The Implementation of Model Pruning to Optimize zk-SNARKs

Abigail Thomas

January 15, 2022

Abstract

Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARK)s are used to convince a verifier that a server possesses certain information without revealing these private inputs [Gro16]. Thus, zk-SNARKs can be useful when outsourcing computations for cloud computing. The proofs returned by the server must be less computationally intensive than the given task, but the more complex the task, the more expensive the proof. We present a method that involves model pruning to decrease the complexity of the given task and thus the proof as well, to allow clients to outsource more complex programs. The proposed method harnesses the benefits of producing accurate results using a lower number of constraints, while remaining secure.

Keywords: privacy, zk-SNARK, model pruning, quantization, machine learning

1 Introduction

As the world becomes more digitized, companies around the world are creating more complex applications to help process this new influx of data. The complexity of computations necessary for this processing are increasing at an exponential rate, and even highly advanced smartphones are not powerful enough to perform such intense computation. In cases like these, less powerful clients outsource tasks to a more powerful server in the cloud, which sends them back the results of the completed task [Gro16]. This is more commonly known as cloud computing and can be seen in many of the applications we use every day. For example, Google Colab provides a way for people without access to GPU or TPU to utilize Google's resources to run complex Python code. Kaggle Notebooks is another similar service that allows people to use GPUs to run Python or R scripts.

When clients run programs through their own computers, they can guarantee that the results are the output of the program of interest. However, when outsourcing complex computations, there lies an issue in that the remote servers may not do exactly what the clients want. They may run a computationally cheaper algorithm in order to save money and computing resources. For example, if a client wanted to run a complicated neural network in order to classify a data set of images, the remote server may return results from a simpler algorithm, such as a support vector machine. In order for the server to prove to the client that they completed the task they requested, the server can include a (zero-knowledge) Succinct Non-Interactive Argument of Knowledge (zk-SNARK) as well [Gro16].

Aside from having practical uses in the realm of cloud computing, zk-SNARKS also have theoretic relevance towards the larger body of cryptography. Specifically, they can be used to convince other parties they are authenticated personnel without leaking any information about the specific inputs. Zero-Knowledge SNARKs do just this by constructing an R1CS to prove that they possess certain private inputs without directly revealing what these inputs are [Gro16].

The proof that is returned back for the client to compute must fall within the computational limits of the client, in order for the client to verify the computation. If the client does not have the computational power to handle even the proof, then the zk-SNARK loses its power. However, as the tasks become more complicated, so do the proofs, which places an "upper bound" on the complexity of tasks that can be securely outsourced.

Another method that allows low resource devices to perform complex computations without outsourcing tasks, is through model pruning. Model pruning aims to reduce the size of a neural network (a type of complex machine learning algorithm), by removing certain unnecessary weights from the network [Ban20]. There are multiple ways to determine what constitutes an unnecessary weight. For example, the most common type of pruning, magnitude pruning, removes weights of magnitudes close to zero [Ban20]. Another method of pruning, movement pruning, is more useful in the case of transfer learning [SWR20]. Movement pruning removes weights that do not change much after fine tuning the model. Both methods can be incredibly powerful in reducing the amount of computations done, while still maintaining a very high level of accuracy [HPTD15].

In this paper, we present a method that involves pruning of complex models specifically a simple convolutional neural network (ShallowNet) in order to decrease the complexity of these proofs. We also evaluate the performance of this method by examining the accuracy and number of constraints before and after pruning. This model provides a trade off between accuracy and the number of constraints (how computationally intensive the proof is).

1.1 Related Works

Before deep diving into our experiment, it is important to understand the necessary background information from other papers on these topics. This section will cover the details of zk-SNARKs as well as magnitude and movement based pruning.

The concept of zk-SNARKs arises from the problem of having a prover try to convince a verifier that a statement is true without revealing any information about this private input. This is called a non-interactive zero-knowledge proof (NIZK). These proofs are defined by three main properties: completeness, soundness, and zero-knowledge. Completeness states that the prover can convince the verifier through a NIZK given a statement and a witness. Soundness states that in the case the prover is a malicious party, the verifier cannot be convinced of a false statement. Zero-Knowledge, as mentioned earlier, states that the prover will not reveal its witness [Gro16].

Groth's NIZK argument involves converting a computational problem to a Rank-1 Constraint System (R1CS) which can then be turned into a usable Quadratic Arithmetic Program (QAP) [Gro16]. An R1CS is a group of three vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$, which has a solution $\boldsymbol{r}$ such that $\boldsymbol{a} \cdot \boldsymbol{r} * \boldsymbol{b} \cdot \boldsymbol{r} - \boldsymbol{c} \cdot \boldsymbol{r} = 0$ [But16]. The size of the vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{r}$ is equal to the the number of logic gates required to represent the computation (there is one constraint per logic gate) [But16]. By following a systematic method (shown in later in this section), the R1CS of any arithmetic circuit can be constructed, and from that a QAP can also be constructed. The process of constructing an R1CS is called quantization. In the case of Boolean circuits, a quadratic span programs (QSP) is constructed instead [Gro16].

To gain an understanding of the process of taking a program and converting it to a QAP and also understand how constraints work, consider the following example [But16] :

$$\text{Find } x, \text{ such that: } x^4 + 2x + 7 == 50$$

This can be broken into the following arithmetic circuits:
$sym_1 = x \times x$
$y = sym_1 \times sym_1$
$z = 2 \times x$
$sym_2 = y + z$
$\sim out = sym_2 + 7$

Let's say the order of variables in $\boldsymbol{r}$ is expressed as $[\sim one, x, sym_1, y, z, sym_2, \sim$

out]. Then the corresponding R1CS is

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\boldsymbol{r} = \begin{bmatrix} 1 & 2.48336 & 6.16707689 & 38.0328374 & 4.96672 & 50 \end{bmatrix}$$

Using a QAP generator program gives [But16]:

$$A_{poly} = \begin{bmatrix} 27.0 & -53.6 & 34.7 & -8.9 & 0.8 \\ 5.0 & -6.4 & 3.0 & -0.6 & 0.04 \\ -10.0 & 17.8 & -9.8 & 2.2 & -0.2 \\ -5.0 & 10.2 & -6.8 & 1.8 & -0.2 \\ -5.0 & 10.2 & -6.8 & 1.8 & -0.2 \\ 1.0 & -2.1 & 1.5 & -0.4 & 0.04 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$B_{poly} = \begin{bmatrix} -4.0 & 8.1 & -5.4 & 1.4 & -0.1 \\ 15.0 & -25.9 & 15.2 & -3.6 & 0.3 \\ -10.0 & 17.8 & -9.8 & 2.2 & -0.2 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$C_{poly} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 5.0 & -6.4 & 3.0 & -0.6 & 0.04 \\ -10.0 & 17.8 & -9.8 & 2.2 & -0.2 \\ 10.0 & -19.5 & 12.25 & -3.0 & 0.25 \\ -5.0 & 10.2 & -6.8 & 1.8 & -0.2 \\ 1.0 & -2.1 & 1.5 & -0.4 & 0.04 \end{bmatrix}$$

If these arguments of knowledge are cheaper than the task being performed, then these proofs gain use as a way to outsource complicated programs to cloud servers and then verify the proof of knowledge. Thus, making these proofs succinct non-interactive argument of knowledge or (SNARKs) is extremely beneficial for its practical uses. If these SNARKs are zero-knowledge, this means that they do not leak any private inputs, which adds to the security of these computations. Thus, we have defined what zk-SNARKs are [Gro16].

Now, let's examine the concept of model pruning. Neural network pruning is a method of removing unnecessary computations from a model in order to compress it [Ban20]. Doing so can decrease complexity and run time. This paper will focus on two main methods network pruning: magnitude pruning and movement pruning. Magnitude pruning is pruning weights on the basis of the magnitude of the weight. If the weight is close to 0 in magnitude, this shows that the node it is connected to is of little importance to the overall model. Thus, by removing these minuscule contributions, we don't affect the accuracy, while decreasing the resource cost of running the program. One implementation of magnitude pruning is to train a neural network on a data set, and then remove weights below a certain threshold. Then, the model can be retrained to further fine tune the remaining weights. Experiments on large data sets using complex neural network models, have shown great success with this method of magnitude pruning. For example, Han et. al were able to reduce the number of computations of their model by 13× without sacrificing accuracy at all [HPTD15].

Movement pruning, on the other hand, prunes weights based on how little they changed between the initial train and the fine tuning process. Movement pruning is more suited to transfer learning, while magnitude pruning is only effective for supervised learning. In transfer learning we have an original model with weights, and after fine tuning that model on a given data set, the weights change accordingly. The change in weight between this original model and fine-tuned model, is what is used to determine the weights that should be pruned [SWR20].

So far in the paper, when referring to model pruning, we have been referring to removing weights from neural networks, but it is also possible to remove neurons as well. Removing nodes preserves the overall shape of the network, but can affect the accuracy much more because removing neurons subtracts a lot more synapses than weight pruning does [Ban20].

2 Methods

We chose to test the pruning optimization on the famous machine learning problem: training the publicly available MNIST dataset. The MNIST dataset

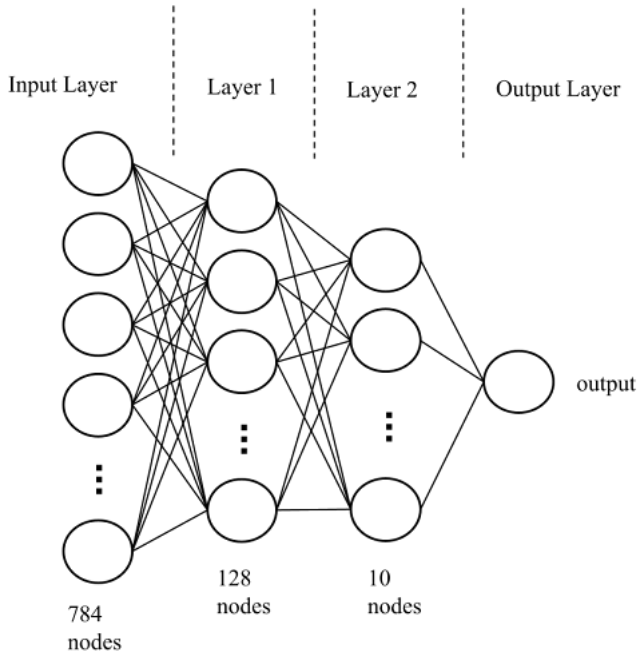

Figure 1: ShallowNet Neural Network Architecture

consists of 28x28 pixel, grayscale images of handwritten digits from 0 to 9, and we use a fully connected neural network with two hidden layers and the ReLu activation function to categorize each image. To input the images into the neural network, each image is flattened from the dimensions 28x28 to a vector of length 784. Figure 1 displays a visual representation of the neural network architecture used.

To implement the zk-SNARKs in our code, we used the publicly available GitHub repository for ZEN: Efficient Zero-Knowledge Proof for Neural Networks compiler. ZEN is an optimizing compiler for zk-SNARKs. ZEN goes about doing this by reducing the number of constraints represented in the R1CS of the program. Since the zk-SNARK derives from a QAP which comes from the R1CS, this will result in a less computationally intensive proof. This compiler reduces the number of constraints using the following quantization schemes: sign-bit grouping and remainder based verification [FQZ$^+$]. An additional characteristic of the ZEN compiler is that it does not support model training when creating the zk-SNARK. Instead, it has two sub tasks: $ZEN_{inference}$ and $ZEN_{accuracy}$. $ZEN_{accuracy}$ calculates the accuracy of the model the zk-SNARK is being made for and $ZEN_{inference}$ finds the number of constraints necessary. So together we can construct a complete picture of the model's complexity and accuracy but these tasks cannot be run simultaneously. In our research, we mostly experi-

ment with the $ZEN_{inference}$ subtask to find the number of contraints used in the R1CS. Another important note is that zk-SNARKs only support computations with positive integers. After the networks were trained many weights had decimal or negative values, so once the network was training we had to input the weights through the ZEN quantization process to make all the weights integer values. This was done using ZEN's novel quantization optimizations as aforementioned (sign-bit grouping and remainder based verification).

To incorporate pruning, we implemented the simplest form of network pruning: global magnitude pruning. We set a percentage threshold for what should be pruned, for example, the smallest 50% of weights would be set to 0 and observed the accuracy of the model. We experimented by pruning 0% of the weights, 50%, and 100% of the weights and found the accuracy and number of constraints of the zk-SNARK. This was tested using PyTorch and the ShallowNet architecture (displated in Figure 1) on the MNIST data set. The ZEN compiler also has a lower limit for the magnitude of numbers, so values below this threshold will automatically be set to 0 because they cannot be adequately represented. Thus, the ZEN compiler will automatically prune some weights while it runs, though this is a trivial level of pruning.

3 Results

The results of running the ShallowNet on the MNIST data set through the ZEN compiler are show in Table 1. In this table, we can see there are a large number of constraints needed for the model without any pruning. We aim to reduce this number through the use of pruning, without significantly impacting the accuracy.

Number of constraints for L1:	343040
Number of constraints for Relu:	456
Number of constraints for L2:	20240
Total number of Full Circuit inference constraints:	363736

Table 1: Table of Constraints for ShallowNet on MNIST

The results of the global pruning trials for the ShallowNet run on the MNIST data set can be seen in Table 2. In this table, we can see as more of the model gets pruned, the accuracy of the model decreases. At first, it does not seem to decrease much, indicating that the weights pruned were trivially important, but as the important weights get set to 0, we see the accuracy suffers greatly.

Amount Pruned	Accuracy	# of Constraints
0%	**0.9516**	**363736**
50%	0.9505	363719
100%	**0.0980**	**363644**

Table 2: Table of Constraints and Accuracies for Different Amounts of Pruning the ShallowNet

4 Discussion

This section contains an interpretation of the results shown in Section 3.

The results from this experiment were that model with 0% of the weights pruned had an accuracy of 0.9516 which was the highest and the most amount of pruning was 0.0980. This makes sense 100% pruning means the model is no better than guessing, and it will guess the correct digit 1/10 of the time which is roughly the 10% we see in the accuracy. The accuracy of the 50% pruned model was actually quite close to that with no pruning and this is probably because a neural network is a bit of an advanced architecture for training a simple dataset like the MNIST dataset. With other datasets and other models we might see a greater distinction between these two values.

Clearly as seen by the trial in Table 1, a large number of constraints are needed for complex models like neural networks without pruning. In the case of the ShallowNet on the MNIST dataset, 363,736 constraints were needed to create a proof for this program. Pruning reduces the number of weights in the model, which lowers the number of constraints because less intensive operations are being performed (multiplication by 0 is trivial cost wise compared to multiplication of two variables). However, pruning also decreases the accuracy as show in Table 2, for example the validation accuracy drops from 95.05% to 9.80% after pruning the last 50% of the weights. Thus, we are presented with a trade off between accuracy and number of constraints.

5 Conclusion

This project aimed to create a model that combines zk-SNARKS with the existing idea of model pruning. The results presented a tradeoff between accuracy and number of constraints, with the higher accuracy model having the largest number of constraints and the lowest accuracy model have the lowest number of contraints. Thus, we can conclude that our method of optimization does reduce the number of constraints in the proof and in turn, reduces the complexity of the proof. This result has particularly useful implications in the realm of cloud computing, as mentioned earlier, smartphones, watches, and laptops will be able

to outsource more advanced computations because the complexity of the proof is reduced by our optimization. It can also be used to decrease the complexity of authentication proofs when proving to a verifier that you are authenticated personnel without revealing any private inputs.

5.1 Limitations

One limitation of our project is that the findings only apply to machine learning tasks that involve neural networks. Network pruning involves removing some of the existing, less important weights in the neural network, and our optimization doesn't apply to other problems. However, many of the most complex machine learning problems utilize neural networks, so our contribution still has major implications.

5.2 Future Work

Future areas of research include using different models on different datasets. The ZEN compiler includes examples of LeNets of different sizes (small, medium) implemented on the CIFAR-10 and ORL datasets, but we could add modify the code so it would work different models as well such as transfer learning, instead of just being constrained to neural networks [FQZ+].

To account for different algorithms like transfer learning, we could also implement different, more effective methods of pruning. For example, movement pruning is known to be more effective for transfer learning [SWR20].

From Table 2, we also see that the majority of constraints come from Layer 1 of the network, so perhaps in future work, we could also examine targeting the pruning in this specific layer as opposed to the other layers.

In the remaining months of this project, we will be examining these questions among others to try and improve upon our current results.

6 Acknowledgements

I would like to thank my mentor Yu Xia for his continued support and guidance for this project, as well as the MIT PRIMES program for giving me this research opportunity.

References

[Ban20] Rohit Bandaru. Pruning neural networks, Sep 2020.

[But16] Vitalik Buterin. Quadratic arithmetic programs: from zero to hero, Dec 2016.

[FQZ+] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. *Advances in Cryptology – EUROCRYPT 2016 Lecture Notes in Computer Science*, page 305–326, Apr 2016.

[HPTD15] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[SWR20] Victor Sanh, Thomas Wolf, and Alexander M Rush. Movement pruning: Adaptive sparsity by fine-tuning. *arXiv preprint arXiv:2005.07683*, 2020.