On the benefits of sparsity in value function approximators for Reinforcement Learning

by

Fatima Davelouis Gallardo

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Fatima Davelouis Gallardo, 2024

Abstract

In machine learning, sparse neural networks provide higher computational efficiency and in some cases, can perform just as well as fully-connected networks. In the online and incremental reinforcement learning (RL) problem, Prediction Adapted Networks (Martin and Modayil, 2021) is an algorithm that can adapt the sparse connectivity of a shallow value network with random hidden-layer weights. Martin and Modayil evaluated Prediction Adapted Networks (PANs) in the RL prediction setting and showed promising results, suggesting that one can use multiple online predictions of input signals to discover high-performing NN sparse topologies with no a priori inductive biases. However, there remain some open questions that one can ask about this algorithm. For instance, do the statistical benefits of PANs carry over to reinforcement learning control in multiple environments? Do PANs provide performance gains when we learn the sparse value network's weights end-to-end in both the prediction and control settings? How does predictive sparsity compare against sparse network structures learned end-to-end? The contributions of this work are two fold. First, we investigate the above questions and provide answers. Second, we devise a methodology that encodes sparse value network structures as binary masks and systematically evaluate their performance. In one RL control environment, we find that predictive sparsity performs on par with both a fully-connected architecture and a sparse network induced by L1 regularization. However, in another domain PANs does not generate a sparse structure that can outperform even random sparsity. Surprisingly, in the same RL pre-

diction environment that was used in the PANs original work, we found that learning the hidden-layer weights does not lead to better performance, suggesting there may be unidentified properties of environments for which PANs is best suited.

To my grandfather

For being my biggest cheerleader.

I still can't believe you're gone.

When you realize there is something you don't understand, then you're generally on the right path to understanding all kinds of things.

– Jostein Gaarder, The Solitaire Mystery, 1990.

Acknowledgements

I would like to thank:

- my thesis advisors, Michael Bowling and John Martin for their continued guidance and support throughout this project.
- Joseph Modayil for helping me specify the questions that I needed to investigate at various instances and the experiments that aim to answer them.
- Professor Rich Sutton for helping me clarify the scope of my research and how to communicate it more effectively.
- the Digital Research Alliance of Canada for the compute resources they provided throughout my graduate studies.
- other graduate students advised by Professor Michael Bowling. I learned a lot from our group meetings and enjoyed our outings.
- the broader community of graduate students with whom I shared a working space on campus. They made my graduate experience all the more fun and memorable.

Contents

1 Introduction **1**

2 Background Material **4**
2.1 Reinforcement Learning 4
2.2 Value Functions, Returns and Policies 5
2.3 RL Prediction vs. Control Problems 6
2.4 Function Approximation in the RL Context 8
2.5 Temporal Difference Learning 9
2.5.1 Eligibility Traces and TD(λ) 10
2.5.2 Forward view of TD(λ) 12
2.6 Off-policy vs. On-policy Learning 12
2.6.1 $QV(\lambda)$: an On-Policy Control Algorithm 13

3 General Value Functions & Prediction Adapted Networks **15**
3.1 General Value Functions 15
3.2 Prediction Adapted Networks 17
3.2.1 Conclusions drawn by the original PANs work and Remaining Questions 19

4 Experiment Methodology for Control **20**
4.1 RL Control Environments 20
4.2 Learning Algorithm 22
4.3 Sparse Network Architectures 22
4.3.1 Baselines 24
4.4 Evaluation Metrics 26

5 Fixed Hidden Layer Weights and Fixed Sparsity **30**
5.1 Hypotheses 30
5.2 Empirical Results in *Breakout* 31
5.3 Empirical Results in *Space-Invaders* 32

6 Learned Hidden Layer Weights and Fixed Sparsity **33**
6.1 Hypotheses 33
6.2 Empirical Results in *Breakout* 33
6.3 Empirical Results in *Space-Invaders* 34

7 Learned Hidden Layer Weights and Learned Sparsity **36**
7.1 Hypotheses 39
7.2 Empirical Results in *Breakout* 40
7.3 Empirical Results in *Space-Invaders* 41

8 Experiment Methodology in Prediction **44**
8.1 Prediction Environment 44
8.2 Learning Algorithm 45
8.3 Value Network Architecture 45
8.4 Baselines 46
8.5 Evaluation Metrics 46

9 Learned Hidden Layer Weights and Fixed Sparsity in Prediction **51**
9.1 Hypotheses 51
9.2 Empirical Results in *Frog's Eye* 52

10 Learning Sparsity in Prediction **54**

11 Conclusion **57**

References **59**

List of Figures

4.1 Visualization of the *Breakout* environment. 21
4.2 Visualization of the *Space-Invaders* environment. 21
4.3 Example of a predictive neighborhood found by the Prediction Adapted Neighborhoods algorithm in *Breakout*. 26
4.4 Example of a spatial neighborhood in *Breakout*. 27
4.5 Example of a random neighborhood in *Breakout*. 27
4.6 Example of a hidden layer binary mask generated by the Prediction Adapted Neighborhoods algorithm in *Space-Invaders*. . 28
4.7 Example of a spatially-biased mask in *Space-Invaders*. 29
4.8 Example of a hidden layer binary mask with random sparsity in *Space-Invaders*. 29

5.1 Performance of sparse architectures with fixed hidden-layer weights in the *Breakout* environment. 31
5.2 Performance of sparse architectures with fixed hidden-layer weights in *Space-Invaders* 32

6.1 Performance when value network's hidden layer is learned in *Breakout*. 34
6.2 Performance when the hidden layer of the value network is learned in *Space-Invaders*. 35

7.1 Average hidden layer weight magnitude with respect to regularization coefficient in *Breakout*. 37
7.2 Percentage of hidden layer weight magnitudes that fall below average in *Breakout*. 38
7.3 Average hidden layer weight magnitude with respect to regularization coefficient in *Space-Invaders*. 38
7.4 Percentage of hidden layer weight magnitudes that fall below average in *Space-Invaders*. 39
7.5 Performance of an L1-sparse value network when the hidden layer is fixed in *Breakout*. 40
7.6 Performance of an L1-sparse value network when the hidden layer is learned in *Breakout*. 41
7.7 Performance of an L1-sparse value network when the hidden layer is fixed in *Space-Invaders*. 42
7.8 Performance of an L1-sparse value network when the hidden layer is learned in *Space-Invaders*. 42

8.1 Example of a predictive neighborhood found by Prediction Adapted Neighborhoods in the *Frog's Eye* environment. 48
8.2 Example of a spatial neighborhoods in the *Frog's Eye* environment. 49

8.3 Example of a random neighborhoods in the *Frog's Eye* environment. 50

9.1 Performance of sparse architectures with fixed hidden-layer weights in the *Frog's Eye* environment; hidden-layer bias units initialized to -4.0. 53

10.1 Average hidden layer weight magnitude with respect to L1-regularization coefficient in *Frog's Eye*. 55
10.2 Percentage of hidden layer weight magnitudes that fall below average in *Frog's Eye*. 56

Chapter 1

Introduction

Reinforcement learning (RL) is a computational approach to learning through interaction to maximize the long term accumulation of a numerical reward signal (Sutton & Barto, 2018). In contrast to other machine learning approaches that rely on static data-sets of inputs and outputs, in RL an agent makes decisions in an unknown environment and learns through sequential trial and error. At each time step, the agent takes an action, which causes the environment to transition from its current state to a new state, and the agent receives a scalar reward.

The process of extracting and forming useful features of raw input data is crucial to reinforcement learning. One reason why this is true is because inputs such as images or sensor readings can be complex, high-dimensional and of unknown structure. Extracting relevant features from observations can make it easier for the RL agent to apply its knowledge from past experiences to new, unseen situations; this is called generalization. For example, consider the case of a self-driving car that has only ever driven in Edmonton. If the car was shipped to Montreal, it ideally should still be able to drive reasonably well without having to learn everything all over again. Although the streets in Montreal are different to those in Edmonton, the basic features that an ideal RL system extracts from its video feed should still be useful to driving: the edges of the road, other vehicles, pedestrians, color of the traffic lights, etc.

A common approach in RL is to use neural network architectures (NNs) as function approximation tools that can construct useful features of the observa-

tions. NNs that connect all of the layer's inputs to the layer's output features have higher representational capacity, but they can be computationally expensive. For this reason, networks with fewer connections are a reasonable alternative (Hoefler et al., 2021). Such sparse NNs may also provide statistical benefits in addition to computational savings, especially in scenarios where the RL agent receives noisy or irrelevant observations. In such settings, a small number of select connections can filter signal from noise (Ahmad & Scheinkman, 2019; Grooten et al., 2023).

Recently, Martin and Modayil (2021) presented *Prediction Adapted Networks* (PANs) — an approach that adapts NN sparsity automatically through sequential interaction. The results suggest that in an environment with thousands of noisy inputs, an RL system can use predictions of the observations to form a sparse NN that approximates the cumulative reward with low average error. Moreover, PANs generated sparsity from scratch without prior knowledge of the input structure. Nonetheless, the resulting NN topology resembled one based on spatial locality. This suggests that it is possible for PANs to discover some useful structure of the input data.

Despite the initial success of Prediction Adapted Networks, there remain some open questions that one can ask about this line of work. Notably, Martin and Modayil's experiments were applied to the RL prediction problem, where the agent's learning goal is to approximate the expected sum of future rewards conditioned on a fixed behaviour. Would the performance gains they found carry over to the RL control problem, where the agent learns its behaviour through experience? Additionally, the PANs algorithm as originally presented holds the hidden layer weights fixed at initial random values. How would performance differ if these weights were learned end-to-end? The **contributions of this thesis are the answers to the following questions:**

1. *Do the statistical benefits of PANs carry over to RL control in multiple environments?*

2. *Do PANs provide performance gains when the hidden layer weights are learned end-to-end in both prediction and control?*

3. *How do sparse structures imposed in PANs compare against a sparsity that is learned end-to-end?*

We believe that these research avenues will help us broaden our current understanding of PANs and its potential limitations.

This thesis is structured as follows: Chapter 2 provides the necessary background on the reinforcement learning problem setting: the concepts, terminology and formalism. Chapter 3 provides a more narrow background material on general value functions (GVFs) and Prediction Adapted Networks (PANs). Chapter 4 describes the experiment methodology in the RL control task — the environment, learning algorithm and evaluation metrics. Chapters 5, 6 and 7 show results that answer each of the above three questions in RL control, respectively. Chapter 8 describes experiments that pertain to the RL prediction problem, within the same environment used in the original PANs work. Chapter 9 and 10 provide answers to 2 and 3, respectively, in the prediction task. Finally, Chapter 11 summarizes the results, contributions and provides future directions.

Chapter 2

Background Material

2.1 Reinforcement Learning

In the Reinforcement Learning (RL) problem setting, an agent interacts with the environment continually over discrete time steps. The agent-environment interaction is formally described as a Markov Decision Process (MDP). An MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the set of all environment states, $\mathcal{A}$ the set of actions that the agent can take, $\mathcal{T}$ the transition function from one time-step to the next, $\mathcal{R}$ the set of rewards that the agent receives and $\gamma \in [0, 1)$ the discount factor; as the name implies, γ down-weights rewards obtained further in the future. At each discrete time step t, the environment (which includes the agent) is in some state $S_t \in \mathcal{S}$. The agent takes action $A_t \in \mathcal{A}$ which causes the environment to transition to state $S_{t+1} \in \mathcal{S}$. In this new state, the agent receives a reward $R_{t+1} \in \mathcal{R}$.

This leads to a sequence of states, actions and rewards, known as a trajectory of experience:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

We say that the environment state is *Markovian* if the likelihood of each value of S_t and R_t only depends on the previous state S_{t-1} and action A_{t-1}. That is, the current state contains enough information to determine the probability of the next state. Mathematically, the Markov property of the state can be summarized as:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}$$

We say that the environment state has this property if and only if it is *fully observable.* Of course, this property does not hold for all environments.

2.2 Value Functions, Returns and Policies

The agent's goal is to accrue the most reward over long periods of time. In the RL problem setting that we consider, rewards received closer to the current time step are given a higher weight compared to those that will come many time-steps later. Thus, formally the agent seeks to maximize the sum of *discounted* future rewards. This is called the *return* at time t, and is denoted as G_t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1}$$

The discount factor $0 \leq \gamma < 1$ gives more weighting to rewards closer in time, making G_t a weighted average of future rewards. Typically, γ is considered part of the problem setting, because $1 - \gamma$ can be viewed as the probability of a trajectory of experience terminating at any given time-step — making it part of the environment dynamics, rather than of the agent. Moreover, γ also specifies the *effective horizon* of a trajectory of experience — meaning the expected number of time steps that will follow t.

Both the environment dynamics and the agent's behavior might contain some amount of randomness affecting the return. The *expectation* of the return starting at state s_t is an estimate for how good that state is to the agent. The *value function* computes this expectation as a mapping from states in $\mathcal{S}$ to scalars $\mathbb{R}$.

On the other hand, a *policy* is a function that determines the actions that the agent takes at any given time step, and is denoted as π, where $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. That is, π outputs the probability that an action $A_t \in \mathcal{A}$ is taken at state S_t. In the case of a deterministic policy, π will output either zero or one for each state-action pair. We also point out that the sum of the policy over all actions must sum to one:

$$\sum_{a \in \mathcal{A}} \pi(s \,|\, a) = 1 \qquad (2.1)$$

Formally, we denote the value function as v_π, since it outputs the expected return that the agent would obtain *under* π. Putting it all together, we define $v_\pi : \mathcal{S} \to \mathbb{R}$:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

We can also define a value function of state-action pairs that estimates how beneficial an action is at any given state. This is typically called the action-value function, denoted by $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

An optimal policy is one that yields a higher expected return compared to all other policies. For a given problem setting, there may be more than one optimal policy. Formally, π^* is optimal if

$$\pi^* = \arg\max_\pi v_\pi(s) \quad \text{for all } s \in \mathcal{S}. \tag{2.2}$$

We can also define an optimal policy in terms of action-value functions:

$$\pi^* = \arg\max_\pi q_\pi(s, a) \quad \text{for all } s \in \mathcal{S} \text{ and all } a \in \mathcal{A}. \tag{2.3}$$

Equation 2.3 states that π^* maximizes the expected return if the agent takes action a in state s and thereafter follows π^*.

2.3 RL Prediction vs. Control Problems

The RL problem setting can be broken down into two sub-problems: the *prediction problem* and the *control problem*. These sub-problems rely on policy evaluation and improvement, respectively.

In the prediction problem — also known as policy evaluation — the agent's goal is to make accurate estimates of the expected return given a policy that selects actions, or given a stream of observed states and rewards. In other words, the objective is to accurately learn the *true* value function from a single stream of experience.

On the other hand, in the control problem the goal is to improve the agent's policy to maximize the return. Reaching the optimal policy often involves two

steps: a policy improvement step, and a policy evaluation step. In other words, the agent must predict the benefits of taking an action at the current state before improving its behaviour strategy. Since the policy changes with each new estimate of the return, the distribution of next states and rewards also changes. Thus, the policy evaluation in RL control problem is fundamentally non-stationary — the data distribution changes and is therefore not i.i.d.

There are two types of RL control methods: *value-based* vs. *policy-gradient* control algorithms. The former case is typically used when the set of actions $\mathcal{A}$ is discrete and relatively small. As the name implies, value-based control algorithms derive the policy directly from an estimate of the state-action values. For instance, this is the case of the Q-learning (Watkins, 1989) and Sarsa (Sutton and Barto, 2018) algorithms, which will be described in section (2.6). On the other hand, policy-gradient techniques represent a policy by parameterizing it with respect to some trainable weights, without resorting to state-action values. In the RL control experiments described in this thesis, we only focus on value-based policies.

One of the main challenges in RL is to balance exploration vs. exploitation; that is, figuring out to what extent the agent should favor states and actions that yield large reward, versus exploring new behaviours or unknown regions of the state space. When it comes to value-based policies, a common strategy used to address this dilemma is to let the policy select a random action some small ϵ fraction of the time. Such policies are called ϵ-greedy policies, where $\epsilon \in [0, 1]$ is the probability of selecting a random action and $1 - \epsilon$ is the probability of selecting an action that maximizes the agent's current value estimate. At each time step, an ϵ-greedy policy operates as follows:

$$A_t \leftarrow \begin{cases} \arg\max_{a \in \mathcal{A}} q_\pi(S_t = s, A_t = a) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases} \tag{2.4}$$

We use an ϵ-greedy policy in the RL control experiments that are presented in chapters 5, 6 and 7.

2.4 Function Approximation in the RL Context

In principle, the state space of the environment is much larger than the agent's computational capacity, and therefore the agent must learn to *generalize* across similar states The agent cannot simply represent every single state in a giant table, as it would run out of memory and it would be infeasible for continuous state spaces.

This means that we might not be able to exactly compute the true value function nor the true optimal policy over the state space, but we can still approximate these. How? We find a lower dimensional representation for these functions. In other words, we map the value function to some space of parameters $\mathbf{w}$ that is smaller than the total number of states in the environment. In the simplest case, suppose that we want to linearly approximate the value function. Then, we would write:

$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \phi(s), \tag{2.5}$$

which means that $\hat{v}(s, \mathbf{w})$ is linear in the parameters $\mathbf{w}$. Here, $\phi(s)$ is called the *state-representation* or the *agent-state*, and it maps the environment states to feature vectors. i.e., $\phi(s) : \mathcal{S} \rightarrow \mathbb{R}^d$, where $d \ll |\mathcal{S}|$. Note that this mapping may not be one-to-one: many states may correspond to the same feature vector.

Moreover, the function ϕ can be a fixed representation — as in the case of tile-coding (Sutton and Barto, 2018) — or a learned representation such as an artificial neural network (NN) trained through back-propagation. Typically, an agent-state ϕ that can generalize across $\mathcal{S}$ would extract the salient features shared among many input states. For example, if the states are pixel images, then some of the underlying features could be comprised of angles, shapes and lines that make up the objects in the images.

Also, we should note that the feature vectors can be constructed from noisy or

incomplete observations of the state, such as a robot's sensor readings. Hence, the *agent-state* can be thought of as the computational mechanism that allows a learning agent to process raw sensory signals of the environment state.

In this thesis, we investigate the effects of different choices of NN architectures that form representations of the agent's observations.

2.5 Temporal Difference Learning

One way to estimate the value of a state directly from experience is to interact with the environment over a fixed number of time steps, and then compute and average the returns from each state. Monte-Carlo techniques precisely adopt this approach (Sutton and Barto, 2018). One of their main drawbacks is that we need to wait until the last time step to compute our estimates. This means that experience needs to be divided into finite sequences called *episodes*. In this sense, Monte-Carlo methods cannot be used to approximate a value function in a purely online and incremental step-by-step way.

An alternative, temporal difference (TD) learning, is a commonly used technique for computing the values from each state without having to wait until the end of an episode. TD learning methods are amenable to the continuing RL problem setting, where in principle, the agent-environment interaction continues forever without interruption nor hard terminations. The general TD update rule is given as:

$$\begin{aligned} \delta_t &= U_t - V(S_t) \\ V(S_t) &\leftarrow V(S_t) + \alpha\delta_t \end{aligned} \tag{2.6}$$

where δ_t is called the TD-error, α is the step-size and U_t denotes the target of our approximation — the new estimate of the value at state S_t. This leads to the question: what exactly is U_t? In Monte-Carlo methods, the target is the entire return $U_t = G_t$. On the other hand, TD methods typically define their target $U_t = R_{t+1} + \gamma V(S_{t+1})$, i.e., they use the current estimate of the next state's value to estimate the return. In other words, they can make updates to the value estimates immediately on each transition, and are therefore fully

online and incremental. Their update rule becomes:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ V(S_t) &\leftarrow V(S_t) + \alpha \delta_t \end{aligned} \tag{2.7}$$

Alternatively, TD algorithms can wait on any number n of next state transitions to update the value estimates, and are hence called n-step TD methods, where $U_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$. We can think of n as a dial between one-step TD and Monte-Carlo algorithms.

All TD methods are characterized by *bootstrapping* — the process of using estimates of next state transitions $V(S_{t+n})$ where $n \in 1, 2, \ldots$ to update current ones. In simple terms, bootstrapping is making a guess from a guess. From now on, we will refer to "one-step TD updates" as "TD updates" for the sake of simplicity.

In the linear function approximation setting, where $\hat{v}(S_t, \mathbf{w}) = \mathbf{w}^T \phi(S_t)$ the TD update is written as:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \mathbf{w}_t^T \phi(S_{t+1}) - \mathbf{w}_t^T \phi(S_t) \\ \mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t + \alpha \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha \delta_t \phi(S_t) \end{aligned} \tag{2.8}$$

where $\mathbf{w}_t$ denotes the weight vector that parameterizes the approximate value function $\hat{v}$, α is the step-size and the state representation $\phi(S_t)$ is equal to the gradient of the value at the current state $\hat{v}(S_t, \mathbf{w}_t)$.

2.5.1 Eligibility Traces and TD(λ)

What if we want a learning target equal to the average of n-step returns, yet we want to make sample-to-sample updates in an online fashion? This is precisely where the TD(λ) algorithm comes in. The TD(λ) update rule involves the computation of *eligibility traces*. When our value function is represented as a table, the eligibility trace is analogous to a form of memory of previous states. More specifically, it quantifies the degree to which each past state is eligible to undergo a learning change. The value estimate at the current state is given the most credit for the current TD error. If a state was visited a long time

ago, then the degree to which it is assigned credit for the current TD error is smaller. The TD(λ) update rules for the tabular setting are given by:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ e_t(s) &= \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \in \{S_0, S_1, \ldots, S_{t-1}\} \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = S_t \end{cases} \\ V(s) &\leftarrow V(s) + \alpha\delta_t e_t(s) \quad \forall s \in \{S_0, S_1, \ldots, S_{t-1}\} \cup \{S_t\} \end{aligned} \tag{2.9}$$

where $\lambda \in [0, 1]$ is called the trace parameter. When $\lambda = 0$, the algorithm becomes the same as one-step TD, also known as TD(0). In the case of $\lambda = 1$, the algorithms becomes equivalent to Monte Carlo methods. In this way, λ can play a similar role to the trajectory length in n-step methods, except that it makes online updates.

In the linear function approximation setting, the eligibility trace is a vector that quantifies the eligibility of each component of the weight vector to be affected by the current TD error:

$$\begin{aligned} \delta_t = R_{t+1} + \gamma \mathbf{w}_t^T \phi(S_{t+1}) - \mathbf{w}_t^T \phi(S_t) \\ \mathbf{z}_t = \gamma\lambda \mathbf{z}_{t-1} + \phi(S_t) \\ \mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t \end{aligned} \tag{2.10}$$

Here, $\mathbf{z}$ denotes the eligibility trace vector, and it is of the same dimension as the weight vector $\mathbf{w}$. If a feature in the vector $\phi(S_t)$ is large, then the weight value that corresponds to it is more heavily affected by the current TD-error. On the same token, weights that correspond to features that are small get updated less. Notice that if $\phi(S_t)$ is a one-hot feature vector with the same dimensionality as the state space, then the update equations (2.10) reduce to those in (2.9). These update rules formalize the backward view of eligibility traces. RL practitioners call it the backward view, because the TD-error at the current state is propagated "backwards" in trajectory of states visited in the past. Values at states visited recently get updated more, and those at states visited a long time ago get updated only a tiny bit. Online TD(λ) allows the agent to more rapidly update the value function at previous states, without waiting to revisit them.

2.5.2 Forward view of TD(λ)

Unfortunately, it is not always convenient to implement the backwards view of eligibility traces, especially when the value function is parameterized by a large NN. This is because the dimensionality of the eligibility trace vectors must be the same as the number of parameters in the NN. To approximate online TD(λ), we can resort to the forward view of eligibility traces. As opposed to the backwards view, the forward view computes many n-step returns starting at the current state and averages them. Each future return is weighted by the parameter λ. Hence, n-step returns of short future trajectories are weighted higher than those of longer trajectories. There are many versions of the finalized average of λ-discounted n-step returns computed in the forward view. We focus on the truncated λ return:

$$G_{t:h}^{\lambda} = (1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{h-t-1}G_{t:h} \tag{2.11}$$

Overall, both the backward and forward views have relatively the same performance (Sutton and Barto, 2018), because they approximate the same learning target.

2.6 Off-policy vs. On-policy Learning

We might want to predict the expected return under some target policy that does not correspond to the agent's behavior in the environment. In the case where the value function's target policy π is identical to the behaviour policy b, we say that the learning task is *on-policy*, and otherwise it is *off-policy*. In off-policy control, the actions that the agent takes have a different distribution from the actions used to update the value function (Sutton and Barto, 2018). One of the most popular off-policy control algorithms is Q-learning (Watkins, 1989), whose update rule is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t)] \tag{2.12}$$

where the current action A_t is sampled from the behaviour policy $A_t \sim b(a|s = S_t)$, while the target policy π is the deterministic policy that maximizes the

bootstrapped target. In this sense, Q-learning allows us to learn about an *optimal policy*, while adopting a more exploratory policy. On the other hand, if the action at the next time step A_{t+1} was sampled from $b(a|s)$, then this would give rise to the Sarsa on-policy update rule (Rummery and Niranjan, 1994):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.13}$$

2.6.1 $QV(\lambda)$: an On-Policy Control Algorithm

As mentioned in section 2.6, there are two types of RL control methods: value-based vs. policy-gradient methods. QV(λ) is a member of the family of value function based control methods that include SARSA, Q-learning and Actor-Critic (Sutton and Barto, 2018). QV(λ) keeps track of two value functions: a state-value function $\hat{v}$ (known as the V-function) that is estimated via TD(λ) and an action value function $\hat{q}$ (known as a Q-function, (Wiering, 2005)). The state-value function estimates are used to update the action values at each step, similar to Q-learning and SARSA. Unlike Q-learning however, QV(λ) remains an on-policy RL algorithm. Furthermore, QV(λ) guarantees that if we have a good enough approximation of the state value function, then the action value estimates will also be accurate.

First, the state values are estimated with eligibility traces in one-step TD(λ) as given in Equation (2.9). The estimates $\hat{v}(S_t)$ then become the bootstrapped targets for the action values:

$$\begin{aligned} \delta_t^q &= R_{t+1} + \gamma\hat{v}(S_{t+1}) - \hat{q}(S_t, A_t) \\ \hat{q}(S_t, A_t) &\leftarrow \hat{q}(S_t, A_t) + \alpha\delta_t^q \end{aligned} \tag{2.14}$$

In the linear function approximation case, QV(λ) updates two weight vectors, one for the state-value function and the other for the action values. The V-function's weight vector $\mathbf{w}_v$ is updated following the linear-TD(λ) update equations (2.10).

Then, the Q-function's weights $\mathbf{w}_a$ are updated as follows:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \mathbf{w}_{v,t}^T \phi(S_{t+1}) - \mathbf{w}_{a,t}^T \phi(S_t, A_t) \\ \mathbf{w}_{a,t+1} &= \mathbf{w}_{a,t} + \alpha \delta_t \phi(S_t, A_t) \end{aligned} \tag{2.15}$$

Note, in the case where the action set $\mathcal{A}$ is small and countable, we can define a vector $\mathbf{w}_a \in \mathbb{R}^d$ for each action.

Chapter 3

General Value Functions & Prediction Adapted Networks

3.1 General Value Functions

Being able to compute multiple predictions in addition to the main value function can be useful to an RL agent — that is, it can help the agent approximate the main value function or improve its policy. Hence, we call these additional predictions *auxiliary predictions*. Since these extra predictions are not limited to the main reward signal, they can capture a broader range of patterns in the data and be more expressive. For example, if a system is learning to play tetris, it could be useful to predict how often 2×2 squares appear in the screen to stack the blocks appropriately and win the game. In this case, at each time step, the signal would be 1 when a 2×2 square appeared and 0 otherwise. Or, it could be useful to estimate how fast the blocks are moving down the screen — i.e., to predict how many time steps it takes for the blocks to move from the top to the bottom of the frame. Moreover, it could be useful for an autonomous driving robot to predict how long a yellow light lasts, or how long it takes to reach a location given constant speed, or how long it will take a pedestrian to cross the road. In RL, auxiliary predictions are formally described as *General Value Functions* (GVFs) (Sutton & Barto, 2018).

As the name implies, GVFs are an extended version of the ordinary value functions described so far. Whereas the value function outputs the expected sum of discounted rewards, a GVF computes the expected future value of any

signal — these could be sensory observations or any function of any signal in the environment. These forecasts are often described in terms of a *question* and an *answer* (Sutton et al., 2011). The question asks: "what is the expectation of some signal of interest over a number of future time-steps, given some behavior?" The question has three principal components: the signal (often called the *cumulant* and is denoted by c), the expected temporal duration of the forecast H, and a target policy π. Notice that the signal we care about depends on the environment state, and thus it is a function of the state: $c(S_t) : \mathcal{S} \to \mathbb{R}$. The duration of the forecast is given by a *continuation function* $\gamma : \mathcal{S} \to [0, 1]$, a generalization of the discount factor introduced for ordinary value functions. The continuation function specifies the probability of an experience trajectory ending at any given state, and is an interpretation that we can also give to the discount factor in conventional RL. At each state s_t, the probability of the trajectory continuing onto the next time step is $\gamma(s_t)$, and the probability of termination is $1 - \gamma(s_t)$. Thus, γ indicates the expected time-span of the trajectory following each state, called the *effective horizon*, denoted by H. Finally, π is the target policy under which we want to predict signal c. Similarly to ordinary value functions, a GVF can be either on-policy or off-policy. In this thesis, we will only consider GVFs that are on-policy. Putting all these three components together, a GVF is defined as:

$$\begin{aligned} \bar{v}_{\pi,\gamma,c}(s) \doteq \bar{v}(s; \pi, \gamma, c) = \mathbb{E}[\bar{G}_t \mid S_t = s, A_t \sim \pi], \text{ where} \\ \bar{G}_t = c(S_{t+1}) + \gamma(S_{t+2})c(S_{t+2}) + \gamma^2(S_{t+3})c(S_{t+3}) + \dots \\ = \sum_{k=0}^{\infty} \gamma^k(S_{t+k+1})\, c(S_{t+k+1}) \end{aligned} \tag{3.1}$$

The "answer" to a GVF question is the mechanism used to approximate the GVF given a trajectory of experience. This includes the type of function approximation (tabular, linear, non-linear, etc), the learning algorithm itself, the learned parameters and how we gathered the data.

When the GVF's cumulant is a raw sensory signal, we call it a *nexting* prediction, because the RL agent's task is to predict the input signal that will appear in the next time steps (Modayil et al., 2011). If the agent's observations are vectors, then the GVF's cumulant might be the ith entry of the vector:

$c(\mathbf{o}_t) = \mathbf{o}_t[i]$. In the case of nexting, the GVF formulation is simplified: the temporal profile γ is typically a constant specified by the designer a priori, thus working in the same way as an ordinary discount factor. In this case, the GVF's nexting return can be written as:

$$\bar{G}_t = \mathbf{o}_{t+1}[i] + \gamma \mathbf{o}_{t+2}[i] + \gamma^2 \mathbf{o}_{t+3}[i] + \ldots = \sum_{k=0}^{\infty} \gamma^k \, \mathbf{o}_{t+k+1}[i] \tag{3.2}$$

3.2 Prediction Adapted Networks

Now that we laid out the necessary reinforcement learning concepts and terminology, we can describe how *Prediction Adapted Networks* (Martin and Modayil, 2021) operate in more detail.

In the original work, Prediction Adapted Networks (PANs) are only applied to the RL prediction problem with partial observability and where no policy is learned. In partially observable problem settings, the environment state is not fully known to the agent. The observations are encoded as vectors $\mathbf{o}_t \in \mathbb{R}^d$ and the sampled rewards r_t are scalars. The value function is approximated as a linear function $\hat{v}(\mathbf{o}_t) = \mathbf{w}^T \phi(\mathbf{o}_t)$, where the state representation — or agent-state — denoted by ϕ is a sparse neural network with a single hidden layer of fixed random weights. The output layer weights $\mathbf{w}$ are learned online through TD(λ).

PANs uses nexting-style predictions to adapt the sparse connectivity of the state representation network fully online. At each time step, the agent receives $\mathbf{o}_t, r_{t+1}$ and constructs m general value functions, whose cumulants are the components of the observation vector $\mathbf{o}_t$: $c(\mathbf{o}_t) = \mathbf{o}_t[i]$. All auxiliary predictions share the same constant discount factor $\gamma \in [0, 1]$ and the same policy as the main value function. We denote the ith GVF as $\bar{v}^i_{\pi,\gamma,c}$:

$$\bar{v}^i_{\pi,\gamma,c}(\mathbf{o}_t) = \mathbb{E}_\pi[\bar{G}^i_t \,|\, O_t = \mathbf{o}_t],$$

where $\bar{G}^i_t$ is given by Equation (3.2). Each GVF is approximated by a linear function of the observation with weights $\bar{\mathbf{w}}^i$, also learned through TD(λ):

$$\bar{v}^i_\pi(\mathbf{o}_t) \approx \hat{\bar{v}}^i(\mathbf{o}_t) = \bar{\mathbf{w}}^{iT} \mathbf{o}_t \tag{3.3}$$

The GVF estimates are used to dictate the sparse hidden layer connections of the network in the following way. First, inputs that are multiplied by the top-k elements of $\bar{\mathbf{w}}^i$ are selected as part of a special subset called a *neighborhood*. Thus, for m GVFs we have a total of m neighborhoods, each containing k input components: $\mathcal{S}_i = \{\mathbf{o}_{t,(1)}, \ldots, \mathbf{o}_{t,(k)}\}$. Each neighborhood is then fed to a linear function of fixed random parameters, followed by a non-linear activation to yield a feature vector. In the end, we obtain m vectors of non-linear features that are concatenated as a single hidden-layer feature vector, denoted as $\phi(\mathbf{o}_t)$. Since the GVF weights are being learned online and incrementally, the hidden layer topology is adapted constantly until said weights converge — hence the name *prediction-adapted networks*. Martin and Modayil suggest that at convergence, inputs that belong to a neighborhood are *predictively related*.

Nonlinear features from the i-th neighborhood are computed as a composition of three functions, $\mathbf{y}_t^i \equiv \mathbf{f}(\mathbf{A}\mathbf{M}^i\mathbf{o}_t + \mathbf{b})$. First is a neighborhood selection matrix $\mathbf{M}^i \in \{0,1\}^{k \times d}$, then a linear projection $\mathbf{A} \in \mathbb{R}^{n \times k}$ and bias units $\mathbf{a} \in \mathbb{R}^n$ both shared between all the neighborhoods, and finally a non-linearity $\mathbf{f}\colon \mathbb{R}^n \to \mathbb{R}^n$. The neighborhood selection matrix $\mathbf{M}^i$ is an orthogonal rank-k matrix with one-hot columns—used to mask out an ordered selection of k elements of the observation. When the matrix $\mathbf{M}^i$ is applied to an input vector $\mathbf{o}_t$, this process is equivalent to selecting k components of $\mathbf{o}_t$. The sparse structure of $\mathbf{M}^i$ is determined by the top-k weights of $\bar{\mathbf{w}}^i$. We denote the transformation of the ith neighborhood under $\mathbf{A}$ and $\mathbf{b}$ as the pre-activation vector $\mathbf{z}^i \in \mathbb{R}^n$: $\mathbf{z}^i = \mathbf{A}\mathbf{M}^i\mathbf{o}_t + \mathbf{b}$. Finally, the function $\mathbf{f}$ applies a fixed non-linearity $f\colon \mathbb{R} \to \mathbb{R}$ to each element of its n-dimensional input: $\mathbf{f}(\mathbf{z}) = (f(\mathbf{z}_1), ..., f(\mathbf{z}_n))$. The full feature vector, $\mathbf{x}_t$, contains nonlinear features from m neighborhoods, $\mathbf{M}^i\mathbf{o}_t$, and the current observation, $\mathbf{x}_t \equiv \text{concatenate}(\mathbf{o}_t, \mathbf{y}_t^1, \ldots, \mathbf{y}_t^m)$. The m neighborhoods encode the architecture's graph topology.

3.2.1 Conclusions drawn by the original PANs work and Remaining Questions

This simple but powerful algorithm achieved similar performance gains compared to sparse neural network architectures built with spatial inductive biases in a stochastic domain with thousands of inputs. Moreover, the authors showed that the predictive neighborhoods contained inputs that were spatially related. Thus, one can conclude that in the chosen domain, the algorithm successfully discovered spatial relationships in the data without *any* a priori knowledge of the environment dynamics. The authors also demonstrated the computational efficiency of PANs: since each neighborhood is computed in parallel, the total amount of computation required is linear in the number of output features!

In the rest of this thesis, we investigate the following research questions:

1. *Do the performance gains found in the RL prediction setting carry over to RL control in multiple environments?*

2. *What would happen if we learn the neural network weight magnitudes end-to-end?*

3. *What about comparing PANs to other kinds of sparse networks, such as those whose connectivity is learned end-to-end?*

Recent work by Modayil and Abbas takes ideas from PANs and applies them to the control setting (Modayil & Abbas, 2023). Our approach and specific research questions however, remain different from the ones they explored.

In the following chapters, we will lay out the methods and experiments that address the above questions.

Chapter 4

Experiment Methodology for Control

In this section, we describe the experiments that investigate our three research questions in RL control.

4.1 RL Control Environments

Our research questions are explored in two environments selected from the MinAtar suite of five, namely *Breakout* and *Space-Invaders* (Young and Tian, 2019). In *Breakout*, each frame has four input channels corresponding to each of the four objects in the environment: (1) the paddle which moves left or right at the bottom of the frame, (2) the ball which bounces off the paddle, (3) the trail which follows the ball's trajectory one time step in the past and (4) the brick wall. Each time a brick is destroyed, the agent gains a +1 reward; otherwise, the reward is zero. The goal of the game is to move the paddle so that the ball bounces off and knocks down as many bricks as possible. Figure 4.1 provides a visualization of the *Breakout* game.

On the other hand, *Space-Invaders* contains six input channels. Four of these correspond to different objects: (1) the cannon representing the player, (2) the aliens representing the enemy, (3) the "friendly bullet" shot by the cannon and (4) enemy bullets shot by the aliens. The extra two channels indicate whether the alien is moving left or right. The objective of the game is for the player to fire bullets at the aliens while trying to avoid getting hit by

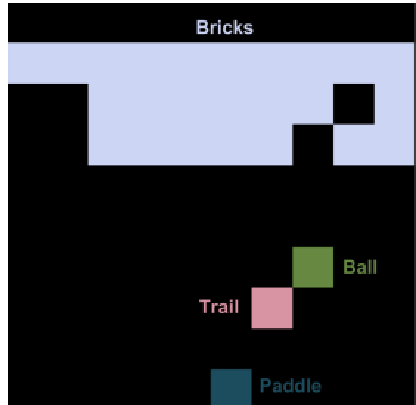

Figure 4.1: Visualization of the *Breakout* environment showing the four objects: the paddle, ball, trail and brick wall (Young & Tian, 2019).

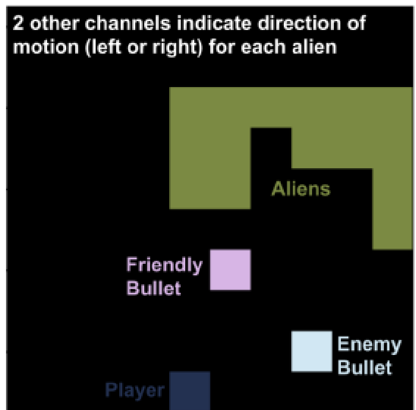

Figure 4.2: Visualization of the *Space-Invaders* environment showing four objects: the cannon, aliens, friendly bullet and enemy bullet (Young & Tian, 2019).

enemy bullets. The player gets a reward of +1 each time an alien is shot, and that alien is also removed. We note that *Space-Invaders* has non-stationary dynamics: the aliens move with an increased speed when few of them are left, or after a wave of them is fully cleared and a new one appears. Figure 4.2 provides a visualization of the *Space-Invaders* game. We refer the reader to the original MinAtar paper for more details about the inherent difficulty of each game.

We chose these two games as they contain an underlying spatial structure; that is, the interactions between objects in the game tend to be spatially local. For instance, in *Breakout* the trail is always found one spatial unit diagonally

behind the ball. Moreover, when the ball hits a brick on the wall, all adjacent bricks will likely be taken down as well. In *Space-Invaders*, the aliens remain stuck to one another and move left or right in a united front. Also, enemy bullets move downwards from the aliens, while friendly bullets move upwards from the cannon's location.

4.2 Learning Algorithm

The algorithm we used to learn a policy is DQN (Mnih et al., 2015), also employed in the original MinAtar paper. We followed the same hyper-parameter settings as done by the MinAtar authors, with the exception of the step-size, which we swept over with a grid search. As opposed to the MinAtar paper however, our Q-network has a single hidden layer instead of two. This allowed us to remain closer to the original PANs work. Furthermore, this layer is not convolutional; rather, we resort to a simple sparse linear layer, followed by a ReLU activation function. The raw inputs are flattened into 1D vectors and then fed to the the network. Finally, we trained our Q-network for 5 million steps using the Adam optimizer.

4.3 Sparse Network Architectures

Recall that PANs adapts NN connectivity online and incrementally. In order to test the benefits of PANs in RL control, we do not need to dynamically adapt the Q-network connectivity while simultaneously learning the policy. Rather, we can apply a static sparse structure to the Q-network hidden layer and then directly test the benefits of each type of sparsity in generating a good policy. Our methodology therefore required two learning phases. In the first phase, we ran PANs until the GVF weights converged, generating predictive masks. In the second phase, we impose the predictive masks onto the hidden layer of the DQN architecture and start learning from scratch. These two phases were done on each environment separately.

Phase One: In this phase, the learning algorithm that updates the value function is QV(0) (i.e., $\lambda = 0.0$) following an ϵ-greedy policy with $\epsilon = 0.1$.

The V-function and Q-function are represented as linear functions with respect to the feature vector $\phi(\mathbf{o}_t)$ generated by PANs online: $v(\mathbf{o}_t; \mathbf{w}_v) = \mathbf{w}_v^T \phi(\mathbf{o}_t)$ and $q(\mathbf{o}_t, A_t) = \mathbf{w}_a^T \phi(\mathbf{o}_t, A_t)$. We ran QV(0) for 5 million steps and extracted the predictive masks at the final time step. This phase was run over a single random seed used to initialize the agent and environment. More specifically, this amounts to initializing the GVF weights and hidden layer weights that generate the feature vector ϕ. The same seed was used to sample random actions from the ϵ-greedy policy and the environment's initial state. Note, the goal of the first phase is to solely generate the masks and is not concerned with finding a near-optimal policy.

Phase Two: In the second phase, we ran DQN with sparse Q-networks whose hidden-layer topology is specified by the masks found in the first phase. All m neighborhood mask matrices were flattened and stacked into a larger mask matrix $\mathbf{M}$, which is imposed onto the Q-network's hidden layer matrix by element-wise multiplication. In our implementation, each column of the weight matrix generates a single scalar feature. Thus, the ith neighborhood mask corresponds to the ith column. We denote the hidden-layer weight matrix as $\mathbf{A} \in \mathbb{R}^{n \times d}$, where d is the number of inputs in the flattened observation vector, and n the number of hidden layer features. The binary mask matrix is denoted as $\mathbf{M} \in \mathbb{R}^{n \times d}$ and the resulting masked hidden-layer matrix becomes $\mathbf{A} \odot \mathbf{M}$. Each column of $\mathbf{M}$ encodes the binary mask vector for a specific neighborhood. In the case where we have m neighborhoods and we want to generate a single scalar feature per neighborhood, then $n = m$. In *Breakout*, we generated 4 features per neighborhood, thus $n = 4m$. As in the original PANs work, the number of neighborhoods is equivalent to the total number of inputs, thus $m = d = 400$ in this domain and our hidden layer weight matrix has 1600 output features. On the other hand, in *Space-Invaders* we generated 3 features per neighborhood, hence $n = 3m$. Since $m = d = 600$ in this environment, we have 600 neighborhoods generating 1800 hidden layer features. The number of hidden layer features were selected in a way that would maintain the overall number of NN parameters more of less consistent across these two environments.

4.3.1 Baselines

In chapters 5 and 6, we compare predictive sparse structures to two other sparse hidden layer structures: random neighborhoods and spatial neighborhoods. For the random sparse baseline, we sample one set of random neighborhoods from a uniform distribution, as a stand-in for any set of random hidden-layer connections. In the spatially sparse architecture, the neighborhoods are repeated across input channels, similar to kernels in a convolutional layer, as shown in Figures 4.4 and 4.7 in *Breakout* and *Space Invaders* respectively. One of the main differences between our predictive masks and a convolutional layer is that the former are not based on spatial locality and they do not slide across the height and width of each input channel. For the sake of completion, we also compare against a dense architecture, a.k.a., a fully connected neural network (FNN). Figures 4.3, 4.4 and 4.5 show examples of hidden layer masks that we generated for each type of network sparsity in *Breakout*. Similarly, Figures 4.6, 4.7 and 4.8 show hidden layer masks in *Space-Invaders*.

In chapter 7, we compare the performance of predictive sparsity against a sparse structure learned end-to-end through L1 regularization. Similar to how we generated predictive sparsity, to run this experiment we follow two phases: one to generate binary masks via L1 regularization and the second to evaluate the performance of the static sparse structure in the RL control task.

In machine learning, L1 regularization is a commonly used technique in which we add a term to the loss function and weight it by a regularization coefficient $\beta \in [0, 1)$:

$$\mathcal{L}(\mathbf{X}, \mathbf{y}; \theta) = l(f_\theta(\mathbf{X}), \mathbf{y}) + \beta \sum_{j=1}^{d} |\theta_j| \tag{4.1}$$

where $\mathbf{X}$ is the data matrix containing inputs, while $\mathbf{y}$ is a vector of targets. Here, θ are the weights that parameterize the learned function f (Hastie et al., 2009). When the coefficient $\beta = 0$, the loss function becomes the standard training loss. As β approaches 1, the weights are pushed towards zero; in practice, they might not actually reach zero, but they should come reasonably

close. The approach we take is to zero out the final weights whose magnitudes fall below average at the end of training; we describe this method in more detail below.

In the first phase, to generate the binary mask we apply L1 regularization to a DQN agent whose fully-connected architecture is learned end-to-end for 5 million steps. Then, at the end of learning we compute two averages over five random seeds: (1) the average hidden layer weight magnitudes stored as a matrix $\mathbf{A}_{ave}$ of size $n \times d$, where d is the number of inputs and n is the number of hidden layer features, and (2) the average hidden layer weight magnitude, which is a scalar: $w_{ave} = average(\mathbf{A}_{ave})$. Each random seed was used to initialize the DQN weights, sample actions from the ϵ-greedy policy and sample the environment's initial state. Once we have $\mathbf{A}_{ave}$ and w_{ave}, we find the elements of $\mathbf{A}_{ave}$ that are larger than w_{ave} and replace them with a "1". The rest are replaced with zeros. The resulting binary mask is given by:

$$\mathbf{M}[i,j] \leftarrow \begin{cases} 1 & \text{if } \mathbf{A}_{ave}[i,j] < w_{ave} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The amount of sparsity is measured as the percentage of zeroes in $\mathbf{M}$. We sweep over the regularization coefficient until we approximately find the desired amount of sparsity. Once we find the L1-induced masks with the desired amount of sparsity, we re-initialize the agent and environment to start phase two, where we impose the masks onto DQN's hidden layer and start the experiment.

One might ask, *Why should the process in phase one lead to sparsity? Why aren't half the weights below the average regardless of the regularization coefficient?* First, notice that the L1-regularized loss in Equation (4.1) tries to reduce both the training error and the sum of all weight magnitudes. Thus, if a subset of weights plays very little contribution to reducing the training error, then they will be more readily drawn towards zero compared to the distinctly more useful ones. Therefore, as β grows from 0 to 1, we expect the distribution of weight magnitudes to become skewed: a greater number of the less-useful weights should fall below average. On the other hand, if all weights are equally useful to minimize the training error, then we expect this approach

to fail at generating sparsity, because around half of them would fall below average regardless of the regularization coefficient.

4.4 Evaluation Metrics

We followed the same evaluation metrics as done by Young and Tian in the MinAtar paper. Namely, we measured the average return incurred by each DQN architecture over 30 independent trials. More specifically, a *trial* refers to a random seed used to (1) initialize the DQN weights at the beginning of learning, (2) sample random actions from an ϵ-greedy policy and (3) sample the environment's initial state. In *Breakout* this amounts to resetting the position of the ball at the start of the game whenever the brick wall is destroyed. On the other hand, in *Space Invaders*, the environment is fully deterministic and thus does not depend on a random seed.

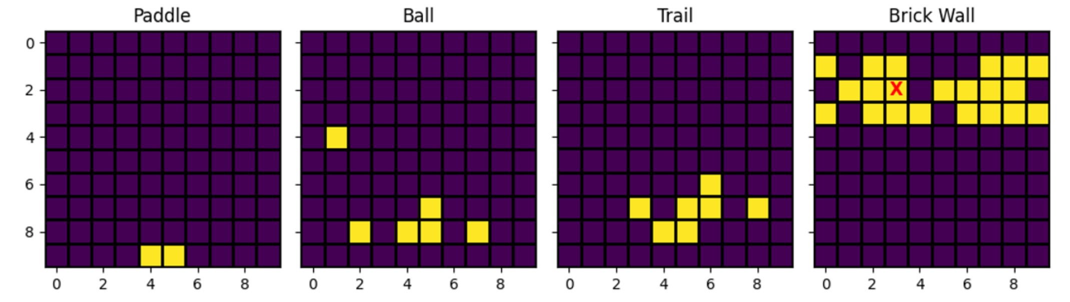

Figure 4.3: One of the predictive neighborhoods in *Breakout*. The red "X" shows the location of the cumulant that the respective GVF predicts. The inputs in yellow are those that are "on" in the mask; they correspond to the top-k weights of the GVF.

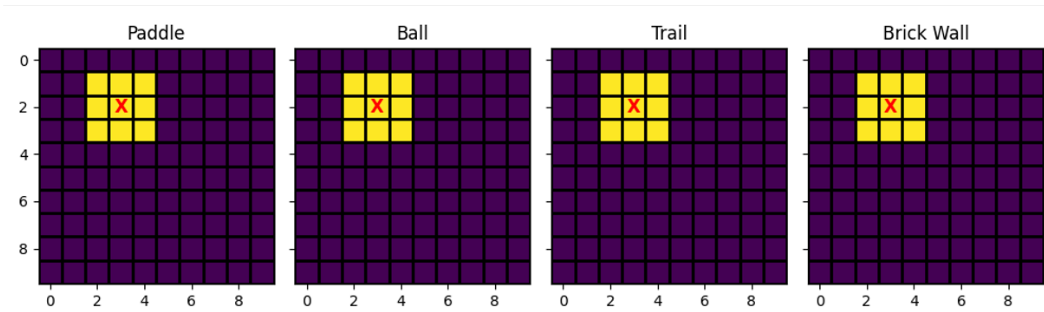

Figure 4.4: One of the spatial neighborhoods in *Breakout*. The inputs that are "on" are located closer to the red "X" in Euclidean distance, across input channels.

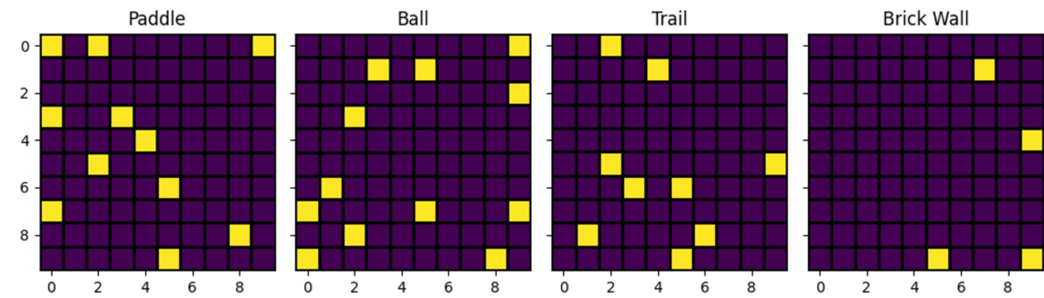

Figure 4.5: One of the random neighborhoods in *Breakout*.

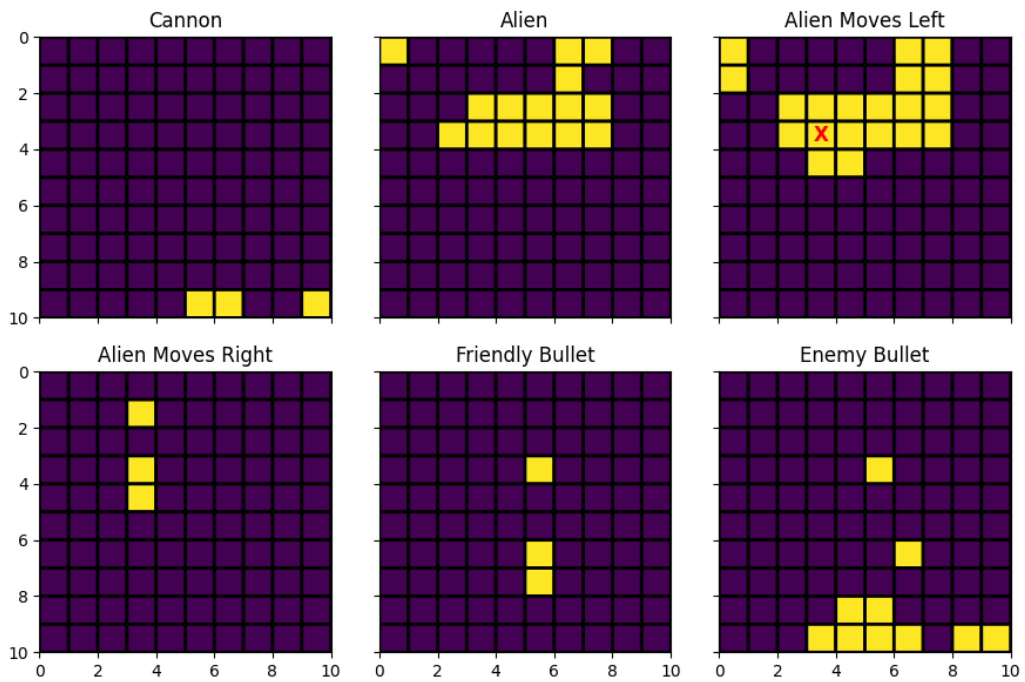

Figure 4.6: One of the predictive masks in *Space-Invaders*. The red “X” shows the location of the cumulant that the respective GVF predicts. The inputs in yellow are those that are “on” in the mask; they correspond to the top-k weights of the GVF.

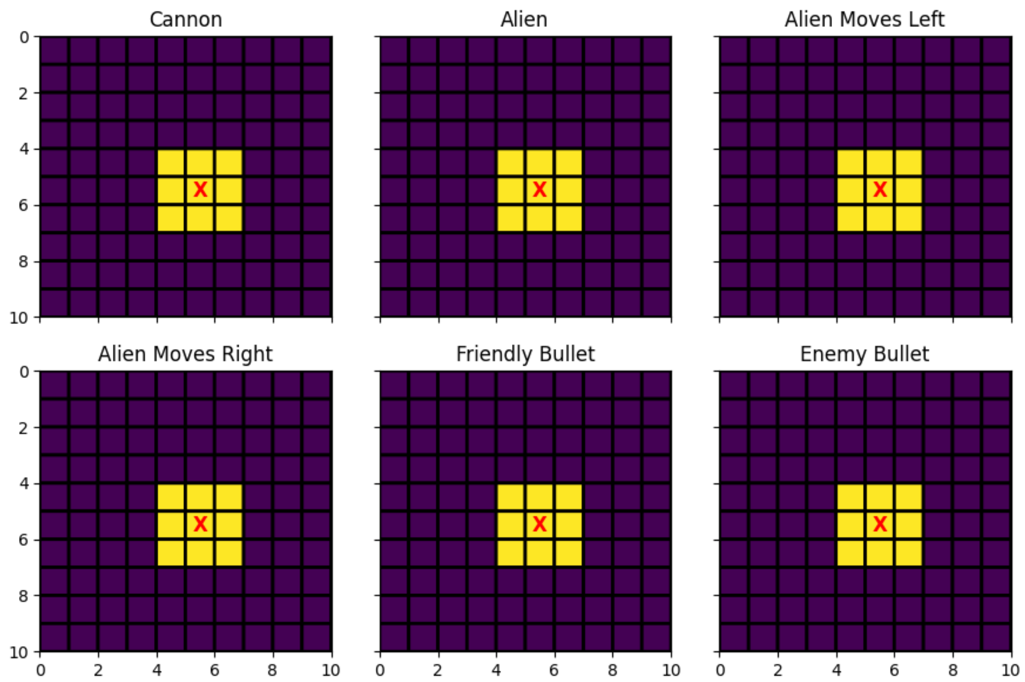

Figure 4.7: One of the spatially-biased masks in *Space-Invaders*. The inputs that are "on" are located closer to the red "X" in Euclidean distance, across input channels.

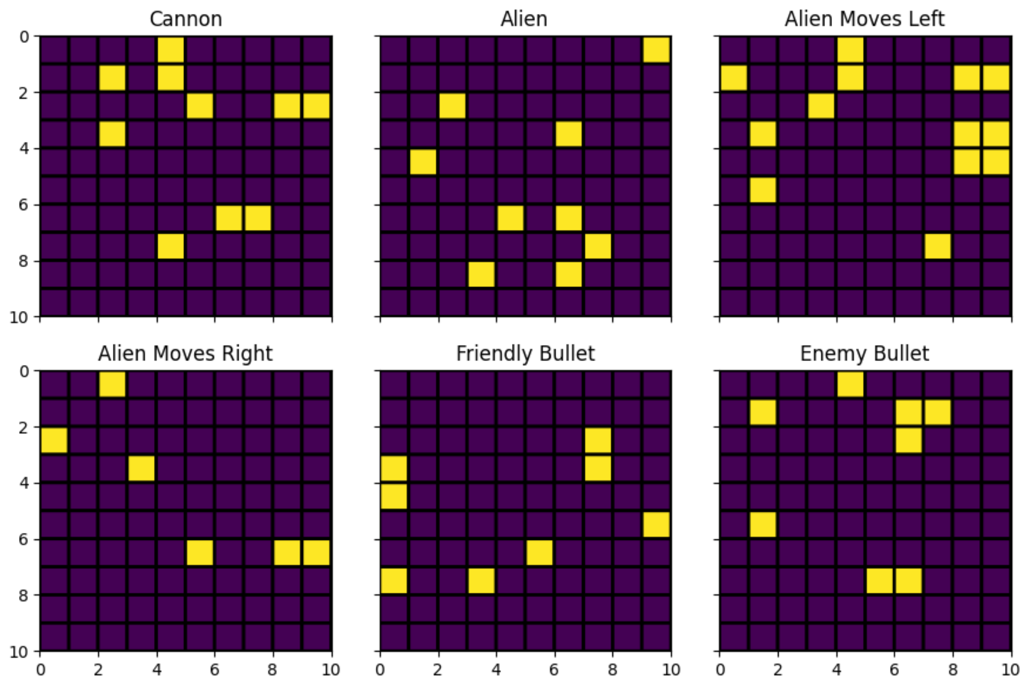

Figure 4.8: One of the random binary masks in *Space-Invaders*.

Chapter 5

Fixed Hidden Layer Weights and Fixed Sparsity

In this chapter, we investigate the first research question: *Do the statistical benefits of PANs carry over to RL control in multiple environments?* To remain consistent with the original PANs work, we randomly initialize the DQN hidden layer weights and hold them fixed.

5.1 Hypotheses

We hypothesize that predictive sparsity will perform better than random sparsity in both environments.We believe this to be the case, because predictive neighborhoods capture information about the temporal structure of the observations: which subset of the inputs are important to predict future (discounted) values of other inputs. Therefore, such temporal relationships could be useful for temporal learning problems, such as predicting the sum of future rewards, or making online estimates of an action-value function that can be used to induce a policy, as in DQN.

Our second hypothesis is that out of the three sparse architectures, the spatially-biased one will yield highest performance, since it is similar to the sparse structure imposed by convolutional layers, which are widely used in domains like MinAtar. Furthermore, as discussed in the previous chapter, the MinAtar games appear to have a distinct spatial structure.

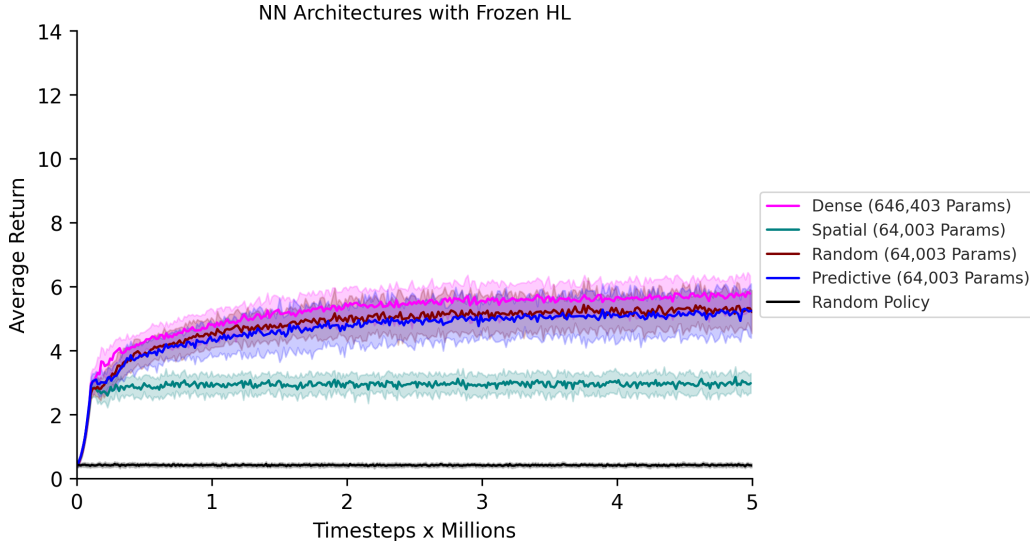

Figure 5.1: Average return incurred on *Breakout* for DQN architectures whose hidden layer is randomly initialized and frozen. The total number of parameters in the Q-networks are listed in the legend.

5.2 Empirical Results in *Breakout*

Figure 5.1 shows the average return incurred by each DQN architecture over 30 trials. Our results suggest that the dense architecture achieves the highest performance. We believe this is the case due to its larger representational capacity. We also note that DQN architectures that achieved the second highest performance have a predictive or random hidden layer sparsity. Although, these do not perform significantly different from the dense NN. Surprisingly, a spatially-biased sparse hidden layer yields the lowest performance – it does not yield anywhere near the performance of predictive nor randomly sparse Q-networks.

These results refute both hypotheses: (1) predictive neighborhoods do not beat random ones, and (2) spatial neighborhoods under-perform all other architectures. The fact that the predictive sparse network performs on par with the randomly-connected network suggests an answer to the motivating question of this chapter: a predictive sparse NN with fixed hidden layer weights does not provide statistical benefits in RL control in the *Breakout* environment; it is just as useful to run DQN with randomly sparse connections.

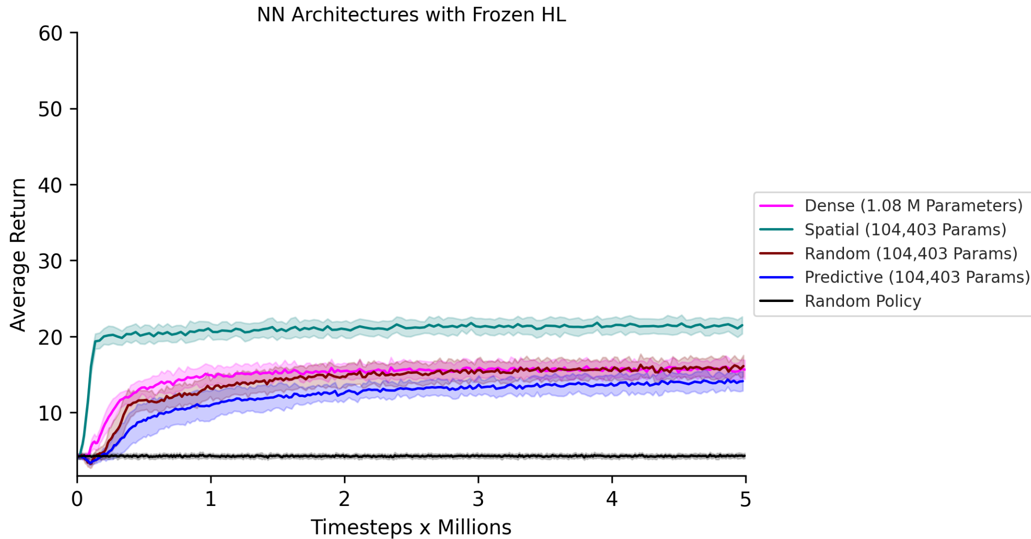

Figure 5.2: Average Return incurred on *Space-Invaders*, for DQN architectures whose hidden layer is randomly initialized and frozen. The total number of parameters in the Q-networks are listed in the legend.

5.3 Empirical Results in *Space-Invaders*

Figure 5.2 shows the average return incurred by each DQN architecture over 30 independent trials on *Space-Invaders*. As shown in the figure, the ordering of the learning curves is almost the reverse compared to *Breakout* (not counting the random policy). Our first hypothesis claim is refuted: on average, predictive sparsity under-performs random sparse connections, although without statistical significance. Moreover, in this domain random sparsity is just as useful as a dense architecture. Our second hypothesis is valid: spatially-biased hidden-layer connections yield the highest average returns — even higher than the dense NN which has 10 times the capacity. In sum, Figure 5.2 suggests that the benefits of PANs do not carry over to the RL control setting in *Space-Invaders*, at least not when the hidden layer weights are fixed to initial random values.

In the following chapter, we take the experiments one step forward to investigate the benefits of predictive sparsity when the systems are allowed to learn their hidden layer weights end-to-end.

Chapter 6

Learned Hidden Layer Weights and Fixed Sparsity

In this chapter, we describe the experiments and results that investigate the second research question: *Do PANs provide performance gains when the hidden layer weights are learned end-to-end for control?* We randomly initialize the DQN weights in the same way as in the previous chapter and allow the entire NN to learn the weight magnitudes end-to-end via backpropagation.

6.1 Hypotheses

Although seemingly trivial, our hypothesis is that the average return of all architectures will be at least as high as when the hidden layer weights are fixed. Moreover, in *Breakout* we hypothesize that spatially-distributed hidden-layer connections are perhaps only useful when the weights are learned end-to-end.

6.2 Empirical Results in *Breakout*

Figure 6.1 shows the average returns for all sparse network architectures, as well as the dense NN on *Breakout*. While spatial sparsity remains the least useful in this domain, we see a change from the learning curves in the previous chapter: now the average return of the predictive sparse network is statistically higher than the randomly sparse network. Moreover, although the average performance of the predictive sparse NN is lower than the dense NN, these are not statistically distinguishable. Hence, there is no advantage

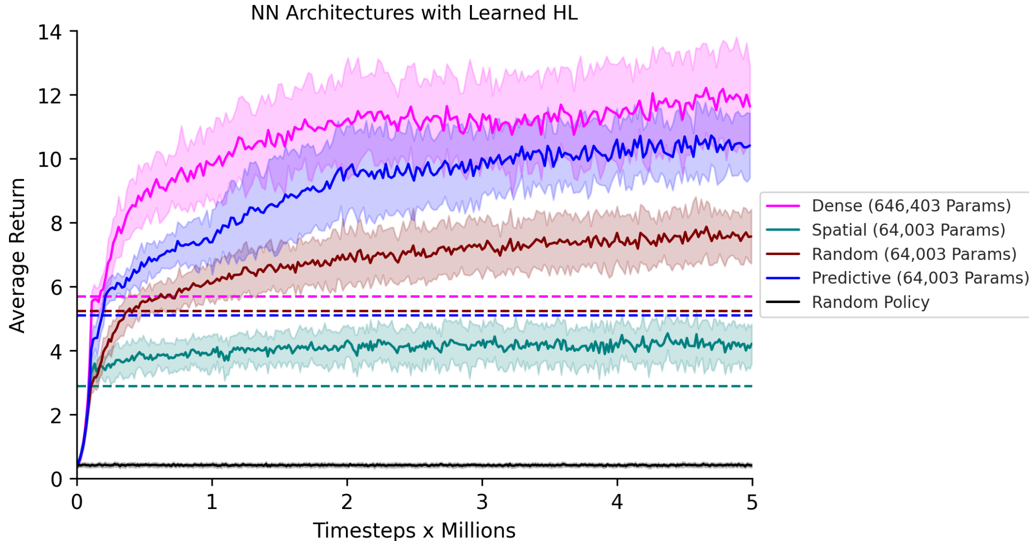

Figure 6.1: Average Return incurred on *Breakout*, for DQN architectures whose hidden layer is learned end-to-end. Horizontal dashed lines indicate the final performances when the hidden-layer weights are never learned, as shown on chapter 6. The total number of NN parameters are listed in the legend.

to using a dense architecture in this domain, since predictive sparsity performs reasonably close to it. Overall, these observations suggest that PANs indeed provide performance gains when the hidden layer weights are learned end-to-end in RL control, as tested in *Breakout*. Further, this result gives evidence that the utility of a sparse architecture is not just associated to its connections, but also to the combination of connectivity and learned weight magnitudes.

6.3 Empirical Results in *Space-Invaders*

Figure 6.2 shows the average return for all network architectures trained on *Space-Invaders*. We observe that spatial sparsity incurs the lowest average return. Similar to the results we found in Chapter 5, predictive sparsity underperforms both random sparsity and the dense NN, now with greater statistical significance. This suggests that in *Space-Invaders*, PANs does not provide performance gains when the hidden layer weights are either fixed or learned end-to-end. We believe that the non-stationary dynamics of this environment make it so that PANs struggles against its competitors.

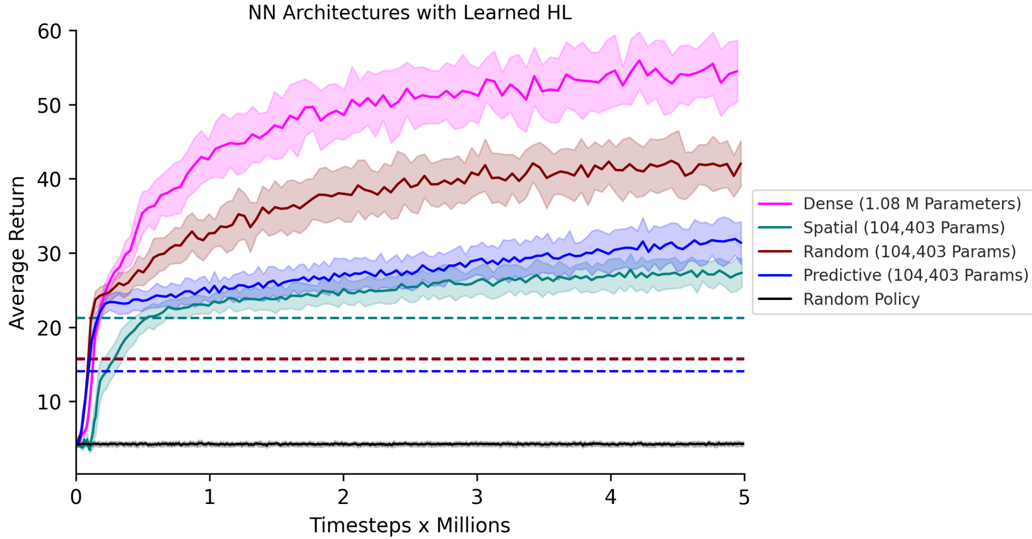

Figure 6.2: Average Return incurred on *Space-Invaders*, for DQN architectures whose hidden layer is learned end-to-end. Horizontal dashed lines indicate the final performances when the hidden-layer weights are never learned, as shown on chapter 6. The total number of NN parameters are listed in the legend.

In both environments, even when the hidden-layer weights are learned, the spatially-biased sparsity is not useful in RL control for these domains. Why, then are convolutional networks — which are spatially biased — so ubiquitous in MinAtar? Is it the fact that CNNs have weight sharing that makes their spatially biased kernels so useful? Or, is it their prevalence when applying them to larger environments such as the Arcade Learning Environment (ALE) (Bellemare et al., 2012)? We leave these questions for future work.

So far, all the sparse hidden layer structures have been obtained by an auxiliary learning mechanism (as in the case of PANs), or hand-coded like the spatially-biased architecture. However, it remains to be seen whether algorithms that generate NN sparsity end-to-end can lead to higher performance gains. In the next chapter, we investigate how predictive sparsity compares against sparse structures learned end-to-end.

Chapter 7

Learned Hidden Layer Weights and Learned Sparsity

In this chapter, we investigate the last research question: *How does predictive sparsity compare against sparse NN structures learned end-to-end for control?*

In the supervised learning setting, previous work that generates NN sparsity end-to-end parameterizes binary masks with respect to the NN weights (Liu et al., 2020). In the backward pass, the authors approximate the derivative of the non-differentiable step-function through a variant of the long-tailed estimator (Xu & Cheung, 2019), yet they do not justify this choice analytically. On the other hand, in the RL control setting, pruning methods zero out a fraction of NN weights periodically using designer-specified heuristics (Evci et al., 2020; Grooten et al., 2023; Sokar et al., 2021); for example, pruning out weights that have the smallest magnitudes. Although pruning techniques are more popular than using trainable masks, they are not *end-to-end* in the sense that the pruning function is not explicitly written in the training loss, and is therefore not differentiated with respect to learnable parameters. In addition, pruning often relies on a number of user-specified hyper-parameters, such as a pruning schedule. A more simple and common approach in machine learning is to apply L1 regularization to the training loss (Hastie et al., 2009). This is the approach we take in this chapter.

We follow the procedure described in section 4.3.1 to generate a binary mask with L1 regularization. Moreover, we sweep over values of the regularizing coefficient to find an amount of sparsity that approximately matches

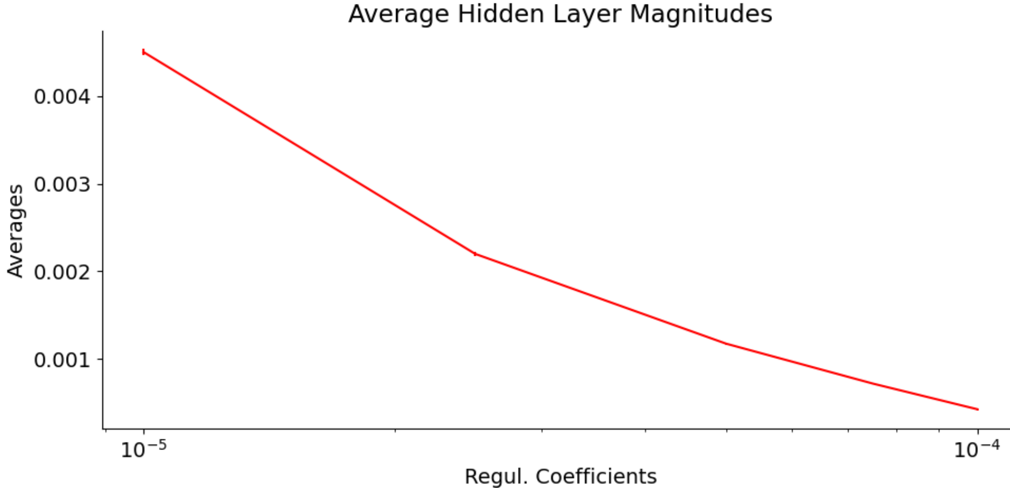

Figure 7.1: Average hidden layer weight magnitude with respect to regularization coefficient in *Breakout*. Averages were computed over 30 independent trials. Error bars are shown as vertical line segments.

that of the predictive, random and spatial architectures for consistency. Recall from section 4.3.1 that we measure the amount of sparsity induced by L1 regularization as the percentage of hidden layer weight magnitudes that fall below average. Figures 7.1 and 7.2 show the average hidden layer weight magnitudes and the percentage of weights that fall below average respectively, in the *Breakout* environment. For the *Space-Invaders* environment, the corresponding plots are shown in figures 7.3 and 7.4.

Note that since the amount of sparsity is dictated by the regularization coefficient, we did not get the exact same number of zeroes in the mask matrix as the other architectures. However, we came reasonably close. For example, in *Breakout*, a regularization coefficient of 2.5×10^{-5} resulted in $58,352$ hidden layer parameters for the L1-sparse architecture. Meanwhile the predictive, random and spatial architectures each have $57,600$ network weights in their hidden layers. On the other hand, in *Space-Invaders*, a regularization coefficient of 2×10^{-5} resulted in a sparse network with $108,158$ hidden layer parameters, while the other three sparse networks had only $97,200$ weights in their hidden layers.

Since L1 regularization is applied to all weights in the network at once,

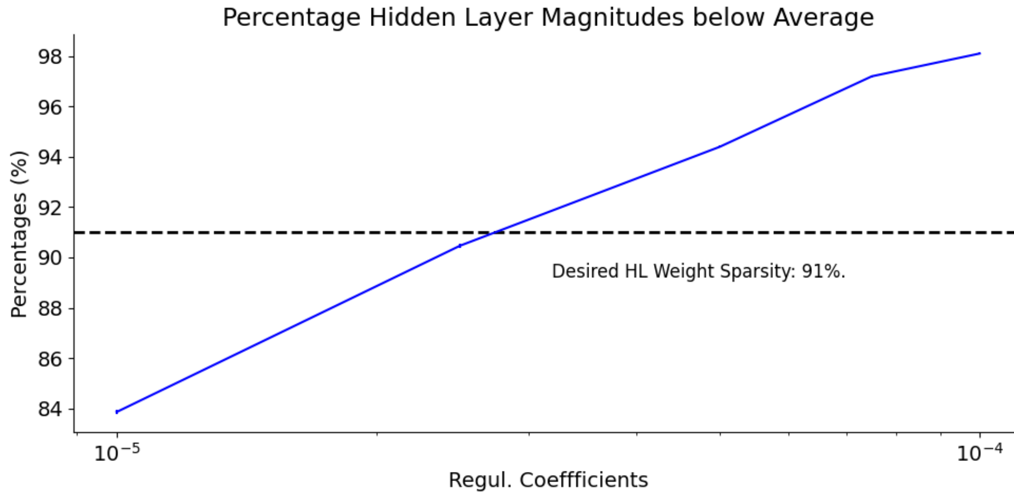

Figure 7.2: Average percentage of hidden layer weight magnitudes that fall below the mean in *Breakout*. Averages were computed over 30 independent trials. Error bars are shown as vertical line segments.

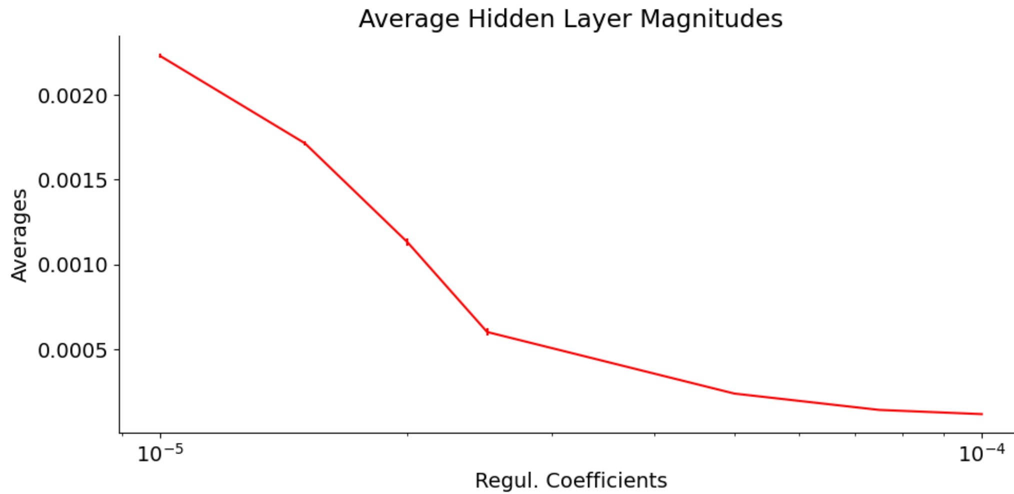

Figure 7.3: Average hidden layer weight magnitude with respect to regularization coefficient in *Space-Invaders*. Averages were computed over 30 independent trials. Error bars are shown as vertical line segments.

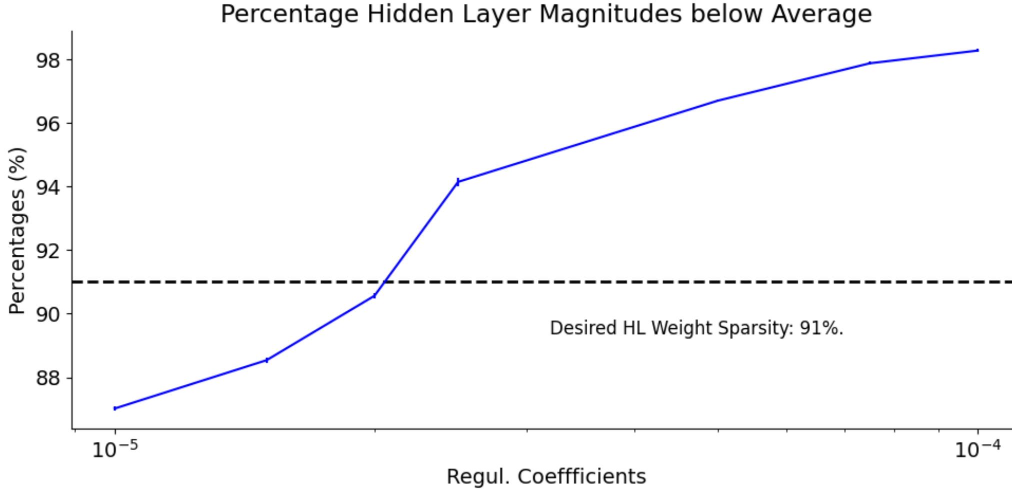

Figure 7.4: Average percentage of hidden layer weight magnitudes that fall below average in *Space-Invaders*. Averages were computed over 30 independent trials. Error bars are shown as vertical line segments.

the distribution of zeroed-out weights is not uniform across all the column vectors of the hidden layer matrix. In this sense, the hidden layer features are not generated from a constant number of inputs – some features might be made from all the elements in the observation vector, while others are made from very few. Once we generate the binary masks through this approach, we impose them onto DQN's hidden layer and restart learning from scratch. We perform two experiments: we compare the average return of the predictively-sparse DQN agent to the L1-sparse agent in two scenarios: (1) when the hidden layer weights are randomly initialized and held fixed, and (2) when the DQN hidden layer weights are learned end-to-end.

7.1 Hypotheses

Due to the greater flexibility that L1 regularization has to mask out weights in a non-uniform fashion throughout the hidden layer, we hypothesize that L1 sparsity will yield a higher performance than all the other sparse networks in both environments.

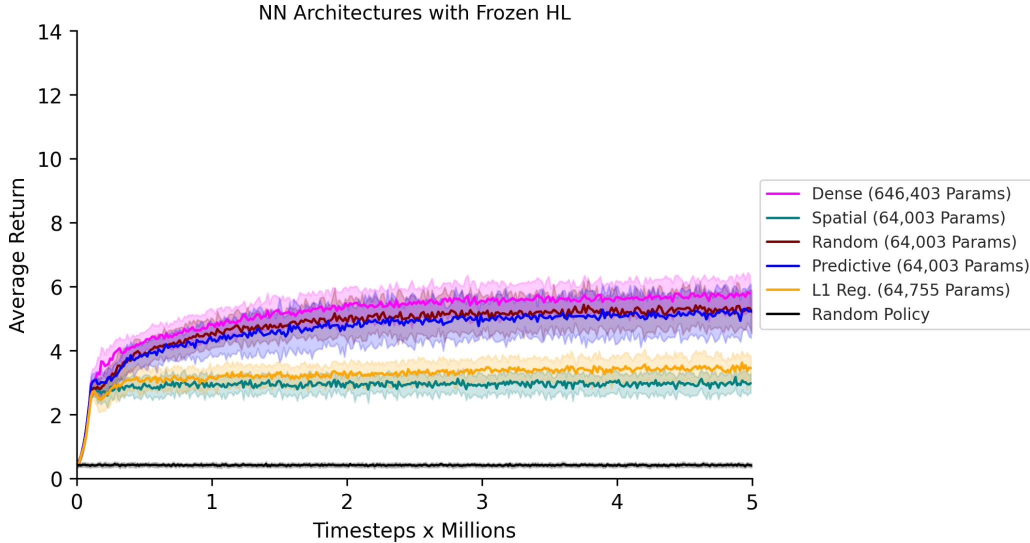

Figure 7.5: Average Return incurred on *Breakout*, for DQN architectures whose hidden layer is fixed. The L1-sparse agent is shown in orange. The total number of NN parameters are listed in the legend.

7.2 Empirical Results in *Breakout*

Figure 7.5 shows the average returns on *Breakout* for all sparse network architectures with fixed hidden layer weights, now including L1-induced sparsity. Clearly, our hypothesis is refuted here, since L1 sparsity performs significantly worse than random and predictive sparsities, yet better than the spatial architecture on average.

We also investigate how the L1 sparse agent performs when the hidden layer weights are learned, as shown in Figure 7.6. In this scenario, we find that the L1 sparse agent performs better than random sparsity, yet there is no significant difference compared to the predictive and fully-connected networks.

Overall, our results in *Breakout* suggest the following answer to the question that prompted this chapter: predictive sparsity is more useful than a sparse NN structure learned end to end via L1 regularization when the hidden layer weights remain fixed. However, when the hidden layer weights are learned, predictive sparsity does not provide significant performance gains compared to the L1 sparse counterparts.

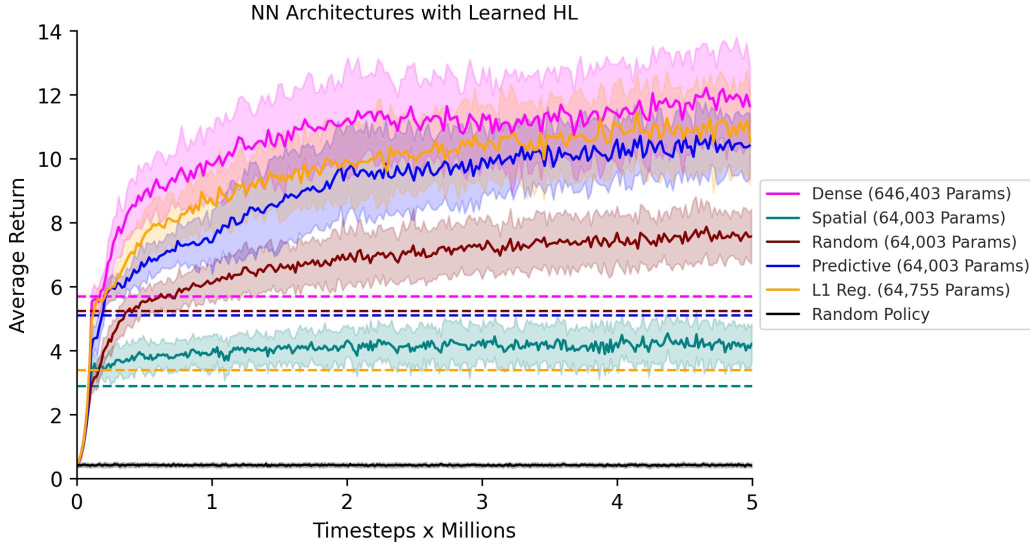

Figure 7.6: Average Return incurred on *Breakout*, for DQN architectures whose hidden layer is learned end-to-end. The L1-sparse agent is shown in orange. Horizontal dashed lines indicate the final performances when the hidden-layer weights are never learned. The total number of NN parameters are listed in the legend.

7.3 Empirical Results in *Space-Invaders*

As for *Space-Invaders*, Figure 7.7 shows the average return for all five network architectures when the hidden layer weights remain fixed at random initial values. In this domain, our hypothesis claim is refuted, as the L1-sparse network with fixed weights yields a significantly lower average return compared to the spatially-biased network.

On the other hand, our hypothesis proves to be valid when the hidden-layer weights are learned. In this case, we see that L1-induced sparsity provides larger performance gains compared to random, predictive and spatial sparsities. *Why?* Recall that in order to arrive at the L1-sparsity, we trained a dense network with a regularized loss function. This means that the L1 sparse structure is optimized for a NN that is learned end-to-end through backpropagation. For this reason, L1 sparsity indeed results in high performance when the NN is trained — this is precisely what Figure 7.8 shows. In other words, the learning task that generated the L1 sparsity is the same task to approximate the value function. On the other hand, Prediction Adapted Networks is

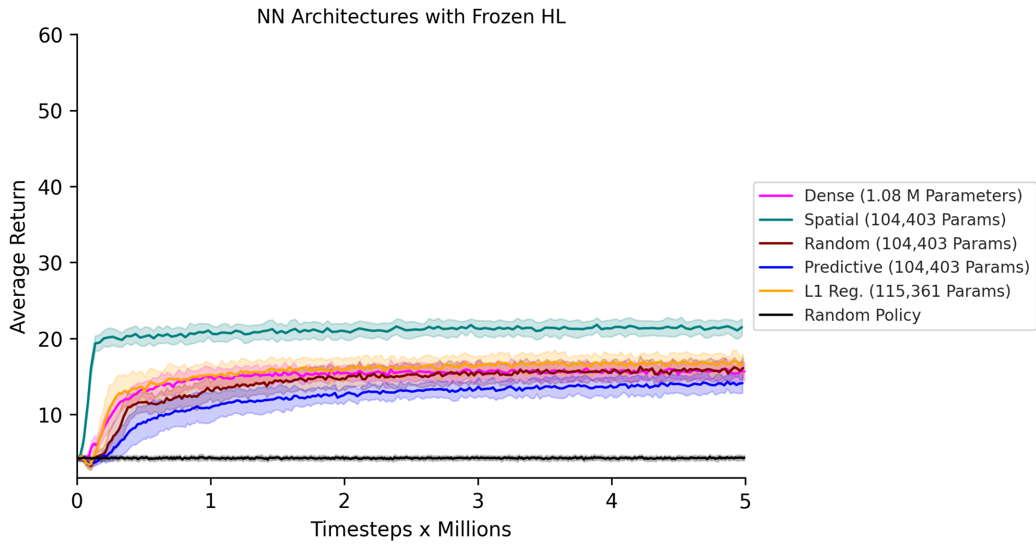

Figure 7.7: Average Return on *Space-Invaders*, for DQN architectures whose hidden layer is fixed. The total number of NN parameters are listed in the legend.

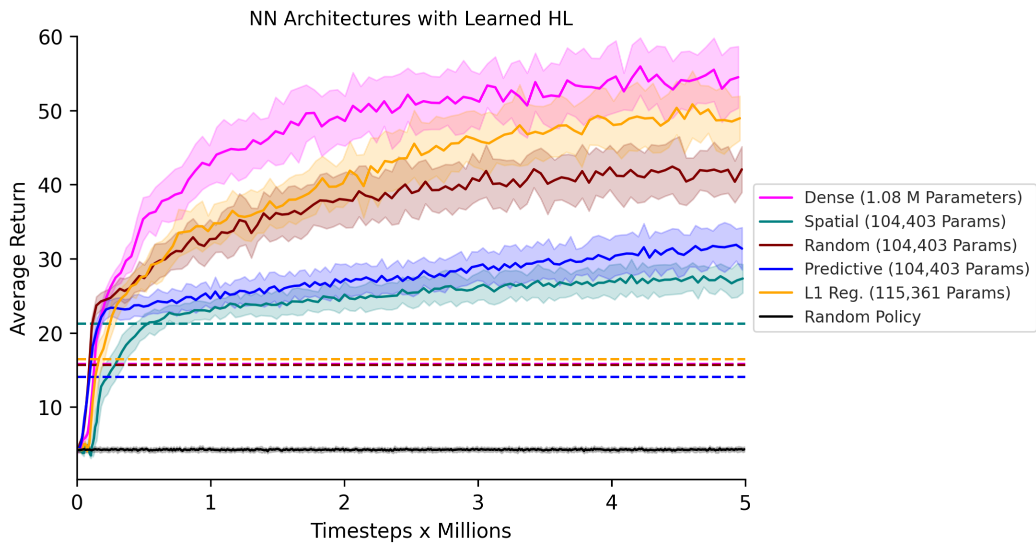

Figure 7.8: Average Return on *Space-Invaders*, for DQN architectures whose hidden layer is learned end-to-end. Horizontal dashed lines indicate the final performances when the hidden-layer weights are never learned, as shown on chapter 7. The total number of NN parameters are listed in the legend.

an auxiliary learning mechanism that generates sparse connections in a way that does not directly optimize DQN's off-policy learning objective.

Lastly, we now address the motivating question of this chapter for the *Space-Invaders* domain: in either the scenario where the hidden layer weights are fixed or learned, our results suggest that predictive sparsity is significantly less useful than a sparse structure learned end-to-end via L1 regularization.

Chapter 8

Experiment Methodology in Prediction

In this chapter, we describe the experiments that investigate our research questions in the RL prediction problem setting. Recall that in the prediction problem, the goal is to approximate the expected return, conditioned on a policy. This is different from the control problem, where we want to learn a policy. The prediction problem can be formulated in two ways. In the first formulation, the agent is handed a fixed policy, and it executes that policy in the environment. As the agent experiences observations and rewards, it can then form an estimate of the expected return. Another way to formulate the prediction problem is the setting where the agent passively experiences observations and rewards as an incoming stream of data, with no actions required and thus, no policy present. The latter is the scenario that we work in.

8.1 Prediction Environment

Specifically, we stick to the same environment used in the original PANs work, namely the *Frog's Eye* environment (Martin and Modayil, 2021). This domain mimics the visual field of a frog, with 4000 input sensors randomly scattered in space. These sensors are encoded as a binary observation vector. When a fly comes in close proximity to a sensor, the sensor turns on – its value turns from zero to one; otherwise, the sensor remains off. The fly is drawn to the center of

the visual field, towards a target region. Once the fly reaches this target, the agent receives a reward of $+1$. Otherwise, the reward is zero. The observations are corrupted with noise in the following manner: with a probability of 0.25, we sample an input and flip its value from zero to one or vice-versa.

8.2 Learning Algorithm

In the original PANs work, the linear GVFs and the output layer of the value network are learned through TD(λ). As mentioned in the Background chapter, implementing TD(λ) to update the hidden layer weights requires much more computational resources, since we need to store and update as many eligibility traces as there are neurons in the value network. For this reason, we adopt the forward view of eligibility traces and compute the truncated n-step λ return as our learning target with $n = 30$. As done in the original PANs work, we set $\lambda = 0.8$ and the discount factor $\gamma = 0.99$.

We initialize the value network randomly according to a standard normal distribution and compare the performance when the hidden-layer weights are held fixed vs. learned end-to-end. We use the stochastic gradient descent optimizer and back-propagation to train the network.

8.3 Value Network Architecture

As done in the RL control chapters, we implement and train a shallow value network with a single hidden layer. To generate the binary mask matrix, we use the same two phase method described in the RL control methodology chapter. In phase one, we ran PANs in the *Frog's Eye* domain for 5 million steps over a single random seed and extracted the top-k weights for all general value functions. The indices of these top-k weights encode the binary columns of the mask matrix.

We follow most of the same hyper-parameters used in the PANs paper to define the value network: the number of neighborhoods $m = 4000$, the number of neighbors $k = 10$ and the activation function is the ReLU. As for the number of non-linear features per neighborhood, the authors set $n = 100$,

but we cut this value to $n = 10$; this was done to maintain our value network at a reasonable size within memory constraints. We made sure that with a smaller number of non-linear features, we still achieved the same order of learning curves shown by the authors. Overall, the hidden layer weight matrix of our value network has dimensions $4000 \times 40,000$ which total up to 16M parameters. As opposed to the ensemble network used in the original PANs work, we do not impose parameter sharing across neighborhood features. This should not negatively affect results, because a higher diversity of weights in the hidden layer only increases the value network's representation capacity. Furthermore, in the original PANs work, the hidden layer bias units are held at -4, in order to allow the ReLU to filter out accidental firing of the sensors. We also initialize the network's hidden layer bias units to -4 for consistency.

8.4 Baselines

Similar to the RL control results, our main baselines are a value network architecture with random connections and an architecture with spatially biased connections. Both of these have the same number of neighbors per neighborhood as the predictive architecture (i.e., the same number of "1"s in each column of the binary mask matrix), and the same number of non-linear features. Figures 8.1, 8.2 and 8.3 show examples of predictive, spatial and random neighborhoods respectively. In chapter 10, we apply L1-regularization to the training loss and sweep over the regularization coefficients to find an amount of sparsity close to that of the other sparse architectures, namely 99.775% sparsity. We follow the same procedure in the attempt to generate L1-induced sparsity as described in chapter 4, under section 4.3.1.

8.5 Evaluation Metrics

For the prediction task, we measure performance in the same way as done by Martin and Modayil: by computing the average square return error. First, the square return errors are calculated for all time steps in the trajectory. Then, we bin these errors over windows that span across $100,000$ time steps. Finally,

the binned quantities are averaged over 30 independent trials. Here, a trial corresponds to a random seed used to initialize the value network's weights.

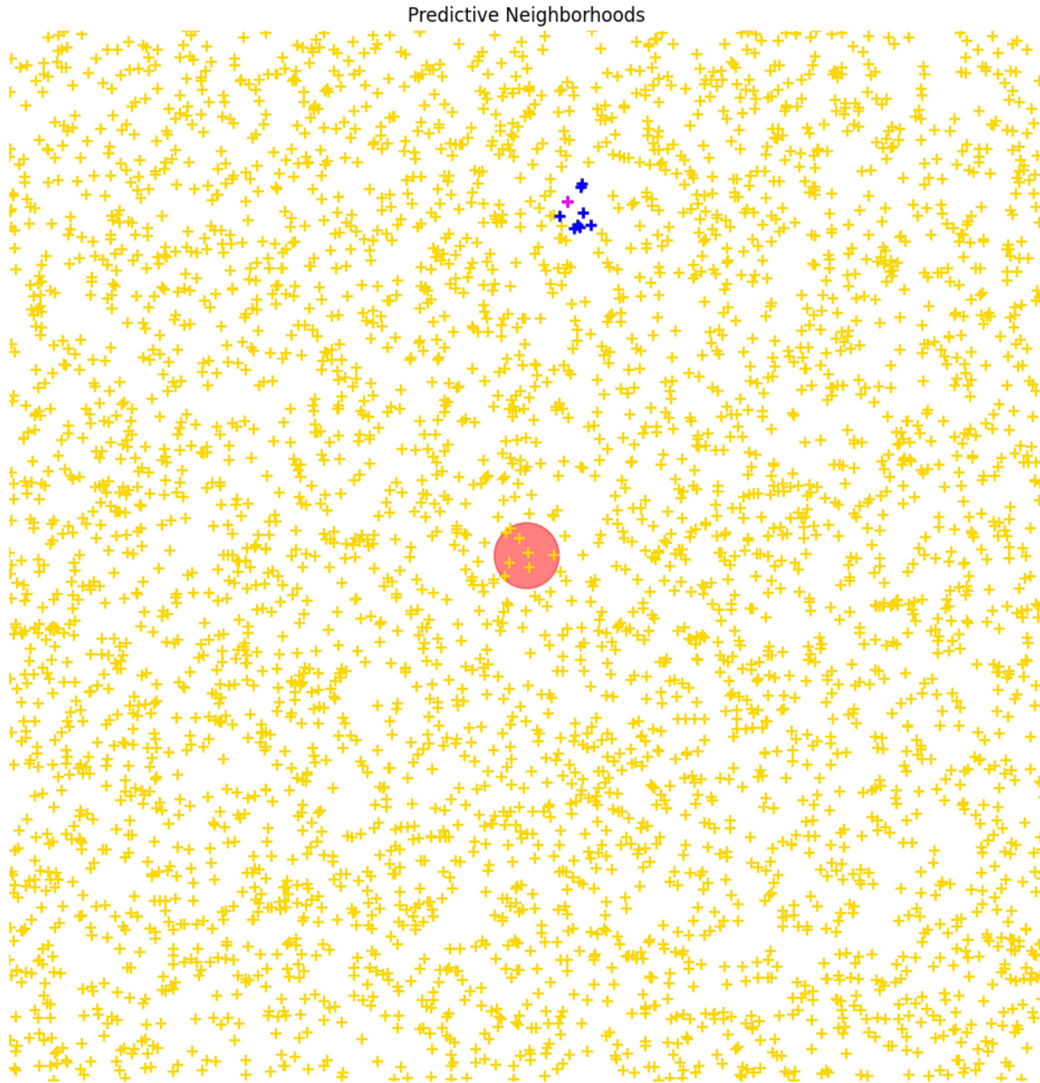

Figure 8.1: One of the predictive neighborhoods. The yellow "+" indicate the sensor locations that make up the observation vector. The blue "+" indicate the observation components that belong to the neighborhood. The violet "+" shows the specific sensor that the GVF predicts (i.e., the cumulant of the GVF). Finally, the red circle in the middle represents the target region.

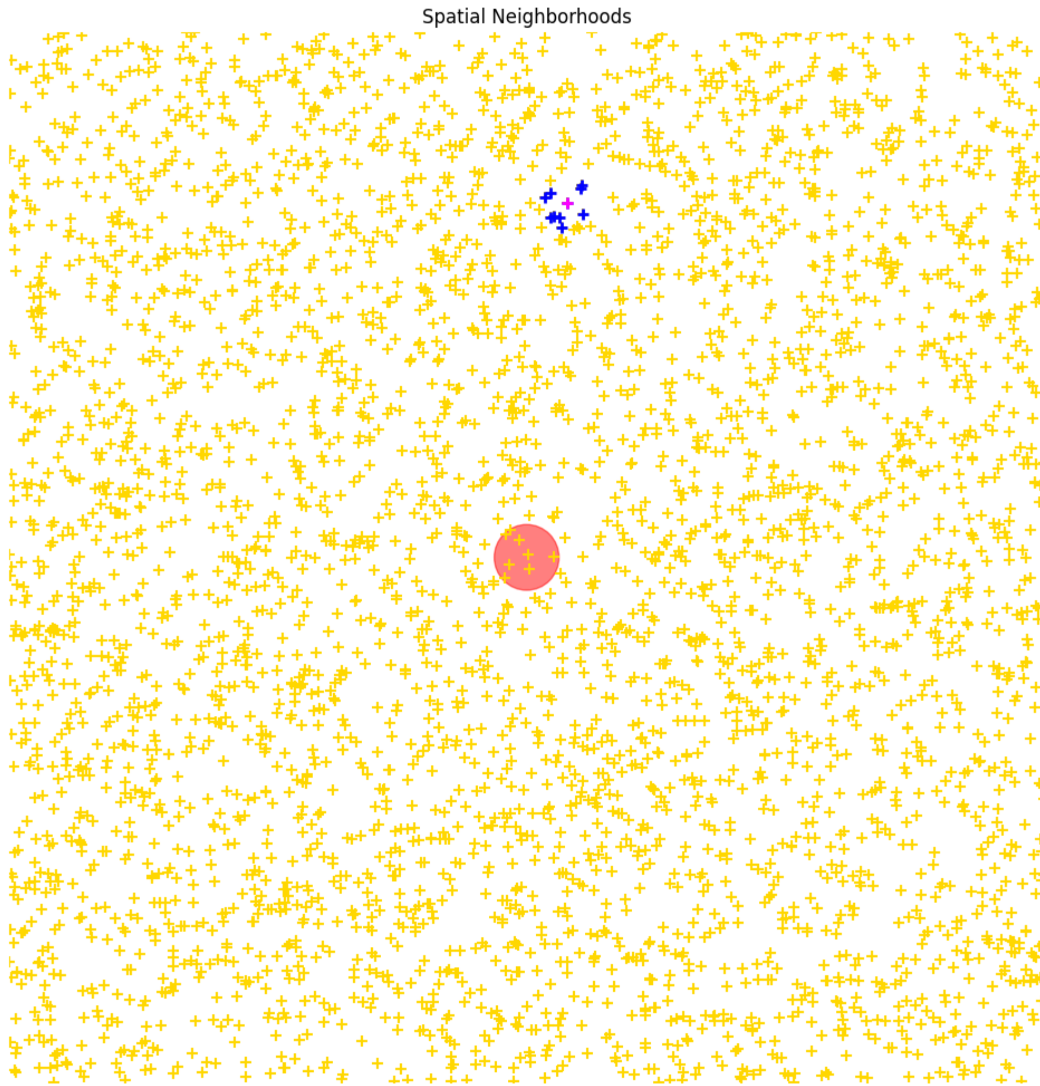

Figure 8.2: One of the spatial neighborhoods in the *Frog's Eye* environment.

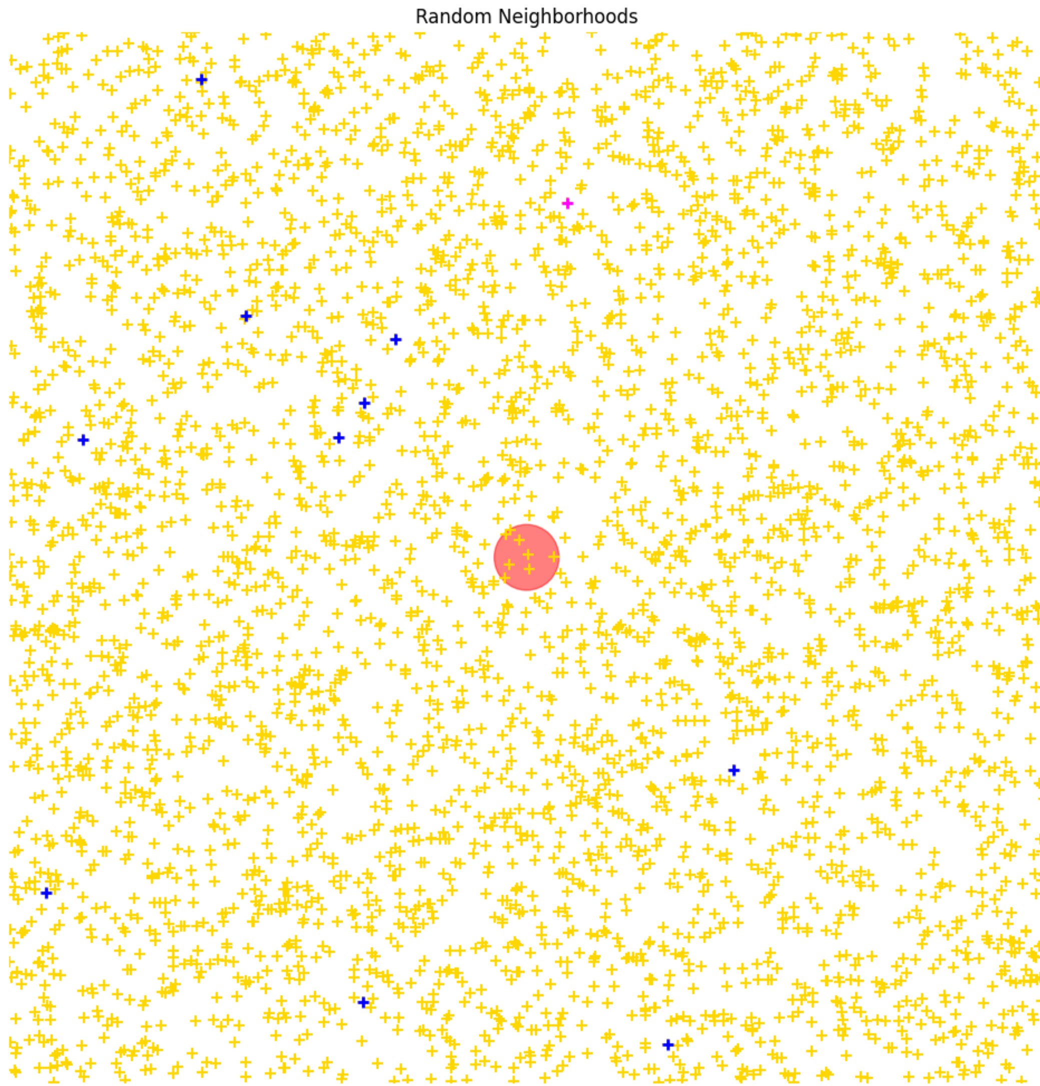

Figure 8.3: One of the random neighborhoods in the *Frog's Eye* environment.

Chapter 9

Learned Hidden Layer Weights and Fixed Sparsity in Prediction

In this chapter, we describe the experiments and results that investigate the following question in the RL prediction problem: *Do PANs provide performance gains when the value network's weights are learned end-to-end?*

Recall that the original PANs work adapted NN connectivity, yet kept the hidden layer weights fixed at random values. We compare the performance of sparse value networks whose hidden-layer weights are fixed at initialization vs. NNs whose weights are learned end-to-end from the start. As mentioned in the methodology chapter, we ran n-step truncated TD(λ) on the *Frog's Eye* domain while sweeping over the step-size parameter. We kept all environment and problem-specific hyper-parameters the same as in the original PANs paper.

9.1 Hypotheses

Our first hypothesis, which we believe to be trivial, is that a sparse value network with learned hidden layer weights will perform at least as well as one that keeps its hidden layer weights fixed at random values. We believe this to be true because SGD optimization can only improve the NN's representations. Moreover, the original PANs paper showed that in the *Frog's Eye* environment, spatially biased sparse connections outperformed both random sparsity and the linear architecture, while predictive sparsity came a close second. These results guided our second hypothesis: out of the three sparse NN architectures

(predictive, spatial and random), spatially biased and predictive sparsities with learned hidden layer weights will outperform random sparsity. Furthermore, we expect the predictive and spatially biased architectures to perform on par, since their topological structures are similar.

9.2 Empirical Results in *Frog's Eye*

Here, we show the performance of all value network architectures in the RL prediction task in two scenarios: when the hidden layer weights are fixed vs. learned. Figure 9.1 shows the average learning curves when the hidden layer weights are learned end-to-end through gradient back-propagation. To our surprise, learning the hidden layer weights does not result in performance gains for any architecture. Perhaps the PANs authors did not bother to learn the value network's hidden layer weights end-to-end, because it would not impact the final performance anyway.

We believe that these unusual results are due to a property of the problem setting, which we call the *many-features regime*. To be in this regime means that a lot of the random features in the NN are enough to approximate a target function with low error. Therefore, as our results suggest, additional gradient-descent steps do not help making more accurate predictions. Further, this regime is a property of both the domain and the type of random features used. The *many-features regime* is not a new concept in machine learning (ML). Previous work in the supervised learning setting has analytically shown that a shallow NN with random hidden layer features and learned outer layer weights can generate a classifier that is not much worse than one where we optimally tune the non-linearities (Rahimi & Recht, 2008). The authors also backed-up this finding empirically on three datasets. In an earlier work, Rahimi and Recht also showed results suggesting that in some regression and classification tasks, simple linear ML algorithms applied to random features of the inputs can outperform certain kernel machines (Rahimi & Recht, 2007). Furthermore, in the online learning setting, Sutton and Whitehead suggested that random linear projections can result in useful and complex features

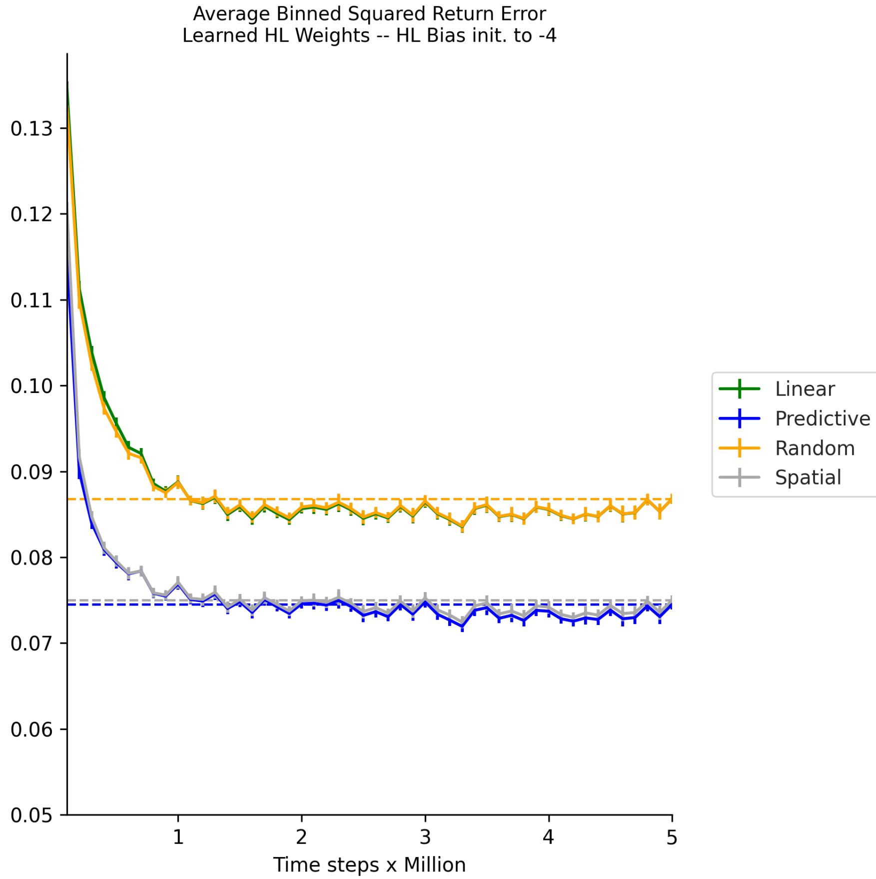

Figure 9.1: Average binned square return error incurred on the *Frog's Eye* domain, for value network architectures whose hidden layer is learned end-to-end. Hidden layer bias units are initialized to -4. The dashed line show the final performance for the same NNs with fixed hidden layer weights that were never learned; this was added for comparison.

(Sutton & Whitehead, 1993).

In the following chapter, we aimed to investigate how predictive sparsity compares agains sparsities learned end-to-end through L1 regularization.

Chapter 10

Learning Sparsity in Prediction

In this chapter, we describe the empirical results that attempt to answer the third and last research question stated in the introduction, in the RL prediction problem: *How does predictive sparsity compare against sparse NN structures learned end-to-end?* In order to generate a sparse architecture through gradient back-propagation, we employ L1 regularization; this was the same strategy used in the RL control experiments of chapter 7.

To our surprise, L1 regularization did not behave as we expected in this domain. First, we expected to see that as the regularization coefficient β introduced in equation (4.1) increases, the average magnitude of the hidden layer weight matrix should decrease. However, the results shown in Figure 10.1 suggest otherwise: the average hidden layer weight magnitude remains roughly the same for values of β in $[0.01, 0.99]$, then decreases steeply when β grows to a value of 2. At first glance, the plateau that we observe for values of β in the set $\{2.0, 10.0, 50.0\}$ seem to make sense, because the L1 regularization term dominates the loss function, reducing all the weights to near zero.

Second, we also expected that greater regularization coefficients would result in a larger proportion of hidden layer weights above average. We were aiming to find a regularization coefficient that yields a sparsity level close to that of the predictive, random and spatial architectures. Instead, results in Figure 10.2 show that for values of β in the range $[0.01, 0.99]$, the proportion of hidden layer weights remains approximately the same, around 51.2%. As β grows above 1.0, this percentage drops very close to 50%.

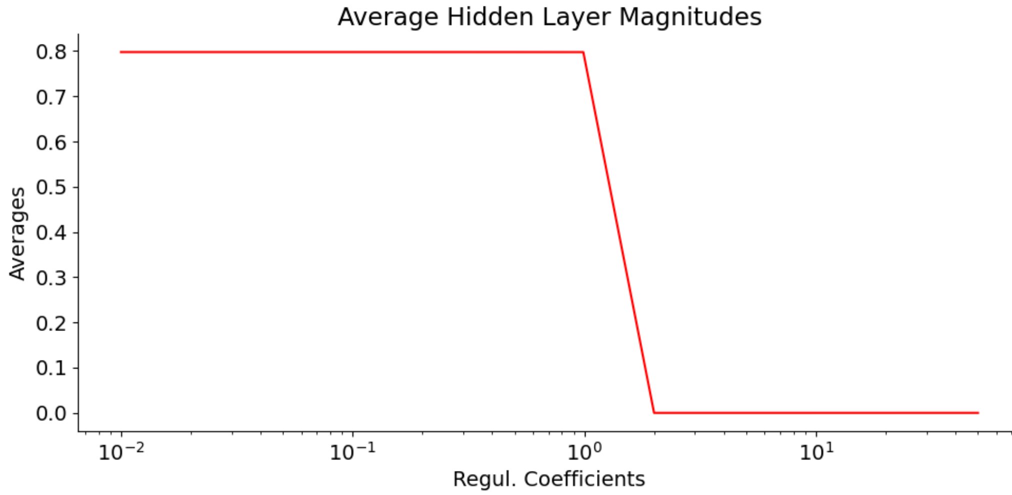

Figure 10.1: Average hidden layer weight magnitude with respect to L1-regularization coefficient in *Frog's Eye*. Averages were computed over 30 independent trials. Error bars, displayed as vertical lines, are indistinguishable, as they range in the order of 10^{-12} to 10^{-6}.

Overall, these results suggest that, regardless of the regularization coefficient's value, all the hidden layer weights are shrinking roughly equally. Hence, the proportion of weights below or above the average remains roughly the same. Thus, L1 regularization is not able to generate sparsity in this regime.

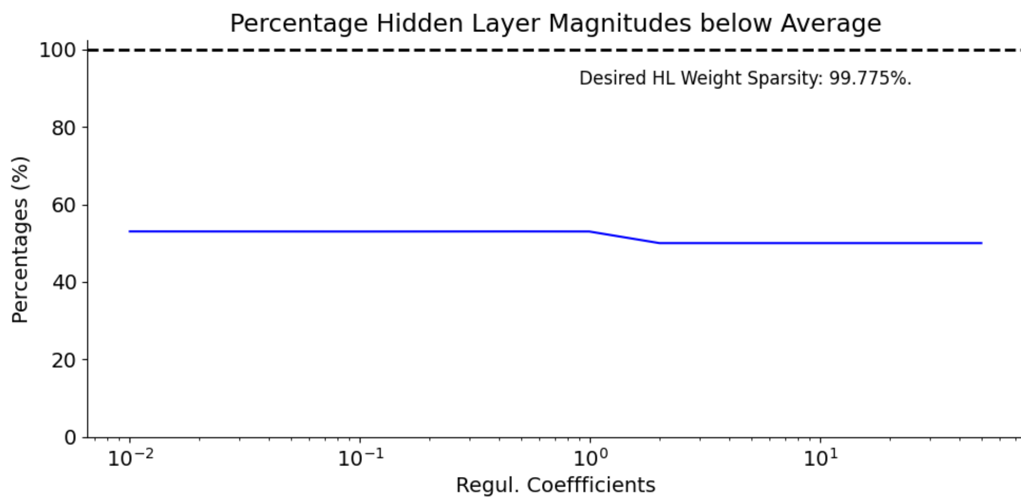

Figure 10.2: Percentage of hidden layer weight magnitudes that fall below average in *Frog's Eye*. Averages were computed over 30 independent trials. Error bars, displayed as vertical lines, are indistinguishable, as they range in the order of 10^{-4} to 10^{-2}.

Chapter 11

Conclusion

In this thesis, we extended previous work that introduced *Prediction-Adapted Networks* (PANs). We investigated three research questions:

1. *Do the statistical benefits of PANs with fixed hidden layer weights carry over to RL control in multiple environments?*
2. *Do we observe the same benefits when we learn the value network's hidden layer weights end-to-end?*
3. *How does predictive sparsity compare against sparse NN structures learned end-to-end?*

The first set of experiments show results in the RL control setting with the DQN algorithm in two MinAtar environments: *Breakout* and *Space-Invaders*. As summarized in chapters 5 and 6, empirical results in *Breakout* suggest that a predictive sparse structure can be useful in the RL control setting when hidden layer weights are learned. On the other hand, when the hidden layer weights remain fixed to a random initialization, the predictive sparse structure does not perform any better than a random sparse architecture, on average. Furthermore, in this environment predictive sparsity can outperform a sparse hidden layer structure learned end-to-end via L1 regularization when the hidden layer weights remain fixed; once these weights are learned, both architectures perform roughly on par. In *Space-Invaders*, predictive sparsity incurred no significant performance gains compared to either a random sparse architecture or an L1-sparse NN. This was observed in either scenario: when

the hidden layer weights were fixed or learned. Overall, the performance gains of PANs can carry over to the control setting in one of our domains but not the other. As our results suggest, we see PANs greatest success against other sparse architectures when the hidden layer weights are learned end-to-end in *Breakout*. We believe that this environment might be better suited for PANs, since it does not have any non-stationary dynamics, compared to *Space-Invaders*.

The second set of experiments investigate the last two research questions in the RL prediction setting in the *Frog's Eye* environment. To our surprise, our results suggest that there is no significant performance gains when the hidden layer weights are either learned vs. fixed to initial random values. We believe this is due to a property inherent to the domain: in the *Frog's Eye* environment, $40,000$ random hidden layer features are enough to make good predictions of the return. Therefore, training the NN does not increase representational capacity. Finally, results in chapter 10 suggest that L1 regularization is not a viable method of generating sparsity in this domain.

In the RL control setting, some future work includes investigating whether QV(λ) with a random behaviour policy yields different neighborhoods compared to an ϵ-greedy policy. Perhaps our choice for a policy biased the GVFs and thereby, the neighborhoods. After all, the distribution of observations that emerge depends on the actions that the agent takes. In addition, another avenue for future work is to compare predictive sparsity to more popular methods that sparsify a NN, such as Dynamic Sparse Training (DST) techniques (Graesser et al., 2022; Grooten et al., 2023; Sokar et al., 2021). These methods rely on pruning algorithms to remove or reset NN weights deemed less useful.

On the other hand, in the RL prediction problem, one direction for future research is to investigate why L1 regularization does not increase the proportion of hidden layer weight magnitudes below average in the *Frog's Eye* domain. Moreover, understanding the connection between the *many features* regime (described in chapter 10) and why L1 regularization does not behave as expected is another interesting avenue that would enhance our understanding of this type of domain.

References

Ahmad, S., & Scheinkman, L. (2019). How can we be so dense? the benefits of using highly sparse representations. *ArXiv*, *abs/1903.11257*.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2012). The arcade learning environment: An evaluation platform for general agents (extended abstract). *International Joint Conference on Artificial Intelligence*.

Evci, U., Elsen, E., Castro, P., & Gale, T. (2020). Rigging the lottery: Making all tickets winners. *Proceedings of Machine Learning and Systems*.

Graesser, L., Evci, U., Elsen, E., & Castro, P. S. (2022). The state of sparse training in deep reinforcement learning. *ArXiv*, *abs/2206.10369*.

Grooten, B., Ghada, S., Dohare, S., Mocanu, E., Taylor, M. E., Pechenizkiy, M., & Mocanu, D. C. (2023, February). Automatic noise filtering with dynamic sparse training in deep reinforcement learning. https://arxiv.org/pdf/2302.06548

Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (Second). Springer.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., & Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, *22*, 241:1–241:124.

Liu, J., Xu, Z., Shi, R., Cheung, R. C. C., & So, H. K. (2020). Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers. *International Conference on Learning Representations*.

Martin, J. D., & Modayil, J. (2021). Adapting the Function Approximation Architecture in Online Reinforcement Learning.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529–533.

Modayil, J., & Abbas, Z. (2023). Towards model-free RL algorithms that scale well with unstructured data. *ArXiv*, *abs/2311.02215*.

Modayil, J., White, A., & Sutton, R. S. (2011). Multi-timescale nexting in a reinforcement learning robot.

Rahimi, A., & Recht, B. (2007). Random features for large-scale kernel machines. *Proceedings of the 20th International Conference on Neural Information Processing Systems*, 1177–1184.

Rahimi, A., & Recht, B. (2008). Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. *Proceedings of the 21st International Conference on Neural Information Processing Systems*, 1313–1320.

Rummery, G., & Niranjan, M. (1994). On-line Q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*.

Sokar, G., Mocanu, E., Mocanu, D. C., Pechenizkiy, M., & Stone, P. (2021). Dynamic sparse training for deep reinforcement learning. *International Joint Conference on Artificial Intelligence*.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., & Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 761–768.

Sutton, R. S., & Whitehead, S. D. (1993). Online learning with random representations. In P. E. Utgoff (Ed.), *Machine learning, proceedings of the tenth international conference, university of massachusetts, amherst, ma, usa, june 27-29, 1993* (pp. 314–321). Morgan Kaufmann.

Watkins, C. J. (1989). *Learning from delayed rewards* [Doctoral dissertation, King's College in Cambridge, England].

Wiering, M. (2005). QV(λ)-learning: A new on-policy reinforcement learning algorithm. *Proceedings of the 7th European Workshop on Reinforcement Learning*, 17–18.

Xu, Z., & Cheung, R. C. C. (2019). Accurate and compact convolutional neural networks with trained binarization. *30th British Machine Vision Conference 2019, BMVC 2019, Cardiff, UK, September 9-12, 2019*, 19.

Young, K., & Tian, T. (2019). MinAtar: An Atari-inspired testbed for thorough and reproducible Reinforcement Learning experiments. *ArXiv*. https://arxiv.org/abs/1903.03176