Finetuning Large Language Models for Vulnerability Detection

James Flora Shijie Zhao Yunhan Qiao Yong-hwan Lee
Oregon State University
{floraj, zhaoshij, qiaoy, leeyongh}@oregonstate.edu

Abstract

The task of Code Vulnerability Detection seeks to identify software source code that has flaws that are able to be exploited by an attacker in order to endanger the software's security. Many current methods at the enterprise level simply leverage an expert's opinion and require software to be checked by a human in code review. However, since the advent of Large Language Models (LLMs), many have sought to turn this code vulnerability problem into one of binary classification. Given a function's source code, can the LLM tell whether or not the code is vulnerable? We finetune WizardCoder, an LLM trained specifically on the "language" of code in order to perform this binary classification. We use the QLoRA method in order to significantly cut down on the runtime and hardware requirements of finetuning the 13b parameter model. We obtain a best ROC AUC of 0.72 and a best F1 Score of 0.66. Further, we perform experimentation on two hyperparameters, namely sequence length and the inclusion of large functions.
This report is primarily a reproduction of the results from the paper of the same title by Shestov et al. [18] The authors' contributions are as follows: implementation of QLoRA and further experimentation on the impact of hyperparameters.

1. Introduction

Since the advent of Large Language Models (LLMs), many machine learning tasks previously thought to be too difficult so solve through conventional techniques have seen leaps and bounds of progress over the last few years. Treating the LLM as the basis for providing knowledge about a language, many researchers have taken to "fine-tuning" models, or making small changes to the LLM that allow it to be adopted for a task that is slightly different from the one it was intended for. In this paper, we use such an idea to adopt the LLM, WizardCoder, to the task of code vulnerability detection via LoRA and QLoRA.

1.1. Code Vulnerability Detection

In the context of machine learning, the research area of code vulnerability detection seeks to train a model to perform binary classification on snippets of code in order to tell if they are "vulnerable" or "not vulnerable". In practice, this area of research can be accurately described as a type of anomaly detection, in which the distinguishing feature is a heavily imbalanced dataset in favor of "negative" instances. One of the most well-known examples of a type of anomaly detection is the classification problem of diagnosing breast cancer in women. Indeed, most women do not have breast cancer, however it is incredibly important that positive instances are detected accurately. Analogously, most written code is not vulnerable, but it is paramount that positive instances are labeled correctly.
Early work in the field utilized alorithmic approaches wherein vector representations of code were determined and compared to known vulnerable ground truth data [12] Although such approaches were successful at the time, they relied on a large pool of known ground truths and such vector representations were not particularly good at capturing the meaning of the code itself. Conversely, the current approach almost entirely revolves around large language models due to their promising results in several different domains and types of tasks. Though most LLMs are focused on the understanding of human language, several LLMs focused on the language of code have been developed, such as WizardCoder [15] and CodeBert [5]. Though these models are originally intended for the generation of language, they are a good target for our code vulnerability detection problem because of the implicit assumption that if one understands the language of code, one can be taught how to detect the vulnerabilities of code given enough data.

1.2. Large Language Models

The popularity of Large Language Models has exploded in recent years since the advent of the transformer architecture and the resulting improved scalability of such models. As noted above, while most LLMs focus on human language, such as GPT-3, BERT [3], and LLaMA [19], others are created for more specific tasks and types of lan-

guage. For example, ContraBERT [14] focuses on contrastive learning, and JukeBox [4] focuses on the generation of music modeled as a language. These tasks are all generative in nature, meaning that they make use of both an encoder and a decoder. The LLM encodes a representation of its input before predicting the next word in the sequence with a decoder. However, it is possible to utilize LLMs for tasks outside of text generation. In the case of something like binary classification, we would generally do away with the decoder part of the architecture and simply take the output of the last ”word” of the sentence, and then perform some classification function (like softmax) into a loss function.
Despite LLM's promising results, they are as the name says, very large. On average, the standard LLM hovers in the billions of parameters, much too large to fit into any modern consumer-grade GPU. This poses a significant challenge in training LLMs and adapting them for tasks that they were not originally designed for. Hence, it is paramount that we adapt our approach in order to move forward with code vulnerability detection.

1.3. Fine-Tuning

The fine-tuning of neural networks is a standard approach in adapting deep networks for tasks that are slightly different from the ones they were intended to handle, this can be viewed as a type of transfer learning. In the vanilla case, additional training data is supplied to the model with a greatly reduced learning rate in order to ”fine-tune” the weights of the model to be better adapted to the new task represented by the new training data.
However, as mentioned, LLMs are far too large to fine-tune all of the weights of the model using this new training data. In order to overcome this challenge, LoRA and QLoRA are introduced as a method of reducing the parameters of the model that we actually need to tune during training time, while only sacrificing a negligible amount of performance. The core of these adaptations relies on the introduction of low rank matrices to the models which are able to ”capture the essence” of the model's parameters, and then only training those matrices. In doing so, we have reduced the number of weights changed in every iteration of training by an exponential amount. Further, QLoRA further reduces the memory required by quantizing the already existing training weights, that is, truncating them to be represented in less bits, thus greatly reducing the overall memory footprint of the model.

2. Related Works

The basis for this experiment is the aforementioned fine-tuning of WizardCoder using LoRA with focal loss and batch-packing in order to perform binary classification on a given input (as well as next token prediction) [18]. While it appears as though the idea of finetuning LLMs is quite a common approach, the authors cite the last major paper in the field of vulnerability detection as *LineVul* [6] which simply adds extra functionality to BERT by putting a LineVul model after it in order to perform binary classification. While this approach appears to work reasonably well, it can be argued that BERT wasn't really adapted to the task at hand, but just used as a impromptu encoding step for the training data. The *LineVul* paper itself was a response to *IVDetect* [11] which uses an interesting combination of GRUs and graph representations of source code in orer to detect vulnerabilities. Whilst this approach is novel, it is summarily beaten by the expanding power of LLMs. Keeping with this theme, Shestov cite one of their main motivations as using a larger LLM along with more sophisticated fine-tuning methods to expand upon what *LineVul* was able to do.
On the topic of larger LLMs, the paper decides to use WizardCoder [15], an LLM specifically created to understand the ”language” of code. Whilst WizardCoder is certainly not the first LLM to try to tackle this domain, at the time the paper was released, it was only beat out by GPT-3.5 and GPT 4. As of today, it currently beats GPT-3.5 in human evaluations. The synthesis of WizardCoder focuses on utilizing the evo-instruct method (or fine tuned instructions) in order to give more diverse and complex tasks to the already built StarCoder. StarCoder [9] itself is an open-source LLM that purports to be trained on only permissively licensed github repositories, which would make it one of the only LLMs to use ethically sourced training data (or the only that we are aware of). Regardless, the WizardCoder used in this paper is around 13B parameters and is currently dwarfed by Code Llama [17] which is around 70B parameters. Despite this, it can be understood that the authors wanted to engage in parameter efficient fine-tuning and opted for the slightly smaller WizardCoder instead.
In the realm of parameter efficient fine-tuning, there are several different ways to go about it with varying levels of success and scale. Whilst one possible option is Prefix-Tuning [10] which is based on the idea of adding trainable ”prefix” vectors to a model while freezing the rest of the parameters in a model in order to train it for a downstream task. Whilst this method is valid, the authors opt for LoRA [7] which creates trainable low-rank matrices that are approximate representations of the layers that they represent. This idea relies on the discovery that over-parameterized models rely on an inherently low-rank dimension that can be represented with less data than is actually used [8]. As it turns out, most LLMs are likely over parameterized and are good candidates for the LoRA method. Lastly, in the present paper we also consider the QLoRA method [2]. As an extension of the LoRA method, QLoRA further quantizes the

weights that are frozen in LoRA, meaning the entirety of the weights can be represented in less memory than if they were represented in full 32bit floating point precision. Empirically, this results in very little degradation of performance, and allows the amount of VRAM used in the fine-tuning of these models to be reduced by entire orders of magnitude.

3. Technical Approach

3.1. LoRA

At a high level, LoRA seeks to make the fine-tuning of large neural networks more efficient by training less parameters overall. This is able to manifest in low rank adapters, which rely on the fact that over-parameterized models can be represented in a lower dimension. This result was empirically proven [8] to be true in most modern neural networks. Functionally, if all of our weight vectors are parameterized by some $\theta \in \mathbb{R}^D$, it is not necessarily true that all D of the parameters ”need” to be correct or are even used in the optimal solution of our neural network. Li formalizes this as

$$D = d_{int} + s$$

where d_{int} refers to the intrinsic dimensionality of our solution (the parts that are necessary to get right) and s as the dimensionality of our solution set, or ways that we could move and still be at the ”correct” solution. Indeed, in the landscape of hundreds of dimensions, it does seem incredibly likely that there are multiple ways one could move in the multi-dimensional space such that the loss does not actually change and for every optimal solution, there is likely hundreds by moving in any given direction that does not correspond to an actual change in the output of the loss function. The results of the paper continue on to show that the solution set s is generally quite large for most neural networks in practice.

LoRA takes this hypothesis and furthers it by making the assertion that it is likely that the changes in weights within a model during fine-tuning also rely on this same low intrinsic rank, and thus we may be able to train just the low intrinsic rank without having to touch most of the model's parameters. Formalized, it follows that any sort of fine-tuning of a neural network can be represented as $\phi_0 + \Delta\phi$ where ϕ_0 are the pre-trained weights and $\Delta\phi$ are our weight updates. In this example, it is clear that $\Delta\phi$ must be the same dimensionality as ϕ_0 in order for this to work. However, to reduce the dimensionality of our parameter updates, and thus the computing resources required, we want to ensure that $\Delta\phi = \Delta\phi(\Theta)$ where $|\Theta| < |\phi_0|$ or that we want to encode our parameter updates into a much smaller set of parameters Θ. This is achieved with the following:

Given some pre-trained weight matrix in a neural network, $W_0 \in \mathbb{R}^{nxm}$, and some update to the weight matrix ΔW, we decompose ΔW into low rank representations $A \in \mathbb{R}^{nxr}$ and $B \in \mathbb{R}^{rxm}$ where $r < min(n, m)$. These low rank representations are the parameters that we will actually train when we fine tune. Given that W_0 is frozen during our training process, we only need to train A and B which are known as the ”adapters” in order to fine-tune our model, before adding our updates back into our frozen parameters by simply multiplying A and B to give us back ΔW. Hence, the forward pass for variable x [7] is:

$$W_0x + \Delta Wx = W_0x + BAx$$

again, this relies on the assumption that most LLM's weight matrices can be well-approximated by low-rank matrices. In essence, we are able to fit weight updates into a much smaller space than the pre-trained model does by exploiting this theory.

As this pertains to transformer models, we only adapt this approach for the attention heads, and not for the subsequent multi-layer perceptron networks.

3.2. QLoRA

Following from LoRA, QLoRA further cuts down on the memory cost of fine-tuning an LLM by quantizing the frozen weights, or discretizing them from full 32-bit floats down to the 4-bit NormalFloat quantization introduced in the paper. 4-bit NormalFloat seeks to solve the problems of common discretization techniques which are generally not robust to outliers or are unable to utilize the full range of bits due to unevenly distributed data. Following this, it is known that most pretrained neural netowrk weights have a normal distribution centered around 0 with some standard deviation σ. We want to fit the range of our distribution into $[-1, 1]$, but in order to do this we need to estimate the quantiles of the data such that they fit into the range. Formally, [2]

$$q_i = \frac{1}{2}(Q_x(\frac{i}{2^k+1}) + Q_x(\frac{i+1}{2^k+1})) \qquad (1)$$

where Q_x is the standard quantile function for a normal distribution and k is the number of bits we want to quantize to. From there, we can simply quantize our weight tensor by normalzing it such that the max is 1 (absolute maximum rescaling), and then quantizing it with the above function.

Further, notice that the above equation is offset by 1 (given i and $i+1$). This is done on purpose to ensure that we have a discrete datapoint that maps to 0. For $k$bit quantization, these quantiles have discrete values that can be computed and listed ahead of time for any such k. Thus, at runtime, only the actual quantization of each weight matrix needs to be done.

3.3. Focal Loss

Given the heavily imbalanced nature of our code vulnerability detection data, it would make sense for our loss function to put more emphasis on the positive (vulnerable) data than we would the negative data. Thus, focal loss [13] takes inspiration from cross-entropy loss but adds a weighting and a focusing parameter that is able to emphasize examples of a certain class. Defined as

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma log(p_t)$$

$$p_t = \begin{cases} p & \textbf{if} \quad y = 1 \\ 1 - p & \textbf{if} \quad o.w. \end{cases}$$

α_t is a hyperparameter weighting factor for each class (in our case, we only have two since this is binary classification), and γ is known as the *focusing* parameter. γ allows us to "focus" less on instances with a certain label, in our case the code that is not vulnerable.
Further, by default, LLMs predict the next token in a sequence, that is, for tokens $\{x_1, x_2, ..., x_n\}$, our model wants to predict x_{n+1}. Or, asks what the probability of any x_{n+1} given the previous tokens. By comparing it with the actual next token (the ground truth) via one-hot encoding, we can then optimize the cross-entropy loss of the entire distribution, that is [18]:

$$L = -\sum_{n=1}^{N-1} y_{n+1} log P(x_{n+1}|x_1, x_2, ...x_n)$$

the authors further derive this into actual classification loss instead of generative next token loss by using the ground truth label of whether or not the code is vulnerable (not the next token):

$$L_{class} = -log P(y|x_1, x_2, ...x_n, \theta)$$

where θ are all the parameters of the model and y is the ground truth. We are asking that given the model parameterized by θ, what is the probability that we assign the right class label? We adopt this generalized cross entropy loss for classification loss into focal loss for some experiments.

3.4. Batch Packing

Due to the wildly variable amount of tokens in any given function that we use as training examples, it is incredibly computationally inefficient to just pad our inputs similarly to how one might naively do for text data. Whilst most of the data falls within the range of 10-100 tokens, there are also plenty of inputs for which there are more than 2,000 tokens. Were we to pad every function out such that it matches the max length of any given input, the majority of our inputs would just be padding. To rectify this, the authors introduce batch packing [18]. At a high level, we define a hyperparameter called sequence length and choose a value for it beforehand (we will see in experimental details how this hyperparameter affects performance). Given this number, we pack as many functions (inputs) as we can until we reach the limit, and then pad out the rest. For example, a sequence length of 55 would allow us to pack 5 functions with 10 tokens each, and then we would have to pad the rest of the 5 spaces left.
Though this is the spirit of the idea, in practice we do something slightly different. We load a batch with as many functions as we can fit into the sequence length, but ensure that the end of each function has an "<EOS>" token (as to separate them). Our model then populates an array of labels where each label corresponds to a singular function in the batch. We then calculate our focal loss with respect to each function in the batch, and use the sum as the loss for that batch (as the average empirically performs worse according to the authors).

4. Experiments

In this section, we discuss the specific setup of our experiments. There is significant overlap between the original paper and our own experimental setup, and we have made our modified code available [1].
We have modified the original code to be able to handle QLoRA including the quantization of the provided model if it is not already quantized. In total we finetune WizardCoder across a few different GPUs in various experiments including a consumer-grade NVIDIA RTX 4080, an NVIDIA A100 Tensor Core (on OSU's HPC Cluster), and an NVIDIA T4 Tensor Core (on Google Colab), for which the results will be presented. In addition, we test the way that different values of the sequence length hyperparameter affects training time and performance on the test data.

4.1. Datasets

As to compare to the original paper's approach with LoRA, we finetune with the same datasets in order to control as many variables as we can. The paper uses several open-source datasets including CVEfixes [1], a manually curated dataset from the NVD (National Vulnerability Database) [16], and VCMatch [20].
These datasets are further filtered into two distinct datasets by the authors that we train on. The dataset dubbed "X_1 with P_3" refers to the raw, heavily imbalanced dataset the same way such code would appear in the real world, with a class imbalance ratio of about 1:34 (1 vulnerable function for every 34 nonvulnerable functions). This dataset has 22,945 samples in it.
The dataset dubbed "X_1 without P_3 refers to a much smaller balanced dataset that is closer to the ideal class balance but with not nearly enough examples for a LLM such as WizardCoder to learn effecitvely. This dataset has 1,334 samples in it.

[1]https://github.com/NLP-Fine-Tunning

	Dataset	Sequence Length	Large Function	ROC AUC	F1 Score	GPU	Training Time (hr)
QLoRA	X_1 without P_3	512	ignore	0.53	0.65	Tesla T4	8.2
	X_1 without P_3	512	include	0.56	0.66	NVIDIA A100 x2	3.4
	X_1 without P_3	256	ignore	0.51	0.63	Tesla T4	2.9
	X_1 with P_3	512	ignore	0.68	0.14	GTX 4080	22.1
	X_1 with P_3	512	include	0.72	0.17	NVIDIA A100 x2	20.4
	X_1 with P_3	256	ignore	0.70	0.14	NVIDIA A100 x2	18.3
LoRA	X_1 without P_3	2048	include	0.69	0.71	NVIDIA V100 x8	?
	X_1 with P_3	2048	include	0.86	0.27	NVIDIA V100 x8	?

Table 1. All experiments run with QLoRA with varying sequence length and inclusion of large functions. Recall that we do not run the LoRA experiment and instead take the author's numbers at their word in the original paper. Observe the slight degradation in performance when switching from LoRA to QLoRa as well as the positive correlation with improved results when including large functions and a larger sequence length.

4.2. Pre-trained Model

As the authors did not specify which version of WizardCoder they finetuned on (and it was not provided anywhere in the code), we opt for the closest model we can find, which is WizardCoder-13B. This lines up with the reported number of parameters and is easily available on the HuggingFace library which the authors claim to have used for experiments, thus this is what we use for all experimentation.

4.3. Evaluation Metrics

As mentioned in the previous section, we use the same focal loss that is used in the original paper, as well as the evaluation metrics of ROC AUC and F1 Score. These are generally common metrics for evaluation, so we skip most of the details regarding these. Though it should be noted that the F1 score we report is only for the positive class of data. This is justified given the heavily imbalanced dataset and that our primary objective is to identify vulnerable code, not safe code. Further, whilst the original authors were able to optimize hyperparameters, we were generally unable to run enough experiments to be able to optimize for these same hyperparameters in regards to QLoRA.

4.4. Setup

Explicitly, we finetune WizardCoder for 15 epochs on the above datasets using QLoRA. We do this for the X_1 with P_3 dataset as well as X_1 without P_3. Further we test the efficacy of the "sequence length" and "include large function" hyperparameters as well as varying GPUs. As we are unable to deploy LoRA given the limitation of our computing resources, we have to make due by taking the authors at their word for their results.

5. Results

After finetuning QLoRA for 15 epochs, we report the results of all experiments in Table 1. In general, it is clear there is a significant amount of performance loss from LoRA to QLoRA with F1 scores dropping by some 10% on average and ROC AUC also seeing a hit to performance. Recall that all F1 score metrics are only for the positively labeled instances, thus an F1 score of 0.17 and 0.27 are not particularly alarming as X_1 with P_3 is our imbalanced dataset, leading to a heavy bias in the model towards negatively labeled data (predicting the majority class). Summarily, the X_1 without P_3 dataset yields much better results in general on the positive class. This proves the result that in this study, less data is better if it is balanced. No amount of loss function weighting and hyperparameter tuning can make up for imbalanced data in either LoRA or QLoRA.
We also provide the detailed training runs of said experiments showing validation loss over time, validation F1 score over time, and validation ROC AUC over time in Figure 1 and Figure 2.

As we can see, the X_1 without P_3 experiments are much more volatile with respect to loss, randomly jumping up much higher than they had previously been despite more training. Intuitively, this makes sense given that we have a lot less data to train and validate on in the first place: it is much more likely that we randomly choose our training data and validation data in a way that is unlucky for that epoch and spike the cumulative loss. Despite that, the model does eventually find its way into a consistent amount of loss as well as scoring metrics. Another point to note is that while loss generally is on a steady decline, evaluation metrics are much more volatile and do not necessarily follow with the trend of loss. This can possibly suggest that the loss metric used is not sufficient to train the model to be able to detect vulnerable code. No matter how far we can push the loss down, we barely move the needle on F1 Score and ROC AUC.
Sequence Length: Further, though the authors claim that sequence length is largely unimportant as a variable, our results tell a different story. Though it is possible that the mix

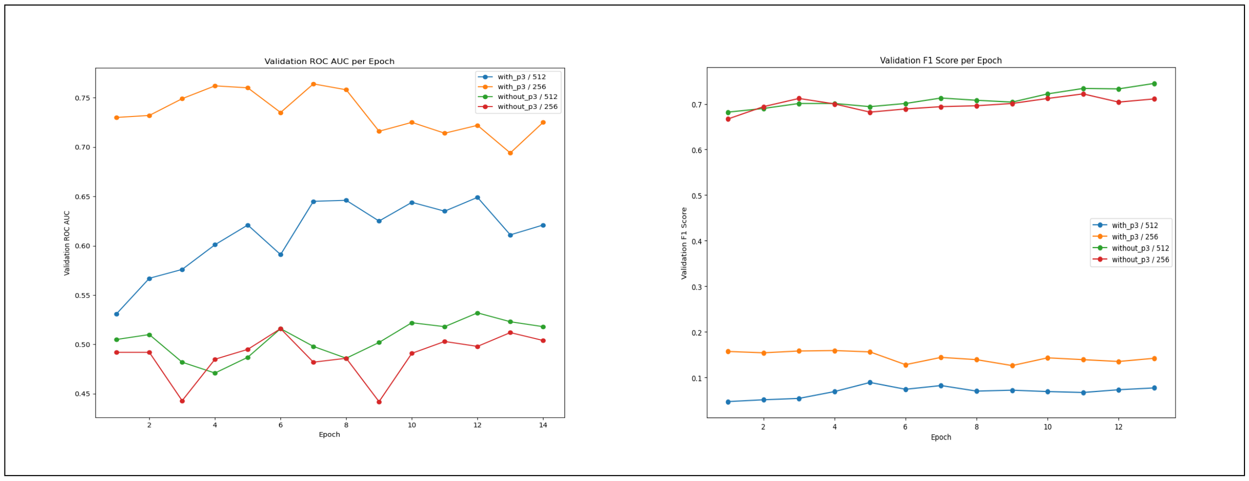

Figure 1. F1 score and ROC AUC per epoch for sequence length experiments

of QLoRA and sequence length had an adverse effect on results in a way that we could not account for, we must take the authors at their word given that we do not have the computing resources to run LoRA. Perhaps inexplicably, the single experiment corresponding to the with p_3 dataset and 256 sequence length started with a much higher ROC AUC, F1 score, and loss compared to the other with p_3 dataset experiment. This however does not translate to a much better result on the testing data as it only yields a 0.02 increase in ROC AUC and no increase in F1 Score. The without p_3 experiment does see some degradation with sequence length however.

Include Large Function: In addition to this, including large functions also has a positive correlation with ROC AUC and F1 Score. This hyperparameter is not mentioned in the original paper, but is a parsed argument in the author's code that we decided to test on. Perhaps unsurprisingly, including larger functions in our training data only helps the model perform better on the test data (likely on analogous large functions in the test data). Though we do not have particularly controlled data on this front, it is likely that including large functions also incurs a larger cost in training time (2 NVIDIA A100's are almost certainly doing more work in the same timeframe as a GTX 4080).

Computing Resources: On the topic of computing resources, while the authors do not report training time in the orginal paper, they do report that they use 8 NVIDIA V100 GPUs. Across our experiments, it was clear that even with the gained memory efficiency of QLoRA, we still had a bottleneck around time efficiency, of which QLoRA is not really an improvement over LoRA. Indeed, quantizing the model led to reducing our memory footprint by a factor of 4 (around 26gb for the full model, and only around 6gb when it was quantized), but we still likely incur the same time cost as the authors of the original paper (depending on how they utilized the parallelization of their computing resources).

6. Conclusion

In this paper, we recreate the findings of Shestov et al. in which we finetune the LLM, WizardCoder, for code vulnerability detection. Whilst the original authors use LoRA to do so, we employ QLoRA to cut down on overall model size and are able to train such a model on a consumer-grade GPU. Despite this, we see significant degradation in performance metrics though it is clear that the model is still doing some sort of ”learning”. Further, we perform experimentation on the hyperparameters ”sequence length” and ”include large function”. We are able to conclude that including large functions is a strict positive for the model's learning capabilities, but the evidence on sequence length is inconclusive due to a baffling experiment with much higher results than the rest.

6.1. Future Work

In the future, it would be incredibly helpful to have the GPU power to be able to run the author's original LoRA experiments to see if LoRA consistently performs as well as the numbers they included in their paper, or if the numbers provided were a ”best case scenario” across several attempts at training the model. Further, procuring more data is almost always a positive, and seems as though it would be especially in this scenario given that our validation loss randomly jumps up when we seem to get ”unlucky” with how the data is split since there is so little (balanced) data in the first place. Lastly, it would be interesting to see how the model performs with more epochs of training. Near the end of 15 epochs it seems as though most of the experiments work themselves into a stable amount of loss, but

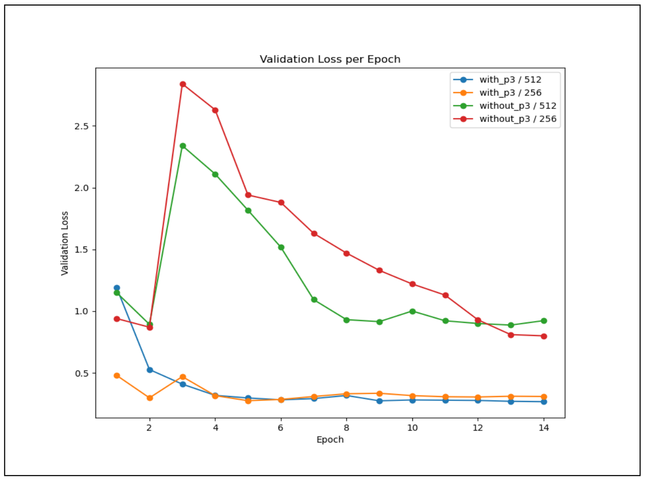

Figure 2. Validation Loss per epoch for sequence length experiments

more training would be needed to make that conclusion with more certainty (again, with reference to our singular outlier experiemnt).

References

[1] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '21. ACM, Aug. 2021. 4

[2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. 2, 3

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. 1

[4] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music, 2020. 2

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. 1

[6] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022. 2

[7] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. 2, 3

[8] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes, 2018. 2, 3

[9] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. 2

[10] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021. 2

[11] Yi Li, Shaohua Wang, and Tien N. Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21. ACM, Aug. 2021. 2

[12] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 201–213, New York, NY, USA, 2016. Association for Computing Machinery. 1

[13] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. 4

[14] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Contrabert: Enhancing code pre-trained models

via contrastive learning, 2023. 2

[15] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023. 1, 2

[16] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software, 2019. 4

[17] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. 2

[18] Alexey Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Anton Cheshkov, and Pavel Zadorozhny. Finetuning large language models for vulnerability detection, 2024. 1, 2, 4

[19] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. 1

[20] Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. Vcmatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 589–600, 2022. 4