Innovative approach to Machine Translation using FS-RNN

A report submitted to

Cluster Innovation Centre

in partial fulfilment of the requirements of paper

IV.7 & VI.7 Semester Long Project

Pankaj Baranwal (Roll No. 11520)

Vaibhav Jain (Roll No. 11634)

Under the supervision of

Dr. Harendra Pal Singh

Cluster Innovation Centre

University of Delhi

Delhi-110007

Certificate

This is to certify that the work embodied in this report entitled “Fast Slow Recurrent Neural Networks” has been submitted to the Cluster Innovation Centre, University of Delhi towards the fulfilment of papers IV.7 & VI.7, “Semester Long Project”. This report has not been submitted in part or in full to any other University/ institution. The work has been carried out under the supervision of Dr. Harendra Pal Singh, assistant professor of Mathematics at Cluster Innovation Centre.

Acknowledgement

Our sincere thanks go to our mentor, Dr. Harendra Pal Singh, who has guided us throughout the stages of this project. His expertise and experience has significantly improved the quality of our final work.

A special gratitude for our colleagues and hostel mates who were always there for those long and quite intuitive discussions we had over tea at odd hours.

We would also like to thank our college, Cluster Innovation Centre (CIC), for providing the facilities and the necessary resources to complete our project.

Abstract

In the past few years, various neural network architectures have been presented and deployed to solve the problem of neural networks not being able to handle ordered or sequential inputs. Among them, recurrent neural networks have been hugely successful. But they have their own strengths and shortcomings. So, various recurrent neural network architectures have been proposed which are better equipped at handling some of these shortcomings.

But there still exists a lot of scope for improvements. For these reasons, in this project, the main ideas behind handling sequential data are first presented and then a comparative analysis is performed over the performance of these different architectures. Then, a recently published novel approach of "Fast Slow recurrent neural networks" is discussed which incorporates the strengths of both multiscale RNNs and deep transition RNNs. This hybrid architecture enables it to process sequential data on different timescales and learn complex transitions between different timesteps. This novel approach is then added to the comparative analysis and then its claim of being able to outperform the state-of-the-art results is tested by implementing the architecture on the NMT based seq2seq machine translation technique which employs a special attention based encoder-decoder mechanism in conjunction with recurrent neural networks to provide state-of-the-art language transition results.

All of our code has been open sourced under MIT license and can be found at:
https://github.com/Pankaj-Baranwal/ML

Keywords - Recurrent Neural Network, LSTM, Neural Machine Translation

Table of Contents

List of Figures

1. Introduction

Our era is seeing the emergence of an intelligence in silicon that rivals our own, and it has led to widespread research and adoption of the tools and techniques involved in machine learning and in general, artificial intelligence. But the field is still in its infancy with a lot of research into improving its accuracy and adaptive ability still needed to make it more efficient and competent.
A major problem with the traditional feed forward neural networks is that they are unable to handle sequential data like speech or text or time-series data like stock market data. But then recurrent neural networks came which are quite good at handling sequential data of variable length. There are still many areas of improvement in the way RNNs operate. For example, each hidden unit in RNNs is shallow in itself. Hence, it is unable to establish complex non linear transition functions between the input and the output. There is also the case of adapting to a completely different input data which traditional RNNs are not very good at handling.

So, in this project, the main ideas behind handling sequential data are first presented and then a comparative analysis is performed over the performance of these different architectures. Then, a recently published novel approach of "Fast Slow recurrent neural networks" is discussed which incorporates the strengths of both multiscale RNNs and deep transition RNNs. This hybrid architecture enables it to process sequential data on different timescales and learn complex transitions between different timesteps. This novel approach is then added to the comparative analysis and then its claim of being able to outperform the state-of-the-art results is tested by implementing the architecture on the NMT based seq2seq machine translation technique which employs a special attention based encoder-decoder mechanism in conjunction with recurrent neural networks to provide state-of-the-art language transition results.

2. Related Work

Neural Machine Translation using Neural Networks is a fairly new concept. The first scientific paper which introduced Neural Networks in machine translation was published in 2014. This was followed by a lot of advances in the next few years. Therefore it is safe to say that NMT is still is in its inception.
But there are few organisations, which have taken a keen interest in this field and put a huge amount of efforts and resources to develop state-of-art NMT models. Following is a brief description of some of these works.

2.1. GNMT by Google

NMT systems are known to be computationally expensive both in training and in translation inference. Also, most NMT systems have difficulty with rare words. These issues have hindered NMT's use in practical deployments and services, where both accuracy and speed are essential. Google address these issues by introducing GNMT, Google's Neural Machine Translation system.

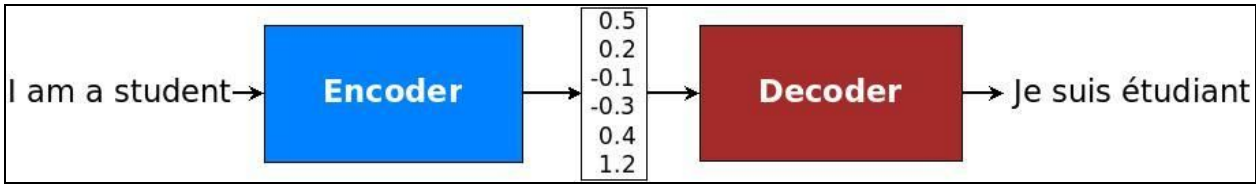


Figure 1: Encoder decoder model employed in NMT architectures
(Source: https://github.com/tensorflow/nmt)

Their model consists of a deep LSTM network with 8 encoder and 8 decoder layers using attention and residual connections. To improve parallelism and therefore decrease training time, they used attention mechanism which connects the bottom layer of the decoder to the top layer of the encoder. This also allows better remembering of text context. To accelerate the final translation speed, they employed low-precision arithmetic during inference computations. To improve handling of rare words, they divided words into a limited set of common sub-word units ("word pieces") for both input and output. This method provides a good balance between the flexibility of "character"-delimited models and the efficiency of "word"-delimited models, naturally handles translation of rare words, and ultimately improves the overall accuracy of the system. Their beam search technique employs a length-normalization procedure and uses a coverage penalty, which encourages generation of an output sentence that is most likely to cover all the words in the source sentence. On the WMT'14 English-to-French and English-to-German

benchmarks, GNMT achieves competitive results to state-of-the-art. Using a human side-by-side evaluation on a set of isolated simple sentences, it reduces translation errors by an average of 60% compared to Google's phrase-based production system.

2.2. OpenNMT by MIT

OpenNMT is an open source initiative for neural machine translation and neural sequence modeling by MIT. The main objective behind this project was that among the several existing NMT implementations, many systems such as those developed in industry by Google, Microsoft, and Baidu, are closed source, and are unlikely to be released with unrestricted licenses. And many other systems such as GroundHog, Blocks, tensorflow, seq2seq, lamtram, exist mostly as research code. These libraries provide important functionality but minimal support to production users. With these problems in mind, the OpenNMT (http://opennmt.net), an opensource framework for neural machine translation was introduced. OpenNMT is a complete NMT implementation. In addition to providing code for the core translation tasks, OpenNMT was designed with three aims: (a) prioritize first training and test efficiency, (b) maintain model modularity and readability, (c) support significant research extensibility.

3. Underlying Concepts

3.1. Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

Neural Network is not a new concept. Theoretical research has been performed on neural network a long back in time, since mid 1940s. But the application of neural networks on real world models was not possible until the introduction of GPUs in the 2000s. Many important advances have been boosted by the use of inexpensive computation power since then.
A neural network usually involves a large number of processors operating in parallel and arranged in tiers. The first tier receives the raw input information -- analogous to optic nerves in human visual processing. Each successive tier receives the output from the tier preceding it, rather than from the raw input -- in the same way neurons further from the optic nerve receive signals from those closer to it. The last tier produces the output of the system.
Each processing node has its own small sphere of knowledge, including what it has seen and any rules it was originally programmed with or developed for itself. The tiers are highly interconnected, which means each node in tier n will be connected to many nodes in tier n-1 -- its inputs -- and in tier n+1, which provides input for those nodes. There may be one or multiple nodes in the output layer, from which the answer it produces can be read.

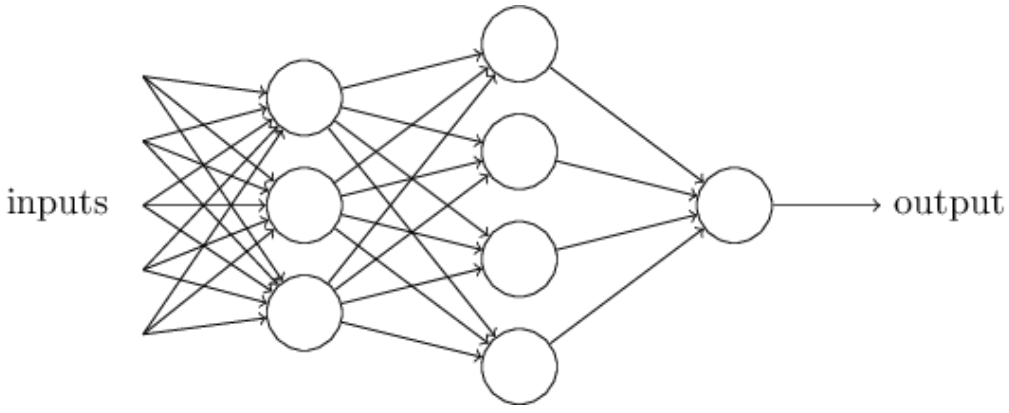


Figure 2: Depiction of a general feed forward neural network
(Source: https://www.sciencedirect.com/science/article/pii/S1364815214001418)

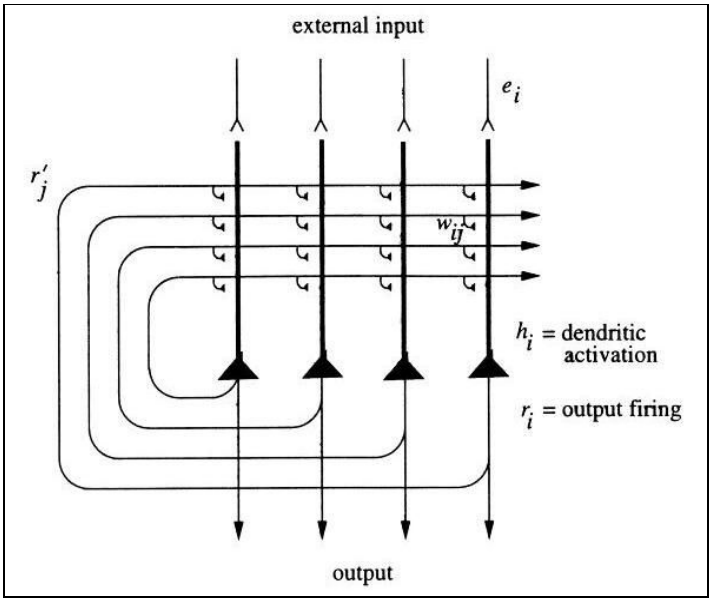
Neural networks are notable for being adaptive, which means they modify themselves as they learn from initial training and subsequent runs provide more information about the world. The most basic learning model is centered on weighting the input streams, which is how each node weights the importance of input from each of its predecessors. Inputs that contribute to getting right answers are weighted higher.
Typically, a neural network is initially trained, or fed large amounts of data. Training consists of providing input and telling the network what the output should be. For example, to build a network to identify the faces of actors, initial training might be a series of pictures of actors, non-actors, masks, statuary, animal faces and so on. Each input is accompanied by the matching identification, such as actors' names, "not actor" or "not human" information. Providing the answers allows the model to adjust its internal weightings to learn how to do its job better.

3.2. Recurrent Neural Net (RNN)

Recurrent Neural Networks are great at processing sequential data where information from the past is also utilized when analysing the present to make predictions of the future. These networks share parameters across different positions / index of time / time steps of the sequence, which makes it possible to generalize well to examples of different sequence length. RNN is usually a better alternative to position-independent classifiers and sequential models that treat each position differently.

- Why are recurrent NNs often good with processing variable-length sequential inputs?

The key feature of a recurrent neural network (RNN) is that part of the network’s state is fed back in as input. This gives the RNN a form of persistent memory about past inputs.



[1]

Figure 3: Unrolled representation of an RNN
(Source: https://stackoverflow.com/questions/45223467/how-does-lstm-cell-map-to-layers)

As shown in the figure above, a recurrent neural network can be considered as multiple copies of same network with each network passing the message to its successor. Because each simulated iteration of the RNN will combine a "running memory of the past" with the latest input, a past of arbitrary length can be incorporated as context in interpreting the current input. And because the "memory of the past" is cumulative and is not hard-coded to be a particular length, that memory can handle pasts of variable length as well.

Here is the static recurrent formula used by traditional RNNs:

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

$\mathbf{h_t}$ is the new form of the RNN layer after taking and processing input vector x.

w is the parameter of the function f

$\mathbf{h_{t-1}}$ is the previous state and

$\mathbf{x_t}$ is the input at the current timestamp

So now, no matter how long or short the x (or our input) is, we take the x inside our network one time stamp at a time and process it inside the RNN with the same above function, every time, thus handling the variable length data.

[1] Rolls et al

3.2.1 Deep Transition RNN

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from the input to the hidden state, x(t) → h(t)
2. from the previous hidden state to the next hidden state, h(t-1) → h(t)
3. from the hidden state to the output, h(t) → o(t)

These transformations are represented as a single layer within a deep MLP in the previous discussed models. However, we can use multiple layers for each of the above transformations, which results in deep recurrent networks.

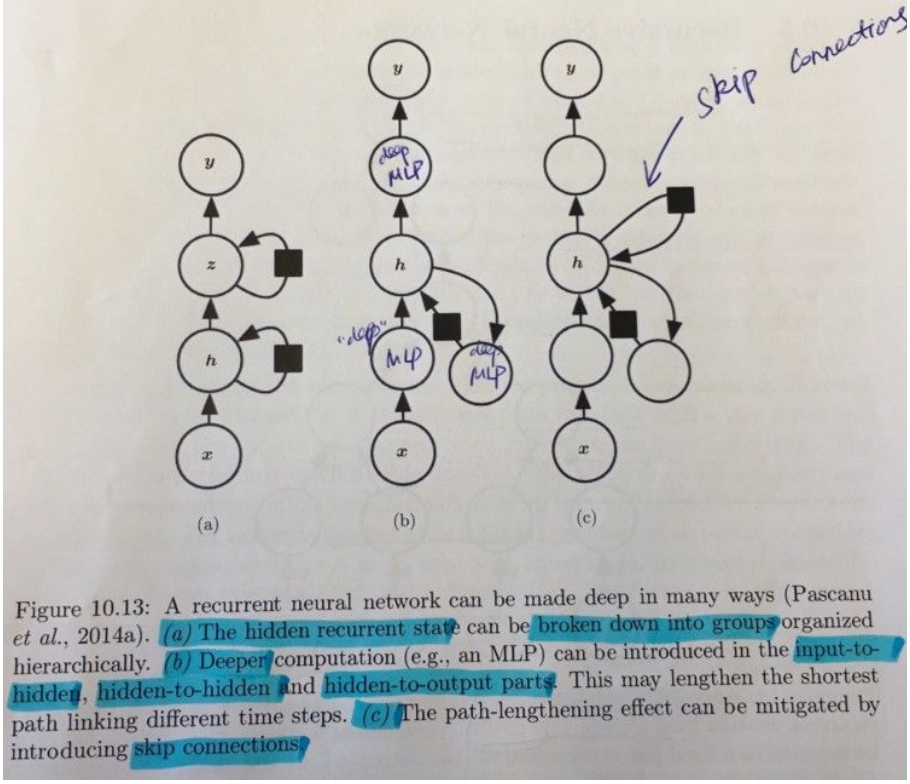


Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu *et al.*, 2014a). *(a)* The hidden recurrent state can be broken down into groups organized hierarchically. *(b)* Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps. *(c)* The path-lengthening effect can be mitigated by introducing skip connections.

Figure 4: Transition between classification of RNN and deep RNN

The above figure shows the resulting deep RNN, if we:

(a) break down hidden to hidden,
(b) introduce deeper architecture for all the 1,2,3 transformations above and
(c) add “skip connections” for RNN that have deep hidden-2-hidden transformations.

Use of deep Recurrent Neural Networks:

1) Deeper models are more powerful when it comes to representing certain functions.
2) When unfolded, RNNs are very deep. But at a particular time-step, RNNs are quite shallow. Deep RNNs take care of that.
3) hidden-to-hidden, hidden-to-output, input-to-hidden functions are shallow - no intermediate layers.

So, to summarize:

In deep RNN architectures, RNNs are stacked layer-wise on top of each other. In these architectures, the hidden states of all the hierarchical layers are updated once per time step (by one time step we refer to the time between two consecutive input elements). The additional layers enable the network to learn complex input to output relations and encourage an efficient hierarchical representation of information.[2]

3.2.2 Multiscale RNN

A multiscale RNN is a neural network model in which the hidden layers are grouped into different layers of cell. Each such group will change its parameters with a different frequency in timesteps. In other words, multiscale RNNs operates on abstractions that build on one another. This property allows it to resolve some inherent problems of standard RNNs: (a) computational efficiency obtained by updating the high-level layers less frequently, (b) efficiently delivering long-term dependencies with fewer updates at the high-level layers, which mitigates the vanishing gradient problem, (c) flexible resource allocation (e.g., more hidden units to the higher layers that focus on modelling long-term dependencies and less hidden units to the lower layers which are in charge of learning short-term dependencies). In addition, the learned latent hierarchical structures can provide useful information to other downstream tasks such as module structures in computer program learning, sub-task structures in hierarchical reinforcement learning, and story segments in video understanding[3].

3.2.3 LSTMs

As we have already discussed, recurrent neural networks are designed to persist information within each RNN cell to predict outputs based on these information. Recurrent neural networks address this issue by creating networks with loops in them.

[2] https://arxiv.org/pdf/1312.6026.pdf
[3] HIERARCHICAL MULTISCALE RECURRENT NEURAL NETWORKS: https://arxiv.org/pdf/1609.01704.pdf

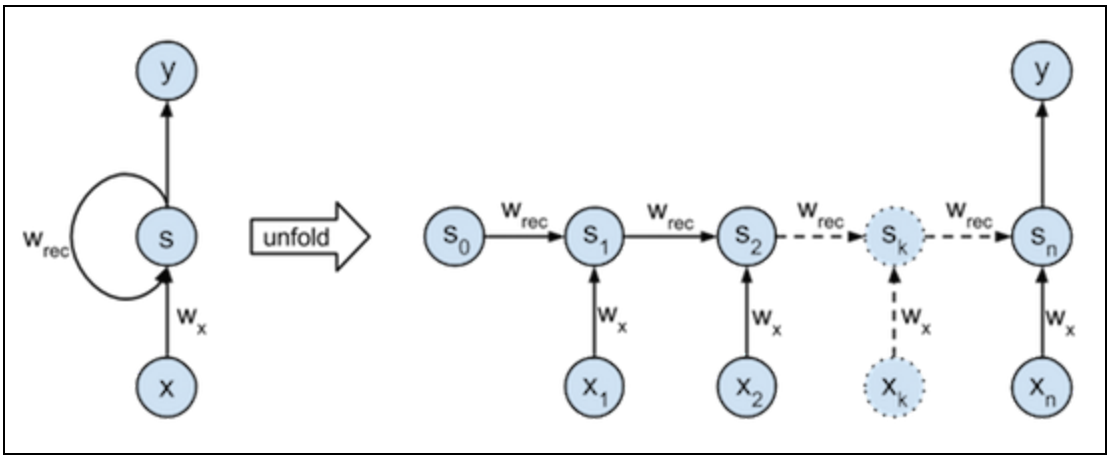


Figure 5: Single RNN cell and its unfolded state
(Source: http://peterroelants.github.io/posts/rnn_implementation_part02/)

But even with this chain like structure specifically designed to contain previous information, the RNNs struggle to create Long term dependencies between the input. This inability greatly limits their applicability in fields like machine translation where the context can be spread across a large amount of data. Consider trying to predict the last word in the text "I grew up in India… I speak fluent ____." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. As this gap grew larger, the RNNs are unable to cope up with it and lose the context of speech/text.

The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult. And later Hochreiter & Schmidhuber (1997) introduced Long Short Term Memory Networks - LSTMs. They were proved to be tremendously successful in a wide variety of works involving long term dependencies.

The main reason behind their success was the internal design of their structure. All RNNs form a chain of layer which is made of a simple structure consisting just an activation function. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

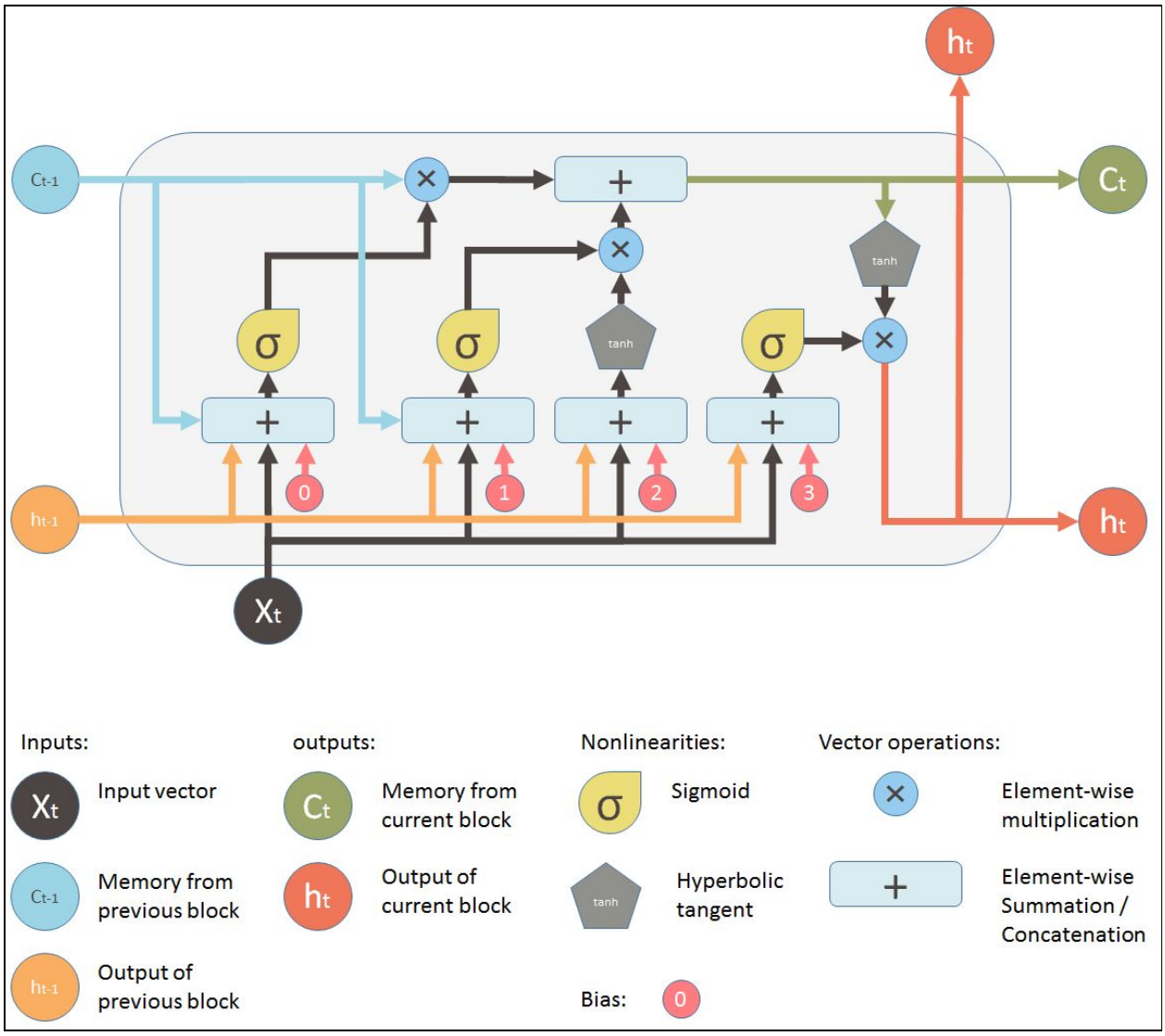


Figure 6: An intuitive representation of what happens inside an LSTM cell
(Source: https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e)

LSTMs adopt gating technique in which each gate, made up of a neural network layer, can control the flow of information through them and decide independently what information is allowed to flow. Although there are many variation of the LSTM models, a general LSTM model contains following four layers:

- Forget Gate Layer

This is the first layer which operates on the input. This layer decides what information is relevant and what information needs to be discarded. It looks at the input and previous cell state and outputs a number between 0 and 1, where a '1' represents completely useful and a '0' means completely get rid of this.

- Input Gate Layer

The next layer decides what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, that could be added to the state. Then we'll combine these two to create an update to the state.

- Output Gate Layer

Finally, we decide what we're going to output. This output gate layer will be based on our cell state. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

As we have already mentioned, not all LSTM networks are same. Many new variations are still evolving and achieving success. And although the differences are slight, some of them are worth mentioning are "The Peephole" variation, "Coupled forget and input gate" and "Gated Recurrent Unit", or GRU.

3.3 Machine Translation by Sequence to Sequence Modeling

Before the advent of the all-powerful machine learning based computations, the traditional phrase-based translation systems used to break up source sentences into multiple chunks and then performed translation on these batches. This led to several problems including disfluency in the translated outputs and didn't seem natural at all[4].

Unlike this traditional approach, NMT mimics the human approach of translation. We read the entire source sentence, understand its meaning, and then produce a translation. Similarly, the NMT model first reads the source sentence using an encoder to build a "thought" vector, a sequence of numbers that represents the sentence meaning; a decoder, then, processes the sentence vector to emit a translation. The sequence-to-sequence models are based on similar approach.

[4] Neural Machine Translation (seq2seq) Tutorial: https://github.com/tensorflow/nmt

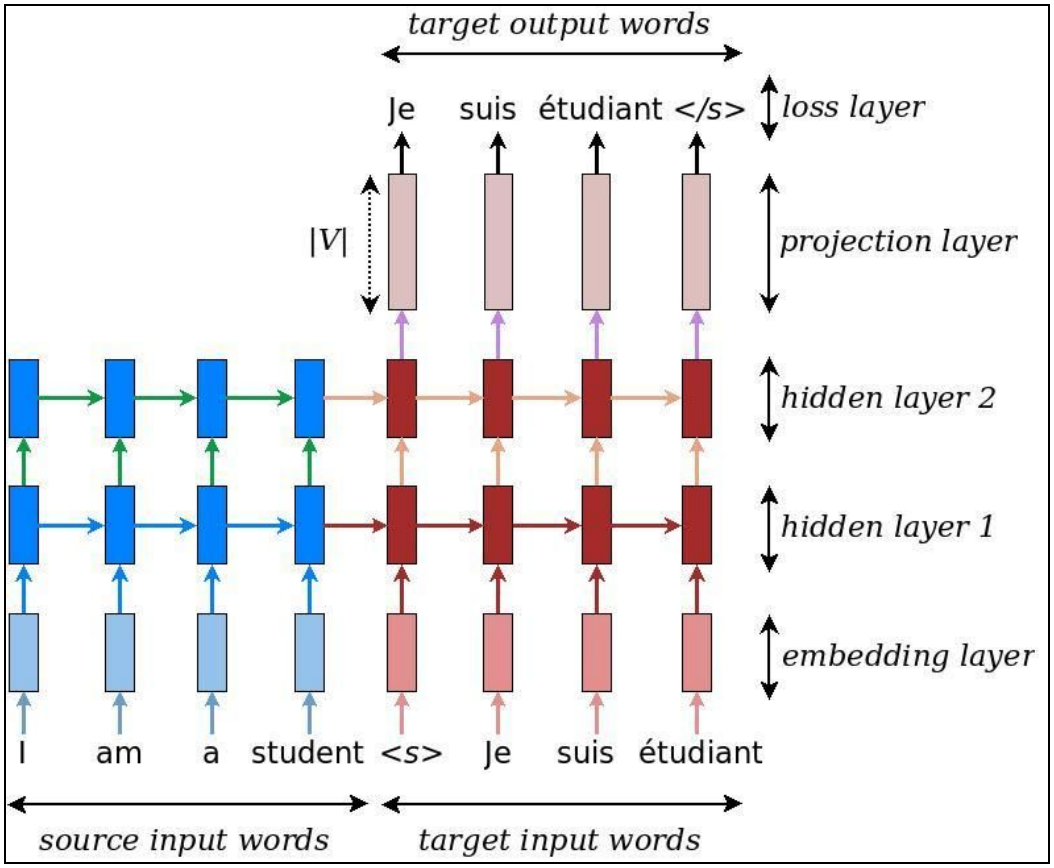


Figure 7: Neural machine translation – example of a deep recurrent architecture proposed by for translating a source sentence "I am a student" into a target sentence "Je suis étudiant". Here, "<s>" marks the start of the decoding process while "</s>" tells the decoder to stop.
(Source: https://github.com/tensorflow/nmt)

Sequence-to-sequence (seq2seq) models have enjoyed great success in variety of applications like, speech recognition, text summarisation, along with machine translation. However, Neural Machine Translation (NMT) was the very testbed for the success of seq2seq models.

The seq2seq models work in a very similar fashion as we explained earlier. A seq2seq model based NMT system first reads the source sentence using an encoder to build a "thought" vector, a sequence of numbers that represents the sentence meaning; a decoder, then, processes the sentence vector to emit a translation, as illustrated in Figure 101010. This is often referred to as the encoder-decoder architecture. In this manner, NMT addresses the local translation problem in the traditional phrase-based approach: it can capture long-range dependencies in languages, e.g., gender agreements; syntax structures; etc., and produce much more fluent translations as demonstrated by Google Neural Machine Translation systems.

At a high level, the NMT model consists of two recurrent neural networks: the encoder RNN simply consumes the input source words without making any prediction; the decoder, on the other hand, processes the target sentence while predicting the next words.

3.4 FS-RNN

The Fast Slow Recurrent Neural Network, FS-RNN was first introduced in "Fast-Slow Recurrent Neural Networks" by A. Mujika et al in 2017. For the FS-RNN architecture, the basic building building block is a Recurrent Neural Network. Because we will be addressing the problem of language translation, a good RNN to incorporate in our architecture would be an LSTM RNN (Long Short Term Memory RNN). Following is the reason why:
LSTM addresses the vanishing gradient problem of simple RNNs. So, it is able to maintain long term contexts which are a common occurence in text data.

So, now that our FS-RNN architecture has been converted to an FS-LSTM one, it is time to explain what the "FS" in the architecture means.

3.4.1 Model Overview

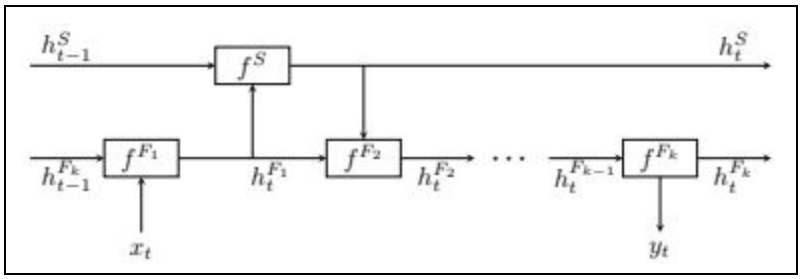


Figure 8: Diagram of a Fast-Slow RNN with k Fast cells
(Source: Cited research paper)

The diagram above consists of k sequentially connected RNN cells $F_1, \ldots, F_k$ on the lower hierarchical layer and one RNN cell S on the higher hierarchical layer. We call $F_1, \ldots, F_k$ the **Fast cells**, S the **Slow cell** and the corresponding hierarchical layers the **Fast and Slow layer**, respectively. S receives input from F_1 and feeds its state to F_2. F_1 receives the sequential input data x_t, and F_k outputs the predicted probability distribution yt for the next element of the sequence from its vocabulary.

This approach performs better than the traditional RNN architectures because of the following reasons:

- The Slow cell captures long-term dependencies.
- The Fast cell only stores short-term information and when multiple Fast cells are stacked one over the other, the gradients decrease from the top layer to the bottom layer thus helping learn complex, non-linear transition functions between the input and its output as we go deeper.

Once the FS-RNN architecture based cell is created, it can be used in any RNN-based model as its internal, individual learning unit.

3.4.2 Results Claimed

In the paper by A. Mujika et al, the FS-RNN model was evaluated on two character level language modeling datasets, namely Penn Treebank and Hutter Prize Wikipedia, which will be referred to as enwik8 in this section. The task was to predict next word given all previous ones.

In their paper, they claimed that the FS-LSTM (a variation of FS-RNN where the RNN cell is a LSTM cell) achieves 1.19 BPC and 1.25 BPC on the Penn Treebank and enwik8 data sets, respectively. These results are compared to other approaches in Table 1 and Table 2[5].

For the Penn Treebank, the FS-LSTM outperforms all previous approaches with significantly less parameters than the previous top approaches. In the enwik8 data set, the FS-LSTM surpasses all other neural approaches.

Table 1: BPC on Penn Treebank

Model	BPC	Param Count
Zoneout LSTM [2]	1.27	-
2-Layers LSTM	1.243	6.6M
HM-LSTM [6]	1.24	-
HyperLSTM - small [15]	1.233	5.1M
HyperLSTM [15]	1.219	14.4M
NASCell - small [44]	1.228	6.6M
NASCell [44]	1.214	16.3M
FS-LSTM-2 (ours)	1.190	7.2M
FS-LSTM-4 (ours)	1.193	6.5M

Table 2: BPC on *enwik8*

Model	BPC	Param Count
LSTM, 2000 units	1.461	18M
Layer Norm LSTM, 1800 units	1.402	14M
HyperLSTM [15]	1.340	27M
HM-LSTM [6]	1.32	35M
Surprisal-driven Zoneout [33]	1.31	64M
ByteNet [22]	1.31	-
RHN - depth 5 [43]	1.31	23M
RHN - depth 10 [43]	1.30	21M
Large RHN - depth 10 [43]	1.27	46M
FS-LSTM-2 (ours)	1.290	27M
FS-LSTM-4 (ours)	1.277	27M
Large FS-LSTM-4 (ours)	1.245	47M
2 × Large FS-LSTM-4 (ours)	1.198	2 × 47M
cmix v13 [24]	1.225	-

[5] Fast Slow RNN by A. Mujika et. al., NIPS 2017

4. Methodology

4.1 Datasets used

To verify the claims made in the paper, we first decided to use one of the datasets mentioned in the paper, Penn Treebank dataset. The PTB dataset contains over one million words of 1989 Wall Street Journal material annotated in Treebank II style.[6]

Since our project heavily uses the NMT architecture by Google for machine translation, we also used the already cleaned dataset of Ted Talks made available by Stanford[7].

Since it did not support Indian languages, we switched to tab-delimited bilingual sentence pairs made available at Anki[8].

4.2 Technologies used

For developing the various traditional neural networks, we used **Keras**. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.

But for developing the code for the novel architecture proposed in the paper, we needed to develop a completely new code for the architecture. High level libraries like Keras are not yet powerful enough to provide that feature. So, we switched to **Tensorflow**, which is a set of core libraries which work on n-dimensional arrays called tensors and provide all the core modules necessary for implementing new architectures.

The programming language used was **python**.

Libraries including **numpy, pandas** and **sklearn** have also been employed to perform mathematical computations and transformations on the initial data.

[6] https://catalog.ldc.upenn.edu/ldc99t42
[7] https://nlp.stanford.edu/projects/nmt/
[8] http://www.manythings.org/anki/

4.3 Comparing with other neural networks

For comparing the performance of Fast Slow Recurrent Neural Network, we developed our own implementation of three different types of neural networks:

- Feed forward neural network
- Long Short Term Memory based Recurrent Neural Network
- Fast Slow Recurrent Neural Network with LSTM cell

Same set of configurations were used in all the models. The configuration is as follows:

- Learning Rate = 0.02
- Dropout Rate = 0.45
- Number of units in an LSTM cell (both LSTM and FS-LSTM) = 256
- Number of hidden layers in feed forward neural network = 256
- Number of LSTM layers in the fast cell = 3
- Total number of epochs = 30
- Batch size = 64

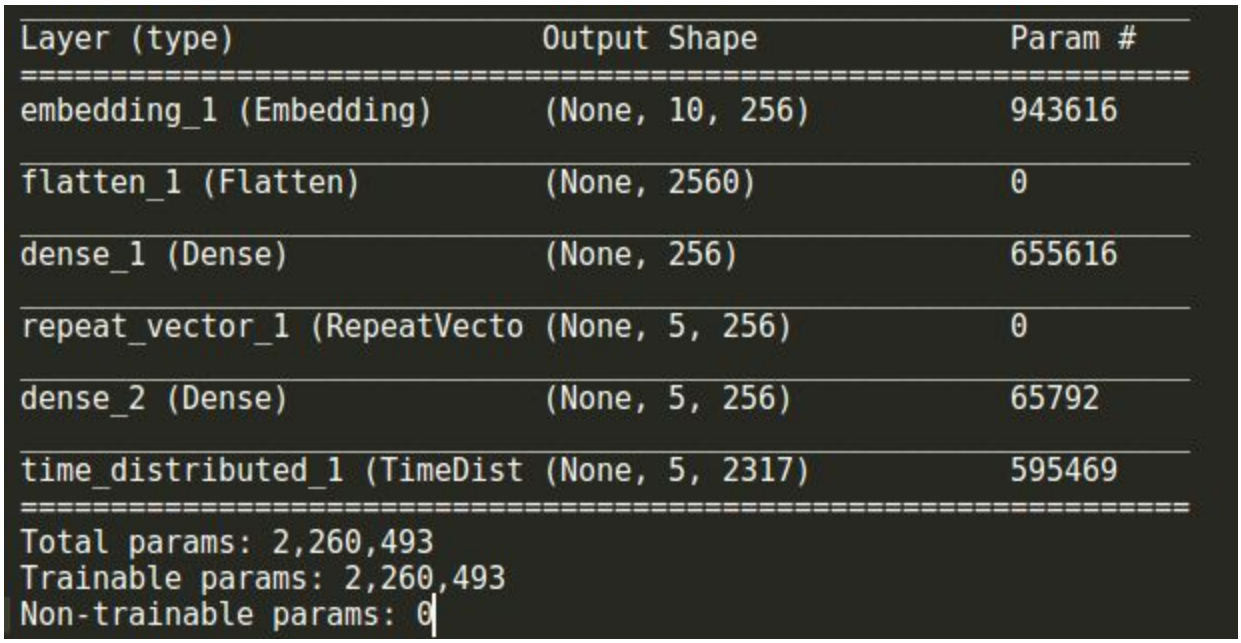


Figure 9: Above is the configuration of each layer of Feed-forward neural network

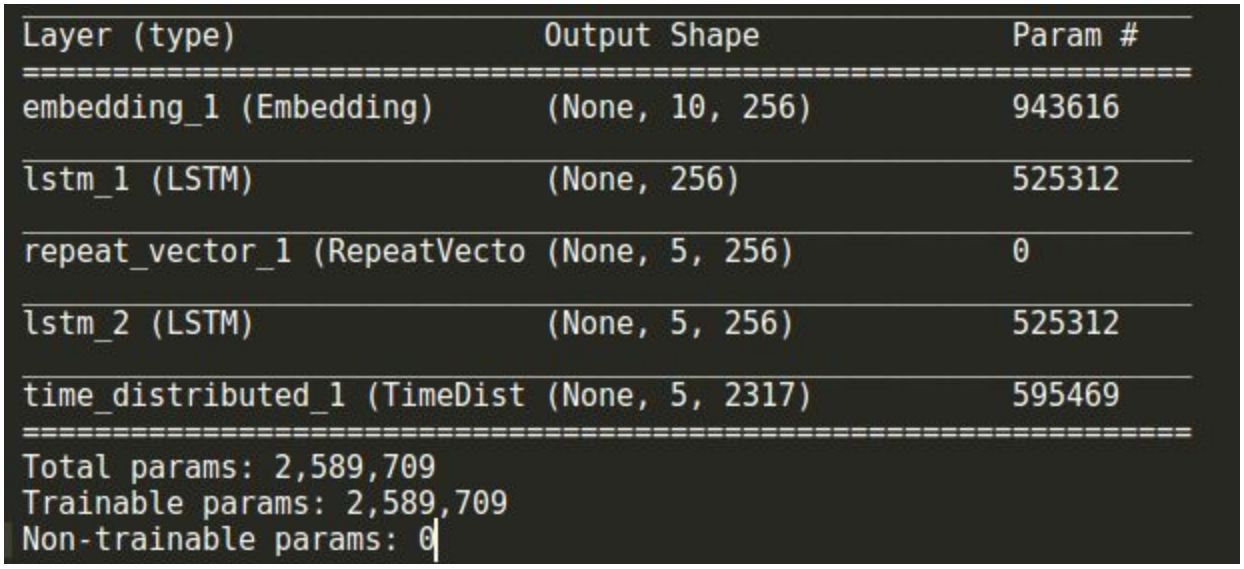


Figure 10: Above is the configuration of each layer of LSTM based neural network

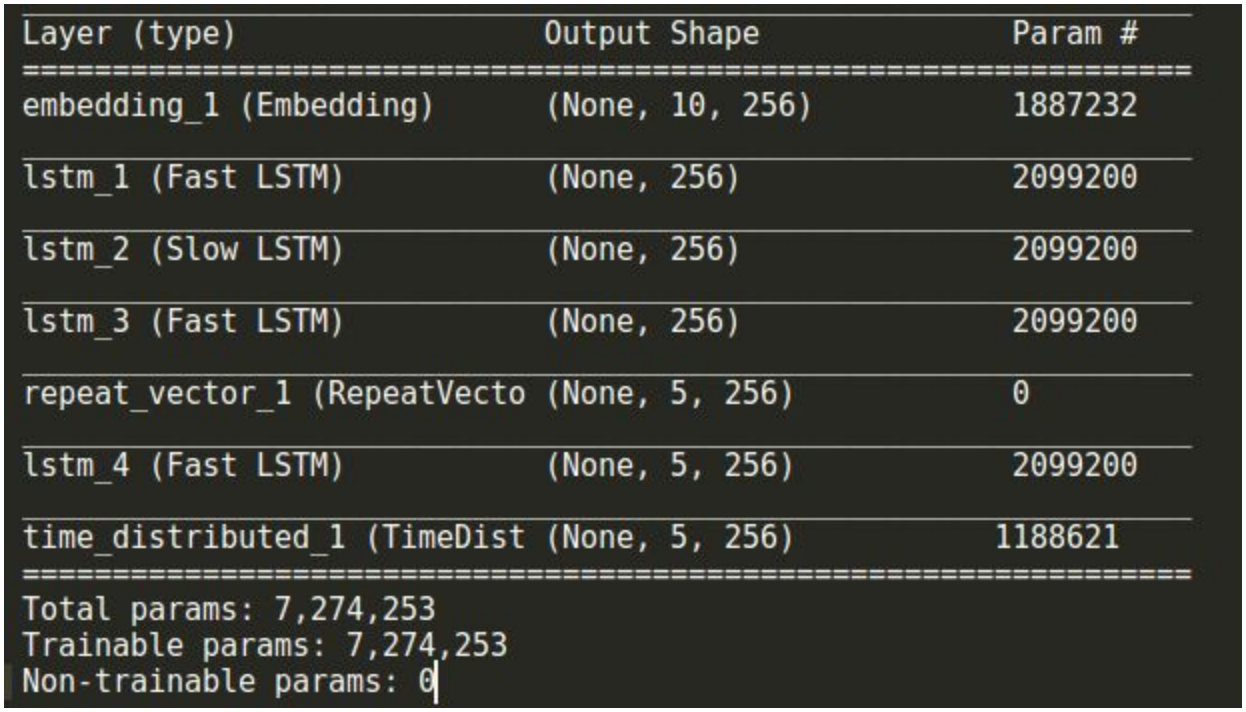


Figure 11: Above is the configuration of each layer of Fast Slow Recurrent neural network

Epoch 1/30

ANN's performance

2018-05-02 01:39:01.
13s - loss: 4.4781 - val_loss: 4.1205

Epoch 00001: val_loss improved from inf to **4.12048**

LSTM's performance

2018-05-01 23:26:40.
25s - loss: 4.3426 - val_loss: 3.5780

Epoch 00001: val_loss improved from inf to **3.57804**

FSRNN's performance

2018-05-02 04:05:39.
64s - loss: 4.0558 - val_loss: 3.4736

Epoch 00001: val_loss improved from inf to **3.47362**

val_loss is the value of cost function for our cross validation data and **loss** is the value of cost function for our training data. On validation data, neurons using dropout do not drop random neurons. The reason is that during training we use dropout in order to add some noise for avoiding overfitting.

As is evident from the values above, the feed forward neural network starts off slow with a big value_loss of 4.12048. This is because feed forward neural networks have a tough time handling sequential data as they treat each time step independently.

In case of LSTM, the val_loss saw an improvement of val_loss from infinity to 3.57804 which is a much better value than the feed forward neural network.

In case of FS-LSTM, since we are using LSTM as the internal cell, and we have three LSTM cells in the Fast layer, it should have shown an increase in accuracy from the start. And this can be clearly seen from the data above where FSRNN improved the val_loss to 3.47362 by the end of first iteration itself.

Epoch 30/30

ANN's performance

2018-05-02 01:39:01.
- 13s - loss: 1.4944 - val_loss: 3.0154
Epoch 00030: val_loss did not improve
The ANN had stopped improving after the 13th iteration where its best accuracy was:

17s - loss: 1.7916 - val_loss: 2.9539

Epoch 00013: val_loss improved from 2.97558 to **2.95392**

LSTM's performance

2018-05-01 23:26:40.
20s - loss: 0.4344 - val_loss: 1.9499

Epoch 00030: val_loss improved from 1.95398 to **1.94993**

FSRNN's performance

2018-05-02 04:05:39.
58s - loss: 0.2946 - val_loss: 1.9233

Epoch 00023: val_loss improved from 1.92972 to **1.92330**

After the 30 (& 23 in case of FSRNN) epochs are complete, LSTM and FSRNN come quite close to one another on the validation set with FSRNN leading by a mere 0.02663.

But on the training set, FS-RNN performs much better than the other two with just 0.2946 loss value.

Another important point to note is that FSRNN not only performs more accurately on the training set, but it is also better equipped at handling more complex mapping between the input and the output. So, in case of new sequence of a type it has not seen before, FSRNN will perform much better than LSTM.

4.4 Neural Machine Translation (seq2seq) model[9]

Here, I will explain some code snippets of the implementation of this model. After that we will discuss some of the common scoring standards for Neural Machine Translation, or in fact any translation in NLP.

Defining some hyperparameters of the model

In [4]:
```
vocab_size= 50000
num_units = 128
input_size = 128
batch_size = 16
source_sequence_length=40
target_sequence_length=60
decoder_type = 'basic' # could be basic or attention
sentences_to_read = 50000
```

In machine learning, a hyperparameter is a parameter whose value is set before the learning process begins. By contrast, the values of other parameters are derived via training. These are the parameters which defines the characteristics of the model, not the properties of the data.

Let's analyse some statistics

In [7]:
```
def split_to_tokens(sent,is_source):
    #sent = sent.replace('-',' ')
    sent = sent.replace(',',' ,')
    sent = sent.replace('.',' .')
    sent = sent.replace('\n',' ')

    sent_toks = sent.split(' ')
    for t_i, tok in enumerate(sent_toks):
        if is_source:
            if tok not in src_dictionary.keys():
                sent_toks[t_i] = '<unk>'
        else:
            if tok not in tgt_dictionary.keys():
                sent_toks[t_i] = '<unk>'
    return sent_toks

# Let us first look at some statistics of the sentences
source_len = []
source_mean, source_std = 0,0
for sent in source_sent:
    source_len.append(len(split_to_tokens(sent,True)))

print('(Source) Sentence mean length: ', np.mean(source_len))
print('(Source) Sentence stddev length: ', np.std(source_len))

target_len = []
target_mean, target_std = 0,0
for sent in target_sent:
    target_len.append(len(split_to_tokens(sent,False)))

print('(Target) Sentence mean length: ', np.mean(target_len))
print('(Target) Sentence stddev length: ', np.std(target_len))
```

```
(Source) Sentence mean length:  26.35934
(Source) Sentence stddev length:  13.9681614669
(Target) Sentence mean length:  28.58758
(Target) Sentence stddev length:  15.1544201388
```

[9] This neural machine translation model was developed while refering to this blog post: http://www.thushv.com/natural_language_processing/neural-machine-translator-with-50-lines-of-code-using-tensorflow-seq2seq/

Here, we first tokenized the input text into a bunch of words. Tokenization is the process of breaking up the given text into units called tokens. The tokens may be words or number or punctuation mark.

Inputs Outputs Masks

In [17]:
```
tf.reset_default_graph()

enc_train_inputs = []
dec_train_inputs = []

# Need to use pre-trained word embeddings
encoder_emb_layer = tf.convert_to_tensor(np.load('de-embeddings.npy'))
decoder_emb_layer = tf.convert_to_tensor(np.load('en-embeddings.npy'))

# Defining unrolled training inputs
for ui in range(source_sequence_length):
    enc_train_inputs.append(tf.placeholder(tf.int32, shape=[batch_size],name='enc_train_inputs_
%d'%ui))

dec_train_labels=[]
dec_label_masks = []
for ui in range(target_sequence_length):
    dec_train_inputs.append(tf.placeholder(tf.int32, shape=[batch_size],name='dec_train_inputs_
%d'%ui))
    dec_train_labels.append(tf.placeholder(tf.int32, shape=[batch_size],name='dec-train_outputs
_%d'%ui))
    dec_label_masks.append(tf.placeholder(tf.float32, shape=[batch_size],name='dec-label_masks_
%d'%ui))

encoder_emb_inp = [tf.nn.embedding_lookup(encoder_emb_layer, src) for src in enc_train_inputs]
encoder_emb_inp = tf.stack(encoder_emb_inp)

decoder_emb_inp = [tf.nn.embedding_lookup(decoder_emb_layer, src) for src in dec_train_inputs]
decoder_emb_inp = tf.stack(decoder_emb_inp)

enc_train_inp_lengths = tf.placeholder(tf.int32, shape=[batch_size],name='train_input_lengths')
dec_train_inp_lengths = tf.placeholder(tf.int32, shape=[batch_size],name='train_output_lengths'
)
```

Here we define the input, output variables. The input words are sent through an embedding layer to obtain embeddings for each word in both source and target sentences. The final inputs fed in to the encoder and decoder will be for following sizes.

- encoder_emb_inp : [source_sequence_length, batch_size, embedding_size]
- decoder_emb_inp: [target_sequence_length, batch_size, embedding_size]

Encoder

In [18]:
```
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

initial_state = encoder_cell.zero_state(batch_size, dtype=tf.float32)

encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
    encoder_cell, encoder_emb_inp, initial_state=initial_state,
    sequence_length=enc_train_inp_lengths,
    time_major=True, swap_memory=True)
```

Essentially we are just using a single LSTM cell with zero-initialized states as the encoder.

tf.nn.dynamic_rnn is a wonderful function introduced by seq2seq library that can handle arbitrary length inputs. And the length of each input is provided to it via *sequence_lengthkeyword*.

Decoder

In [19]:
```
# Build RNN cell
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

projection_layer = Dense(units=vocab_size, use_bias=True)

# Helper
helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, [tgt_max_sent_length-1 for _ in range(batch_size)], time_major=True)

# Decoder
if decoder_type == 'basic':
    decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)

elif decoder_type == 'attention':
    decoder = tf.contrib.seq2seq.BahdanauAttention(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)

# Dynamic decoding
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, output_time_major=True,
    swap_memory=True
)
```

There are a bit of lines more than the encoder, because the Decoder uses a softmax layer to produce predictions belonging to the target language. Concisely the above snippet is saying. Define a decoder that uses a LSTM cell, encoder state as its initial state and a softmax layer as the final output layer. Then we use *tf.contrib.seq2seq.dynamic_decode* function to decode the predictions over time. Here, a question comes across our minds:

- Why do we need the last encoder state as the first state of the decoder?

This is the single link that's responsible for the communication between the encoder and the decoder (the arrow connecting the encoder and the decoder in the above figure). In other words, this last encoder state provides the context for the decoder in terms of what the translated prediction should be about. The last state of the encoder can be interpreted as a "language-neutral" thought vector.

Loss and Predictions

In [20]:
```
logits = outputs.rnn_output

crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=dec_train_labels, logits=logits)
loss = (tf.reduce_sum(crossent*tf.stack(dec_label_masks)) / (batch_size*target_sequence_length
))

train_prediction = outputs.sample_id
```

Here we use the *tf.nn.sparse_softmax_cross_entropy_with_logits* function to parse softmax cross entropy between logits and labels.

Defining Optimizer with Gradient Clipping

In [21]:
```
print('Defining Optimizer')
# Adam Optimizer. And gradient clipping.
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)
learning_rate = tf.train.exponential_decay(
    0.01, global_step, decay_steps=10, decay_rate=0.9, staircase=True)

with tf.variable_scope('Adam'):
    adam_optimizer = tf.train.AdamOptimizer(learning_rate)

adam_gradients, v = zip(*adam_optimizer.compute_gradients(loss))
adam_gradients, _ = tf.clip_by_global_norm(adam_gradients, 25.0)
adam_optimize = adam_optimizer.apply_gradients(zip(adam_gradients, v))

with tf.variable_scope('SGD'):
    sgd_optimizer = tf.train.GradientDescentOptimizer(learning_rate)

sgd_gradients, v = zip(*sgd_optimizer.compute_gradients(loss))
sgd_gradients, _ = tf.clip_by_global_norm(sgd_gradients, 25.0)
sgd_optimize = sgd_optimizer.apply_gradients(zip(sgd_gradients, v))

sess = tf.InteractiveSession()
```

```
Defining Optimizer
```

We use the Adam optimizer initially (for example, for the first 10000 iterations) and then switch to a Stochastic Gradient Descent later. This is because using Adam continuously gave some unexplained results. Also, we use Gradient clipping to avoid gradient explosion.

Now, since we know what is happening in the code, let's see the results. We measure the model based on two factors, BLEU score and Perplexity score. These results are based on English to Vietnamese dataset.

BLEU Score: The Bilingual Evaluation Understudy Score, or BLEU for short, is a metric for evaluating a generated sentence to a reference sentence. "The closer a machine translation is to a professional human translation, the better it is" – this is the central idea behind BLEU. BLEU was one of the first metrics to claim a high correlation with human judgements of quality, and

remains one of the most popular automated and inexpensive metrics[10].

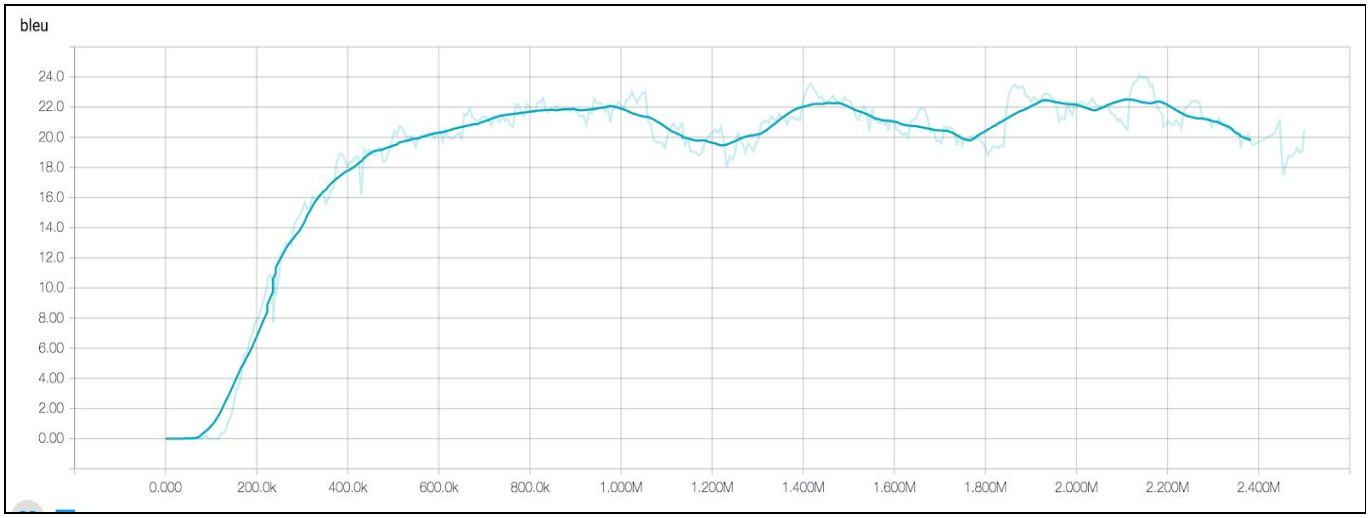


Figure 12: BLEU score during training of the NMT model for Vietnamese to English

Perplexity: Perplexity refers to the log-averaged inverse probability on unseen data. A low perplexity score indicates a good NMT model, but it does not necessarily mean that it will work equally fine in real world application.

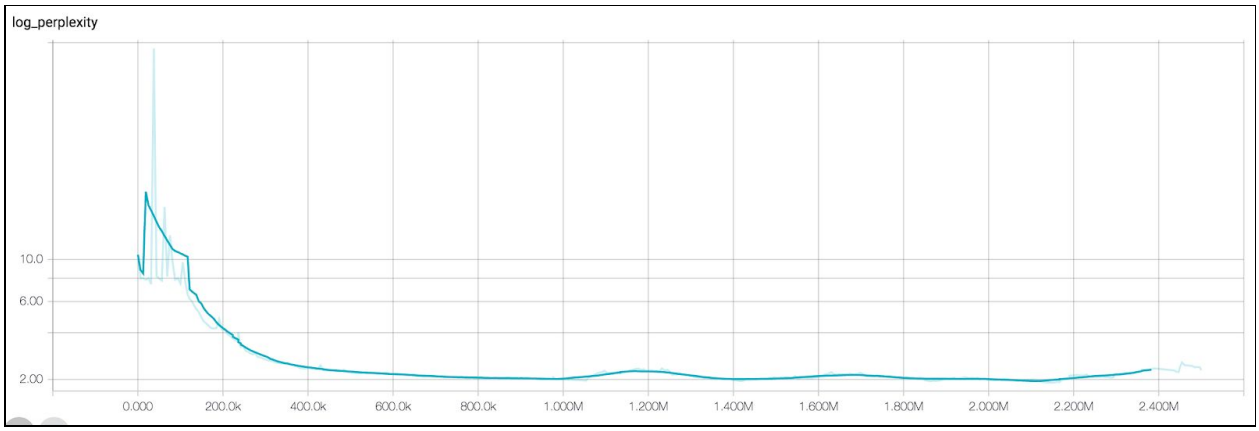


Figure 13: Perplexity score during training of the NMT model for Vietnamese to English

[10] Wikipedia article on BLEU: https://en.wikipedia.org/wiki/BLEU

5. Conclusion

While the Fast Slow Recurrent Neural Network performed much better than LSTM based recurrent neural network, it took thrice as much time to perform computations over each epoch. On further testing with different types of input sequences, FSRNN, again performed exceptionally well.

Hence, it can be said that this novel architecture which does indeed provide the best of both worlds by making the RNN cells deep instead of shallow, while simultaneously providing the adaptive properties of Multiscale RNNs. So, if better accuracy and the ability to handle diversified data is what one wants, FSRNN is the way to go.

But the fact that it takes much more time than the LSTM cell of the same configuration calls for more research and improvements into its configuration and architecture.

6. Future Prospects

Currently, the various neural models implemented as part of this project run only for 30 epochs which is a very insignificant amount to properly analyse the efficiency and accuracy of the training. So, these models need to be trained on bigger datasets and much more powerful GPU-based systems to properly assess their accuracy.

Also, our NMT based seq2seq adaption did not have attention model integrated with it. Currently, all state-of-the-art results in sequence based translation problems have been achieved using Attention models. So, integrating it with our FS-RNN based NMT model should theoretically give much better and more adaptive results than the current state-of-the-art.

Also, the code for FS-RNN is still under active development. Many optimizations need to be made and the hyperparameters need to be well adjusted so that the full ability of this novel approach may be appreciated.

7. References

1.) Asier Mujika, et al (2017). “Fast-Slow Recurrent Neural Networks”, NIPS Journal

2.) Yonghui Wu, et al (2016). “Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”, arxiv.org

3.) Ilya Sutskever, et al (2014). “Sequence to Sequence Learning with Neural Networks”, NIPS Journal

4.) Razvan Pascanu, et al (2013). “How to Construct Deep Recurrent Neural Networks”, arxiv.org

5.) Junyoung Chung, et al (2017). “Hierarchical Multiscale Recurrent Neural Networks”, arxiv.org

6.) Marcus, Mitchell, et al (1999). “Treebank-3 LDC99T42”, Philadelphia: Linguistic Data Consortium

7.) Trang Ho (2010). “Tab-delimited Bilingual Sentence Pairs”

8.) http://www.thushv.com/natural_language_processing/neural-machine-translator-with-50-lines-of-code-using-tensorflow-seq2seq/