Master of Science in Informatics at Grenoble
Master Informatique
Specialization Data Science

Sample-Efficiency and Transfer in Reinforcement Learning

Merging Episodic Control and Successor Features

David Emukpere

August 22, 2021

Research project performed at INRIA - RobotLearn team

Under the supervision of:
Chris Reinke and Xavier Alameda-Pineda

Defended before a jury composed of:
Massih-Reza Amini
Julien Mairal
Frank Iutzeler

August 2021

Acknowledgement

First and foremost, I would like to thank my supervisors, Chris and Xavi, for the incredible amount of human and technical support I received throughout my internship. I am indeed grateful for your help and advice on various areas of life.

I will also like to express my gratitude to the IDEX Université Grenoble Alpes scholarship for providing me with the financial support to make my studies possible in Grenoble.

An immense pillar of support and essence of my existence is my family. I owe everything I am today to you. To my father, a shining light of what it means to live in harmony with others, showing respect, wisdom, gratefulness, the joy of living, humility, and sincere will to help. To my mother, who nourishes with eternal love and superhuman strength and selflessness unheard of in the entire world. To my sister, whose insistent pestering is coloured with the sweetest care and unwavering fandom of her younger brother, I say a huge thank you.

Likewise, I would like to thank all of my colleagues at INRIA, RobotLearn, for making my stay there a pleasure. For being very welcoming, and for numerous chats about everything from research to life matters and learning about foreign cultures. To new friends I've made there, Gaetan, Anand, Zhiqi, Yihong, Natanael, Louis, I say a huge thank you.

I would be remiss to not mention my long-term friends who are like family to me in Europe from back home in Nigeria. Obinna, Paula, Abraham, I love you guys. To Amira, Clement, Eslam, Hadi, Ksenia, Mohammed, Oumar my lovely friends who've made my time in Grenoble a period of unparalleled growth and great friendship. Many thanks, and here's to many more years of great friendship ahead!

Abstract

A longstanding goal in artificial intelligence is to build intelligent systems that are flexible and fast learners, akin to human and animal learning. Reinforcement learning as a sub-discipline of artificial intelligence has discovered good techniques for learning by interaction with the environment, as is common in humans and animals alike. However, the flexibility and learning speed is not close to the abilities biological organisms exhibit. This problem remains a challenge for modern-day artificial intelligence. Our approach in this study investigates two frameworks for tackling the problems of learning speed and flexible transfer of skills, namely: episodic control and successor features for transfer learning, respectively. These two techniques have shown impressive results in vastly improved sample efficiency and the elegant reuse of previously learned policies. Our goal is to define and investigate a framework that seamlessly combines both benefits in a single reinforcement learning agent. We show that such a combination is possible and indeed allows us to take a step towards implementing reinforcement learning agents that learn fast and can also flexibly transfer learned skills across tasks.

Résumé

Un objectif de longue date de l'intelligence artificielle est de construire des systèmes intelligents qui soient flexibles et apprennent rapidement, à l'instar de l'apprentissage humain et animal. L'apprentissage par renforcement, une sous-discipline de l'intelligence artificielle, regroupe des techniques d'apprentissage par interaction avec l'environnement, à la manière des humains et des animaux. Cependant, la flexibilité et la vitesse d'apprentissage ne sont pas égales aux des capacités des organismes biologiques. Ce problème reste un défi pour l'intelligence artificielle moderne. Dans cette étude, nous étudions deux cadres pour résoudre les problèmes de vitesse d'apprentissage et de transfert flexible des compétences, à savoir : le contrôle épisodique et les caractéristiques du successeur pour l'apprentissage par transfert. Ces deux techniques ont montré des résultats impressionnants en améliorant considérablement l'efficacité de l'échantillonnage et la réutilisation élégante des politiques apprises précédemment. Notre objectif est de définir et d'étudier un cadre qui combine de manière transparente ces deux avantages dans un seul élément. Nous montrons qu'une telle combinaison est possible et qu'elle nous permet de faire un pas vers la mise en œuvre d'éléments qui apprennent rapidement et peuvent également transférer de manière flexible les compétences acquises entre les tâches.

Contents

— 1 —
Introduction

The idea of building intelligent agents and systems that learn purely by interaction with their environment, known as reinforcement learning [39], is an appealing approach to artificial intelligence with solid connections to neuroscience and psychology [29, 11]. Reinforcement learning has generated significant interest both in the research community and in public awareness, especially as reinforcement learning combined with deep learning [21], a paradigm known as deep reinforcement learning [2], has given rise to impressive achievements in various contexts. These include building champion game players [38, 37, 45, 30], and solving long-standing problems in biology [18] to list a few. Furthermore, reinforcement learning appears to be a framework that holds the promise of a path toward achieving human-level intelligence, a long-standing goal of the field of artificial intelligence.

However, human intelligence has defining characteristics lacking in state-of-the-art deep reinforcement learning systems. Compared to learning in humans, these systems tend to require significantly huge amounts of data to learn [19, 42], as they need a lot of (repeated) exposure to learn rules/concepts contained in data samples which manifest as *slow* learning. Technically, this is known as the sample/data efficiency problem. In contrast, humans can learn quickly and efficiently, making use of little data. As pointed out in [10], the source of slowness in deep reinforcement learning can be attributed to the requirement for incremental parameter adjustment in gradient-based optimization of deep neural networks, as well as weak inductive bias. A number of techniques have been proposed to tackle the data efficiency problem in deep reinforcement learning such as model-based reinforcement learning [26], advanced exploration mechanisms [31], episodic control [22, 9, 32], hierarchical reinforcement learning [27] and transfer learning [35].

Another typical trait of human learning and intelligence is the ability to seamlessly transfer concepts across tasks leading to efficient and flexible reuse of knowledge gained across tasks with some similarity. A key ingredient that makes this possible is called *inductive bias*. Essentially, this means that when we encounter a new problem, we typically consider a restricted set of solutions to the new problem that most likely contains an appropriate solution based on previous experiences with other problems. This contributes to vastly improved learning speed compared to if we had no previous knowledge and had to try all possible methods to solve a problem. Thus, we can describe learning in humans as typically involving strong inductive biases when faced with problems similar to previously encountered problems. Unfortunately, the ability to reuse old knowledge as efficiently in this manner is not as yet replicated in deep learning systems in general, and

this problem is typically tackled under the frameworks of transfer learning [40, 20] and meta learning [15] in deep learning and deep reinforcement learning.

1.1 Problem statement

Having identified the need for methods to improve the speed of learning, flexibility, and adaptability of reinforcement learning systems as well as proposed solutions to these problems in the literature on sample efficiency and transfer, the research question investigated in this study is the following:

Research question: Can we define a framework that integrates the speed of learning with flexible transfer using the frameworks of sample-efficient learning and transfer learning in reinforcement learning?

1.2 Investigative method and contributions

The posed research question is answered in the affirmative by this study. Our approach involves combining two techniques developed independently for sample efficiency and flexible transfer in reinforcement learning, namely:

- Episodic control [32] and,
- Successor features and generalized policy improvement(SF&GPI) [4].

We chose these two frameworks for the following reasons. First, for episodic control, it has both a well-founded biological inspiration and displays impressive sample efficiency results in reinforcement learning tasks [9, 32]. Likewise, for successor features, the elegance of the SF&GPI framework and the connections of successor representation [12, 13] to neuroscience form the basis of our motivation. Additionally, a recent study provided some evidence that humans use a strategy similar to SF&GPI for multi-task reinforcement learning [41].

For our study on integrating these two methods, we took an empirical approach. We started by defining a model which we call Successor Feature Neural Episodic Control (SFNEC). Using this model, we aimed to learn successor features for flexible transfer *rapidly* using episodic control. After implementing an agent according to this model, we evaluated it using a two-dimensional navigation task introduced in [4]. Our experimental results show that combining successor features and episodic control in SFNEC can lead to fast and adaptable reinforcement learning agents. To summarize, our main contributions in this study are listed below:

- We define a model that integrates sample-efficient learning using episodic control with transfer learning using successor features and generalized policy improvement
- We empirically validated this approach on various environments by implementing an agent developed according to the defined model

1.3 Organization of the report

The rest of this thesis is organized as follows. Chapter 2 gives a brief background on reinforcement learning, covering the field's basic concepts with a particular focus on those most relevant in the development of ideas in this study. Following this, we begin Chapter 3 with sections on previous work done in episodic memory and episodic control, elucidating its roots in psychology, then covering approaches proposed for its implementation in reinforcement learning. We end Chapter 3 with an overview of transfer learning approaches in reinforcement learning focusing on successor representations and successor features. In Chapter 4, we begin by discussing the approach taken in this study and subsequently defining the model designed and implemented in detail. We also describe all the agents implemented, which we used in our experiments. Next, Chapter 5 covers all aspects relating to the experiments used to study the performance of the proposed model with details of experimental setups followed by results for each environment used in the experiments. We continue discussing the results from our experiments in Chapter 6, where we try to understand more about the characteristics of the implemented model by evaluating different aspects of it. Finally, we round up this report in Chapter 7, summarizing the presented work and outlining perspectives for future research.

— 2 —
Background

We give a short introduction to reinforcement learning and review its main concepts in this chapter.

2.1 Reinforcement learning

Reinforcement learning[39] refers to a learning process driven by trial and error where an agent/organism attempts to maximize cumulative rewards it can obtain while interacting with its environment. As a discipline, reinforcement learning can trace its roots to two distinct fields: the psychology of animal learning and optimal control. From a computational perspective, the key tenet of this field is to build agents that learn through this agent-environment interaction illustrated in Figure 2.1:

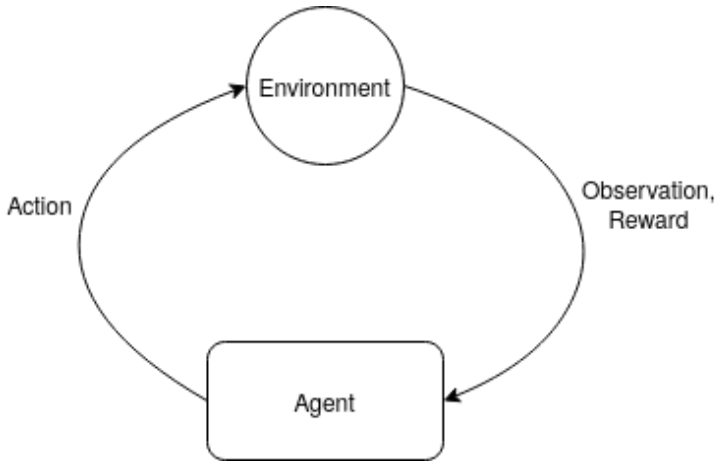

Figure 2.1 – Agent environment interaction in reinforcement learning

To formalize learning by interaction in an environment, the field makes use of the *Markov Decision Process* formalism[33]. A Markov Decision Process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, p, \mathcal{R}, \gamma)$. $\mathcal{S}$ represents the state space, while $\mathcal{A}$ is the action space. Using this formalism, the agent interacts with the environment at discrete time steps t. Furthermore, p is the state transition probability distribution function $p(s_{t+1}|s_t, a_t)$ which gives the distribution of the state at the next time step $t+1$ after an agent takes action $a_t \in \mathcal{A}$ in state $s_t \in \mathcal{S}$ at the current time step. $\mathcal{R}$ defined as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward

function associated with a transition (s_t, a_t, s_{t+1}), and $\gamma \in [0, 1]$ is the discount factor used to determine how much weight is accorded to future rewards. The goal of a reinforcement learning agent is to learn a *policy* π – a mapping between states and actions $\pi : \mathcal{S} \rightarrow q(\mathcal{A})$ – so as to maximize the expected sum of discounted rewards $G_t = \mathbb{E}^{\pi}[\sum_{i=t}^{\infty} \gamma^{t-i} r_i]$, called the *return*, where r_i are the rewards received at each time step. q in the policy definition above refers to a probability distribution over actions which accounts for the fact that the policy can either be deterministic or stochastic. The state $s_t \in \mathcal{S}$ in the MDP formalism is defined to have the *Markov* property, which means the distribution of the next state in the decision process only depends on the current state and action taken in this state. However, in practice, we might not be able to directly observe the state in a reinforcement learning problem. Thus the term *observation* is used to depict what an agent can observe from the environment, which may not directly be the state.

One way of classifying reinforcement learning methods is based on the representation of the policy and how it is optimized in order to solve a MDP. *Value function* based methods do not directly represent a policy but rather implicitly represent it using a value function which the agent learns. On the other hand *policy gradient* methods represent and optimize the policy directly, while *actor-critic* methods combine a direct policy representation termed the *actor* with a learned value function called the *critic*. Our focus from here on will be on value function based approaches. A classical method of solving MDPs using a value function is *dynamic programming*[7]. Here, the value function typically used is called the *state-action value function* $Q^{\pi}(s, a)$ which writes:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}^{\pi}\big[G_t | s_t = s, a_t = a\big]. \tag{2.1}$$

For a given Q-function, we can define a deterministic greedy policy π that is optimal with respect to Q as $\pi(s) = \text{argmax}_a Q(s, a)$. This policy is the workhorse of dynamic programming and methods based on it. The critical insight in dynamic programming is that the action-value function(Q-value function), when expanded, leads to a recursive definition:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}^{\pi, p}\big[r_t + \gamma Q^{\pi}\big(s_{t+1}, \pi(s_{t+1})\big)\big]. \tag{2.2}$$

Equation 2.2 is known as the *Bellman equation*. A fundamental result from dynamic programming is that an optimal Q-value function, Q^*, satisfies the Bellman equation:

$$Q^*(s_t, a_t) = r_t + \gamma \max_{a_{t+1}} \big[Q^*(s_{t+1}, a_{t+1})\big]. \tag{2.3}$$

Thus, solving an MDP with dynamic programming proceeds by instantiating a table of Q-values for each state-action pair and alternating between two processes until stabilization. This first process is known as *policy evaluation*. It consists of applying the Bellman equation at each state-action value pair in the table. While the second process known as *policy improvement* consists of deriving a new policy π' that is greedy with respect to the resulting Q-value function of the policy evaluation step i.e., $\pi'(s) \in \text{argmax}_a Q^{\pi}(s, a)$. However, dynamic programming requires knowledge of the transition dynamics p, typically unknown in a reinforcement learning problem, to compute expectations in the policy evaluation phase. Here, another dimension along which to classify reinforcement learning approaches emerges. If we are given the transition model or learn this model through interaction with the environment, which we can then use for dynamic programming, then this class

of methods is known as *model-based reinforcement learning*. In contrast, if we assume no knowledge of the model and attempt to estimate the action-values using samples from environment interaction, this is known as *model-free* reinforcement learning.

For model-free methods, there exist two major ways of estimating the Q-values. The first method estimates Q-values by averaging samples of entire episode rollouts i.e. $r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots + \gamma^T r_T$. These samples are known as *Monte-Carlo* estimates, and the basic idea is that by the law of large numbers, the average of a large number of these estimates will converge to the true expected value for Q. The second method for estimating Q-values involves a technique known as *bootstrapping*. In this case, Q-values are estimated using the Bellman equation by replacing the one-step expectation over the distribution of the next state and rewards, with samples gotten from the agent's interaction with the environment i.e. $Q^\pi(s_t, a_t) = r_t + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})$. This method is known as temporal difference learning. Both have their benefits; Monte-Carlo learning uses unbiased estimates with high variance due to intervening actions driven by stochastic transitions (and rewards). Meanwhile, temporal difference learning typically has low variance but is biased because we use the Q-value function in bootstrapping. Thus, a method for defining a tradeoff between both approaches is to extend temporal difference from using a single step to the use of N-step estimates which combine rewards up to a horizon length of N with bootstrapped value beyond the horizon i.e.

$$Q^\pi(s_t, a_t) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N Q^\pi\big(s_{t+N}, \pi(s_{t+N})\big). \tag{2.4}$$

In general, for all the above mentioned methods, learning the Q-value proceeds with the agent's experience by updating the Q-function according to the stochastic approximation algorithm [34]:

$$old\_estimate = old\_estimate + \alpha(new\_estimate - old\_estimate), \tag{2.5}$$

where α is a step size parameter used to determine how much to move the old estimate towards the new estimate, and $new\_estimate$ corresponds to either Monte-Carlo, single-step, or N-step temporal difference estimates.

A quintessential problem in reinforcement learning is the exploration-exploitation dilemma which refers to the tradeoff an agent faces in deciding whether to try new actions or stick to its current strategy. Naturally, an agent needs to exploit knowledge it has acquired in order to accumulate rewards. Nonetheless, it is also vital for an agent to explore new actions to learn about its environment and avoid getting stuck in the over-exploitation of a sub-optimal policy. A simple approach used to handle this dilemma while learning is known as ϵ-greedy exploration. This method corresponds to a behavioural policy controlled by exploration parameter ϵ wherein an agent chooses to act optimally according to its value function when the result of drawing a random number in $[0, 1]$ is greater than ϵ and acts uniformly at random otherwise.

In the above treatment, the learning methods were all *on policy*. For example, in temporal-difference learning, the subsequent actions were chosen using the value function being learned. However, a more widely used approach in the literature is an *off policy* method called Q-learning[46, 47]. In Q-learning the action-value estimate becomes $Q^\pi(s_t, a_t) = \mathbb{E}^\pi\big[r_t + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1})\big]$. The advantage of this approach is that it

provides an elegant way to directly estimate the optimal policy $Q*$, which is the target policy we are interested in learning while acting according to a possibly different behavioral policy, e.g., ϵ-greedy.

Q-learning and the methods we have considered so far are adequate to solve MDPs, but they are restricted to small problems because they require a table of all state-action pairs. Therefore, these methods suffer from the curse of dimensionality, which means that they cannot scale to high dimensional state spaces because of prohibitive memory requirements. In order to tackle this, tabular methods have been replaced with powerful function approximators such as deep neural networks, which allow methods like Q-learning to scale to high dimensional state spaces. However, using non-linear function approximators like neural networks can destabilize Q-learning due to overestimating Q-values. This overestimation explodes when bootstrapping is used with function approximators. One technique to tackle this problem called *experience replay* was introduced in [23]. This involves storing transitions and rewards an agent has experienced in a buffer and replaying samples from this buffer to train the agent. More recently, Deep Q-Network [25] was introduced, where it was demonstrated how a combination of experience replay and a slowly changing copy of the function approximator called a *target network* can be used to stabilize the training of a reinforcement learning agent with deep neural networks and Q-learning.

— 3 —
Related work

We discuss related work by giving an overview of the literature on episodic control and transfer in reinforcement learning.

3.1 Episodic memory and episodic control

Episodic memory[44] is a model from the field of psychology, which refers to an autobiographical kind of memory about one's personal experiences. For example, this kind of memory is related to recalling and linking different aspects of particular events, such as recalling one's first beer drinking experience. This contrasts with a different memory model called semantic memory, which is concerned with knowledge of general facts. Episodic memory is believed to be a more recently evolved memory system on top of semantic memory and also seems to be unique to humans, unlike semantic memory found in other species [43].

Lengyel & Dayan [22] argued for the inclusion of a third mechanism for reinforcement learning in addition to the well-established model-free and model-based paradigms. This mechanism, termed episodic control, was based on the utilization of episodic memory for reinforcement learning by replaying sequences of actions based on stored episodic memory of previous experiences. It was argued that this would be particularly useful in low data regimes by allowing agents to latch on to good strategies quickly once experienced. Similar arguments have also been made by Gershman et. al [14] where they argued for a non-parametric method based on episodic reinforcement learning to model sample efficiency displayed in humans and animals in low-data regimes.

Subsequently, a simple computational model of this kind of learning, called model-free episodic control(MFEC), was investigated in [9]. They instantiated a non-parametric model, which utilized a key-value store Q^{EC} to represent the state-action value function shown in Equation 3.1. The two cases here show the strategy employed by the agent while learning, depending on a state-action pair encountered. If an instance of a state-action pair already exists in memory, it is updated by taking the maximum of the new return R_t associated with this state-action pair and its previous value in memory.

$$Q^{\mathrm{EC}}(s_t, a_t) \leftarrow \begin{cases} R_t & \text{if } (s_t, a_t) \notin Q^{\mathrm{EC}}, \\ \max\left\{Q^{\mathrm{EC}}(s_t, a_t), R_t\right\} & \text{otherwise,} \end{cases}. \tag{3.1}$$

Furthermore, at action selection time, an MFEC agent might encounter states and actions for which it has no previously stored values in Q^{EC}. In order to handle this case, the agent estimates a state-action value using a number k of its nearest neighbours. Equation 3.2 shows how state-action values are estimated when selecting actions both for state-action pairs in memory as well as those not previously existing in memory.

$$\widehat{Q^{\text{EC}}}(s,a) = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} Q^{\text{EC}}(s_i,a) & \text{if } (s,a) \notin Q^{\text{EC}}, \\ Q^{\text{EC}}(s,a) & \text{otherwise,} \end{cases} . \tag{3.2}$$

where s_i, $i = 1, \ldots, k$ are the the k-nearest neighbouring states

In this episodic memory model, the keys used to index memory correspond to a compact state representation derived from environmental observations using random projections to a lower-dimensional space or a pre-trained variational auto-encoder represented as $s_t = \phi(o_t)$. Furthermore, the values stored in memory correspond to Monte-Carlo Q-value estimates. Learning in this model proceeds by making updates to the episodic memory at the end of each episode, associating each encountered sate with the total discounted reward experienced in that episode. The algorithm for this method is shown in Algorithm 1.

Algorithm 1 Model-Free Episodic Control [9]

for each episode **do**
  for $t = 1, 2, 3, \ldots, T$ **do**
    Receive observation o_t from environment.
    Let $s_t = \phi(o_t)$.
    Estimate return $\widehat{Q^{\text{EC}}}$ for each action a via (3.2)
    Let $a_t = \arg\max_a \widehat{Q^{\text{EC}}}(s_t, a)$
    Take action a_t, receive reward r_{t+1}
  end for
  for $t = T, T-1, \ldots, 1$ **do**
    Update $Q^{\text{EC}}(s_t, a_t)$ using R_t according to (3.1).
  end for
end for

Neural episodic control

In a bid to improve the MFEC architecture to allow for an end-to-end differentiable learning architecture, as well as shaping state representations that adapt to the current task being learned, Neural Episodic Control [32] was proposed. A convolutional neural network was used to obtain the compact representation of states used as keys in memory. Additionally, a memory structure called a *differentiable neural dictionary* (DND) was introduced, which, put together with a convolutional neural network for obtaining keys, gives rise to an end-to-end differentiable model illustrated in Figure 3.1. In NEC, there is one DND per action, which is a growing table M_a of a pair of dynamically growing arrays (K_a, V_a) of keys and values. Furthermore, a lookup is performed with the DND for a query key h to obtain the corresponding output o using the following equation.

$$o = \sum_i w_i v_i, \tag{3.3}$$

where v_i corresponds to values stored in V_a and w_i are weights corresponding to the result of a normalized kernel k between the query key h and keys h_i in K_a as follows

$$w_i = k(h, h_i) / \sum_j k(h, h_j). \tag{3.4}$$

Two techniques were employed to enable the scalability of this model. First, the number of elements involved in lookups was limited to the top 50 nearest neighbours, efficiently found using a k-d tree. Second, the size of the DNDs was kept limited by removing the least recently used items.

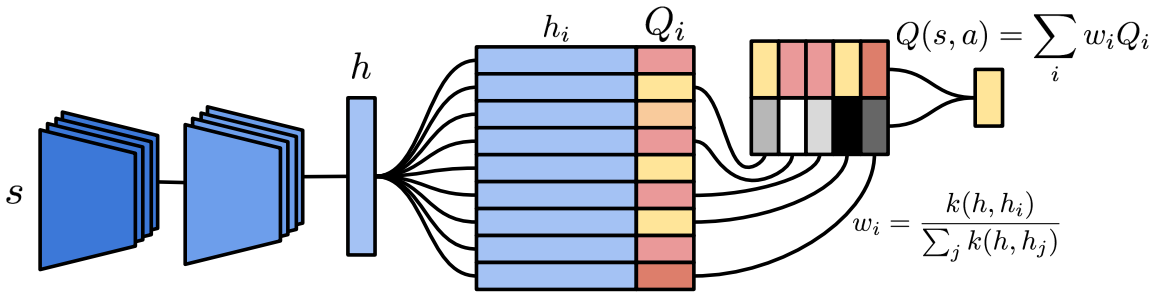

Figure 3.1 – NEC architecture from [32]

Furthermore, the values stored in memory were changed from Monte Carlo estimates used in MFEC to N-step Q-value estimates:

$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a'). \tag{3.5}$$

In contrast to MFEC, updates to keys already found in the DND memory store while learning were done using Q-learning as $Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s,a) - Q_i)$. The NEC algorithm introduced therein is given in Algorithm 2.

Algorithm 2 Neural Episodic Control [32]

$\mathcal{D}$: replay memory.
M_a: a DND for each action a.
N: horizon for N-step Q estimate.
for each episode **do**
  for $t = 1, 2, \ldots, T$ **do**
    Receive observation s_t from environment with embedding h.
    Estimate $Q(s_t, a)$ for each action a from M_a via (3.3)
    $a_t \leftarrow \epsilon$-greedy policy based on $Q(s_t, a)$
    Take action a_t, receive reward r_{t+1}
    Computer $Q^{(N)}(s_t, a_t)$ via (3.5)
    Append $(h, Q^{(N)}(s_t, a_t))$ to M_{a_t}.
    Append $(s_t, a_t, Q^{(N)}(s_t, a_t))$ to $\mathcal{D}$.
    Train on a random minibatch from $\mathcal{D}$.
  end for
end for

Further to this, several improvements and extensions have been proposed, which we highlight next. [28] proposed a method termed NEC2DQN in which NEC is used at the early stages of learning before making a transition to DQN at later stages. This was done to benefit from both the advantages of sample efficient NEC in a low-data regime at the beginning of training and the powerful generalization capability of DQN when data is plentiful in later stages of training. In [24] episodic memory deep q network was proposed, an architecture that augments DQN with an episodic memory-based estimate. They found that combining this with a TD estimate improved sample efficiency and long-term performance over baseline DQN and NEC agents.
[36] investigated adding principled exploration to NEC by combining episodic control with maximum entropy mellowmax policy. [1] on the other hand, proposed using dynamic online k-means to improve the memory efficiency of NEC. Likewise, [48] proposed to further optimize the usage of the contents of the episodic memory store by considering the relationship between contents of episodic memory. Finally, [16] recently introduced *Generalizable Episodic Memory* which extends the applicability of episodic control to continuous action domains.

In this study, we will rely on the basic NEC model as proposed in [32] as the backbone for developing our model.

3.2 Transfer learning in reinforcement learning

Transfer learning [40, 20] refers to methods developed to allow knowledge learned from one or several tasks referred to source tasks to be reused when faced with new tasks referred to as target tasks. In reinforcement learning, these tasks are defined by MDPs, and in order for transfer to occur between two or more MDPs, some shared structure must exist between some components of the MDPs. A survey on transfer in deep reinforcement learning [49] highlights different kinds of transfer in reinforcement learning based on what

knowledge is transferred. Namely these are *reward shaping*, *learning from demonstrations*, *policy transfer*,*inter-task mapping*, and *representation transfer*.

For the work in this study, we are interested in the form of transfer enabled by a framework introduced in [4] termed successor features and generalized policy improvement (SF&GPI). This approach belongs to both classes of policy transfer and representation transfer.

Successor representations and successor features

Successor representation [12] refers to a state representation using an expected occupancy measure of future states when following a policy π defined as:

$$\mu^{\pi}(s, s') = \mathbb{E}^{\pi}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} \mathbb{1}[s_i = s'] | s_t = s\Big], \tag{3.6}$$

where $\mathbb{1}$ is an indicator function.

With successor representations, the value function of a state can be represented as a linear combination of the successor representation of all feasible successor states and their associated rewards written as:

$$V^{\pi}(s) = \sum_{s'} \mu^{\pi}(s, s') R(s'). \tag{3.7}$$

However, a problem with successor representations is that it does not work in high dimensional state spaces, given it requires a tabular representation of states, nor in continuous spaces. Therefore in order to deal with these shortcomings, successor features [4] which extend successor representations to continuous state and action spaces were introduced.

Successor features, like successor representations, are based on the idea of learning a value function representation that decouples rewards from environment dynamics. A fundamental assumption behind this approach is that a useful and common scenario for transfer is when tasks, defined by MDPs, only differ in their reward function. The first step towards defining an extension of Equation 3.7 for a state-action value function as outlined in [4] is to suppose the one-step expected reward associated with a transition (s, a, s') writes:

$$r(s, a, s') = \boldsymbol{\phi}(s, a, s')^{\top} \mathbf{w}. \tag{3.8}$$

Now, we can rewrite Q-values as follows:

$$\begin{aligned} Q^{\pi}(s, a) &\equiv \mathbb{E}^{\pi}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}^{\pi}[r_{t+1} + \gamma r_{t+2} + ... | S_t = s, A_t = a] \\ &= \mathbb{E}^{\pi}[\boldsymbol{\phi}_{t+1}^{\top} \mathbf{w} + \gamma \boldsymbol{\phi}_{t+2}^{\top} \mathbf{w} + ... | S_t = s, A_t = a] \\ &= \mathbb{E}^{\pi}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} \phi_{i+1} | S_t = s, A_t = a\Big]^{T} \mathbf{w} \\ &= \boldsymbol{\psi}^{\pi}(s, a)^{T} \mathbf{w}, \end{aligned} \tag{3.9}$$

where $\boldsymbol{\psi}^{\pi}(s, a)$ are known as the successor features (SFs) of (s, a) under policy π.

Also, SFs satisfy a Bellman equation:

$$\boldsymbol{\psi}(s,a) = \boldsymbol{\phi}_{t+1} + \gamma \mathbb{E}^{\pi}[\boldsymbol{\psi}^{\pi}(S_{t+1}, \pi(S_{t+1}))|S_t = s, A_t = a], \tag{3.10}$$

and thus can be learned using RL methods such as temporal difference methods.

Generalized policy improvement

Generalized policy improvement(GPI) is an operation which generalizes policy improvement in classic dynamic programming to multiple Q functions. Given a set of value functions Q_i's, a GPI policy writes as: $\pi(s) \in \text{argmax}_a \max_i Q_i(s,a)$. The decomposition provided by Equation 3.9 allows the transfer of a value function that summarizes the environment dynamics induced by a given policy decoupled from a reward function. Once the successor features for a policy are learned for one task, evaluating the value function for the policy on a different task is straightforward once $\mathbf{w}$ is given or estimated. This property enables the efficient implementation of generalized policy improvement. Based on this, an algorithm for transfer using successor features and generalized policy improvement was proposed in [4] called SFQL. Additionally, a theoretical guarantee on the performance of the GPI policy was stated in the following theorem

Theorem 1. (Generalized Policy Improvement)[4]
Let π_1, π_2, ..., π_n be n decision policies and let $\tilde{Q}^{\pi_1}$, $\tilde{Q}^{\pi_2}$, ..., $\tilde{Q}^{\pi_n}$ be approximations of their respective action-value functions such that

$$|Q^{\pi_i}(s,a) - \tilde{Q}^{\pi_i}(s,a)| \leq \epsilon \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}, \text{ and } i \in \{1,2,...,n\}. \tag{3.11}$$

Define

$$\pi(s) \in \operatorname*{argmax}_a \max_i \tilde{Q}^{\pi_i}(s,a). \tag{3.12}$$

Then,

$$Q^{\pi}(s,a) \geq \max_i Q^{\pi_i}(s,a) - \frac{2}{1-\gamma}\epsilon \tag{3.13}$$

for any $s \in \mathcal{S}$ and $a \in \mathcal{A}$, where Q^{π} is the action-value function of π.

Next, we mention a few extensions to these ideas found in the literature. As a follow up to [4], a relaxation of the condition that reward functions be expressed as in Equation 3.8, as well as a demonstration of how to combine deep neural networks with SF&GPI in a stable manner, was introduced in [3] with corresponding theorems based on this relaxation. Going even further, in [5], a generalization of policy evaluation was introduced, called generalized policy evaluation, in addition to generalized policy improvement to speed up solving reinforcement learning tasks.

The SF&GPI framework and SFQL algorithm as introduced in [4] will be used as the backbone of the transfer component in this study.

— 4 —
Method

The central idea proposed and investigated in this study is an architecture that combines neural episodic control with successor features (and generalized policy improvement) which we call Successor Feature Neural Episodic Control (SFNEC). We hypothesize that this would bring advantages from both approaches into fusion, namely improved data efficiency and transfer learning, leading to a better algorithm in terms of learning speed and flexible reuse of previously gained knowledge in new tasks. We proceed by giving an overview of our proposed model, SFNEC, followed by agents we used as baselines. Finally, we organize discussing implementation details of all agents according to the two settings under which they will be evaluated: learning a single task and transferring learning across multiple tasks.

4.1 SFNEC model

We designed our SFNEC model by extending NEC to deal with vector-valued successor features in place of scalar valued action-values. Our model follows very closely the NEC architecture with the following modifications. We use an NEC-like algorithm to learn successor features. To do this, first we rely on the assumption outlined in [4] that the reward function can be decomposed into an inner product: $r(s, a, s') = \boldsymbol{\phi}(s, a, s')^\top \mathbf{w}$. This decomposition leads to the definition of $\boldsymbol{\psi}(s, a)$, called successor features (SFs) which also follows a Bellman equation and thus can be learned with conventional reinforcement learning methods as explained in Section 3.2. With this observation, we re-purpose the NEC algorithm for our model to learn N-step $\boldsymbol{\psi}$-values:

$$\boldsymbol{\psi}^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j \boldsymbol{\phi}_{t+j} + \gamma^N \max_{a'} \boldsymbol{\psi}(s_{t+N}, a') \tag{4.1}$$

in place of N-step action-values $Q^{(N)}$ which were used as the values stored in memory.

We highlight two things about Equation 4.1. First, $\boldsymbol{\psi}^{(N)}$ has a dependence on the current behavioural policy π of an agent, which we have dropped for notational convenience. Second, we are actually interested in learning the *expected* $\boldsymbol{\psi}$-values, but the equation shows how we estimate this in practice by using samples, similar to how the expected $Q^{(N)}$-values are estimated in NEC.

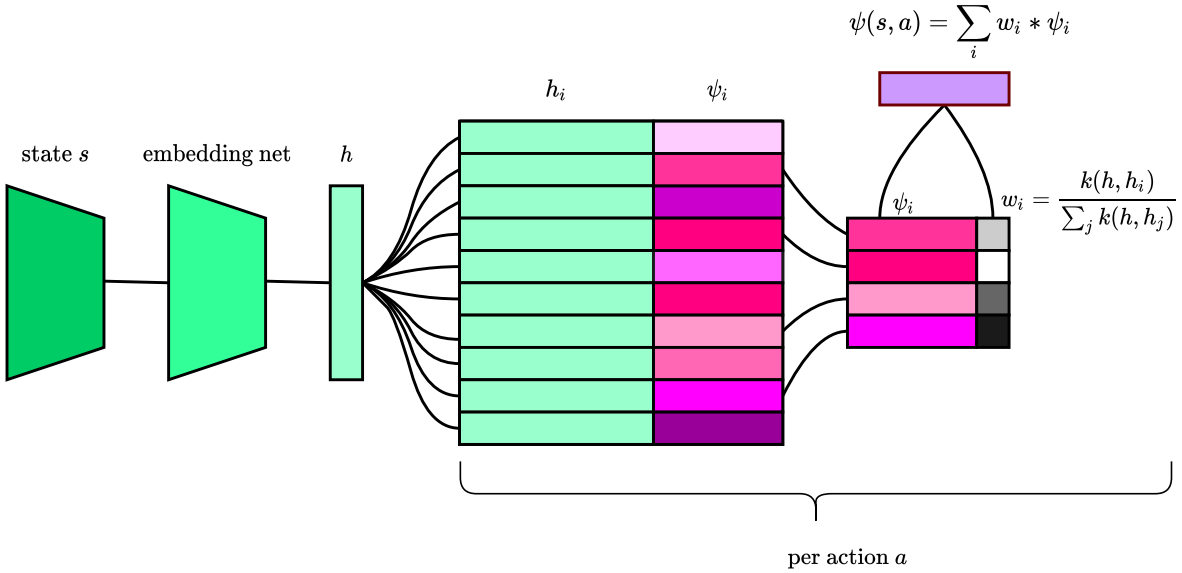

Figure 4.1 – SFNEC model. This architecture is similar to the NEC architecture shown in Figure 3.1 with the major difference being the storage of ψ-values instead of Q-values in memory.

To perform a lookup using the SFNEC model, we use the following equation:

$$\boldsymbol{\psi}(s_t, a) = \sum_i \frac{k(h(s_t), h_i)}{\sum_j k(h(s_t), h_j)} * \boldsymbol{\psi}_i \tag{4.2}$$

where $\boldsymbol{\psi}_i$ corresponds to a previously stored $\boldsymbol{\psi}_i$-value for a key h_i, and k is the kernel used to compute a similarity score between the query key h and keys in memory h_i. We note that similar to the lookups done in NEC, we limit the elements in memory used to a number of nearest neighbours e.g. 50. Likewise, we also used the inverse distance kernel used in [4] which writes:

$$k(h, h_i) = \frac{1}{\|h - h_i\|_2^2 + \delta} \tag{4.3}$$

We show the architecture of our SFNEC model in Figure 4.1 and defer the details of the algorithm for training a reinforcement learning agent using this model to Section 4.3.

4.2 Single task learning

NEC agent: We chose to implement an agent following the Neural Episodic Control architecture and algorithm outlined in Section 3.1 as the baseline for learning speed on a single task, henceforth referred to as NEC agent. We relied on a publicly available implementation provided by Kai Arulkumaran on GitHub[1] as an inspiration for the implementation of our NEC agent. For our purposes, we modified our implementation to allow memory updates to occur either immediately after a horizon of N steps or batching updates at the end of episodes. This is different from the original description and the reference implementation, which suggest batching memory updates at the end of episodes.

[1] `https://github.com/Kaixhin/EC`

Our primary motivation for this modification is that we would like our method to be applicable even in learning scenarios that do not divide into episodes, i.e., continuing tasks. Also, we suspected that this end-of-episode update method would not work as well in tasks where it might take a while to reach the end of an episode. We study the effect of this choice experimentally in Chapter 5.

We recall that estimating action values with NEC involves combining the values for previously-stored keys in memory similar to a query key representing the agent's state. Thus, it is necessary to perform an efficient similarity search. In [32] it was suggested to enable this efficiency by performing approximate searches using a k-d tree [8]. Contrary to this, to keep our initial implementation simple while laying out the general idea of our framework, we utilize brute-force searches for all agents that need to perform a similarity search in stored memory(NEC and SFNEC). For this, we used the Facebook AI similarity search library[2] [17]. We chose to use this library because it is open-source, simple to use, and well-optimized for brute-force and approximate nearest neighbour searches.

4.3 Transfer learning

In this section, we are concerned with the more relevant setting for our study in which a reinforcement learning agent can transfer knowledge across tasks.

SFQL agent: As our baseline agent for transfer learning, we used an implementation of SF&GPI according to the pseudocode provided by Barreto et. al in [4] which we call SFQL agent. Chris Reinke provided this implementation in his transfer_rl library[3] in addition to other software modules used for running the experiments in this study.

SFNEC agent: To implement our SFNEC model in a reinforcement learning agent, we designed the algorithm given in Algorithm 3 based on both the SFQL algorithm in [4] and the NEC algorithm in [32]. Henceforth, the agent implemented according to the SFNEC model and algorithm is called the SFNEC agent. We now discuss essential aspects of the algorithm. Firstly, we note that we allow for both cases when reward descriptions $\mathbf{w}_i$ for each task are provided or not with the boolean condition learn_w. Furthermore, similar to the NEC agent described in Section 4.2, our SFNEC agent implementation included the option to perform the memory updates in lines 19-20 either immediately after N steps or at the end of episodes. Additionally, we note that we can use the algorithm without GPI by simply making j equal i in line 8 similar to the SFQL algorithm described in [4]. All these options allow us to study the model under different setups in our experiments.

Finally, we highlight that we also update policies used for GPI action selection in lines 22-25 as done in the SFQL algorithm. The essence of this update is to continually refine the successor features of policies that remain pertinent for GPI action selection in line 8. A crucial difference for updating these policies is that we cannot obtain the features $\boldsymbol{\phi}$ needed to compute N-step $\boldsymbol{\psi}$ estimates as the agent is acting according to a different policy at this update point. Thus, we are constrained to utilizing a single-step *off-policy* update.

[2]https://github.com/facebookresearch/faiss

[3]`https://gitlab.inria.fr/creinke/transfer_rl`

Algorithm 3 Successor Feature Neural Episodic Control (SFNEC)

	α_w	learning rate for $\mathbf{w}$
	learn_w	condition to indicate if to learn $\mathbf{w}$ for each task
	$\boldsymbol{\phi}$	features of state transitions
	$\mathbf{w}_i$	optionally given for each task i
Require:	N	n-step horizon for $\psi^{(N)}$ estimates
	$\mathcal{D}_i$	replay buffer of (h, a, $\psi^{(N)}$) tuples for each task i
	M_{ai}	a DND for each action a per task i
	num_tasks	number of tasks to be learned

for $i = 1, \ldots,$ num_tasks **do**
  if learn_w **then**
    Initialize $\mathbf{w}_i$ with small random values
  end if
  for each episode **do**
    for $t = 1, \ldots, T$ **do**
      Get observation s_t from the environment and its embedding h_t
      $j \leftarrow \text{argmax}_{k \in \{1,\ldots,i\}} \max_b \boldsymbol{\psi}_k(s_t, b)^\top \mathbf{w}_i$
        // j is the index of policy selected according to GPI
      if $rand[0, 1) < \epsilon$ **then**
        $a_t \leftarrow$ select an action uniformly at random
      else
        $a_t \leftarrow \text{argmax}_b \boldsymbol{\psi}_j(s_t, b)^\top \mathbf{w}_i$
      end if // ϵ-greedy action selection
      Take action a_t and observe reward r_t, and observation s_{t+1}

      if learn_w **then**
        $\mathbf{w}_i \leftarrow \mathbf{w}_i + \alpha_w[r - \boldsymbol{\phi}(s_t, a_t, s_{t+1})^\top \mathbf{w}_i]$ // learn $\mathbf{w}$ for task i
      end if
      Compute $\boldsymbol{\psi}^{(N)(s_t,a_t)}$ using eqn. 4.1
      Append $(h_t, \boldsymbol{\psi}^{(N)}(s_t, a_t))$ to $M_{a_t i}$
      Append $(s_t, a_t, \boldsymbol{\psi}^{(N)}(s_t, a_t))$ to $\mathcal{D}_i$
      Train on a random minibatch from $\mathcal{D}_i$

      if $j \neq i$ **then**
        $a' = \text{argmax}_b \boldsymbol{\psi}_j(s_{t+1}, b)^\top \mathbf{w}_j$
        update $\boldsymbol{\psi}_j(s_t, a_t)$ using the one-step TD target: $\boldsymbol{\phi}(s_t, a_t, s_{t+1}) + \gamma \boldsymbol{\psi}_j(s_{t+1}, a')$
      end if
    end for
  end for
end for

— 5 —
Evaluation

We evaluated our proposed model and other agents presented in Chapter 4 using the experiments detailed in this chapter. Our organization here also follows a similar format where we have a section dedicated to learning a single task and another section dedicated to transfer learning.

5.1 Single task evaluation

Here we started by evaluating two aspects of our NEC agent implementation. First, we aimed to observe the effect of the two modes of updating memory we allowed. We recall that these correspond to updating memory either at the end of episodes of an agent's interaction with its environment or immediately after N steps, the horizon length needed to compute the N-step action-value estimates, in the environment. Secondly, it was reported in [4] that they found using N-step estimates to perform better than Monte-Carlo estimates, which were used to layout the initial episodic reinforcement learning framework in [9]. Thus, we wanted to observe how the NEC agent's performance varies depending on the horizon length N. We ran these experiments using an implementation of the CartPole[6] environment in OpenAI gym[1].

CartPole

The CartPole, illustrated in Figure 5.1, is a classical control task in which the goal of an agent or a controller is to balance a pole attached to a cart in an upright position. When this task is set up as a reinforcement learning problem, an agent receives a reward of $+1$ for each time step it manages to keep the pole upright within a specified orientation. The task is considered solved when an agent can obtain an average of at least 195 across 100 consecutive episodes. A new episode starts with the cart and pole positioned in the middle and ends in one of three situations. The first is when the cart goes beyond the track's boundaries, the second is when the pole deviates beyond the allowed orientation, and the third is after a maximum of 500 steps. For the agent, the state is a two-dimensional vector containing the speed and orientation of the pole, while it has two discrete actions allowed, which are either to move left or right.

[1]OpenAI gym is an open-source library for reinforcement learning problems

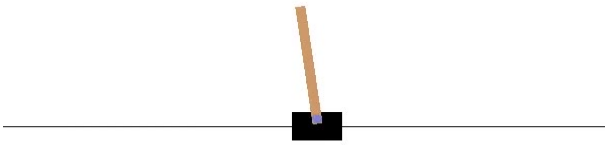

Figure 5.1 – CartPole task from OpenAI gym. The goal of a reinforcement learning agent here is to keep the pole upright by choosing between two actions (left or right) at each time step.

Setup

Next, we discuss our experimental setup for NEC agent in the CartPole environment. We trained the NEC agent following the NEC algorithm as outlined in Algorithm 2. For the embedding network used to obtain keys stored in the differentiable neural dictionary(DND) memory module, we used a single layer fully connected neural network which took the agent's state, a vector of dimension 2, as input and produced an output vector of dimension 4. Thus, our key size for each DND, one per action, was 4. We set the capacity of the replay buffer to $100,000$ while the DNDs each had a capacity of $15,000$. We used 50 nearest neighbours to estimate the Q-values and set the DND learning rate for updating exact matches during training to 0.3. For training, the agent followed an ϵ-greedy exploration strategy using $\epsilon = 0.1$, and the entire network was optimized by gradient descent using an RMSprop optimizer with a learning rate of 0.01. Finally, for varying horizon lengths of N we tested from the set $\{1, 2, 4, 8, 16, 32, 64, 100\}$.

Results

The results shown for these experiments on CartPole correspond to a running average of the reward per episode for the 50 most recent episodes. This was done over 10 random seeds and shaded areas around the solid lines corresponding to the standard error. First, Figure 5.2 shows the results of the two memory update modes. We reckon the difference is minimal because of the combination of a dense reward signal gotten in the Cartpole task and the easy reachability of the end of episodes. We presume that in tasks where reaching the end of an episode might require significant exploration, the update method that waits till the end of episodes will slow down learning performance.

However, the major takeaway is that we have implemented a flexible method, which allows learning to commence as soon as N steps are taken in the environment without dependence on reaching episode ends. We believe this should work better in the case mentioned above, and from our results, it also performs well in more straightforward settings like CartPole.

Next, we have the results of varying the horizon length in Figure 5.3. Note that setting values for N corresponds to setting a tradeoff between Monte Carlo estimates at $N =$ episode length and one-step off policy estimates at $N = 1$. As expected, the NEC does

not perform well when using a single step because this corresponds to keeping no actual episodic memory from an agent's own experience. Instead, it boils down to performing standard Q-learning but using a non-parametric approximator based on nearest neighbours. We observe that the best performance is somewhere around $N = 16$, improving until then and dropping afterward. Note that moving higher corresponds to shifting the tradeoff towards Monte-Carlo estimates, and we notice that the performance starts dropping off. This is consistent with the reported finding in NEC[32] that using N-step estimates performed better than Monte-Carlo estimates originally used in MFEC[9].

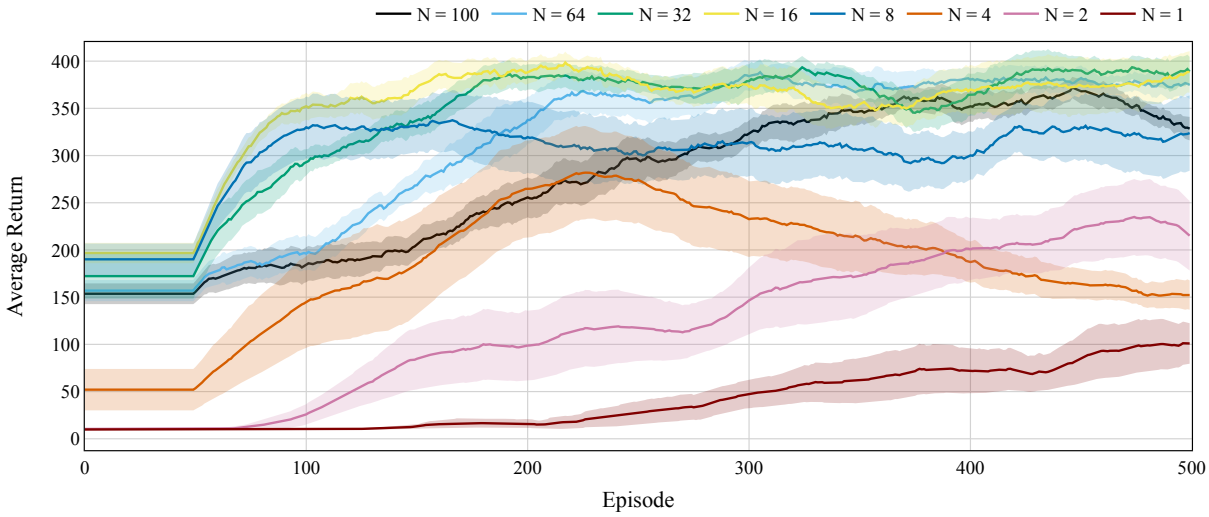

Figure 5.3 – NEC agent performance on CartPole according to the length N-step horizons. The optimal value for N is between single-step and Monte-Carlo estimates.

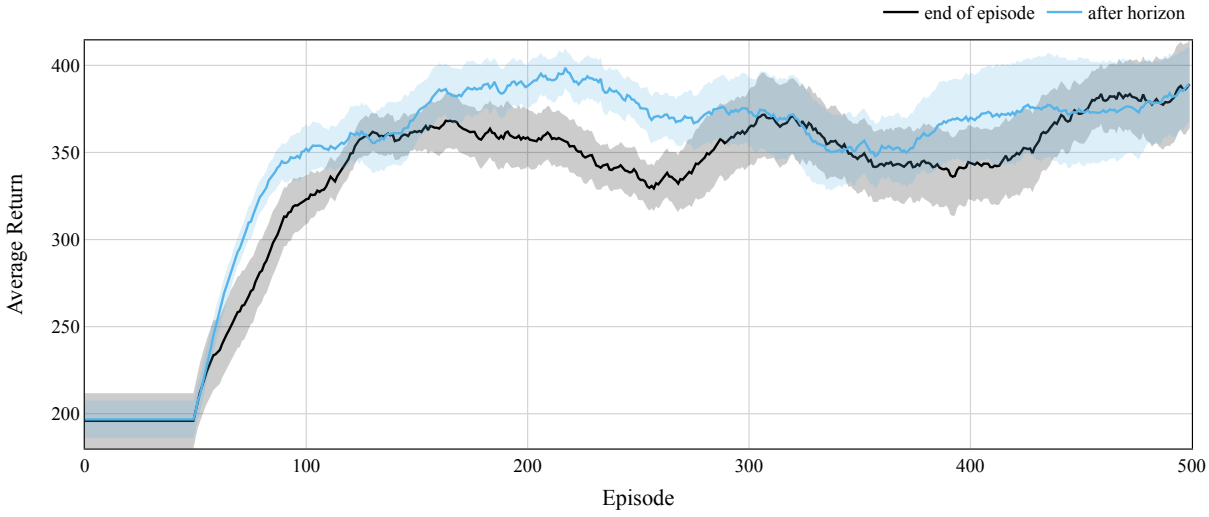

Figure 5.2 – NEC agent performance on CartPole according to memory update modes. Performance is similar between both modes, with updating after N-steps being slightly better than batching memory updates at the end of episodes.

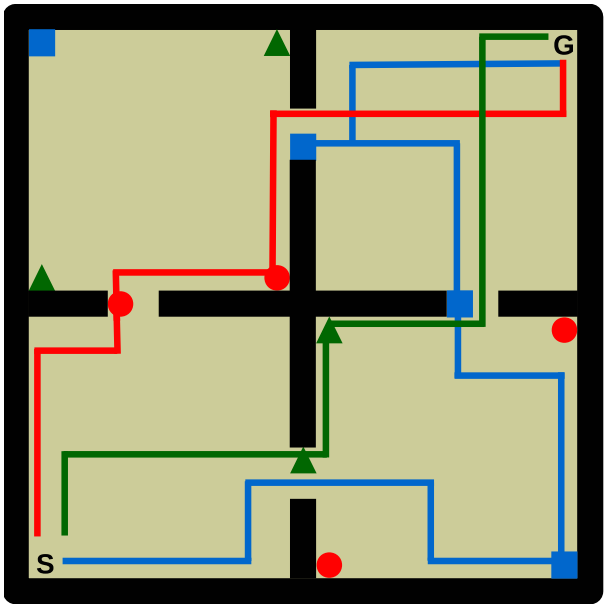

Figure 5.4 – 2-D four room environment proposed in [4]. An episode starts with an agent at position 'S' and ends when it reaches the goal position 'G'. The different shapes indicate classes of objects associated with different rewards.

Moving forward, as a first evaluation step for our proposed SFNEC agent, we decided to run an experiment on a single task in a simplified version of the environment proposed by Barretto et. al in [4] for evaluating transfer learning which we describe in the next section. This was also advantageous in setting up a smooth transition from evaluating our model in a single task setting to a transfer learning setting.

Four-room environment

Barreto et. al [4] proposed using a family of navigation tasks to evaluate the transfer performance of their SFQL agent with a two-dimensional four-room environment. The environment, shown in Figure 5.4, consists of four rooms with a start location in the bottom left room and a goal location in the top right room. Within the rooms, there are multiple objects belonging to different classes. Each class is associated with a reward, and the instantiation of rewards per class defines a task within this environment. The goal of an agent is to navigate from the start position to the goal position while picking up objects associated with positive rewards and avoiding objects with negative rewards. Objects once picked up disappear from the environment and reappear at the beginning of a new episode. Thus, to demonstrate good performance, an agent faces a series of tasks, each being a different instantiation of rewards with the aim of maximizing the sum of rewards accumulated by the agent. In general, when we use this environment, we follow the same setup for this environment as in [4] for our experiments, with essential details recapitulated below. There are twelve objects in the environment and three object classes (four objects per class). The rewards associated with each class changes after $20,000$ transitions, and they are sampled uniformly at random from $[-1, 1]$ while reaching the goal always gave a reward of $+1$. The agent's observations provided from the environment consists of two parts. The first part is the activations of the agent's (x, y) position on a

10×10 grid of radial basis functions over the entire four rooms. The second part consists of object detectors indicating the presence or absence of objects in the environment.

For our experiments with this environment, we provide both the state features $\boldsymbol{\phi}(s_t, a_t, s_{t+1})$ and reward weight vector $\mathbf{w}$. The state features are boolean vectors containing elements indicating whether the agent is over an object present in the environment or over the goal position. The reward weight vector contains rewards associated with picking up an object and reaching the goal position. We note that providing both elements to the agent means the reward function is fully specified to the agent according to the decomposition: $r(s_t, a_t, s_{t+1}) = \boldsymbol{\phi}(s_t, a_t, s_{t+1})^\top \mathbf{w}$. Nonetheless, it is possible to approximate these quantities as shown in [4]. We refer the reader to Appendix B in [4] for further details on this environment.

When evaluating our NEC and SFNEC agents, we utilize two different methods for obtaining the keys used as the compact state representation in memory. The first method directly uses the observation from the environment as the key, which we refer to as the identity embedding. Meanwhile, the keys were the output of a single layer fully connected neural network for the second method, which we refer to as learned embedding. Additionally, we set up a training method for NEC and SFNEC where we only keep a single sample in the replay buffer of these agents. Training proceeds by using each sample sequentially as they are collected in the environment. We did this to allow a more direct comparison to SFQL, which uses a "true" stochastic gradient descent method for its optimization, i.e. gradients are estimated using single samples rather than batches of multiple samples. We will refer to this as a single-sample training method in contrast to a batch training method that trains on randomly sampled batches of a large replay buffer.

Single task in simplified room environment

We continued our single task evaluation experiments by deriving a simplified version of the four-room environment. Our simplified environment corresponds to taking down the walls between the rooms in the original environment shown in Figure 5.4 and placing two objects of different classes at the top left and bottom right corners. The top left object had a reward of -1 while the bottom eight object had $+1$. In this case, we limit the total number of steps allowed to $10,000$, and the agent's goal is to accumulate as much reward as possible within this period by learning to pick up the good object while avoiding the bad object on its way to the goal location. Now we give details on hyperparameters. We set the key size when learning an embedding to 4, the DND capacity to 1000, N to 8, replay buffer size to 1 or 5000, and batch size to 1 or 16 depending on if we are in a single-sample or batch training method respectively.

Results

We show the results of our experiments in the simplified four-room environment in Figures 5.5-5.6, split according to the training method. In both figures, we include the SFQL agent as a baseline for comparison.

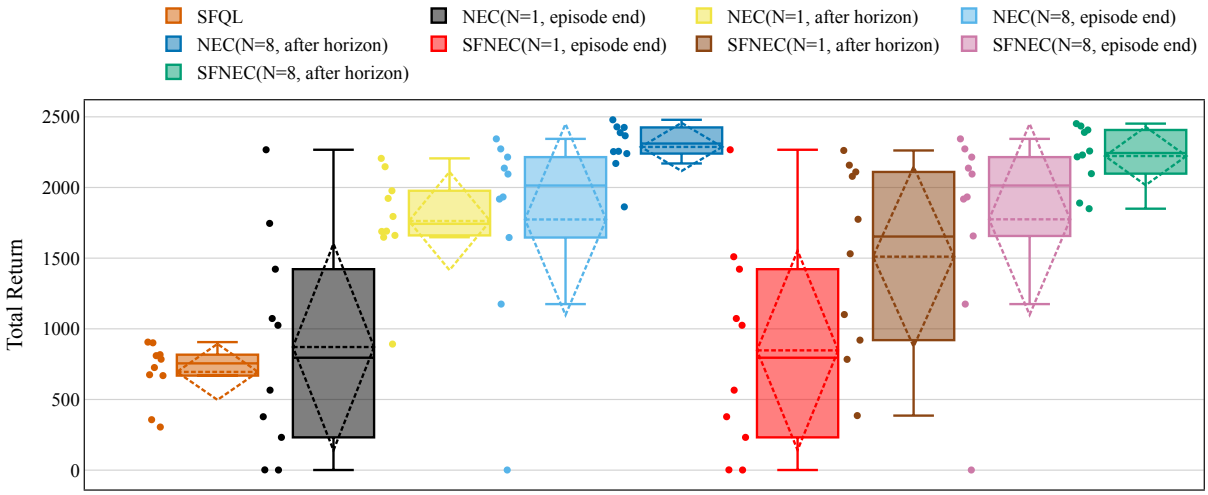

Figure 5.5 – Total return of agents when using the single-sample training method and an identity embedding in the single room environment. NEC and SFNEC outperform the SFQL baseline. Returns are shown for 10 runs of each algorithm.

In Figure 5.5, we have the results for the configurations where we use the single-sample training method. We notice here that our NEC and SFNEC agents outperform the SFQL agent as they can learn faster. However, updating at the end of episodes performs poorly for SFNEC and NEC agents when using a horizon length $N = 1$. Although this update method performs better when the horizon length N is increased, it still performs worse than its counterpart, where we update immediately after N steps. We believe this is because the rewards are sparser, i.e. many transitions will yield 0 reward. Likewise, the end of episodes might not occur quickly in this environment, leading to much slower learning.

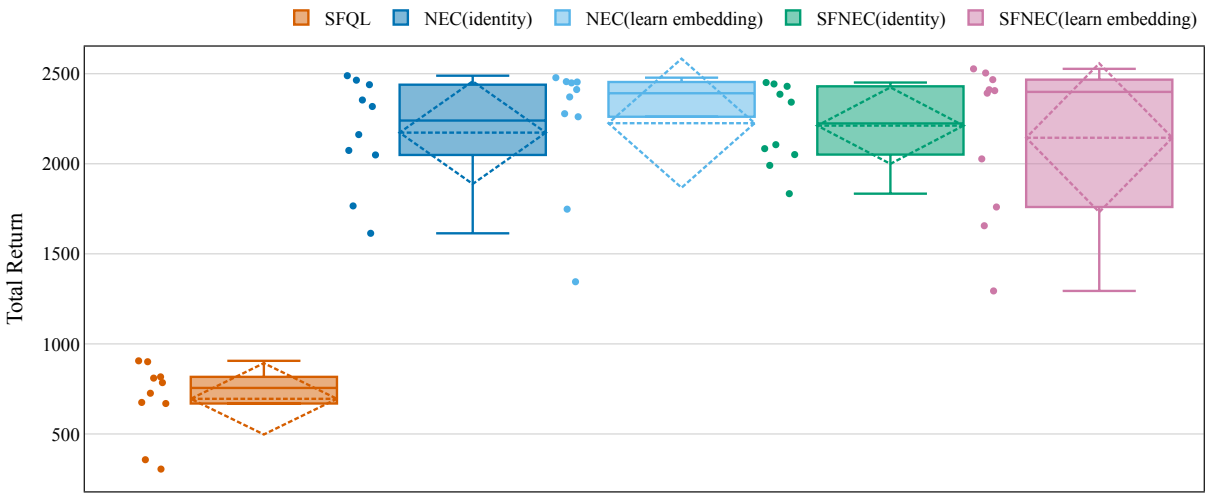

Figure 5.6 – Total return of agents when using the batch training method and a learned embedding in the single room environment. SFNEC and NEC outperform the SFQL baseline. Returns are shown for 10 runs of each algorithm.

Furthermore, in Figure 5.6, we show results for the batch training method. Here, we only used the update after horizon method since it performed better, as discussed above. These results show that using an identity embedding for the keys suffices for this environment, and it performs on par with a learned embedding. This is because the observations provided to the agent in the environment are good enough to support learning. Moreover, the dimension of these observations is not too large, specifically in this case, each observation is vector $\boldsymbol{o_t} \in \mathbb{R}^{103}$, which makes learning a lower-dimensional embedding not strictly necessary.

Overall, we found that using N-step estimates with a single-sample method and an identity embedding performs best for NEC and SFNEC agents.

5.2 Transfer evaluation

We proceed with our experiments in this section by evaluating the SFQL, NEC, and SFNEC in the complete four-room environment as described in Section 5.1. We give additional details about our setup in the next section.

Setup

In this round of experiments, we use 10 tasks to evaluate the performance of our agents. For obtaining our hyperparameters, we carried out a non-exhaustive grid search for the NEC and SFNEC agents. At the same time, we relied on reported values for SFQL agent parameters from [4]. We now report values used for our search and the final values chosen for our experiments. We tested values in $\{0.05, 0.15\}$ for ϵ used for ϵ-greedy exploration. For the learning rate used in network optimization: $\{0.01, 0.05, 0.1\}$, for number of neighbours used in estimating ψ-values: $\{1, 4, 10, 20, 50\}$. Furthermore, for the fast learning rate used to update re-encountered keys in the DND, we tried values in $\{0.1, 0.3, 0.5\}$, and for horizon length N, we used the set: $\{8, 16, 32\}$. We show the best configurations found for each agent in Table 5.1.

Agent	ϵ	Network learning rate	Neighbours	DND learning rate	N
SFQL	0.15	0.01	-	-	-
NEC	0.15	0.01	20	0.1	8
SFNEC	0.15	0.05	20	0.1	8

Table 5.1 – Best parameter configurations found for agents in the four-room environment

Finally, we used the single-sample training method when using an identity embedding and the batch training training method when using a learned embedding.

Results

The results of our experiments in the four room environment are shown in Figures 5.7-5.8. We grouped these results according to embedding used for keys i.e. identity and learned embedding. In our results, we depict the average return over 10 runs of each algorithm.

As shown in Figure 5.7, when using a single-sample training method with an identity embedding, which is most comparable with SFQL, SFNEC with GPI performs best, outperforming NEC and SFQL agents. This is expected because it combines learning speed on each task with the strong transfer conferred by GPI. We note that the NEC agent and SFNEC without GPI already slightly outperform the SFQL baseline even though they do not transfer knowledge across tasks compared to SFQL. These agents can reach such good performance because of the speed of learning benefits due to their usage of episodic control.

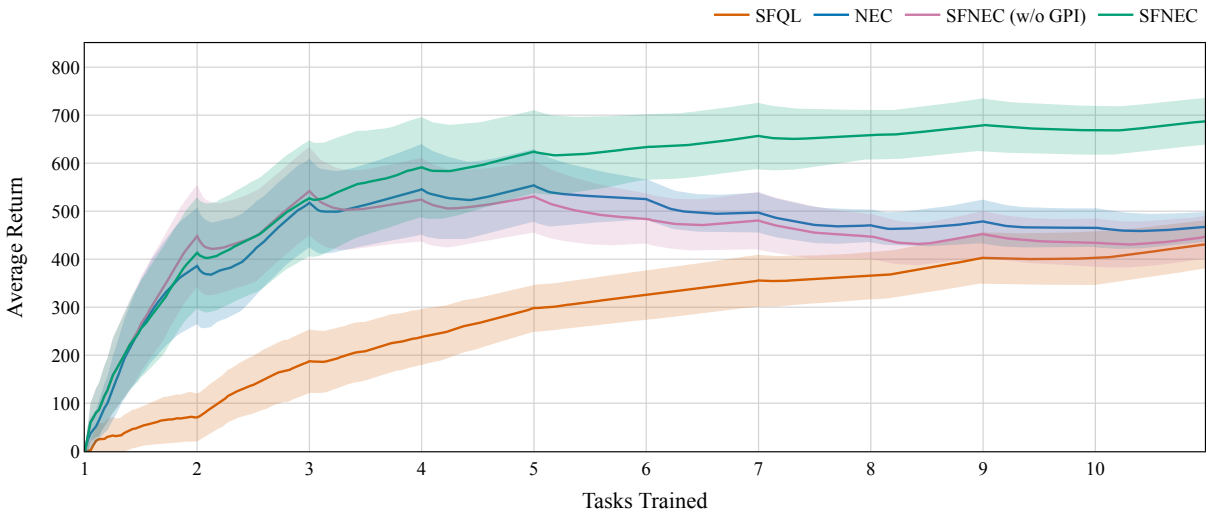

Figure 5.7 – Average return of SFQL, NEC, and SFNEC when using single-sample method on the four-room environment. SFNEC with GPI outperforms other agents when using single-sample method with an identity embedding. Averages are taken over 10 runs with the standard error shown as the shaded region around solid lines.

Figure 5.8 – Average return of SFQL, NEC, and SFNEC when using batch method on the four-room environment. NEC performs best and SFNEC with GPI performs poorly. Averages are taken over 10 runs with the standard error shown as the shaded region around solid lines.

On the other hand, Figure 5.8 shows the results when we use a learned embedding for NEC and SFNEC agents. We observed that once we introduce the embedding layer SFNEC with GPI performs poorly. Nonetheless, SFNEC without GPI performs similarly to NEC, which performs best in this setting. In the next chapter, we discuss further the reasons we believe might cause the poor performance of SFNEC with GPI when it is combined with learning an embedding.

— 6 —
Discussion

This chapter discusses a few more aspects of our results and different experiments run to understand our model. For all the additional experiments run in this chapter, we utilized the SFQL, SFNEC, and NEC agents from Chapter 5. For the SFNEC and NEC agents, we use an identity embedding, horizon length $N = 8$, and the single-sample training method on the four-room environment. These correspond to the best-performing agents for all algorithms.

6.1 How stable is the combination of SF & GPI with NECs

We noted in our experiments that using SFNEC with GPI when learning an embedding does not work well. We hypothesize that this might be due to higher approximation errors being introduced when we learn an embedding. We note a crucial difference between using identity embedding and learned embedding. With identity embedding, we have the advantage that the keys are stable coming directly from environmental observation. This makes learning easier for the agent as it is not burdened with learning another representation layer, as in the learned embedding setup. Logically, we can expect that if the agent cannot learn a good embedding to produce keys in the DND for a particular task, then reusing this on a new task can lead to erroneous predictions about the value of a state-action pair. Essentially, this would mean the approximation error ϵ in Theorems 1 could be high for the policies involved in the GPI procedure because of inaccurately learned embeddings, which could result in poor performance.

6.2 What is the effect of learning $\mathbf{w}$

We ran supplemental experiments where the reward weight vector $\mathbf{w}$ was not provided to agents; rather, it was approximated while interacting with the environment for agents using algorithms that need $\mathbf{w}$ i.e., SFQL and SFNEC. As shown in Figure 6.1, we noticed a reduction in the performance across all agents that rely on $\mathbf{w}$. This drop in performance is expected due to errors in approximating $\mathbf{w}$. Crucially, we observed that learning $\mathbf{w}$ resulted in SFNEC not being the best performing agent anymore in this setup that uses the identity embedding and single-sample training method.

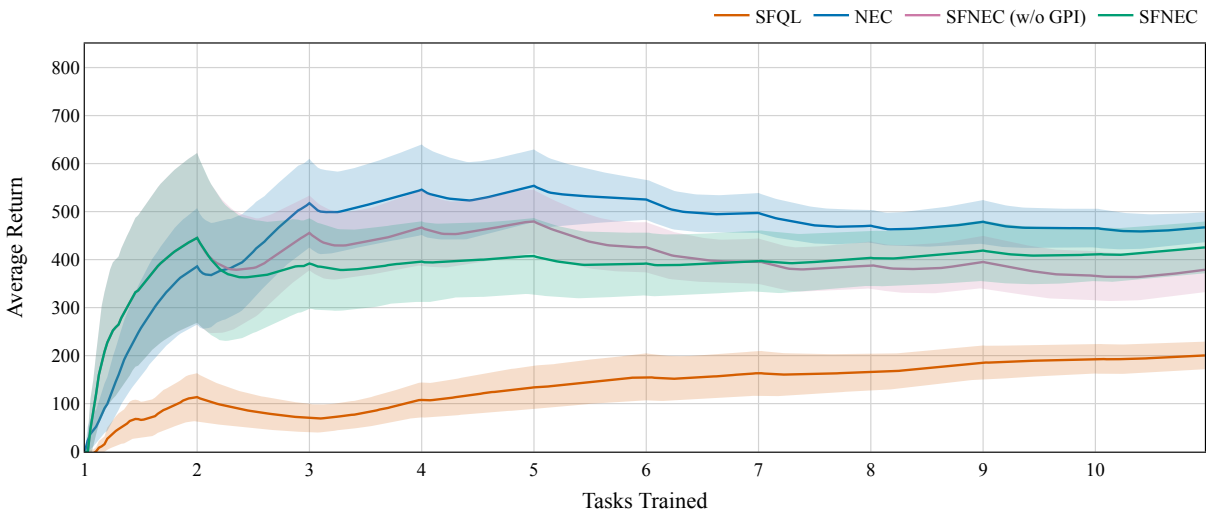

Figure 6.1 – Average return of SFQL, NEC, and SFNEC agents on the four-room environment while learning $\mathbf{w}$. Performance drops for SFQL and SFNEC agents that depend on $\mathbf{w}$ compared to Figure 5.7. NEC performance remains the same and performs best in this setting. Averages are taken over 10 runs with the standard error shown as the shaded region around solid lines.

We posit that the compounding of approximation errors from estimating $\mathbf{w}$ and the successor features $\boldsymbol{\psi}$ might lead to this reduced performance for agents using successor features, especially as the best performing agent in this setup is the NEC agent that does not rely on $\mathbf{w}$ nor successor features. We also notice here that SFNEC without GPI performs slightly better than its counterpart with GPI, and believe this is also due to similar reasons as discussed in Section 6.1. Nonetheless, there are many application domains where the reward function given by $\mathbf{w}$ would be known, and the SFNEC model with GPI would perform best in this case.

6.3 What is the effect of varying memory capacity

We know an essential consideration in a real-life implementation of our proposed SFNEC model and algorithm is how well it will scale with many tasks. However, the scheme scales linearly, and this might be too expensive if one keeps a large DND memory per action per task. Thus, we were interested in observing the degradation of the agent's performance with the DND capacity.

As shown in Figure 6.2, there is a point beyond which increasing capacity does not lead to improvement in performance. Practically, this means we might be able to use the minimum possible capacity for memory that guarantees good performance, knowing that a larger capacity would not result in performance gains. Also, the gradual degradation in performance shows that the method can be flexibly tuned to achieve the desired tradeoff between performance and memory requirements for a particular application.

6.4 What is the effect of varying number of neighbours

A critical parameter when performing value estimation using our model is the number of neighbours used. This induces a sort of bias-variance tradeoff, and we experimented with different settings shown in Figure 6.3.

Our results indicate that varying the number of neighbours should follow a 'U-curve' for a particular application, reducing performance when going towards either extreme of estimating with a single neighbour or the entire memory. This is expected because, on one end, we will have a method that obtains estimates that overfits to its nearest neighbour

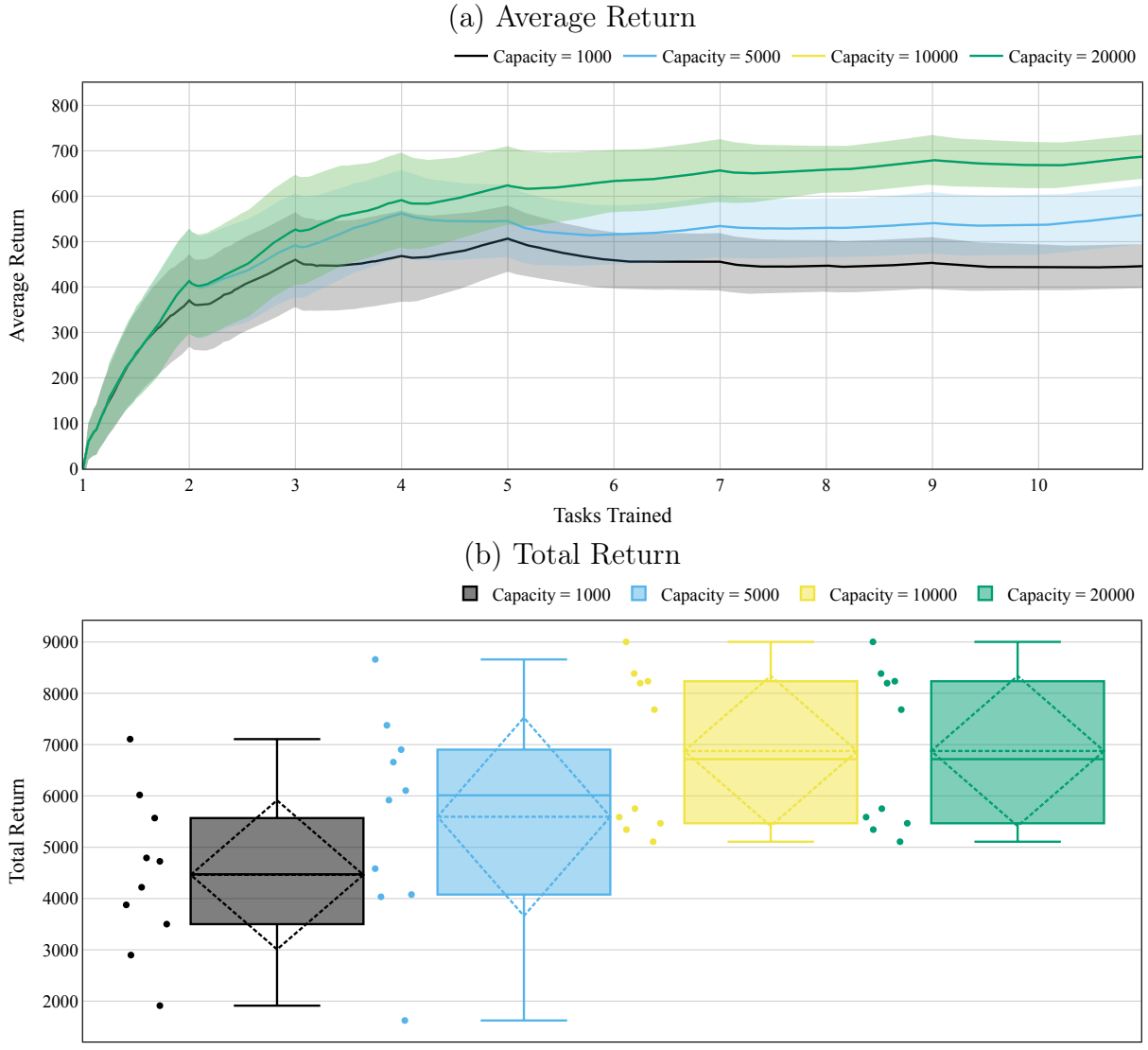

Figure 6.2 – Average and Total return of SFNEC agents on the four-room environment when varying the DND memory capacity. Averages are taken over 10 runs with the standard error also depicted as shaded regions. Beyond 10000, further increase in capacity does not lead to improved performance. This is indicated by perfectly overlapping lines in the return curves. The box plots also show this by showing the total return for 10 runs.

(i.e. high variance). On the other end, we have a method that tries to fit its estimate over the entire dataset provided by the memory module (i.e., high bias).

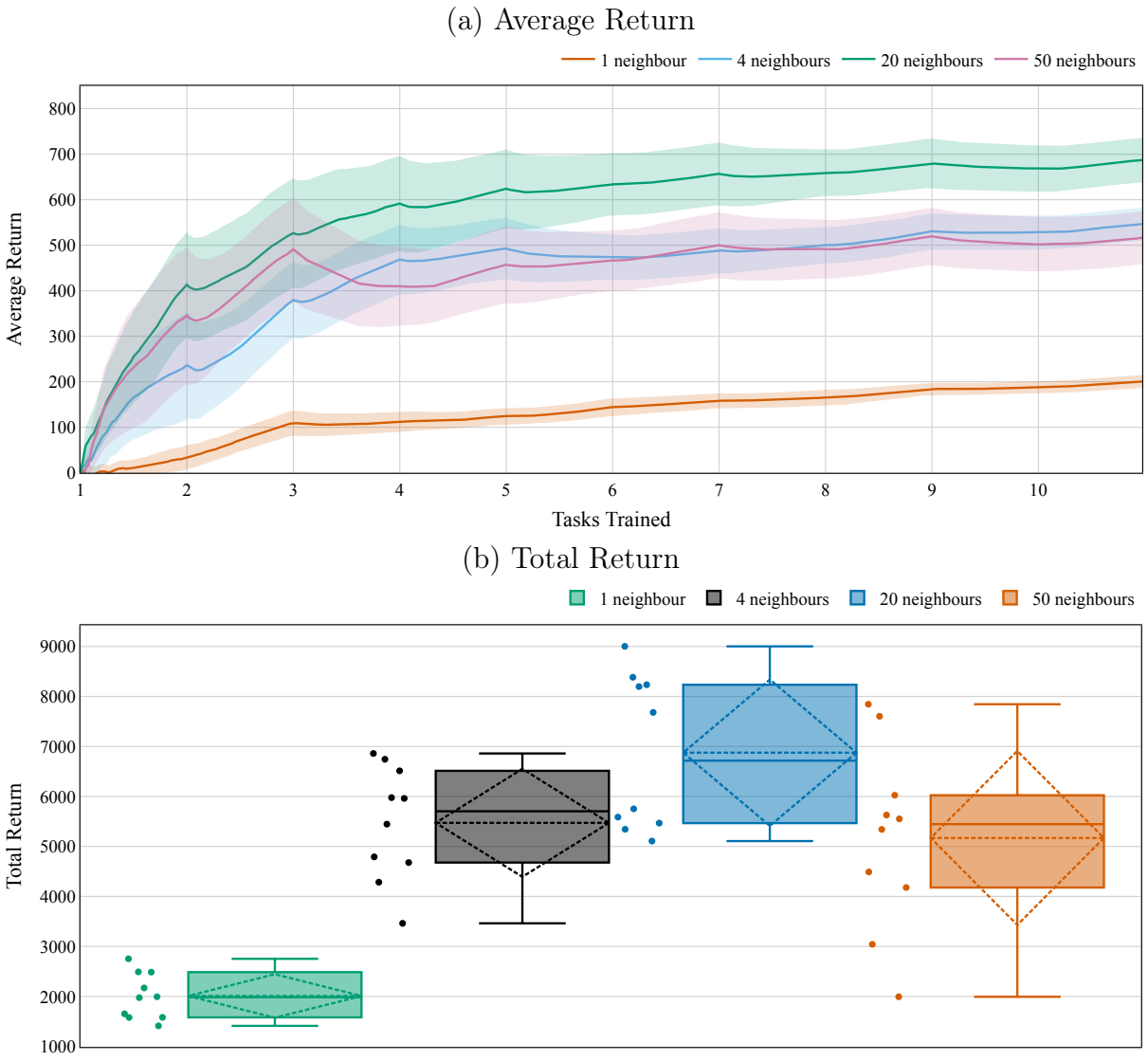

Figure 6.3 – Average and Total Return of SFNEC agents on the four-room environment when varying number of neighbours used for value estimation. Averages are taken over 10 random runs with the standard error also depicted as shaded regions. The box plots depicts more clearly that the performance follows a 'U-curve'. It shows the total return for 10 runs.

— 7 —
Conclusion and Future Work

In the beginning, we posed the research question if it is possible to define a framework combining the benefits of episodic control with the SF&GPI framework. Embarking on answering this question led to our SFNEC model and algorithm definition. With the experimental evaluation detailed in this study, we showed the viability of this method. Nevertheless, we have observed limitations in this approach which we believe is fruitful ground for future work.

We noted in our discussion how learning an embedding for the keys used in memory might lead to higher approximation errors, resulting in a poor GPI procedure's performance when used in our model. Therefore, we believe it is also interesting to develop techniques that allow for stable integration of batch methods and experience replay with our framework.

Another exciting avenue for further investigation is to analyze the performance guarantees of our SFNEC algorithm theoretically. The benefit would be a more profound and principled understanding of the algorithm, which could also lead to practical insights on improving it, especially in cases where it performs sub-optimally.

In conclusion, we see this framework as applicable and beneficial in the real world, for example, in domains like robotics, where learning could be in continual or lifelong settings. The data efficiency/learning speed combined with the ability to transfer knowledge across tasks as demonstrated here will be beneficial for such applications. However, a few problems would need to be overcome to define a good implementation. An example would be determining which tasks to keep in memory as suitable base tasks for generalization as memory requirements are a real-world constraint, and we would like to keep as many diverse valuable skills in memory as possible. Similarly, deciding when to learn tasks or automatically detect task switches is an area to tackle. Some suggestions for tackling such problems have been pointed out in [4], and we believe it would be fruitful work to investigate applying SFNEC on real-world tasks with an evaluation of different techniques to handle these various challenges.

Bibliography

[1] Andrea Agostinelli, Kai Arulkumaran, Marta Sarrico, Pierre Richemond, and Anil Anthony Bharath. Memory-efficient episodic control reinforcement learning with dynamic online k-means. *arXiv:1911.09560 [cs, stat]*, November 2019. arXiv: 1911.09560.

[2] Kai Arulkumaran, M. Deisenroth, Miles Brundage, and A. Bharath. A brief survey of deep reinforcement learning. *ArXiv*, abs/1708.05866, 2017.

[3] André Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel J. Mankowitz, Augustin Zídek, and Rémi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. *CoRR*, abs/1901.10964, 2019.

[4] André Barreto, Will Dabney, R. Munos, Jonathan J. Hunt, Tom Schaul, D. Silver, and H. V. Hasselt. Successor features for transfer in reinforcement learning. In *NIPS*, 2017.

[5] André Barreto, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117(48):30079–30087, December 2020.

[6] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, September 1983.

[7] Richard Bellman and Stuart Dreyfus. *Dynamic programming*. Princeton Landmarks in mathematics. Princeton University Press, Princeton, NJ, 1. princeton landmarks in mathematics ed., with a new introduction edition, 2010.

[8] J. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, 1975.

[9] Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z. Leibo, Jack W. Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *ArXiv*, abs/1606.04460, 2016.

[10] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, 23(5):408–422, May 2019.

[11] Matthew Botvinick, Jane X. Wang, Will Dabney, Kevin J. Miller, and Zeb Kurth-Nelson. Deep reinforcement learning and its neuroscientific implications. *Neuron*, 107(4):603–616, August 2020.

[12] Peter Dayan. Improving generalization for temporal difference learning: the successor representation. *Neural Computation*, 5(4):613–624, July 1993.

[13] Samuel J. Gershman. The successor representation: its computational logic and neural substrates. *The Journal of Neuroscience*, 38(33):7193–7200, August 2018.

[14] Samuel J. Gershman and Nathaniel D. Daw. Reinforcement learning and episodic memory in humans and animals: an integrative framework. *Annual Review of Psychology*, 68(1):101–128, January 2017.

[15] Timothy M. Hospedales, Antreas Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, PP, 2021.

[16] Haotian Hu, Jianing Ye, Zhizhou Ren, Guangxiang Zhu, and Chongjie Zhang. Generalizable episodic memory for deep reinforcement learning. In *ICML*, 2021.

[17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[18] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, July 2021.

[19] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40:e253, 2017.

[20] A. Lazaric. Transfer in reinforcement learning: A framework and a survey. In *Reinforcement Learning*, 2012.

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[22] M. Lengyel and P. Dayan. Hippocampal contributions to control: The third way. In *NIPS*, 2007.

[23] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, May 1992.

[24] Zichuan Lin, Tianqi Zhao, Guangwen Yang, and Lintao Zhang. Episodic memory deep q-networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2433–2439, Stockholm, Sweden, July 2018. International Joint Conferences on Artificial Intelligence Organization.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[26] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: a survey. *arXiv:2006.16712 [cs, stat]*, February 2021. arXiv: 2006.16712.

[27] Ofir Nachum, S. Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *NeurIPS*, 2018.

[28] Daichi Nishio and Satoshi Yamane. Faster deep q-learning using neural episodic control. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 486–491, Tokyo, Japan, July 2018. IEEE.

[29] Yael Niv. Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154, June 2009.

[30] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv:1912.06680 [cs, stat]*, December 2019. arXiv: 1912.06680.

[31] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[32] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech Badia, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2827–2836, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[33] Martin L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley Series in Probability and Statistics. Wiley, 1 edition, April 1994.

[34] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, September 1951.

[35] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, J. Kirkpatrick, K. Kavukcuoglu, Razvan Pascanu, and R. Hadsell. Progressive neural networks. *ArXiv*, abs/1606.04671, 2016.

[36] Marta Sarrico, Kai Arulkumaran, Andrea Agostinelli, Pierre Richemond, and Anil Anthony Bharath. Sample-efficient reinforcement learning with maximum entropy mellowmax episodic control. *arXiv:1911.09615 [cs, stat]*, November 2019. arXiv: 1911.09615.

[37] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[38] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[39] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction.* Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[40] Matthew E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, 2009.

[41] Momchil S. Tomov, Eric Schulz, and Samuel J. Gershman. Multi-task reinforcement learning in humans. *Nature Human Behaviour*, 5(6):764–773, June 2021.

[42] Pedro Tsividis, Thomas Pouncy, J. L. Xu, J. Tenenbaum, and S. Gershman. Human learning in atari. In *AAAI Spring Symposia*, 2017.

[43] Endel Tulving. Episodic memory: from mind to brain. *Annual Review of Psychology*, 53(1):1–25, February 2002.

[44] Endel Tulving, Wayne Donaldson, Gordon H. Bower, and United States, editors. *Organization of memory.* Academic Press, New York, 1972.

[45] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019.

[46] C. Watkins. Learning from delayed rewards. 1989.

[47] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.

[48] Guangxiang Zhu, Zichuan Lin, G. Yang, and Chongjie Zhang. Episodic reinforcement learning with associative memory. In *ICLR*, 2020.

[49] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: a survey. *arXiv:2009.07888 [cs, stat]*, March 2021. arXiv: 2009.07888.