Minimalism in deep learning

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

MINIMALISM IN DEEP LEARNING

by

LOUIS JENSEN

B.S., University of Notre Dame, 2017
M.S., Boston University, 2020

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2021

Approved by

First Reader

Peter Chin, PhD
Research Professor of Computer Science

Second Reader

George Kollios, PhD
Professor of Computer Science

Third Reader

Bryan Plummer, PhD
Assistant Professor of Computer Science

Fourth Reader

Steve Homer, PhD
Professor of Computer Science

Acknowledgments

Thank you to my adviser, Dr. Peter Chin, who guided my research. Thank you to my professors, who shared with me their knowledge. Thank you to my parents, who shaped me as a person.

A special thanks to Professors Steve Homer, George Kollios, Bryan Plummer, and Derry Wijaya for serving as my dissertation committee. A special thanks also to fellow PhD candidates Ryan, Trung, Peilun, Hieu, and Xiao for providing me helpful feedback on my dissertation document.

Louis Jensen
PhD Candidate
Computer Science Department

MINIMALISM IN DEEP LEARNING

LOUIS JENSEN

Boston University Graduate School of Arts and Sciences, 2021

Major Professor: Peter Chin, PhD
Research Professor of Computer Science

ABSTRACT

As deep learning continues to push the boundaries with applications previously thought impossible, it has become more important than ever to reduce the associated resource costs. Data is not always abundant, labelling costs valuable human time, and deep models are demanding of computer hardware. In this dissertation, I will examine questions of minimalism in deep learning. I will show that deep learning can learn with fewer measurements, fewer weights, and less information. With minimalism, deep learning can become even more ubiquitous, succeeding in more applications and on more everyday devices.

Contents

List of Tables

List of Figures

List of Abbreviations

AD	Anomaly Detection
AE	autoencoder
ARIMA	AutoRegressive Integrated Moving Average
AUC	Area-under-curve
AWS	Amazon Web Services
BMS	Balle, Minnen, Singh
BN	Batch normalization
BPG	Better Portable Graphics
BPP	Bits per Pixel
CIFAR10	Canadian Institute for Advanced Research 10 Class Dataset
CIFAR100	Canadian Institute for Advanced Research 100 Class Dataset
CNN	Convolutional Neural Network
COIN	Compressed Implicit Neural representation
CST	Cheng, Sun, Takeuchi
DNN	Deep Neural Network
EM	Electromagnetic
F1	F-score
FN	False Negatives
FP	False Positives
GAN	Generative Adversarial Network
GAT	Graph Attention Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit

HTM	Hierarchical Temporal Memory
JPEG	Joint Photographic Experts Group codec
LSTM	long-short term memory
MBT	Minnen, Balle, Toderici
MHz	megahertz
MNIST	Modified National Institute of Standards and Technology database
MSL	Mars Science Laboratory rover
MTAD-GAT	Multivariate Time-series Anomaly Detection Graph Attention Network
NAB	Numenta Anomaly Benchmark
NASA	National Aeronautics and Space Administration
PSNR	Peak Signal to Noise Ratio
RAM	Random Access Memory
RGB	Red, Green, Blue
RNN	Recurrent Neural Network
SIREN	Sinusoidal Representation Networks
SMAP	Soil Moisture Active Passive satellite
TADGAN	Time-series Anomaly Detection Generative Adversarial Network
TN	True Negatives
TP	True positives
VAE	variational autoencoder
VEEGAN	Variational Encoder Enhancement to Generative Adversarial Networks
VGG	Visual Geometry Group
VRAM	Video Random Access Memory
VTM	VVC Test Model

Chapter 1

Introduction

In the last decade, deep learning has revolutionized how we use computers to solve problems. The improved capability of computer hardware and the increased abundance of data made possible the meteoric rise of deep learning where before previous limitations had made widespread application impossible. Despite its meteoric rise, deep learning continues to be limited by its demands for heaping data and advanced hardware. Neural networks need training information to learn from, random access memory (RAM) to process the information in its models, and internet bandwidth to transmit the information of its models. In this dissertation I will examine questions of minimalism in deep learning. If deep learning can succeed with less, then deep learning can succeed at more.

There are many different types of information minimalism that are important in the advance of deep learning, and there exists a wealth of literature exploring these topics.

I will look at minimal information requirements, and mechanisms to encourage minimal information in deep learning. My research examines four different aspects of information minimalism in deep learning found in four unique application contexts.

In chapter two we study the application of neural networks to side-channel signal analysis. We trained a deep learning model to classify what type of program block was executed on a processor using only measurements of the processor's power drain and electromagnetic radiation. In particular, we determined the minimal number of measurements required to classify program block types, showing that measurements over a span of only four instruction cycles were sufficient to distinguish block types for our experimental set up.

In chapter three I propose the NodeDrop technique for minimizing the architecture size of neural networks. NodeDrop is experimentally demonstrated to be an effective mechanism for reducing the size of neural networks. NodeDrop also offers a theoretically proved guarantee that the network output will not be affected by its pruning at the end of training. This guarantee allows the network to adjust to changes in architecture continuously during training. The NodeDrop technique reduced the number of parameters in a VGG-like network by a factor of 114x while maintaining classification accuracy on the CIFAR10 benchmark.

Chapter four offers an autoencoder-based approach to time-series anomaly detection. The autoencoder reconstructs windows of time-series data from its latent dimension, learning to encode and decode frequent patterns appearing in data. When an anomaly occurs, the autoencoder struggles to reconstruct the anomalous data because it does not conform to learned patterns. The autoencoder relies on reducing information to a bottleneck at the latent dimension in order to fit only frequent patterns of data.

Chapter five gives a perspective on using neural networks to compress image data. We show how to use structured sparsity in neural networks to build upon recent work with neural representations [Dupont et al., 2021]. We show that adding structured sparsity to Dupont's technique is able to improve image compression ratio with maintained signal preservation.

All of these chapters address issues of minimal information in deep learning. This dissertation does not address all aspects of minimalism in deep learning, nor does it address information theory. Instead, this dissertation experimentally probes minimal information for deep learning in four diverse applications.

Chapter 2

Deep Learning for Minimal-Context Block Tracking through Side-Channel Analysis

2.1 Abstract

It is well known that electromagnetic and power side-channel attacks allow extraction of unintended information from a computer processor. However, little work has been done to quantify how small a sample is needed in order to glean meaningful information about a program's execution. This paper quantifies this minimum context by training a deeplearning model to track and classify program block types given small windows of side-channel data. We show that a window containing approximately four clock cycles suffices to predict block type with our experimental setup. This implies a high degree of information leakage through side channels, allowing for the external monitoring of embedded systems and Internet of Things devices.

2.2 Introduction

Computer processors consume power and emit low energy electromagnetic (EM) radiation according to the currents in their transistors and circuitry. Side-channel attacks utilize this unintended, external information to infer the current state of the processor. These attacks were first used to break cryptographic systems by analyzing the timing of processes to gain information about security keys [Kocher, 1996]. Since then, additional studies have demonstrated that power [Kocher et al., 1999], sound [Genkin et al., 2013], temperature [Hutter and Schmidt, 2014], and EM [Agrawal et al., 2003] outputs can all lead to viable side-channel attacks.

Side-channel attacks are frequently aimed at extracting private cryptographic secrets. Information from side channels can also be used in less malicious ways. As a defensive measure, analysis of side-channel signals can be used to classify machine states for malfunction detection. This is especially important on embedded systems, which are becoming more common in dedicated Internet of Things applications. Because of the specialized nature of these systems, they are frequently low in memory and processor power. As a result, traditional defensive methods such as on-device malware detection and self-monitoring are infeasible for ensuring security and maintenance. However, external malware detection and monitoring for these small systems can be achieved through side-channel analysis. Ideally, a monitoring system could detect both whether and when any deviation from a program's normal execution occurred. This requires a monitoring system to be able to track program execution.

A program's execution can be viewed as consisting of basic blocks separated by control flow, with each basic block containing a portion of assembly level instructions within which no branch or jump occurs. As a first step towards the monitoring of embedded systems, we aim to use side-channel analysis to track a program's sequence of basic blocks. We measure both the power consumption and EM radiation surrounding the target device, and then use a convolutional neural network (CNN) to segment the measured EM signals according to labelled block types at each observed time sample. In particular, we aim to show that these block types can be classified after observing side channels for only a very small number of clock cycles—showing that even these small windows of side-channel data still carry relevant leaked information.

2.3 Related Works

Problems in side-channel analysis include cryptanalysis, program-level classification, block-level tracking, and instruction level tracking. Many papers have demonstrated the ability of machine learning algorithms to tackle these problems through EM and power side channels [Riley et al., 2018, Maghrebi et al., 2016, Prouff et al., 2018, Wang et al., 2018, Hospodar et al., 2011, Liu et al., 2016]. Support vector machines have successfully been used for both cryptanalysis [Maghrebi et al., 2016, Hospodar et al., 2011] and program classification [Riley et al., 2018]. A handcrafted machine learning algorithm based on hidden Markov models has also been successfully applied to the sequencing of program execution at the instruction level [Liu et al., 2016].

Deep learning models have also been repeatedly demonstrated to successfully analyze the EM and power side channels. CNN models are frequently used for side-channel cryptanalysis [Maghrebi et al., 2016, Prouff et al., 2018], and recurrent neural network (RNN) models have also been used for cryptanalysis [Maghrebi et al., 2016]. We build upon the success of CNN and RNN models in this domain in order to implement our block-tracking model.

Deep learning models have been used less frequently for applications of side-channel analysis outside of cryptanalysis. A multilayer perceptron model was used for program classification [Wang et al., 2018] with good results. That model analyzed the same experimental traces we analyze in this paper, but used program labels instead of the block labels we use to approach the more difficult problem of block tracking.

As side-channel analysis becomes a topic of growing interest, these applications of machine and deep learning techniques have achieved many important results in this domain. To the best of the authors' knowledge, none of these works have explored the minimum bounds of context needed for analysis. The capacity of the EM and power side channels has been studied in specific cases designed to understand the most vulnerable instructions [Yilmaz et al., 2018, Zajic and Prvulovic, 2014, Callan et al., 2014]. Instead of focusing on specific components of the program, we test the minimum required context for block tracking across programs similar to those in real-world applications.

2.4 Data Collection

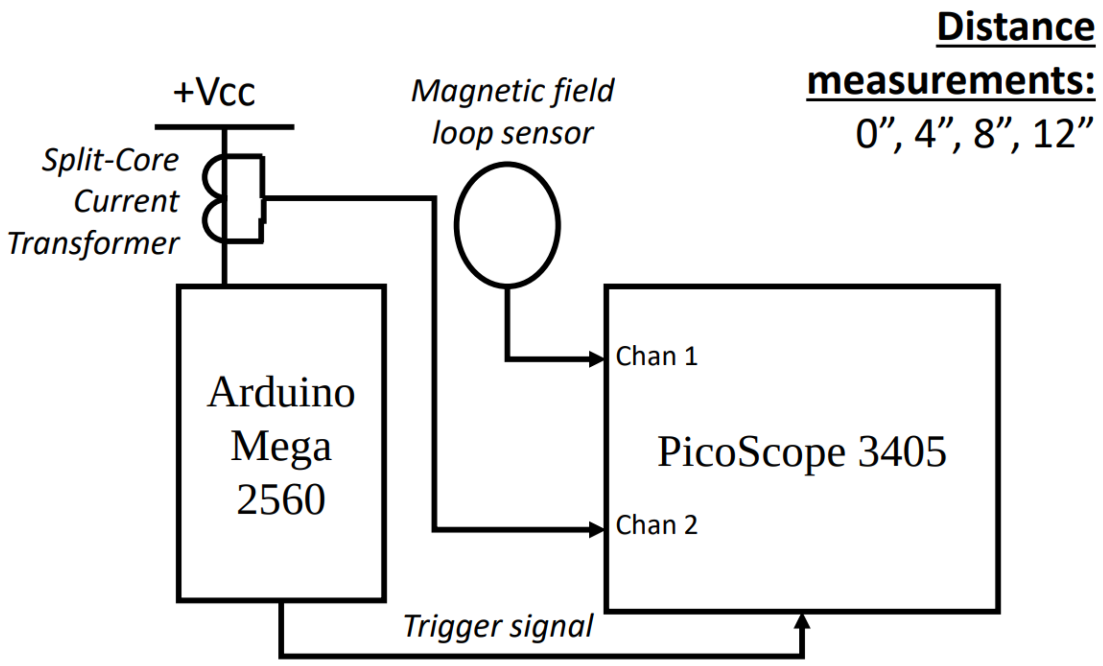

Figure 2·1: The two-channel data collection setup measures power consumption and EM radiation at 500 MHz.

Our data was generated on an Arduino Mega 2560 processor, which has a clock speed of 16 MHz. The time samples were measured at a rate of 500 MHz—providing roughly 31 samples per instruction. The samples were gathered on a PicoScope 6405, with each time sample containing two simultaneous measurements: power consumption and EM radiation. We measure power consumption by measuring direct current drain from the power source, and EM radiation by measuring the magnetic field strengths. The direct current measurements were taken via an ETS-Lindgren 94430 Split-Core Current Transformer. For distance measurements, the magnetic field was measured via an Aaronia MDF 9400 broadband magnetic field tracking antenna. For close measurements, a custom version of Beehive's 100A EMC Probe was pressed to the top of the processor.

Experiments were run under four different physical setups to test signal degradation over increasing distances. The "ideal" 0" samples were gathered with a magnetic probe directly on the processor and the current transformer placed after the power regulator. The 4", 8", and 12" measurements were gathered with the magnetic antenna at those distances from the processor. For these measurements, the current transformer was at a single fixed location before the regulator on the main power connector.

Signals were measured for two different programs: math and bit-toggle. The math program repeatedly generates random numbers and was labelled with five block types: write-low, write-high, random number seed, random number generation, and loop.

The bit-toggle program repeatedly flips the register between 0x00000000 and 0xffffffff. It was labeled with three block types: write-low, write-high, and loop.

2.5 Methods

2.5.1 Data Processing

During data collection we extracted the raw data as signal traces. Each time step of the traces was then labelled by block type using the program execution, i.e. write-low, write-high, etc. We first separated the data according to the 0", 4", 8", and 12" distances used in our data collection setups. We then subsequently separated the traces in each of these datasets into a training set of 2800 traces and a testing set of 180 traces. Each math program trace contained approximately 57,000 labeled samples and

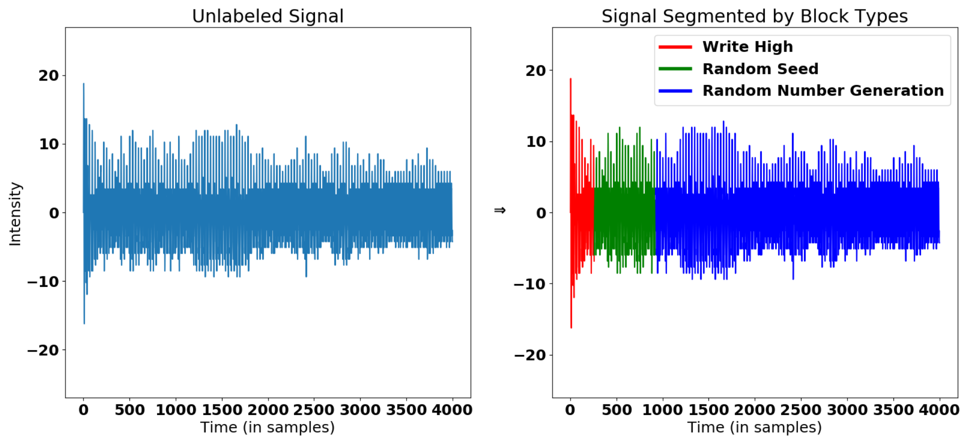

Figure 2·2: Example trace and correct block type classifications for each time sample. The colors in the figure on the right represent different block classes at those time samples. Note that only a fraction of a complete program trace is shown above

each toggle program trace contained approximately 9,000 samples. During training, we randomly sampled windows from the training traces with a bias to balance an unbalanced label distribution. For testing, we sampled without bias.

In order to find the minimal context required for block tracking, we broke the traces into windows of varying length. We tested our models with window sizes of 48, 64, 96, 128, 192, and 256 samples in length. Most windows were contained in a single block type. For the windows in transitions between block types, labels were proportional to the number of samples from each block.

2.5.2 Deep Learning Model Architecture

In the past several years, deep learning has achieved remarkable results in domains where data is plentiful and rich in structure [Goodfellow et al., 2016]. These domains include signal, image, and video processing. For signal processing, one-dimensional CNNs have proven very effective for extracting important signal features. Originally applied to images, CNNs learn layers of filters that are convolved with the input,

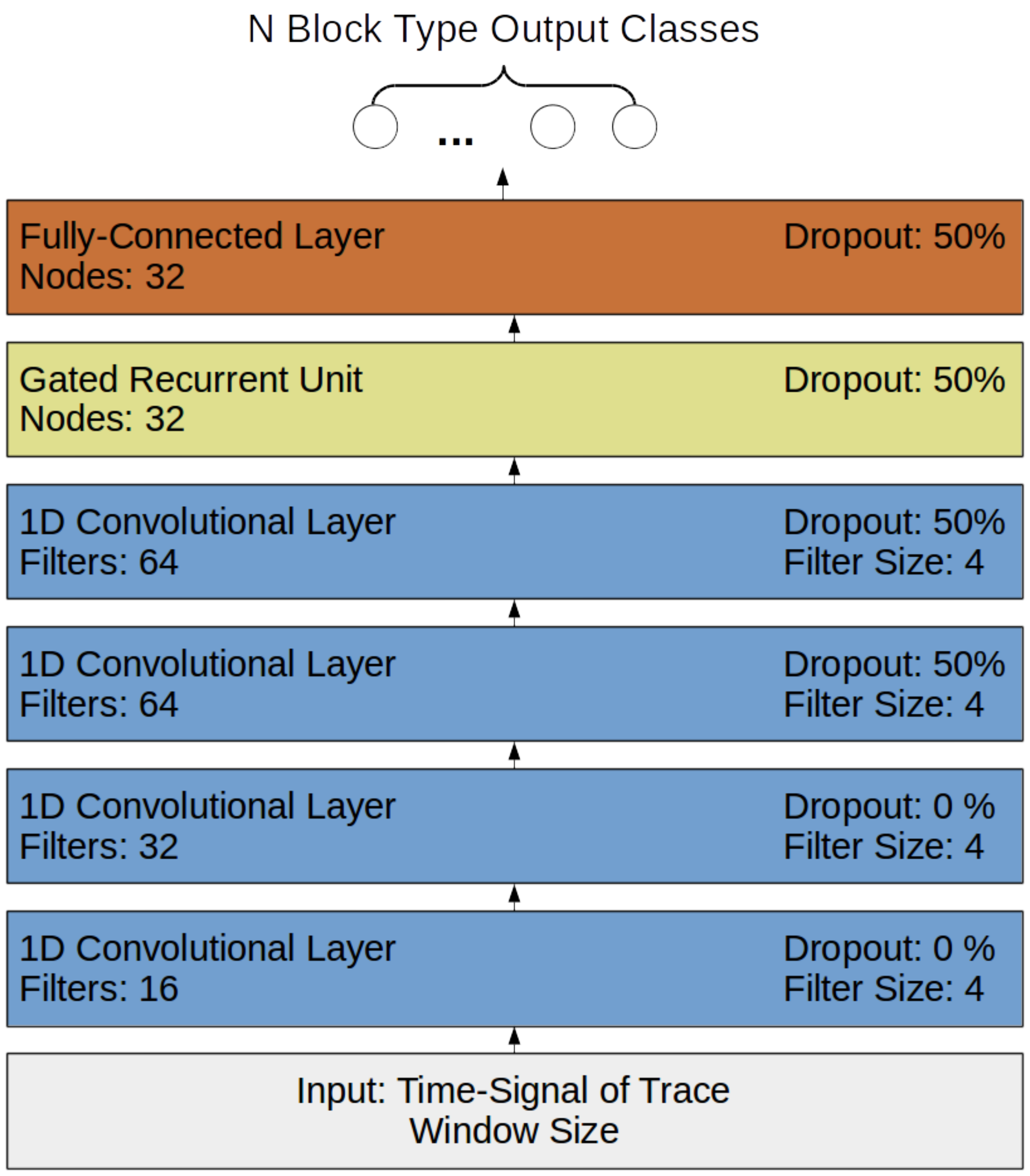

Figure 2·3: Our combined CNN and GRU architecture learns basic signal features in the lower layers and combines these features across time using a gated recurrent unit. This architecture classified block types with high accuracy across many different window sizes.

and then use the output of these repeated convolutions for classification or encoding. Another powerful deep learning model is the RNN, which operates sequentially on data while updating a hidden "memory" state. This hidden state allows the RNN to relate features across time. One successful RNN technique for handling the memory update is the gated recurrent unit (GRU) [Cho et al., 2014].

Our model architecture is shown in Figure 3. It consists of four convolution layers, a single GRU layer, and a final fully-connected layer. Inputs are processed by the convolution layers, generating basic signal features. We use convolution strides of length two instead of max-pooling in order to reduce dimension. The GRU layer then runs over the output of the convolution layers. The output of the GRU layer is then passed into the fully-connected layer which outputs a block classification. We utilized exponential linear unit activation functions [Clevert et al., 2015] on each layer except for the output layer, which uses a softmax to produce the classification. We implemented batch normalization [Ioffe and Szegedy, 2015a] on each layer and included dropout [Srivastava et al., 2014] on the four last layers to combat overfitting on the training data. Models for each dataset and window size were trained from random initialization using the Keras framework [Chollet et al., 2015] and optimized using the Adam optimizer [Kingma and Ba, 2014]. Each model was trained until a pre-determined stopping epoch, sufficient to ensure model convergence.

In our tests this model outperformed both models using a pure RNN applied to a spectrogram and models using fully-connected layers applied to the frequency domain. We trained an independent model for each window size and distance.

2.6 Experimental Results

We trained neural networks to perform block-type classification with different window sizes for the math program. The testing results of our models trained on the

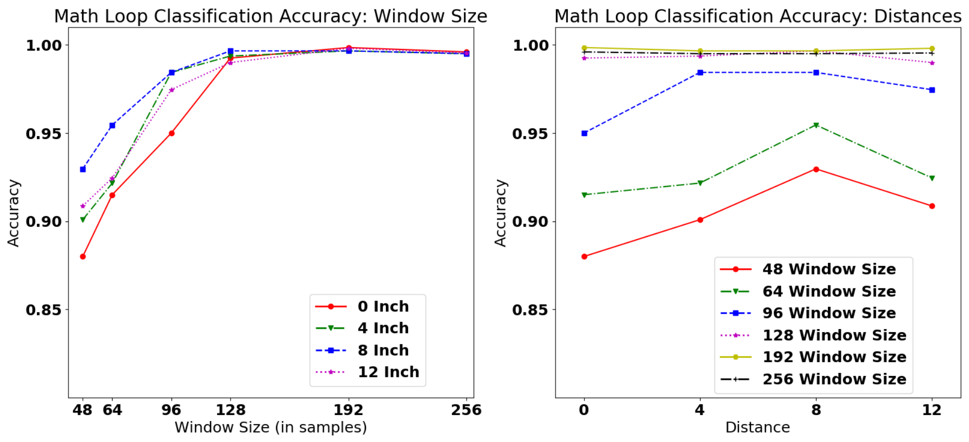

Figure 2·4: Classification accuracy across various window sizes and distances. Note how accuracy plateaus above a window size of 128 samples. We estimate this window size as the minimum context required for block-type classification—equivalent to approximately four clock cycles.

math datasets are plotted in Figure 4 for all window sizes and distances. We were pleased that our model achieved high classification accuracy for multiple window sizes. However, we noted that the smallest window sizes resulted in a noticeable decrease in classification accuracy. We used this drop in accuracy to estimate a minimum window length in which the windowed data still carries sufficient information to consistently classify block types. From our results, we estimated this minimum to be 128 sample windows, equivalent to approximately four clock cycles or 256 ns in our experiment setup.

We proceeded to verify this chosen window size using the bit-toggle program. We applied the 128 sample window size to the 0”, 4”, 8”, and 12” bit-toggle measurements. We found that our model again achieved high accuracy on the bit-toggle program. This verifies that the 128 sample window transfers well to new program executions in the same experimental setup. Both the math and toggle program results are found in Table 2.1.

		Distance			
Program	Window	0”	4”	8”	12”
Math	48	87.9	90.1	93.0	90.9
	64	91.5	92.2	95.5	92.5
	96	95.0	98.4	98.4	97.5
	128	99.2	99.4	99.7	99.0
	192	99.9	99.7	99.7	99.8
	256	99.6	99.4	99.5	99.5
Toggle	128	99.7	98.3	100	99.7

Table 2.1: Classification accuracy for each model's performance on the distinct test sets.

We did not observe any meaningful relationship between accuracy and measurement distance. This suggests that the measurement noise in this distance regime is negligible in comparison to the signal.

2.7 Chapter Conclusion

In this paper we empirically estimate an upper bound for the minimum window context required to carry information for predicting block type. With experimental setups measuring from multiple distances, we achieve a high block-type classification testing accuracy, over 99 percent, using a window context of four clock cycles. Our results can help to estimate baseline window sizes for different experimental setups than ours (e.g. different processors, sampling rates, distances, etc.).

Many directions remain in which to extend this research. The fine-grained classification of individual blocks directly supports program-level classification. Like all supervised classification, this work is limited by the need for labelled datasets. In the future, we plan to employ unsupervised learning techniques to detect anomalies without training on explicitly-labelled datasets. We hope to recognize deviations from desired program execution such as malware or malfunction. In particular, we hope to both identify anomalies in program execution and, using the window size determined in this work, identify when in program execution these anomalies occur.

Chapter 3

NodeDrop: A Method for Finding Sufficient Network Architecture Size with Improved Generalization

3.1 Abstract

Determining an appropriate number of features for each layer in a neural network is an important and difficult task. This task is especially important in applications on systems with limited memory or processing power. Many current approaches to reduce network size either utilize iterative procedures, which can extend training time significantly, or require very careful tuning of algorithm parameters to achieve reasonable results. In this paper we propose NodeDrop, a new method for eliminating features in a network. With NodeDrop, we define a condition to identify and guarantee which nodes carry no information, and then use regularization to encourage nodes to meet this condition. We find that NodeDrop drastically reduces the number of features in a network while maintaining high performance, reducing the number of parameters by a factor of 114x for a VGG like network on CIFAR10 without a drop in accuracy.

3.2 Introduction

A prime difficulty in neural network design is the appropriate tuning of network architectures. Choosing a size for each layer of a neural network is usually done by rough estimate, trial, and error. This imprecise process can often lead to network designs larger than needed to perform a particular task. Although the capacity for training large and complex networks grows with improving graphics processing unit (GPU) technology, designing too large a network can result in applications impracticable for general hardware use. Mobile devices and embedded systems limit compute, memory, and storage consumption, and as a result can only run small, minimally designed networks. A designer aiming to create such a minimal network is faced with the time-consuming task of manually tuning the number of neurons in each layer. This tuning process can result in many extended tuning experiments in order to balance the space and performance of the neural network.

The issues involved with using deep neural networks (DNN's) on constrained systems has inspired significant research. One interesting area of research is the design of systems which can automatically prune a network's parameters. Ideally these techniques can still maintain high performance while pruning as many parameters as possible, ensuring the network can fit on smaller systems. Many state-of-the-art methods for network pruning generally involve an iterative process of repeatedly pausing training, pruning parameters, and resuming training in order for the network to reconverge. Such iterative procedures can lead to long training times. Other techniques use regularization in order to eliminate nodes. The final performance of these networks are often highly variable with the hyper-parameters of the algorithm. Thus, while these techniques do offer parameter reduction benefits, the network designer will still be faced with similar difficulties as before: a time consuming training process and a potential hyper-parameter tuning headache.

We address the problem of parameter reduction with our novel NodeDrop technique, which prunes the network during training. The NodeDrop technique only drops nodes which carry no information and drops them fluidly during the training process.

First, we formally define the conditions necessary to guarantee a neuron carries no information. We then propose a simple variant of L_1 regularization which drives nodes toward this condition. Second, we extend the NodeDrop technique to networks which use batch normalization [Ioffe and Szegedy, 2015b]. We test our technique on modern architectures for the MNIST, CIFAR10, and CIFAR100 datasets, and show that we are able to drop a significant number of nodes without a loss in performance. Our method requires no iterative retraining and only a modest increase in training time. We demonstrate effective results with a wide range of hyperparamaters, indicating our method does not require precise hyperparameter tuning. At best case we produce a network which reduces the number of parameters by 93.27, 99.12, and 87.82 percent for MNIST, CIFAR10, and CIFAR100 respectively, with no perceivable loss in performance.

3.3 Related Works

3.3.1 Pruning

Network pruning comprises a set of techniques which take a pretrained network and then prune off connections using some heuristic. This is usually followed by a retraining of the network and sometimes by an iterative process of pruning and retraining the network several times. Pruning techniques first appeared in the 1990s, with the first instances using second order gradients of connections to determine which neurons should be pruned [Hassibi and Stork, 1993, LeCun et al., 1990, Reed, 1993]. More recent approaches have taken on a wide array of methods for determining which connections should be pruned. These approaches include correlation [Sun et al., 2015, Han et al., 2016, Srinivas and Babu, 2015], regularization [Han et al., 2015, Li et al., 2017], particle filtering on misclassification rate [Anwar et al., 2017], low rank approximation [Denton et al., 2014], vector quantization [Gong et al., 2014] and tensor decomposition [Kim et al., 2015].

All network pruning techniques suffer from extended training time due to the iterative retraining of the network. This can lengthen training times significantly, and often makes tuning the various parameters in each method a lengthy chore.

3.3.2 Regularization

A more recently developed approach to network parameter reduction is to disable parameters through regularization. A majority of these techniques have focused on the sparsification of network connections using a group sparsity approach [Wen et al., 2016, Zhou et al., 2016, Alvarez and Salzmann, 2016, Lebedev and Lempitsky, 2016]. This involves grouping the weights for every neuron and attempting to sparsify each group by penalizing its L_2 norm. These techniques require all weights to be driven to zero before a node can be guaranteed to carry no information. In practice nodes are removed based on a threshold since this guarantee is difficult to meet. Because of this, regularization methods can be difficult to use as they require very precise tuning of the regularization and threshold terms.

The most similar technique to ours, Liu et al. [Liu et al., 2017], uses L_1 regularization to drive the scale parameter in batch norm, γ, towards zero. This is similar in principle to our own experiments with batch norm. However, Liu et al. requires retraining after pruning in order to reconverge. We provide a more absolute condition to guarantee a node is off, eliminating the need for a retraining procedure and making node removal a more fluid process.

Our technique falls within the regularization category. Key differences in our approach involve special regularization of the bias for each neuron and a condition for node removal guaranteeing no effect on network output. Our condition is also more relaxed, utilizing the "dead" region in a node's activation function, instead of

Table 3.1: MNIST Network Architectures: Number of Features by Layer

Network Name	Layer 1 Conv2d	Layer 2 Conv2d Maxpool	Layer 3 Conv2d	Layer 4 Conv2d Maxpool	Layer 5 Dense	Layer 6 Output
Dense160	16	16	32	32	64	10
Dense240	24	24	48	48	96	10
Dense320	32	32	64	64	128	10
Dense480	48	48	96	96	192	10
Dense640	64	64	128	128	256	10

requiring the node's weights to be zero.

3.3.3 Other approaches

Several other approaches have appeared which do not fit into the categories of the previous two subsections. Many of these approaches focus on reducing precision as opposed to reducing the number of parameters. [Hubara et al., 2016, Vanhoucke et al., 2011, Gupta et al., 2015, Rastegari et al., 2016]. As such, these approaches are largely orthogonal to our own work, and can be used in conjunction with our work in order to compound the reduction on memory and computation. One example of this approach is quantized and binarized neural networks [Hubara et al., 2016], which take this approach to new levels by using $\{-1, 1\}$ weights and an XOR to replace multiplication.

An additional noteworthy paper is that of Molchanov et al. [Molchanov et al., 2017]. They achieve impressive results by sparsifying a network's connections during training using variational dropout. Again, in theory this work should be usable in conjunction with our own.

3.4 Experiments

Having established a theoretical basis for the NodeDrop condition and regularization technique, we will now establish NodeDrop's practical viability as a method for

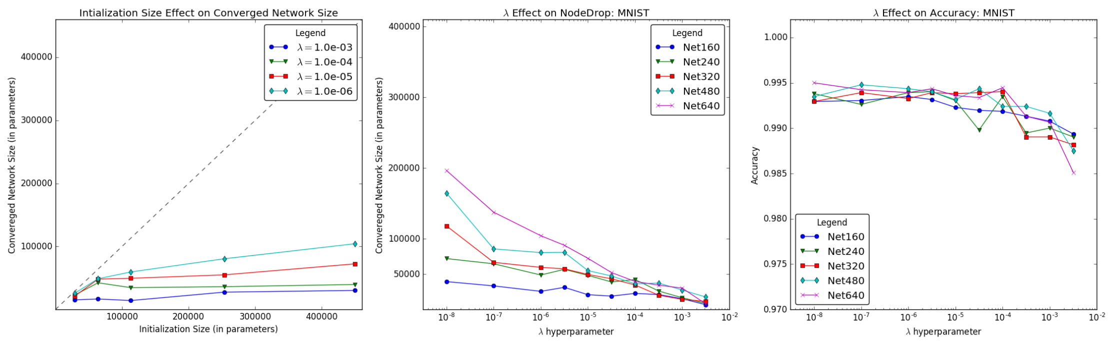

Figure 3·1: In the right and center figures, the λ parameter values plotted on the y-axis are on a logarithmic scale. We note that the performance and parameter reduction both maintain desirable levels for a large range of λ values (over several orders of magnitude). This indicates the ease of tuning the NodeDrop technique. In the leftmost figure, networks of different starting size converge to nearly the same size for a given λ. The dashed diagonal line represents networks without pruning. Note that increased initialization size has a slight effect on final size, as indicated by the slight upward slopes. This effect is greater for larger λ.

shrinking networks. The NodeDrop technique requires two hyperparameters: C and λ. The C value is unimportant, and can be set to almost any positive value without impacting results or parameter reduction. However, the λ parameter is crucial in determining the balance between learning the objective and dropping nodes. Therefore, we closely examine the effect that choosing different λ values has on both network performance and parameter reduction. We test many λ values on the MNIST and CIFAR10 datasets. We also test a few λ values on the CIFAR100 dataset.

The network initalization size should affect the number of nodes dropped. We show that if a network starts near optimal size, NodeDrop will maintain accuracy and only drop what few nodes it can. Furthermore, we show that if a network is grossly oversized at initialization, NodeDrop will drop many nodes and converge towards the same size as a smaller network initialization. This result is desirable, as

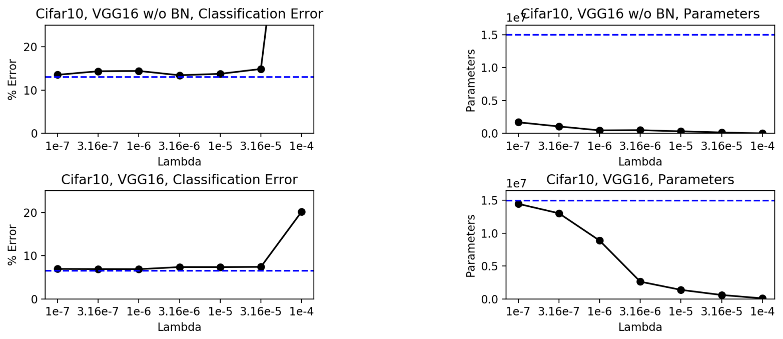

Figure 3·2: Results on CIFAR10 for VGG with and without Batch Normalization over a spread of λ choices. Top Left: Classification error for VGG without Batch Normalization. Top Right: Final parameters after training using NodeDrop. Bottom Left: Classification error for VGG with Batch Normalization. Bottom Right: Final parameters after training using NodeDrop-BN. For both NodeDrop and NodeDrop-BN, a range of λ values are acceptable. Baseline accuracy and network size is indicated by the dashed lines.

it demonstrates NodeDrop is largely unaffected by poor layer size choices. NodeDrop uses λ to determine the balance between performance and number of nodes utilized. Therefore, a network architect using NodeDrop can afford to initialize a large network, and remain confident that NodeDrop will eliminate needless nodes. Using the MNIST dataset, we demonstrate this ability by showing that networks will converge to the same size from multiple initialization sizes, for a fixed λ.

Many pruning methods require an increase in training time to be effective. The NodeDrop technique does not delay performance or accuracy convergence, but in order to allow the number of network nodes to converge, one must train for a longer time. We examine the training time required for this convergence with experiments on the CIFAR10 datasets.

Most importantly we test to ensure NodeDrop maintains performance and effectively drops nodes. We find that NodeDrop regularization does not affect a network's performance for a large swath of λ values, only reducing testing accuracy if extreme

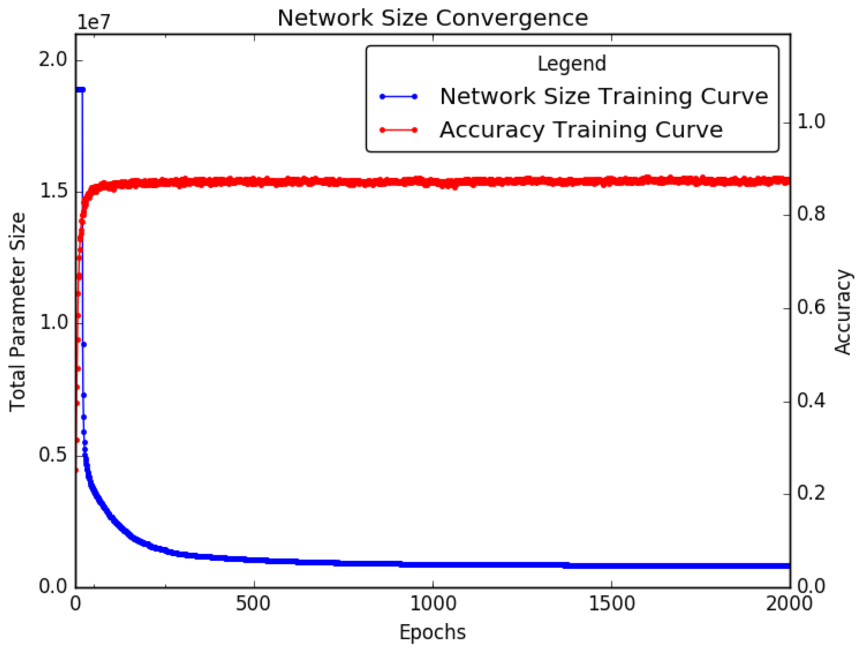

Figure 3·3: Accuracy stabilizes after less than 100 epochs in this CIFAR10 run, indicating the NodeDrop technique does not delay performance convergence. Training for another 400 epochs helps maximize parameter reduction.

λ values are chosen.

Furthermore, we demonstrate that NodeDrop is able to drop more than 100x parameters from popular networks such as VGG16, while continuing to maintain classification accuracy on the CIFAR10 dataset. We test NodeDrop network performance and parameter reduction on MNIST, CIFAR10, and CIFAR100.

3.4.1 MNIST Experiments

The MNIST dataset [LeCun and Cortes, 1998] provides an opportunity to perform a large number of experiments because of the datasets rapid accuracy convergence. Thus, we used this dataset to sweep across λ values for five differently sized, but otherwise similar, network architectures, as shown in table 3.1. We demonstrate NodeDrop's ability to rapidly converge to similarly sized networks from different

Table 3.2: Cifar10 Classification Results

Network	λ	Error	Params	Prune %	Factor	Nodes	Prune %
VGG 16 w/o BN	Baseline	13.01	15.04M	0.0	1.0	4736	0.0
	1.0×10^{-6}	14.14	0.45M	97.00	33.28	1115	76.46
	1.0×10^{-5}	13.27	0.31M	97.96	48.98	859	81.9
	3.2×10^{-5}	**13.76**	**0.13M**	**99.12**	**114.00**	**612**	**87.08**
	1.0×10^{-4}	90.00	0.0M	100.0	-	0	100.0
VGG 16	Baseline	6.50	15.04M	0.0	1.0	4736	0.0
	1.0×10^{-6}	6.88	8.88M	40.7	1.69	3624	23.48
	1.0×10^{-5}	7.36	1.39M	90.75	10.81	1164	75.42
	3.2×10^{-5}	**7.41**	**0.61M**	**95.96**	**24.76**	**751**	**84.14**
	1.0×10^{-4}	20.16	0.10M	99.35	152.84	308	93.50
DenseNet40 w/o BN	Baseline	14.94	1.04M	0.0	1.0	456	0.0
	1.0×10^{-6}	15.21	0.66M	35.69	1.55	363	20.39
	1.0×10^{-5}	14.74	0.41M	60.47	2.54	291	36.18
	1.0×10^{-4}	**14.99**	**0.08M**	**91.96**	**12.43**	**154**	**66.22**
DenseNet40	Baseline	6.80	1.05M	0.0	1.0	456	0.0
	1.0×10^{-6}	7.13	0.99M	4.19	1.04	447	1.97
	1.0×10^{-5}	6.75	0.98M	5.67	1.06	443	2.85
	1.0×10^{-4}	**7.79**	**0.55M**	**47.12**	**1.89**	**333**	**26.73**

starting sizes.

For all MNIST experiments we used a simple network design: four convolution layers and a single fully connected layer. We used 3x3 filters in all convolution layers, and performed max-pooling after every second convolution layer. We varied the width of the layers in order to test the effects of changing network initialization size. We did not investigate the effects of changing network depth, but suspect that prudent selection of network depth remains important. The network architectures are described in table 3.1. The following consistent hyperparameters were used across all MNIST runs: learning rate = 1.0×10^{-3}, batch size = 1024, optimizer = *Adam*, loss function = *cross entropy*, epochs = 480.

Choosing Lambda

Choosing an appropriate value for the NodeDrop's λ parameter remains an important task. In order to prove that the NodeDrop technique remains robust for many selec-

Table 3.3: Cifar100 Classification Results

Network	λ	Error	Params	Prune %	Factor	Nodes	Prune %
VGG 16	Baseline	27.65	15.04M	0.0	1.0	4736	0.0
	1.0×10^{-6}	27.69	9.78M	34.99	1.54	3914	17.35
	1.0×10^{-5}	**28.04**	**1.83M**	**87.82**	**8.21**	**1623**	**65.73**
	1.0×10^{-4}	38.49	0.46M	96.93	32.58	729	84.6
DenseNet40	Baseline	26.5	1.05M	0.0	1.0	456	0.0
	1.0×10^{-6}	26.92	1.05M	2.27	1.02	451	1.09
	1.0×10^{-5}	**27.01**	**1.03M**	**4.74**	**1.05**	**445**	**2.41**
	1.0×10^{-4}	29.38	0.744M	31.12	1.45	376	17.54

Table 3.4: ImageNet Classification Results

Network	λ	Error	Params	Prune %	Factor	Nodes	Prune %
VGG 19	Baseline	33.79	143.65M	0.0	1.0	14696	0.0
	1.0×10^{-5}	34.85	23.75M	83.47	6.05	6670	54.61

tions of λ, we tested five different network initialization sizes to observe differences in convergence across λ values. The network architectures and hyperparameters are discussed in section 3.4.1. We tested ten different λ values between $\lambda = 1.0 \times 10^{-8}$ and $\lambda = 1.0 \times 10^{-3}$.

Our results indicate that easy tuning is a benefit of the NodeDrop technique. We found that λ selections across orders of magnitude yielded desirable results, as shown in figure 3.1. For $\lambda > 10^{-4}$ we noticed a drop in MNIST accuracy, and for $\lambda < 1.0 \times 10^{-7}$ we judged there to be a significant sacrifice in parameter reduction. Choosing appropriate λ will always be dependent on both application and loss function. Because of these MNIST experiments, we expect that the NodeDrop technique is robust for a large range of λ selections. For a network designer using the popular cross-entropy loss objective function, as we did, we would suggest $\lambda = 1.0 \times 10^{-5}$.

Network Sizes

In the previous section (3.4.1) we experimentally observed that tuning the λ parameter of the NodeDrop technique should not cause a network designer grief. In this section, we will experimentally observe that choosing initialization layer sizes should also prove easy. We use the same experiments from the previous section (3.4.1), but instead plot the effect of initializing with differently sized networks. This plot, shown in figure 3.1, demonstrated that the NodeDrop technique will converge to a similar "equilibrium" from many differently sized initialization networks. The size of the final network is instead mostly dependent on λ. A network designer should err towards too large a network in order to ensure desirable performance.

3.4.2 CIFAR10 and CIFAR100 Experiments

Dataset

The CIFAR dataset [Krizhevsky et al., 2009] consists of 32x32 colored natural images. Both CIFAR10 and CIFAR100 are designed for classification, containing 10 and 100 classes respectively. There are $50,000$ training images and $10,000$ testing images for both. We adopt a standard data augmentation scheme where the training images are shifted and mirrored horizontally [He et al., 2016, Liu et al., 2017].

Architectures and Training

We implement our technique on two standard models, VGG [Simonyan and Zisserman, 2014] and DenseNet [Gao Huang, 2017]. Our VGG network is a slight variant of the standard VGG16 model. We follow the standard modification of VGG for CIFAR [Liu et al., 2017, Molchanov et al., 2017], by removing the three final fully connected layers of size 4096 and instead using only a single fully connected layer of size 512. We train the network using SGD with momentum of 0.9. The network is trained for 200 epochs with an initial learning rate of 0.1 which is decayed by 0.1

at epochs 80 and 130. We tested both with and without batch normalization, and discovered that batch normalization is necessary for the large VGG16 initialization when applied to the more difficult CIFAR100 dataset. Therefore results without batch normalization are excluded for CIFAR100.

For DenseNet we implement the standard DenseNet-40 given in the original paper with $L = 40$ and $k = 12$. We train the model as per the original paper with SGD and momentum 0.9. The network is trained for 300 epochs with an initial learning rate of 0.1 and is decayed by 0.1 at epochs 150 and 225. As with VGG we found that the CIFAR100 dataset required batch normalization, but we were again able to train a variant on CIFAR10 without batch normalization.

Lambda Parameter Tests

As with the MNIST experiments, we tested a range of λ's on CIFAR10 in order to determine the choices which suit the network and dataset well. Furthermore, here we test NodeDrop-BN, which was not tested in the MNIST experiments. Results for VGG on CIFAR10 with varying choices of λ are shown in figure 3·2.

For the case without batch normalization our network maintains performance and prunes a large number of nodes over many choices of λ. As with the MNIST case, this suggests that choosing λ is relatively easy. All choices of $\lambda \leq 3.2 \times 10^{-5}$ achieved high performance with significant pruning. For $\lambda \geq 1.0 \times 10^{-4}$ the regularization parameter proved too high, causing an entire layer to turn off, which in turn caused the network to turn off all other layers.

For NodeDrop-BN, we find that $\lambda \leq 3.2 \times 10^{-5}$ is appropriate for maintaining performance. However, NodeDrop-BN requires more precise tuning than NodeDrop, as only $\lambda \geq 3.2 \times 10^{-6}$ achieved desirable parameter reduction. Based on the above results we continue to recommend an initial lambda setting of $\lambda = 1 \times 10^{-5}$ for the cross-entropy loss objective function.

Network Convergence Time

Sometimes it is important to avoid needlessly extending training time. In this section we analyze NodeDrop's effect on training time. Using $\lambda = 10^{-5}$, we train a network for 2000 epochs in order to observe network parameter and performance convergence over time. This experiment used the VGG16 network without batch normalization on the CIFAR10 dataset. Our results, shown in figure 3.3, indicate that while accuracy convergence is not delayed by the NodeDrop technique, one will need to wait longer to maximize NodeDrop's parameter reduction.

Parameter Reduction

Results for CIFAR10 and CIFAR100 are given in tables 3.2 and 3.4 respectively. We highlight the rows which provide the highest parameter reduction while maintaining high accuracy.

For the VGG network we are able to drop a significant number of parameters without degradation to the accuracy of the network. For NodeDrop-BN, we can prune 95 percent of the parameters for CIFAR10 and 88 percent for CIFAR100. For vanilla NodeDrop, we can prune 99 percent of the parameters on CIFAR10. This suggests that VGG is a significantly oversized network for application to the CIFAR datasets.

It is more difficult to prune nodes from the DenseNet architecture than for VGG. We are only able to prune approximately 5 percent of the parameters from DenseNet on CIFAR100. We believe this suggests that the DenseNet architecture is already well sized for CIFAR100. DenseNet starts at around 1 million parameters, which is close to the number of remaining parameters after our best case pruning of the VGG network.

3.5 Conclusion and Outlook

In this paper, we proposed the novel NodeDrop technique for reducing parameters in neural networks. The NodeDrop technique consists of a condition for identifying nodes which are guaranteed to carry no information, and a regularization term to encourage this condition to be met. We also propose a modified version of NodeDrop, NodeDrop-BN, for use in networks with batch normalization. Experiments on the MNIST and CIFAR10 datasets show that NodeDrop does not significantly increase training time, and facilitates network design with the easily tuneable hyperparameter λ. With experiments on MNIST, CIFAR10, and CIFAR100 datasets, using VGG16 and DenseNet architectures, we demonstrate that NodeDrop compares favorably with other parameter reduction techniques. NodeDrop reduces the number of parameters in a network by up to a factor of 114x. We hope that NodeDrop and NodeDrop-BN will prove useful in neural network design, and will help to make the implementation of neural networks on constrained systems more practical.

Chapter 4

How Dense Autoencoders can still Achieve the State-of-the-art in Time-Series Anomaly Detection

4.1 Abstract

Time series data has become ubiquitous in the modern era of data collection. With the increase of these time series data streams, the demand for automatic time series anomaly detection has also increased. Automatic monitoring of data allows engineers to investigate only unusual behavior in their data streams. Despite this increase in demand for automatic time series anomaly detection, many popular methods fail to offer a general purpose solution. Some demand expensive labelling of anomalies, others require the data to follow certain assumed patterns, some have long and unstable training, and many suffer from high rates of false alarms. In this paper we demonstrate that simpler is often better, showing that a fully unsupervised multi-layer perceptron autoencoder is able to outperform much more complicated models with only a few critical improvements. We offer improvements to help distinguish anomalous subsequences near to each other, and to distinguish anomalies even in the midst of changing distributions of data. We compare our model with state-of-the-art competitors on benchmark datasets sourced from NASA, Yahoo, and Numenta, achieving improvements beyond competitive models in all three datasets.

4.2 Introduction

In recent years we have seen an increase in the amount of time series data. Much of this time series data requires monitoring in order to determine whether the time series system is behaving normally or anomalously. Systems are designed to account for normal behavior, behavior that is well understood, but are not designed to account for anomalies, portions of data not within the well understood range of normal behavior. Therefore, it is desirable to design monitoring algorithms which will automatically detect anomalous behavior. Identification of anomalous data enables data to be investigated, reacted to, and accounted for in the future.

In time series data, an anomaly is a point or period of unusual behavior. Anomalies are difficult to define with specificity because anomalies vary in type, only united by the fact that each does not conform with "usual" behavior. A few examples of types of anomalies include point anomalies, contextual anomalies, and collective anomalies [Chandola et al., 2009]. Point anomalies are data instances where a point is at an unusual value relative to the global distribution of points. Contextual anomalies are data instances where a point is at an unusual value relative to other the local distribution of points (near in time) in the data. Collective anomalies are subsequences that are unusual relative to other subsequences in the data. Together, these categories account for anomalous spikes, unusual trends, anomalies by omission of a periodically repeated sequence, unusual changes in correlation for multivariate datasets, and more.

General purpose anomaly detection is challenging not only because there are many different types of anomalies, but also because many different types of time series data are in need of monitoring. Time series data appear in domains such as finance, social media analytics, healthcare, computer hardware monitoring, weather, and many others. Different data streams feature trends, periodicity, noise, and can be either univariate or multivariate.

Many techniques have been employed for time series anomaly detection, ranging from statistical methods to deep learning methods. A good anomaly detection technique must model the "usual" in order to identify the unusual. Statistical methods [Din, 2015] make assumptions about the underlying data distribution, and deep learning methods [Hundman et al., 2018] learn those assumptions. Most deep learning models will learn either to forecast or to reconstruct windows of time series data. If the model forecasts or reconstructs incoming new data with low error, the new data is expected to be normal, but when the model struggles to forecast or reconstruct accurately, then the new data is flagged anomalous. These models divide well understood normal data from poorly understood anomalous data.

Despite deep learning's unmatched ability to learn both simple and complex features of data, deep learning for time series anomaly detection faces a few significant hurdles. In fact, deep learning's ability to learn complex features threatens to overfit data [Zhang et al., 2017] and interpret too much data as normal. While simple statistical approaches are often inadequate to explain the complex structures of many time series datasets, deep learning algorithms can often explain both anomalous and normal data, making distinction impossible. Finally, while statistical methods rely on assumptions, e.g. domain specific knowledge, deep learning methods rely on a large amount of representative data.

In order to overcome the difficulties of applying deep learning to time series anomaly detection, a few approaches have been used. A common approach is to limit the model's capacity through the latent dimension of an autoencoder in order to avoid overfitting. Combining multiple loss functions is another technique which can prevent a deep learning model from overfitting [Geiger et al., 2020, Zhao et al., 2020, Kukacka et al., 2017].

The high demand for monitoring of temporal data streams with a large diversity of datasets and anomaly types requires a general purpose approach capable of detecting many types of anomalies in many types of data. Because data labeling is expensive, a good general purpose approach must not rely on labeled data or supervised techniques.

In this paper we demonstrate that a simple autoencoder with a few critical post-processing steps is able to outperform more complex techniques as an unsupervised, general purpose approach for anomaly detection. Our model is able to outperform more complex state-of-the art-competitors on three benchmark collections of data: Yahoo, Numenta (NAB), and NASA.

4.3 Related Works

In this section we will address the current literature for time series anomaly detection. Three families of techniques stand out in particular: the forecasting techniques, the reconstruction techniques, and the distribution techniques. We will discuss each of these techniques and also specific methods that use one or a combination of these techniques.

4.3.1 Forecasting Techniques

One of the standard techniques applied to time series anomaly detection depends on the idea that normal data should be predictable from the immediately previous data, and anomalies should be unpredictable. This family of techniques includes both statistical models and deep learning models.

Statistical models such as ARIMA [Din, 2015] make assumptions about the distribution of the data. ARIMA in particular assumes stationarity in the data distribution in order to forecast future values. The model predicts the next value using these assumptions, and flags anomalies where predictions have high error. For best results the time series may require domain-specific featurization as a preprocessing step.

Deep learning forecasting models address a supervised regression problem. The model is given as input a past window, and trained to predict the value or values immediately following. The model flags anomalies where predictions have high errors. This approach is unsupervised in the sense that no anomaly labels are required. However, deep forecasting approaches fall in danger of overfitting, especially if the network is not carefully parametrized to a simple size.

Examples that use forecasting to determine anomalies include Hundman [Hundman et al., 2018] and MTAD-GAT [Zhao et al., 2020]. Another example of a deep forecasting approach is the Hierarchical Temporal Memory (HTM) model [Ahmad et al., 2017]. This model first encodes a hidden state, and then predicts the next hidden state encoding instead of the next raw data value. Again, large error suggests an anomaly.

Forecasting approaches are an important and popular family of techniques for time series anomaly detection because of their intuitive design and strong results, but deep learning forecasting models frequently suffer from overfitting if not tuned carefully.

4.3.2 Reconstruction Techniques

Most deep learning algorithms rely at least in part on using autoencoder reconstructions to detect anomalies. Reconstruction techniques encode a window of data into a low dimensional latent space and then decode back to a reconstruction of the original window. Because the latent space is lower in dimension than the input window, information is lost in this mapping, and if the model reconstructs the original window poorly, this indicates significant information loss. Ideally, the information lost in an autoencoder is noise information rather than signal information.

Reconstruction techniques make use of the standard Auto-Encoder [Bank et al., 2020] or the Variational Auto-Encoder [Kingma and Welling, 2019] to map from the high to low dimensional space and back again. These models must be trained on past data to learn to reconstruct common patterns in the data.

Deep reconstruction models benefit from the low-dimensional latent space, making overfitting less likely, but like all deep models, they still often suffer from overfitting. Reconstruction loss has been combined with other loss functions such as a forecasting loss [Zhao et al., 2020], a generative adversarial loss [Geiger et al., 2020], or a regularization loss [Kukacka et al., 2017] in order to avoid overfitting.

TADGAN [Geiger et al., 2020] uses reconstruction error in combination with an adversarial approach, and MTAD-GAT uses forecasting in combination with reconstruction error. Omnianomaly [Su et al., 2019] uses a recurrent variational autoencoder as a reconstruction model to detect anomalies.

Our technique is a reconstruction technique using post-processing to overcome a few common pitfalls.

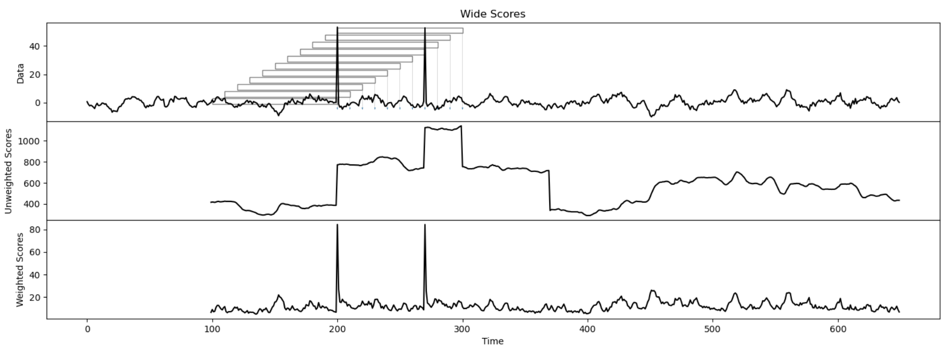

Figure 4·1: Without any post-processing, a sliding window over the data (1st plot) will yield a high anomaly score for any window in which the anomaly is overlapping, resulting in wide regions of high predicted anomaly scores (2nd plot). This is undesirable behavior, especially when two anomalies appear in rapid succession, as these anomalies will not be distinguished from each other by a threshold. If anomaly scores are calculated by weighted sum (3rd plot) over the sliding window instead of a simple sum, anomalies can easily be distinguished. This data was synthetically created with two spike anomalies to best illustrate this concept.

4.3.3 Distribution Based Techniques

Distribution techniques model the underlying frequency distribution of the data in order to determine whether an event is rare and anomalous or common and normal.

To determine whether a person's height is unusually low or high one can compare that person's height with the Gaussian distribution of heights to understand the rarity of that height; similarly, distribution techniques either assume or learn the underlying distribution to establish rarity.

The distribution technique is frequently implicit in the other techniques [Probst and Rothlauf, 2020], and is also explicitly combined with the reconstruction technique in TADGAN [Geiger et al., 2020], but to our knowledge has not yet been effectively used stand-alone for time series anomaly detection.

Schlegl [Schlegl et al., 2019] uses the discriminator of a generative adversarial network to predict whether medical image data is in-distribution or out-of-distribution, and [Geiger et al., 2020] similarly uses a discriminator score to predict anomalies in time series, but does so in combination with reconstruction error. GANs represent the current state-of-the-art in the distributional approach for time series anomaly detection.

While the theory of distributional techniques is most attractive, and some recent success has been achieved, a GAN approach has never been successfully used stand-alone for general purpose time series anomaly detection because of one major drawback: mode collapse. Even state-of-the-art techniques combating mode collapse such as VEEGAN [Srivastava et al., 2017] achieve only a reduction of mode collapse. For many GAN applications, mode collapse is not particularly damaging, but for detecting anomalies based on the assumption that the discriminator has learned the true underlying distribution, mode collapse can be catastrophic. GANs also suffer from training instability and long convergence times [Kodali et al., 2017]. TADGAN performs well because it combines the distribution approach with the reconstruction approach, as indicated by the ablation study found in their paper. [Geiger et al., 2020].

4.4 Methods

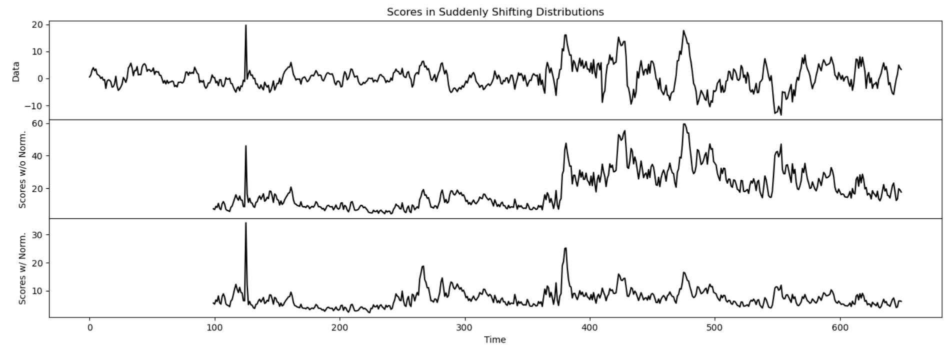

Figure 4·2: Noise levels may also change suddenly in a time series (1st plot). Using normalization (3rd plot) again improves anomaly scoring when compared with no normalization (2nd plot). With normalization a range of thresholds are able to be selected that will still yield good results.

After experimenting with TADGAN and Omnianomaly, we realized that a multilayer perceptron autoencoder was able to perform competitively with state-of-the-art approaches. Using a simple autoencoder also offers several advantages beyond performance, including fast and stable training, simple hyperparameter tuning, easy debugging, and easy identification of issues.

First, we use a sliding window across the time series to generate training samples, then the model learns to reconstruct these samples, and finally we predict anomalies by thresholding reconstruction error. This approach is simple, practical, and powerful for time series anomaly detection. Despite this model's effectiveness, the model still requires post-processing steps to reduce a high rate of false alarm. For all the datasets we used for benchmarking, we used a sliding window size of 100 and a latent dimension of 10.

Our contributions come not from our deep learning model, but instead in how that model is used to create meaningful anomaly scores. In particular we address three issues that repeatedly appear in autoencoder results – the issue of predicted anomaly width corresponding to sliding window width, the issue of anomaly scores changing with either sudden or trended distribution shifts in the data, and finally the issue of flagging sudden changes between two normal modes of data. We will discuss each of these issues and corresponding solutions with illustrations in the following sections.

Once the model is trained, the anomaly pipeline begins by passing the data window through the trained autoencoder. Then the L2 reconstruction loss is calculated at each of the points in the window. At this point, we add our post-processing steps to improve the model. We take a weighted sum instead of a simple sum of the L2 losses at each data point, and we normalize the score according to the standard deviation of the reconstruction errors in the window. Finally, a threshold is selected to predict anomaly labels from the anomaly scores.

4.4.1 Why a Weighted Sum?

A simple sum of anomaly scores will result in anomalies of length 1 becoming predictions of the length of the window size. The reason for this is that a length 1 anomalous spike appears in many shifted overlapping windows according to the window size as shown in figure 4·1. This behavior, predicting anomalies much wider than the true anomalies, is undesirable because it does not represent the true anomaly width. This is especially undesirable if two anomalies appear within one window length, and can not be distinguished from one another.

In order to address this dilemma, we take a weighted sum of the reconstruction errors, with much more significant weights given to indices at the leading edge of the sliding window.

4.4.2 Why Normalization?

Time series data often contain changes in trends and shifts in distribution. These changing and shifting distributions represent different modes of “usual” data, but unfortunately they still impact the anomaly scores, making good threshold selection across the entire data distribution impossible. For example, a dataset’s noise level or amplitude may trend higher over a region of data, and the anomaly scores will also trend higher since the reconstruction model cannot fit the high noise. This creates another dilemma : no threshold will perform well for both the high and low noise portions of data. A threshold will either fail to identify true anomalies in the high noise portion, or flag many false alarms in the low noise portion (see figure 5·1).

In order to address this dilemma we normalize the weighted sum of reconstruction errors by the standard deviation of reconstruction errors in the sliding window. If most points in a window have a low reconstruction error in a low noise portion of data, a few high noise points in this portion will still result in a high anomaly score because they have been divided by a small standard deviation. On the other hand, higher reconstruction errors in a higher noise portion of data will be normalized such that an anomaly will not be flagged unless a much higher reconstruction error relative to the surrounding data is observed.

4.4.3 Delay Improves Results

Continuing upon the idea of changing distributions of data, sometimes a data distribution will shift suddenly between two modes of ”usual” instead of trending gradually. When the sliding window first observes a changed distribution at its leading edge, it cannot possibly predict whether this is an anomalous spike or simply a transition between two modes of usual data without observing future data to see if the new

Table 4.1: Experimental Results, F1 scores

Model	**NASA**		**YAHOO**			
	SMAP	MSL	A1	A2	A3	A4
DenseAE w/ Post	0.623	**0.797**	**0.916**	**0.995**	**0.976**	**0.912**
DenseAE w/o Post	**0.655**	0.608	0.496	0.283	0.097	0.041
TADGAN [Geiger et al., 2020]	0.623	0.704	0.8	0.867	0.685	0.6
LSTM [Hundman et al., 2018]	0.46	0.69	0.744	0.98	0.772	0.645
ARIMA [Din, 2015]	0.492	0.42	0.726	0.836	0.815	0.703
Deep AR [Salinas et al., 2020]	0.583	0.453	0.532	0.929	0.467	0.454
HTM [Ahmad et al., 2017]	0.412	0.557	0.588	0.662	0.325	0.287
MADGAN [Li et al., 2019]	0.111	0.128	0.37	0.439	0.589	0.464
MS Azure [Ren et al., 2019]	0.218	0.118	0.352	0.612	0.257	0.204

Table 4.2: Experimental Results, F1 scores

Model	**NAB**						
	Art	AdEx	AWS	Traf	Tweets	μ	σ
DenseAE w/ Post	**0.8**	0.762	0.762	**0.8**	**0.71**	**0.82**	0.12
DenseAE w/o Post	0.667	0.533	**0.764**	0.333	0.742	0.47	0.25
TADGAN [Geiger et al., 2020]	**0.8**	**0.8**	0.644	0.486	0.609	0.69	0.11
LSTM [Hundman et al., 2018]	0.375	0.538	0.474	0.634	0.543	0.62	0.17
ARIMA [Din, 2015]	0.353	0.583	0.518	0.571	0.567	0.6	0.16
Deep AR [Salinas et al., 2020]	0.545	0.615	0.39	0.6	0.542	0.56	0.14
HTM [Ahmad et al., 2017]	0.455	0.519	0.571	0.474	0.526	0.49	0.11
MADGAN [Li et al., 2019]	0.324	0.297	0.273	0.412	0.444	0.35	0.14
MS Azure [Ren et al., 2019]	0.125	0.066	0.173	0.166	0.118	0.22	0.15

mode persists (if it is a change in distribution) or disappears (if it is an anomalous spike). Therefore, we must predict anomalies at some delay in order to avoid flagging false alarms at these transition points.

In order to distinguish these transition points with a low anomaly score, the sliding window is divided at a delay index into "past" points and "future" points. We normalize based on both "past" and "future" standard deviations. In particular, we normalize by the power mean of the "past" and "future" standard deviations because the power mean biases towards the maximum value. We found that the degree 4 power mean works well. Finally, our weighted sum weights the "current" point and those points closest to the "current" highest in order to center the predicted anomaly score at the index between "past" and "future". The "future" points are not actually in the future because these have already been observed, and thus this method trades a small prediction delay for a reduced false alarm rate.

4.4.4 Putting it all Together

There are a variety of weighting function and normalization approaches that yield improved results, but for benchmark reproducibility we will now describe the settings of our weighting and normalization schemes. We use a window size of 100 time steps and a delay of 15 time steps dividing 84 "past" time steps, 1 "current", and 15 "future" time steps. We normalize the reconstruction scores by dividing by the degree 4 power mean of the standard deviations of the "future" and "past" reconstruction errors $\sqrt[4]{\sigma_{past}^4 + \sigma_{future}^4}$. We weight the reconstruction errors by multiplying each reconstruction value by $(size_{window} - |index - delay|)^2$ creating a spike at the "current" time step and a quadratic decay to either side. Because the "current" time step is especially important, we further amplified this value.

We set the autoencoder's latent dimension to 10 and trained with the mean squared error loss function. Experiments were run using PyTorch [Paszke et al., 2019]. We use the simple deep multilayer perceptron (MLP) architecture with layer sizes 100-80-40-20-10-20-40-80-100.

4.5 Experiments

4.5.1 Datasets

Three collections of datasets stand out in the literature for benchmarking time series anomaly detection algorithms – the Yahoo collection, the Numenta collection, and the NASA collection. We have selected to benchmark using the same subset of these datasets as TADGAN's recent state-of-the-art work. We benchmark using the entire Yahoo collection, consisting of four univariate datasets, A1 sourced from real Yahoo service metrics, and A2-A4 synthetically created. We also benchmark on the entire NASA collection, consisting of multivariate telemetry measurements from systems aboard the Mars Science Laboratory (MSL) rover and Soil Moisture Active Passive (SMAP) satellite. From the Numenta Anomaly Benchmark (NAB) collection we test on five datasets – advertisement clicking rate data, Amazon Web Services (AWS) server metrics, Twitter volume data, vehicle traffic data, and a synthetically created artificial dataset. Benchmarking on these datasets together provides a method for comparing performance between different anomaly detection algorithms across a variety of domains. Although each of these datasets is labeled, the training must remain unsupervised in order to ensure that good benchmark performance implicates good performance on other unlabeled datasets.

4.5.2 Measuring Performance

We measure the performance using the F1 metric. $F1 = \frac{TP}{TP + \frac{1}{2} * (FP + FN)}$. A good monitoring system will correctly identify anomalies without flooding the user with false alarms. The F1 metric rewards true positive prediction (TP) and penalizes both false negatives (FN) and false positives (FP). Unfortunately, time-series anomaly detection algorithms are plagued by a lack of standardization for scoring models. In an effort to remedy this situation, Numenta released their own scoring metric [Lavin and Ahmad, 2015] along with their collection of datasets, but perhaps because people desire to use one metric easily across multiple datasets, including the Yahoo and NASA datasets, Numenta's scoring function has not been widely adopted. Area under the curve (AUC) struggles to differentiate between any models with decent performance. Therefore, most literature uses the F1 metric, a well established metric used across multiple domains, but many have used time series specific variations, which can either drastically boost or plunge scores. Variations agree that partially overlapping prediction and anomaly windows should still count as true positives, but variations differ on how to count anomalies. One group measures each anomaly window as a singular anomaly, and another measures an anomaly window according to the window's width. This difference can have a drastic effect. If a correctly predicted anomaly window is length 100, then the first of these variations will record one true positive and the second will record 100, giving full credit even in the case of partial overlap and significantly boosting scores. We think it is more sensible to count each anomaly window as a singular true positive or false negative, and each contiguous false positive also as a singular value. Each of these is a singular event, and should be counted as such. Therefore, we use the F1-metric time series variation used by [Geiger et al., 2020]: (1) If a known anomalous window overlaps any predicted windows, a single true positive is recorded. (2) If a known anomalous window does not overlap any predicted windows, a single false negative is recorded. (3) If a predicted window overlaps with no anomalous windows, a single false positive is recorded. We selected optimal thresholds in order to measure the performance of our anomaly scoring function.

4.5.3 Results

Table 4.2 shows the experimental results of our model both with and without post-processing alongside competitor results. With the post-processing included, our model achieves the best results on 8 of 11 datasets, and near the best results for the other 3.

The difference in F1 scores with post-processing is much more noticeable in some datasets than others. In particular, the Yahoo datasets, which have many examples of multiple anomalies near to each other, benefit from the weighted sum in order to distinguish true positives. Yahoo's A4 dataset also features many sudden distribution shifts, benefitting from the normalization step.

The NASA dataset performance is also state-of-the-art. SMAP features anomalies spaced in time from each other and no major distribution shifts, explaining why results are similar with and without post-processing. The NAB dataset also sees noticeable improvements with post-processing, and noticeable improvements against many competitors even without post-processing for the AWS dataset.

The results shown demonstrate that a simple autoencoder can achieve state-of-the-art results with simple post-processing. For most applications, complex and large model architectures boost performance, but for time series anomaly detection, simple is better.

4.6 Conclusion

In conclusion, sometimes simple does work better. Deep learning is a powerful tool, able to extract meaningful features from complex data, but for time series anomaly detection a simpler model is important to avoid overfitting. A deep autoencoder with a small latent dimension offers the feature flexibility and automation of deep learning along with the power of simplicity. In order to improve results, it is better to focus on optimized post-processing rather than using a fancy new architecture that increases the chance of overfitting. More work can be done in optimizing how deep learning can be used for anomaly detection. In particular, our method and many others still struggle to determine the best resolution and window size to scan for anomalies. Deep learning already assists with many of the burdens of engineering time series, but the more steps that can be automated the better.

Chapter 5

Using Block Sparse Weights with Compressed Implicit Neural Representation Networks

5.1 Abstract

We use structured sparsity to improve on an existing approach for compressing images. A small neural network is trained to encode an image's RGB values according to pixel location. This small neural network is then used as the compressed representation of the image. The approach outlined in this work demonstrates that block-sparse weight matrices offer more expressive power per weight than dense fully-connected weight matrices, resulting in better compression performance. While this approach does not yet outperform the traditional state-of-the-art algorithms for image compression, it further develops a radically novel approach that offers flexibility and great potential for rapid advancement.

5.2 Introduction

Compression of files is an important mechanism for reducing bandwidth load over computer networks. Image, video, and audio mediums often allow for lossy compression as a pixel perfect representation is rarely necessary to view an image. In the image domain, recent work has developed deep learning techniques to compete with more traditional image codecs such as JPEG.

Most deep compression techniques use an autoencoder architecture to encode an image into a latent space, and then the sender transmits a latent encoding instead of the image. These approaches, however, require both the sender and receiver to have the weights of the autoencoder, weights that use orders-of-magnitude more memory than the images themselves.

One novel approach, however, takes a completely different approach to the problem of neural compression. This approach uses compressed implicit neural representations networks (COINs) [Dupont et al., 2021] to offers several benefits. COIN represents the image as a learned function mapping pixel coordinates to RGB values, and the weight values of this learned function are then the compressed representation of the image. In this approach, there is no larger network that must be shared beforehand between the two devices, signal is preserved competitively with existing methods at low bit-rates, and images can be decoded to any desired resolution.

COIN offers a new unexplored paradigm where many further improvements can be made, increasing performance to unknown new measures. In this work, we improve on this novel approach by introducing structured sparsity into the model design, slightly boosting PSNR values over the original work.

5.3 Related Work

Currently image compression is dominated by traditional approaches which make use of color transformations, wavelets, Fourier transformations, and more. The workings of these approaches is outside the scope of this work, which deals with minimizing information in deep learning, but offer an important baseline of comparison.

Over the past few years, deep learning approaches [Lee et al., 2019, Minnen et al., 2018, Ballé et al., 2018] using an autoencoder to compress an image into a latent dimension have proven to be effective, and these offer one potential avenue for bringing deep learning to a state-of-the-art performance level in image compression.

Finally, beyond the traditional algorithms and the deep latent approach, neural representations [Mildenhall et al., 2020,Sitzmann et al., 2020,Dupont et al., 2021] are a possible contender for state-of-the-art image compression.

5.4 Background and Methods

Last year, SIREN networks were introduced as a method for encoding images as a neural network. The original SIREN paper [Sitzmann et al., 2020] shows that periodic activation functions dramatically improve the expressiveness of neural representation networks, enabling them to represent sharp edges and gradients in the image. A sinusoidal periodic activation also offers the benefit of being easily differentiable into another SIREN network since the derivative of a sinusoid activation function is also a sinusoid. The initial SIREN paper does not probe neural representations as an option for compression, but demonstrates the effectiveness of SIRENs for audio, image, and video encoding, for image inpainting, and for differentiable equation solving.

In March of 2021 COIN demonstrated SIREN networks capable of compressing images competitively with the JPEG codec at low bit-rates. COIN representations outperformed JPEG at the lowest bit-rates of compression. Despite COIN representing a possible paradigm shift in compression, COIN still underperforms the best compression algorithms even at low bit-rates. Therefore, demonstrating techniques for improvement on COIN is important for the development of this new technology.

Using COIN's paradigm of neural representations for image compression, image compression becomes closely related to the problem of model compression (such as techniques discussed in chapter 3 of this dissertation), but not all model compression techniques are suited to compression of SIREN networks. The NodeDrop technique of chapter three cannot compress a SIREN because of its sinusoidal activation functions.

Edge pruning techniques can compress weight matrices at the cost of expressive power, but weight matrices stored in sparse format include storing indices of values, and so are not as efficient as dense matrices per weight stored.

Random block sparsity has successfully been used to accelerate the training of neural networks. Random block sparsity replaces a dense fully connected layer with a blocked weight matrix, In this work we will use predetermined block sparsity to improve COIN's expressive performance. Each weight matrix will be tiled diagonally by blocks of predetermined size. A BlockCOIN model architecture is determined by layer depth, block size, and layer size. Because many weights are predetermined to be 0, we can afford to make much wider layers than COIN, with many sparsely connected nodes.

5.5 Experiment

We tested BlockCOIN using the same benchmarks and baselines as COIN. Testing was performed on the Kodak image dataset against three deep autoencoder competitors, BMS, MBT, and CST. Performance was also compared to more traditional codecs JPEG, JPEG2000, BPG, and VTM. Experiments show that using block-sparse matrices yields a marginal performance improvement over the original COIN models, as shown in Figure 5·1. Figure 5·1 measures the fidelity of the compressed image using the peak-signal-to-noise-ratio (PSNR) at different bit rates in bits per pixel (bpp). This is a first step toward becoming more competitive with the other models. Training was performed at 32-bit precision, but testing was performed at 16-bit precision in order to improve compression.

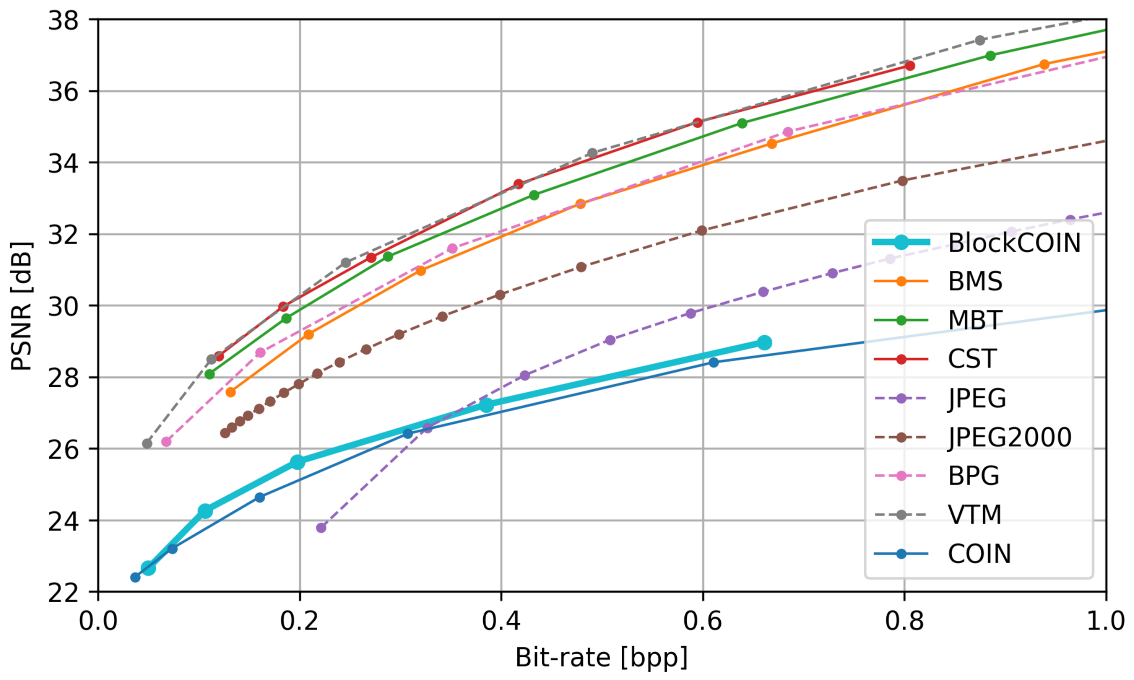

Figure 5·1: The block sparse version of COIN's rate-distortion curve marginally outperforms COIN's rate-distortion curve. Both curves are not yet competitive with the state-of-the-art deep learning approachs or popular codecs except for JPEG.

5.6 Outlook

This work builds directly upon the work of COIN, and yields only a slight performance gain. More work needs to be done before this chapter is published as a standalone paper. Despite this, the improvement in PSNR scores is noticeable enough to incorporate this technique along with future techniques. I think that a careful study of quantization could yield improved results. Also the suggestions from COIN's initial paper, using entropy coding and learning a distribution of weights, both could yield improvements. Neural representations for image compression offer benefits that other deep learning techniques do not, with a small model size that could be executed on even devices with only kilobytes of memory hardware. Low bit-rate performance looks promising, and one-to-many applications such as Netflix or Youtube could benefit from improved compression even at the cost of the long encoding time associated with training a neural network. Neither COIN nor BlockCOIN are application ready, but these novel approaches are rife for further exploration.

Chapter 6

Conclusions

6.1 Conclusion

In this dissertation, I have discussed four projects, each related to minimalism in deep learning. After introducing the dissertation topic in the introduction, I addressed using deep learning with minimal measurements in side-channel analysis, a general purpose technique to reduce the size of neural networks while maintaining performance, how minimalism in deep learning can help avoid overfitting in anomaly detection, and an avenue of research for bringing minimalism in deep learning to the new application of image compression. Each of the aspects of minimalism in deep learning discussed in this dissertation were advanced by my work, and can be advanced further by future work.

In chapter two, I used deep learning to classify types of repeated blocks in program execution using only side-channel signals of power and EM radiation. This progressed beyond techniques that classified entire programs, but unlocking reliable instruction-level tracking remains an exciting goal in side-channel research. With reliable instruction-level tracking one could mimic program execution on an external device by measuring only a processor's side-channels.

It would also be useful to further explore the architecture space to see if different architectures can yield performance with even less context. In our architecture exploration we found the 1D CNN layers critical to success, but it is possible that removing the GRU layer could still yield optimal performance. Further analysis using different side-channels such as temperature and fan noise is another direction of future work.

Chapter three explored a novel technique for reducing the size of neural networks, with provable benefits. Model compression through pruning, quantization, and other novel techniques will always aim towards better achievement. Combinations of these techniques especially deserve further exploration. Combining NodeDrop with entropy-coding on the weights also deserves consideration.

Most of all, expanding the applications of NodeDrop is important. NodeDrop should be used as a step before neural architecture searches in order to constrain the search space. Also, expanding beyond convolutions and fully-connected layers to different architectures and settings, such as the recurrent architecture or transformer, deserves further work. A clever use of skip layers and regularization could even make possible the elimination of entire layers, tuning the depth of the model instead of the width. Similar conditions and regularizers should be sought for these architectures and settings.

Chapter four achieved strong performance on unsupervised time-series anomaly detection benchmarks. For time-series anomaly detection, a simple technique based on the minimalism of an autoencoder can perform better than powerful techniques such as GAN's and VAE's. Sometimes a problem should be solved with a simple model, and as a research community it is important to try the simple before the complex.

Chapter five explored a very new application of deep learning. Neural representations are exciting for many reasons, not just compression, but the use of neural representations for image compression has at least given us an understanding of just how efficiently deep learning models store information in their weights. Can we make neural representations more efficient? And if we can, will the techniques used prove beneficial to make other deep learning models also more efficient? These questions are important to answer even if deep learning never becomes the standard for image compression.

Finally, there are also many more aspects of minimalism in deep learning not discussed in this dissertation that must continue to be explored. Unsupervised learning continues to advance rapidly in applications beyond anomaly detection. This research minimizes the infamous labelling demands of effective models. Research in few-shot learning, transfer learning, and data augmentation help models to train with less data.

Deep learning is limited by its demands. The less demanding deep learning becomes, the more ubiquitous it will also become.

References

Agrawal, D., Archambeault, B., Rao, J., and Rohatgi, P. (2003). The em side – channel(s): Attacks and assessment methodologies.

Ahmad, S., Lavin, A., Purdy, S., and Agha, Z. (2017). Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147. Online Real-Time Learning Strategies for Data Streams.

Alvarez, J. M. and Salzmann, M. (2016). Learning the number of neurons in deep networks. *Neural Information Processing Systems (NIPS).*

Anwar, S., Hwang, K., and Sung, W. (2017). Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC).*

Ballé, J., Minnen, D., Singh, S., Hwang, S. J., and Johnston, N. (2018). Variational image compression with a scale hyperprior. In *International Conference on Learning Representations.*

Bank, D., Koenigstein, N., and Giryes, R. (2020). Autoencoders. *CoRR*, abs/2003.05991.

Callan, R., Zajic, A., and Prvulovic, M. (2014). A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.*

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3).

Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111. Association for Computational Linguistics.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Clevert, D., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. *Neural Information Processing Systems (NIPS).*

Din, M. (2015). Arima by box jenkins methodology for estimation and forecasting models in higher education.

Dupont, E., Goli'nski, A., Alizadeh, M., Teh, Y., and Doucet, A. (2021). Coin: Compression with implicit neural representations. *ArXiv*, abs/2103.03123.

Gao Huang, Zhuang Liu, L. v. d. M. K. Q. W. (2017). Densely connected convolutional networks. *Computer Vision and Pattern Recognition (CVPR).*

Geiger, A., Liu, D., Alnegheimish, S., Cuesta-Infante, A., and Veeramachaneni, K. (2020). Tadgan: Time series anomaly detection using generative adversarial networks. In *2020 IEEE International Conference on Big Data (IEEE BigData)*. IEEE.

Genkin, D., Shamir, A., and Tromer, E. (2013). Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. (2014). Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115.*

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. *Internatinal Conference on Machine Learning (ICML).*

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR).*

Han, S., Pool, J., Tran, J., and Dally, W. (2015). Learning both weights and connections for efficient neural network. *Neural Information Processing Systems (NIPS).*

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. *Neural Information Processing Systems (NIPS).*

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. *Computer Vision and Pattern Recognition (CVPR).*

Hospodar, G., Gierlichs, B., Mulder, E. D., Verbauwhede, I., and Vandewalle, J. (2011). Machine learning in side-channel analysis: a first study. *J. Cryptographic Engineering*, 1(4):293–302.

Hubara, M. C. I., , Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks. *Neural Information Processing Systems (NIPS).*

Hundman, K., Constantinou, V., Laporte, C., Colwell, I., and Soderstrom, T. (2018). Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '18, page 387–395, New York, NY, USA. Association for Computing Machinery.

Hutter, M. and Schmidt, J.-M. (2014). The temperature side channel and heating fault attacks. *IACR Cryptology ePrint Archive*, 2014:190.

Ioffe, S. and Szegedy, C. (2015a). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org.

Ioffe, S. and Szegedy, C. (2015b). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning (ICML).*

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. (2015). Compression of deep convolutional neural networks for fast and low power mobile applications. *International Conference on Learning Representations (ICLR).*

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Kingma, D. P. and Welling, M. (2019). An introduction to variational autoencoders. *CoRR*, abs/1906.02691.

Kocher, P. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Koblitz, N., editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In Wiener, M., editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kodali, N., Abernethy, J. D., Hays, J., and Kira, Z. (2017). How to train your DRAGAN. *CoRR*, abs/1705.07215.

Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar-10 (canadian institute for advanced research).

Kukacka, J., Golkov, V., and Cremers, D. (2017). Regularization for deep learning: A taxonomy. *CoRR*, abs/1710.10686.

Lavin, A. and Ahmad, S. (2015). Evaluating real-time anomaly detection algorithms - the numenta anomaly benchmark. *CoRR*, abs/1510.03336.

Lebedev, V. and Lempitsky, V. (2016). Fast convnets using group-wise brain damage. *Computer Vision and Pattern Recognition (CVPR).*

LeCun, Y. and Cortes, C. (1998). MNIST handwritten digit database.

LeCun, Y., Denker, J. S., Solla, S., Howard, R. E., and Jackel, L. D. (1990). Optimal brain damage. *Neural Information Processing Systems (NIPS).*

Lee, J., Cho, S., and Beack, S. (2019). Context-adaptive entropy model for end-to-end optimized image compression. In *ICLR*.

Li, D., Chen, D., Shi, L., Jin, B., Goh, J., and Ng, S. (2019). MAD-GAN: multivariate anomaly detection for time series data with generative adversarial networks. *CoRR*, abs/1901.04997.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2017). Pruning filters for efficient convnets. *Internation Conference on Learning Representations (ICLR).*

Liu, Y., Wei, L., Zhou, Z., Zhang, K., Xu, W., and Xu, Q. (2016). On code execution tracking via power side-channel. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1019–1031, New York, NY, USA. ACM.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. (2017). Learning efficient convolutional networks through network slimming. *International Conference on Computer Vision (ICCV).*

Maghrebi, H., Portigliatti, T., and Prouff, E. (2016). Breaking cryptographic implementations using deep learning techniques. Cryptology ePrint Archive, Report 2016/921.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. (2020). Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*.

Minnen, D. C., Ballé, J., and Toderici, G. (2018). Joint autoregressive and hierarchical priors for learned image compression. In *NeurIPS*.

Molchanov, D., Ashukha, A., and Vetrov, D. P. (2017). Variational dropout sparsifies deep neural networks. *International Conference on Machine Learning (ICML).*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Probst, M. and Rothlauf, F. (2020). Harmless overfitting: Using denoising autoencoders in estimation of distribution algorithms. *Journal of Machine Learning Research*, 21(78):1–31.

Prouff, E., Strullu, R., Benadjila, R., Cagli, E., and Canovas, C. (2018). Study of deep learning techniques for side-channel analysis and introduction to ascad database. *IACR Cryptology ePrint Archive*, 2018:53.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. *European Conference on Computer Vision (ECCV).*

Reed, R. (1993). Pruning algorithms-a survey. *IEEE transactions on Neural Networks.*

Ren, H., Xu, B., Wang, Y., Yi, C., Huang, C., Kou, X., Xing, T., Yang, M., Tong, J., and Zhang, Q. (2019). Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '19, page 3009–3017, New York, NY, USA. Association for Computing Machinery.

Riley, R., Graham, J., Fuller, R., Baldwin, R., and Fisher, A. (2018). Generalization of algorithm recognition in rf side channels between devices. In *Cyber Sensing 2018*, volume 10630, page 106300C. International Society for Optics and Photonics.

Salinas, D., Flunkert, V., Gasthaus, J., and Januschowski, T. (2020). Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191.

Schlegl, T., Seeböck, P., Waldstein, S., Langs, G., and Schmidt-Erfurth, U. (2019). f-anogan: Fast unsupervised anomaly detection with generative adversarial networks. *Medical Image Analysis*, 54.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR).*

Sitzmann, V., Martel, J. N. P., Bergman, A. W., Lindell, D. B., and Wetzstein, G. (2020). Implicit neural representations with periodic activation functions. *CoRR*, abs/2006.09661.

Srinivas, S. and Babu, R. V. (2015). Data-free parameter pruning for deep neural networks. *British Machine Vision Conference (BMVC).*

Srivastava, A., Valkov, L., Russell, C., Gutmann, M. U., and Sutton, C. (2017). Veegan: Reducing mode collapse in gans using implicit variational learning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Su, Y., Zhao, Y., Niu, C., Liu, R., Sun, W., and Pei, D. (2019). Robust anomaly detection for multivariate time series through stochastic recurrent neural network. pages 2828–2837.

Sun, Y., Wang, X., and Tang, X. (2015). Sparsifying neural network connections for face recognition. *Computer Vision and Pattern Recognition (CVPR).*

Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. *Workshop on Deep Learning and Unsupervised Feature Learning (NIPS).*

Wang, X., Zhou, Q., Harer, J., Brown, G., Qiu, S., Dou, Z., Wang, J., Hinton, A., Gonzalez, C. A., and Chin, P. (2018). Deep learning-based classification and anomaly detection of side-channel signals. In *Cyber Sensing 2018*, volume 10630, page 1063006. International Society for Optics and Photonics.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. *Neural Information Processing Systems (NIPS).*

Yilmaz, B. B., Callan, R., Prvulovic, M., and Zajic, A. (2018). Capacity of the em covert/side-channel created by the execution of instructions in a processor. *IEEE Transactions on Information Forensics and Security*, 13(3):605–620.

Zajic, A. and Prvulovic, M. (2014). Experimental demonstration of electromagnetic information leakage from modern processor-memory systems. *IEEE Transactions on Electromagnetic Compatibility*, 56(4):885–893.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017). Understanding deep learning requires rethinking generalization.

Zhao, H., Wang, Y., Duan, J., Huang, C., Cao, D., Tong, Y., Xu, B., Bai, J., Tong, J., and Zhang, Q. (2020). Multivariate time-series anomaly detection via graph attention network. *CoRR*, abs/2009.02040.

Zhou, H., Alvarez, J. M., , and Porikli, F. (2016). Less is more: Towards compact cnns. *European Conference on Computer Vision (ECCV).*

CURRICULUM VITAE

[redacted]

[redacted]