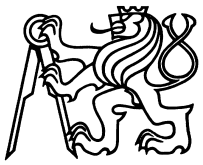
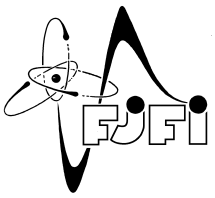


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering
Department of Mathematics



Transfer learning in Reinforcement learning tasks - learning the task correspondence

Study for Dissertation

Author: **Ing. Marko Ruman**

Supervisor: **Ing. Tatiana Valentine Guy, Ph.D.**

Supervisor-specialist: **Ing. Miroslav Kárný, DrSc.**

Academic year: 2020/2021

Contents

Abstract: Deep reinforcement learning has shown an ability to achieve super-human performance in solving complex reinforcement learning tasks only from raw-pixels. However, it fails to reuse knowledge from previously learnt tasks to solve new, unseen ones. To generalize and reuse knowledge is one of the fundamental requirements for creating a truly intelligent agent. This study summarizes the problem of transfer learning in reinforcement learning tasks and offers a method for one-to-one task transfer learning.

Keywords: deep reinforcement learning, transfer learning, Markov decision process

List of Symbols

α	Learning rate
$\mathbf{\Phi}$	Set of activation functions
δ^k	Intermediate value used in backpropagation
γ	Discount factor
$\mathcal{L}$	Neural network loss
$\mathcal{L}_D$	Discriminator loss
$\mathcal{L}_G$	Generator loss
$\mathcal{L}_{Cyc}$	Cycle-consistency loss
$\mathcal{L}_{GAN}$	GAN loss
$\mathcal{L}_M$	Model loss
$\mathcal{L}_Q$	Q loss
λ_{Cyc}	Cycle-consistency loss weight
λ_M	Model loss weight
λ_Q	Q loss weight
$\mathbb{N}$	Natural numbers
$\mathbb{R}$	Real numbers
C	Correspondence function
ϕ^k	Activation function of the k-th layer
Π	Decision policy
π	Decision rule
π_t	Decision rule used at the t-the time step
$\mathbf{A}$	Set of actions
$\mathbf{E}$	Set of edges
$\mathbf{G}$	Network graph
$\mathbf{M}$	Experience memory
$\mathbf{S}$	Set of states
$\mathbf{V}$	Set of neurons
$\mathbf{X}$	Dataset
θ	Network parameters
a_t	Action at the t-th time step
C_{in}	Number of input channels of convolutional layer
C_{out}	Number of output channels of convolutional layer
D	GAN discriminator
d^t	Observed states, actions and rewards up to the t-th time step
$E[\cdot]$	Expected value of $\cdot$
e^k	Activation of the k-th layer
$E_*[\cdot]$	Expected value of $\cdot$ with respect to probability distribution $*$
F	Environment model
f	Mapping formed by a neural network
f_K	Knowledge function
G	GAN generator
h	Convolution filter
K	Knowledge
N	Number of time steps

N_B	The size of sample batch	R	Reward function
N_M	The size of experience memory	r_t	Reward received at the t-th time step
N_U	The number of steps after which the networks in Q-learning are synchronized	s_t	State at the t-th time step
o^k	Output of the k-th layer	T	Transition function
Q	Q-function	t	Time step

Introduction

Motivation and objectives of the research

The field of deep reinforcement learning (RL) has received much attention recently. Model-free methods, for instance Deep Q-network (DQN), have shown an ability to achieve super-human performance in Atari games, [26], [**?**], where the algorithms find optimal discrete actions. Moreover, the model-free methods are successfully used also e.g. in MuJoCo environments with continuous actions, [22], or real-world robotic applications, [24]. Advances were also made in model-based deep RL methods such as AlphaZero, [31], or PlaNET, [13].

However, unlike humans, all the methods lack an essential ability to *generalize* across different but similar tasks. Humans have a remarkable ability to learn motor skills by mimicking the behaviours of others. For example, developmental psychologists have shown that 18-month-old children can infer the intentions and imitate the behaviours of adults, [25]. Imitation is not easy: children likely need to infer correspondences between their observations and their internal representations, which effectively aligns the two domains.

In machine learning, artificial intelligence (AI) or robotics, using knowledge learnt in different but related tasks is called *transfer learning* (TL), [27]. In the RL context, transfer learning is important for developing agents with lifelong learning, [1], for simulation-to-real knowledge transfer used in robotics, [19, 14, 30, **?**], or for developing the general AI, [6].

The shortcomings of transfer learning in RL and especially in deep RL contexts often stem from the inability to generalize. For example, when deep learning was used with image data, the output of a deep convolutional network can be dramatically changed by 1-pixel perturbations of the input image, [32]. Naturally, as image data often form the observable states in RL tasks, this problem also occurs in RL, for instance 1-pixel perturbations lead to useless policies, [29]. The RL-methods fail to reuse previously learnt knowledge even in very similar tasks, i.e. when the original image is rotated or some colours have been changed. It was also shown that learning from scratch can be more efficient than fine-tuning the previously gained model, [8]. All that contrast significantly with human abilities to generalize and reuse previously acquired knowledge.

This work formalises the problem of transfer learning in RL tasks and offers a novel method for one-to-one TL based on finding correspondence between *unpaired* data.

Formalisation of TL problem

State of the art

Other approaches: This work focuses on transfer learning by finding correspondence between data. Besides learning the correspondence, however, some other approaches for transfer learning in deep RL

are being researched. One research direction aims to learn the general features of RL tasks transferable across multiple tasks. Work [5] showed that learning policies on so-called mid-level features can generalize better than learning policies directly on image observations. Work [11] learns the general features of two tasks with different dynamics. However, their method is based on paired image observations which are hard or impossible to obtain in practice. Work [3] achieved success in tasks differing in reward function by maintaining successor features and decoupling environment dynamic and reward function. Work [**?**] introduces task similarity criterion and builds a transfer learning framework based on knowledge shaping, where for similar tasks, positive transfer is theoretically guaranteed. Work [**?**] achieved good results in tasks differing in environment dynamics parameters such as gravitation constant by inferring environment model on the target task.
Correspondence search: The pioneering work that used task correspondence was based on unsupervised image-to-image translation models CycleGAN[1], [38], and UNIT[2], [23]. Work [8] achieved results on a specific set of tasks by finding correspondence between states of two RL tasks. The application potential of the approach is rather limited as problems like mode-collapse were present. Works [30] and [14] improved the approach proposed in [8] by introducing learnt Q-function or object detection into the learning of the task correspondence. One of the recent approaches, [37], includes an environment model in the learning of the task correspondence. This approach is strongly inspired by the video-to-video translation model, [2].

Main thesis contributions

- It introduces a method for *knowledge transfer between two similar RL tasks* based on RL-specific modification of CycleGAN.
- The work establishes a *correspondence function computing similarity between the data* from two different RL tasks. The method can successfully transfer knowledge between RL tasks while respecting their similarity.
- The work proposes a new loss function consisting of three meaningful components reflecting different types of losses. The proposed modification *introduces Q-function and environment model into the learning*. It is shown that each component plays a significant role.
- The method is general and *does not require any extra task-specific* manual engineering. It works with RL tasks regardless of data format of states (e.g. images, sounds, numerical vectors, etc.).

Experiments with Atari game Pong, [4], demonstrated that the proposed method notably speeds up the learning and increases the average reward. The proposed approach is able to cope with the tasks for which standard approaches (GAN, CycleGAN) fail and learning from scratch remains to be the most efficient method.

Layout of the thesis

The paper layout is as follows: Section **??** recalls the required background and formulates the TL problem. Section **??** constructs the correspondence function and proposes a novel method of its learning. Section 4 describes the experimental evaluation of the proposed approach and compares it with baseline methods. Section 5 provides concluding remarks and outlines future research directions.

[1] Cycle generative adversarial network
[2] Unsupervised Image-to-Image Translation Networks

Chapter 1

Preliminaries and Research Methodology

1.1 Artificial Neural Networks

1.1.1 General Neural Networks

Definition 1 (Fully-connected neural network)**.** A fully connected neural network is a tuple

$$(\mathbf{G}, \mathbf{\Phi}, \Theta_0, \mathcal{L}), \tag{1.1}$$

where

- $\mathbf{G}$ is a directed graph defining the architecture of the network, $\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$, where
  - $\mathbf{V} = \left\{\left\{n_i^k\right\}_{i=1}^{r_k}, k \in \{1, 2, ..., m\}\right\}$ is a set of neurons, where n_i^k stands for i-th neuron in layer k, the set $\left\{n_i^k\right\}_{i=1}^{r_k}$ is referred to as *k-th layer*,
  - m is the number of layers,
  - r_k is the number of neurons in the k-th layer,
  - $\mathbf{E}$ is the set of edges, which in the case of fully-connected network is defined as follows - $\mathbf{E} = \left\{\left(n_i^k, n_j^{k+1}\right), i \in \{1, 2, ..., r_k\}, j \in \{1, 2, ..., r_{k+1}\}, k \in \{1, 2, ..., m\}\right\}$ and
    - every neuron has an edge with each neuron from the "neighbor layer"
    - to each edge of the network $\left(n_i^k, n_j^{k+1}\right)$, there is a corresponding weight $\theta_{ij}^k \in \mathbb{R}$
    - the weights can be organized into matrices Θ^k, where θ_{ij}^k is its (i,j)-th entry, $\Theta = \left\{\Theta^k\right\}_{k=1}^m$
- $\mathbf{\Phi}$ is a set of activation functions, $\mathbf{\Phi} = \left\{\phi^k\right\}_{k=1}^m$, where $\phi^k : \mathbb{R} \mapsto \mathbb{R}$ is activation function at layer k
- Θ_0 is an initial value of weights, $\Theta_0 = \left\{\Theta_0^k\right\}_{k=1}^m$
- $\mathcal{L} = \mathcal{L}(\mathbf{X}, \Theta)$ is the loss function, that has the following form

$$\mathcal{L}(\mathbf{X}, \Theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_p\left(x_i, y_i, \Theta\right) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i, \tag{1.2}$$

  where:

  - $\mathcal{L}_p$ is chosen usually as squared L2 norm or L1 norm,

– $\mathbf{X} = \{(x_i, y_i)\}_{i=1}^N$ is *dataset*, i.e. set of input-output pairs of the size $N \in \mathbb{N}$,
– $\mathcal{L}_i$ is the loss for the i-th input-output pair, i.e. $\mathcal{L}_i = \mathcal{L}_p(x_i, y_i, \Theta)$.

The neural network forms a mapping f in the following form:

$$f(x|\Theta) = \phi^m\left(\Theta^m \phi^{m-1}\left(\Theta^{m-1} ... \phi^1\left(\Theta^1 x\right)...\right)\right). \tag{1.3}$$

□

1.1.2 Backpropagation

The neural network is learnt through the weight updates. They are updated by using *gradient descent*, [12], with the loss function $\mathcal{L}$ as follows[1]:

$$\Theta_{t+1} = \Theta_t - \alpha \frac{\partial \mathcal{L}(\mathbf{X}, \Theta_t)}{\partial \Theta}, \qquad \text{i.e.} \qquad \theta^k_{t+1;ij} = \theta^k_{t;ij} - \alpha \frac{\partial \mathcal{L}(\mathbf{X}, \Theta_t)}{\partial \theta^k_{ij}} = \theta^k_{t;ij} - \alpha \frac{1}{N} \sum_{l=1}^{N} \frac{\partial \mathcal{L}_l}{\partial \theta^k_{ij}}, \tag{1.4}$$

where α is a parameter called *learning rate*.

Backpropagation is an algorithmic way how to compute the needed derivatives $\frac{\partial \mathcal{L}_i}{\partial \theta^k_{ij}}$ efficiently.

The algorithm is summarized in Algorithm 5, the derivation of it is the following.

- The following extra notation is used:
  – $o^k = \phi^k(e^k) = \begin{bmatrix} o^k_1 \\ o^k_2 \\ \vdots \\ o^k_{r_k} \end{bmatrix}$ is the *output* of k-th layer, o^k_i is the output of i-th node in k-th layer
  – $e^k = \Theta^{k-1} o^{k-1} = \begin{bmatrix} e^k_1 \\ e^k_2 \\ \vdots \\ e^k_{r_k} \end{bmatrix}$ is the *activation* of k-th layer, where $e^k_i = \sum_{j=1}^{r_{k-1}} \theta^{k-1}_{ij} o^{k-1}_j$
- The derivation begins by applying the chain rule for partial derivative:

$$\frac{\partial \mathcal{L}_i}{\partial \theta^k_{ij}} = \frac{\partial \mathcal{L}_i}{\partial e^k_i} \frac{\partial e^k_i}{\partial \theta^k_{ij}}. \tag{1.5}$$

  – Let us denote the first term as

$$\delta^k_i = \frac{\partial \mathcal{L}_i}{\partial e^k_i}, \tag{1.6}$$

  – the second term is computed as follows:

$$\frac{\partial e^k_i}{\partial \theta^k_{ij}} = \frac{\partial}{\partial \theta^k_{ij}} \sum_{l=1}^{r_{k-1}} \theta^{k-1}_{il} o^{k-1}_l = o^{k-1}_j \tag{1.7}$$

[1] The time indexes of weights (e.g. Θ_t or $\theta^k_{t;ij}$) denote its values at the t-th step of gradient descent

- A recursive formula for computing δ_i^k can be derived using chain rule:

$$\delta_i^k = \frac{\partial \mathcal{L}_i}{\partial e_i^k} = \sum_{j=1}^{r_k} \frac{\partial \mathcal{L}_i}{\partial e_j^{k+1}} \frac{\partial e_j^{k+1}}{\partial e_i^k}, \text{where} \tag{1.8}$$

  - $\frac{\partial \mathcal{L}_i}{\partial e_j^{k+1}} = \delta_j^{k+1}$
  - $\frac{\partial e_j^{k+1}}{\partial e_i^k} = \sum_{l=1}^{r_k} \theta_{jl}^k \frac{\partial}{\partial e_i^k} \phi^k(e_l^k) = \theta_{ji}^k \frac{d}{de_i^k} \phi^k(e_i^k)$

- Hence, substituting (A.6) and (A.7) into (A.5) the following formula is obtained:

$$\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k} = \delta_i^k o_j^{k-1} = \frac{d}{de_i^k} \phi^k(e_i^k) o_j^{k-1} \sum_{l=1}^{r_k} \delta_l^{k+1} \theta_{li}^k. \tag{1.9}$$

- It remains to compute all o_j^k and e_j^k, which is done in the forward phase when calculating the output by (A.3), and all $\delta_j^m = \frac{\partial \mathcal{L}_i}{\partial e_j^m}$ which depends on the specific form of loss function $\mathcal{L}$.

Forward phase (or forward propagation) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Mini-batch gradient descent, [21] In practice, each step of gradient descent is not computed on the full dataset as in (A.4), but on small subsets of the dataset $\mathbf{X}$ called mini-batches. A hyper-parameter $N_B < N$, called *mini-batch size*, is chosen and then for each step a mini-batch $\left\{(x_i, y_i)_{i=1}^{N_B}\right\} \subset \mathbf{X}$ is generated and the mini-batch gradient descent is performed as follows:

$$\theta_{t+1;ij}^k = \theta_{t;ij}^k - \alpha \frac{1}{N_B} \sum_{l=1}^{N_B} \frac{\partial \mathcal{L}_l}{\partial \theta_{ij}^k} \tag{1.10}$$

The main advantages of mini-batch gradient descent are its more robust convergence, i.e. avoiding local minima, and lower memory requirements - only small subset of dataset is stored in memory.

1.1.3 Convolutional Neural Networks

As the name suggests, the convolutional neural networks contain convolution that is a mapping defined as follows.

Definition 2 (Convolution)**.** For tensors x and h the convolution produces a tensor $y = x * h$ that is defined as follows:

$$y_{i_1,i_2,\ldots,i_n} = \sum_{j_1} \sum_{j_2} \cdots \sum_{j_n} x_{i_1-j_1,i_2-j_2,\ldots,i_n-j_n} h_{j_1,j_2,\ldots,j_n} \tag{1.11}$$

□

Definition 3. The convolutional neural network is a fully-connected neural network (see Definition 6) that contains one or more (even exclusively) convolutional layers. □

A convolutional layer can be defined for arbitrary dimension, but since this work considers images, the 2-D convolutional layers will be used.

Algoritmus 1: Backpropagation on mini-batches

1. Calculate the forward phase for each input-output pair (x_i, y_i) from a mini-batch and store the results $f(x_i)$ (A.3), e_j^k and o_j^k for each node j in layer k by proceeding from layer $k = 0$, the input layer, to layer $k = m$, the output layer.
2. Calculate the backward phase for each input-output pair (x_i, y_i) from a mini-batch and store the results $\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k}$ for every node and every layer.
   - Evaluate the loss term for the final layer $\delta_j^m = \frac{\partial \mathcal{L}_i}{\partial e_j^m}$ for each node.
   - Backpropagate the loss terms for the other layers δ_j^k, working backwards from the layer $k = m - 1$, by repeatedly using the equation (A.8)
   - Evaluate the partial derivatives of the individual loss $\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k}$ by using the equation (A.9)
3. Update the weights according to the equation (A.10)

Definition 4 (2-D Convolutional layer)**.** A 2-D convolutional layer with the parameters *input channels* $C_{in} \in \mathbb{N}$, *output channels* $C_{out} \in \mathbb{N}$ and *kernel size* $k \in \mathbb{N}$ contains:

- C_{out} parameter matrices $h_1, ...h_{C_{out}}$ referred to as *filters*. Each $h_i \in \mathbb{R}^{k,k,C_{in}}$.
- activation function ϕ.

The layer takes $x \in \mathbb{R}^{C_{in},n,n}$ as an input that has dimensions $C_{in}, n, n \in \mathbb{N}$ and gives the output o in the following way:

- $e = [h_1 * x, h_2 * x, ..., h_{C_{out}} * x] \in \mathbb{R}^{C_{out},n_{out},n_{out}}$ is the activation of the layer, where $n_{out} = n - k + 1$,
- $o = \phi(e)$ is the output of the layer.

□

Remark 1. The learning of convolutional neural network uses the backpropagation algorithm from the Section A.1.1. The learned parameters of convolutional layers are contained in the filters h_i from Definition 9. The neural network is differentiable with respect to these parameters and the activation e and output o from Definition 9 can be directly used for the backpropagation algorithm. □

1.1.4 Generative Adversarial Network

Generative Adversarial Network (GAN) algorithm assumes a training dataset $\mathbf{X} \subset \mathbb{R}^n$, $n \in \mathbb{N}$ that is assumed to be generated from an unknown distribution p. The goal of GAN is to learn the unknown distribution p. GAN assumes a known distribution q defined on $\mathbb{R}^d$, $d \in \mathbb{N}$ (q can be chosen as e.g. uniform or normal distribution). GAN finds a mapping $G(z) : \mathbb{R}^d \mapsto \mathbb{R}^n$ such that if a random variable $z \in \mathbb{R}^d$ has the distribution q, then $G(z)$ has the distribution p.

There are mainly two problems associated with creating GAN: 1) how to find G, 2) if we have G, how do we know that $G(z)$, where z is drawn from distribution q, has distribution p.

GAN forms an adversarial system from which G receives updates to improve its performance. More specifically, it trains also a discriminator $D(x) : \mathbb{R}^n \mapsto \mathbb{R}$ that represents a probability of x coming from

the training set $\mathbf{X}$ or from the samples generated from $G(z)$. Both G and D are constructed as neural networks $G(x|\theta_G)$, $D(x|\theta_D)$ and their parameters θ_G, θ_D are learnt.

This adversarial game can be mathematically defined as the following minimax problem of a target function[2]:

$$\min_G \max_D E_p[\log D(x)] + E_q[\log(1 - D(G(z)))], \tag{1.12}$$

Practically, the mini-batch stochastic gradient method is used to train the neural networks. The specific algorithm is summarized in Algorithm 6

Algoritmus 2: GAN learning

Input: training set $\mathbf{X}$, noise distribution q, minibatch size m

for *number of training iterations* **do**

Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from the distribution q

Sample minibatch of m examples $\{x_1, ..., x_m\}$ from the training set $\mathbf{X}$

Calculate the discriminator loss:

$$\mathcal{L}_D = -\frac{1}{2m}\left(\sum_{i=1}^{m} \log D(x_i|\theta_D) + \log(1 - D(G(z_i|\theta_G)))\right)$$

Update θ_D by gradient descent

Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from the distribution q

Calculate the generator loss:

$$\mathcal{L}_G = -\frac{1}{m}\left(\sum_{i=1}^{m} \log D(G(z_i|\theta_G))\right)$$

Update θ_G by gradient descent

end

Output: The learnt generator G

1.2 Markov Decision Processes

Definition 5 (Markov decision process)**.** An MDP is a tuple

$$(\mathbf{S}, \mathbf{A}, T, R, \gamma), \tag{1.13}$$

where

- $\mathbf{S}$ is a set of possible states,
- $\mathbf{A}$ is a set of possible actions,
- $T(s_{t+1}, s_t, a)$ is a *transition function* from state $s_t \in \mathbf{S}$ to a new state $s_{t+1} \in \mathbf{S}$ while taking action $a \in \mathbf{A}$,
- $R(s_{t+1}, a_t, s_t)$ is a *reward function* that outputs a real number based on states $s_t, s_{t+1} \in \mathbf{S}$ and action $a \in \mathbf{A}$,

[2] In practice, instead of minimizing $\log\left(1 - D\left(G\left(z|\theta_g\right)\right)\right)$, maximizing $\log\left(D\left(G\left(z|\theta_g\right)\right)\right)$ is often used for training the generator G.

- $\gamma \in [0, 1]$ is a *discount factor*.

The set $\mathbf{S} \times \mathbf{A}$ will be referred to as *task domain*.

□

In each time step t, the agent chooses action $a_t \in \mathbf{A}$ while the environment is in state s_t, then the environment transits to a new state s_{t+1} and the agent receives a reward $r_t = R(s_{t+1}, a_t, s_t)$ depending on the action a_t and states s_t and s_{t+1}. The *goal* of the agent is to maximize the accumulated discounted reward by taking the appropriate actions. Mathematically speaking, his aim is to find the optimal decision policy

$$\Pi^* = \{\pi_t^*\}_{t=1}^N, \tag{1.14}$$

where $\pi_t^* \in \boldsymbol{\pi}$ is a deterministic decision rule used in epoch t, $\boldsymbol{\pi} = \{\pi : \mathbf{S} \mapsto \mathbf{A}\}$ is the set of all decision rules (i.e. mappings from $\mathbf{S}$ to $\mathbf{A}$) and $N \in \mathbb{N} \cup \{\infty\}$ is a number of time steps of the task. Since the policy will be evaluated in each decision epoch, from now on, it is vital to assume that the solution of a task is the *optimal decision rule* π^* that satisfies:

$$\pi^* = \underset{\pi \in \boldsymbol{\pi}}{\operatorname{argmax}}\, E\left[\sum_{t=1}^{N} \gamma^t R(s_{t+1}, \pi(s_t), s_t)\right]. \tag{1.15}$$

The decision rule π^* is called the *solution* of the MDP.

1.3 Reinforcement learning

Reinforcement learning (RL) task involves an agent interacting with an *environment* through his *actions* in order to influence the *state* of the environment in a targeted way (i.e. the agent has a goal wrt the environment). The agent receives a numerical *reward* after each action and his goal is to maximize the accumulated rewards obtained throughout the task. The *Markov decision process* (MDP) formalism is used for describing such task, [28].

A typical RL task assumes that the state and action sets, $\mathbf{S}$ and $\mathbf{A}$, are *known* by the agent, while the transition T and reward R functions are *unknown*. In general, the state and/or the action sets are assumed to be *very large*.

Since reward function R and transition function T are unknown, the agent estimates π^* (1.15) using previously observed data d^t consisting of past states, actions and obtained values of rewards (denoted as r_t), i.e.

$$d^t = (s_\tau, a_\tau, r_\tau, s_{\tau+1})_{\tau=1}^t. \tag{1.16}$$

The observed data d^t can easily become too large, thus, in practice, only a transformation of d^t necessary for solving (1.15) is saved in memory. For instance parameter estimation uses the whole data record (thus comprising the info contained in data), and only a small part of d^t is stored. *Knowledge* K^t will stand for the transformed data up to time t[3]. For example, the knowledge K^t may contain value of parameters that define the optimal decision rule π^* from (1.15).

There are various methods for solving a RL task, i.e. finding the solution of (1.15). Vast majority of them is based on Bellman equation and use dynamic programming. The ability to solve individual RL task is a necessary prerequisite of transfer learning. Throughout the study, a model-free method, deep Q-learning, will be used for solving RL tasks. It is derived from regular Q-learning, [36], and described below.

[3]The index t in K^t will be used only when necessary

1.3.1 Q-learning

Q-learning, [36], is a model-free approach to solving RL tasks. Instead of using unknown reward function R directly, the estimate of (state-action) Q-function is used, i.e. knowledge K is the estimate of the Q-function:

$$K = \hat{Q}(s,a) \quad : \quad \mathbf{S} \times \mathbf{A} \to \mathbb{R}, \tag{1.17}$$

where

$$Q(s,a) = E_{\pi^*}\left[\sum_{t=1}^{N} \gamma^t R_t | s_1 = s, a_1 = a\right], \tag{1.18}$$

i.e. Q-function gives the expected value of future discounted reward while using the optimal decision rule π^* for given starting state s and action a.

The estimate of the Q-function (1.18) for s and a, denoted as $\hat{Q}(s,a)$, is continuously learnt on the stream of data records (s_{t+1}, a_t, s_t, r_t) using so-called temporal difference error, [36], as follows:

$$\hat{Q}_{t+1}(s_t,a_t) = (1-\alpha)\hat{Q}_t(s_t,a_t) + \alpha(r_t + \gamma \max_{a \in \mathbf{A}} \hat{Q}_t(s_{t+1},a)), \tag{1.19}$$

where α is a parameter called *learning rate*, γ is *discount factor*. The learning starts with the initial conditions $\hat{Q}_0(s,a)$.

Algoritmus 3: Q-learning

Input: initial Q-function $Q_0(s,a)$, learning rate α, discount factor γ
get the initial state s_0
for *t=1,2,..., till convergence* **do**
  select action $a_t = \underset{a \in \mathbf{A}}{\operatorname{argmax}} Q(s_t,a)$
  get the next state s_{t+1} and reward r_t
  if s_t *is terminal* **then**
    target = r
  else
    target = $r + \gamma \max_{a \in \mathbf{A}} Q(s_{t+1},a)$
  end
  $Q_{new}(s_t,a_t) = (1-\alpha)Q(s_t,a_t) + \alpha(\text{target})$
end
Output: Q-function $Q(s,a)$

The solution, i.e. the optimal decision rule outputting the optimal action, $\pi^*(s|K)$ is then found as

$$\pi^*(s|K) = \pi^*\left(s|\hat{Q}\right) = \underset{a \in \mathbf{A}}{\operatorname{argmax}}\, \hat{Q}(s,a). \tag{1.20}$$

1.3.2 Deep Q-learning

As mentioned earlier, the state space is assumed to be huge. Therefore, to efficiently learn the Q-function, a function approximator has to be used. The current state-of-the-art of function approximators are deep neural networks, [7], which are used to learn so-called deep Q-network (DQN) approximating the Q-function. DQN algorithm is based on the standard Q-learning algorithm, [36] (see Algorithm 3).

In DQN algorithm, the Q-function is approximated by a deep neural network with parameters θ, which can be trained in a similar way as in the supervised learning. However, in the supervised learning,

the input data are assumed to be i.i.d., moreover, it is assumed that for the same input, its target value (i.e. the desired output) stays the same, [9].

In RL tasks, those two assumptions are not met. The consecutive states of the system (e.g. video frames) are usually highly correlated and thus very far from being i.i.d. Also, the target values contain the currently learnt Q-function which evolves during the learning process. This makes the target values and the whole learning process unstable.

These problems are solved by introducing the following two additional steps within the learning algorithm (see [26] for details):

Experience replay The last N_M data records (so-called *experience memory*, denoted as $\mathbf{M}$) are stored in a memory buffer. In each learning step, mini-batch of data records of length N_B (see Appendix A.1) are randomly sampled from the memory buffer and used for updating the neural network that approximates the Q-function. This makes the learning data closer to being i.i.d.

Target network In order to stabilize the target values, two separate networks are used. One of the networks (so-called *original*) is updated in every learning step, while the other network (so-called the *target network*) is used to retrieve target values and stays static, i.e. its parameters does not change in every learning step. Each N_U steps, the target network is synchronized with the original network used for learning (i.e. the parameters of the original networks are copied to the target network).

The whole DQN algorithm is summarized in Algorithm 4.

Algoritmus 4: DQN

Input: initial parameters θ of the Q-function, learning rate α, discount factor γ, exploration rate ϵ, size of the experience memory N_M, size of the learning minibatch N_B, number of steps for target network synchronization N_U

Initialize the experience memory to the size N_M

Set the parameters θ^- of the target network as $\theta^- = \theta$

for *t=1,2,..., till convergence* **do**

With probability ϵ perform random action a_t

otherwise select action $a_t = \underset{a \in \mathbf{A}}{\operatorname{argmax}} Q(s, a|\theta)$

Get next state s_{t+1} and receive reward r_t

Store (s_t, a_t, r_t, s_{t+1}) in the experience memory **M** and (if the memory is full) remove the oldest data record

Sample random minibatch $(s_j, a_j, r_j, s_{j+1})_{j \in Random(N_B)}$

for *every j* **do**

if s_{j+1} *is terminal* **then**

$\text{target}_j = r_j$

else

$\text{target}_j = r_j + \gamma \max_{a' \in \mathbf{A}} Q(s_{j+1}, a'|\theta^-)$

end

end

Perform a gradient descent step on $\left(\left(\text{target}_j - Q(s_j, a_j|\theta)\right)^2\right)_{j \in Random(N_B)}$ with Huber loss, [15], with respect to parameters θ

Every N_U steps set $\theta^- = \theta$

end

Output: Q-function $Q(s, a)$, experience memory **M**

Chapter 2

Transfer Learning in Reinforcement Learning

2.1 Overview

DQN algorithm is efficient in learning the solution of individual RL tasks even with high-dimensional states (e.g. raw video frames). However, it lacks any ability to use previously learnt knowledge for solving related tasks in the future (i.e. transfer learning).

It is assumed that the agent solves a new RL task using a standard algorithm like DQN from Section **??** during which he continuously accumulates task-related knowledge K. Apart from that, it is further assumed that the agent has pieces of knowledge $K_1, ..., K_n$ from n different, previously solved RL tasks. Transfer learning (TL), [27], in this context means the ability of the agent to choose the useful part of previously acquired knowledge $K_1, ..., K_n$, and use it for more efficient solving of the current task.

Mathematically expressed, the agent has (or learned) a *knowledge function* f_K that maps knowledge accumulated on all tasks (past and current) $K, K_1, ..., K_n$ to a *transformed knowledge* K_T that will be used to more effectively solve the current RL task[1],

$$f_K = f_K(K_1, ..., K_n, K) = K_T. \tag{2.1}$$

The construction and learning the knowledge function (2.1) is a challenging problem even when transfer knowledge between similar tasks is supposed, [8]. Main obstacles that may arise are the following:

- **Different task domains**. In general, it is assumed that the agent can use knowledge from the previous tasks performed in different domains. A simple example is a game called Mountain car, [33]. The agent has learned how to solve its 2D-version and should be able to reuse the learned knowledge to solve the 3D-version much more effectively (see Figure 2.1).

- **Scalability**. The number of the previously solved tasks n can become large over time. The agent should be able to scale well, so adding more tasks to the "database" of solved tasks should improve the agent's abilities rather than bring computational complexity.

[1] The specific form of the function and its properties depend on the specific form of the used pieces of knowledge $K_1, ..., K_n, K$ and will be specified later on. The performance metrics measuring the efficiency of transfer learning are introduced in the next section.

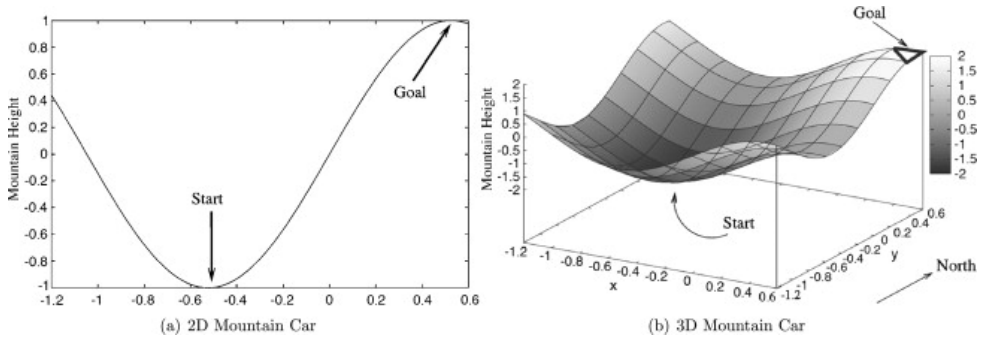


Figure 2.1: The game Mountain car in 2D and 3D version, [33]

2.2 Performance metrics of TL

The performance of the agent solving an individual RL task can be measured in a various way, see [16]. In this study, the moving average of reward will be considered as *performance*.

To evaluate effectiveness of TL, the following measures comparing an agent using TL with an agent solving the task from scratch, [34], may be used, see Figure 2.2:

- **Learning speed improvement**. The number of samples needed to achieve a desired performance is reduced.
- **Asymptotic improvement**. The maximum value of performance that agent reaches to is improved.
- **Jumpstart improvement**. The performance in the beginning of the target task is improved by using knowledge from the source tasks.

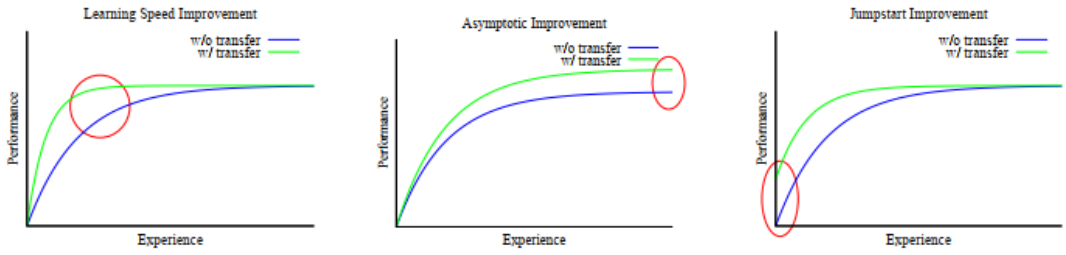


Figure 2.2: The performance criteria of transfer learning. The red circles highlight the improvement in the performance in the learning process expected by using TL compared to solving the task from scratch. There are 3 types of TL criteria - a) learning speed improvement, b) asymptotic improvement and c) jumpstart improvement

2.3 Transfer Learning using Q-function - REDO: this is 1 to 1

For the purposes of this study, the simplest example of transfer learning scenario is considered. It consists of two tasks - *source* and *target task*. The agent learns the solution of the source task from

scratch. His main objective is to use the knowledge learnt from the *source task*, K_1, for more efficient solving the *target task* (e.g. the agent finds the optimal policy using fewer steps).

The agent uses DQN algorithm (see Algorithm 4) for solving the source task. The created knowledge (see Chapter **??**) in the source task, K_1, is formed of Q-function and the experience memory $\mathbf{M}_1$ (see Algorithm 4)

$$K_1 = (Q_1(s,a), \mathbf{M}_1). \tag{2.2}$$

The agent continuously updates the knowledge of the target task K_2 in the form of Q-function, and experience memory $\mathbf{M}_2$, $K_2 = (Q_2(s,a), \mathbf{M}_2)$, while interacting with the target task.

It is assumed that the source and target tasks share dynamics and the knowledge from the source task is useful to some extend for learning the target task. However, generally the tasks may differ in:

1. States and actions
    - states and actions from the source task may correspond to different states and actions in the target tasks or the actions space in target task can contain more possible actions etc.
2. Environment differences
    - while the environments of two tasks may share some properties, e.g. laws of physics, it can differ in other aspects, e.g. the target task can contain other objects to interact with. An agent learning how to drive a motorcycle while previously learned to drive a car is an example of this situation.

One possible approach how to use knowledge K_1 gathered in the *source task* is to assume that "situations" the agent faces in the *target task* have already appeared in the *source task*. In other words, it is assumed that there exists a *correspondence function*

$$C : \mathbf{S}_1 \times \mathbf{A}_1 \to \mathbf{S}_2 \times \mathbf{A}_2 \tag{2.3}$$

that finds the correspondence between state-action pairs in the source and target tasks. $\mathbf{S}_1$ and $\mathbf{A}_1$ are the state and action spaces of the source task and $\mathbf{S}_2$ and $\mathbf{A}_2$ are the state and actions spaces of the target task, respectively. The correspondence function C outputs a state-action pair from the target task $(s_2, a_2) \in \mathbf{S}_2 \times \mathbf{A}_2$ for a given state-action pair from the source task $(s_1, a_1) \in \mathbf{S}_1 \times \mathbf{A}_1$.

It is used to produce the transferred knowledge K_T (therefore, it defines the knowledge function (2.1)) in the following way:

$$f_K(Q_1, \mathbf{M}_1, Q_2, \mathbf{M}_2) = K_T = Q_1\left(C(.,.|\mathbf{M}_1, Q_1, \mathbf{M}_2, Q_2)\right). \tag{2.4}$$

It means that the Q-function from the source task Q_1 is used for solving the target task. However its input - the state-action pair (s_2, a_2) is mapped by the correspondence function to a state-action pair $(s_1, a_1) = C(s_2, a_2)$ from the source task.

Thus, the remaining part of the proposed approach is to learn the correspondence function C from the available data contained in the knowledge from the source task K_1 and in the partial knowledge from the target task K_2. The learning uses mainly the experience memories $\mathbf{M}_1$ and $\mathbf{M}_2$, but also the Q-functions Q_1 and Q_2 can be used to improve the learning of the correspondence function C.

The specific construction and learning of the function $C = C(s, a|\mathbf{M}_1, Q_1, \mathbf{M}_2, Q_2)$ is presented in the next chapter.

2.4 Correspondence Function

2.4.1 Model of Loss

Let us assume (for brevity) that the action spaces of source and target RL task are identical, i.e. $\mathbf{A}_S = \mathbf{A}_T$. Let the mutually corresponding actions are found using an identity mapping independently of the current state[2]. Thus, we need to learn a mapping that indicates the corresponding states, correspondence function for states.

The proposed learning is inspired by CycleGAN, [38], and extends it by employing environment dynamics and optimal strategy to learn the correspondence function. The gained correspondence function is then used for knowledge transfer between the related RL tasks (see **??**). Let us explain the main idea.

CycleGAN is based on GAN[3], [10], and was developed for learning image-to-image transformations. Similarly, in considered case of TL we have two experience memories $\mathbf{M}_S$ and $\mathbf{M}_T$ with *mutually unpaired* entries. The task is to learn the correspondence function C that will help to match them.

In CycleGAN, two mappings G_S and G_T called *generators* are learnt,

$$G_S : \mathbf{S}_S \to \mathbf{S}_T \quad \text{and} \quad G_T : \mathbf{S}_T \to \mathbf{S}_S. \tag{2.5}$$

They are learnt as two GANs, thus generators G_S and G_T are learned simultaneously with respective discriminators D_S and D_T. The generators learn to map the states from $\mathbf{S}_S$ to $\mathbf{S}_T$ and vice-versa, while the discriminators learn to *distinguish* a real state from a state mapped by a generator.

Mappings G_S, G_T, D_S and D_T are constructed as neural networks with their architecture depending on the data format, for instance if states are images, convolutional layers are often used.

We propose the loss for the generators in the following form

$$\mathcal{L} = \mathcal{L}_{GAN} + \lambda_{Cyc}\mathcal{L}_{Cyc} + \lambda_Q\mathcal{L}_Q + \lambda_M\mathcal{L}_M, \tag{2.6}$$

where losses $\mathcal{L}_{GAN}, \mathcal{L}_{Cyc}, \mathcal{L}_Q$ and $\mathcal{L}_M$ are explained below. λ_{Cyc}, λ_Q and λ_M are *loss parameters* that define relative influence (weight) of the respective loss components.

Learning in CycleGAN minimises a loss consisting of two parts. The first part of the loss, $\mathcal{L}_{GAN}$, comes from GAN and it is given by the adversarial loss for generators G_S, G_T and their respective discriminators D_S, D_T as follows:

$$\begin{aligned}&\mathcal{L}_{GAN} = \\ &E_{s_S}\left[\log D_S(s_S)\right] + E_{s_T}\left[\log\left(1 - D_S\left(G_S\left(s_T\right)\right)\right)\right] \\ &+ E_{s_T}\left[\log D_T(s_T)\right] + E_{s_S}\left[\log\left(1 - D_T\left(G_T\left(s_S\right)\right)\right)\right]\end{aligned} \tag{2.7}$$

The adversarial training can learn mappings G_S and G_T (2.5) to produce outputs indistinguishable from respective sets $\mathbf{S}_S$ and $\mathbf{S}_T$. However, a network can learn to map the same set of input images to any permutation of images in the target domain while minimising the adversarial loss $\mathcal{L}_{GAN}$. Thus, even if its global minimum can be found loss (2.7) itself cannot guarantee that the desired correspondence function is learnt.

To further reduce the search space of possible mappings G_S and G_T, CycleGAN introduces a so-called cycle-consistency requirement: every state $s_S \in \mathbf{S}_S$ must be recoverable after mapping it back to $\mathbf{S}_T$, i.e. $G_T(G_S(s_S)) \approx s_S$. The same requirement applies to every state $s_T \in \mathbf{S}_T$. This cycle-consistency

[2]More specifically, all the actions in the source and target task have the same labels and meanings (e.g. $a = 1$ stands for "go up"), therefore no mapping between source and target task action spaces is necessary

[3]Generative adversarial network

requirement is expressed in the proposed loss (2.6) by minimising the *cycle-consistency* loss $\mathcal{L}_{Cyc}$ that has the following form:

$$\begin{aligned}\mathcal{L}_{Cyc} = & E_{s_S}\left[\|G_T\left(G_S\left(s_S\right)\right) - s_S\|_1\right] \\ & + E_{s_T}\left[\|G_S\left(G_T\left(s_T\right)\right) - s_T\|_1\right].\end{aligned} \tag{2.8}$$

Relation to RL

Even direct application of CycleGAN to the states brought some success in policy transfer, see for instance [8]. However, data records in experience memories comprise richer yet unused information that may be helpful for the transfer of knowledge between tasks. Respecting this information by including additional components into the loss function that is minimised in CycleGAN learning, will make the learned correspondence even more relevant to RL tasks.

This work proposes to add two other components of the loss:

- $\mathcal{L}_Q$ - a loss that describes how Q-function learned from the source task copes with impreciseness in learned generators G_T and G_S,
- $\mathcal{L}_M$ - a loss that reflects the influence of the environment model of the source task.

***Q* loss**

The available Q-function, Q_S, should be incorporated in the learning of the correspondence C as it determines the optimal policy. The *cycle-consistency* loss (2.8) forces generators G_S and G_T to be inverses of each other. By requiring that, the values of the Q-function of state s_S and the state mapped to the other domain and back, $G_T(G_S(s_S))$, are the same. So, it is assumed that the learning of the generators will focus on the states relevant to the Q-function. Thus we propose to introduce the loss $\mathcal{L}_Q$ in the following form:

$$\mathcal{L}_Q = E_{s_S}\left[\|Q_S\left(G_T\left(G_S\left(s_S\right)\right)\right) - Q_S\left(s_S\right)\|_1\right] \tag{2.9}$$

The loss (2.9) will make the learned correspondence more suitable for transferring knowledge between RL tasks because the learned correspondence function C will retain the parts of the states that are the most important for choosing the optimal action.

Model loss

So far, all considered losses (2.7) - (2.9) are associated with state values. However, any RL task is dynamic, and the time sequence of states is essential. Consider states of the target and the source tasks at times t and $t+1$. If generator G_T ensures mapping s_{Tt} on s_{St+1} and generator G_S maps s_{St+1} back to s_{Tt}, then losses $\mathcal{L}_{GAN}, \mathcal{L}_{Cyc}$ and $\mathcal{L}_Q$ (2.7) - (2.9) are minimal. However, it would not help to solve the target RL task.

Therefore, to capture the essential dynamic relationship of the source and target task in correspondence function C, it is also vital to consider the loss respecting environment model F of the source task:

$$\mathcal{L}_M = E_{s_{Tt}, a_{Tt}, s_{Tt+1}}\left[\|F\left(G_T\left(s_{Tt}\right), a_{Tt}\right) - G_T\left(s_{Tt+1}\right)\|_1\right] \tag{2.10}$$

The correspondence function minimising (2.10) will respect the environment dynamics in the following way: whenever action a_{Tt} is applied at state s_{Tt} resulting in state s_{Tt+1}, the triplet $(\bar{s}_{St}, a_{Tt}, \bar{s}_{St+1})$ should be observable in the source task, where $\bar{s}_{St} = G_T\left(s_{Tt}\right)$ and $\bar{s}_{St+1} = G_T\left(s_{Tt+1}\right)$.

Final loss

The proposed loss comprises all the components (2.7), (2.8), (2.9) and (2.10) and, thus, has the following form:

$$\mathcal{L} = \mathcal{L}_{GAN} + \lambda_{Cyc}\mathcal{L}_{Cyc} + \lambda_{Q}\mathcal{L}_{Q} + \lambda_{M}\mathcal{L}_{M}, \tag{2.11}$$

where λ_{Cyc}, λ_{Q} and λ_{M} are *loss parameters*.

The searched correspondence function C is then obtained as follows:

$$C(s_T, a_T) = (G_T(s_T), I(a_T)), \quad \forall (s_T, a_T) \in \mathbf{S}_T \times \mathbf{A}_T, \tag{2.12}$$

where G_T is the generator from (2.5) mapping states from the *target task* to states from the *source task* and $I(.)$ is an identity mapping.

2.4.2 Learning of Correspondence Function

Add the algorithm how the batches are generated,A_real, B_real etc.

2.4.3 Knowledge Transfer Among Several Tasks

TODO

2.4.4 Similarity Issues

??

2.5 Data Set

2.6 Algorithm

2.7 Comparison

2.8 Discussion and Related Work

Chapter 3

Experimental part

To test the transfer learning algorithm, the Atari game **Pong** was used as a playground, [4]. There are 6 available actions (do nothing, fire, move up, move down, move up fast, move down fast) and the last 4 image frames were used as a state. The agent learned to play the game using the DQN algorithm from Section **??** and, thus, obtained the Q-function. Moreover, to test the approach described in Section **??**, the agent learned environment model F.

The proposed method for TL was tested in two experiments. The *source* task in both of them was the original game **Pong**, which the agent learned how to play.

The *target* task in the **first experiment** was the same original Pong and the agent was searching for the identity transformation using the proposed approach. In the **second experiment**, the *target* task was **rotated Pong**, where the game remained the same, but the image frames were all rotated by 90 degrees.

The states in both of the tasks were the last 4 frames (images) of the game, the discrete actions spaces have the same amount of actions.

The experiment was as follows. The agent

1) learned to play **standard Pong** (the *source* task) and obtained the Q-function and environment model F,

2) observed 10 000 frames of the target task while using random actions,

3) was learning the state correspondence from the *target* task to the *source* task,

4) each 1000 steps of the correspondence learning, the agent played 5 games of the *target* task using the learned state correspondence and the Q-function from the *source* task (see 2.4) and saved the current correspondence if achieved the best reward so far,

5) played the **target task** using the learned correspondence and the Q-function from the *source* task (while continuously fine-tuning the Q-function, the correspondence was fixed).

The key metric to study was the average accumulated reward in one game while playing the *target* task.

3.1 Experiment 1 - original Pong

The generators G_1 and G_2 (see Section **??**) in this experiment were constructed as neural networks with convolutional layers (see Appendix A.2). Their specific architecture was taken from [20].

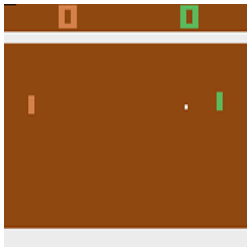


Figure 3.1: **Standard Pong**, [4]

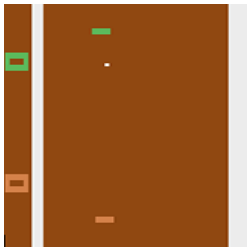


Figure 3.2: **Rotated Pong by 90 degrees**, [4]

The discriminators D_1 and D_1 were also constructed as neural networks with convolutional layers with the same architecture as in [17].

The parameters of all of the networks were initialized from a Gaussian distribution $N(0, 0.02)$.

The method was tested for various settings of the loss parameters λ_{Cyc}, λ_Q and λ_M (2.11). More specifically, it was run for all the combination of the parameters from the following sets - $\lambda_{Cyc} \in \{0, 1, 10\}$, $\lambda_Q \in \{0, 1\}$ and $\lambda_M \in \{0, 1, 10\}$.

3.1.1 Results discussion

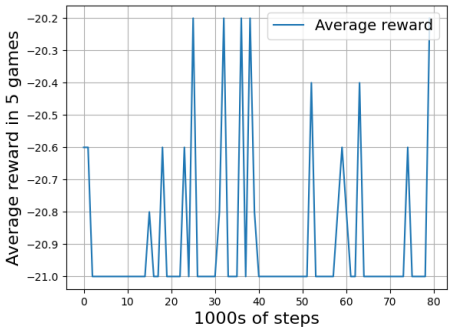
The results are presented in the following figures.

Figure 4.3 shows the average reward gained in 5 games that the agent played each 1000 steps of the correspondence learning for different settings of the parameters λ_{Cyc}, λ_Q and λ_M. The maximum reward in one game was 21 and the figure shows that the best results were achieved with settings involving all the parts of the loss $\lambda_{Cyc} = \lambda_Q = \lambda_M = 1$. A good performance was achieved also when only the cycle loss was applied, i.e. $\lambda_{Cyc} = 10$, $\lambda_Q = \lambda_M = 0$. However, the learning soon become unstable, which was not observed in the previously discussed setting.

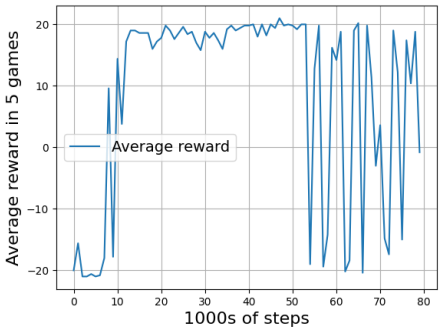
Figure 4.4 shows the progress of the correspondence learning. It confirms that the best results were achieved for the settings with all the loss parts active, $\lambda_{Cyc} = \lambda_Q = \lambda_M = 1$, while good results were achieved also for the setting $\lambda_{Cyc} = 10$, $\lambda_Q = \lambda_M = 0$. In other settings the correspondence learning failed to learn a useful state correspondence.

Figure 4.5 shows the evolution of the individual loss parts $\mathcal{L}_{GAN}$, $\mathcal{L}_{Cyc}$, $\mathcal{L}_Q$ and $\mathcal{L}_M$ in the correspondence learning for the best settings of the learning parameters $\lambda_{Cyc} = \lambda_Q = \lambda_M = 1$. It can be seen that all the losses were decreasing in the beginning of the experiment, but after some time the trend had changed. Especially $\mathcal{L}_{GAN}$ started increasing rapidly even though the average reward in Figure 4.3 was slightly improving.

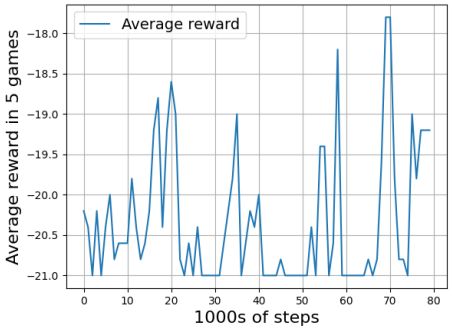
Lastly, Figure 4.6 shows the comparison of the agent learning to play the game from scratch and the agent that learned the correspondence and uses the previously learned Q-function. As the figure shows, in this simple situation, the agent was able to fully reuse the previously learned knowledge.



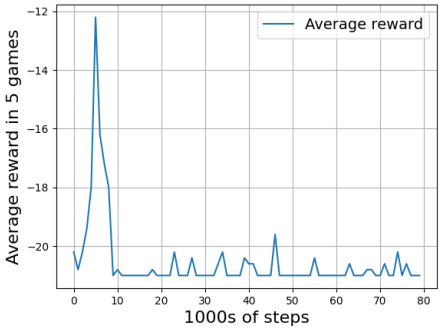
a) $\lambda_{Cyc} = \lambda_Q = \lambda_M = 0$



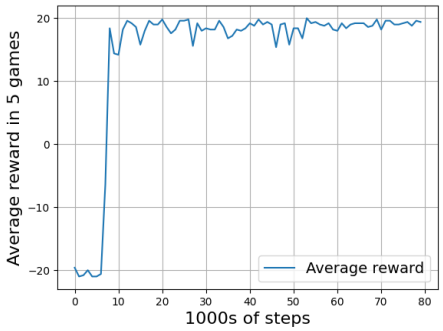
b) $\lambda_{Cyc} = 10, \lambda_Q = 0, \lambda_M = 0$



c) $\lambda_{Cyc} = 0, \lambda_Q = 1, \lambda_M = 0$

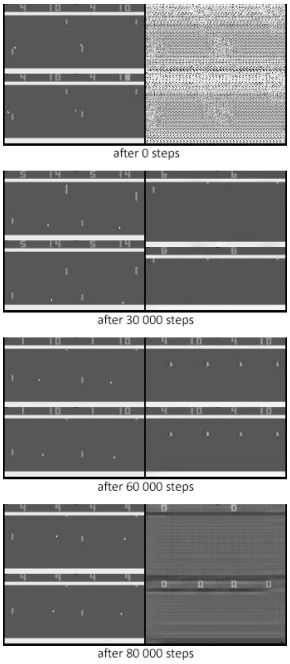


d) $\lambda_{Cyc} = 0, \lambda_Q = 0, \lambda_M = 10$

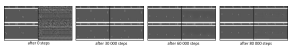


e) $\lambda_{Cyc} = 1, \lambda_Q = 1, \lambda_M = 1$

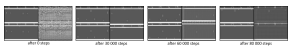
Figure 3.3: Average accumulated reward in 5 games when playing Pong by using the transformed Q-function (2.4). The reward is computed each 1000 steps of the correspondence learning. The results are shown for different values of loss weights λ_{Cyc}, λ_Q and λ_M.



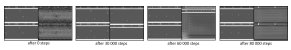
a) $\lambda_{Cyc} = 0$, $\lambda_Q = 0$, $\lambda_M = 0$



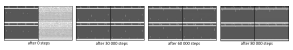
b) $\lambda_{Cyc} = 10, \lambda_Q = 0$, $\lambda_M = 0$



c) $\lambda_{Cyc} = 0$, $\lambda_Q = 1$, $\lambda_M = 0$

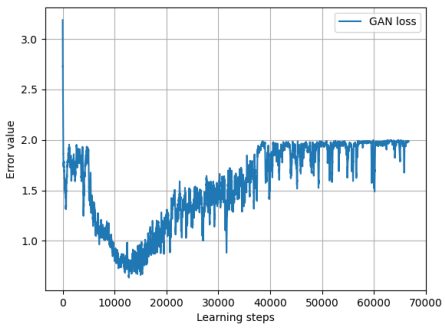


c) $\lambda_{Cyc} = 0, \lambda_Q = 0$, $\lambda_M = 10$

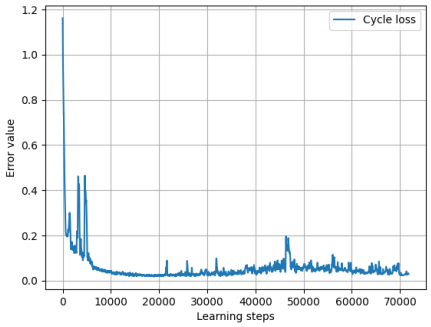


d) $\lambda_{Cyc} = 1, \lambda_Q = 1$, $\lambda_M = 1$

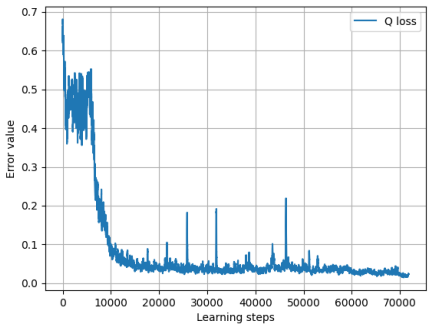
Figure 3.4: Progress of learning the correspondence function C (2.3) after 0, 20000, 40000 and 60000 iterations). The results are shown for different values of loss weights λ_{Cyc}, λ_Q and λ_M. The left part of the pictures is a state of the *target* task, the right part is the same state transformed by the correspondence C.



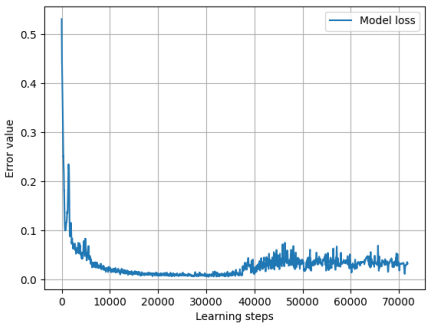
a) $\mathcal{L}_{GAN}$



b) $\mathcal{L}_{Cyc}$



c) $\mathcal{L}_{Q}$



d) $\mathcal{L}_{M}$

Figure 3.5: Evolution of the loss parts $\mathcal{L}_{GAN}$ (2.7), $\mathcal{L}_{Cyc}$ (2.8), $\mathcal{L}_{Q}$ (2.9) and $\mathcal{L}_{M}$ (2.10) while learning the correspondence function C (2.3) for the loss weights $\lambda_{Cyc} = 1, \lambda_{Q} = 1$, $\lambda_{M} = 1$.

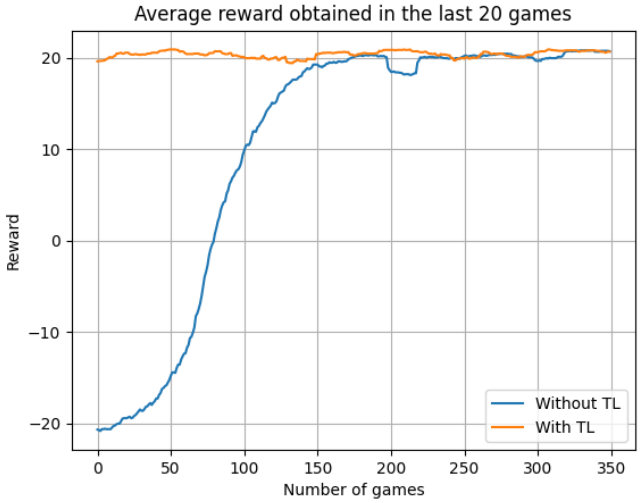


Figure 3.6: Average reward of the last 20 games depending on the number of played games for the game Pong without using TL (learning from sratch) vs. with using TL

3.2 Experiment 2 - rotated Pong

The generators G_1 and G_2 (see Section **??**) in this experiment were also constructed as neural networks. Two types of generators were tested in this experiment. The architecture of the first one, referred to here as **resnet generator**, was taken from [20] as in the previous experiment, but it was followed by a rotation layer as in [18]. The second type, referred to as **rotation generator**, was composed only by the mentioned rotation layer.

The discriminators D_1 and D_1 were also constructed, same as in the first experiment, as neural networks with convolutional layers with the same architecture as in [17].

The method was tested in this parameters for various settings of the loss parameters λ_{Cyc}, λ_Q and λ_M, only the best settings for both generator types are presented.

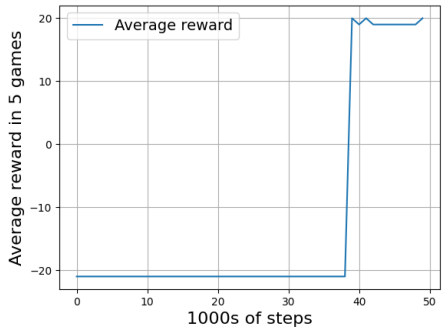
3.2.1 Results discussion

The results are presented in the following figures.

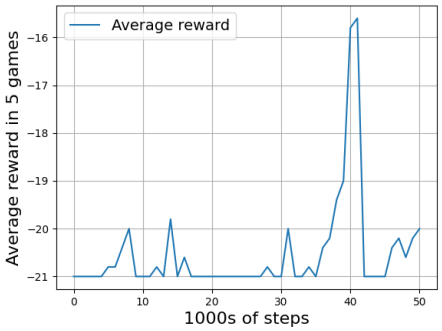
Figure 4.7 shows the average reward gained in 5 games that the agent played each 1000 steps of the correspondence learning for both types of the generators and for the best settings of the parameters λ_{Cyc}, λ_Q and λ_M. The maximum reward in one game was 21 and the figure shows that the perfect correspondence was found only in the case of **rotation generator**. The correspondence learning with **resnet generator** showed also some success, the correspondence from it was subsequently used for fine-tuning the Q-function to see if it can be helpful.

Figure 4.8 shows the progress of the correspondence learning. It shows the successful learning in the case of **rotation generator**, however the correspondence learned in the case of **resnet generator** looked visually close to the optimal one (especially after 60 000 steps).

Lastly, Figure 4.9 shows how the agent learned to play the *target* task - rotated Pong. It compares an agent learning to play the game from scratch, an agent with the correspondence learned i) with resnet generator, ii) with rotation generator and finally iii) an agent that reused the Q-function without any correspondence. As the figure shows, the agent with rotation generator was able to fully reuse the previously gained knowledge, the agent with resnet generator performed significantly better in the beginning than the agent learning from scratch, while the agent reusing only the Q-function achieved significantly worse performance than even the agent learning from scratch.



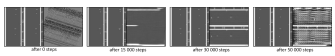
a) **Rotation generator** with $\lambda_{Cyc} = \lambda_Q = 0$, $\lambda_M = 10$



b) **Resnet generator** with $\lambda_{Cyc} = 1, \lambda_Q = 0$, $\lambda_M = 1$

Figure 3.7: Average accumulated reward in 5 games when playing Pong by using the transformed Q-function (2.4). The reward is computed each 1000 steps of the correspondence learning. The results are shown for **rotation** and **resnet** generator for the best settings of the loss parameters.

a) Rotation generator with $\lambda_{Cyc} = 0$, $\lambda_Q = 0$, $\lambda_M = 0$



b) Resnet generator with $\lambda_{Cyc} = 10, \lambda_Q = 0$, $\lambda_M = 0$

Figure 3.8: Progress of learning the correspondence function C (2.3). The results are shown for rotation and resnet generator for its best settings of the loss weights λ_{Cyc}, λ_Q and λ_M. The left part of the pictures is a state of the *target* task, the right part is the same state transformed by the correspondence C.

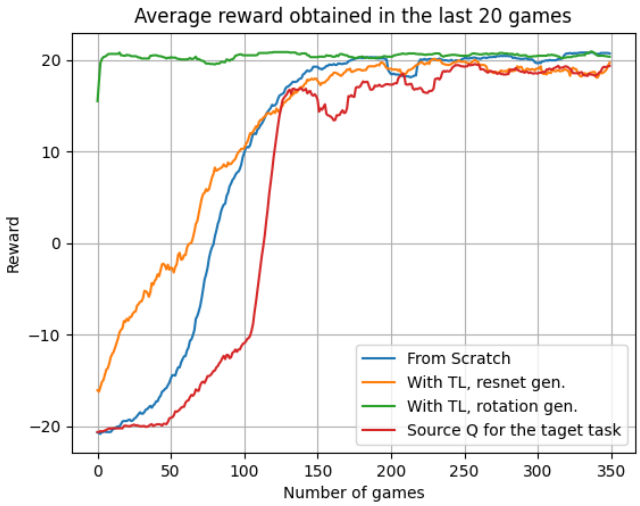


Figure 3.9: Average reward of the last 20 games depending on the number of played games for the game rotated Pong for four different agents - an agent learning the game from scratch, an agent using the correspondence learned with resnet generator, an agent using the correspondence learned with rotation generator and an agent reusing only the Q-function without any correspondence.

Chapter 4

Conclusions and Future Work

The paper considers transfer learning in RL tasks. As mentioned, state-of-the-art RL methods such as those presented in [26], [**?**] achieve superhuman performance on individual RL tasks, but they are unable to transfer knowledge across related tasks. The ability to transfer knowledge across tasks is a crucial element in developing general AI, [6], lifelong learning, [1], or simulation-to-real transfer, [**?**].

As the experiments have shown, fine-tuning the function approximators learned on previous tasks does not bring good results in RL context and can be even contra-productive. The work proposes a novel correspondence function (2.11) indicating the *source* and *target task* similarity. The correspondence function respects the Q-function and environment model of the source task and establishes them into learning. It allows the agent to gradually create a set of skills, adapt and use them while interacting with the *dynamically* changing environment.

The method was evaluated on simulated experiments involving 2-D Atari game Pong and compared against two baselines using GAN and CycleGAN method. Experiment 1, with identical source and target tasks, served as verification of the proposed method. It was shown that the method successfully learned the tasks correspondence. It also illustrated the importance of all the proposed loss components, as the gained performance was the most successful when all the components were respected. Comparing to baselines shows that TL using the loss relevant only to images was either unsuccessful (when only GAN loss was used) or unstable (when both GAN and Cycle-consistency loss are respected).

Experiment 2 shows how the proposed approach copes with different tasks. Two types of network architecture for generators in the learning of correspondence function were tested. It was shown that choice of the architecture is crucial for achieving good results. The generator composed of rotation and convolutional layers yielded worse performance (but still acceptable) than the generator with only a rotation layer. It shows that the convolutional layers traditionally used in image processing may not be suitable for the transfer learning scenario. Thus other architectures, like transformers, [35], should be investigated.

In both experiments the proposed method successfully learned the mapping between the states of the *source task* and the states of the *target task*. The resulting TL was successful and the agent played the target task very well from the very beginning.

The developed method is suitable for transferring between any related RL tasks, where only the network architecture has to be chosen and aligned with the task's state format. The most significant advantage of the proposed method is its practical applicability. It learns the correspondence function between RL tasks in an unsupervised way where the corresponding state pairs are unavailable. It is the typical situation that an RL agent faces and the proposed method has a clear potential to solve the transfer learning problem in RL tasks.

Many open problems remain unsolved, in particular: i) how to perform transfer learning between tasks having low similarity, ii) how to identify and transfer relevant knowledge from several source tasks, iii) how to choose the only relevant source tasks similarly to [**?**], iv) what is a better network architecture for the correspondence function learning.

Method implementation: The method implementation in Python is available at *https://github.com/marko-ruman/RL-Correspondence-Learner*

Data availability statement: The datasets generated and/or analysed during the current study are available from the corresponding author on reasonable request.

Ethical approval: This article does not contain any studies with human participants performed by any of the authors.

Bibliography

[1] Haitham Bou Ammar, Eric Eaton, José Marcio Luna, and Paul Ruvolo. Autonomous cross-domain knowledge transfer in lifelong policy gradient reinforcement learning. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[2] Aayush Bansal, Shugao Ma, Deva Ramanan, and Yaser Sheikh. Recycle-gan: Unsupervised video retargeting. In *Proceedings of the European conference on computer vision (ECCV)*, pages 119–135, 2018.

[3] Andre Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel Mankowitz, Augustin Zidek, and Remi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. In *International Conference on Machine Learning*, pages 501–510. PMLR, 2018.

[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[5] Bryan Chen, Alexander Sax, Gene Lewis, Iro Armeni, Silvio Savarese, Amir Zamir, Jitendra Malik, and Lerrel Pinto. Robust policies via mid-level visual representations: An experimental study in manipulation and navigation. *arXiv preprint arXiv:2011.06698*, 2020.

[6] Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

[7] Balázs Csanád Csáji et al. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24(48):7, 2001.

[8] Shani Gamrian and Yoav Goldberg. Transfer learning for related reinforcement learning tasks via image-to-image translation. In *International Conference on Machine Learning*, pages 2063–2072. PMLR, 2019.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[11] Abhishek Gupta, Coline Devin, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *arXiv preprint arXiv:1703.02949*, 2017.

[12] Jacques Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, volume 33. Imprimerie nationale, 1908.

[13] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[14] Daniel Ho, Kanishka Rao, Zhuo Xu, Eric Jang, Mohi Khansari, and Yunfei Bai. Retinagan: An object-aware approach to sim-to-real transfer. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10920–10926. IEEE, 2021.

[15] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.

[16] Javier Insa-Cabrera, David L Dowe, and José Hernández-Orallo. Evaluating a reinforcement learning algorithm with a general intelligence test. In *Conference of the Spanish Association for Artificial Intelligence*, pages 1–11. Springer, 2011.

[17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[18] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. *Advances in neural information processing systems*, 28, 2015.

[19] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12627–12637, 2019.

[20] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[21] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[22] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[23] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. *Advances in neural information processing systems*, 30, 2017.

[24] A Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. In *Conference on robot learning*, pages 561–591. PMLR, 2018.

[25] Andrew N Meltzoff. Understanding the intentions of others: re-enactment of intended acts by 18-month-old children. *Developmental psychology*, 31(5):838, 1995.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[27] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[28] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[29] Xinghua Qu, Zhu Sun, Yew-Soon Ong, Abhishek Gupta, and Pengfei Wei. Minimalistic attacks: How little it takes to fool deep reinforcement learning policies. *IEEE Transactions on Cognitive and Developmental Systems*, 13(4):806–817, 2020.

[30] Kanishka Rao, Chris Harris, Alex Irpan, Sergey Levine, Julian Ibarz, and Mohi Khansari. Rl-cyclegan: Reinforcement learning aware simulation-to-real. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11157–11166, 2020.

[31] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[32] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.

[33] Matthew E Taylor, Nicholas K Jong, and Peter Stone. Transferring instances for model-based reinforcement learning. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 488–505. Springer, 2008.

[34] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[36] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[37] Qiang Zhang, Tete Xiao, Alexei A Efros, Lerrel Pinto, and Xiaolong Wang. Learning cross-domain correspondence for control with dynamics cycle-consistency. *arXiv preprint arXiv:2012.09811*, 2020.

[38] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

Appendices

Appendix A

Neural network models

A.1 General neural network + backpropagation

Definition 6 (Fully-connected neural network)**.** A fully connected neural network is a tuple

$$(\mathbf{G}, \mathbf{\Phi}, \Theta_0, \mathcal{L}), \tag{A.1}$$

where

- $\mathbf{G}$ is a directed graph defining the architecture of the network, $\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$, where
  - $\mathbf{V} = \left\{\left\{n_i^k\right\}_{i=1}^{r_k}, k \in \{1,2,...,m\}\right\}$ is a set of neurons, where n_i^k stands for i-th neuron in layer k, the set $\left\{n_i^k\right\}_{i=1}^{r_k}$ is referred to as *k-th layer*,
  - m is the number of layers,
  - r_k is the number of neurons in the k-th layer,
  - $\mathbf{E}$ is the set of edges, which in the case of fully-connected network is defined as follows - $\mathbf{E} = \left\{\left(n_i^k, n_j^{k+1}\right), i \in \{1,2,...,r_k\}, j \in \{1,2,...,r_{k+1}\}, k \in \{1,2,...,m\}\right\}$ and
    - every neuron has an edge with each neuron from the "neighbor layer"
    - to each edge of the network $\left(n_i^k, n_j^{k+1}\right)$, there is a corresponding weight $\theta_{ij}^k \in \mathbb{R}$
    - the weights can be organized into matrices Θ^k, where θ_{ij}^k is its (i,j)-th entry, $\Theta = \left\{\Theta^k\right\}_{k=1}^m$
- $\mathbf{\Phi}$ is a set of activation functions, $\mathbf{\Phi} = \left\{\phi^k\right\}_{k=1}^m$, where $\phi^k : \mathbb{R} \mapsto \mathbb{R}$ is activation function at layer k
- Θ_0 is an initial value of weights, $\Theta_0 = \left\{\Theta_0^k\right\}_{k=1}^m$
- $\mathcal{L} = \mathcal{L}(\mathbf{X}, \Theta)$ is the loss function, that has the following form

$$\mathcal{L}(\mathbf{X}, \Theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_p(x_i, y_i, \Theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i, \tag{A.2}$$

  where:
  - $\mathcal{L}_p$ is chosen usually as squared L2 norm or L1 norm,
  - $\mathbf{X} = \{(x_i, y_i)\}_{i=1}^N$ is *dataset*, i.e. set of input-output pairs of the size $N \in \mathbb{N}$,

– $\mathcal{L}_i$ is the loss for the i-th input-output pair, i.e. $\mathcal{L}_i = \mathcal{L}_p(x_i, y_i, \Theta)$.

The neural network forms a mapping f in the following form:

$$f(x|\Theta) = \phi^m\left(\Theta^m\phi^{m-1}\left(\Theta^{m-1}...\phi^1\left(\Theta^1 x\right)...\right)\right). \tag{A.3}$$

□

A.1.1 Backpropagation

The neural network is learnt through the weight updates. They are updated by using *gradient descent*, [12], with the loss function $\mathcal{L}$ as follows[1]:

$$\Theta_{t+1} = \Theta_t - \alpha\frac{\partial\mathcal{L}(\mathbf{X}, \Theta_t)}{\partial\Theta}, \qquad \text{i.e.} \qquad \theta^k_{t+1;ij} = \theta^k_{t;ij} - \alpha\frac{\partial\mathcal{L}(\mathbf{X}, \Theta_t)}{\partial\theta^k_{ij}} = \theta^k_{t;ij} - \alpha\frac{1}{N}\sum_{l=1}^{N}\frac{\partial\mathcal{L}_l}{\partial\theta^k_{ij}}, \tag{A.4}$$

where α is a parameter called *learning rate*.

Backpropagation is an algorithmic way how to compute the needed derivatives $\frac{\partial\mathcal{L}_i}{\partial\theta^k_{ij}}$ efficiently.

The algorithm is summarized in Algorithm 5, the derivation of it is the following.

- The following extra notation is used:
  - $o^k = \phi^k(e^k) = \begin{bmatrix} o^k_1 \\ o^k_2 \\ \vdots \\ o^k_{r_k} \end{bmatrix}$ is the *output* of k-th layer, o^k_i is the output of i-th node in k-th layer
  - $e^k = \Theta^{k-1}o^{k-1} = \begin{bmatrix} e^k_1 \\ e^k_2 \\ \vdots \\ e^k_{r_k} \end{bmatrix}$ is the *activation* of k-th layer, where $e^k_i = \sum_{j=1}^{r_{k-1}}\theta^{k-1}_{ij}o^{k-1}_j$
- The derivation begins by applying the chain rule for partial derivative:

$$\frac{\partial\mathcal{L}_i}{\partial\theta^k_{ij}} = \frac{\partial\mathcal{L}_i}{\partial e^k_i}\frac{\partial e^k_i}{\partial\theta^k_{ij}}. \tag{A.5}$$

  - Let us denote the first term as

$$\delta^k_i = \frac{\partial\mathcal{L}_i}{\partial e^k_i}, \tag{A.6}$$

  - the second term is computed as follows:

$$\frac{\partial e^k_i}{\partial\theta^k_{ij}} = \frac{\partial}{\partial\theta^k_{ij}}\sum_{l=1}^{r_{k-1}}\theta^{k-1}_{il}o^{k-1}_l = o^{k-1}_j \tag{A.7}$$

- A recursive formula for computing δ^k_i can be derived using chain rule:

$$\delta^k_i = \frac{\partial\mathcal{L}_i}{\partial e^k_i} = \sum_{j=1}^{r_k}\frac{\partial\mathcal{L}_i}{\partial e^{k+1}_j}\frac{\partial e^{k+1}_j}{\partial e^k_i}, \text{where} \tag{A.8}$$

[1] The time indexes of weights (e.g. Θ_t or $\theta^k_{t;ij}$) denote its values at the t-th step of gradient descent

- $\frac{\partial \mathcal{L}_i}{\partial e_j^{k+1}} = \delta_j^{k+1}$
- $\frac{\partial e_j^{k+1}}{\partial e_i^k} = \sum_{l=1}^{r_k} \theta_{jl}^k \frac{\partial}{\partial e_i^k} \phi^k(e_l^k) = \theta_{ji}^k \frac{d}{de_i^k} \phi^k(e_i^k)$

- Hence, substituting (A.6) and (A.7) into (A.5) the following formula is obtained:

$$\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k} = \delta_i^k o_j^{k-1} = \frac{d}{de_i^k} \phi^k(e_i^k) o_j^{k-1} \sum_{l=1}^{r_k} \delta_l^{k+1} \theta_{li}^k. \tag{A.9}$$

- It remains to compute all o_j^k and e_j^k, which is done in the forward phase when calculating the output by (A.3), and all $\delta_j^m = \frac{\partial \mathcal{L}_i}{\partial e_j^m}$ which depends on the specific form of loss function $\mathcal{L}$.

Forward phase (or forward propagation) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Mini-batch gradient descent, [21] In practice, each step of gradient descent is not computed on the full dataset as in (A.4), but on small subsets of the dataset $\mathbf{X}$ called mini-batches. A hyper-parameter $N_B < N$, called *mini-batch size*, is chosen and then for each step a mini-batch $\left\{(x_i, y_i)_{i=1}^{N_B}\right\} \subset \mathbf{X}$ is generated and the mini-batch gradient descent is performed as follows:

$$\theta_{t+1;ij}^k = \theta_{t;ij}^k - \alpha \frac{1}{N_B} \sum_{l=1}^{N_B} \frac{\partial \mathcal{L}_l}{\partial \theta_{ij}^k} \tag{A.10}$$

The main advantages of mini-batch gradient descent are its more robust convergence, i.e. avoiding local minima, and lower memory requirements - only small subset of dataset is stored in memory.

Algoritmus 5: Backpropagation on mini-batches

---

1. Calculate the forward phase for each input-output pair (x_i, y_i) from a mini-batch and store the results $f(x_i)$ (A.3), e_j^k and o_j^k for each node j in layer k by proceeding from layer $k = 0$, the input layer, to layer $k = m$, the output layer.

2. Calculate the backward phase for each input-output pair (x_i, y_i) from a mini-batch and store the results $\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k}$ for every node and every layer.

   - Evaluate the loss term for the final layer $\delta_j^m = \frac{\partial \mathcal{L}_i}{\partial e_j^m}$ for each node.
   - Backpropagate the loss terms for the other layers δ_j^k, working backwards from the layer $k = m - 1$, by repeatedly using the equation (A.8)
   - Evaluate the partial derivatives of the individual loss $\frac{\partial \mathcal{L}_i}{\partial \theta_{ij}^k}$ by using the equation (A.9)

3. Update the weights according to the equation (A.10)

A.2 Convolutional neural network (CNN)

As the name suggests, the convolutional neural networks contain convolution that is a mapping defined as follows.

Definition 7 (Convolution)**.** For tensors x and h the convolution produces a tensor $y = x*h$ that is defined as follows:

$$y_{i_1,i_2,...,i_n} = \sum_{j_1} \sum_{j_2} \cdots \sum_{j_n} x_{i_1-j_1,i_2-j_2,...,i_n-j_n} h_{j_1,j_2,...,j_n} \tag{A.11}$$

□

Definition 8. The convolutional neural network is a fully-connected neural network (see Definition 6) that contains one or more (even exclusively) convolutional layers. □

A convolutional layer can be defined for arbitrary dimension, but since this work considers images, the 2-D convolutional layers will be used.

Definition 9 (2-D Convolutional layer)**.** A 2-D convolutional layer with the parameters *input channels* $C_{in} \in \mathbb{N}$, *output channels* $C_{out} \in \mathbb{N}$ and *kernel size* $k \in \mathbb{N}$ contains:

- C_{out} parameter matrices $h_1, ...h_{C_{out}}$ referred to as *filters*. Each $h_i \in \mathbb{R}^{k,k,C_{in}}$.
- activation function ϕ.

The layer takes $x \in \mathbb{R}^{C_{in},n,n}$ as an input that has dimensions $C_{in}, n, n \in \mathbb{N}$ and gives the output o in the following way:

- $e = [h_1 * x, h_2 * x, ..., h_{C_{out}} * x] \in \mathbb{R}^{C_{out},n_{out},n_{out}}$ is the activation of the layer, where $n_{out} = n - k + 1$,
- $o = \phi(e)$ is the output of the layer.

□

Remark 2. The learning of convolutional neural network uses the backpropagation algorithm from the Section A.1.1. The learned parameters of convolutional layers are contained in the filters h_i from Definition 9. The neural network is differentiable with respect to these parameters and the activation e and output o from Definition 9 can be directly used for the backpropagation algorithm. □

A.3 Generative Adversarial Network (GAN)

Generative Adversarial Network (GAN) algorithm assumes a training dataset $\mathbf{X} \subset \mathbb{R}^n$, $n \in \mathbb{N}$ that is assumed to be generated from an unknown distribution p. The goal of GAN is to learn the unknown distribution p. GAN assumes a known distribution q defined on $\mathbb{R}^d$, $d \in \mathbb{N}$ (q can be chosen as e.g. uniform or normal distribution). GAN finds a mapping $G(z) : \mathbb{R}^d \mapsto \mathbb{R}^n$ such that if a random variable $z \in \mathbb{R}^d$ has the distribution q, then $G(z)$ has the distribution p.

There are mainly two problems associated with creating GAN: 1) how to find G, 2) if we have G, how do we know that $G(z)$, where z is drawn from distribution q, has distribution p.

GAN forms an adversarial system from which G receives updates to improve its performance. More specifically, it trains also a discriminator $D(x) : \mathbb{R}^n \mapsto \mathbb{R}$ that represents a probability of x coming from the training set $\mathbf{X}$ or from the samples generated from $G(z)$. Both G and D are constructed as neural networks $G(x|\theta_G)$, $D(x|\theta_D)$ and their parameters θ_G, θ_D are learnt.

This adversarial game can be mathematically defined as the following minimax problem of a target function[2]:

$$\min_G \max_D E_p\left[\log D(x)\right] + E_q\left[\log\left(1 - D(G(z))\right)\right], \tag{A.12}$$

Practically, the mini-batch stochastic gradient method is used to train the neural networks. The specific algorithm is summarized in Algorithm 6

Algoritmus 6: GAN learning

Input: training set $\mathbf{X}$, noise distribution q, minibatch size m

for *number of training iterations* **do**

Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from the distribution q

Sample minibatch of m examples $\{x_1, ..., x_m\}$ from the training set $\mathbf{X}$

Calculate the discriminator loss:

$$\mathcal{L}_D = -\frac{1}{2m}\left(\sum_{i=1}^{m} \log D(x_i|\theta_D) + \log\left(1 - D\left(G(z_i|\theta_G)\right)\right)\right)$$

Update θ_D by gradient descent

Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from the distribution q

Calculate the generator loss:

$$\mathcal{L}_G = -\frac{1}{m}\left(\sum_{i=1}^{m} \log D\left(G(z_i|\theta_G)\right)\right)$$

Update θ_G by gradient descent

end

Output: The learnt generator G

[2]In practice, instead of minimizing $\log\left(1 - D\left(G\left(z|\theta_g\right)\right)\right)$, maximizing $\log\left(D\left(G\left(z|\theta_g\right)\right)\right)$ is often used for training the generator G.