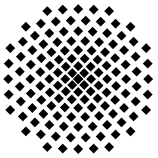


University of Stuttgart
Institute for Signal Processing and System Theory
Professor Dr.-Ing. B. Yang

Masterthesis D1346

Addressing Catastrophic Interference in Policy Optimization Agents

Behandlung katastrophalen Vergessens in Agenten mit Policy-Optimierung

Author: Dennis Klau

Date of work begin: January 17, 2020
Date of submission: December 3, 2020

Supervisor: Mario Döbler

Keywords: Deep learning, Reinforcement learning, Neural networks, Continual learning, Catastrophic forgetting, Atari, PPO

English: Catastrophic forgetting in (deep) reinforcement learning is not only an issue for continual learning on different environments, but also inside the same domain. By re-building a model proposed in [1] and applying it to the intra-environment task setting of [2], we present an adapted model and training procedure, the Memento-PNN, that addresses the problem of catastrophic intra-environment interference.

Deutsch: In dieser Arbeit zeigen wir, dass im Bereich des (tiefen) verstärkenden Lernens katastrophales Vergessen nicht nur beim Trainieren von verschiedenen, sondern auch innerhalb der gleichen Umgebungen auftritt. Wir stellen die Memento-PNN Methode vor, die katastrophales Vergessen in der selben Umgebung deutlich reduziert, indem wir eine Netzwerk Architektur aus [1] nutzen und sie auf die Trainings-Prozedur aus [2] anwenden.

Contents

Glossary

ALE Arcade Learning Environment. 27, 28, 33–35, 37–39, 49, 50

ANN Artificial Neural Network. 1, 20

catastrophic forgetting . 2, 4, 20, 22, 23, 25, 49

continual learning . 2, 4, 20–23, 25, 27, 28, 38, 41

DL Deep Learning. 1, 2, 5, 6, 20, 27, 28

DNN Deep Neural Network. 19

DRL Deep Reinforcement Learning. 1, 2, 5, 6, 8, 12, 15

HP hyperparameter. 18, 34, 37, 45

ICL Incremental Class Learning. 22

IDL Incremental Domain Learning. 21–23

ITL Incremental Task Learning. 21, 22, 25

KL Kullback-Leibler divergence. 18, 37

MDP Markov Decision Process. 7–13, 17, 21

ML Machine Learning. 1, 2

MLP Multilayer Perceptron. 25–27, 43

NN Neural Network. 2, 12, 21, 22

PDF probability density function. 6, 8

PNN Progressive Neural Network. 3, 4, 25–27, 29, 30, 33, 40, 49, 50

PPO Proximal Policy Optimization. 17, 18, 25, 29, 30, 33, 38, 40

RL Reinforcement Learning. 1–10, 12, 13, 15, 16, 20–22, 27, 33

SGA stochastic gradient ascent. 15, 17

SGD stochastic gradient descent. 37

TD Temporal Difference learning. 18, 19

TRPO Trust Region Policy Optimization. 15, 17, 18

1. Introduction

Reinforcement Learning (RL) is considered one of the three major branches of Machine Learning (ML) [3]. In contrast to supervised and unsupervised learning, RL picks up the approach of self-teaching through observation and feedback, inspired by the human nature of learning. An RL *agent* shall have the ability to learn rules and goals, grasp the concept of cause and effect and understand the consequences of actions – all of that without supervision by acting on its own in a certain environment.
The term Deep Reinforcement Learning (DRL) is a specification and refers to the method, that is used to represent the learned knowledge of an agent. In DRL, deep learning models like Artificial Neural Networks (ANNs) are used to embody and approximate the behavior of learners, that move in environments too complex to fully represent.

Due to major obstacles, particularly the poor sample efficiency and high variance in the results together with unstandardized evaluation protocols, the RL branch was less relevant in the beginning of broad ML research than e.g. supervised learning, which quickly showed promising performance after breakthroughs in Deep Learning (DL). Because of this, RL and especially DRL were long time a niche application in the field of ML and DL respectively, until major advances in the field caught the attention of researchers again.[1]

This can be attributed to an increase of available and standardized open-source frameworks, which made it easier for researchers to start in RL and quickly produce prototype algorithms. Another benefit is the amount of simulation and physics engines[2] nowadays, that facilitate faster training of these sample inefficient methods by simulating the agent and necessary part of the world, instead of requiring to have a physical learner (e.g. a robot) and collecting samples by acting in the real world.
Much contribution to this gain in popularity of DRL can also be accounted to the recent success of lately published algorithms, that not only proved to work on controlled and comparable easy environments, but also on much harder tasks. Examples for this would be *AlphaGo* [5] (which defeated the world champion of the board game "Go"), its successor *AlphaZero* [6] and *AlphaStar* [7] (an improvement of the idea of AlphaZero to the more complex environment of "StarCraft 2"[3]) or *OpenAI Five* [8] (that was able to beat the world champions of the popular *MOBA*[4] game "Dota 2" by Valve multiple times), which showed that such agents can catch up and even surpass human-level planning and acting skills in specific but also quite complicated and only partially observable environments.

[1] MIT-Technology-Review:Where-is-AI-headed-next (accessed at 06-02-2020)
[2] for example *Gym* [4], *MuJoCo*
[3] a real-time strategy game by Blizzard Entertainment, that requires a great amount of micro- and macro-management
[4] MOBA stands for **M**ultiplayer **O**nline **B**attle **A**rena

1.1. Motivation

Reinforcement learning is the ML branch, that focuses most on goal-directed learning from interaction and tends to mimic learning like biological brains best. The ability of life-long learning by incorporating new information into old knowledge and applying variations of already obtained abilities to new problems are some of the main reasons, why humans and other biological agents are so effective in navigating through, adapting to, and solving problems in unknown and seemingly random environments.

A major shortcoming of modern DL methods like Neural Networks (NNs) is their inability to successfully implement such continual learning. This is largely because of their susceptibility to catastrophic forgetting. Catastrophic forgetting is the phenomenon in which any significant alteration to the parameters of an already trained model leads to an abrupt loss of what has been learned. A more detailed coverage of this topic is made in Section 2.2.
This poses a fundamental constraint on the way in which neural networks can be trained and deployed: At training time, the neural network needs to be exposed to the entire training data (or a carefully selected and very good representation of it), and once deployed, it can't be incrementally taught more new data. This can only be done by recovering all the original information in addition to the new data, and then essentially re-training the model.
Currently, this problem is sidestepped by either preemptively collecting all the data that might be needed for every task a model should be able to solve and then running a single optimization job on the whole dataset, or by training different independent models for each problem. This works indeed reasonably well for fewer or smaller problems, making the argument for sequential or incremental learning weaker, but extrapolating the evolution of DL based on past development, the trend towards agents, that are responsible for more and more complex tasks is clearly visible. This in turn means, that catastrophic forgetting is headed for catastrophic costs and a major barrier towards general artificial intelligence. [9]

In reinforcement learning, this is even more important: For agents, that learn by exploring and collecting experience inside an environment in an online manner, it is crucial to overcome these challenges in order to obtain agents, that can learn and solve arbitrary tasks in a reasonable time.

1.2. Goal of this thesis

Continual learning is often only considered to apply, when a sequence of individual, clearly separated tasks are at hand. Then, agents are usually trained on one environment and, when the use-case changes or is extended later, trained on the next one. However, the side-effects of continual learning do not only appear on the transition from one task to another, but can also occur inside the same problem. Previous works already showed, that catastrophic interference not only happens for separated high-level tasks, but also inside the same RL challenge [2]. This is especially true, if the environment or objective is abstract or complex, with many individual subtasks required to successfully learn it.

In this work, we will extend the concept of continual learning in DRL by not only applying countermeasures against catastrophic forgetting to individual tasks with a distinct set of goals, but elaborating on techniques, that counter interference inside the *same* superordinate task /

environment.
For that, we will utilize and extend on the findings of two preceding papers, namely [1] and [2], that study an advanced model architecture and a training protocol for mitigating intra-environment interference. Combining both approaches, we create a novel algorithm, the Memento-PNN, to account for catastrophic forgetting inside the same environment of gradient-based RL agents.

1.3. Related Work

This work is based on two papers from [1] and [2]. The first developed the idea of Progressive Neural Networks (PNNs), a novel network architecture to attenuate catastrophic forgetting and the problem of network initialization for multi-task learning. While these networks enable transfer learning and are immune to catastrophic forgetting by design, they exhibit a growth in model parameters with the number of tasks though. The ideas for this novel architecture are addressed in detail in Section 3.1.
[2] studies the effects of catastrophic interference inside the same environment of an RL agent and specifies a model-free training protocol to reduce it. The proposed algorithm, the Memento experiment, identifies performance plateaus of the agent during training, stops it and creates clones of this agent to continue training from the plateau. We elaborate in much more detail on this method in Section 3.2.

Elastic weight consolidation (EWC) is a regularization approach showcased in [10] to mitigate catastrophic forgetting. The regularizer protects neuron, that were found to be important for earlier tasks. For that, EWC computes a Gaussian approximation of the posterior distribution of each task using the class centers as mean and diagonal Fisher information as covariance. EWC takes inspiration from synaptic consolidation in neurobiology. A major restriction is, that task overlaps in the feature space are required for the algorithm work well and that the Fisher information can only be computed in an offline manner after completion of a task training.
In the area of supervised classification, [11] proposes *intelligent synapses* as a replacement for neurons in neural networks. Each synapse has a local importance estimation measure for past tasks, that defines how much its weight is allowed to change. Intelligent synapses are inspired by and similar to EWC, but incorporate the regularizer into each individual synapse (neuron) and compute the regularization value in an online manner.
Another research [12] follows an approach similar to intelligent synapses which equips value-based RL agents with a synaptic model. Instead of regularizing parameter changes over different tasks (EWC, intelligent synapses), the authors use a synaptic plasticity model over a range of timescales inspired by long and short-term memory of the brain. The work shows, that catastrophic forgetting, as well as the necessity for experience replay buffers can be alleviated. The setup is also tested against intra-environment interference for simple tasks.

The work of [13] is a similar approach to mitigate catastrophic forgetting via design specialties of an agent's model. The authors propose a combination of PNNs, EWC and knowledge distillation methods called *Progress & Compress*. The framework consists of a PNN with two columns, the active column and knowledge base, which are alternately used to first learn a new task (active column) and then compress the learned knowledge into the knowledge base.

In the work of [14], off-policy, on-policy and imitation learning (behavioral cloning) are leveraged to create a replay-based "Continual Learning with Experience And Replay" (CLEAR) technique to reduce catastrophic forgetting in RL agent. The underlying actor-critic model (Importance Weighted Actor-Learner (IMPALA) [15]) uses on-policy learning for adaptation to new tasks and off-policy (V-Trace) learning from replay experiences to maintain previous knowledge. To further ensure the stability of old experience, behavioral cloning between the old and new version of the learner is used.

1.4. Thesis Structure

In this work, we will start with a theoretical part, that introduces the basic concepts of Reinforcement Learning and continual learning. Especially, the mathematical and conceptual definitions of the reinforcement learning problem and its learning approaches are covered in Section 2.1. Afterwards, the concept of continual learning and phenomenon of catastrophic forgetting are outlined in Section 2.2.

In Chapter 3, the methods and protocols of the Progressive Neural Network (PNN) architecture (Sec. 3.1) and Memento experiment (Sec. 3.2) are described, before the ideas of the novel Memento-PNN algorithm are specified in Section 3.3.

We continue with an outline of the used RL environments, the conducted experiments and their setup in Chapter 4. More precisely, Sections 4.1.3 and 4.1.4 cover the environments (Atari games) and models of the used agents and Section 4.2.1 reports the results of the proposed Memento-PNN compared to other methods. Section 4.2.2 shows the performance of the algorithm on more than one intra-environment contexts. In Section 4.2.3 we investigate a possible direction of how to make the Memento-PNN approach task label independent.

The last Chapter 5 concludes the thesis with a final discussion of the results and an outlook for possible future improvements of the algorithm.

2. Theory

Here the basic principles and terminology for Deep Reinforcement Learning (DRL) and other DL concepts will be explained, that are used to formalize the ideas elaborated throughout this work. The Chapter aims to provide a common understanding of concepts like:

- Markov processes and RL environments
- agents, actions, rewards and return
- exploration vs. exploitation
- the RL optimization problem
- the concept of continual learning

Each of the individual parts here will not be covered in full detail, as an exhaustive explanation of the theory behind these concepts fill entire books. For a comprehensive coverage of Deep Learning and Reinforcement Learning, one can resort to [16, 17, 18] or [3].

2.1. Background to (Deep) Reinforcement Learning

A reinforcement learning system consists of several parts, that interact continuously with each other. In this Section, an attempt is made to briefly describe and derive the most important concepts for these parts in order to give a good overview of the RL problem, the challenges when training RL agents and the mathematical formalism.

Many notional conventions and derivations in this Section are inspired by [18] and [17].

2.1.1. Classification of Reinforcement Learning

The essence of RL is a decision maker, that interacts with its world over time to achieve a desired goal. For that, the learner must be able to understand its current situation to some extend and is required to take actions which affect it. The learner and decision maker are called *agents* and the part of the agent, that determines behavior is called the *policy*. The world is called the *environment* and comprehends everything outside the agent. The goal of the agent always depends on the environment states.
DRL is different to *supervised learning* and *unsupervised learning*. In *supervised learning*, models are trained from a predefined data set of *labeled* examples provided by a knowledgable expert. In this field, the correct system response (action) is known for the samples and the primary task for the learner is to generalize in a meaningful way beyond the explicit examples provided in the training set. In *unsupervised learning*, the main objective is generally to find hidden structures in collections of unlabeled data.

(a) Schematic diagram of the agent-environment interaction loop. The agent receives a state and direct performance feedback and chooses an action depending on the learned knowledge.

(b) Iterative adaption procedure of an arbitrary RL agent.

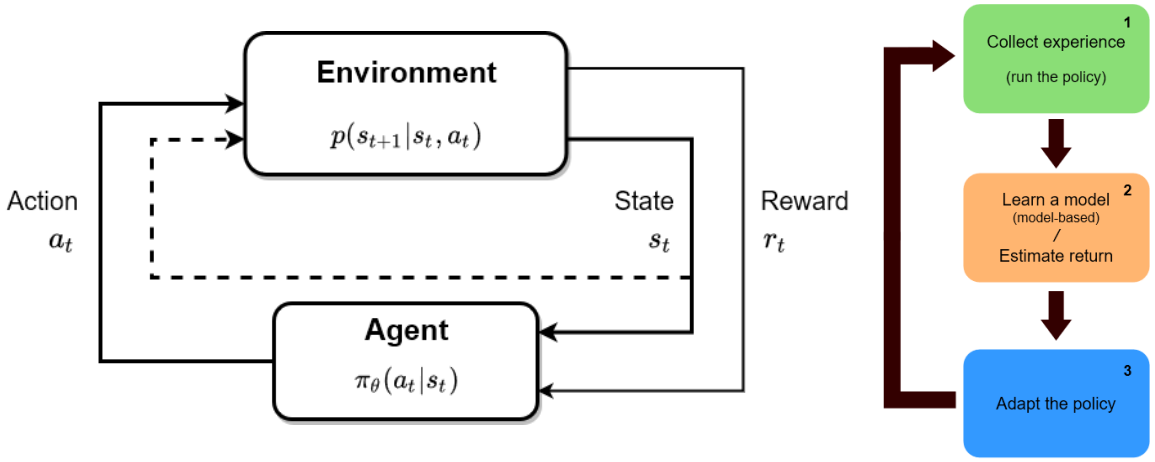


Figure 2.1.: Figures inspired by [17, Fig. 3.1] and [18, Anatomy of RL algorithms]

In the RL area, the goal is to maximize a reward signal by self-acting inside the given environment. The RL paradigm can be understood as a continuous loop of interactions between an agent and the environment, where the agent takes actions and receives responses returned by the environment. This concept is depicted in Figure 2.1a. In the graph, the environment is abstracted by a probability density function (PDF) that takes an action and the current state and generates the next state from it. The agent, similar to other DL areas, can be expressed by a mapping function π_θ, that depends on the internal parameters θ of the used algorithm (e.g. the weights of a neural network in the case of DRL) and infers actions from observed states.

Figure 2.1b shows the general learning approach for arbitrary RL agents: 1) the current policy is rolled out in the environment to collect samples, 2) the RL system fits a model of the world (if the agent is model-based) and/or estimates its performance under the current policy, and 3) the parameters of the policy are updated, depending on which type of policy representation is used.

Model-free vs Model-based RL algorithms can be divided into agents, that learn – additionally to an explicit or implicit policy – a *model* of the environment (*model-based*) and agents, that learn exclusively from trial-and-error (*model-free*).
Model-based systems try to learn a replica of the environment (more precisely the transition function), that mimics its original behavior and allows to make predictions about the future states of the environment. This allows the agent to *plan* through the environment by predefining a sequence of actions before any of the actual states are encountered, that would lead to this action. A big challenge, however, is the availability of a ground-truth model of the environment. If such a model is not available, the agent has to learn it purely from interactions. This often induces bias to the learned model, because not every state of a generally high-dimensional state space environment can be reached in a reasonable time. In that case, the agent can learn to exploit that bias, leading to good performance with respect to the learned

model, but poor behavior in the real environment.
Model-free learners only require experience from the environment as input, whereas model-based agents use experience collected from their model to compute performance estimates in an intermediate backup step and use them to update the policy. [17, Ch. 8.1]

Exploration vs Exploitation Another challenge specific to the area of RL is the trade-off between *exploration* and *exploitation*. In order to maximize the received reward, a reinforcement learning agent must prefer the actions that are *known* to be good in the sense of performance. However, in order to identify such actions being effective in the current situation, the agent must also make decisions it never tried before. On the one hand, the learner has to *exploit* what it has already seen, but on the other hand it has to *explore* other possible actions to make better decisions in the future. The exploration-vs-exploitation dilemma is that optimizing one area automatically impairs the other and neither can be perfectly solved without failing the task (the "No free lunch theorem" [19] applies here). The dilemma has been intensively studied for decades but so far remains unresolved. [17, Ch. 1.1]

2.1.2. The Environment

An environment is terminologically defined as the world an RL agent has to live in and mathematically formulated to be a Markov Decision Process (MDP) [20]. It is explained by a set of attributes, that a learner uses to *sense* its situation and derive decisions. These attributes are generally tied to a certain time when they occur and can be discretized into steps, at which the environment and agent progress[1].
Before we can describe how the environment is integrated inside a reinforcement learning system, we must introduce the mathematical concept the environment is based on and the properties it inherits.

Markovian Processes

Markov Decision Processes are a direct formulation for the problem of learning from interaction. They are an extension of the Markov Chain by introducing decision making possibilities. The Markov chain is defined as a tuple $\mathcal{M} = \{\mathcal{P}_0, \mathcal{S}, \mathcal{T}\}$ consisting of a *state space* $\mathcal{S}$ (discrete or continuous) of all possible states in an environment, a transition operator $\mathcal{T} : p(s_{t+1} \mid s_t)$, that describes the probability of encountering the successive state s_{t+1} given the current state at time t, and an *initial state distribution* $\mathcal{P}_0 : p_0(s_0)$, that defines the first state s_0 in a Markov Chain, since it cannot be influenced by any action from the agent. An important property of Markov Chains is, that the next state s_{t+1} is conditionally independent of the previous state s_{t-1} and only depends on the current state s_t (*Markov property*).

MDPs are described by $\mathcal{M} = \{\mathcal{P}_0, \mathcal{S}, \mathcal{A}, \mathcal{T}, R\}$ with an additional set of possible *actions* $\mathcal{A}$. The *transition probability* $p(s_{t+1} \mid s_t, a_t)$ is extended by the chosen action a_t to include the decision making process. Also, a *reward signal* $R : \mathcal{S} \times \mathcal{A} \to \mathcal{R} \subseteq \mathbb{R}$ is introduced, that maps a current state-action pair to an immediate feedback value.

[1] The derivations made here are done in the discrete-time scenario to keep things simpler, although many formulas and assumptions can be extended to the continuous-time case. Refer e.g. to [21]

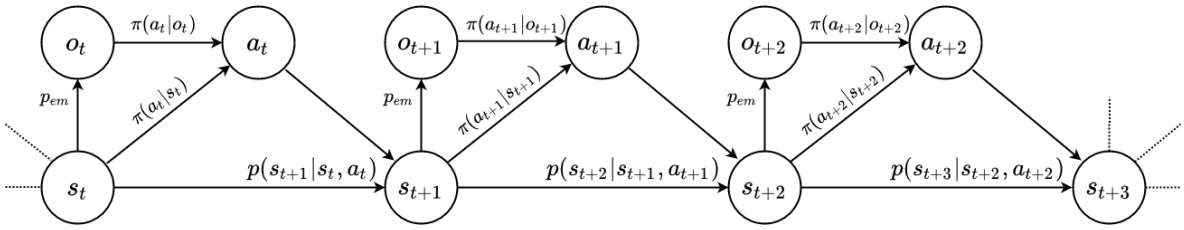


Figure 2.2.: MDP in form of a directed transition graph. For each state, the agent has to make a decision by executing its policy $\pi(\cdot)$. o_t is generated by the emission process $p_{em}(s_t)$. After executing the inferred action, the environment reacts by realizing a state from the transition distribution $p(\cdot)$.

The MDP is called *partially observable*, if the learner must deal with *observations* instead of states. Then, the process is a tuple $\mathcal{M} = \{\mathcal{P}_0, \mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{T}, \mathcal{E}, R\}$ and additionally expanded by adding the *observation space* $\mathcal{O}$ and an *emission probability* $\mathcal{E} : p_{em}(o_t|s_t)$. Observations are instances, that are generally created by some sort of stochastic filter process of the environment and yield less information than the corresponding state they are generated from. In this case, $o_t \sim p_{em}(\cdot|s_t)$ represents the probability with which a specific observation o_t is generated by the environment in state s_t.

The graph of a general MDP is shown in Figure 2.2 with the quantities described above, as well as the decision mapping functions (policies) $\pi(a_t|s_t)$ and $\pi(a_t|o_t)$ introduced later. Note that we restrict ourselves to *finite* MDPs, meaning that all sets $\mathcal{S}, \mathcal{O}, \mathcal{A}$ and $\mathcal{R}$ have a finite number of elements. This ensures well defined discrete PDFs for the random variables s_t, o_t and r_t, that satisfy the Markov property. [17, Eq. 3.2]
Note, that when we train model-free algorithms, the transition probability $p(s_{t+1} \mid s_t, a_t)$ is generally not known at any time and it is assumed, that the agent can only interact with it to sample new states from the environment.

State-Observation Differentiation

In that way, reinforcement learning uses ideas from dynamical system theory to formalize problems using optimal control of MDPs.
The state $s_t \in \mathcal{S}$ includes all information and is a *complete* representation of the environment at the current time t, meaning that the information in s_t uniquely identifies each environment state. Closely tied to it is the observation $o_t \in \mathcal{O}$ and represents all information, that is *available* to the agent through its sensorial capabilities. It is normally a subset of s_t in terms of information entropy. In many problems, the state is not available to the agent, which has then to learn the task from observations with possibly missing or ambiguous information. When an DRL agent encounters a state or observation, it has to make a decision (action) $a_t \in \mathcal{A}$, that influences the environment and leads to a next state at time step $t + 1$. This concept is visualized in Figure 2.3 with an agent that uses observations to infer actions.

Many literature sources use s_t and o_t interchangeably and mathematically this holds for most derivations since they can be generalized, but in practice there is a difference: Consider for example a robot that uses cameras to locate objects to grab on an assembly line with a locomotion system controlled by an RL agent. Here, the state could include all information about the coordinates of every robot part as well as all objects, their dimensions (maybe even

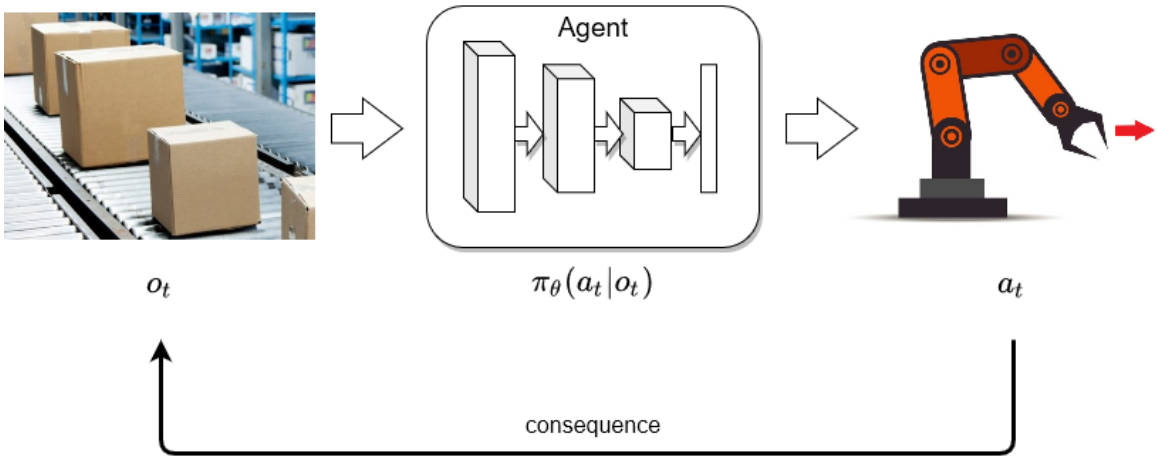


Figure 2.3.: Schematic representation for the observation-action loop of an RL agent in the example of an autonomous robot.

material types), movement dynamics (e.g. velocity), etc. in the world space. The information available to the system, however, are only the pixel images returned by the cameras (the observations), which in fact can be misleading and not fully descriptive. For instance just from one observation in Fig. 2.3, the agent cannot know in which direction the assembly line actually moves and how fast.

In the practical part of this work, we will deal exclusively with partially observable MDPs, where the model has to learn from image observations. The derived methods still hold, however, these tasks present a bigger challenge for the agent during training, than dealing with states in which all information about the environment is available. Minor changes in the observation space (i.e. pixel values) can belong to fundamentally different environment states, leading to decision processes of high variance. Because in general not enough information can be retrieved from a single observation, a sequence of consecutive images is given to the learner at each training step. This obviously violates the Markov property (as it is often the case for more complex problems), but ensures, that ambiguous or missing state information is reduced by choosing actions based on monitoring behavior of the environment in the near past. This requires the policy to generalize well during training by exploring many possible trajectories. The way the observations are constructed in this work and how they look like is presented in Section 4.1.2.

Trajectories

In order to derive the RL objective of accumulating high long-term reward, the stochastic description of a complete MDP is needed. We define a *trajectory* as a consecutive sequence of states or observations (whatever the available information of the environment is) and associated actions retrieved from the current version of the agents' policy. The experience collected in a trajectory is called an episode. Given, that the states of the world can be accessed, a trajectory is can be defined as

$$\tau = (s_0, a_0, s_1, a_1, \ldots s_t, a_t, \ldots) \tag{2.1}$$

where t denotes the temporal index for successive states-actions pairs. The *initial state* s_0 often has a special standing in the definitions, as the agent cannot influence this state by any taken action and is in most cases either a well-defined (deterministic) initial entry point to the environment or sampled from a *start-state distribution* $s_0 \sim p_0(\cdot)$. Either way, s_0 does not contribute to any variation of – and can therefore often be omitted in – the objective formulation. The definition of the trajectory can additionally be restricted to a *finite horizon state space*

$$\tau = (s_0, a_0, s_1, a_1, \ \dots \ s_T, a_T) \tag{2.2}$$

where all actions lead at some point in time T to a defined *termination state* that finalizes an episode. This simplifies the formulation of the objective in 2.1.5 slightly, but also introduces some requirements, that we sometimes do not want to or can enforce in RL. [18]

By utilizing the chain rule and Markov property, we can write down the joint probability distribution over a whole trajectory as [22]

$$\begin{aligned} p_\theta(\tau) &= p_\theta\Big(\{a_t, s_t\}_{t=0}^{T}\Big) \\ &= p_0(s_0) \prod_{t=0}^{T} p(s_{t+1}|s_t, a_t)\, \pi_\theta(a_t|s_t) \end{aligned} \tag{2.3}$$

In deep RL, we formulate the joint distribution $p_\theta(\cdot)$ with the subscript θ, because the experience in the trajectories is collected by rolling out a current (or other)[2] version of the agent's policy and receiving feedback from the environment. The fact, that the policy is a parametric method defined by the variables in θ means, that the actions a_x and thus the states s_{x+1}, that follow these actions in the trajectory, are dependent on the agent parameters.

2.1.3. Rewards and Return

Rewards are the immediate feedback of the environment in terms of how well the agent acted towards a defined goal. The *reward function* $R : \mathcal{S} \times \mathcal{A} \to \mathcal{R} \subseteq \mathbb{R}$, introduced in the MDP (2.1.2), specifies the instantaneous reward value r_t of a learner as

$$r_{t+1} = R(s_t, a_t) \tag{2.4}$$

The reward is defined to be returned by the environment at the next possibility, but cannot be received before the action is taken. Therefore, r_t is the reward for taking action a_{t-1}. This is why Eq. (2.4) returns the feedback at time step $t + 1$. The function depends on both: the current state s_t, as well as the chosen action a_t in that state, but not on the past (as required for the Markov property). The reward is the primary basis for altering the policy and is inherently part of the environment. This means, that the agent has no full control over the reward signal and cannot arbitrarily change it. The goal of the learner is then to maximize the total reward it can receive in the long run. [17]

The amount of cumulative reward an agent can achieve from a specific time onward (until the termination of the episode or infinitely) is called the *return*. Assuming we have an *episodic* task, which means the underlying MDP has a finite horizon and termination state, we can

[2] This difference is defined by the used algorithm and whether it is on- or off-policy (see Sec. 2.1.4)

define the *finite-horizon undiscounted return* as the sum of all future rewards beginning from time t: [22]

$$\begin{aligned} R_t &= r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \\ &= \sum_{t'=t+1}^{T} r_{t'} \\ &= r_{t+1} + R_{t+1} \qquad \forall t \in [0, T) \end{aligned} \tag{2.5}$$

with $R_T \stackrel{!}{=} 0$. Note, that in general the termination time T is a random variable, since the episodes normally have varying length.
However, when we deal with *continuous* tasks, where $T \to \infty$, the naive formulation of (2.5) is problematic, because an infinite-horizon sum is not guaranteed to converge to a finite value and hence hard to deal with. To alleviate this problem, we can define an *infinite-horizon discounted return* similar to the finite case:

$$R_t = \sum_{t'=t+1}^{\infty} \gamma^{t'-t-1} r_{t'} = r_{t+1} + \gamma R_{t+1} \tag{2.6}$$

with a discount factor $\gamma \in (0, 1)$ and r_t as in (2.4). This ensures not only convergence as long as the reward sequence $\{r_k\}$ is bounded, but also directs the agent's attention more towards the present rather than the future (immediate reward is better than at an unknown distant time step) [17, Eq. 3.7-3.10]. For $\gamma = 1$ and $T < \infty$, the discounted return becomes (2.5), making it a generalization of the finite case.
Note, that we use $R(\cdot)$ to refer to the reward-generating function of the MDP as in (2.4) and R_t to denote the return at an arbitrary time step t. In that sense, with the return definition (2.6), we use $R(\tau)$ to indicate the return over a whole trajectory

$$R(\tau) = R_t\big|_{t=0} = \sum_t \gamma^t r_t \qquad \forall t \tag{2.7}$$

which is identical to the sum of all received rewards for every time step t.[3]

Since receiving reward is a stochastic process, that depends on the characteristics of the transition function, reward signal and policy, our goal is to maximize the *expected return* under the trajectory distribution: [22]

$$\begin{aligned} J(\pi) &= \int_\tau p_\theta(\tau) R(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \right] \\ &\stackrel{!}{=} \mathbb{E}_{\tau \sim \pi} \left[\sum_t R(s_t, a_t) \right] \end{aligned} \tag{2.8}$$

using the finite MDP formulation (2.5) for slightly simpler notation. The expectation is taken under the assumption, that τ was generated by sampling from the environment according to the policy π, in which case we can set $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\cdot] \simeq \mathbb{E}_{\tau \sim \pi}[\cdot]$.

[3] For the rest of the derivations, we assume the case of finite-horizon MDPs with (2.5). While this may be not as general, it simplifies the notations and is more relevant in the presented setting.

2.1.4. Policies and Value Functions

The policy defines the actual behavior of an agent at each point in time. In general, policies may be stochastic, specifying probabilities for each action [17, Ch. 1.3]. In that sense, policies are a mapping of a perceived state to probabilities of selecting each possible action available in that state:

$$a_t \sim \pi(\cdot|s_t) \tag{2.9}$$

The two types of policies, that an agent can learn, together with their requirements are shown in the MDP graph of Fig. 2.2. Whether $\pi(a_t|s_t)$ or $\pi(a_t|o_t)$ can be learned is depending on the environment as described in Section 2.1.2. The policy is the core of an RL agent in the sense that it alone is sufficient to determine behavior.

An agent can use parametric function classes like Neural Networks as function approximators to represent the policy (explicitly or implicitly) to deal with problems like the *curse of dimensionality*, when classical methods are inefficient or not applicable. In that case, we speak of DRL and use the subscript θ to indicate the parametrization. In cases, where it doesn't matter whether the policy is parametric or not (like in most derivations below), we omit the subscript to simplify the notation.

While the reward signal represents the immediate feedback of the environment for reaching a specific state, the *value function* formalizes the "worth" of a state in the long term. It describes the final reward the agent can expect to accumulate over time starting from a specific state. This takes into account which states are likely to follow and what reward can be expected in that states. [17, Ch. 1.1,3.5]

Value functions have their justification in a simple consideration: the objective, that we are trying to optimize, is the expected sum of future rewards. However, as shown in Section 2.1.6, the direct optimization of the policy with gradient-based methods tries to find empirical estimates of this reward-to-go by single-sample evaluation, considering only the current trajectory as it was collected by rolling out π_θ. This is visualized in Figure 2.4 as the black arrow. At a specific time step t, the agent could take multiple possible actions in every state that follows in the trajectory, thus leading to many possible trajectories with different outcomes (blue arrows). The value functions attempt to give a better estimate of the reward-to-go by considering statistics of the multitude of trajectories that could follow a specific state. [18]

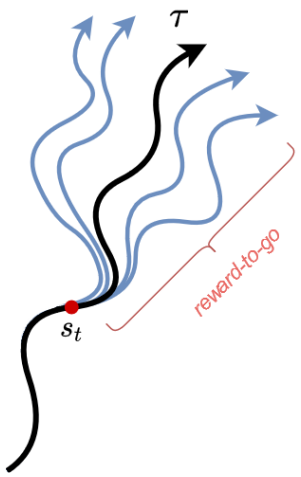


Figure 2.4.

In general, the value function V_π depends on a specific state s_t and policy π and expresses the expected return, when starting in s_t at time t and following π for the rest of the trajectory (for the rest of time in the continuous case). Assuming a finite MDP as before and using (2.5), (2.7) and (2.8) together with the linearity of expectation, we can formalize this function as:

$$\begin{aligned} V_\pi(s_t) &= \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R_t|s_t\right] \\ &= \sum_{t'=t}^{T} \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R(s_{t'},a_{t'})|s_t\right] \qquad \forall s_t \in \mathcal{S} \end{aligned} \tag{2.10}$$

where the meaning of $\mathbb{E}_{(s_t,a_t)\sim\pi}[\cdot]$[4] is still the same as $\mathbb{E}_{\tau\sim\pi}[\cdot]$, but indicating, that we only require the *rest* of the trajectory τ, starting from s_t, to follow the policy π. V_π is defined for arbitrary time steps t and how the state s_t was reached is not relevant for the function. [22]

Similarly, the *action-value function* – or *Q-function* – is defined as the expected return, when being in a state s_t, choosing an arbitrary action a_t at time t and *then* following π thereafter:

$$\begin{aligned} Q_\pi(s_t, a_t) &= \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R_t | s_t, a_t\right] \\ &= \sum_{t'=t}^{T} \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R(s_{t'}, a_{t'}) | s_t, a_t\right] \quad \forall s_t \in \mathcal{S},\ a_t \in \mathcal{A} \end{aligned} \tag{2.11}$$

Equivalent to the return value, $V_\pi(s_t)$ and $Q_\pi(s_t, a_t)$ are defined to be zero at time $t = T$. [17, Ch. 3.5]
An important property is, that the value function can be expressed in terms of the Q-function by marginalizing out the action:

$$V_\pi(s_t) = \mathbb{E}_{a_t\sim\pi}\left[Q_\pi(s_t, a_t)\right] \tag{2.12}$$

and the Q-function can be formulated w.r.t the value function, ensuring that the next state s_{t+1} follows the transition dynamics of the MDP:

$$Q_\pi(s_t, a_t) = R(s_t, a_t) + \mathbb{E}_{s_{t+1}\sim p(\cdot|s_t,a_t)}\left[V_\pi(s_{t+1})\right] \tag{2.13}$$

Both, (2.10) and (2.11) can be expressed as a *Bellman equation*: [17, Eq. 3.14], [22]

$$\begin{aligned} V_\pi(s_t) &= \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R_t | s_t\right] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi}\left[r_{t+1} + R_{t+1} | s_t\right] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R(s_t, a_t) + V_\pi(s_{t+1})\right] \end{aligned} \tag{2.14}$$

$$Q_\pi(s_t, a_t) = \ldots = \mathbb{E}_{s_t\sim p_\theta(\tau)}\left[R(s_t, a_t) + \mathbb{E}_{a_{t+1}\sim\pi}\left[Q_\pi(s_{t+1}, a_{t+1})\right]\right] \tag{2.15}$$

This recursive formulation is important to be able to compute, approximate and learn the value or Q-function in the RL framework. [17, Ch. 3.5]
Especially important for value-based algorithms is the *advantage function*. It describes the benefit of taking a particular action a_t over a randomly sampled action according to π:

$$\begin{aligned} A_\pi(s_t, a_t) &= Q_\pi(s_t, a_t) - V_\pi(s_t) \\ &\overset{(2.13)}{=} R(s_t, a_t) + \mathbb{E}_{s_{t+1}\sim p(\cdot|s_t,a_t)}\left[V_\pi(s_{t+1})\right] + V_\pi(s_t) \end{aligned} \tag{2.16}$$

This formulates the relative improvement of the particular action to the average performance of the current policy. [22]

Agent training algorithms either use the policy (2.9) directly, an implicit representation utilizing value functions (2.10), (2.11) or a combination of both to formulate the actual objective they are aiming to improve.

[4] shorthand for $s_t \sim p_\theta(\tau), a_t \sim \pi$ and means, that the state was sampled following the transition rules of the environment when taking actions according to the policy π

On-policy and Off-policy Differential So far, the derived functions above always assumed, that the experience in τ, that is used to compute their values, was generated using the current policy[5]. This assumption is required for the family of *on-policy* algorithms, which need to sample new trajectories each time their policy (i.e. the generating trajectory distribution) changes. This is typically the case after each optimization step.
A distinction to that approach are the *off-policy* methods. They are able to improve the policy, even when the samples in the trajectory that are used for optimization, are in fact not from the joint trajectory distribution of the current policy ($\tau \nsim p_\theta(\tau)$ for the current θ).

2.1.5. The RL Objective

The goal of reinforcement learning is to find an optimal policy π^*, that maximizes the achievable reward over the full trajectory. This means, that the agent must not only be able to choose actions that yield good immediate feedback, but also to actively select actions that may result in bad immediate performance in order to achieve higher future reward. The central objective of finding this optimal policy is achieved by searching the space of stochastic policies. Using Eq. (2.8), we can define the *optimal policy* as the result of the optimization

$$\pi^* = \arg\max_\pi J(\pi) = \arg\max_\pi \mathbb{E}_{\tau\sim\pi}\left[\sum_t R(s_t, a_t)\right] \tag{2.17}$$

where (2.17) uses the finite return definition of (2.5), but also holds for the infinite horizon case when using the discounted return definition of (2.6).

Both, value (2.10) and Q-function (2.11) have optimality conditions similar to the optimal policy (2.17). The *optimal value function* and *optimal action-value function* and their Bellman formulations are:

$$\begin{aligned} V^*(s_t) &= \max_\pi \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R_t|s_t\right] \\ &= \max_{a_t} \mathbb{E}_{s_t\sim p_\theta(\tau)}\left[R(s_t,a_t) + V^*(s_{t+1})\right] \end{aligned} \tag{2.18}$$

$$\begin{aligned} Q^*(s_t,a_t) &= \max_\pi \mathbb{E}_{(s_t,a_t)\sim\pi}\left[R_t|s_t,a_t\right] \\ &= \mathbb{E}_{s_t\sim p_\theta(\tau)}\left[R(s_t,a_t) + \max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})\right] \end{aligned} \tag{2.19}$$

Like their non-optimal versions, they define the expected return under predefined state (and action) conditions but always act according to the optimal policy afterwards. As in (2.12), the optimal value can be expressed w.r.t. the optimal Q-function:

$$V^*(s_t) = \max_{a_t} Q^*(s_t, a_t) \tag{2.20}$$

With the equations above, the RL objective (2.17) can alternatively be expressed in terms of the value function (2.10):

$$\pi^* = \arg\max_\pi V_\pi(s_t)\Big|_{t=0} = \arg\max_\pi \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0\right] \tag{2.21}$$

[5] This was e.g. indicated by the subscripts $\tau \sim \pi$, $\tau \sim p_\theta(\tau)$, $(s_t, a_t) \sim \pi$ for the expectations or by π for the value functions respectively

by a maximization under the expectation of the earliest possible state, given that the initial state $s_0 \sim p_0(\cdot)$ follows the initial state distribution. [18]

Since in DRL the policy is depending on the parameters θ and we ultimately want to define an algorithm that tells us how to change these parameters, all operators (like the expectations or maximizations) depending on π can also be formulated equivalently with respect to θ.

2.1.6. Policy Optimization

Many different algorithms and methods have been developed to train agents on the reinforcement learning objective. Most of these methods can be divided into categories of how they attempt to improve behavior:

- *Policy gradient*: as the name suggests, these methods try to apply gradient-based optimization to the RL objective directly. We will cover them in more detail in Sec. 2.1.6, since we use a method from this family in the agents of Chapter 3 and 4.
- *Value-based*: the value-based methods try to estimate either a good Q-function (2.11) or value function (2.10) of the optimal policy. Typically, such methods only learn and improve the policy implicitly (e.g. as an *argmax* of a Q-function). They are covered shortly in Section 2.1.6
- *Actor-critic*: is a hybrid between policy gradients and value methods. They learn any of the value function representations and use them to improve the policy (typically by calculating a better gradient of the policy).
- *Model-based*: as mentioned in model comparison of Sec. 2.1.1, they try to learn a transition model and then use it to either plan through the environment (without an explicit policy), or to improve the policy by applying one of the above methods.

The reasons for so many different methods are, that each of them introduce different trade-offs (like stability and sample efficiency) or use different assumptions (like continuous or discrete states / actions, finite or infinite horizons, etc.). The type of algorithm, that shall be used, is a design choice and also depending on the type of problem. [18]
We will give an introduction to the ideas behind the first two categories here, since they represent the fundamental ideas of behavior optimization.

Policy Gradient Methods

Policy gradient methods are one of the general categories of model-free RL algorithms and attempt to directly calculate a derivative of the objective in Eq. (2.17) with respect to the parametrization θ of the policy. Some of the fundamental algorithms in this category are e.g. REINFORCE [23], Natural Policy Gradient [24] or Trust Region Policy Optimization (TRPO) [25]. The general approach of such methods work by computing an estimator of the policy (π) or objective ($J(\pi)$) gradient and applying a well-known stochastic gradient ascent (SGA) algorithm to it. They are more straight forward than the other methods since they are principled, i.e. they directly optimize the explicit policy and thus tend to be more stable and reliable. However, because of their structure they are mostly on-policy, which results also in less sample efficiency. [17]

The analytical gradient of the RL objective can be derived using:

1) the log-probability of the trajectory (2.3) under the finite horizon formulation (2.2): [22]

$$\log p_\theta(\tau) = \log p_0(s_0) + \sum_{t=0}^{T} \Big(\log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \Big) \tag{2.22}$$

2) the log-derivative trick combined with the chain rule to get the derivative of $p_\theta(\tau)$ with respect to θ: [18]

$$p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau)\, \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta\, p_\theta(\tau) \tag{2.23}$$

3) the combination of (2.22) and (2.23) to get the gradient of the log-probability of the trajectory:

$$\nabla_\theta\, \log p_\theta(\tau) = \sum_{t=0}^{T} \Big(\nabla_\theta \log \pi_\theta(a_t|s_t) \Big) \tag{2.24}$$

Neither the initial state probability $p_0(\cdot)$, nor the transition probability $p(s_{t+1}|s_t, a_t)$ of the environment in (2.24) are depending on θ and become zero in the differentiation.

The most common policy gradient estimator can then be formulated by applying Equations (2.22)–(2.24) (which implies a finite horizon)[6] to the cost function $J(\theta)$[7] (2.8): [22, 26, 18]

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta\, \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \\
&= \int_\tau \nabla_\theta p_\theta(\tau) R(\tau)\, d\tau \\
&= \int_\tau p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) R(\tau)\, d\tau \\
&= \mathbb{E}_{\tau\sim\pi}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau) \Big] \\
&= \mathbb{E}_{\tau\sim\pi}\Big[\Big(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \Big) \Big(\sum_{t=0}^{T} R(s_t, a_t) \Big) \Big]
\end{aligned} \tag{2.25}$$

Because in reality the expectation is evaluated using a finite number of samples / trajectories, we can get an unbiased sample estimate of (2.8) as

$$\begin{aligned}
J(\theta) &= \mathbb{E}_{\tau\sim p_\theta(\tau)} \Big[\sum_t R(s_t, a_t) \Big] \\
&\approx \frac{1}{|C|} \sum_{\tau\in C} \sum_t R(\tau) = \frac{1}{N} \sum_n \sum_t R(s_{n,t}, a_{n,t})
\end{aligned} \tag{2.26}$$

by collecting a set of $C = \{\tau_n\}_{n=0}^{N}$ trajectories with $|C| = N$ using the policy π_θ and averaging the rewards over the number of collected trajectories.

[6] Note that this derivation scheme also applies to the infinite horizon case

[7] We define $J(\pi) \overset{!}{=} J(\theta)$ but replace the (parametric) policy in the argument with the parameters θ to indicate, that our true goal is to adapt the parametrization of π in a favorable way

With the sample estimate (2.26) and analytical derivative (2.25) of J, we can get the *empirical* estimator for the policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_n \left[\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_{n,t}|s_{n,t}) \right) \left(\sum_{t=0}^{T} R(s_{n,t}, a_{n,t}) \right) \right] \tag{2.27}$$

which can then be applied to any SGA algorithm:

$$\theta_{m+1} = \theta_m + \eta \, \nabla_\theta J(\theta)\big|_{\theta_m} \tag{2.28}$$

For example, consider the vanilla policy gradient algorithm REINFORCE [23]:

REINFORCE

1: **for** each parameter update of θ **do**
2: sample $\{\tau_i\}$ using $\pi_\theta(a_t|s_t)$
3: compute gradient estimate $J(\theta) \approx \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \, R(\tau_i)$
4: perform policy update : $\theta \leftarrow \theta + \eta \, \nabla_\theta J(\theta)$
5: **end for**

This only requires to represent the policy in a way to be able to calculate $\nabla_\theta \log \pi_\theta(a_t|s_t)$.

The problem with the basic policy gradient estimator (2.27) in practice is, that it suffers from very high variance when calculating the update steps of a finite sized batch C. Figure 2.4 explains this visually: if the agent collects new experience and (if ever) reaches the same state at time t as in a trajectory collected before, it will likely select different actions in some of the subsequent states due to the randomness in the MDP and policy itself. This creates a totally different trajectory despite the same starting conditions. That is why advances in policy gradient algorithms mainly focus on reducing the variational dependence on the samples.
There can be much more said about the approaches to stabilize and improve the vanilla policy gradient, like causality assumptions, importance sampling, natural gradients, baselines and more, but this would go beyond the scope of this introduction. Instead, we refer to more detailed sources like [18, 17, 24, 27, 26, 28], that study, outline and elaborate on these topics in more detail.

Proximal Policy Optimization The Proximal Policy Optimization (PPO) algorithm [26] is one of the recent and more advanced approaches in gradient-based optimization methods. We will cover the ideas behind PPO here shortly, because we use it in our trained agents later in Chapters 3 and 4. The Section is based on [26] and [22, Algorithms].
PPO is an on-policy actor-critic algorithm based on TRPO with a regularized objective function to prevent performance collapse. It takes advantage of many of the improvements mentioned in the last part of the general derivation of policy gradients above. There are two versions of PPO in the original implementation of [26], that change the basic objective in (2.26):

– *Clipped surrogate loss*: this formulation introduces a surrogate objective to prevent, that the policy updates move the parameters too far away from the original distribution:

$$J_{\text{CLIP}}(\theta, \theta_k) = \mathbb{E}_{\tau \sim \pi_{\theta_k}} \left[\min \left\{ \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} \hat{A}_{\pi_{\theta_k}}(s,a) \, , \; g(\epsilon, \hat{A}_{\pi_{\theta_k}}(s,a)) \right\} \right] \tag{2.29}$$

where g is a clipping function:

$$g(s, A) = \begin{cases} (1+\epsilon)A, & A \geq 0 \\ (1-\epsilon)A, & A < 0 \end{cases} \tag{2.30}$$

The hyperparameter (HP) ϵ expresses roughly how far the new policy π_θ is allowed to move away from the current policy π_{θ_k}.

– *Adaptive KL penalty objective*: the original TRPO used a hard constraint on the Kullback-Leibler divergence (KL)[8] of the old and new policy. Instead, the PPO formulation uses a regularization approach:

$$J_{\text{PEN}}(\theta, \theta_k) = \mathbb{E}_{\tau \sim \pi_{\theta_k}} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \hat{A}_{\pi_{\theta_k}}(s_t, a_t) - \beta_k \, KL(\pi_{\theta_k}(\cdot|s_t), \, \pi_\theta(\cdot|s_t)) \right] \tag{2.31}$$

with an adaptive coefficient β for the KL penalty term to ensure update steps, that do not diverge too much from the old trajectory distribution. This keeps the policy update close to the last one. β is then changed for the next policy update according to a predefined target d_{target} of the divergence:

$$\beta_{k+1} \leftarrow \begin{cases} \beta_k / 2 & , \; d < d_{\text{target}} / 1.5 \\ \beta_k \times 2 & , \; d > d_{\text{target}} \times 1.5 \end{cases} \tag{2.32}$$

with $d = KL(\pi_{\theta_k}(\cdot|s_t), \, \pi_\theta(\cdot|s_t))$.

Both versions use an *advantage estimate baseline* (indicated by $\hat{A}$) instead of the cumulative reward to obtain a better target and reduce variance. The critic part of the network is responsible for fitting the advantage estimate using a regression target (see Section 2.1.6).
Ultimately, the parameters are updated as in Eq. (2.28):

$$\theta_{k+1} = \arg\max_\theta \mathbb{E}_{\tau \sim \pi_{\theta_k}} \left[J(\theta, \theta_k) \right] \tag{2.33}$$

Value-based Methods

Value-based optimization is covered here briefly, since it is (together with the gradient methods) one of the more fundamental approaches to optimize behavior. There are multiple types of algorithms in this family like policy iteration, Temporal Difference learning (TD), Fitted Value Iteration or Q-iteration.
A purely value-based algorithm either uses the value-function (2.10), Q-function (2.11), or a combination (e.g. (2.16)) as a proxy for the policy. They only *indirectly* optimize behavior by training a value function to satisfy a self-consistency condition. Because of that, the algorithms incline less stability due to many failure modes they can develop during training. On the other hand, they are often more sample efficient than gradient-based methods, since most of them can be found in the category of off-policy algorithms. [22, Part 2]

The optimal action in each state is represented by the connection between the used value function (e.g. A_π (2.16)) and the policy π:

$$a_t^* = \arg\max_{a_t} A_{\pi_\theta}(s_t, a_t) \tag{2.34}$$

[8] KL divergence is a measure for the dissimilarity of two probability distributions: $D(P||Q) = KL(P, Q) = \int p(x) \log \frac{p(x)}{q(x)} dx \geq 0$

This is why we don't need to optimize the policy explicitly to determine optimal behavior. The actual policy that is executed in each state can be expressed as:

$$\tilde{\pi} = \begin{cases} 1 & \text{if} \quad a_t = \arg\max_{a_t} A_{\pi_\theta}(s_t, a_t) \\ 0 & \text{otherwise} \end{cases} \tag{2.35}$$

and determines the action for the given state. [22, Q-learning]

A famous example is Q-iteration, which learns the approximator Q_θ for the optimal action-value function Q^*. Normally, the Bellman objective formulation (2.15) is utilized for these methods. Fitted Q-iteration typically collects sampled trajectories into a replay buffer $\mathcal{B}$ from which batches of experience are drawn at random using some distribution $\mu \sim P(\mathcal{B})$. The algorithm is off-policy, because it does not matter from which version of the policy the transitions in $\mathcal{B}$ stem from. The optimal policy is implicitly represented by a tabular or function approximator, that tries to minimize the deviation from its prediction to the current Q value by using for example a logistic regression cost on the TD error:

$$\mathcal{L}(\theta) = \mathbb{E}_\mu \left[\frac{1}{2} \left\| Q_\theta(s, a) - y \right\|^2 \right] = \frac{1}{2} \sum_n \left\| Q_\theta(s_n, a_n) - y_n \right\|^2 \tag{2.36}$$

where y_n is the target value for the n-th sample and derived (with some intermediate steps) from the optimal advantage value (2.16): [18]

$$y_n = R(s_n, a_n) + \max_{a_{n+1}} Q_\theta(s_{n+1}, a_{n+1}) \tag{2.37}$$

The subscript n refers to a sample from the buffer $\mathcal{B}$[9] and Q_θ corresponds to (2.11). We also want to emphasize with this notation, that $C(\theta)$ doesn't depend on an explicit representation of π. The Q-iteration algorithm basically consists of three steps and is given as:

Fitted Q-iteration

1: **for** collect dataset $\mathcal{B} = \{(s_n, a_n, s_{n+1}, R(s_n, a_n))\}$ of transitions **do**
2: **for** fixed number of updates K using the current $\mathcal{B}$ **do**
3: set (2.37): $y_n \leftarrow R(s_n, a_n) + \max_{a_{n+1}} Q_\theta(s_{n+1}, a_{n+1})$
4: update parameters: $\theta \leftarrow \arg\min_\theta \frac{1}{2} \sum_n \left\| Q_\theta(s_n, a_n) - y_n \right\|^2$
5: **end for**
6: **end for**

As mentioned before, a problem of value fitting methods is that no convergence guarantees can be made in the non-linear case, when function approximators like Deep Neural Networks (DNNs) are used to represent Q_θ. [18]
As before for the gradient-based approaches, there are also much more sophisticated ideas to make value based methods work better. Replay buffers are actually one of these advances already and not used in the original value-iteration algorithm. Some other examples are using (variations of) target networks, double Q-learning to mitigate value overestimation or multi-step returns to counter initial ill-conditioning of the value function. The reader can refer to [29, 18, 17] or *Q-learning* [30] and its deep learning equivalent DQN [31] for explanation of some advanced methods.

[9] We simplified the expression of the expectation of the sample buffer distribution μ in (2.36) to a collection process of some samples

2.2. Continual Learning

Biological networks (where DL often draws inspiration from) can acquire new knowledge over their whole lifespan and continually adapt to changing domains [12]. Continual learning is a field of machine learning, where an artificial learner shall gain the same ability of solving a series of (not necessarily) related, sequentially trained tasks, while alleviating the effect of forgetting old ones.
The desire for continual learning abilities in artificial connectionist models has a few orthogonal motivations. One is the ease of use: training a model only on the task at hand solely with the data available and without caring about other (older) use-cases or their performance. The convenience is, that neither the data of all preceded tasks must be stored, if a new problem shall be learned at some point, nor does the learner need to incorporate this data into the training process (which generally slows down training time significantly).
Another motive is the usage of *transfer knowledge*. In some way related tasks can share a common subset of useful representations in the model, that do not need to be re-learned to solve the new problem. This is the same effect as in *Transfer Learning*, but with the difference, that old knowledge is preserved instead of fine-tuned to the new task. If a model can exploit such behavior, it exhibits *positive forward*[10] */ backward*[11] *transfer*. Being able to use the same network for multiple problems in that way is likely to yield performance gains in training (convergence) speed and prediction accuracy.
This is especially of interest in (deep) RL, where learning from interaction towards a goal-directed target is primarily encouraged by the reward signal alone and successfully learned low-level concepts are expected to reoccur across many different tasks.[12]

Like biological brains, the ANN should exhibit a good balance between both: stability and plasticity. Stability is the ability to preserve learned knowledge, enabling the agent to perform good on the trained task in the future. Plasticity on the other hand, allows the model to acquire new knowledge, which is the prerequisite for continual learning. So far, both parts interfere with each other and concentrating on one completely degrades the other (stability-plasticity dilemma [32]).
More precisely, the challenge of continual learning is the shift of the data distribution of samples for each encountered task. When the input data distribution starts to diverge from the trained one, without having access to data from the previous task(s), the learner will adapt its parameters during training to the new distribution and override the old one. This is referred to as catastrophic forgetting and shown in Figure 2.5 for the simple case of a binary classifier. While the model learns a good decision boundary during the training of the first task, when the distribution shift occurs for the inputs of task 2, the model learns in general not the ideal new boundary as the best trade-off between task 1 and 2. Instead, the lack of stability leads to a decision boundary, that does not account for task 1 anymore.
Continual learning can pose different challenges in terms of available and required information and can therefore be formally grouped into different classes.

[10] Positive forward transfer means, that the model can utilize knowledge from previous tasks to solve the current problem more efficiently.

[11] Positive backward transfer means, that the model can reuse (some of) the new knowledge from current problem for old tasks. This is desired but unrealistic and also most of the time not needed, if the model is initially trained good enough.

[12] An example for this would be teaching a robot to pick up boxes first and later extending this task to also pick up other things. The (lower-level) capability of grabbing objects will be useful for both tasks.

2.2.1. Categories

Different categories of continual learning have been proposed and adopted [33, 34, 35], each defining a certain change of the *feature* or *task domain* of a task sequence. These categories were originally defined by [33, 34] on supervised learning tasks, but can be transferred to the RL case. Similar to MDPs, we can define tasks as a tuple of sets and operators:

$$T_i = (\mathcal{S}_i, P, \mathcal{A}_i, \pi, t_i) \tag{2.38}$$

where i indicates the specific task, $\mathcal{S}$ the set of possible states[13], $\mathcal{A}$ the set of all possible actions, $P(\mathcal{S})$ the marginal probability distribution of the environment and $\pi : \mathcal{S} \rightarrow \mathcal{A}$ the mapping from states to actions, which inherently represents the goal of the agent in this task. Finally, t describes the task identity.
The proposed continual learning categories always imply a certain context, to which the problem is related:

Incremental Task Learning (ITL) In the first scenario, the task identity t_i is known for the agent. This refers to the easiest continual learning scenario, since knowledge about the task label allows to have a model with task-specific components [34]. A multi-headed NN can be used to address each task individually by selecting the corresponding head of the output layer according to t_i.
In ITL, everything in the task tuple (2.38) is allowed to change between two tasks T_1 and T_2: the states in $\mathcal{S}$ can have different appearances, the behavior of the environment $P(\mathcal{S})$ or goal π of the agent can change and / or the available actions in $\mathcal{A}$ and their meaning can differ. The correct part of the model can always be identified during training and inference, when the task identity is known. This assumption however is also quite strong, because in many practical applications the task label is unknown in production or even during training.

Incremental Domain Learning (IDL) The difference of IDL to ITL is, that the task identity is unknown. Also, the continual learning task is restricted to only change the input domain. This means, that the structure of the environment can change either in its appearance or its behavior / logic or both, but the goal of the agent and the available actions stays the same for the tasks. Because of this, the task label is not required for the model to make predictions.
In IDL, at least one of either the state set $\mathcal{S}$ or distribution $P(\mathcal{S})$ change between two tasks T_1 and T_2, formulating $\mathcal{S}_1 \neq \mathcal{S}_2 \ \vee \ P(\mathcal{S}_1) \neq P(\mathcal{S}_2)$. At the same time, the action set $\mathcal{A}_1 = \mathcal{A}_2$ stays the same and therefore a single-headed model can be used to infer model responses for both tasks.
An RL example would be an agent trained to survive in game as long as possible (T_1). Applying this agent to a different game (T_2) with the same goal and set of possible actions refers to IDL. The agent does not need to identify, in which particular environment it currently is, since it can follow the same target and use the same actions.

[13] We use $\mathcal{S}$ here to denote any kind of representation for the information of a state to unclutter notation.

Incremental Class Learning (ICL) The name of this category originates from the supervised learning scenarios of [33, 34] and refers to the learning of entirely new classes by training on disjoint training sets of different domains. At the same time, like in IDL, the number and meaning of the model outputs does not change, but the task identity must be inferred together with the model prediction.
Translated to the RL case, this means that for the task sequence $\{T_k\}\forall k$, environment contexts ($\mathcal{S}_k$ / $P(\mathcal{S}_k)$) can change, but here also the goal π of the agent can vary for each task. However, the action set $\mathcal{A}$ does not change and is the union of all individual action sets $\mathcal{A} = \bigcup_k \mathcal{A}_k$. This implies, that a single-headed model can be used for each task, like in IDL.

2.2.2. Catastrophic Forgetting

Catastrophic interference [36], also known as catastrophic forgetting, is a phenomenon that occurs in artificial connectionist models like Neural Networks, whereby learning new information can lead to abrupt erasure of previously acquired knowledge [10, 11, 35]. In the context of the previous Sections is catastrophic forgetting the failure of stability in neural networks. This behavior is independent of the class (ITL, IDL, ICL) to which the learning problem belongs to.
When confronted with a sequential learning task (e.g. $\{T_1, T_2\}$), catastrophic forgetting manifest not until the information of the next task is tried to be incorporated into the model. If the learner delivered good performance after training on T_1, normally an abrupt functional degradation on that task can be observed, after the model is started to be trained on T_2. How much the network suffers from catastrophic forgetting on the previous task depends on the dissimilarity of the marginal data distributions of samples in task 1 and 2, i.e. the distribution shift as depicted in Figure 2.5. In the figure, the distributional shift from task 1 to 2 is rather small, resulting still in a decent performance on the first task (only a part of samples from class $y = 1$ is misclassified). Such a training process is shown schematically in Figure 2.6, where the test performance on task 1 is still above zero, despite training on task 2.

However, in general the distribution shifts are big and unpredictable, which leads to severe performance drops in practical continual learning applications. To systematically mitigate the effects of catastrophic interference is therefore currently a very active field of research. [35, 10, 1, 13, 12, 11, 14, 2]
In this work, we will especially elaborate and extend more on the continual learning approaches of [1] and [2] in Sections 3 and 4.

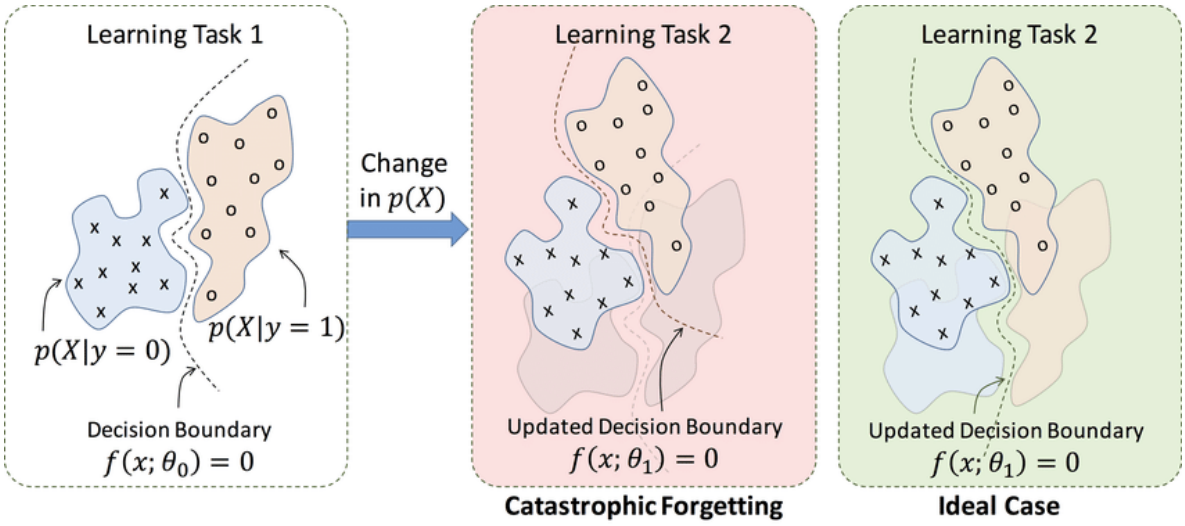


Figure 2.5.: Update of the decision boundary for binary classification tasks, when applying IDL continual learning. First, the model is trained on task 1, then a change of the marginal probability distribution happens in the transition to task 2. The effects of catastrophic forgetting and the optimal boundary for the task set are shown on the right. Figure reference is [37].

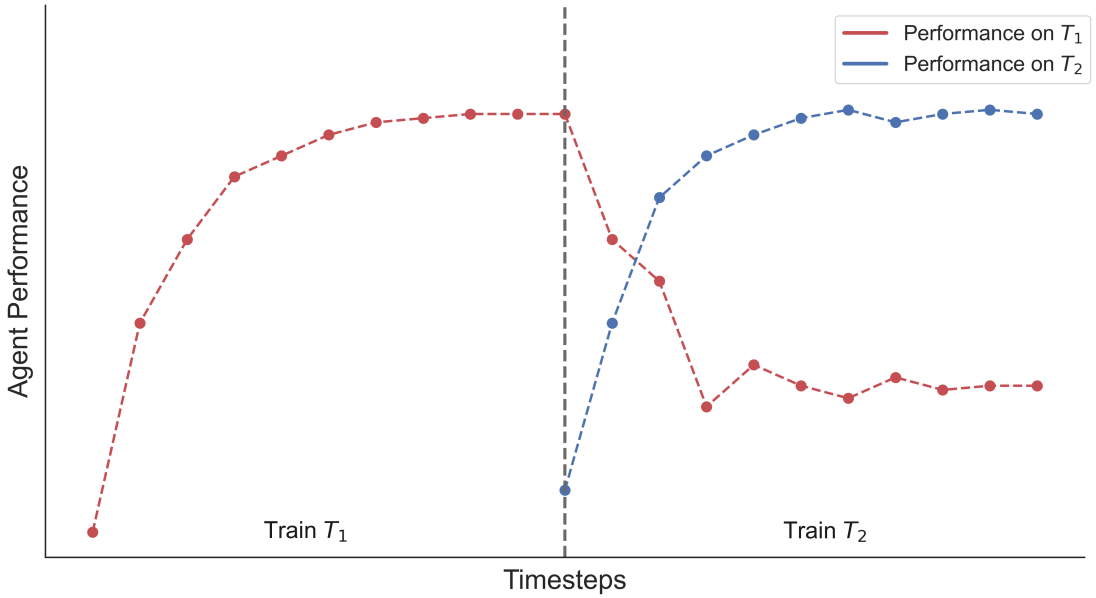


Figure 2.6.: Common trend of learner performance in the setting of continual learning. The agent performs well on the first task after training, but when trained on the second task, test performance on T_1 degrades rapidly.

3. Approach

This Section extends the theoretical Chapter by explaining the underlying concepts of the algorithms this work is based on in Section 3.1 and 3.2. Afterwards, these concepts are combined and extended in Section 3.3.

3.1. Progressive Neural Networks

PNNs [1] are an approach to progress towards the long-standing goal of continual learning by preventing catastrophic forgetting "by design". The underlying concept is a special network architecture, where the complete model is logically divided into *columns*. Each column is a complete sub-network of common structure and trained to solve one of the required tasks. While this alone would not yield any benefit compared to the training of a completely new model for each task, lateral connections (so called *adapters*) are added to each layer input (after the input layer).
Additionally to the previous layer of the same column, the adapters also connect the previous layers of all preceding columns to the current layer input. Transfer knowledge in the form of intermediate feature representations from old tasks can be accessed by the current column through these connections. The adapters are general single-layer MLPs used for initial conditioning and dimensional adaption, followed by a non-linearity Φ and preceded by an additional importance scaling factor α. The scaling factor is applied to the raw feature vectors of the previous columns and responsible for adjusting the different layer scales as well as learning the importance of intermediate layer features of previous tasks.

The logical structure of a PNN, as it is implemented in this work, is depicted in Figure 3.1a and shows the complete layout of the PNN. Graph 3.1b reveals the structure of a single *PNN block* used for each layer and column of the model. The PNN blocks combine the standard forward pass from its previous layer of the same column and the adapters, which are responsible for aggregating and combining the previous layer outputs of all preceding columns. The output layers in Fig. 3.1a needs special handling because of the *GAE*[1] [28, 26], which needs an additional head that returns an estimate of the value function. The value heads are *shared*, meaning that the value outputs use the same hidden blocks (layers) during the forward pass as the action prediction.
Since PNNs create new columns for each task, the most fitting category of continual learning is ITL because the sub-networks are capable of handling both: a change in the feature domain (where the data generating distribution changes) and in the task domain (where the label space and/or mapping function changes).[2] [35]

[1] **G**eneralized **A**dvantage **E**stimation is a recommended method to use in PPO, that learns to estimate the advantage function from the current state via a weighted sum of n-step return estimators

[2] Note, that the proposed PNN can also be easily extended further to handle a change in the feature space as well by using customized input layers for each column

Assuming the PNN was trained on c tasks, the forward pass of a layer l to the next layer $l+1$ in the last column c for the implementation in this work is given as:

$$\mathbf{h}_c^{(l+1)} = \Phi\Big(\mathbf{W}_c^{(l)}\,\mathbf{h}_c^{(l)} \;+\; \mathbf{A}_c^{(l)}\Phi\Big(\sum_{t<c}\mathbf{V}_{t:c}^{(l)}\,\alpha_{t:c}^{(l)}\,\mathbf{h}_t^{(l)}\Big)\Big) \quad , \mathbf{h}_c^{(0)} = \mathbf{x} \quad \forall c \tag{3.1}$$

where $\mathbf{h}_c^{(l)}$ is the input to the l-th layer in the c-th column, $\mathbf{x}$ the input trajectory sampled from the environment and $\Phi(\cdot) = \mathrm{ReLu}(\cdot) = \max(\cdot, 0)$ the non-linearity. $\mathbf{A}_c^{(l)}$ is the projection matrix required to map the sum of adapter networks from all prior columns onto the same feature space as the current layer output. For the adapter single-layer MLP the subscripts $t:c$ denotes the connection from column $t \in [0, c)$ to c. Additionally, the learned scaling parameter α is initialized from values $\{1, 0.1, 0.01\}$ for each adapter to account for different scaling of the feature spaces between the columns.

The ability to solve c problems independently and that no assumption of overlapping feature spaces between the tasks is made (as it is required for some functional approaches like *EWC* [10], where task similarity is required for the algorithm to lead to a stable performance) is a big benefit of PNNs.
In exchange, since the number of parameters stemming from the lateral connections in each additional column is in the same order as the model parameters for each sub-network $|\Theta^{(1)}|$, the total number of model parameters scale with the number of tasks c: $\sum_{n=1}^{c} n\,|\Theta^{(1)}|$.
Also, task labels must be known at inference time and the training order of the tasks can play a role, when positive forward transfer shall be maximized. Additionally, with increasing number of trained tasks the contribution of new columns to the agents decision can decrease, if a non-linear combination of prior columns can solve the new task sufficiently good, leading to poor capacity utilization.

It is shown in [1], that great forward transfer can be achieved for similar problems (e.g. for *Pong Soup*) and comparable performance could be reached for different Atari game constellations emphasizing the intuition, that forward transfer is limited for significant task variation ([1, Fig. 4, Fig. 6]), but on the other hand, that much knowledge can be reused for similar problems.

3.2. Memento Agent

3.2.1. Memento Experiment

Continual RL research (especially in the Arcade Learning Environment (ALE) [38]) often assumes, that tasks are defined per environment / game and that multi-task continual learning therefore corresponds to multiple games or game modes being trained sequentially. [2]
For continual RL, training on a whole environment was long time considered as a sufficient definition for task boundaries [10, 13, 1]. As this may be true for other DL areas like supervised learning[3], in RL however, it is often more difficult to solve a single environment, since the task the agent shall learn is often closely tied to the environment where it has to act in. On

[3] Where task boundaries can easier be identified and defined. E.g. in image classification one can easily select, exclude or pre-process the images in the training-set in order to hand-craft distinct task boundaries.

top of that, the agent normally has to perform multiple independent primary tasks to be able to successfully navigate through an environment and collect any reward [39]. As a result, *catastrophic interference* does not necessarily occur only when an agent is (re-)trained on a new environment (game) but can also arise inside a single environment.

To explicitly control for and overcome *catastrophic interference* inside a single game, [2] proposes the *Memento experiment*[4], in which *intra-environment tasks* are defined (similar to multi-task learning) as different stages of the game, where the requirements for the agent change significantly in order to make further progress beyond this stage. The task boundaries in this case are identified by plateaus in the agent performance (typically the return as a proxy measure), where even with more network capacity or training time the policy cannot improve further [2, Fig. 5]. Logically, one would expect such boundaries to appear at contextual changes in a game, like entering a different room or receiving a new objective. The work of [2, Ch. 3] shows, that especially in hard exploration games like in Atari `Montezuma's Revenge` or `Private Eye` such context changes appear, but even learning a new basic tasks can mean a big contextual change for the agent (since DL models normally don't have any deeper comprehension of higher level concepts or abstractions [40]). This is why non-exploration games like `Asterix` or `Breakout` show similar interference patterns during training. [2, Fig. 4]

If an agent suffers from such intra-environment interference effects and continual learning dynamics, modeling the game beyond the plateau during training leads to destructive policy updates, which negatively impact decision-making earlier in the game. This means, the agent starts to degrade its own policy, inhibiting further progress. To prevent the agent to interfere with the policy that led to the plateau, a new, identical agent (the Memento agent) is started from this position. This approach shown in Figure 3.3. After the first (base) agent stagnates in performance, a new Memento agent is started from that state.

In [2], the Memento agent is a completely new and independent Rainbow-CTS or DQN agent, launched from the exact state the base agent plateaued and continues to train from there. Since the new agent begins each episode from the final position of the first one, it trains exclusively on states beyond the plateau and is not limited by the constraint to perform well in earlier game stages too. The agent is able to update its policy without interfering with the original policy of the base agent, and thus can perform parameter updates independently. [2, Sec. 3]

3.2.2. Memento Environment

In this work, the Memento environment was created to enable the Memento agent described above to train on states beyond performance plateaus. As required for the Memento experiment, this environment provides the functionality of defining the start state distribution (see 2.1.2) and initializing arbitrary ALE environments with specific states (the *Memento states*) from a buffer (the *Memento state buffer*). This allows for training of subsequent agents, independently of the trajectory (and corresponding policy) required to end up in the Memento

[4] This is a reference to the psychological thriller where the protagonist suffers from amnesia and must deduce a reasonable plan without any memory of how he arrived at certain point or what his initial goal was. [2, footnote 2]

state. In contrast to the approach in [2], here the Memento buffer is generated during an additional policy evaluation step and contains one environment state from each sampled episode. The selection approach is shown in Figure 3.2b. During evaluation, for each observed state $s_{i,k}$ in the currently sampled trajectory $i : \tau_i = \{(s_{i,k}, a_{i,k}, r_{i,k})\}\big|_{k=0}^{t}$ of present length t, the up-to-date cumulative return (i.e. game score)

$$R_{i,max} = \max_t \left\{ R_i(t) \right\} \quad with \quad R_i(t) = \sum_{k=0}^{t} R(\tau_{i,k}) = \sum_{k=0}^{t} r_{i,k} \tag{3.2}$$

is calculated. Then, the Memento state $s_{i,max}$ of the current episode is identified, with the earliest time-step, that reached the maximum game score $R_{i,max}$ (i.e. the intra-game interference criteria):

$$s_{i,max} = \min_t \left\{ s_{i,t} \ : \ R_i(t) = R_{i,max} \right\} \tag{3.3}$$

The Memento state buffer $\mathcal{S}_{max}$ is filled with the state-return pairs $\{ s_{i,max}, R_{i,max} \}$ of independent trajectories, from which the Memento environment then samples its initial state from at each reset.

3.3. Memento-PNN Agent

The Memento-PNN agent combines the techniques of 3.1 and 3.2. This means, it uses a Memento environment for each new (intra-environment) task to generate ongoing experience beyond a performance plateau and has a PNN as its model, whose additional columns are then trained exclusively on trajectories starting from the Memento states.

The differences to the Memento agent in Section 3.2 are, that the Memento-PNN 1) is a gradient-based PPO agent; 2) integrates the ability to solve all tasks into a single model, i.e. does not spawn a new and independent agent for each intra-environment task; 3) is able to utilize and combine static transfer knowledge of *all* previous tasks instead of overwriting the last training accomplishments for each new problem; and 4) is trained on a variety of initial states from the Memento state buffer $\mathcal{S}_{max}$ as a starting point of the environment (which shall encourage the agent to generalize better at environment start), instead of continuing training from a single state.

3.3.1. Training Procedure

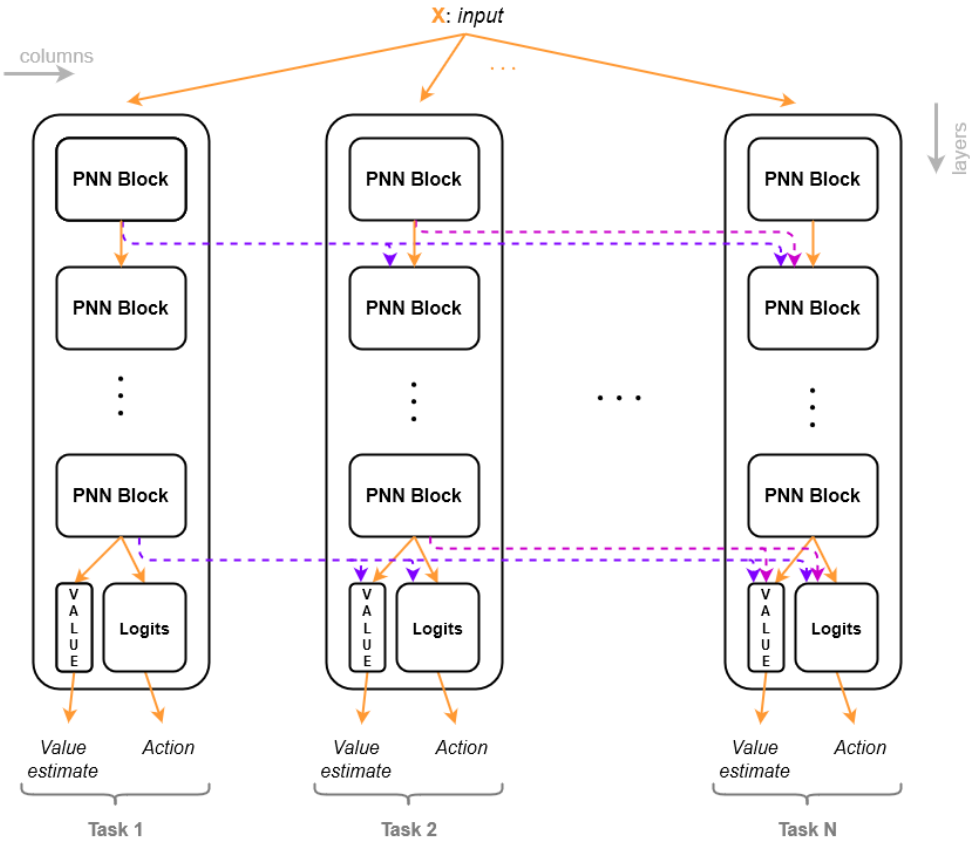
The training procedure for obtaining an intra-environment interference-free agent using the Memento-PNN method is similar to the one for the standard Memento experiment (3.2). As mentioned previously, because of the lack of intrinsic replay buffers in PPO, the training consists of three repeating independent stages:

Train until convergence The initial *base agent* is trained until a stable performance level is reached. Such a state is considered to be reached, when the learning curve of the agent plateaus and does not improve anymore, even with sufficient additional training time. The game score is used as a performance proxy for the agent. [2, Sec. 1]

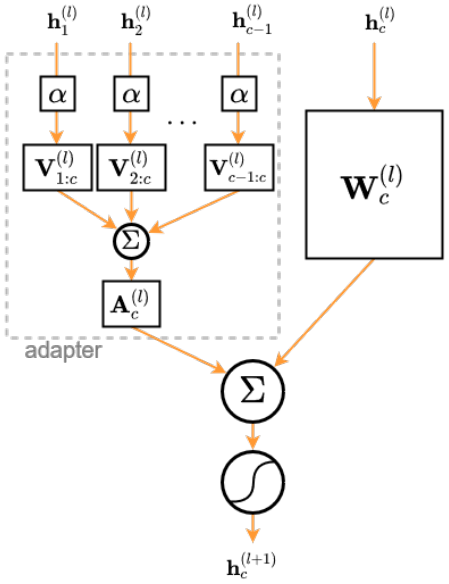
In this work, we determine the convergence state of an agent manually by supervising its reward progress. When a plateau is reached, the training is stopped and a checkpoint of the model is selected, where the agent reached maximum reward. The selection process is shown in Figure 3.2a. The model parameters from the checkpoint *on the plateau* are selected, where the agent showed best performance. These model weights of the current PNN column are then frozen for all consecutive tasks and used to evaluate the policy in the next training step.

Evaluate and create state buffer Opposed to Section 3.2.1, we identify Memento states by running a separate evaluation step. After the initial training is completed, the base agent / current column is evaluated on complete trajectories to generate the *Memento state buffer*, using the model parameters from the checkpoint of the previous step. This means, that we only end an episode when the underlying Atari environment reports termination. Because of that, we use ϵ-greedy action sampling with a very small $\epsilon < 0.002$ here to ensure, that the agent is able to escape certain states of the game, where it can get stuck (e.g. repeating the same action over and over again without making progress). The justification of ϵ-greedy sampling for agent evaluation is based on the findings in Appendix B.
After the episode is finished, the earliest state that yields maximum return $R_{i,max}$ is then a Memento state candidate $s_{i,max}$ and added to the Memento state buffer $\mathcal{S}_{max}$ (see Eq. (3.2) and (3.3)). This is schematically depicted for 4 episodes in Figure 3.2b: The maximum game score and associated Memento state are extracted for each of the episode trajectories and the Memento state buffer would contain 4 state-return pairs in this case. The size of the Memento buffer can be chosen freely, however, evaluating too few episodes can lead to biased start state distributions in the Memento environment. In this work, the buffer size was chosen to be at least 1000 to ensure a diverse set of start states.
The additional evaluation procedure after the actual training phase is required, since we use a policy gradient method (PPO) which – in contrast to the value-based Rainbow-CTS or DQN agents in [2] – does not have an intrinsic replay buffer.

Extend and retrain PNN The last phase is a slightly adapted version of the first one: each training is done by initializing the PNN model of the agent with a checkpoint from the previous training (or equivalently evaluation) step in a converged state and then extending it by one column. While the previous columns are frozen, the new column is trained on the next intra-environment task using the Memento state buffer created by the preceding evaluation step. The Memento environment uses the states in $\mathcal{S}_{max}$ to parameterize its start state distribution to train exclusively from the intra-game task boundary onward, which caused the agent to degrade the policy and inhibited learning. In contrast to the Memento agent, the Memento-PNN can not only train independently on the new task, but can also utilize and combine intermediate features from the previous columns to solve it as described in Sec 3.1. How a consecutive training of a Memento-PNN agent can be expected to look like is qualitatively shown in Figure 3.3 and is the same as for the original Memento agent of Section 3.2.1. The difference is, that the *same* agent is used to generate the second training curve by adding a new PNN column to the base agent.
To train the Memento-PNN agent on additional intra-environment contexts, the phases can be repeated by training the current task until a performance plateau is reached again, evaluating the new PNN column on the environment using the same Memento buffer and gathering new Memento states for the next task boundary.

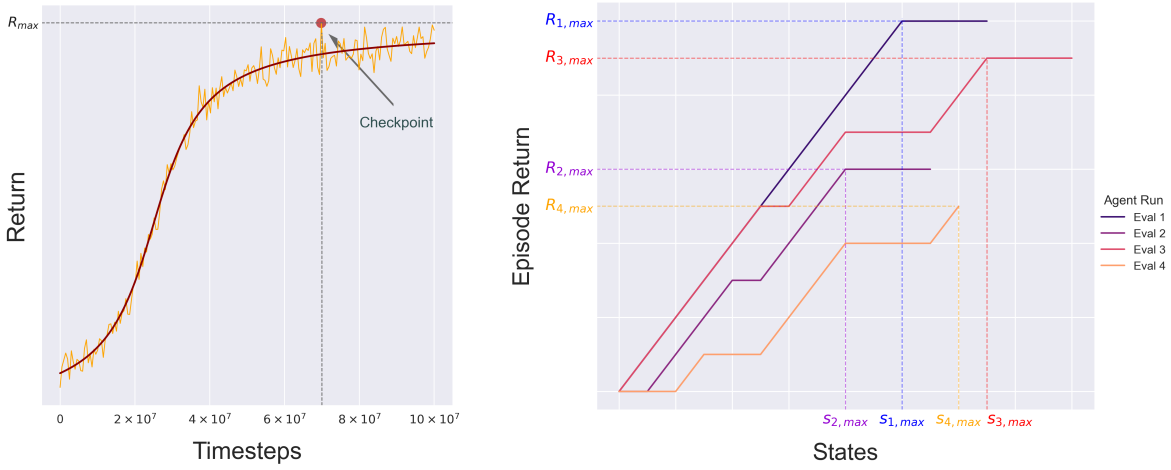


(a) Structure of an arbitrary PNN. Each encapsulating outer block corresponds to a column of the PNN, which is a complete sub-network on its own and capable of solving one task. Each hidden layer in each column consists of a special PNN block described in Fig. 3.1b. The violet arrows depict the *residual connections* from each column to all successive ones and are present in each layer of the model. These connections allow transfer knowledge to propagate to subsequent tasks.



(b) Layout of a PNN block in the l-th layer of an arbitrary column c. The adapter Multilayer Perceptron (MLP) learns the transfer importance of the preceding columns and combines it with the forward pass of the current task. Notations are according to Equation (3.1).

Figure 3.1.: PNN network and adapter structure



(a) (b)

Figure 3.2.: (a) Selection scheme for the checkpoint at the intra-environment interference state of the Memento(-PNN) agents. The checkpoint is selected by the earliest best agent performance in a converged state (performance plateau).

(b) Generation procedure for the Memento Buffer showing 4 exemplary evaluation episodes. The episodes can vary in length and performance. In each episode, the evaluated trajectory is stored as return-observation tuples for each timestep. Finally, the first observation that yields maximum reward is added to the Memento state buffer.

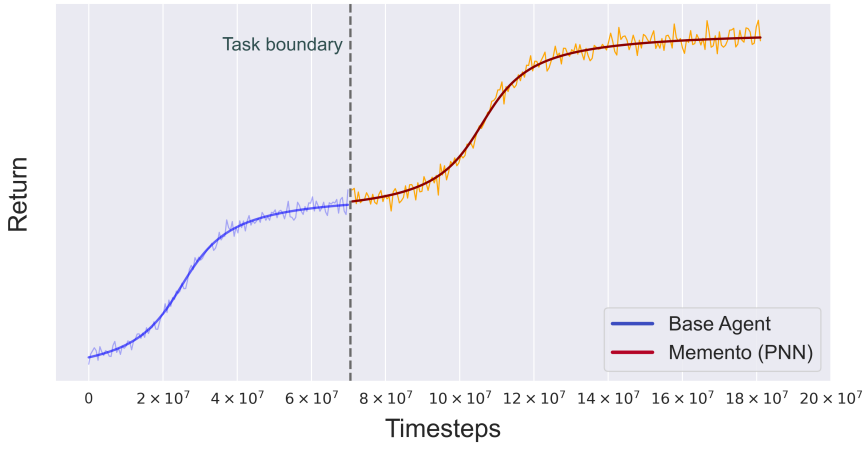


Figure 3.3.: Exemplary training curves of the base and Memento agent or the base and next column of an Memento-PNN agent respectively. After best model parameters in the converged state of the agent are identified according to Fig. 3.2b and the state buffer is created, a new agent (Memento agent) or column (Memento-PNN agent) is trained starting from the intra-environment task boundary.

4. Methodology

In this Chapter, the findings on continual training and catastrophic intra-environment interference are presented. Before that, the experiment setup and used environments are described and the baselines and individual approaches are shown.
All agents are trained using PPO in combination with exchanging parts of the original algorithm with the methods described in Chapter 3.
We refer to the different approaches as:

- **PNN Agent**, if we only substitute the model of an agent with a PNN. Each new environment (i.e. task) will be represented by a new column in the model.

- **Memento Agent**, if we train a standard PPO agent (or equivalently a PNN agent with only a single column) on a Memento environment. Training is then performed according to Section 3.2. A new agent is used for each task, which is initialized with the checkpoint of the previous task as determined by Eq. 3.3.

- **Memento-PNN Agent**, if we combine the Memento environment with the PNN agent to train it on multiple (intra-environment) tasks as described in Section 3.3.1.

All algorithms are conducted on the Atari ALE suite [4] for a standardized way of evaluation and comparison to other papers and use the returned images as observations (partial observability).

4.1. Frameworks and Setup

4.1.1. Framework

Because model-free RL is highly sample inefficient [5, 41] and for the purpose of continual learning many models must be trained, an efficient scaling and parallelization framework had to be selected in order to train (and re-train) the agents in a reasonable time. The used frameworks in this work are covered here shortly.

Ray is a parallelization framework for building distributed applications [42]. It provides a universal Python API for job assignments to parallel actors, automatic worker management and communication synchronization. It also integrates additional libraries for machine learning applications such as *Tune* and *RLlib*, making it a good choice for rapid prototyping with high efficiency demands.

Tune is a hyper-parameter optimization library [43], that can launch multi-node distributed HP sweeps for the same learner and integrates with many optimization libraries and algorithms, such as *Bayesian Optimization* or *HyperBand*. Additionally, it provides automatic visualization capabilities using TensorBoard [44].

RLlib is an open-source library for reinforcement learning and used for implementing the algorithms showcased in Section 3. It is built on top of *Ray* and can either use *Tensorflow* [44] or *PyTorch* [45] for its auto-grad backend. It must be mentioned, however, that since continual learning is not considered nor implemented in RLlib, more effort was needed to integrate this functionality into the Ray/RLlib framework and much time of this thesis was spent on implementing and adapting the standard agents to the required problem formulation and structure. The models for this work are implemented in PyTorch.

4.1.2. Experiment Setup

In order to create comparable results over the different environments in the ALE suite, a common set of wrappers for preprocessing the observations of each environment is used. The raw Atari games are wrapped by the *deepmind* wrapper[1], where observation images are processed by a sequence of filters before they are passed to the agent. In the Ray framework, the following filter classes are summarized under the deepmind wrapper:

- **NoopReset** performs a random number of "NOOP"s at the start of each episode. This shall randomize the start state.
- **MaxAndSkip** skips a given number of frames (usually 4) between each returned observation. For the skipped frames, the initial action is repeated, their reward is accumulated and the observation is returned that yields maximum reward.
- **EpisodicLife** soft-terminates an episode when a life is lost. The environment continues to run even when a life is lost, but the wrapper reports end-of-episode to the subsequent processors.
- **FireReset** executes the "fire" action, when the environment is reset. This is necessary for some games to start.
- **WarpFrame** rescales the images to a fixed size and converts it to gray-scale.
- **FrameStack** stacks the last k observations together (typically $k = 4$), introducing some temporal information. This is necessary, since one frame is ambiguous (e.g. no movement information).

The experiments are run on a SLURM cluster[2], which allocates submitted jobs dynamically to one of 4 Nodes (servers) depending on the load and available resources.
All experiments use one primary worker process for model training and seven parallel sample workers for experience collection. The sample workers synchronize the model weights after each parameter update with the primary worker. The training worker runs on a GPU for

[1] Under this name, a fixed set of preprocessing wrappers are chained together, which were used in the same way in the DeepMind DQN paper [31, Methods].
[2] SLURM is a cluster management and job scheduling system for Linux clusters (see SLRUM official website; accessed at 11/2020).

efficient backpropagation, while the sample worker only use the model for inference and don't have GPU's assigned to them.

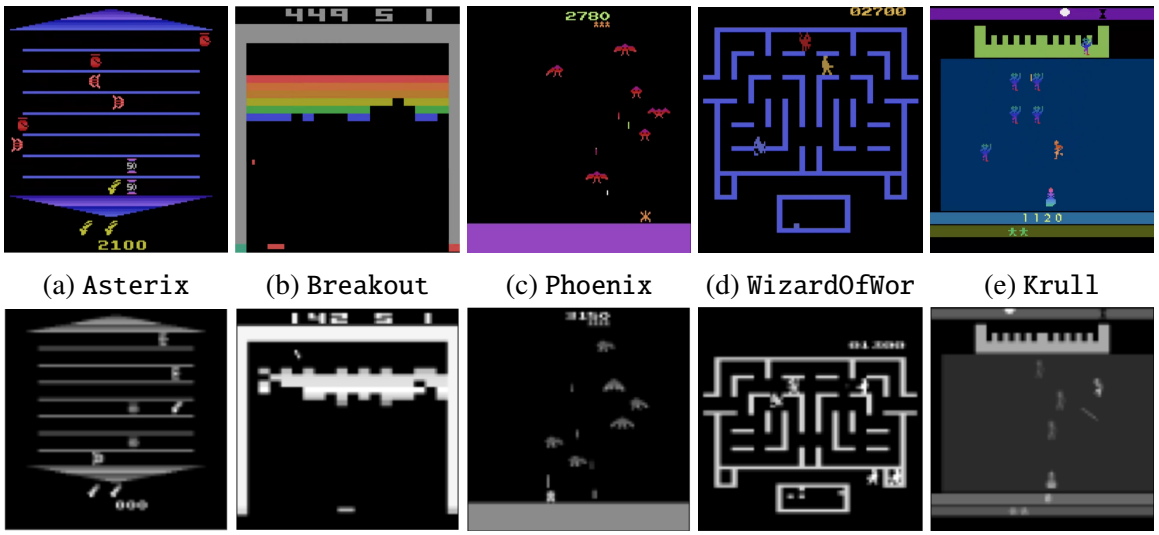
The base agents are always trained until convergence (regardless how many time-steps it takes), where the model parameters are then stored. After the checkpoint, the base agent and continual methods are resumed for a fixed number of additional 100M timesteps.

4.1.3. Game Overview

Due to time limitations, the experiments were carried out on a selected sub-set of Atari games, that offer a range of increasing task complexity and context variety, both in terms of observations (visual representation) and action strategy. The proposed algorithms are tested on the Atari games `Asterix`, `Breakout`, `Phoenix`, `WizardOfWor` and `Krull`. For each game, the base agent is trained until it reaches a performance plateau, where the respective Memento state buffer is created. In the following, the games considered in this work are briefly explained.

`Asterix` In Atari `Asterix`, the objective is to collect as much pots (and later other objects) on multiple layers of horizontal lanes, while avoiding the enemies (harps). Over the course of the game, the pots give more points. Objects and enemies are spawning randomly on a lane and side and are moving horizontally to the other one. The player can move freely on and in between lanes and has three lives. As the game progresses, all objects and foes become gradually faster. A raw game frame is shown in Fig. 4.1 (a) as returned by the ALE, as well as one of the stacked frames from a processed observation, when the wrappers from Sec. 4.1.2 are applied.
`Asterix` is one of the simpler games chosen here. While the screen environment and appearance of the enemies stay the same over the course of the game, only the reward objects change. This lets us expect to have a limited amount of intra-game interference and therefore a lower benefit from the proposed algorithms compared to the baselines.

`Breakout` The game was inspired by the predecessor `Pong`. A game frame is shown in Figure 4.1 (b) along with an agent observation. The objective is to rebound the ball as long as possible, while hitting all colored bricks top third of the screen. At each hit, a brick is destroyed and rewards the player with one point. The ball bounces from each edges of the screen (gray area) except the bottom. When the agent misses the ball, one of the five lives is removed and it has to press the "fire" button to continue. While the game progresses, the ball gets faster.
It is expected, that less intra-environment interference occurs in this game, since it does not introduce some logical task boundaries neither has different game stages. The screen environment stays the same for the whole game and the objective never changes. Less benefit from the presented algorithms is anticipated here, since the subjective amount of diverse subtasks is lower.



(a) Asterix (b) Breakout (c) Phoenix (d) WizardOfWor (e) Krull

Figure 4.1.: Raw emulator images for all the investigated games (top) and one of the stacked gray-scale images from the observation wrapper, which is passed to the model of the agent.

Phoenix Atari Phoenix is a fixed shooter and similar to the popular game *Space Invaders*. The player controls a spaceship, that moves horizontally at the bottom of the screen and has to shoot enemies hovering over him, which can in turn fire back. There is no time limit, but the player can only proceed to the next stage when all opponents are defeated. The game consists of multiple levels, in which the appearance of the enemies changes. The last level is a boss fight. Figure 4.1 (c) displays a section from the original game without any filters at the top, while the bottom shows one of the stacked frames from an observation.
Phoenix is a more difficult game compared to Asterix or Breakout, because it requires different approaches for the normal and boss levels, i.e. the agent must learn context and objective changes inside the same game.

WizardOfWor This game is a maze shooter hybrid, where the objective is to shoot all enemies (called Worlings) and not get shot by them. There are different kinds of foes, some can turn invisible. After clearing out all opponents in the maze, the player has to fight a boss. Defeating the boss allows the player to advance to the next stage, where the maze layout changes and the objective repeats. If the boss can escape from the maze using one of the two exits, it forces the player to fight an even harder boss (the Wizard of Wor), which can teleport around the maze. As the game progresses, the pace of the enemies picks up gradually.
This game presents a bigger challenge in terms of contextual changes inside the same environment. Catastrophic interference is expected to occur more likely, since the agent has to deal with different screen layouts, opponents and their behavior across various levels. A clip of one of the stages is shown in Fig. 4.1 (d).

Krull The game is based on the science fantasy movie Krull from 1983 and takes place on four separate screens. It is a mix of weak exploration and fighting game and unites multiple completely different game stages and associated tasks. As for the others, a raw game image of the first level and filtered training observation are depicted in Fig. 4.1 (e).

The different contexts (levels) for `Krull` are quite diverse and described in detail in appendix C.1. This environment poses the biggest challenge among the chosen ALE games. With each completed level, the goal and with it the agent objective changes, which requires the agent to learn an almost unrelated sub-task. Even further, while [2] reports noticeable performance gains for all games above, the Memento agent for `Krull` *decreases* by approximately 4% in performance compared to the base agent. [2, Fig. 4]

4.1.4. Model Structure and Parameters

Here, the general model structure and settings used for the trained baseline and Memento/-PNN agents are presented. All models share the common base architecture and HPs for comparison. We use the standard PPO algorithm with a shared action and value network (GAE) for each task. As PPO does not roll out complete episodes, we collect less correlated episode fragments from a total of 35 independent environments running on seven workers in parallel. The remaining set of HPs for the PPO configuration is shown in Table 4.1. The reward and action values are clipped to prevent excessive over-estimation.

The networks of the agents share the same number of convolutional (Conv) and fully-connected (FC) layers. Each column of a Memento-PNN agent has this base structure too, with the difference, that additional adapter sub-networks are added to each layer to establish the forward transfer as described in Sec. 3.1. Every collected sample is preprocessed using the *deepmind*-wrapper from above. The network architecture is shown in Table 4.2. For the convolutional layers, also their respective filter specification is shown. Note, that we use zero-padding in front of the convolution layers to adjust for the correct image size. This results in a final input image size of 88×88 pixels for the observation samples. The rectified linear activation $\mathrm{ReLu}(x) = \max\{0, x\}$ is used in the convolutional layers and identity mapping for the network heads to get the logits output.

We trained three different agent types with respect to total parameter count: Baseline and standard Memento agents, which are equivalent to single-column PNNs and have the same number of parameters; Memento-PNN agents trained on two intra-environment tasks, which use a PNN with 2 columns; and Memento-PNNs with three contexts and columns. Their parameter count and estimated model size (in MB) is shown in Table 4.3. A full overview of the model architectures can be found in Appendix A.

Table 4.1.: HP configuration for all agents

Discount factor γ	0.99
GAE - λ	0.95
Learning rate	5.0×10^{-5}
Batch size	5000
Rollout (episode fragment) length	100
SGD batch size	500
PPO Clip value ϵ	0.1
PPO KL coefficient β	0.5
PPO KL target d_{target}	0.01
PNN importance α initialization	0.01

Table 4.2.: Base network architecture for all agents. For Conv layers, values are equal for all dimensions, thus only one value is shown. Zero-padding is used to adjust the input sizes.

Layer	Output Size	Spec			
		Activation	Filter Depth	Kernel	Stride
Conv	21	ReLu	16	8	4
Conv	11	ReLu	32	4	2
Conv	1	ReLu	256	11	1
FC (Action head)	*# actions*	Identity	–	–	–
FC (Value head)	1	Identity	–	–	–

Table 4.3.: Model capacities $|\Theta|$ for all agents

Model	Network Capacity	Trainable Parameters	Estimated Size
Baseline	1,006,394	1,006,394	4.60 MB
PNN (2 columns)	3,147,982	2,141,588	13.81 MB
PNN (3 columns)	5,422,482	2,274,500	23.71 MB

4.2. Experiments

This Section reports the results on the combined measures for overcoming catastrophic interference in intra-environment training introduced in Chapter 3. For that, we train Memento-PNN agents and then compare the results in terms of performance to Memento agents as well as the effect of inter-task interference to standard PPO agents trained on the complete environment.

4.2.1. Agent Performance

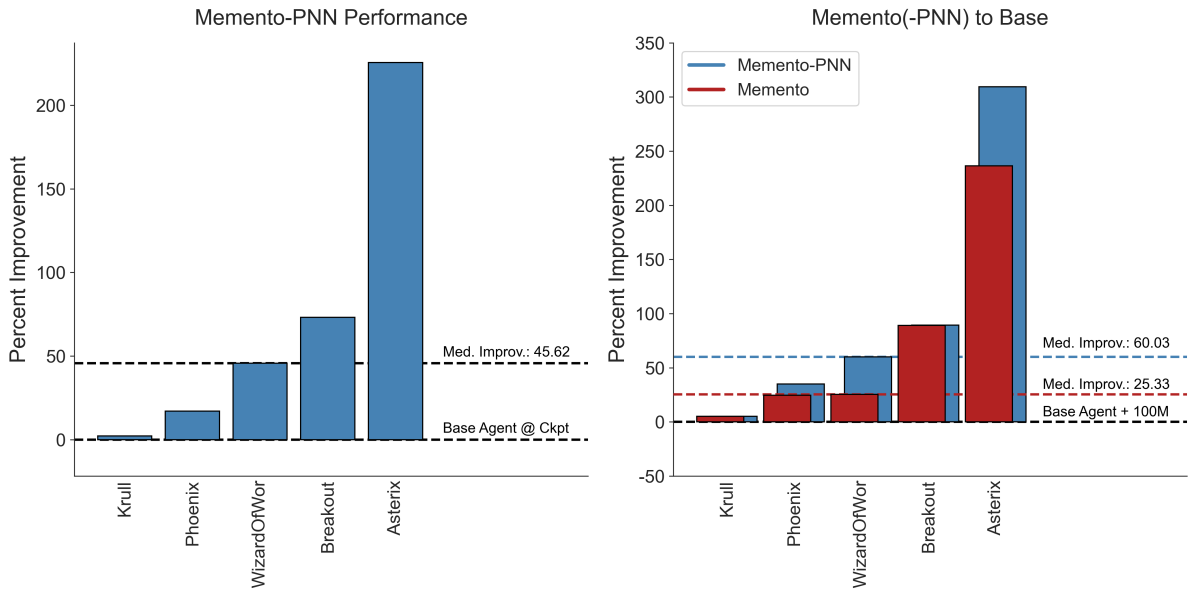
In this Section, we evaluate the performance of our proposed Memento-PNN agent on intra-environment continual learning tasks. For reference and comparison, we train a baseline agent and a Memento agent from Section 3.2. The standard Memento approach, using the same training algorithm, is included here, since comparison to the proposed Memento agent in [2] is hard, due to the disparity of value- (DQN) and gradient-based (PPO) training methods and the significant difference in obtaining the Memento buffer.
Table 4.4 shows the performance results, when we train the three different agents on the stated games of the ALE suite. The timesteps for the checkpoint of the Memento(-PNN) agents are chosen based on peak performance in a converged state (scheme of Fig. 3.2a) and shown (in million steps) next to the agent reward at that time. Starting from the timestep of the checkpoint, all agents are trained for an additional 100M environment steps and the mean return of the final 1M steps is reported in the table. The averaging of the final steps is also the reason why the reported performance of the continued base agent is in general slightly

lower than the peak snap-shot performance at the checkpoint.
The Memento and Memento-PNN agents are trained by continuing from the checkpoint, according to their respective approaches, also for 100M timesteps. The reported game score is overall way higher for these agents, showing a significant performance gain over the base agent, that was trained on the whole environment in one run. This indicates the presence of intra-environment interference as defined in [2], which can be mitigated by the advanced algorithms.

As shown in Figure 4.2a, the Memento-PNN method leads to an overall increased agent performance, even when we compare the absolute peak performance of the base agent to the averaged final 1M timestep return of the Memento-PNN. The evaluation of the training results on all showcased games yielded a median performance boost of +45.62%, showing a clear benefit of training agents on different contexts inside the same environment.
Figure 4.2b aligns the ultimate reward achievements of the normal Memento agent with the Memento-PNN. This time based on the *final* base agent performance, as stated in the second column of Table 4.4. In our experiments, even with a much smaller set of evaluated ALE games, we were able to reproduce approximately the same overall performance gain (+25.33%) for the Memento agent as originally reported in [2, Fig. 4] (+25.08%). While the advantage of the Memento-PNN approach over Memento agents without constant access to prior task information is smaller in some games, the median improvement difference between the two agent types is still +34.7%.

Table 4.4.: Performance of the base agent compared to the single-column Memento Agent and the Memento-PNN agent on different Atari games. The *base agent at checkpoint* column states the achieved return at the specific timestep where the agent performance peaked and the checkpoint was taken from. The advanced agents are initialized from this checkpoint. All other columns report the respective agent performance when trained for additional 100M environment steps after the checkpoint and the return is averaged over the final 1M steps.

	Agent Performance			
	Base (@ checkpoint)	Base final	Memento	Memento-PNN
`Asterix`	4 428.0 (57.4M steps)	3 519.1	11 838.6	14 413.7
`Breakout`	288.9 (162.4M steps)	264.2	499.6	500.0
`Phoenix`	6 752.6 (41.3M steps)	5 852.0	7 298.2	7 903.0
`WizardOfWor`	7 714.0 (35.7M steps)	7 019.5	8 797.0	11 233.1
`Krull`	9 336.6 (60.2M steps)	9 073.3	9 547.6	9 548.6



(a) Performance gains of the Memento-PNN compared to its base agent at peak performance (checkpoint) for all games. Median reward increase is at ~ 45%.

(b) Performance comparison between the Memento and Memento-PNN agent based on the final base agent performance. Median performance difference between the agents is ~ 35%.

Figure 4.2.: Relative improvement of the continual agents

For `Asterix`, `Phoenix` and `WizardOfWor`, the Memento-PNN method also yields further performance boosts over the standard Memento agent between +10.3 and +34.7%. For `Breakout` and `Krull`, there is no additional benefit. This was already expected for `Breakout`, since the game is considered to be relatively static and without many different contexts to learn (as pointed out before). The access to previous game contexts is not of much benefit for the agent in this case, because the new PNN column doesn't need to learn a new context of the game, that diverges much from the previous one. On the other hand, it is surprising for `Asterix`, that there are clear benefits when using a PNN model, since we rank the game to the same level of complexity as `Breakout`. Our guess is, that the appearance change of the collectibles over the course of the game is enough of a context difference to actually benefit from previous columns.

Of particular note here is the `Krull` environment. It was especially chosen, because of its weak exploration aspects[3] and because [2] reported a degradation of the Memento agent, resulting in less performance than the baseline. In the experiments both, the Memento and Memento-PNN agent benefit the least from their advanced structure but still perform slightly better (~ +5%) than the base agent. However, the trained `Krull` agents did not learn to play the game properly. Instead, the base, as well as the advanced agents, learned to exploit the game, using a known scoring glitch in the "spider den" level (see C.1 for further information). Noting, that even the base agent didn't learn to play the game correctly indicates, that not the tested Memento/-PNN approaches are the problem, but the underlying PPO algorithm, when trained on harder games (with e.g. exploration requirements). This is supported by the

[3] Weaker than compared to e.g. `Montezuma's Revenge` or `PrivateEye`.

additional methods, that are sometimes required to learn hard exploration environments with sparse rewards, for example enforcing the agent to follow initial trajectories from human demonstration as done in [39]. We also tried to train the agents on `Montezuma's Revenge` (a hard exploration game of the ALE suite), but could not get the agents to learn without using additional conditioning like for example the one mentioned before.

Overall, it is interesting to note that seemingly simpler games benefit more from the Memento/-PNN approaches, than more complex games like `Phoenix` or `WizardOfWor`, even though we would assume, that these algorithms are especially designed for being better in grasping new and unrelated sub-tasks inside an environment.

Also, a better training progression can be observed for most of the tested games. While the base agent mostly stagnates on the performance plateau and never overcomes the policy degradation happening there, the Memento and Memento-PNN agents reliably overcome this state and continue to collect more reward. The training progress curves for all games are shown in Figure 4.3. The point, at which the base agent was stopped, varies for each plot and corresponds to the timesteps stated in Table 4.4. This is naturally justified by the different amount of time each agent needs to reach a performance plateau. For all games, three to five equivalent base agents with different RNG seeds were trained to reduce training variance. The checkpoint was chosen at a performance peak from the best base agent.
For the training curves of `Asterix`, `Breakout` and `WizardOfWor`, a significant progress of the Memento (yellow) and Memento-PNN (red) agent compared to the continued base agent (green) can be observed. Unfortunately, for `Breakout` a data loss of the continued base agent occurred after about 5M timesteps indicated by the red $\times$.
In the progress curve for `Phoenix`, the gain of the continual learning agents is smaller and for `Krull`, there is hardly any benefit over the base agent (because of the mentioned reasons). The plots also emphasize, that the PNN-based model approach is always better or equally good (in the worst case) than the plain Memento agent.

The argument, that the Memento-PNN improvement is only attributed to more network capacity or not enough training time for the base agent can also be vitiated. Even a longer training time for the base agent does not yield comparable performance gains as shown in Appendix C.2, where we trained the an agent on `Phoenix` even longer than the defined additional 100M timesteps. As shown there, the performance gets even worse, indicating even more interference over time. In addition, [2] also showed, that increased model capacity will not lead to a comparable rise in performance either.

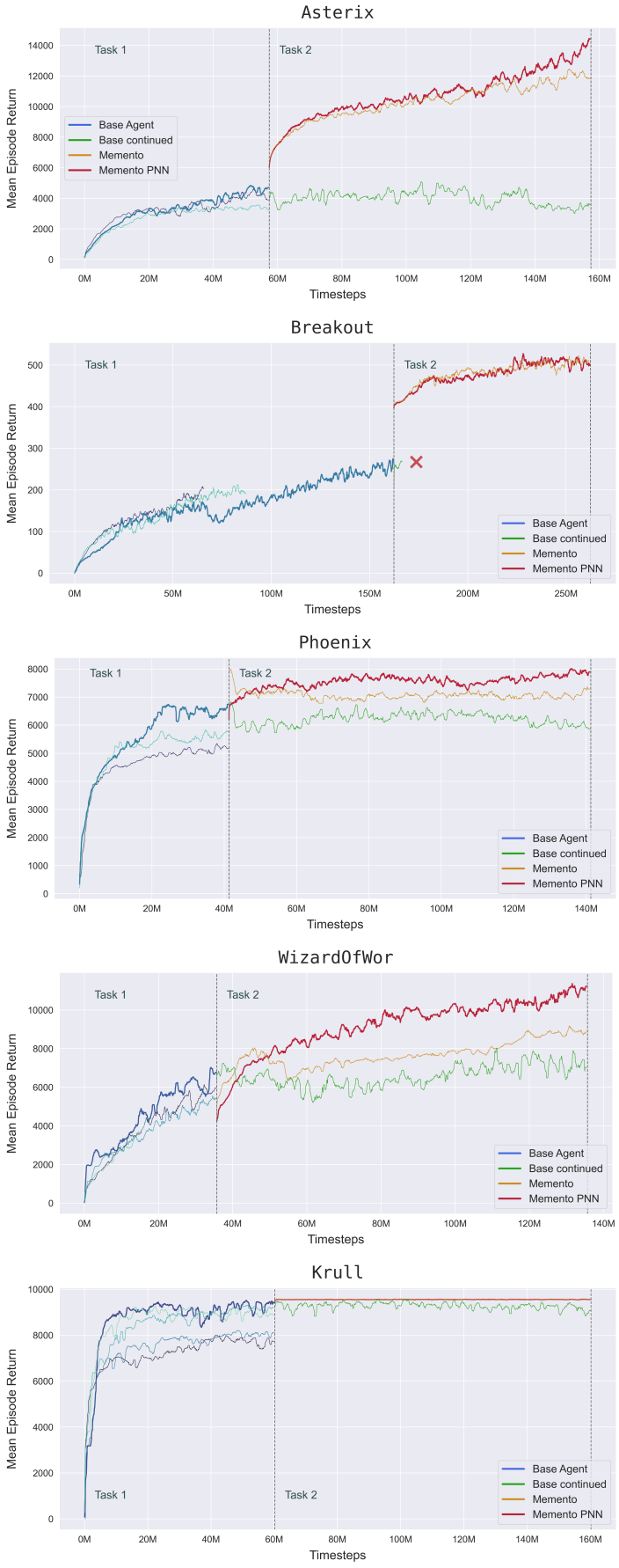


Figure 4.3.: Training curves of the baseline and all continuous agents. The thick line represents the best agent from each task. A median filter is applied to all training curves for better visibility. A red × indicates data loss. Best viewed in color.

4.2.2. Memento-PNN on three Contexts

To further test the limits of the proposed algorithm, the Memento-PNN is trained on an additional context for two of the selected environments. The training procedure is performed in the same way as outlined in the previous Section 4.2.1: the agents are trained for an additional 100M timesteps on third intra-game task by continuing from the last checkpoint of the previous one. For that, a third column of the PNN model is trained exclusively from the Memento states of the performance plateaus in context 2. The results are shown in Figure 4.4 for the Atari games `Asterix` and `Phoenix`. Table 4.5 supports the plots with the associated game scores, averaged over the last 1M timesteps of each task.

The training curve of `Asterix` shows great performance gains, even for the third task. While the base agent already degrades in the second context, the Memento-PNN continuously collects more reward with each new column added to the agent. The steep reward jumps in the beginning of each new training (gray vertical lines) can be explained by the random initialization of the new PNN column and its adapter MLPs. Transfer knowledge from the previous columns is available and the agent leans quickly to incorporate that knowledge through the adapters.
The progress plot for `Phoenix` in Fig. 4.4, however, does not indicate further improvement of the agent through incorporation of additional intra-game tasks. Even though the Memento-PNN agent shows still noticeable improvement in the second context, the third column ends up on approximately the same reward level as the previous one. This could be attributed to difficult initial states in the Memento buffer, but also due to the nature of the game itself. States of the boss fight in `Phoenix` are observed to often occur in the Memento buffer as a performance plateau. Because the boss fight is especially difficult and contextual different to the previous levels, the PPO agent struggles to learn a strategy for this level at all and rarely is able to progress beyond the boss stage.

The training progress plots suggest, that the performance of every next column of the Memento-PNN is approximately lower-bounded by the performance of the previous one. This is an important feature of the algorithm indicating that transfer knowledge is always useful for those agents and easier incorporated into the new column (via training of the adapters) than learning useful features from scratch.

Table 4.5.: Memento-PNN mean performance for the last 1M timesteps over all three contexts of `Asterix` and `Phoenix` respecively.

Intra-game task	Game Score	
	`Asterix`	`Phoenix`
Context 1	4 428.0	6 752.6
Context 2	14 413.7	7 903.0
Context 3	22 367.7	7 788.7

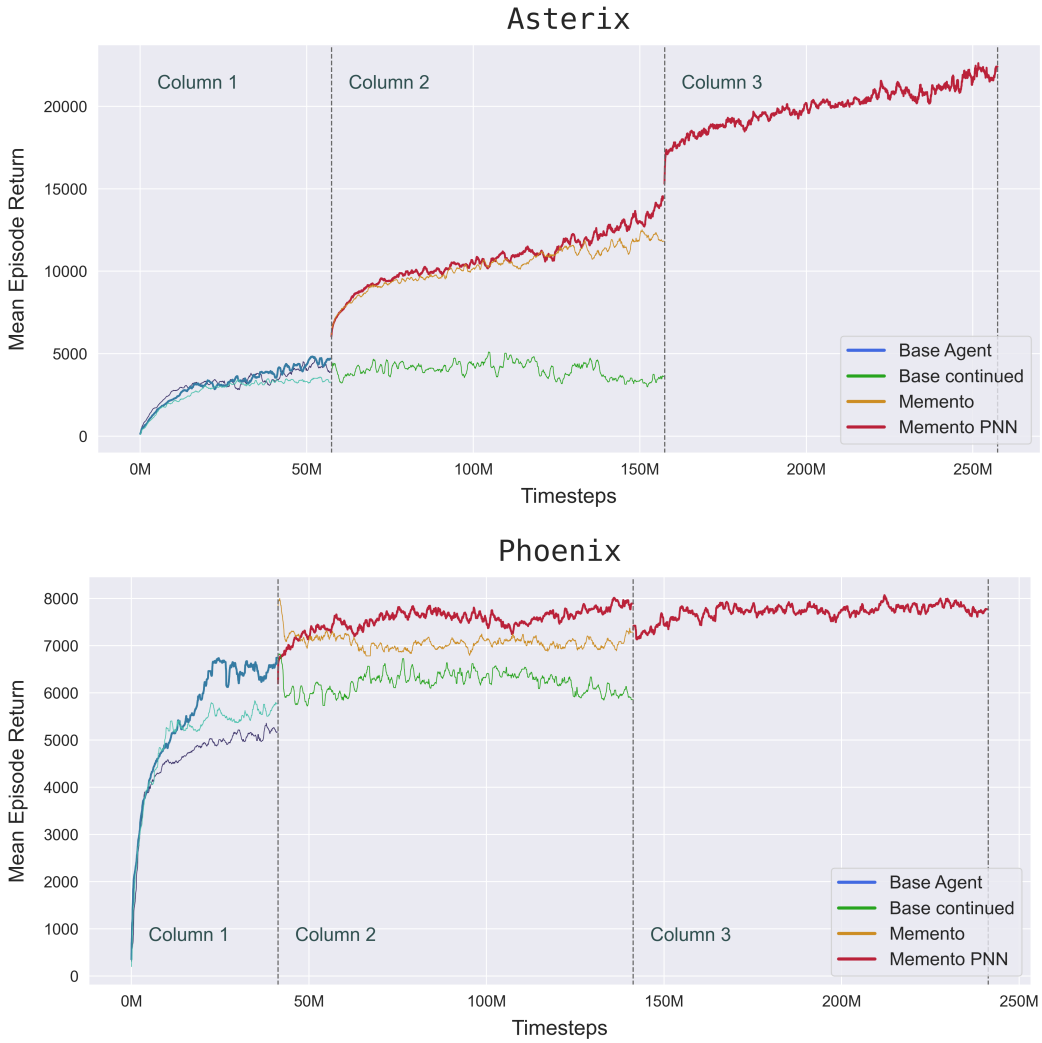


Figure 4.4.: Training curves of the Memento-PNN agent for three intra-environment contexts. The thick line represents the best agent in each context. A median filter is applied to all training curves for better visibility. Best viewed in color.

4.2.3. Intra-environment Task Identification

A big problem of the outlined Memento approaches is the difficulty to apply such agents outside of training. A trained Memento/-PNN agent would need to know at which state of the game which agent / column is responsible for sampling an action. While it is easy to identify the task boundaries during training by performance proxies like the agent return, no such control mechanism is available during real-time inference. The agent cannot sample a complete trajectory and evaluate it offline afterwards.

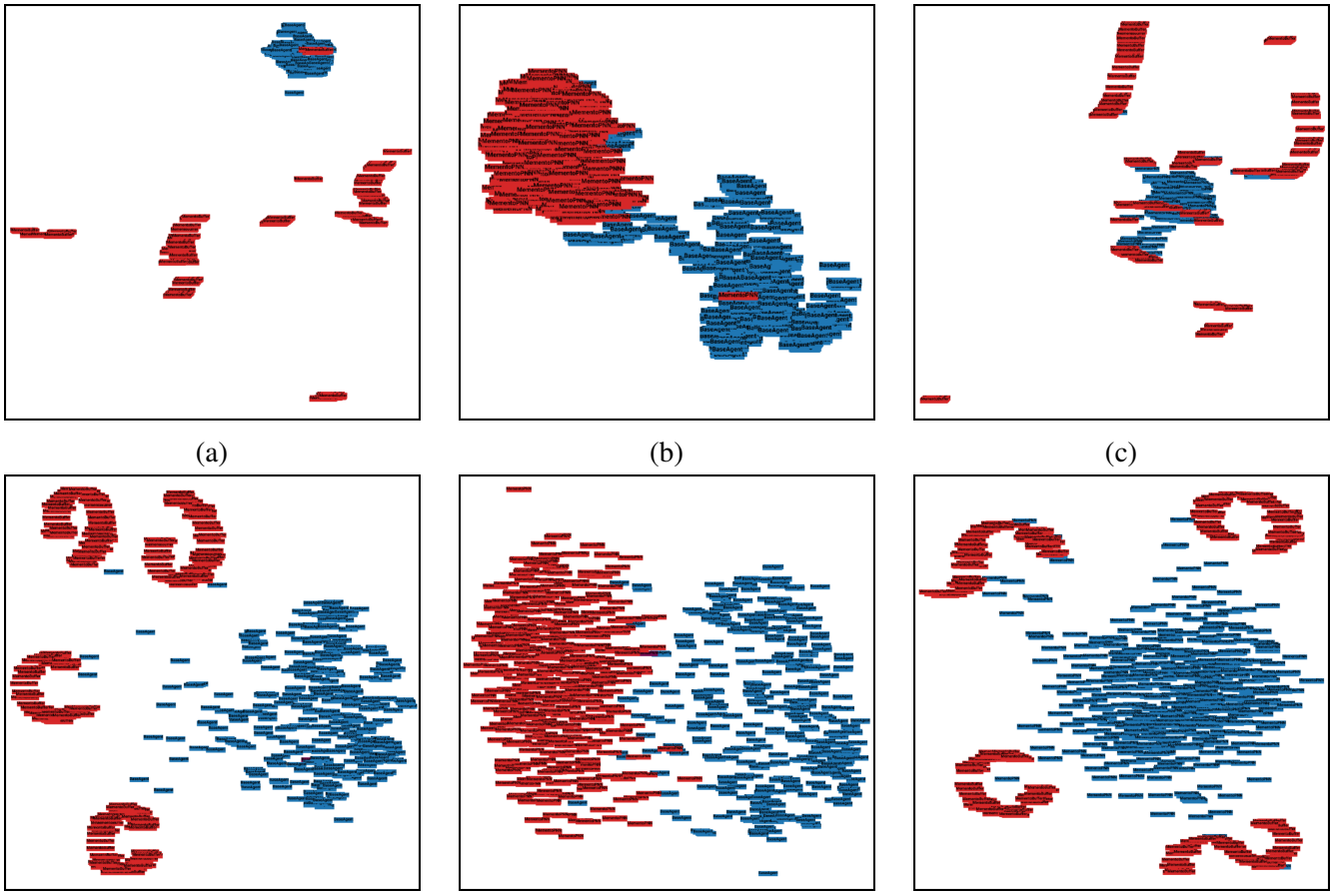
In order to see, whether the task boundaries can be identified automatically, the dimensional reduction techniques UMAP[4] [46] and t-SNE[5] [47] are applied to the observations of trajectories returned by the base agent, the Memento-PNN agent and its associated Memento buffer in all combinations. This provides a visual impression of the $255^{84\times84\times4}$ dimensional feature space.
The compilation of the results are exemplary shown for `Asterix` in Figures 4.5a – 4.5c. The dimension reduction visualizes the high-dimensional similarities of observations for one agent and disparity between the different agents / buffers. For the UMAP plot of the base agent against the Memento buffer (Fig. 4.5a (top)) or Memento-PNN agent (Fig. 4.5b (top)) respectively, the different classes group into (almost) linear separable clusters, indicating an easy way for identifying the intra-game task boundary. Even the Memento-PNN and its associated buffer (Fig. 4.5c (top)), from which the agent was originally started from, show some easy differentiation areas, even though this is naturally not as distinct as for the other two.
The t-SNE results are shown in the bottom rows. The algorithm needs some fine-tuning of its HPs in order to give a good visualization, but for `Asterix` the separation is even better visible than for UMAP. The t-SNE Plots 4.5a (bottom) and 4.5c (bottom) compare the two agents to the Memento buffer. They show, how the buffer states form clusters indicating that even for independent validation episodes during step two of the Memento-PNN training protocol (Sec. 3.3.1), the performance proxy as selection criteria still chooses Memento states s_{max} that share similar representation attributes and are therefore close in the state space.

The findings suggest, that automatic task discrimination by solely analyzing the current state / observation is possible. The distinction rules are agent and task specific and could be learned together with e.g. each column of the Memento-PNN agent.

For the rest of the games, the visualizations can be found in Appendix D. Not for all games showcased there, an easy distinction can be made based on the used mapping techniques. Especially, when the environment becomes more complex or the observations are very similar for different levels, their distinction becomes difficult. This can be observed for `Phoenix` (D.2) and `WizardOfWor` (D.3), where the mapped observations from both classes are completely intertwined.

Figure 4.6 presents some examples for observations from different areas of the UMAP visualization of the base agent and the Memento-PNN (Fig. 4.5b). The closer the samples of both classes are, the more similarities can be found between the observations.

[4] Uniform Manifold Approximation and Projection
[5] t-Distributed Stochastic Neighbor Embedding



(a) (b) (c)

Figure 4.5.: UMAP (top) and t-SNE (bottom) of `Asterix` for observation trajectories of:
(a) Base agent (blue) vs. Memento buffer (red)
(b) Base agent (blue) vs. Memento-PNN agent (red)
(c) Memento-PNN agent (blue) vs. Memento buffer (red)
For each class, 450 images were selected randomly from a pool of 10 000 observations. Best viewed in color.
HP for UMAP Epochs: 500, Neighbors: 15
HP for t-SNE Iterations: 500, Perplexity: 40, Learning rate: 10

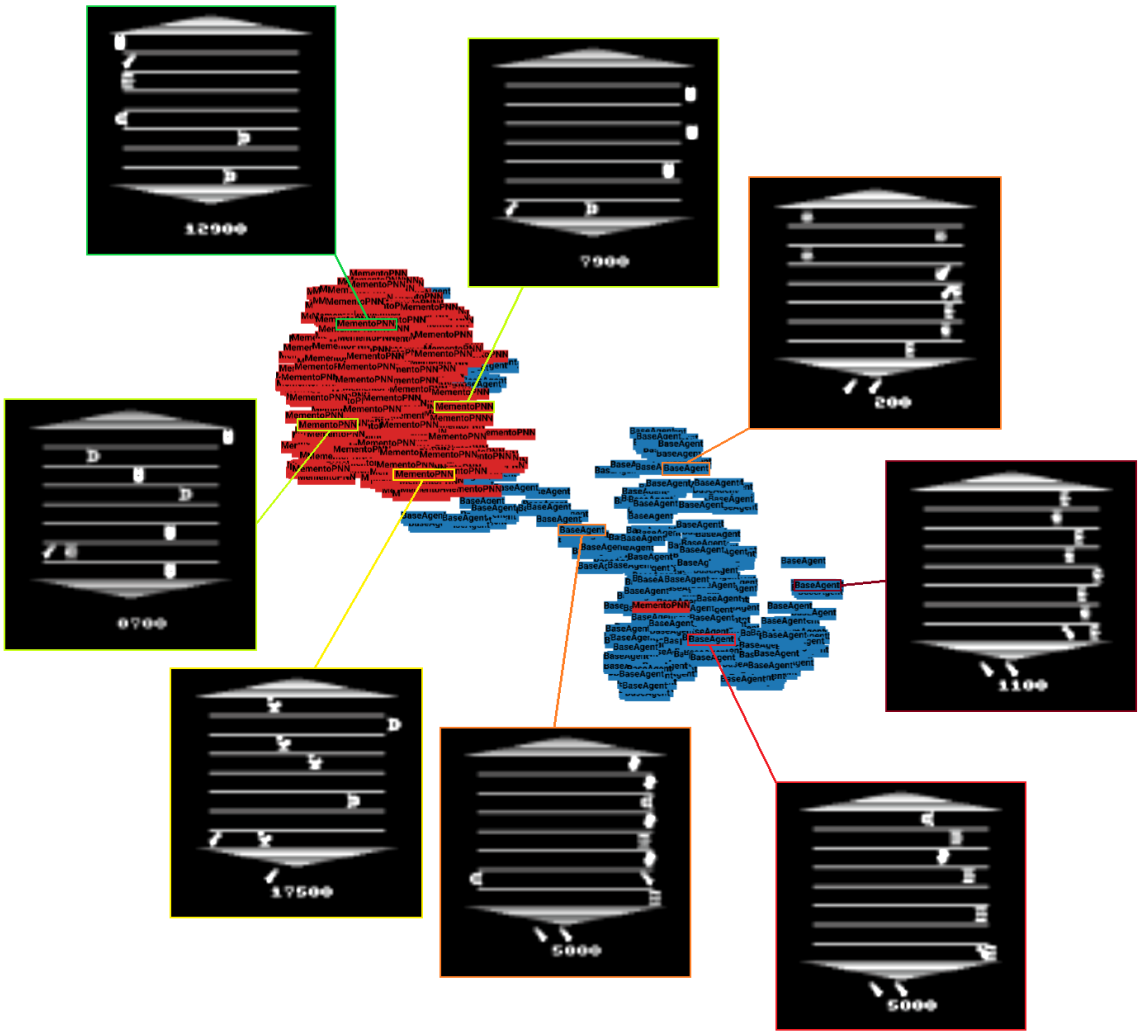


Figure 4.6.: UMAP with observations samples of the Baseline vs. Memento-PNN agent for Asterix. The mapping is the same as in Fig. 4.5b. Red labels mark the observations of the Memento-PNN, blue the experience of the baseline.

5. Conclusion and future work

5.1. Conclusion

Catastrophic forgetting in the continual learning domain is a major impediment on the way to life-long learning agents. This issue becomes even more relevant in the area of reinforcement learning, where knowledge is exclusively collected through consistent interaction and learning in a changing environment. However, recent research showed, that catastrophic interference not only occurs in multi-task learning, but also inside the same environment (task) of RL agents. [2, 12]

In this work, we presented a novel algorithm, the Memento-PNN, as a combination of the ideas in [1] and [2] to address catastrophic interference inside the same environment, rather than between isolated and independent tasks. The findings suggest, that the Memento training protocol combined with the PNN architecture effectively mitigates catastrophic intra-environment interference also for gradient-based models in a simple way by training a standard PPO agent, equipped with a PNN model, on Atari games of varying difficulty from the ALE suite.
The experiments come to the same conclusion as [2]: a significant amount of within-task interference normally happens in RL agents, which inhibits learning past a certain point. By using the Memento-PNN, the same model can successfully learn beyond the performance plateau and continue to improve its policy. By that, great performance gains for many of the tested environments could be achieved. The experiments also show, that the agent performance of the proposed method is naturally lower bounded by the compared baselines, due to its PNN architecture. The advanced architecture helps the agent to transfer knowledge to consecutive sub-tasks, resulting in a selective performance increase compared to individual agents trained according to the Memento experiment. The trade-off for this performance gain is increased model capacity, which grows with the number of tasks.
The advantage of the proposed algorithm is, that the intra-task boundaries are determined automatically during training and does not require human supervision or labeling. However, no such detection scheme is available in a production environment, and selecting the right column of the model there requires additional methods.

Finally, possibilities for an automatic task boundary identification were investigated. The visualization showed, that for some of the used environments, a simple mapping algorithm like UMAP or t-SNE suffices to already be able to linearly separate model inputs for the different columns of the PNN model. Although, tests on all the environments showed, that this approach is not general enough to suffice for every problem. Since the task distinction is problem dependent, using a simple classifier to determine the boundary could meet the requirements.

5.2. Future work

As mentioned in Section 4.2.3, the suggested Memento-PNN method is not usable in production yet. For this, a task discrimination method is needed in order to know, at what time / environment state which column of the PNN model must be active. Learning the task distinction cloud be done using a discriminator model, but the figures in Section 4.2.3 and Appendix D respectively imply, that even a linear learner could suffice in some cases.
A further direction to investigate could be, whether for observation clusters of different agent columns (game stages) – which are close or non-separable – any of the columns for the respective context would be eligible to make an action estimation.

To test the general validity and boundaries of the approach, the Memento-PNN method could be applied to the full set of problems in the ALE suite, or even harder challenges like robot simulations. When the environment gets more complex, it is expected that more intra-environment interference occurs and thus more columns are needed in the PNN model. For this, also an automated training framework becomes useful to reduce the manual work steps needed to train a new context in the agent. So far, the training procedure consists of three separate steps (see Sec. 3.3), which must be executed sequentially to train a new column. This could be combined into a single process without much effort.

Additionally, to verify that actual transfer knowledge is used, the same evaluation methods as in [1, Ch. 3] could be implemented[1], to test for transfer flows between columns. The work of [1] also figured out, that much capacity of new PNN columns may not be used, if a combination of old knowledge is sufficient to solve the current task. This suggests, that the problem of unbounded growth in the model parameters can be mitigated by controlling for column contribution to final model prediction. When identifying layers or whole columns not being utilized, network pruning could reduce excessive capacity usage. This property also emphasizes, to combine the Memento-PNN with NAS[2] methods (e.g. *Supernetwork search*) for each new column of the model, to optimize the utility of the transfer knowledge and reduce unnecessary network capacity.

As a prospect, we logically would assume that, provided having a PNN with enough columns trained on a wide range of diverse tasks, at some point the agent should be able to learn each additional task as a combination of all previous tasks. Hence eliminating the need for additional columns at some point and only requiring the fitted adapters.

[1] Average Perturbation Sensitivity (APS) and Average Fisher Sensitivity (AFS) are used there.
[2] Network Architecture Search

List of Figures

List of Tables

Bibliography

[1] A. A. Rusu *et al.*, "Progressive neural networks," Google DeepMind, p. 14, 2016. [Online]. Available: https://arxiv.org/abs/1606.04671

[2] W. Fedus *et al.*, "On catastrophic interference in atari 2600 games," Google Brain and University of Montreal, p. 18, 2020. [Online]. Available: https://arxiv.org/abs/2002.12499

[3] D. Silver, "Reinforcement learning," in *COMPM050/COMPGI13: Reinforcement Learning*. University College London and Google DeepMind, 2015 accessed at 11/2020. [Online]. Available: https://www.davidsilver.uk/teaching

[4] G. Brockman *et al.*, "Openai gym," 2016.

[5] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016.

[6] D. Silver *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017. [Online]. Available: https://doi.org/10.1038/nature24270

[7] O. Vinyals *et al.*, "Alphastar: Mastering the real-time strategy game starcraft ii," 2019, accessed at 11/2020. [Online]. Available: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii

[8] OpenAI *et al.*, "Dota 2 with large scale deep reinforcement learning," 2019. [Online]. Available: https://arxiv.org/abs/1912.06680

[9] cerberusd, "Paper review: Progress and compress: A scalable framework for continual learning," 2018, as at 11/2020. [Online]. Available: https://blog.lunit.io/2018/05/31/progress-compress-a-scalable-framework-for-continual-learning

[10] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *CoRR*, vol. abs/1612.00796, 2016. [Online]. Available: http://arxiv.org/abs/1612.00796

[11] F. Zenke *et al.*, "Continual learning through synaptic intelligence," *CoRR*, vol. abs/1703.04200, 2017. [Online]. Available: https://arxiv.org/abs/1703.04200v3

[12] C. Kaplanis *et al.*, "Continual reinforcement learning with complex synapses," 2018. [Online]. Available: https://arxiv.org/abs/1802.07239

[13] J. Schwarz *et al.*, "Progress and compress: A scalable framework for continual learning," 2018.

[14] D. Rolnick *et al.*, "Experience replay for continual learning," University of Pennsylvania and Google DeepMind, p. 14, 2019. [Online]. Available: https://arxiv.org/abs/1811.11682

[15] L. Espeholt *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," Google DeepMind, p. 22, 2018. [Online]. Available: https://arxiv.org/abs/1802.01561

[16] C. M. Bishop, in *Pattern Recognition and Machine Learning*, vol. 2. Springer, 2006.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[18] S. Levine, "Deep reinforcement learning," in *CS 285: Deep Reinforcement Learning*. UC Berkeley, 2020 accessed at 11/2020.

[19] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[20] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 04 1994.

[21] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*, 01 1996, vol. 27.

[22] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018. [Online]. Available: https://spinningup.openai.com

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[24] S. Kakade, "A natural policy gradient," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01. Cambridge, MA, USA: MIT Press, 2001, p. 1531–1538.

[25] J. Schulman *et al.*, "Trust region policy optimization," Separtment of Electrical Engineering and Computer Sciences, Berkeley, University of California, p. 16, 2017. [Online]. Available: https://arxiv.org/abs/1502.05477

[26] J. Schulman *et al.*, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[27] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, pp. 682 – 697, 2008, robotics and Neuroscience. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608008000701

[28] J. Schulman *et al.*, "High-dimensional continuous control using generalized advantage estimation," Department of Electrical Engineering and Computer Science, University of California, Berkeley, p. 14, 2018. [Online]. Available: https://arxiv.org/abs/1506.02438

[29] R. Munos *et al.*, "Safe and efficient off-policy reinforcement learning," Google DeepMind and Vrije Universiteit Brussel, p. 18, 2016. [Online]. Available: https://arxiv.org/abs/1606.02647

[30] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[31] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, feb 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[32] S. Grossberg, "How does a brain build a cognitive code?" *Psychological review*, vol. 87, pp. 1–51, 02 1980.

[33] H. Yen-Chang *et al.*, "Re-evaluating continual learning scenarios: A categorization and case for strong baselines," *CoRR*, vol. abs/1810.12488, 2018. [Online]. Available: http://arxiv.org/abs/1810.12488

[34] G. M. van de Ven and A. S. Tolias, "Three scenarios for continual learning," Center for Neuroscience and Artificial Intelligence, Baylor College of Medicine, Houston, p. 18, 2019. [Online]. Available: https://arxiv.org/abs/1904.07734

[35] F. Wiewel and B. Yang, "Localizing catastrophic forgetting in neural networks," *CoRR*, vol. abs/1906.02568, 2019. [Online]. Available: http://arxiv.org/abs/1906.02568

[36] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," ser. Psychology of Learning and Motivation, G. H. Bower, Ed. Academic Press, 1989, vol. 24, pp. 109 – 165. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0079742108605368

[37] S. Kolouri *et al.*, "Attention-based selective plasticity," HRL Laboratories, LLC. and University of California Irvine, p. 7, 03 2019. [Online]. Available: https://www.researchgate.net/publication/331609030_Attention-Based_Selective_Plasticity

[38] M. G. Bellemare *et al.*, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, p. 253–279, Jun 2013. [Online]. Available: http://dx.doi.org/10.1613/jair.3912

[39] T. Salimans and R. Chen, "Learning montezuma's revenge from a single demonstration," 12 2018. [Online]. Available: https://www.researchgate.net/publication/329568138_Learning_Montezuma%27s_Revenge_from_a_Single_Demonstration

[40] G. Marcus and E. Davis, "Gpt-3, bloviator: Openai's language generator has no idea what it's talking about," 8 2020, accessed at 10/2020.

[41] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 02 2015.

[42] P. Moritz *et al.*, "Ray: A distributed framework for emerging ai applications," 2017.

[43] R. Liaw *et al.*, "Tune: A research platform for distributed model selection and training," 2018.

[44] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[45] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," pp. 8024–8035, 2019. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[46] L. McInnes *et al.*, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020.

[47] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html

A. Network Architectures

Here we present the PyTorch implementation of the used networks as a sequence of consecutive layers and their respective output shapes and trainable parameters.

A.1. Network Structure of Baseline and Memento Agents

```
----------------------------------------------------------------
Layer (type)               Output Shape         Param #
================================================================
ZeroPad2d-1             [-1, 4, 88, 88]               0
Conv2d-2               [-1, 16, 21, 21]           4,112
SlimConv2d-3           [-1, 16, 21, 21]               0
ReLU-4                 [-1, 16, 21, 21]               0
PNNConvBlock-5         [-1, 16, 21, 21]               0
ZeroPad2d-6            [-1, 16, 24, 24]               0
Conv2d-7               [-1, 32, 11, 11]           8,224
SlimConv2d-8           [-1, 32, 11, 11]               0
ReLU-9                 [-1, 32, 11, 11]               0
PNNConvBlock-10        [-1, 32, 11, 11]               0
Conv2d-11               [-1, 256, 1, 1]         991,488
SlimConv2d-12           [-1, 256, 1, 1]               0
ReLU-13                 [-1, 256, 1, 1]               0
PNNConvBlock-14         [-1, 256, 1, 1]               0
Flatten-15                    [-1, 256]               0
Linear-16                       [-1, 9]           2,313
SlimFC-17                       [-1, 9]               0
Identity-18                     [-1, 9]               0
PNNLinearBlock-19               [-1, 9]               0
Linear-20                       [-1, 1]             257
SlimFC-21                       [-1, 1]               0
Identity-22                     [-1, 1]               0
PNNLinearBlock-23               [-1, 1]               0
================================================================
Total params: 1,006,394
Trainable params: 1,006,394
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.65
Params size (MB): 3.84
Estimated Total Size (MB): 4.60
----------------------------------------------------------------
```

A.2. Network Structure of 2-column Memento-PNN

```
----------------------------------------------------------------
Layer (type)               Output Shape         Param #
================================================================
ZeroPad2d-1             [-1, 4, 88, 88]               0
Conv2d-2               [-1, 16, 21, 21]           4,112
SlimConv2d-3           [-1, 16, 21, 21]               0
ReLU-4                 [-1, 16, 21, 21]               0
PNNConvBlock-5         [-1, 16, 21, 21]               0
ZeroPad2d-6             [-1, 4, 88, 88]               0
Conv2d-7               [-1, 16, 21, 21]           4,112
SlimConv2d-8           [-1, 16, 21, 21]               0
ReLU-9                 [-1, 16, 21, 21]               0
PNNConvBlock-10        [-1, 16, 21, 21]               0
ZeroPad2d-11           [-1, 16, 24, 24]               0
Conv2d-12              [-1, 32, 11, 11]           8,224
SlimConv2d-13          [-1, 32, 11, 11]               0
ReLU-14                [-1, 32, 11, 11]               0
PNNConvBlock-15        [-1, 32, 11, 11]               0
ZeroPad2d-16           [-1, 16, 24, 24]               0
Conv2d-17              [-1, 32, 11, 11]           8,224
SlimConv2d-18          [-1, 32, 11, 11]               0
Conv2d-19              [-1, 16, 21, 21]             272
SlimConv2d-20          [-1, 16, 21, 21]               0
ReLU-21                [-1, 16, 21, 21]               0
ZeroPad2d-22           [-1, 16, 24, 24]               0
Conv2d-23              [-1, 32, 11, 11]           8,224
SlimConv2d-24          [-1, 32, 11, 11]               0
ReLU-25                [-1, 32, 11, 11]               0
PNNConvBlock-26        [-1, 32, 11, 11]               0
Conv2d-27               [-1, 256, 1, 1]         991,488
SlimConv2d-28           [-1, 256, 1, 1]               0
ReLU-29                 [-1, 256, 1, 1]               0
PNNConvBlock-30         [-1, 256, 1, 1]               0
Flatten-31                    [-1, 256]               0
Conv2d-32               [-1, 256, 1, 1]         991,488
SlimConv2d-33           [-1, 256, 1, 1]               0
Conv2d-34              [-1, 32, 11, 11]           1,056
SlimConv2d-35          [-1, 32, 11, 11]               0
ReLU-36                [-1, 32, 11, 11]               0
Conv2d-37               [-1, 256, 1, 1]         991,488
SlimConv2d-38           [-1, 256, 1, 1]               0
ReLU-39                 [-1, 256, 1, 1]               0
PNNConvBlock-40         [-1, 256, 1, 1]               0
Flatten-41                    [-1, 256]               0
Linear-42                       [-1, 9]           2,313
SlimFC-43                       [-1, 9]               0
Identity-44                     [-1, 9]               0
PNNLinearBlock-45               [-1, 9]               0
Linear-46                       [-1, 1]             257
SlimFC-47                       [-1, 1]               0
Identity-48                     [-1, 1]               0
PNNLinearBlock-49               [-1, 1]               0
Linear-50                       [-1, 9]           2,313
SlimFC-51                       [-1, 9]               0
Linear-52                     [-1, 256]          65,792
SlimFC-53                     [-1, 256]               0
Identity-54                   [-1, 256]               0
Linear-55                       [-1, 9]           2,313
SlimFC-56                       [-1, 9]               0
Identity-57                     [-1, 9]               0
PNNLinearBlock-58               [-1, 9]               0
```

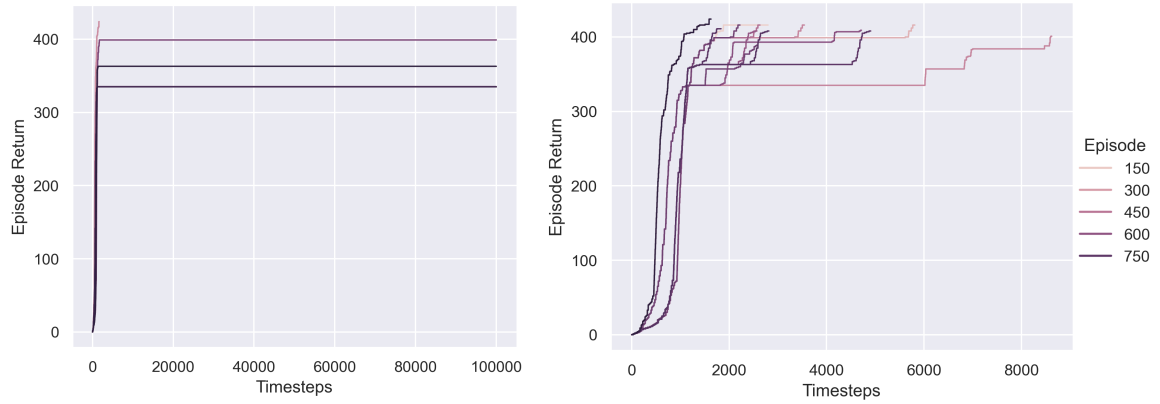
```
Linear-59                      [-1, 1]             257
SlimFC-60                      [-1, 1]               0
Linear-61                    [-1, 256]          65,792
SlimFC-62                    [-1, 256]               0
Identity-63                  [-1, 256]               0
Linear-64                      [-1, 1]             257
SlimFC-65                      [-1, 1]               0
Identity-66                    [-1, 1]               0
PNNLinearBlock-67              [-1, 1]               0
================================================================
Total params: 3,147,982
Trainable params: 2,141,588
Non-trainable params: 1,006,394
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 1.70
Params size (MB): 12.01
Estimated Total Size (MB): 13.81
----------------------------------------------------------------
```

A.3. Network Structure of 3-column Memento-PNN

```
----------------------------------------------------------------
Layer (type)               Output Shape         Param #
================================================================
ZeroPad2d-1             [-1, 4, 88, 88]               0
Conv2d-2               [-1, 16, 21, 21]           4,112
SlimConv2d-3           [-1, 16, 21, 21]               0
ReLU-4                 [-1, 16, 21, 21]               0
PNNConvBlock-5         [-1, 16, 21, 21]               0
ZeroPad2d-6             [-1, 4, 88, 88]               0
Conv2d-7               [-1, 16, 21, 21]           4,112
SlimConv2d-8           [-1, 16, 21, 21]               0
ReLU-9                 [-1, 16, 21, 21]               0
PNNConvBlock-10        [-1, 16, 21, 21]               0
ZeroPad2d-11            [-1, 4, 88, 88]               0
Conv2d-12              [-1, 16, 21, 21]           4,112
SlimConv2d-13          [-1, 16, 21, 21]               0
ReLU-14                [-1, 16, 21, 21]               0
PNNConvBlock-15        [-1, 16, 21, 21]               0
ZeroPad2d-16           [-1, 16, 24, 24]               0
Conv2d-17              [-1, 32, 11, 11]           8,224
SlimConv2d-18          [-1, 32, 11, 11]               0
ReLU-19                [-1, 32, 11, 11]               0
PNNConvBlock-20        [-1, 32, 11, 11]               0
ZeroPad2d-21           [-1, 16, 24, 24]               0
Conv2d-22              [-1, 32, 11, 11]           8,224
SlimConv2d-23          [-1, 32, 11, 11]               0
Conv2d-24              [-1, 16, 21, 21]             272
SlimConv2d-25          [-1, 16, 21, 21]               0
ReLU-26                [-1, 16, 21, 21]               0
ZeroPad2d-27           [-1, 16, 24, 24]               0
Conv2d-28              [-1, 32, 11, 11]           8,224
SlimConv2d-29          [-1, 32, 11, 11]               0
ReLU-30                [-1, 32, 11, 11]               0
PNNConvBlock-31        [-1, 32, 11, 11]               0
ZeroPad2d-32           [-1, 16, 24, 24]               0
Conv2d-33              [-1, 32, 11, 11]           8,224
SlimConv2d-34          [-1, 32, 11, 11]               0
Conv2d-35              [-1, 16, 21, 21]             272
SlimConv2d-36          [-1, 16, 21, 21]               0
Conv2d-37              [-1, 16, 21, 21]             272
SlimConv2d-38          [-1, 16, 21, 21]               0
ReLU-39                [-1, 16, 21, 21]               0
ZeroPad2d-40           [-1, 16, 24, 24]               0
Conv2d-41              [-1, 32, 11, 11]           8,224
SlimConv2d-42          [-1, 32, 11, 11]               0
ReLU-43                [-1, 32, 11, 11]               0
PNNConvBlock-44        [-1, 32, 11, 11]               0
Conv2d-45             [-1, 256, 1, 1]         991,488
SlimConv2d-46         [-1, 256, 1, 1]               0
ReLU-47               [-1, 256, 1, 1]               0
PNNConvBlock-48       [-1, 256, 1, 1]               0
Flatten-49                  [-1, 256]               0
Conv2d-50             [-1, 256, 1, 1]         991,488
SlimConv2d-51         [-1, 256, 1, 1]               0
Conv2d-52              [-1, 32, 11, 11]           1,056
SlimConv2d-53          [-1, 32, 11, 11]               0
ReLU-54                [-1, 32, 11, 11]               0
Conv2d-55             [-1, 256, 1, 1]         991,488
SlimConv2d-56         [-1, 256, 1, 1]               0
ReLU-57               [-1, 256, 1, 1]               0
PNNConvBlock-58       [-1, 256, 1, 1]               0
```

```
Flatten-59                  [-1, 256]               0
Conv2d-60               [-1, 256, 1, 1]         991,488
SlimConv2d-61           [-1, 256, 1, 1]               0
Conv2d-62              [-1, 32, 11, 11]           1,056
SlimConv2d-63          [-1, 32, 11, 11]               0
Conv2d-64              [-1, 32, 11, 11]           1,056
SlimConv2d-65          [-1, 32, 11, 11]               0
ReLU-66                [-1, 32, 11, 11]               0
Conv2d-67               [-1, 256, 1, 1]         991,488
SlimConv2d-68           [-1, 256, 1, 1]               0
ReLU-69                 [-1, 256, 1, 1]               0
PNNConvBlock-70         [-1, 256, 1, 1]               0
Flatten-71                  [-1, 256]               0
Linear-72                     [-1, 9]           2,313
SlimFC-73                     [-1, 9]               0
Identity-74                   [-1, 9]               0
PNNLinearBlock-75             [-1, 9]               0
Linear-76                     [-1, 1]             257
SlimFC-77                     [-1, 1]               0
Identity-78                   [-1, 1]               0
PNNLinearBlock-79             [-1, 1]               0
Linear-80                     [-1, 9]           2,313
SlimFC-81                     [-1, 9]               0
Linear-82                   [-1, 256]          65,792
SlimFC-83                   [-1, 256]               0
Identity-84                 [-1, 256]               0
Linear-85                     [-1, 9]           2,313
SlimFC-86                     [-1, 9]               0
Identity-87                   [-1, 9]               0
PNNLinearBlock-88             [-1, 9]               0
Linear-89                     [-1, 1]             257
SlimFC-90                     [-1, 1]               0
Linear-91                   [-1, 256]          65,792
SlimFC-92                   [-1, 256]               0
Identity-93                 [-1, 256]               0
Linear-94                     [-1, 1]             257
SlimFC-95                     [-1, 1]               0
Identity-96                   [-1, 1]               0
PNNLinearBlock-97             [-1, 1]               0
Linear-98                     [-1, 9]           2,313
SlimFC-99                     [-1, 9]               0
Linear-100                  [-1, 256]          65,792
SlimFC-101                  [-1, 256]               0
Linear-102                  [-1, 256]          65,792
SlimFC-103                  [-1, 256]               0
Identity-104                [-1, 256]               0
Linear-105                    [-1, 9]           2,313
SlimFC-106                    [-1, 9]               0
Identity-107                  [-1, 9]               0
PNNLinearBlock-108            [-1, 9]               0
Linear-109                    [-1, 1]             257
SlimFC-110                    [-1, 1]               0
Linear-111                  [-1, 256]          65,792
SlimFC-112                  [-1, 256]               0
Linear-113                  [-1, 256]          65,792
SlimFC-114                  [-1, 256]               0
Identity-115                [-1, 256]               0
Linear-116                    [-1, 1]             257
SlimFC-117                    [-1, 1]               0
Identity-118                  [-1, 1]               0
PNNLinearBlock-119            [-1, 1]               0
================================================================
Total params: 5,422,482
Trainable params: 2,274,500
Non-trainable params: 3,147,982
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 2.92
Params size (MB): 20.69
Estimated Total Size (MB): 23.71
----------------------------------------------------------------
```

B. Greedy Policy Evaluation



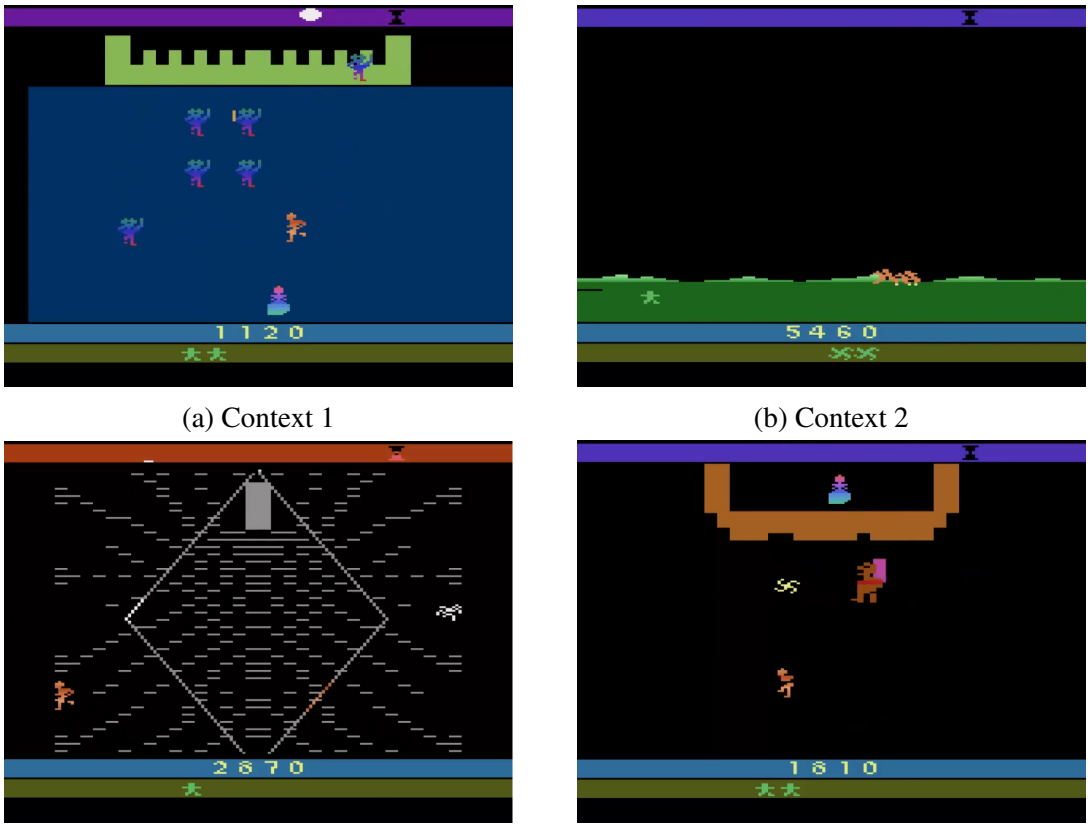
(a) (b)

Figure B.1.: Agent performance of Atari `Breakout` showing some of the trajectories of the 1000 evaluation episodes. One episode is a full environment circle, i.e. using all lives of the agent. With greedy action sampling (a), the agent is stuck at certain points in the game without progressing further, sometimes because required reset actions after life loss were never learned. Thus leading to inefficient evaluation until the max. number of steps in the emulator are reached. Using ϵ-greedy action sampling (b) with a small $\epsilon < 0.002$, the agent can reliably escape plateaus and continue to collect more reward.

C. Additional agent results for ALE

C.1. Elaboration on `Krull`

Learning contexts of `Krull`



(a) Context 1

(b) Context 2

(c) Context 3

(d) Context 4

Figure C.1.: The different objectives (contexts) in Atari `Krull`

C.1 shows the different learning contexts of the game `Krull`:

- **Context 1**: Defending the princess. Opponents spawn on the top of the screen and move downwards. The player must prevent them to reach the bottom. Enemies keep spawning until the player is overwhelmed and the princess is abducted.
- **Context 2**: Follow the kidnapper. Riding the horse to get to a new level. This context appears every time a level is finished. The player can't do anything but collecting objects (lives or glaives) on the way by pressing a button at the right moment.

- **Context 3**: Spider den. The solid gray square must be reached while a spider (right center) moves towards the player and must be avoided. After the square is reached, the player must move to the edge of the screen in order to progress to the next level. He is taken back to screen (context) 2. If the player doesn't leave the den at the correct time of day (according to a timer at the top of the screen), instead of getting to the boss level, he loses a life and returns to the den.
- **Context 4**: Beast fight. With the glaive (yellow disc), the player must destroy the box around the princess (center top), similar to `Breakout`. The disc bounces back after hitting and must be caught, while the boss moves horizontally, blocking the disc and shooting downwards. If the prison is broken, the princess escapes and the player can shoot one fireball which can defeat the boss. The game then restarts at a higher difficulty.

Exploiting the game

In contrast to the other games used in this work, all versions of the trained agents (Base, Memento & Memento-PNN) did *not* learn to play `Krull` properly. Instead, they learned an exploit in the spider den stage, where constantly hovering over the moving diamond-shaped lines (see Context 3 in C.1) gives the player small amounts of reward. The glitch can be used infinitely, when the player resets the spider spawn by moving into the gray square. Our agents didn't learn this detail of the exploit and thus reach a finite score and eventually terminate the game at some point, but also never solve the level.
An example of the full abuse of the glitch can be found e.g. on YouTube "Atari VCS/2600 Krull scoring trick and glitch"[1].

[1] https://www.youtube.com/watch?v=MpBSqgh1FNA (accessed at 11/2020)

C.2. Longer training of base agents

In order to show, that even further training does not help the base agent to overcome the performance plateau and the policy degradation happening there, we continued to train the baseline of `Phoenix`. As reported in Table 4.4, the performance peak for this agent occurred after 41.3M steps and the comparison to the Memento methods is depicted in Figure 4.3. Here we show the same base agent trained for approximately 224M timesteps in Figure C.2. The graph supports the statement we made in Sec. 3.2.1 and 4.2.1, that agents trained on the whole environment as a single task can not achieve comparable performance as the proposed algorithm, even with excessively more training time.

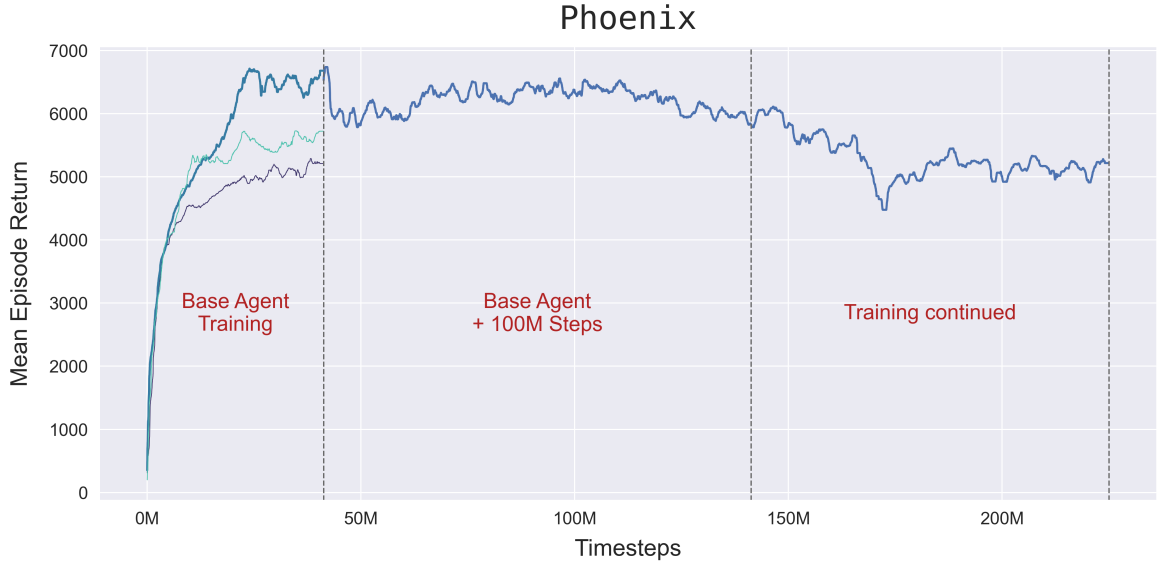


Figure C.2.: Training curve for the best baseline agent trained on the whole environment of `Phoenix` for almost twice the time than the advanced agents. The vertical lines show the boundaries, where the original base agent was stopped and checkpointed (at the performance peak), the continued baseline trained for the defined 100M steps, and finally additional +80M timesteps of training showing that there is no improvement.

D. Observation space visualization for additional games

D.1. UMAP and t-SNE for `Breakout`

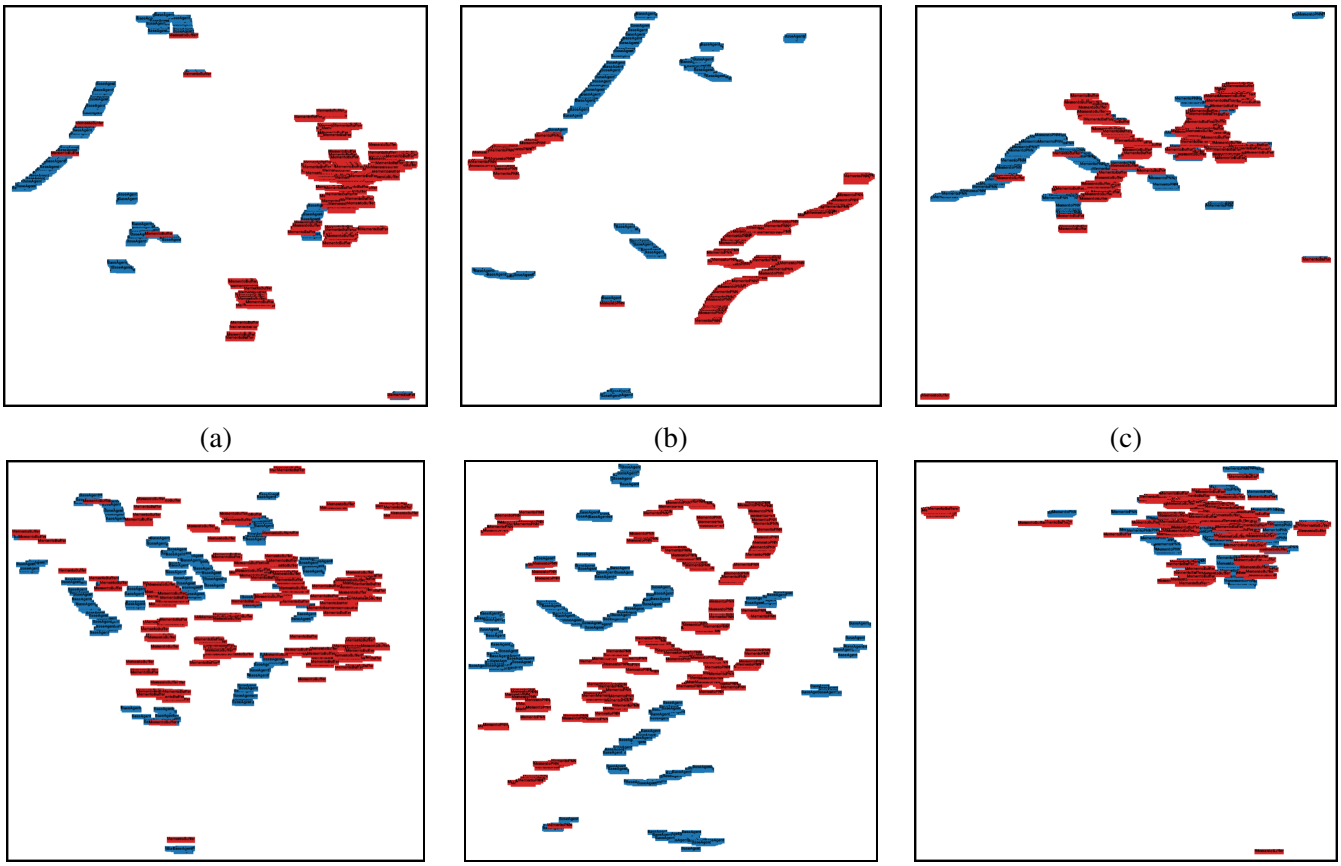


Figure D.1.: UMAP (top) and t-SNE (bottom) of `Breakout` for observation trajectories of:
(a) Base agent (blue) vs. Memento buffer (red)
(b) Base agent (blue) vs. Memento-PNN agent (red)
(c) Memento-PNN agent (blue) vs. Memento buffer (red)
For each class, 450 images were selected randomly from a pool of 10 000 observations. Best viewed in color.
HP for UMAP Epochs: 500, Neighbors: 15
HP for t-SNE Iterations: 1000, Perplexity: 8, Learning rate: 10

D.2. UMAP and t-SNE for `Phoenix`

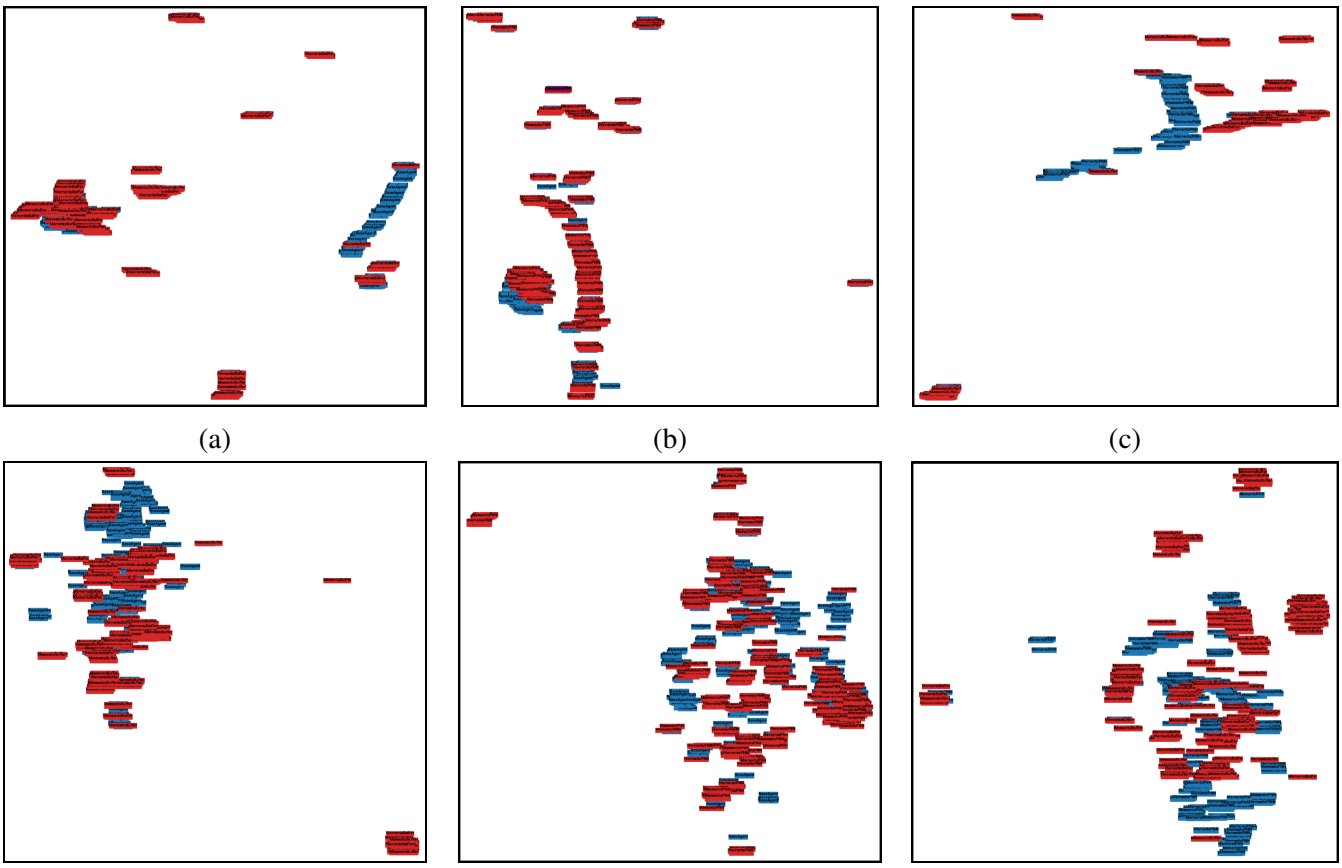


Figure D.2.: UMAP (top) and t-SNE (bottom) of `Phoenix` for observation trajectories of:
(a) Base agent (blue) vs. Memento buffer (red)
(b) Base agent (blue) vs. Memento-PNN agent (red)
(c) Memento-PNN agent (blue) vs. Memento buffer (red)
For each class, 450 images were selected randomly from a pool of 10 000 observations.
HP for UMAP Epochs: 500, Neighbors: 15
HP for t-SNE Iterations: 1000, Perplexity: 8, Learning rate: 10

D.3. UMAP and t-SNE for WizardOfWor

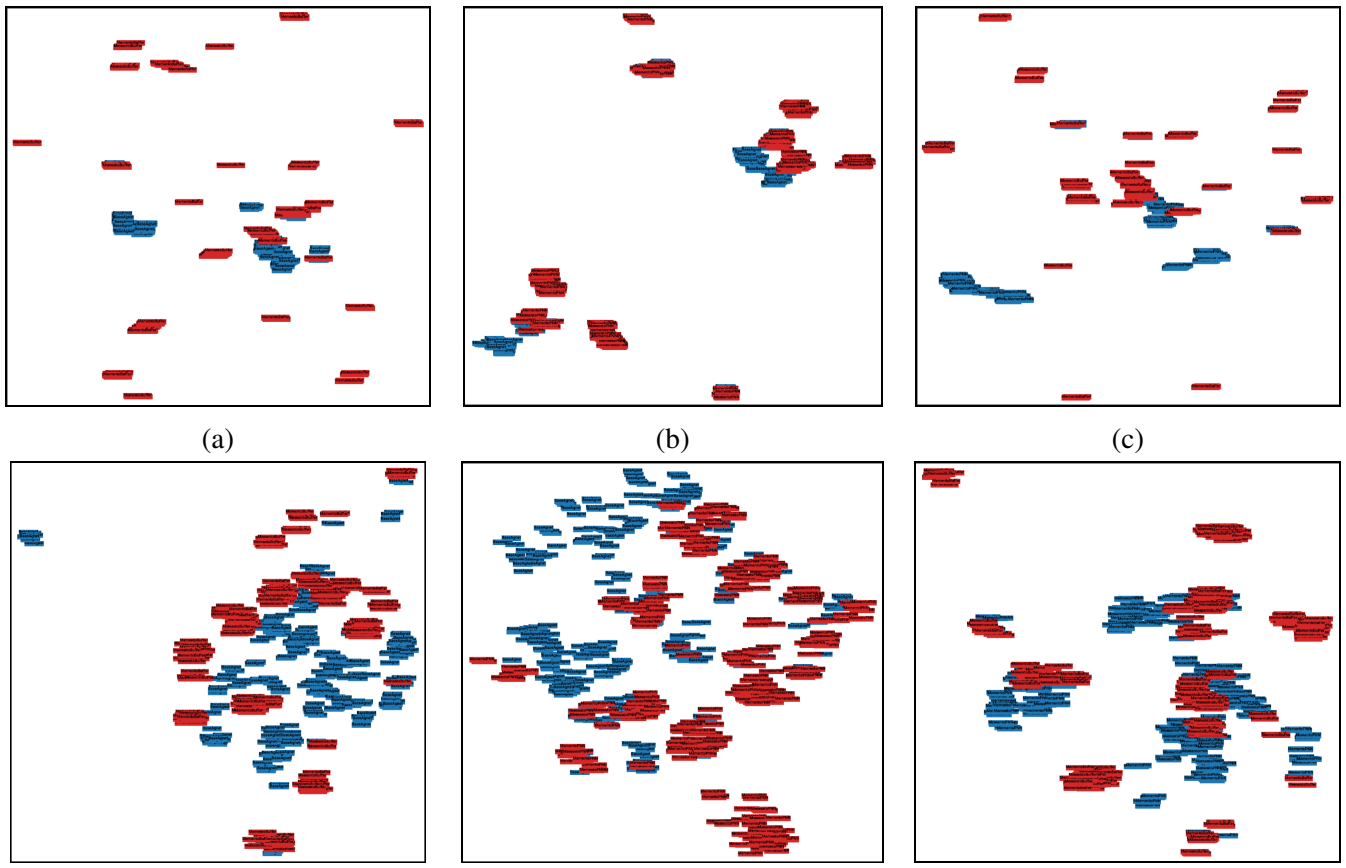


Figure D.3.: UMAP (top) and t-SNE (bottom) of WizardOfWor for trajectories of:
(a) Base agent (blue) vs. Memento buffer (red)
(b) Base agent (blue) vs. Memento-PNN agent (red)
(c) Memento-PNN agent (blue) vs. Memento buffer (red)
For each class, 450 images were selected randomly from a pool of 10 000 observations.
HP for UMAP Epochs: 500, Neighbors: 15
HP for t-SNE Iterations: 500, Perplexity: 10, Learning rate: 10

D.4. UMAP and t-SNE for `Krull`

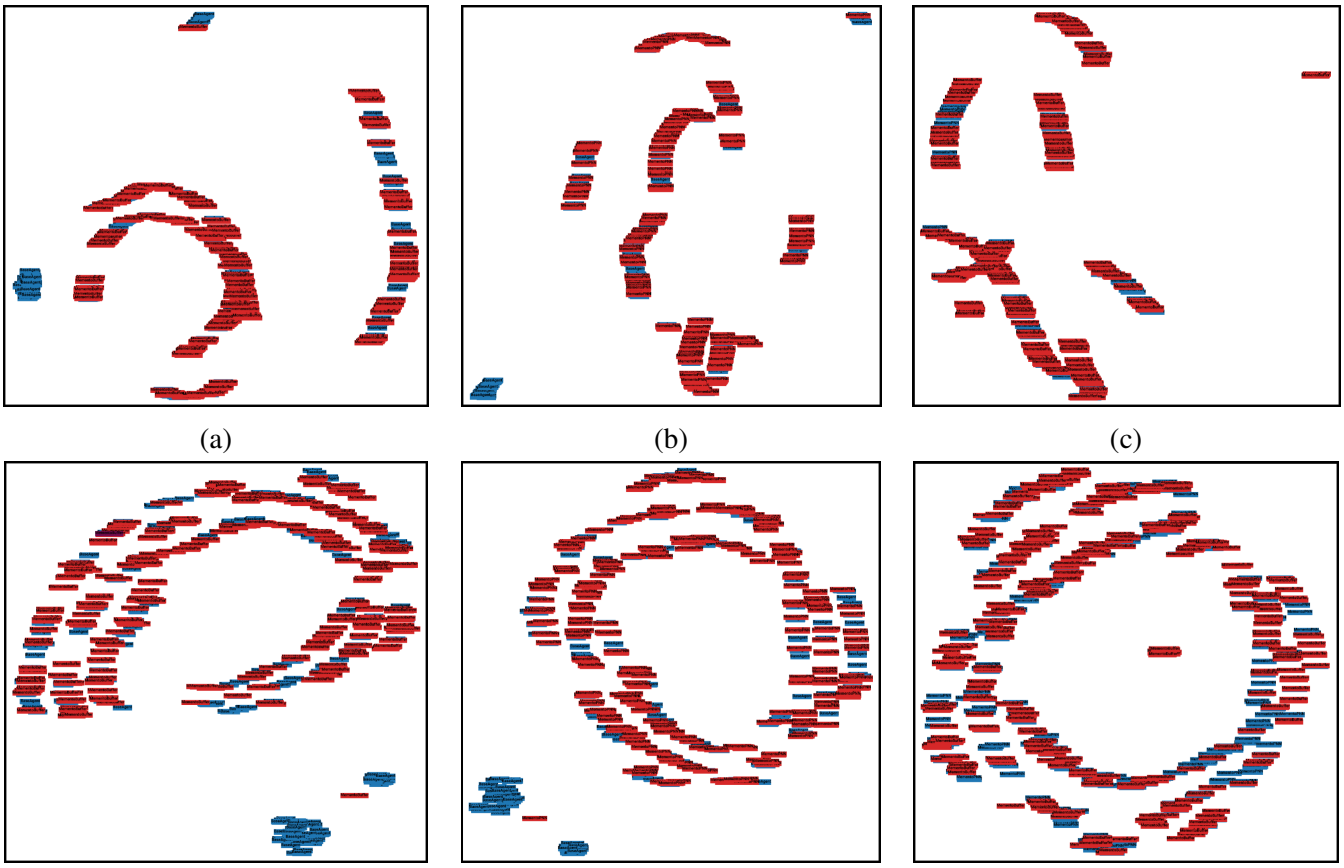


Figure D.4.: UMAP (top) and t-SNE (bottom) of `Krull` for observation trajectories of:
(a) Base agent (blue) vs. Memento buffer (red)
(b) Base agent (blue) vs. Memento-PNN agent (red)
(c) Memento-PNN agent (blue) vs. Memento buffer (red)
For each class, 450 images were selected randomly from a pool of 10 000 observations.
HP for UMAP Epochs: 500, Neighbors: 15
HP for t-SNE Iterations: 1000, Perplexity: 50, Learning rate: 10