5-26-2018

Theory, Operation, and Application of Neural Networks

John Andersen
Portland State University

Portland State University

June 2018

Undergraduate honors thesis submitted in fulfillment of requirements
for the degree of Bachelor of Science
in
University Honors College
&
Computer Engineering

Theory, Operation, and Application of Neural Networks

Author: John Andersen

Advisor: Dr. Douglas Hall

Abstract

This paper examines the history and current state of machine learning. It examines neural networks, theory behind neural networks, how they are implemented, and how they are used. The systems and networks examined have up to three modes of learning. Theory behind machine learning is broken up into three approaches; rule-based, Bayesian, and neural networks. Operation of machine learning algorithms has been enabled by several prevalent libraries in the open source community, as well as various hardware technologies. Due to this surge in resources application, developers have been able to apply machine learning in novel ways. An application of machine learning to evaluate the security practices of open source software was undertaken as the culmination of this thesis.

Table Of Contents

Introduction

Machine Learning (ML) has an important role to play in the advancement of the computer science and engineering fields. Classification of handwritten digits using the MNIST dataset is a frequently used example that shows the power of machine learning. Without the aid of ML, programmers can only reach approximately 50% accuracy on this problem. However, when neural networks are applied, accuracy above 90% can easily be achieved [1]. To solve a problem using machine learning we create a model. Model's process input data, known as features, and produce a given result. The type of result is dependent upon the mode of learning. One mode allows us to make predictions, another gives us insight into the dataset. Within the model is the learning system, which could be rule-based, Bayesian, or a neural network.

The learning part of machine learning comes from the correction made to the mathematics occurring in the model. For instance, if data depicting a cat is input and the model outputs dog, then the model is adjusted, and hence learning has occured. Adjustment to the model is modification of the mathematical operations done on the inputs. With enough inputs and outputs as examples, the model will eventually settle on mathematical operations that work for most inputs. The accuracy of a model is defined by how often it correctly predicts the output for a set of inputs.

Theory

Modes of Learning

The process of learning can be done in one or more of three modes; supervised, unsupervised, and reinforcement. Each mode is applicable to different situations and algorithms. Supervised mode learns what the correct answer is based on events in the past. Unsupervised mode finds correlations between inputs. Reinforcement mode weighs risks and rewards in order to make the best decision.

There are many popular publicly available datasets for use with machine learning. One of them is on the Portland Real Estate market. It contains data on home prices, number of beds, baths, square footage, and distance from downtown. For a supervised application of machine learning, the price would be what we want our model to predict given the other data about a home. After training the model, it could be given the number of rooms, etc., and be able to predict the price of the house. Supervised learning is used in situations where there is a known correct result for a set of input data.

An example of unsupervised learning is the Netflix problem. The Netflix problem is to determine what a user wants to watch given their previous viewing history. In this case, Netflix knows what a user has and hasn't enjoyed in the past, as well as a set of information about those films. Not all films are of the same genre however, and a user's interest in a film in one genre may translate to an interest in a similar film of another genre. Unsupervised learning takes films and groups them according to the data it has, thus determining that respective groups of films have a certain *je ne sais quoi,* which make them similar to each other. Neither Netflix, the user, or the model could explain what it is, however, aspects about them revealed in the data show that they are in some way related.

Reinforcement learning (RL) is learning how to achieve a goal. Actions which make no progress, or negative progress towards a goal teach a model not to repeat those actions. Actions which help the model make progress towards the goal are repeated, or variations of them are explored further. Researchers have used RL to train models to play video games. One such project involved the game Space Invaders [16]. To learn and play the game, the model selects an action, such as move right, left, or shoot, and observes the results. Aliens killing the player is bad, the player killing aliens is good. As the model selects actions, it tracks the feedback it receives. Should it continuously perform actions that get the player killed, it will learn from this and change its course of actions.

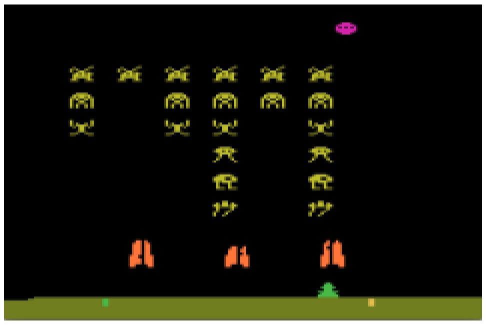


Figure 1: Space Invaders played by RL [16]

Rule Based Systems

When we think about how to solve a problem, or how to determine what something is, we usually create a set of rules to guide us to the right answer. While driving, if there are pedestrians, we stop. If a bump is coming up, we slow down. If the light is green, we start moving. Like driving, most problems can be broken into these "if condition, do something" rules. This is the idea behind rule-based machine learning. We know what rules there are, but just as with driving, context matters. Rule-based machine learning uses the rules we give it and learns how important each rule is. This way it would know that even if the light is green, we still need to stop for pedestrians. Rule-based systems work primarily based on reinforcement learning. With driving, we would give feedback that whichever route gets us to the destination quicker, is better. In a reinforcement system, we could do this by giving the system negative "points" so that it tries to pick rules which would decrease the amount of negative points it gets. We would give it many negative points if it ran a stop sign, and therefore it would know that the rules which tell it not to run stop signs are more important than the rules which make its route shorter. Over time this teaches the rule-based system how important each rule is.

Learning Classifier Systems (LCS) are rule-based machine learning systems combined with genetic algorithms. Rules defined in an LCS produce outputs which are the inputs to the genetic algorithm [13]. Genetic algorithms simulate population growth. Inputs become genes which become either more or less prevalent in the system as a whole as the simulation progresses from generation to generation. Genes are representative of an answer to the classification problem presented to the LCS. Reinforcement learning in a genetic algorithm is done by applying a cost function. A cost function is a calculation which decides which genes are closest to the correct answer. To create the next generation in the GA's population, genes are mutated slightly, meaning their answer to the problem is changed slightly. The number of permutations of a gene within the next generation is relative to the cost function. If the gene is close to the correct answer, then it will produce more offspring for the next generation. Each of them slightly mutated so that the next might be even closer to correct.

Bayesian Networks

The Bayesian belief network is a way that statistics and probability have been applied to machine learning. Connections between events are created in a directed acyclic graph [14]. The probabilities of events are connected to each other such that an event’s probability is determined by the probability of its previous events. The first layer of events have probability functions determined only by their inputs. Subsequent events interior to the network have probability functions determined by the joint probability of the system’s best knowledge of previous events [15].

$$P(A|B) \; = \; \frac{P(B|A) * P(A)}{P(B)}$$

Bayes Rule

In the above formula, *P(B)* is the probability of the previous event occuring. *P(A)* is the probability that our network will guess correctly without being given any events as examples. *P(A|B)* is therefore the probability that the model predicts event occurrences accurately. As more events are added to the belief network the model becomes more accurate because *P(B)* gets closer to the theoretical probability [20].

Neural Networks

Both Bayesian belief networks and rule-based learning require that we have knowledge of data we are working with. Neural networks however, are versatile enough to give us meaningful results solely based on input data. Unlike rule-based approaches they learn what pieces of the input data and features are important through training. This effectively is the neural network creating its own rules.

Neural networks are meant to mimic the human brain. From what we know about brains, there are neurons and synapses. When neurons fire, synapses carry their signal to connected neurons. Upon receiving a signal, the connected neuron must decide if it will fire or not. In machine learning, artificial neural networks (ANNs) operate on the same premise. In an ANN, neurons must be trained on when to fire and what value to propagate upon firing to produce the desired output from the network [3]. The concepts of perceptrons, sigmoid neurons, layers, feedforward networks, cost functions, and backpropagation are key to understanding ANNs.

The most basic neural network is the perceptron network. Perceptrons are the most basic representation of a neuron, having at least one input and producing a single output. Internally, each has a weight and bias, which are multiplied by the input to produce floating point outputs.They are then interpreted as either a one or zero. This makes them function similarly to a logic gate, which is not applicable to real world applications. Sigmoid neurons are a variation on a perceptron, which allow numbers between zero and one. Their range of output can be thought of as a degree of certainty.

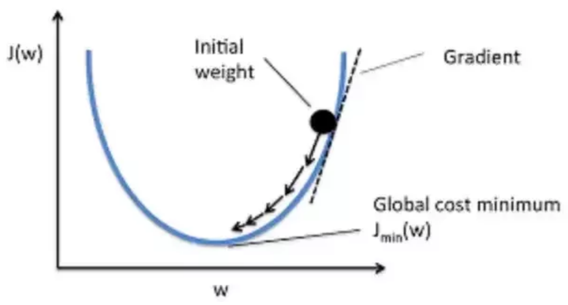


Figure 2: Stochastic Gradient Descent

Various algorithms, referred to as cost functions, are used to learn the appropriate weight and bias for each neuron, so that for various inputs, the output is as close to correct as possible. Stochastic gradient descent is one of the most commonly used cost functions. Each time data is fed through the network during supervised learning, weights are adjusted and derivatives are taken to determine how far off the last adjustment was from producing the correct output. For instance, a prediction of 0.9 when the correct output is 0.1 is far from correct. With figure 2 for reference, 0.1 would be near the bottom where the slope is rather level. 0.9 is in the exponentially increasing portion. Calculation of the cost function would indicate a large change is needed to the weight due to the steep slope of the line between the two. Neural networks learn by changing weights recomputed using the cost function.

As one might imagine, calculation of the cost function is a costly computational process. Each neuron's weight and bias must be recalculated for each pass through the network. A pass through the network is the process of feeding inputs to the network and carrying the values of one layer to the next [1]. This is also known as a feedforward neural network. Feedforward networks are networks which feed their outputs forward between layers, rather than back into themselves.

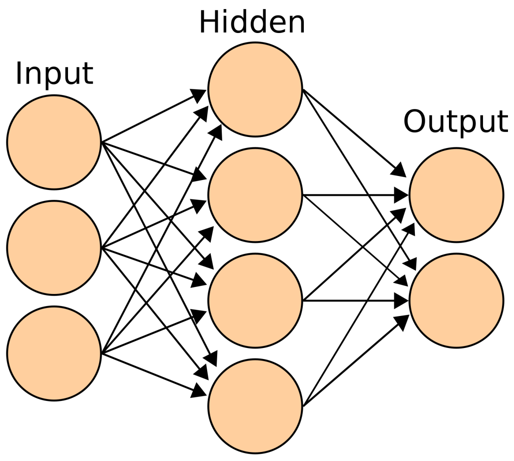


Figure 3: Artificial Neural Network

Layers in a neural network are a group of neurons. In the first layer, neurons are given inputs from external data. Subsequent layers are connected together so that outputs of one layer become inputs of the next layer. The output layer contains the result of the network. Equal number of output neurons and classification options allow for a one-to-one encoding. This is where an output neuron's result is a percentage confidence in input data belonging to its classification. Layers internal to the network (not the input and output layers), are known as hidden layers. Neural networks with more than one hidden layer are sometimes referred to as Deep Neural Networks (DNNs).

Convolutional Neural Networks

Computer vision (CV) tasks such as image classification involve feeding each pixel's data into a network. Even small images contain thousands of pixels, forcing networks built for CV to contain large numbers of neurons. CV tasks commonly require identification of features within an image. Convolutions are a way to identify features within a neural network. These can be thought of as a neural network within a neural network. Inputs to a convolution are a subset of the inputs to the larger Convolutional Neural Network (CNNs) [10]. As seen in figure 4, the inputs to a convolution cycle through the inputs to the CNN allowing them to identify a feature within the input. This makes CNNs good at classification tasks, such as object recognition.

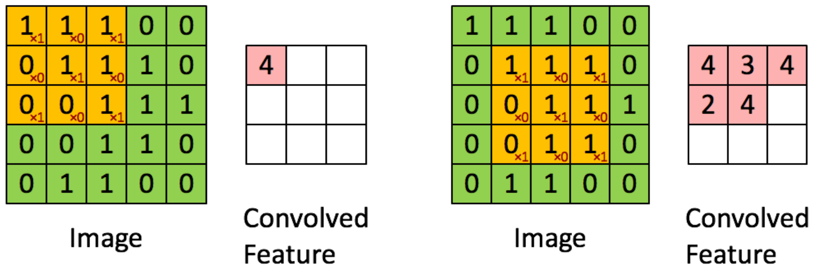


Figure 4: Feature extraction using convolutions

CNNs used for CV often employ multi-stage learning, which involves an explicitly defined layer and a learned layer. An explicitly defined layer might be an algorithm which detects edges in an image. The learned layer would make use of convolutions to identify features from associated edge and color image data. Two stage systems such as this are approximately 10% more effective than single stage systems [9].

Implementation Methods

Implementation details of machine learning algorithms and neural networks in both software and hardware are readily available thanks to an array of open source libraries, as well as research funded by influential corporations. Google, Microsoft, Facebook, Intel and others have created and collaborated on software libraries such as MXNet and TensorFlow, which have speed advancement of the field. Algorithm implementation is done in software, and hardware carries out the calculations. At the lowest level of the software stack, developers must choose which of four hardware options to perform ML calculations on; Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), Central Processing Units (CPUs), or Graphics Processing Units (GPUs). Custom designed logic in ASICs / FPGAs, extensions in CPUs for vector processing, and SMDI pipelines in GPUs allow for parallel computation, speeding the process of learning. Though each have advantages and disadvantages, GPUs have, in practice, become the most widely used and developed for hardware to quickly train machine learning models.

ASICs

Analog and digital designs have been used to implement ANNs. The challenge with digital is that sigmoid neurons have weights which are between zero and one [4]. Therefore, data has to be represented with multiple bits or extensive use of ADCs / DACs. Layout and connections on chip are another issue. There is only so much space, and connections can't always go over or under each other, making it nearly impossible to layout large networks. Calculations prove difficult because of the non-linearity of the cost function, as well as the need to do many different mathematical operations at each neuron to calculate the activation.

Analog designs can lead to more power and area efficient ASICs. However, the precision required to make analog work is higher and there are less automated design tools available to do this. Learning on an ASIC can be implemented in two ways. Chip-in-the-loop learning uses the ASIC to perform the calculations, but the software implements the algorithm. On-chip learning requires feeding data to the ASIC from an external source, such as a CPU, and it uses the learning algorithm which has been implemented on chip to run the algorithm and calculations.

FPGAs

FPGAs provide high speed calculations, near that of an ASIC. This allows hardware designers to choose what operations get done in parallel. Due to a predefined number of logic gates, FPGAs suffer from physical space constraints, putting a finite limit to the size of a neural network which can be implemented in a given FPGA. Machine learning is used to work on large datasets, thus a downside of FPGAs is their constraint on the amount of data to which parallel calculations can be done.

Sigmoid based neural networks use floating point numbers and arithmetic, which are resource intensive on computation and memory constrained FPGAs. Two-Bit Networks allow for replacement of full floating point numbers used for weights with two-bit numbers. Two-bit numbers can encode weights of -2, -1, 0, and 1, using 2's complement. For machine learning implementations, the lowest (-2), would not be used in order to provide a balanced scale. However, increased accuracy can be achieved by eliminating the use of zero, and using all possible 4 values that can be encoded using two-bits. To keep a balanced scale, the values of -2, -1, 1, 2 can be used instead. This is only possible because implementation is done at the hardware level. CPUs and GPUs do not allow for custom number interpretation . Researchers were able to save a gigabyte worth of memory by using this technique, while still maintaining reasonably good accuracy (~84%) [5].

While both ASICs and FPGAs allow for customization at a low level, FPGAs can be reprogrammed. Reprogramming of FPGAs can be done in sections. For large neural networks, processing can be completed in one section, move to another section, then back into the initial section, which has been reprogrammed. This could be the next network in a chain whos backpropagation calculations were performed in parallel, or another layer of the same network. The major downsides of FPGAs are long compile and debug times, a steep learning curve, and lack of software libraries which support them.

OpenCL and CUDA are the popular frameworks for programming GPUs. Minimal OpenCL support has been added by Xilinx and Altera for operations on FPGAs. OpenCL is currently the only major option for software integration with FPGAs. OpenCL can be used for calculations within higher level software libraries, such as Caffe and Torch. These libraries are the driving factors to increase FPGA adoption for machine learning applications [10].

Power savings over CPUs and GPUs are a strength of FPGAs. Power consumption is most important on battery powered embedded devices, as well as in data centers where hardware resources are widespread, and power consumption is a significant cost [6].

CPUs

All computers have a Central Processing Unit (CPU). CPUs can do many operations and have a wide variety of special instructions, which can increase the speed at which some mathematical operations are performed. While this is a benefit to the mathematically intensive machine learning systems described, CPUs are limited in their multitasking abilities. They were originally designed to perform sequential operations. While most CPUs now have multiple cores which allow for parallel processing, most operations are still performed sequentially within each core.

Some libraries, such as TensorFlow, have taken advantage of recent advances in CPU parallel processing abilities, known as Advanced Vector Extensions (AVX). This allows for matrix calculations to be done all at once. Neural network inputs can be fed through to multiple neurons, and backpropagation calculations can be done in parallel, by putting weights and inputs into matrices supplied to AVX.

The bottleneck for calculation intensive programs is always memory access. Feature data and correct outputs in the case of supervised learning must all be stored in memory. Computer architecture is such that the CPU controls access to main memory. Therefore, the CPU is tasked with loading data from external sources into memory, and preprocessing it before feeding it to machine learning algorithms. The CPU must also orchestrate data transfer and consistency between the memory of ASICs, FPGAs, or GPUs, and main memory. This can be a significant time overhead compared to calculation time within those devices. Therefore, some models perform faster running calculations on the CPU.

GPUs

Multitasking can be found in abundance in GPUs (Graphics Processing Units). These processors are designed to do thousands of operations at the same time in order to compute a value for every pixel on the screen. Using the principle of Single Instruction Multiple Data (SIMD), GPUs do the same operation on multiple pieces of data. In a neural network, the operation is multiplication, and the data is an input and a weight for each neuron.

Unlike FPGAs and ASICs, GPUs have well defined interfaces for communicating with the CPU and main memory. GPUs connect to the processor over the PCI bus. PCI provides high data transfer rates, and the ability to attach multiple GPUs to the same CPU. Machine learning involves large datasets that must be transferred over the internet to each machine using them for training. Data transfer is a significant time overhead. Having multiple GPUs per computer significantly increases their parallel processing capacity, thus eliminating the overhead of data transfer associated with a one GPU per machine configuration.

A popular dataset for image classification created in 2010 is called ImageNet. It contains over 14 million images. GitHub’s dataset of source code contains over 145 million unique Git commit messages, each associated with hundreds of lines of code. To train on the ever growing datasets available in a reasonable amount of time, machines must be networked together. Multi-machine, multi-GPU training increases the complexity of data transfer and consistency problems.

GeePS

Parameter servers are applications designed to orchestrate training across machines and individual GPUs. GeePS is a parameter server for the Caffe library. Training using GeePS can see up to 13x speedups compared to distributed CPU learning or single GPU Caffe learning. GeePS uses a parameter cache to store neural network parameters. It syncs parameters across hosts and manages data moving between CPU and GPU memory.

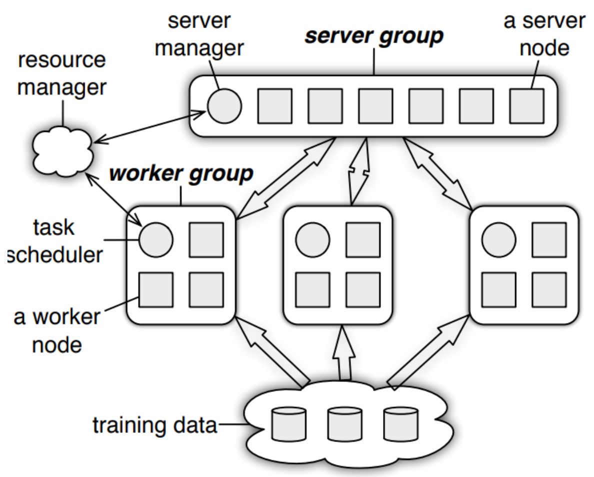


Figure 5: Parameter server architecture [17]

GeePS scales linearly with the addition of machines to its network. Training machines can run at different speeds, syncing parameters at an interval. This allows heterogeneous machines to train nearly as fast as they would in an non-networked setup, while still ensuring that machines don't compute using stale data that another machine hasn't finished computing. GeePS performs well on very large datasets due to its management of host and GPU memory. Rather than attempting to load entire datasets from main memory to GPU memory, loading is done iteratively. GPU memory only holds data being actively calculated on, all other data is stored in main memory, which speeds the parameter syncing process [12].

Caffe

Convolutional Architecture for Fast Feature Embedding (Caffe), is software developed by Berkeley AI Research and the Berkeley Vision and Learning Center with input from industry partners. It is a community maintained machine learning project which gives users the ability to construct models which can be run on CPUs or GPUs. Being written in C++ lets it use NVIDIA cuDNN and CUDA libraries which provide profoment access to NVIDIA GPUs.

A major goal of Caffe is to provide reproducible results. They do this by making it easy for researchers to share their machine learning models and trained networks. Bindings to Python and MATLAB enable quick prototyping of models. Caffe excels at full utilization of GPUs, thanks to its management of data moving between main memory and GPU memory. Caffe's architecture is defined by layers. Models are defined through specification of connections between layers. This allows users to easily construct CNNs and other large networks by linking the first, last, or interior layers of different models together [11]. Layers handle computation and have both CPU and GPU implementations. Having a choice of hardware backend allows researchers to discover which is most performant for their use case.

Caffe's biggest strength is its simple model definition format, as well as the ability to serialize models. Model Zoo is a collection of models defined and shared by researchers using Caffe. Serialization and specification simplicity allow researchers to easily share model definitions and trained models with each other via Model Zoo. Sharing models and layer based definition enable creation of increasingly complex and capable models based on previously created ones.

TensorFlow

Developed by Google, TensorFlow is the most widely used machine learning library. It has built in support for training on multiple machines. Written in C++, it has bindings for Python, Java, Go, Swift, and JavaScript, as well as support for mobile devices. Graph based execution differentiates TensorFlow from other machine learning libraries [18]. Models define the graph and are constructed using Application Programing Interfaces (APIs) providing input processing, choice of computations, and connections between layers. Due to the graph architecture, input and neuron values are not accessible until the computation graph runs. Therefore, all input processing and calculations must be implemented in C++ within the TensorFlow library. Similarly to Caffe, C++ implementation of operations means GPUs can be used to accelerate computation. To speed CPU calculations, TensorFlow has taken advantage of AVX instructions' quick matrix multiplication.

While attractive from a performance perspective, the graph based architecture requires that preprocessing of data all be done within C++. At the same time, TensorFlow only promises API stability for its Python bindings. Programing of data preprocessing is therefore often done in Python so as to avoid having to reprogram when updating to new versions of TensorFlow. Models which require require data preprocessing not included within the C++ API are unable to be serialized due to a limitation of the Python API [19]. This deficiency is due to graph values not being available until the computation phase, which must be done in C++ to allow for serialization and portability.

Applications using TensorFlow define their models via high level estimation APIs, or lower level APIs that provide control over individual layers in a neural network. Estimators only require the number of neurons within each layer, specification of the learning algorithm (such as gradient descent), and the data types of the inputs. TensorFlow's low level APIs allow developers to provide the same customizations on a per layer basis. Customization via this interface allows developers to increase their model's accuracy by fine tuning per-layer options. With support from industry partners, development of TensorFlow has lead to a stable, well documented, widely deployable solution, enabling machine learning within a variety of applications.

Application

Theory behind machine learning and neural networks is well established. Research has lead to a plethora of implementations in software and hardware. Datasets to train models can be found in abundance through research papers and on Kaggle, a centralized repository for ML data. At this point in time software and hardware developers are fortunate enough to have access to a variety of libraries, technologies, and past research, allowing them to apply machine learning to solve problems and create useful applications.

FPGAs relative low power consumption and availability have made them the optimal choice for embedded applications. GPUs have made training on large datasets, images, and videos feasible, pushing the boundaries of computer vision. Even with minimal training data, low training times, and simplistic machine learning models, high accuracy can be achieved on what were once thought to be non-automatable processes.

Mobility Aid

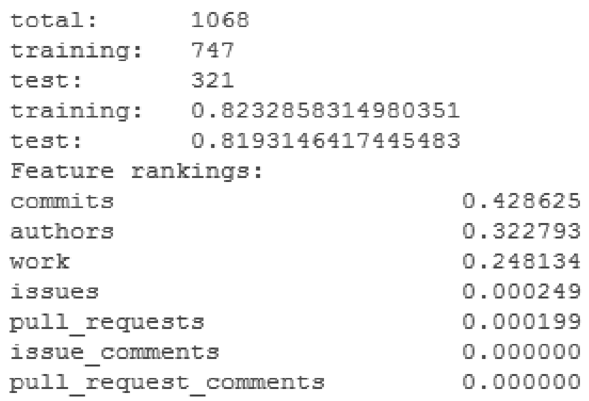
An embedded system which illustrates the capabilities of neural networks and ability to deploy them in a wearable device is the mobility aid developed by Poggi, et al. These researchers created a solution which uses machine learning powered computer vision to provide the same assistance to a user as a guide dog. Feeding data from cameras containing image and depth information through a neural network implemented in a Xilinx Spartan 6 FPGA, they were able to accurately classify objects to alert visually impaired users of objects in their surroundings. They created a CNN with multiple hidden layers using Stochastic Gradient Descent as their cost function. Convolutions in their CNN helped them perform image classification. Their solution identifies objects in the user's surroundings using the trained network on the FPGA, then alerts the user to potential obstructions [7].

Open Source Software Security Evaluation

To test my understanding of machine learning and neural networks I applied them to a problem facing my team at Intel. Open Source Software (OSS) is software where source code is released for free, and usage of the source code must adhere to the specified license. Licenses are legally binding documents which explain how the software may be used, modified, or redistributed. Caffe and TensorFlow are both OSS projects. Most software relies on open source libraries to implement complex tasks. When one software project relies on another, the relied upon project is a dependency. The benefit of OSS is that many people and organizations can contribute to these projects and create reliable software for the benefit of everyone.

Intel has an Open Source Technology Center (OTC), which employs developers who work on important OSS projects relied on by critical infrastructure and companies around the world. I am a part of the software security team within OTC. One of our team's responsibilities is ensuring product teams and OSS developers choose secure libraries for use in their software. This is important because software is only as strong, or secure, as its weakest dependency. Being that OSS can be put on the internet by anyone, not all libraries follow strict security practices, or fix security issues when notified. Internally we have a tool for handling submission and review of dependencies. Product Security Experts manually assess the security of requests and classify them as either Whitelist, Conditional, or Blacklist. This process is known as whitelisting, and has consumed hundreds of hours of our engineers time.

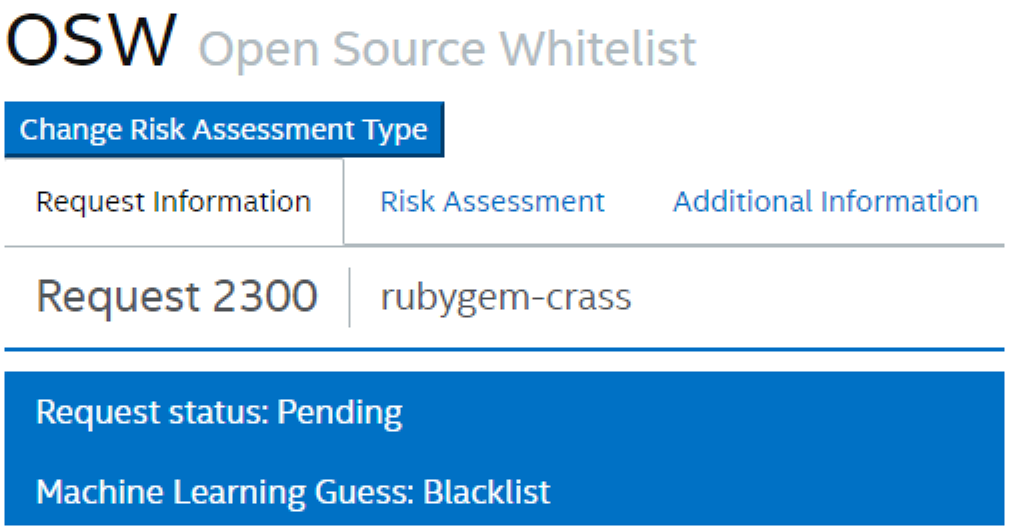
Previous attempts to automate the whitelisting process using machine learning achieved 60% accuracy. This was due to unreliable numeric data stored in the whitelist tool. Thus, the first step in applying ML was to automate the collection of metrics explicative of each request's maintenance cycle. In order to collect these metrics, the DFFML library [22] was created. DFFML provides a pipeline wherein metrics are collected, models are defined, and model accuracies are compared.



```
total:       1068
training:    747
test:        321
training:    0.8232858314980351
test:        0.8193146417445483
Feature rankings:
commits                         0.428625
authors                         0.322793
work                            0.248134
issues                          0.000249
pull_requests                   0.000199
issue_comments                  0.000000
pull_request_comments           0.000000
```

Figure 6: Importance of metrics as assessed by Random Forest

Using DFFML, metrics were collected for each of the 4000+ manually assessed libraries in the whitelist. Models were defined using TensorFlow as well as the scikit.learn APIs. Due to ambiguity associated with the conditional rating, only whitelisted and blacklisted libraries were used to train and assess the model. In order to understand the importance of each metric collected, the Random Forest algorithm was used. It provides the percentage importance of each input feature to the classification of the request. As seen in figure 6, it was found that the model considered many of the features our team initially identified to be unimportant.



Request 2300 | rubygem-crass

Request status: Pending

Machine Learning Guess: Blacklist

Figure 7: Integration of DFFML into whitelist tool

Once meaningful features had been identified using the Random Forest model, another model was created using the DNNClassifier in TensorFlow's estimator API. This trained model, having hidden layers of 10, 20, and 10 neutrons respectively, achieved ~90% accuracy. Deep Neural Network accuracy was only ~9% more than the Random Forest model. The DNN model was integrated with our whitelisting tool as seen in figure 7. Our team is now conducting a pilot of the model's assistance in the whitelisting process. After further evaluation we will implement automatic classification of incoming requests, reducing time spent on manual review.

Conclusion

Open source machine learning libraries, such as Caffe and TensorFlow, have accelerated the field by allowing researchers to contribute implementations of their theoretical advancements for others to build on. An abundance of OSS libraries have made application of machine learning to solve once un-automatable problems an exercise in integration. As machine learning algorithms and methods of data collection improve, accuracy on once difficult problems will increase. In the future, Internet of Things (IoT) devices will create a digital representation of our physical world. IoT devices provide data needed to train machine learning models and make decisions reflected in physical systems. An already present application of this is self driving cars.

Future research and application of ML and neural networks will revolutionize industries. Similarly to how robotics automated the assembly line, machine learning will enable automation of other fields, such as healthcare. Application of machine learning in the medical field is providing doctors with new ways to identify and fight cancer. This year, machine learning models have become more accurate than doctors at identifying melanoma, achieving 96% accuracy [21]. With the right data and model, ML can surpass humans in its prediction abilities, allowing us to create applications which save time, money, and lives.

Bibliography

[1] Krieg, Mark L. *A tutorial on Bayesian belief networks.* No. DSTO-TN-0403. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION SALISBURY (AUSTRALIA) ELECTRONICS AND SURVEILLANCE RESEARCH, 2001.
[2] Renelle, Tyler. "Algorithms - Intuition." Audio blog post. *Machine Learning Guide*. OCDevel, 11 Feb. 2017. Web.
[3] Wang, Haohan, Bhiksha Raj, and Eric P. Xing. "On the Origin of Deep Learning." *CoRR* abs/1702.07800 (2017): n. pag. 1 Mar. 2017. Web.
[4] M. Forssell, “Hardware Implementation of Artificial Neural Networks,” http://users.ece.cmu.edu/~pgrover/teaching/files/NeuromorphicComputing.pdf, 2014.
[5] Meng, Wenjia, et al. "Two-Bit Networks for Deep Learning on Resource-Constrained Embedded Devices." *arXiv preprint arXiv:1701.00485* (2017).
[6] Hegde, Gopalakrishna, Siddhartha, Nachiappan Ramasamy, Vamsi Buddha, and Nachiket Kapre. "Evaluating Embedded FPGA Accelerators for Deep Learning Applications." in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2016).
[7] Poggi, Matteo, and Stefano Mattoccia. "A Wearable Mobility Aid for the Visually Impaired Based on Embedded 3D Vision and Deep Learning." in *2016 IEEE Symposium on Computers and Communication (ISCC)* (2016).
[8] Henaff, Mikael, et al. "Unsupervised learning of sparse features for scalable audio classification." *ISMIR*. Vol. 11. No. 445. 2011.
[9] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, “What is the best multi-stage architecture for object recognition?” in *Proc. International Conference on Computer Vision (ICCV’09)*. IEEE, 2009.
[10] Lacey, Griffin, Graham W. Taylor, and Shawki Areibi. "Deep learning on fpgas: Past, present, and future." *arXiv preprint arXiv:1602.04283* (2016).
[11] Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014.
[12] Cui, Henggang, et al. "GeePS: Scalable deep learning on distributed gpus with a gpu-specialized parameter server." *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.
[13] Urbanowicz, Ryan J., and Jason H. Moore. "Learning classifier systems: a complete introduction, review, and roadmap." *Journal of Artificial Evolution and Applications* 2009 (2009): 1.
[14] Pearl, Judea. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.
[15] Cooper, Gregory F. "The computational complexity of probabilistic inference using Bayesian belief networks." *Artificial intelligence* 42.2-3 (1990): 393-405.
[16] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
[17] Li, Mu, et al. "Scaling Distributed Machine Learning with the Parameter Server." *OSDI*. Vol. 14. 2014.
[18] Abadi, Martín, et al. "TensorFlow: A System for Large-Scale Machine Learning." *OSDI*. Vol. 16. 2016.
[19] TensorFlow py_func documentation. https://www.tensorflow.org/versions/r1.7/api_docs/python/tf/py_func
[20] Nielsen, Michael A. "Neural Networks and Deep Learning." *Neural Networks and Deep Learning*. Determination Press, Jan. 2017. Web.
[21] Gaudiuso, Rosalba, et al. "Using LIBS to diagnose melanoma in biomedical fluids deposited on solid substrates: Limits of direct spectral analysis and capability of machine learning." *Spectrochimica Acta Part B: Atomic Spectroscopy* 146 (2018): 106-114.
[22] John Andersen, Intel, "Data Flow Facilitator for Machine Learning” https://github.com/intel/dffml