Language Modeling on Penn Treebank

Shannon Rumsey

Abstract

A decoder-only language model is created to generate sentences from the Penn Treebank dataset. This model achieves a perplexity of 269, compared to GPT-2 which gets around 65 (Radford et al., 2019).

1 Introduction

1.1 Task

The task is to create a language model that is able to generate text based on the Penn Treebank dataset hosted by Hugging Face. Based on the data, the model learns what sentences "look" like and generate similar sentences, one word at a time. This dataset consists of American English sentences taken from 1989 Wall Street Journal material. Rare words were replaced with an <unk> token and the dataset was already split into a train, validation, and test set. Because the dataset only consists of sentence excerpts, there are no formal target outputs defined, making this task unsupervised. The purpose of generative models is to predict the next token in a sequence, therefore the data the Penn Treebank data is used as both an input for the model to predict from, but also the target since the model tries to predict the input token in the next time step.

1.2 Dataset

The dataset did not need any preprocessing as it was already relatively clean. An example sentence from the training data showcases this: "the <unk> group said its earlier projection that group profit for all of N would be close to the N million francs posted for N remains valid". The <unk> tokens were already included and all words were lowercase, with the exception of the N token which is a placeholder for numbers. The language model is expected to provide similar, if not the same the sentences that it was fed.

As seen in Figure 1, there is a drastic difference in frequency in even the thirty most common words,

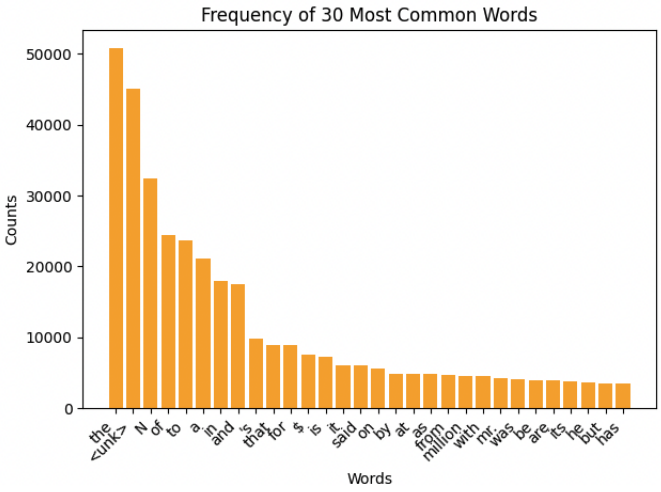

Figure 1: The thirty most common words in the training dataset.

suggesting that the rest of the data is imbalanced. In fact, the median word frequency in the training dataset is 15 and the mean is 88. While this dataset would be a good candidate for resampling, the nature of text data makes it unsuitable. If the words in a sentence were resampled, this would make the outputs of the language model seem unnatural.

2 Model

The model created was based on the encoder portion of the model from Vaswani et al. (2023), except it contains look-ahead masking and pad masking, making it more similar to a decoder-only transformer. The transformer architecture was chosen due to its performance on generative tasks, and the decoder-only specification for its simplicity to implement while still maintaining performance (Radford and Narasimhan, 2018).

A transformer block was created using PyTorch's `nn.TransformerEncoder` module. This model is a collection of encoder layers that are defined using `nn.TransformerEncoderLayer`, and takes the positional embeddings as inputs. Before

being passed to the encoder layers, the embeddings were masked to prevent the model from "looking-ahead" at the future time steps it is intended to predict, and to ignore padding. A final linear layer was used to project the model outputs.

The number of epochs that the model was trained on was based on convergence, which is determined by the validation loss failing to increase. Inspired by Brown et al. (2020), the hyperparameters experimented with were the number of attention heads, number of layers, number of embedding dimensions, batch size, and learning rate.

2.1 Embeddings

Dynamic embeddings were used due to their superior performance when compared to static embeddings (Wang et al., 2020). To create dynamic embeddings that are learning through PyTorch's `nn.Embedding` module, each word in the target sentence was mapped to a unique number. The semantic representations of these sequences of numbers were learned and stored as one type of embedding, and also used as inputs to learn the positional embeddings as well. Positional embeddings are important for the model to understand where words should be in a sentence, since it lacks the sequential structure of models such as the RNN (Vaswani et al., 2023). The class that learns these positional embeddings, `PositionalEncoding`, takes in the learned semantic embeddings, and calculates the physical distance that each word in these embeddings are from the beginning of the sentence. With this information, the class learns more nuanced positional embeddings that help the model learn where words are typically located in a sentence. The positional embeddings and semantic embeddings are added together into one embedding structure.

3 Experiments

The baseline model has an embedding size of 256, 4 layers, 4 attention heads, a learning rate of 0.001, and batch size of 64. The number of heads being experimented with must be divisible by the number of embeddings, so the number of heads experimented with follows this rule. The hyperparameters observed were 2, 4, and 6 layers, 2, 4, and 8 heads, embedding sizes of 128, 256, and 512, learning rates of 0.001, 0.003, and 0.01, and batch sizes of 32, 64, and 100.

Batches of data were created using PyTorch's `DataLoader`, with a special padding function that alters the `collate\_fn` parameter. To calculate loss, `nn.CrossEntropyLoss` was used, and it was modified to ignore padding tokens.

The metrics used to determine the hyperparameters chosen for the final model were loss and perplexity. Perplexity was calculated using the sentence-wise loss of the model's outputs, which was then exponentiatied to create sentence-wise perplexities. These individual perplexities were only considered when outputting the final model's results to a CSV file. Instead, to determine which hyperparameters to use, the average perplexity across all sentences in the validation set were used, alongside loss. Validation loss was calculated across the entire dataset, instead of the average per sentence.

4 Results

Interestingly, the training loss was between 4 and 8 across all model variations. This is substantially higher than the validation loss values seen during experimentation.

Layers	Heads	E. Size	LR	Batch Size	Perplexity	Val Loss
4	4	256	0.001	64	295.87	4.82
6	4	256	0.001	64	1604.31	7.14
2	4	256	0.001	64	346.04	4.92
4	8	256	0.001	64	300.33	4.83
4	2	256	0.001	64	299.82	4.84
4	8	512	0.001	64	1000.71	6.62
4	8	128	0.001	64	292.68	4.88
4	8	256	0.003	64	863.75	6.55
4	8	256	0.01	64	841.95	6.54
4	8	256	0.001	32	294.00	4.82
4	8	256	0.001	100	298.82	4.85

Table 1: Hyperparameter values experimented with and their effects on loss and perplexity.

The best model is the one with 4 layers, 8 heads, an embedding size of 128, a learning rate of 0.001, and a batch size of 64. The model parameters experimented with in Vaswani et al. (2023) are larger from being trained on significantly more data, therefore it is reasonable that the hyperparameters in this experiment are smaller to accommodate the smaller dataset. Their large model had 6 layers, which explains why this project's model with the same number of layers performed worse when trained on a smaller dataset. This is similar to the embedding sizes, where anything too large would cause performance degradation. The findings from Michel et al. (2019) suggest that a single head per layer is sufficient, whereas the model created has a 2:1 ra-

tio. Changing the batch size does not significantly impact model performance, which is interesting because batch size is related to the frequency at which the optimizer is updated, making it more or less prone to being trapped in a local minima. Poor performance on larger learning rates suggest that the model fails to converge towards the minimum of the cost function.

Val Perplexity	Test Perplexity	Train Loss	Val Loss
294.74	269	4.52	4.88

Table 2: Performance of best model.

References

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. 2020. Language models are few-shot learners.

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one?

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

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

Yile Wang, Leyang Cui, and Yue Zhang. 2020. How can bert help lexical semantics tasks?