Using Neural Networks to Train an Operating System Scheduler

Changhao Chenli, Ryan Karl, and Jonathan Takeshita
University of Notre Dame, Department of Computer Science and Engineering
CSE 60641: Graduate Operating Systems
Email Addresses: {cchenli, rkarl, jtakeshi}@nd.edu

Abstract—**Optimal process scheduling is a common problem in computing. In this paper, we investigate whether ideas from machine learning and neural networks can be applied to improve process scheduling. We build a neural network-based process scheduler and compare its performance against standard and experimental schedulers to determine its overall performance. After training, our scheduler is able to output a mostly optimal sequence to run a set of processes in, after receiving as input a random list of processes to be run. After analyzing several experiments, we describe how our scheduler has generally comparable or partially inferior performance to current scheduling algorithms. We offer an analysis of the shortcomings of our technique and discuss what challenges still exist that prevent this technique from being successfully applied to the scheduling domain, along with the future directions of our work.**

I. Introduction

All modern operating systems support multitasking, or the ability to concurrently run multiple processes at once. This is accomplished either through context switching and/or parallelism. Context switching involves cycling through current processes and allowing each to enter the running state one at a time while the others wait until all the processes have finished, whereas parallelism is when multiple CPU resources are assigned to perform work on current processes to allow for the parallel execution of multiple tasks at the same time. However, if processes are permitted to naively interact with computing resources, the overall system often has poor performance. Instead, a special kernel-level process called a scheduler is used to decide the order and duration for which a process should be assigned CPU time, to prevent resource contention and optimize overall performance [10].

A variety of scheduling policies currently exist that optimize the order of processes for certain performance goals, such as high responsiveness or throughput. However, there has been relatively little work investigating whether using ideas from artificial intelligence and machine learning to train a scheduler based on a user's unique workload can improve on classic scheduling algorithms. For this project we investigate whether ideas from neural networks (a subfield of machine learning) can be used to create scheduling algorithms that can outperform both canonical and experimental scheduling algorithms.

The novel contributions of this paper include:

- A neural network-based scheduling framework.
- An investigation into the viability of using neural networks to learn trends relevant to real-world process scheduling.
- An investigation into the viability and performance of using neural networks in a simulated real-time framework, with comparisons against ordinary scheduling algorithms.
- An analysis of results from the comparison of ordinary and neural network-based scheduling algorithms.
- A discussion of the strengths and weakness of neural networks when being adapted to solve the problem of process scheduling, our conclusions as to why they are not currently a practical solution, and what technological advances need to develop in order for them to become a viable solution.

A. Organization of the Paper

II. Related Work

There has been some previous investigation into this area. A useful survey paper in this area is [17], but many of the example techniques they discuss are applied to general manufacturing systems and are over two decades old. There is a study applying machine learning to schedulers to optimize energy efficiency [4], but while this work aims for reasonable performance, their systems tolerate some performance degradation if it allows for high energy conservation. Also, [18] applies machine learning to schedulers, but is only interested in real-time systems, while we target general systems. The paper [5] is similar to our work, but only works in domains with low knowledge and is over a decade old. Among modern studies in this area, the most prominent is [15], which applied off the shelf machine learning technology to moderately improve the existing Linux scheduler. Also of note is [5], which used machine learning in combination with a novel Bayesian classifier to improve a scheduler's dynamic choice of what algorithm to use. While their work is similar to ours, over a decade has passed since they performed their analyses. Thus, their conclusions may no longer be accurate due to modern improvements to both neural networks and Linux process scheduling. We proposed to create a scheduling algorithm based on a neural network (NN), a higher-level and more precise machine learning scheme, while other works used older models. Negi and Pusukuri used machine learning schemes to estimate the execution time and classification of processes, based on various traits [15]. In contrast to their work, our project is going to attempt to create a neural network to mimic different scheduling algorithms.

III. Background

Our project draws heavily on a variety of fields including scheduling, artificial intelligence, and neural networks. We give an overview of each below.

A. Scheduling Algorithms

Scheduling algorithms are used to allocate the computational resources among different subjects which ask for service simultaneously. The main goals of scheduling algorithms are to prevent processes from starvation, ensure processes meet deadlines, and to try to make CPU allocation (more) fair. Concurrency is a very important notion we consider, as in scheduling and operating systems it can improve performance for multitasking systems. However, for simplicity in this project, we focus on only uniprocessor algorithms. As most uniprocessor algorithms can be adapted to multiprocessor systems, this does not make a uniprocessor-focused approach less meaningful in a time of concurrent computing. In step with other advances in computing, scheduling algorithms are also improving very quickly [1].

We focus on three scheduling algorithms. The Earliest Deadline First (EDF) algorithm is very simple in operation: it simply selects which job to run by choosing the one with the earliest deadline. The Rate Monotonic (RM) algorithm similarly selects the job with the highest assigned priority. A more complicated algorithm is the Global Fair Lateness (G_FL) algorithm, which recalculates job priority in order to minimize upper bounds on lateness (defined as the difference between a job's deadline and its actual time of completion) [9].

B. Artificial Intelligence and Machine Learning

Our project relies on recent advances in artificial intelligence (AI), a subfield of computer science that aims to build machines that can learn from data, in the same manner as humans. AI systems have been built that can anticipate or adaptively assist people with problem solving in a variety of sophisticated tasks, including speech recognition, image classification, and language translation, among others [21]. There are currently many active areas of AI research, but for our work we mostly leverage ideas from its subfield of machine learning.

Machine Learning can be broadly defined as the effort of training computers to learn patterns from data. Successful machine learning algorithms are able to function on data that is not drawn from training samples and classify information that has not been previously seen [7]. More specifically, a machine is fed a representation of data that has been classified in a manner the machine can parse, referred to as training data. After this, the accuracy of the classifications is scored to build evaluation frameworks the machine can use to classify future data. This process is repeated several times with training data to optimize the system and find the framework with the best score. This process aims to construct a framework that is highly accurate after it has run through all of the training data.

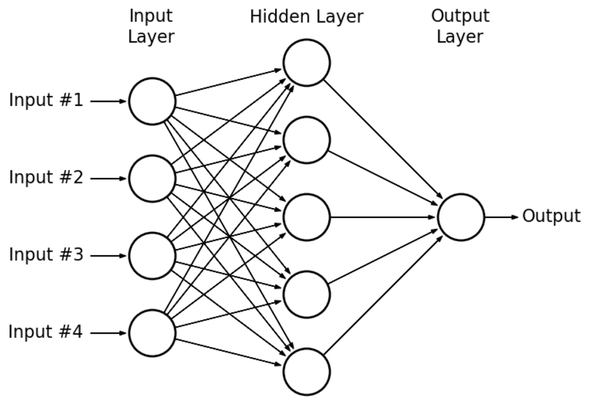

Fig. 1: Diagram of a Typical Neural Network

C. Neural Network (Overview)

A neural network (NN), also known as an artificial neural network, is a type of machine learning model that draws inspiration from a biological neural network, often exemplified by the human brain. The neural network itself is not an algorithm but a kind of structure to combine many machine learning algorithms together and deal with complex and large inputs. In contrast to the structure of traditional machine learning algorithms, which have an input layer and output layer, neural networks add hidden layer(s) of neurons into the frame which greatly improve the accuracy. The models trained by a neural network are usually considered to think like a human, but judge like a program, which means they work in a more rational way with a simple human-like brain [11].

A neural network contains many processing nodes that are interconnected into layers. These nodes are fed data that moves in one direction through the network. Each node will receive data from several nodes connected to it in the previous layer and send data to several nodes that it connects to in the following layer. A node assigns a weight (a numerical value) to each connection from the previous layer, and each time it receives a piece of data along this connection, it multiplies it by the corresponding weight. After this, it adds all of the products generated together into an output value. If the value computed is less than a specified threshold then the node does not transmit any data to the following layer, but if the value is greater than the threshold, the node transmits the value to all of the nodes it connects to in the following layer (other activation functions that determine how the node transmits information exist, but the step function described here is the biologically inspired, intuitive model). Between the first input layer and the final output layer, there are typically many intermediate layers called ”hidden layers”, which allow the system to better tune its final output [16]. Generally speaking, neural networks learn better and become more accurate as the number of hidden layers increases. See Figure 1 for a diagram of a typical neural network [16].

When a neural network is being trained, all of its weights and thresholds are initially set to random values. Training data is fed to the bottom layer (the input layer) and it passes through the succeeding layers, until it finally arrives, radically transformed, at the output layer. During training, the weights and thresholds are continually adjusted until training data with the same labels consistently yield similar outputs. Neural networks learn via backpropagation, a feedback process where the output of the final layer of a neural network is compared against the output it was supposed to produce. By analyzing the difference between the two values, the network can modify the weights of each connection between the nodes by traversing the network topologically from the final output layer to the input layer [20]. The goal of this process is to reduce the difference between the two until they are almost identical, so the network responds accurately when posed a problem [19].

Many parameters besides the data affect the behavior of the neural network. These are called hyperparameters, to distinguish them from ordinary parameters of the model. One of the most important of these is the learning rate, which determines the speed at which the neural network learns from its input. If this parameter is set too high, then the neural network's weights and thresholds can diverge. In the opposite case, the network can fail to learn quickly enough to evolve significantly based on its given training data. As our group learned the hard way, this can lead to problems ranging from a total lack of meaningful output to programs crashing due to numerical overflow. A similar hyperparameter is the momentum, which prevents the network from targeting a local minimum or maximum (as opposed to global minimums/maximums).

Another important hyperparameter is the choice of activation function. The step function described above is the one most analogous to biological neurons. However, other functions may be more suited to a particular type of work. Most activation functions will be (at least stepwise) continuous, and will have output in an interval of $[0, 1]$ or $[-1, 1]$. One exception to this is the ReLU function $f(x) = max(x, 0)$ and its smoother variations, who can produce outputs over all positive numbers. In situations where a numerical output is desired (e.g. the predicted execution time of a process), the ReLU function may be the most useful.

D. Neural Network (Strengths)

The various strengths of neural networks in pattern recognition make them an attractive tool for use in scheduling problems. Additionally, neural networks are one of the most powerful and popular artificial intelligence models nowadays, for a variety of reasons [2]:

- Function approximation: A neural network, given enough data and training, can approximate any continuous function of its inputs, to a high degree of accuracy [3]. Other models (machine learning or conventional) lack this type of versatility.
- Simplicity: Though some aspects of neural networks are complicated, the overall structure can be easily understood as a result of its biological inspiration. Further, a neural network lends itself to simple implementations, which allows programmers to later extend the functionality of the neural network.
- Performance: One of the main reasons for the current popularity of the neural network is that it can outperform both conventional and artificial intelligence models in correctness of results. This makes it a highly attractive choice of model to use.
- Options: The user has a wide variety of hyperparameters they can choose to tune, including momentum, learning rate, and the activation function. The user also has the choice of when and how to change these parameters - with careful programming, the learning rate or momentum can be dynamically adjusted to decrease as training progresses, which helps to prevent overfitting of data.

E. Neural Network (Weaknesses)

While neural networks are highly versatile and effective compared to other artificial intelligence models, they do have some critical weaknesses we (and others [2]) have discovered that make them unsuitable to some applications:

- Volume of data required: Neural networks require an immense volume of data to be useful in practice. There is the additional problem of gathering data that is representative and free of anomalies, though this is not a problem unique to neural networks.
- Fixed input/output size: Traditional algorithms can be designed to handle a variable number of inputs, with the input size limited only by system resources or parameters. Neural networks, in contrast, have a fixed size of input and output nodes, which limits them to a constant-size input and output. This is a highly undesirable property for algorithms such as schedulers, that must be able to take in a variable number of inputs.
- Computational effort: To either train or get results from a neural network, a large amount of computing may be required. While this can be mitigated by decreasing the size and complexity of the network, this approach may also reduce the accuracy of the network.
- Domain: As noted above, a neural network can learn any continuous function of its inputs (with some caveats). However, this domain of problems does not describe the full range of problems that an algorithm must be designed to solve. For example: a scheduling algorithm may ask the question ”Which of these processes should run next?” This problem's input is not always easily expressed numerically, nor is the output continuous (or even over a continuous range).

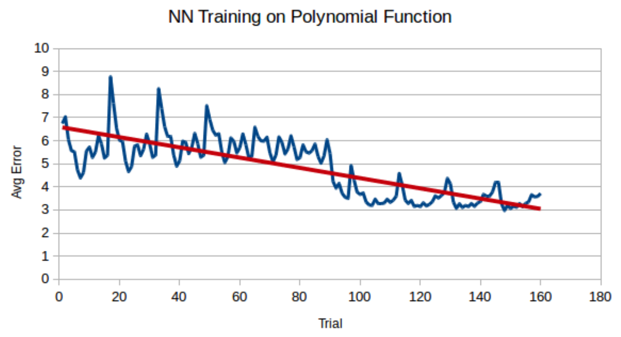

Fig. 2: Neural Network Error in Evaluating a Polynomial Function

IV. Approach and Methodology

A. Neural Network Implementation

Our first task in this project was to create an implementation of a neural network. The purpose of this step was not only to construct a neural network module to use in our network, but also to allow our group to gain experience with artificial intelligence, neural networks, hyperparameters, and other relevant topics.

Our neural network is written in C++, and adapted from the SimpleNeuralNetwork module by Zehao Huang [12]. Several changes to the original source code were made for the reasons of safety, ease of future modifications, usability, and best software engineering practices. We started by splitting the single file containing all the code into multiple files. We next rewrote portions of the code to be consistent with modern coding practices, including more standard data typing and parameter passing. Later, we also had to take precautions at both compilation and runtime to avoid numerical overflow. In order to allow permanence of training beyond a single process' execution, we endowed the network with the ability to write its weights and biases to a file, or initialize itself from a file. Finally, we built a front-end interface to the neural network. This interface allows an ordinary user to train or evaluate the neural network on data from a file. Using the front-end, a user can also specify hyperparameters such as the learning rate or momentum, or the number of passes to make over training data.

B. Basic Neural Network Validation

We then worked to demonstrate the efficacy and applicability of our neural network. When training the neural network to learn a multivariate polynomial, we were able to reduce the reported recent average error by half from its original value (shown in Figure 2). We next worked to show that the scheduler could be trained on data more pertinent to our project than polynomial functions.

C. Data Collection and Further Validation

To collect real-world data, we used two Linux utilities. The first of these was the built-in *size* tool, which displays the total file size and sizes of individual execution segments when called on an ELF (Executable and Linkable Format) executable. The *size* utility displays sizes of the *.text*, *.data*, and *.bss* segments in addition to total file size. The second utility was the *forkstat* tool, which can track system events (e.g. fork, exec, exit) of processes. This utility gives information about a process' PID, child/parent status, time, caught event, and executed command. Most pertinently, *forkstat* can be used to display the duration of a process that has exited. We wrote a C++ program to parse the output of *forkstat* upon a process' exit and use *size* to collect data on the size of the deceased process' executable file. We used a shell script to connect *forkstat* and our program calling *size*, and ran the script on a team member's personal computer. This yielded us a large amount of real-world process data that we could use to train our neural network on.

We were able to train our network on the collected real-world process data, and obtain similar results as with the polynomial data. However, we decided to change our approach from a real-world environment to a simulated one, for several reasons. First, our training data was highly representative of a single group member's use patterns, but not of more general usage. For example, the data included many uses of the Brave web browser, which we found to have a zero-size *.data* segment and whose runtime is user-determined. Second, our data was not satisfactorily granular - *forkstat* only reports durations down to the millisecond. Third, our data-collection framework was not usable on the CRC or other research computing environments - *forkstat* requires root access. Besides these deficiencies with the real-world approach, we believe that a simulated environment is more suited to testing scheduling, especially at earlier states of development where anomalies or failures may have greater effects.

D. Switching to a Simulated Environment

To that goal, we then began investigating the SimSo scheduling simulation framework [6]. SimSo has the advantage that it provides predictable and deterministic simulation of tasks. SimSo is written in Python, causing an efficiency loss, but we believe the other advantages of Python make the tradeoff beneficial: we can easily write new scheduling algorithms, extend new or preexisting algorithms to output data for our neural network, and run SimSo without root access. Another advantage of working with SimSo (and simulated environments in general) is that the computing environment we use becomes irrelevant, precluding the need to run tests in virtual machines or on CRC computers.

The design of our training framework is shown in Figure 3, and is the standard method of training a neural network. During this process, we gathered data attributes about typical process workloads from a SimSo process set. We then ran a SimSo scheduler on the process set, and recorded the sequence in which the processes were scheduled. Following this, we combined this ground truth data with the original attributes we gathered to act as a training data set for our neural network based scheduler. We then inputted the training data into the

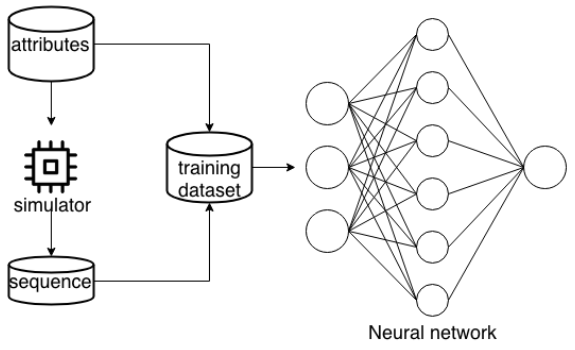

Fig. 3: Process for Training Our Neural Network

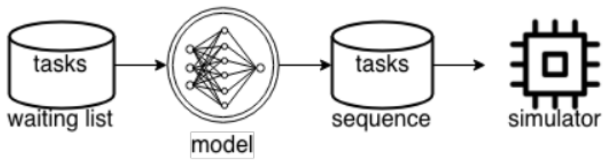

Fig. 4: Neural Network-based Scheduler Design

neural network to train it to find the optimal order to run the processes in the workload, in terms of standard metrics detailed at length in the Experimental Results section.

When switching to SimSo, we had the choice of continuing to use our previously created neural network code, or starting from scratch with a Python implementation. We decided to retain the C++ codebase, both due to the desire to strike a balance between efficiency and convenience and the time constraints upon this project.

Our first step in integrating our scheduler with SimSo was to update the neural network code with a Boost::Python interface. This allowed us to compile our C++ network to a shared object file (.so), which allowed exposure of the neural net's C++ classes and functions to a Python program.

E. Training and using our Neural Network

For each scheduler we wished to mimic, we modified the original scheduler to output training data (groups of the relevant inputs and correct output). Our neural network uses a fixed number of inputs. In it, we use the scheduler's heuristics (e.g. the deadline for an Earliest Deadline First algorithm) to select the most likely candidates, up to the number of inputs. If not enough inputs exist, data duplicating existing input is used to fill the empty spots. The candidate processes are then randomly reordered (so that the network does not simply learn to choose the first or last inputs), and formulated into training data, which is then written to a file.

In addition, we have written a scheduler that uses the neural net and attempts to mimic the original scheduling algorithm. In this scheduler, we again must prune or pad the input data to fit the fixed-sized input vector of the neural network (the need to prune or pad data is a weakness of the neural net model, noted above). Our scheduler then uses the output of the neural network to determine which process to run next. The structure of this scheduler is depicted in Figure 4. From a waiting list of eligible tasks, our trained neural network based scheduler selects the optimal order for the processes to be run, and after arranging this sequence executes them in order within the SimSo simulation framework.

V. Experimental Results

In this section, we discuss our experiments that we used to evaluate the ability of a neural network to be used as part of a scheduling algorithm.

A. System Specifications

Most of our work and much of the preliminary computation was carried out on a team member's computer. This computer was running Ubuntu 16.04 LTS, with 8GB RAM and an Intel(R) Core(TM) i5-6500 CPU running at 3.2 GHz with 4 cores. Our neural net was written in C++14, and we ran SimSo under Python 2.7. The neural net was integrated into SimSo using the Boost::Python interface under Boost version 1.58.

B. Benchmarks

In our experiments, we use SimSo's interface to give each scheduler various predetermined workloads to represent common use cases and measure how well each performs with regards to eight standard metrics.

The first three we discuss are qualitative: fairness (whether each process has an equal chance of being executed as the other), predictability (whether the next process to be executed after the process currently being executed can be determined), and preemption (whether the scheduler has the power to interrupt and later resume other tasks in the system).

We also measure five quantitative metrics: throughput (number of jobs per cycle interval), number of abortions (the number of times a job does not finish before the deadline), abortion rate (the percentage of total jobs in the workload that are aborted), number of preemptions (the number of times a running job is paused to let another job run), and average task completion time (the average number of cycles it takes a job to finish).

C. Experiment Setup

We trained and tested our neural network-based scheduler against three scheduling algorithms: Earliest Deadline First (EDF), Rate Monotonic (RM), and Global Fair Lateness (G_FL). The EDF and RM scheduling algorithms are well-known, canonical, scheduling policies. The more complex G_FL algorithm, proposed by [9], aims to minimize the maximum bounds of lateness (the difference between deadlines and completion times). This set of scheduling algorithms is representative of traditional scheduling paradigms, and strikes a balance between simplicity and complexity.

For each of these algorithms, we trained our neural network-based scheduler to mimic the chosen algorithm. Training data was generated by applying the chosen algorithm to task

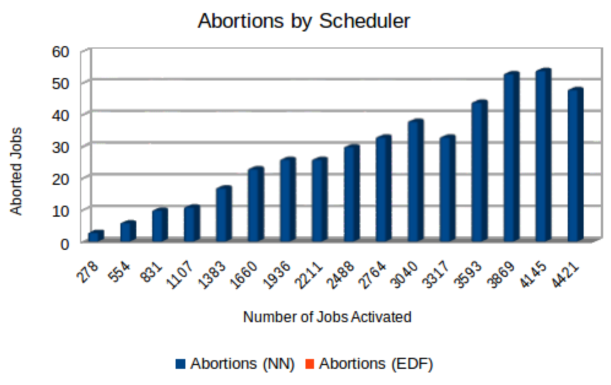

Fig. 5: EDF vs. NN Number of Abortions

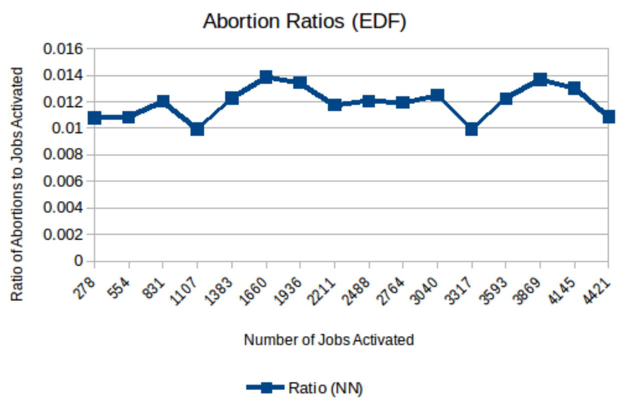

Fig. 6: EDF vs. NN Abortion Ratio

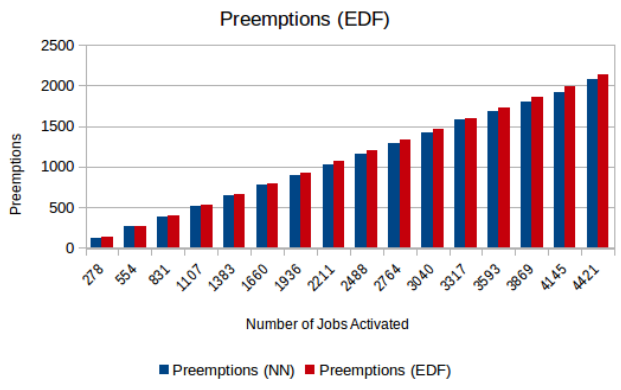

Fig. 7: EDF vs. NN Number of Preemptions

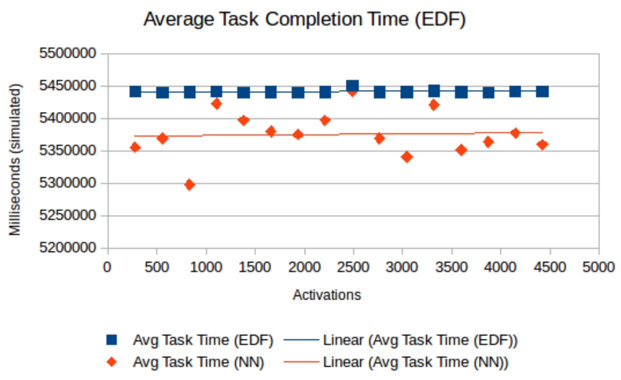

Fig. 8: EDF vs. NN Average Task Completion Time

scheduling problems generated using Stafford's Fixed Random Sum algorithm [8]. We then evaluated the neural network-based scheduler against the original algorithm by running both on sample workloads (different from the workload used for training). We then parsed SimSo output logs from the schedulers' executions to collect relevant data.

When utilizing our neural network for scheduling, we chose to use a 4-layer structure, where the output layer's size is the same as the number of jobs inputted. This allows us to use the neural network similarly to a classifier: the output node with the highest value corresponds to one of the input jobs. Thus to use the network to select a process, the algorithm uses the process with the highest assigned value outputted by the network.

D. *Fairness, Predictability, and Preemption*

With regards to fairness, the EDF and G_FL schedulers are both fair, because they allow each process an equal opportunity to run based on how soon the nearest deadline will be reached and how much lateness the system can tolerate respectively. RM is not fair because the user statically configures the priority of each process and can force the system to give preference to some processes over others. Our own NN scheduler is fair because when it is being trained each process has an equal opportunity to run in the system and the order is simply determined by the current workload of the system. In terms of predictability, all of EDF, RM, and G_FL are predictable, because the next process to run is determined in a deterministic manner based on the closest deadline, next highest fixed priority task, or least lateness incurred. Our NN scheduler is not predictable because it determines the optimal process to run next simply based on past system behavior and does not make use of a statically preconfigured metric. In terms of preemption, all of the schedulers allow preepmtions of currently running tasks if this allows a process to run that has a closer deadline, a higher priority, less lateness, or a more optimal ordering.

E. *Performance When Trained Against EDF*

When trained and compared against the EDF algorithm, our neural network-based scheduler was not able to replicate EDF's guarantee of not missing deadlines, and as a result our scheduler had some number of aborted tasks, as shown in Figure 5. However, the number of aborted jobs compared to the number of total jobs activated during the simulation is small. The ratio of abortions to activations was between 1% and 1.5% in our experiments, as shown in Figure 6.

For both EDF and our neural network-based scheduler, the number of preemptions increases roughly linearly with the number of jobs activated. This is shown in Figure 7. The

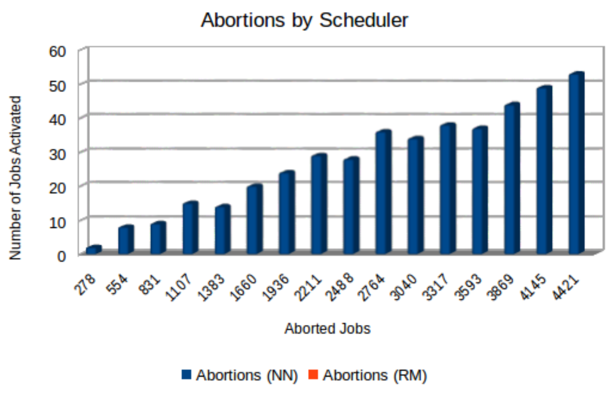

Fig. 9: RM vs. NN Number of Abortions

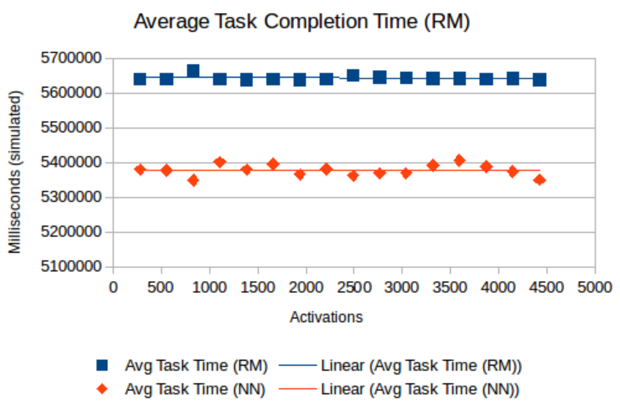

Fig. 12: RM vs. NN Average Task Completion Time

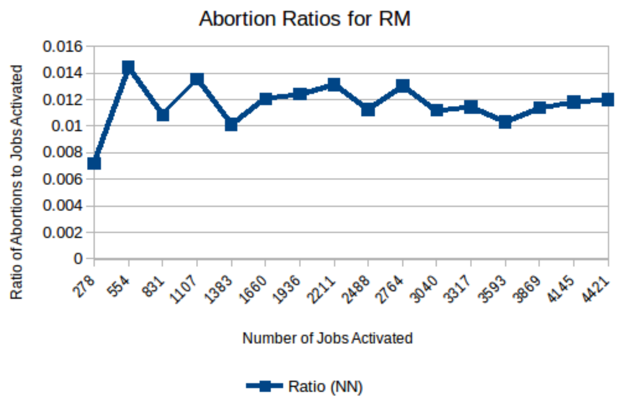

Fig. 10: RM vs. NN Abortion Ratio

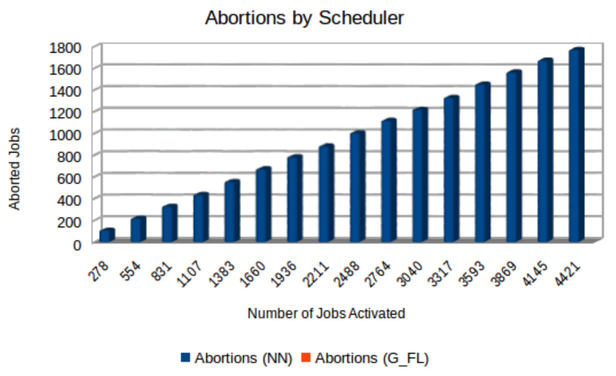

Fig. 13: G_FL vs. NN Number of Abortions

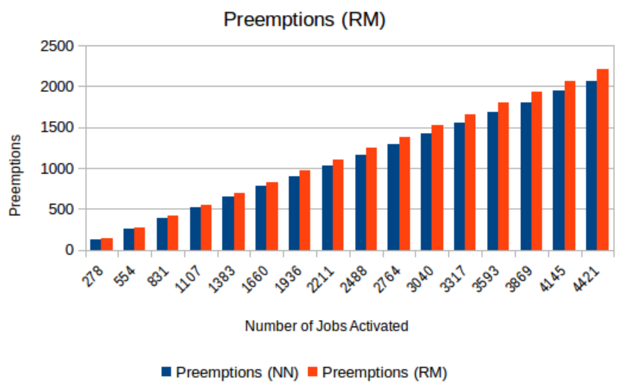

Fig. 11: RM vs. NN Number of Preemptions

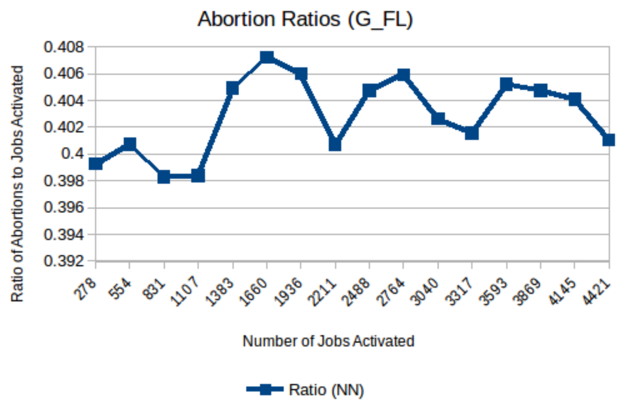

Fig. 14: G_FL vs. NN Abortion Ratio

neural network-based scheduler actually outperformed EDF by having fewer preemptions. However, we hypothesize that this improvement is due to the greater number of aborted jobs that the neural network-based scheduler produces. More aborted tasks will result in fewer remaining tasks to complete, which will lead to fewer preemptions overall.

We observe similar results and take similar conclusions from an analysis of average task completion times, shown in Figure 8. The average task completion time is smaller (with much higher variance) for the neural network-based scheduler. Again, we believe that this is due to the neural network-based scheduler's abortions of some jobs. Aborting some jobs means that the remaining jobs can achieve a better average runtime.

Similarly, in terms of tasks completed per epoch, we see in Figure 17 that both EDF and our neural network-based scheduler have generally comparable performance. The num-

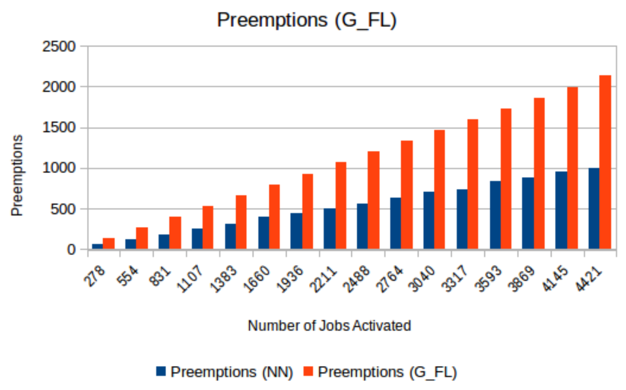

Fig. 15: G_FL vs. NN Number of Preemptions

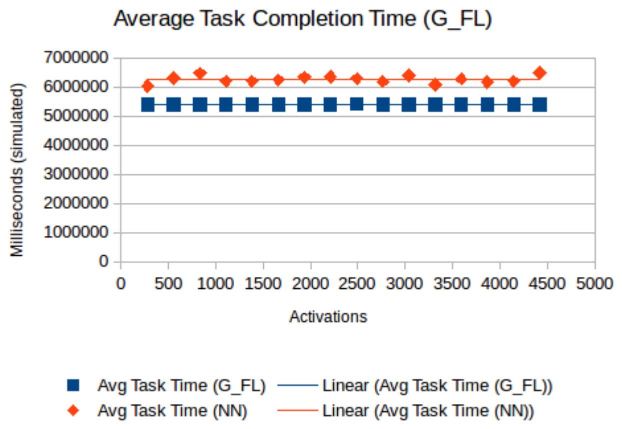

Fig. 16: G_FL vs NN Average Task Completion Time

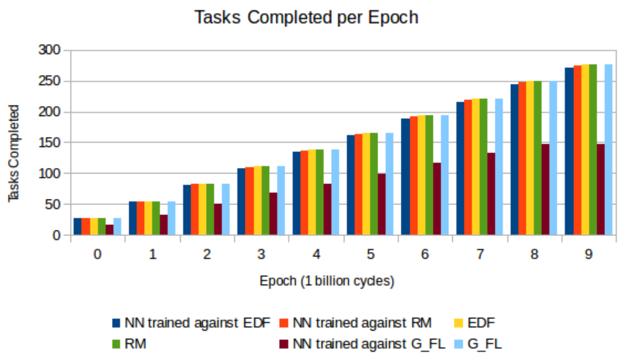

Fig. 17: Cross Scheduler Comparison of Tasks Completed per Epoch

ber of tasks completed is slightly lower (with somewhat higher variance) for the neural network-based scheduler. We again hypothesize that this is likely due to the neural network-based scheduler's tendency to abort some jobs.

F. Performance When Trained Against RM

When we trained and evaluated our neural network-based scheduler against the RM algorithm, the results were very similar to the EDF comparison. Again, the RM algorithm was able to guarantee making deadlines, while our neural-network based scheduler was not. This resulted in the neural network-based scheduler aborting some jobs, as shown in Figure 9. Again, as shown in Figure 10, the proportion of aborted jobs to the total number of activated jobs was small, in the neighborhood of 0.7% to 1.5%.

As with the EDF comparison, the number of preemptions scaled roughly linearly with the number of jobs activated for both RM and the neural network-based scheduler. This is shown in Figure 11. Again, the neural network-based scheduler slightly outperformed RM, but this is most likely due to the proclivity of the neural network-based scheduler to abort some jobs.

The trend of similarity to the EDF results continues when analyzing average task completion time. As shown in Figure 8, the neural network-based scheduler again manages to gain a better average completion time than RM, though we hypothesize that this is due to the neural network-based scheduler aborting some jobs.

Once again, we see similar results in terms of tasks completed per epoch in Figure 17 where both RM and our neural network-based scheduler have generally comparable performance. The number of tasks completed is again slightly lower for the neural network-based scheduler, and we feel that this is likely due to the neural network-based scheduler's tendency to abort some jobs. Note that the neural network-based scheduler trained against RM performs slightly better than its counterpart trained against EDF, and this suggests that RM style schedulers, with statically preconfigured priorities and attributes, may be easier for a neural network to learn and identify in practice.

G. Performance When Trained Against G_FL

When trained and evaluated against the more complex G_FL algorithm, our neural network was less successful. G_FL, like EDF and RM, was able to guarantee no missed deadlines and thus no abortions. Our neural network-based scheduler was unable to make that guarantee, and aborted jobs at a linear rate (shown in Figure 13), similarly to the evaluations against EDF and RM. However, the abortions came at a much higher rate than what we observed when comparing against EDF and RM, approximately 40%, as shown in Figure 14.

Figure 15 shows that the number of preemptions is about half for our neural network-based scheduler, compared to G_FL. While this result appears to be encouraging, the rate of preemptions appears to be due to the high abortion rate of the neural network-based scheduler, and is thus not a sign of our scheduler's high performance but rather of its shortcomings.

When evaluating average task completion time, our neural network-based scheduler is somewhat inferior. Despite aborting a large number of jobs, its average task completion time was still higher than that of the G_FL algorithm's.

For our neural network-based scheduler, training it from a G_FL algorithm appears to give it the worst performance for tasks completed per epoch, as shown in Figure 17. Our scheduler's performance was only about half as good as other schedulers, both our other neural network-based ones and the EDF, RM, and G_FL algorithms.

H. Overall Analysis

Based on our experiments, our neural network-based scheduler has roughly comparable performance to existing schedulers in some cases, but in general was not able to match the performance of traditional scheduling algorithms. When trained and compared against the EDF and RM algorithms, the neural network-based schedulers were able to show similar performance, with miss/abortion rates of roughly 1%. In some applications with a high degree of fault tolerance, such a failure rate may be acceptable. For the more complicated algorithm of G_FL, our neural network-based scheduler's performance was much less optimal, with a miss/abortion rate close to 40%. Very few applications would be able to tolerate such a high failure rate.

A similar pattern repeated for the metrics of preemptions, average task completion time, and tasks completion by epoch - for the EDF and RM algorithms, neural network-based schedulers performed similarly to the original heuristic-based algorithms, but a neural network-based scheduler trained against G_FL performed noticeably worse than all other schedulers (both heuristic or neural network-based).

We hypothesize that the disparity in results is from the differences between the algorithms. EDF and RM are simpler algorithms, making it easy for a machine learning model to learn from their output. In contrast, the G_FL algorithm is more complex, and is thus not as well-suited to being learned by a neural network.

We also note that there is no theoretical guarantee that G_FL (or even EDF and RM) can be learned by a neural network: the Universal Approximation Theorem [3] only concludes the learnability of *continuous* functions. "Which job do I schedule next" (or the low-level equivalents we wrote) is not a continuous function. While scheduling may still be learnable, (as in the case of EDF and RM) there are no guarantees.

VI. Conclusions

A. Positive Conclusions

Artificial intelligence and neural networks are extremely powerful tools, applicable to a wide variety of domains. Both in our work and past work in scheduling, artificial intelligence and neural networks have shown promise in several ways:

- Trainability: Neural networks have been shown to be trainable on data from process scheduling workloads. One of the most notable examples of this is the one of Negi and Pusukuri [15], who used machine learning models to classify processes' work types (e.g. system, background, user-facing) and utilize that information in process scheduling. Their results also showed their improvement in the area of process scheduling. In the first part of our project, we similarly showed the ability of a neural network to determine process attributes. In the second part of our project, we showed that this ability can be directly utilized in scheduling.
- Intuition: Neural networks (and artificial intelligence in general) are capable of expressing rules not easily expressed in a formulaic fashion. In a complex application such as scheduling, this ability is extremely useful. Compared to traditional machine learning methods, neural networks add hidden layers into their structure, making it more accurate while the users will not have to care about the exact procedure neural networks do during the training step. For scheduling, as there may be many attributes from each process, using neural networks can solve the problem of multi-inputs after we know the output of a certain process.

B. Negative Conclusions

We have concluded that a neural net-based scheduling algorithm is a poor choice at this time. This is due in part to the computational effort and large amounts of data needed to train the neural network. Additionally, running the neural network may also be computationally intensive, which is undesirable for a scheduling algorithm in a real-time operating system. (This is one of the main reasons the $\mathcal{O}(n)$ scheduler used in Linux 2.6 was quickly replaced [13].) Another downside of neural networks is the fixed number of inputs and outputs, which makes an application to scheduling extremely challenging. Finally, a network trained on one user's workload (e.g. scientific computing) may give less optimal results for another user's workload (e.g. gaming).

Despite these pitfalls, we still think there may be a place for neural networks in scheduling, if some things were to change. We believe the following events or conditions could mitigate the weaknesses of neural networks with respect to scheduling:

- Cheaper computation: If neural network implementations become computationally lightweight enough to be comparable to a $\mathcal{O}(1)$ scheduler, then a neural network-based solution would become computationally viable.
- Specialized hardware: Similarly, if dedicated hardware for evaluation (and/or training) of neural networks becomes commonplace in computers, such as Google's TPU [14], then the burden of computation required for a neural network will become much cheaper. Additionally, the computation will be on the dedicated hardware and not on the CPU, decreasing CPU load. While this may seem unlikely, many types of specialized hardware (e.g. Trusted Platform Modules, Application-Specific Integrated Chips) have become more common.
- Solutions to input/output: In our project, we have been forced to work around the limitations of a neural network's fixed-size input and output. We have used a pad or prune approach to keep the input data a constant size, and have used a constant number of output nodes to indicate which input to choose. These approaches (especially the former) are not the best ways to use a neural network. Further research in this area might explore the use of

strategies such as recurrent neural networks or other ways to mitigate this weakness.

- Diverse and plentiful data: One of our difficulties with this project was that our data was limited, not only in size but in its ability to represent general use patterns. If future research was endowed with larger amounts of data that was representative of a wider variety of usage patterns, then the quality and quantity of the data would be able to justify a neural network-based approach.

VII. Future Work

Should we continue our research into neural network-based schedulers, we could pursue the following directions:

- More and better data: There are many scheduling algorithms, but we only chose three of them for implementation of our project. Therefore, we can choose a greater variety of popular scheduling algorithms to compare against our NN scheduler in the future. Also, as the quality and the quantity of training datasets has a significant impact on the NN model, we can also make our scheduler better by finding better datasets on which to train our model.
- Security and privacy of data: The security and the privacy of data is always a concerning problem, especially in DL fields where the training data is usually related with commercial secrets or individual privacy. As our research group specializes in cryptography, based on our DL and NN knowledge acquired this semester, we may move on to work on the problem of how to securely share training data in an efficient way.
- Real-world experimentation: For our project, we only prove the feasibility of neural network-based schedulers within a simulated environment. This approach is enough for a simple implementation, but would be interested in testing our idea in the real world by integrating our neural network-based scheduler with a Linux operating system.

References

[1] YA Adekunle et al. "A comparative study of scheduling algorithms for multiprogramming in real-time systems". In: (2014).

[2] Swati G. Anantwar and Rajeshri R. Shelke. "Simplified Approach of ANN: Strengths and Weakness". In: *International Journal of Engineering and Innovative Technology* 1.4 (), pp. 73–77.

[3] "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080.

[4] Josep Ll Berral et al. "Towards energy-aware scheduling in data centers using machine learning". In: *Proceedings of the 1st International Conference on energy-Efficient Computing and Networking*. ACM. 2010, pp. 215–224.

[5] Tom Carchrae and J Christopher Beck. "APPLYING MACHINE LEARNING TO LOW-KNOWLEDGE CONTROL OF OPTIMIZATION ALGORITHMS". In: *Computational Intelligence* 21.4 (2005), pp. 372–387.

[6] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. "SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms". In: *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2014, 6–p.

[7] Pedro Domingos. "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10 (2012), pp. 78–87.

[8] Paul Emberson, Roger Stafford, and Robert I Davis. "Techniques for the synthesis of multiprocessor tasksets". In: *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11.

[9] Jeremy P Erickson, James H Anderson, and Bryan C Ward. "Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling". In: *Real-Time Systems* 50.1 (2014), pp. 5–47.

[10] Aakash Goel and Gaurav Sharma. "Scheduling Challenges in Operating System". In: *Bilingual International Conference on Information Technology:Yesterday,Today,Tommorrow* (2015), pp. 179–182.

[11] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[12] Zehao Huang. *SimpleNeuralNetwork*. https://github.com/huangzehao/SimpleNeuralNetwork. 2015.

[13] *Inside the Linux scheduler*. June 2006. URL: https://www.ibm.com/developerworks/library/l-scheduler/index.html.

[14] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017, pp. 1–12.

[15] Atul Negi and Kishore Kumar Pusukuri. *Applying Machine Learning Techniques to Improve Linux Process Scheduling*. Dec. 2005.

[16] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press USA, 2015.

[17] Paolo Priore et al. "A review of machine learning in dynamic scheduling of flexible manufacturing systems". In: *AI EDAM* 15 (2001), pp. 251–263.

[18] K. Ramamritham and J. A. Stankovic. "Scheduling algorithms and operating systems support for real-time systems". In: *Proceedings of the IEEE* 82.1 (Jan. 1994), pp. 55–67. ISSN: 0018-9219. DOI: 10.1109/5.259426.

[19] Tariq Rashid. *Make your own neural network*. CreateSpace Independent Publishing Platform, 2016.

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), p. 533.

[21] S Shukla Shubhendu and Jaiswal Vijay. "Applicability of Artificial Intelligence in Different Fields of Life". In: *International Journal of Scientific Engineering and Research* ().