Fast Weights - The First Transformer Variants?

1st Semester of 2023-2024

Bratan Radu-George
radu-george.bratan@s.unibuc.ro

Cristea Eduard-Gabriel
eduard-gabriel.cristea@s.unibuc.ro

Abstract

Abstract. This document aims to explain how fast weights work as well as how they can be applied to a sequence modelling problem (implemented in Python), such as generating predictions based on a large text corpus. Section 1 will offer a short summary of the paper along with what our code is supposed to accomplish, section 2 will dive into the inner workings of our models, section 3 will outline all the limitations that keep our implementation from being ideal, while the last section, 4, adds references to the original authors of all the works we used (including the YouTube videos we were inspired by - NeuralNine (2023), Science (2023), Team (2021)).

1 Introduction

Following the recent explosion in popularity of LLMs such as ChatGPT and Microsoft Copilot, we decided to create a simple text completion/prediction model of our own by using the "fast weights" approach in combination with a sequence modeling problem.

The primary problem we are addressing in our project is the challenge of sequence modeling in the context of text prediction. Sequence modeling refers to the task of predicting the next element in a sequence, given the previous elements. In the case of text prediction, the goal is to forecast the next word or character in a sentence based on the preceding text. This is a fundamental problem in natural language processing (NLP) with applications in areas like machine translation, speech recognition, and text auto-completion.

In the paper Schmidhuber (1992), the author introduces a novel approach to sequence modeling using a mechanism called "fast weights". This method enhances the abilities of neural networks by allowing them to rapidly alter the connection strengths (weights) between neurons within the network. This dynamic adjustment of weights provides the network with a short-term memory capability, enabling it to better handle temporal sequences and dependencies that are vital for tasks like text prediction.

The paper outlines a theoretical framework for fast weights and demonstrates their potential through various experiments. The approach is shown to be effective in enabling networks to remember and utilize recent information, which is crucial for making accurate predictions in sequence modeling.

Other research on fast weights and sequence modeling includes:

- Ba et al. (2016) which further explores the concept of fast weights, showing how they can be applied to recurrent neural networks to improve performance on tasks that require processing of recent information ;
- Zaremba et al. (2015), which discusses various techniques for regularizing recurrent neural networks, including those that could be applied to models using fast weights to improve their generalization.

Previous work in this area includes the development of Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber, 1997), which address the vanishing gradient problem often encountered in sequence modeling and are closely related to the idea of maintaining a form of memory for sequence prediction. Another example is the Transformer model introduced by Vaswani et al. (2023), which uses self-attention mechanisms to weigh the influence of different parts of the input data and has been highly successful in a variety of NLP tasks.

Our approach is comprised of two pieces of code, an LSTM-based model (RNN) and a Transformers-based model (FNN, similar to Fast Weights), both of them meant to act as text generation models trained to predict the next word based on three previous words. They use a text file comprised of various sentences written in the English language which are then split and organized into sequences.

The first approach sounds simple, but in reality it's complicated enough that we couldn't quite grasp every concept used in its creation. For example, we weren't able to understand how tokenization or one-hot encoding works. We did, however, expand our knowledge of model architecture by learning about Embedding layers and LSTM layers, as well as deepen our understanding of activation functions, loss functions, optimizers ant the Keras API.

Similar difficulties and revelations were encountered in the second approach, which uses an FNN instead of an RNN.

When it comes to contributions, Radu did the following:

- researched all the provided papers;
- created a summary of the papers;
- wrote some of the detailed explanation of the approach;
- optimized the models;

while Eduard:

- researched Schmidhuber (1992);
- researched additional papers;
- created a summary of the additional papers;
- wrote some of the detailed explanation of the approach;
- trained the models.

2 Approach

Our code is stored on this GitHub repository.

The first implementation was written in the Python language, using the TensorFlow platform, the Keras API and Kaggle as our notebook. The model was created using Embedding layers, LSTM layers and Dense layers, provided by Keras. Training for the model took between 5 hours with the accurate model and 20 minutes with the fast model.

The second implementation took a similar approach and had similar results, albeit we decided to skip the accurate implementation for the sake of saving time.

The following encapsulates the end-to-end process of training both of our models for text generation, from initial data handling to interactive prediction:

1. **Data preprocessing.** We used a text file comprised of multiple natural speech phrases taken from various sources such as books, articles, internet comments, blogs, etc. The file is then preprocessed to remove special characters and unwanted whitespace.

2. **Tokenization.** The preprocessed text data was then tokenized using Keras's Tokenizer class, an essential step for preparing the data because raw text cannot be directly processed.

3. **Creating Sequences.** After tokenization, the data was organized into sequences. Each sequence consisted of a fixed number of words and the corresponding label was the next word in the text. These sequences were then converted into NumPy arrays for the model to train on.

4. **One-Hot Encoding.** The labels (next words) were then one-hot encoded.

5. **Model Training.** The model was compiled with the categorical cross-entropy loss function and the Adam optimizer. Training involved fitting the model to the data, using model checkpoints to save the model with the best performance.

6. **Prediction.** After training, the model could predict the next five words based on a given input sequence. The prediction function took the last three words of the input, tokenized them, and fed them into the model, which then outputted the most likely next words.

7. **User Interaction.** Finally, the model was set up to take user input for generating text predictions. Users could input a three-word

phrase, and the model would predict the next word. If the user entered "0", the program ended.

The last key point of the process are the architectures of the models themselves, which will be compared here:

1. **LSTM Model Architecture.** We built a Sequential LSTM model using Keras, adding an Embedding layer, two LSTM layers, and two Dense layers, including one with a softmax activation function to output a probability distribution over the vocabulary for the next word.

2. **Transfomers Model Architecture.** We built a Transformers-based model using Keras using similar approaches, adding two Conv1D layers, one with a relu activation function, two Dense layers including one with a softmax activation function, in order to output similar results.

Considering the results, we report both models were able to consistently predict the next five words of the sentence. The fast models tried to match the provided training data as close as possible, while the accurate LSTM model provided creative text completions.

The fast LSTM model went from a 7.4280 loss in the first epoch to 2.4175 in the last (40th) epoch. Training took about 13ms/step, a full epoch taking about 24s. The accurate model went from a 7.32514 loss in the first epoch to 0.5997 in the last (31st/70) epoch. Training took about 392ms/step, a full epoch taking about 351s. The accurate model was not fully trained for the purpose of this implementation due to time constraints.

The fast Transformers model went from a 7.3006 loss in the first epoch to a 2.8562 loss in the last (20th) epoch. Training took about 26ms/step, a full epoch taking about 45s.

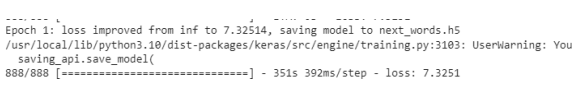

Figure 1: First epoch of the accurate LSMT model

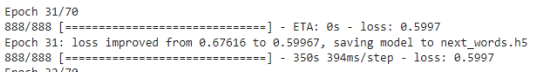

Figure 2: Last epoch of the accurate LSMT model

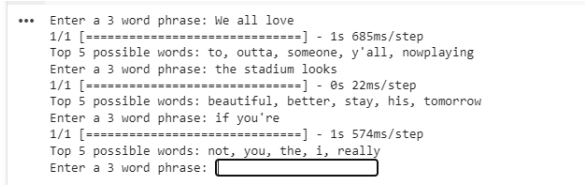

Figure 3: Example output 1 of the accurate LSMT model

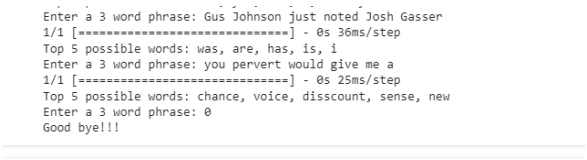

Figure 4: Example output 2 of the accurate LSMT model

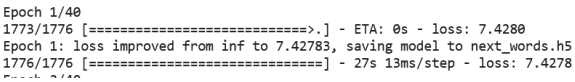

Figure 5: First epoch of the fast LSMT model

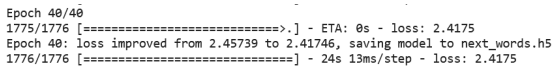

Figure 6: Last epoch of the fast LSMT model

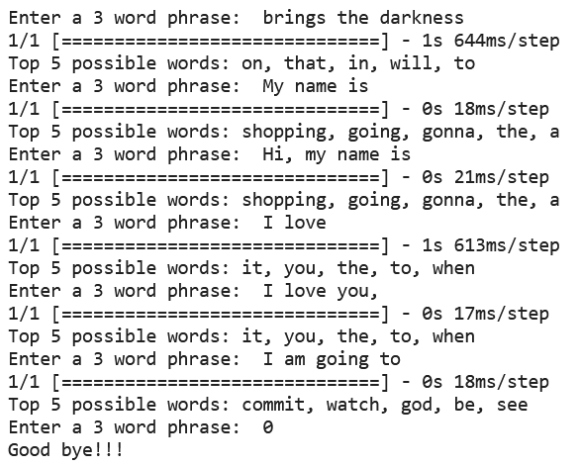

Figure 7: Example output 1 of the fast LSMT model

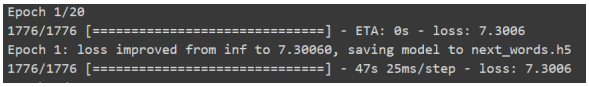

Figure 8: First epoch of the fast Transformer model

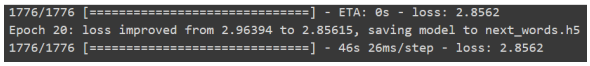

Figure 9: Last epoch of the fast Transformer model

3 Limitations

While those text generation models offer powerful capabilities, there are several limitations that may affect the practical application and scalability of such models:

- **Computational Resources.** Training such models, especially on large datasets, requires significant computational resources. This includes high-performance GPUs capable of handling large matrix operations. Given our limited resources, training times for accurate models are excessively long, taking potentially days or even weeks for very large datasets.

- **Time Constraints.** As mentioned above, training models is a time-consuming process. Due to our specific configuration of the models, it was not feasible to wait for the model to fully train within a reasonable time frame.

- **Difficulty in Understanding and Tuning.** Those are complex models with many hyper parameters to tune, such as the number of hidden units, learning rate, batch size, etc. Finding the optimal configuration was challenging and would definitely require further extensive experimentation.

- **Memory Usage.** The LSTM model specifically consumes a lot of memory during training, which is a limiting factor, especially for cloud environments with limited memory capacity such as Kaggle. On the other hand, the Transformer-based model is inherently designed to be more efficient than RNNs.

- **Overfitting.** Although LSTMs are less prone to overfitting compared to traditional RNNs due to their gating mechanisms, they can still suffer from overfitting, especially when dealing with a small dataset like ours. Overfitting occurs when the model learns the training data too well, capturing noise and anomalies along with the underlying patterns, which leads to poor performance on unseen data.

- **Dependency on Quality Data.** The quality and quantity of the training data significantly impact the performance of our models. The poorly written data we used leads to subpar model performance.

4 Conclusions and Future Work

We really enjoyed this project, we mean it. We're not big fans of creating AI models, but it's really wonderful to dive deep into the history of computer science and create something that *feels* like it has a mind of its own, even if it's just regular old math. Not to mention, now we know how to write a (basic) LaTeX document, which is extremely useful for creating our Bachelor's thesis with.

Now, when it comes to the course itself and its projects, we think that what was accomplished so far (the organization, the teaching, the communication, etc) were pretty good, but our opinions are limited by one factor: we didn't really participate to enough courses/labs to accurately evaluate everything, due to our full time jobs. However, one thing is clear: the difficulty of the course is... perfect - not too hard, not too easy, not too demanding, not too effortless - and we hope it stays that way.

All in all, we hope the future generations will have the same opportunity to take this course as we did, especially those interested in AI. If there's one thing we would change though, that would be the name. Honestly, we had no idea this course was AI related, the theme wasn't immediately obvious like with other subjects.

Acknowledgements

This document has been adapted by Steven Bethard, Ryan Cotterell and Rui Yan from the instructions for earlier ACL and NAACL proceedings, including those for ACL 2019 by Douwe Kiela and Ivan Vulić, NAACL 2019 by Stephanie Lukin and Alla Roskovskaya, ACL 2018 by Shay Cohen, Kevin Gimpel, and Wei Lu, NAACL 2018 by Margaret Mitchell and Stephanie Lukin, BibTeX suggestions for (NA)ACL 2017/2018 from Jason Eisner, ACL 2017 by Dan Gildea and Min-Yen Kan, NAACL

2017 by Margaret Mitchell, ACL 2012 by Maggie Li and Michael White, ACL 2010 by Jing-Shin Chang and Philipp Koehn, ACL 2008 by Johanna D. Moore, Simone Teufel, James Allan, and Sadaoki Furui, ACL 2005 by Hwee Tou Ng and Kemal Oflazer, ACL 2002 by Eugene Charniak and Dekang Lin, and earlier ACL and EACL formats written by several people, including John Chen, Henry S. Thompson and Donald Walker. Additional elements were taken from the formatting instructions of the *International Joint Conference on Artificial Intelligence* and the *Conference on Computer Vision and Pattern Recognition*.

References

Jimmy Ba, Geoffrey Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. 2016. Using fast weights to attend to the recent past.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9:1735–80.

NeuralNine. 2023. Text generation ai - next word prediction in python.

Jürgen Schmidhuber. 1992. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4:131–139.

Unfold Data Science. 2023. Lstm next word prediction in python | lstm python tensorflow | lstm python keras | lstm python.

IG Tech Team. 2021. Project 2: Next word prediction using lstm | complete project with source code.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2015. Recurrent neural network regularization.