Recurrent Neural Networks for Machine Translation

Nicholas Gambirasi
The University of Texas at Dallas
Department of Engineering and Computer Science
Richardson, TX, USA
nag170330@utdallas.edu

Nadhiya Ganesan
The University of Texas at Dallas
Department of Engineering and Computer Science
Richardson, TX, USA
nxg200032@utdallas.edu

Ruturaj Nikam
The University of Texas at Dallas
Department of Engineering and Computer Science
Richardson, TX, USA
rrn200000@utdallas.edu

Ameya Potey
The University of Texas at Dallas
Department of Engineering and Computer Science
Richardson, TX, USA
anp200000@utdallas.edu

***Abstract*—This report examines the general theoretical background to recurrent neural networks (RNNs) for machine translation, as well as the methods and results of a recurrent neural network implementation specific to machine translation from English to French and French to English.**

***Index Terms*—deep learning, machine learning, machine translation. natural language processing, neural networks, recurrent neural networks**

I. Introduction

Thanks to the technological advances of the late 20th and early 21st centuries, it has never been easier for people from different countries, continents, and cultures to connect with one another. It remains true, however, that one of the largest obstacles to establishing those deep connections is effortless intercultural communication. The world can only be so connected if people from different cultures are not able to understand one another's speech and writing.

It is for the above reason that machine translation is currently one of the most researched problems in deep learning for natural language processing. Though research began early in the 21st century, only recently have large-scale applications become truly effective at translating from language to language. This newfound success is due to improvements in the architecture of **Recurrent neural networks (RNNs)** that have enabled machines to not only translate words and characters in different languages, but also understand underlying semantic and grammatical differences between the languages.

In this report, we explore the historical background of recurrent neural networks for machine translation. Additionally, we will highlight the mathematical theory behind training the most basic encoder-decoder model using deep RNNs, and analyze our own implementation of this model on a small English-French translation dataset. Additionally, we will detail some future work that would build upon ours to allow for better results in a large-scale translation application.

II. Background

A. RNN Implementation for Machine Translation

Research into neural machine translation (the use of neural networks for machine translation) began in the late 20th century, to serve as competition for the most prominent method at the time, statistical machine translation (SMT) [1]. Due to poor initial results, research into neural machine translation was halted almost immediately following its introduction. But, with the expansion of the use of deep learning and deep neural networks in the 2010s, neural nets were once again at the forefront of research in machine translation.

One of the first deep learning architectures considered for machine translation was the recurrent neural network. Neural machine translation (NMT) researchers were drawn to the use of RNNs because of their reliance on previous model states to determine the current state. When applied to the use case of machine translation, RNNs allow for better learning of semantic and grammatical rules in languages than other deep neural networks.

From their initial implementation, the use of RNNs yielded impressive results. The premier implementation of a pure deep RNN for machine translation acheived results near that of the best statistical machine translation method, which was the product of decades of research and development [2]. From this point forward, recurrent neural networks dominated the field of neural machine translation. Eventually, improvements such as the introduction of attention mechanisms by led to RNN-based approaches achieving results far superior to SMT methods. The first industry-level machine translation model was implemented in Google Translation. It was an RNN-based method with a feature attention mechanism, and the results achieved were so substantial that it is considered a

milestone in neural machine translation [3]. Most models designed for machine translation today are some derivative of a basic reurrent neural network.

III. Conceptual and Theoretical Background

A. RNN Architecture

Recurrent neural networks are very similar to standard deep neural networks, with the exception that the recurrent neural networks are time-dependent. Simply put, the present state of a recurrent neural network is based partially on the input presently being received, while also being partially based on the network's state in the previous time state. This property allows for more accurate prediction of a time-dependent event (such as the probability of a word appearing after another word), and thus models language more accurately than standard neural networks would.

Similar to a standard deep neural network, a deep RNN has an input layer that takes a set of sequences, typically of integers. Where these networks differ however, is how the **hidden layers** are handled. In a standard neural network, the weights of each hidden layer are only calculated once, and the whole input sequence is used to determine the values of the hidden nodes. However, in recurrent neural networks, the weights of each hidden layer are calculated repeatedly, treating each element of the input sequence as it's own instance in time (i.e. the first element of the input sequence represents time $t = 1$, and the last element represents time $t = T$). For an input sequence of length T, the hidden node values are calculated T times (see figure below). Standard and recurrent

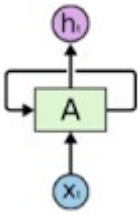

Fig. 1. Repeated calculation of hidden states from input state

neural networks also have similar output layers, but in the case of the recurrent neural network only the final time state is used to calculate the output values of the network.

B. The encoder-decoder

Historically, the encoder-decoder structure has been the most frequently and longest used structure when building RNN models for machine translation. It has gained popularity due to its ability to handle both inputs and outputs where the lengths of the sequences are not constant. As a quick example to demonstrate, take the English phrase, "I love you." and the equivalent French phrase, "Je t'aime." Tokenizing the English phrase yields a sequence of length four (*"I", "love", "you", "."*), whereas performing the same tokenization on the French phrase yields a sequence of length three (*"Je", "t'aime", "."*). Using a single RNN to translate these two sequences may yield an effective word-to-word map for this sequence in particular, but the model will not have been trained to correctly handle other cases with differing usage of the same words.

The encoder-decoder structure avoids this problem by building two connected recurrent neural networks (an encoder and a decoder, respectively)(see figure below). The purpose

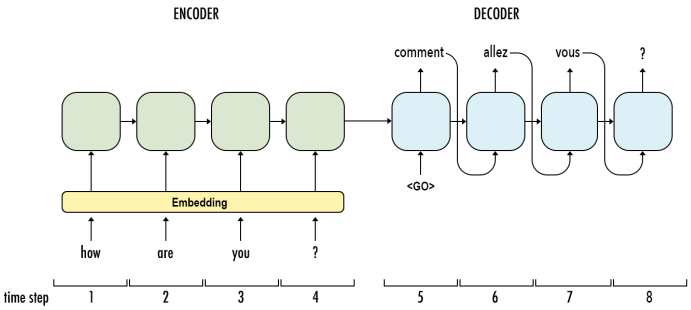

Fig. 2. Encoder-decoder example — from [6]

of the encoder is to take the sequence of words from the first language, and give that sentence a meaning, which is represented by a vector called a **semantic vector**. This semantic vector is used as the input of the decoder RNN, which learns how to decode the input into a sequence of words in the translated language. Using the encoder-decoder structure allows for accurate sequence-to-sequence modelling while also taking the semantics of a sequence into account.

C. How the encoder-decoder is Trained

When trying to understand how the weights of an encoder-decoder model are learned, there are a lot of parallels that can be drawn between the training of a standard neural network, and the training of a recurrent neural network. Generally, the encoder-decoder model is trained by going through a full forward pass of the model, where the input sequences are first fed through the encoder RNN to generate the semantic vector, then the semantic vector is fed into the decoder, which generates a predicted output sequence. The loss of the prediction is calculated by comparing the predicted sequence to the expected output sequence, using the cross-entropy loss function. Finally, the weights for both the encoder and the decoder are updated using the Adam optimizer combined with a learning rate that has been chosen optimally by the programmer. This cycle repeats until loss convergence, or for the specified number of iterations. In order to establish a deeper theoretical understanding, the mathematical formulae used in each part of a training iteration are included and explained in the below sections.

1) The Forward Pass: In all neural network training, the purpose of the forward pass is to use the current model state

to make make predictions that are then compared to the ground truth sequence values to calculate a loss value. In an encoder-decoder system, the forward pass feeds the sequences through both the encoder and decoder. Since the encoder and decoder are recurrent neural networks, the sequences are read through the model one time state at a time, rather than all at once as in standard neural network training. For each time state, the hidden states are saved for use in the next iteration. The hidden states of the current time state are calculated with respect to the previous time state, as well as the current input sequence:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t \tag{1}$$

where $W^{(hh)}$ is the weights matrix from hidden state to hidden state in the encoder, $W^{(hx)}$ is the weights matrix from the input to the first hidden state, and f is the selected activation function for the hidden layers. Typical activation functions used between the hidden layers are ReLU, tanh, or sigmoid activation functions, though others can be used where appropriate.
Once all of the time states have been completed in the encoder, the final time state is fed, in reverse order, into the decoder as the initial time state. The next hidden state is calculated using the prvious hidden state:

$$h_t = f(W^{(hh)}h_{t-1}) \tag{2}$$

where both $W^{(hh)}$ and f are defined the same as in Equation (1). For each of the hidden states of the decoder, the output at that time state is also calculated:

$$y_t = softmax(W^s h_t) \tag{3}$$

where W^s is the weights matrix between the hidden state and the output. In the case of sequence to sequence matching, the softmax function is the most commonly used activation function between the hidden layers and the output layer, though in rare cases other activation functions can be used to improve the results. Typically, the final timestate is used in loss calculations, but the hidden states at all times are used in backpropagation.

2) Loss Calculation: All different types of neural networks evaluate their performance using a loss function. While this loss functionc can vary from use case to use case, the goal is always to optimize the weight matrices so as to minimize the loss function. For the machine translation use case, categorical cross-entropy is used almost exclusively. The categorical cross-entropy loss function is defined below:

$$J(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{M} y_i^{(n)} log\hat{y}_i^{(n)} \tag{4}$$

The categorical cross-entropy function measures the distance between the predicted probability distribution of a sequence and the actual probability distribution of a sequence. Optimally minimizing the categorical cross-entropy assures that the predicted and actual sequence are as close to each other as possible, element-wise.

3) Backpropagation: The similarity between standard neural network training and encoder-decoder training is again demonstrated in the backpropagation step of training. The first goal of backpropagation is to calculate the direction that maximizes each individual weight matrix. These maximally steep steps are called **gradients**, and are then paired with the network's **learning rate** to update the weights matrices for the next training iteration. Calculating the gradients for each of the weight matrices is not as easy as simple one-dimensional differentiation — the chain rule for partial derivatives must be applied. This step is referred to as backpropagation because it traverses the neural net *backwards* and computes the partial derivatives in reverse order to get to the gradients of the weight matrices. This begins with the calculation of the gradient of the loss function with respect to the predicted output:

$$\frac{\partial L}{\partial \hat{y}_t} = \frac{\partial l(\hat{y}_t, y_t)}{T \cdot \partial \hat{y}_t} \tag{5}$$

This gradient is calculated for each time step *t*. All of these gradients are used in conjunction with their respective hidden states (that is, the hidden state of the decoder at time *t*) to calculate the gradient of the loss function with respect to the matrix between the decoder's hidden states and the output states:

$$\frac{\partial L}{\partial W_s} = \sum_{t=1}^{T} prod(\frac{\partial L}{\hat{y}_t}, \frac{\partial \hat{y}_t}{\partial W_s}) = \sum_{t=1}^{T} \frac{\partial L}{\partial \hat{y}_t} h_t^\top \tag{6}$$

In order to be able to calculate the gradients for the other two weight matrices of importance, it is necessary to understand how the loss function changes with respect to each of the hidden states of the decoder. For the final time state of the decoder, this is:

$$\frac{\partial L}{\partial h_T} = prod(\frac{\partial L}{\partial \hat{y}_t}, \frac{\partial \hat{y}_t}{\partial h_T}) = W_s^\top \frac{\partial L}{\partial \hat{y}_t} \tag{7}$$

Now, for each of the preceding time states, the gradient equation is reliant on the gradient of the loss function with respect to the *next* time state (which is opposite to the dependency seen in the forward pass step):

$$\frac{\partial L}{\partial h_t} = prod(\frac{\partial L}{\partial h_{t+1}}, \frac{\partial h_{t+1}}{\partial h_T}) + prod(\frac{\partial L}{\partial \hat{y}_t}, \frac{\partial \hat{y}_t}{\partial h_T}) \tag{8}$$

This equation can be simplified to:

$$\frac{\partial L}{\partial h_t} = W_{hh}^\top \frac{\partial L}{\partial h_{t+1}} + W_s^\top \frac{\partial L}{\partial \hat{y}_t} \tag{9}$$

All of the necessary partials to calculate the gradients of the other two weights matrices are complete. The gradients are now computed using the product and chain rules of multivariable calculus:

$$\frac{\partial L}{\partial W_{hx}} = \sum_{t=1}^{T} prod(\frac{\partial L}{\partial h_t}, \frac{\partial h_t}{\partial W_{hx}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} x_t^\top \tag{10}$$

and, finally:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} prod(\frac{\partial L}{\partial h_t}, \frac{\partial h_t}{\partial W_{hh}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} h_{t-1}^\top \tag{11}$$

These gradients, paired with a developer-chosen learning rate, are used to update the weight matrices until the loss function from the previous subsection is optimally minimized.

IV. A Simple Encoder-Decoder for Machine Translation

A. The Implementation

1) Dataset Selection: The dataset on which the practical implementation was trained and tested is a subset of the same English-French dataset used by Google to train Google Translate, the parallel corpus English-French dataset from Europarl. Due to time constraints, only a very small subset of this dataset was used.

2) Preprocessing of Data: As in other types of natural language processing, the sentences must be converted into numerical vectors so that the encoder-decoder model can understand it. In a practical environment, there are numerous NLP libraries that can be used to achieve this task, but the most simple implementation was done via Tensorflow's preprocessing library. Several simple steps are used in the preprocessing pipeline of the practical implementation, those being:

- **Lowercase conversion**: covert each word in each input and output sequence to lowercase
- **Tokenization**: for each input and output sequence, assign each word a term frequency value based on how common it is in the input or output vocabulary
- **Beginning** and end of sequence appending: 'SOS' and 'EOS' tokens to the beginning and end of the output vectors, so the decoder knows where the sequence starts and ends.
- **Sequence padding**: Pad the sequences with zeros so that each input sequence is of the same length — this allows for model simplification without having an effect on the output
- **One-hot encoding**: For each input and output vector, create a new sequence vector that spans the whole vocabulary, indicating whether each word in the vocabulary is present in the sequence.

3) Model Construction: An encoder-decoder structure was constructed using the standard architecture discussed previously. The weight matrix between the input state and the hidden state was initialized with dimensions that reflect both the size of the preprocessed input vectors and the number of neurons in the hidden layer. The weight matrix was constructed as a square matrix of dimension equal to the number of neurons in the hidden layer. Finally, the decoder weight matrix between the final hidden state and the output state was initialized with dimensions that reflected the number of hidden neurons and the length of the desired output sequence. In the case of weight initialization specifically, each individual weight in the matrix was selected as an independent and identically distributed event from standard distributions that varied based on the size of the input/output vector and the size of the hidden layer.

B. Results

The following are the results obtained when training and testing model iterations. Due to computational constraints and time constraints, these models were trained for a very short period of time, and it is expected that the loss values would continue to improve the longer the models were trained for. Each of these models was trained for 5 epochs, then tested against our test dataset:

TABLE I
Training and Test Error Measurements

Batch Size	Learning Rate	Train Loss	Test Loss
20	1e-3	0.999132	0.935120
20	1e-4	0.978612	0.841516
50	1e-3	0.959976	0.823567
50	1e-4	0.958498	0.793189
60	1e-3	0.939891	0.731812
60	1e-4	1.07588	0.624120

As previously discussed in the theoretical portion of this report, the error function used in both the training and test methods are the categorical cross-entropy function, which simply measures the distance between the probability distributions of the predicted and actual outputs. Aside from the outlying training results at a batch size of 60 and a learning rate of *1e-4*, as the batch size increased and the learning rate decreased, the training and testing errors continuously improved. This is an observation derived strictly from the hyperparameter combinations that were tested in the practical implementation, although at some point too small a learning rate would lead to an increased loss versus a more optimal learning rate.

V. Results Discussion

It is important to note that the ability to completely and accurately train large recurrent neural networks for this task was severely inhibited by our inability to train on multiple machines at once (due to most of the work being done in colab, where only one runtime instance can be active), as well as the lack of computational power to train models. In ideal working conditions the results would have been much more accurate across both the training and test sets. It should also be noted that due to time constraints the size of the training and test sets were reduced significantly below the originally expected sizes. Additionally, the Colab environment had a very difficult time training on even the smaller datasets. This led to a lot of unforseen and unoptimal training and testing results, as Colab continuously crashed in the latter stages of training epochs.

VI. Future Work

The implementation seen in the previous sections is that of a pure deep recurrent neural network. However, recent developments in this field have resulted in the discovery of better RNN implemenations than a pure RNN. In the future,

implementation of a bidirectional RNN, as well as the use of LSTM and GRU methods, would certainly yield better training results than that above. It would be worth coding some of these implementations and running comparisons between the pure RNN and the other variations that have been uncovered in the latest research, just to understand how substantial of a difference there is between them. Implementing attention mechanisms would also greatly improve results across both training and testing of the encoder-decoder setup.

ACKNOWLEDGMENTS

We would like to thank Dr. Anurag Nagar for his support on this project and for serving as a mentor during the implementation phase of our project.

REFERENCES

[1] A. Soleimany, "MIT 6.S191: Recurrent neural networks - youtube," Youtube, 12-Feb-2021. [Online]. Available: https://www.youtube.com/watch?v=qjrad0V0uJE. [Accessed: 30-Nov-2021].

[2] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," arXiv, 19-May-2016. [Online]. Available: https://arxiv.org/pdf/1409.0473. [Accessed: 30-Nov-2021].

[3] D. Lazar, "Creating a Simple RNN from Scratch with Tensorflow," Medium, 01-Jul-2021. [Online]. Available: https://medium.com/nabla-squared/creating-a-simple-rnn-from-scratch-with-tensorflow-8995a03c976d. [Accessed: 30-Nov-2021].

[4] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," arXiv, 14-Dec-2014. [Online]. Available: https://arxiv.org/pdf/1409.3215. [Accessed: 21-Nov-2021].

[5] R.B. Allen (1987). Several Studies on Natural Language ·and Back-Propagation.

[6] S. Wang, Y. Wang, and X. Chu, "A Survey on Deep Learning Techniques for Neural Machine Translation," arXiv, 18-Feb-2020. [Online]. Available: https://arxiv.org/pdf/2002.07526.pdf. [Accessed: 30-Nov-2021].

[7] T. Tracey, "Language translation with RNNS," Medium, 25-Feb-2019. [Online]. Available: https://towardsdatascience.com/language-translation-with-rnns-d84d43b40571?gi=d2d44e5b1bb0. [Accessed: 30-Nov-2021].