

UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Automatic Learning of Block Storage Access Time Models

Permalink

<https://escholarship.org/uc/item/9gs8x5n8>

Author

Crume, Adam

Publication Date

2015

Copyright Information

This work is made available under the terms of a Creative Commons Attribution-ShareAlike License, available at <https://creativecommons.org/licenses/by-sa/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

**AUTOMATIC LEARNING OF BLOCK STORAGE ACCESS TIME
MODELS**

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Adam Crume

September 2015

The Dissertation of Adam Crume
is approved:

Professor Carlos Maltzahn, Chair

Professor Scott Brandt

Professor Seshadhri Comandur

Dean Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by
Adam Crume
2015

Table of Contents

List of Figures	vii
List of Tables	ix
Abstract	x
Dedication	xi
Acknowledgments	xii
1 Introduction	1
1.1 Research questions	4
1.2 Contributions	5
1.3 Overview of this thesis	6
2 Related work	7
2.1 Predicting request latencies	7
2.2 Learning periodic functions	9
3 Storage devices	12
3.1 Scope of work	12
3.2 Challenges	12
3.2.1 State	13
3.2.2 Periodicity	15
3.2.3 Complex geometry	15
3.2.4 Request reordering	17
3.2.5 Caching	17
3.2.6 Nonlinearity and discontinuity	18
3.2.7 Nondeterminism	18
3.2.8 Other internal and external factors not addressed	20
3.3 Devices studied	20
3.4 Capturing response times	20

4	Latency modeling	22
4.1	Challenges	22
4.1.1	Lack of feedback	22
4.1.2	Low footprint	23
4.1.3	Low prediction latency	23
4.2	Quantifying response time accuracy	24
4.2.1	L_1 versus L_2 type error functions	24
4.2.2	Request-level accuracy	26
4.2.3	Trace-level accuracy	26
4.2.4	Window-based error functions	28
4.2.5	Selected error function	28
4.3	Quantifying scheduling accuracy	29
4.3.1	Per-request error functions	29
4.3.2	Permutation-based error functions	32
4.3.3	Selected error function	34
5	Inherent uncertainties	35
5.1	Nondeterminism	35
5.2	Seek versus rotation time	36
6	Common machine learning approaches	39
6.1	Models	41
6.1.1	Linear regression	42
6.1.2	Neural nets	42
6.1.3	Associative memory	43
6.1.4	Decision trees	44
6.1.5	Nearest neighbor	45
6.1.6	Support vector regression	45
6.1.7	Input-output hidden Markov models	46
6.1.8	Genetic programming	47
6.2	Optimization methods	47
6.2.1	Gradient descent (GD)	48
6.2.2	Stochastic gradient descent (SGD)	48
6.2.3	Conjugate gradient (CG)	49
6.2.4	Resilient propagation (Rprop)	49
6.2.5	RMSProp	50
6.2.6	Quickprop	50
6.2.7	L-BFGS	51
6.2.8	Levenberg-Marquardt (LM)	51
6.2.9	Simulated annealing	51
6.2.10	Genetic algorithms (GA)	51
6.3	Ensemble methods	52
6.3.1	Bagging	52

6.3.2	Boosting	52
7	Automatic learning	54
7.1	Neural nets	54
7.1.1	General configuration	55
7.1.2	Shared weights	56
7.1.3	Training	56
7.2	L_1 norm versus L_2 norm	57
7.3	Hyperparameter tuning	58
7.4	Results	62
7.4.1	Baseline data	62
7.4.2	Errors	62
7.4.3	Model creation time	63
8	Periodicity	68
8.1	Scalable discovery of periodicity	68
8.1.1	Useful frequencies	69
8.1.2	Finding strong frequencies	69
8.1.3	Scaling the search for strong frequencies	70
8.1.4	Input augmentation	72
8.1.5	Hyperparameters	72
8.2	Changes in periodicity	73
8.3	Period composition	75
8.4	Learning to compose periods	79
8.5	Period composition results	81
9	Discontinuities	83
9.1	Cause and appearance	83
9.2	Removal	83
9.3	Results	84
10	Future work	88
11	Conclusion	92
	Bibliography	95
A	Notation	108
B	Symbols	109
C	Glossary	111

D Diameter of the set of q-bounded length-n permutations	114
D.1 Background	114
D.2 Theorems	115
Index	118

List of Figures

2.1	Classically hard periodic problems	10
2.2	Predicting positioning times using a recurrent model	11
3.1	Response times after a seek	14
3.2	Response time for mostly-sequential reads	14
3.3	Serpentine layouts, platters viewed edge-on	16
3.4	Queueing in a storage device with no internal parallelism	18
3.5	Seek times from track 0 for the first 1000 tracks of the test hard disk drive	19
4.1	Demerit compares CDFs	27
5.1	Time from A to B plus B to A	37
5.2	Time from A to B plus B to A, one dimension	38
6.1	HMM	46
6.2	IOHMM	46
7.1	Network architecture when subnets are not used	55
7.2	Network architecture when subnets are used	57
7.3	Comparison of L_1 versus L_2 norm	58
7.4	Error versus generation	60
7.5	Configuration versus generation length	61
7.6	Error of the neural net over time while training	64
7.7	Access time over the first 4 tracks of a hard disk drive, in milliseconds .	67
8.1	Fourier spectrum for the first 237,631 sectors	71
8.2	Convergence with and without period penalty	73
8.3	Messy Fourier transform of function with changing periodicity	74
8.4	Block space is split into regions which are handled separately	74
8.5	Mapping from request blocks to sets of periods	76
8.6	Fourier transform misses complex periodicity	76
8.7	Non-integer frequencies	77
8.8	Neural net generating non-integer frequency	79
8.9	Predicted versus actual access times using period composition	82

9.1	A 2D sawtooth can be rolled up into a 3D helix to remove its discontinuities.	84
9.2	Predicted versus actual access time over the first 237,631 sectors of HD1, without and with discontinuities removed	85
9.3	Error versus sector for HD1	86
9.4	Predicted versus actual access times for the first half of HD1	87

List of Tables

7.1	Effect of minimizing L_1 versus L_2	58
7.2	Average errors for access time predictions	65
7.3	Average scheduling error	66

Abstract

Automatic Learning of Block Storage Access Time Models

by

Adam Crume

Performance models for storage devices are an important part of simulations of large-scale computing systems. Storage devices are traditionally modeled using discrete event simulation. However, this is expensive in terms of computation, memory, and configuration. Configuration alone can take months, and the model itself requires intimate knowledge of the internal layout of the device. The difficulty in white-box model creation has led to the current situation, where there are no current, precise models. Automatically learning device behavior is a much more desirable approach, requiring less expert knowledge, fewer assumptions, and less time. Other researchers have created behavioral models of storage device performance, but none have shown low per-request errors. By making use of only a few high-level domain-specific assumptions, such as spatial periodicity and the existence of a logical-to-physical mapping, neural nets can learn to predict access time with low per-request errors. Providing neural nets with specific sinusoidal inputs allows them to generate periodic output, which is necessary for this problem. Weight sharing in the neural net accounts for regularities in the structure of the input, which reduces data requirements and error by reducing the number of free parameters. A trigonometric change of variables in the output of the neural net removes a discontinuity in the objective function, which makes the problem more amenable to neural nets and reduces error. Combining these approaches, we demonstrate that a neural net can predict access times with a mean absolute error of about 0.187 ms over a small portion of a hard disk drive, and a mean absolute error of about 2.120 ms over half of a hard disk drive.

To Megan,

who stood by me, put up with me, and kept me sane.^[citation needed]

Acknowledgments

I would like to thank Scott Brandt and Carlos Maltzahn for allowing me to join their research in Storage Performance Modeling. Thanks also to Joe Buck, Noah Watkins, Dimitris Skourtis, Michael Sevilla, Ivo Jiminez, Andrew Shewmaker, Jeff LeFevre, and other students in the Systems Research Laboratory.

Thanks to Samy Bengio for providing an implementation of IOHMMs.

Thanks to the other researchers I have collaborated with, including (in alphabetical order) John Bent, Philip Carns, Christopher Carothers, Seshadhri Comandur, Carolyn Conner, Jason Cope, Matthew Curry, Stephen Eidenbenz, Gary Grider, David Helmbold, Curtis Janssen, Philip Kegelmeyer, Thomas Kroeger, Sam Lang, Ning Liu, Meghan McClelland, Patrick McCormick, Misbah Mubarak, James Nunez, Ron Oldfield, Robert Ross, Lee Ward, Manfred Warmuth, Patrick Widener, David Wolpert, and anyone else I may have left off this list.

A portion of this work was supported by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This work was also supported by the DATA Management in Scientific Computing project (DAMASC) and by the Institute for Scalable Scientific Data Management (ISSDM).

Chapter 1

Introduction

Storage device performance modeling is an integral part of modeling computing systems. These models are used for simulating supercomputers [77], for parallel file system simulation [78], for storage configuration [7], and for data placement [53]. The most accurate models are discrete event simulation-based models such as DiskSim [19], which is used by many people [22, 117, 78]. Discrete event simulation is a type of white-box modeling based on processing simulation events that modify state and/or generate new events. White-box models require intimate knowledge of the internal layout of the device, which can take months to extract. Automatically learning device behavior is a much more desirable approach, potentially requiring less expert knowledge, fewer assumptions, and less time. Others have created behavioral models of storage device performance, but none have shown low per-request errors. Barriers to machine learning of request latencies include periodicity, caching effects, stateful behavior, discontinuities, sparsity of samples, and many other factors. We propose a method for generating black-box models of block storage device performance using machine learning.

We chose the block interface as the level to model because of its simplicity, universality, and low-level aspect. Other storage interfaces are available, including OSD and POSIX. Since most storage devices (including hard disk drives, SSDs, and RAID arrays) expose a block interface, object storage and file systems are built on top of the block interface. These higher-level interfaces are more complex, offering a richer set of operations, and hide details about lower levels. We believe that modeling these higher-level interfaces would produce useful results, but that modeling them with machine

learning is likely to be more difficult than modeling the block interface. Furthermore, the higher-level interfaces are more transparent and thus more amenable to white-box modeling.

Modern storage devices are complex, meaning that white-box models of them are also complex. The simulator must know the storage device’s caching and scheduling algorithms, as well as the physical placement of each block. The very large number of parameters and the reluctance of manufacturers to provide implementation details make characterization a long, difficult, and error-prone process. In fact, configuring DiskSim to accurately model a modern hard disk drive took six months in one case [87]. Other researchers have also noted the difficulty of parameterizing DiskSim [86, 72].

Many researchers use the outdated disk models that come with DiskSim, presumably because this is easier than obtaining or creating models for newer disk drives. Hard disk drives commonly modeled with DiskSim include the Quantum Atlas 10K [26, 103, 71, 114], which was released in 1999 [71]; the Seagate Cheetah 9LP ST39102LW [103, 75, 108, 76], which appears to have been released in 1998 [84]; and the Seagate Barracuda 4LP ST32171 [117], which appears to have been released between 1996 and 1998 [9]. All of these papers used models which come with DiskSim, and these models were at least 11 years old when the papers were published. The capacities of the modeled devices were at most 9.2 GB, with sustained data transfer rates of at most 41 MB/s [1, 9, 84]. For comparison, consider top of the line models released in 2010, the year of the earliest publication above. The WD Caviar Green 3TB [31] and the Seagate 3TB FreeAgent GoFlex Desk [2], had capacities of around 3000 GB and sustained data transfer rates of around 123 MB/s [3] (sizes measured using $1 \text{ GB} = 10^9$ bytes). This means that the models used in the papers were at least a factor of 326 times smaller and 3 times slower than top of the line drives of the time. Using such out of date models likely hampers research relevance, but it is nearly inevitable when models are so hard to create.

Tools such as DIG can extract some of the necessary DiskSim parameters [46]. Unfortunately, they require assumptions about the internal structure of the storage device. This structure is highly proprietary and frequently changing. Examples of changes introduced include zoning, zero-latency reads, and serpentine layouts. Manufacturers

do not release this information, so researchers must reverse-engineer a device before modifying DIG and DiskSim to support the new layout. Furthermore, models are extremely different between storage device types, and new models must be created by hand whenever a new storage device type appears.

Because of the complexity of modern hard disk drives, there are no publicly available request-level models. This means that for drives on the market today, the only way to really know how a system will perform with those drives is to build it and test it. Furthermore, accurately reproducing today’s experiments in the future will be impossible once the hardware is gone, since models do not exist. Existing simulators, such as DiskSim, are so difficult to configure that comparing the accuracy of new methods to them is fallacious; the accuracy of a simulator is only relevant when paired with a correct parameterization.

Rather than white-box simulation, a more desirable approach is to use machine learning to generate models that can reproduce the behavior with as few assumptions as possible. Some progress has been made in *behavioral modeling* of storage device performance [59, 26, 109, 117], but none of the models described can accurately model individual requests. Optimizing layout of multidimensional data on disk [94], auto-tuning parallel I/O [10], and real-time scheduling [102] are use cases where request-level predictions would certainly be beneficial. Request-level accuracy in machine learning-based storage device performance models warrants further research.

Machine learning approaches can be divided into two classes, online and offline. With online approaches, models receive feedback from the system being modeled and continue to adapt. With offline approaches, models do not receive feedback, and therefore do not change after being created. Many of our use cases do not allow for feedback. Furthermore, simpler approaches have been shown to work in some online cases [97]. Therefore, we limit ourselves to offline machine learning.

Because of the challenges in machine learning of storage device performance models, preprocessing the data is an important step. In the preprocessing stage, we use high-level domain knowledge to represent the data in a fashion that is more amenable to machine learning. This knowledge includes the importance of phenomena such as periodicity and a single device-wide logical-to-physical mapping. Better representing

periodicity requires explicitly including periodic data as additional inputs (see chapter 8). Recognition of a single logical-to-physical mapping allows us to share part of the model across requests (see section 7.1.2). Embedding high-level assumptions into the model using periodic inputs and weight sharing is especially important because the amount of data that can be sampled is very sparse relative to the amount of complexity in the raw data.

In this thesis, we focus on a random, read-only workload. This emphasizes the effect of access time, and de-emphasizes other effects such as transfer time, caching, and read-ahead. Transfer time can be addressed by simply including a size input [109]. Caching can be captured by techniques such as including LRU stack distance [59]. Read-ahead can be addressed by including a sequentiality flag in the input [109]. Access time is a component of the overall latency prediction problem which has proven difficult to solve in the past, which is why we focus on that component. We demonstrate that with specific periodic inputs, and with customizations such as weight sharing and remapped output which removes discontinuities, we can train a neural net to predict access times over the majority of a modern hard disk drive.

1.1 Research questions

The general question is how to create a practical black-box access time model of a block storage device. This can be broken down into specific questions:

- What assumptions are necessary?
- What model is most appropriate for this problem?
- What data representation is best, or in machine learning parlance, what features are important?
- How should error be measured?
- How much data is necessary?
- Is the model fast enough?
- Is the model accurate enough?

1.2 Contributions

The overall contribution of this work is a method for automatically generating storage device request access time models for random, read-only workloads. Specific sub-contributions include:

- Selection of neural networks as a useful model. This applies to the question of which model is appropriate.
- Use of weight sharing to reduce the number of neural network parameters. This applies to the question of which model is appropriate.
- Method of input augmentation to allow the neural network to internally generate necessary frequencies in recognition of the importance of periodicity. This applies to the questions of representation and assumptions. We find that periodicity is a very useful assumption to make.
- Use of L_1 norm in recognition of non-Gaussian errors. This applies to the question of how error should be measured.
- Method for removing rotation-related discontinuities in the access time function. This applies to the questions of representation and assumptions.
- Method for tuning neural network hyperparameters. This applies to the question of which model is appropriate.
- Identification of 29.5 hours of trace data as sufficient for 2.120 ms mean absolute error over 250 GB of storage. This applies to the question of how much data is necessary.
- Identification of 9.5 ms as the real time necessary to generate an access time prediction over 250 GB of storage with a model not tuned for speed. This compares with the median device access time over the same range, which is 12.495 ms. This means that using the model is (slightly) faster than using the actual hard disk drive, which makes the model plausibly fast enough for some use cases, such as emulating a hard disk drive's performance with faster hardware. Since the model was not tuned for speed, the time per prediction can almost certainly be reduced.

- Identification of 2.120 ms mean absolute error over 250 GB of storage. Most of this error comes from mispredicting rotation misses. After adding or subtracting an integer multiple of the rotational latency to predictions to correct for this fact, the mean absolute error drops to 0.485 ms. This compares to a constant-value model, which has a mean absolute error of 2.792 ms over the same range. This applies to the question of whether the model is accurate enough.

1.3 Overview of this thesis

The rest of this thesis is organized as follows: In chapter 2 we discuss related work for storage device modeling and machine learning of periodic functions. In chapter 3 we describe our assumptions about storage devices as well as challenges in modeling them. In chapter 4 we discuss methods for quantifying prediction errors. In chapter 5 we discuss inherent uncertainties in modeling block storage device performance. In chapter 6 we compare machine learning approaches and discuss their relative merits. In chapter 7, after selecting neural nets as an appropriate machine learning approach, we propose a modification (weight sharing) to decrease error. In chapter 8, we describe our method for finding periodicity, which many machine learning approaches do not handle well, and feeding it into the machine learning algorithm. In chapter 9, we discuss the problem of modeling discontinuous functions with neural nets, and how the problem can be mitigated for hard disk drives. Future work is discussed in chapter 10, and we conclude in chapter 11.

Chapter 2

Related work

2.1 Predicting request latencies

Storage device latency models can be categorized in many ways. Perhaps the most basic is online versus offline models. Online models receive feedback from the actual storage device and can therefore be continuously updated. Offline models receive no such feedback and are not modified after being created. Another crucial dividing point is white-box versus black-box models. White-box models use information about the internal behavior of the device; this class includes analytical models and models based on discrete event simulation. Virtually all white-box models are offline models since they are constructed by humans. Black-box models look only at the input and output of the device; this class includes machine learning models such as decision trees, neural nets, support vector regression, etc. Models can also be characterized by the granularity of their input and output. Some take individual requests as inputs, and some take aggregate statistics over many requests. Some predict latencies for individual requests, and some predict average latencies over time slices or entire workloads. Even when predicting individual request latencies, the black-box models described below have only shown the predictions to be accurate in aggregate. None of the existing black-box approaches we have seen have shown low per-request errors.

DiskSim [19] is a well-regarded disk model based on discrete event simulation. It has been validated to produce request-level accuracy. However, it is computationally expensive and difficult to configure for modern disks. As detailed in chapter 1, the

difficulty is so great that researchers frequently use decade-old models rather than create new ones.

Many analytic models have been created, including work by Lebrecht, Dingle, and Knottenbelt [67], as well as work by Lingenfelter, Khurshudov, and Vlassarev [72]. Analytic models are relatively easy to understand and can be extremely fast due to their compact formulae. Unfortunately, they require very detailed expert knowledge to create. Often, they are limited to certain classes of workloads and are not useful alone in generalized contexts.

Kelly *et al.* describe a black-box probabilistic model [59] similar to table-based models such as the one by Garcia *et al.* [42]. Requests are categorized based on features including size, LRU stack distance, number of pending reads, number of pending writes, and some RAID-specific information. Oddly, they do not appear to include any sort of sequentiality feature. Table-based models are limited by the table size. If too many dimensions are used, or the granularity within dimensions is too fine, the table grows far too large. This limits the ability of the models to capture smooth transitions or complex cross-dimensional features. The work by Kelly *et al.* ameliorates the issue by essentially not requiring the entire table to be filled in, but the problem is not fully solved.

Mesnier *et al.* create a model for relative performance of storage devices [81]. This first requires workload characterization which may not be feasible ahead of time for a large-scale system simulation, especially if the workload is dynamic. Furthermore, given a relative model for device B compared to device A, simulating device B still requires a model for device A.

A popular approach is to use decision trees to predict response time. Dai *et al.* predict performance with a combination of decision trees and support vector regression [26]. However, their models are workload-specific, and their prediction errors are based on one-second averages rather than per-request latencies. Wang *et al.* also calculate errors based on windows, using one-minute windows in their case [109]. (See section 6.1.4 for more information on decision trees, and section 6.1.6 for more information on support vector regression.)

Erazo *et al.* created FileSim to simulate parallel file systems. They used a

simple disk model instead of DiskSim and were able to reproduce behavior of a real system [35]. Liu *et al.* modeled the performance of burst buffers for smoothing traffic to storage systems attached to supercomputers [77]. Tarasov *et al.* demonstrate how accurate models of hard disk drive latencies can be used to improve performance in a real-time scheduler [102]. All of these are applications which could benefit by including models generated by our approach.

Our approach is orthogonal to storage device emulators in that our generated models can provide high-fidelity timing models to emulators, such as those by Griffin *et al.* [48] and Agrawal *et al.* [5]. Griffin *et al.* implemented a storage device emulator using DiskSim as the underlying performance simulator. Agrawal *et al.* also implemented a storage device emulator, but with an analytical model for the simulator. Some storage device emulators do not alter performance to mimic any particular device and essentially map from one interface to another [88, 29]; these are unrelated.

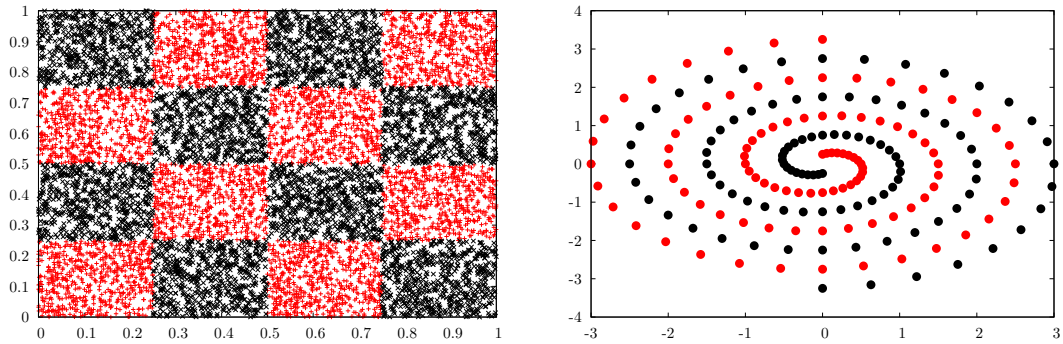
2.2 Learning periodic functions

We posit that periodicity in block number is an important yet high-level feature of storage device performance. In hard disk drives, periodicity is seen in the division of sectors into tracks, the skew between tracks, spare sectors, and so on. Periodicity is also seen in other storage devices, such as in the striping patterns of RAID arrays.

Unfortunately, a limitation of many general-purpose machine learning approaches, including neural nets and decision trees, is their difficulty in recognizing periodic patterns in the data. This can be seen with the checkerboard problem [100] (fig. 2.1a) as well as the two-spirals problem [37, 95] (fig. 2.1b).

Neural nets can be trained on periodic functions in many ways. Some setups predict the value of the function directly from the input. These invariably use a fixed interval with a very small number of oscillations [50, 49, 90, 38, 116]. Extrapolating beyond the training range leads to poor performance [62]. This approach is infeasible for our problem because the number of oscillations is on the order of a million.

If the function is known to have period p , another approach is to map the input x into the range $(0, p)$ using $x \bmod p$. Common examples include time-of-day or day-of-year inputs for functions that are daily or yearly periodic [64, 34]. An alternative



(a) Checkerboard problem: given X and Y coordinates, determine whether point is in a red or black square
 (b) Two-spirals problem: given X and Y coordinates, determine whether point is on the red or black spiral

Figure 2.1: Classically hard periodic problems

is to map it to $\sin(2\pi x/p)$ and $\cos(2\pi x/p)$ [43]. (This pair of functions has many nice properties; they are both continuous and bounded, and weighted sums are equal to other sinusoids with varying phase.) This is similar to our approach, except that we determine the period empirically, and we augment the original values with the transformed values rather than replace them.

Neurons in a neural net calculate their output by taking a weighted sum of the inputs and applying an *activation function* or *transfer function*, typically $\tanh(x)$ or $\frac{1}{1+e^{-x}}$ (logistic sigmoid). Rather than using one of these as the activation function, one may use $\sin(x)$. However, $\sin(x)$ does not approach a limit for large x , while $\tanh(x)$ and $\frac{1}{1+e^{-x}}$ do. Lack of a such a limit can lead to instability [113]. Periodic activation functions also introduce many local minima [99].

A powerful tool for time series data is recurrent or delay networks [85, 56]. These feed the network back into itself, so that the network predicts output values from other output values, rather than from input values. In other words, they predict $f(x)$ given $f(x-\delta)$ instead of predicting $f(x)$ given x . This style of prediction, where f values are predicted from other f values, we will call a *recurrent model*. Recurrent models also include Kalman filters [111] in addition to recurrent neural nets. Unfortunately, any recurrent model will require either enormous amounts of computation or an enormous training dataset to accurately generate predictions for storage device access times. Since

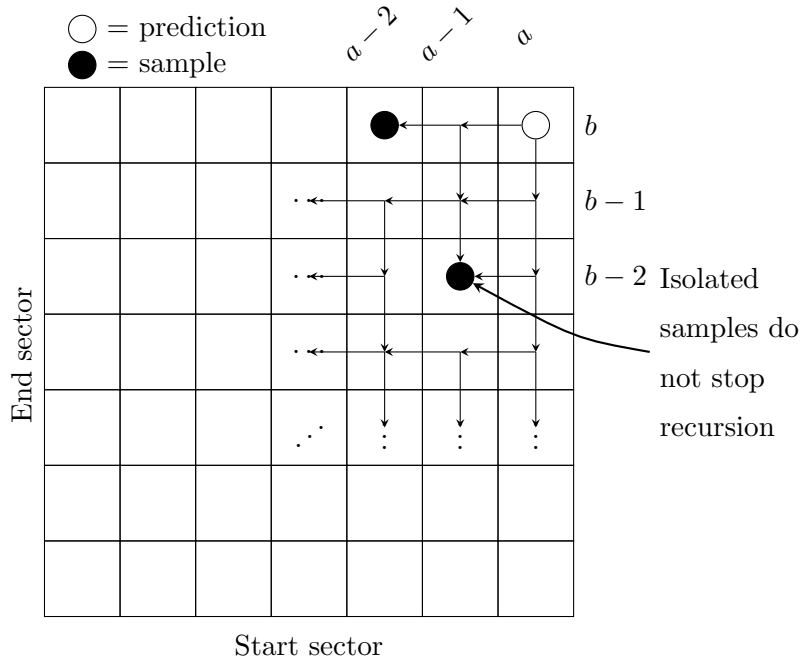


Figure 2.2: Predicting positioning times using a recurrent model. Sample sparsity leads to deep recursion.

we have a 2D problem, we would predict $f(a, b)$ from $f(a - \delta, b)$ and $f(a, b - \delta)$. Since we only have a very sparse sampling of f , we will almost certainly need to recursively compute $f(a - \delta, b)$ from $f(a - 2\delta, b)$ and $f(a - \delta, b - \delta)$, and compute $f(a, b - \delta)$ from $f(a - \delta, b - \delta)$ and $f(a, b - 2\delta)$, where $\delta = 1$ sector (fig. 2.2). Limiting the recursive computation to one million predictions would require sampling full rows and columns every thousand sectors. Given that the size of a modern 500 GB hard disk drive is approximately a billion sectors, this comes to two million rows and columns, each with a billion entries. If we assume an average access time of 10 ms, it would take $2 \times 10^6 \cdot 10^9 \cdot 10 \text{ ms} \approx 634,000$ years to capture the dataset. Thus, no recurrent model will be practical for predicting access times (when recursing in space).

Chapter 3

Storage devices

3.1 Scope of work

This thesis is limited to block storage devices, rather than other storage models such as object storage devices or file systems. Block storage devices work with *blocks*, which are fixed-size (traditionally 512 bytes) pieces of data numbered sequentially. Requests are sent to the storage device and responses are sent back. Multiple requests may be in flight at a time, and responses may be sent back out of order. Each request consists of a read/write flag, the start block, the block count, and, for writes, the data to be written. We assume that response times are independent of the actual contents written or read, so we ignore the contents (but not their size) in our models. Given a sequence of requests, our task is to predict the timestamps of their responses.

3.2 Challenges

Storage device performance modeling is a challenging problem. In this section we outline the major challenges that we believe are important to overcome. We intend to address these challenges except for those in section 3.2.8.

These challenges are phenomena that impact storage device performance and are difficult for many general-purpose machine learning approaches to handle “out of the box”. Challenges were selected based on three criteria. First, the phenomenon must have a large impact on storage device performance. Since we are not aiming

for perfection, phenomena with small impact are considered unimportant. Second, the phenomenon must occur relatively often in the environment of interest. Again, if it occurs rarely, it is considered unimportant. We simply accept that we will have high error on rare occasions. Third, the phenomenon must require preprocessing of the data, require reformulation of the problem, or otherwise cause large errors for multiple machine learning approaches.

These challenges arise from assumptions made by machine learning approaches that do not match well with storage device performance prediction. For example, most machine learning approaches assume that training examples are independent of each other. This assumption clashes with the existence of state, caching, and reordering. Most machine learning approaches also assume that the function to be learned is relatively smooth. This assumption clashes with the existence of complex geometry and discontinuities. Reformulation of the problem is necessary to align the problem with the assumptions made by the machine learning algorithm.

3.2.1 State

Storage devices are stateful, and requests are not independent. The model must accept a series of requests and predict a series of response times. Because of mechanical state, caching, read-ahead, garbage collection, etc., the time to complete a request is dependent on previous requests seen.

State changes can also persist for long periods. Figure 3.1 shows hard disk drive response times after a seek, after queueing effects are removed. This is a mostly sequential workload that occasionally seeks to a random position. Here, response times are affected for approximately 1100 requests after the seek. This same signature also occurs spuriously in the same trace (fig. 3.2). The cause is uncertain but may be due to remapped sectors. Whatever the cause, it is unpredictable, and this effect amplifies rare occurrences.

Many machine learning approaches do not natively support time series. The most straightforward way to model a stateful process with most machine learning approaches is to include past stimuli in the model's input. Unfortunately, the number of relevant past requests can number in the thousands or more, so many must be excluded.

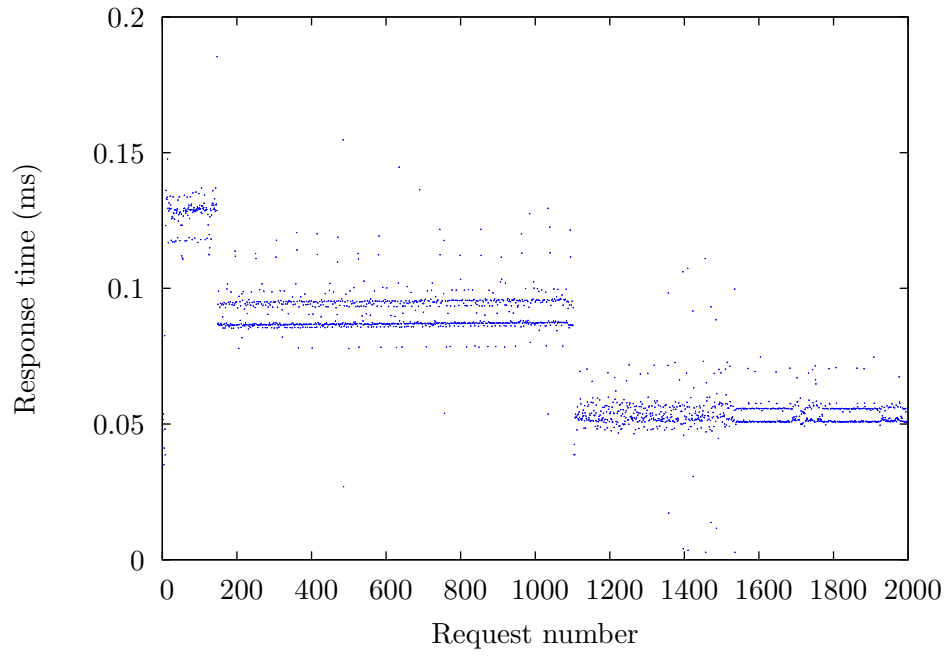


Figure 3.1: Response times after a seek

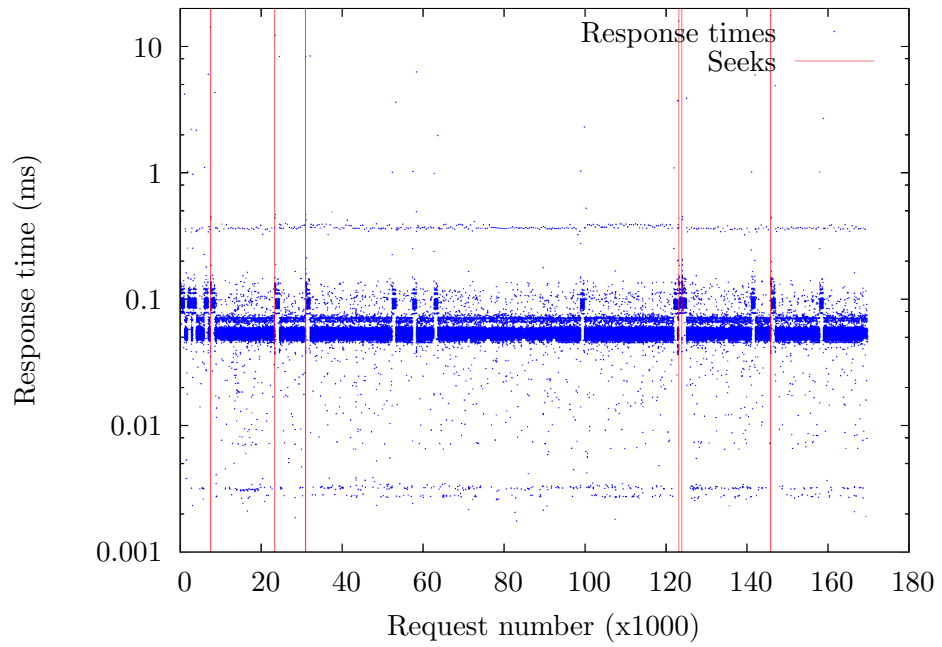


Figure 3.2: Response time for mostly-sequential reads

This is especially a problem for linear models, because adding nonlinear combinations of raw inputs leads to a combinatorial explosion.

3.2.2 Periodicity

Periodicity shows up in the behavior of many types of storage devices. In hard disk drives, periodicity comes from rotation. The rotational nature of hard disk drives leads to many periodic effects such as the arrangement of sectors into tracks, track skew, and so on. Hard disk drives are a classic example of storage devices with periodicity. In RAID arrays, periodicity comes from the regular striping of data across component devices. Requests for sectors that are a multiple of a stripe width apart will hit the same component device and contend with each other. In SSDs, although hidden by the flash translation layer (FTL), periodicity comes from the hierarchy of channels, packages, dies, planes, blocks, and pages [6, 32]. Groups of requests may have different degrees of parallelism based on how they are arranged in the hierarchy [58]. Periodicity is likely to occur in other device types as well, because of the benefits of regular repetition in designs.

As described in section 2.2, periodicity is challenging for general-purpose machine learning algorithms. If they do not recognize periodicity, each oscillation appears to add unique information about the function being learned, causing the necessary number of parameters to rise as the number of oscillations rises. Our test hard disk drive has roughly a million tracks and therefore roughly a million times a million oscillations in the access time function. Having on the order of a trillion parameters is clearly infeasible, so periodicity must be addressed.

3.2.3 Complex geometry

Storage devices often have complex internal geometry. Hard disk drives in particular have long ago moved away from a linear cylinder-head-sector mapping. Modern hard disk drives use zoning, which varies the number of sectors per track in each cylinder. Skewed layouts, in which the angular position of the first sector of each track varies across tracks, are employed to reduce rotational latency caused by misses when switching tracks. Many modern hard disk drives also use serpentine layouts, in which the

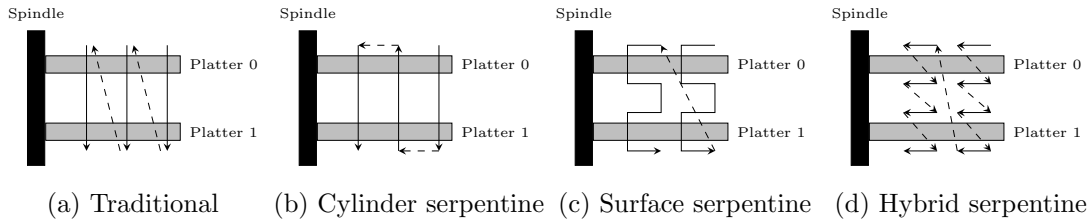


Figure 3.3: Serpentine layouts, platters viewed edge-on (adapted from fig. 3 in Gim and Won 2010)

logical tracks zig-zag inward and outward across the disk (see fig. 3.3). This is intended to reduce head switching. There are multiple variants, including cylinder, surface, and hybrid [46]. Geometrical details are not typically provided by the manufacturers and are likely to change as better layouts are found. Even worse, geometry varies from drive to drive even within a model line [63], meaning that perfect accuracy requires this extraction be done for the specific disk being modeled.

After modification, we were able to extract the layout of a drive using DIG [46]. This is very time-consuming, requiring days to complete. The results were also full of errors that had to be manually corrected using hopefully reasonable assumptions about the disk’s layout. These were most likely caused by fluctuations in timing causing DIG to report a false positive or negative about a track boundary. DIG’s MIMD algorithm, while necessary and useful, often multiplied these errors across tracks. We suspect that the final result still contains undetected errors.

Another tool, DIXtrac, should be faster and more accurate but does not support drives with serpentine layouts [93]. DIXtrac works by issuing SCSI commands to directly query the drive about the physical locations of sectors, and therefore does not support interfaces such as SATA.

In RAID arrays, the geometry may be viewed as the particular striping and parity strategies in use. For large arrays with many component drives, geometry also includes the arrangement of controllers, busses, etc. that may be shared among some but not all drives.

SSDs are internally divided into channels, packages, dies, planes, blocks, and pages [6, 32]. These allow for differing degrees and granularities of parallelism [58]. The

sizes and layouts of these structures define the SSD geometry.

3.2.4 Request reordering

Many storage devices support a request queue within the device, a feature called Native Command Queuing (NCQ) in SATA. Since this queue is held in the device, the requests can be reordered using details not readily available to the host. For hard disk drives, this includes the exact cylinder and angular position of the sectors, as well as the angular position of the platter. Devices may also use different scheduling algorithms to determine the ordering, so predicting the ordering is difficult even if the exact geometry is known. Reordering also means that a request's response time can depend on requests issued later. In other words, if request A is sent to the device, then request B, it is possible for A's time to depend on B's parameters and access time.

Request reordering is a large factor due to the amount of time a request spends in the queue. For a queue size of 32 (the size of the native command queue in our test device, and the maximum size of an NCQ queue), a sequential request on a loaded device will spend approximately 1/32 of its time being physically processed and 31/32 of its time in the queue. If the queue is heavily reordered (due to random accesses or bad sectors), the amount of time spent in the queue will vary wildly, causing response times to vary wildly. Random block accesses can be caused by fragmented files or metadata access, so even sequential file access may induce reordering.

Simple models may assume that there is only one queue and that only one request may be physically serviced at a time, as shown in fig. 3.4. This is mostly true for hard disk drives but is not true for RAID arrays and SSDs. This assumption would allow us to split the model into two parts, a request scheduler model and an access time model. These are then black box models on their respective subproblems. Unfortunately, this assumption is not valid in general.

3.2.5 Caching

Caching strategies can be very complex, vary between models, and are not exposed to the user. Beyond simply the cache size, caches may have an internal structure, as well as many possibilities for replacement policies and flush policies. By design,

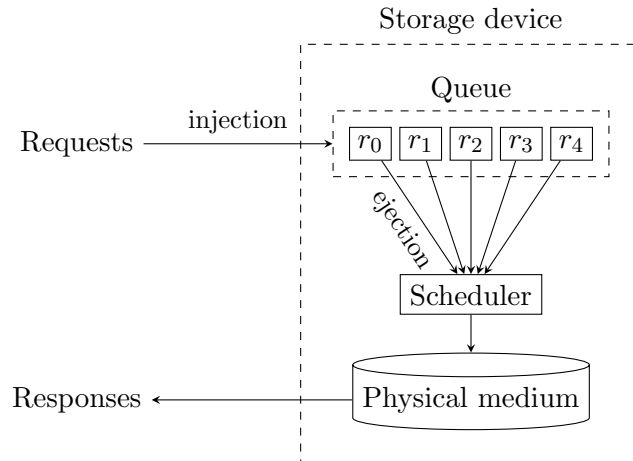


Figure 3.4: Queuing in a storage device with no internal parallelism

caches can have a significant impact on performance and therefore should not be ignored by a performance model.

3.2.6 Nonlinearity and discontinuity

Response time is highly nonlinear. Read-ahead, caching, reordering, etc. introduce step functions in the response time function. Access time is also highly nonlinear, partly due to the rotational nature of the drive (for hard disk drives). Rotational latency adds a sawtooth to the access time function which is extremely high frequency. As an example of nonlinearity, fig. 3.5 shows seek times for the first 1000 tracks of the test hard disk drive.

3.2.7 Nondeterminism

Response time is not entirely deterministic. Small variations should be expected, since this involves timing measurements of a physical process. Measurement error can creep in through scheduler noise on the CPU. Actual response times can vary as well, due to unrelated bus traffic and variations in seek time, settle time, and rotation time.

Rotation misses can amplify small variations into quite large ones. An arbitrarily small change in seek time can mean that a sector which could barely be read

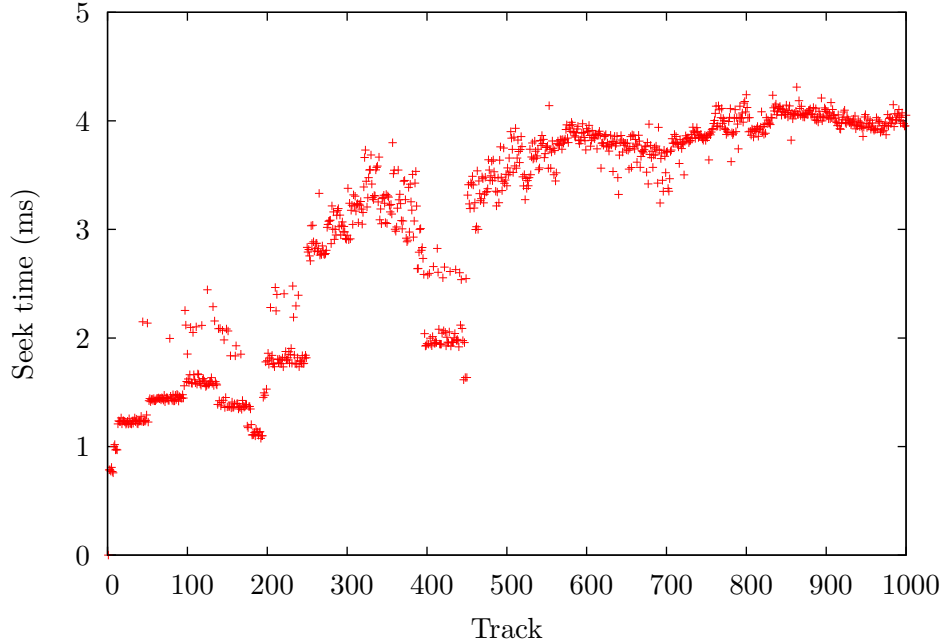


Figure 3.5: Seek times from track 0 for the first 1000 tracks of the test hard disk drive

during the current rotation has to wait for another rotation, adding a rotational latency to the response time. For our test devices, this means that response time can easily vary nondeterministically by 8.33 ms.

We measured the nondeterminism of our test devices by issuing identical pairs of requests repeatedly. We calculated the mean absolute deviation from the median (MAD-median) for the access time of the second request. This is equivalent to measuring the mean absolute error when using the median as the prediction. For HD1, we measured the MAD-median to be 0.142 ms, and for HD2 it is 0.024 ms. When rotation misses are excluded, these fall to 0.003 ms and 0.003 ms. For further details about HD1 and HD2, see section 3.3.

The nature of the nondeterminism affects the design and evaluation of the model. The non-Gaussian nature of the deviations affects our choice of error function (section 4.2), making mean square error less desirable. Nondeterminism also means that a predictor will have a non-zero lower bound on error, because even a perfect predictor cannot predict inherently nondeterministic fluctuations. For our selected error function

(section 4.2.5), this lower bound is given by the MAD-median above.

3.2.8 Other internal and external factors not addressed

Modern storage devices can be affected by factors difficult or impossible to account for. These include vibration [106, 92] and temperature [91]. Because these are difficult to measure and have little impact in normal operating environments, we consider their effects to be out of scope. Devices may also include power saving features [54]. Because our use cases focus on performance of heavy workloads, we consider power saving features to be out of scope.

3.3 Devices studied

We tested our approach against two hard disk drives. The first, which we call HD1, is a Western Digital Black WD5002AALX 500 GB 7200 RPM, with a total of 976,773,168 sectors. It has a cache size of 32 MB and NCQ queue length of 32. The second hard drive, HD2, is a Seagate ST31000340SV 1 TB 7200 RPM, with a total of 1,953,525,168 sectors. It also has a cache size of 32 MB and NCQ queue length of 32.

We originally studied HD1 in a Kingwin EZ-Dock external enclosure (model EZD-2535), connected via e-SATA. Due to predictability problems that seem to be caused by the enclosure or connector, we moved the drive to an internal bay.

In the final setup, both drives were installed internally and connected with SATA. The host runs 64-bit Ubuntu Linux, kernel version 2.6.35-28-server. It has an Intel Core i7 quad-core CPU (plus hyper-threading) running at 2.80 GHz and 12 GB of RAM.

3.4 Capturing response times

Block-level traces were usually captured using blktrace. The value that we define to be response time is the D2C or device-to-completion time. This is the time between the OS IO scheduler sending the request to the device driver and receiving a response from the device driver. Tracing was done below the OS IO scheduler because that is better understood and will likely be modeled separately. To the best of our

knowledge, there is no simple way to capture traces excluding time spent in the device driver, short of bus-level tracing. This requires special hardware setup, though, and the benefit likely to be gained was very small compared to the work required. Because of the speeds of modern CPUs compared to hard disk drives, the time spent in the device driver should be negligible compared to the time spent in the hard disk drive.

In some cases, especially for random read workloads against hard disk drives, traces were captured with a user space program. The program opens the device as a file in direct mode, and latency is defined as the latency of a `pread64` call. While slightly less accurate, this method makes it much easier to retrieve the data and is immune to kernel state issues with `blktrace` which occasionally require reboots. For random read workloads against hard disk drives, the latency of the hard disk drive is significantly greater than latency introduced by the user-space-level tracing, so the amount of noise in the data should be acceptable.

Before capturing data, the OS's scheduler was set to `noop` (FIFO), software read-ahead was disabled, and the read and write caches were flushed. This should ensure that the OS performs minimal processing of I/O requests so that observed patterns can be trusted to be caused by the storage device.

Chapter 4

Latency modeling

This chapter is about access time modeling from a machine learning point of view, specifically, challenges and how error should be measured.

Access time modeling is challenging for our use cases (see chapter 1) particularly because of lack of feedback, low memory constraints, and low latency constraints. These challenges shape our choices when designing a model.

Determining how to measure error is crucial in machine learning, because not only is it used to report results, but the choice of error function actually affects what model is produced. We chose mean absolute error as the error function for access times (section 4.2.1) and mean square error of 1-of- k encoding as the error function for scheduling (section 4.3.1.3).

4.1 Challenges

Besides the difficulties in modeling storage devices themselves, there are additional challenges posed by the environment in which our model would run.

4.1.1 Lack of feedback

When used as part of a larger simulation, the model would not have feedback from the storage device being modeled. This means that the model cannot correct for how long a request actually took, or know how accurate its predictions are. In machine learning terminology, we are limited to *offline* models, as opposed to *online* models,

which receive feedback.

When feedback is available, a linear model can be sufficient for many problems. The feedback corrects small deviations between the predicted and actual state, preventing state errors from building up. Without feedback, these errors accumulate, and more accurate models are needed to prevent the errors in the first place. Since the phenomena of interest are inherently nonlinear, linear models would yield large errors, so nonlinear models are necessary.

For quality of service applications, the model will be able to receive feedback, but it is not required to use that information. Making use of feedback when available is left for future work.

4.1.2 Low footprint

The model is almost certainly going to be used as part of a larger system, rather than by itself, so it is important to use as little memory and CPU as possible. This is especially true for quality of service applications, where the entire purpose is to make performance more reliable. This is also true for large system simulations, where a storage device model is just one small part of a large system and cannot use up all the simulation resources.

The model may also be used in parameter sweeps. These are when many simulations are run with slightly varying parameters to see how those parameters affect the result. Because the simulations are independent of each other, they may be run concurrently. The storage device model must not be so heavyweight that it prevents many simulations from being run concurrently.

4.1.3 Low prediction latency

The model must also generate predictions reasonably fast. If the model is 100% accurate but takes an hour to predict each individual latency, it is worthless. This is especially true for quality of service applications, where the time to predict a request latency must be much less than the request latency itself. The prediction time must also be less than the request latency when used in emulation. If the model is slower than the device it is emulating, then the emulated performance cannot match the performance

of the original device.

4.2 Quantifying response time accuracy

Given a sequence of actual values y_0, y_1, \dots, y_{N-1} and a sequence of predicted values $\hat{y}_0, \hat{y}_1, \dots, \hat{y}_{N-1}$, we need an *error function* to define the difference between the two sequences. Choice of error function during the training phase is crucial because different functions will lead to different models being generated. Typically, best results are seen when the training error function and test error function are the same.

Response times are obviously positive, since a response arrives after the corresponding request was sent, and they vary over several orders of magnitude. For our test hard disk drive, response time can be as low as 0.002 ms for a read cache hit and as high as 300 ms for a random read with a full queue. We want accuracy in both regimes, so a good error function will look at the ratio of predicted versus actual response times.

4.2.1 L_1 versus L_2 type error functions

An L_1 type error function is one that depends on the absolute value of the error, such as mean absolute error (MAE):

$$\frac{1}{N} \sum |y_i - \hat{y}_i| \quad (4.1, \text{MAE})$$

whereas an L_2 type error function is one that depends on the square of the error, such as mean square error (MSE):

$$\frac{1}{N} \sum (y_i - \hat{y}_i)^2 \quad (4.2, \text{MSE})$$

We use the terms “ L_1 type” and “ L_2 type” instead of “ L_1 ” and “ L_2 ” because the error functions we discuss do not necessarily exactly match the definition of the L_p norm, which is

$$\|v\|_p := \left(\sum_i |v_i|^p \right)^{1/p} \quad (4.3, L_p \text{ norm})$$

where $v = y - \hat{y}$ for an error function. Note that eq. (4.2, MSE) is equal to $\frac{1}{N} \|y - \hat{y}\|_2^2$. While $\frac{1}{N} \|y - \hat{y}\|_2^2$ and $\|y - \hat{y}\|_2$ have the same minima, they differ in other properties. One important difference is that the derivative of $\frac{1}{N} \|y - \hat{y}\|_2^2$ is separable in terms of the y_i s, while the derivative of $\|y - \hat{y}\|_2$ is not. This makes $\frac{1}{N} \|y - \hat{y}\|_2^2$ easier to optimize.

Many machine learning approaches have traditionally used an L_2 type error function, partly because it is easier to study analytically, and partly because its gradient is linear, making it simpler to minimize. However, it is sensitive to outliers. A single value that is very wrong can affect the error disproportionately. An L_1 type error function punishes outliers to a smaller extent, making it more robust.

Another way of looking at it is that an L_2 type error function yields models that predict the mean of possible values, given their uncertainty, whereas an L_1 type error function yields models that predict the median of possible values. To see this, consider

$$\arg \min_{\hat{y}} \sum_i (\hat{y} - y_i)^2 \quad (4.4)$$

To solve, we set the derivative to zero:

$$0 = \frac{\partial}{\partial \hat{y}} \sum_i (\hat{y} - y_i)^2 \quad (4.5)$$

$$= \sum_i 2(\hat{y} - y_i) \quad (4.6)$$

$$= 2n\hat{y} - 2 \sum_i y_i \quad (4.7)$$

$$\hat{y} = \frac{1}{n} \sum_i y_i \quad (4.8)$$

Or for L_1 , consider

$$\arg \min_{\hat{y}} \sum_i |\hat{y} - y_i| \quad (4.9)$$

Again, we set the derivative to zero:

$$0 = \frac{\partial}{\partial \hat{y}} \sum_i |\hat{y} - y_i| \quad (4.10)$$

$$= \sum_i \text{sgn}(\hat{y} - y_i) \quad (4.11)$$

$$= \sum_i \mathbb{1}_{\hat{y} > y_i} - \sum_i \mathbb{1}_{\hat{y} < y_i} \quad (4.12)$$

$$\sum_i \mathbb{1}_{\hat{y} < y_i} = \sum_i \mathbb{1}_{\hat{y} > y_i} \quad (4.13)$$

In other words, this is solved when equal numbers of y_i are less than and greater than \hat{y} , which means \hat{y} is the median.

4.2.2 Request-level accuracy

The default error function for most machine learning algorithms is mean square error (eq. (4.2, MSE)). This has many nice properties from a theoretical machine learning point of view, including convexity (for linear models) and a linear derivative. Convex functions are functions in which the value of every point in the interior of an interval is less than or equal to the value of a linear function connecting the endpoints of the interval. More intuitively, it means that the function is bowl-shaped and curves up far from the minimum. This error function is most applicable when the values can range over the entire real line. Unfortunately for our purposes, this is an L_2 -type error, which is sensitive to outliers (section 4.2.1). Since our data contains a lot of outliers, we want to avoid this error function.

A common definition of *relative error* is

$$\frac{1}{N} \sum \frac{|y_i - \hat{y}_i|}{y_i} \quad (4.14)$$

Unfortunately, this penalizes overestimation more than underestimation. For example, a prediction with $y = 1, \hat{y} = 10$ gives an error of 9, whereas $y = 10, \hat{y} = 1$ gives an error of 0.9. This asymmetry produces models that nearly always underestimate the true value.

Instead, we may predict the log of the response time instead of the response time. Let $y_i = \ln u_i$. Then

$$\begin{aligned} \frac{1}{N} \sum |y_i - \hat{y}_i| &= \frac{1}{N} \sum |\ln u_i - \ln \hat{u}_i| \\ &= \frac{1}{N} \sum \left| \ln \frac{u_i}{\hat{u}_i} \right| \end{aligned} \quad (4.15)$$

In other words, when using the mean absolute error function and predicting the log of the response time, we minimize the ratio of the actual and predicted response times. Furthermore, underestimation and overestimation are penalized by the same amount.

4.2.3 Trace-level accuracy

Assuming a loaded device, an intuitive error function is the relative difference in total time to complete a series of requests. Ignoring queueing and reordering, this is

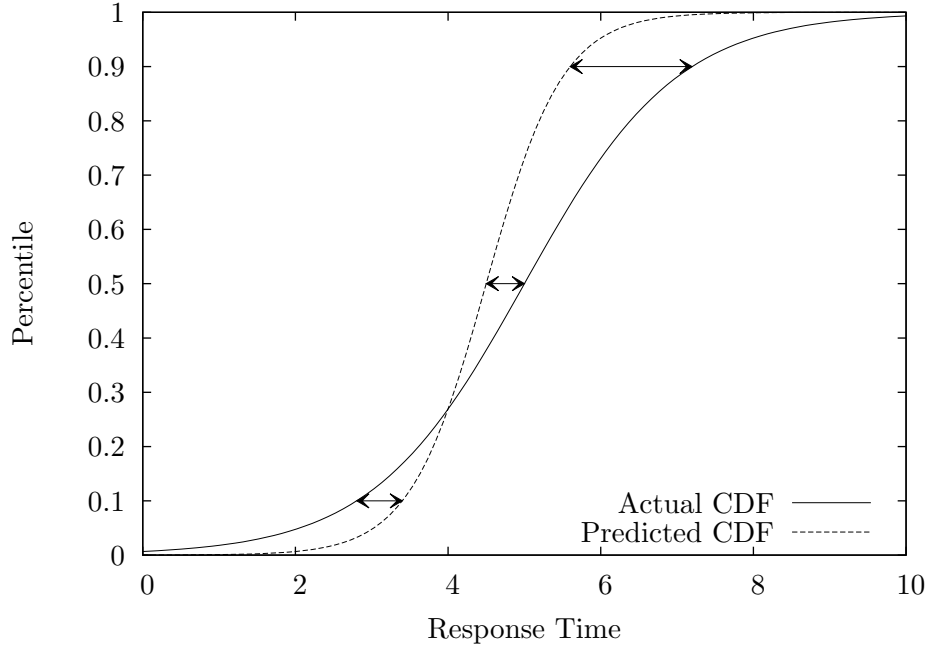


Figure 4.1: Demerit compares CDFs

roughly equivalent to

$$\frac{\max(\sum y, \sum \hat{y})}{\min(\sum y, \sum \hat{y})} - 1 \quad (4.16)$$

Demerit is a more complex error function that is useful when the distribution of response times is important, such as when there are downstream queueing effects. Demerit (as defined by Ruemmler [91]) is the root mean square horizontal distance between the cumulative distribution functions (CDFs) of the actual times and simulated times (see fig. 4.1). More precisely:

$$demerit = \sqrt{\int_0^1 (f^{-1}(y) - g^{-1}(y))^2 dy} \quad (4.17)$$

where f is the CDF of actual response times and g is the CDF of simulated response times.

Of course, any trace-level error function is very coarse-grained. A specific deficit is that trace-level error functions are unable to capture how closely the simulation tracks changing workloads.

4.2.4 Window-based error functions

A compromise between request-level and trace-level error functions is to use window-based error functions, which allow for an adjustable degree of granularity. An example error function for a window size w would be

$$\frac{w}{N} \sum_{i=0}^{N/w-1} \left(\frac{\max(s_i, \hat{s}_i)}{\min(s_i, \hat{s}_i)} - 1 \right) \quad (4.18)$$

where

$$s_i = \sum_{j=0}^{w-1} y_{wi+j} \quad (4.19)$$

$$\hat{s}_i = \sum_{j=0}^{w-1} \hat{y}_{wi+j} \quad (4.20)$$

Note that when $w = N$, this is equivalent to eq. (4.16).

Very small window sizes are similar to request-level accuracy and will reflect a model's ability to reproduce fine-grained detail. Small windows also provide more instances (training examples), since the number of windows is larger, which helps prevent overfitting. Very large window sizes are similar to trace-level accuracy and are appropriate for models which are unable to reproduce fine-grained detail, when fine-grained data is noisy, or when long-term biases are important. If the model is not able to capture long-term state, windows should probably be at least 1000 requests wide, if not wider, to capture stateful behavior mentioned in section 3.2.1. If sliding windows are not used, the width should probably be several multiples of this to minimize the effect of states starting in one window and ending in the next.

4.2.5 Selected error function

Because of our goal of request-level accuracy, and because of our current focus on fixed-size random reads, our current error function is mean absolute error:

$$\frac{1}{N} \sum_i |\hat{y}_i - y_i| \quad (4.21, \text{MAE})$$

For more varied workloads, with a larger range in access times, a more appropriate error function would likely be the mean absolute error of the logarithm of the

access time:

$$\frac{1}{N} \sum_i |\log \hat{y}_i - \log y_i| \tag{4.22}$$

but since this thesis uses only random reads, we use eq. (4.21, MAE).

4.3 Quantifying scheduling accuracy

For a random-read workload, access time alone is insufficient to predict the total latency. If multiple requests are outstanding, devices can schedule the requests in a non-FIFO fashion. Predicting scheduling order is also necessary to predict the total latency. Note that this thesis is focused solely on access time; scheduling is considered as a factor relevant for future work.

We want to find a request scheduler that mimics the scheduler in a real device as closely as possible. The input to a scheduler is a sequence of n requests, $R = \{r_0, r_1, r_2, \dots, r_{N-1}\}$, and the output is a permuted sequence, $R' = \{r'_0, r'_1, r'_2, \dots, r'_{N-1}\}$. The scheduler has a fixed, finite maximum queue length Q , meaning that at any time, at most Q requests have been issued for which responses have not been received.

A simple storage device with one queue and one physical medium is shown in fig. 3.4. We define an *injection* event to be the time a request is sent to the device and consequently inserted into the queue. An *ejection* event is when the scheduler removes a request from the queue and sends it to the physical medium. Since we cannot measure this directly, we define the ejection time to be the time the response is received for the previous request, or for an idle drive, the ejection time is the same as the injection time.

To quantify a scheduler’s accuracy, we need an error function, a few of which are outlined here. In this section, all indexing is 0-based.

4.3.1 Per-request error functions

For these error functions, we assume that the actual ejected request is the request ejected from both the device and simulator queues. In other words, the contents of the device and simulator queues are always the same (ignoring order). Therefore, we can refer to the set of requests in the queue without ambiguity.

A per-request error function computes, for each ejection, some notion of distance between the actual ejected request and the predicted ejected request. On ejection ($eject_t$), let a_t be the actual ejected request, p_t be the predicted ejected request, and q_t be the set of requests in the queue. The error function defines a per-request distance $d(a_t, p_t, q_t)$ satisfying $0 \leq d(a_t, p_t, q_t) \leq Q$, and the total error is given by the mean of d normalized by the queue length:

$$error = \frac{1}{N} \sum_{eject_t} \frac{d(a_t, p_t, q_t)}{|q_t|} \quad (4.23)$$

satisfying $0 \leq error \leq 1$.

4.3.1.1 Mistake count

The distance is 0 if the prediction is correct, and $|q|$ otherwise. This means that $error$ is simply the number of mistakes divided by the number of predictions. This is simple, but not ideal, because every mistake is given equal weight.

4.3.1.2 Scheduler rank distance

The scheduler predicts ejections by applying a function $w : Request \rightarrow \mathbb{R}$ which takes a request in the queue (and implicit context such as the previous ejected request) and assigns a weight. The queued request with the highest weight is the predicted ejection. The distance is the number of requests with higher weight than a_t .

$$d(a, p, q) = |\{r \in q : r \neq a, w(r) \geq w(a)\}| \quad (4.24)$$

If the error is non-zero, it tells us how “close” we were to being right; if the error is 1, our second guess was correct, if the error is 2, our third guess was correct, etc. While this error function may be interesting for training, it is less useful for evaluation. In the end, it does not matter if the scheduler was close to predicting the correct answer; only the actual prediction is relevant.

4.3.1.3 Mean square error of 1-of- k encoding

Similar to scheduler rank distance, the scheduler predicts ejections by applying a function $w : Request \rightarrow \mathbb{R}$ which takes a request in the queue (and implicit context

such as the previous ejected request) and assigns a weight. The queued request with the highest weight is the predicted ejection. The distance is the mean square error of w compared to the 1-of- k encoding.

$$d(a, p, q) = \frac{1}{|q|} \sum_{r \in q} (w(r) - \mathbb{1}_{r=a})^2 \quad (4.25)$$

This is a poor error function for determining how well the scheduler performs, because identical ejection sequences can have different errors, but it is useful for training because it is differentiable. We discuss this further in section 4.3.3.

4.3.1.4 Injection distance

The distance is the number of requests in the queue injected between a_t and p_t , plus one if $a_t \neq p_t$. Let $a \prec_{inject} b$ mean that request a was injected before request b .

$$d(a, p, q) = \begin{cases} |\{r \in q : a \preceq_{inject} r \prec_{inject} p\}| & a \prec_{inject} p \\ |\{r \in q : p \preceq_{inject} r \prec_{inject} a\}| & p \prec_{inject} a \\ 0 & a = p \end{cases} \quad (4.26)$$

This error function is probably of little use, because injection order is largely irrelevant. For example, consider the request sequence

$$a, b, c, d, e, f, g \quad (4.27)$$

and the response sequences

$$predicted = \mathbf{a}, \mathbf{g}, b, c, d, e, f \quad (4.28)$$

$$actual = \mathbf{g}, \mathbf{a}, b, c, d, e, f \quad (4.29)$$

the injection distance error function would report a large error, because a and g are far apart in the request stream, even though the response sequences are actually very similar.

4.3.1.5 Predicted ejection distance

The predicted ejection order is not defined at any particular time step, because the scheduler only predicts the next ejection, not the ones that follow it (although, see

section 4.3.1.2). Furthermore, because the actual ejected request is the one removed from the queue, the scheduler may predict the same request multiple times, or not at all. This means that a global predicted ejection order is not defined.

If it was defined, a non-zero error would mean that the actual ejected request was ejected sooner than predicted, and the error measures how much. The predicted ejected request is ejected early, and the error function doesn't tell us how much, but we could infer it from the actual ejection order.

4.3.1.6 Actual ejection distance

The distance is the number of requests in the queue that will be ejected after a_t and before p_t , plus one if $a_t \neq p_t$. Let $a \prec_{eject} b$ mean that request a is ejected before request b . Note that $a_t \preceq_{eject} p_t$.

$$d(a, p, q) = |\{r \in q : a \preceq_{eject} r \prec_{eject} p\}| \quad (4.30)$$

This is the most useful per-request error function. It is computable (unlike predicted ejection distance), utilizes available relevant information (unlike mistake count), and ignores irrelevant information (unlike injection distance and scheduler rank distance).

4.3.2 Permutation-based error functions

A permutation-based error function compares the full actual order output by the device and the full predicted order output by the simulator. Simulator error is compounded, and the two outputs of size N are compared.

In this section, a permutation of length N is a sequence of the numbers 0 through $N - 1$ (inclusive), with each number occurring exactly once. The identity permutation 1_S is simply the numbers in order ($\{0, 1, 2, \dots, N - 1\}$).

The sections below will define an error function $M(\tau)$ which measures the distance from the permutation τ to 1_S . We require $M(\tau) = M(\tau^{-1})$, $M(1_S) = 0$, and $M(\tau) > 0$ for $\tau \neq 1_S$. Let \hat{R}' be the predicted sequence of responses. Let σ be the permutation from R to R' , and let $\hat{\sigma}$ be the permutation from R to \hat{R}' . The error is then

$$error = M(\sigma^{-1}\hat{\sigma}) \quad (4.31)$$

Note that when $\hat{R}' = R'$, then $\hat{\sigma} = \sigma$, so $error = M(\sigma^{-1}\sigma) = M(1_S) = 0$, and the error is otherwise positive.

Our set of possible permutations P (which is limited by the queue size q) is known as the set of q -bounded permutations. For our values of q and N , P is much smaller than the entire set of permutations. Unfortunately, P is not closed under composition or inversion.

4.3.2.1 Inversion count

The inversion count of a permutation σ is the number of out-of-order pairs:

$$I(\sigma) = |\{(i, j) : i < j, \sigma_i > \sigma_j\}| \quad (4.32)$$

We let $M = I$. This is also known as the unscaled or unnormalized Kendall's tau. The maximum error for a queue of size q is

$$max_error = \begin{cases} \frac{1}{2}N(N-1) & N \leq 2q \\ (2N - 2q + 1)(q - 1) & N > 2q \end{cases} \quad (4.33)$$

so we can calculate percent error if needed. (For proof, see appendix D).

The inversion count tells us how many responses are out of order between the actual and predicted outputs; how many pairs of requests (a, b) such that a comes before b in the actual order, but b comes before a in the predicted order (or vice-versa). For example, given

$$R' = \{1, 2, 0\} \quad (4.34)$$

and

$$\hat{R}' = \{1, 0, 2\} \quad (4.35)$$

the inversion count is 1 because the pair $(0, 2)$ is inverted, *i.e.* 2 comes before 0 in the actual order, but 0 comes before 2 in the predicted order.

4.3.2.2 Spearman's footrule

Spearman's footrule measures how far each element is moved from its original position:

$$F(\sigma) = \sum_i |i - \sigma_i| \quad (4.36)$$

and we let $M = F$.

This tells us how far apart each response is in the actual and predicted outputs; if a particular response is at position 3 in one output, and it is at position 4 in the other output, then its contribution to the error is 1. For example, given

$$R' = \{1, 2, 0\} \tag{4.37}$$

and

$$\hat{R}' = \{1, 0, 2\} \tag{4.38}$$

the footrule is 2 because 0 moved by one slot and 2 moved by one slot.

Interestingly, the Diaconis-Graham (DG) inequality states that Spearman's Footrule and the inversion count are within a constant factor of each other [30]:

$$I(\tau) \leq F(\tau) \leq 2I(\tau) \tag{4.39}$$

Because of this similarity, neither has much intrinsic benefit over the other unless one has a specific application in mind. However, since we have proof of the upper bound for the inversion count, which makes it easier to interpret, we use the inversion count as the preferred permutation-based measure of scheduler error.

4.3.3 Selected error function

Training and testing with the same error function virtually always results in the lowest error. In other words, training with error function A and testing with error function A will almost certainly give lower error than training with error function B and testing with error function A. When the simulator is actually used, the error will compound, since our assumption of offline machine learning means that there is no source of correction. Testing is therefore best done with a permutation-based error function such as inversion count. Unfortunately, training a neural net (described in sections 6.1.2 and 7.1) with a permutation-based error function and a gradient-based optimization algorithm is extremely difficult due to the lack of separability and impossible due to the fact that the gradient is zero everywhere it is defined. Since we use a RMSprop for training, which is a gradient-based optimization algorithm, it is an absolute requirement that the error function have a useful gradient. Because of this, training is performed using mean square error of the 1-of- k encoding described in section 4.3.1.3.

Chapter 5

Inherent uncertainties

Due to data capture limitations, and the physical nature of devices, any model is limited in how accurately it can reproduce storage device performance. Here we discuss some of those limitations and sources of uncertainty as well as the lower bound they imply on model accuracy.

5.1 Nondeterminism

Since hardware is not entirely deterministic, we measured the amount of nondeterminism in the hard disk drives. Request pairs were submitted repeatedly and each access time measured. Each pair was submitted and measured 25 times. The median of those 25 measured access times was taken to be the “true” value of the access time function for that pair of sectors. We then computed the mean absolute error of all the measurements compared with the median, treating the measurements as if they were predictions. (In statistics, this is known as the “sample mean absolute deviation from the median,” or sample MAD-median.) This error was computed for many independent sectors pairs and averaged.

More formally, for each of $N = 100,000$ random sector pairs (a_i, b_i) , $K = 25$ samples s_{ik} were taken. To capture sample s_{ik} , a read is issued for sector a_i , then a read is issued for sector b_i . s_{ik} is then defined to be equal to the completion time for b_i

minus the completion time for a_i . The MAD-median is

$$m_i = \frac{1}{K} \sum_k |s_{ik} - \text{median}(s_{i1}, s_{i2}, \dots, s_{iK})| \quad (5.1)$$

and this is averaged over all generated sector pairs:

$$\text{average MAD-median} = \frac{1}{N} \sum_i m_i \quad (5.2)$$

$$= \frac{1}{NK} \sum_{ik} |s_{ik} - \text{median}(s_{i1}, s_{i2}, \dots, s_{iK})| \quad (5.3)$$

After sampling from the first 237,631 sectors of HD1, the mean absolute deviation from the median is 0.048 ms. This means that the lowest mean absolute error achievable over this range is about 0.048 ms.

After sampling from the entirety of HD1, the mean absolute deviation from the median is 0.147 ms. This means that the lowest mean absolute error achievable is about 0.147 ms. When adjusting for rotation misses, the mean absolute deviation from the median is 0.003 ms. This shows that the vast majority of nondeterminism comes from the number of rotations not being fixed.

After sampling from the entirety of HD2, the mean absolute deviation from the median is 0.029 ms. When adjusting for rotation misses, the mean absolute deviation from the median is 0.008 ms. This shows again that the vast majority of nondeterminism comes from the number of rotations not being fixed, although the ratio is lower than for HD1.

5.2 Seek versus rotation time

We also looked at the sum of the access time of reading sector B after sector A, plus reading sector A after sector B (fig. 5.1). In other words, we read sector A, then B, then A, and the value is the time difference between the two A responses. This has the intriguing property of always being an integer multiple of the rotational latency of the drive. (Unfortunately, predicting exactly which integer appears to be no simpler than the prediction problem in general.) To better understand this, ignore B and consider A only. Obviously, an integer number of rotations occurs between two reads of A, so the time is an integer multiple of the rotational latency. Incorporating this knowledge into

the structure of the model could either reduce error or reduce the amount of training data required, at the expense of making further assumptions about the storage device.

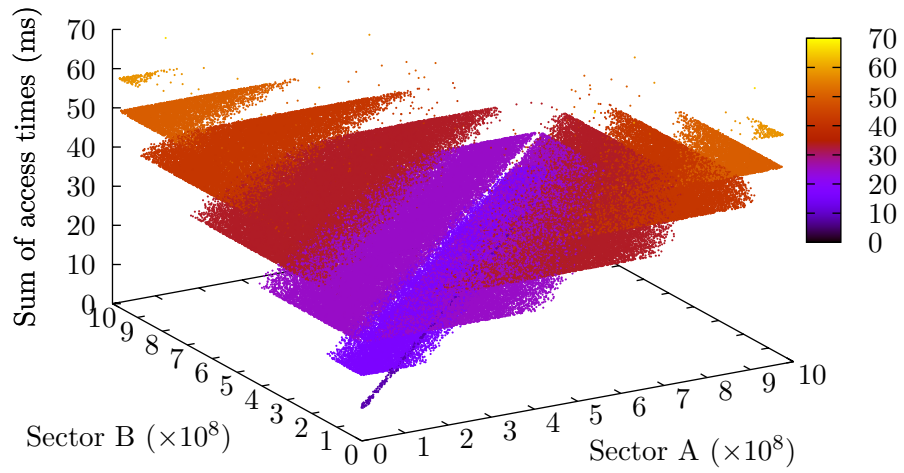


Figure 5.1: Time to read sector B after sector A, plus the time to read sector A after sector B. Note that the total is an integer multiple of the rotational latency, 8.33 ms. Also note that the segments overlap.

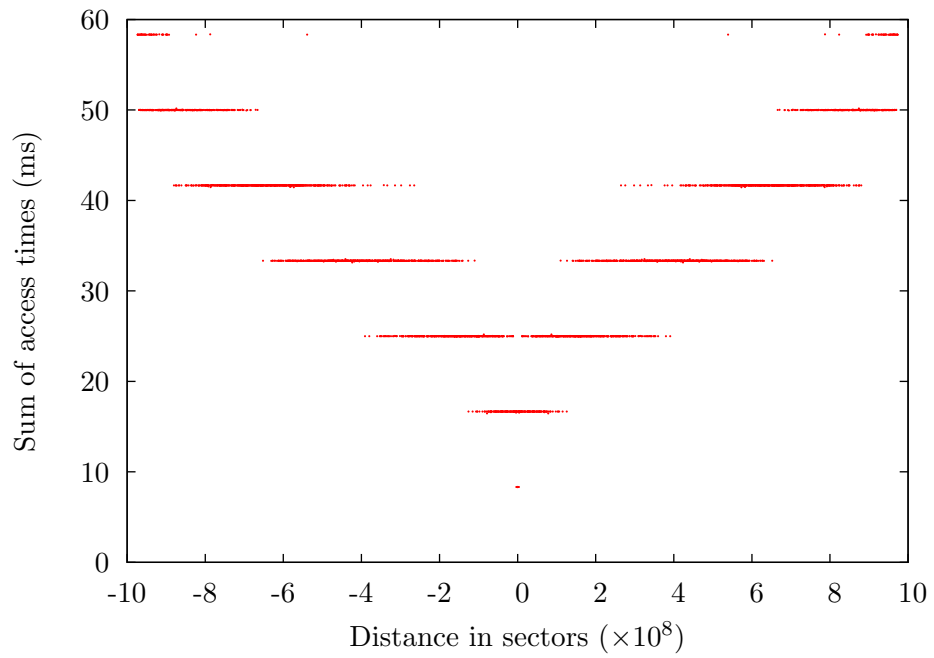


Figure 5.2: One-dimensional slice through fig. 5.1. Note that, ignoring fuzzy edges, any x -coordinate intersects exactly two lines.

Chapter 6

Common machine learning approaches

Because we want to automatically generate models by computer, we turned to machine learning. Our goal is an application of machine learning, not an advancement of machine learning itself, so we wanted to avoid creating any new machine learning approaches and instead rely on existing approaches.

Machine learning problems generally break down into choosing a data representation, model, error function, and optimization method. Data representation is how real-world observations are translated into numbers which can be manipulated by computer. A model (more formally, a hypothesis space) describes describes how pieces of data relate to each other, and specifically in the case of supervised machine learning (described below), how the output relates to the input. Choice of model affects what types of relationships are easily learned from the data. Error functions determine which particular models are better than other models. Optimization methods are used to choose a particular model from a set of models by minimizing the error function. We discussed error functions in section 4.2, and non-trivial aspects of data representation will be discussed in chapter 8. That leaves the choice of model and optimization method, which are discussed in this chapter. Ensemble methods are a special category which are difficult to split into model and optimization method, so they are discussed separately.

Our problem is naturally framed in terms of *supervised* machine learning. In supervised machine learning, the model has access to sample input, output pairs. The model is trained so its output matches the desired output as closely as possible. We evaluated several models and optimization methods to see which best fit our needs.

Our most important criteria are prediction speed, memory footprint, ability to use a sparse data set, and accuracy. Training speed is less important, as long as it is reasonable. Data can be captured at roughly 100 data points per second (depending on workload details), so models are not limited by extremely small data set sizes. However, since there are roughly 10^{18} data points which could potentially be captured, which would take roughly half a billion years to capture, the model must be able to use an extremely sparse sampling. Increased data set sizes also increase training time in addition to data capture time. Since we are producing black-box models, we are not interested in whether or not the approach produces models that are interpretable by humans. In most of our use cases, a reference device is not available at runtime, so only offline machine learning approaches are applicable.

A few properties of the access time function make machine learning difficult. For one, the access time function has a large number of discontinuities. For example, there are roughly half a million uniformly distributed values where increasing the start sector of the current request by 1 will change the output by a large amount. Also, the access time function has periodic or quasi-periodic components. If the machine learning algorithm does not support periodic functions, this makes the access time function appear to be much more information rich than it really is. A successful machine learning approach must be able to handle these difficulties, either directly or through preprocessing the data.

In general, models can be trained with multiple optimization methods, and optimization methods can be applicable to multiple models. There are constraints, though, that prevent all possible combinations from being meaningful or useful, and we specify which combinations are commonly used. In some cases, a model has a single commonly-used optimization method, or an optimization method is commonly used for only a single model, and the distinction between model and optimization method can be a little unclear. This is especially true for non-parametric models (defined in section 6.1).

We chose neural nets as the most appropriate model. They are expressive, interpolate smoothly, are relatively fast to evaluate, straightforward to train, and allow us to embed the problem structure into the network. (See section 7.1.2 for details on

subnets and weight sharing.)

6.1 Models

Models fall roughly into two categories: *parametric* and *non-parametric*.

A parametric model consists of a set of parameters θ in a parameter space $\Theta \subset \mathbb{R}^d$ that parameterize a function f_θ such that $f_\theta(X_n) = \hat{Y}_n \approx Y_n$. Parametric models can be trained by defining an associated *error function* $J(\hat{Y}_n, Y_n) : \mathbb{R}^L \times \mathbb{R}^L \rightarrow \mathbb{R}$ that is small when \hat{Y}_n is near Y_n and large when they are very different. (In general, J could be defined on all instances simultaneously, *i.e.* on \hat{Y} and Y , but we assume here that it is separable.) This error function is then minimized, which finds a set of parameters θ that makes the predicted values \hat{Y}_n as close as possible to the actual values Y_n . More formally, training solves the problem

$$\theta^* = \arg \min_{\theta \in \Theta} \sum_n J(f_\theta(X_n), Y_n) \quad (6.1)$$

and the trained model is f_{θ^*} . Methods of solving this are discussed in section 6.2.

Non-parametric is a misnomer, as it does not mean that the model has no parameters, but rather that the number of parameters is not fixed. In other words, there is still a notion that $\theta \in \Theta$, but there is no finite d independent of the number of training examples such that $\Theta \subset \mathbb{R}^d$. The number of parameters grows without bound, in some cases even requiring a parameter per input variable per example. In the limit, some models essentially store the entire set of training data in their parameters.

Some models can be either parametric or non-parametric depending on the optimization method, such as neural nets. Multilayer perceptrons, a traditional formulation of neural nets, have a fixed number of neurons and connections and are therefore parametric. However, more generic neural nets can be trained using algorithms that allow adding new neurons and connections as training goes on, making the model non-parametric. Models may also have a varying number of parameters, but with a fixed upper bound, making them somewhere between parametric and non-parametric.

The distinction between parametric and non-parametric models is important because many optimization methods are only applicable to models with a fixed number of parameters, and particularly to models where $\partial J / \partial \theta$ is defined and easy to calculate.

6.1.1 Linear regression

Given a feature vector x , a linear regression model uses a learned matrix A and vector b to predict

$$\hat{y} = Ax + b \tag{6.2}$$

Linear models are fast to train, fast to execute, and easy to understand. However, they are not able to capture interactions between features or nonlinear behavior of a single feature. This can be partially addressed by adding nonlinear functions of the original features as new features. Unfortunately, this quickly leads to an explosion in the number of features, which leads to a larger memory footprint and slower prediction speed.

When feedback is available, and when the function to predict is relatively smooth, linear models are attractive. Their simplicity and speed are very advantageous, and feedback prevents large errors from building up. QBox is an example of linear latency models with feedback being used successfully for performance isolation [97].

Linear regression is parametric.

6.1.2 Neural nets

Artificial Neural Nets (ANNs) are inspired by simplified biological processes [73]. Neural nets consist of relatively simple neurons, each of which accepts multiple real-valued inputs and generates one real-valued output. Neurons apply an activation function (also called a transfer function) to a weighted combination of their input vector in :

$$out = a(b + in \cdot w) \tag{6.3}$$

where common choices of activation function are

$$a(x) = x \text{ (linear)} \tag{6.4}$$

$$a(x) = \frac{1}{1 + e^{-x}} \text{ (logistic sigmoid)} \tag{6.5}$$

$$a(x) = \tanh(x) \tag{6.6}$$

(For a linear activation function, multiplicative and additive constants are not necessary because the same flexibility is accomplished by the combination of inputs.) We use the

more general definition of *sigmoid* to mean any S-shaped curve, including tanh and logistic sigmoid.

In the simplest and most common case, neurons are arranged in layers, often with an input layer, a hidden layer, and an output layer. Each neuron in the hidden and output layers is connected to each of the neurons in the previous layer. The input layer simply outputs the inputs of the problem, the hidden layer uses a sigmoid activation function, and the output uses either a linear or sigmoid activation function. In this configuration, with the output neurons using a linear activation function, the universal approximation theorem or Cybenko theorem, states that a neural net is a universal approximator for continuous functions on compact subsets of \mathbb{R}^n [25].

General neural nets are stateless. The most common way to add state is to add past requests as additional inputs. Another approach is to use recurrent networks, where the outputs for one instance are used as additional inputs for the next instance. Unfortunately, this approach is slow to learn patterns involving large distances in time, at least when using gradient descent [13]. RMSProp (section 6.2.5) should alleviate this problem, since large time distances in a recurrent network is equivalent to a deep feedforward network. Another approach is to change the structure of the recurrent net and use Long Short-Term Memory (LSTM) [52].

An advantage of neural nets is their ability to learn complex interactions between features.

Neural nets are usually formulated as parametric models. However, some training algorithms, such as NEAT [101], change the number of connections or neurons. In that case, the neural net is a non-parametric model.

6.1.3 Associative memory

An associative memory, also known as content-addressable memory, is a type of neural net that stores patterns and will converge to a full pattern when presented with a partial pattern [73]. Features take on boolean values, so continuous features must be converted to boolean features, typically by binning. In other words, a single continuous feature will be represented as multiple boolean features, each representing a range of values. The discreteness requirement holds not only for the inputs but the

outputs as well. The large number of continuous inputs and the fine granularity needed for inputs and outputs means that a large number of bins is needed.

Hopfield networks, the traditional form of associative memory, do not scale well. The number of unique input vectors that can be stored is sublinear in the length of the input vector [80], although the number of parameters is quadratic. Hopfield networks are parametric models.

6.1.4 Decision trees

Decision trees are trees where each inner node specifies a decision and each leaf node specifies a result [70]. For a particular input, the tree is traversed starting at the root, and descending left or right based on the decision. In the simplest case, decisions are of the form “Is the value of feature x greater than the constant y ?” and results are constants. Decision trees are typically fast and capable of learning nonlinear functions.

Decision trees are formed top-down, and nodes are formed by looking at the set of instances, which is initially the entire set. If the set is small, a leaf node is formed, and the result at that node is the mean of the values. Otherwise, an inner node is formed by finding a decision that divides the instances well. Essentially, the mean of the values going left should be as different as possible from the mean of the values going right, and the variances should be small. The left and right child nodes are then recursively created with their respective partitions.

One disadvantage of decision trees is their statelessness. Again, this can be addressed by adding past requests as inputs.

Another disadvantage is their difficulty in learning patterns with certain symmetries. For example, consider inputs (x, y, z) with $x, y \in [0, 1]$, $z \in [0, 0.1]$, and the function $f(x, y, z) = (x + y \bmod 1) + z$. Assuming the inputs are uniformly distributed, $\{f(x, y, z) : x < x^*\}$ is distributed the same as $\{f(x, y, z) : x \geq x^*\}$ for any $0 < x^* < 1$, and likewise for y . This means that there is no x^* or y^* that splits the set well, and the algorithm will use something like $z^* = 0.05$ instead. Note that *after* using a “bad” split such as $x^* = 0.5$, the next level down will have good splits, and the end result would be much better than splitting on z^* . This is a result of the algorithm being greedy. (Finding an optimal tree is NP-hard [66] and therefore impractical.) Since we see these

symmetries in our data, decision trees are unlikely to have high accuracy.

Decision trees are non-parametric.

6.1.5 Nearest neighbor

Nearest neighbor simply finds the example that most closely matches the input, and uses its value as the prediction [23]. Nearest neighbor is obviously capable of learning any function as long as the examples are dense enough. The main disadvantages are the required density, large memory footprint (all the examples must be stored), and long search time. Indexing with a structure such as a k-d tree can ameliorate the search cost.

Nearest neighbor is non-parametric.

6.1.6 Support vector regression

SVR (related to support vector machines, SVM) works by finding a subset of the most useful training examples (so called *support vectors*) and combining them to produce predictions [98]. SVR performs a modified form of linear regression and can be extended to nonlinear regression by performing linear regression in an internal space that may not be equal to the input space. Vectors in this internal space (which may be quite large) do not need to be materialized, thanks to the *kernel trick*.

The kernel trick relies on the fact that only inner products between vectors in the internal space are ever used. Therefore, instead of specifying a function ϕ from the input space to the internal space, one need only specify a *kernel function* that is equal to the inner product of outputs of some ϕ , *i.e.*

$$k(x, y) = \langle \phi(x), \phi(y) \rangle \tag{6.7}$$

Common choices for the kernel function include:

$$k(x, y) = (x \cdot y + 1)^d \text{ (polynomial)} \tag{6.8}$$

$$k(x, y) = \exp(-\gamma \|x - y\|^2) \text{ for } \gamma > 0 \text{ (Gaussian)} \tag{6.9}$$

$$k(x, y) = \tanh(\kappa x \cdot y + c) \text{ for some } \kappa > 0 \text{ and } c < 0 \text{ (hyperbolic tangent)} \tag{6.10}$$

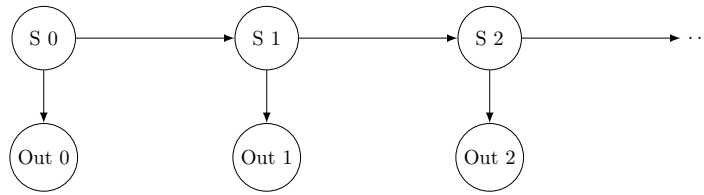


Figure 6.1: HMM

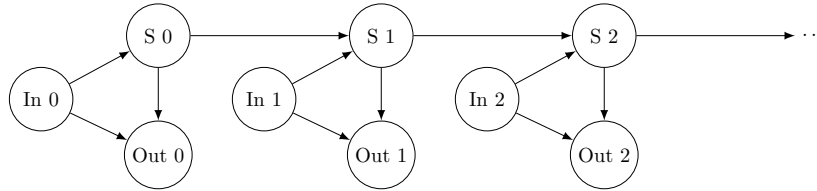


Figure 6.2: IOHMM

Choice of kernel function determines the type of nonlinearity introduced and affects how well SVR will perform on a particular problem.

SVR is non-parametric.

6.1.7 Input-output hidden Markov models

An IOHMM is an HMM augmented to include input [12] (see fig. 6.2). The state transition probabilities and output probabilities are then conditioned on the input [12]. This is done by using neural nets to generate the state transition matrix and output distribution per state, given the input.

IOHMMs have the advantage of natively supported time series data. However, they assume that the state space is discrete. Some components of the state space for storage devices, such as angular position for hard disk drives or time of the previous request, are effectively continuous. IOHMMs could be mapped to the problem by discretizing the state space, but the number of states would be too large to practically learn.

IOHMMs are parametric.

6.1.8 Genetic programming

Genetic programming is a specific type of genetic algorithm which entails evolving programs [83]. These programs are typically represented as trees, with operators as inner nodes and constants or variables as leaves. This is more expressive than the usual approach with genetic algorithms, because genetic programming evolves the structure of g , not just constants within a fixed formula.

One major advantage of this approach is its native support of time series. The most straightforward way to model a stateful process with most machine learning approaches is to include past stimuli in the model's input. With a genetic programming model, it is possible to include state directly as state variables.

We initially attempted to model latency using a stateful genetic programming model. As with all evolutionary algorithms, genetic programming is trivial to parallelize as long as the number of individuals in the population is not less than the number of threads. We took advantage of this by training on multiple machines concurrently. This model was successful in reproducing seek times and queue congestion, although it was not able to model rotational effects. We saw slight improvements by using a cellular variation, in which individuals are laid out on a grid and can only mate with neighbors. This appears to help by reducing mixing and encouraging diversity.

Unfortunately, genetic programming is slow to train, due to the large number of models that need to be created and evaluated.

Genetic programming is non-parametric.

6.2 Optimization methods

An optimization method takes a model and modifies the parameters to generate a better model. Equivalently, it takes a point in parameter space θ and finds another point θ' such that $J(\theta') \leq J(\theta)$ (with good probability). Most provide the guarantee, or at least the intent, that running the optimization longer asymptotically results in a model at a local optimum. Most cannot guarantee convergence to the global optimum unless the error function J is convex in terms of θ .

Most of the optimization methods here are applicable only to parametric mod-

els, and particularly to models where the gradient of the error function with respect to the parameters, *i.e.*

$$\nabla_i J_n^{(t)} := \frac{\partial}{\partial \theta_i} J(\hat{Y}_n^{(t)}, Y_n) \quad (6.11)$$

is defined and easy to calculate. All of the methods are applicable to at least one model described in section 6.1.

We use RMSProp because it is a stochastic method and our large data sets means that stochastic methods vastly outperform batch methods, and because RMSProp performs better than its competitors for deep neural networks, which are our chosen model.

6.2.1 Gradient descent (GD)

Gradient descent (GD), sometimes called steepest descent or method of steepest descent, simply takes steps “downhill,” in the direction opposite the gradient. In the context of neural networks, GD is also known as backpropagation or batch backpropagation. Given parameters $\theta^{(t)}$ at time t and a learning rate γ , the GD update is

$$\theta^{(t+1)} = \theta^{(t)} - \gamma \frac{1}{N} \sum_n \nabla J_n^{(t)} \quad (6.12)$$

Gradient descent has good convergence properties in terms of the number of iterations [16]. However, it is a batch algorithm, meaning that each iteration requires summing over all training examples. Better results in practice can be seen with stochastic gradient descent (SGD).

6.2.2 Stochastic gradient descent (SGD)

Stochastic gradient descent (SGD) is a modification of gradient descent that takes more frequent, less accurate steps. Instead of averaging the gradient over all training examples, SGD takes a step based on the gradient evaluated on a single, randomly-chosen training example. In the context of neural networks, SGD is also known as stochastic backpropagation. Given parameters $\theta^{(t)}$ at time t and a learning rate $\gamma^{(t)}$, the SGD update is

$$\theta^{(t+1)} = \theta^{(t)} - \gamma^{(t)} \nabla J_n^{(t)} \quad (\text{with randomly chosen } n) \quad (6.13)$$

To guarantee convergence, the learning rate $\gamma^{(t)}$ must satisfy [16]

$$\sum_{t=0}^{\infty} \gamma^{(t)} = \infty \tag{6.14}$$

$$\sum_{t=0}^{\infty} \left(\gamma^{(t)}\right)^2 < \infty \tag{6.15}$$

One good choice is to use

$$\gamma^{(t)} = \frac{\gamma^{(0)}}{1 + \eta t} \tag{6.16}$$

where $\gamma^{(0)}$ is the initial learning rate and η is the learning rate dropoff rate. The optimal value of η is $\gamma^{(0)}\lambda_{min}$, where λ_{min} is the smallest eigenvalue of the Hessian at the optimum [16]. Since λ_{min} may not be known, η may instead be chosen using cross-validation or other hyperparameter tuning techniques.

SGD converges more slowly than GD in terms of iterations, since each step uses a noisy estimation of the gradient. However, since steps are much faster, SGD is often much faster in practice for large data sets. SGD is also more resistant to local minima [15]. This resistance to local minima is shared with genetic optimization.

6.2.3 Conjugate gradient (CG)

Conjugate gradient (CG) chooses search directions that are (in some sense) orthogonal to previous search directions [96]. The idea is to reduce redundancy by not searching in the same direction multiple times. CG is a batch algorithm (like GD), so each iteration is slow to compute. Nonlinear CG requires a line search at each step. This means that at each step, once a search direction is chosen, a new optimization problem must be solved to find how far to move in that direction. The nested optimization problem further increases the cost of each iteration.

6.2.4 Resilient propagation (Rprop)

Rprop is a batch algorithm that looks at the sign but not the magnitude of each term in the gradient. One advantage of this approach is that it is unaffected by the fact that for deeper networks, the gradient can be very small or very large for weights near the input. Rprop comes in at least four variants called iRprop+, iRprop-, Rprop+,

and Rprop-. It performs well compared to other algorithms [55]. Unfortunately, we find that Rprop often diverges on our test dataset when combined with weight sharing.

6.2.5 RMSProp

RMSProp [104] is a stochastic algorithm similar to Rprop. The learning rate for a parameter is divided by a running average of the magnitudes of the gradients for that parameter. This acts as a form of preconditioning which normalizes the impact of the gradient to be roughly the same for each parameter. This is especially useful for deep neural networks, where the gradients of weights in early layers tend to have much smaller or much larger magnitudes than gradients of weights in later layers.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \gamma^{(t)} \frac{\nabla_i J_n^{(t)}}{\sqrt{\kappa_i^{(t)}}} \quad (\text{with randomly chosen } n) \quad (6.17)$$

$$\kappa_i^{(t+1)} = (1 - \alpha)\kappa_i^{(t)} + \alpha \left(\nabla_i J_n^{(t)} \right)^2 \quad (\text{with same randomly chosen } n) \quad (6.18)$$

Here, $\kappa_i^{(t)}$ is the running average of the squared magnitude of the i -th component of the gradient at time t , and $\alpha \in (0, 1)$ determines how quickly the running average responds to changes. Since

$$\frac{\nabla_i J_n^{(t)}}{\sqrt{\kappa_i^{(t)}}} \approx 1 \quad (6.19)$$

each component in the parameter update has approximately equal magnitude. While not as ideal theoretically as making use of the Hessian [27], RMSProp is easy to implement and only requires local information when scaling parameter updates.

6.2.6 Quickprop

Quickprop (sometimes called Qprop) assumes that the error surface is locally parabolic and jumps to the minimum of the parabola [110]. Equivalently, it assumes that the gradient is locally linear and jumps to the root. Quickprop does not work well in non-convex areas of the error surface. In concave areas of the error surface, Quickprop will construct a concave parabola and actually jump toward a local *maximum*, not minimum.

6.2.7 L-BFGS

Limited-memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) is a quasi-Newton optimization method. It uses an implicit representation of an approximation of the inverse Hessian matrix and does not require computing second derivatives [74]. It does require storing several recent parameter and gradient deltas, however, which increases memory consumption and compute cost.

6.2.8 Levenberg-Marquardt (LM)

This is an optimization algorithm that interpolates between the Gauss-Newton algorithm and gradient descent. One major disadvantage is that it requires multiplying large matrices ($N \times M$ and $M \times N$, where N is the number of parameters and M is the number of training examples) [115].

6.2.9 Simulated annealing

Simulated annealing [47] is intended to be analogous to annealing in metallurgy. The process starts at a high temperature, where the parameters change almost randomly. The temperature is slowly reduced, meaning the probability of switching from a good set of parameters to a worse set of parameters is reduced. Finally, the temperature reaches zero, where the algorithm behaves the same as hill climbing, *i.e.* parameters are updated only if the new configuration is better than the old one. Simulated annealing does not require gradient information and is good at escaping local minima, but it can be slow to converge. Simulated annealing can be used with parametric or non-parametric models.

6.2.10 Genetic algorithms (GA)

Genetic algorithms are optimization methods that “evolve” a set of inputs to minimize or maximize a particular function [21]. If the function is to be minimized, it is typically called the error function or loss function, and if it is to be maximized, it is typically called the fitness function. Each set of inputs, or *individual*, is usually represented as a string.

A population of individuals is initialized to randomly generated strings. Each individual in the population is evaluated using the fitness function/error function. Individuals with low fitness/high error are discarded. A new population is generated by applying crossover and mutation operators to the remainder. Then, the individuals in the new population are evaluated, another population is created, and so on.

Genetic algorithms are similar to simulated annealing in that they do not require gradient information and are good at escaping local minima, but can be slow to converge. They are trivial to parallelize, with each individual being evaluated in a separate thread. Genetic algorithms can be used with parametric or non-parametric models.

6.3 Ensemble methods

Ensemble methods create a composite model from many base models. They are especially useful with base models which are fast to evaluate, since the compute cost usually grow linearly with the number of base models in the ensemble.

6.3.1 Bagging

Bagging, named for Bootstrap AGgregation, averages many independent models [18]. This is commonly used with decision trees.

Given a training dataset D , many resampled datasets B_1, B_2, \dots, B_k are created. Each consists of elements chosen with replacement from D , and are often the same size as D . Each resampled dataset B_i is used to train a base model \hat{f}_i , and the final model is simply the average over \hat{f}_i .

Bagging does not typically cause overfitting.

6.3.2 Boosting

Boosting is a general framework which generates base models serially, training each new base model to reduce the residual error. The final model is then the sum of the base models.

There are many boosting algorithms, but we will use gradient boosting [41]

as an example and assume we are minimizing mean square error. The first model \hat{f}_0 is simply a constant, which is the mean of the values to predict \bar{y} . Each new model \hat{f}_m is then trained on the residual error $(x, y - \sum_{i=0}^{m-1} \hat{f}_i(x))$, and the final model is $\hat{f}(x) = \sum_i \hat{f}_i(x)$.

Because the base models are not independent, each new base model increases the number of parameters in the composite model. This can lead to overfitting if steps are not taken to prevent it.

Chapter 7

Automatic learning

This chapter is about our basic approach and how it performs. In this chapter, because of the difficulty of the general problem, we focus on a simpler subproblem by restricting ourselves to a single zone of a hard disk drive. Scaling and cross-zone effects are addressed in later chapters. Our basic approach is to feed sines and cosines into specially constructed neural nets. In this chapter, we use Fourier analysis to discover the sines and cosines, although we describe the disadvantages of Fourier analysis and an alternate approach in chapter 8. We show that with Fourier analysis and specially constructed neural nets, we can achieve low error over a single zone of a hard disk drive. We describe empirical results of training with different error functions described in section 4.2.2 and show that mean absolute error is most appropriate for our problem. We also describe a method for automatically tuning neural network hyperparameters, since the hyperparameters (which describe the network structure) are very influential on the final results but difficult to tune.

7.1 Neural nets

Neural nets are the machine learning algorithm that we use for predicting response times. Neural nets have many advantages, including that they interpolate smoothly, have small memory requirements, support continuous inputs, and are reasonably fast to train (compared to genetic programming). Neural nets do not require choosing a kernel function, which is a critical choice for SVR [20], but not at all an obvi-

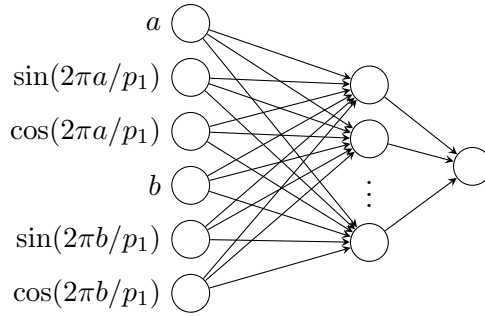


Figure 7.1: Network architecture when subnets are not used

ous choice for response time prediction. Neural nets also allow us to use weight sharing, described in section 7.1.2. Finally, neural nets are capable of reproducing behavior with arbitrary periods based on a base set of periodic inputs, described in section 8.3.

7.1.1 General configuration

We start with a configuration that is commonly used and well-studied. We use a multilayer perceptron (MLP), which is a neural network that consists of layers of neurons, where each layer is typically fully connected with the next. MLPs are feed-forward, which means that signals propagate in one direction, with no loops.

As is common with MLPs, the neural net has three layers: input, hidden, and output. (When using shared weights, described in section 7.1.2, there are multiple hidden layers.) The input layer has a neuron per input feature. Neurons in this layer do not perform any computation; they simply output the value in their part of the input vector. The hidden layer consists of a relatively large number of neurons, all with a nonlinear transfer function. The output layer consists of a single neuron with the linear transfer function x , which allows its value to vary over the entire real line.

Remember that commonly used transfer functions are $\frac{1}{1+e^{-x}}$ and $\tanh(x)$ (eqs. (6.5) and (6.6)). The transfer function that we use for hidden neurons is $\frac{1}{1+e^{-x}}$. We have also tried $\tanh(x)$, but it seems to make no substantive difference, which makes sense because $\frac{1}{1+e^{-x}} = \frac{1}{2} \tanh\left(\frac{1}{2}x\right) + \frac{1}{2}$. Another transfer function worth exploring is $\frac{x}{1+|x|}$. While similar in shape to $\tanh(x)$, $\frac{x}{1+|x|}$ is roughly 10 times faster to calculate than $\frac{1}{1+e^{-x}}$.

7.1.2 Shared weights

A useful and device-independent assumption is that all sectors map from LBNs to geometrical space in the same way. This is encoded into the neural net by applying weight sharing across two subnets (see fig. 7.2). This means that the weights in each neuron in subnet 1 are identical to the weights in the corresponding neuron in subnet 2. This is essentially a very simple convolutional neural net with only two instances of the convolving subnet and no overlap of inputs. A similar approach was used by Kindermann *et al.* for solving functional equations with neural nets [60]. Another interpretation is that the subnets perform feature extraction, knowing that the two sectors are instances in the same input space. This is often used in image processing [105]. Effectively, this means that instead of a single data point from a 2D function for each sector pair, the subnets get two data points from a 1D function. This makes our sampling effectively much denser.

More formally, the net assumes f can be expressed as

$$f(a, b) = h(g(a), g(b)) \tag{7.1}$$

where g (the subnet) maps from LBNs to geometry, and h (the main net) maps from geometry to access time. Reuse of g , which is equivalent to weight sharing in the neural network, formalizes the assumption that all sectors map to geometry in the same way. Technically, f is not restricted as long as the intermediate space is at least as large as the input space, and if g is expressive enough, since one could always use $g(x) = x$ and $h = f$, but weight sharing predisposes the neural net to learn more useful decompositions.

All neurons use a sigmoidal activation function except the final output, which is linear. This is common practice for neural nets performing regression [38]. The number of layers and their sizes are chosen as per section 7.3.

7.1.3 Training

We use RMSProp (section 6.2.5) with a minibatch size of 10 to train the neural net. The networks can be relatively deep (3–10 hidden layers), which causes problems with plain gradient descent due to the exploding/vanishing gradient problem. We also found that stochastic variants are much faster than batch training, probably due to

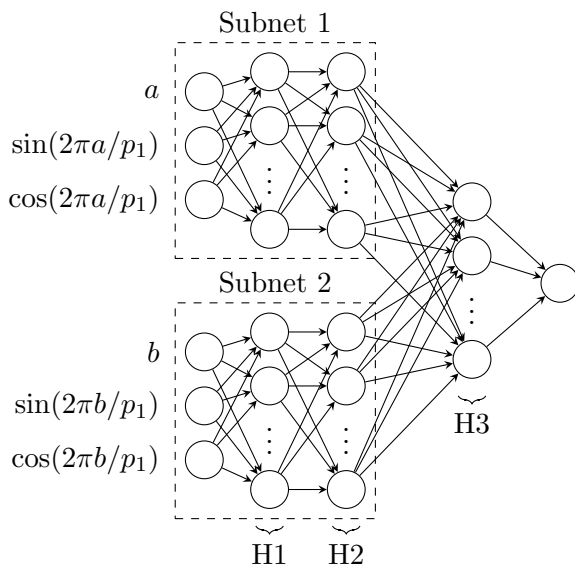


Figure 7.2: Network architecture when subnets are used. Note that the weights in subnet 1 are identical to the weights in subnet 2.

our relatively large dataset sizes. Use of RMSProp actually turns out to be crucial for training the neural net in a reasonable amount of time, since plain gradient descent does not work for deep neural networks.

7.2 L_1 norm versus L_2 norm

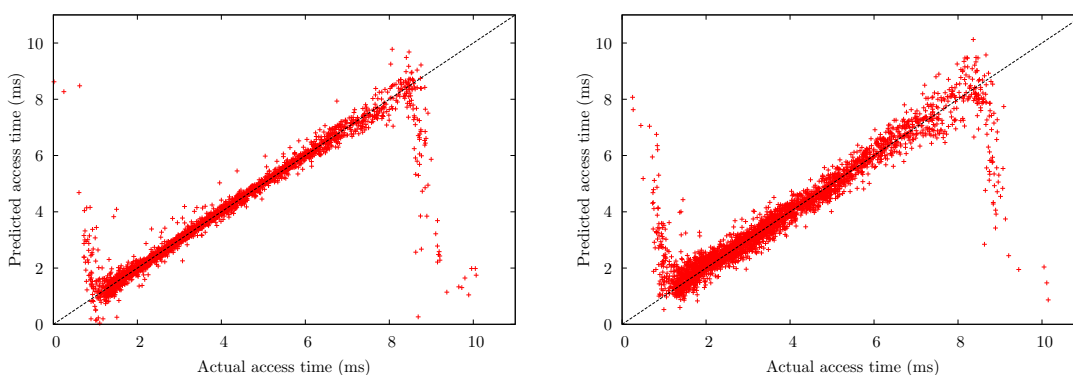
We chose to minimize the L_1 norm (equivalent to minimizing Mean Absolute Error, or MAE) instead of the L_2 norm (equivalent to minimizing Root Mean Square Error, or RMSE) mainly because the L_2 norm is known to be sensitive to outliers [8, 112].

Empirically, minimizing the L_1 norm gives a tighter correspondence between predicted and actual values compared with the L_2 norm (fig. 7.3). We also found that minimizing the L_1 error reduces both the L_1 error and, surprisingly, the L_2 error (table 7.1). In other words, minimizing the L_1 error during training resulted in lower L_2 error during testing than minimizing the L_2 error during training. We believe this is probably due to the sensitivity of the L_2 norm to non-Gaussian errors causing the model to generalize poorly.

	L_1 test error	L_2 test error
Minimizing L_1 training error	0.139 ± 0.003	0.730 ± 0.019
Minimizing L_2 training error	0.273 ± 0.005	0.799 ± 0.019

Table 7.1: Effect of minimizing L_1 versus L_2 as measured by L_1 and L_2 , with 95% confidence interval over 50 runs

Figure 7.3 shows large errors at extreme values. The cause is the fact that the access time function has many large discontinuities (see fig. 7.7a) where it “wraps” from the maximum to minimum value. Near a discontinuity, a small error may push the predicted value to the wrong side of the discontinuity compared to the actual value, yielding a large error. Hence, extreme values have a higher chance of large error.



(a) Predicted versus actual access times when trained with L_1 norm

(b) Predicted versus actual access times when trained with L_2 norm

Figure 7.3: Comparison of L_1 versus L_2 norm for neural nets with subnets. (A narrow cluster along the diagonal $y = x$ is better.) High errors are seen at extreme values due to discontinuities in the access time function. For details, see section 7.2.

7.3 Hyperparameter tuning

Hyperparameters are variables which affect what biases a model has or how an algorithm learns and usually must be set before training begins. Neural network hyperparameters include layer sizes, learning rate, momentum, and magnitudes of the

initial weights. Hyperparameters are often tuned using some combination of manual search and grid search [51, 68, 65, 14], although genetic algorithms are also used [69, 4, 57, 39]. Grid search is known to be inefficient if not all hyperparameters have equal importance [14], and manual search is not reproducible. We use a genetic algorithm to tune the machine learning algorithms’ hyperparameters.

A set of hyperparameters is evaluated by running the machine learning algorithm with those parameters and evaluating its performance. Since training a neural network or building an ensemble of decision trees is relatively expensive, we use a relatively small population size to reduce the amount of computation needed.

The representation used in our genetic algorithm includes booleans (for periods to be included or not, discussed more in section 8.1.5), positive integers (for layer sizes), and positive reals (for the learning rate, momentum, and initial weight magnitudes). Using a representation that matches our problem should provide better performance than using a simple bit string [82]. When mutating, booleans are flipped, and integers are incremented or decremented, and reals are multiplied by a log-normally distributed value.

Initial hidden layer sizes were sampled from a log-normal distribution with $\mu = \ln 10$ and $\sigma = \ln 10$ (then cast to an integer). The initial learning rates were sampled from a log-normal distribution with $\mu = \ln(4 \times 10^{-3})$ and $\sigma = \ln 100$. Momenta were sampled uniformly from 0 to 1, which is the range of momentum values which are useful and meaningful. The initial weight standard deviations were sampled from a log-normal distribution with $\mu = 0$ and $\sigma = \ln 10$. Other than momentum, distributions are chosen to be somewhat reasonable values based on experience and literature. The idea is that the values do not need to be precise, since the genetic algorithm will tune them. There just needs to be a reasonably high probability of choosing a value that is not outrageous.

Individuals were evaluated by training a neural net with stochastic backpropagation and minimizing the L_1 error. Initial weights were sampled from a normal distribution with mean 0 and standard deviation that is a hyperparameter that depended on the layer. A random 10% of the dataset was set aside as a test set, and the neural net was trained on the remainder for 10 epochs. A penalty of 1.8×10^{-5} per connection was added to the error to discourage unnecessary bloat. This value for the penalty was

chosen to be roughly 10% of the raw error.

After each generation, the best 25% was kept for mating, and the rest were discarded. Random pairs were selected for mating, and attributes were randomly swapped to generate pairs of children. Each child was mutated such that the expected number of attributes changed was $5/8$. On mutation, a layer size would be incremented or decremented, and real values would be multiplied by a value sampled from a log-normal distribution with $\mu = 0$, $\sigma^2 = 10^{-2}$ (so that an “average” change would be $\pm 10\%$).

Multiple generation lengths were tested to see if the optimal parameters depend on the generation length. Generation length is the number of epochs spent training each neural network.

The error of the genetic algorithm drops quickly, with most of the gain seen in the first 50 generations (fig. 7.4). We ran the genetic algorithm for 400 generations to ensure maximum benefit, but this is not necessary if one wants faster results.

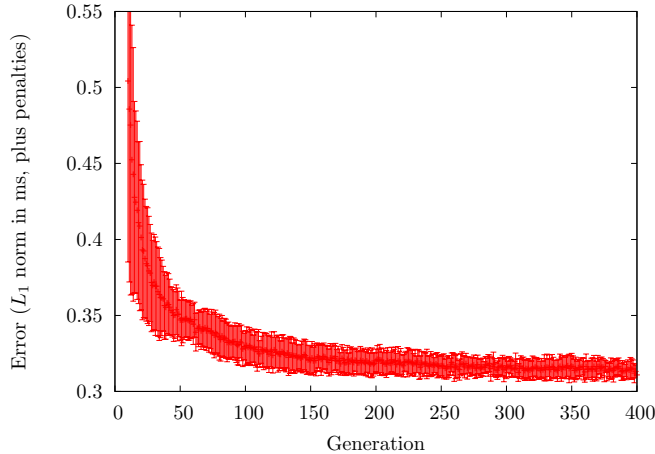


Figure 7.4: Error versus generation of the genetic algorithm for neural nets with subnets. The neural net is trained to predict access times over the first zone of HD1. Note that this is higher than the final error because 1) this error includes penalties, and 2) this error is calculated based on neural nets trained for a smaller number of epochs.

For neural nets with subnets, the H2 layer (fig. 7.2), which corresponds to the output of the g subnets, evolves to a relatively small size (about 6 or 7 neurons) with little variance (fig. 7.5a). This suggests that the dimensionality of the space of physical

sector locations is low, and that allowing for a larger intermediate space may actually be detrimental.

When comparing the actual and predicted access time functions (fig. 7.7), we see that the shape is matched very well. Even the notches in the stripes are in the predicted locations.

Different experiments converge to the same values for most hyperparameters regardless of generation length (fig. 7.5). This is useful because it means that the hyperparameters can be chosen using a short generation length, and then the final neural network can be trained for a longer period of time. This is the approach we take, with neural nets being trained for 10 epochs during hyperparameter tuning, then trained for 1000 epochs for the final evaluation. Using 10 epochs per generation for hyperparameter tuning limits the amount of time spent in the hyperparameter tuning phase, and training for 1000 epochs once the hyperparameters are chosen gives the neural net time to reduce its error to a near-minimal value.

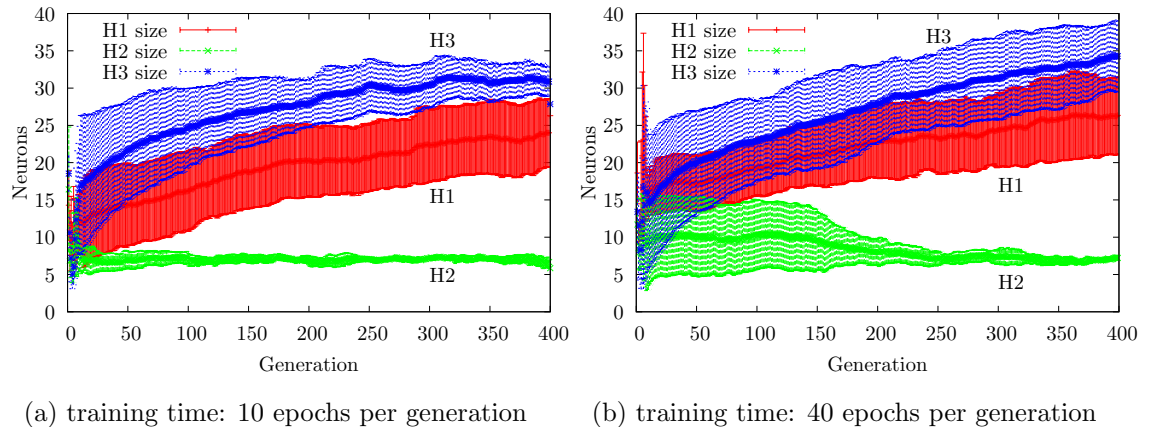


Figure 7.5: Size of hidden layers versus generation. Training only for 10 epochs versus 40 epochs reveals about the same results.

Unsurprisingly, a higher learning rate is chosen when the generation length is very short. This makes intuitive sense because if the generation stops while a neural network’s error is still dropping rapidly, a small increase in the learning rate will provide a large benefit. Even though the higher learning rate increases noise in the training error, the early cutoff means that the training is stopped before this becomes a problem.

Since the optimal learning rate (and an associated hyperparameter, the momentum) are dependent on how many epochs the neural net is trained for, and since the number of epochs is significantly different for the final training phase (1000) than the hyperparameter tuning phase (10), we discard those learned values and used fixed values of 4×10^{-4} for the learning rate and 0.3 for the momentum when training a network for evaluation. Learning for the same number of epochs during the hyperparameter tuning phase and the training phase would allow us to use the automatically tuned learning rate and momentum, at the expense of the hyperparameter tuning phase taking 100 times as long to complete.

7.4 Results

7.4.1 Baseline data

For reference, we ran DIG (section 3.2.3) against HD1 (section 3.3). From the DIG data, the track length at the beginning of the drive is 2528 sectors, and the skew is 1.1908 ms, which corresponds to 361.37 sectors (given the rotation time of 8.33 ms). We expected to see a dominant period of $2528 - 361.37 = 2166.63$ sectors, which is a frequency of $1/2166.63 = 4.62 \cdot 10^{-4}$. From the Fourier analysis, the actual dominant period is $1/(4.52 \cdot 10^{-4}) = 2212.99$ sectors, which is close to our prediction of 2166.63.

7.4.2 Errors

Access time errors are listed in table 7.2. The first entry is the error for a model that always predicts the mean value of the dataset.

When using decision trees, adding the sines and cosines as additional features reduced the error noticeably. Bagging the decision trees reduced error further, but only when the periodic information was included. The lowest mean absolute error achieved with decision trees was 0.526 ms.

When using neural nets, adding the sines and cosines reduced the error significantly. Weight sharing reduced the error further by a small but noticeable amount. The lowest mean absolute error achieved with neural nets was 0.149 ms, which is close to the inherent noise level of 0.048 ms (section 5.1).

Most of the time, neural net tests with random periods fared worse than tests with no period information. However, one test with random periods performed much better, although not as well as tests with the correct period information. This appears to have been caused by one of the random periods being nearly equal to twice the most useful period.

When comparing the actual and predicted access time functions (fig. 7.7), we see that the shape is matched very well. Even the notches in the stripes are in the predicted locations.

Effectiveness of scheduling prediction was also tested (table 7.3). The entire contents of the queue was presented to the machine learning algorithm, and the index of the next request to complete was predicted (using a 1-of- k encoding with the neural net). Neural nets outperformed decision trees, at least when the correct periodicity information was provided. Again, we see that including the correct periodicity information is crucial for the performance of both decision trees and neural nets. Weight sharing had a much more significant impact than in the response time experiment. This may be due to the fact that the scheduling experiment feeds a larger number of requests into the neural net simultaneously, thereby increasing the number of weights that can be shared. As previously noted, this thesis is focused solely on access time; scheduling is considered as a factor relevant for future work.

7.4.3 Model creation time

The GA was run on a cluster with 72 cores. A population size of 144 individuals was used, which gives 2 individuals per core for reasonable CPU utilization. For the workload of 40,000 half-zero random reads over the first zone (237,631 sectors), each individual was trained for 5 minutes, which gives 10 minutes per generation. Reasonable results can be achieved after about 50 generations, which gives a total run time of 8 hours and 20 minutes. Larger regions will increase the amount of data and size of the neural nets required, which will increase the training time. This time could be decreased by starting with a better initial population and running for fewer generations.

When training an individual neural net, error initially drops rapidly, then declines slowly (fig. 7.6). To give a sense of magnitude of training times, in one example,

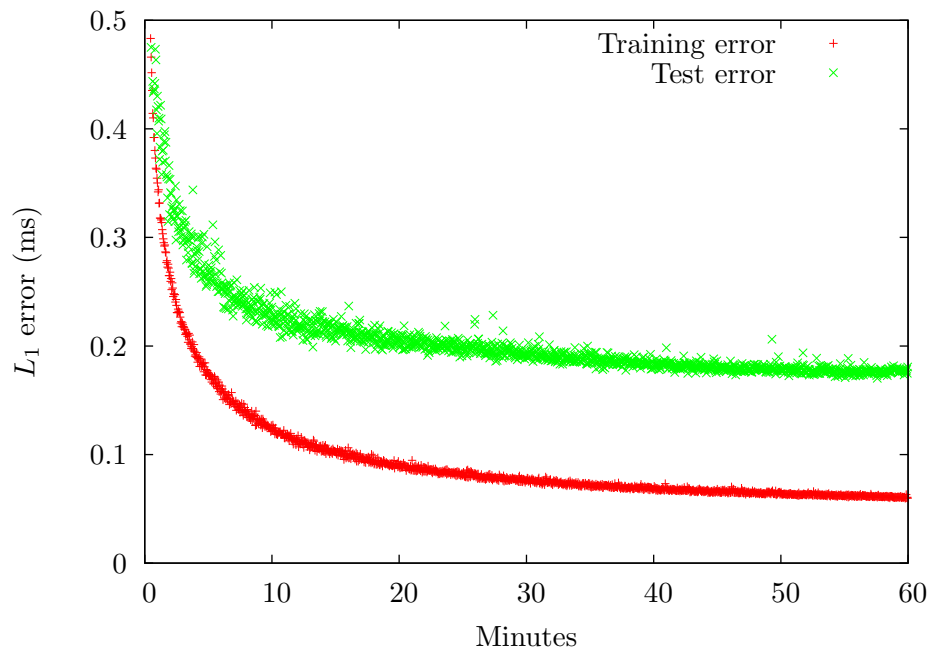


Figure 7.6: Error of the neural net over time while training

training for 5 minutes yields an error of about 0.27 ms, while training for an hour yields an error of about 0.18 ms. Of course, all training time depends on the configuration chosen and the computer running the training.

	Configuration	L_1 error (ms)	L_2 error (ms)
	constant value	1.651 ± 0.017	2.014 ± 0.019
Manually tuned decision trees	no periods, w/o bagging	1.650 ± 0.001	2.014 ± 0.008
	no periods, w/ bagging	1.655 ± 0.000	2.023 ± 0.004
	random periods, w/o bagging	1.608 ± 0.062	1.969 ± 0.056
	random periods, w/ bagging	1.545 ± 0.152	1.925 ± 0.180
	Fourier periods, w/o bagging	1.082 ± 0.085	1.544 ± 0.183
	Fourier periods, w/ bagging	0.748 ± 0.009	1.349 ± 0.338
Auto-tuned decision trees	no periods, w/o bagging	1.640 ± 0.009	2.006 ± 0.016
	no periods, w/ bagging	1.649 ± 0.001	2.016 ± 0.005
	random periods, w/o bagging	1.519 ± 0.169	2.022 ± 0.015
	random periods, w/ bagging	1.058 ± 0.336	1.507 ± 0.295
	Fourier periods, w/o bagging	0.865 ± 0.046	1.717 ± 0.129
	Fourier periods, w/ bagging	0.526 ± 0.019	1.032 ± 0.018
Manually tuned neural nets	no periods, w/o subnets	1.603 ± 0.016	2.058 ± 0.017
	no periods, w/ subnets	1.593 ± 0.023	2.043 ± 0.026
	random periods, w/o subnets	1.616 ± 0.024	2.072 ± 0.030
	random periods, w/ subnets	1.401 ± 0.426	1.894 ± 0.365
	Fourier periods, w/o subnets	0.308 ± 0.009	0.969 ± 0.034
	Fourier periods, w/ subnets	0.236 ± 0.010	0.808 ± 0.036
Auto-tuned neural nets	no periods, w/o subnets	1.613 ± 0.020	2.079 ± 0.026
	no periods, w/ subnets	1.608 ± 0.027	2.059 ± 0.030
	random periods, w/o subnets	1.270 ± 0.455	1.791 ± 0.390
	random periods, w/ subnets	1.394 ± 0.417	1.862 ± 0.363
	Fourier periods, w/o subnets	0.298 ± 0.012	0.961 ± 0.060
	Fourier periods, w/ subnets	0.157 ± 0.015	0.785 ± 0.065
	“unrestrained” neural net*	0.149 ± 0.021	0.781 ± 0.072

Table 7.2: Average errors for access time predictions. Neural nets were trained for 1000 epochs. Solid bars show the mean error, and thin bars show the standard deviation of the error. For more details, see section 7.4.

*The “unrestrained” neural net was evolved with all penalties set to 0, so it has lower error at the expense of larger size.













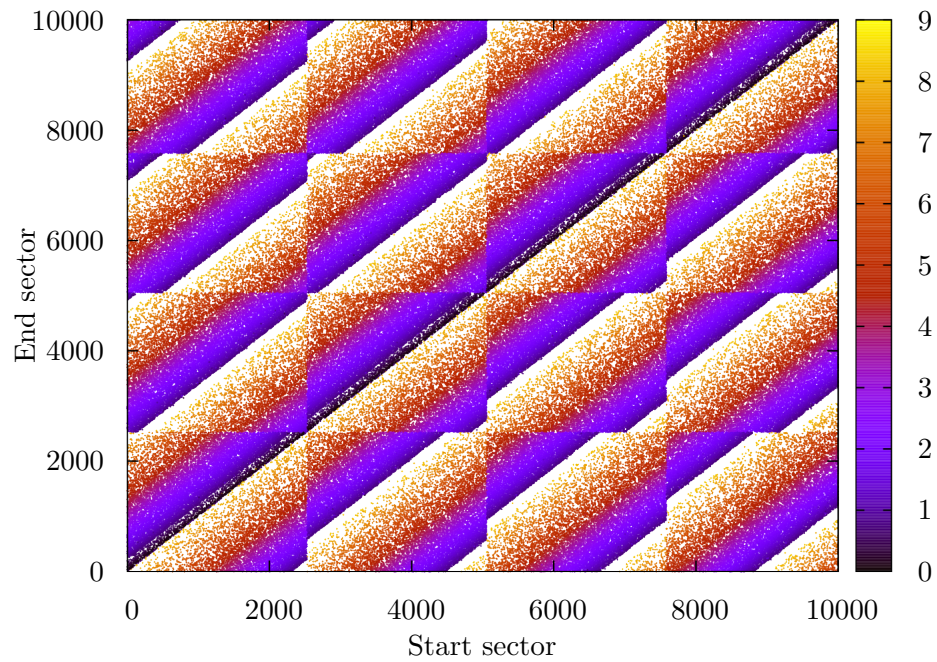
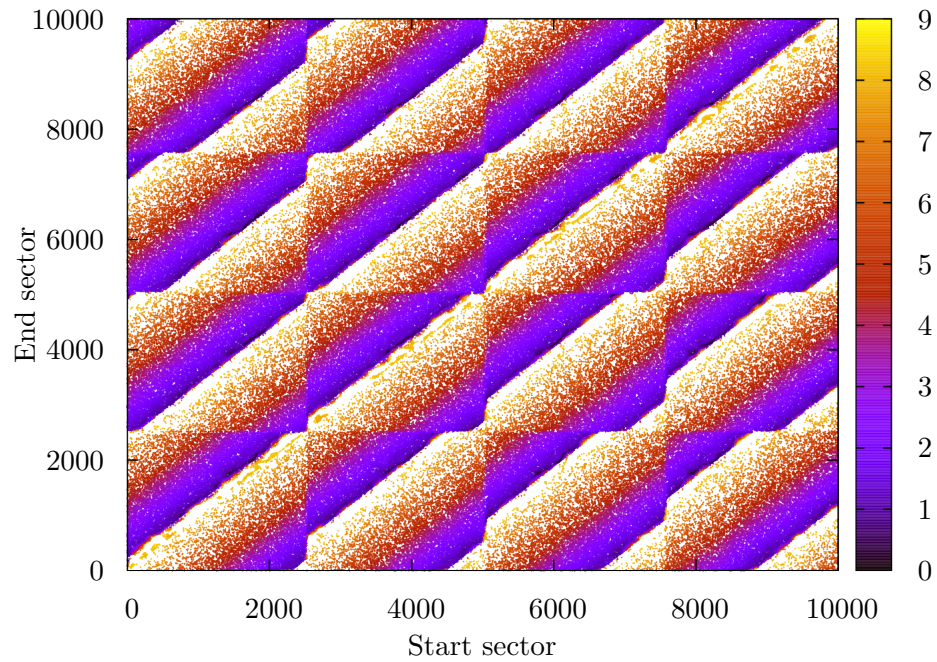
	Configuration	Error (%)
Decision trees	no periods, without bagging	33.11 ± 1.02 
	no periods, with bagging	32.20 ± 0.64 
	6 random periods, without bagging	34.33 ± 0.01 
	6 random periods, with bagging	33.42 ± 0.95 
	6 periods, without bagging	31.24 ± 0.70 
	6 periods, with bagging	20.50 ± 1.04 
Neural nets	no periods, without subnets	38.80 ± 1.91 
	no periods, with subnets	33.60 ± 0.29 
	6 random periods, without subnets	42.59 ± 0.55 
	6 random periods, with subnets*	38.26 ± 14.80 
	6 periods, without subnets	10.40 ± 2.69 
	6 periods, with subnets	5.50 ± 0.75 

Table 7.3: Average scheduling error, as measured in section 4.3.1.1.

*One run had an error of 11.80%, and the others averaged 44.88%.



(a) Actual times



(b) Predicted times

Figure 7.7: Access time over the first 4 tracks of a hard disk drive, in milliseconds

Chapter 8

Periodicity

8.1 Scalable discovery of periodicity

Periodicity shows up in the behavior of many types of storage devices. In hard disk drives, periodicity comes from rotation, arrangement of sectors into tracks, track skew, and so on. In RAID arrays, periodicity comes from the regular striping of data across component devices. In SSDs, although hidden by the FTL, periodicity comes from the hierarchy of channels, packages, dies, planes, blocks, and pages [6, 32]. As described in section 2.2, periodicity is challenging for general-purpose machine learning algorithms. To make up for this, we incorporate the periodicity explicitly into the machine learning algorithm's input.

We assume that the access time function may have components that are *locally periodic*. By locally periodic, we mean that the function's domain can be partitioned such that the function is periodic when restricted to any particular partition. This is a fairly lax assumption. First of all, it is well-known that these functions do have locally periodic components for existing hard disk drives, due to track lengths and track skews that are constant within a serpentine. Periodicity is likely to occur in other devices as well, because of the benefits of regular repetition in designs. Secondly, as described in section 8.1.4 the cost is minor if these functions do not have locally periodic components.

8.1.1 Useful frequencies

The most accurate method of determining which frequencies are useful is to train models with different frequencies included and observe how the frequencies affect the models' errors. Unfortunately, this is prohibitively expensive. The number of frequencies to search is roughly the number of sectors in the storage device, or 976,773,168 for our test hard disk drive. Training a model takes a few minutes to a few hours, depending on the approach and size of the dataset, so training a model for each frequency would take a very long time.

To identify useful frequencies faster, we can calculate the Fourier transform of the access time function and find frequencies with large Fourier components, which we call strong frequencies. Although they correlate well, useful frequencies do not necessarily correspond one-to-one with strong frequencies. (Section 8.3 further discusses this problem and how to work around it by bypassing the Fourier transform entirely.) A useful frequency may occur with multiple phases in different locations and cancel itself out, thereby causing its Fourier component to be small. A strong frequency may be useless because it is a harmonic of a useful frequency and therefore redundant.

It is unlikely that a frequency will occur in multiple phases in such a way to cancel itself out entirely, so the set of useful frequencies is likely to be a subset of the set of strong frequencies. After strong frequencies are isolated, more expensive methods may be used to cull the useless frequencies using a class of machine learning techniques known as feature selection. This is discussed in section 8.1.5.

8.1.2 Finding strong frequencies

We refer to the previously accessed sector as the *start sector*, which is the current position of the head, and the first sector of the current request as the *end sector*, which will be the new position of the head. Let $f(a, b)$ be the access time function, where a and b are the LBNs of the start and end sectors, and f returns the access time in milliseconds. Capturing the access time for every sector pair takes a very long time. Given our test device's mean access time of 15.5 ms and 976,773,168² pairs of sectors, the time to capture all of them would be roughly 469 million years. Obviously, this is infeasible, so we are limited to a very sparse sampling.

The sparsity means that we cannot use the Fast Fourier Transform, so we fall back to the brute force method of calculation. Let $x = (a, b)$ and $\xi = (u, v)$, where u and v are coordinates in frequency space. Further define N as the number of data points (in our case, the number of requests in the trace), M as the number of frequencies to search, and \tilde{f} as the Fourier transform of f . The discrete Fourier transform is defined:

$$\tilde{f}(\xi) = \frac{1}{N} \sum_k f(x_k) e^{-2\pi i x_k \cdot \xi} \quad (8.1)$$

which takes $O(N)$ time to calculate for a single frequency vector ξ , so calculating \tilde{f} for M frequencies takes $O(MN)$ time. Searching all frequencies in the 2D space leads to infeasible computation time for large datasets or datasets over larger regions of the drive.

One way to approach the Fourier analysis is to use a search-based method, rather than scanning uniformly. Assuming the number of strong frequencies is small, this should be much faster, roughly $O(kN \log M)$ for k strong frequencies. Unfortunately, these methods require performing convolutions which result in dense matrices that are far too large to calculate. Instead, we rely on uniform scanning, but only in a small portion of the space.

8.1.3 Scaling the search for strong frequencies

Note that the access time depends mostly on the difference between the sectors. This is intuitively true, as the time to move from sector 0 to sector 9 should be roughly the same as the time to move from sector 1 to sector 10. This causes stripes in the access time function along $a = b$ (see fig. 7.7a), and the Fourier transform of a function with stripes has strong components in the direction orthogonal to the stripes [44], which means that strong components of the access time function should lie on $u = -v$. We see this empirically in fig. 8.1. By searching only this diagonal instead of the entire space, the computation time becomes feasible.

Off-diagonal frequencies occur, but they are mostly related in a straightforward fashion to frequencies on the diagonal. An example would be a dataset that includes track sizes of 10 and 11. Strong coefficients are likely at periods (10, 10), (10, 11), (11, 10), and (11, 11). Since the diagonal provides (10, 10) and (11, 11) (or just 10 and

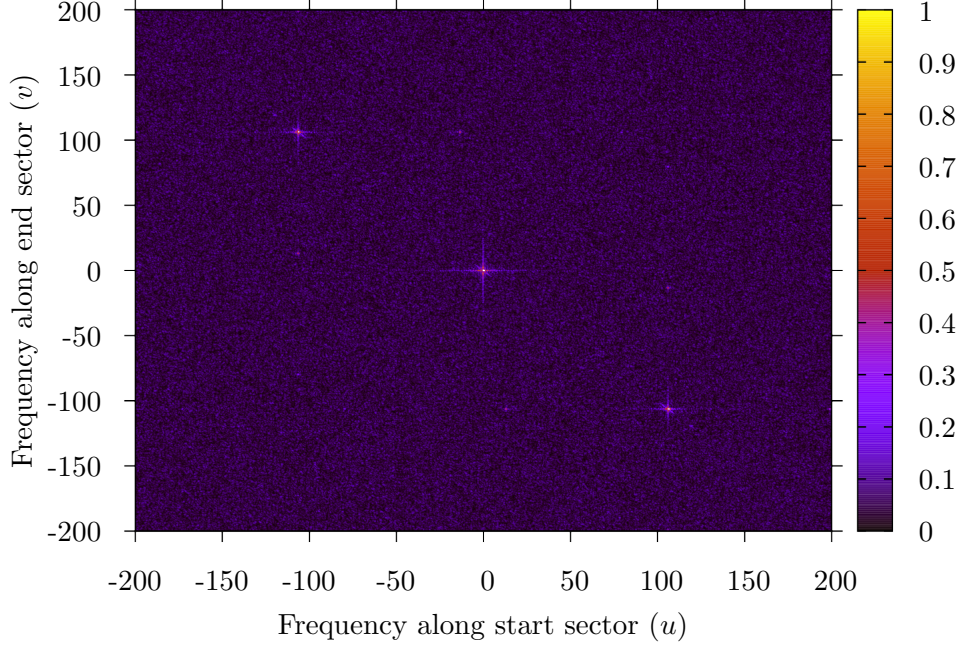


Figure 8.1: Full 2D Fourier spectrum for the first $K = 237,631$ sectors out to $\frac{200}{K} \times \frac{200}{K}$, which corresponds to periods of at least $\frac{K}{200} = 1188.155$ sectors. Values for u and v are in oscillations per K sectors. Plot is clipped to magnitude 1 to show detail, but central spike goes up to 8.6, and other diagonal spikes go up to 3.9. This figure shows that strong frequencies do lie on the diagonal $v = -u$.

11 in 1D), the locations of the others could be inferred, although that is not actually necessary.

To determine a threshold for strong frequencies, we sample 1000 frequencies at random and calculate $|\tilde{f}|$. The threshold is then set to the mean plus six standard deviations. Assuming the magnitudes are normally distributed, this means that a frequency has roughly a 1 in 500 million chance of being spuriously flagged.

We scan across $v = -u$ with a step size of $0.1/blockCount$, looking for local maxima above the threshold. The peaks were often not centered on integer multiples of $1/blockCount$, which is why the step size is not $1/blockCount$. After finding a maximum, we perform a local search to fine-tune its position.

Due to structures such as serpentines, f may have higher-order periods. In

other words, f may have period p_1 for a subrange, then p_2 , then p_1 , then p_2 , etc., with the switch between p_1 and p_2 being periodic. Currently, we have no method for detecting these explicitly, although we have seen them show up as useful periods in their own right. These actually cause further problems by reducing the utility of the lower-level periods. The multiple regions using period p may be out of phase, causing $|\tilde{f}|$ to drop. Essentially, this is a form of mixed interference which results in a weaker signal than if the signals interfered purely constructively. This issue is sidestepped in section 8.3 by dropping the Fourier transform.

8.1.4 Input augmentation

Once all useful periods have been found, the input vectors for the neural net are augmented. Given an input (a, b) and periods p_1, \dots, p_k , the augmented input vector is $(a, \cos(2\pi a/p_1), \sin(2\pi a/p_1), \dots, \cos(2\pi a/p_k), \sin(2\pi a/p_k), b, \cos(2\pi b/p_1), \sin(2\pi b/p_1), \dots, \cos(2\pi b/p_k), \sin(2\pi b/p_k))$. We use sinusoids of a and b separately rather than sinusoids of $b - a$ because the period changes for each input separately. For example, if a is in a region where p_1 dominates, and b is in a region where p_2 dominates, then $\sin(2\pi a/p_1)$ and $\sin(2\pi b/p_2)$ are useful, but $\sin(2\pi(b - a)/p_3)$ is not likely to be useful for any period p_3 .

If no useful periods are found, then the input vectors are unchanged. This means that for devices with no periodicity, the cost of the periodicity assumption is just the time to search for periods. The neural net is unchanged, and therefore the speed and accuracy of the model is unchanged.

8.1.5 Hyperparameters

The top 25 periods with the largest Fourier magnitudes were used as candidate periods. Adding to the hyperparameters discussed in section 7.3, each period has an associated boolean which determines whether that period is used or not. This allows us to find the subset of strong frequencies that are useful frequencies. Each period was initially included (its boolean set to true) with a probability of 10%. On mutation, a period's inclusion would be toggled.

A penalty per included period is added to prevent inclusion of unnecessary

periods and to improve convergence. When no penalty term is added, the included periods do not converge well (fig. 8.2a). Although some periods are strongly selected for (top of the plot), and others are strongly selected against (bottom of the plot), many flip back and forth within a run and are inconsistent across runs. By including the penalty term, the included periods converge quickly (clear middle area, fig. 8.2b).

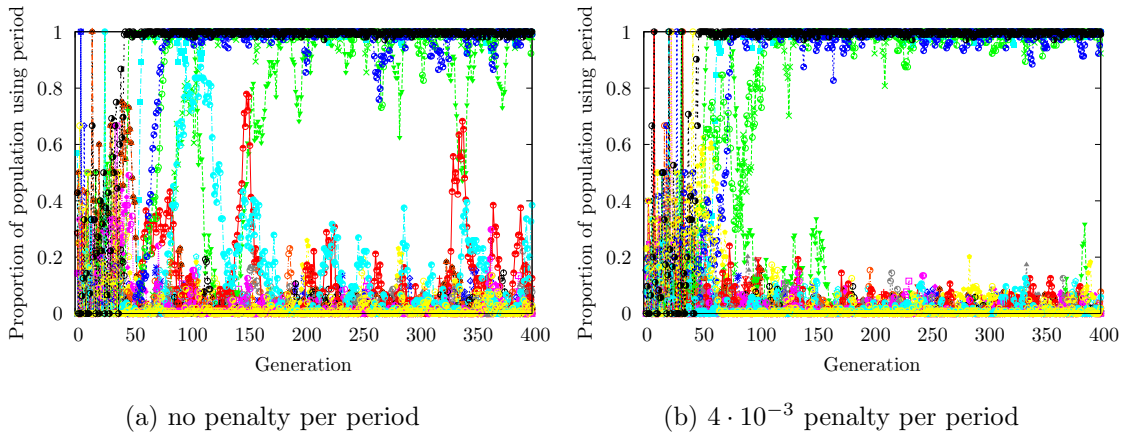


Figure 8.2: Usage of periods versus generation (neural nets with subnets). Note that with no penalty, periods converge poorly, *i.e.* cross $y = 0.5$ even in later generations.

The value of the penalty was chosen by starting with 10^{-3} , which would give a penalty term roughly equal to 10% of the L_1 norm. We then doubled the penalty until we saw that the periods converged well, which happens with a penalty of 4×10^{-3} .

8.2 Changes in periodicity

As track lengths change across the platter, the periodicity changes. This causes the Fourier transform to be messy fig. 8.3. Peaks are short, misshapen, and not centered exactly on the relevant frequency. Temporal information (*i.e.* the location of the frequency shift) is also lost.

To account for this, we split the logical block space into regions. Each region has its Fourier transform computed separately. Each region also gets its own subnet, which is swapped into the neural net when a request lies in that region fig. 8.4.

Treating each region entirely separately would require too many subnets to be

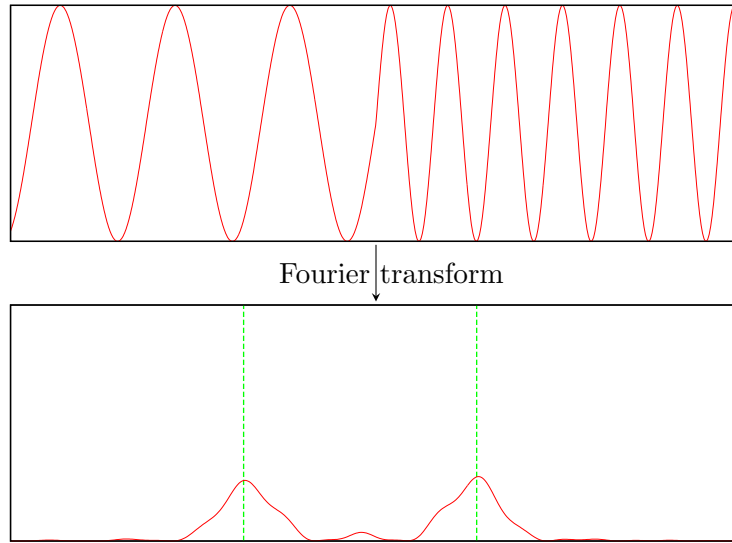


Figure 8.3: If periodicity is not constant, the Fourier transform is messy. Peaks are short, misshapen, and off-center.

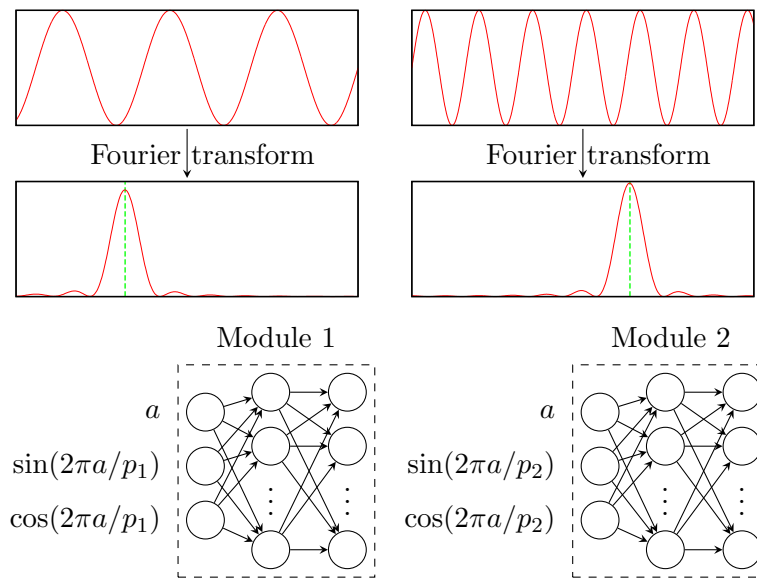


Figure 8.4: Block space is split into regions which are handled separately

feasible, about 3000. Thankfully, many non-contiguous regions have the same spectra. This appears to be caused partly by serpentines switching between sides of the platter. Grouping regions together by spectrum allows us to reuse subnets across regions, reducing the number of subnets. This is important because it reduces the number of parameters in the model, which means the model can be trained with less data and will generalize better.

Regions that have the same spectra may be out of phase with each other, so the input needs to be adjusted to reuse the subnets effectively. Phase offset parameters are added that shift the sines and cosines for each region, but otherwise the inputs and subnet are the same across regions with the same spectra.

To be formal: The raw input for a particular instance is a sequence of blocks $x = \{x_1, x_2, \dots, x_m\}$. (Currently, $m = 2$.) Each block x_m lies in a region ρ with spectrum s which consists of K periods $p_{s,1}, p_{s,2}, \dots, p_{s,K}$. For each block x_m in the input, the input is augmented with $\{\cos \omega_1, \sin \omega_1, \cos \omega_2, \sin \omega_2, \dots, \cos \omega_K, \sin \omega_K\}$. Without subnet reuse, we use:

$$\omega_k(x_m) = 2\pi x_m / p_{s,k} \tag{8.2}$$

When subnets are reused, we allow for different regions having the same spectrum to have different phases, with a per-period phase offset $\phi_{\rho,k}$ and/or per-region phase offset ψ_ρ . Then the angles are:

$$\omega_k(x_m) = 2\pi x_m / p_{s,k} + \phi_{\rho,k} + \psi_\rho \tag{8.3}$$

8.3 Period composition

As detailed in section 8.1, the Fourier transform is difficult to calculate for our problem. Furthermore, because the data contains many large discontinuities, the Fourier transform is not well-behaved. In particular, a dominant pattern in the data is a sawtooth, whose Fourier transform has harmonics whose coefficients decay as $1/n$. In other words, a very large number of harmonics have significant power. This masks even the fundamental frequency of other waves with lower power.

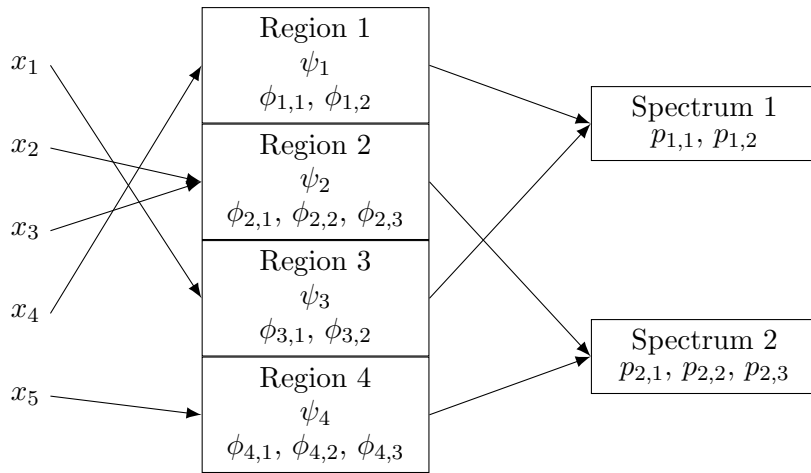


Figure 8.5: Mapping from request blocks to sets of periods

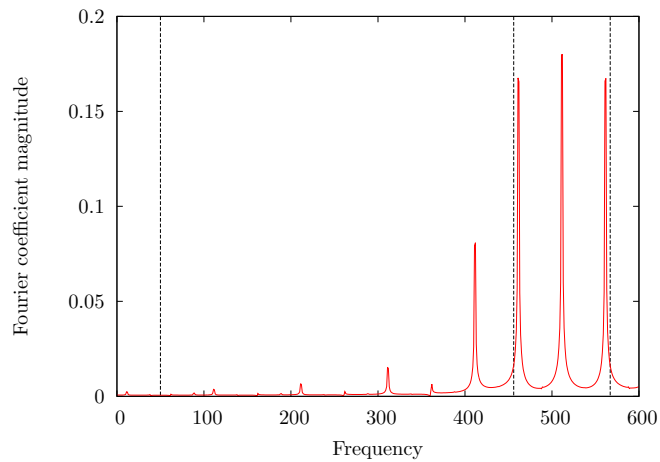


Figure 8.6: Fourier transform of signal has useful frequencies at the marked locations, but correlation between usefulness and Fourier coefficient magnitude is weak.

Signals which are not simple and precisely periodic may also have useful frequencies that do not show up in the Fourier transform. For example, consider:

$$f(x) = \begin{cases} \sin(456 \cdot 2\pi x + \psi_k) & \sin(50 \cdot 2\pi x) < 0 \\ \sin(567 \cdot 2\pi x + \psi_k) & \text{otherwise} \end{cases} \quad (8.4)$$

where

$$k = \lfloor 2 \cdot 50x \rfloor \quad (8.5)$$

with ψ defined to make f continuous. In other words, the frequency of this signal switches between 456 and 567, with the switching occurring with frequency 50. The Fourier transform of this function is seen in fig. 8.6. None of the frequencies mentioned correspond exactly to a peak in the Fourier transform. Of course, the Fourier transforms of individual zones of this function have sharp peaks at the appropriate frequencies, but in the real data the locations of the boundaries and even the number of zones are not known ahead of time.

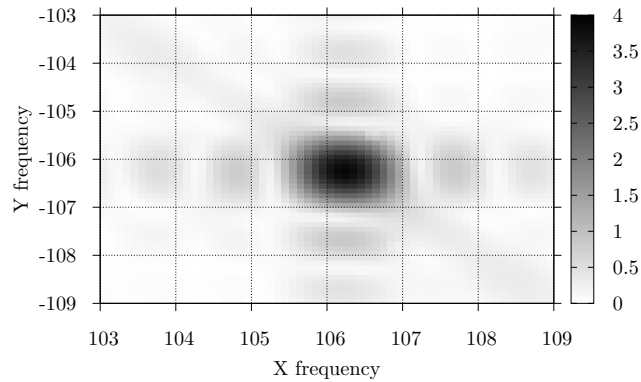


Figure 8.7: Section of the Fourier transform of access times from the first few tracks, with darker colors indicating higher magnitude. Note that the peak is not centered on a pair of integer coordinates.

Another problem is that our data contains *off-grid* frequencies (fig. 8.7), or frequencies which are not exact integers. Many sparse Fourier algorithms do not work well with off-grid frequencies [17], and will instead report many frequencies near the

frequency of interest, but with lower magnitudes. A sparse Fourier algorithm which is two-dimensional, supports off-grid frequencies, is computationally efficient, is efficient in the number of samples required, has reasonable constant factors, and which generally scales to the size of our problem is currently unknown.

Because of the difficulties of computation and limitations in the result of the Fourier transform, we decided that the best approach was to let the neural net compute the frequencies for itself. Computing periodic waveforms directly is a difficult problem (see checkerboard problem [100] and two-spirals problem [37, 95]), so we feed the neural net a base set of sinusoids from which it can compute arbitrary sinusoids.

From basic trigonometry, the angle sum formulas for sine and cosine are:

$$\cos(\alpha + \beta) = \cos(\alpha) \cos(\beta) - \sin(\alpha) \sin(\beta) \quad (8.6)$$

$$\sin(\alpha + \beta) = \sin(\alpha) \cos(\beta) + \cos(\alpha) \sin(\beta) \quad (8.7)$$

Adding a time parameter gives:

$$\cos((\alpha + \beta)t) = \cos(\alpha t) \cos(\beta t) - \sin(\alpha t) \sin(\beta t) \quad (8.8)$$

$$\sin((\alpha + \beta)t) = \sin(\alpha t) \cos(\beta t) + \cos(\alpha t) \sin(\beta t) \quad (8.9)$$

which means that a neural net can construct sines and cosines with frequency $\alpha + \beta$ very simply given sines and cosines with frequencies α and β . By extension, sinusoids of any integer frequency n can be constructed by a neural net with depth $O(\log \log n)$ given $O(\log n)$ input sinusoids with frequencies of the form 2^k . Of course, this is not necessarily the mechanism by which the neural net computes its output, but it serves as an explanation to show that the idea has theoretical merit.

Since the number of sectors, and therefore the number of periods, is on the order of a billion, and since we use frequencies of the form 2^k , the number of input frequencies is roughly $\log_2 10^9 \approx 30$. Combining an arbitrary subset of these may be too difficult an operation for a neural net with a reasonable number of neurons to perform in a single layer, so we allow the neural net to have multiple hidden layers. This corresponds to the $O(\log \log n)$ depth mentioned in the previous paragraph.

To demonstrate that this is possible, we constructed a simple neural net (no subnets) to predict a sine wave with a non-integer frequency (fig. 8.8). Even though the

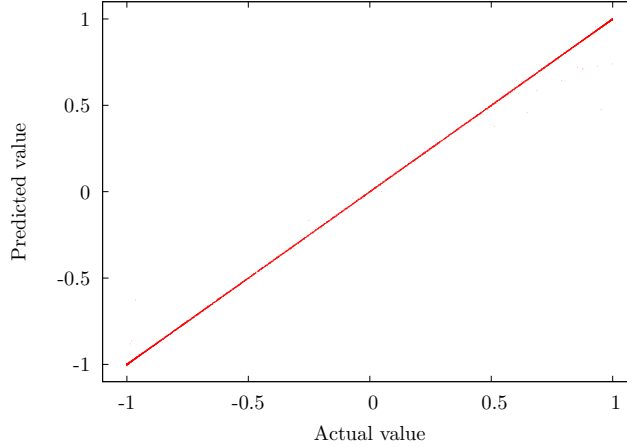


Figure 8.8: Predicted versus actual value of $\sin(1000.47 \cdot 2\pi x)$ generated by a neural net. The neural net has 4 hidden layers of 100 neurons each. The input consists of sines and cosines of x with frequencies of the form 2^k with $0 \leq k < 30$. The mean absolute error is 0.00215. This shows that the neural net can generate non-integer frequencies well.

frequency was not fed in directly, and even though it was not an integer, the neural net was able to construct a sine wave with that frequency. The neural net actually performs the task extremely well, with a mean absolute error of 0.00215. This is an interesting direction for future research (see chapter 10), since an efficient method for performing a sparse Fourier transform fitting our requirements (extremely sparse dataset, non-integer frequencies, multidimensional) is not yet known.

8.4 Learning to compose periods

The problem of learning to compose periods, find the difference between the phases of the composed periods, and generate an appropriate output waveform is too difficult for a neural net to learn all at once. In fact, our attempts to do so failed. One problem is that there is a large number of inputs, about 30 sines and cosines per input request, and learning to combine the appropriate subset of these is a difficult problem in and of itself. Another problem is that different zones of the disk have different track lengths, meaning that different composite periods are required, so the neural net must

learn to switch between them.

Our intent is that these subproblems are handled by different parts of the neural net, whose architecture is shown in fig. 7.2. Specifically, period choice and composition should be performed by the input subnets, and the phase comparison and output generation should be performed by the output subnet. By breaking the training into stages that focus on parts of the neural net, the difficulty of each stage of training is reduced. This idea of multi-stage training shares a conceptual thread with the pre-training used with deep learning for classifying images [36]. In these approaches, the researchers pretrain layers close to the input before moving to the main training phase. Although their motivation is different (making use of unlabeled data), it has much the same effect.

One approach is to start by training with data from a tiny portion of the drive. We then increase the covered portion of the drive, training along the way, until the entire drive is covered. When the covered portion is much less than one composite period, the phase is approximately linear in the sector number. This means that approximating the composite period is very easy, and most of the training is focused on the output subnet, which computes the phase difference and output waveform. As more of the drive is covered, the difficulty of approximating the composite periods is gradually increased. Unfortunately, this approach does not work very well. When we stop expanding the coverage and focus on reducing the error, we are unable to reduce the error to a reasonable amount. We believe this is because the neural net gets stuck in a configuration that is good for the early parts of training, essentially in a local minimum and unable to change to a configuration that is good for the larger area.

A better approach is to train the period composition first, and then train the phase differencing. This is accomplished by capturing a training set where the start sector is always sector 0, and the end sector varies. Since one of the sectors is constant, computing the difference is trivial, and training is focused on learning to generate the composite periods. The start sector can then be allowed to vary more and more until it can be any sector on the drive.

We use a workload that consists of random reads, except with half of the reads randomly changed to sector 0. This means that (previous, current) sector pairs in the

workload consist of a mix of $(0, 0)$, $(0, x)$, $(x, 0)$, and (x, y) . The $(0, 0)$ pairs are discarded, since they are useless, leaving an even mix of $(0, x)$, $(x, 0)$, and (x, y) . Training on these all at once avoids the local minimum problem in the previous approach, which seems to be caused by changing characteristics of the training set over time. Doing so also avoids the question of when to switch data sets. Although the data is provided all at once, the neural net presumably learns the $(0, x)$ and $(x, 0)$ pairs first, since they are easier, and then learns the (x, y) pairs. Testing is performed only against the (x, y) pairs to eliminate the unrealistic bias of the $(0, x)$ and $(x, 0)$ pairs.

8.5 Period composition results

Predicted versus actual access times when using period composition can be seen in fig. 8.9. This is for a workload of 40,000 half-zero random reads over the first zone of HD1 (237,631 sectors). Testing was performed against a purely random read workload. The subnet hidden layer sizes were 39, 56, 164, 9, and 5. The subnet output layer size was 18, and the main net hidden layer sizes were 51 and 15. The mean absolute error is 0.333. If we correct for predictions which are off by exactly one rotation, the mean absolute error falls to 0.176.

When automatically tuning hyperparameters, the genetic algorithm consistently chooses deep rather than shallow architectures. It makes sense that deep neural networks are more efficient for this task than shallow networks because of the compositional nature of the task. This is also in line with the $O(\log \log n)$ depth mentioned in section 8.3.

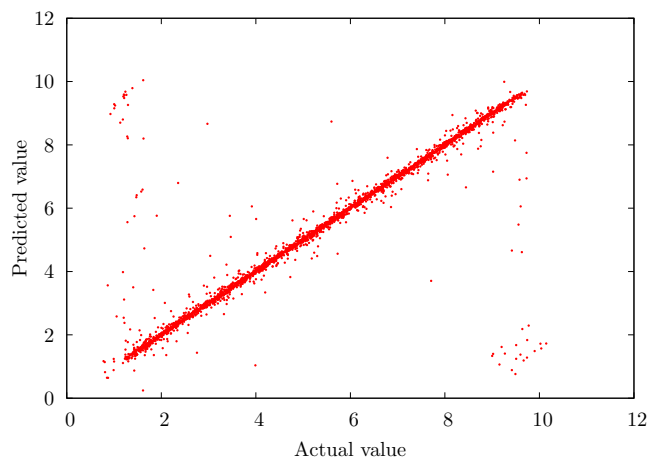


Figure 8.9: Predicted versus actual access times over the first 237,631 sectors of HD1 using period composition. The mean absolute error is 0.333. If we correct for predictions which are off by exactly one rotation, the mean absolute error falls to 0.176. (This plot is identical to fig. 9.2a.)

Chapter 9

Discontinuities

9.1 Cause and appearance

A basic characteristic of neural nets is that they assume that the function being approximated is continuous, and so they do not handle discontinuities well. This is very unfortunate, because the function we are approximating is full of large discontinuities. Thankfully, these discontinuities have a structure that makes them removable.

The discontinuities are caused by the rotational nature of a hard disk drive. In this paragraph, seek time is defined to include settle time and head switch time. If the seek time is less than the rotation time, then the head simply waits on the destination track until the destination sector rotates underneath it. If the seek time is greater than the rotation time, the head will miss the sector and must wait for an additional rotation. The discontinuities in the access time function are (mostly) places where the seek time is almost exactly equal to the rotation time, and a slight change in the sector number, which implies a slight change in rotation time, causes a rotation miss. This explanation predicts that the discontinuities will have a height which is equal to the rotational period of the drive, which is borne out by data.

9.2 Removal

Consider a continuous section of the access time function between two discontinuities. This section runs from a minimum to a maximum (or from a maximum to a

minimum). The section can be translated and scaled so that it runs from $-\pi$ to π . If it is now treated as an angle, instead of as a real number, then $-\pi$ and π are equal, so the discontinuity is gone. This is shown geometrically in fig. 9.1.

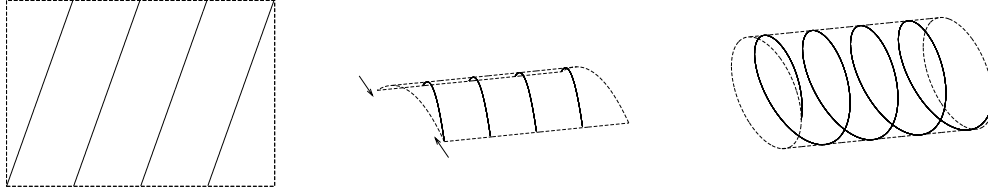


Figure 9.1: A 2D sawtooth can be rolled up into a 3D helix to remove its discontinuities.

In more concrete terms, consider the sawtooth $f(x)$ which has a minimum of $-h$ and a maximum of h . Instead of predicting $f(x)$ directly, we predict $c = \cos\left(\frac{\pi}{h}f(x)\right)$ and $s = \sin\left(\frac{\pi}{h}f(x)\right)$. These are continuous functions of x , so the neural net sees no discontinuities. The value of $f(x)$ can be reconstructed using $\hat{f}(x) = \frac{h}{\pi}\text{atan2}(c, s)$.

The problem is slightly complicated by the fact that while the access time function looks locally like the sawtooth described, globally it has a greater range between its minimum and maximum. To account for this, we use a sliding window approach. The neural net predicts a lower bound for the access time function, l , in addition to c and s . The final predicted access time is then $\hat{f}(x) = \frac{h}{\pi}\text{atan2}(c, s) + 2kh$, where k is the smallest integer such that $\frac{h}{\pi}\text{atan2}(c, s) + 2kh \geq l$. In other words,

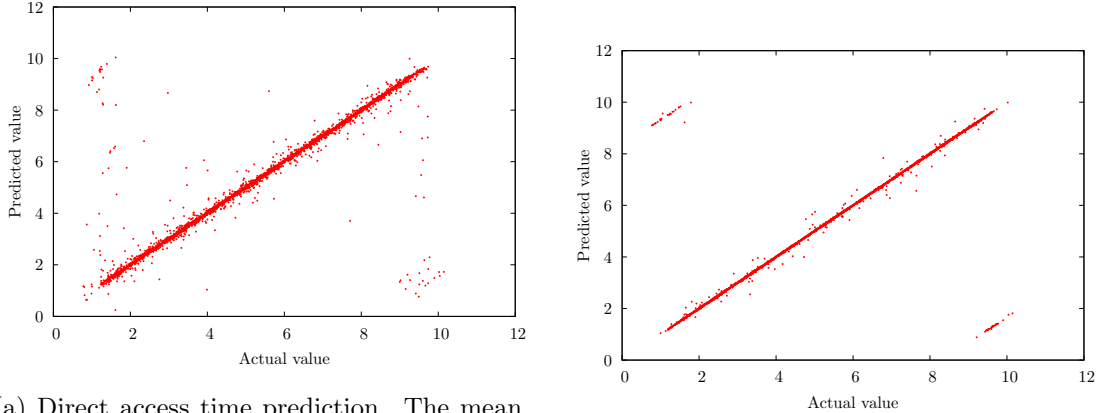
$$\hat{f}(x) = \frac{h}{\pi}\text{atan2}(c, s) + 2h \left\lceil \frac{l}{2h} - \frac{1}{2\pi}\text{atan2}(c, s) \right\rceil \quad (9.1)$$

which guarantees $\hat{f}(x) \geq l$.

9.3 Results

The result of removing discontinuities over the first zone of the drive are shown in fig. 9.2. This is for a workload of 40,000 half-zero random reads over the first zone of HD1 (237,631 sectors). (See section 8.4 for a description of the workload and its purpose.) Testing was performed against a purely random read workload. The mean absolute error is reduced significantly, from 0.333 to 0.187. If we correct for predictions

which are off by exactly one rotation, the mean absolute error falls from 0.176 to 0.029. Note that when discontinuities are removed (fig. 9.2b), predictions may be off by exactly one rotation, but they do not fall in between as they do when discontinuities are not removed (left and right side of fig. 9.2a).



(a) Direct access time prediction. The mean absolute error is 0.333. If we correct for predictions which are off by exactly one rotation, the mean absolute error falls to 0.176. (This plot is identical to fig. 8.9.)

(b) Predictions with the discontinuities removed. The mean absolute error is 0.187. If we correct for predictions which are off by exactly one rotation, the mean absolute error falls to 0.029.

Figure 9.2: Predicted versus actual access time over the first 237,631 sectors of HD1, without and with discontinuities removed.

When combining period composition, half-zero sampling, and discontinuity removal, we can get a neural net to cover roughly 60% of HD1 (fig. 9.3). (Note that only angular error and not seek error is plotted to make the plot more readable.) The plot of predicted versus actual values for the first half of the drive is seen in fig. 9.4. This snapshot was taken at iteration 36, after which improvement was minimal. Unfortunately, it took 90 hours to train to this point, which makes auto-tuning with the GA infeasible, so hyperparameters were manually tuned. (Note that the 90 hour training time is still significantly faster than the months of model creation time for existing methods (chapter 1).) The subnet hidden layer sizes were 368, 1016, 528, 244, 112, and 100. The subnet output layer size was 13, and the main net hidden layer size was 37. Given the

gains seen with auto-tuning in table 7.2, tuning these parameters would probably result in a significant improvement. A total data set size of 8,480,000 requests was used, which took about 29.5 hours to capture.

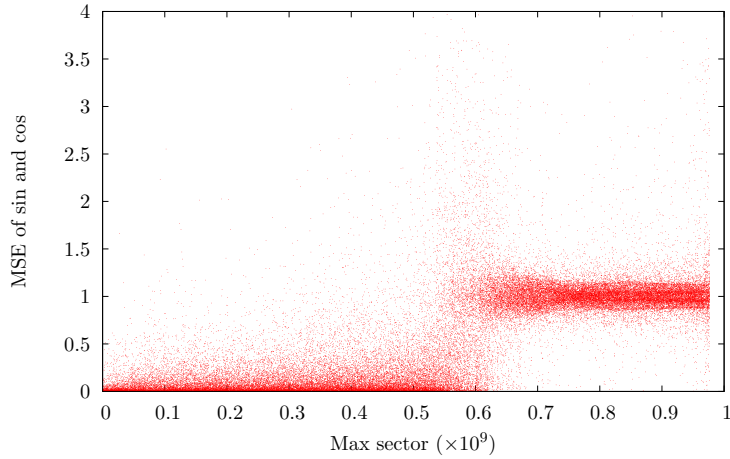


Figure 9.3: Error versus sector for HD1. The neural net was fed data sets with exponentially increasing range, all intermingled. The neural net was able to cover approximately 60% of the drive. The error stabilizes at 1 on the high end because if learning fails, the neural net will output the constant value with the lowest error, and the constant output that yields the lowest MSE is a “sin” and “cos” of 0, which gives an MSE of 1.

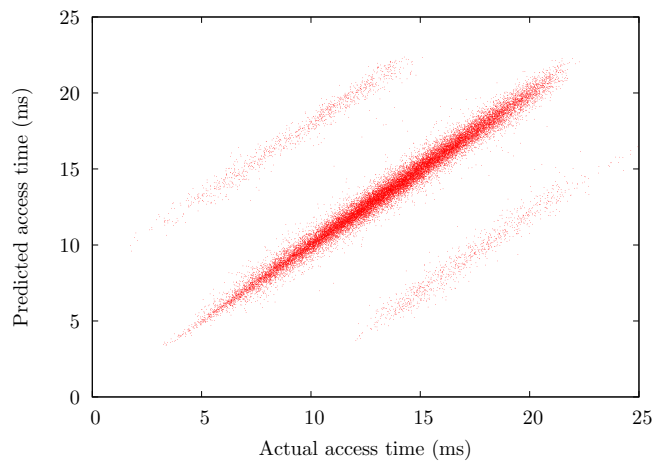


Figure 9.4: Predicted versus actual access times for the first half of HD1. The mean absolute error is 2.120 ms, and the mean absolute error when correcting for rotation misses is 0.485 ms. The real time necessary to generate each prediction is 9.5 ms.

Chapter 10

Future work

There are many, many ways this work could be extended, and we describe some of them in this chapter.

When composing periods by providing 2^k -period inputs (see section 8.3), the latent frequencies generated are not immediately apparent. It is not clear how the generated frequencies could be determined, but doing so would provide a method for performing a sparse Fourier transform on an extremely sparse, multidimensional dataset with non-integer frequencies.

Our approach could be combined with existing approaches mentioned in section 2.1 to support more diverse workloads. Some workload characteristics can be addressed by adding input features. For example, a read/write flag [26, 109] and request size [72, 26, 109] parameters are natural extensions. Additionally, LRU stack distance has been used to approximate caching effects [59], and a sequentiality flag has been used to approximate read-ahead [26, 109]. Better handling of cross-request interactions, such as caching and read-ahead, can also be handled by providing more than just the latest and previous request to the neural net. A simple extension is to use the past k requests as input [109, 26]. An alternative would be to use a recursive neural net (RNN) to explicitly store state, which is often used when modeling time-series data [85, 45]. This would avoid imposing an artificial bound on the length of time the neural net can retain state.

Other optimization methods for training the neural net could be explored. Our experience has been that stochastic methods such as RMSProp give the fastest results.

However, stochastic methods are intrinsically difficult to parallelize, and a batch method may be preferable, especially with increasing core counts. New optimization methods are continually being developed, such as the saddle-free Newton [28] (and its stochastic version, equilibrated stochastic gradient descent (ESGD) [27]), stochastic Hessian-free optimization (SHF) [61], and target propagation [11]. (Note that these have not yet been peer-reviewed except for the saddle-free Newton.)

Better hyperparameter tuning methods could be explored. Surrogate-based optimization [40], also known as sequential model-based optimization (SMBO), is of particular interest, since the function evaluation (training and evaluating a neural net) is very expensive. Another possibility is gradient-based hyperparameter optimization [79].

Transfer learning [89] could allow for faster or more accurate learning on new hard drives, once a model has been trained on other hard drives. Such a model would essentially have a sense of how hard drives in general behave, not just the current one being modeled.

A feedback loop could be incorporated if a reference device is available at run time. This would benefit quality of service and other scheduling applications. The model would then be able to predict what will happen based not only on requests issued, but also on the latencies of past requests. This may allow simpler models to be used.

Higher-level interfaces such as POSIX calls could be modeled. Although this would obscure block-level information, it adds file system contextual information which is unavailable at the block level. For example, file content reads may tend to cache differently than metadata reads, even though these would be difficult to distinguish in terms of block reads. Being coarser-grained, POSIX-level modeling may require less overhead than block-level modeling. Furthermore, it would allow non-block-based file systems such as NFS or future file systems based on non-volatile memory (NVM).

If the issues with explicitly identifying periods, such as by using Fourier analysis, can be worked out (see section 8.3), then the question of the relevance of higher-order periods becomes important again. Future research could explore how easy higher-order periods are to identify and whether they are best used as additional neural net inputs, as help in segmenting the drive into zones, or in other ways.

More domain knowledge could be incorporated into the neural network. This

is already done with the separate subnets and the weight sharing between them (see section 7.1.2) and with discontinuity removal (see chapter 9). Another piece of knowledge which could theoretically be incorporated is the fact that the time to read sector a then b plus the time to read sector b then a is an integer multiple of the rotational latency.

This fact is only applicable for hard drives, and other such facts may be limited to specific types of device, moving the model from black-box to gray-box.

More architectural flexibility could be allowed in the neural net. Currently, the net consists of fully-connected layers with no skip-layer connections (see fig. 7.2). Arbitrary network topologies could be allowed by using an algorithm such as NEAT [101]. The transfer function could also be tuned, possibly even allowing a different transfer function for each neuron. An even more esoteric possibility is using product units instead of sum units [33].

Our approach is currently approximately 14 times slower than DiskSim, and both use only a few kilobytes of model state. Measuring process memory usage is unhelpful, because different codebases will have different constant overhead, and because memory used to store trace data is model independent. The neural net has 4928 neurons (counting 64 input neurons per subnet for the sines and cosines, and counting the subnets separately), each of which has one number as state (its activation), which takes a total of 19 KB of storage using single-precision floats. DiskSim’s memory usage varies based on the number of events in its queue, but the size of the structs containing disk state (ignoring statistics structs) plus all queued events averages roughly 15 KB to 20 KB. This is comparing our final neural net, which covers a 250 GB range and is unoptimized, to the Cheetah 9LP model which comes with DiskSim and is a 9.1 GB disk. Automatically tuning the neural net and penalizing for evaluation time should significantly speed up the neural net, especially if sparser network topologies are allowed. Benchmarking our MSST results [24] shows a 10% speedup of the automatically tuned neural net over the manually tuned neural net. The speedup is likely to be more dramatic for our final neural net, since the longer evaluation time meant that fewer tuning iterations were performed. Further improvements, such as the use of vector CPU instructions or GPU processing, especially if processing multiple requests at once, should also speed up the neural net. In fact, careful optimization of a neural net,

including use of SIMD instructions, can speed up evaluation by a factor of 10, even without using a GPU [107].

Other devices could be modeled, not just hard disk drives. Obvious extensions include RAID arrays, raw flash devices, or non-volatile memory. SSDs would be difficult to model with low per-request error because of the flash translation layer (FTL).

Chapter 11

Conclusion

We have created methods for successful machine learning in a difficult domain, where sampling is, by necessity, extremely sparse. Storage devices, and particularly hard disk drives, have a large amount of internal structure which is not readily taken advantage of by existing general-purpose machine learning algorithms. We have found high-level assumptions, such as periodicity and the existence of a logical-to-physical mapping, which allow machine learning algorithms to make use of the structure without requiring a human expert to hand-craft specific features.

Black-box approaches to storage device performance modeling, used by many other researchers in addition to ourselves, stand in stark contrast to white-box approaches such as DiskSim. While potentially very accurate as long as properly configured, and as long as the codebase reflects reality, DiskSim is difficult to configure and maintain due to its tight coupling with the often proprietary internal details of complex devices.

While this thesis looks at predicting access times for random reads, models of read-only workloads have several use cases. First of all, many read-only workloads exist. These include static web serving (assuming access logs are on a separate device), media serving, and analytic query processing. Second, if write-back is disabled, reads and writes have nearly identical performance in hard disk drives. Furthermore, the approach in this thesis could be combined with existing approaches for storage latency modeling to create a model capable of predicting individual request latencies for general read/write workloads.

We have shown that accurately predicting individual request access times with black-box models is possible for random read workloads on hard drives. Periodicity is inherent in the task, caused partly by the rotational nature of hard disk drives, but also more generally by the benefits of a regularly repeating design. This periodicity manifests itself in the access time function and makes the problem difficult for unmodified machine learning algorithms. Recognition of periodicity and explicitly addressing it (by augmenting the input vector with sines and cosines) is crucial for success on this task.

We have also shown that neural nets are a good choice of model for this task. The structure inherent in neural nets makes them amenable to modifications such as weight sharing, which reduces the number of parameters, thus speeding up learning and reducing error.

Deep neural networks are more efficient for this task than shallow networks, probably because of the compositional and periodic nature of this task. When automatically tuning hyperparameters, the genetic algorithm consistently chooses deep rather than shallow architectures. Although training deep networks would have been inefficient a few years ago, relatively recent advances such as RMSProp make them more effective.

Discontinuities in the access time function are a major hurdle, especially for models such as neural nets which assume a smooth function. Although reasonable results can be achieved with an unmodified learning algorithm, embedding the output in a higher-dimensional space can remove the discontinuities and reduce error. This is made possible by knowing the rotational latency of a hard drive, which is provided by the manufacturer and also easy to determine empirically.

Although neural nets are proven to be universal function approximators, the idea that applying a plain neural net to unprocessed input data will automatically yield good results is naive. The universal approximation theorem says nothing about the number of hidden units necessary, and using a single-hidden-layer network with unprocessed inputs may require an infeasible number of hidden units and an infeasible amount of training data to yield useful results. Furthermore, although a neural network with sufficiently low error conceptually exists for any given problem, actually finding the neural network can be a difficult task. Training is usually performed using a gradient-based optimization algorithm, which is a greedy algorithm, and no guarantee is possible

that the global minimum will be found. In addition to local minima, training can also be confounded by saddle points, plateaus, deep ravines, and other undesirable features in the parameter space. Domain knowledge, good strategies, good algorithms, tuning, and patience are required to achieve good results.

Bibliography

- [1] Quantum atlas 10k ii. http://www.seagate.com/staticfiles/maxtor/en_us/documentation/data_sheets/atlas_10k_ii_datasheet.pdf, 2000. AI-IDS0600.
- [2] Seagate breaks capacity ceiling with world's first 3 terabyte external desktop drive. <http://media.seagate.com/2010/06/seagatetechnology/seagate-breaks-capacity-ceiling-with-worlds-first-3-terabyte-external-desktop-drive/>, June 2010.
- [3] WD caviar green series disti spec sheet. <http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701229.pdf>, January 2012. 2879-701229-A25.
- [4] Hussein A Abbass. Speeding up backpropagation using multiobjective evolutionary algorithms. *Neural Computation*, 15(11):2705–2726, 2003.
- [5] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating goliath storage systems with david. In *Proceedings of the 9th USENIX conference on File and storage technologies*, FAST'11, pages 15–15, Berkeley, CA, USA, 2011. USENIX Association.
- [6] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance.
- [7] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, 2005.

- [8] J Scott Armstrong and Fred Collopy. Error measures for generalizing about forecasting methods: Empirical comparisons. *International Journal of Forecasting*, 8(1):69–80, 1992.
- [9] D. Ashby, B. Norman, and K. Tan. *Barracuda 4LP Family - Product Manual*. Seagate, D edition, Feb 1998.
- [10] Babak Behzad, L Huong Vu Thanh, Joseph Huchette, Surendra Byna, R Aydt Prabhat, Quincey Koziol, and Marc Snir. Taming parallel I/O complexity with auto-tuning. In *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)*, 2013.
- [11] Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.
- [12] Yoshua Bengio and Paolo Frasconi. Input-output HMMs for sequence processing. *Neural Networks, IEEE Transactions on*, 7(5):1231–1249, 1996.
- [13] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.
- [14] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 2012.
- [15] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nimes*, 91:8, 1991.
- [16] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.
- [17] Petros Boufounos, Volkan Cevher, Anna C Gilbert, Yi Li, and Martin J Strauss. What’s the frequency, Kenneth?: Sublinear Fourier sampling off the grid. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 61–72. Springer, 2012.
- [18] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

- [19] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and Contributors. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. Carnegie Mellon University, Pittsburgh, PA, May 2008.
- [20] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.
- [21] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.
- [22] Ying Chen, Windsor W. Hsu, and Honesty C. Young. Logging RAID - an approach to fast, reliable, and low-cost disk arrays. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1302–1311. Springer Berlin Heidelberg, 2000.
- [23] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.
- [24] Adam Crume, Carlos Maltzahn, Lee Ward, Thomas Kroeger, and Matthew Curry. Automatic generation of behavioral hard disk drive access time models. In *Mass Storage Systems and Technologies, 2014. MSST 2014. 30th IEEE Conference on*. IEEE, 2014.
- [25] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [26] Chenjun Dai, Guiquan Liu, Lei Zhang, and Enhong Chen. Storage device performance prediction with hybrid regression models. In *PDCAT'12*, Beijing, China, December 2012.
- [27] Yann N Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. RM-SProp and equilibrated adaptive learning rates for non-convex optimization. *arXiv preprint arXiv:1502.04390*, 2015.
- [28] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem

- in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 2933–2941, 2014.
- [29] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and Nawab Ali. Attribute storage design for object-based storage devices. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 263–268. IEEE, 2007.
- [30] Persi Diaconis and Ronald L Graham. Spearman’s footrule as a measure of disarray. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 262–268, 1977.
- [31] Jesus Diaz. This is the largest SATA drive in the world. <http://gizmodo.com/5667594/this-is-the-largest-sata-drive-in-the-world>, October 2010.
- [32] Cagdas Dirik and Bruce Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 279–289. ACM, 2009.
- [33] Richard Durbin and David E Rumelhart. Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation*, 1(1):133–142, 1989.
- [34] David Elizondo, Gerrit Hoogenboom, and RW McClendon. Development of a neural network model to predict daily solar radiation. *Agricultural and Forest Meteorology*, 71(1):115–132, 1994.
- [35] Miguel A Erazo, Ting Li, Jason Liu, and Stephan Eidenbenz. Toward comprehensive and accurate simulation performance prediction of parallel file systems. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.
- [36] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of

- unsupervised pre-training. In *International Conference on artificial intelligence and statistics*, pages 153–160, 2009.
- [37] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, February 1990.
- [38] Silvia Ferrari and Robert F Stengel. Smooth function approximation using neural networks. *Neural Networks, IEEE Transactions on*, 16(1):24–38, 2005.
- [39] Dario Floreano, Peter Dürri, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.
- [40] Alexander IJ Forrester and Andy J Keane. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45(1):50–79, 2009.
- [41] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.
- [42] J.D. Garcia, L. Prada, J. Fernandez, A. Nunez, and J. Carretero. Using black-box modeling techniques for modern disk drives service time simulation. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 139 –145, april 2008.
- [43] M.W. Gardner and S.R. Dorling. Neural network modelling and prediction of hourly NO_x and NO_2 concentrations in urban air in london. *Atmospheric Environment*, 33(5):709 – 719, 1999.
- [44] John M. Gauch. Ch4 - Fourier transform. <http://www.csce.uark.edu/~jgauch/5683/notes/ch04a.pdf>.
- [45] Felix Gers. Long short-term memory in recurrent neural networks. *Unpublished PhD dissertation, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*, 2001.
- [46] Jongmin Gim and Youjip Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *Trans. Storage*, 6:6:1–6:26, July 2010.

- [47] William L Goffe, Gary D Ferrier, and John Rogers. Global optimization of statistical functions with simulated annealing. *Journal of Econometrics*, 60(1):65–99, 1994.
- [48] John Linwood Griffin, Jiri Schindler, Steven W Schlosser, John S Bucy, and Gregory R Ganger. Timing-accurate storage emulation. In *FAST*, volume 2, page 6, 2002.
- [49] Allon Guez and Ziauddin Ahmad. Solution to the inverse kinematics problem in robotics by neural networks. In *Neural Networks, 1988., IEEE International Conference on*, pages 617–624. IEEE, 1988.
- [50] Martin T Hagan and Mohammad B Menhaj. Training feedforward networks with the Marquardt algorithm. *Neural Networks, IEEE Transactions on*, 5(6):989–993, 1994.
- [51] Geoffrey Hinton. A practical guide to training restricted Boltzmann machines. Technical report, University of Toronto, August 2010. UTML TR 2010-003.
- [52] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [53] Hai Huang, Wanda Hung, and Kang G Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. *ACM SIGOPS Operating Systems Review*, 39(5):263–276, 2005.
- [54] A. Hylick, R. Sohan, A. Rice, and B. Jones. An analysis of hard drive energy consumption. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1 –10, sept. 2008.
- [55] Christian Igel and Michael Hüsken. Empirical evaluation of the improved rprop learning algorithms. *Neurocomputing*, 50:105–123, 2003.
- [56] Sarangapani Jagannathan and Frank L Lewis. Multilayer discrete-time neural-net controller with guaranteed performance. *Neural Networks, IEEE Transactions on*, 7(1):107–130, 1996.

- [57] Yaochu Jin, Tatsuya Okabe, and Bernhard Sendhoff. Neural network regularization and ensembling using multi-objective evolutionary algorithms. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 1–8. IEEE, 2004.
- [58] Myoungsoo Jung, Ellis H Wilson III, and Mahmut Kandemir. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 404–415. IEEE Press, 2012.
- [59] Terence Kelly, Ira Cohen, Moises Goldszmidt, and Kimberly Keeton. Inducing models of black-box storage arrays. Technical Report HPL-2004-108, HP Laboratories, Palo Alto, CA, June 2004.
- [60] Lars Kindermann, Achim Lewandowski, and Peter Protzel. A framework for solving functional equations with neural networks. In *Proceedings of Neural Information Processing (ICONIP2001)*, volume 2, pages 1075–1078. Fudan University Press, Shanghai, 2001.
- [61] Ryan Kiros. Training neural networks with stochastic hessian-free optimization. *arXiv preprint arXiv:1301.3641*, 2013.
- [62] K Kosanovich, A Gurumoorthy, E Sinzinger, and M Piovoso. Improving the extrapolation capability of neural networks. In *Intelligent Control, 1996., Proceedings of the 1996 IEEE International Symposium on*, pages 390–395. IEEE, 1996.
- [63] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks are like snowflakes: No two are alike. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS’13, pages 14–14, Berkeley, CA, USA, May 2011. USENIX Association.
- [64] Jaimyoung Kwon, Benjamin Coifman, and Peter Bickel. Day-to-day travel-time trends and travel-time prediction from loop-detector data. *Transportation Research Record: Journal of the Transportation Research Board*, 1717(1):120–129, 2000.

- [65] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480. ACM, 2007.
- [66] HYAF Laurent and Ronald L RIVEST. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 1976.
- [67] Abigail S. Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. A performance model of zoned disk drives with I/O request reordering. In *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, QEST '09, pages 97–106, Washington, DC, USA, 2009. IEEE Computer Society.
- [68] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*, pages 9–50. Springer-Verlag, 1998.
- [69] Frank Hung-Fat Leung, Hak-Keung Lam, Sai-Ho Ling, and Peter Kwong-Shun Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79–88, 2003.
- [70] Roger J Lewis. An introduction to classification and regression tree (CART) analysis. In *Annual Meeting of the Society for Academic Emergency Medicine in San Francisco, California*, pages 1–14. Citeseer, 2000.
- [71] Mingqiang Li and Jiwu Shu. DACO: A high-performance disk architecture designed specially for large-scale erasure-coded storage systems. *Computers, IEEE Transactions on*, 59(10):1350–1362, 2010.
- [72] D.J. Lingenfelter, A. Khurshudov, and D.M. Vlassarev. Efficient disk drive performance model for realistic workloads. *Magnetics, IEEE Transactions on*, 50(5):1–9, May 2014.

- [73] Richard Lippmann. An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2):4–22, 1987.
- [74] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.
- [75] Liu Liu, Zhen Han Liu, Lu Xu, and JunWei Zhang. FBctrl - a novel approach for storage performance virtualization. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 268–272. IEEE, 2011.
- [76] Liu Liu, Lu Xu, Zhen Han Liu, and JunWei Zhang. QClock: An interposed scheduling algorithm for performance virtualization in shared storage systems. In *Networked Computing (INC), 2011 The 7th International Conference on*, pages 17–21. IEEE, 2011.
- [77] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI 2012*, Pacific Grove, CA, April 16 - 20 2012.
- [78] Yonggang Liu, Renato Figueiredo, Dulcardo Clavijo, Yiqi Xu, and Ming Zhao. Towards simulation of parallel file system scheduling algorithms with PFSSim. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O (May 2011)*, 2011.
- [79] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv preprint arXiv:1502.03492*, 2015.
- [80] Robert J McEliece, Edward C Posner, Eugene R Rodemich, and Santosh S Venkatesh. The capacity of the Hopfield associative memory. *Information Theory, IEEE Transactions on*, 33(4):461–482, 1987.
- [81] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *SIGMETRICS 2007*, 2007.

- [82] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.
- [83] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [84] L. Newman and J. Nowitzke. *Cheetah 9LP Family - Product Manual*. Seagate, C edition, August 1998.
- [85] Gregory Noone and Stephen D Howard. Investigation of periodic time series using neural networks and adaptive error thresholds. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1541–1545. IEEE, 1995.
- [86] Alberto Núñez, Javier Fernández, Rosa Filgueira, Félix García, and Jesús Carretero. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simulation Modelling Practice and Theory*, 20(1):12–32, 2012.
- [87] Ron Oldfield. Personal communication, January 2013.
- [88] Jiaming Pan, Jiwu Shu, Suqin Zhang, and Weimin Zheng. Design and implementation of scsi target emulator. *Tsinghua Science & Technology*, 11(1):38–43, 2006.
- [89] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010.
- [90] Bruce E Rosen. Ensemble learning using decorrelated neural networks. *Connection Science*, 8(3-4):373–384, 1996.
- [91] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, mar 1994.
- [92] Thomas M Ruwart and Yingping Lu. Performance impact of external vibration on consumer-grade and enterprise-class disk drives. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on*, pages 307–315. IEEE, 2005.

- [93] Jiri Schindler, John Bucy, and Greg Ganger. Comments in `extract_layout.c` in DiskSim 4.0 source. <http://www.pdl.cmu.edu/PDL-FTP/DriveChar/disksim-4.0-with-dixtrac.tar.gz>.
- [94] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.
- [95] Yi Shang and Benjamin W Wah. Global optimization for neural network training. *Computer*, 29(3):45–54, 1996.
- [96] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.
- [97] Dimitris Skourtis, Shinpei Kato, and Scott Brandt. Qbox: guaranteeing i/o performance on black box storage systems. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 73–84. ACM, 2012.
- [98] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.
- [99] Josep M Sopena, Enrique Romero, and Rene Alquezar. Neural networks with periodic and monotonic activation functions: a comparative study in classification problems. In *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, volume 1, pages 323–328. IET, 1999.
- [100] Donald F. Specht and P.D. Shapiro. Generalization accuracy of probabilistic neural networks compared with backpropagation networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume i, pages 887–892 vol.1, 1991.
- [101] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

- [102] Vasily Tarasov, Gyumin Sim, Anna Povzner, and Erez Zadok. Efficient i/o scheduling with accurately estimated disk drive latencies. In *OSPERT 2013*, Paris, France, July 9 2013.
- [103] Alexander Thomasian. Survey and analysis of disk scheduling methods. *ACM SIGARCH Computer Architecture News*, 39(2):8–25, 2011.
- [104] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.
- [105] Fok Hing Chi Tivive and Abdesselam Bouzerdoum. Texture classification using convolutional neural networks. In *TENCON 2006. 2006 IEEE Region 10 Conference*, pages 1–4. IEEE, 2006.
- [106] Julian Turner. Effects of data center vibration on compute system performance. In *Proceedings of the First USENIX conference on Sustainable information technology*, SustainIT’10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.
- [107] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, 2011.
- [108] Elizabeth Varki, Allen Hubbe, and Arif Merchant. Improve prefetch performance by splitting the cache replacement queue. In *Advanced Infocomm Technology*, pages 98–108. Springer, 2013.
- [109] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *MASCOTS 2004*, 2004.
- [110] Sam Waugh and Anthony Adams. A practical comparison between quickprop and backpropagation. In *The Eighth Australian Conference on Neural Networks*, pages 143–147. Citeseer, 1997.
- [111] Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995.

- [112] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research*, 30(1):79, 2005.
- [113] K-W Wong, C-S Leung, and S-J Chang. Use of periodic and monotonic activation functions in multilayer feedforward neural networks trained by extended Kalman filter algorithm. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 149, pages 217–224. IET, 2002.
- [114] Tao Xie and Yao Sun. Dynamic data reallocation in hybrid disk arrays. *Parallel and Distributed Systems, IEEE Transactions on*, 21(9):1330–1341, 2010.
- [115] Hao Yu and BM Wilamowski. Levenberg-marquardt training. *The Industrial Electronics Handbook*, 5:1–15, 2011.
- [116] Zarita Zainuddin and Ong Pauline. Function approximation using artificial neural networks. *WSEAS Transactions on Mathematics*, 7(6):333–338, 2008.
- [117] Lei Zhang, Guiquan Liu, Xuechen Zhang, Song Jiang, and Enhong Chen. Storage device performance prediction with selective bagging classification and regression tree. *Network and Parallel Computing*, pages 121–133, 2010.

Appendix A

Notation

$\mathbb{1}_P$	indicator function; 1 when the predicate P is true and 0 otherwise
\hat{x}	predicted value of x
\tilde{x}	Fourier transform of x
x_i	i th component of vector x
X_i	row i of matrix X
$X_{i,j}$	entry i, j of matrix X
x^*	optimal value of x
$x^{(t)}$	value of x at time t
$r_1 \prec_{inject} r_2$	request r_1 was issued (injected into the queue) before request r_2
$r_1 \preceq_{inject} r_2$	request r_1 was issued (injected into the queue) before request r_2 or is request r_2
$r_1 \prec_{eject} r_2$	request r_1 was ejected from the queue before request r_2
$r_1 \preceq_{eject} r_2$	request r_1 was ejected from the queue before request r_2 or is request r_2
$\lfloor x \rfloor$	floor function; “round down”

Appendix B

Symbols

i	imaginary number
i	index
k	index to particular period in a region's spectrum
K_s	number of periods in spectrum s
m	index to particular feature within an instance
M	number of input features
n	index to particular instance
N	number of instances
p_s	periods in the spectrum s
$p_{s,k}$	period k of the spectrum s
Q	maximum queue depth
r	request
R	number of regions
s	index to particular spectrum
S	number of spectra
X	$N \times M$ matrix of input data, instance/row n is the vector X_n
Y	output data; the output for instance n is Y_n
\hat{Y}	predicted output data; the predicted output for instance n is \hat{Y}_n
lat	latencies, latency of instance n is lat_n
\widehat{lat}	predicted latencies, predicted latency of instance n is \widehat{lat}_n

θ	parameters of a model
ρ	index to particular region
$\phi_{\rho,k}$	phase offset of period k in region ρ
ψ_{ρ}	phase offset of region ρ
$\omega_k(x)$	angle k of block x

Appendix C

Glossary

Some definitions may not be standardized and simply refer to how the term is used in this thesis.

access time

This is the time between a request being ejected from the on-device queue and when the head starts reading or writing. This includes seek time, settle time, and rotational time.

cylinder

Outdated term which refers to a set of tracks at a fixed distance from the spindle. This term is no longer useful because track widths can vary on opposite sides of a platter and across platters, meaning that tracks on opposite sides of a platter do not necessarily align with each other.

feature/attribute

A feature is a specific property of an instance, which may be considered a particular input or output. For example, when learning $f(a, b) = a + b$, the features are a , b , and $a + b$. Equivalent to a column in a database.

feature extraction/generation

Process of creating new features based on existing ones. Often used for dimensionality reduction, but may also be used to convert complex features into simpler ones.

instance

An instance is a tuple consisting of a value for each feature, which may be used as a training example provided to a machine learning algorithm. Equivalent to a row in a database.

response time/request latency

This is the D2C time according to blktrace, which is the interval between the OS sending the request to the driver and receiving the response. This should not include queuing time within the OS, but does include time spent in any on-device queue.

LBA/LBN

Short for Logical Block Address or Logical Block Number. One-dimensional numbering scheme for blocks, starting at 0. This is the scheme used to identify blocks between the device and the host computer.

locally periodic

If a function is locally periodic, then its domain can be partitioned such that the function is periodic when restricted to any particular partition.

NCQ

Short for Native Command Queuing. This is part of the SATA specification which allows a device to support an internal request queue with a length of up to 32 requests. This queue gives the device the ability to schedule requests based on information not available to the host.

rotational latency

Time for the platter to rotate until the desired sector is under the read/write head.

scheduler

Algorithm in a storage device that determines when, or in which order, pending requests are processed.

sector/block

Smallest unit of data transferable between the storage device and the host computer. For hard disk drives, this is typically 512 bytes.

seek latency

Time for the arm to move to the destination track.

serpentine layout

Zig-zag manner of numbering tracks to reduce head switches. This prevents seek time from being a monotonic function of difference in track numbers.

strong frequency

A frequency is considered strong if its coefficient in the latency function's Fourier transform has a large magnitude. Compare: useful frequency.

track

Collection of sectors on the same side of the platter and same distance from the center.

transfer function/activation function

Function used by a neuron in a neural net, applied to the weight sum of the inputs to generate the output.

transfer time

Time to physically transfer the data to/from the media. This is typically proportional to the size of the request.

useful frequency

A frequency is considered useful if its inclusion as an additional input decreases the model's error. Compare: strong frequency.

zone

We define a zone to be a contiguous set of logical sectors which all fall in tracks of the same length. More traditional definitions of zone may be problematic because of the fact that track lengths can vary on opposite sides of a platter or on different platters.

Appendix D

Diameter of the set of q -bounded length- n permutations

D.1 Background

Definition 1. The symmetric group S_n consists of all permutations of n unique elements. We use the set of elements $\{1, 2, \dots, n\}$, so a permutation $\pi \in S_n$ can be represented as a sequence of positive integers. If i is the index of x in π , *i.e.* $\pi_i = x$, then $\pi_x^{-1} = i$.

Definition 2. The rank of index i in permutation π is one more than the number of smaller elements to the right:

$$\text{rank}(i, \pi) = |\{j : i \leq j, \pi_i \geq \pi_j\}| \quad (\text{D.1})$$

Definition 3. The set $P_n^q \subseteq S_n$ is the set of length- n , q -bounded permutations, *i.e.* permutations that can be generated by a machine with a memory of size q . (In the literature, k is usually used instead of q .) Note that $\pi \in P_n^q$ if and only if $\max_i \text{rank}(i, \pi) \leq q$.

Definition 4. Given permutations π, σ , the distance $d(\pi, \sigma)$ is (in this document) the inversion count of $\tau = \pi^{-1}\sigma$, *i.e.* the number of pairs (i, j) such that $i < j$ and $\tau_i > \tau_j$. This is also the minimum length of a sequence of adjacent transpositions which transforms π to σ .

Definition 5. The diameter of a set S (w.r.t. adjacent transpositions) is $\max_{x, y \in S} d(x, y)$

D.2 Theorems

Lemma 1. *If $\pi \in P_n^q$, moving the maximal element in π to the end results in an element in P_n^q .*

Proof. Let $i = \pi_n^{-1}$ be the index of the maximal element. Let π' be the transformed π , *i.e.*

$$\pi'_k = \begin{cases} \pi_k & k < i \\ \pi_{k+1} & i \leq k < n \\ n & n \end{cases} \quad (\text{D.2})$$

For $j < i$, $\text{rank}(j, \pi') = \text{rank}(j, \pi) \leq q$, because rearranging the elements to the right of j does not affect its rank. For $i \leq j < n$, $\text{rank}(j, \pi') = \text{rank}(j+1, \pi) \leq q$, because adding a larger element (the maximal element) to the right of j does not affect its rank. For $i = n$, $\text{rank}(i, \pi') = 1 < q$. Since $\text{rank}(j, \pi') \leq q$ for all j , $\pi' \in P_n^q$. \square

Lemma 2. *The diameter of P_n^q is at most*

$$\text{diameter}(P_n^q) \leq \begin{cases} \frac{1}{2}n(n-1) & n \leq 2q \\ (2n-2q+1)(q-1) & n > 2q \end{cases} \quad (\text{D.3})$$

Proof. Let π and σ be permutations in P_n^q . We always have that $d(\pi, \sigma) \leq \frac{1}{2}n(n-1)$, because the number of (i, j) pairs where $i < j$ is $\frac{1}{2}n(n-1)$, which provides the bounds for the case $n \leq 2q$.

When $n = 2q$,

$$(2n-2q+1)(q-1) = (2n-n+1)\left(\frac{1}{2}n-1\right) \quad (\text{D.4})$$

$$= \frac{1}{2}(n+1)(n-2) \quad (\text{D.5})$$

$$\geq \frac{1}{2}n(n-1) \quad (\text{for } n \geq 1) \quad (\text{D.6})$$

so $d(\pi, \sigma) \leq (2n-2q+1)(q-1)$ for $n = 2q$.

Otherwise, assume $n > 2q$ and that $d(\pi, \sigma) \leq (2n'-2q+1)(q-1)$ for $n' = n-1$. Since $\pi \in P_n^q$, its maximal element is at an index $i \geq n-q+1$, or else its rank would be greater than q . We can move the element to the end, generating π' , using at most $q-1$ adjacent transpositions. Likewise, we can transform $\sigma \in P_n^q$ to σ' . By lemma 1, we

have $\pi', \sigma' \in P_n^q$, and since the maximal element is at the end of each, we can consider them to be in P_{n-1}^q . By induction we have that

$$d(\pi', \sigma') \leq (2(n-1) - 2q + 1)(q-1) \quad (\text{D.7})$$

We then have

$$d(\pi, \sigma) \leq d(\pi, \pi') + d(\pi', \sigma') + d(\sigma', \sigma) \quad (\text{D.8})$$

$$= q - 1 + (2(n-1) - 2q + 1)(q-1) + q - 1 \quad (\text{D.9})$$

$$= 2(q-1) + (2n - 2 - 2q + 1)(q-1) \quad (\text{D.10})$$

$$= (2n - 2q + 1)(q-1) \quad (\text{D.11})$$

□

Lemma 3. *The diameter of P_n^q is at least*

$$\text{diameter}(P_n^q) \geq \begin{cases} \frac{1}{2}n(n-1) & n \leq 2q \\ (2n - 2q + 1)(q-1) & n > 2q \end{cases} \quad (\text{D.12})$$

Proof. For $n \leq 2q - 2$, define

$$\pi_i = \begin{cases} i & i < q \\ q + n - i & i \geq q \end{cases} \quad (\text{D.13})$$

and let σ be the reflection of π (i.e. $\sigma_i = \pi_{n-i+1}$). Both are in P_n^q , which can be seen by looking at the ranks of the elements. Since they are reflections, the distance is $\frac{1}{2}n(n-1)$.

Otherwise, $n > 2q - 2$. Define

$$\pi = \begin{cases} i & i < q \\ i + q - 1 & q \leq i \leq n - q + 1 \\ q + n - i & i > n - q + 1 \end{cases} \quad (\text{D.14})$$

$$\sigma = \begin{cases} q - 1 + i & i < q \\ i + q - 1 & q \leq i \leq n - q + 1 \\ n + 1 - i & i > n - q + 1 \end{cases} \quad (\text{D.15})$$

and note that π and σ are in P_n^q . (Example: for $q = 4$ and $n = 32$, we have $\pi = \{1, 2, 3, 7, 8, 9, \dots, 30, 31, 32, 6, 5, 4\}$ and $\sigma = \{4, 5, 6, 7, 8, 9, \dots, 30, 31, 32, 3, 2, 1\}$.)

Then

$$\pi^{-1} = \begin{cases} i & i < q \\ n + q - i & q \leq i \leq 2q - 2 \\ i - q + 1 & i > 2q - 2 \end{cases} \quad (\text{D.16})$$

(Example: for $q = 4$ and $n = 32$, we have $\pi^{-1} = \{1, 2, 3, 32, 31, 30, 4, 5, 6, \dots, 27, 28, 29\}$.)

Then

$$\pi^{-1}\sigma = \begin{cases} n + 1 - i & i < q \\ i & q \leq i \leq n - q + 1 \\ n + 1 - i & i > n - q + 1 \end{cases} \quad (\text{D.17})$$

(Example: for $q = 4$ and $n = 32$, we have $\pi^{-1}\sigma = \{32, 31, 30, 4, 5, 6, \dots, 27, 28, 29, 3, 2, 1\}$.)

Now we consider the inversion count of $\tau = \pi^{-1}\sigma$. We count the inverted pairs $\{(i, j) : i < j, \tau_i > \tau_j\}$ by their start index:

Case 1 ($i < q$). Each i has $n - i$ inversions, totaling

$$\sum_{1 \leq i < q} n - i = n(q - 1) - \frac{1}{2}q(q - 1) \quad (\text{D.18})$$

$$= (q - 1)(n - \frac{1}{2}q) \quad (\text{D.19})$$

Case 2 ($q \leq i \leq n - q + 1$). Each i has $q - 1$ inversions, totaling $(q - 1)(n - 2q + 2)$.

Case 3 ($i > n - q + 1$). Each i has $n - i$ inversions, totaling

$$\sum_{n - q + 1 < i \leq n} n - i = \sum_{0 \leq j < q - 1} j \quad (\text{substitute } j = n - i) \quad (\text{D.20})$$

$$= \frac{1}{2}(q - 1)(q - 2) \quad (\text{D.21})$$

The final total is then

$$d(\pi, \sigma) = (q - 1)(n - \frac{1}{2}q) + (q - 1)(n - 2q + 2) + \frac{1}{2}(q - 1)(q - 2) \quad (\text{D.22})$$

$$= (q - 1)(n - \frac{1}{2}q + n - 2q + 2 + \frac{1}{2}q - 1) \quad (\text{D.23})$$

$$= (q - 1)(2n - 2q + 1) \quad (\text{D.24})$$

Finally, note that for $n = 2q - 1$:

$$(2n - 2q + 1)(q - 1) = (2n - (n + 1) + 1)\left(\frac{1}{2}(n + 1) - 1\right) \quad (\text{D.25})$$

$$= (2n - n)\left(\frac{1}{2}n - \frac{1}{2}\right) \quad (\text{D.26})$$

$$= \frac{1}{2}n(n - 1) \quad (\text{D.27})$$

and that again for $n = 2q$:

$$(2n - 2q + 1)(q - 1) \geq \frac{1}{2}n(n - 1) \quad (\text{D.28})$$

so $d(\pi, \sigma) \geq \frac{1}{2}n(n - 1)$ for both cases where $n \leq 2q$.

□

Theorem 4. *The diameter of P_n^q is exactly*

$$\text{diameter}(P_n^q) = \begin{cases} \frac{1}{2}n(n - 1) & n \leq 2q \\ (2n - 2q + 1)(q - 1) & n > 2q \end{cases} \quad (\text{D.29})$$

Proof. The upper and lower bounds shown by lemmas 2 and 3 are equal.

□

Index

- access time, 10, 11, 17, 18, 56, 62, 67, 69, 70, 111
- access time function, 18, 63, 68–70
- accuracy, 40
- activation function, *see* transfer function
- aggregation, 7
- analytical models, 7, 8
- associative memory, 43–44
- attribute, *see* feature
- auto-tuning, 3, 85

- backpropagation, 48
- bagging, 52, 62, 65, 66
- behavioral modeling, x, 3
- black-box modeling, 1, 7, 8, 40, 90, *see also* white-box modeling
- blktrace, 20, 112
- block, 9, 12, 15–17, 68, 112
- block interface, 1
- block storage device, 12
- boosting, 52–53
- burst buffers, 9
- bus-level tracing, 21

- caching, 1, 2, 4, 13, 17–18, 21, 24, 88
- CDF, 27
- checkerboard problem, 9, 10, 78

- conjugate gradient, 49
- content-addressable memory, *see* associative memory
- contributions, 5
- convergence, 48, 49, 51, 52, 73
- convexity, 26, 50
- convolution, 70
- CPU utilization, 63
- crossover, 52
- Cybenko theorem, 43
- cylinder, 111

- D2C, 20, 112
- data placement, 1
- decision trees, 7–9, 44–45, 59, 62, 65, 66
- deep learning, 80
- demerit, 27
- device characterization, 2
- Diaconis-Graham inequality, 34
- diagonal, 70, 71
- diameter, 114
- DIG, 2, 3, 16, 62
- dimensionality, 61
- direct mode, 21
- discontinuities, 13, 40, 58, 75, 83–86, 90
- discrete event simulation, x, 1, 7

- DiskSim, 1–3, 7, 9
- DIXtrac, 16
- dynamic workloads, 8
- e-SATA, 20
- ejection, 29–32
- emulation, 23
- emulator, 9
- ensemble methods, 52
- equilibrated stochastic gradient descent, 89
- error function, 24, 26, 51, 54
- ESGD, *see* equilibrated stochastic gradient descent
- expert knowledge, 1, 8
- external enclosure, 20
- Fast Fourier Transform, 70
- feature, 8, 42–44, 55, 62, 111, 112
- feature extraction, 56, 111
- feature selection, 69
- feed-forward, 55
- feedback, 7, 89
- FFT, *see* Fast Fourier Transform
- FIFO, 21
- file system, 12
 - simulation, 1
- file systems
 - parallel, 8
- FileSim, 8
- fitness function, 51, 52
- flash, 91
- Fourier analysis, 54, 62, 68
 - search-based, 70
- Fourier transform, 70, 75
- frequencies, 62
 - high, 18
- frequency threshold, 71
- FTL, 15, 68, 91
- fully-connected layers, 90
- garbage collection, 13
- Gauss-Newton, 51
- genetic algorithm, 47, 51–52, 58–63
- genetic programming, 47
- geometry, 15–17, 56
- gradient, 48–52
- gradient descent, 43, 48, 51
- granularity, 7, 8, 16, 28, 44
- gray-box modeling, 90
- grid search, 59
- half-zero random reads, 79–81, 84
- hard disk drive, 1, 2, 9, 13, 15, 17, 18, 24, 46, 62, 68, 69, 113
- harmonics, 69, 75
- head switch, 16, 83, 113
- helix, 84
- Hessian, 51
- high frequencies, 18
- higher-order periods, 71, 89
- hill climbing, 51
- HMMs, 46, *see also* IOHMMs
- Hopfield networks, 44
- hyperparameters, 54, 58–62, 85

idle, 29
 image processing, 56
 individuals, 52
 injection, 29, 31
 input augmentation, 72
 instability, 10
 instance, 28, 43, 44, 56, 111, 112
 interference, 72
 inversion count, 33, 34, 114
 IOHMMs, xii, 46

 k-d tree, 45
 Kalman filter, 10
 Kendall's tau, 33
 kernel function, 45, 55
 kernel trick, 45

 L-BFGS, 51
 L_1 norm, 24–25, 57–58
 L_2 norm, 24–25, 57–58
 layer size, 59, 60
 LBA, *see* LBN
 LBN, 56, 69, 112
 learning rate, 59, 61, 62
 Levenberg-Marquardt, 51
 line search, 49
 linear regression, 42
 local minima, 10, 51, 52
 locally periodic, 68, 112
 log-normal distribution, 59
 logistic sigmoid, 10, 42
 LRU stack distance, 4, 8, 88

 machine learning, 1, 3, 9, 12, 13, 15, 26,
 39, 40, 47, 54, 68, 69, 112
 offline, 7, 22, 40
 online, 7, 23
 supervised, 39
 MAD-median, *see* mean absolute deviation from the median
 MAE, *see* mean absolute error
 manual search, 59
 manufacturers, 2, 16
 mating, 60
 mean, 25
 mean absolute deviation from the median, 19, 35
 mean absolute error, 19, 54, 57, *see also*
 L_1 norm, mean square error
 mean square error, 19, 26, *see also* L_2
 norm, mean absolute error, root
 mean square error
 median, 19, 25
 memory footprint, x, 40, 45, 54
 method of steepest descent, *see* gradient
 descent
 mistake count, 30
 momentum, 59, 62
 MSE, *see* mean square error
 multi-stage training, 80
 multidimensional data, 3
 multilayer perceptron, 55
 mutation, 52, 59

 NCQ, 17, 20, 112

nearest neighbor, 45
 NEAT, 90
 neural nets, 7, 9, 10, 41–43, 46, 48, 50,
 54, 59–60, 62, 63, 65, 66, 72, 83,
 113
 convolutional, 56
 neuron, 10, 42, 43, 55, 56, 113
 NFS, 89
 noise, 28
 non-Gaussian, 19
 non-Gaussian errors, 57
 non-parametric models, 41
 non-volatile memory, 89, 91
 nondeterminism, 18, 35–36
 noop, 21
 NVM, *see* non-volatile memory

 object storage device, 12
 off-diagonal frequencies, 70
 off-grid frequencies, 77
 orthogonal, 49
 OSD, 1
 outdated models, 2
 outliers, 25, 57
 overfitting, 28, 52, 53

 parallelism, 15, 16, 52, 89
 parameter sweeps, 23
 parametric models, 41, 48
 penalty, 59, 72–73
 period composition, 75, 79–81
 periodic functions, 9
 periodicity, 4, 9–11, 15, 40, 68, 72
 in block number, 9
 permutation, 32, 114
 q -bounded, 33, 114
 platter, 17, 112, 113
 population, 52
 POSIX, 1, 89
 power saving, 20
 pread64, 21
 preprocessing, 3, 13, 40
 pretraining, 80
 probabilistic model, 8
 product unit, 90

 QoS, *see* quality of service
 quality of service, 23, 89
 quasi-Newton, 51
 queue, 13, 17, 20, 24, 27, 29–33, 111, 112
 quickprop, 50

 RAID, 1, 8, 9, 15–17, 68, 91
 random periods, 63, 65, 66
 random reads, 21
 rank, 30, 114
 read-ahead, 13, 18, 21, 88
 read/write head, 112
 real-time scheduling, 3, 9
 reboot, 21
 recurrent model, 10, 11
 recurrent neural net, 10, 43, 88
 regression tree, *see* decision tree
 relative error, 26
 relative performance, 8
 request reordering, 13, 17–18, 27

request-level accuracy, 26, *see also* trace-level accuracy
 resilient propagation, 49–50
 response time, 8, 12, 13, 17, 18, 20, 24, 26, 27, 54, 112
 RMSE, *see* root mean square error
 RMSProp, 50, 89
 root mean square error, 57, *see also* mean square error
 rotation, 15, 18, 68, 83
 rotation miss, 36, 83
 rotation time, 83
 rotational latency, 18, 36, 90, 111, 112
 Rprop, *see* resilient propagation

 saddle-free Newton, 89
 SATA, 16, 17, 20, 112
 sawtooth, 18, 75, 84
 scheduler, 20, 21, 30, 112
 scheduling, 2, 89

- accuracy, 29
- per-request error functions, 29

 SCSI, 16
 sector, *see* block
 sector remapping, 13
 sector sparing, 9
 seek, 13
 seek latency, 111, 113
 seek time, 18, 83
 sequential model-based optimization, *see* surrogate-based optimization
 sequentiality, 8
 sequentiality flag, 88
 serpentine layout, 2, 16, 113
 serpentine, 68, 71
 settle time, 83, 111
 shared weights, 4, 50, 55–56, 62, 90
 shingling, 2
 sigmoid, 43, 56
 simulated annealing, 51–52
 simulation

- parallel file system, 1

 sinusoids, 10, 72
 skew, 9, 15, 68
 skip-layer connections, 90
 SMBO, *see* surrogate-based optimization
 sparse Fourier, 77
 sparse sampling, 69
 Spearman’s footrule, 33, 34
 speed, x, 7, 40, 45
 SSDs, 1, 15–17, 68, 91
 state, 1, 13–15, 28, 43, 44, 46, 47, 88

- long-term, 13

 state space, 46
 state transition matrix, 46
 steepest descent, *see* gradient descent
 step size, 71
 stochastic backpropagation, 48, 59
 stochastic gradient descent, 48–49, 89
 stochastic Hessian-free optimization, 89
 storage configuration, 1
 stripes, 63, 70
 striping, 9, 15, 16, 68

- strong frequencies, 69–72, 113
- supercomputers, 9
- support vector regression, 7, 8, 45–46, 55
- surrogate-based optimization, 89
- SVR, *see* support vector regression

- table-based models, 8
- target propagation, 89
- temperature, 20, 51
- time series, 10, 13, 46
- time slices, 7
- topology, *see* geometry
- trace-level accuracy, 26, *see also* request-level accuracy
- tracing, 20
- track length, 62, 68
- track skew, 15
- tracks, 9, 15, 68, 113
- training, 9, 24, 34, 56
- training data, 10
- training time, 40, 54, 69
- transfer function, 10, 42, 55, 90, 113
- transfer learning, 89
- transfer time, 113
- transposition, 114
- two-spirals problem, 9, 10, 78

- unlabeled data, 80
- useful frequencies, 69, 113
- user space, 21

- vibration, 20

- weight sharing, *see* shared weights
- white-box modeling, 1, 7, *see also* black-box modeling
- window-based error functions, 28
- windows
 - sliding, 28
- workload, 7, 8, 13, 20, 27
- workload characterization, 8
- workload-specific, 8

- zero-latency reads, 2
- zone, 54, 113
- zoning, 2, 15