language model a DECODER.

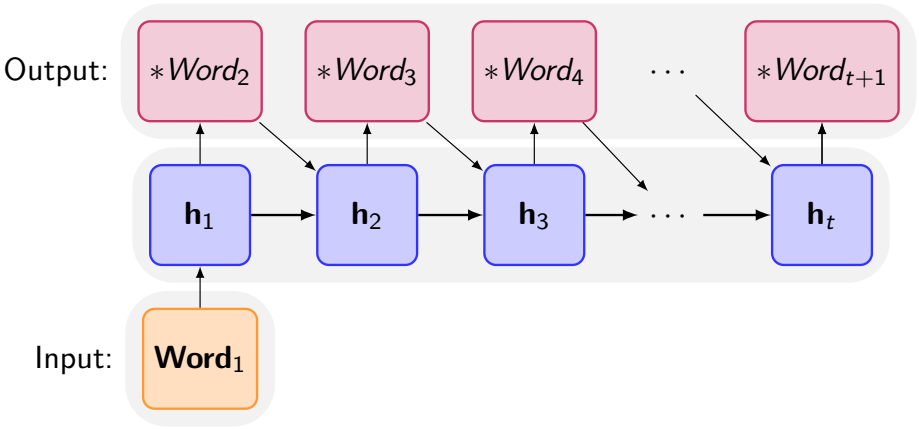

Figure 13: Using an RNN Language Model to Generate (Hallucinate) a Word Sequence

9 Neural Machine Translation

We now have all the pieces we need to do machine translation (MT) with neural networks. To do MT with neural networks we connect an RNN encoder with a RNN decoder language model. The RNN encoder processes the sentence in the source language word by word and generates a representation of the input sentence. The RNN decoder (or language model) takes the output from the encoder as input and generates the translation of the input sentence word by word. Figure 14 illustrates how we can connect the encoder and decoder models together. This model architecture for machine translation is known as an ENCODER-DECODER model, see [9] for more details.

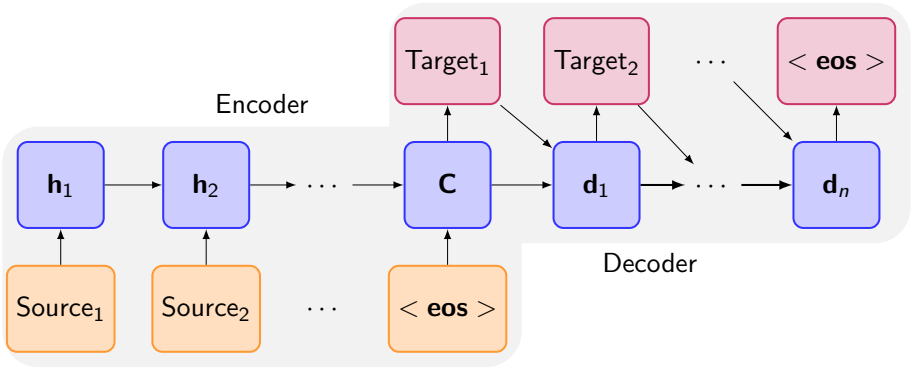

Figure 14: Sequence to Sequence Translation using an Encoder-Decoder Architecture

Figure 15 illustrates how an ENCODER-DECODER MT system would generate an English translation of a French sentence. The ENCODER processes the French sentence word by word including the $< eos >$ (END OF SEQUENCE) symbol. Notice that in this example we pass the source sentence in backwards, doing this has been found give better translation results. We then pass the encoded representation of the source sentence to the DECODER (language model) and we let this language model generate the translation word by word until it outputs an $< eos >$ (END OF SEQUENCE) symbol.

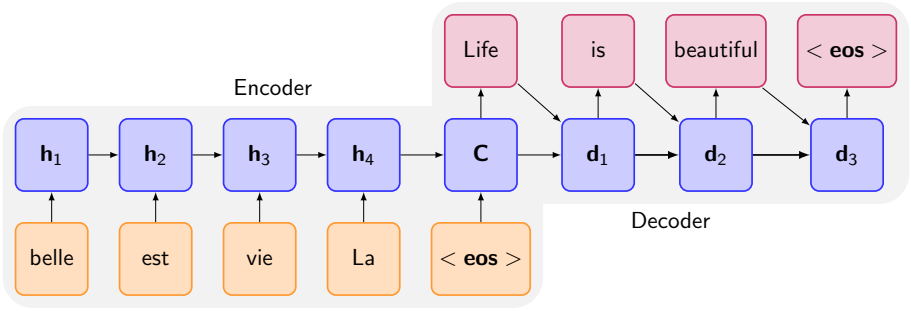

Figure 15: Example Translation using an Encoder-Decoder Architecture

10 Conclusions

An advantage of the ENCODER-DECODER architecture is that the system processes the entire input before it starts translating. This means that the decoder can use what it has already generated and the entire source sentence when generating the next word in the translation. There is ongoing research on what is the best way to present the source sentence to the encoder. There is also ongoing research on giving the decoder the ability to attend to different parts of the input during translation. This is done by extending the encoder-decoder architecture with an attention module that acts as an alignment mechanism between the words in the input and output sentences, see [1] and [4] for more on this. Finally, it is worth noting that data driven computational models tend to learn the average (or most common) behaviour found in the data. In fact, the real challenge in machine learning is to create models that model the real variation in the data (while excluding the noise in the data) and hence make predictions away from the central tendency of the data when it is appropriate [3]. The implication of this for machine translation systems (be they statistical models or neural machine translation models) is that these models tend to struggle with non-compositional, figurative, or idiomatic language (see for example [7, 8]). So one of the challenges facing neural machine translation researchers is to develop translation systems that handle these forms of language.

11 Acknowledgements:

This work was partly supported by the ADAPT centre. The ADAPT Centre is funded under the SFI Research Centres Programme (Grant 13/RC/2106) and is co-funded under the European Regional Development Fund.

References

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

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[3] John D. Kelleher, Brian Mac Namee, and Aoife D'Arcy. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT Press, 2015.

[4] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[6] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *The 2013 Conference of the North Americal Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 746–751, 2013.

[7] Giancarlo D. Salton, Robert J. Ross, and John D. Kelleher. An Empirical Study of the Impact of Idioms on Phrase Based Statistical Machine Translation of English to Brazilian-Portuguese. In *Third Workshop on Hybrid Approaches to Translation (HyTra) at 14th Conference of the European Chapter of the Association for Computational Linguistics*, 2014.

[8] Giancarlo D. Salton, Robert J. Ross, and John D. Kelleher. Evaluation of a substitution method for idiom transformation in statistical machine translation. In *The 10th Workshop on Multiword Expressions (MWE 2014) at 14th Conference of the European Chapter of the Association for Computational Linguistics*. 2014.

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. 2014.