Enhancing Text Generation Models, From Vanilla RNN to Deep LSTM and Transformers

Michel Le Dez
KTH Royal Institute of Technology
Stockholm, Sweden
micld@kth.se

Karim Chakroun
KTH Royal Institute of Technology
Stockholm, Sweden
chakroun@kth.se

Pierre Falconnier
KTH Royal Institute of Technology
Stockholm, Sweden
piefal@kth.se

Abstract

This report investigates advancements in text generation models, comparing vanilla RNNs, LSTMs, and Transformers. Initially, we implemented and evaluated a vanilla RNN and one-layer and two-layer LSTM networks using character-level inputs. Subsequently, we explored tokenization techniques like Word2vec and Byte-Pair Encoding (BPE) to enhance LSTM performance. Finally, we implemented a decoder-only Transformer model. Our results indicate that while LSTMs outperform vanilla RNNs in capturing long-term dependencies, the Transformer model achieves superior performance, particularly in reducing misspelling rates. The best model utilized BPE with a vocabulary size of 5000 demonstrated improvements in text generation tasks.

1 Introduction

The field of text generation has evolved significantly, progressing from simple statistical models to advanced neural network models. Early methods like n-gram models were limited by their inability to capture long-term dependencies. Recurrent Neural Networks (RNNs) [1] addressed this by processing sequential data, but they struggled with long-range dependencies due to the vanishing gradient problem.

Long Short-Term Memory (LSTM) networks [2], a type of RNN, improved on this by using gating mechanisms to better capture and retain long-term dependencies. The next major advancement was the Transformer model, introduced in 2017, which used self-attention mechanisms for parallel data processing, significantly improving training efficiency and the ability to understand complex text relationships.

Building on Transformers, Large Language Models (LLMs) like BERT[3] and GPT-3 [4] have set new benchmarks in text generation, capable of producing highly coherent and contextually relevant text.

Motivated by these advancements in the field of text generation, we will review some of this major breakthrough. First, we will implement a vanilla RNN and compare its performance with LSTM networks. Then, we will delve into tokenization techniques, specifically Word2vec [5] and Byte-Pair Encoding (BPE) [6], aiming to enhance LSTM performance. Finally, we will implement a decoder-only transformer and compare its performance with LSTM models. Our implementation were done in Python, mainly using the libraries Pytorch. The scripts are available here: `https://github.com/PierreFalconnier/text-generation-project`.

2 Vanilla RNN and LSTM

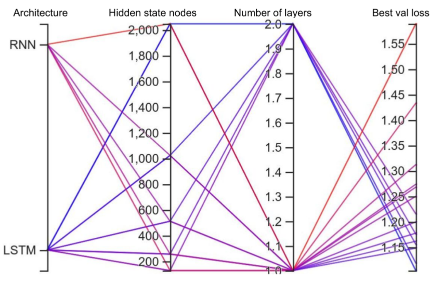
Firstly, we investigated three network architectures: a vanilla one-layer RNN, and both a one-layer and two-layer LSTMs. We used as a dataset all the Harry Potter books, which correspond to 101 unique characters for a total of 6,251,637 characters and 1,100,342 words. The inputs of the networks are sequences of one-hot-encoded characters. From a given sequence, the goal is to predict the input sequence shifted by one character. This allows the network to generate text by literally predicting new characters. The sequences built from the dataset were randomly split into a train set (70%), a validation set (15%) and a test set (15%), always using the same seed for repeatability.

Training procedure Each training was done using Adam optimizer. The loss function used is the cross entropy since the task boils down to a classification task. For now, the input characters are simply one-hot-encoded. The sequences that constitute the training batches are selected randomly. Hence, the hidden states (and the memory cells for LSTMs) are set to the null vector between each batch. The RNN gradients norms were clipped to 5 to prevent exploding gradients issues. In order to prevent overfitting, we used early stopping with a tolerance of 5 epochs.

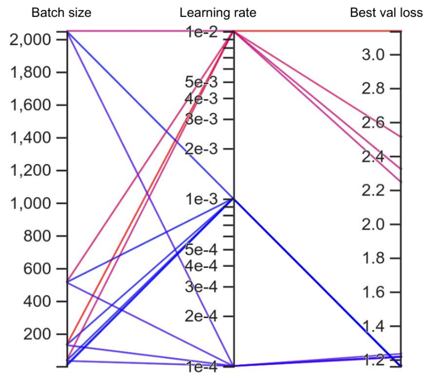
Sequence length We first trained each model with input sequence lengths of 25, 50 and 100. We used a batch size of 64, a learning rate of 10^{-3} and 1024 hidden nodes. Using a length of 100 yielded the lowest validation loss so we used it for the next experiments.

Number of the nodes of the hidden state We then investigated the influence of the number of nodes on the models' performance. To assess the models during training, in addition to the validation loss, we generated 10,000 characters long sequences after each epoch and used the Python library "spellchecker" to compute the percentage of misspelled words. For each of the three models, we tried the following number of nodes: 128, 256, 512, 1024 and 2048. Our results are shown on the Figure 1 (a).

We noticed that the more hidden nodes are used, the sooner the model over-fits the training data, which is expected since its capacity is higher. The figure 1 (a) below shows that the 2-layer LSTM performs better than the one-layer LSTM which also performs better than the RNN. Increasing the number of nodes allows better performance for the LSTMs but not necessarily for the RNN. The misspelling percentage of the 2-layers LSTM with 2048 nodes is about 8%, which is not good enough for a useful text generation task.



(a) Results of our experiments on the number of hidden nodes.



(b) Results of our experiments on the batch sizes and learning rates.

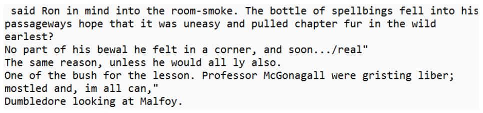
Figure 1: Results of our experiments for the different hyper-parameters

Influence of the batch size and the learning rate We performed a grid search in order to investigate both the effects of the batch size and learning rate. We used a one-layer LSTM with 1024 hidden nodes. We tried the learning rates $\{10^{-2}, 10^{-3}, 10^{-4}\}$ since they can be considered respectively as high, commonly used, and low learning rates. For the batch sizes, we tried {2048, 512, 128, 32}, as well as {1, 8} when using 10^{-3} as a learning rate. Our results are shown on the Figure 1 (b).

Using a learning rate of 10^{-2}, which is large, leads to oscillations in the training losses since the weights updates are too large. Regardless of the batch size, the losses decrease but remain relatively high, even if using a batch size of 32 seems to lead to better results.

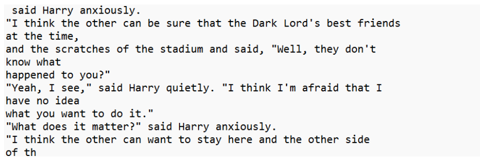
Using a low learning rate like 10^{-4} leads to slow convergence while using 10^{-3} seems to strike a good balance between fast convergence and stability. In both cases, the batch size has little to no impact on the results, even if using a small batch size such as 1 or 8 give slightly better results. The summary of the experiments are shown Figure 1 below.

Text generation To generate text, we used an initial sequence like "'Where are you?'". Then, the sequence is one-hot-encoded and given to the network. We apply the softmax function of the last vector output, which corresponds to the next character prediction. From this probability distribution over the characters, we sample a new one and add it to the initial sequence. The procedure is repeated until the desired text length is reached. The Figure 2 (a) is a generated text using the classic procedure described earlier in this report, when the initial sequence is "'Where are you?'". The model used is a 2-layers LSTM with 1024 hidden nodes per layer.



```
 said Ron in mind into the room-smoke. The bottle of spellbings fell into his
passageways hope that it was uneasy and pulled chapter fur in the wild
earlest?
No part of his bewal he felt in a corner, and soon.../real"
The same reason, unless he would all ly also.
One of the bush for the lesson. Professor McGonagall were gristing liber;
mostled and, im all can,"
Dumbledore looking at Malfoy.
```

(a) Temperature value of 1.0



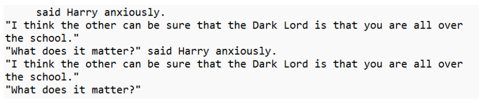
```
 said Harry anxiously.
"I think the other can be sure that the Dark Lord's best friends
at the time,
and the scratches of the stadium and said, "Well, they don't
know what
happened to you?"
"Yeah, I see," said Harry quietly. "I think I'm afraid that I
have no idea
what you want to do it."
"What does it matter?" said Harry anxiously.
"I think the other can want to stay here and the other side
of th
```

(b) Temperature value of 0.01

Figure 2: Text generation with and without temperature

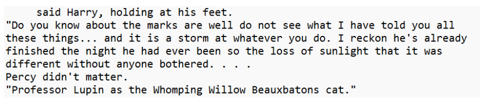
Temperature The logits obtained from the model are divided by the temperature parameter. The higher temperatures result in a smoother probability distribution, while lower temperatures emphasize high-probability words and suppress low-probability ones, making the generated text less diverse. Figure 2 (b) is an example using a temperature of 0.01. The text is not very diverse. As we generated several texts, certain sentences came back often like "What are you talking about" or "said Harry anxiously". Using a high temperature like 2 led basically to random characters, which is expected.

Nucleus sampling Instead of sampling from the entire probability distribution, we define a threshold probability p. The model selects a subset of characters whose cumulative probability exceeds p and samples from this nucleus. This method balances diversity and coherence by exploring diverse options while favoring high-probability characters. A low p (e.g., 0.01) results in low diversity and loops, while a higher p (e.g., 0.9) allows for more diversity while avoiding unlikely characters (see Figure 3).



```
     said Harry anxiously.
"I think the other can be sure that the Dark Lord is that you are all over
the school."
"What does it matter?" said Harry anxiously.
"I think the other can be sure that the Dark Lord is that you are all over
the school."
"What does it matter?"
```

(a) Nucleus sampling value of 0.01



```
     said Harry, holding at his feet.
"Do you know about the marks are well do not see what I have told you all
these things... and it is a storm at whatever you do. I reckon he's already
finished the night he had ever been so the loss of sunlight that it was
different without anyone bothered. . . .
Percy didn't matter.
"Professor Lupin as the Whomping Willow Beauxbatons cat."
```

(b) Nucleus sampling value of 0.9

Figure 3: Text generation with different values of nucleus sampling

3 Enhancing Input Tokens: Leveraging Word2vec and Byte-Pair Encoding (BPE)

With the aim of improving the text generation ability of the model, two embedding techniques were used: Word2vec and BPE tokenization. From now, the task is to predict words, or tokens.

3.1 Word2vec

Word2vec is a neural network-based method for creating word embeddings. It has two versions: Skip-Gram and Continuous Bag of Words (CBOW). Skip-Gram uses as input a one-hot encoded vector of the surrounding words (the context) of the target words and as output a one-hot encoded vector representing the target word. The neural network contains a single hidden layer, where the number of nodes corresponds to the embedding dimension. The network is trained using back-propagation with cross-entropy loss, and techniques such as negative sampling enable efficient training. For a given word, its associated weights give its embedding. In contrast, CBOW takes as input a one-hot encoded vector representing the target word and outputs a one-hot encoded vector representing its context. The primary hyperparameters of Word2vec include the embedding dimension, the number of epochs, the learning rate, and the context size.

The Skip-Gram version was arbitrarily chosen, using its implementation from the gensim library, while keeping its default parameters: 5 epochs, a learning rate of 0.025, and a window size of 5. An embedding has been created for every word, including those that are infrequent, with upper case characters transformed to lower case. Additionally, embeddings for punctuation marks have been created to enable the network to learn their usage and use them during text generation.

3.1.1 Training procedure

Given that better results were obtained with an LSTM network in the first part, it was retained for this part of the study. Initially, hyperparameters of the LSTM were tuned using a subset of the training data, specifically the fourth book: *Harry Potter and the Goblet of Fire*. The data was split sequentially with the following proportions: 70% for training, 15% for validation and 15% for testing. The tuning phase was conducted in a greedy manner by analyzing each result and guiding subsequent experiments. The Adam optimizer was used, and early stopping with a tolerance of 5 epochs was implemented to prevent overfitting. Additionally, batches of size 1 were used.

3.1.2 Hyperparamaters tuning

Learning rate The process began with a small LSTM network consisting of one hidden layer with 100 nodes. The embedding dimension was initially set to 100, the default value in the gensim library. The first hyperparameter analyzed was the learning rate. Initial runs with a learning rate of 1e-3 showed rapid overfitting, indicating a need to decrease the learning rate. Several values ranging from 1e-3 to 1e-5 were tested, and 2.5e-5 yielded the lowest validation loss.

Sequence length and weight decay Next, the length of the input sequence for the LSTM was analyzed. Sequence lengths ranging from 5 to 50, in steps of 5, were tested. A sequence length of 35 yielded the lowest validation loss. However, analysis of the training and validation loss revealed overfitting. To address this, weight decay was added to the Adam optimizer. Various values (0, 5e-6, 1e-5, 5e-5, and 1e-4) were tested, with 1e-5 resulting in the lowest validation loss.

Number of hidden nodes in the LSTM network To further enhance the performance of the LSTM, an analysis of both the number of nodes in the hidden layer and the number of hidden layers was conducted. A grid search was performed with values 1, 2 for the number of hidden layers and 128, 256, 512, 1024 for the number of nodes. Examination of the validation loss revealed that 128 nodes with one hidden layer is the optimal choice.

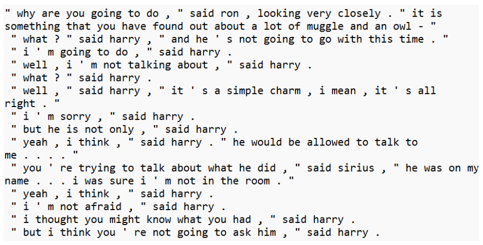
Embedding dimension of Word2vec Lastly, the impact of the Word2vec embedding dimension was analyzed. Currently, the hyperparameters values are set at 2.5e-05 for the learning rate, 35 for the sequence length, 1 hidden layer with 128 nodes for the LSTM network and 1e-05 for the L2 regularization. The embedding dimensions tested were 50, 100, 200, and 300, with 200 yielding the lowest validation loss.

3.1.3 Final model

The hyperparameters are now fine-tuned, with values set at 2.5e-05 for the learning rate, 35 for the sequence length, 1 for the number of hidden layers, 128 for the number of hidden nodes, 1e-05 for the regularization term, and 200 for the embedding dimension. To further enhance the model's performance, the entire training set, comprising all the Harry Potter books, was utilized. The data was split sequentially with the following proportions: 70% for training, 15% for validation, and 15% for testing. As in the previous part, batches of size 1 were used.

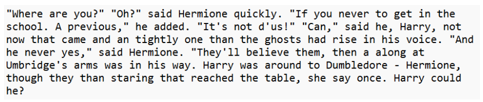
To further improve validation performance, one additional hidden layer was added, with a dropout value of 0.4 between the two layers. This adjustment resulted in a slightly lower validation loss and reduced overfitting. However, this improvement is more evident on the test set, where the cross-entropy loss decreased from 10.64 to 4.38.

Finally, text generation experiments were conducted with this model for various values of nucleus sampling. Subjectively, a nucleus sampling value of 0.6 appeared effective. The generated text is presented in Figure 4 (a) below. It should be noted that words have been converted to lowercase, so any uppercase letters should be reintroduced during post-processing. Additionally, punctuation has been treated as a token, resulting in white space surrounding it, which needs to be corrected during post-processing. In conclusion, although the generated text shows some familiarity with the Harry Potter dialogue style, it still lacks diversity and coherence. This suggests that Word2vec didn't improve the text generation ability of our model.



```
" why are you going to do , " said ron , looking very closely . " it is
something that you have found out about a lot of muggle and an owl - "
 " what ? " said harry , " and he ' s not going to go with this time . "
 " i ' m going to do , " said harry .
 " well , i ' m not talking about , " said harry .
 " what ? " said harry .
 " well , " said harry , " it ' s a simple charm , i mean , it ' s all
right . "
 " i ' m sorry , " said harry .
 " but he is not only , " said harry .
 " yeah , i think , " said harry . " he would be allowed to talk to
me . . . . "
 " you ' re trying to talk about what he did , " said sirius , " he was on my
name . . . i was sure i ' m not in the room . "
 " yeah , i think , " said harry .
 " i ' m not afraid , " said harry .
 " i thought you might know what you had , " said harry .
 " but i think you ' re not going to ask him , " said harry .
```

(a) Generated text using Word2vec



```
"Where are you?" "Oh?" said Hermione quickly. "If you never to get in the
school. A previous," he added. "It's not d'us!" "Can," said he, Harry, not
now that came and an tightly one than the ghosts had rise in his voice. "And
he never yes," said Hermione. "They'll believe them, then a along at
Umbridge's arms was in his way. Harry was around to Dumbledore - Hermione,
though they than staring that reached the table, she say once. Harry could
he?
```

(b) Generated text using BPE

Figure 4: Generated text using a LSTM for the two different tokenization techniques, namely Word2vec and BPE.

3.2 BPE tokenization

One of the main issues encountered in NLP and text generation in particular is how to deal with rare words and subword variations. To avoid this problem, we investigate using Byte-Pair Encoding (BPE) before training our model. BPE is a subword tokenization method that is used in most modern LLMs including ChatGPT [7] models. The potential benefits of BPE include improved model training dynamics, reduced out-of-vocabulary rates, and enhanced generalization to unseen text [6]. BPE works by iteratively merging the most frequent pairs of characters or character sequences in the corpus, effectively reducing the number of unique tokens while maintaining the ability to represent rare words and morphological variants. This results in a more compact and efficient vocabulary that can capture both common and uncommon word forms.

Training procedure : For each experiment, we train for 60 epochs and a batch size of 128. Prior tests showed overfitting issues so we used L2-regularization with $\lambda = 0.001$. For all training sessions using BPE, we use a cyclical learning rate (4 cycles) with a base learning rate of 5e-4. The sequence length was chosen to capture sufficient context (25 tokens for bpe and `word` mode, 200 for `character` mode). These parameters were pre-selected based on several training tests on a smaller dataset. We compare the models with metrics such as best validation loss, test loss, and misspelling percentage and qualitative assessments of generated text samples. The experiments were divided into three main categories: in the first setting, we evaluated the effectiveness of BPE compared to traditional tokenization methods by training the model using character-level, word-level, and BPE (with a vocabulary size of 5000) tokenization. The embedding dimension was set to 100, with sequence lengths of 25 tokens for BPE and words, and 200 for characters. In the second setting, we explored the impact of BPE vocabulary size on model performance by testing three different sizes: 1000, 5000, and 15000, while keeping the embedding dimension at 100 and maintaining consistent training parameters. The third setting investigated the impact of embedding dimensions on model performance using BPE, testing dimensions of 50, 100, and 300. Initially, the BPE vocabulary size was fixed at 5000, then experiments were conducted with varying combinations of vocabulary size and embedding dimensions: (1000, 50), (5000, 100), and (15000, 300).

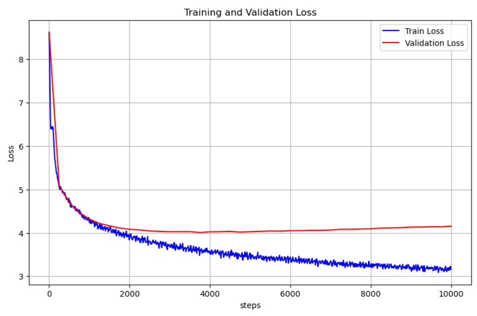
After conducting these experiments, we identified the optimal model configuration. The best model utilized a BPE vocabulary size of 5000 and an embedding dimension of 100, achieving the lowest validation loss of 4.93 and a test loss of 9.39, with a misspelling percentage of $\sim 10\%$. However, in our case, using BPE didn't yield better results than using simple characters or complete words. Figure 4 (b) shows a sample of a text generated using our model trained with BPE. From a subjective point of view, the generated texts using BPE still lack coherence, suggesting that BPE did not improve the generation ability of our model.

4 A small Transformer

Finally, we used a Transformer as the SOTA performance is achieved by transformer-based models. The model is made of decoder layers. The LN layers are placed before the Multi-Head attention and the MLP as it can speed up the training according to [8]. After the last decoder, we apply a LN layer followed by a MLP. The input sequence and the positions are embedded by a learnt embedding layer. The sequence embedding shares its weights with the last MLP, since it reduces the number of trained parameters without harming the performance [9].

Our choices were based on [10]. We trained the networks for 5,000 steps using AdamW optimizer [11]. The learning rate scheduler is a short warm-up phase followed by a Cosine decay. We tried two network configurations: 8 layers with 8 heads, 6 layers with 6 heads, both with an embedded dimension of 384 and a dropout of 0.2 in the MLPs.

For character prediction, the transformer models reached similar test losses as our best previous LSTM models (~ 1.1) but out-performed them in terms of misspelling percentages ($\sim 3\%$ against $\sim 8\%$). As expected, the larger transformer configuration led to slightly better performance ($\sim 0.5\%$ better). We also tried to use the position embedding from "Attention is all you need" [12] but we got slightly worse results. We also tried BPE tokenization before training the transformer model. Again, we tested different configurations before selecting the final training parameters. Ultimately, we used the larger architecture mentioned previously, a BPE vocabulary size of 5000 and a larger dropout of 0.5 to prevent over-fitting. Other parameters remained the same. We obtain slightly better results compared to character mode with a misspelling percentage of $\sim 2\%$.



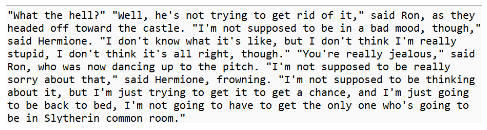

"What the hell?" "Well, he's not trying to get rid of it," said Ron, as they headed off toward the castle. "I'm not supposed to be in a bad mood, though," said Hermione. "I don't know what it's like, but I don't think I'm really stupid, I don't think it's all right, though." "You're really jealous," said Ron, who was now dancing up to the pitch. "I'm not supposed to be really sorry about that," said Hermione, frowning. "I'm not supposed to be thinking about it, but I'm just trying to get it to get a chance, and I'm just going to be back to bed, I'm not going to have to get the only one who's going to be in Slytherin common room."

Figure 5: Left: Evolution of the cross-entropy loss on the training and validation sets using the best transformer model (with BPE). Right: Generated text using the best model with a nucleus sampling value of 0.6

The text generated using transformers and BPE showcases some interesting elements in the dialogue between the book's characters. The model captures the conversational tone and attempts realistic exchange. However, there are sometimes some repetitions.

5 Conclusion

In conclusion, our experiments confirmed that Transformer models outperform LSTMs and vanilla RNNs in text generation tasks, particularly in terms of coherence and misspelling rates. The optimal configuration for our models involved using BPE tokenization. These findings underscore the importance of advanced tokenization methods and Transformer architectures in achieving state-of-the-art performance in text generation.

References

[1] Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural networks.

[2] Harsha Vardhana Krishna Sai Buddana, Surampudi Sai Kaushik, PVS. Manogna, and Shijin Kumar P.S. Word level LSTM and recurrent neural network for automatic text generation. In *2021 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4. ISSN: 2329-7190.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[6] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[7] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[8] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture.

[9] Ofir Press and Lior Wolf. Using the output embedding to improve language models. version: 3.

[10] Andrej. karpathy/nanoGPT. original-date: 2022-12-28T00:51:12Z.

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

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