Language Modeling and Machine Translation with Recurrent Neural Networks

Aditi Nair and Akash Shah

May 10, 2016

1 Introduction

Though linguists and scientists have developed many approaches to language translation over time, high-quality robust machine translation is especially valued today for its scalability. Recently, the application of neural networks to statistical machine translation has gained traction as a machine translation methodology which, unlike other statistical approaches to machine translation, does not require extensive manual feature engineering. While scientists have also recently explored the application of neural networks to "sub-problems" of machine translation systems – for example, at the feature engineering stage – we will focus on the direct application of neural networks for the purpose of end-to-end machine translation from the source language to the target language. In particular, we will present the example of a simple Encoder-Decoder model, which uses recurrent neural networks for both encoding and decoding. We will preface our presentation of the Encoder-Decoder model with a brief review of statistical machine translation, an introduction to recurrent neural networks, and a discussion of language modeling with recurrent neural networks.

2 Framework for Statistical Machine Translation

When translating from one language to another, there are often multiple acceptable translations for a sentence or phrase in the source language to a sentence or phrase in the target language. For this reason, we prefer to develop models of translation that will express the conditional probability $p(Y|X)$ of a translation Y given the source sentence X. Generally, given a source sentence X and its correct translation Y, we would like to find a model which assigns a high value to the conditional probability $p(Y|X)$, or, equivalently, gives a high value for the log-likelihood $\log p(Y|X)$. Assuming that we have a training set $\mathrm{D} = (X_1, Y_1), \ldots, (X_n, Y_n)$ of source sentences X_i and their translations Y_i, we define the score of a translation model on the training set as:

$$L(\theta, D) = \sum_i \log(p(Y_i|X_i))$$

where θ represents a parameterization of the model. By maximizing this score, we can find the model that provides the most likely or best translations on the

training set; this is called the maximum likelihood estimator (MLE).

In traditional approaches to statistical machine translation, researchers often approximate the log likelihoods $\log(p(Y_i|X_i))$ with a linear combination of featurization functions f, which take the source sentence X_i or components of it as input. This approach reduces the modeling task to choosing the optimal functions f and their coefficients which, as mentioned above, usually requires careful selection and tuning.

By contrast, the recurrent neural network model, which we will describe below, can be trained to algorithmically determine the optimal featurizations f and parameters which approximate the MLE.

3 A Brief Introduction to Recurrent Neural Networks

A reader familiar with neural networks may recognize the diagram below as a simple recurrent neural network with a single hidden layer:

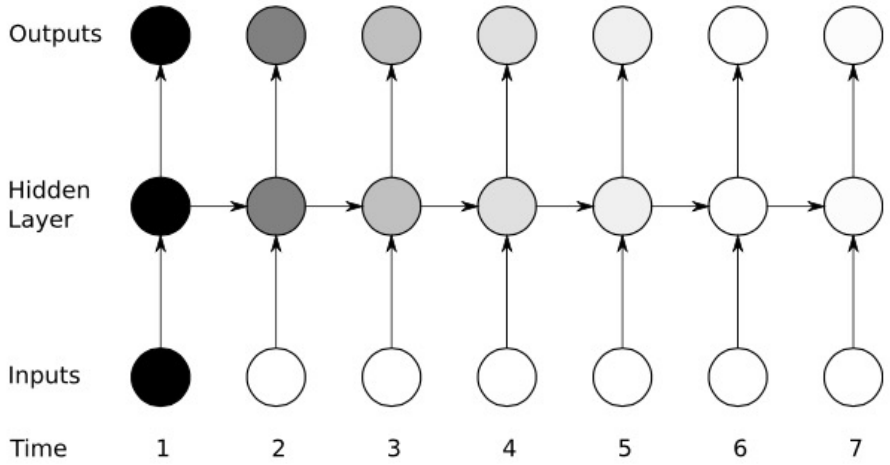

Figure 1: A Simple Recurrent Neural Network[1]

At each time t, the network receives external data through the input nodes, which it processes into the appropriate real-valued vector format. This information is transferred to the hidden layer, which manipulates or interprets the data. Importantly, at each time t, the processing done at the hidden layer is dependent on the input at time t as well as on the results of the hidden layer node at time $t-1$. At each time t, the model can be used to generate output via the output nodes, which emit the model's interpretation in the preferred format. In practice, of course, each of these nodes are actually mathematical functions, the exact nature of which we will describe in more detail later. More

[1]Image: `http://eric-yuan.me/wp-content/uploads/2015/06/1.jpg`

complex recurrent networks are also used in practice, but the above diagram sufficiently illustrates the core methodology that we will be discussing below. Before continuing, it is important to clarify why we use recurrent neural networks (RNN) here over a simpler neural network model, such as the feed-forward neural network below:

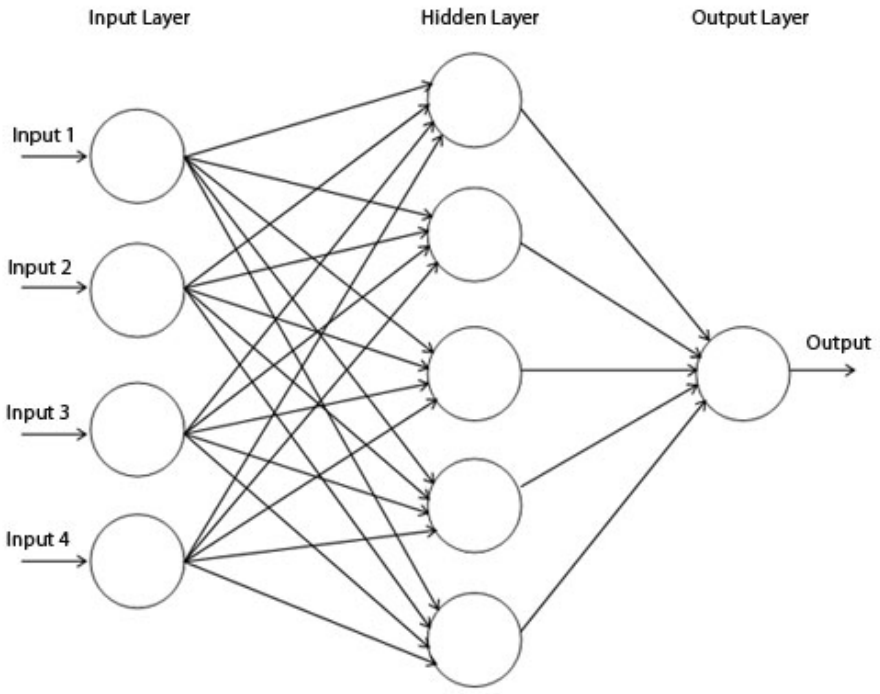

Figure 2: A Feed-Forward Neural Network[2]

The feed-forward neural network traditionally assumes that all inputs to the model have fixed uniform length and therefore conveniently receives all input simultaneously, as shown in the diagram. In a language setting, the above network would only accept sentences of length four. In comparison, at any time t, the RNN computes the current hidden state of the model as a function of both the previous hidden state and the current input, giving us the flexibility to use inputs of variable length. For example, Figure 1 shows the application of a RNN to a sequence of length seven. However, we can easily use the same RNN to model properties of a sequence of length twenty, simply by recursively computing the hidden state of the network twenty times on the new input sequence. The value of this in machine translation is obvious: sentences have variable length, and we want to build a translation system that is robust to changes in sentence or phrase length.

[2]Image: `http://www.codeproject.com/KB/dotnet/predictor/network.jpg`

4 Reviewing the Mathematics of Recurrent Neural Networks

Now we will elaborate on the actual mathematics behind recurrent neural networks. In the discussion that follows, we will consider an input sequence $X = (x_1, \ldots, x_T)$. For the purpose of this paper, we will assume that X is a sequence of words and each x_t represents a single word in the input sequence, though in practice smaller or larger units of language may be used.

The RNN model achieves flexibility to handle variable length inputs by the use of a time-indexed and recursive history vector which we will call h_t at time t. (h_t is often also called the memory state at time t.) Let x_t be an element in the sequence $X = (x_1, \ldots x_T)$. We define h_t recursively as:

$$h_t = \phi(x_t, h_{t-1})$$

with h_0 initialized to some vector (often the zero vector) and h_{t-1} representing the history of the sequence until time t. A common example of the transformation ϕ is:

$$\phi(x_t, h_{t-1}) = g(Wf(x_t) + Uh_{t-1})$$

where $f(x_t)$ is a function that turns the input x_t to a real vector.

In the language setting where x_t is a single word in a sentence or phrase, a commonly used method of creating $f(x_t)$ in neural networks is to first create an ordered dictionary of known tokens in the source language - we will call this the vocabulary V. We can express x'_t as a one-hot encoding of the token x_t which is nonzero and equal to 1 only at the index corresponding to the position of the token x_t in the vocabulary. This means $x'_t \in \mathbb{R}^{|V|}$.

Then we choose a matrix $E \in \mathbb{R}^{|V| \times d}$ and compute:

$$f(x_t) = E^T x'_t$$

Now $f(x_t) \in \mathbb{R}^d$.

Returning to the definition of $\phi(x_t, h_{t-1})$, W is called the input weight matrix and it is an element in $\mathbb{R}^{d \times n}$. Similarly U is called the recurrent weight matrix and it is an element in $\mathbb{R}^{n \times n}$. Finally, g, the activation function, is usually element-wise non-linear. The element-wise hyperbolic tangent function is a popular choice for g. Thus the following is a final example for the recursive definition for h_t:

$$h_t = \tanh(W^T f(x_t) + U^T h_{t-1})$$

Notice that the recursion formula forces h_t to be an element of $\mathbb{R}^n$ for all t, since the input to *tanh* is a vector in $\mathbb{R}^n$ and *tanh* is an element-wise function here. (Even if the choice of g was not defined element-wise, all h_t would be mapped to the co-domain of g.) Thus we have a method for transforming a single word and its context (the words in the sequence before it) into a continuous real vector. That is to say, using this methodology we can represent any

sentence or sequence of words with a vector in $\mathbb{R}^n$, regardless of the number of words in the sentence or sequence. This method constitutes what is known as the transformation layer of the neural network - it is where feature extraction occurs. In a learning context, the parameters E, W, and U would be optimized to achieve a better featurization.

Finally, it is also worth noting that for simplicity we have defined ϕ as a function which receives two inputs x_t and h_{t-1}. As we will see later, this is the exact form of ϕ used in the encoder of the translation model. In the decoder model, however, we will see that functions ϕ which accept more than two inputs are also used in practice. These are functionally quite similar to the ϕ described above and for brevity we will abstain from discussing them further.

5 Language Modeling with Recurrent Neural Networks

This framework equips us to model the probability of sentences of variable length. Recalling the definition of joint probability that $p(x, y) = p(x)p(y|x)$, we derive the following factorization of the probability of the sequence $(x_1, \ldots, x_T)$:

$$\begin{aligned} p(x_1, \ldots, x_T) &= p(x_1)p(x_2|x_1) \ldots p(x_T|x_1 x_2 \ldots x_{T-1}) \\ &= \prod_{t=1}^{T} p(x_t|x_{<t}) \end{aligned}$$

We can use neural networks to approximate $p(x_t|x_{<t})$. Recall that the history vector $h_t \in \mathbb{R}^n$ (for some n) summarizes the sequence $(x_1, ...x_t)$. Now choose $M \in \mathbb{R}^{n \times |V|}$ and $c \in \mathbb{R}^{|V|}$ and compute:

$$y_t = M^T h_t + c$$

Suppose each row i in M^T encapsulates our expectation of what the i^{th} word in the vocabulary looks like in a continuous vector representation. Then by the multiplication above, y_t is a vector in $\mathbb{R}^{|V|}$ where each row i in the vector contains a compatibility score of h_t to the word in the vocabulary V at index i. Each row of y_t contains the dot product of h_t with the i^{th} row of M^T - so large values for the dot product will indicate that these two vectors are closer in vector space, and therefore more similar.

We can transform these compatibility scores to probability scores using the soft-max function. For any word v_j at the j^{th} index in the vocabulary V we can define its probability at time t in the sequence as:

$$p(x_t = v_j) = \frac{e^{y_{t,j}}}{\sum_i e^{y_{t,i}}}$$

where $y_{t,j}$ indicates the value at the j^{th} index of y_t.

Now for any sequence $(x_1, ...x_T) = (v_1, ...v_T)$, we can model its probability as:

$$p(x_1, ...x_T) = \prod_{t=1}^{T} p(x_t = v_t)$$

where $p(x_t = v_t)$ can be determined by following the steps above: computing the appropriate h_t, computing y_t through matrix multiplication, and transforming the compatibility scores to probabilities using the soft-max function.

It is worth noting, of course, that the quality of these probability estimates is highly dependent on the choice of parameters for the neural model which determine the value of h_t, and on the choice of the parameters M and c. In our discussion of the Encoder-Decoder model, we will discuss a similar neural language modeling problem in which we make explicit the parameter tuning methodology for such applications.

6 Neural Machine Translation

6.1 Reframing the Machine Translation Problem

Now that we have introduced neural networks along with the statistical machinery needed for language modeling, we can begin to frame how neural networks can be used to produce the desired output - probability distributions over possible translations from a source language to a target language.

We begin with a sentence in our source language, $X = (x_1, x_2, \ldots, x_T)$, where x_1,...,x_T are the T words in sentence X. Our goal is to translate X by producing a sentence in our target language, $Y = (y_1, y_2, \ldots, y_{T'})$. We note that T is not necessarily equal to T' as the translation will most likely not comprise of the same number of words as the source sentence. Using the foundation of the statistical approach to machine translation, we model a specific word y_t of the target sentence by using the $t-1$ previous words of the target sentence, y_1,...,y_{t-1}, as well as the source sentence X. That is, we can express the likelihood of the t^{th} word in the translation as the conditional probability $P(y_t|y_1, \ldots, y_{t-1}, X)$. Now that we have neural networks at our disposal, we want to create a function to model that conditional probability which can then be trained by our neural networks.

6.2 The Encoder-Decoder Model

Generally speaking, the application of neural networks to machine translation that we are about to discuss follows an encoder-decoder model in which an encoder model is used to convert some source input into a "common language" representation. Then we apply a decoder model, which is used to take an input of the "common language" form and produce an output in the target language.

Let us first focus on the source sentence: $X = (x_1, x_2, \ldots, x_T)$. We are given a variable-length input sentence, and we want to create a summarization of this sentence in vector form that we can then feed into neural networks. This immediately provides the motivation for using a RNN as our encoder since it can handle variable length inputs as we discussed earlier. At a high level, the

function ϕ of our RNN encoder will be applied to each word of the source sentence in order, creating a memory state h_t for word x_t; after applying it to x_T, the last memory state h_T will be our continuous vector representation of the entire input sentence X. Then the continuous vector representation h_T is in a "common language" form, and we will see that this can be passed as input to another neural network.

Following the steps of section 4, we can transform the sequence X into a "common language" form represented by the memory state h_T of an encoding recurrent neural network. For a given word x_t in X, let x'_t be a one-hot-encoding of x_t. We also have an input weight matrix $E \in \mathbb{R}^{d \times |V_X|}$, with as many rows d as we desire (Cho et al. suggests somewhere in the hundreds[3]) and as many columns as there are words in our source language vocabulary V_X. The function of this weight vector parameter is to project each one-hot-encoded vector x'_t for the input word x_t into a continuous vector by defining the projection $f(x_t) = Ex'_t$. We then recursively pass each projection, along with the most recent memory state h_{t-1}, into our parameterized function ϕ_θ to get the next memory state h_t. In practice, we initialize h_0 to be a vector of zeros. In summary, the full implementation of the encoder recurrent neural network is $h_t = \phi_\theta(x_t, h_{t-1})$. Following the example from Section 4, we could define $h_t = \tanh(W^T f(x_t) + U^T h_{t-1})$.

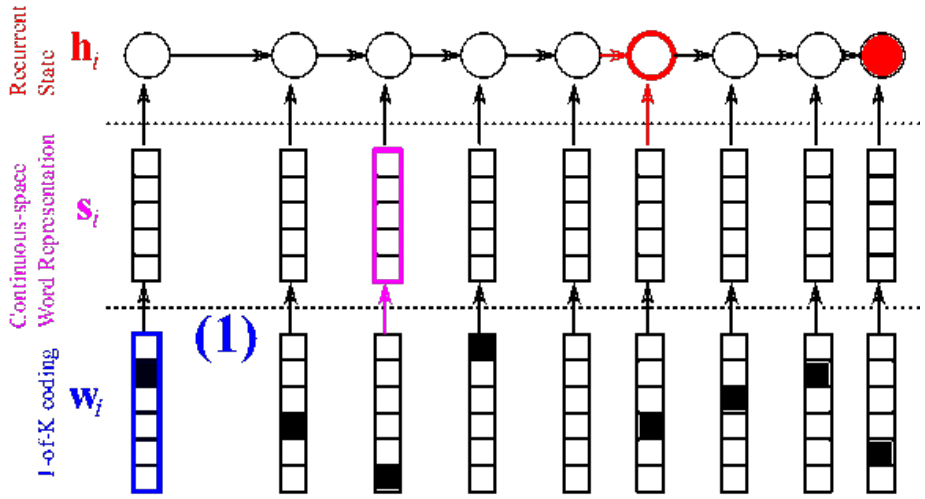

Figure 3: Diagram of the RNN encoder[4]

In the diagram above, we can see how each memory state in our neural network takes as input the previous memory state, as well as the projected representation of the word corresponding to that state. Furthermore, we can see that after

[3]Blog: https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-2/

[4]Image: https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2015/06/Figure-3_one-hot.png

computing h_T, we have a continuous vector form that summarizes our entire input sentence. This vector is often referred to as the context vector, and we will simply denote it as c.

Once we have used the encoder to transform X into the context vector c, we can use another recurrent neural network to serve as our decoder, taking c as its input at $t = 0$ and eventually producing a translation Y for X. Note that at any step t in the decoding process, we will have generated memory states, now denoted by m, for $m_1, m_2, ..., m_{t-1}$. As in the encoder, we will want these previous memory states to serve as inputs to the current memory state. Conceptually, these previous memory states will encode the probability distributions over the words in the vocabulary of our target language V_Y for $y_1, ..., y_{t-1}$, which will be the final inputs to m_t. Finally, once m_t is calculated, it will then be used to generate a prediction for y_t.

Notice that in the encoder recurrent neural network, each memory state only took in the corresponding word and the previous memory state. In the decoder, each memory state will take in the the context c, the previous word y_{t-1}, and the previous memory state m_{t-1}. We can see that at each time state t, we make full use of context information as well as the previous translation predictions that have been generated. Another difference between the encoder step and the decoder step is that rather than setting m_0 to a vector of all zeros as we did in encoding, we have the context c as information, so we set $m_0 = \phi(c)$. The full implementation of our decoder is $m_t = \phi_{\theta'}(m_{t-1}, y_{t-1}, c)$.

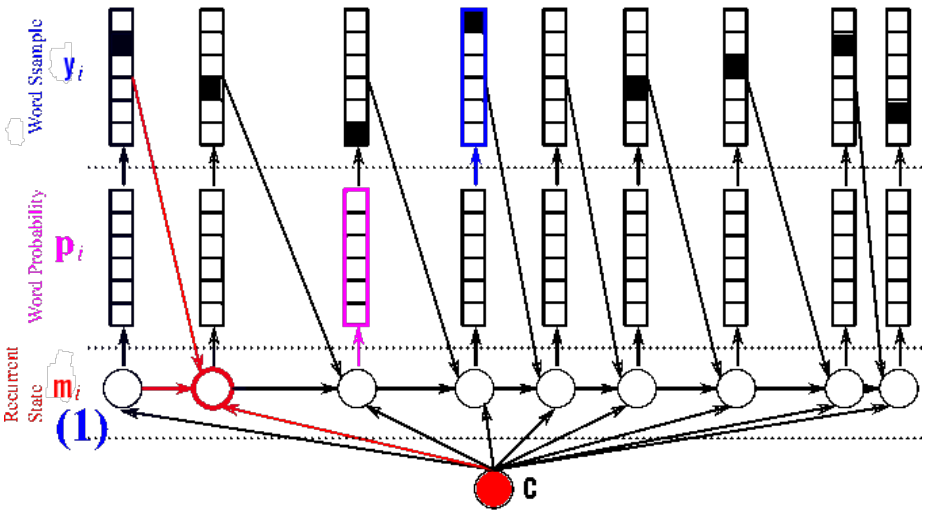

Figure 4: Diagram of the RNN decoder[5]

[5]Image: https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2015/06/Figure4_continuous-space.png

While this shows how each memory state, m_t, is computed until we reach the end of the sentence, we still need to be able to use the memory states to generate a target word $\hat{y}_t$. As we showed in the Language Modeling section, we can use these memory states, along with a parameter matrix M and a bias vector b, to compute a score for each word and use the soft-max function to create a probability distribution over each word in V_Y. After doing so, we can simply sample from this distribution to obtain a translation word, $\hat{y}_t$.

Algorithmically, we can summarize the entire Encoder-Decoder Model as follows:

1. Given an input sentence X of length T, recursively compute $h_t = \phi_\theta\left(h_{t-1}, x_t\right)$
2. Set $c = h_T$
3. Initialize $m_0 = \phi_{\theta'}(c)$
4. While $\hat{y}_t \neq \langle \text{eos} \rangle$, where $\langle \text{eos} \rangle$ represents an end-of-sentence tag:
   (a) Compute $m_t = \phi_{\theta'}\left(m_{t-1}, y_{t-1}, c\right)$
   (b) Calculate $P(y_t|y_1, ..., y_{t-1}, X)$ and sample $\hat{y}_t$ from the distribution

6.3 Training our Encoder-Decoder Model

With the Encoder-Decoder Model set up, we see how it can be used to generate translations, Y, for an input X. However, the model must first be trained before we can use it to make "good" translations. Given a parallel corpus D, with n pairs (X_i, Y_i) of a source sentence X and its corresponding translation Y in the target language, we can calculate the log-likelihood of the corpus given a model and parameters Θ as

$$L(\Theta, D) = \sum_{i=1}^{n} \log P(Y_i|X_i, \Theta)$$

Ultimately, we want to maximize the log-likelihood over the model parameters, which in the case of the Encoder-Decoder model are the parameters θ and θ' of ϕ. The details of the optimization process are outside the scope of this paper, but common methods for actually finding the maximizing parameters involve stochastic gradient descent, which in the case of neural networks can be done by a method called backpropagation[6].

6.4 Sampling as a Means to an End?

We elaborated on how we could use two recurrent neural networks to create a pipeline for the entire machine translation process, ultimately yielding a probability $P(Y|X)$ on a possible translation from X to Y. However, we have no idea whether this is a good translation. A naive solution is to repeat this process multiple times, obtaining many translations each with a conditional probability, and pick the one with the highest probability. This leads to an obvious

[6]Text Book: https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf

question: How many times must one sample from $P(Y|X)$ to get a "sufficiently good" translation? The answer is not clear, and we are most likely unwilling to subject ourselves to the computational complexity and risk of a bad translation in order to find out. This motivates the need for a better method of finding a good translation.

Ultimately, we wish to solve

$$Y_{\text{opt}} = \arg\max_Y \log P(Y|X)$$

by simply applying the log-likehood model to our conditional probability problem. Let us examine what happens if we try to solve this explicitly. At each step t in the decoder portion of the Encoder-Decoder model, we obtain a probability for each word in V_Y as a prediction for y_t. Theoretically, starting at $t = 0$ we could maintain a probability for each possible sequence as we traverse the neural network until we reach the last memory state $m_{T'}$ and reach an end-of-sentence tag. However, this means we would be maintaining $|V_Y|^{T'}$ sequences. Our vocabulary could be anywhere in the range of tens of thousands of words to hundreds of thousands of words, while a lower bound on our average sentence length might be 6 words. This results in having a lower bound of 10^{24} sequences that we are keeping track of, something that is computationally infeasible.

6.5 Beam Search

While we cannot keep track of every possible sequence, we can however keep track of a fixed number of sequences. At any point in the process, if we could only keep track of a fixed number, B, of the possible sequences, it is clear that we would want to keep track of the B sequences up to that point with the highest probability. This simple idea is the crux of the Beam Search, a method in which at every step of the decoder model, we keep track of the B sequences with the highest probability up to that point. We can see that if we take $B = 1$, we obtain a single sample from the sampling method discussed earlier. On the other hand, taking $B = \infty$ yields the infeasible explicit maximization of the log-likelihood model. Based on our computational constraints, the size of our target vocabulary, and an expected length for Y, we can explicitly calculate the beam size B that is most appropriate to use. Then, at each step t, we calculate $P(y_1, y_2, ..., y_t|X)$ given the B best probabilities from $P(y_1, y_2, ..., y_{t-1}|X)$ and a probability for each word in V_Y. Thus, we will have $V_Y \times B$ possible sequences, of which we will pick the B with the highest probabilities.

There is one more loose end to tie-up: Is this sufficient? In other words, given the computational constraints, are we picking the globally optimal solution by using the best B probabilities at each step? As it turns out, this may not be optimal. It is possible that a particular $B' \in \{1, 2, ..., B-1\}$ can yield a translation with a higher likelihood than the output of the beam search for B. An example of this possibility is shown in the link provided[7]. We can get around this by doing a beam search for each $B' \in \{1, 2, ..., B\}$, where B is determined by our computational constraints, and pick the translation with the highest probability out of all of the searches.

[7] pg. 95 of `https://github.com/nyu-dl/NLP_DL_Lecture_Note/blob/master/lecture_note.pdf`

7 Model Performance and Improvements

The Encoder-Decoder model described above, as well as improved iterations of it, were described and tested by Cho et al.[8] in 2012, even though the simple Encoder-Decoder had been proposed before then. The corpus used for training and evaluation was the English and French parallel corpus provided at the ACL 2014 Workshop on Statistical Machine Translation[9]. The models were used to generate translations from English sentences to French sentences in this case. The simple version of the Encoder-Decoder model, as described above, achieved a BLEU score of 17.82 on the test set. As a reference point, a baseline model using the Moses[10] statistical machine translation system, with default settings, achieved a BLEU score of 33.30 on the test set, while a state-of-the-art SMT model achieved a BLEU score of 37.03 on the test set. Given the poor performance of the Encoder-Decoder model, it is natural to ask if a complete neural network model has any chance of competing with state-of-the-art SMT models, which have been tuned over many years and are trained over enormous vocabulary datasets. As it turns out, implementations and improvements have recently been researched that build upon and rectify the weaknesses of the simple Encoder-Decoder model, creating full neural network models and ensemble models that can compete with, or even exceed, the state-of-the-art SMT models.

7.1 Effect of Long-Term Dependencies

In the Encoder-Decoder model, we used the recurrent neural network in its most fundamental formulation, and for that reason it is often called a simple recurrent neural network. Even though each memory state h_t summarizes the memory states that preceded it, it turns out that the simple recurrent neural network has trouble capturing long-term dependencies in its inputs. A long-term dependency refers to some relationship or meaning between parts of the input that are not close in terms of the time index t. In our language setting, it could be a semantic relationship between words or phrases in the input sentence that are far away from each other. Looking at the simple recurrent neural network in Figure 1, the gradient of the color at each time t shows how the information in the first memory state from the first input is slowly diluted as t increases. We can see how a long-term dependency between the first word and the last word might not be fully extracted with this kind of model.

In the late 90's, long short-term memory units, referred to as LSTMs, were introduced as a modification to recurrent neural networks that would allow them to better capture long-term dependencies. While recurrent neural networks memory states definitively pass on the information that passes through them to proceeding memory states, LSTMs use gates to determine what information to pass on. It is this property that gives them the ability to better pass on long-term dependencies through the recurrent neural network. The first gate, called the forget gate, determines what information should be thrown away within a particular state. The next gate, the input gate decides what information should be used to create an update to the state. Finally, the output gate determines

[8] Paper introducing Encoder-Decoder: http://arxiv.org/pdf/1406.1078v3.pdf
[9] Source of parallel corpus: http://statmt.org/wmt14/translation-task.html
[10] Moses system homepage: http://www.statmt.org/moses/

what information, and how much of it, the current state will show and pass on the rest of the recurrent neural network.

Sutskever et al.[11] researched the improvements of LSTM models over other models and showed promising, and even some surprising, results. In addition to using LSTMs as the base recurrent neural network, the paper found that by reversing the order of the source language sentences being input into the LSTM, there were significant improvements in BLEU score, even for longer sentences. Even though the average distance between words or phrases with a dependency stayed the same, now the minimal distance was much smaller. The authors surmised that this decrease in minimal distance between words or phrases with a dependency resulted in a more efficient memory allocation by the LSTM which allowed for improved performance during the stochastic gradient descent optimization process. This reversal method improved the BLEU score from 25.6 for their base LSTM model to 30.6 on the same English to French WMT'14 translation task mentioned above. By using ensembles of LSTMs, they were able to improve the BLEU score to 34.81, beating out the phrase-based Moses system by a statistically significant margin.

7.2 Mapping to a Single Context Vector

As we have discussed extensively, the ability of recurrent neural networks to map input sequences of any length to a fixed-length history vector h_t makes them particularly suited for applications to language modeling problems including translation. However, this application also demands that we must build incredibly complex neural networks that are capable of summarizing and adequately modeling natural language sentences of arbitrary form, meaning, and length with a single context vector h_t of fixed length. Even then, as verified empirically by Cho et. al[12] in 2014, the performance of our neural translation models will often degrade rapidly as the length of the input sequences or sentences increases. This is ultimately because the recursive definition of h_t means that for longer sentences we must compress more information into fixed-length vectors.

Recall the definition of h_t:

$$h_t = \phi(x_t, h_{t-1})$$

In order to build robust and high-performing "simple" Encoder-Decoder models, we must find functions ϕ which are both highly non-linear and complex - which also means producing encodings which are extremely large so that we can compress more information into them. This is a significant mathematical and computational burden – the former from the necessary complexity of the function ϕ and the latter from the computational demands of working with extremely large vectors and matrices.

In response to this problem, researchers have begun to question the very premise of language models using recurrent neural networks – that we summarize an en-

[11]Paper on LSTMs: https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[12]Paper analyzing neural machine translation: http://arxiv.org/pdf/1409.1259v2.pdf

tire sentence with a *single, fixed-size vector*. Rather, they are interested in a more flexible approach, one that summarizes variable length input sequences with variable length representations in vector form.

An interesting solution which has been posed is to summarize an input sequence $X = (x_1, \ldots, x_T)$ with a *set* of context vectors h_t, often called the context set C. In this approach, we build two neural networks – one which reads the input sequence in the given order, and one which, as described in section 7.1 reads the input sequence in reverse order. For each x_t in the sequence X, the two neural networks produce two context vectors $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ - the 'forward' context vector and the 'backwards' context vector respectively. We then define h_t as the concatenation of these two vectors:

$$h_t = (\overrightarrow{h_t}, \overleftarrow{h_t})$$

and store each h_t in the set $C = \{h_1, \ldots, h_T\}$. With this formulation, each h_t encapsulates the information of the sequence elements preceding the term x_t as well as the information in the sequence elements following the term x_t. The size of the set C is equal to the sequence length T.

Though this formulation helps us create a more expressive encoding of the input sequence X, we still need to find a way to summarize the new context set C as a single vector which can be passed to the decoder of the neural translation model. To do this, at each time t in the decoding step, we compute a compatibility score for each h_t in C as:

$$e_{i,t} = f_{score}(z_{t-1}, \hat{y}_{t-1}, h_i)$$

where z_{t-1} is the previous state of the decoder network and $\hat{y}_{t-1}$ is the last word generated in the translation. It is also worth noting that f_{score} is a parameter of the model and accordingly needs to be tuned according to the specific application and data set.

Using the soft-max function we normalize these scores:

$$\alpha_i, t = \frac{e_i}{\sum_i e_{i,t}}$$

and we write a new context vector c_t as:

$$c_t = \sum_i \alpha_{i,t} h_i$$

which is used at time t to compute the memory state of the decoder.

Ultimately this approach gives us a more complex Encoder-Decoder model which is capable of incorporating more of the nuance of the original source sentence, but does not require the enormous computational resources that come with working with much larger vectors and matrices at the encoder stage. The performance of this improved model was 28.45, which is a staggering increase over the BLEU score of 17.82 by the simple Encoder-Decoder model.

7.3 Neural Network's Vocabulary Limit

One aspect of the Encoder-Decoder model that has a strong impact on both computational complexity and performance is the choice of the target vocabulary, V_Y, and in particular, its size. As $|V_Y|$ increases, we will see an increase in the training time for the model as each target word must be considered in the optimization process. The trade-off is that we should see an increase in performance as well, as V_Y is more likely to contain the words in all of our target labels Y_n. Especially when using the BLEU metric for scoring, there will be a harsh penalty when the model encounters target words in training that are not in V_Y. In practice, the typical size of V_Y that is computationally feasible is in the tens of thousands.

A breakthrough was made by Cho et al.[13] when they proposed a method that can utilize a very large vocabulary without the additional computational complexity in training time. During the training process, rather than using the entire vocabulary they suggest taking a much smaller subset $V'_Y \subset V_Y$. By using importance sampling to select the subset, they were able to significantly reduce the training time, while still using the entire vocabulary at the prediction stage. Building upon the context set improvement in 7.2, they combined this large vocabulary implementation with translation-specific techniques and ensemble methods, and were able to achieve a BLEU score of 37.19, which surpasses the state-of-the-art threshold mentioned earlier of 37.03.

8 Conclusion

By introducing neural networks as a tool for language modeling, we were able to create an entire translation pipeline consisting of encoder and decoder recurrent neural networks. With the improvements and modifications to the simple encoder-decoder model mentioned in section 7, researchers have been able to surpass many industry-standard and state-of-the-art statistical machine translation systems. This is a tremendous accomplishment given that the field of neural machine translation has only risen to relative prominence in the last couple years. However, machine translation is a notoriously difficult task. There is still much room for improvement in the BLEU scores on the corpus described and in more difficult translation tasks such as Mandarin to English. Current research in the field of neural machine translation includes efforts to improve translations between languages without common roots or internal structure, to better encapsulate the many nuances of human language use, and to generally build more robust translation systems with higher BLEU scores.

[13]Paper with Large Vocabulary: http://arxiv.org/pdf/1412.2007v2.pdf

References

Aside from the explicit citations above for direct references of methods, results and borrowed images, there were some resources that proved invaluable to us in understanding the general theory of neural networks and their use in machine translation. In particular, Kyunghyun Cho's course notes and blog posts regarding the topic were especially useful and served as the foundation for our understanding of the field. Specific sources are listed below:

1. Course notes from Kyunghyun Cho's Natural Language Understanding with Distributed Representation class:
   `https://github.com/nyu-dl/NLP_DL_Lecture_Note/blob/master/lecture_note.pdf`

2. A Brief Introduction to Neural Networks by David Kriesal:
   `http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf`

3. Slides from the presentation 'From Sequence Modeling to Translation' by Kyunghyun Cho at the University of Montreal:
   `https://drive.google.com/file/d/0B16RwCMQqrtdNEhwbHN2bXJzdXM/view`

4. The online tutorial 'Recurrent Neural Networks Tutorial' by Denny Britz:
   http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

5. Cho's blog series 'Introduction to Neural Machine Translation with GPUs' (parts 1-3) for NVIDIA:
   https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-with-gpus/
   https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-2/
   https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-3/

6. An overview of Long Short-Term Memory Units 'Understanding LSTM Networks' by Christopher Olah:
   `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`