NLP using RNNs, LSTMs, and Transformers
Project Report
DD2424 Deep Learning in Data Science

David Alm
20020207-4795
davialm@kth.se

Theofanis Georgakopoulos
20000814-0576
thegeo@kth.se

Antony Zhang
20011231-8159
antonyz@kth.se

Abstract

In this work, we present a comprehensive evaluation of sequence modeling techniques for natural language processing and text generation. We investigate neural text generation using RNNs, LSTMs, and Transformers on Shakespeare's complete works. We first implement a TensorFlow/Keras character-level vanilla RNN baseline. We then develop LSTMs from scratch, that perform better than the baseline RNN. We explore hyperparameter tuning (hidden units, layer depth), regularization (dropout, L2), and character-level data augmentation. Next, we transition to word-level modeling with pretrained GloVe embeddings and compare against Byte Pair Encoding subword tokenization. Finally, we implement a transformer network, employing multi-head self-attention. Our findings highlight the key trade-offs for neural language modeling.

1 Introduction

Natural language processing (NLP) and generation lies at the heart of many modern artificial intelligence applications. Sequence modeling techniques such as recurrent neural networks (RNNs) and their gated variants (*e.g.*, long short-term memory (LSTM)) have historically been the backbone of these systems, yet they often struggle with long-range dependencies and coherent text generation. More recently, Transformer-based architectures have demonstrated superior capability in capturing context across lengthy inputs, leading to significant advances in language modeling.

In this project, we build on the vanilla character-level RNN baseline from Assignment 4 but implemented using `TensorFlow` [1] and `Keras` [2] by developing and evaluating deeper sequence models on Shakespeare's complete works (approximately 5 million characters). Next, we implement LSTMs and compare their performance—both quantitatively and qualitatively with each other and the baseline RNN. We then explore various strategies to improve performance. During text synthesis, we investigate probabilistic sampling methods such as temperature scaling and nucleus (top-p). Next, we upgrade to word-level inputs using pre-trained embeddings (GloVe) and compare against byte-pair encoding (BPE) tokenization. Finally, we implement a Transformer-based model.

2 Related Work

Character-level RNNs first demonstrated the feasibility of neural text generation. Sutskever *et al.* [3] trained one of the first character-level RNN language models. However, vanilla RNNs suffered from unstable training and short memory due to vanishing/exploding gradients. The introduction of the LSTM network by Hochreiter and Schmidhuber [4] addressed these issues. LSTMs enabled much longer effective memory, allowing recurrent models to capture long-range structure in text. For example, Graves [5] showed the promise of LSTM-based models, leading to the popularity of multi-layer LSTM char-RNNs (*e.g.* Karpathy's Char-RNN [6]) as a baseline for text generation.

While character-level models learn language from first principles, they can be slow to train and may struggle to capture higher-level semantic structure. A natural progression was to move to word-level modeling with pre-trained word embeddings. An influential approach is GloVe by Pennington *et al.* [7], which learns word vectors by factorizing a global word co-occurrence matrix. Other works such as word2vec by Mikolov *et al.* [8, 9] are also popular.

A challenge with word-level models is their fixed vocabulary, which cannot gracefully handle rare or unseen words. Subword tokenization techniques address this limitation. A prominent method is Byte Pair Encoding (BPE), introduced by Sennrich *et al.* [10]. BPE starts with characters and iteratively merges the most frequent pairs of symbols, effectively learning a vocabulary of common subword units. Subword tokenization has since become standard in language models.

Beyond input representation improvements, architectural advances have driven rapid progress in text generation. The Transformer architecture introduced by Vaswani *et al.* [11] was a breakthrough that moved beyond recurrence altogether. The transformer uses self-attention mechanisms to model sequence relationships in parallel. Vaswani *et al.* demonstrated that transformers achieved state-of-the-art machine translation results while being faster to train than LSTM-based models.

3 Data

Our experiments are built on The Complete Works of William Shakespeare. Specifically, we shall use the UTF-8 encoded plain text version which can be downloaded from Project Gutenberg. The dataset is large, roughly 5 million characters, and we have made splits into training, testing, and validation data.

Character-level pre-processing For the character-level pre-processing, we numerate the full set of unique characters (letters, digits, punctuation, whitespace, and special symbols) to form a vocabulary of size $K = 106$. Each character is mapped to a zero-based index, and the text is converted into an integer sequence. For sequence modelling, we generate overlapping input–target pairs of length 100 where each input sequence of 100 characters predicts the next character. We use `TensorFlow`'s `tf.data.Dataset` API to (1) slice the entire integer sequence into sequences of length $100 + 1$, (2) map each chunk to an (input, target) pair by shifting by one, (3) shuffle the data with a buffer of 10 000, and (4) batch into mini-batches of size 64.

Word-level pre-processing For the word-level pre-processing to use GloVE Tokenization, we turned the text into lowercase text, removed any punctuation and special characters, in order to split into sequencies and then tokenize into words. We then converted words to integer indices and we then constructed the tokens by creating a vocabulary that maps words to integer indices. We set a `seq_length = 30` and an `embedding_dim = 100` as the dimension of the embedding matrix constrcuted to load pre-trained GloVE vectors from `glove.6B.100d.txt`

Subword (BPE) pre-processing In Byte Pair Encoding Tokenization instead of splitting into words, we split words into frequent subwords set by a `min_frequency` that we set as 2 and a `vocab_size = 5000`. We encoded the text into a sequence of subwords and then prepared the input/target sequences to train, by:

```
sequences_bpe = [ids[i-seq_length:i+1] for i in range(seq_length, len(ids))]
```

4 Methodology and Results

In the present section, we shall describe what we have done and the results, in order. Much of the setup is as was in Assignment 4.

4.1 A Baseline RNN using Tensorflow

First, we implement a baseline RNN using `TensorFlow` and `Keras` on character-level inputs and targets. Essentially, this emulates the architecture explored in Assignment 4 but implemented using `Tensorflow`.

The architecture can be found found in detail in the Appendix, specifically in the code snippet Listing 1. We utilize cross-entropy loss and an Adam optimizer to train for 10 epochs. The training

history is given by Table 1. Finally, we also show some sample text generation that shows that it works, albeit bad. See Listing 2 in the Appendix.

Epoch	1	2	3	4	5	6	7	8	9	10
Train Loss	2.8769	2.0198	1.8911	1.8251	1.7825	1.7535	1.7338	1.7185	1.7037	1.6935
Val Loss	2.0838	1.9462	1.8880	1.8560	1.8350	1.8184	1.8075	1.7972	1.7901	1.7833

Table 1: Training and Validation Loss over Epochs

4.2 Implementing an LSTM

Next, we implement an LSTM without relying on `TensorFlow` and `Keras` for anything but the feeding the data as previously described. Recall that an LSTM is similar to an RNN but it introduces a memory cell state. We omit the precise formulation of an LSTM and the necessary equations for its implementation for brevity. Instead, we refer the reader to e.g. *Dive into Deep Learning* [12] by Zhang *et al.*

For full details on the implementation, we refer to the file `test.py` and specifically the functions `initialize_lstm(...)`, `fp_lstm(...)`, `bp_lstm(...)`, and `train_adam_lstm(...)`. We start by training a one layer LSTM using the following parameters choices:

```
params = {'eta': 0.001, 'num_epochs': 1, 'beta1': 0.9, 'beta2': 0.999, 'eps': 1e-8, 'l2': 1e-4,
↪  'dropout': 0.2}
```

Our first LSTM was trained for one epoch resulting in a train_loss = 2.4543 and a val_loss = 2.1108 after sufficient regularization applied, described below in section 4.4 where we also look at more layers and varying parameters. We generate some sample text:

```
ROMEO.
I glavooy sinst
Efart's to pran eawivill.

SVhe be levient thecomsbe flook of.

Aten And have manday, kent, is nolok.
The is onde sto han se to preeand theus soon. With and,
 nyour seam line toI his,
And win, 'll ther Bacuere
Forer sawe bouct teailorve as noth weas'd bady.
Whe soned dead dam rhal
```

4.3 Comparing simple RNN with LSTM

We observe a higher loss in both training and validation for the LSTM than the RNN. This is normal, we applied several regularization techniques to enhance our LSTM's generalization capabilities.

We observe in the generated text that our LSTM has managed to capture context better than the RNN, generating words and mimicking Shakespeare structure of poetry.

4.4 Optimizing performance of the LSTM

In order to increase performance of our character-level LSTM we implemented several common techniques. Namely, we use dropout of 0.2, some data augmentation for noisy data (With 3% probability: Swap it with the next character. Drop it (remove). Duplicate it.), L2 Regularization, varied the number of hidden nodes, and varied the number of hidden layers.

After running several experiments, we had a set configuration of parameters as defined in params above, as we had significant performance training for one epoch. Models of higher capacity would benefit of training for more than one epoch, but it requires significant amount of training time. We stick to indicative results that already showcase tendencies in performance of the different architectures.

We built several architectures, varying both the number of hidden nodes and the number of hidden layers and made a comparison with training on augmented data. We also implemented nucleus

sampling, generating text with $p - value = 0.9$, meaning we generate the most probable characters with probability ≥ 0.9. We also implemented the following metrics to compare our results and evaluate performance on the generated nucleus samples: (1) Spelling Accuracy, to measure the proportion of words that are actual English words; (2) Bigram Overlap, to measure the proportion of 2-character sequences that are also present in the original text; (3) Trigram Diversity, to measure the proportion of unique trigrams to the total number of trigrams.

We build the following LSTM architectures and the trained on augmented data version of them: Arch_1,100 with 1 hidden layer, 100 hidden units; Arch_1,100_augmented with 1 hidden layer, 100 hidden units trained on augmented data; Arch_2,30: 2 hidden layers, 30 hidden units each; Arch_2,30_augmented: 2 hidden layers, 30 hidden units each trained on augmented data; Arch_2,60: 2 hidden layers, 60 hidden units each; Arch_2,60_augmented: 2 hidden layers, 60 hidden units each trained on augmented data; Arch_3,30: 3 hidden layers, 30 hidden units each; and Arch_3,30_augmented: 3 hidden layers, 30 hidden units each trained on augmented data.

Model Architecture	Spelling Accuracy	Bigram Overlap	Trigram Diversity
Arch_1,100	0.37	1.00	0.87
Arch_1,100_augmented	0.54	1.00	0.85
Arch_2,30	0.59	1.00	0.83
Arch_2,30_augmented	0.52	1.00	0.83
Arch_2,60	0.62	1.00	0.82
Arch_2,60_augmented	0.60	1.00	0.84
Arch_3,30	0.37	1.00	0.86
Arch_3,30_augmented	0.43	1.00	0.84

Table 2: Comparison of different architectures on spelling accuracy, bigram overlap, and trigram diversity

Studying the table 4, we observe that in general, training on augmented data, can siginificantly boost the models performance either in spelling Accuracy (Arch_1,100) or in Trigram Diversity (Arch_2,60).

We also made the following observations. Adding a hidden layer, can significantly improve the models performance, while the time complexity is mainly affected by the number of hidden nodes. For example, the architecture 2x60 was trained at approximately same amount of time as 1x100, while showcasing boosted performance. There is a trade-off though of due to higher capacity, more training is required to reach the same levels of performance. This is observed in Architecture 3x30. We see a dropped Spelling Accuracy as we only trained for one epoch, not sufficient for this level of model capacity.

We indicatively provide some generated text from our 2x60 lstm structure using nucleus sampling:

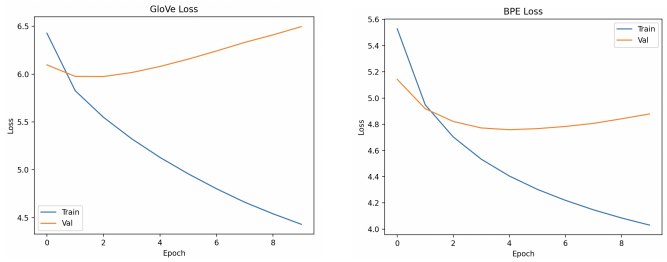
```
Generated text from LSTM trained on original data (nucleus sampling):

ROMEO.
But stanly he seelver, for to shere the lakes in fore, of lomes?
With for nevery downers'd. I flith' selle in of furtut I fath and. Heraw herat, for dinces. The our
↪  greaver, an I should as mefon to whin he dowleth cupad and on granters; liged ean ther she
↪  stombridion.

[Amnow a I hat he worls conot

Generated text from LSTM trained on augmented data (nucleus sampling):

ROMEO. Leath, no live wild steent, my leverae, with baccans's bendere, for my then hirs it withe him
↪  bansted with of more, dore good bood bay coness derion, me
This reves ifo homother! Leare of theal our that sie the copen atleas be lontane.

MOSSTHIULIO.
An with wor of It hered, liter, a we pit moys,
The
```

4.5 Word and Subword embedding

For our GloVe and BPE Tokenizers we implemented a standard one hidden layer LSTM with 128 hidden units, using Tensorflow. We trained for 3 epochs each and surprisingly dominated over any

other manually implemented LSTM. As we observed that training and validation loss were still evolving, we ran a bigger training process for 10 epochs, where for the BPE we introduced a higher `vocab_size = 8000` to hopefully observe greater diversity.

In that case, we observe that after 4-5 epochs, both models are starting to overfit creating a big generalization gap. One could handle this for future work by introducing dropout and networks of greater capacity with more hidden layers.



(a) GloVe Loss (b) BPE Loss:

Figure 1: Loss evolution in training using GloVe and BPE Tokenizers.

Finally, we display our metrics for the different training procedures of GloVe and BPE in Table 3:

Model Architecture	Spelling Accuracy	Bigram Overlap	Trigram Diversity
GloVe, 3 epochs	1.00	1.00	0.77
GloVe, 10 epochs	1.00	1.00	0.80
BPE, 3 epochs	1.00	1.00	0.35
BPE, 10 epochs	0.97	1.00	0.79

Table 3: Comparison of different architectures on spelling accuracy, bigram overlap, and trigram diversity

```
Generated text using GloVe tokenizer trained for 10 epochs:

romeo and juliet sheathed for mine error moon whereat dissentious controller space chorus scene i
↪  possessed with thee perdita take heed the ugly lad that in the least be blessed indeed thy mine
↪  usurp my sweet loves my brother canst not kill a father with me with a dauphins eye hate or more

Generated text using BPE tokenizer trained for 10 epochs:

romeo and juliet
furs be thy sheep

rosalind
why then he is a very good lover of a parthian

antipholus of syracuse
i am a villain

mercutio
why then i am a gentleman of a good world

rosalind
```

4.6 Transformer

In our project, we implemented a Transformer model for character-level language modeling. Input characters are first converted into dense vector embeddings of dimension 256, capturing syntactic and semantic information. These are combined with learned positional embeddings to preserve token order, a critical aspect as the self-attention mechanism itself doesn't keep track of sequence order. The core architecture comprises multiple Transformer blocks stacked sequentially, processing these combined embeddings.

Each Transformer block employs multi-head self-attention with four heads, enabling simultaneous focus on different input sequence parts from various representational subspaces. This is followed

by position-wise feed-forward networks that apply nonlinear transformations independently at each position. To ensure stable training and effective gradient flow, residual connections and layer normalization are applied around each sub-layer. Crucially, a look-ahead mask is used during attention calculations to maintain the autoregressive property essential for next-character prediction, ensuring that predictions rely only on preceding characters.

For training the Transformer model detailed above, we experimented with several configurations. Specifically, we varied the number of stacked Transformer blocks (L) and the total number of training epochs. These models were optimized using AdamW, which incorporates weight decay for regularization to help prevent overfitting. The learning rate was managed with a warm-up phase, gradually increasing to a peak, followed by a cosine decay schedule to promote stable convergence.

Transformer Parameters	Test Loss	Test Accuracy	Spelling Accuracy	Bigram Overlap	Trigram Diversity
1 Layer, 10 epochs	0.038	0.991	0.11	0.96	0.76
1 Layer, 200 epochs	0.024	0.994	0.71	0.98	0.55
2 Layer, 100 epochs	0.026	0.994	0.76	0.95	0.76
2 Layer, 200 epochs	0.038	0.994	0.50	0.71	0.83
4 Layer, 500 epochs	0.037	0.994	0.79	0.94	0.62
8 Layer, 1000 epochs	0.034	0.994	0.75	0.98	0.75

Table 4: Comparison of transformer configurations on final test loss and test accuracy, and these spelling accuracy, bigram overlap, and trigram diversity for the nucleus sampling

Besides the test loss and accuracy we also evaluate the the performance of the model, we sample and use the same metrics as implemented in section 4.4. We calculate the spelling accuracy, bigram overlap, and trigram diversity for all text generated by the nucleus sampling (p=0.9) all with the original starting string "ROMEO:". One thing that we observed was that all generated samples, started off with writing disorganized characters before forming real words (shown in 3).

5 Conclusions and Future Work

Optimizing an LSTM architecture is a multidimensional task, in order to find optimal hyperparameters. Given enough resources, one could run a grid search to find better values for dropout, data augmentation noise levels, L2 regularization parameter and apply different learning rate schemes, such as adaptive learning rate or cyclic learning rate. One main task is also to introduce networks of higher capacity (2x128 or 3x128 architectures) and train for sufficient epochs to observe best performance. Furthermore, one could train using pre-trained word embeddings such as GloVe or use a sub-word tokenization scheme introducing a higher capacity LSTM with regularization methods applied and a `vocal_size > 8000`.

One main observation is that introducing new hidden layers with fewer hidden units, can be as expressive while dropping time complexity. One could further study in which cases this scenario holds and provide theoretical and experimental insights.

With regard to the transformer, we believe there is a lot more potential to be realized. For instance, scaling the model to include more Transformer layers and larger embedding sizes could improve representational power. Additionally, exploring different tokenization methods such the word-level preprocessing with GloVE or subword pre-processing would help the model better fit to handle rare or compound tokens. Finally, training on larger and more diverse datasets or through data augmentation could further enhance the quality and generalization of generated text.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[2] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[3] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML)*, Bellevue, WA, USA, 2011, pp. 1017–1024.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[5] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013. [Online]. Available: https://arxiv.org/abs/1308.0850

[6] A. Karpathy, "`char-rnn`: Multi-layer Character-Level RNNs in Torch," https://github.com/karpathy/char-rnn, 2015, accessed: 2025-05-16.

[7] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, October 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162/

[8] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[9] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013. [Online]. Available: https://arxiv.org/abs/1310.4546

[10] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," 2016. [Online]. Available: https://arxiv.org/abs/1508.07909

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[12] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*. Cambridge University Press, 2023, https://D2L.ai.

Appendix

Baseline RNN

```
rnn_units = 100
embedding_dim = rnn_units//2

model = tf.keras.Sequential([
    layers.Embedding(input_dim=K, output_dim=embedding_dim),
    layers.SimpleRNN(rnn_units, return_sequences=True),
    layers.Dense(K)
])
```

Listing 1: Baseline RNN architecture

```
Generated text pre-training:
ROMEO.BJ:H_k'L49
™%G1Di‘Bv-.&hYÆOBy&2[a        ;àh’É1sHk ‘dîVxRpa&kE'To,àgré#QX7S/OId/•Ph3I
;$t&3zhë        x...?êÆ)BÉjæv(ÇAl9gMXâÀUêA;U8’[Wæ#KcPDgÇfmGâg,/xJkI        Y_ê_A)u/Yy]!oÆzÀè(‘X/qj/ÆOêV
↪  ÇK]LKQfè4v;!QUëUV
c’zM
JY/XX;?27%i m7d$,OVpO’%j(/dT‘âê’V*•zj•ÉëDQsR9nhZPê        lz,‘WëE...çyâTOèQ%
G1        D‘/_SxZYc1ë)iî)f!3S*ZXvV]vB),ÀéVç;ëq

Generated text post-training:
ROMEO.
The

! by’sw Thited nclan igray anicequt thed:
TH ton bluplout shencoproutwiges.
MILYes orase y llder.
 mige Pat cay om,
APLer are, t ssa geak tesit wig azergad, lskimatinin hopan we y, inis o bengese alot De’ Secor t
↪  ARine’T s
 m lone kn y T. ounok
MUThels plily,
SHoxayoumingusinoncu Engre Thes
```

Listing 2: Baseline RNN sample text

```
Generating text with trained Transformer (Nucleus Sampling):
ROMEO:OMORORORORORORORORORORORORORO.
OLLLO.
O.
MARDERDERDER.
 Rxerolles nothesheares forith on the world.


DUKE SY.
I will dare hum thing.

KING.
[_Rich a me in and Rominess and to be not here,
nor I on the could sake you were to you should the struth of the vert.

Enter Grear Old Clemband, Bashed

Quantitative Evaluation for Transformer (Nucleus Sampled Text):
Transformer Model (Nucleus) -> Spelling Accuracy: 0.75, Bigram Overlap: 0.98, Trigram Diversity: 0.75
```

Listing 3: Nucleus sampling of 8 Layer, 1000 Epoch Transformer

GitHub Repository The full code can be found by following the below link:

https://github.com/DavidAlm123/DD2424-Project