NLP Assignment 3
Rahul Aditya (UNI: ra3261)

Task 1

1. RNNs suffer from the problem of capturing long range dependencies. This is because of the exploding gradient and/or vanishing gradient problem. This problem is particularly exemplified in the case of longer sentences. In RNNs, if weight matrices in the recurrent connections have eigenvalues larger than 1, gradients can explode as they backpropagate through time, making training unstable. Vanishing gradient problem occurs due to the repeated multiplication of weights in the recurrent connections where weight matrices have eigenvalues smaller than 1. As a result, RNNs struggle to capture relationships between distant tokens or time steps in the sequence.
   This problem is solved in the case of transformers because it uses an attention mechanism to capture long range dependencies. It calculates separately for each token how much attention it should give to every other token. Another important architectural marvel is the fact that transformers are devoid of recurrent connections and hence training transformers can be done efficiently in parallel on GPUs. In order to retain the sense of sequential data, transformers instead use positional encodings which are superpositions of sinusoidal functions.

2. First let us talk about the word-level tokenizer. There are some obvious advantages of such a tokenizer. First of all, the word level tokens are highly intuitive to humans and hence we would be able to interpret the output of the model in a much better way. This has also got to do with retaining the semantic characteristics of the language which are nothing but the words. However, word-level tokenizers suffer from the problem of OOV words. This is because the vocabulary is already set to whatever was there in the training corpus. So for tasks like Machine translation, if there comes a word in

the source language which the model has not seen before, it might start producing [UNK] tokens in the output.
In the case of character-level tokenizers, it is obvious that the problem of OOV words is solved trivially in **most** of the cases as we expect the training data to atleast have all the characters in the language. It can also thus handle a very morphologically rich language because the atomic tokens are characters. Hence such a tokenizer is more robust, can generalize well and captures fine-grained details. An immediate disadvantage is that the tokenizer would actually produce lots of tokens and hence the training as well as the inference time increases substantially. The tokenizer also loses the semantic structure of the language and if not trained properly can create potential problems.

3. Architecturally, the encoder-decoder transformer model uses separate encoder and decoder components where in an autoregressive model like GPT-2, the decoder is the entire model. In an encoder-decoder model, the encoder tries to come up with a contextual representation of the input to be used by the decoder for task specific generation like Machine Translation or Natural Language Summarization. The decoder then uses this contextualized representation to generate an output sequence step-by-step. Autoregressive models process the input sequence step-by-step which is conditioned on whatever the model has generated so far. In other words, the model predicts the next token based on the context of previous tokens. This is especially useful for language modeling and text generation tasks.

   There is an encoder-decoder interaction in encoder-decoder based models. The transformer in this case tries to construct a representation of the input which is a fixed size embedding vector via an encoder and then it uses the decoder to train the model to learn the parameters based on the specific task we are training the model for. In the case of a decoder, the model comes up with a conditional probability distribution of the output tokens based on whatever it has produced so far. It generates the output sequence token-by-token, conditioning each prediction on the previously generated tokens. In simple terms, an encoder-decoder transformer concentrates on encoding input data and enabling the creation of sequences from start to finish, while an autoregressive transformer generates sequences incrementally, predicting the next tokens based on the context of previously generated ones.

4. For an encoder-decoder architecture, all that is needed is a parallel corpus of sentences in the two languages. The source sentence goes into the encoder after tokenization which then creates a contextualized embedding vector to be used by the decoder to generate tokens step-by-step in the target language. For this case, we would have pairs of sentences in the two languages.
   Example: **{"I am Rahul", "Soy Raul"}** where the 1st sentence is in English and the 2nd sentence is in Spanish.

   On the other hand, for an autoregressive model which is based on conditional generation of tokens based on the context. During training time, we can combine the two sentences in the two languages separated by the token. For example, a sample sentence for an autoregressive model with only a decoder can be "**I am Rahul <LAN_SWITCH> Soy Raul**". We would pass the prefix "**I am Rahul <LAN_SWITCH>**" as a prompt and use the suffix "**Soy Raul**" to train the model using a loss function which in our case could be cross entropy loss. But during training we are using the entire sentence "**I am Rahul <LAN_SWITCH> Soy Raul**". **<LAN_SWITCH>** is a special token to inform the decoder that it needs to switch the language. It can however depend on the exact model that we are using. During inference, we will use only the source language sentence and the special token as prompt. It would look something like this: "**I am Rahul <LAN_SWITCH>**". Hence, during inference the decoder would know after seeing the token **<LAN_SWITCH>** that it needs to attend at the source language sentence and start generating tokens in the target language.

5. It is possible to structure numerous NLP tasks in a way suitable for autoregressive models to learn but it is important to keep in mind that the main objective of autoregressive models is in language generation. So for example, in an NER task, we can provide the input sentence as context to the decoder and let it generate NER tokens sequentially but it is not an ideal way to do it because the number of NER labels equals the number of words in the input sentence. It is not clear as to why the model should keep generating tokens even though it has already generated the required number of tokens. There are ways to mitigate this problem by considering only the required tokens as needed and setting a lower limit on the number of tokens generated while training using maybe a cross entropy loss. In a similar way, machine translation, summarization as well as QA can be restructured to fit the generation requirements of the decoder. For example, in the case of machine translation, we can provide the source language sentence as input and train the model on the translated output it generates, conditioned on whatever it has generated so far. Similarly, for summarization we can give the entire paragraph as a prompt but even in this case it becomes a bit difficult for the decoder to get a good contextualized representation of the entire paragraph as it lacks an encoder. It is also a bad idea to use decoders for image captioning as we would need an encoder to better capture the contextual representations in the image.

6. Beam search is also a decoding algorithm that can be used to generate tokens in a sequence-to-sequence tasks much like greedy decoding which generates the most like token at a specific time step conditioned on whatever it has generated so far. Beam search differs from greedy decoding in the sense that unlike greedy decoding which takes into consideration only the most probable token in a specific time step, Beam search maintains multiple most-like sequences in each time step. The number of most-likely sequences maintained by the Beam search algorithm is a hyperparameter called beam-size or beam-width. For example, if beam size is set to 3, then at a specific

time step, we will have 3 most likely next tokens. The next 3 most likely tokens are determined by computing the conditional probabilities of 9 possible choices and then picking the best 3 out of them. It mitigates the problem of outputting sub-optimal sequences by a huge margin although it does not completely solve it. Trivially speaking, for a beam size of 1, beam search is the same as greedy decoding. In some cases where computational resources like memory and time are limited, greedy decoding may be a better choice owing to its simplistic design and efficient computational memory and time requirements.

7. The performance of a fine-tuned model on a dataset with a lot of random <JUNK> tokens would be a lot worse than a dataset without such random <JUNK> tokens. This is because a transformer, and more so an autoregressive model, generates conditional probability distributions on whatever it has seen so far. The introduction of <JUNK> tokens to the dataset would unnecessarily make the model capture irrelevant patterns of <JUNK> tokens. The introduction of <JUNK> tokens would reduce the natural language coherency and confuse the model's understanding of true language text and semantic relationships between them. Also, since the transformer models are based on self-attention (in this case masked self-attention), the model would unnecessarily pay attention to <JUNK> tokens which were anyways useless and would be dropped in the post-processing step. It also reduces the model generalization to proper natural language sentences which don't have <JUNK> tokens. It is because of the noise added to the dataset we can expect the translation quality to be pretty bad. On the other hand, in the absence of these <JUNK> tokens, the transformer model would be able to capture the relevant patterns in a much better way. It would understand the context better and be able to generate nice conditional probabilities based on the source language sentences.

Task 2

Note: Layers of GPT-2 are indexed from 1 to 12. (1-based indexing)

1. Preparation

   I downloaded the SNLI corpus from this link. This saved me from uploading the corpus everytime I used the Google notebook.

   It is made sure that each of the three classes (neutral, entailment, and contradiction) are equally represented in the lines used in training, validating and testing our classifier. This is done by fixing the number of lines from each class to be one-third of the total number of lines we want to read and then while reading the dataset, such lines are ignored for which the required number of lines have already been taken from that class. This is implemented in the "**read_jsonl()**" function.

   Finally the GPT2 model as well as tokenizer are loaded from the Huggingface Transformers library.

2. Obtain Internal States

   Each example is transformed as:
   transformed_sent = sentence1 + ' This has the same meaning as: ' + sentence2

   This transformation is done in the "**transformed_data()**" function.

   Now coming to using the Trace library for getting internal hidden states, it turns out that there is a little caveat for ensuring that only sentence-2 hidden states are used in the classifier that I designed for the latter part. This is because of the way the GPT2 tokenizer is tokenizing each sentence. For example, full-stop ('.') is a separate token during tokenization. In order to avoid any ambiguity, I tokenized the combined sentence, transformed_sent, as well as only sentence-2. After getting the hidden states from GPT2 using the Trace library, I calculated the effective number of tokens in the transformed sentence if we were to leave out sentence 2 by subtracting the number of tokens in sentence 2 from the number of tokens in transformed sentence. Finally, I used the quantity to slice the hidden states tensor in the 2nd dimension to get the hidden states of sentence 2 as it decoded over the entire transformed sentence.

This is implemented in functions “**get_representation()**” and “**get_all_hidden_states**”. These hidden states are stored as a pickle object to be used later while training a classifier.

3. Train a classifier

   We were allowed to have 500 training examples. I tried with 50 validation and 50 testing examples. The loss function was a bit unstable in the beginning so I used 100 validation and 100 testing examples. Although this does not seem to have much of an impact on training the classifier as well as on the final classification error rate.

   As was stated in the problem description, I used padding to make sure that all the sequences are of equal length. This is especially useful in my case because I will be using NNs as my classifier for which I need to know the input dimensions after flattening each example. This is implemented in the function “**get_padded_sequence()**”. It is implemented by taking note of the maximum sequence size in all of the training, validation and testing examples and later padding the smaller ones with 0s. Finally, data loaders are made in batches of size 25 after having created a dataset “**Data_X_y**” class.

   While designing the architecture of the neural network, I started with 3 layers but was getting a modest accuracy of around 41% on test data. However, when moving on to 4 layers, accuracy improved to around 54%.

While designing the number of hidden states in each layer, I made the assumption that the number of hidden units does not decrease too rapidly as this will create a lot of pressure on such a layer to classify labels using high-dimensional features to classify them now using low-dimensional features. The input dimension of feature vectors after flattening is around 20,000. With four layers, I thought of having the following units in each layer: [2000, 200, 20, 3]. The last dimension is fixed because we are solving a 3-class classification problem. However, this was not giving good results. The accuracy of around 54% (my best possible accuracy) was achieved with the following size of hidden units: [1024, 1024, 512, 3]. I believe this is because of the fact that the earlier model with hidden units as [2000, 200, 20, 3] was a bit simplistic and not able to capture the patterns in the dataset accurately. I also used ReLU activation function so that the model is more articulate in capturing complex patterns if the data is not linearly separable. Another important observation was that the accuracy does not seem to be increasing with more layers. Hence, I stuck with 4 layers in all the later experiments.

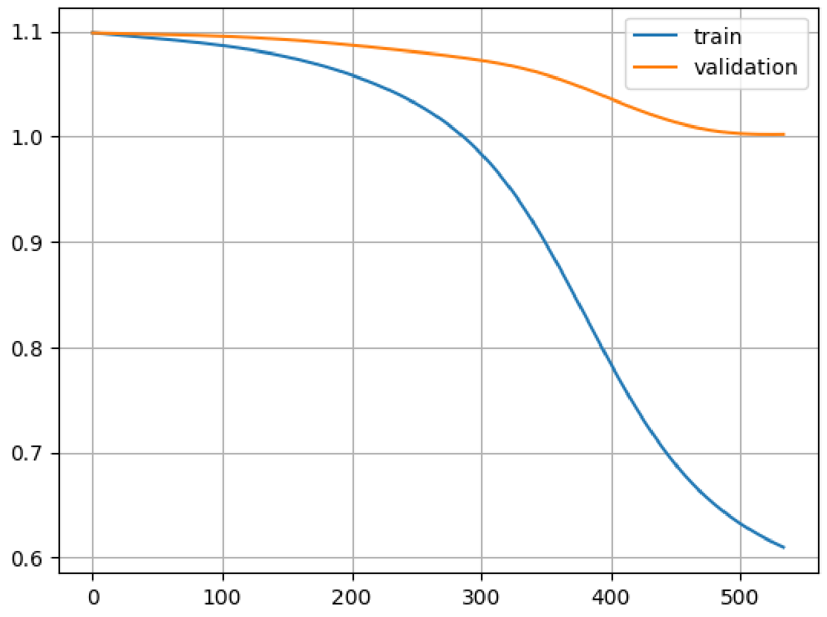
I am using Cross-Entropy loss while training the neural networks as this is a standard loss function to use for classification problems and has all nice and well defined characteristics. I used the usual 0-1 loss for checking final evaluation performance. Basically, if our predicted label matched the target label, we get 1 point and 0 otherwise. The average number of points we get for our testing data is the measure of how good our classifier does on unseen examples. I used this metric because it is very intuitive and a common metric people use for classification problems. Other metrics like precision and recall can be used as well.

In order to improve the model's generalization, I used dropout in each of the 4 layers before ReLU with a standard dropout probability of 0.2.

Following is the comparison between the NNs with same architecture ([1024, 1024, 512, 3]) with and without dropout:

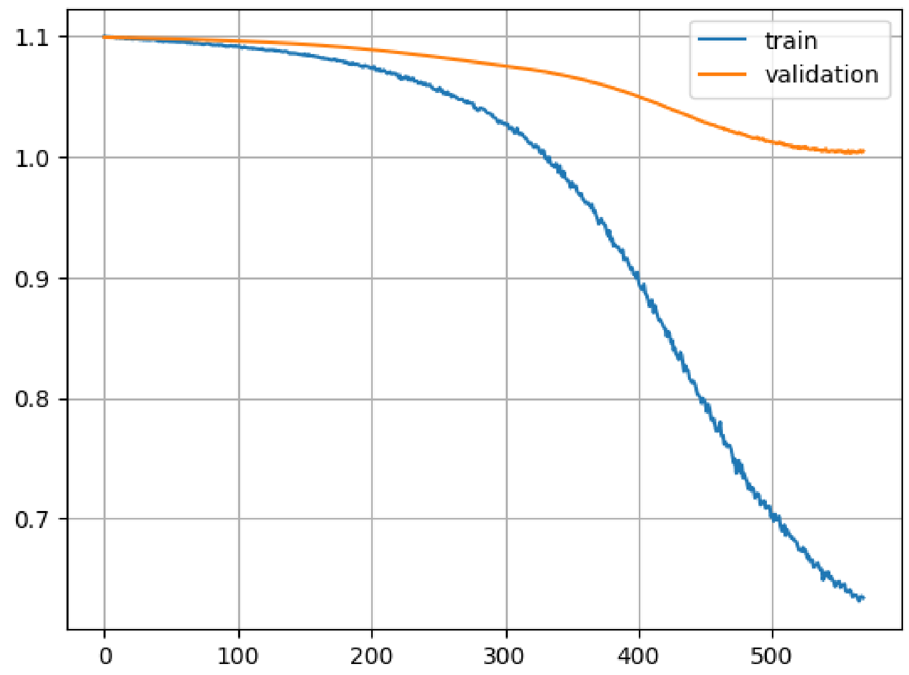
a) w/o Dropout: **Test accuracy = 54%**

Following is the plot of training and validation losses v/s # of epochs w/o Dropout.



b) w/ Dropout: **Test accuracy = 56%**

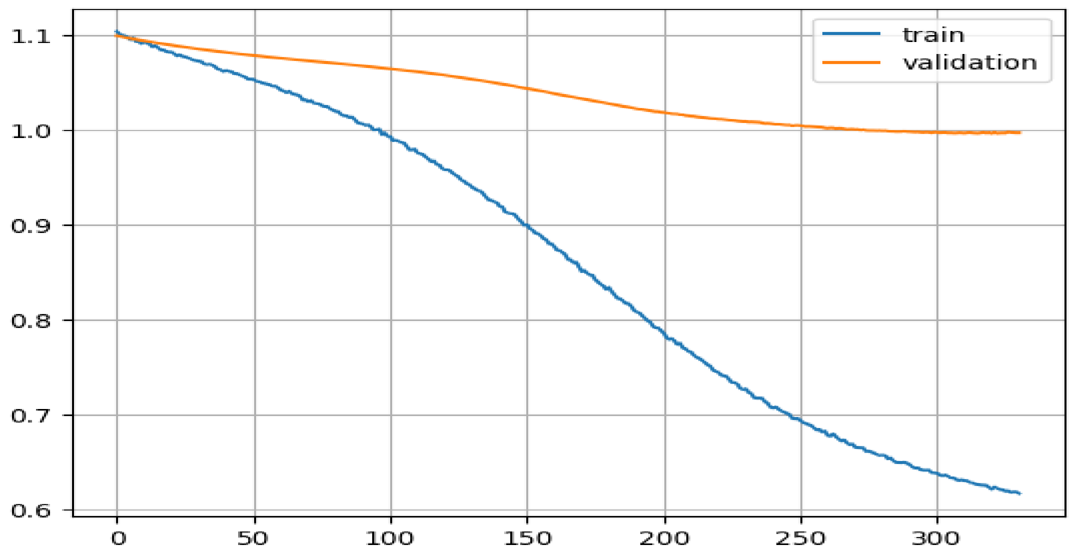
Following is the plot of training and validation losses v/s # of epochs w/ Dropout.



The accuracy improves by 2% while using dropout. It is not a significant improvement but I decided to keep dropout in all the later experiments as dropout improves generalization and makes the model more robust by avoiding overfitting the training data unnecessarily. In addition to this, I am using a patience counter while training. If the validation loss does not improve for the number of times this counter was set in the beginning compared to the best validation accuracy thus far, then the training stops and the best model is returned. This prevents our model from capturing idiosyncratic patterns in the training data.

Finally, taking inspiration from the original paper (by Bengio et. al) in the problem set, I experimented with training the same architecture classifier without any activation function.

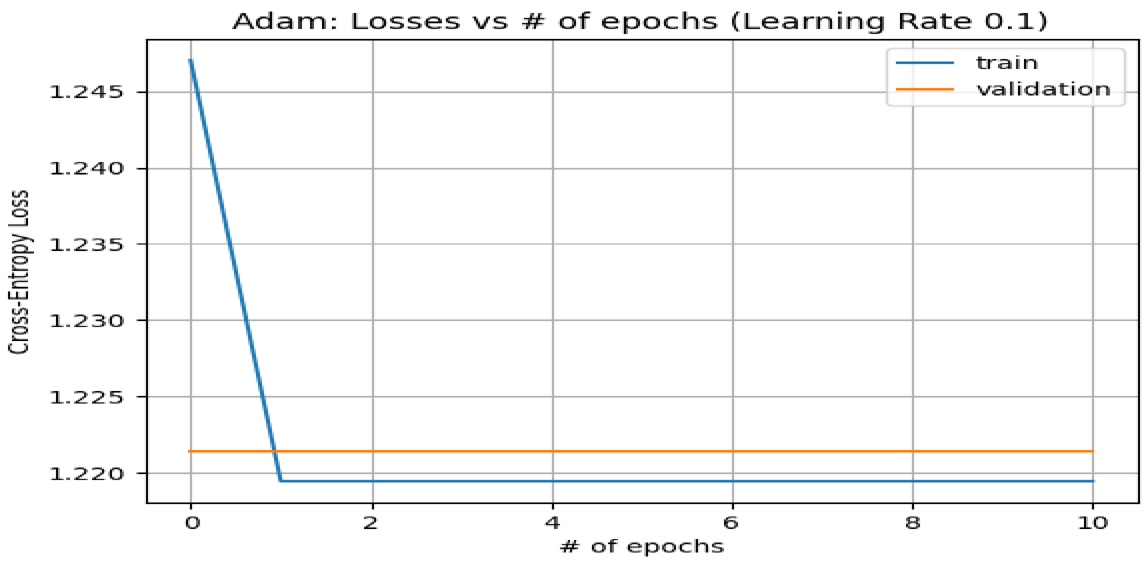
I have tried different forms of multi-layer neural networks. It turns out that the test accuracy is **53%** which is comparable to all the other cases we dealt with. Following is the training/validation losses v/s # of epochs plot for the same:

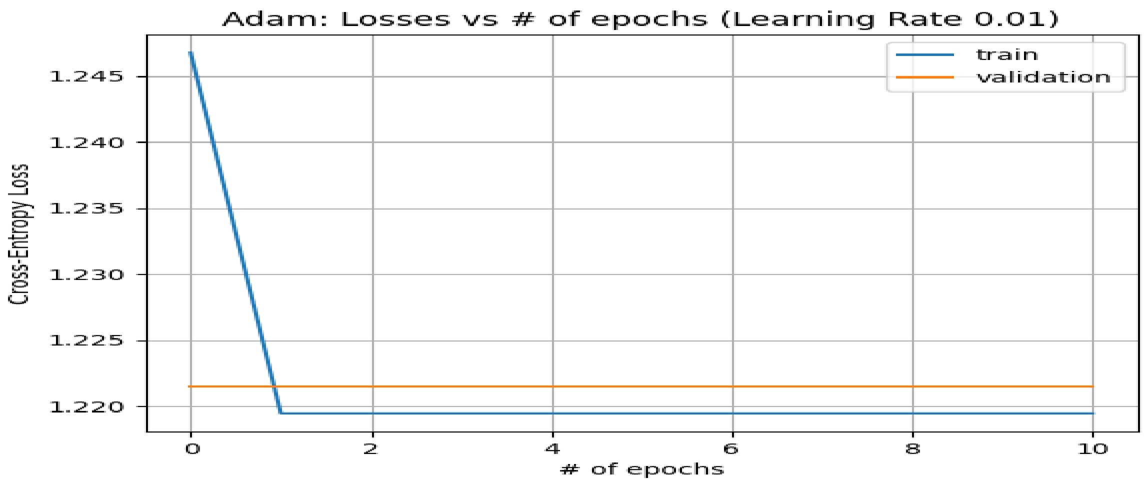


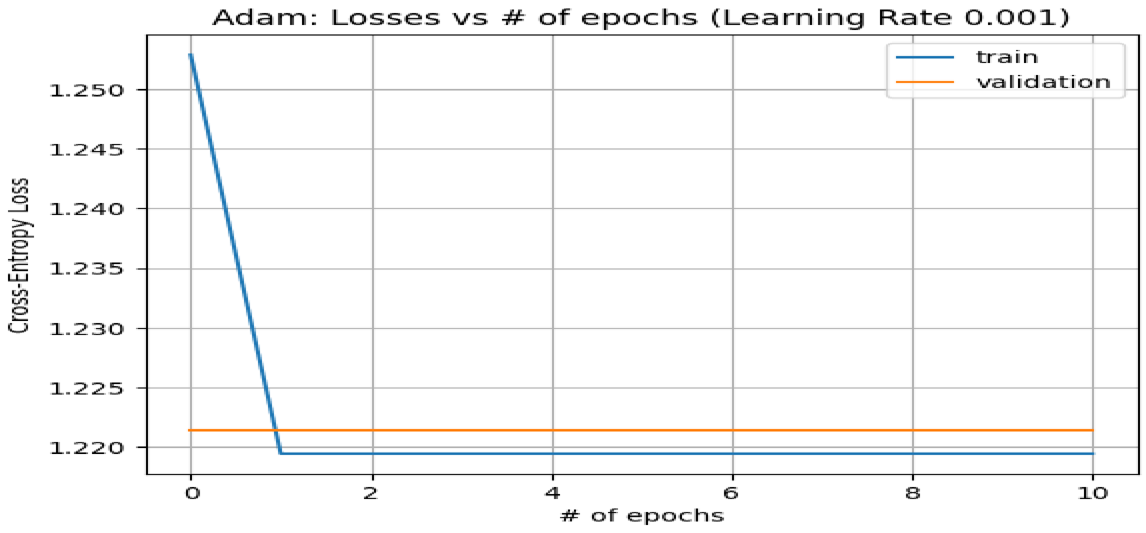
These different architectures are implemented as different classes in the code. These were NNs w/ ReLU: **NeuralNetwork(nn.Module)**, NNs w/ ReLU w/ Dropout: **NeuralNetworkDropout(nn.Module)**, and NNs w/o ReLU w/ Dropout: **NeuralNetwork_No_ReLU(nn.Module)**.

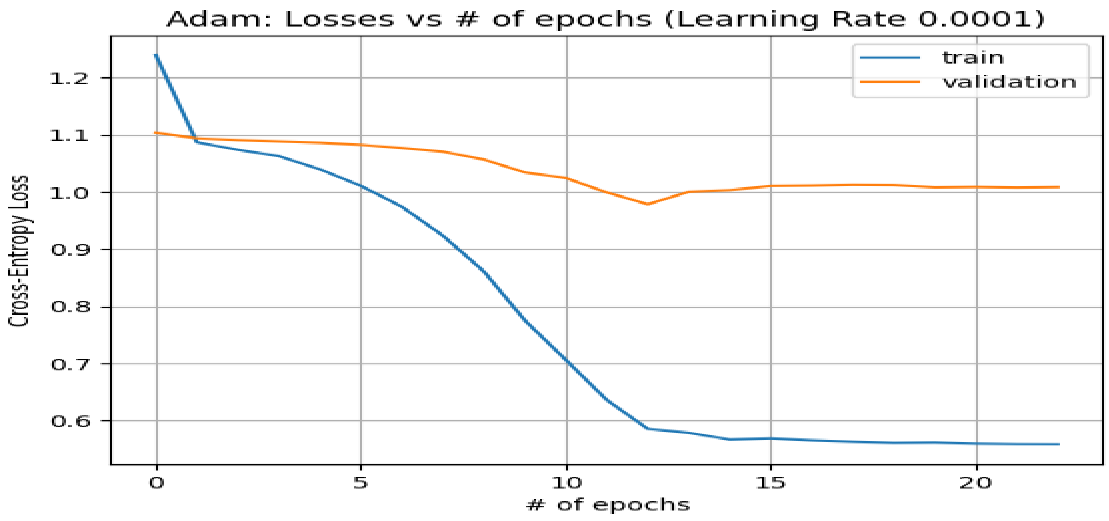
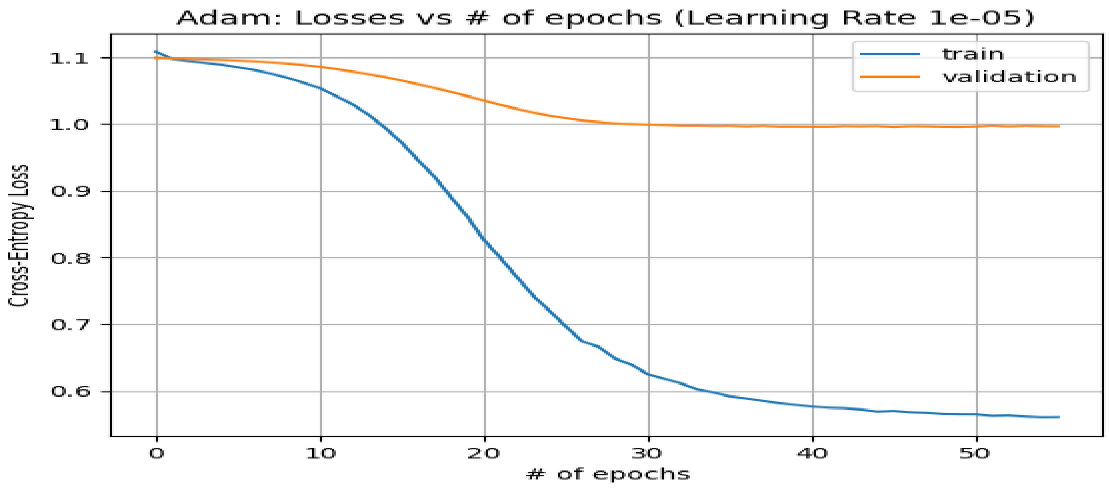
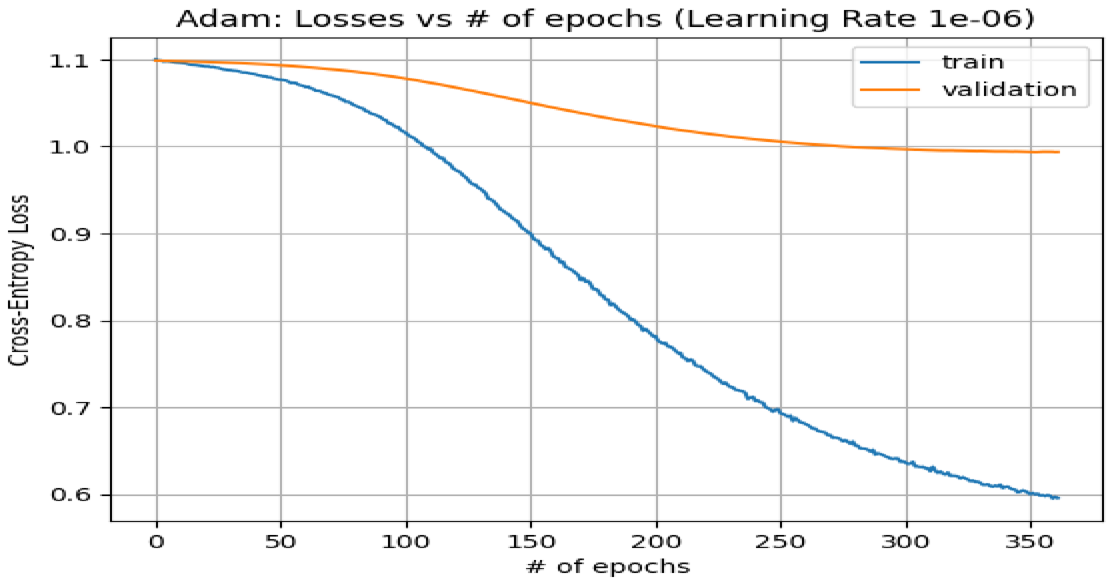
Finally, I tried Adam optimizer. The convergence seems to be much faster compared to the standard SGD optimizer. However, it is very susceptible to learning rate. For a learning rate of 10^{-3}, the model finishes in 10 epochs and does not seem to be learning much. This is evident from the test accuracy of around **33%**. However, with a learning rate of 10^{-5}, the model finishes training in around 50 epochs with test accuracy of **53%**.

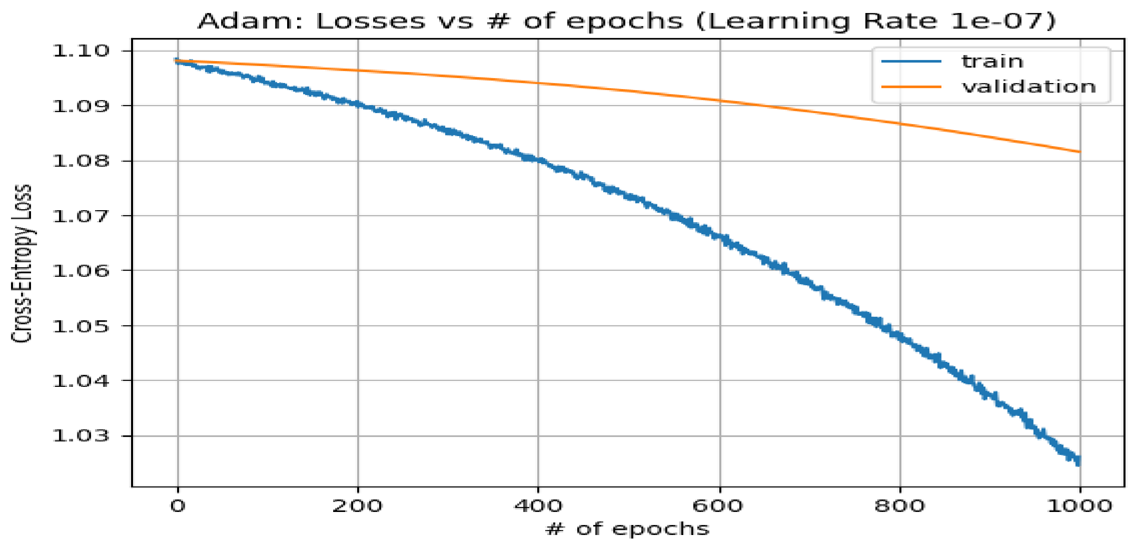
Following are the training and validation loss plots for Adam optimizer v/s # of epochs:


Adam: Losses vs # of epochs (Learning Rate 0.1)
train
validation
Cross-Entropy Loss
1.245
1.240
1.235
1.230
1.225
1.220
0
2
4
6
8
10
# of epochs


Adam: Losses vs # of epochs (Learning Rate 0.01)
train
validation
Cross-Entropy Loss
1.245
1.240
1.235
1.230
1.225
1.220
0
2
4
6
8
10
# of epochs

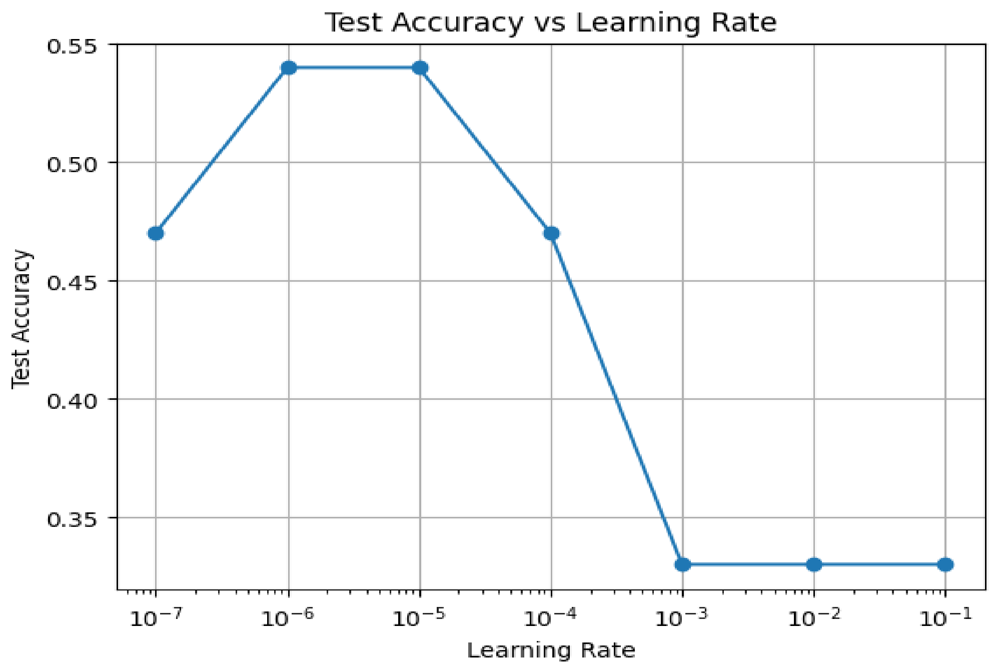

Adam: Losses vs # of epochs (Learning Rate 0.001)
train
validation
Cross-Entropy Loss
1.250
1.245
1.240
1.235
1.230
1.225
1.220
0
2
4
6
8
10
# of epochs


Adam: Losses vs # of epochs (Learning Rate 0.0001)
Cross-Entropy Loss
# of epochs
train
validation
0
5
10
15
20
0.6
0.7
0.8
0.9
1.0
1.1
1.2

Adam: Losses vs # of epochs (Learning Rate 1e-05)
Cross-Entropy Loss
# of epochs
train
validation
0
10
20
30
40
50
0.6
0.7
0.8
0.9
1.0
1.1

Adam: Losses vs # of epochs (Learning Rate 1e-06)
Cross-Entropy Loss
# of epochs
train
validation
0
50
100
150
200
250
300
350
0.6
0.7
0.8
0.9
1.0
1.1



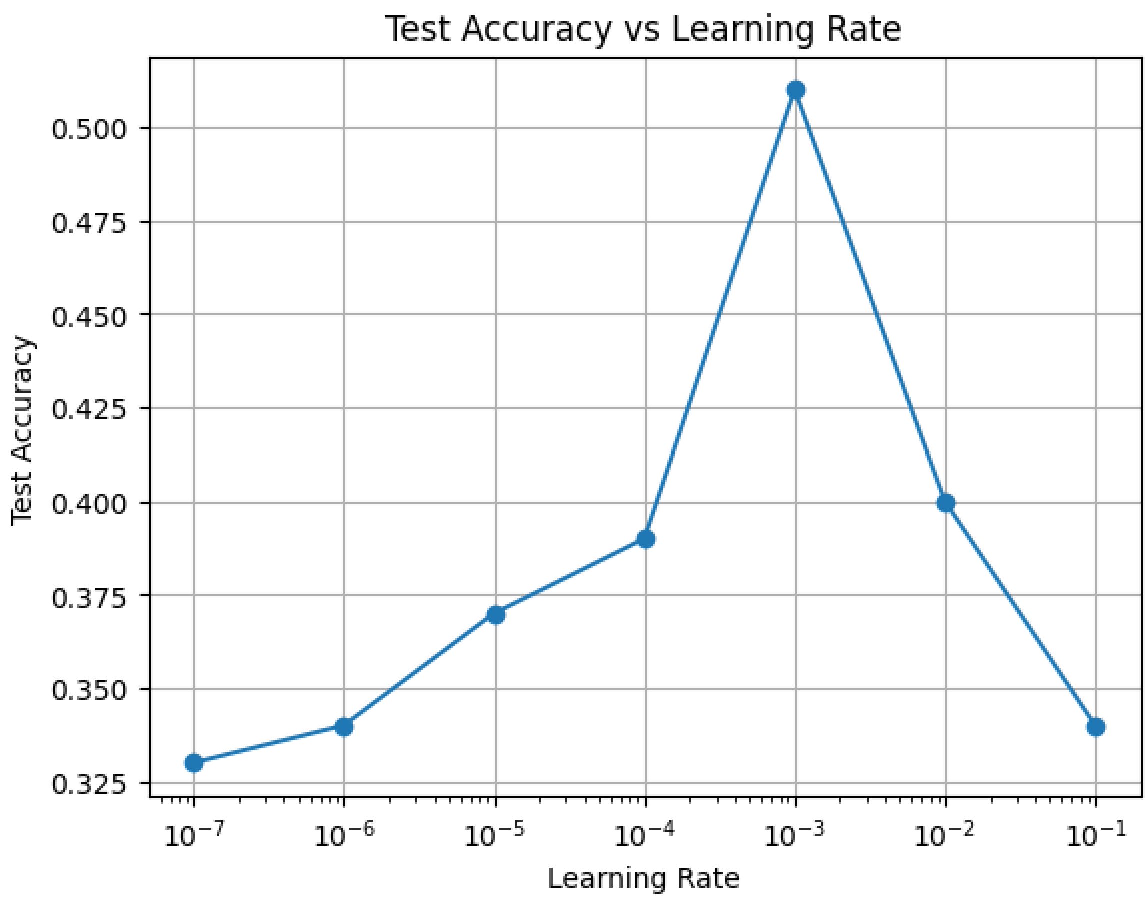
The accuracy of Adam optimizer in each of the cases are [0.32999998, 0.32999998, 0.32999998, 0.46999997, 0.53999996, 0.53999996, 0.46999997] for learning rates from 10^{-1}, 10^{-2}, …, 10^{-7}.

The test accuracy plot for Adam optimizer is shown below:

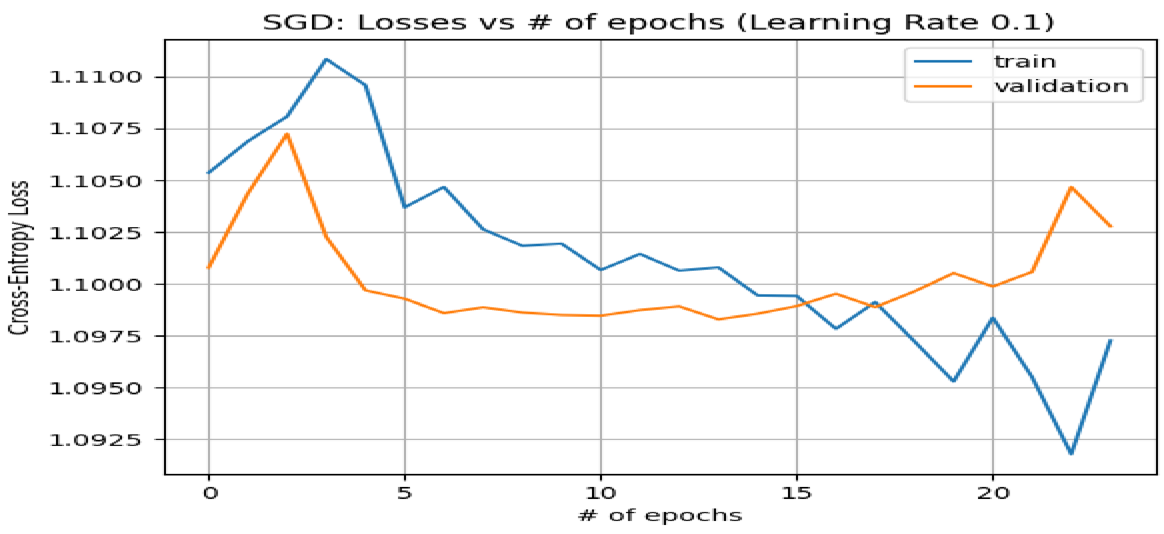


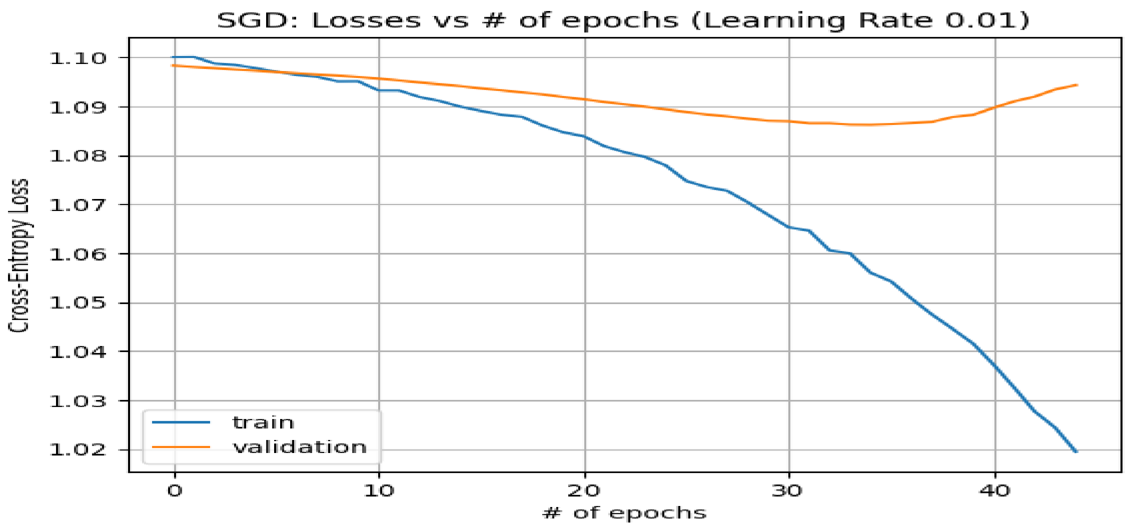
For usual SGD optimizer the accuracies are [0.34, 0.39999998, 0.51, 0.39, 0.36999997, 0.33999997, 0.32999998] for learning rates from 10^{-1}, 10^{-2}, …, 10^{-7}.

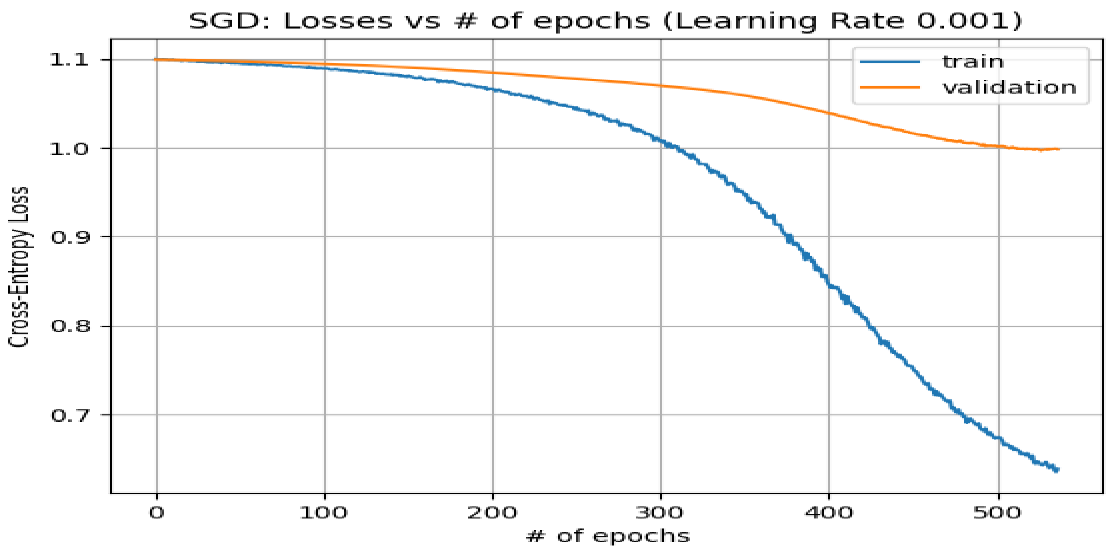
The test accuracy plot for Adam optimizer is shown below:

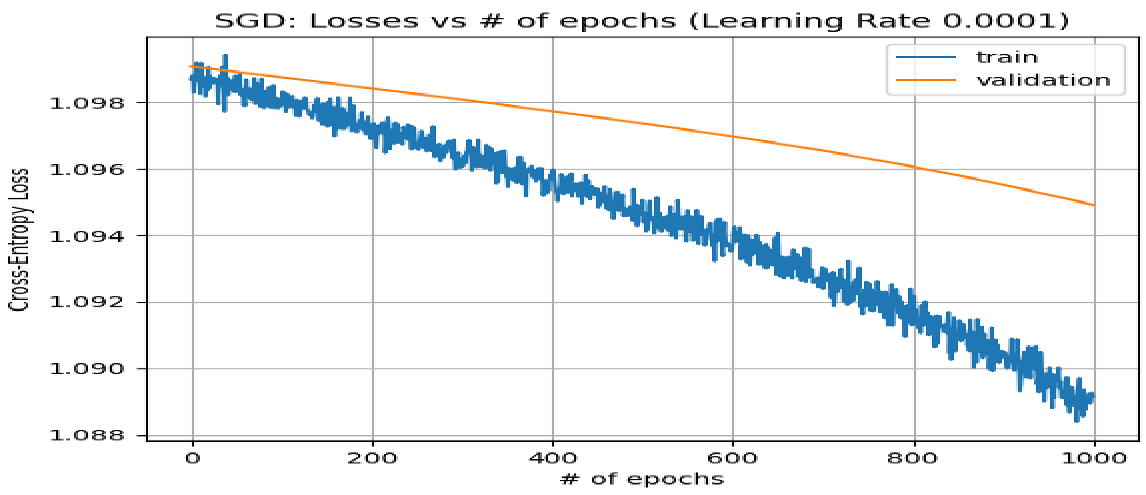


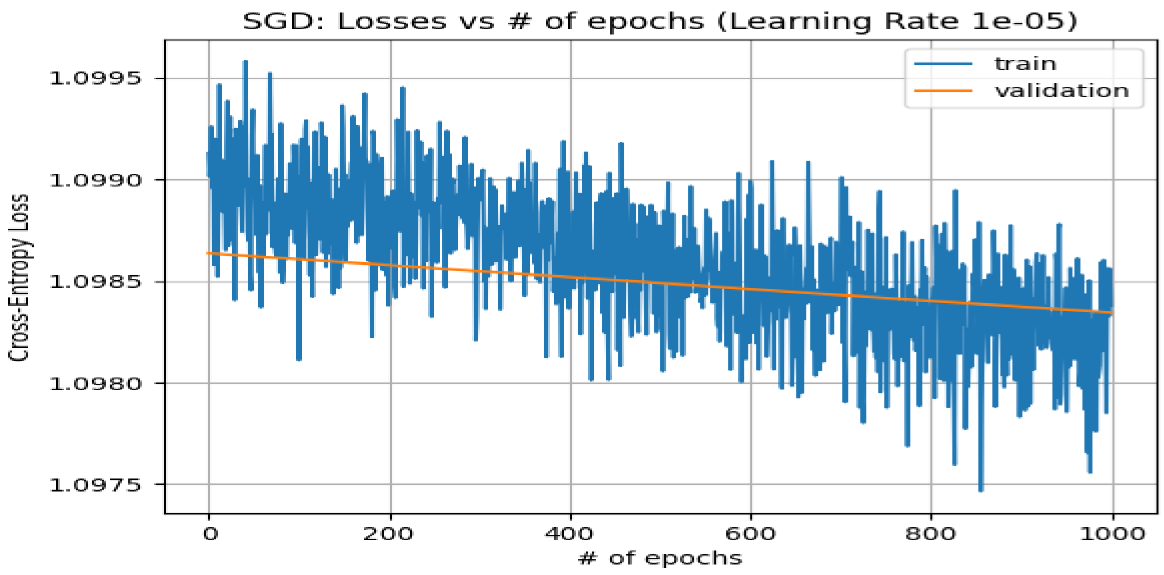
Following are the training and validation plots for losses in SGD optimizer v/s # of epochs:

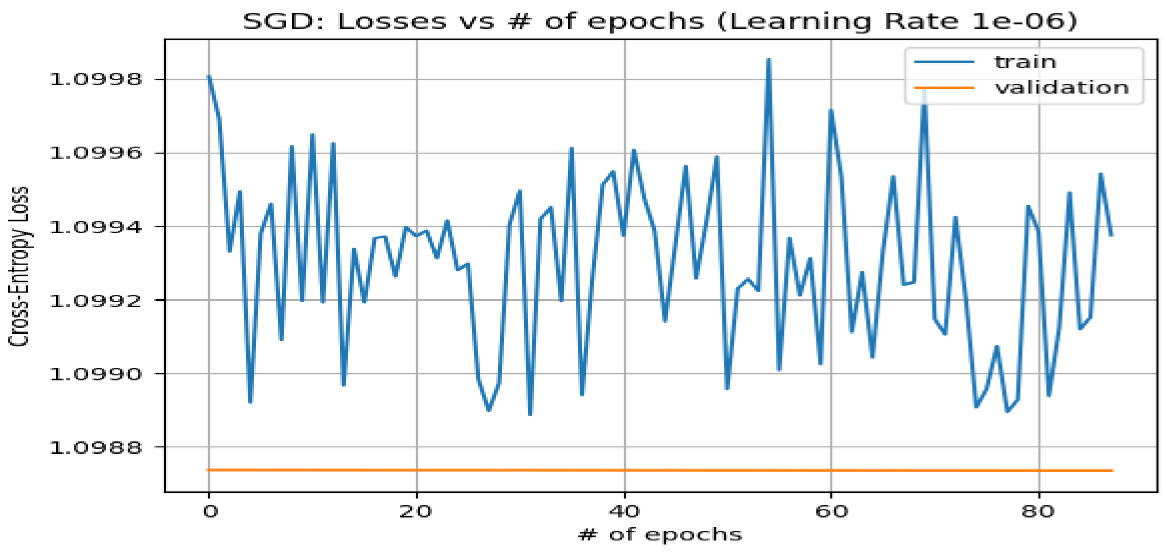

SGD: Losses vs # of epochs (Learning Rate 0.1)
Cross-Entropy Loss
# of epochs
train
validation
1.1100
1.1075
1.1050
1.1025
1.1000
1.0975
1.0950
1.0925
0
5
10
15
20

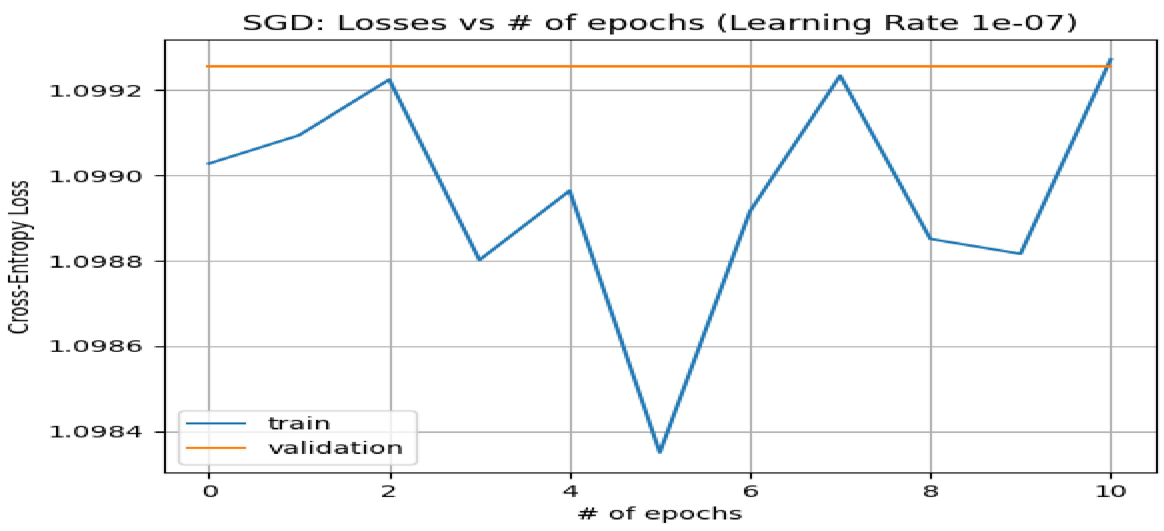

SGD: Losses vs # of epochs (Learning Rate 0.01)
Cross-Entropy Loss
# of epochs
train
validation
1.10
1.09
1.08
1.07
1.06
1.05
1.04
1.03
1.02
0
10
20
30
40


SGD: Losses vs # of epochs (Learning Rate 0.001)
Cross-Entropy Loss
# of epochs
train
validation
1.1
1.0
0.9
0.8
0.7
0
100
200
300
400
500


SGD: Losses vs # of epochs (Learning Rate 0.0001)
train
validation
Cross-Entropy Loss
# of epochs
1.098
1.096
1.094
1.092
1.090
1.088
0
200
400
600
800
1000


SGD: Losses vs # of epochs (Learning Rate 1e-05)
train
validation
Cross-Entropy Loss
# of epochs
1.0995
1.0990
1.0985
1.0980
1.0975
0
200
400
600
800
1000


SGD: Losses vs # of epochs (Learning Rate 1e-06)
train
validation
Cross-Entropy Loss
# of epochs
1.0998
1.0996
1.0994
1.0992
1.0990
1.0988
0
20
40
60
80



Summary: We got decent results with around 56% with SGD optimizer when the learning rate was 10^{-3} and around 53% with Adam optimizer when the learning rate was 10^{-5}.

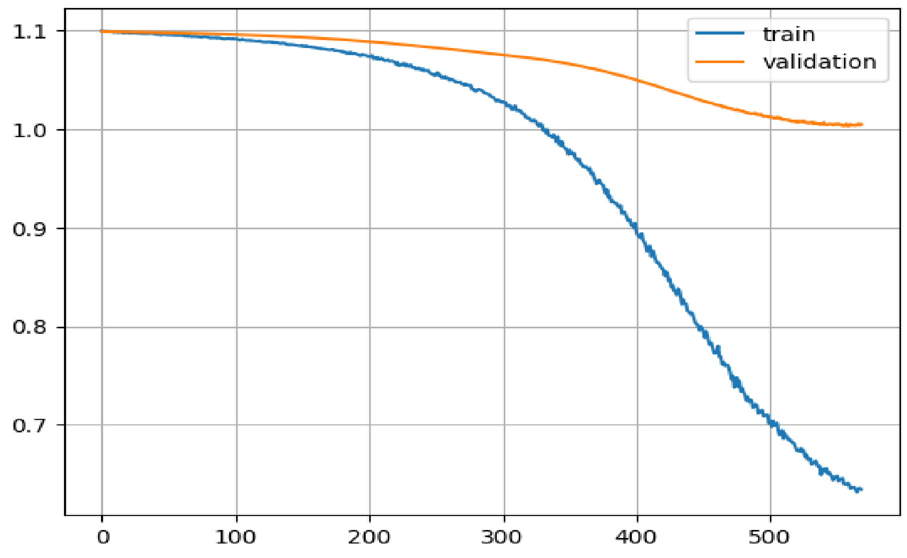
Reflecting back on the experiments, our classifier achieved 56% accuracy and hence it is obviously better than random guessing (33%), but the assessment of its performance depends on various factors such as the complexity of the task, the quality of the data, and the baseline performance of other methods. For entailment prediction, 56% accuracy might indicate some understanding or learning, but it might not be sufficient for robust real-world applications where higher accuracy is desired.

Regarding the saliency of entailment relations in GPT-2, interpreting the neural network's knowledge or understanding of contradictions or entailments based solely on the classification accuracy can be challenging. A classifier trained on hidden states can capture some implicit information but may not explicitly represent the "understanding" of entailment relations in GPT-2. To gain deeper insights, we will explore layer analysis in the next section. Another very useful yet difficult way would be to visualize or interpret the learned representations to understand what aspects of the input are being captured by the model for entailment detection.

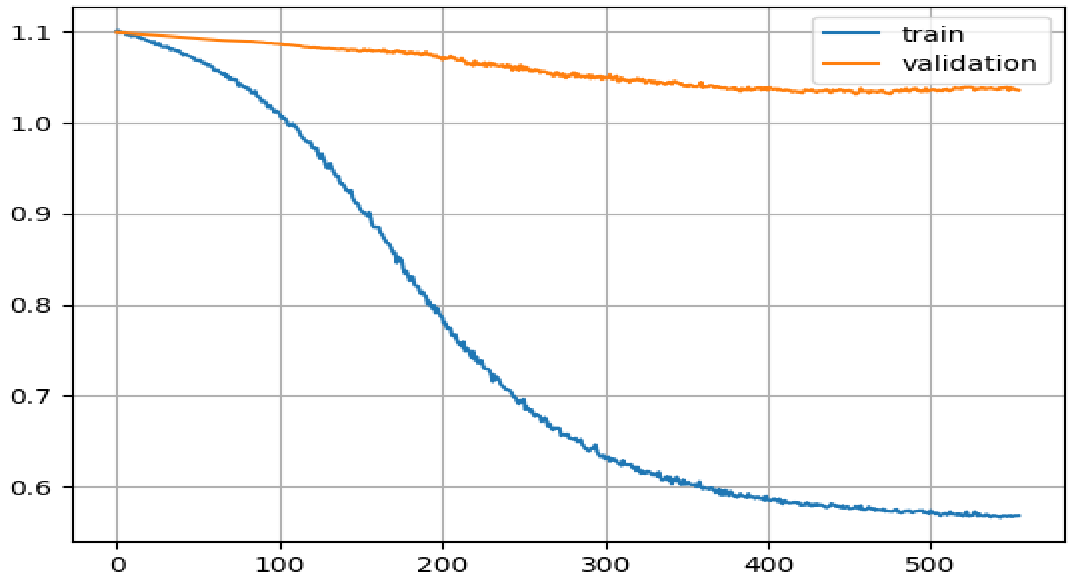
4. Probe other layers

   In this section, I experimented with different MLP layers of GPT-2 using SGD optimizer. I got the following results:

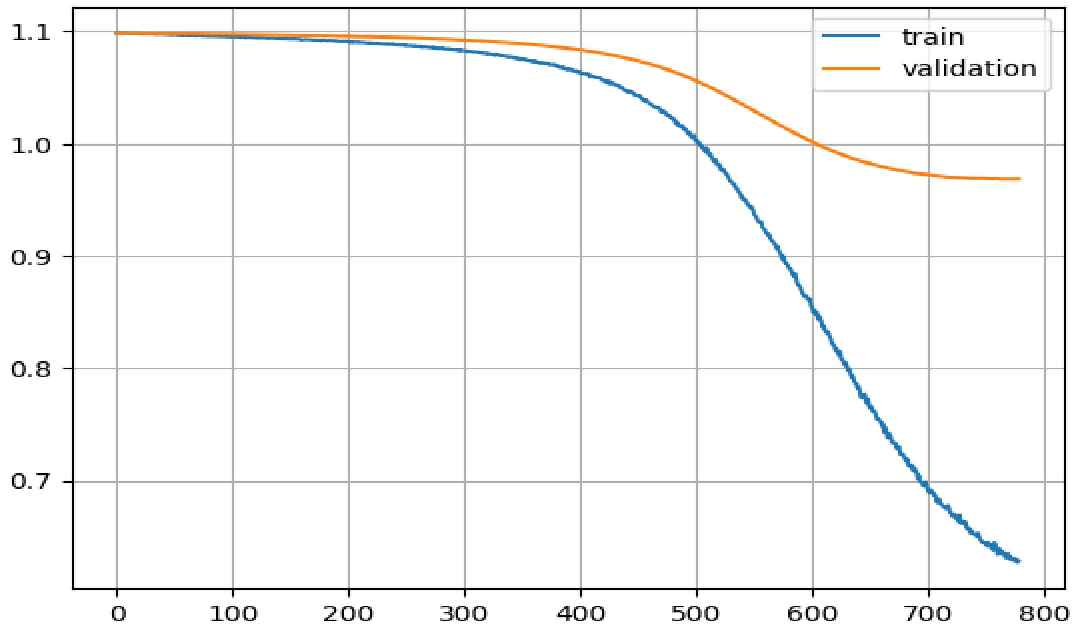
   a. **Layer 1:** Test accuracy = 56% (Training-validation losses v/s # of epochs is shown).



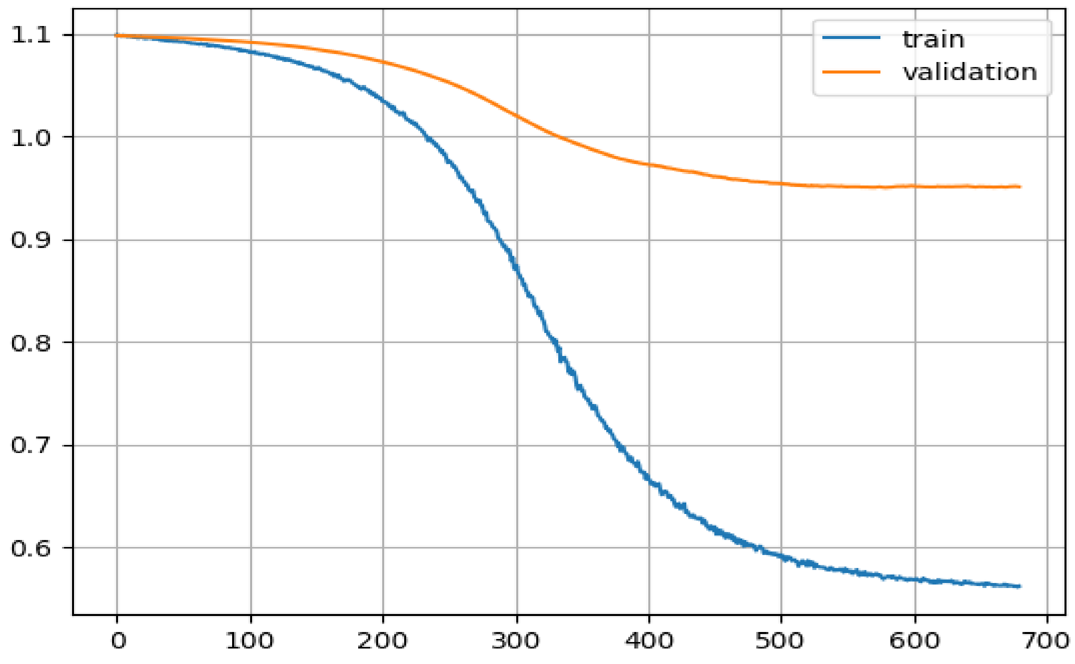
   b. **Layer 4:** Test accuracy = 55.9% (Training-validation losses v/s # of epochs is shown).



c. **Layer 6:** Test accuracy = 61% (Training-validation losses v/s # of epochs is shown).



d. **Layer 10:** Test accuracy = 64% (Training-validation losses v/s # of epochs is shown).



We see that the test accuracy increases if we take hidden states from later layers in the GPT2 model. In fact, the accuracy goes up to 64% if we use hidden states from the 10th MLP layer.

We can say the lower layers contain basic or low-level features but may lack higher-level representations needed for accurate predictions in our task. In deeper layers, the model is able to understand the sentence it is decoding in a much better way by repeatedly combining the context vectors it gathered from previous layers. Deeper layers might encode more abstract, task-specific, or high-level features that are crucial for better performance in decoding whether the sentence followed by prompt is an entailment/contradiction/neutral to sentence 1.

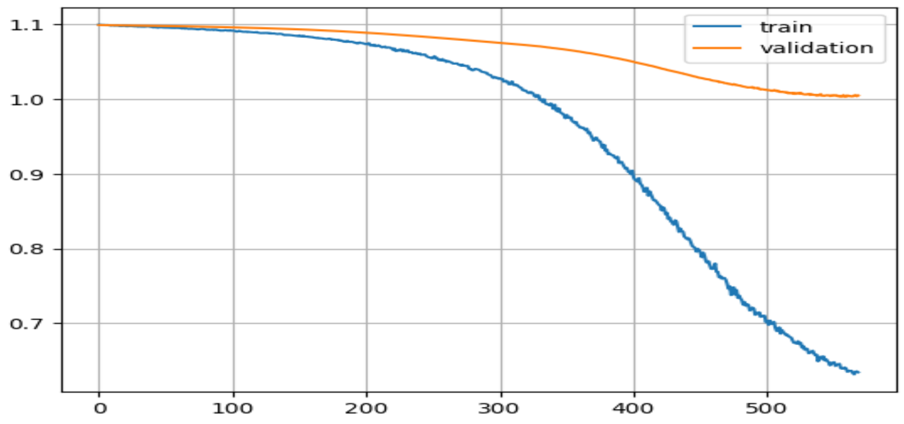
5. Bonus Task 1

I tried two different separators in between sentence 1 and sentence 2. I got the following results:

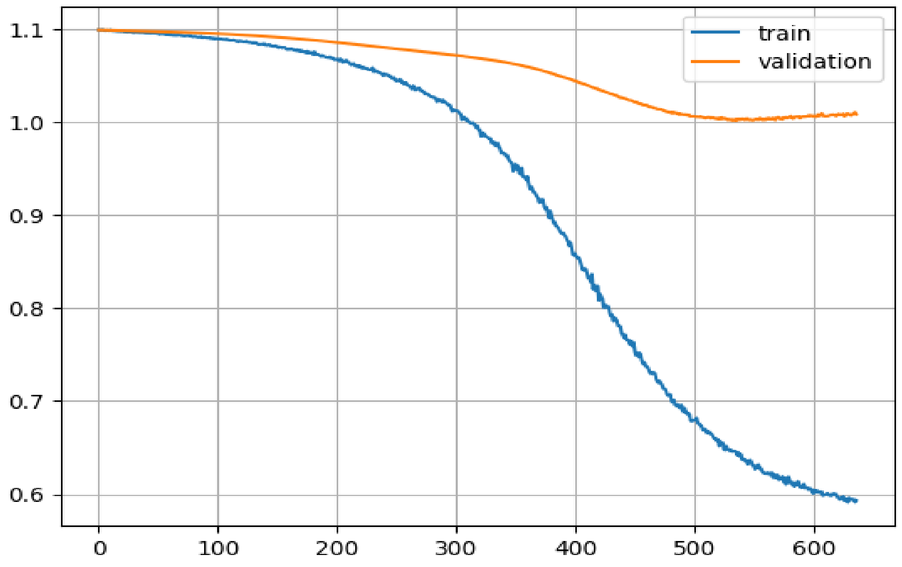
a. Accuracy for prompts of this type **<sent1> This has the same meaning as: <sent2>** equals **56%**.
b. Accuracy for prompts of this type **<sent1> [CAUSED] <sent>** equals **54%**.
c. Accuracy for prompts of this type **<sent1> [Has Different Meaning] <sent>** equals **52%**.

These are the training plots for the 3 cases:

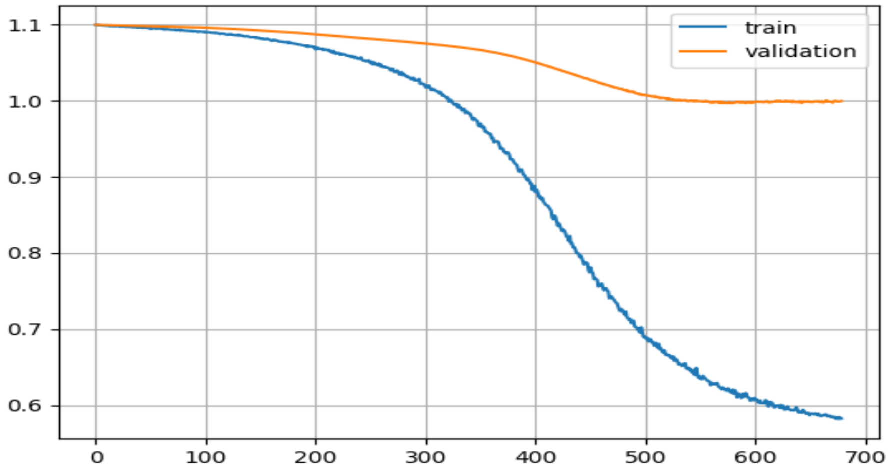
a. Prompt = This has the same meaning as



b. Prompt = [CAUSED]



c. Prompt = [Has Different Meaning]



Important takeaways from this experiment are:

a. **Accuracy:** The prompt "This has the same meaning as" resulted in the highest accuracy (56%) among the tested prompts. Other prompts ("[CAUSED]" and "[Has Different Meaning]") resulted in slightly lower accuracies of 54% and 52%, respectively.

b. **Observation**: The performance differences among the prompts suggest that the choice of prompt impacts the model's ability to classify entailment/contradiction/neutral relations. The prompt "This has the same meaning as" might provide clearer or more conducive contextual information for the model, resulting in slightly better performance compared to the other prompts. Alternative prompts like "[CAUSED]" or "[Has Different Meaning]" might introduce different linguistic contexts or complexities that could affect the model's understanding and classification accuracy.

c. **Intuition for different test accuracies**: The wording and context provided by each prompt might influence how well the model captures the intended relationships between the sentences. Some prompts might offer clearer or more explicit cues for entailment or contradiction, aiding the classifiers trained on top of them in making more accurate classifications. The complexity or ambiguity introduced by certain prompts might make it more challenging for the model to discern entailment/contradiction relationships accurately.

6. Bonus Task 2

   LLMs have a lot of parameters going up to as many as billions. We experimented with the GPT-2 model which has around 117M parameters. Having those many parameters gives LLMs the ability to capture very complex morphological as well as semantic contexts. They can understand the text in a much better way because of more layers as well as attention heads. They are also expected to have amazing generalization capabilities. However, there is no direct proven relationship between model size and better natural language understanding. In fact, using large language models for determining entailment or contradiction have to involve use of more complicated classifiers to get an idea of whether the LLM “knows” what it is outputting. This is primarily due to the fact that LLMs will also have larger embedding dimension size as well and hence using less complicated classifiers would lead to overfitting. In conclusion, Larger models may be able to better understand inconsistencies, but it is difficult to gauge how much better they will actually be able to understand and how this would affect practice. Due to their expanded parameters, these models are excellent at extracting complex features and have the potential to detect contradictions more accurately.

7. Bonus Task 3

   Because GPT-2 was trained on a wide variety of online texts showcasing a wide range of linguistic settings and styles, it is possible that subtle linkages and contradictions were incorporated into the training process. There were most likely a lot of textual instances in this rich training set that contradicted or supported one another. Consequently, GPT-2 acquired the ability to encode these complex patterns and relationships within its internal workings during its training. Thus, during its training process, the model internalized these environmental patterns and linkages. These features may have been more apparent inside the internal structure of the model due to the training data's repeated repetition of contradicting or supportive material. This could make it easier for the model to spot similar trends or inconsistencies while creating or managing text.