Hardware Acceleration of Lattice Networks

Matthew Feldman
Department of Electrical Engineering
Stanford University
mattfel@stanford.edu

Tushar Swamy
Department of Electrical Engineering
Stanford University
tswamy@stanford.edu

Abstract

The ability to quickly distinguish malicious packets from normal packets in a network is becoming an important line of research. The goal is to drop packets whose metadata indicates that they are not coming from a benign source. We employ machine learning to make these decisions quickly on both CPUs and FPGAs while exploring two different classes of deep learning architectures to achieve this task. We use standard networks with multiple fully-connected layers, as well as lattice regression networks, which are an emerging class of networks that are designed to provide quick inference, quick training, and intuitive visibility in the intermediate layers. We show that lattice networks are able to exchange some accuracy for inferences that take less than 10 nanoseconds. In addition, they allow an order of magnitude less malicious packets into the network.

1 Introduction

The focus of our project is to use FPGAs to accelerate neural network applications. We are particularly interested in lattice regression networks (1), as they are good candidates for tasks with tight latency constraints and expose a level of debuggability and visibility that is not present in other ML architectures. Lattice regression is a new network architecture currently used by Google for applications such as ranking and ads.

The task that we will focus on in this project is network anomaly detection. We believe anomaly detection to be a good use case because data center networks need to classify each and every packet that enters the network while maintaining high data throughout and low latency. In this task, packets must be labelled as either "normal" or "malicious" so that the bad ones can be dropped from the network. In a real-life setting, misclassifying malicious packets as normal is very bad, while dropping normal packets is wrong but not the worst kind of mistake.

Hardware acceleration is a promising tool for high-performance ML and an FPGA implementation of lattice regression has not been done before. We believe that we can prove lattice networks map well to these devices, resulting in an impactful result for both the machine learning and hardware acceleration communities. There are a number of features of the algorithm that make it a much better fit for hardware rather than software. Furthermore, FPGAs are more energy-efficient than CPUs because they remove a lot of the overhead by baking instructions directly into hardware.

2 Related work

2.1 Lattice Networks

The bulk of work on lattice networks has come from the Glassbox team at Google (1; 7; 8; 9). They are theory based papers so we take advantage of their description of lattice networks to build our models. However, they have little to no implementation or systems level details. As such, we believe this is one of the main contributions of our work. There are several key equations that we use from their work to build and train our models. First, actual feature inputs are mapped to learned values using an input calibration layer built from shifted ReLUs:

$$c(x[d]; a, b) = \sum_{k=1}^{K} \alpha[k] ReLU(x - a[k]) + b[1]$$

Typical lattice networks will have a calibration layer for input features followed by the actual lattices. We then use their formulation for multilinear interpolation across the k vertices of a lattice structure where v_k are selectors that have a value of 0 or 1 and selectively remove certain vertices. $x[d]$ are input features to the lattice:

$$\phi_k(k) = \prod_{d=0}^{D-1} x[d]^{v_k[d]}(1 - x[d])^{1-v_k[d]}$$

The results of the interpolation are applied to the lattice parameters θ:

$$f(x) = \theta^T \phi(x)$$

For training, the objective function used is similar to the traditional deep learning objective functions which apply a loss function over n points with a regularization function $R(\theta)$:

$$\theta = argmin_\theta \sum_{i=1}^{n} l(y_i, \theta^T \phi(x_i)) + R(\theta)$$

Finally, the lattices lend themselves to a different type of regularization called torsion regularization which punishes twisting (resulting from overfitting) of the lattice structure by checking differences in adjacent vertex parameters (r, s, t, u):

$$R(\theta) = \sum_{d=1}^{D} \sum_{d'=1, d' \neq d}^{D} \sum_{r,s,t,u} ((\theta_r - \theta_s) - (\theta_t - \theta_u))^2$$

2.2 Intrusion Detection with Machine Learning

There are very few papers using deep neural networks as a learned intrusion detection system. A good baseline that we found was from Mane et al. (2). However, this is a very simple network and in our project, we explore some different topologies in addition to the one shown in this paper and see how it affects accuracy. There is more literature on using traditional machine learning techniques such as support vector machines but this is outside the scope of our project.

2.3 FPGAs and Spatial

There is no prior literature on mapping lattice networks to FPGAs. However, there is more general work mapping pattern based applications to FPGAs (5; 6). We use a language called Spatial to take advantage of these techniques (4). Spatial is a high level hardware description language embedded in Scala that would allow us to easily produce synthesizable Verilog. We utilize this embedding to metaprogram Spatial kernels in Scala that could be automatically wired together to form the whole network.

3 Dataset and Features

We are using the 1999 KDD Cup Dataset for this project (10). This dataset consists of 743MB of network data from MIT Lincoln Lab over the course of 9 weeks. Despite its age, it remains the primary dataset for anomaly detection since no data set of this scale has since been made public. The malicious packets have 14 different types of attacks spread across 4 categories:

- DOS: Denial of service attacks
- Probes: Probing attacks that look for vulnerabilities
- R2L: Unauthorized access from remote machines
- U2R: Unauthorized access to root privileges

The data points themselves are not full packets but information about the packet spanning three categories. There are connection level features, server level features, and traffic features collected over a two second window. The features in the dataset are a combination of strings, integers, and floating point numbers, and the labels are strings. For FCNNs (fully connected neural networks), we preprocess the dataset by converting strings to integers based on an arbitrary mapping, and then treat all numbers as floats so that the deep neural network architectures are compatible. The lattice networks don't require this preprocessing because the input calibrators perform this mapping with learned parameters.

We have about 5,000,000 points in the dataset and we chose to use 2% for training, 2% for dev, and the rest for testing. The reason for this unusual distribution is because a claimed property of lattices is to generalize from very small amounts of data. One tricky thing about the dataset is that some of the attacks have very few data points. This is because certain kinds of attacks are rare. Some attacks, such as smurf, involve flooding a network to achieve DoS, so there are many packets with these labels. We considered data augmentation but were unable to find a meaningful way of mixing and matching properties of each data point to generate new data without making unsubstantiated assumptions about attack types. For example, a buffer overflow should have a larger payload because it contains shell code but this is not always true and it is difficult to back this up with literature so we left the data set as is.

4 Methods

In this problem, we chose to only consider the number of malicious packets misclassified as normal for our evaluation metric (false negatives). This is because there is a much bigger penalty in the real world for permitting malicious packets than for dropping normal packets as malicious (these would simply be resent by the client). As a result, our primary metric was recall because it punishes false negatives only. The misclassification rate of normal packets is also important for a functioning network (a network that drops all normal packets is very useless), but we decided to put much more emphasis on the opposite problem.

4.1 FCNN (Fully Connected Neural Networks) Construction and Tuning

Our baseline for this project was standard, fully connected, multi-layer FCNNs. Our FCNN was originally based on the simple FCNN described in (2). However, we tried several variations on the models to explore how layer depth affected accuracy. Because of the processing rate requirement, we were constrained on how complex the networks could be. We had to sacrifice complexity (and therefore inference accuracy) for inference latency. This left us with a few knobs in terms of choosing FCNN architectures: depth of the network and width of each layer. We swept these parameters coarsely to get an understanding of the design space and how different designs affect accuracy and runtime. We used the ADAM optimizer (equation not shown for lack of space) along with a categorical crossentropy loss function, learning rate of 0.001, and mini-batches of size 16.

When implementing the FCNNs on the FPGA, we had a few more hyperparameters to work with, including parallelization factors, pipeline factors, loop ordering, and tiling parameters. We found the the best performing hardware designs were the ones that fused the linear function with the activation function for each layer into a single operation. Each fused layer of the network was pipelined, which improves throughput of the network at the expense of latency. The parallelization factors were chosen to be larger in the layers with more neurons as this is better for amortizing the cost of the resulting deep reduction trees. Tiling factors were chosen to fit the majority of the problem on-chip in order to get the most benefit from the DRAM controller and short latency for on-chip memory accesses.

4.2 Lattice Network Construction and Tuning

We first started our exploration of lattices with simple hyperparameter sweeps based on existing models used by Google. They presented a model that would predict income of an individual based on various characteristics taken during a census (13). Because this shared characteristics with our problem (high dimensional inputs with interactions between small sets of inputs) we used their model as a basis and started with lattices of size two. We chose the rank of each lattice based on simple inspection of the data set features. We noticed that sets of features of size three to five should have some correlation (for example, there are four different features based on same host connections) and as such we set the rank of each lattice to be four. Next we performed a geometric parameter sweep of both the learning rate as well as the number of lattices. We found the optimal set to be 16 to 32 lattices with a learning rate of 0.02. Our final set of architectures involved a layer of calibrators (one per feature) followed by a layer of lattices (16 or 32) and a reduction layer that combined all the results together and assigned a score that was compared against a threshold for anomalies. We noticed that we tended to overfit as evidenced by a high train accuracy and subsequent dropoff in test accuracy. To this end, we increased our torsion regularization from 0.002 to 0.2 in a geometric fashion. We found that this had little to no effect on the test accuracy. We believe this to be a result of the immaturity in this area. There will probably be more complex regularization schemes in the future that have larger effects. Similar to the FCNNs we used an ADAM optimizer, a squared error loss, and mini batches of size 10000 as recommended by the literature and existing models.

When implementing the lattice networks on the FPGA, we used some of the same general techniques as used for the regular FCNNs. However, these networks have unique features that allow us to do optimizations that don't make sense for other kinds of networks. We made heavy use of metaprogramming to fully pipeline the entire computation. This gives the hardware designs a significant advantage over software implementations in certain aspects, such as bi-directional sort for simplex layers and fully-expanded combination trees for hypercube layers.

4.3 Hardware Generation

We created an end to end flow that would allow us to take a model on the CPU and generate a bitstream that could be mapped onto an FPGA. First, we train a model in a high level deep learning framework like Tensorflow (11; 13) or TFLearn (12). Then we extract the structure of the neural network as well as the various parameters into sets of CSV files. These CSV files are then read into a Scala program that invokes metaprogrammed kernels, populates them with the appropriate parameters, and wires the kernels together. This process produces a Spatial graph that represents the network. This graph is passed through the Spatial compiler which in turn produces Verilog. This Verilog is synthesized using Xilinx tools and mapped to an FPGA.

5 Experiments/Results/Discussion

In this section, we present a variety of designs on both the CPU and FPGA for FCNNs and lattices. The CPU that we used was an Intel(R) Xeon(R) CPU E7-8890 v3 @ 2.50GHz. The FPGA that we used was a Xilinx ZCU102 Ultrascale MPSoC @ 100MHz. This FPGA is towards the lower end of FPGAs in the market today, but is a common board for embedded tasks such as networking and image processing.

In terms of accuracy, Table 2 shows that lattice regression performs worse than standard fully connected FCNNs. This was because of the overfitting problem mentioned earlier. We believe that a better regularization method may be needed for our lattice networks. Both sets of models had very high recall which is promising. However, lattices generally have an order of magnitude less false negatives which is crucial for security. The FCNN architectures always misclassify thousands of malicious packets as benign (although these errors account for only 0.01% of the test set), which is not acceptable for a real system. As expected, these mispredicts tended to happen on attack classes with few samples. We experimented with having more complicated, deep networks versus shallow, wide networks and found that adding complexity does not necessarily improve accuracy, likely because the malicious packets don't need to be processed in a more abstract space (deeper layers) to be detected.

In terms of FCNN runtime, Table 1 shows that the CPU has much more trouble with throughput as the networks get deeper. However, the FPGA is able to maintain throughput even with deeper

	Arch.	CPU (infs/s)	FPGA (infs/s)	Speedup from CPU to FPGA	Single Inference Runtime on FPGA (ns)	DSPs (% of FPGA)	LUTs (% of FPGA)
FCNN	[10,1]	13358	617238	46.2	162.01	3.97	52.15
	[96,1]	10205	114667	11.24	872.1	3.97	51.64
	[28,28,28,1]	9663	234714	24.29	426.0	7.14	59.39
	[10 Layer*]	3182	38389	12.06	2604.8	10.32	73.95
Lattice	Simplex [16]	17.9	1e8	5.6e6	9.99	12.70	17.75
	Simplex [32]	18.5	1e8	5.4e6	9.99	25.40	24.84
	Hypercube [16]	17.9	1e8	5.6e6	9.99	86.35	21.19
	Hypercube [32]	17.5	1e8	5.7e6	9.99	172.22	33.74

Architectures are described in format [X, Y] indicating a layer with X hidden units then a layer with Y hidden units. *The 10 Layer Architecture is [128,128,64,64,64,64, 64,64,32,32,1]

Table 1: Table of Speed Metrics

	Arch.	Train Accuracy	Test Accuracy	False Negatives	Recall
FCNN	[10,1]	0.9999	0.9990	4202	0.9999
	[96,1]	0.9999	0.9990	2981	0.9999
	[28,28,28,1]	0.9999	0.9991	3266	0.9998
	[10 Layer*]	0.9999	0.9937	3041	0.9999
Lattice	Simplex [16]	0.9988	0.7685	253	0.9999
	Simplex [32]	0.9970	0.7699	6090	0.9935
	Hypercube [16]	0.9998	0.7687	377	0.9996
	Hypercube [32]	0.9998	0.7686	879	0.9990

Architectures are described in format [X, Y] indicating a layer with X hidden units then a layer with Y hidden units. *The 10 Layer Architecture is [128,128,64,64,64,64, 64,64,32,32,1]

Table 2: Table of Accuracy Metrics

networks. The component that slows down the FPGA is the widest network. This is because of the FPGAs ability to fully pipeline packets. While latency may be lower, throughput can be maintained. These throughput numbers are all acceptable, but unfortunately deeper networks do not result in better accuracy so this pipelining is not useful for this task. The FPGA implementation of the lattices showed a huge improvement in speed against the CPU code. This is in part because the CPU code is written in TensorFlow Lattice which is not yet highly optimized. However, the structure of the lattices also maps better to hardware because of the abundance of lookup tables and tree structures. The lattice layers are small enough that they can all be computed in parallel so the latency across models is constant. They all operate within less than 10 nanoseconds for a single inference making these models viable for even high end switching devices. We see from the DSP and LUT usage that they generally require less resources on the FPGA than FCNNs except for hypercubes which require many DSPs for weighted sum computations. Simplexes are very resource efficient though and emerge as the single best candidate resource-wise.

6 Conclusion/Future Work

To conclude, we recommend using fully-pipelined lattice regression on FPGAs for tasks with relatively small feature spaces that require high throughput and low latency. We ran experiments for two different classes of networks on two different devices to come to these results. We see from the results that neither lattices nor FCNNs are advantageous across every single metric. The FCNNs tended to have overall better accuracy while having more false negatives. The lattices, while having worse test accuracy, usually let an order of magnitude less malicious packets through. In addition, they had significantly shorter runtimes on the FPGA. High end switch latencies range from 200 to 400 nanoseconds so in the more extreme cases, the lattices are the only option for a machine learning solution. In the future, we would like to attempt to build networks that combine traditional FCNN units with lattices to achieve both speed and accuracy. We would also like to see the effect that monotonicity constraints in embedded lattice networks (3; 9) have in this situation.

Contributions

Both Tushar and Matt worked on all portions of the project. Matt focused more on the standard FCNN networks for both the CPU and the FPGA. This includes training, hyperparameter search, FPGA kernel implementations, and data analysis. Tushar's work was more focused on the lattice regression networks, for both the CPU and FPGA implementations as well as parameter and structure extraction. Both students worked together to figure out the most efficient way to implement lattice inference kernels on the FPGA. Because lattice regression is such a new field, it was challenging to figure out exactly how to work with it and debug it. The students spent a lot of time running experiments with these kinds of networks to get a baseline understanding and intuition about them. We feel that we have a much better understanding of how to manage and succeed in machine learning projects for tasks and frameworks that are novel and sparsely documented.

We received help from an external member, Nathan Zhang, who is a domain expert in lattice regression and was able to provide guidance and code support throughout the process. This involved things like initial setup, making reusable code components for the FPGA, helping us get started with lattice regression in Tensorflow, and overall guidance on how to navigate the hyperparameter space.

Code

Github Link: https://github.com/pyprogrammer/spatial-lattice
The README in our repository explains how to run both the Python and Spatial portions of our code. The installation script will install missing python libraries, as well as SBT to run the FPGA kernels. The repo includes a few sample points from our data set so that it is easy to quickly test the code. The repo is private so the CS230 Github account has been added as a collaborator.

References

[1] Garcia, Eric, and Maya Gupta. "Lattice regression." Advances in Neural Information Processing Systems. 2009.

[2] Mane, Vrushali D., and S. N. Pawar. "Anomaly based ids using backpropagation neural network." International Journal of Computer Applications 136.10 (2016): 29-34.

[3] Fard, Mahdi Milani, et al. "Fast and flexible monotonic functions with ensembles of lattices." Advances in Neural Information Processing Systems. 2016.

[4] Koeplinger, David, et al. "Spatial: a language and compiler for application accelerators." Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2018. https://spatial.stanford.edu

[5] Raghu, Prabhakar, et al. "Generating configurable hardware from parallel patterns." Proceedings of 21st ASPLOS: 651-665.

[6] Koeplinger, David, et al. "Automatic generation of efficient accelerators for reconfigurable hardware." Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. Ieee, 2016.

[7] You, Seungil, et al. "Deep lattice networks and partial monotonic functions." Advances in Neural Information Processing Systems. 2017.

[8] Garcia, Eric, and Maya Gupta. "Optimized construction of ICC profiles by lattice regression." Color and Imaging Conference. Vol. 2010. No. 1. Society for Imaging Science and Technology, 2010.

[9] Gupta, Maya, et al. "Monotonic calibrated interpolated look-up tables." The Journal of Machine Learning Research 17.1 (2016): 3790-3836.

[10] https://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data

[11] https://www.tensorflow.org

[12] https://www.tflearn.org

[13] https://github.com/tensorflow/lattice

[14] https://spatial.stanford.edu