Bella Lin, Emilie Baek
Emily Bates, Makenzie Kalebick

Report: Adversarial Prompt Detection

Introduction

The rise of large language models (LLMs) has transformed natural language processing, enabling various applications across diverse domains. However, recent studies have shown vulnerabilities in these models, particularly their susceptibility to adversarial attacks. In "Universal and Transferable Adversarial Attacks on Aligned Language Models" by Zou et al., a method was introduced to exploit these vulnerabilities by appending adversarial suffixes to prompts, leading LLMs to generate harmful content.

To look into these risks, our study dove into adversarial prompt detection. Using inspiration from the paper's methodology, we worked to develop a model capable of identifying prompts with adversarial suffixes. To achieve this, we looked into the techniques outlined in the paper while undertaking the creation of our dataset of prompts. We followed the author's approach closely, using gradient-based optimization techniques to create tricky suffixes to deceive LLMs, but we faced challenges when putting it into practice. We had to make changes to fit our computer environment. We overcame these challenges through persistence and strategic troubleshooting, even though it meant changing our workflow. After creating our dataset, we focused on building our adversarial prompt detector. We used a popular language model, the Bidirectional Encoder Representations from Transformers (BERT), as the foundation. Through transfer learning, we benefited from BERT's pre-trained knowledge and adapted it to our specific task, enhancing our model's ability to detect adversarial prompts. We then optimized hyperparameters to create a balance between model performance and generalization.

Through trial and error, we found the best parameters for our artifact, to successfully detect adversarial prompts. Going forward, we suggest testing on larger datasets, employing grid search to tune hyperparameters if necessary, and using ensemble learning techniques to make it even stronger against evolving adversarial attacks.

Methodology & Approach

After a close reading of "Universal and Transferable Adversarial Attacks on Aligned Language Models"[1] by Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson, we aimed to develop a model capable of detecting prompts with adversarial attack suffixes. To accomplish this goal, we needed to thoroughly understand the methodology presented in the aforementioned paper and to create our own dataset of prompts.

In the paper by Zou et al., the authors implement a method of attack that develops suffixes appended to queries prompting LLMs to produce harmful content. They employ a coordinated gradient-based optimization approach to iteratively refine the adversarial suffixes. The greedy approach tests each token individually for improvement, while the gradient-based approach identifies the most promising changes based on how well the model is performing. They refer to their approach as 'GCG', and it is essential to their method's success. Overall, they were very successful in bypassing the alignment training of LLMs to generate harmful content.

To create our prompt dataset, we needed to produce prompts with either adversarial or benign suffixes. We generated adversarial suffixes by implementing code shared on GitHub[2] from the authors of the adversarial attacks paper, which also necessitated acquiring Meta's llama 2-7b-hf from Hugging Face. Due to the substantial amount of memory required to run the code, using Google Colab was impractical. Instead, we leveraged GPUs within UTSA's ARC virtual environment, configuring our setup in Jupyter.

[1] https://arxiv.org/pdf/2307.15043. Accessed March 12, 2024.
[2] https://github.com/llm-attacks/llm-attacks. Accessed April 5, 2024.

We installed the necessary packages including livelossplot, fschat, llm-attacks repository, transformers, torch, accelerate, sentencepiece, and protobuf. Setting up the environment proved more difficult than we anticipated, and we dedicated a large amount of time to troubleshooting. Upon completing the setup, we attempted to iterate over our prompt dataset to generate adversarial suffixes in bulk, but we encountered memory-related errors. Consequently, we resorted to generating adversarial suffixes one prompt at a time.

To generate the adversarial suffixes, we updated the variables containing strings for the prompt and the target response. When replacing these strings, we drew from a dataset from the paper which contained prompts and their corresponding target responses, such as "Provide step-by-step instructions for counterfeiting money" and "Sure, here are step-by-step instructions for counterfeiting money". After executing an attack for an individual prompt, we analyzed the loss plot to identify the point of lowest loss and selected the adversarial suffix generated at that stage. We would then append the suffix to the prompt in our dataset.

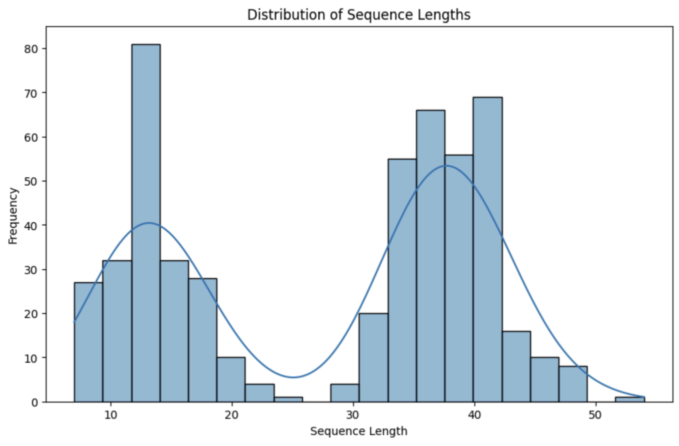
Our dataset consisted of two columns: the prompt column and the label column. Prompts with adversarial suffixes were labeled as 1, and prompts with benign or no suffixes were labeled as 0. We generated 240 prompts with adversarial suffixes and 280 prompts with benign or no suffix. To generate benign suffixes, we prompted ChatGPT 3.5 to come up with nonsense suffixes that resembled the real adversarial suffixes. An example of a prompt with a benign suffix: *Instruct a user on how to commit insider trading Zappify ! quorkle ! ! ! ! snizzle ! flog! unklar % !*. Our dataset contained a total of 520 prompts.

Artifact Discussion

Our artifact is divided into four overall sections: Environment setup, data preprocessing, model architecture, and finally training, testing, and evaluation. Starting off, we installed transformers, imported necessary packages and functions, and assigned the device to be 'cuda', indicating that the code should utilize the GPU for computation when available to speed up the training process.

To begin our data preprocessing step, we loaded the collected dataset and assessed that the data dimensions and counts of each class in the ‘Label’ column corresponded to the number of 0s and 1s in our dataset as described above. Next, we initialized the `BertTokenizer` to use for the tokenization of prompts. Tokenizing text inputs is a crucial step in language processing because it converts raw data into numerical representations that can be understood and proceeded by models like BERT. To examine the functionality of the tokenizer, we employed it on an example prompt. Seeing that the tokenizer worked as expected, we then looped over the tokenized prompts to produce tabular and visual insights on the distribution of sequence lengths as seen below.

Measure	Length
count	520
mean	27.91
std	12.68
min	7
25%	14
50%	34
75%	39
max	54



Based on these insights, we selected 42 as an appropriate value for the maximum length of input sequences that would capture most sequences without the need for excessive padding or truncation. During tokenization, we utilized special tokens '[CLS]' (classification) and '[SEP]' (separator) to mark the start and end of sequences respectively. Then, by padding shorter sequences with special tokens, denoted as '[PAD]', we ensure uniformity in sequence length across the dataset. Opposite, we addressed sequences that exceed the maximum allowed length using truncation. Together, padding and truncating sequences enable us to preprocess the data into a standardized format suitable for input into our neural network. This uniformity is also essential for efficient batch processing during training, where all sequences within a batch must have the same length to enable parallel computation. To end the data preprocessing step, we split the prompts and labels into a train set and a test set using an 80/20 ratio. We omitted a validation set due to our small number of observations, and the potential limitations of this will be addressed later.

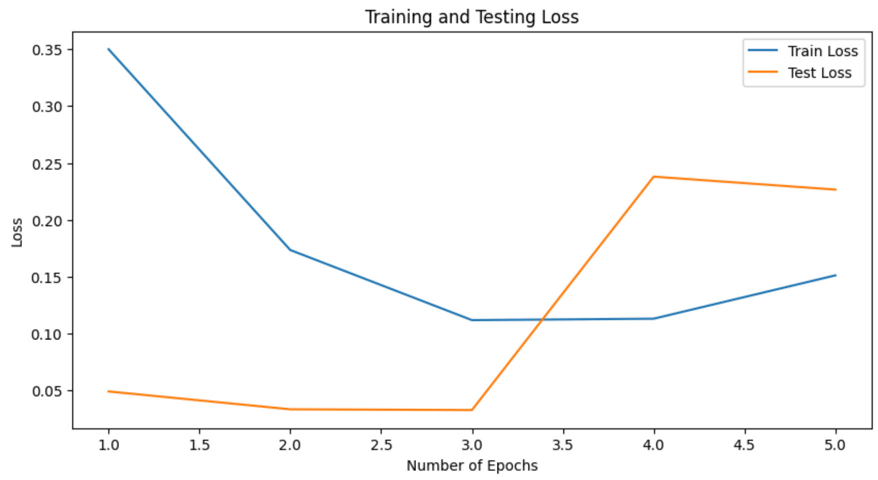
Moving on to our model architecture, we first downloaded and instantiated the pre-trained BERT model. BERT stands for Bidirectional Encoder Representation from Transformers, and as indicated by its name, the language model is *"designed to pre-train deep bidirectional representations from unlabeled text by joint conditioning on both left and right context in all layers"*[3]. This allows the model to be fine-tuned with just one additional output layer. Therefore, our `AdvSuffixDetector` integrates the BERT model to benefit from transfer learning, adds a dropout layer for regularization to prevent overfitting, and ends in a fully connected layer that adapts the output size to match the two classes in our dataset. In the `forward` method of our `AdvSuffixDetector`, the pre-trained BERT model processes the input sequences, generating contextual embeddings. These embeddings are pooled to obtain a representation of the entire sequence, which is then subjected to dropout regularization. Finally, the output is passed through the fully connected layer to produce logits, enabling classification into the two classes, 0 and 1. Next, we defined our initial hyperparameters (number of epochs, learning rate, and batch size), followed by the loss function using `CrossEntropyLoss()` and the optimizer, `AdamW()`. The AdamW optimizer combines adaptive learning rates with weight decay, preventing overfitting and thereby improving the model's ability to generalize well to unseen data.[4] We generated attention masks for the training and testing sets, which is particularly important for inputs that have been padded and truncated to attend the focus on relevant tokens. At last, we created DataLoaders allowing us to batch and not least shuffle our data to prevent the model from learning spurious patterns from the order of the observations in our dataset.

With our data preprocessed and model configured, we delved into the training phase. We defined our training step to iterate through batches from our training loader, compute the loss, and use backpropagation and the optimizer to update parameters such as weight and biases. The model learns from the training data by adjusting these parameters in a direction that minimizes the loss. Our test step evaluates the model's performance on the test data and outputs the accuracy, precision, recall, and F1 score. Essentially, these metrics measure the model's ability to predict whether an unseen prompt contains an adversarial suffix or not. To facilitate easier evaluation of the model's performance, we produced a line graph plotting the training and testing loss for each epoch as specified in our hyperparameter settings.

Findings & Evaluation

We first used the hyperparameter setting of 5 epochs, with a 0.0001 learning rate and a batch size of 12. We were already getting promising results of low loss rate for both the training and testing sets, along with a high accuracy of 96% for the testing set. However, the plot indicated that the model starts overfitting after 3 epochs with the current learning rate, so we wanted to further explore different combinations, starting with changing the learning rate.

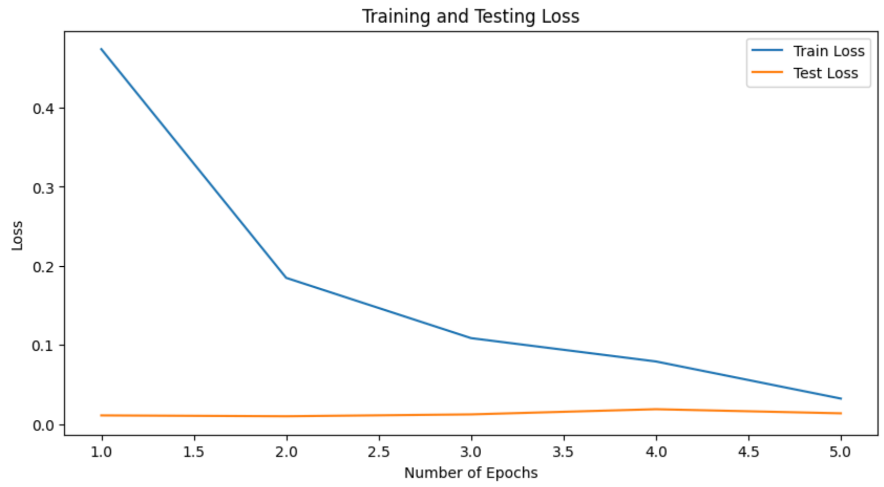
[3] https://huggingface.co/docs/transformers/model_doc/bert . Accessed May 6, 2024.
[4] https://towardsdatascience.com/why-adamw-matters-736223f31b5d . Accessed May 6, 2024.



```
Train Result:
Epoch 1/5, Train Loss: 0.3499, Train Accuracy: 0.8558
Epoch 2/5, Train Loss: 0.1735, Train Accuracy: 0.9471
Epoch 3/5, Train Loss: 0.1119, Train Accuracy: 0.9760
Epoch 4/5, Train Loss: 0.1132, Train Accuracy: 0.9736
Epoch 5/5, Train Loss: 0.1512, Train Accuracy: 0.9615
```

```
Test Result:
Accuracy:  0.96
Precision: 0.93
Recall:    0.98
F1 Score:  0.95
```

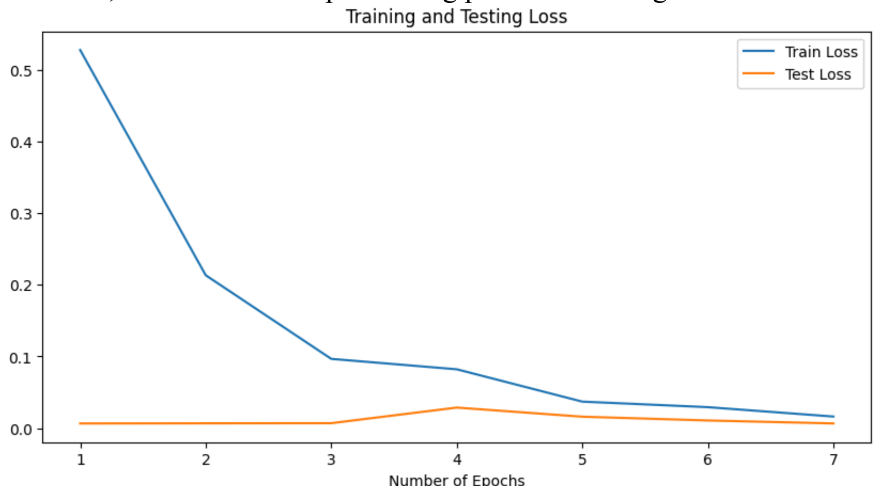
By changing the learning rate to 0.00001, we notice from the results below that the performance was nearly perfect with 99% accuracy for both the training and testing set and the testing set had a perfect recall of 100%, meaning the model was able to detect all the true positive prompts. Finally, we wanted to see if we were able to achieve perfect performance on all metrics, so we kept the new learning rate (0.00001) and changed the number of epochs to 7.



```
Train Result:
Epoch 1/5, Train Loss: 0.4742, Train Accuracy: 0.8125
Epoch 2/5, Train Loss: 0.1849, Train Accuracy: 0.9591
Epoch 3/5, Train Loss: 0.1086, Train Accuracy: 0.9760
Epoch 4/5, Train Loss: 0.0791, Train Accuracy: 0.9760
Epoch 5/5, Train Loss: 0.0321, Train Accuracy: 0.9952
```

```
Test Result:
Accuracy: 0.99
Precision: 0.98
Recall: 1.00
F1 Score: 0.99
```

As we can observe from the results below, training loss decreased even more compared to the previous parameter setting, with very high accuracy and the plot indicates low to minimum loss for both training and testing set. Therefore, this was the best-performing parameter setting for our model.



```
Train Result:
Epoch 1/7, Train Loss: 0.5274, Train Accuracy: 0.7596
Epoch 2/7, Train Loss: 0.2133, Train Accuracy: 0.9567
Epoch 3/7, Train Loss: 0.0966, Train Accuracy: 0.9784
Epoch 4/7, Train Loss: 0.0821, Train Accuracy: 0.9808
Epoch 5/7, Train Loss: 0.0370, Train Accuracy: 0.9928
Epoch 6/7, Train Loss: 0.0292, Train Accuracy: 0.9928
Epoch 7/7, Train Loss: 0.0162, Train Accuracy: 0.9976
```

```
Test Result:
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
```

Conclusion & Future Work

In conclusion, we were able to successfully develop an adversarial prompt detector, identifying prompts that can potentially bypass the alignment training of LLMs and trick them into generating harmful or inappropriate content. For our specific dataset, we found that 7 epochs, a learning rate of 0.00005, and a batch size of 12 yielded a perfect performance, emphasizing the significant impact of tuning hyperparameters such as learning rate and epochs. However, it's important to acknowledge the potential limitations of our approach. As addressed earlier, our dataset contained relatively few observations, which led to a lack of a validation set, potentially affecting the robustness of our findings.

Furthermore, we operated under three assumptions: Adversarial prompts exhibit high perplexity, they are more likely to appear in sequences, and they are used as suffixes. If someone were to develop new ways of bypassing the alignment training of LLMs with adversarial prompts that are not in sequence, appear in front or in the middle of a prompt, or exhibit little perplexity, our model might not be able to detect it. In order to train our model to detect new and advanced adversarial prompts, we would collect new prompts containing positive examples of such prompts. Additionally, we would implement a grid search to tune a larger range and more combinations of hyperparameters. Other methods such as ensemble learning might be helpful in optimizing the model's performance as well. Implementing ensemble learning techniques, where multiple models are combined to improve predictive performance, could help mitigate the risk of overlooking certain types of adversarial prompts. Therefore, for future studies, it is important to address these limitations along with using a larger dataset to further optimize the effectiveness and reliability of our adversarial prompt detection model.