Joke Generation Using Decoder Models and GPT-2 Fine-tuning

Kanishk Aman
Indian Institute of Science
UMC 301: Assignment 3

Contents

Introduction

In this report, I describe my approach to training a decoder model to generate jokes and then later fine-tuning a GPT-2 model for the same purpose, and then comparing them both. The project involved preprocessing the dataset, training and validation of the created models, and generating results & analyzing their performance. The assignment is structured into two main questions: **Q1** focuses on creating and training the decoder model, and **Q2** on fine-tuning GPT-2. My workflow is shown in the following sections.

1 Question 1: Training a Decoder Model

1.1 Part 1: Dataset Loading, Preprocessing, and Splitting

For this task, I worked with three joke datasets from the GitHub repo given to us: 'reddit_jokes.json', 'stupidstuff.json', and 'wocka.json'. My goal was to preprocess these datasets and split them into training, validation, and test sets. Below, I explain what I did, how I did it, and why I made my specific choices.

1.1.1 Loading and Combining Datasets

To begin, I wrote functions to load the datasets from JSON files and merge them into one unified collection. I combined them to increase the diversity of jokes and create a robust training set. Although it wasn't very necessary as 'reddit_jokes.json' was a very large file, and constituted most of the training data compared to the other two json files. (Also the body of the JSON files were different, which I later had to filter.)

1.1.2 Preprocessing the Data

Once the datasets were merged, I cleaned and filtered the jokes. This step was critical to ensure the quality of the training data. My preprocessing involved:

- **Cleaning Unicode Characters:** Some jokes contained special characters that could interfere with training. I normalized these using the `unicodedata` library.
- **Removing Extra Whitespace and Special Characters:** I eliminated redundant spaces, tabs, and newlines to maintain text consistency.
- **Filtering Short Jokes:** Jokes with titles or bodies shorter than five characters were excluded. I kept the min_length = 5, although it could be changed.
- **Profanity Filter:** I added this initially to censor out some of the vulgar and offensive words, but later removed this part as *many* of the jokes in my training data had them. (and to be honest jokes are more funny *with* them. Lol.)

This process reduced noise in the dataset, leaving only "high-quality jokes" suitable for training the decoder model.

1.1.3 Splitting the Dataset

The cleaned dataset was then divided into three parts:

- **Training Set (80%):** Used to train the model.
- **Validation Set (10%):** Used to tune hyperparameters.
- **Test Set (10%):** Used to evaluate the final performance of the model.

I used the usual `train_test_split` function from `sklearn` to create these splits randomly while maintaining the specified proportions.

1.1.4 Saving the Splits

Finally, I saved each split of the combined file as a separate json, making it easier to access them during model training and evaluation.

1.2 Part 2: Model Creation and Training

This section describes how I created the decoder model and my experiments with hyperparameter tuning to improve the model performance.

1.2.1 Model Description

The model used in this task is designed for generating jokes based on tokenized input sequences. The architecture is built around a **Transformer decoder**, which includes:

- A word embedding layer to convert tokens into dense vector representations.
- Positional embeddings to encode positional information of tokens in sequences.
- A stack of decoder units, each comprising:
  - Multi-head self-attention to capture relationships between tokens.
  - Feedforward layers for non-linear transformations.
  - Layer normalization and residual connections for stability.
- A final linear layer to project embeddings into vocabulary space.

1.2.2 Explanation of Classes

The `JokeDataset` class is responsible for preparing and managing the dataset:

- Loading and preprocessing jokes from a JSON file.
- Generating a vocabulary based on word frequencies, with special tokens for padding (`<pad>`) and end-of-sequence (`<eos>`).
- Tokenizing text into sequences with padding and truncation for uniformity.
- Providing input, target, and attention mask tensors for training.

The `JokeGenerator` class implements the Transformer decoder architecture:

- Word and positional embeddings for input encoding.
- A configurable stack of decoder units, with adjustable number of layers, heads, and hidden dimensions.
- Dropout for regularization and a final linear layer for generating output logits.

1.2.3 Training Process

The training process optimizes the model using Cross-entropy loss, calculated between predicted logits and target sequences. Key aspects include:

- Gradients are clipped to prevent exploding gradients.
- The OneCycle learning rate scheduler is used to improve convergence.
- The final weights are saved for further usage.

Training and validation losses are monitored across epochs, and then visualized using plots for each set of hyperparameters.

1.3 Hyperparameter Tuning Experiments

1.3.1 Random Trial

In the baseline configuration, I experimented with a model using 2 layers and a hidden dimension of 128. Initially, I trained the model for a random set of hyperparameters for 10 epochs. However, I noticed that the gap between the training and validation errors kept increasing as the number of epochs increased. Therefore, for the rest of the experimentation process, I ran the model for 5 epochs only as I felt it was sufficient. Here is the general trend that the training and validation errors followed (for 10 epochs):

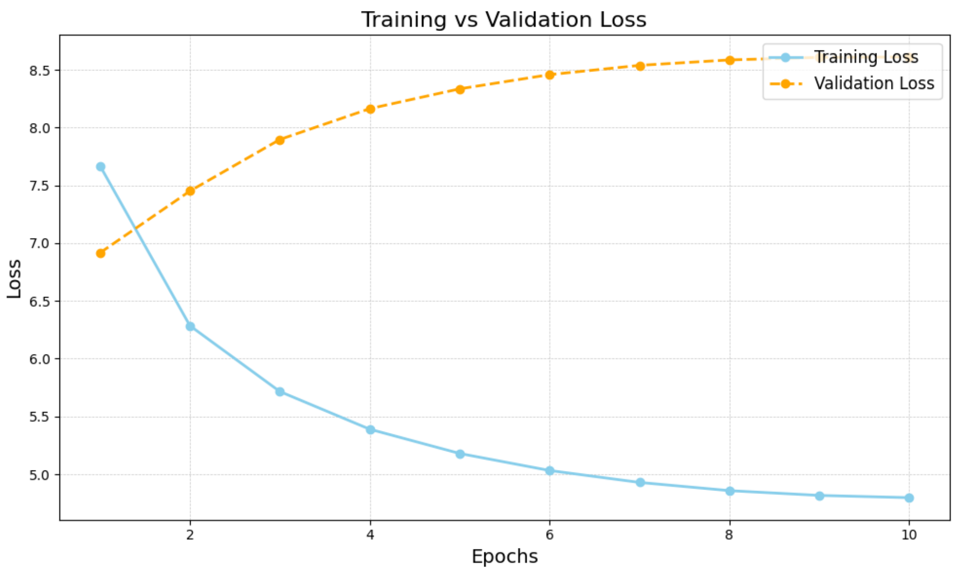


Figure 1: Training vs Validation Loss (Random Trial)

Epoch	Training Loss	Validation Loss
1	7.669	6.916
2	6.285	7.450
3	5.715	7.897
4	5.389	8.163
5	5.180	8.335
6	5.032	8.458
7	4.929	8.538
8	4.858	8.585
9	4.817	8.607
10	4.797	8.611

Table 1: Training and Validation Loss for Baseline Try

The above model failed to generalize well, as evidenced by the increasing validation loss. So, I performed the following experiments to find the best set of hyperparameters:

1.3.2 Experiment 1: Lower Learning Rate and Increased Dropout

This experiment aimed to reduce overfitting by lowering the learning rate to 5×10^{-5} and increasing the dropout rate to 0.2. Results are shown in Table 2.

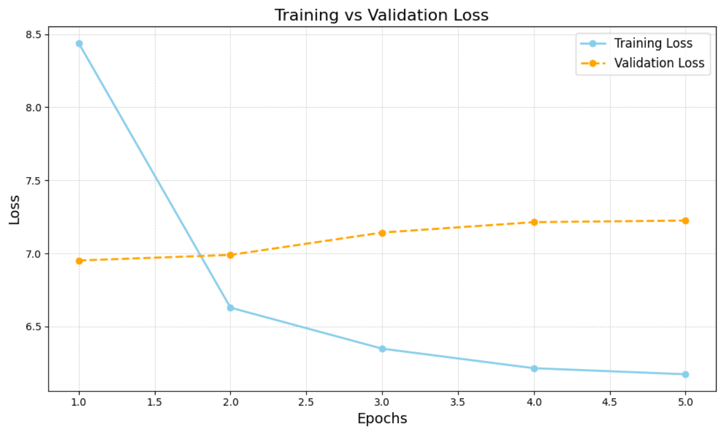


Figure 2: Training vs Validation Loss for Experiment 1

Epoch	Training Loss	Validation Loss
1	8.439	6.952
2	6.629	6.990
3	6.348	7.143
4	6.215	7.214
5	6.173	7.225

Table 2: Training and Validation Loss for Experiment 1

While the validation loss stabilized, the model underperformed overall. A lower learning rate and higher dropout might have overly constrained learning.

1.3.3 Experiment 2: More Layers and Heads, Lower Learning Rate

This experiment increased the number of layers to 4 and attention heads to 8 while slightly increasing the learning rate to 2×10^{-4}. Results are shown in Table 3.

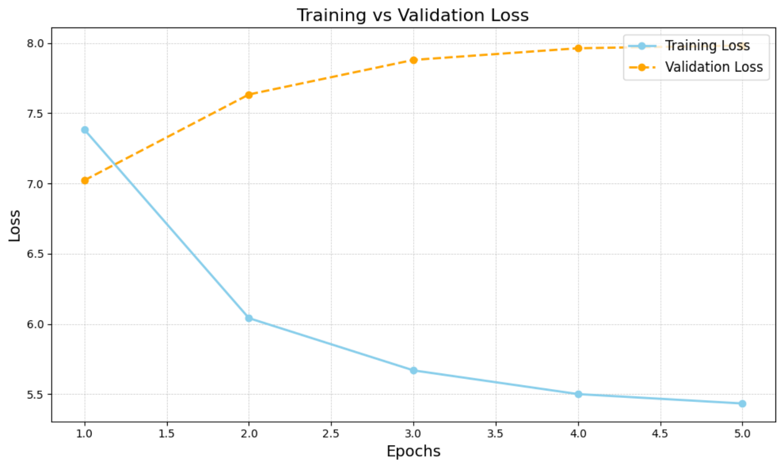


Figure 3: Training vs Validation Loss for Experiment 2

Epoch	Training Loss	Validation Loss
1	7.383	7.022
2	6.041	7.633
3	5.669	7.879
4	5.500	7.962
5	5.434	7.981

Table 3: Training and Validation Loss for Experiment 2

1.3.4 Experiment 3: Increased Dropout and Smaller Batch Size

In this experiment, dropout was set to 0.3, and the batch size was reduced to 16 to improve gradient updates. Results are shown in Table 4.

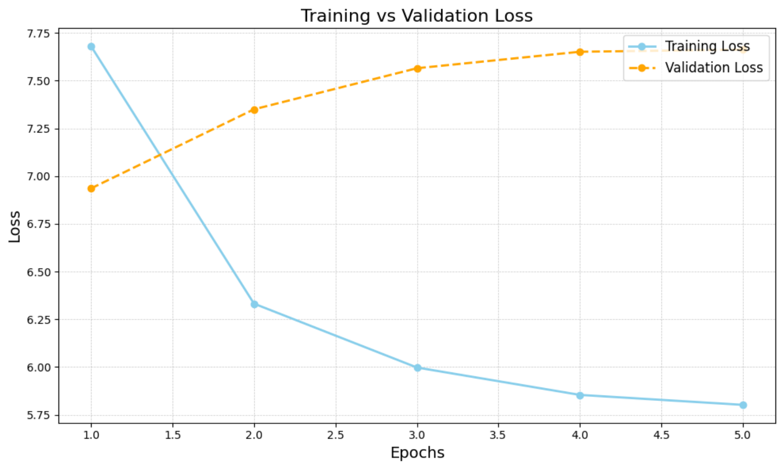


Figure 4: Training vs Validation Loss for Experiment 3

Epoch	Training Loss	Validation Loss
1	7.680	6.935
2	6.332	7.350
3	5.997	7.565
4	5.854	7.651
5	5.802	7.662

Table 4: Training and Validation Loss for Experiment 3

The smaller batch size improved gradient precision, but the increased dropout likely hindered the model's ability to fully learn the training data.

1.3.5 Experiment 4: Increased Layers & Heads and Hidden Dimension

This experiment used 4 layers, 8 heads, and increased the hidden dimension to 256 while maintaining a learning rate of 2×10^{-4}. Results are shown in Table 5.

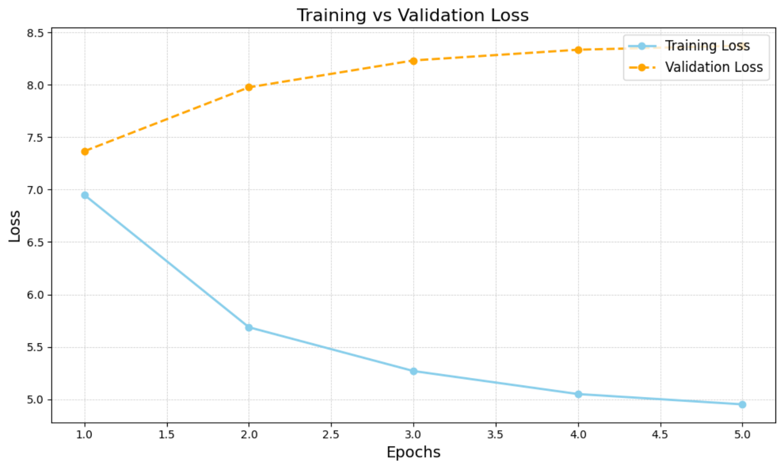


Figure 5: Training vs Validation Loss for Experiment 4

Epoch	Training Loss	Validation Loss
1	6.949	7.366
2	5.687	7.975
3	5.270	8.231
4	5.051	8.333
5	4.952	8.372

Table 5: Training and Validation Loss for Experiment 4

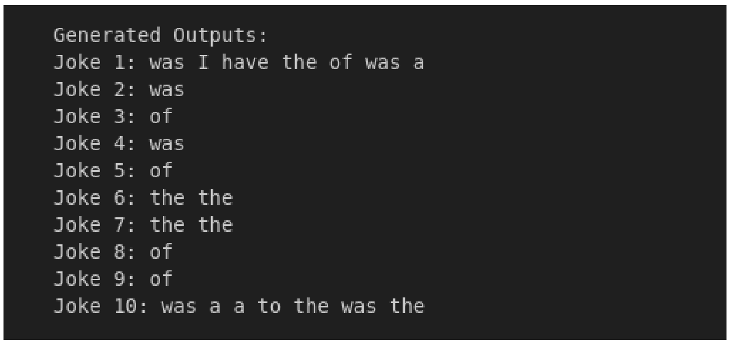
Despite using a larger model, the validation loss increased, indicating that even this configuration might be too complex for the dataset.

1.3.6 Best Hyperparameters:

The best performing model in my opinion was obtained in **Experiment 1**, which balanced model complexity and optimization settings. Both the Training and Validation loss stayed low here compared to the other experiments, suggesting better model performance.

1.4 Part 3: Generating Samples

I selected random samples from the test dataset and passed them through the trained decoder model. However, the generated outputs were weird and didn't make any sense. This could probably be due to insufficient data points, very basic methodology of the model architecture or some other other training issues. (I really couldn't figure out)
Here are some of the outputs that it generated:



```
Generated Outputs:
Joke 1: was I have the of was a
Joke 2: was
Joke 3: of
Joke 4: was
Joke 5: of
Joke 6: the the
Joke 7: the the
Joke 8: of
Joke 9: of
Joke 10: was a a to the was the
```

Figure 6: Outputs Generated by the Decoder Model

2 Question 2: Fine-Tuning the GPT-2 Model

For the second part of the assignment, I fine-tuned a GPT-2 model on the joke dataset. The fine-tuning process involves the following steps:

1. Load the pre-trained GPT-2 model and tokenizer from Transformers library.
2. Prepare the dataset by combining joke titles and bodies.
3. Tokenize the dataset and create input-output pairs for training.
4. Train the model using a suitable optimizer and scheduler over multiple epochs.
5. Save the fine-tuned model and tokenizer for generating jokes.

The model is fine-tuned on a dataset where each example consists of a joke title and body, and the model learns to predict the next word in the sequence during training.

2.1 Joke Generation

Once the model is fine-tuned, it can be used to generate jokes by providing a prompt. (something like, "Tell me a joke: ") The generation process leverages the model's ability to predict sequences of text based on the given input.

The fine-tuned model uses a sampling-based approach to produce diverse and coherent jokes. The model generates one joke at a time based on the input prompt and can be adjusted for creativity by tweaking parameters such as `top-k`, `top-p`, and `temperature`. I have currently set the number of jokes to 10 and `max_length` to 100, but we can change it to any number.

2.2 Comparison of Results

As observed, even after many attempts the **Decoder model** failed to produce coherent or meaningful outputs. Most of the generated "jokes" consisted of repetitive or nonsensical words (whatever the model could learn), with no structure or humor. This lack of meaningful outputs could be attributed to the following reasons:

- Insufficient training data or lack of diversity in the dataset.
- The decoder model's limited capacity to capture and understand linguistic nuances required for joke generation.
- The absence of a pre-trained base model to leverage pre-existing language info.

2.2.1 Results from the Fine-Tuned GPT-2 Model:

The **Fine-tuned GPT-2 model**, on the other hand, was able to produce "well-structured jokes". Although the jokes were mostly not funny (or they probably were, I just couldn't understand them), they demonstrated coherence and utilized words from the custom joke dataset effectively (one could literally see this with the amount of offensive jokes it produces). The model's outputs were syntactically correct and exhibited logical sentence construction. This improvement can be attributed to:

- The pre-trained GPT-2 model's ability to encode linguistic patterns and structures from extensive prior training on diverse datasets.
- Effective fine-tuning on the custom joke dataset, which allowed the model to adapt and incorporate specific terms and phrases relevant to the task.
- The ability of GPT-2 to handle sequence generation tasks with better context retention and prediction capabilities.

Here are some of the jokes generated by the fine-tuned GPT-2 model (Of course, I have removed all the NSFW content in the examples provided below, but the model still generates such jokes, so please be careful.)

Joke 1: Tell me a joke: Where do you put a piece of candy on the ground? Ground up.

Joke 2: Tell me a joke: What do you call a person with no arms and no legs in a bar?: A duck.

Joke 3: Tell me a joke: What did the chicken say to the tree?: "You can't eat this!"

Joke 4: Tell me a joke: A man walks into a bar and says, "My name is Bill Cosby and I would like to have a drink with you." The barman says "no, you have to go to the bathroom." So the man goes to his bathroom and asks the barkeep, who says he has a rule. If you tell me your name, I will go in and get you a free drink. So Bill goes in, sits down and takes a seat. He sits back down

Joke 5: Tell me a joke: So I was driving down the street and I hit a wall and hit my head on the glass

Joke 6: Tell me a joke: What do you call a Mexican who is in the closet? A closet-nazi

Joke 7: I want to get a job as an accountant. The first thing I have to do is write a check.

Joke 8: Tell me a joke: What do you call an Englishman flying a plane? A pilot.

Joke 9: Tell me a joke: I'll tell you one.

Joke 10: Tell me a joke: I heard a guy was dying to be a firefighter. His wife asked him what he did. He said he worked out on a tractor.

Conclusion

This project explored two approaches to joke generation: Training a Decoder model and Fine-tuning GPT-2. The preprocessing step ensured high-quality input data, while experiments with hyperparameters led to a well-tuned decoder model. Fine-tuning the GPT-2 model demonstrated its ability to adapt to the joke dataset much efficiently, offering a benchmark for comparison. The report summarizes the challenges and insights gained during this process.

The report also highlights the significant advantages of using a pre-trained language model like GPT-2 for fine-tuning, especially in tasks that require structured and coherent text generation. While the decoder model struggled to produce meaningful outputs, the fine-tuned GPT-2 demonstrated its potential to create syntactically correct jokes, leveraging the custom dataset much more effectively.