UNIVERSITY OF CALIFORNIA,
IRVINE

Deep Learning for Puzzles and Circadian Rhythms

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Forest Agostinelli

Dissertation Committee:
Professor Pierre Baldi, Chair
Professor Babak Shahbaba
Assistant Professor Sameer Singh

2019

DEDICATION

In memory of my mom
and to my many family members who gave so much to raise and educate me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Pierre Baldi, who has played a vital role in my scientific education. I am truly fortunate to have the opportunity to learn from someone so passionate about science.

I would like to thank the many professors at the University of California, Irvine who have contributed to my scientific education including Professor Babak Shahbaba, Professor Ramesh Jain, Professor Padhraic Smyth, Professor Sameer Singh, Professor Rick Lathrop, and Professor Paolo Sassone-Corsi.

I would like to thank Yuzo Kanomata for ensuring the functionality of all the machines, software, and web servers.

I would like to thank all of my labmates for their many ideas and valued feedback.

I would like to thank the ICS staff and graduate division staff for ensuring the functionality of the program.

I would like to thank the editors of *Nature Machine Intelligence*, *Nucleic Acids Research*, and *Bioinformatics* for publishing parts of this work.

I would like to thank everyone at the Black Student Union and the Nigerian Student Association.

I would like to thank the GEM Fellowship program as well as the National Science Foundation Graduate Research Fellowship program for helping to fund my studies as well as NVIDIA for the hardware donations.

I would also like to give a very special thanks to Nima Lamu Yolmo. I could write another thesis on why.

CURRICULUM VITAE

Forest Agostinelli

EDUCATION

Doctor of Philosophy in Computer Science **2019**
University of California Irvine *Irvine, California*

Masters in Computer Science **2014**
University of Michigan *Ann Arbor, Michigan*

Bachelor of Science in Electrical and Computer Engineering **2008**
The Ohio State University *Columbus, Ohio*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2019**
University of California Irvine *Irvine, California*

Deep Reinforcement Learning Research Intern **Summer 2017**
DeepMind *London, UK*

Machine Learning Research Intern **Summer 2015**
Microsoft Research *Beijing, China*

Machine Learning Research Intern **Summer 2014**
Adobe Research *San Francisco, California*

TEACHING EXPERIENCE

Teaching Assistant **2018-2019**
Introduction to Artificial Intelligence, University of California *Irvine, California*

Teaching Assistant **2018-2019**
Programming in Java, The Ohio State University *Columbus, Ohio*

REFEREED JOURNAL PUBLICATIONS

Solving the Rubik's Cube with Deep Reinforcement Learning and Search **2019**
Nature Machine Intelligence

CircadiOmics: Circadian Omic Data Web Portal **2018**
Nucleic Acids Research

What Time is It? Deep Learning Approaches for Circadian Rhythms **2016**
Bioinformatics

REFEREED CONFERENCE PUBLICATIONS

Solving the Rubik's Cube with Approximate Policy Iteration **2019**
International Conference on Learning Representations

Improving Survey Aggregation with Sparsely Represented Signals **2016**
SIGKDD Conference on Knowledge Discovery and Data Mining

Adaptive Multi-Column Deep Neural Networks with Application to Robust Image Denoising **2013**
Neural Information Processing Systems

PEER REVIEWED BOOK CHAPTERS

From Reinforcement Learning to Deep Reinforcement Learning: An Overview **2018**
Key Ideas in Learning Theory from Inception to Current State: Emmanuel Braverman's Legacy, pp. 298-328. Springer, Cham

SOFTWARE

DeepCube `http://deepcube.igb.uci.edu/`
Solve the Rubik's Cube with deep learning.

BIO_CYCLE `http://circadiomics.igb.uci.edu/biocycle`
Analyze circadian -omic experiments with deep learning.

ABSTRACT OF THE DISSERTATION

Deep Learning for Puzzles and Circadian Rhythms

By

Forest Agostinelli

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Pierre Baldi, Chair

The combination of deep learning with reinforcement learning and the application of deep learning to the sciences is a relatively new and flourishing field. We show how deep reinforcement learning techniques can learn to solve problems, often in the most efficient way possible, when faced with many possibilities but little information by designing an algorithm that can learn to solve seven different combinatorial puzzles, including the Rubik's cube. Furthermore, we show how deep learning can be applied to the field of circadian rhythms. Circadian rhythms are fundamental for all forms of life. Using deep learning, we can gain insight into circadian rhythms on the molecular level. Finally, we propose new deep learning algorithms that yield significant performance improvements on computer vision and high energy physics tasks.

Chapter 1

Introduction

Deep learning has shown state-of-the-art performance on a variety of tasks involving vision, speech, and scientific discovery. Additionally, The combination of deep learning with reinforcement learning has yielded human level and super-human level performance on a variety of games. While much progress has been made in recent years, the space of possibilities that have yet to be investigated remains large. In this dissertation, we investigate how to learn to solve problems that have many different possible permutations, but only one solution. Solutions to these problems carry implications for agents learning in environments with little information. Furthermore, we show that deep reinforcement learning can be applied to the study of circadian rhythms in multiple ways. Finally, we investigate general algorithmic improvements for deep neural networks.

1.1 Deep Reinforcement Learning for Puzzles

Puzzles such as the Rubik's cube pose a unique challenge for deep reinforcement learning problems due to its large state space (4.3×10^{19} different states) that contains only a single

goal state. If an agent only knows whether or not it has reached the goal, then the information it receives is uniform in all states except one. We design an agent that can learn to solve this puzzle, among others, by starting in reverse from the goal state. With this method, the agent learns to solve increasingly difficult states by bootstrapping from knowledge it has obtained on simpler states. We can then combine this knowledge with path finding algorithms, in particular, A* search. Application of this technique has yielded success on the Rubik's cube and six other puzzles: the 15-puzzle, 24-puzzle, 35-puzzle, 48-puzzle, Lights Out, and Sokoban, finding a shortest path in the majority of verifiable cases.

1.2 Deep Learning for Circadian Rhythms

Circadian rhythms are fundamental for life. Disruptions in circadian rhythms have been linked to diseases such as early aging, sleep disorders, and cancer. High throughput technology allows us to analyze circadian rhythms on a molecular level, presenting new challenges. The first challenge is to determine whether or not a molecular species is oscillating in a circadian fashion. The second challenge is, given a set of measurements taken at a single timepoint, to infer when that measurement was taken. This second challenge is particularly applicable to the plethora of experiments that have been taken at a single timepoint but have not been labeled with a timepoint.

1.3 Learning Activation Functions to Improve Deep Neural Networks

The state-of-the-art function approximation ability of deep neural networks to is due partly to the type of activation function used; that is, the point-wise non-linear function applied after

a matrix multiplication. However, while the values of these matrices are learned through backpropagation, the activation function is usually fixed. In this section, we investigate learned activation functions and show that they consistently improve the performance of deep neural networks for computer vision tasks and a high-energy physics task.

Chapter 2

Deep Reinforcement Learning for Puzzles

The Rubik's Cube is a prototypical combinatorial puzzle that has a large state space with a single goal state. The goal state is unlikely to be accessed using sequences of randomly generated moves, posing unique challenges for machine learning. We solve the Rubik's Cube with DeepCubeA, a deep reinforcement learning approach that learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge. DeepCubeA solves 100% of all test configurations, finding a shortest path to the goal state 60.3% of the time. DeepCubeA generalizes to other combinatorial puzzles and is able to solve the 15-puzzle, 24-puzzle, 35-puzzle, 48-puzzle, Lights Out, and Sokoban, finding a shortest path in the majority of verifiable cases.

2.1 Introduction

The Rubik's Cube is a classic combinatorial puzzle that poses unique and interesting challenges for artificial intelligence and machine learning. Although the state space is exceptionally large (4.3×10^{19} different states), there is only one goal state. Furthermore, the Rubik's Cube is a single-player game and a sequence of random moves, no matter how long, is unlikely to end in the goal state. Developing machine learning algorithms to deal with this property of the Rubik's Cube might provide insights into learning to solve planning problems with large state spaces. While machine learning methods have previously been applied to the Rubik's Cube, these methods have either failed to reliably solve the cube[76, 113, 16, 59] or have had to rely on specific domain knowledge[67, 3]. Outside of machine learning methods, methods based on pattern databases have been effective at solving puzzles such as the Rubik's Cube, 15-puzzle, and 24-puzzle [68, 70], but these methods can be memory intensive and puzzle-specific.

More broadly, a major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge. The classical 3x3x3 Rubik's Cube is only one representative of a larger family of possible environments, broadly sharing the characteristics described above, including: (1) cubes with longer edges or in higher dimension (e.g. 4x4x4 or 2x2x2x2); (2) sliding tile puzzles (e.g. the 15-puzzle, 24-puzzle, 35-puzzle, and 48-puzzle); (3) Lights Out; as well as (4) Sokoban. As the size and dimensions are increased, the complexity of the underlying combinatorial problems rapidly increases. For instance, while finding an optimal solution to the 15-puzzle takes less than a second on a modern day desktop, finding an optimal solution to the 24-puzzle can take days, and finding an optimal solution to the 35-puzzle is generally intractable[35]. Not only are the aforementioned puzzles relevant as mathematical games, but they can also be used to test planning algorithms [14] and to assess how well a machine learning approach may generalize to different environments. Furthermore, since the

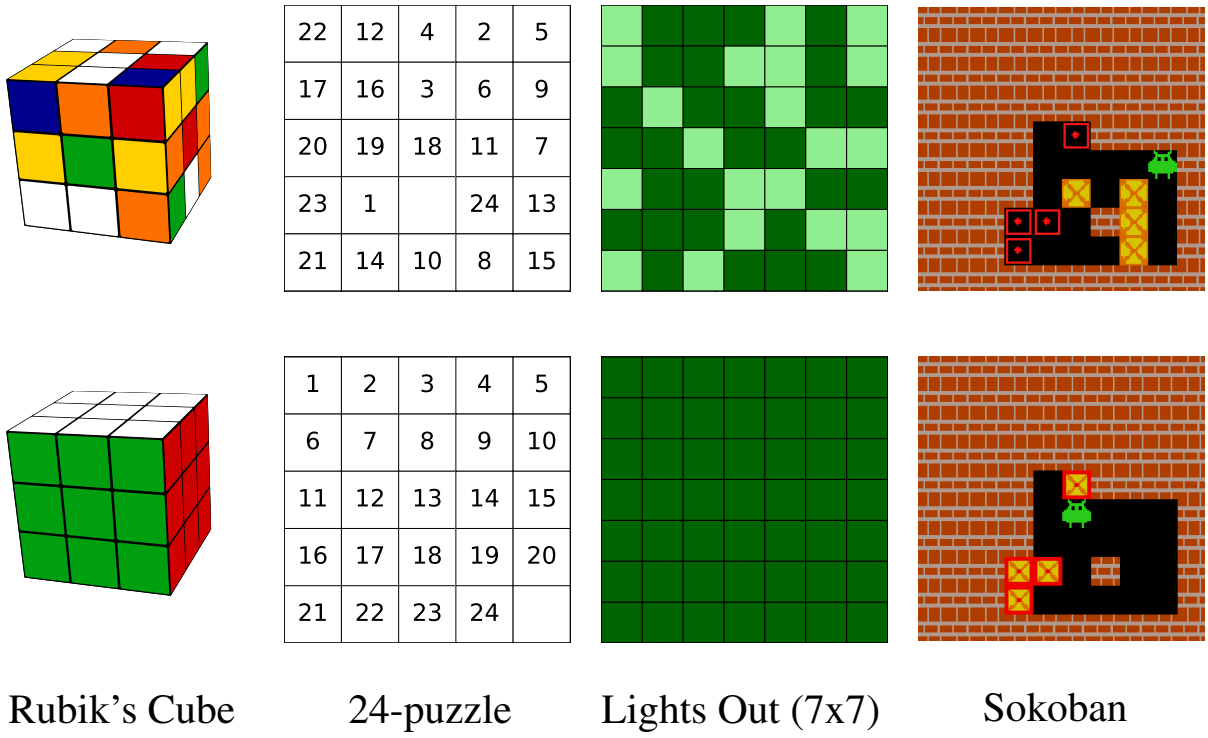


Figure 2.1: A visualization of a scrambled state (top) and the goal state (bottom) for four of the puzzles investigated in this paper.

operation of the Rubik's Cube and other combinatorial puzzles are deeply rooted in group theory, these puzzles also raise broader questions about the application of machine learning methods to complex symbolic systems, including mathematics. In short, for all these reasons, the Rubik's Cube poses interesting challenges for machine learning.

To address these challenges, we develop DeepCubeA which combines deep learning [109, 42] with classical reinforcement learning [122] (approximate value iteration[11, 98, 13]) and path finding methods (weighted A* search[50, 97]). DeepCubeA is able to solve combinatorial puzzles such as the Rubiks Cube, 15-puzzle, 24-puzzle, 35-Puzzle, 48-puzzle, Lights Out, and Sokoban (see Figure 2.1). DeepCubeA works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function). Since random play is unlikely to end in the goal state, DeepCubeA trains on states obtained by starting from the goal state and randomly taking moves in reverse. After training, the learned cost-to-go function is used as

a heuristic to solve the puzzles using weighted A* search [50, 97, 29].

DeepCubeA builds upon DeepCube [85], a deep reinforcement learning algorithm that solves the Rubik's Cube using a policy and value function combined with Monte Carlo tree search (MCTS). MCTS combined with a policy and value function is also used by AlphaZero which learns to beat the best existing programs in chess, Go, and shogi [112]. In practice, we find that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path. In contrast, DeepCubeA finds a shortest path to the goal for puzzles for which a shortest path is computationally verifiable: 60.3% of the time for the Rubik's Cube, and over 90% of the time for the 15-puzzle, 24-puzzle, and Lights Out.

2.2 Deep Approximate Value Iteration

Value iteration [98] is a dynamic programming algorithm [11, 13] that iteratively improves a cost-to-go function J. In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s. Value iteration loops through each state s and updates $J(s)$ with a new value $J'(s)$ until convergence:

$$J'(s) = min_a \sum_{s'} P^a(s, s')(g^a(s, s') + \gamma J(s')) \tag{2.1}$$

Here $P^a(s, s')$ is the *transition matrix* representing the probability of transitioning from state s to state s' by taking action a; $g^a(s, s')$ is the *cost* associated with transitioning from state s to s' by taking action a; and γ is the *discount factor*. In principle, this update equation can also be applied to the puzzles investigated in this paper. However, since these puzzles

are deterministic, the transition function is a degenerate probability mass function for each action, simplifying Equation 2.1. Furthermore, because we wish to assign equal importance to all costs, $\gamma = 1$. Therefore, we can update $J(s)$ using:

$$J'(s) = min_a(g^a(s, A(s, a)) + J(A(s, a))) \tag{2.2}$$

where: $A(s, a)$ is the state obtained from taking action a in state s and $g^a(s, s')$ is the cost to transition from state s to state s' taking action a. For the puzzles investigated in this paper, $g^a(s, s')$ is always 1.

Given the size of the state space of the Rubik's Cube, maintaining a table to store the cost-to-go of each state is not feasible. Therefore, we resort to *approximate value iteration* [13]. Instead, J is represented by a parameterized function implemented by a DNN. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go of state s, $J(s)$, and the updated cost-to-go estimation $J'(s)$. We call the resulting algorithm *deep approximate value iteration* (DAVI).

In order to train the DNN, we have two sets of parameters: the parameters being trained θ, and the parameters used to obtain an improved estimate of the cost-to-go θ_e. The output of $J_{\theta_e}(s)$ is set to 0 if s is the goal state. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go and the estimation obtained from Equation 2.2. Every C iterations, the algorithms checks if the error falls below a certain threshold ϵ; if so, then θ_e is set to θ. The entire DAVI process is shown in Algorithm 1. While we tried updating θ_e at each iteration, we found that the performance saturated after a certain point and sometimes became unstable. Updating θ_e only after the error falls below a threshold ϵ yields better, more stable, performance.

Algorithm 1: Deep Approximate Value Iteration

Input:

B: Batch size
K: Maximum number of scrambles
M: Training iterations
C: How often to check for convergence
ϵ: Error threshold

Output:

θ: Trained neural network parameters

$\theta \leftarrow initialize\_parameters()$
$\theta_e \leftarrow \theta$
for $m = 1$ to M **do**
  $X \leftarrow get\_scrambled\_states(B, K)$
  for $x_i \in X$ **do**
    $y_i \leftarrow \min_a(g^a(x_i, A(x_i, a)) + J_{\theta_e}(A(x_i, a)))$
  $\theta, loss \leftarrow train(J_\theta, X, \mathbf{y})$
  if *(M mod C = 0) and (loss* $< \epsilon$*)* **then**
    $\theta_e \leftarrow \theta$
Return θ

While the update in Equation 2.2 is only a one-step lookahead, it has been shown that, as training progresses, J approximates the optimal cost-to-go function J^*[13]. This optimal cost-to-go function computes the total cost incurred when taking a shortest path to the goal. Instead of Equation 2.2, multi-step lookaheads such as a depth-N search or Monte Carlo tree search can be used. We experimented with different multi-step lookaheads and found that multi-step lookahead strategies resulted in, at best, similar performance to the one-step lookahead used by DAVI (see Results for more details).

2.2.1 Training Set State Distribution

In order for learning to occur, we must train on a state distribution that allows information to propagate from the goal state to all the other states seen during training. Our approach for achieving this is simple: each training state x_i is obtained by randomly scrambling the goal state k_i times, where k_i is uniformly distributed between 1 and K. During training,

the cost-to-go function first improves for states that are only one move away from the goal state. The cost-to-go function then improves for states further away as the reward signal is propagated from the goal state to other states through the cost-to-go function. This can be seen as a simplified version of prioritized sweeping[88]. Exploring in reverse from the goal state is a well-known technique and has been used in means-end analysis[90] and STRIPS[36]. In future work we will explore different ways of generating a training set distribution.

2.3 Batch Weighted A* Search

After learning a cost-to-go function, we can then use it as a heuristic to search for a path between a starting state and the goal state. The search algorithm that we use is a variant of A* search [50], a best-first search algorithm that iteratively expands the node with the lowest cost until the node associated with the goal state is selected for expansion. The cost of each node x in the search tree is determined by the function $f(x) = g(x) + h(x)$, where $g(x)$ is the path cost, which is the distance between the starting state and x, and $h(x)$ is the heuristic function, which estimates the distance between x and and the goal state. The heuristic function $h(x)$ is obtained from the learned cost-to-go function:

$$h(x) = \begin{cases} 0 & \text{if } x \text{ is associated with the goal state} \\ J(x) & \text{otherwise} \end{cases} \tag{2.3}$$

We use a variant of A* search called weighted A* search [97]. Weighted A* search trades potentially longer solutions for potentially less memory usage by using instead the function $f(x) = \lambda\, g(x) + h(x)$, where λ is a weighting factor between zero and one. Furthermore, using a computationally expensive model for the heuristic function $h(x)$, such as a DNN,

could result in an intractably slow solver. However, $h(x)$ can be computed for many nodes in parallel by expanding the N lowest cost nodes at each iteration. We call the combination of A* search with a path-cost coefficient λ and a batch size of N *batch weighted A* search* (BWAS).

In summary, the algorithm presented in this paper uses DAVI to train a DNN as the cost-to-go function on states whose difficulty ranges from easy to hard. The trained cost-to-go function is then used as a heuristic for BWAS to find a path from any given state to the goal state. We call the resulting algorithm DeepCubeA.

2.4 Description of Puzzles

2.4.1 The Rubik's Cube

The Rubik's Cube state space has 4.3×10^{19} possible states. Any valid Rubik's Cube state can be optimally solved with at most 26 moves in the quarter-turn metric, or 20 moves in the half-turn metric [104, 102]. The quarter-turn metric treats 180 degree rotations as two moves, whereas the half-turn metric treats 180 degree rotations as one move. We use the quarter-turn metric.

The 3x3x3 Rubik's Cube consists of smaller cubes called cubelets. These are classified by their sticker count: center, edge, and corner cubelets have 1, 2, and 3 stickers, respectively. The Rubik's Cube has 26 cubelets with 54 stickers in total. The stickers have colors and there are six colors, one per face. In the solved state, all stickers on each face of the cube are the same color. The representation given to the DNN encodes the color of each sticker at each location using a one-hot encoding. Since there are 6 possible colors and 54 stickers in total, this results in a state representation of size 324.

Moves are represented using face notation: a move is a letter stating which face to rotate. *F*, *B*, *L*, *R*, *U*, and *D* correspond to turning the *front*, *back*, *left*, *right*, *up*, and *down* faces, respectively. Each face name is in reference to a fixed front face. A clockwise rotation is represented with a single letter, whereas a letter followed by an apostrophe represents a counter-clockwise rotation. For example: *R* rotates the right face by 90° clockwise, while *R'* rotates it by 90° counter-clockwise.

2.4.2 Sliding Tile Puzzles

Another combinatorial puzzle we use to test DeepCubeA is the n piece sliding puzzle. In the n-puzzle, n square sliding tiles, numbered from 1 to n, are positioned in a square of length $\sqrt{n+1}$, with one empty tile position. Thus, the 15-puzzle consists of 15 tiles in a 4x4 grid, the 24-puzzle consists of 24 tiles in a 5x5 grid, the 35-puzzle consists of 35 tiles in a 6x6 grid, and the 48-puzzle consists of 48 tiles in a 7x7 grid. Moves are made by swapping the empty position with any tile that is horizontally or vertically adjacent to it. The objective is to move the puzzle into its goal configuration shown in Figure 2.1. The 15-puzzle has $16!/2 \approx 1.0 \times 10^{13}$ possible states, the 24-puzzle has $25!/2 \approx 7.7 \times 10^{24}$ possible states, the 35-puzzle has $36!/2 \approx 1.8 \times 10^{41}$ possible states, and the 48-puzzle has $49!/2 \approx 3.0 \times 10^{62}$ possible states. Any valid 15-puzzle configuration can be solved with at most 80 moves [17, 69]. The largest minimal number of moves required to solve the 24-puzzle, 35-puzzle, and 48-puzzle is not known. For both puzzles, the representation given to the neural network uses one-hot encoding to specify which piece (tile or blank position) is in each position. For example, the dimension of the input to the neural network for the 15-puzzle would be $16 * 16 = 256$.

2.4.3 Lights Out

Lights Out is a grid-based puzzle consisting of an N by N board of lights that may be either active or inactive. The goal is to convert all active lights to inactive from a random starting position as seen in Figure 2.1. Pressing any light in the grid will switch the state of that light and its immediate horizontal and vertical neighbors. At any given state, a player may click on any of the N^2 lights. However, one difference of Lights Out compared to the other environments is that the moves are commutative. The representation given to the DNN is a vector of size N^2. Each element is 1 if the corresponding light is on and 0 if the corresponding light is off.

2.4.4 Sokoban

Sokoban [26] is a planning problem that requires an agent to move boxes onto target locations. Boxes can only be pushed, not pulled. The Sokoban environment we use is a 10 by 10 grid which contains four boxes that an agent needs to push on to four targets. In addition to the agent, boxes, and targets, Sokoban also contains walls. Since boxes can only be pushed, not pulled, some actions are irreversible. For example, a box pushed into a corner can no longer be moved, creating a sampling problem because some states are unreachable when starting from the goal state. To address this, for each training state, we start from the goal state and allow boxes to be pulled instead of pushed. The representation given to the DNN contains four binary vectors of size 10^2 that represent the position on the agent, boxes, targets, and walls.

2.5 Results

To test the approach, we generate a test set of 1,000 states by randomly scrambling the goal state between 1,000 and 10,000 times. Additionally, we test the performance of DeepCubeA on the three known states that are the furthest possible distance away from the goal (26 moves)[102]. In order to assess how often DeepCubeA finds a shortest path to the goal, we need to compare our results to a shortest path solver. We can obtain a shortest path solver by using iterative deepening A* search (IDA*)[66] with an admissible heuristic computed from a pattern database. Initially, we used the pattern database described in Korf's work on finding optimal solutions to the Rubik's Cube[68]; however, this solver only solves a few states a day. Therefore, we use the optimal solver provided by Rokicki [103]. This human-engineered solver relies on large pattern databases [21] (requiring 182GB of memory) and sophisticated knowledge of group theory to find a shortest path to the goal state. Comparisons between DeepCubeA and shortest path solvers are shown in Table 2.3.

The first two hidden layers of the DNNs have size 5,000 and 1,000 respectively, with full connectivity. This is then followed by 4 residual blocks [51], where each residual block has two hidden layers of size 1,000. Finally, the output layer consists of a single linear unit representing the cost-to-go estimate (see Figure 2.2). We used batch normalization [56] and rectified linear activation functions [40] in all hidden layers. The DNN was trained with a batch size of 10,000, optimized with ADAM [61], and did not use any regularization. The maximum number of random moves applied to any training state K was set to 30. The error threshold ϵ was set to 0.05. We checked if the loss fell below the error threshold every 5,000 iterations. Training was carried out for 1 million iterations on two NVIDIA Titan V GPUs, with six other GPUs used in parallel for data generation. In total, the DNN saw 10 billion examples during training. Training was completed in 36 hours. When solving scrambled cubes from the test set, we use 4 NVIDIA X Pascal GPUs in parallel to compute the cost-to-go estimate. For the 15-puzzle, 24-puzzle, and Lights Out we set K to 500. For the

35-puzzle, 48-puzzle, and Sokoban, we set K to $1,000$. For the 24-puzzle, we use 6 residual blocks instead of 4. When performing BWAS, the heuristic function is computed in parallel across four NVIDIA Titan V GPUs.

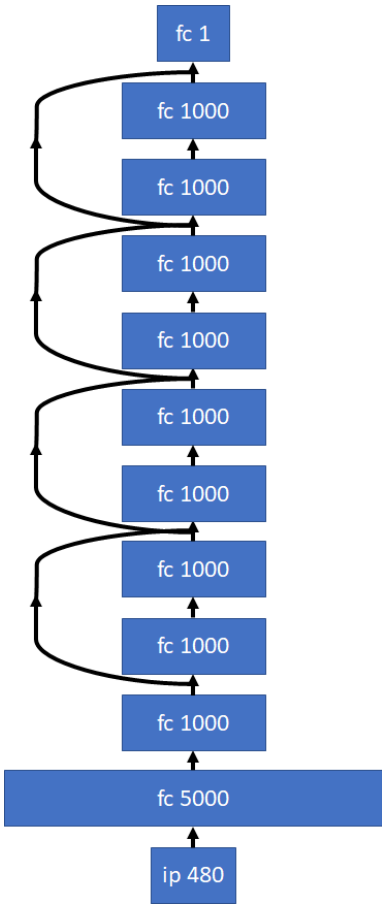
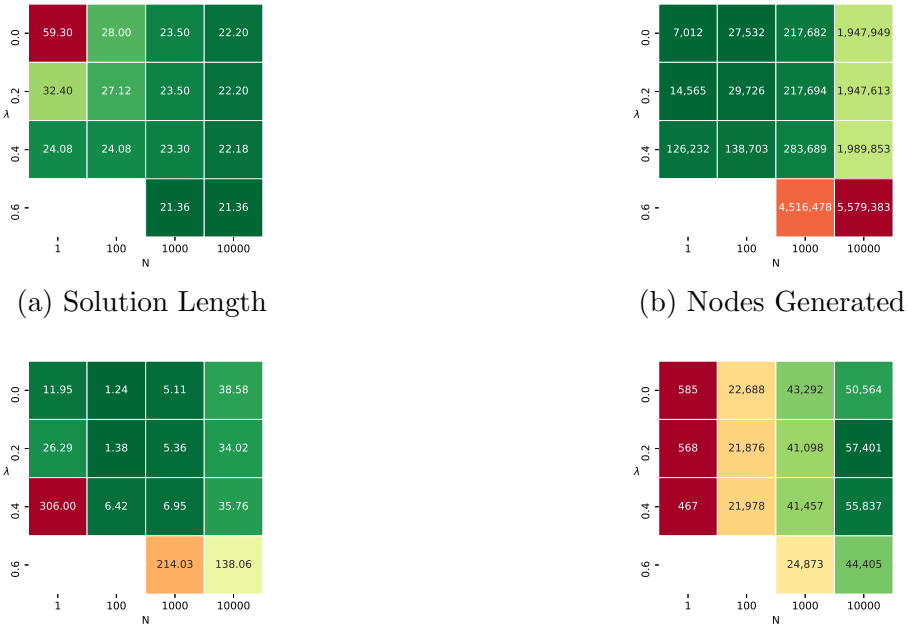


Figure 2.2: The deep neural network architecture used by DeepCubeA. The cube state is entered in the input layer (ip) and activities are propagated forward through a series of fully connected layers (fc) to produce the cost-to-go estimate at the output layer.

To choose the hyperparameters of BWAS, we did a grid search over λ and N. Values of λ were 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0 while values of N were 1, 100, 1,000, and 10,000. The grid search was performed on 100 cubes that were generated separately from the test set. The GPU machines available to us had 64GB of RAM. Hyperparameter configurations that reached this limit were stopped early and thus not included in the results. Figure 2.3 shows how λ and N affect performance in terms of average solution length, average number of nodes generated, average solve time, and average number of nodes generated per second.

The figure shows that as λ increases, the average solution length decreases, however, the time to find a solution typically increases as well. The results also show that larger values of N lead to shorter solution lengths, but generally also require more time to find a solution; however, the number of nodes generated per second also increases due to the parallelism provided by the GPUs. Since $\lambda = 0.6$ and $N = 10,000$ resulted in the shortest solution lengths, we use these hyperparameters for the Rubik's Cube. For the 15-puzzle, 24-puzzle, and 35-puzzle we use $\lambda = 0.8$ and $N = 20,000$. For the 48-puzzle we use $\lambda = 0.6$ and $N = 20,000$. We increased N from $10,000$ to $20,000$ because we saw a reduction in solution length. For Lights Out we use $\lambda = 0.2$ and $N = 1,000$. For Sokoban we use $\lambda = 0.8$ and $N = 1$.



(a) Solution Length

(b) Nodes Generated

(c) Solve Time

(d) Nodes/Second

Figure 2.3: Effects of BWAS hyperparameters on Rubik's Cube solving performance.

Pattern databases (PDBs)[21] are used to obtain a heuristic using lookup tables. Each lookup table contains the number of moves required to solve all possible combinations of a certain subgoal. For example, we can obtain a lookup table by enumerating all possible combinations of the edge cubelets on the Rubik's cube using a breadth-first search. These

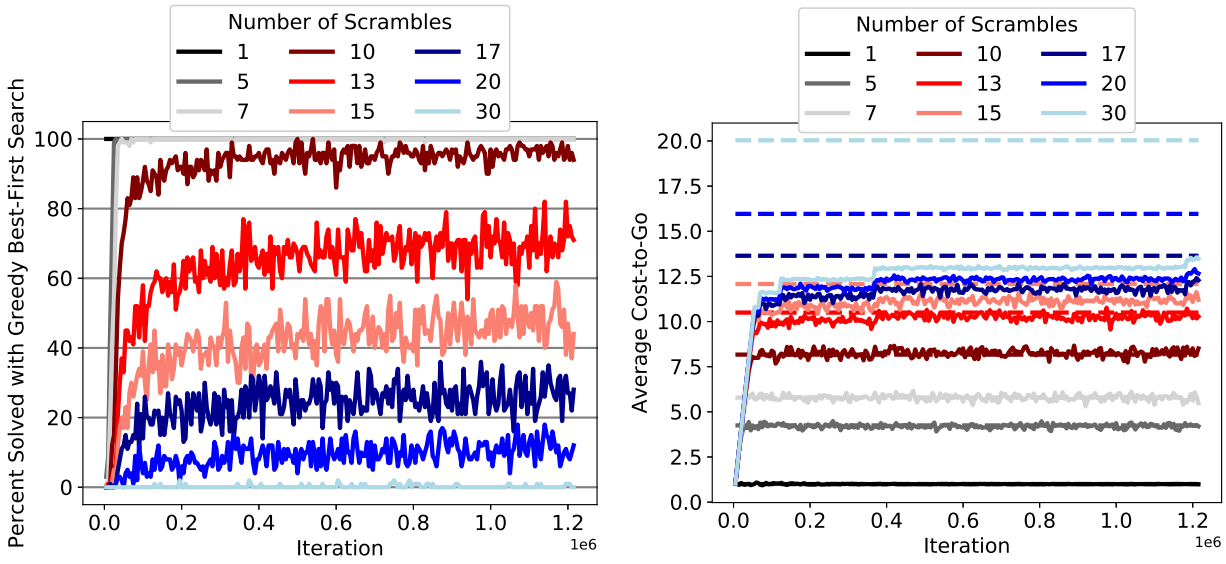


Figure 2.4: The performance of DeepCubeA. The plots show that DeepCubeA first learns how to solve cubes closer to the goal and then learns to solve increasingly difficult cubes. The dashed lines represent the true average cost-to-go.

lookup tables are then combined through either a max operator or a sum operator (depending on independence between subgoals)[68, 70] to produce a lower bound on the number of steps required to solve the problem. Features from different pattern databases can be combined with neural networks for improved performance [106].

For the Rubik's Cube, we implemented the pattern database that Korf uses to find optimal solutions to the Rubik's Cube [68]. For the 15-puzzle, 24-puzzle, and 35-puzzle, we implement the pattern databases described in Felner et. al's work on additive pattern databases[35]. To the best of our knowledge, no one has created a pattern database for the 48-puzzle. We create our own by partitioning the puzzle into 9 subgoals of size 5 and one subgoal of size 3. For all the n-puzzles, we also save the mirror of each PDB to improve the heuristic and map each lookup table to a representation of size p^k where p is the total number of puzzle pieces and k is the size of the subgoal. Though this uses more memory, this is done to increase the speed of the lookup table[35]. For the n-puzzle, the optimal solver algorithm (IDA*[66]) adds an additional optimization by only computing the location of the beginning state in the lookup table and then only computing offsets for each subsequently generated state.

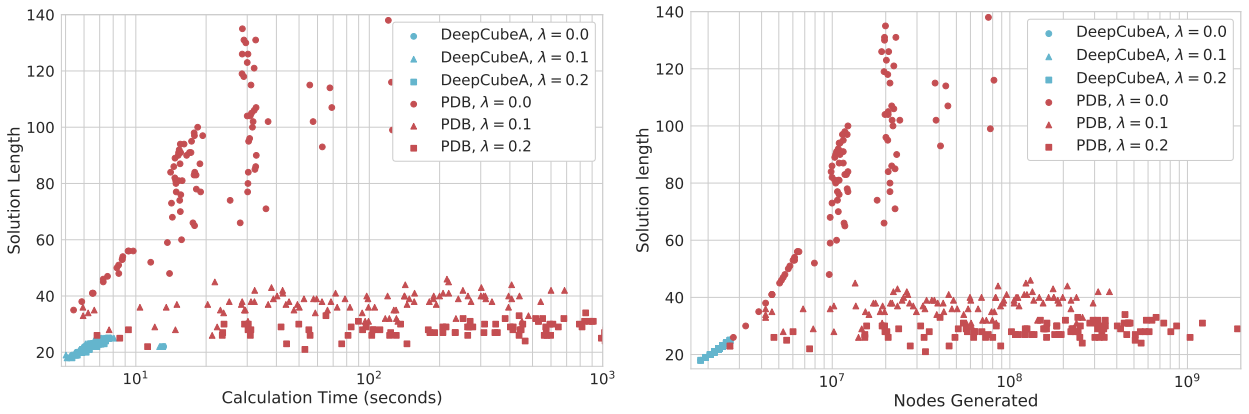


Figure 2.5: The performance of DeepCubeA vs pattern databases (PDBs)[68] when solving the Rubik's cube with BWAS. N = 10,000 and λ is either 0.0, 0.1, or 0.2. Each dot represents the result on a single state. DeepCubeA is both faster and produces shorter solutions.

2.5.1 Performance

DeepCubeA finds a solution to 100% of all test states. DeepCubeA finds a shortest path to the goal 60.3% of the time. Aside from the optimal solutions, 36.4% of the solutions are only two moves longer than the optimal solution, while the remaining 3.3% are four moves longer than the optimal solution. For the three states that are furthest away from the goal, DeepCubeA finds a shortest path to the goal for all three states (see Table 2.3). While we relate the performance of DeepCubeA to the performance of shortest path solvers based on pattern databases, a direct comparison cannot be made because shortest path solvers guarantee an optimal solution while DeepCubeA does not.

While pattern databases can be used in shortest path solvers, they can also be used in BWAS in place of the heuristic learned by DeepCubeA. We use Korf's pattern database heuristic for BWAS and compare to DeepCubeA. We perform BWAS with $N = 10,000$ and $\lambda = 0.0, 0.1, 0.2$. We compute the pattern database heuristic in parallel across 32 CPUs. Note that at $\lambda = 0.3$ BWAS runs out of memory when using pattern databases. Figure 2.5 shows that performing BWAS with DeepCubeA's learned heuristic consistently produces shorter solutions, generates fewer nodes, and is overall much faster than Korf's pattern database

heuristic.

We also compare the memory footprint and speed of pattern databases to DeepCubeA. In terms of memory, for pattern databases, it is necessary to load lookup tables into memory. For DeepCubeA, it is necessary to load the DNN into memory. Table 2.1 shows that DeepCubeA uses significantly less memory than pattern databases. In terms of speed, we measure how quickly pattern databases and DeepCubeA compute a heuristic for a single state, averaging over 1,000 states. Since DeepCubeA uses neural networks, which benefit from GPUs and batch processing, we measure the speed of DeepCubeA with both a single CPU and a single GPU, and with both sequential and batch processing of the states. Table 2.2 shows that, as expected, pattern databases on a single CPU are faster than DeepCubeA on a single CPU, however, the speed of pattern databases on a single CPU is comparable to the speed of DeepCubeA on a single GPU with batch processing.

During training we monitor how well the DNN is able to solve the Rubik's cube using greedy best-first search; we also monitor how well the DNN is able to estimate the optimal cost-to-go function (computed with Rokicki's shortest path solver [104]). How these performance metrics change as a function of training iteration is shown in Figure 2.4. The results show that DeepCubeA first learns to solve states closer to the goal before it learns to solve states further away from the goal. Cost-to-go estimation is less accurate for states further away from the goal; however, the cost-to-go function still correctly orders the states according to difficulty.

Comparison to Multi-Step Lookahead Update Strategies

Instead of using Equation 2.2, which may be seen as a depth-1 breadth-first search (BFS), to update the estimated cost-to-go function, we experimented with a depth-2 BFS. To obtain a better perspective on how well DeepCubeA's learning procedure trains the given DNN,

we also implemented an update strategy of trying to directly imitate the optimal cost-to-go function calculated using the handmade optimal solver[104] by minimizing the mean squared error between the output of the DNN and the oracle value provided by the optimal solver. We demonstrate that the DNN trained with DAVI achieves the same performance as a DNN with the same architecture trained with these update strategies. The performance obtained from a depth-2 BFS update strategy is shown in Figure 2.6. While the final performance obtained with depth-2 BFS is similar to the performance obtained with depth-1 BFS, its computational cost is significantly higher. Even when using 20 GPUs in parallel for data generation (instead of 6), the training time is 5 times longer for the same number of iterations. Figure 2.7 shows that the DNN trained to imitate the optimal cost-to-go function predicts the optimal cost-to-go more accurately than DeepCubeA for states scrambled 20 or more times. The figure also shows the performance on solving puzzles using greedy best-first search with this imitated cost-to-go function suffers for states scrambled fewer than 20 times. We speculate that this is because imitating the optimal cost-to-go function causes the DNN to overestimate the cost to reach the goal for states scrambled fewer than 20 times.

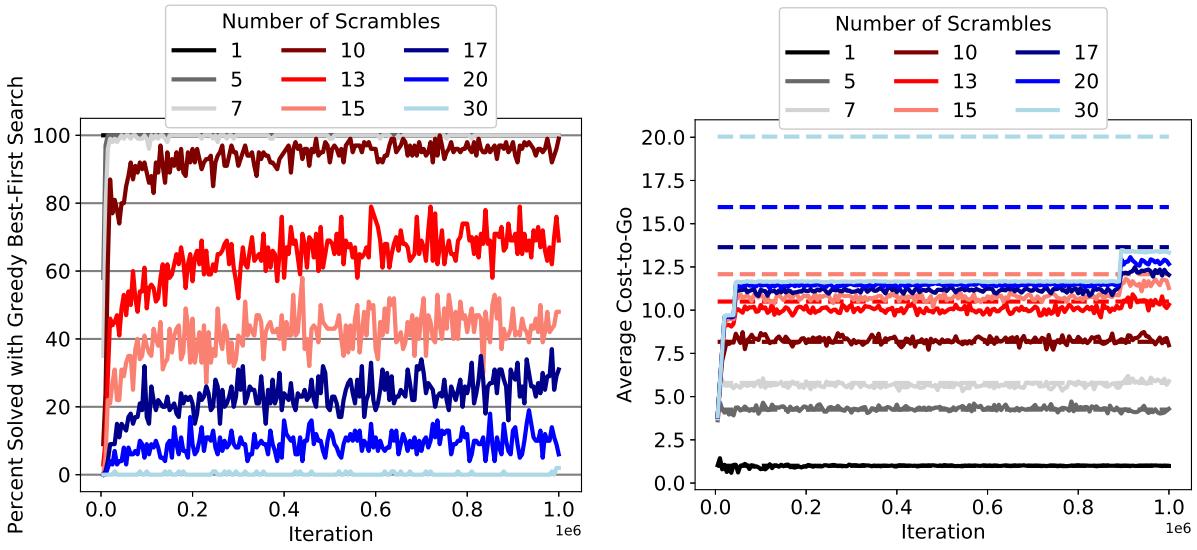


Figure 2.6: The performance of DeepCubeA using a depth-2 breadth-first search. While a depth-2 breadth first search uses more information than a depth-1 breadth-first search, the plots show that the performance of a depth-2 breadth-first search is the same as DeepCubeA. However, it is considerably more expensive computationally. The dashed lines represent the true average cost-to-go.

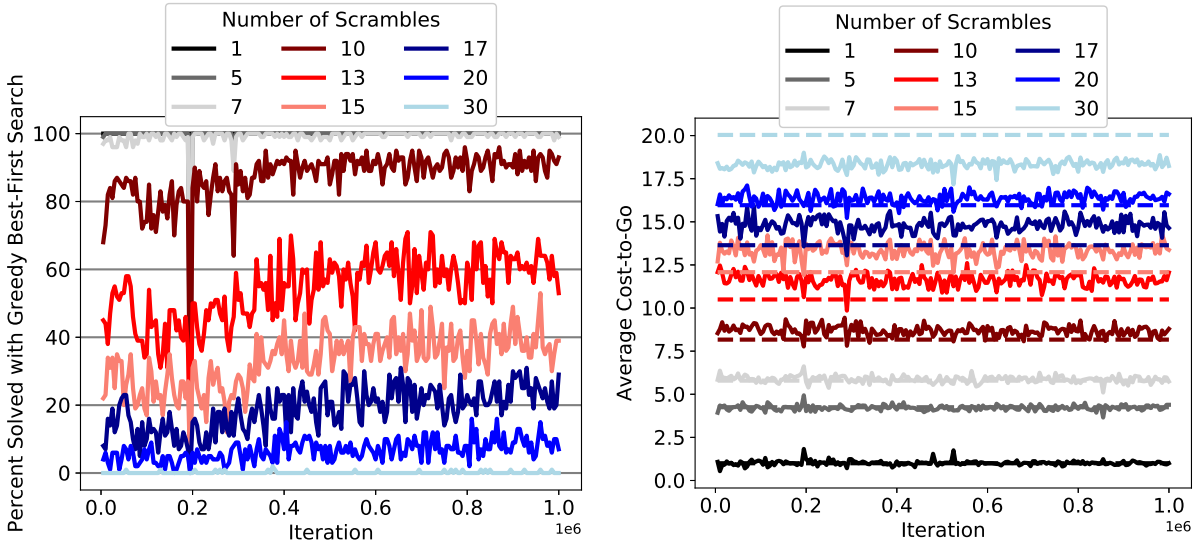


Figure 2.7: The performance of imitating the optimal cost-to-go function. Attempting to imitate the optimal value function leads to, at best, similar performance to DeepCubeA. The dashed lines represent the true average cost-to-go.

Conjugate Patterns and Symmetric States

Since the operation of the Rubik's Cube is deeply rooted in group theory, solutions produced by an algorithm that learns how to solve this puzzle should contain group theory properties. In particular, conjugate patterns of moves of the form aba^{-1} should appear relatively often when solving the Rubik's Cube. These patterns are necessary for manipulating specific cubelets while not affecting the position of other cubelets. Using a sliding window, we gathered all triplets in all solutions to the Rubik's Cube and found that aba^{-1} accounted for 13.11% of all triplets (significantly above random) while aba accounted for 8.86%, aab accounted for 4.96%, and abb accounted for 4.92%. To put this into perspective, for the optimal solver, aba^{-1}, aba, aab, and abb accounted for 9.15%, 9.63%, 5.30%, and 5.35% of all triplets, respectively.

In addition, we found that DeepCubeA often found symmetric solutions to symmetric states. One can produce a symmetric state for the Rubik's Cube by mirroring the cube from left to right, as shown in Figure 2.8. The optimal solutions for two symmetric states have the same length; furthermore, one can use the mirrored solution of one state to solve the other. To see if this property was present in DeepCubeA, we created mirrored states of the Rubik's Cube test set and solved them using DeepCubeA. The results showed that 58.30% of the solutions to the mirrored test set were symmetric to those of the original test set. Of the solutions that were not symmetric, 69.54% had the same solution length as the solution length obtained on the original test set. To put this into perspective, for the handmade optimal solver, the results showed that 74.50% of the solutions to the mirrored test set were symmetric to those of the original test set.

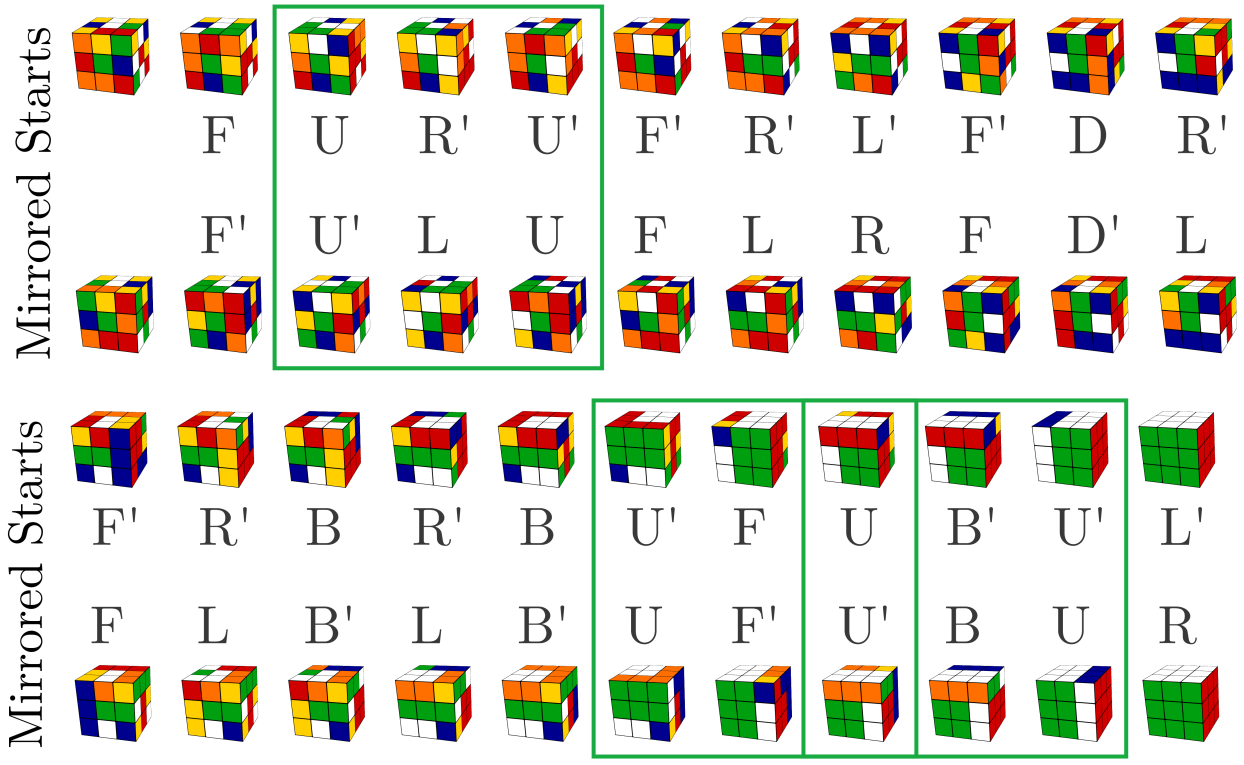


Figure 2.8: An example of symmetric solutions that DeepCubeA finds to symmetric states. Conjugate triplets are indicated by the green boxes. Note that the last two conjugate triplets are overlapping.

Web Server

We have created a web server, located at http://deepcube.igb.uci.edu/, to allow anyone to use DeepCubeA to solve the Rubik's Cube. In the interest of speed, the hyperparameters for BWAS are set to $\lambda = 0.2$ and $N = 100$ in the server. The user can initiate a request to scramble the cube randomly or use the keyboard keys to scramble the cube as they wish. The user can then use the "solve" button to have DeepCubeA compute and post a solution, and execute the corresponding moves. The basic web server's interface is displayed in Figure 2.9.

	RC	**15-p**	**24-p**	**35-p**	**48-p**	**LightsOut**	**Sokoban**
PDBs	4.67	8.51	1.86	0.64	4.86	-	-
PDBs$^+$	182.00	-	-	-		-	-
DeepCubeA	**0.06**	**0.06**	**0.08**	**0.08**	**0.10**	**0.05**	**0.06**

Table 2.1: Comparison of the size (in GB) of the lookup tables for pattern databases (PDBs) and the size of the DNN used by DeepCubeA. The RC column corresponds to the Rubik's Cube and columns with a "-p" suffix correspond to n-puzzles. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[103, 104]. The table shows that DeepCubeA always uses memory that is orders of magnitude less than PDBs.

	RC	**15-p**	**24-p**	**35-p**	**48-p**	**LightsOut**	**Sokoban**
PDBs	2E-06	1E-06	2E-06	3E-06	4E-06	-	-
PDBs$^+$	6E-07	-	-	-	-	-	-
DeepCubeA (GPU-B)	6E-06	6E-06	7E-06	8E-06	9E-06	7E-06	6E-06
DeepCubeA (GPU)	3E-03	3E-03	3E-03	2E-03	3E-03	4E-03	3E-03
DeepCubeA (CPU-B)	7E-04	6E-04	9E-04	9E-04	1E-03	1E-03	7E-04
DeepCubeA (CPU)	6E-03	6E-03	8E-03	8E-03	1E-02	2E-01	6E-03

Table 2.2: A suggestive comparison of the speed (in seconds) of the lookup tables for pattern databases (PDBs) and the speed of the DNN used by DeepCubeA when computing the heuristic for a single state. Results were averaged over 1,000 states. DeepCubeA was timed on a single CPU and on a single GPU when doing sequential processing of the states and batch processing of the states (batch processing is denoted by the "-B" suffix). The RC column corresponds to the Rubik's Cube and columns with a "-p" suffix correspond to n-puzzles. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[103, 104]. On a GPU, DeepCubeA is comparable to PDBs.

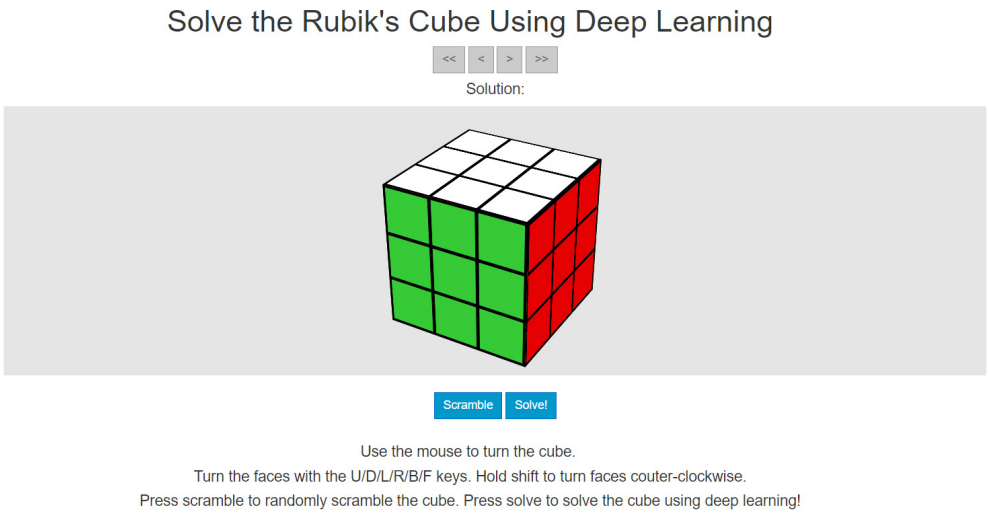


Figure 2.9: A visualization of the DeepCubeA web server (http://deepcube.igb.uci.edu/).

2.5.2 Generalization to Other Combinatorial Puzzles

The Rubik's Cube is only one combinatorial puzzle among many others. To demonstrate the ability of DeepCubeA to generalize to other puzzles, we applied DeepCubeA to four popular sliding tile puzzles: the 15-puzzle, the 24-puzzle, 35-puzzle, and 48-puzzle. Additionally, we applied DeepCubeA to Lights Out and Sokoban. Sokoban posed a unique challenge for DeepCubeA because actions taken in its environment are not always reversible.

Sliding Tile Puzzles

For these sliding tile puzzles, we generated a test set of 500 states randomly scrambled between 1,000 and 10,000 times. The same DNN architecture and hyperparameters that are used for the Rubik's Cube are also used for the n-puzzles with the exception of the addition of two more residual layers. We implemented an optimal solver using additive pattern databases[35]. DeepCubeA not only solved every test puzzle, but also found a shortest path to the goal 99.4% of the time for the 15-puzzle and 96.98% of the time for the 24-puzzle. We also test on the 17 states that are furthest away from the goal for the 15-puzzle (these states

are not known for the 24-puzzle)[63]. Solutions produced by DeepCubeA are, on average, 2.8 moves longer than the length of a shortest path and DeepCubeA finds a shortest path to the goal for 17.6% of these states. For the 24-puzzle, on average, pattern databases take 4,239 seconds and DeepCubeA takes 19.3 seconds, over 200 times faster. Moreover, in the worst case we observed that the longest time needed to solve the 24-puzzle is 5 days for pattern databases and two minutes for DeepCubeA. The average solution length is 124.76 for the 35-puzzle and 253.53 for the 48-puzzle; however, we do not know how many of them are optimal due to the optimal solver being prohibitively slow for the 35-puzzle and 48-puzzle. The performance of DeepCubeA on the 24-puzzle and 35-puzzle are summarized in Table 2.3.

Although the shortest path solver for the 35-puzzle and 48-puzzle was prohibitively slow, we compare DeepCubeA to pattern databases using BWAS. The results show that, compared to pattern databases, DeepCubeA produces shorter solutions and generates fewer nodes, as shown in Figure 2.10 and Figure 2.11. In combination, these results suggest that as the size of the n-puzzle increases, DeepCubeA scales favorably compared to pattern databases.

Lights Out

We tested DeepCubeA on the 7 by 7 Lights Out puzzle. A theorem by Scherphuis[107] shows that, for the 7 by 7 Lights Out puzzle, any solution that does not contain any duplicate moves is an optimal solution. Using this theorem, we found that DeepCubeA found a shortest path to the goal for all test cases.

Sokoban

To test our method on Sokoban, we train on the 900,000 training examples and test on the 1,000 testing examples used by previous research on single-agent policy tree search applied to

Sokoban[4]. DeepCubeA successfully solves 100% of all test examples. We compare solution length and number of nodes expanded to this same previous research[91]. Although the goals of the aforementioned paper are slightly different than ours; DeepCubeA finds shorter paths than previously reported methods and also expands, at least, 3 times fewer nodes (see Table 2.3).

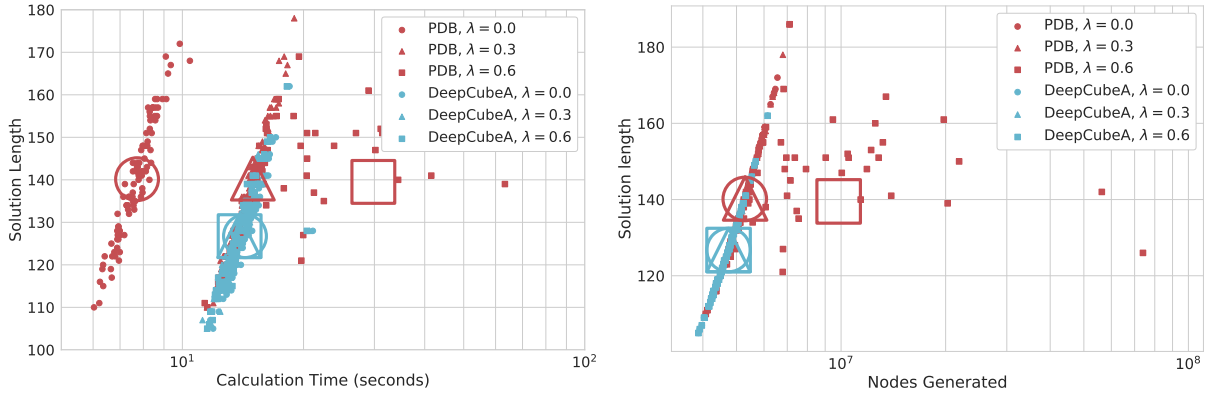


Figure 2.10: The performance of DeepCubeA vs pattern databases (PDBs) when solving the 35-puzzle with BWAS. N = 10,000 and $\lambda = 0.0, 0.3, 0.6$. Each dot represents the result on a single state. DeepCubeA is almost always faster than PDBs and always produces shorter solutions. The large shapes represent the average of the respective run. The results show that DeepCubeA, on average, always produces shorter solutions and, on average, is faster than PDBs for two of the three assignments of λ.

2.6 Discussion

DeepCubeA is able to solve planning problems with large state spaces and few goal states by learning a cost-to-go function, parameterized by a deep neural network, which is then used as a heuristic function for weighted A* search. The cost-to-go function is learned by using approximate value iteration on states generated by starting from the goal state and taking moves in reverse. DeepCubeA's success on solving the seven problems investigated in this paper suggests that DeepCubeA can be readily applied to new problems given an input representation, a state transition model, a goal state, and a reverse state transition model that can be used to adequately explore the state space.

Puzzle	Solver	Len	% Opt	Nodes	Secs	Nodes/Sec
Rubik's Cube	PDBs[68]	-	-	-	-	-
	PDBs$^+$[103]	20.67	100.0%	2.05E+06	2.20	1.79E+06
	DeepCubeA	21.50	60.3%	6.62E+06	24.22	2.90E+05
Rubik's Cube$_h$	PDBs[68]	-	-	-	-	-
	PDBs$^+$[103]	26.00	100.0%	2.41E+10	13,561.27	1.78E+06
	DeepCubeA	26.00	100.0%	5.33E+06	18.77	2.96E+05
15-Puzzle	PDBs[35]	52.02	100.0%	3.22E+04	0.002	1.45E+07
	DeepCubeA	52.03	99.4%	3.85E+06	10.28	3.93E+05
15-Puzzle$_h$	PDBs[35]	80.00	100.0%	1.53E+07	0.997	1.56E+07
	DeepCubeA	82.82	17.65%	2.76E+07	69.36	3.98E+05
24-Puzzle	PDBs[35]	89.41	100.0%	8.19E+10	4,239.54	1.91E+07
	DeepCubeA	89.49	96.98%	6.44E+06	19.33	3.34E+05
35-Puzzle	PDBs[35]	-	-	-	-	-
	DeepCubeA	124.64	-	9.26E+06	28.45	3.25E+05
48-Puzzle	PDBs	-	-	-	-	-
	DeepCubeA	253.35	-	1.96E+07	74.46	2.63E+05
Lights Out	DeepCubeA	24.26	100.0%	1.14E+06	3.27	3.51E+05
Sokoban	LevinTS[91]	39.80	-	6.60E+03	-	-
	LevinTS[91] (*)	39.50	-	5.03E+03	-	-
	LAMA[91]	51.60	-	3.15E+03	-	-
	DeepCubeA	32.88	-	1.05E+03	2.35	5.60E+01

Table 2.3: Comparison of DeepCubeA with optimal solvers based on pattern databases (PDBs) along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), and number of nodes generated per second. The datasets with an "h" subscript represent the dataset containing the states that are furthest away from the goal state. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[103, 104]. For Sokoban, we compare nodes expanded instead of nodes generated to allow for direct comparison to previous work. DeepCubeA often finds a shortest path to the goal. For the states that are furthest away from the goal, DeepCubeA either finds a shortest path or a path close in length to a shortest path.

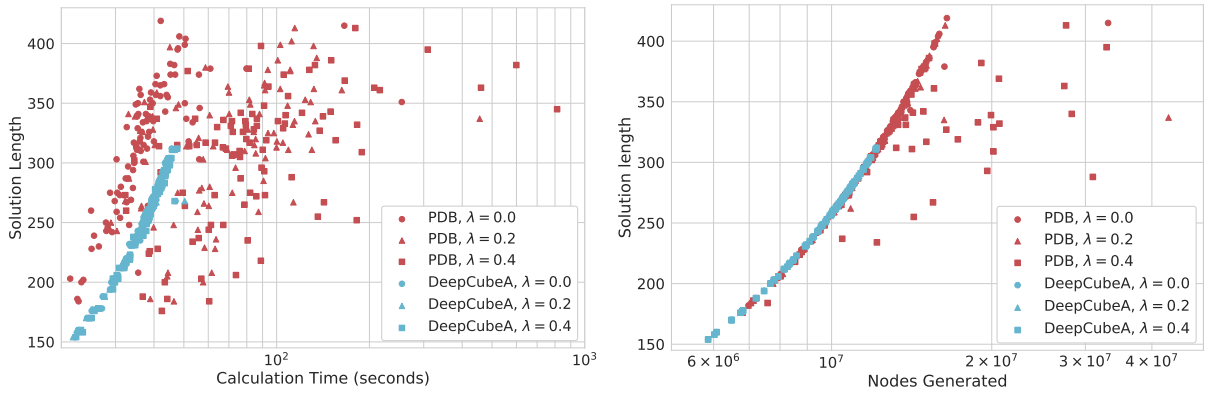


Figure 2.11: The performance of DeepCubeA vs pattern databases (PDBs) when solving the 48-puzzle with BWAS. N = 10,000 and λ is either 0.0, 0.2, or 0.4. Each dot represents the result on a single state. DeepCubeA is generally faster than PDBs and generally produces shorter solutions than PDBs.

To satisfy the theoretical bounds on how much the length of a solution will deviate from the length of an optimal solution, the heuristic used in weighted A* search must be admissible. That is to say that the heuristic can never overestimate the cost to reach the goal. While DeepCubeA's value function is not admissible, we empirically evaluate by how much DeepCubeA overestimates the cost to reach the goal. To do this, we obtain the length of a shortest path to the goal for 100,000 Rubik's cube states scrambled between 1 and 30 times. We then evaluate those same states with DeepCubeA's heuristic function J_θ. We find that DeepCubeA's heuristic function does not overestimate the cost to reach the goal 66.8% of the time and 97.4% of the time it does not overestimate it by more than 1. The average overestimation of the cost is 0.24. In future work, we will examine how to obtain admissible heuristic functions using DNNs.

The generality of the core algorithm suggests that it may have applications beyond combinatorial puzzles, as problems with large state spaces and few goal states are not rare in planning, robotics, and the natural sciences.

Chapter 3

Deep Learning for Circadian Rhythms

Circadian rhythms date back to the origins of life, are found in virtually every species and every cell, and play fundamental roles in functions ranging from metabolism to cognition. Modern high-throughput technologies allow the measurement of concentrations of transcripts, metabolites, and other species along the circadian cycle creating novel computational challenges and opportunities, including the problems of inferring whether a given species oscillate in circadian fashion or not, and inferring the time at which a set of measurements was taken.

We first curate several large synthetic and biological time series data sets containing labels for both periodic and aperiodic signals. We then use deep learning methods to develop and train BIO_CYCLE, a system to robustly estimate which signals are periodic in high-throughput circadian experiments, producing estimates of amplitudes, periods, phases, as well as several statistical significance measures. Using the curated data, BIO_CYCLE is compared to other approaches and shown to achieve state-of-the-art performance across multiple metrics. We then use deep learning methods to develop and train BIO_CLOCK to robustly estimate the time at which a particular single-time-point transcriptomic experiment was carried. In most

cases, BIO_CLOCK can reliably predict time, within approximately one hour, using the expression levels of only a small number of core clock genes. BIO_CLOCK is shown to work reasonably well across tissue types, and often with only small degradation across conditions. BIO_CLOCK is used to annotate most mouse experiments found in the GEO database with an inferred time stamp.

3.1 Introduction

The importance of circadian rhythms cannot be understated: circadian oscillation have been observed in animals, plants, fungi, and cyanobacteria and date back to the very origins of life on Earth. Indeed, some of the most ancient forms of life, such as cyanobacteria, use photosynthesis as their energy source and thus are highly circadian almost by definition. These oscillations play a fundamental role in coordinating the homeostasis and behavior of biological systems, from the metabolic [30, 38, 124, 134] to the cognitive levels [39, 33]. Disruption of circadian rhythms has been directly linked to health problems [124, 62, 73] ranging from cancer, to insulin resistance, to diabetes, to obesity, and to premature ageing [60, 2, 37, 64, 110, 126, 111, 65, 38]. At their most fundamental level, these oscillations are molecular in nature, whereby the concentrations of specific molecular species such as transcripts, metabolites, and proteins oscillate in the cell with a 24h periodicity. Modern high-throughput technologies allow large-scale measurements of these concentrations along the circadian cycle thus creating new data sets and new computational challenges and opportunities. To mine these new datasets, here we develop and apply machine learning methods to address two questions: (1) which molecular species are periodic?; and (2) what time or phase is associated with high-throughput transcriptomic measurements made at a single timepoint?.

At the molecular level, circadian rhythms are in part driven by a genetically encoded, highly

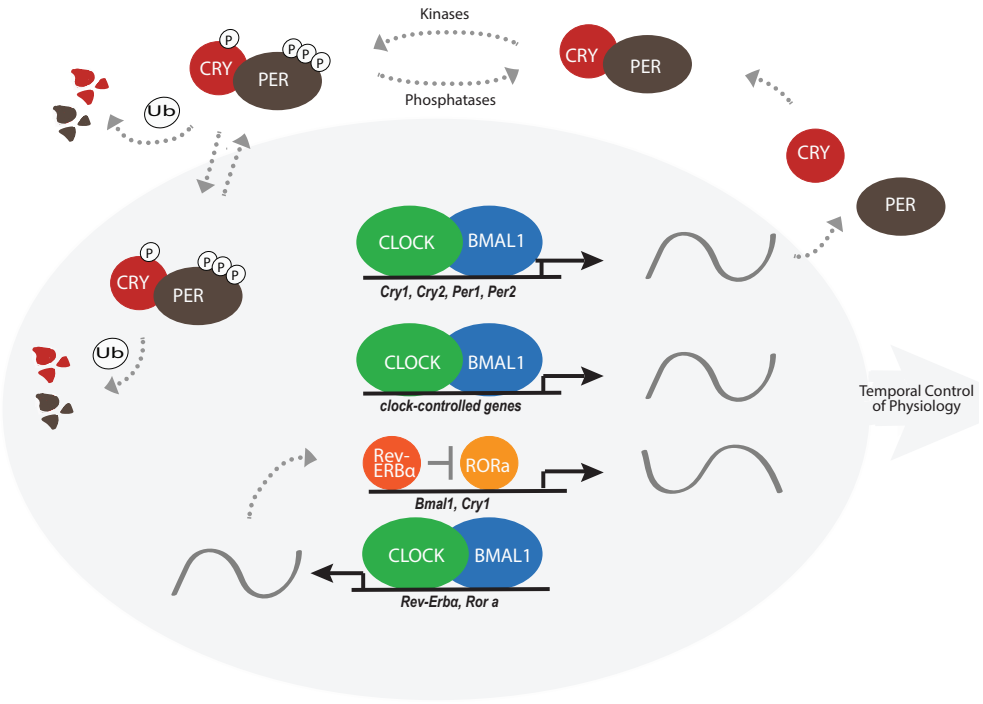


Figure 3.1: Core clock genes and proteins and the corresponding transcription/translation negative feedback loop.

conserved, core clock found in nearly every cell based on negative transcription/translation feedback loops, whereby transcription factors drive the expression of their own negative regulators [108, 93], and involving only a dozen genes [131, 93]. In the mammalian core clock (Figure 3.1), two bHLH transcription factors, CLOCK and BMAL1 heterodimerize and bind to conserved E-box sequences in target gene promoters, thus driving the rhythmic expression of mammalian Period (*Per1*, *Per2*, and *Per3*) and Cryptochrome (*Cry1* and *Cry2*) genes [119]. PER and CRY proteins form a complex that inhibits subsequent CLOCK:BMAL1-mediated gene expression [15, 25, 93]. The master core clock located in the suprachiasmatic nucleus (SCN) [89, 100] of the hypothalamus interacts with the peripheral core clocks throughout the body [134, 124].

In contrast to the small size of the core clock, high-throughput transcriptomic (DNA microarrays, RNA-seq) or metabolomic (mass spectrometry) experiments [54, 86, 32, 31, 92, 1, 83, 125], have revealed that a much larger fraction, typically on the order of 10%, of all

transcripts or metabolites in the cell are oscillating in a circadian manner. Furthermore, the oscillating transcripts and metabolites differ by cell, tissue type, or condition [92, 118, 131]. Genetic, epigenetic, and environmental perturbations–such as a change in diet–can lead to cellular reprogramming and profoundly influence which species are oscillating in a given cell or tissue [32, 82, 10, 31, 28, 84]. When results are aggregated across tissues and conditions, a very large fraction, often exceeding 50% and possibly approaching 100%, of all transcripts is capable of circadian oscillations under at least one set of conditions, as shown in plants [48, 19], cyanobacteria and algae [128, 87], and mouse [135, 95].

In a typical circadian experiment, high-throughput omic measurements are taken at multiple timepoints along the circadian cycle under both control and treated conditions. Thus the first fundamental problem that arises in the analysis of such data is the problem of detecting periodicity, in particular circadian periodicity, in these time series. The problem of detecting periodic patterns in time series is of course not new. However, in the cases considered here the problem is particularly challenging for several reasons, including: (1) the sparsity of the measurements (the experiments are costly and thus data may be collected for instance only every 4 hours); (2) the noise in the measurements and the well known biological variability; (3) the related issue of small sample sizes (e.g. $n = 3$); (4) the issue of missing data; (5) the issue of uneven sampling in time; and (6) the large number of measurements (e.g. 20,000 transcripts) and the associated multiple-hypothesis testing problem.

Here we develop and apply deep learning methods for robustly assessing periodicity in high-throughput circadian experiments, and systematically compare the deep learning approach to the previous, non-machine learning, approaches [55, 132, 41]. While this is useful for circadian experiments, the vast majority of all high-throughput expression experiments have been carried, and continue to be carried, at single timepoints. This can be problematic for many applications, including applications to precision medicine, precisely because circadian variations are ignored creating possible confounding factors. This raises the second problem

of developing methods that can robustly infer the approximate time at which a single-time high-throughput expression measurement was taken. Such methods could be used to retrospectively infer a time stamp for any expression data set, in particular to improve the annotations of all the datasets contained in large gene expression repositories, such as the Gene Expression Omnibus (GEO)[34], and improve the quality of all the downstream inferences that can be made from this wealth of data. There may be other applications of such a method, for instance in forensic sciences, to help infer a time of death. In any case, to address the second problem we also develop and apply deep learning methods to robustly infer time or phase for single-time high-throughput gene expression measurements.

3.2 Datasets

3.2.1 Periodicity Inference from Time Series Measurements

To train and evaluate the deep learning methods, we curate BioCycle, the largest dataset including both synthetic and real-world biological time series, and both periodic and aperiodic signals. While the main goal here is to create methods to analyze real-world biological data, relying only on biological data to determine the effectiveness of a method is not sufficient because there are not many biological samples which have been definitively labeled as being periodic or aperiodic. Even when one can be confident that a signal is periodic, it can be difficult to determine the true period, phase, and amplitude of that signal. Therefore, we rely also on synthetic data to provide us with signals that we can say are definitely periodic or aperiodic, and whose attributes–such as period, amplitude, and phase–can be controlled and are known. Furthermore, previous approaches were developed using synthetic data and thus the same synthetic data must be used to make fair comparisons.

Synthetic Data

We first curate a comprehensive synthetic dataset $\text{BioCycle}_{\text{Synth}}$, which includes all previously defined synthetic signals found in JTK_Cycle [55] and ARSER [132], but also contains new signals. $\text{BioCycle}_{\text{Synth}}$ is in turn a collection of two different types of datasets: a dataset in which signals are constructed using mathematical formulas ($\text{BioCycle}_{\text{Form}}$), and a dataset in which signals are generated from a Gaussian process [101] ($\text{BioCycle}_{\text{Gauss}}$). In previous work, synthetic data was generated with carefully constructed formulas to try to mimic periodic signals found in real-world data (see below). While this gives one a lot of control over the data, it can create signals that are too contrived and therefore not representative of real-world biological variations. In addition, the noise added at each timepoint is independent of the other timepoints, which may not be the case in real-world data. The $\text{BioCycle}_{\text{Gauss}}$ dataset uses Gaussian processes to generate the data and address these problems.

The datasets used in JTK_Cycle contain the following types of formulas or signals: cosine, cosine with outlier timepoints, and white noise. The ARSER dataset contains cosine, damped cosine with an exponential trend, white noise, and an auto-regressive process of order 1 (AR(1)). In addition to all the aforementioned signals, $\text{BioCycle}_{\text{Form}}$ contains also 9 additional kinds of signals: combined cosines (cosine2), cosine peaked, square wave, triangle wave, cosine with a linear trend, cosine with an exponential trend, cosine multiplied by an exponential, flat, and linear signals (many of which can be found in [22]). Figure 3.2 shows an example of each type of signal found in the $\text{BioCycle}_{\text{Form}}$ dataset. For clarity, the periodic signals are shown without noise. Signals in the $\text{BioCycle}_{\text{Form}}$ dataset have an additional random offset chosen uniformly between -200 and 200, random amplitudes chosen uniformly between 1 and 100, signal to noise ratios (SNRs) of 1-5, random phases chosen uniformly between 0 and 2π, and periods between 20 and 28. At each timepoint sample, zero mean Gaussian noise is added with the proper SNR variance.

The $\text{BioCycle}_{\text{Gauss}}$ dataset is obtained from a Gaussian process. The value of the covariance matrix corresponding to the timepoints x and x' is determined by a kernel function $k(x, x')$. Equation 3.1 is the kernel function used to generate the periodic signals, and Equation 3.2 is the kernel function used to generate the aperiodic signals in $\text{BioCycle}_{\text{Gauss}}$.

$$k_p(x, x') = exp(\frac{-sin^2(|\pi\frac{1}{p}(x - x')|)}{2l^2}) + \sigma^2\delta(x, x') + \beta x x' \tag{3.1}$$

$$k_a(x, x') = exp(\frac{-(x - x')^2}{2l^2}) + \sigma^2\delta(x, x') \tag{3.2}$$

The parameter l controls how strong the covariance is between two different timepoints, σ controls how noisy the synthetic data is, and β can add a non-stationary, linear, trend to the signals [27]. The parameter p in equation 3.1 is the period of the signal. To generate the data in $\text{BioCycle}_{\text{Gauss}}$, the values of l, σ, β, p, as well as the offset and the scale are varied, in a way similar to the data in $\text{BioCycle}_{\text{Form}}$. Examples of signals from the $\text{BioCycle}_{\text{Gauss}}$ dataset are given in Figure 3.3.

JTK_Cycle analyzes synthetic signals sampled over 48 hours with a sampling frequency of 1 and 4 hours. ARSER analyzes synthetic signals sampled over 44 hours with a sampling frequency of 4 hours. BioCycle analyzes synthetic signals sampled over 24 hours and 48 hours. Signals sampled over 24 hours have a sampling frequency of 4, 6, and an uneven sampling at timepoints 0, 5, 9, 14, 19, and 24. Signals sampled over 48 hours have sampling frequencies of 4, 8, and an uneven sampling at timepoints 0, 4, 8, 13, 20, 24, 30, 36, 43. The sampling frequencies in these datasets are intentionally sparse to mimic the sparse temporal sampling of real-world high-throughput data. The number of synthetic signals at each sampling fre-

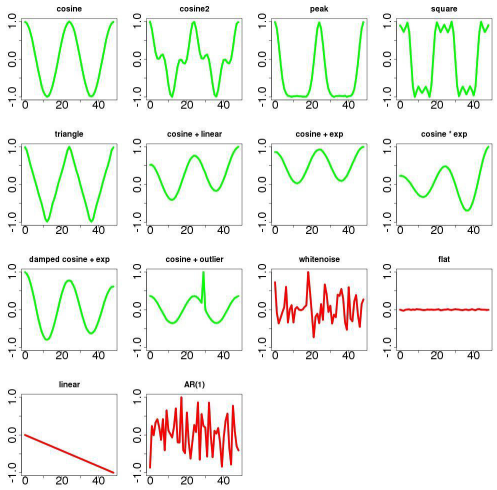


Figure 3.2: Samples of synthetic signals in the $\text{BioCycle}_{\text{Form}}$ dataset. Signals in green are periodic; signals in red are aperiodic.

quency is 1024 for JTK_Cycle, 20,000 for ARSER, and 40,000 for $\text{BioCycle}_{\text{Synth}}$. Finally, each signal in $\text{BioCycle}_{\text{Synth}}$ has three replicates, obtained by adding random Gaussian noise to the signal, to mimic typical biological experiments.

Biological Data

The performance of any circadian rhythm detection method requires extensive validation on biological datasets. In previous work, due to the aforementioned difficulty of not having ground truth labels, the biological signals detected as being periodic had to be inspected by hand, or loosely assessed by comparison to other methods [120]. In addition to the scaling problems associated with manual inspection, this approach did not allow the computation of precise classification metrics [6], such as the AUC- the Area Under the Receive Operating Characteristic (ROC) Curve. The repository of circadian data hosted on CircadiOmics [94] includes over 30 high-throughput circadian transcriptomic studies, as well as several circadian high-throughput metabolomic studies, that provide extensive coverage of different tissues and experimental conditions. From the CircadiOmics data, a high-quality biological dataset $\text{BioCycle}_{\text{Real}}$ is created with periodic/aperiodic labels.

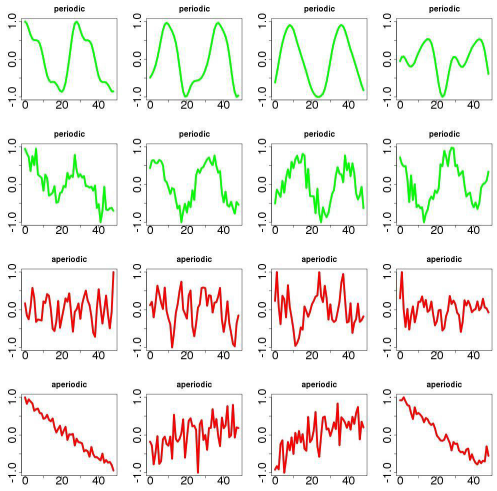


Figure 3.3: Samples of synthetic signals in the $\text{BioCycle}_{\text{Gauss}}$ dataset. Signals in green are periodic; signals in red are aperiodic.

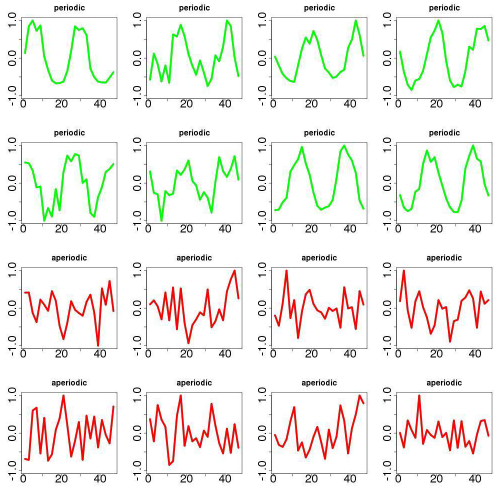


Figure 3.4: Samples of biological time series in the $\text{BioCycle}_{\text{Real}}$ dataset. Signals in green are periodic; signals in red are aperiodic. [Note the signals are spline-smoothed.]

To curate $\text{BioCycle}_{\text{Real}}$, we start from 36 circadian microarray or RNA-seq transcriptome datasets, 32 of which are currently publicly available from the CircadiOmics web portal (28 of these are also available from CircaDB[96]). Five datasets are from ongoing studies and will be added to CircadiOmics upon completion. All included datasets correspond to experiments carried out in mice, with the exception of one dataset corresponding to measurements taken in *Arabidopsis Thaliana*. $\text{BioCycle}_{\text{Real}}$ comprises experiments carried over a: 24-hour period with a 4 hour sampling rate; 48-hour period with a 2 hour sampling rate; and 48-hour period with a 1 hour sampling rate.

To extract from this set a high-quality subset of periodic time series, we focus on the time series associated with the core clock genes (Figure 3.1) in the control experiments. These gene include Clock, Per1, Per2, Per3, Cry1, Cry2, Nr1d1, Nr1d2, Bhlhe40, Bhlhe41, Dbp, Npas2, and Tel [49] for mouse, and the corresponding orthologs in *Arabidopsis* [48]. Arabdiposis orthologs were obtained from Affymetrix NetAffx probesets and annotations [78]. These core gene time series were further inspected manually to finally yield a set of 739 high-quality periodic signals.

To extract a high-quality biological aperiodic dataset, we start from the same body of data. To identify transcripts unlikely to be periodic, we select the transcripts classified as aperiodic consistently by all three programs JTK_Cycle, ARSER, and Lomb-Scargle with an associated p-value of 0.95. After further manual inspection, this yields a set of 18,094 aperiodic signals. Examples of signals taken at random from the $\text{BioCycle}_{\text{Real}}$ are shown in Figure 3.4.

3.2.2 Time Inference from Single Timepoint Measurements

To estimate the time associated with a transcriptomic experiment conducted at a single time-point, we curate the BioClock dataset starting from the same data in CircadiOmics, focusing on mouse data only for which we have enough training data. While in principle inference

of the time can be done using the level of expression of all the genes, exploratory feature selection and data reduction experiments (not shown) show that in most cases it is sufficient to focus on the set of core clock genes, or even a subset (see Results). Thus the reduced BioClock dataset contains microarray and RNA-Seq single time measurements for each gene transcript in the core clock with the associated timepoint. The BioClock dataset is organized by tissue and condition. Tissues include liver, kidney, heart, colon, glands (pituitary, adrenal), skeletal muscle, bone, white fat, and brown fat. Brain specific tissues include SCN (Suprachiasmatic nucleus), hippocampus, hypothalamus, and cerebellum. There are also several cell-specific datasets including mouse fibroblasts and macrophages. All the datasets in BioClock contain both control and treatment conditions. There is great variability among the treatment conditions (e.g. [31, 84]), varying from gene knock out and knock down (SIRT1 and SIRT6), to changes in diet (high fat, ketogenic), to diseases (epilepsy). It is important to be able to assess the ability of a system to predict time across tissues and conditions.

3.3 Methods

We experimented with several machine learning approaches for the the two main problems considered here. In general, the best results were obtained using neural networks. This is perhaps not too surprising since it is well known that neural networks have universal approximation properties and deep learning has led to state-of-the art performance, not only in several areas of engineering (e.g. computer vision, speech recognition, natural language processing, robotics) [123, 46, 75], but also in the natural sciences [23, 99, 79, 8]. Thus here we focus exclusively on deep learning approaches to build two systems, BIO_CYCLE and BIO_CLOCK, to address the two main problems.

3.3.1 Periodicity Inference from Time Series Measurements

Classifying Between Periodic and Aperiodic Signals

To classify signals as periodic or aperiodic, we train deep neural networks (DNNs) using standard gradient descent with momentum [105, 121]. We train separate networks for data sampled over 24 hours and 48 hours. The input to these networks are the expression time-series levels of the corresponding gene (or metabolite). The output is computed by a single logistic unit trained to be 1 when the signal is periodic and 0 otherwise, with relative entropy error function. We experimented with many hyperparameters and learning schedules. In the results reported, the learning rate starts at 0.01, and decays exponentially according to $\frac{0.1}{1.000087^t}$, where t is the iteration number. The training set consists of 1 million examples, a size sufficient to avoid overfitting. The DNN uses a mini-batch size of 100 and is trained for 50,000 iterations. Use of dropout [116, 7], or other forms of regularization, leads to no tangible improvements. The best performing DNN found (Figure 3.5a) has 3 hidden layers of size 100. We are able to obtain very good results by training BIO_CYCLE on synthetic data alone and report test results obtained on $\text{BioCycle}_{\text{Form}}$, $\text{BioCycle}_{\text{Gauss}}$, and $\text{BioCycle}_{\text{Real}}$.

Estimating the Period

In a way similar to how we train DNNs to classify between periodic and aperiodic signals, we can also train DNNs to estimate the period of a signal classified as periodic. During training, only periodic time series are used as input to train these regression DNNs . The output of the DNNs are implemented using a linear unit and produce an estimated value for the period. The error function is the squared error between the output of the network and the true period of the signal, which is known in advance with synthetic data. Except for the difference in the output unit, we use the same DNNs architectures and hyperparameters as

for the previous classification problem.

Estimating the Phase and the Lag

After the period p, we estimate the phase ϕ of a signal $\mathbf{s}$ by finding the value ϕ that maximizes the following expression: $\sum_{t \in T} cos(\frac{2\pi t}{p} - \phi)\mathbf{s}[t]$, where T is the set of all timepoints. Given ϕ, the lag (i.e. at what time the periodic pattern starts) is given by $\frac{\phi p}{2\pi}$.

Estimating the Amplitude

After the phase ϕ, we estimate the amplitude α by first removing any linear trend and then comparing the variance of the signal to the variance of a cosine signal with parameters ϕ, p, and amplitude 1. The formula is shown in Equation 3.3, where $\mu_{\mathbf{s}} = \frac{1}{|T|}\sum_{t \in T} \mathbf{s}[t]$ and $\mu_c = \frac{1}{|T|}\sum_{t \in T} cos(\frac{2\pi t}{p} - \phi)$

$$\alpha = \sqrt{\frac{\frac{1}{|T|}\sum_{t \in T}(\mathbf{s}[t] - \mu_{\mathbf{s}})^2}{\frac{1}{|T|}\sum_{t \in T}(cos(\frac{2\pi t}{p} - \phi) - \mu_c)^2}} \tag{3.3}$$

We cannot claim this approach is new, however, we have not seen it in previous literature. An alternative is to measure the amplitude on the smoothed time series.

Calculating p-values and q-values

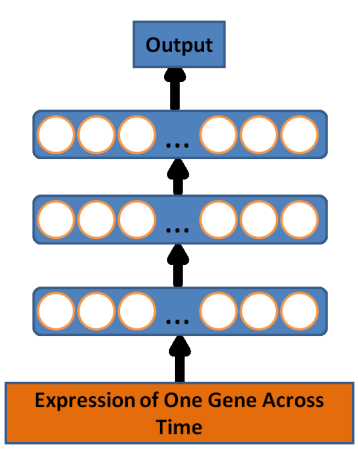
To calculate p-values, the distribution of the null hypothesis must first be obtained. To do this, N aperiodic signals are generated from one of the two $\text{BioCycle}_{\text{Synth}}$ datasets. Then we calculate the N output values $V(i)$ $(i = 1, \ldots, N)$ of the DNN on these aperiodic signals. The p-value for a new signal $\mathbf{s}$ with output value V is now $\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(V(i) > V)$, where $\mathbb{1}$

is the indicator function. This equation provides an empirical frequency estimate for the probability of obtaining an output of size V or greater, assuming that the signal $\mathbf{s}$ comes from the null distribution (the distribution of aperiodic signals). Therefore, the smaller the p-value, the more likely it is that $\mathbf{s}$ is periodic. The q-values are obtained through the Benjamini and Hochberg procedure [12].

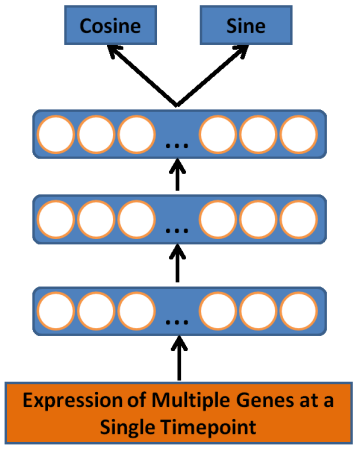
3.3.2 Time Inference from Single Timepoint Measurements

For this task, different machine learning methods were investigated, including simple linear regression, k-nearest neighbors, decision trees, shallow learning, and deep learning, including unsupervised compressive autoencoders [5] with two coupled phase (cosine/sine) units in the bottleneck layer (see Section 3.4.7). Supervised deep learning methods give the best results and are used in the final BIO_CLOCK system. The output of the DNNs is implemented using two coupled output units, representing the cosine and the sine of the phase angle (Figure 3.5b). If the total weighted inputs into these two units are S_1 and S_2 respectively, then the values of the two outputs units are given by: $S_1/\sqrt{S_1^2 + S_2^2}$ and $S_2/\sqrt{S_1^2 + S_2^2}$. These are then automatically converted into a time (ZT).

In order to better assess the effect of having data from different tissues, we experiment with both training specialized predictors trained on data originating from a single tissue, as well as predictors trained on data from all tissues. The final general-purpose predictor corresponds to a DNN trained on all the data. In each one of these experiments, we use 5-fold cross validation on the corresponding subset of the BioClock dataset, using architectures with 2 to 9 layers, and 100 to 600 units, to select the best network. A learning rate of 0.1 is typically used, with an exponential decay according to $\frac{0.1}{1.002^t}$. A visualization of the DNN is provided in Figure 3.5b.



(a) BIO_CYCLE. The output is either the binary periodic/aperiodic classification, or the regression estimate of the period of the signal.



(b) BIO_CLOCK. The outputs are the cosine and sine of the phase angle associated with the expression measurement of the core clock genes.

Figure 3.5: Visualizations of the deep neural networks (DNNs).

3.3.3 Data Normalization

For both the periodicity and time inference problems, training and testing examples are normalized to have a mean of zero and a standard deviation of one.

3.3.4 Software and Run Time

Downloadable software is currently written in R and Python and is intended to be easy for biologists to use. While exploring different models both Pylearn2 [43] and Caffe [58] were used. The DNNs typically take hours for training but, once trained, can process a real-world dataset (˜20,000 time series) in about one minute, both run times corresponding to a single CPU.

3.4 Results

In all the tables, the best results are shown in bold.

3.4.1 Periodic/Aperiodic Classification

For comparison, the methods ARSER (ARS), Lomb-Scargle (LS), and JTK_Cycle (JTK) are all evaluated along with the DNNs used by BIO_CYCLE, trained on the $\text{BioCycle}_{\text{Form}}$ and $\text{BioCycle}_{\text{Gauss}}$ datasets. In addition, we compare to MetaCycle (MC) [130]. To identify periodic signals, ARSER uses autoregressive spectral estimation, Lomb-Scargle uses a periodogram, and JTK_Cycle uses the Jonckheere-Terpstra's and the Kendall's tau tests. MetaCycle combines ARSER, Lomb-Scargle, and JTK_Cycle into one method.

To determine if the BIO_CYCLE results are significantly different from other methods, the testing set is randomly split into 10 equal-size, non-overlapping, subsets and the results from each subset are obtained. Then, a Student's t-test is performed between the results of the best of the two DNNs and the best of the previously existing methods. Finally, the p-value from that test is obtained to assess if the result differences are statistically significant. Small p-values (such as 0.05 and below) indicate that there is a significant difference between the methods. The p-values from the t-tests are shown in the rightmost column in all the tables. The results focus on periodic signals with periods around 24 hours, the most common case, however periods of 12 and 8 hours, corresponding to the second and third harmonics, are analyzed in Section 3.4.4.

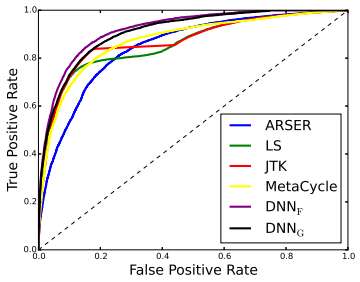
In the tables, the datasets $\text{BioCycle}_{\text{Form}}$, $\text{BioCycle}_{\text{Gauss}}$, and $\text{BioCycle}_{\text{Real}}$ are referred to as BC_{F}, BC_{G}, and BC_{R}, respectively.

Synthetic Data ($\mathbf{BioCycle_{Synth}}$)

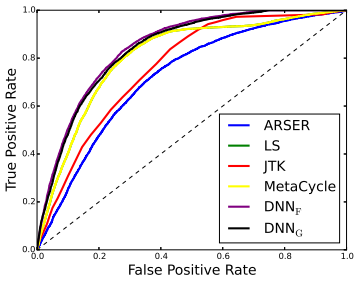
Results for the area under the receiver operating characteristic curve (AUC) for the task of classifying signals as periodic or aperiodic are shown in Table 3.1, and the ROC curves computed on $\text{BioCycle}_{\text{Form}}$ are shown in Figure 3.6. The DNN_{F} label corresponds to the DNN that has been trained on the $\text{BioCycle}_{\text{Form}}$ data, and the DNN_{G} label corresponds to the DNN that has been trained on the $\text{BioCycle}_{\text{Gauss}}$ data. The ROC curves computed on $\text{BioCycle}_{\text{Gauss}}$ are similar (not shown). The results from Table 3.1 show that the DNN method has better AUC than all the other published methods on the $\text{BioCycle}_{\text{Form}}$ and $\text{BioCycle}_{\text{Gauss}}$ datasets. Though the DNN does better when tested on data from the same distribution as it was trained on, it still outperforms all the other previous methods, regardless of which data it is trained on. A plot showing how the signal to noise ratio (SNR) affects performance is shown in Figure 3.7. This plot cannot be done for the $\text{BioCycle}_{\text{Gauss}}$ dataset, since in this case the exact SNR is not known. The DNN outperforms all the other published methods at all SNRs.

Table 3.1: AUC performance on synthetic data.

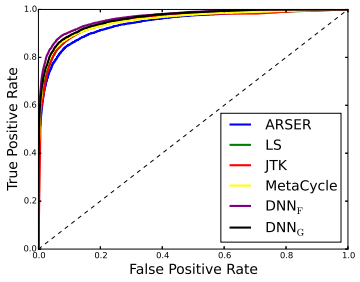
	ARS	LS	JTK	MC	DNN_{F}	DNN_{G}	t-test
BC_{F} (24_4)	0.85	0.86	0.87	0.87	**0.92**	0.91	0E+00
BC_{F}(24_6)	0.72	0.81	0.76	0.81	**0.85**	0.84	0E+00
BC_{F} (48_4)	0.94	0.95	0.95	0.95	**0.97**	0.96	3E-06
BC_{F} (48_8)	0.83	0.86	0.78	0.86	**0.89**	**0.89**	1E-06
BC_{F} (24_U)	0.80	0.84	0.85	0.84	**0.89**	0.88	0E+00
BC_{F} (48_U)	0.89	0.92	0.83	0.92	**0.94**	0.93	0E+00
BC_{G} (24_4)	0.85	0.89	0.89	0.89	0.92	**0.94**	0E+00
BC_{G} (24_6)	0.73	0.85	0.78	0.85	0.88	**0.89**	1E-06
BC_{G} (48_4)	0.96	0.95	0.95	0.96	**0.97**	**0.97**	5E-04
BC_{G} (48_8)	0.90	0.91	0.80	0.92	**0.93**	**0.93**	2E-06
BC_{G} (24_U)	0.84	0.89	0.88	0.89	0.91	**0.92**	0E+00
BC_{G} (48_U)	0.93	0.94	0.85	0.94	0.95	**0.96**	2E-06
ARS (44_4)	**0.99**	0.98	0.97	**0.99**	**0.99**	**0.99**	0E+00
JTK (48_1)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	2E-01
JTK (48_4)	0.96	0.97	**0.98**	**0.98**	**0.98**	0.97	1E+00



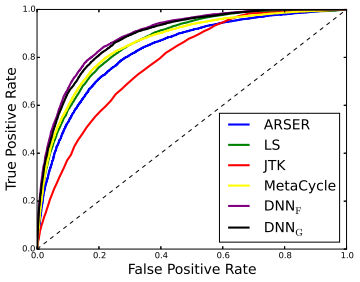
(a) BioCycle$_{\text{Form}}$ (24_4)



(b) BioCycle$_{\text{Form}}$ (24_6)

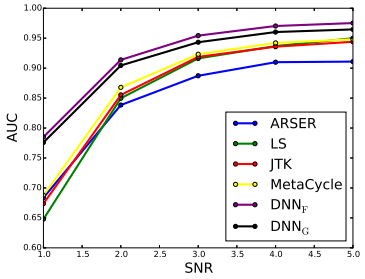


(c) BioCycle$_{\text{Form}}$ (48_4)

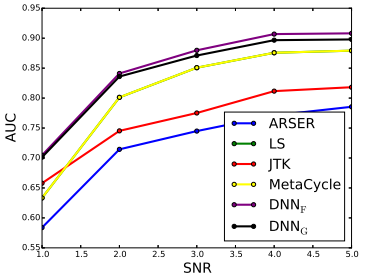


(d) BioCycle$_{\text{Form}}$ (48_8)

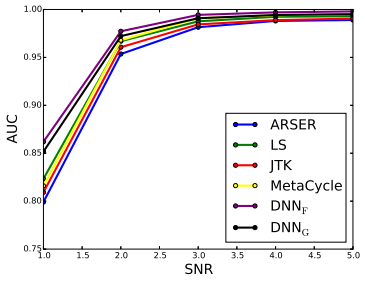
Figure 3.6: ROC Curves of different methods on the BioCycle$_{\text{Form}}$ dataset.



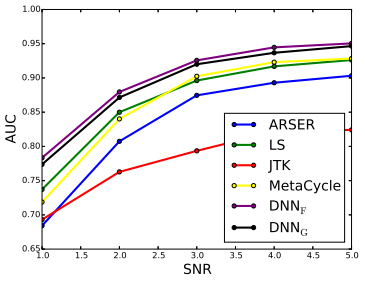
(a) BioCycle$_{\text{Form}}$ (24_4)



(b) BioCycle$_{\text{Form}}$ (24_6)



(c) BioCycle$_{\text{Form}}$ (48_4)



(d) BioCycle$_{\text{Form}}$ (48_8)

Figure 3.7: AUC at various signal-to-noise ratios (SNRs) on the BioCycle$_{\text{Form}}$ dataset. The lower the SNR the noisier the signal is.

Biological Data ($\mathbf{BioCycle_{Real}}$)

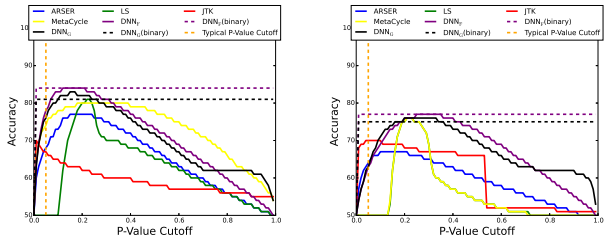
The performance on the biological dataset is shown in Table 3.2. Although the ARSER, LS, and JTK_Cycle methods achieve good performance on the aperiodic data, as can be expected since they were used to label the aperiodic data, the DNN method remains very competitive, often outperforming at least one of the other published methods.

Table 3.2: AUC performance on the biological dataset.

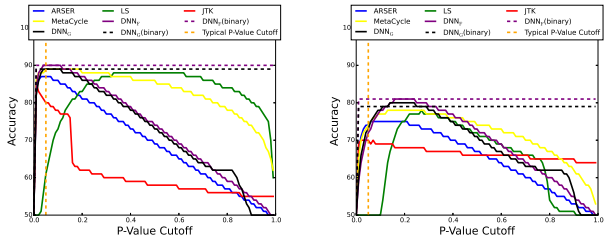
	ARS	LS	JTK	MC	DNN_F	DNN_G	t-test
BC_R (24_4)	**0.97**	**0.97**	0.89	**0.97**	**0.97**	**0.97**	7E-01
BC_R (48_1)	0.96	0.94	0.91	**0.98**	**0.98**	0.97	5E-01
BC_R (48_2)	**0.98**	0.97	0.95	0.96	0.94	0.95	3E-01

Evaluation of p-value Cutoffs

To investigate if the p-values obtained by BIO_CYCLE are reasonable, the accuracy of the periodic/aperiodic classification at different p-value cutoffs is evaluated. In addition to a p-value, BIO_CYCLE produces a binary classification. If the output of the DNN is greater than 0.5 the signals is labeled as periodic, otherwise, it is labeled as aperiodic. The accuracy using this binary classification is also evaluated. Results are shown in Figure 3.8. The vertical dashed line corresponds to a common p-value cutoff of 0.05. However, a proper p-value does not guarantee that the best accuracy will be at the cutoff of 0.05. Results show that BIO_CYCLE has the highest potential accuracy. It also has the best accuracy at 0.05 for 2 out of the 4 plots. In addition, the binary classification of BIO_CYCLE is almost always better than the accuracy of all the other methods at any p-value cutoff. The distribution of the p-values is shown in Figure 3.9.

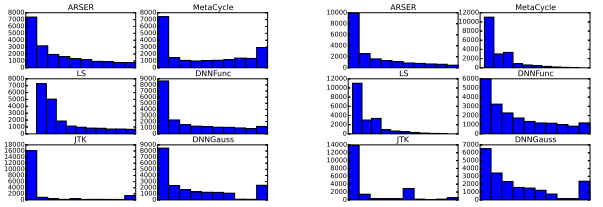


(a) BioCycle$_{\text{Form}}$ (24_4) (b) BioCycle$_{\text{Form}}$ (24_6)

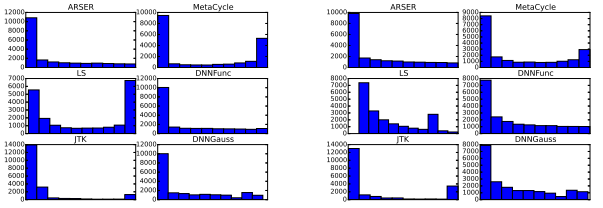


(c) BioCycle$_{\text{Form}}$ (48_4) (d) BioCycle$_{\text{Form}}$ (48_8)

Figure 3.8: Accuracy of periodic/aperiodic classification at different p-value cutoffs on the BioCycle$_{\text{Form}}$ dataset.



(a) BioCycle$_{\text{Form}}$ (24_4) (b) BioCycle$_{\text{Form}}$ (24_6)



(c) BioCycle$_{\text{Form}}$ (48_4) (d) BioCycle$_{\text{Form}}$ (48_8)

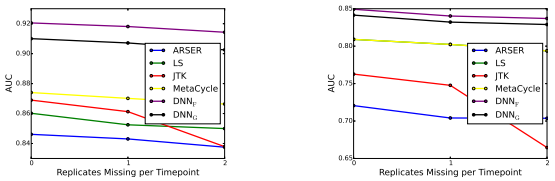
Figure 3.9: Histograms of p-values.

3.4.2 Period, Lag, and Amplitude Estimation

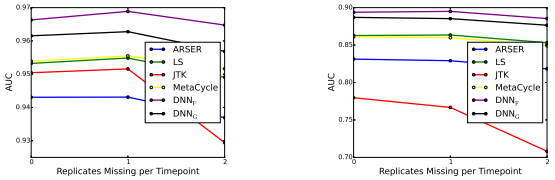
The metric to determine how well each method estimates the period, lag, and amplitude is given by the coefficient of determination R^2. The line $y = x$ corresponds to perfect prediction. In this case, y is the estimated value given by the method and x is the true value. R^2 measures how well the line $y = x$ fits the points that correspond to the true value vs the estimated value. Perfect prediction corresponds to $y = x$ and corresponds to $R^2 = 1$. The results for estimating the period, lag, and amplitude are shown in Tables 3.3, 3.4, and 3.5, respectively. For the $\text{BioCycle}_{\text{Gauss}}$ dataset we cannot control or know the exact lag or amplitude, so there are no results for $\text{BioCycle}_{\text{Gauss}}$ in Tables 3.4 and 3.5. These tables tell a similar story as Table 3.1. The DNN outperforms the other methods in most of the categories. Even when the DNN is tested on data associated with a distribution that is different from the distribution of its training set, in the majority of the cases it gives superior performance compared to ARSER, LS, and JTK_Cycle.

3.4.3 Missing Replicates and Missing Data

In gene expression experiments, replicate measurements can be missing. To investigate how missing replicates affect performance, the $\text{BioCycle}_{\text{Form}}$ dataset which has 3 replicates for each timepoint was used to assess performance with 0 replicates removed at each timepoint, 1 replicate removed at each timepoint, and 2 replicates removed at each timepoint. The results are shown in Figure 3.10 and show that JTK_Cycle is significantly affected in a negative way by missing replicates, while the performance of all the other methods degrades gracefully with the number of missing replicates, and minimally compared to JTK_Cycle. Missing data (timepoints at which there are no replicates) is also handled gracefully by BIO_CYCLE, while it is not handled at all by some of the other methods (not shown).



(a) BioCycle$_{\text{Form}}$ (24_4) (b) BioCycle$_{\text{Form}}$ (24_6)



(c) BioCycle$_{\text{Form}}$ (48_4) (d) BioCycle$_{\text{Form}}$ (48_8)

Figure 3.10: AUC at different levels of missing data.

3.4.4 Detecting Periods of 8 and 12 Hours

BioCycle$_{\text{Form}}$ and BioCycle$_{\text{Gauss}}$ datasets can be generated for different period ranges. In the main text, this range is 20-28, focusing on detecting signals with periods of 24 hours. Since there have also been genes discovered with periods of 12 and 8 hours, we generate BioCycle$_{\text{Form}}$ and BioCycle$_{\text{Gauss}}$ datasets with period ranges from 10-14 and from 7-9 to focus on the 12 and 8 hour periods, respectively, corresponding to the second and third harmonics. The results for detecting periods of 12 hours are shown in Ttables 3.6, 3.7, 3.8, and 3.9. The results for detecting periods of 8 hours are shown in tables 3.10, 3.11, 3.12, and 3.13.

The JTK and ARSER methods did not run on these datasets, so we only compare to LS and MetaCycle. We also note that the meta predictor MetaCycle, which uses JTK, ARSER, and LS, only chose to use LS for these datasets. The results show that BIO_CYCLE is the best choice in almost all cases.

Table 3.3: Coefficients of determinations (R^2) for the periods.

	ARS	LS	JTK	MC	DNN_F	DNN_G	t-test
BC_F (24_4)	0.02	0.22	0.17	0.19	**0.31**	0.27	0E+00
BC_F(24_6)	0.04	0.16	0.02	0.16	**0.22**	0.19	3E-04
BC_F (48_4)	0.59	0.64	0.51	0.65	**0.74**	0.73	5E-05
BC_F (48_8)	0.36	0.48	0.00	0.42	**0.57**	0.55	0E+00
BC_F (24_U)	0.05	0.20	0.06	0.20	**0.28**	0.24	0E+00
BC_F (48_U)	0.33	0.52	0.02	0.52	**0.62**	0.60	0E+00
BC_G (24_4)	0.02	0.27	0.20	0.24	0.35	**0.40**	0E+00
BC_G (24_6)	0.07	0.26	0.01	0.26	0.32	**0.36**	0E+00
BC_G (48_4)	0.70	0.68	0.53	0.72	0.80	**0.81**	0E+00
BC_G (48_8)	0.56	0.54	0.00	0.53	0.67	**0.69**	0E+00
BC_G (24_U)	0.06	0.25	0.03	0.25	0.32	**0.37**	0E+00
BC_G (48_U)	0.42	0.63	0.02	0.63	0.73	**0.75**	0E+00
ARS (44_4)	0.74	0.85	0.66	0.83	**0.89**	**0.89**	0E+00
JTK (48_1)	0.66	**0.94**	0.91	0.90	0.93	0.93	3E-03
JTK (48_4)	0.67	0.84	0.62	0.80	**0.85**	0.83	3E-02

3.4.5 BIO_CYCLE Web Server

The BIO_CYCLE portal within CircadiOmics at http://circadiomics.ics.uci.edu/biocycle allows users to upload an unpublished dataset for processing with BIO_CYCLE. For each experiment and each molecular species, individual P-value, q-value, period, amplitude, and phase can be obtained. Additionally, summary figures are generated for the distribution of each statistic in the user provided dataset. Trends for individual trajectories in user-provided data are available for search and visualization through the supplied set of molecular IDs. An example dataset is provided to give the user a sample of portal features and provide a template for desired data format. The BIO_CYCLE R package is also available for download through the main portal.

Table 3.4: Coefficients of determination (R^2) for the lags. The blank sqaures in LS and MC is due to the programs crashing on this dataset.

	ARS	LS	JTK	MC	DNN_F	DNN_G	t-test
BC_F (24_4)	0.36	0.37	0.27	0.42	**0.49**	**0.49**	8E-03
BC_F(24_6)	0.30		0.07		**0.45**	0.43	0E+00
BC_F (48_4)	0.50	0.14	0.31	0.50	**0.52**	0.51	5E-01
BC_F (48_8)	0.37	0.12	0.02	0.35	**0.42**	0.41	6E-03
BC_F (24_U)	0.34	0.31	0.10	0.32	**0.47**	**0.47**	0E+00
BC_F (48_U)	0.36	0.07	0.21	0.38	**0.49**	0.48	3E-04
ARS (44_4)	0.67	0.12	0.41	**0.69**	0.65	0.65	1E-01
JTK (48_1)	0.60	0.16	**0.80**	0.70	0.72	0.79	9E-01
JTK (48_4)	0.47	0.12	0.30	**0.55**	0.49	0.50	5E-01

Table 3.5: Coefficients of determination (R^2) for the amplitudes. The blank sqaures in LS and MC is due to the programs crashing on this dataset.

	ARS	LS	JTK	MC	DNN_F	DNN_G	t-test
BC_F (24_4)	0.81	0.63	0.86	**0.87**	0.81	0.81	2E-04
BC_F(24_6)	**0.81**		0.76		0.80	0.80	0E+00
BC_F (48_4)	0.82	0.55	**0.87**	0.84	0.75	0.75	0E+00
BC_F (48_8)	**0.80**	0.57	0.48	0.79	0.75	0.75	2E-02
BC_F (24_U)	0.68	0.62	0.84	**0.85**	0.80	0.80	2E-05
BC_F (48_U)	0.78	0.56	0.79	**0.83**	0.77	0.77	1E-03
ARS (44_4)	0.97	0.82	0.93	**0.99**	0.98	0.98	0E+00
JTK (48_1)	0.86	0.64	0.90	**0.93**	0.91	0.92	0E+00
JTK (48_4)	0.72	0.43	0.71	**0.74**	0.71	0.72	9E-01

3.4.6 Time Inference from Single Timepoint Measurements

Overall Performance

BIO_CLOCK is trained using 16 core clock genes: Arntl, Per1, Per2, Per3, Cyr1, Cry2, Nr1d1, Nr1d2, Bhlhe40, Bhlhe41, Dbp, Npas2, Tef, Fmo2, Lonrf3, and Tsc22d3. When trained and tested on all the data, using 70% of the data for training and the remaining 30% for testing, it accurately predicts the time of the experiment with a mean absolute error of 1.22 hours (less than 75 minutes) (Table 3.14). We experimented also with training BIO_CLOCK with an even smaller number of genes. For example, using only Arntl, Per1,

Table 3.6: AUC performance on synthetic data. Periodic data has periods between 10 and 14.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.89	0.89	0.87	**0.90**
BC_F(24_6)	0.83	0.83	**0.86**	**0.86**
BC_F (48_4)	0.94	0.94	**0.96**	0.95
BC_F (48_8)	0.81	0.81	**0.85**	0.83
BC_F (24_U)	0.86	0.86	**0.87**	**0.87**
BC_F (48_U)	0.90	0.90	**0.92**	0.91
BC_G (24_4)	0.93	0.93	0.92	**0.94**
BC_G (24_6)	0.88	0.88	**0.91**	**0.91**
BC_G (48_4)	0.95	0.95	0.96	**0.97**
BC_G (48_8)	0.88	0.88	0.89	**0.90**
BC_G (24_U)	0.91	0.91	**0.92**	**0.92**
BC_G (48_U)	0.93	0.93	**0.94**	**0.94**

Per2, Per3, Cry1, and Cry2, produces a mean absolute error of 3.72 hours. Adding Nr1d1 and Nr1d2 to this set reduces the mean absolute error to 1.65 hours.

Training and Testing on Different Organs/Tissues

Table 3.14 shows the mean absolute errors obtained when training BIO_CLOCK on data from certain organs/tissues and testing it on data from a different set of organs/tissues. All the data is from mice and under WT condition. The only datasets for which we have enough data for training correspond to liver and brain (when aggregating all the corresponding data sets). We form two additional sets (Set 1 and Set 2) by combining data from other organs. The first corresponds to combined data from the adrenal gland, fat, gut, kidney, lung, and muscle (Set 1). The second corresponds to combined data from the aorta, colon, fibroblast, heart, macrophages, and pituitary gland (Set 2). Finally, all of the aforementioned data is combined to form a bigger dataset (All). In all the experiments reported in Table 3.14, the data are split using a 70/30 training/test ratio, and tests sets never overlap with any of the corresponding training sets. The DNNs perform best when trained and tested on the

Table 3.7: Coefficients of determinations (R^2) for the periods. Periodic data has periods between 10 and 14.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.50	0.50	**0.57**	**0.57**
BC_F(24_6)	0.00	0.00	**0.01**	**0.01**
BC_F (48_4)	0.79	0.79	**0.85**	0.84
BC_F (48_8)	0.60	0.60	**0.70**	0.69
BC_F (24_U)	0.29	0.29	**0.47**	0.45
BC_F (48_U)	0.52	0.52	**0.66**	0.64
BC_G (24_4)	0.57	0.57	0.66	**0.68**
BC_G (24_6)	0.00	0.00	**0.01**	**0.01**
BC_G (48_4)	0.79	0.79	0.86	**0.87**
BC_G (48_8)	0.66	0.66	0.77	**0.78**
BC_G (24_U)	0.43	0.43	0.57	**0.60**
BC_G (48_U)	0.60	0.60	0.72	**0.74**

Table 3.8: Coefficients of determinations (R^2) for the lags. Periodic data has periods between 10 and 14.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.13	0.24	**0.39**	**0.39**
BC_F(24_6)			0.00	0.00
BC_F (48_4)	0.00	0.00	**0.47**	0.44
BC_F (48_8)	0.03	0.06	**0.33**	0.30
BC_F (24_U)	0.13	0.30	**0.34**	0.32
BC_F (48_U)	0.00	0.00	**0.34**	0.33

same organ/tissue or sets of organ/tissues or when trained on all the organs/tissues. The DNNs perform significantly worse when trained and tested on data with diverging origins. However, in all cases, the DNN trained on the combined dataset does almost as well as, or better than, the corresponding specialized DNN.

Training and Testing on Different Conditions

The collected data also includes data from mice under experimental conditions. The experimental conditions include high-fat and ketogenic diets, epilepsy, and SIRT1 and SIRT6 knockouts. This dataset is too small to build a training and testing set. However, one can

Table 3.9: Coefficients of determinations (R^2) for the amplitudes. Periodic data has periods between 10 and 14.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.61	**0.88**	0.86	0.86
BC_F(24_6)			**0.59**	0.58
BC_F (48_4)	0.42	0.58	0.83	**0.84**
BC_F (48_8)	0.57	0.56	**0.77**	**0.77**
BC_F (24_U)	0.60	0.75	**0.82**	**0.82**
BC_F (48_U)	0.44	0.63	**0.81**	**0.81**

Table 3.10: AUC performance on synthetic data. Periodic data has periods between 7 and 9.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.90	0.90	**0.92**	**0.92**
BC_F(24_6)	0.61	0.61	**0.80**	0.79
BC_F (48_4)	0.95	0.95	0.95	**0.96**
BC_F (48_8)	0.59	0.59	**0.85**	0.81
BC_F (24_U)	0.85	0.85	**0.88**	0.87
BC_F (48_U)	0.90	0.90	**0.92**	**0.92**
BC_G (24_4)	0.93	0.93	**0.94**	**0.94**
BC_G (24_6)	0.69	0.69	0.81	**0.85**
BC_G (48_4)	0.95	0.95	0.94	**0.97**
BC_G (48_8)	0.66	0.66	0.66	**0.73**
BC_G (24_U)	0.90	0.90	0.91	**0.92**
BC_G (48_U)	0.92	0.92	0.93	**0.94**

test the BIO_CLOCK DNN trained on the combined mice organs under normal conditions on this dataset. This experiment yields a mean absolute error of 2.57 hours.

3.4.7 Autoencoders and Manifold Learning

We also investigated an alternative unsupervised manifold learning approach for automatically extracting the time associated with a high-throughput transcriptomic measurement taken at a single timepoint. The basic idea is to use a compressive autoencoder with a bottleneck consisting of two special units (Figure 3.11). The autoencoder can be applied to the full sets of measurements, or to a subset (e.g. the core clock genes). In trying to

Table 3.11: Coefficients of determinations (R^2) for the periods. Periodic data has periods between 7 and 9.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.01	0.01	**0.03**	0.01
BC_F(24_6)	0.36	0.36	**0.44**	0.42
BC_F (48_4)	0.02	0.02	**0.14**	0.10
BC_F (48_8)	0.00	0.00	**0.02**	0.01
BC_F (24_U)	0.40	0.40	**0.50**	0.48
BC_F (48_U)	0.50	0.50	**0.65**	0.63
BC_G (24_4)	0.00	0.00	0.01	**0.02**
BC_G (24_6)	0.46	0.46	0.54	**0.56**
BC_G (48_4)	0.04	0.04	0.14	**0.16**
BC_G (48_8)	0.01	0.01	0.02	**0.04**
BC_G (24_U)	0.48	0.48	0.59	**0.62**
BC_G (48_U)	0.59	0.59	0.74	**0.76**

Table 3.12: Coefficients of determinations (R^2) for the lags. Periodic data has periods between 7 and 9.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.00	0.00	0.00	**0.01**
BC_F(24_6)			**0.45**	0.44
BC_F (48_4)	0.00	0.00	0.00	0.00
BC_F (48_8)	0.00	0.00	0.00	0.00
BC_F (24_U)	0.07	0.12	**0.26**	0.25
BC_F (48_U)	0.01	0.02	**0.17**	**0.17**

reconstruct the input data in the final output layer, the autoencoder must compress the data through these two units optimally in a way that hopefully correspond to the cosine and sine of the phase angle, up to a circular shift. If the activations of these two units are S_1 and S_2, then their two outputs are given by: $S_1/\sqrt{S_1^2 + S_2^2}$ and $S_2/\sqrt{S_1^2 + S_2^2}$. The autoencoder can be trained using large amounts of unlabeled data, for instance taken in GEO. While this approach generates interesting results, the supervised approach used to train BIO_CLOCK so far yields better results.

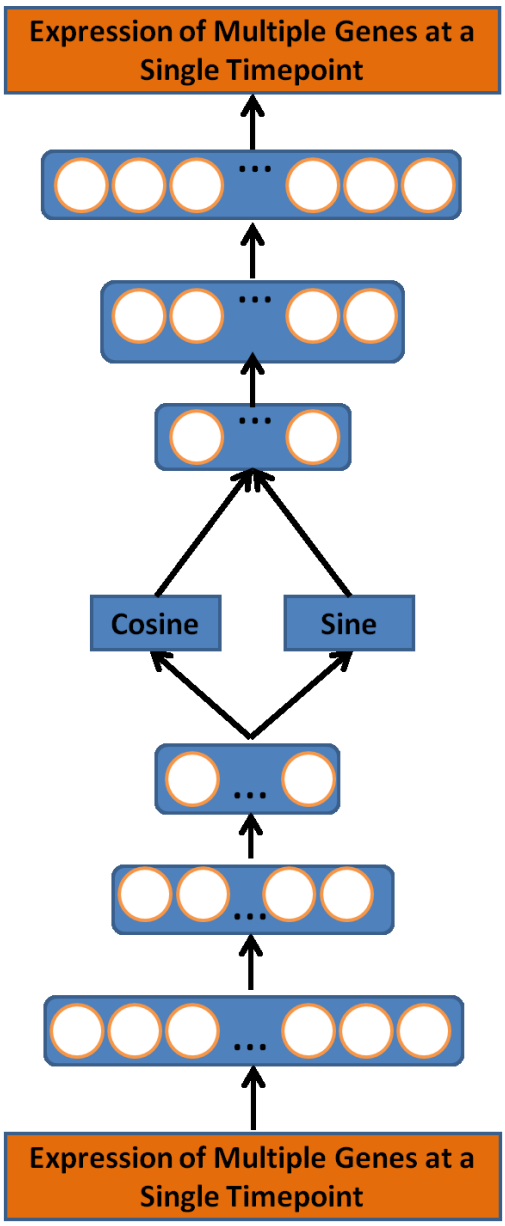


Figure 3.11: A visualization of an autoencoder with a cosine and sine unit as the bottleneck.

Table 3.13: Coefficients of determinations (R^2) for the amplitudes. Periodic data has periods between 7 and 9.

	LS	MC	DNN_F	DNN_G
BC_F (24_4)	0.52	0.49	**0.69**	**0.66**
BC_F(24_6)			**0.80**	**0.80**
BC_F (48_4)	0.42	0.10	0.57	**0.59**
BC_F (48_8)	**0.47**	0.01	0.03	0.02
BC_F (24_U)	0.59	0.61	**0.74**	**0.74**
BC_F (48_U)	0.46	0.62	**0.77**	**0.77**

Table 3.14: Cross organ mean absolute error (MAE) comparison of BIO_CLOCK.

		Testing				
		Liver	Brain	Set1	Set2	All
Training	Liver	**1.21**	5.18	3.78	4.77	3.78
	Brain	3.94	**1.50**	3.28	5.39	3.84
	Set1	4.06	4.25	2.03	4.69	3.58
	Set2	2.31	4.10	2.14	0.75	2.00
	All	1.28	1.66	**1.49**	**0.70**	**1.22**

3.5 Conclusion

Deep learning methods can be applied to high-throughput circadian data to address important challenges in circadian biology. In particular, we have developed BIO_CYCLE to detect molecular species that oscillate in high-throughput circadian experiments and extract the characteristics of these oscillations. Remarkably, BIO_CYCLE can be trained with large quantities of synthetic data preventing any kind of overfitting. We have also developed BIO_CLOCK to infer the time at which a transcriptomic sample was collected from the level of expression of a small number of core clock genes. Both methods will be improved as more data becomes available and, more generally, deep learning methods are likely to be useful to address several other related circadian problems, such as analyzing periodicity in high-throughput circadian proteomic data, or inferring sample time in different species. In particular, developing methods for annotating the time of all the *human* gene expression experiments, contained in GEO, and other similar repositories, would be valuable. Such annotations could be important for improving the interpretation of both old and new data

and discovering circadian driven effects that may be important in precision medicine and other applications, for instance to help determine the optimal time for administering certain drugs.

Chapter 4

Learning Activation Functions to Improve Deep Neural Networks

Artificial neural networks typically have a fixed, non-linear activation function at each neuron. We have designed a novel form of piecewise linear activation function that is learned independently for each neuron using gradient descent. With this adaptive activation function, we are able to improve upon deep neural network architectures composed of static rectified linear units, achieving classification performance of 7.51% on CIFAR-10, 30.83% on CIFAR-100, and improving on a benchmark from high-energy physics involving Higgs boson decay modes.

4.1 Introduction

Deep learning with artificial neural networks has enabled rapid progress on applications in engineering [72, 47] and basic science [24, 80, 8]. Usually, the parameters in the linear components are learned to fit the data, while the nonlinearities are pre-specified to be a logistic,

tanh, rectified linear, or max-pooling function. A sufficiently large neural network using any of these common nonlinear functions can approximate arbitrarily complex functions [53, 18], but in finite networks the choice of nonlinearity affects both the learning dynamics (especially in deep networks) and the network's expressive power.

Designing activation functions that enable fast training of accurate deep neural networks is an active area of research. The rectified linear activation function [57, 40], which does not saturate like sigmoidal functions, has made it easier to quickly train deep neural networks by alleviating the difficulties of weight-initialization and vanishing gradients. Another recent innovation is the "maxout" activation function, which has achieved state-of-the-art performance on multiple machine learning benchmarks [44]. The maxout activation function computes the maximum of a set of linear functions, and has the property that it can approximate any convex function of the input. [115] replaced the max function with a probabilistic max function and [45] explored an activation function that replaces the max function with an L_P norm. However, while the type of activation function can have a significant impact on learning, the space of possible functions has hardly been explored.

One way to explore this space is to *learn* the activation function during training. Previous efforts to do this have largely focused on genetic and evolutionary algorithms [133], which attempt to select an activation function for each neuron from a pre-defined set. Recently, [127] combined this strategy with a single scaling parameter that is learned during training.

In this paper, we propose a more powerful adaptive activation function. This parametrized, piecewise linear activation function is learned independently for each neuron using gradient descent, and can represent both convex and non-convex functions of the input. Experiments demonstrate that, like other piecewise linear activation functions, this works well for training deep neural networks.

4.2 Adaptive Piecewise Linear Units

Here we define the adaptive piecewise linear (APL) activation unit. Our method formulates the activation function $h_i(x)$ of an APL unit i as a sum of hinge-shaped functions,

$$h_i(x) = \max(0, x) + \sum_{s=1}^{S} a_i^s \max(0, -x + b_i^s) \tag{4.1}$$

The result is a piecewise linear activation function. The number of hinges, S, is a hyperparameter set in advance, while the variables a_i^s, b_i^s for $i \in 1, ..., S$ are learned using standard gradient descent during training. The a_i^s variables control the slopes of the linear segments, while the b_i^s variables determine the locations of the hinges.

The number of additional parameters that must be learned when using these APL units is $2SM$, where M is the total number of hidden units in the network. This number is small compared to the total number of weights in typical networks.

Figure 4.1 shows example APL functions for $S = 1$. Note that unlike maxout, the class of functions that can be learned by a single unit includes non-convex functions. In fact, for large enough S, $h_i(x)$ can approximate arbitrarily complex continuous functions, subject to two conditions:

Theorem 4.1. *Any continuous piecewise-linear function $g(x)$ can be expressed by Equation 4.1 for some S, and a_i, b_i, $i \in 1, ..., S$, assuming that:*

1. *There is a scalar u such that $g(x) = x$*

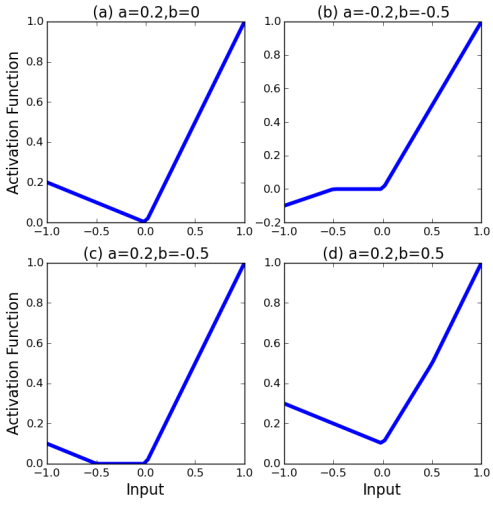


Figure 4.1: Sample activation functions obtained from changing the parameters. Notice that figure b shows that the activation function can also be non-convex. Asymptotically, the activation functions tend to $g(x) = x$ as $x \to \infty$ and $g(x) = \alpha x - c$ as $x \leftarrow -\infty$ for some α and c. $S = 1$ for all plots.

for all $x \geq u$.

2. *There are two scalars* v *and* α *such that* $\nabla_x g(x) = \alpha$ *for all* $x < v$.

This theorem implies that we can reconstruct any piecewise-linear function $g(x)$ over any subset of the real line, and the two conditions on $g(x)$ constrain the behavior of $g(x)$ to be linear as x gets very large or small. The first condition is less restrictive than it may seem. In neural networks, $g(x)$ is generally only of interest as an input to a linear function $wg(x) + z$; this linear function effectively restores the two degrees of freedom that are eliminated by constraining the rightmost segment of $g(x)$ to have unit slope and bias 0.

Proof. Let $g(x)$ be piecewise linear with $K+2$ linear regions separated by ordered boundary points b^0, b^1, ...,b^K, and let a^k be the slope of the k-th region. Assume also that $g(x) = x$ for all $x \geq b^K$. We show that $g(x)$ can be expressed by the following special case of Equation 4.1:

$$\begin{aligned} h(x) \equiv & -a^0 \max(0, -x + b^0) \\ & + \textstyle\sum_{k=1}^{K} a^k (\max(0, -x + b^{k-1}) - \max(0, -x + b^k)) \\ & - \max(0, -x) + \max(0, x) + \max(0, -x + b^K), \end{aligned} \tag{4.2}$$

The first term has slope a^0 in the range $(-\infty, b^0)$ and 0 elsewhere. Each element in the summation term of Equation 4.2 has slope a^k over the range (b^{k-1}, b^k) and 0 elsewhere. The last three terms together have slope 1 when $x \in (b^K, \infty)$ and 0 elsewhere. Now, $g(x)$ and $h(x)$ are continuous, their slopes match almost everywhere, and it is easily verified that $h(x) = g(x) = x$ for $x \geq b^K$. Thus, we conclude that $h(x) = g(x)$ for all x. □ □

4.2.1 Comparison with Other Activation Functions

In this section we compare the proposed approach to learning activation functions with two other nonlinear activation functions: maxout [44], and network-in-network [77].

We observe that both maxout units and network-in-network can learn any nonlinear activation function that APL units can, but require many more parameters to do so. This difference allows APL units to be applied in very different ways from maxout and network-in-network nonlinearities: the small number of parameters needed to tune an APL unit makes it practical to train convolutional networks that apply different nonlinearities at each point in each feature map, which would be completely impractical in either maxout networks or network-in-network approaches.

Maxout. Maxout units differ from traditional neural network nonlinearities in that they take as input the output of *multiple* linear functions, and return the largest:

$$h_{\text{maxout}}(x) = \max_{k \in \{1,\ldots,K\}} w^k \cdot x + b^k. \tag{4.3}$$

Incorporating multiple linear functions increases the expressive power of maxout units, allowing them to approximate arbitrary convex functions, and allowing the difference of a pair of maxout units to approximate arbitrary functions.

Networks of maxout units with a particular weight-tying scheme can reproduce the output of an APL unit. The sum of terms in Equation 4.1 with positive coefficients (including the initial $\max(0, x)$ term) is a convex function, and the sum of terms with negative coefficients is a concave function. One could approximate the convex part with one maxout unit, and

the concave part with another maxout unit:

$$h^{\text{convex}}(x) = \max_k c_k^{\text{convex}} w \cdot x + d_k^{\text{convex}}; \quad h^{\text{concave}}(x) = \max_k c_k^{\text{concave}} w \cdot x + d_k^{\text{concave}}, \tag{4.4}$$

where c and d are chosen so that

$$h^{\text{convex}}(x) - h^{\text{concave}}(x) = \max(0, w \cdot x + u) + \textstyle\sum_s a^s \max(0, w \cdot x + u). \tag{4.5}$$

In a standard maxout network, however, the w vectors are not tied. So implementing APL units (Equation 4.1) using a maxout network would require learning $O(SK)$ times as many parameters, where K is the size of the maxout layer's input vector. Whenever the expressive power of an APL unit is sufficient, using the more complex maxout units is therefore a waste of computational and modeling power.

Network-in-Network. [77] proposed replacing the simple rectified linear activation in convolutional networks with a fully connected network whose parameters are learned from data. This "MLPConv" layer couples the outputs of all filters applied to a patch, and permits arbitrarily complex transformations of the inputs. A depth-M MLPConv layer produces an output vector f_{ij}^M from an input patch x_{ij} via the series of transformations

$$f_{ijk}^1 = \max(0, w_k^1 \cdot x_{ij} + b_k^1), \ldots, f_{ijk}^M = \max(0, w_k^M \cdot f_{ij}^{M-1} + b_k^M). \tag{4.6}$$

As with maxout networks, there is a weight-tying scheme that allows an MLPConv layer to reproduce the behavior of an APL unit:

$$f_{ijk}^1 = \max(0, c_k w_{\kappa(k)} \cdot x_{ij} + b_k^1), f_{ijk}^2 = \textstyle\sum_{\ell|\kappa(\ell)=k} a_k f_{ij\ell}^1, \tag{4.7}$$

where the function $\kappa(k)$ maps from hinge output indices k to filter indices κ, and the coeffi-

cient $c_k \in \{-1, 1\}$.

This is a very aggressive weight-tying scheme that dramatically reduces the number of parameters used by the MLPConv layer. Again we see that it is a waste of computational and modeling power to use network-in-network wherever an APL unit would suffice.

However, network-in-network can do things that APL units cannot—in particular, it efficiently couples and summarizes the outputs of multiple filters. One can get the benefits of both architectures by replacing the rectified linear units in the MLPconv layer with APL units.

4.3 Experiments

Experiments were performed using the software package CAFFE [58]. The hyperparameter, S, that controls the complexity of the activation function was determined using a validation set for each dataset. The a_i^s and b_i^s parameters were regularized with an L2 penalty, scaled by 0.001. Without this penalty, the optimizer is free to choose very large values of a_i^s balanced by very small weights, which would lead to numerical instability. We found that adding this penalty improved results. The model files and solver files are available at https://github.com/ForestAgostinelli/Learned-Activation-Functions-Source/tree/master.

4.3.1 CIFAR

The CIFAR-10 and CIFAR-100 datasets [71] are 32x32 color images that have 10 and 100 classes, respectively. They both have 50,000 training images and 10,000 test images. The images were preprocessed by subtracting the mean values of each pixel of the training set from each image. Our network for CIFAR-10 was loosely based on the network used in [116].

It had 3 convolutional layers with 96, 128, and 256 filters, respectively. Each kernel size was 5x5 and was padded by 2 pixels on each side. The convolutional layers were followed by a max-pooling, average-pooling, and average-pooling layer, respectively; all with a kernel size of 3 and a stride of 2. The two fully connected layers had 2048 units each. We applied dropout [52] to the network as well. We found that applying dropout both before *and* after a pooling layer increased classification accuracy. The probability of a unit being dropped before a pooling layer was 0.25 for all pooling layers. The probability for them being dropped after each pooling layers was 0.25, 0.25, and 0.5, respectively. The probability of a unit being dropped for the fully connected layers was 0.5 for both layers. The final layer was a softmax classification layer. For CIFAR-100, the only difference was the second pooling layer was max-pooling instead of average-pooling. The baseline used rectified linear activation functions.

When using the APL units, for CIFAR-10, we set $S = 5$. For CIFAR-100 we set $S = 2$. Table 4.1 shows that adding the APL units improved the baseline by over 1% in the case of CIFAR-10 and by almost 3% in the case of CIFAR-100. *In terms of relative difference, this is a 9.4% and a 7.5% decrease in error rate, respectively.* We also try the network-in-network architecture for CIFAR-10 [77]. We have $S = 2$ for CIFAR-10 and $S = 1$ for CIFAR-100. We see that it improves performance for both datasets.

We also try our method with the augmented version of CIFAR-10 and CIFAR-100. We pad the image all around with a four pixel border of zeros. For training, we take random 32 x 32 crops of the image and randomly do horizontal flips. For testing we just take the center 32 x 32 image. To the best of our knowledge, the results we report for data augmentation using the network-in-network architecture *are the best results reported for CIFAR-10 and CIFAR-100 for any method.*

In section 4.3.4, one can observe that the learned activations can look similar to leaky rectified linear units (Leaky ReLU) [81]. This activation function is slightly different than the ReLU

because it has a small slope k when the input $x < 0$.

$$h(x) = \begin{cases} x, & \text{if } x > 0 \\ kx, & \text{otherwise} \end{cases}$$

In [81], k is equal to 0.01. To compare Leaky ReLUs to our method, we try different values for k and pick the best value one. The possible values are positive and negative 0.01, 0.05, 0.1, and 0.2. For the standard convolutional neural network architecture $k = 0.05$ for CIFAR-10 and $k = -0.05$ for CIFAR-100. For the network-in-network architecture $k = 0.05$ for CIFAR-10 and $k = 0.2$ for CIFAR-100. APL units consistently outperform leaky ReLU units, showing the value of tuning the nonlinearity (see also section 4.3.3).

4.3.2 Higgs Boson Decay

The Higgs-to-$\tau^+\tau^-$ decay dataset comes from the field of high-energy physics and the analysis of data generated by the Large Hadron Collider [9]. The dataset contains 80 million collision events, characterized by 25 real-valued features describing the 3D momenta and energies of the collision products. The supervised learning task is to distinguish between two types of physical processes: one in which a Higgs boson decays into $\tau^+\tau^-$ leptons and a background process that produces a similar measurement distribution. Performance is measured in terms of the area under the receiver operating characteristic curve (AUC) on a test set of 10 million examples, and in terms of discovery significance [20] in units of Gaussian σ, using 100 signal events and 5000 background events with a 5% relative uncertainty.

Our baseline for this experiment is the 8 layer neural network architecture from [9] whose architecture and training hyperparameters were optimized using the Spearmint algorithm [114]. We used the same architecture and training parameters except that dropout was used in the

top two hidden layers to reduce overfitting. For the APL units we used $S = 2$. Table 4.2 shows that has higher performance than the dropout-trained baseline and the ensemble of 5 neural networks from [9].

4.3.3 Effects of APL unit Hyperparameters

Table 4.3 shows the effect of varying S on the CIFAR-10 benchmark. We also tested whether learning the activation function was important (as opposed to having complicated, *fixed* activation functions). For $S = 1$, we tried freezing the activation functions at their random initialized positions, and not allowing them to learn. The results show that learning activations, as opposed to keeping them fixed, results in better performance.

4.3.4 Visualization and Analysis of Adaptive Piecewise Linear Functions

The diversity of adaptive piecewise linear functions was visualized by plotting $h_i(x)$ for sample neurons. Figures 4.2 and 4.3 show adaptive piecewise linear functions for the CIFAR-100 and Higgs$\rightarrow \tau^+\tau^-$ experiments, along with the random initialization of that function.

In figure 4.4, for each layer, 1000 activation functions (or the maximum number of activation functions for that layer, whichever is smaller) are plotted. One can see that there is greater variance in the learned activations for CIFAR-100 than there is for CIFAR-10. There is greater variance in the learned activations for Higgs$\rightarrow \tau^+\tau^-$ than there is for CIFAR-100. For the case of Higgs$\rightarrow \tau^+\tau^-$, a trend that can be seen is that the variance decreases in the higher layers.

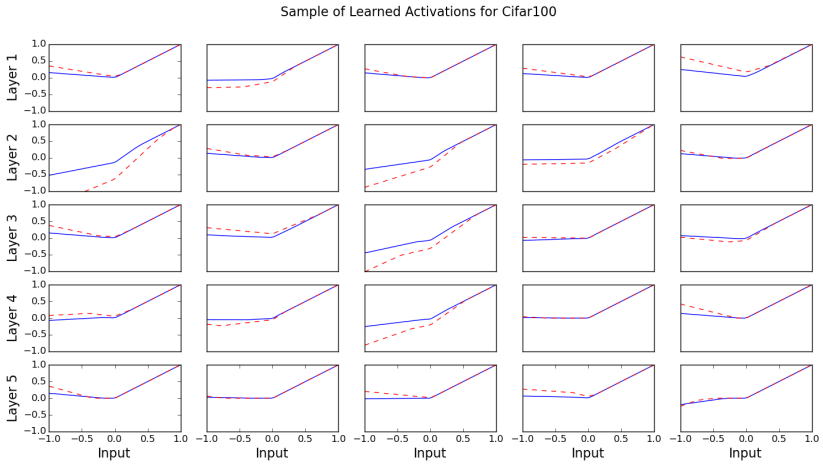


Figure 4.2: CIFAR-100 Sample Activation Functions. Initialization (dashed line) and the final learned function (solid line).

4.4 Conclusion

We have introduced a novel neural network activation function in which each neuron computes an independent, piecewise linear function. The parameters of each neuron-specific activation function are learned via gradient descent along with the network's weight parameters. Our experiments demonstrate that learning the activation functions in this way can lead to significant performance improvements in deep neural networks without significantly increasing the number of parameters. Furthermore, the networks learn a diverse set of activation functions, suggesting that the standard one-activation-function-fits-all approach may be suboptimal.

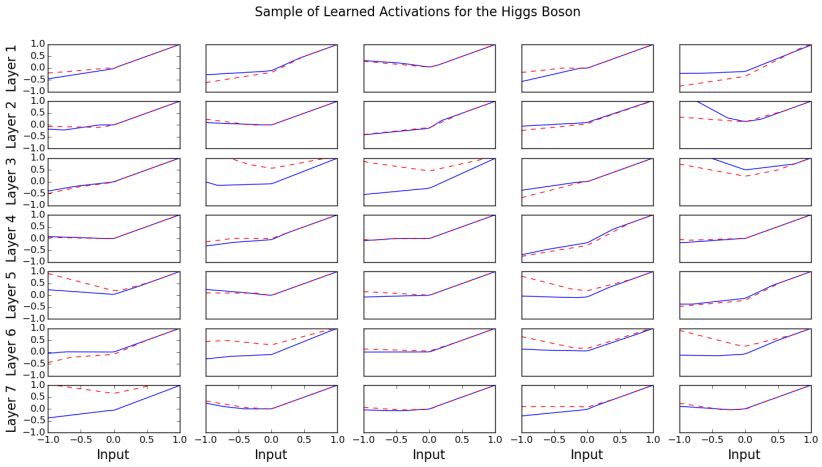


Figure 4.3: Higgs$\rightarrow \tau^+\tau^-$ Sample Activation Functions. Initialization (dashed line) and the final learned function (solid line).

Table 4.1: Error rates on CIFAR-10 and CIFAR-100 with and without data augmentation. This includes standard convolutional neural networks (CNNs) and the network-in-network (NIN) architecture [77]. The networks were trained 5 times using different random initializations — we report the mean followed by the standard deviation in parenthesis. The best results are in bold.

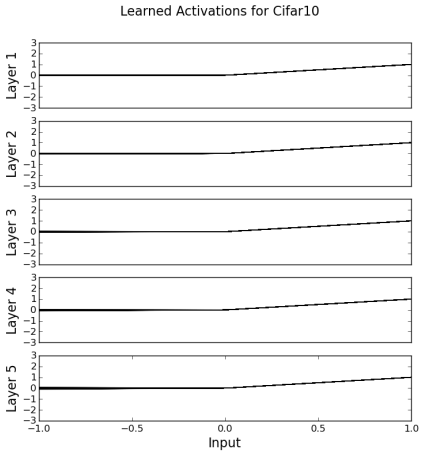
Method	**CIFAR-10**	**CIFAR-100**
Without Data Augmentation		
CNN + ReLU [116]	12.61%	37.20%
CNN + Channel-Out [129]	13.2%	36.59%
CNN + Maxout [44]	11.68%	38.57%
CNN + Probout [115]	**11.35%**	38.14%
CNN (Ours) + ReLU	12.56 (0.26)%	37.34 (0.28)%
CNN (Ours) + Leaky ReLU	11.86 (0.04)%	35.82 (0.34)%
CNN (Ours) + APL units	11.38 (0.09)%	**34.54 (0.19)%**
NIN + ReLU [77]	10.41%	35.68%
NIN + ReLU + Deep Supervision [74]	9.78%	34.57%
NIN (Ours) + ReLU	9.67 (0.11)%	35.96 (0.13)%
NIN (Ours) + Leaky ReLU	9.75 (0.22)%	36.00 (0.36)%
NIN (Ours) + APL units	**9.59 (0.24)%**	**34.40 (0.16)%**
With Data Augmentation		
CNN + Maxout [44]	9.38%	-
CNN + Probout [115]	9.39%	-
CNN + Maxout [117]	9.61%	34.54%
CNN + Maxout + Selective Attention [117]	**9.22%**	**33.78%**
CNN (Ours) + ReLU	9.99 (0.09)%	34.50 (0.12)%
CNN (Ours) + APL units	9.89 (0.19)%	33.88 (0.45)%
NIN + ReLU [77]	8.81%	-
NIN + ReLU + Deep Supervision [74]	8.22%	-
NIN (Ours) + ReLU	7.73 (0.13)%	32.75 (0.13)%
NIN (Ours) + APL units	**7.51 (0.14)%**	**30.83 (0.24)%**

Table 4.2: Performance on the Higgs boson decay dataset in terms of both AUC and expected discovery significance. The networks were trained 4 times using different random initializations — we report the mean followed by the standard deviation in parenthesis. The best results are in bold.

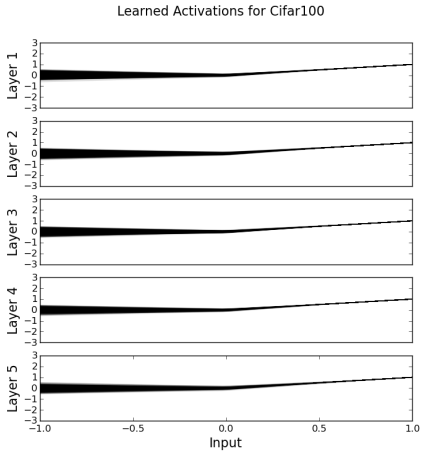
Method	**AUC**	**Discovery Significance**
DNN + ReLU [9]	0.802	3.37σ
DNN + ReLU + Ensemble[9]	0.803	3.39σ
DNN (Ours) + ReLU	0.803 (0.0001)	3.38 (0.008) σ
DNN (Ours) + APL units	**0.804 (0.0002)**	**3.41 (0.006)** σ

Table 4.3: Classification accuracy on CIFAR-10 for varying values of S. Shown are the mean and standard deviation over 5 trials.

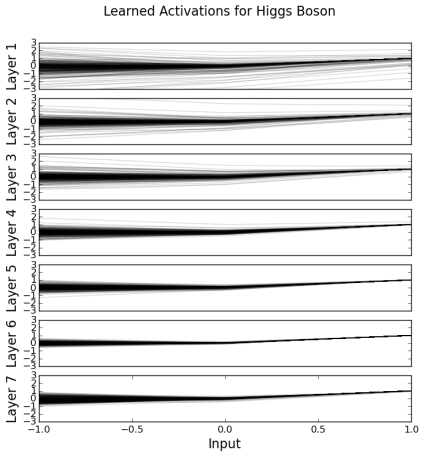
Values of S	**Error Rate**
baseline	12.56 (0.26)%
$S = 1$ (activation not learned)	12.55 (0.11)%
$S = 1$	11.59 (0.16)%
$S = 2$	11.73 (0.23)%
$S = 5$	**11.38 (0.09)%**
$S = 10$	11.60 (0.16)%



(a) CIFAR-10 Activation Functions.



(b) CIFAR-100 Activation Functions.



(c) Higgs$\rightarrow \tau^+\tau^-$ Activation Functions.

Figure 4.4: Visualization of the range of the values for the learned activation functions for the deep neural network for the CIFAR datasets and Higgs$\rightarrow \tau^+\tau^-$ dataset.

Chapter 5

Conclusion

We have investigated how to solve puzzles that have large state spaces with only one solved state using a deep reinforcement learning method called DeepCubeA. We showed that not only can DeepCubeA solve these problems, it can often solve them in the most efficient way possible. Next, we investigated how deep learning can be applied to problems in circadian rhythms. Our deep learning algorithm BIO_CYCLE could detect oscillating circadian species better than current software and that our deep learning software BIO_CLOCK could determine the time measurements were taken. Finally, we introduced the adaptive piecewise linear units (APLs). These activation functions were learned using gradient descent and lead to significant increases in performance.

APLs have been shown to work for classification tasks and could possibly be used for other tasks that can be solved with deep neural networks, such as dimensionality reduction and regression. Deep reinforcement learning algorithms, such as DeepCubeA, could also potentially benefit, however, the loss function is often non-stationary for deep reinforcement learning tasks. Therefore, APLs may need modification for deep reinforcement learning applications.

DeepCubeA could be applied problems in the sciences that have many possibilities, where

only a small fraction of those possibilities are considered solutions. For example, protein structure prediction can be viewed as a local search problem in a large state space. While solving the Rubik's cube is a path finding problem and not a local search problem, the concepts used to solve this puzzle could be extended to other search problems found in the sciences. Furthermore, finding solutions to problems with large state spaces should also be efficient in order to conserve time and resources. While DeepCubeA was able to solve the majority of test configurations in the most efficient way possible, or close to the most efficient way possible, there were no guarantees that this would be the case. It is possible that this algorithm can be extended to guarantee efficiency with high probability.

Bibliography

[1] J. L. Andrews, X. Zhang, J. J. McCarthy, E. L. McDearmon, T. A. Hornberger, B. Russell, K. S. Campbell, S. Arbogast, M. B. Reid, J. R. Walker, J. B. Hogenesch, J. S. Takahashi, and K. A. Esser. Clock and bmal1 regulate myod and are necessary for maintenance of skeletal muscle phenotype and function. *Proc Natl Acad Sci U S A*, 107(44):19090–5, 2010.

[2] L. C. Antunes, R. Levandovski, G. Dantas, W. Caumo, and M. P. Hidalgo. Obesity and shift work: chronobiological aspects. *Nutr Res Rev*, 23(1):155–68, 2010.

[3] S. J. Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[4] K. G. R. K. S. R. T. W. D. R. A. S. L. O. T. E. G. W. D. S. T. L. V. V. Arthur Guez, Mehdi Mirza. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.

[5] P. Baldi. Autoencoders, Unsupervised Learning, and Deep Architectures. *Journal of Machine Learning Research. Proceedings of 2011 ICML Workshop on Unsupervised and Transfer Learning*, 27:37–50, 2012.

[6] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16:412–424, 2000.

[7] P. Baldi and P. Sadowski. The dropout learning algorithm. *Artificial Intelligence*, 210C:78–122, 2014.

[8] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5, 2014.

[9] P. Baldi, P. Sadowski, and D. Whiteson. Enhanced higgs to $\tau^+\tau^-$ searches with deep learning. *Physics Review Letters*, 2015. In press.

[10] M. M. Bellet, E. Deriu, J. Z. Liu, B. Grimaldi, C. Blaschitz, M. Zeller, R. A. Edwards, S. Sahar, S. Dandekar, P. Baldi, et al. Circadian clock regulates the host response to salmonella. *Proceedings of the National Academy of Sciences*, 110(24):9897–9902, 2013.

[11] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[12] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[13] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[14] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[15] S. A. Brown, E. Kowalska, and R. Dallmann. (re)inventing the circadian feedback loop. *Dev Cell*, 22(3):477–87, 2012.

[16] R. Brunetto and O. Trunda. Deep heuristic-learning in the Rubik's cube domain: an experimental evaluation. In *Information Technologies - Applications and Theory*, pages 57–64, 2017.

[17] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90:45–63, 1999.

[18] Y. Cho and L. Saul. Large margin classification in infinite neural networks. *Neural Computation*, 22(10), 2010.

[19] M. F. Covington, J. N. Maloof, M. Straume, S. A. Kay, and S. L. Harmer. Global transcriptome analysis reveals circadian regulation of key pathways in plant growth and development. *Genome Biol*, 9(8):R130, 2008.

[20] G. Cowan, K. Cranmer, E. Gross, and O. Vitells. Asymptotic formulae for likelihood-based tests of new physics. *Eur.Phys.J.*, C71:1554, 2011.

[21] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[22] A. Deckard, R. C. Anafi, J. B. Hogenesch, S. B. Haase, and J. Harer. Design and analysis of large-scale biological rhythm studies: a comparison of algorithms for detecting periodic signals in biological data. *Bioinformatics*, 29(24):3174–3180, 2013.

[23] P. Di Lena, K. Nagata, and P. Baldi. Deep architectures for protein contact map prediction. *Bioinformatics*, 28(19):2449–2457, 2012.

[24] P. Di Lena, K. Nagata, and P. Baldi. Deep architectures for protein contact map prediction. *Bioinformatics*, 28:2449–2457, 2012. doi: 10.1093/bioinformatics/bts475. First published online: July 30, 2012.

[25] C. Dibner, U. Schibler, and U. Albrecht. The mammalian circadian timing system: organization and coordination of central and peripheral clocks. *Annu Rev Physiol*, 72:517–49, 2010.

[26] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[27] D. Duvenaud. *Automatic model construction with Gaussian processes*. PhD thesis, University of Cambridge, 2014.

[28] K. A. Dyar, S. Ciciliot, L. E. Wright, R. S. Biensø, G. M. Tagliazucchi, V. R. Patel, M. Forcato, M. I. Paz, A. Gudiksen, F. Solagna, et al. Muscle insulin sensitivity and glucose metabolism are controlled by the intrinsic muscle clock. *Molecular metabolism*, 3(1):29–41, 2014.

[29] R. Ebendt and R. Drechsler. Weighted A* search–unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[30] K. Eckel-Mahan and P. Sassone-Corsi. Metabolism control by the circadian clock and vice versa. *Nat Struct Mol Biol*, 16(5):462–7, 2009.

[31] K. L. Eckel-Mahan, V. R. Patel, S. de Mateo, R. Orozco-Solis, N. J. Ceglia, S. Sahar, S. A. Dilag-Penilla, K. A. Dyar, P. Baldi, and P. Sassone-Corsi. Reprogramming of the circadian clock by nutritional challenge. *Cell*, 155(7):1464–1478, 2014/04/07 2013.

[32] K. L. Eckel-Mahan, V. R. Patel, R. P. Mohney, K. S. Vignola, P. Baldi, and P. Sassone-Corsi. Coordination of the transcriptome and metabolome by the circadian clock. *Proc Natl Acad Sci U S A*, 109(14):5541–6, 2012.

[33] K. L. Eckel-Mahan, T. Phan, S. Han, H. Wang, G. C. Chan, Z. S. Scheiner, and D. R. Storm. Circadian oscillation of hippocampal mapk activity and camp: implications for memory persistence. *Nature neuroscience*, 11(9):1074–1082, 2008.

[34] R. Edgar, M. Domrachev, and A. E. Lash. Gene expression omnibus: NCBI gene expression and hybridization array data repository. *Nucleic acids research*, 30(1):207–210, Jan. 2002. PMID: 11752295.

[35] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[36] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[37] O. Froy. Metabolism and circadian rhythms–implications for obesity. *Endocr Rev*, 31(1):1–24, 2010.

[38] O. Froy. Circadian rhythms, aging, and life span in mammals. *Physiology (Bethesda)*, 26(4):225–35, 2011.

[39] J. R. Gerstner, L. C. Lyons, J. Wright, K. P., D. H. Loh, O. Rawashdeh, K. L. Eckel-Mahan, and G. W. Roman. Cycling behavior and memory formation. *J Neurosci*, 29(41):12824–30, 2009.

[40] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[41] E. F. Glynn, J. Chen, and A. R. Mushegian. Detecting periodic patterns in unevenly spaced gene expression time series using lomb–scargle periodograms. *Bioinformatics*, 22(3):310–316, 2006.

[42] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[43] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[44] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[45] C. Gulcehre, K. Cho, R. Pascanu, and Y. Bengio. Learned-norm pooling for deep feedforward and recurrent neural networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 530–546. Springer, 2014.

[46] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deepspeech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[47] A. Y. Hannun, C. Case, J. Casper, B. C. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[48] S. L. Harmer, J. B. Hogenesch, M. Straume, H.-S. Chang, B. Han, T. Zhu, X. Wang, J. A. Kreps, and S. A. Kay. Orchestrated transcription of key pathways in arabidopsis by the circadian clock. *Science*, 290(5499):2110–2113, 2000.

[49] S. L. Harmer, S. Panda, and S. A. Kay. Molecular bases of circadian rhythms. *Annual review of cell and developmental biology*, 17(1):215–253, 2001.

[50] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[51] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[52] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[53] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[54] M. E. Hughes, L. DiTacchio, K. R. Hayes, C. Vollmers, S. Pulivarthy, J. E. Baggs, S. Panda, and J. B. Hogenesch. Harmonics of circadian gene transcription in mammals. *PLoS Genet*, 5(4):e1000442, 2009.

[55] M. E. Hughes, J. B. Hogenesch, and K. Kornacker. JTK_CYCLE: an efficient nonparametric algorithm for detecting rhythmic components in genome-scale data sets. *J Biol Rhythms*, 25(5):372–80, 2010.

[56] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[57] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[58] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[59] C. G. Johnson. Solving the Rubik's cube with learned guidance functions. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2082–2089. IEEE, 2018.

[60] B. Karlsson, A. Knutsson, and B. Lindahl. Is there an association between shift work and having a metabolic syndrome? results from a population based study of 27,485 people. *Occup Environ Med*, 58(11):747–52, 2001.

[61] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

[62] A. Knutsson. Health disorders of shift workers. *Occup Med (Lond)*, 53(2):103–8, 2003.

[63] H. Kociemba. 15-puzzle optimal solver. `http://kociemba.org/themen/fifteen/fifteensolver.html`.

[64] A. Kohsaka, A. D. Laposky, K. M. Ramsey, C. Estrada, C. Joshu, Y. Kobayashi, F. W. Turek, and J. Bass. High-fat diet disrupts behavioral and molecular circadian rhythms in mice. *Cell Metab*, 6(5):414–21, 2007.

[65] R. V. Kondratov, A. A. Kondratova, V. Y. Gorbacheva, O. V. Vykhovanets, and M. P. Antoch. Early aging and age-related pathologies in mice deficient in bmal1, the core componentof the circadian clock. *Genes & development*, 20(14):1868–1873, 2006.

[66] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[67] R. E. Korf. Macro-operators: A weak method for learning. *Artificial intelligence*, 26(1):35–77, 1985.

[68] R. E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 700–705. AAAI Press, 1997.

[69] R. E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6):26, 2008.

[70] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134(1-2):9–22, 2002.

[71] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[72] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[73] K. A. Lamia, K. F. Storch, and C. J. Weitz. Physiological significance of a peripheral tissue circadian clock. *Proc Natl Acad Sci U S A*, 105(39):15172–7, 2008.

[74] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

[75] I. Lenz, H. Lee, and A. Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

[76] P. Lichodzijewski and M. Heywood. The Rubik's cube and GP temporal sequence learning: an initial study. In *Genetic Programming Theory and Practice VIII*, pages 35–54. Springer, 2011.

[77] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[78] G. Liu, A. E. Loraine, R. Shigeta, M. Cline, J. Cheng, V. Valmeekam, S. Sun, D. Kulp, and M. A. Siani-Rose. Netaffx: Affymetrix probesets and annotations. *Nucleic acids research*, 31(1):82–86, 2003.

[79] A. Lusci, G. Pollastri, and P. Baldi. Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. *Journal of chemical information and modeling*, 53(7):1563–1575, 2013.

[80] A. Lusci, G. Pollastri, and P. Baldi. Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. *Journal of chemical information and modeling*, 53(7):1563–1575, 2013.

[81] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[82] S. Masri, V. R. Patel, K. L. Eckel-Mahan, S. Peleg, I. Forne, A. G. Ladurner, P. Baldi, A. Imhof, and P. Sassone-Corsi. Circadian acetylome reveals regulation of mitochondrial metabolic pathways. *Proceedings of the National Academy of Sciences*, 110(9):3339–3344, 2013.

[83] S. Masri, P. Rigor, M. Cervantes, N. Ceglia, C. Sebastian, C. Xiao, M. Roqueta-Rivera, C. Deng, T. Osborne, R. Mostoslavsky, P. Baldi, and P. Sassone-Corsi. SIRT6 defines circadian transcription leading to control of lipid metabolism. *Cell*, 2014. in press.

[84] S. Masri, P. Rigor, M. Cervantes, N. Ceglia, C. Sebastian, C. Xiao, M. Roqueta-Rivera, C. Deng, T. F. Osborne, R. Mostoslavsky, et al. Partitioning circadian transcription by sirt6 leads to segregated control of cellular metabolism. *Cell*, 158(3):659–672, 2014.

[85] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi. Solving the Rubik's cube with approximate policy iteration. *International Conference on Learning Representations (ICLR)*, 2019.

[86] B. H. Miller, E. L. McDearmon, S. Panda, K. R. Hayes, J. Zhang, J. L. Andrews, M. P. Antoch, J. R. Walker, K. A. Esser, J. B. Hogenesch, and J. S. Takahashi. Circadian and clock-controlled regulation of the mouse transcriptome and cell proliferation. *Proceedings of the National Academy of Sciences*, 104(9):3342–3347, 2007.

[87] A. Monnier, S. Liverani, R. Bouvet, B. Jesson, J. Q. Smith, J. Mosser, F. Corellou, and F.-Y. Bouget. Orchestrated transcription of biological processes in the marine picoeukaryote ostreococcus exposed to light/dark cycles. *BMC genomics*, 11(1):192, 2010.

[88] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, Oct 1993.

[89] R. Y. Moore and V. B. Eichler. Loss of a circadian adrenal corticosterone rhythm following suprachiasmatic lesions in the rat. *Brain Res*, 42(1):201–6, 1972.

[90] A. Newell and H. A. Simon. GPS, a program that simulates human thought. Technical report, RAND CORP SANTA MONICA CALIF, 1961.

[91] L. Orseau, L. Lelis, T. Lattimore, and T. Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

[92] S. Panda, M. P. Antoch, B. H. Miller, A. I. Su, A. B. Schook, M. Straume, P. G. Schultz, S. A. Kay, J. S. Takahashi, and J. B. Hogenesch. Coordinated transcription of key pathways in the mouse by the circadian clock. *Cell*, 109(3):307–20, 2002.

[93] C. L. Partch, C. B. Green, and J. S. Takahashi. Molecular architecture of the mammalian circadian clock. *Trends in cell biology*, 24(2):90–99, 2014.

[94] V. Patel, K. E. Maha, P. Sassone-Corsi, and P. Baldi. Circadiomics: Integrating circadian genomics, transcriptomics, proteomics, and metabolomics. *Nature Methods*, 2012. In press.

[95] V. R. Patel, N. Ceglia, M. Zeller, K. Eckel-Mahan, P. Sassone-Corsi, and P. Baldi. The pervasiveness and plasticity of circadian oscillations: the coupled circadian-oscillators framework. *Bioinformatics*, 31(19):3181–3188, 2015.

[96] A. Pizarro, K. Hayer, N. F. Lahens, and J. B. Hogenesch. Circadb: a database of mammalian circadian gene expression profiles. *Nucleic acids research*, page gks1161, 2012.

[97] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4):193–204, 1970.

[98] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[99] D. Quang, Y. Chen, and X. Xie. Dann: a deep learning approach for annotating the pathogenicity of genetic variants. *Bioinformatics*, 31(5):761–763, 2015.

[100] M. R. Ralph, R. G. Foster, F. C. Davis, and M. Menaker. Transplanted suprachiasmatic nucleus determines circadian period. *Science*, 247(4945):975–8, 1990.

[101] C. E. Rasmussen. *Gaussian processes for machine learning*. MIT Press, 2006.

[102] T. Rokicki. God's number is 26 in the quarter-turn metric. `http://www.cube20.org/qtm/`, Aug 2014.

[103] T. Rokicki. cube20. `https://github.com/rokicki/cube20src`, 2016.

[104] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. The diameter of the Rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

[105] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

[106] M. Samadi, A. Felner, and J. Schaeffer. Learning from multiple heuristics. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, 2008.

[107] J. Scherphuis. The mathematics of Lights Out. `https://www.jaapsch.net/puzzles/lomath.htm`, 2015.

[108] U. Schibler and P. Sassone-Corsi. A web of circadian pacemakers. *Cell*, 111(7):919–22, 2002.

[109] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[110] A. Sharifian, S. Farahani, P. Pasalar, M. Gharavi, and O. Aminian. Shift work as an oxidative stressor. *J Circadian Rhythms*, 3:15, 2005.

[111] S.-q. Shi, T. S. Ansari, O. P. McGuinness, D. H. Wasserman, and C. H. Johnson. Circadian disruption leads to insulin resistance and obesity. *Current Biology*, 23(5):372–381, 2013.

[112] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[113] R. J. Smith, S. Kelly, and M. I. Heywood. Discovering Rubik's cube subgroups using coevolutionary GP: A five twist experiment. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 789–796. ACM, 2016.

[114] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[115] J. T. Springenberg and M. Riedmiller. Improving deep neural networks with probabilistic maxout units. *arXiv preprint arXiv:1312.6116*, 2013.

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

[117] M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber. Deep networks with internal selective attention through feedback connections. In *Advances in Neural Information Processing Systems*, pages 3545–3553, 2014.

[118] K. F. Storch, O. Lipan, I. Leykin, N. Viswanathan, F. C. Davis, W. H. Wong, and C. J. Weitz. Extensive and divergent circadian gene expression in liver and heart. *Nature*, 417(6884):78–83, 2002.

[119] M. Stratmann and U. Schibler. Properties, entrainment, and physiological functions of mammalian peripheral oscillators. *J Biol Rhythms*, 21(6):494–506, 2006.

[120] M. Straume. Dna microarray time series analysis: automated statistical assessment of circadian rhythms in gene expression patterning. *Methods in enzymology*, 383:149, 2004.

[121] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1139–1147, 2013.

[122] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[123] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[124] J. S. Takahashi, H. K. Hong, C. H. Ko, and E. L. McDearmon. The genetics of mammalian circadian order and disorder: implications for physiology and disease. *Nat Rev Genet*, 9(10):764–75, 2008.

[125] P. Tognini, M. Murakami, K. Eckel-Mahan, J. Newman, E. Verdin, Y. Liu, P. Baldi, and P. Sassone-Corsi. *in preparation*, 2014.

[126] F. W. Turek, C. Joshu, A. Kohsaka, E. Lin, G. Ivanova, E. McDearmon, A. Laposky, S. Losee-Olson, A. Easton, D. R. Jensen, R. H. Eckel, J. S. Takahashi, and J. Bass. Obesity and metabolic syndrome in circadian clock mutant mice. *Science*, 308(5724):1043–5, 2005.

[127] A. J. Turner and J. F. Miller. Neuroevolution: Evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, pages 1–20, 2014.

[128] V. Vijayan, R. Zuzow, and E. K. O'Shea. Oscillations in supercoiling drive circadian gene expression in cyanobacteria. *Proceedings of the National Academy of Sciences*, 106(52):22564–22568, 2009.

[129] Q. Wang and J. JaJa. From maxout to channel-out: Encoding information on sparse pathways. *arXiv preprint arXiv:1312.1909*, 2013.

[130] G. Wu, R. C. Anafi, M. E. Hughes, K. Kornacker, and J. B. Hogenesch. Metacycle: an integrated r package to evaluate periodicity in large scale data. *bioRxiv*, page 040345, 2016.

[131] J. Yan, H. Wang, Y. Liu, and C. Shao. Analysis of gene regulatory networks in the mammalian circadian rhythm. *PLoS Comput Biol*, 4(10):e1000193, 2008.

[132] R. Yang and Z. Su. Analyzing circadian expression data by harmonic regression based on autoregressive spectral estimation. *Bioinformatics*, 26(12):i168–i174, 2010.

[133] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[134] S. H. Yoo, S. Yamazaki, P. L. Lowrey, K. Shimomura, C. H. Ko, E. D. Buhr, S. M. Siepka, H. K. Hong, W. J. Oh, O. J. Yoo, M. Menaker, and J. S. Takahashi. Period2::luciferase real-time reporting of circadian dynamics reveals persistent circadian oscillations in mouse peripheral tissues. *Proc Natl Acad Sci U S A*, 101(15):5339–46, 2004.

[135] R. Zhang, N. F. Lahens, H. I. Ballance, M. E. Hughes, and J. B. Hogenesch. A circadian gene expression atlas in mammals: Implications for biology and medicine. *Proceedings of the National Academy of Sciences*, 111(45):16219–16224, 2014.