Department of Mathematics and Computer Science

University of Southern Denmark

Data Science

Offline Reinforcement Learning with Kalman Filters

Author
Sarah Anne Pedersen
Exam nr: 493408
saped13@student.sdu.dk

Supervisor
Melih Kandemir

June 21, 2022

Abstract

English: Offline reinforcement learning can introduce optimal policies for control problems in the world. However, uncertainty and risks are road blockers for the application of current models in critical settings. There is a need for system dynamics models that can handle non-linear environments and distribution shifts. The report addresses dynamics models through behavioral cloning of a collected target policy. A deep Kalman filter is constructed by assuming that the latent state space governs the system dynamics. Furthermore, the state transition distribution and the observation distribution are normal random variables. Therefore we can inferred the latent state through the Kalman filter equation. A neural ordinary differential equation's net is used to model the dynamics of the state transition parameters. The performance of the deep Kalman filter and the baseline models is evaluated through the accuracy, the expected calibration error, and the cumulative reward. In the test, the deep Kalman filter yields a conservative accuracy score, the lowest expected calibration error, and the highest cumulative reward when deployed online. The model would benefit from further testing in different non-linear environments.

Danish: Offline reinforcement learning har et potentiale for at flytte optimale polices for kontrolproblemer fra data til virkelighed. Usikkerhed og risici forhindre anvendelsen af eksisterende modeller i kritiske miljøer. Der er et behov for modeller som tager højde for den underliggende dynamic i et datasæt og som kan håndtere data fra ulinæearer miljøer, samt distributionsskifte fra træning til virkelighed. I rapporten bruger vi behavioral cloning til at lave modeller som håndtere underliggende dynamikker fra indsamlet data. Vi kombinerer deep learning med et Kalman filter og antager at det latent space fanger den underliggende dynamik i dataen. Yderligere antages det at state transition distributionen og observationsdistributionen indeholder normale stokastiske variabler. Vi kan udlede latent state ved brug af Kalman filters filterligning. Vi bruger et netværk af neural ODEs til at modellere dynamikken af state transition parameterne. Kalman Filteret og de to baseline modeller bliver evalueret ved brugen af modellernes nøjagtighed, den forventede kalibreringfejl og den accumuleret belønning modellerne samler i de virtuelle miljøer. Under test af modellerne fandt vi at det dybe Kalman filter giver en konservativ nøjagtighedsscore, modellen havde den laveste kalibreringsfejl og den største samlet belønning når den blev testet i miljøet. Det dybe Kalman filter ville drage fordel af blive testet i forskellige komplekse miljøer for at klarlægge modellens potentiale yderligere.

Contents

List of figures

List of tables

List of Algorithms

Chapter 1

Introduction

Reinforcement learning has a yet to be realized potential to change the world. One of the main blockers is the need for interaction when learning from the environment. By turning Reinforcement learning "offline" and thereby data-driven, we can utilize existing data to learn optimal policies safely. Offline reinforcement learning is limited by uncertainty in the world and - the data. The models often need additional policy fine-tuning before they can be deployed safely online, limiting the models' use to non-critical applications. In reinforcement learning we work with time series trajectories. Many existing time series prediction models assume linearity and stationarity in their applied data sets. These assumptions rarely hold true in the real world. Non-linear time series prediction needs methods that can extract information about the dynamics and the underlying model. Such methods involve reconstructing the state space where the data was sampled. Neural networks are widely used with reinforcement learning due to their impressive results. However, recent literature has shown that deep learning models tend to overshoot in the accuracy of the true predictions a model delivers. This is not desirable in an offline reinforcement learning setting.

In this thesis, we combine model-free and model-based reinforcement learning. We solve for target policy to develop an environment dynamics model. We show how to utilize Open AI gym for non-linear dynamic problems. The environments CartPole and AcroBot are used for data collection and testing. We will construct a Partially Observable Markov Decision Process by assuming the latent state space governs the system dynamics. The Kalman Filter's filtering equation filters for the latent state at an arbitrary time point, which is mapped to an action. We will compare our dynamics model to two baseline models. The models are based on theory from time series prediction, reinforcement learning, behavioral cloning, state space models, and Neural Ordinary Differential Equations. By combining deep learning and state space models we aim to improve the performance of the learned policies and the calibration of the predictions. We will include calibration measurements of the networks' outputs as a performance measure. Furthermore, we measure the cumulative reward collected in the environments and the accuracy score of a test set.

We found that it possible to model the environment's dynamics through the use of ODEs and a Kalman filter. The Deep Kalman filter performed better than

the two baseline models when deployed online. The model delivered a conservative validation accuracy and the lowest expected calibration error. The Deep Kalman Filter requires few parameters and tends to overfit easily. Therefore, early stopping when training the model was used. The Kalman Filter would benefit from additional research to see if the found properties would transfer to more complex time series data.

Chapter 2

Literature review

2.1 Time series classification

Time series classification (TSC) is concerned with making classification based on the trajectory of time steps t. In TSC univariate and multivariate sequences exists. An univariate time series is defined as $X = [x_1, x_2, ..x_T]$, where X is an ordered set of real values. A multivariate time series consist of M different univariate time series with $x^i \in R^T$.[20] In time series prediction several prediction methods can be found: statistical models, state space models, machine learning algorithms, and deep learning.[2] In this thesis we will focus on the latter since we are interested in using deep learning in the context of offline reinforcement learning to model dynamics for real-world like applications. The prior methods handle linear dynamics and stationary data sets and these assumptions rarely apply to the real world.[29] When predicting time series a simple feedforward network have proven to give a good performance in accuracy for non-linear time series.[60][20][2] Due to the non-existing state in feedforward networks, other models that provides context for the sequence of the observations are preferred.[15][20] Convolutional neural networks (CNN's) are quite famous for its breakthrough in image generation. Nevertheless, their underlying mechanisms and assumptions do not fit well with time series and CNNs are more often seen with multivariate time series data. The CNN is often a part of the network architecture rather than a stand-alone element in TSC.[20] A popular type of architecture for sequence processing in deep learning are the recurrent neural network (RNN) family.[2] Especially autoencoders are a go-to model when performing time series prediction or other sequence processing. Autoencoders have two recurrent layers. The first layers process the entire sequence to end and output its state. The state serves as input for the second layer, which processes the state and creates its output. These autoencoders have been applied in different fields and different variations and combinations with other tools.[4][2][22] Another exciting development in deep learning sequence processing is Transformers which utilize self-attention to access long-range context regardless of distance in the network.[55][32] Furthermore, combinations of deep learning models and state space models can be seen in the literature for handling uncertainty in time series predictions.[40][46][45]

2.2 Uncertainty Calibration

Deep learning models are included in our daily decision pipelines. They can be found in the fields of finance, healthcare, and more.[18][24] A common mistake when working with deep learning is to believe that modern neural networks only produce true predictions. However, a high accuracy does not only equal correct predictions. Therefore it is crucial to measure when the predictions can be incorrect. In deep learning we work with two types of uncertainty aleatoric uncertainty and epistemic uncertainty. Aleatoric uncertainty is uncertainty in the data itself due to noisiness. No matter how much data you collect, the noise will still be in the underlying process and the only way to change the aleatoric uncertainty is to change to better sensors that can collect more accurate data. Epistemic uncertainty is the uncertainty of the model, and it reflects the confidence of the predictions the model produces. Epistemic uncertainty can be mitigated by collecting more training data. In this thesis we are interested in the epistemic uncertainty, and thereby how confidence our model is in its predictions. We can use the knowledge of the confidence to estimate when the model can not provide a reliable answer, and when additional training data is needed. Estimation of epistemic uncertainty is difficult. However several approaches has emerged in the later years.[1] Bayesian networks are used for uncertainty estimation. The models produce a sampling distribution for each weight in the network. The idea may seem smart, but the process is memory demanding and challenging for real time operations.[43] Another is post-processing techniques before implementation of the algorithms. Temperature scaling is fairly easy to implement since it is applied to the output logits of the network; and thereby softens the softmax of the model's output and does not affect the model's accuracy.[16] Uncertainty calibration error (UCE) utilizes regularization of the network to normalize through Bayesian inference theory.[28] Platt-scaling is often found in time series and reinforcement learning literature and has generated nicely calibrated predictions.[26]

2.3 Offline Reinforcement learning

Reinforcement learning (RL) is concerned with finding the optimal behavior for an agent to perform by a given control task. In the RL framework methods allows a system to learn a strategy, also known as policy π, for making good decisions by observing the systems own behavior in an environment and using mechanisms for improving their actions through reinforcement.[6][49] RL has been an active research area for years. The blooming of deep learning has enabled several impressive results in areas such as games: Atari and Alpha go, robotics where Boston dynamics has found success and Natural language Processing.[36][19][31] Figure 2.1. demonstrates how an agent learns from interaction with the environment. The important thing to notice here is that the agent is online with the environment and updates its policy π based on the feedback from the environment.

[1]https://www.youtube.com/watch?v=toTcf7tZK8c

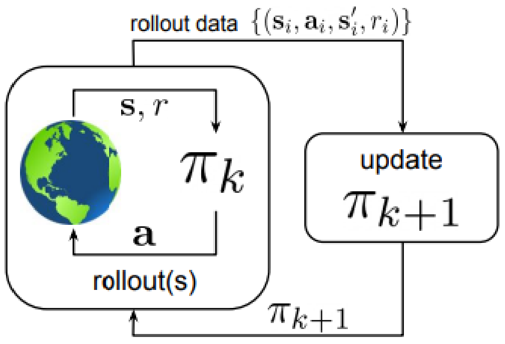


Figure 2.1: Online reinforcement learning paradigm [30]

The combination of RL and deep learning is called deep reinforcement learning (DRL).[33] Especially one distinct deployment of DRL is called a deep Q-network (DQN). The agent interacts with an environment through a sequence of observations, actions and rewards (s, a, s', r) as in figure 2.1. The goal for the agent is to select actions that maximize cumulative future reward. DQN uses a neural network to approximate the optimal action-value pairs. This kind of set up has proven to learn successfully control policies in different environments with minimal prior knowledge and abstract representations of the environment.[36] RL may seem promising to achieve artificial general intelligence.[36] However, the demand for interaction with the environment makes it challenging to move RL from a simulated environment to real-world application. The real world is complex. Environments can be partially observable, the rewards are unspecified multi-objective or sensitive and there are rarely room for trail and error approaches.[14]

Offline reinforcement learning (ORL), which is also known as batch reinforcement learning[30][13], uses off-policy RL algorithms. This covers all RL algorithms that can collect policies from existing data sets. Q-learning, actor-critic algorithms that uses the Q-function, and several model-based RL algorithms falls within the category of off-policy algorithms.[30] In literature the term fully off-policy can be found which means that no additional data was collected online to improve the policy.[30] By turning RL offline and into a data-driven discipline, the adoption of the framework for real-world applications can be more attractive to industries where an online approach would be cost demanding and potentially dangerous. In figure 2.2 the process for learning a policy by ORL is seen. Notice that the training of π is isolated form the deployment, and the trajectory of state, action, next state and reward are collected prior to the training phase.[30]

One model-free ORL algorithm is the previous mentioned DQN. DQN was applied to a data set of 2600 Atari games which showed promising results.[1] Contrary to previous existing work which had found the DQN to fail in an offline setup.[1][56] Many other algorithms for model-free ORL exists, and many of them uses Q-learning in some version.[50] Another method for ORL is imitation learning. In imitation learning an agent learns a behavior with an unknown

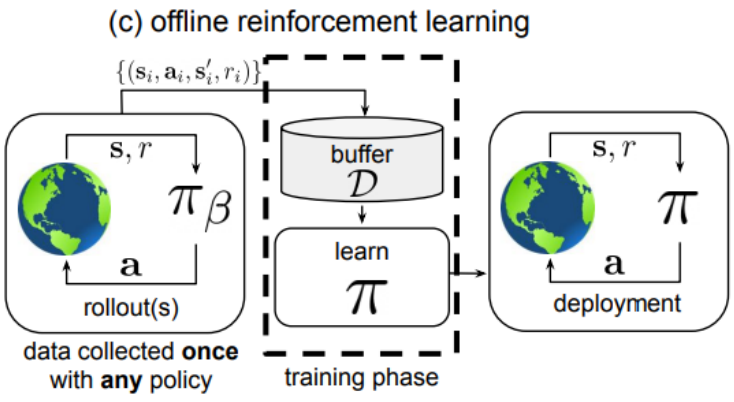


Figure 2.2: Offline reinforcement learning paradigm [30]

cost function to mimic an expert policy.[17] Specially pessimistic and minimalist approaches have been studied to minimize risk.[41] One of the most fully off-policy friendly imitation learning methods is behavioral cloning since it minimizes the need for additional interaction and fine-tuning online.[54] The goal is to learn the relationship between states and optimal actions like in supervised learning.[39] Imitation learning has shown to be effective in learning policies fast for sequential predictions.[48] Both regular feedforward networks and Long Short-Term Memory (LSTM) networks have been tested and found useful for learning policies.[48] Behavioral cloning (BC) is also related to model-based learning in RL since it makes use of learned models of the environment.[54] When mapping from state to action it was proven that BC required fewer post demonstrations after deployment.[54] Furthermore, the BC models can be transferred across tasks which optimizes training time.[51] Another promising area is the use of transformers for sequence modelling which is found effective for imitation learning and offline reinforcement learning. However, for now it is deemed to be slow and resource-intensive compared to models that predict single time steps.[21]

Risk and uncertainty are essential issues in ORL and ORL algorithms often perform best when additional data is collected after policy deployment which fine-tunes the policy.[37][30] Training offline from fixed data logs may not be sufficient due to a lack of exploration or the underlying dynamics learned is hard to generalize.[14] Uncertainty is a blocker for adopting ORL to sensitive control tasks where post-training may not be desirable.[30] One could say that a data set just needs to be large enough, but when is a data set large enough to accommodate the extrapolation risk and distribution shift, and due to poor calibration can we even guarantee that our model suggests the right actions? Model-based ORL may seem like an ideal fit for the learning dynamics of the environment, but this type of learning is also challenged by the complexity of different MDPs and predicting for longer horizons. A combination of model-based and model-free learning yields the best results for now.[30]

Chapter 3

Research Methodology

This chapter will cover the problem, process and theories behind the thesis experiments. Section 3.1 covers the problem statement and the experimental pipeline. Section 3.2 and 3.3, cover the Reinforcement learning preliminaries and deep supervised learning theory. Section 3.4 covers state space models, Section 3.5 Neural ODEs, 3.6 the Deep Kalman filter and section 3.7 Evaluation Metrics.

3.1 Research process

Reinforcement learning (RL) is a core technology in artificial intelligence (AI) for building intelligent robot control skills. A fundamental road blocker for RL algorithms is to find widespread use in real-world applications. The need to interact with the target environments many times and learn from repetitive trial and error, such a setup is infeasible, costly, or dangerous for many use cases such as robotic surgery and autonomous driving. Offline reinforcement learning emerges as a solution by aiming to fit a control policy to data observed from the target environment prior to the learning process. Then the modeler has the chance to do prior checks to decide whether to implement the learned policy into the target environment. In this thesis, we will investigate solutions that can accommodate uncertainty in the adoption of offline reinforcement learning. We will develop an environment dynamics model, which can handle non-linear environments and test it against other baseline models found in the literature. We aim to predict the action a_t given the current state s_t for a single time step. We conduct the experimental pipeline in figure 3.1 on collected policies from two Open AI Gym environments.

As seen in 3.1, Q-learning is applied to the environments to efficiently solve for target policies. Our target behavior policy will collect 4-tuples of the current state, action, reward and next state (s_t, a_t, r, s_{t+1}). By utilizing open AI Gym, the problem is made dynamic and non-stationary.

We will develop an environment dynamics model. We construct a Partially Observable Markov Decision Process (POMDP) by assuming a latent state space that governs the system dynamics. We will assume that the state transition and observation distribution are normal random variables. Hence the latent state of

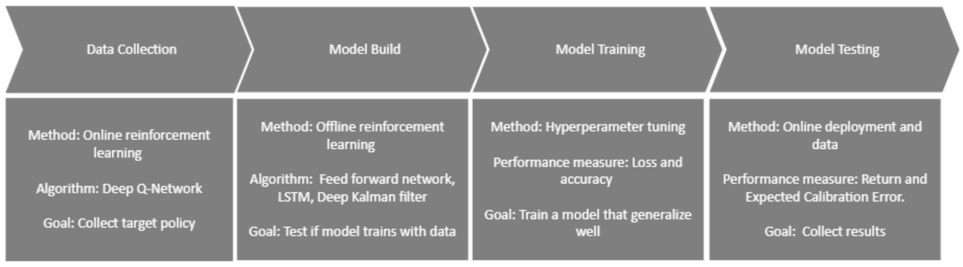

Data Collection
Model Build
Model Training
Model Testing
Method: Online reinforcement learning
Algorithm: Deep Q-Network
Goal: Collect target policy
Method: Offline reinforcement learning
Algorithm: Feed forward network, LSTM, Deep Kalman filter
Goal: Test if model trains with data
Method: Hyperperameter tuning
Performance measure: Loss and accuracy
Goal: Train a model that generalize well
Method: Online deployment and data
Performance measure: Return and Expected Calibration Error.
Goal: Collect results

Figure 3.1: Experimental pipeline

an arbitrary time point can be inferred analytically via Bayesian filtering using the Kalman filter equations. Furthermore, we will assume that the parameters of the state transition distribution are governed by a Neural Ordinary Differential Equation (Neural ODE). A feedforward network and an LSTM are our baseline models. The feedforward network gives an indication of the complexity of the time-series dynamics, and the LSTM is applied as a non-linear state space model to learn environment dynamics. These models are tuned and trained based on the accuracy and loss of our collected data. As the last activity in figure 3.1, we benchmark the models based on expected calibration error from the collected test data and the return from the online environment when the models are deployed.

3.2 Offline RL preliminaries

This section will cover the reinforcement learning theory applied in the thesis. Most material will originate from the reinforcement learning course's lecture notes.[1] [2] [3]

3.2.1 Building blocks of RL

To do offline Reinforcement learning (ORL) we need the main building blocks of a RL system: agent, environment, states, actions, policy, a reward signal, value functions and a task to solve.

An agent is the learning object. It can consist of one or more: policies, value functions, and models. The agent fulfills actions, and the system has the power to change the agent's actions based on, e.g., the value functions expectations.

An action on a given time step a_t is a part action space $a_t \in A$ given the current state. Actions are used to transition between states of the environment. The agent's goal is to maximize its accumulated reward through its actions. The agent can choose to sacrifice immediate reward to maximize the long-term. Actions like in real life have long-term consequences. The agent can choose them based on the values given by a value function. The agent seeks actions which

[1]https://imada.sdu.dk/ kandemir/rl-lecture2.pdf
[2]https://imada.sdu.dk/ kandemir/rl-lecture3.pdf
[3]https://imada.sdu.dk/ kandemir/rl-lecture6.pdf

bring the highest values and maximize overall return.[49]

The environment is what the agent is 'living' in and learning from. The agent has limited or no control of the environment. Nevertheless, the environment produces information that is relevant for archiving the overall goal. The environment state is information that describes the environment. The state can be hidden or partially observable for the agent.

State s_t contains all relevant information about the environment of a given time step t. The state is part of the state space $s_t \in S$ of the environment. It is a set of features that the environment can take on a given time step.

Policy π is the strategy of the agent. It defines the agent's behavior in a given state. The policy maps states to actions. The policy is optimized to find the optimal state-action pairs. The policy is modified through time as the agent gain more experience and calculates more precise values. Policies may be stochastic, and specify probabilities for each action.

The reward signal defines the goal of the reinforcement learning problem. On each time step the environment sends the agent a numerical reward r based on the agent's previous action. The ultimate goal is to find a policy that maximizes the overall return.[49]

Value functions help the agent predict which behavior is best in the long run. It defines the value of a state and estimates the total return an agent can expect to accumulate over the future given the current state. The values help the agent evaluate its actions. Values are estimated and re-estimated based on the agent's trajectory.

A task is what the agent is solving. It can be described as episodic if the state sequences break naturally and is continuous e.g. as in a board game.

Markov Decision Process

Markov Decision Process (MDP) is a model for stochastic decision problems.[47] It is an abstraction of the problem of goal-directed learning from interaction. It proposes to reduce the problem of goal-directed behavior to tree signals passing back and forth between the agent and its environment, see figure 3.2.[49]

- The state s can be discrete or continuous and can both be high or low dimensional. The most important thing is that the state is Markovian.
- The action a in the MDP can also be discrete or continuous.
- The reward, r is generated from the reward function by the state and action.

4

[4]CS 182: Lecture 15: Part 1: Policy Gradients, https://youtu.be/$_AYvYUrDohw?t = 386$

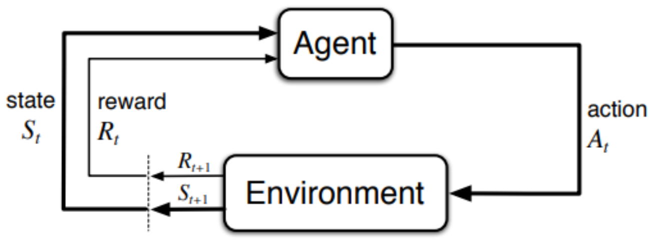


Figure 3.2: Interaction between agent and environment in Markov decision process [49]

Through interaction between the agent and the environment as seen in figure 3.2 a trajectory of signals are created: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \ldots R_n, S_n, A_n$, over time[49]. A state is Markovian if it fulfills the Markov property. Any environment, task, or game that exhibits the Markov property, is said to be a MDP. The Markov property assumes that the current state alone contains enough information for the agent to choose an optimal action to maximize future rewards: A random state S_t satisfies the Markov property if:

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, ..., S_t) \tag{3.1}$$

Or equation 3.1. in words, the future is independent of the past. A normal MDP assumes that the environment is fully observable. This means the current state captures all relevant information from history. [49]

Important building blocks of the MDP.
The Markov process is a simple version of the MDP.[47] It glues each time step together and ensures that the model of the environment can run for an arbitrarily long time.[35] It enables the transition between states. The definition of the Markov process:
A Markov process (MP) is a tuple of two entities $< S, P >$ where:

S is the set of environment states: $S_t = s$ with $s \in S$ at all time steps.

$P = P(S_{t+1}|S_t)$ is the environment dynamics model.

We introduce the reward in the Markov process to measure if we are happy with the given state or not. The Markov reward process tells the agent about the reward it receives from the state current.
A Markov reward process (MRP) is a tuple of 4 entities $< S, P, R, \gamma >$ where:

S is the set of environment states: $S_t = s$ with $s \in S$ at all time steps.

R is the set of rewards, $R_t = r$ with $r \in R$.

$\gamma \in [0, 1]$ is the discount factor.

$P = P(R_{t+1}, S_{t+1}|S_t)$ is the environment model, which is written as

$$P(R_{t+1}, S_{t+1}|S_t) = P(R_{t+1}|S_{t+1}, S_t).P(S_{t+1}|S_t). \tag{3.2}$$

Where:

$$P(R_{t+1}|S_{t+1}, S_t).$$

is the Reward model, and:

$$P(S_{t+1}|S_t).$$

is the transition model.

The agent needs to know the overall return it can get given that the agent solves the task.
The return is defined as the cumulative discounted reward starting from time step t.

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{3.3}$$

$\gamma^k R_{t+k+1}$ defines the present value of the future reward R_{t+k+1}.
$\gamma = 0$ discount all future reward and $\gamma = 1$ gives a model which is strongly dependent on future reward. The return is calculated by recursion, which makes it possible to do problem-solving by breaking problems into smaller sub-problems and solving those problems recursively combining the solutions and thereby solving the original problem.[49]

The value functions are crucial for the agent to observe, evaluate and improve its behavior. In other words, sharpen the heuristic search for optimal state-action pairs. A state-value function (value function) is the expected return from the starting state following the policy:

$$v(s) = E[G_t|S_t = s]. \tag{3.4}$$

An action-value function is the expected return from the starting s, taking action a and following the policy π.

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]. \tag{3.5}$$

The state value function tells the agent how good it is to be in a state and the action value function provides an estimate of how good it is to take a certain action in a certain state. These two functions are connected and tells the agent which action-state pairs are most promising and thereby the value function of the policy π.[47] The combination is seen in equation 3.6.

$$v_\pi(s) = \sum_{a \in A} \pi(A_t = a|S_t = s) q_\pi(s, a). \tag{3.6}$$

We now know the building blocks of the MDP, let us define it. The Markov

Decision Process MDP has 5 entities $< S, A, P, R, \gamma >$ and a policy function π, where:

S is the set of environment states, $S_t = s$ with $s \in S$, for all time steps t.

R is a set of rewards, $R_t = r$ with $r \in R$.

$\gamma \in [0, 1]$ is the discount factor.

A is the set of actions such that $A_t \in A$ follows $A_t \sim \pi(A_t|S_t)$.

$P = P(R_{t+1}, S_{t+1}|S_t, A_t)$ is the environment model that can be decomposed into the reward model $P(R_{t+1}|S_t, A_t)$ and the transition model $P(S_{t+1}|S_t, A_t)$.

Not all problems are MDPs, but a process can be modified to contain more information and thereby fulfill the Markov property.[49] Many versions of the Markov decision processes can be found in literature, for this thesis we will work with a partially observable Markov decision process (POMDP) which is an extension of the above-mentioned. This is used for when the environment is not fully observable e.g. road conditions when driving. The Partially Observable Markov Decision process POMDP is identified as a 6 entities $< S, A, P, O, R, \gamma >$ and a policy function π, where:

S is the set of environment states, $S_t = s$ with $s \in S$, for all time steps t.

O is the set of observations such that $O_t = o$, $o \in O$, for all time steps t.

R is a set of rewards, $R_t = r$ with $r \in R$.

$\gamma \in [0, 1]$ is the discount factor.

A is the set of actions such that $A_t \in A$ follows $A_t \sim \pi(A_t|S_t)$.

$P = P(R_{t+1}, O_{t+1}, S_{t+1}|S_t, A_t)$ is the environment model that can be decomposed into the reward model $P(R_{t+1}|S_t, A_t)$, observation model $P(O_{t+1}|S_t)$ and the transition model $P(S_{t+1}|S_t, A_t)$.
The POMDP allows us to introduce potentially non-markovian observations into the process.[5]

3.2.2 Model-free reinforcement learning

In model-free RL we want to predict given rewards accurately and thereby maximize our return. In model-free RL the agent does not have to learn the underlying dynamics of the environment, since it does not have use for it. It can just learn based on experience through trial and error and evaluate the consequences of its actions through some value function.[49]

[5]CS 182: Lecture 15: Part 1: Policy Gradients, $https : //youtu.be/_AYvYUrDohw?$

Q-learning

Q-learning is a model-free reinforcement learning algorithm. It makes it possible for an agent to learn how to act optimally in a Markovian domain by observing the consequences of the actions without the need to build maps of the domain.[57][11] Q-learning is not a new idea, and some of the first algorithms have been done in tables and have later been developed into deep Q-networks.[11] Q-learning is related to temporal differences methods (TD). The basic idea of TD is to match the learner's current predictions for the current input more closely to the following prediction for the next time step.[52] Q-learning utilizes the action-value function which we saw in section 3.3.1, equation 3.5. The Q-learning algorithm aims to learn the optimal action-values by predicting the values of each state and action pair and thereby find the best action at the given state to maximize the return. In Q-learning, the agent uses exploration and exploitation. Exploration is when the agent "explorer" a random action that has not already been performed. Exploitation is when the agent utilizes existing knowledge to operate in the environment it is in.[53]

In equation 3.7. the Q-learning formula is seen.

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \text{max}_a Q(S',a) - Q(S,A)] \tag{3.7}$$

In equation 3.7. the variables are defined as: $Q(S,A)$ is the Q-values. On the left side it is the updated q value, and the right side is the current q-value. α is a step size value e.g. the learning rate in a neural network. R_{t+1} is the observed reward. γ is the discount factor. $\text{max}_a Q(S',a)$ is the max Q value for all actions. Algorithm 1 contains the process for Q-learning.

Algorithm 1: Q-learning

```
Initialize Q(s,a), $\forall s \in S$ , $a \in A$, $Q(s_{end}) = 0$
for $e \in episode$ do
    Initialize S
    for time step in episode do
        Choose A from S using policy derived from Q e.g. $\epsilon - greedy$
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \text{max}_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
        End when $S$ terminate
```

Bellman equation. The Bellman equation is essential in reinforcement learning and origins back in dynamic programming.[5] In dynamic programming, algorithms seek to simplify complex problems by breaking them into smaller sub-problems and solving the sub-problems recursively.[5] When working with the value functions for state-action pairs, see section 3.3.1, equation 3.4, 3.5, the bellman equation tells us how to update the Q-function when rewards are observed.

$$Q_\pi(s_t, a_t) \leftarrow r_t + \gamma \bullet V_\pi(s_{t+1}) \tag{3.8}$$

where:

$$V_\pi(s_{t+1}) = \max[Q_\pi(s_{t+1}, a)] \tag{3.9}$$

It provides the answer to: what overall return can the agent expect, and what is the value of the state the agent is in and thereby helps the agent to evaluate the advantage or disadvantage of the state.[49]
Furthermore, the Bellman equation can help solve each individual policy π given the MDP. One MDP can have several optimal policies. However, this is not feasible in many real-world applications. The process of value function update by the Bellman equation is seen in Algorithm 2.

Algorithm 2: Bellman equation: Value function update.

```
1: Initialize V(s) with random values
2: for $s \in S$ do
3:     for $a \in A$ do
4:         $Q_\pi(s_t, a_t) \leftarrow r_t + \gamma \bullet V_\pi(s_{t+1})$
5:         $V(s) \leftarrow \max Q(s, a)$
```

3.3 Model-based learning

In model-based RL the agent makes decisions based on a constructed model of the dynamics of the environment the agent is operating in. When knowing the environment dynamics the agent can plan a sequence of actions to achieve the final goal. The most straight-forward approach to train an environment model is by having a separate deep learning network that predicts the future states. Potentially we can predict a longer sequence of states by giving a model a state (s_t), let it predict the next state (s_{t+1}) and feed the predicted state to the model to predict (s_{t+2}). The accuracy of such prediction would depend on the randomness of the environment and the accuracy of the model.[49] In our case, we are interested in predicting s_{t+1} and matching it to an action a_t.

This is not right

One of the challenges with offline model-based RL is the distributional shift between training data and the distribution the model encounters when deployed to the real world.[30] Within offline model-based reinforcement learning several methods exists. One has already been addressed which is the naive approach where a policy is trained directly on offline data. However, it is not really useful when it comes to uncertainty. Other methods involves uncertainty modification, fine-tuning online or both. Different combinations of model-free and model-based methods have proven to be effective for ORL e.g. for policies for mobile robot navigation.[30] One way to deal with uncertainty and risks are developing policies with safe regions of the state space, this has been done successfully with deep imitative models.[42] MoREL and MOPO aims to use conservative value estimates to create bounds for performance.[30]

Should have been in lit revies

3.3.1 Imitation learning

Imitation learning is related to RL. The main difference between ORL and imitation learning is how the data is collected for training. In imitation learning an expert policy collects data from the environment, while in offline reinforcement learning any policy can be used to collect data for the training phase of a policy.[27] When that said behavioral cloning (BC) and model-based ORL interlace in literature and system dynamics can be learned with BC to predict actions. This can be done by learning the inverse dynamics model of the system.[54] The goal is to learn a policy given a trajectories of an agents MDP. In BC deep learning moves from predictions to decision-making with overall abstract goals. The goal is for the neural network to provide "actions" to complete a task. In BC we exploit existing knowledge of deep supervised learning to learn policies π for control tasks.[54][58] We estimate the parameters of the model that fits best to the target policy. This means we map a model's parameters to the joint probability of a set of observations and thereby estimate a policy. The terminology in behavioral cloning:

- o_t for observation or s_t for state is the input to our neural network.
- $\pi\theta(a_t|o_t)$ is policy by action given observation.
- a_t, action, is the output.

- The subscript t indicates that we work with time steps in a sequence.

s_t and a policy can be observed by $\pi\theta(a_t|s_t)$, if that is the case the environment is fully observable. States differentiates from observations by describing multiple elements in the space, e.g., speed, angle, of a car. Where observations only describes if a car is seen in an image or not. States fit the Markov property so the current state contains enough information about the previous state to predict the next time step. Through observations it is not possible to predict the following observation without the previous ones. For a trajectory T we have state action pairs (s_t, a_t). The policy probability $\pi(a_t|s_t)$ of taking a_t given the state s_t and the transition probability $T(\bullet)$ of arriving in s_{t+1} is given by the action pair (s_t, a_t), and is expressed in the equation 3.10. and 3.11.

$$T = (s_0, a_0), (s_1, a_1), \ldots, (s_n, a_n) \tag{3.10}$$

Should have been O

$$P(T) = \Pi_{t=0}^{n} \pi(a_t|s_t) T(s_{t+1}|a_t, s_t) \tag{3.11}$$

The likelihood would be the product of all probabilities of each individual trajectory of a data set.

$$D = T^1, \ldots, T^n \tag{3.12}$$

$$L(D) = \Pi_{i=1}^{n} P(T^i) \tag{3.13}$$

By tuning the parameters in a supervised learning model we can approximate the maximum likelihood estimation of action given state.[6]

Algorithm 3: Behavioral Cloning from target policy

```
Require: Target policy form data set, $D : [(s_1, a_1)..(s_T, a_T)]$
Require: Initialize model parameter randomly,$\theta_{BC}$,
1: while not done do:
2:     sample sequence from $T(s_k, a_k) = [(s_{k:k+k}, a_{k:k+k})] \sim D$
3:     Split $T(s), T(a) = T(s_k, a_k)$
4:     $P(a_k|s_k) = model(T(s))$
5:     Compute action loss: $L_{BC} = -\sum_{c=1}^{M} y_{a,c} \log(p_{a,c})$
6:     Update model$\theta$ with gradient step to maximize $L_{BC}$
```

Loss and likelihood same notation

To miximize likelihood

Supervised learning and BC differentiate by dependence and independence of observation. In supervised learning, all observations are assumed to be independent. In BC because we are dealing with control tasks, some of our observations can be dependent and affect later decisions made. We will include three models in this thesis that utilize BC and inverse modelling. A very simple inverse model

[6]https://ml.berkeley.edu/blog/posts/bc/

Mix op in notation:
Maximize parameters likelihood
minimize los of model

takes a state s_t and the next state s_{t+1} and maps it to the action at current state a_t.[59] The models we will develop are more complex and take uncertainty into account. In algorithm 3 an example of training for inverse modelling is seen. This will be the high-level approach for training the models for the thesis on line 4 s will be a sequence of states, and the model will output the probabilities of the probability distribution for the action space.

Depending on type of model. FF is just random

Is this even true

3.3.2 Deep learning and Neural networks

This subsection is concerned with the theory of supervised deep learning and neural networks. In the thesis, we will use supervised learning for behavioral cloning and deep Q-learning and deep learning models in different versions. To simplify and avoid confusion, we keep the notation from the RL theory in this subsection.

Deep learning is a sub-field of artificial intelligence and machine learning and falls within the area of representational learning. It differentiates from other modern machine learning algorithms by the increased input representation through the several layers that comprise the neural network. The feature abstraction increases and thereby mapping several input representations are possible. To perform supervised deep learning, we need the same basic ingredients for supervised machine learning:

- Input data for the algorithm noted as s
- Examples of the expected output, our "ground truth," noted as a
- A performance measure of the algorithm. This will be the distance between our output of the model and the ground truth, noted $L(\Theta, a, a')$

During training, the algorithm transforms the input into a meaningful output, which means the algorithm learns different meaningful representations of the input in different layers. The meaningfulness of the representation is determined by output's a' distance to the ground truth a. The distance measure is used to optimize the weights and thereby the representations of the input in the layers.

Feedforward networks

feedforward networks consist of densely connected layers. For each input an individual transformation is conducted, and no state is kept between the inputs. feedforward networks can also be illustrated as acyclic graphs since no knowledge flows backwards in the model. If a sequence is passed to a feedforward model it has to be done in one input as one data point.[15] This may seem counter-intuitive for time-series data since we know that dependence can occur between time steps. However, feedforward networks have proven to be a good baseline for comparison with other models for time-series predictions since a feedforward network's performance indicates how complex and dependent the data sequences are.[60] Furthermore, feedforward networks have proven a high accuracy on non-linear time-series forecasting.[60]

A neural network consists of layers in each layer f' a transformation of the input s is produced as in equation 3.14.

$$f' = (W * s) + b. \tag{3.14}$$

The weights W parameterize the layers and finds the optimal set of values θ for the W and biases b. The optimal θ values are important to map an input to the target output. When a new network is constructed θ are initialized randomly. Through training θ is adjusted which improves the mapping from a' to a. In other words, the network approximates a function f so s can be mapped to a as seen in equation 3.15.

$$a' = f(s; \theta). \tag{3.15}$$

[15]
In figure 3.3 a simple feedforward neural network is displayed. The network comprises of an input layer, one hidden layer and an output layer. Within the layers hidden units are located. All hidden units in one layer are connected to the hidden units in the previous layer. The weight of the weighted sum between the hidden units are the strength of the connections between the layers. In figure 3.3. the different colors of lines indicate different connections between the hidden units. The bias is illustrated by different black, white and grey scaled colors in the hidden units of the hidden layer and the output layer. The bias indicates the activation of the hidden unit. This should not be seen as active or not, but as a percentage of activation and therefore the grey scaling is used.[7]

[7]But what is a neural network? — Chapter 1, Deep learning, $https : //www.youtube.com/watch?v = aircAruvnKkt = 2s$

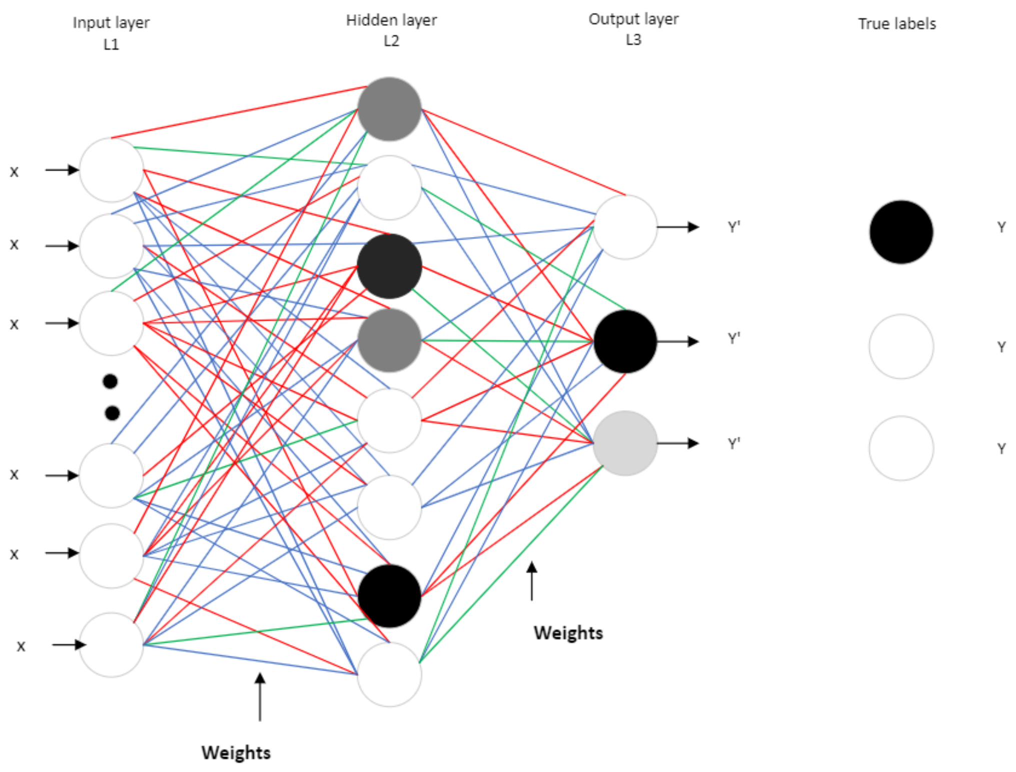


Figure 3.3: Feedforward network

Activation functions introduces non-linearity in the transformation of the mapping from input to output. This expands the hypothesis space and increases the representation possibilities of the network. Without the activation functions, the only transformations happening within the layer would be linear as in equation 3.14. One of the most common activation functions is ReLu. ReLu will output the input directly if it is positive. Otherwise, it will output zero. It is a decision between max values as seen in equation 3.16.

$$a(f') = \max[0, f']. \tag{3.16}$$

An activation function of a hidden unit would be applied like in the equation 3.17.

$$Unit(s) = g(f(s)). \tag{3.17}$$

For the output layer we want a percentage of likelihood for each label. This will activation functions like Sigmoid and Softmax provide. In equation 3.18 the calculation of Softmax is seen.

$$p(y = i|x) = softmax(f\theta(x))[i] = exp(f\theta, i(x))/\Sigma_{j=1}^{N}(f\theta, j(x)). \tag{3.18}$$

[15]
The likelihood output makes it possible to observe how far the network's output

is from the target. The loss is used as a feedback signal to adjust the layers' weights in a direction that will lower the loss score. An example of a loss function would be cross-entropy which is used for multi-classification problems. In equation 3.19 C is the output vector of the Softmax score and Q would be the true labels. In figure 3.3. C is a vector of the three variables Y', and L is a vector of the Y variables. The true labels would be encoded so they match a distance from zero to one.

$$L(C, Q) = -\sum_i Q_i log(C_i). \tag{3.19}$$

[8]

The Gradient is used in several optimization algorithms by most modern machine. The gradient finds the local minimum of a differential function. The gradient of a function gives the direction of the steepest descent by taking the negative of the gradient we move towards the deepest decent. To get an intuitive understanding of gradient descent imagine a function's slope. If the gradient of that slope is positive we move in the opposite direction and vice versa. When this is done continuously we will approach a local minimum of the function.[15] Equation 3.17. is a function of a unit in a hidden layer. The calculation for derivative of that unit is in equation 3.20.

$$\frac{\delta unit}{\delta w} = \frac{\delta unit}{\delta f(s)} \frac{\delta f(s)}{\delta W}. \tag{3.20}$$

The derivative is the slope of a function at each point tangent line. In the equation 3.20 we have two partial of the derivative since we have two variables. The gradient ∇ of our function is the collection of all the partial derivatives in a vector.[9]

A local minimum is not optimal for a network, we want to optimize for the global minimum of multiple functions chained together. The back-propagation algorithm enables this so we can train a neural network. In back-propagation the cost function and the gradients of the network is used to optimize the functions approximation. The cost function indicates how bad our network overall performance is on our data set. The gradient of the functions tells us how to adjust the weights and biases most efficiently to decrease the overall cost of the network. Back-propagation uses the vector chain rule to compute the gradient ∇ of the network. The chain rule tells us how to find the derivative of a composite function $f(g(x))$. In equation 3.21. the chain rule is seen.

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x) \tag{3.21}$$

To perform back-propagation we must:

- Make a forward pass to compute the cost of the network.
- Compute the local gradients.

[8]https://medium.com/data-science-bootcamp/understand-cross-entropy-loss-in-minutes-9fb263caee9a

[9]Tutorial 96 - Deep Learning terminology explained - Back propagation and optimizers $https://www.youtube.com/watch?v = KR3l_E fINdw$

- Make a backward pass to compute the derivates of $dLoss/dWeights$ using the chain Rule.

Algorithm 4 demonstrates a forward pass of a feedforward network. A single input x is used for simplicity and a linear transformation and activation are computed for the input at the lines 3 and 4. The output from the last layer is named y' as our prediction. The overall cost J is computed by the loss between our prediction y' and target y. Θ comprises all the parameters both weights and biases. This will correspond to moving from the input layer to the output layer in figure 3.3.

[15]

Algorithm 4: Feedforward pass

```
Require: Depth of network, $l$
Require: the weights matrices of the network, $W^{(i)}, i \in 1, ..., l$
Require: the bias parameters of the network, $b^{(i)}, i \in 1, ..., l$
Require: the input to process, $x$
Require: the target output, $y$
  $h^{(0)} = x$
  for $K - 1, \dots$ do
      $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
      $h^{(k)} = f(a^{(k)})$
      Get Labels $y' = h^{(l)}$
      Get Cost function $J = L(y', y)$
```

COST og LOSS skal rykkes ud

When the forward pass is achieved the gradient is computed by the output layer. The gradients are computed for each function by the chain rule from the output layer to the input layer, figure 3.3. This is called a back propagation in algorithm 5 you see the process. When this is done the gradient is given to a gradient-based optimizer.

[15]

Algorithm 5: Feedforward network, backward computation

```
$g \leftarrow \nabla_g J = \nabla_g L(y', y)$
for $K = l, l-1, \ldots 1$ do
    $g \leftarrow \nabla_{a^{(k)}} J = g \cdot f'(a^{(k)})$
    $g \leftarrow \nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \theta)$
    $g \leftarrow \nabla_{W^{(k)}} J = g h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Theta)$
    $g \leftarrow \nabla_{h^{k-1}} J = W^{(k)T} g$
```

Gradient descent can not just be applied to the network since several optimization challenges exist in the landscape of a network. We need a good optimizer for the job which can contribute with different tools for handling challenges like local minimum, saddles and more. Adam is a popular choice as optimizer. It uses an adaptive learning rate, it is a fairly robust optimizer and often works well with its default values.[15] In algorithm 4 the Adam algorithm is displayed.

[15]

Algorithm 6: The Adam Algorithm

```
Require: : Step size $\epsilon$ (default 0.001)
Require: : Exponential decay rates for moment estimates, p1 and p2 in the
  [0,1). default (0.9-0.9999)
Require: : Initial parameters $\theta$.
  Initialize 1st and 2nd moment variables s=0, r=0.
  Initialize time step t = 0
  while stopping criterion not met do
      Sample minibatch m
      Compute gradient: $g \leftarrow \frac{1}{m} \nabla\Theta \sum_i L(f(x^{(i)} < \Theta), y^{(i)})$
      $t \leftarrow t + 1$
      Update $s \leftarrow p1s + (1 - p1)g$
      Update $r \leftarrow p2r + (1 - p2)g \bullet g$
      Correct bias $\hat{s} \leftarrow \frac{s}{1-p_1^t}$
      Correct bias $\hat{r} \leftarrow \frac{r}{1-p_2^t}$
      Compute update: $\delta\Theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$
      Apply update $\Theta \leftarrow \Theta + \Delta\Theta$
```

Recurrent neural networks

Recurrent neural networks (RNN) differentiate from feedforward networks by having a state between inputs where information seen so far is kept. Therefore they are specialized in processing sequence data.[44] Each element of the output is a function of the previous elements of the output. In short RNNs have a recursive property. Each output element is produced using the same update rule applied to the previous outputs. This results in sharing of parameters through a deep computational graph. Contrary to the feedforward network the RNN does include cycles in the computational graph to facilitate the influence of previous values x^{t-1} of a time step x^t to the current value.

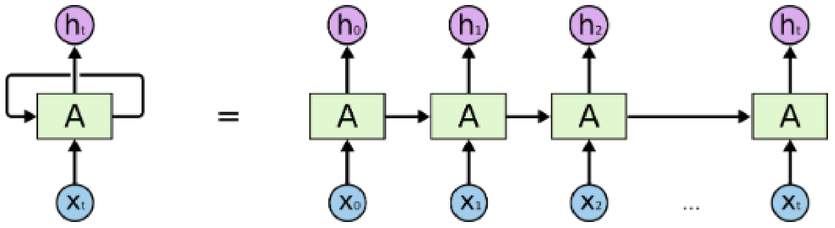


Figure 3.4: A fold and unrolled RNN

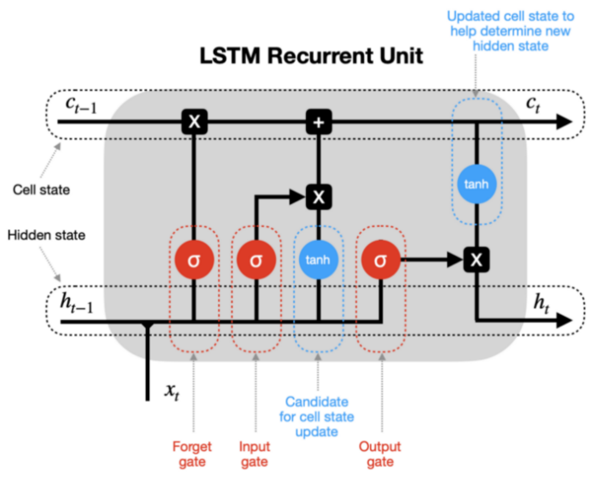
The forward pass in RNN is quite similar to the feedforward network except for the shared parameters in the computation. The forward computation of an RNN is seen in algorithm 7.

Algorithm 7: Forward propagation RNN

```
**Require:** Input $x = (x_1..x_T)$
  Initialize $h^0$
  **for** $t = 1..T$ **do**
    $a^t = b + Wh^{(t-1)} + Ux^t$
    $h^t = tanh(a^t)$
    $o^t = c + Vh^t$
    $\hat{y}_{(t)} = softmax(o^t)$
```

In algorithm 7 b is the bias vector and c, U, V, W is the weight matrices for input hidden, output hidden and hidden to hidden. o is the output, $\hat{y}$ post processed output for normalized probabilities by the softmax activation function. The back-propagation is pretty straight forward like the feedforward network. However, with a simple RNN we can encounter challenges with the gradient computation. This is known as the vanishing gradient and exploding gradient problem. In short the vanishing gradient problem is when the network is unable to back-propagate the gradient to the input layer.[15]

Long short-term memory (LSTM) is a type of RNN which handles the vanishing gradient problem. The gradient problem is solved by adding a cell unit to the framework. This creates a 'carrier convoy' where relevant information can jump on and off during training and keep the information through the layers to the output.[15] This also makes the LSTM more complex than a regular RNN and thereby more parameter heavy. The LSTM has a particular unit called the memory block in the recurrent hidden layer. These blocks have an input gate, a forget gate, and output gate.[44] The input gate controls the flow of input to the memory cell. The forget gate adjusts the internal state of the cell before adding it as input to the cell, and the output gate controls the flow of the output to the rest of the network.[44] Like most other neural network architectures the LSTM consist of an input layer, a transformation layer, and an output layer. The transformation layer in the LSTM's case is a recurrent LSTM layer.[44] In figure 3.5. a recurrent LSTM unit is seen. [10]



Difference between LSTM unit and LSTM Cell

Figure 3.5: An recurrent LSTM unit

In the top of the LSTM unit, figure 3.5., the 'carrier convoy' c_t is seen. In the bottom you see the hidden state. In algorithm 8 the computation from input to output of an LSTM is seen where: W is the weight matrices e.g. W_{ix} is the weight matrix from the input gate to the input. b Is the bias vector. σ is the logistic sigmoid function. i, f, o, c are the respective gates: input, forget, output and cell activation. m is the cell output activation vector and g, h is the cell input and output activation. Forward propagation of an LSTM cell can is visualized in figure 3.6.[44]

The back-propagation computation in an LSTM network is slightly different from feedforward and regular RNNs. The LSTMs utilize regularization to handle the vanishing gradient problem. We will not go into details here, but just note that the computation is different.[15]

[10]https://towardsdatascience.com/lstm-recurrent-neural-networks-how-to-teach-a-network-to-remember-the-past-55e54c2ff22e

Algorithm 8: LSTM mapping from input to output

```
Require: Input $x = (x_1..x_T)$
  for $t = 1..T$ do
    $i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i)$
    $f_t = \sigma(W_{fx}x_t + W_{mf}m_{t-1} + W_{cf}c_{t-1} + b_f)$
    $c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c)$
    $o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_{t-1} + b_o)$
    $m_t = o_t \odot h(c_t)$
    $y_t = W_{ym}m_t + b_y$
```

alternative standard architecture from 44

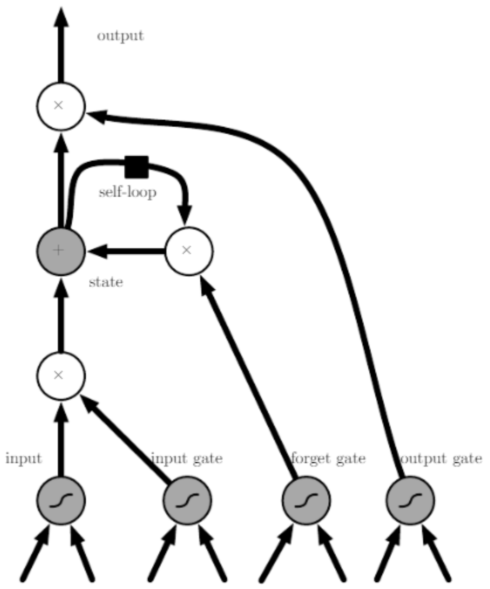


Figure 3.6: LSTM cell [15]

3.4 State space models

State space models (SSM) origins from control theory. They are concerned with how to factor in measurement errors when making estimates and inject prior knowledge into these estimates. SSMs can be used for filtering, forecasting, and smoothing.[23] In our case, we are interested in forecasting, also called predicting. SSMs uses state variables to describe a system by a set of first-order differential equations. The state variables can be reconstructed from other observations, but are not measured e.g. our collected data. We have a sequence of our observed states $u_{1:T} = (u_1, .., u_T)$ and a corresponding sequence of latent states $x_{1:T} = (x_1, .., x_T)$. SSMs exists in a linear and non-linear form. In the linear form we have the state model equation 3.22. and the output model equation 3.23.

$$x_t = Ax_{t-1} + Bu_t \tag{3.22}$$

$$y_t = Cx_t + Du_t \tag{3.23}$$

In equation 3.22. x_{t-1} is the hidden state vector, containing all the variables of the state from a previous time step. x_t is current hidden state vector and it tells us how the state changed. The change is given by A a linear combination of the previous state, and B a linear combination of external inputs and u_t is the control input. In that way we can focus on each state variable and how these variables relate to each other. This gives us a system of linear equations, and we can turn our equation 3.22. into matrices. A describes the underlying dynamics of the system, and the B matrix describes how the inputs enters into the system and which states are they affecting. Equation 3.23. tells us how the hidden state is related to our observation vector y_t. y_t is the vector of the output of the system. The C matrix describes how the states are combined to get the y_t. The D allows inputs to bypass the system and feedforward to the output. Often the last part Du_t is left out. Our hidden state vectors x_t fulfill the Markov property from section 3.2.

The Kalman filter is a widely deployed linear SMM. It has been applied in diverse areas of signal processing for engineering purposes, for robotics and vision, real-time industrial control systems and time series analysis. [3][34] It has a recursive property like other SSMs where it inference for the state.[34] The filter has proven useful for multiple measurements of different variables of the same observation simultaneously, e.g., a state, s_t, with four parameters. In equation 3.24 and 3.25 the state model and the output model for the Kalman filter is seen.[2]

$$x_t = Fx_{t-1} + Bu_t + wt. \tag{3.24}$$

$$y_t = Ax_t + v_t. \tag{3.25}$$

Many versions of the Kalman filter exists, but the fundamental equation for updating the Kalman filter with new information is:

$$\hat{x}_t = K_t y_t + (1 - K_t)\hat{x}_{t-1}. \tag{3.26}$$

where:
y_t is our observation.
K_t is the balance between old and new information.
$\hat{x}_{t-1}$ is our old information.

Maybe would have reduced the size of the section

When the Kalman filter is applied to a process we have known and estimated values for time step 0, then we iterative predict and update our values in two phases prediction and filtering.

Prediction:

$$\hat{x}_t = F\hat{x}_{t-1} + Bu_t. \tag{3.27}$$

where:
$\hat{x}_t$ is our estimated state.
F The forces of the state transition.
B, u_t The impact of external forces on state.

$$P_t = FP_{t-1} + F^TQ. \tag{3.28}$$

where:
P_t is the estimate of the covariance.
F The forces of the state transition.
Q Estimates covariance for state stochasticity.

Filtering:

$$\hat{x}_t = \hat{x}_{t-1} + K_t(y_t - A\hat{x}_{t-1}). \tag{3.29}$$

where:
$\hat{x}_t$ is our estimated state.
K_t is the Kalman gain.
A The dependency between state and measurement.

$$P_t = (I - K_tA)P_{t-1}. \tag{3.30}$$

where:
I is the low error.

The Kalman gain:

$$K_t = P_{t-1}A^T(AP_{t-1}A^T + R)^{-1}. \tag{3.31}$$

where:
R is the variance of measurement error.[2]

To introduce non-linearity into our state space model we add an activation function like in section 3.3.2. We adapt equation 3.22. and 3.23. and observe the change.

$$x_t = \sigma(Ax_{t-1} + Bu_t) \tag{3.32}$$

$$y_t = Cx_t \tag{3.33}$$

Not nesserecyly with outer activation function for LSTM due to all the internal

If we revisit the forward propagation of RNN in section 3.3.2. algorithm 7 and take line 3 and 4 computation of our hidden layer and line 5 computation of the output, rename o^t to y^t, and remove the bias for similarity. We can see in equation 3.34. and 3.35. that an RNN is a non-linear state space model.

$$h^t = \sigma(Wh^{t-1} + Ux^t). \tag{3.34}$$

$$y^t = Vh^t. \tag{3.35}$$

The RNN family falls under non-linear state space models, and for the thesis we want to learn our transition dynamics in the state space instead of the sample space. We expect the hidden state to solve for the dynamics of a sequence and model the transition from state to state. The final output is a state which contains all the information we need about the dynamics, and uses a linear layer to map our state to an action.[11][54]

3.5 Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (neural ODEs) has proven to be useful for time series modeling and supervised learning. Neural ODEs are memory efficient, adaptive in computation, great for normalizing of flows and can be used for continuous time-series. Further more they should be better than RNNs to fit and extrapolate time series data. The basic idea behind a ODE net is to have a shared layer to parameterize each time step and add the input to the output of the layer as a "skip-connection" similar to ResNet.[12] To learn a dynamic system we need an ODE, a neural network that models our dynamics and an optimization process.[10] The ODE is just a simple neural network see subsection 3.3.2, the dynamics model is a state space model see section 3.4. and the optimization is already available from back-propagation section 3.3.2.

It describes evolution in time of some process that depends on one variable.

we can regulate the depth of the neural network, just with choosing the discretizing scheme, hence, making the solution (aka neural network) more or less accurate, even making it infinite-layer like

(discretizing: making discrete)

3.6 Deep Kalman Filter

In this section, we want to create our Deep Kalman Filter. As we already know from literature several examples for deep state space models exists. Kalman filters with deep learning has e.g. been used for normalizing multivariate time series and noise reduction of images.[12][25] We want to create a Deep Kalman Filter that can make reliable fitting and extrapolation for time series predictions for offline reinforcement learning purposes. We can use the theory of state space models for dynamics modelling and deep neural networks to parameterize the mapping of covariance of the Kalman Filter[40], and finally theory from BC and supervised learning to map state to action. Our states collected with the MDP serves as noised observations as a part of the POMDP of the system. The underlying state is filtered through the kalman filter equation.

$$state = (\delta w \bullet t) + (\delta Q \bullet e) \tag{3.36}$$

[11] //www.youtube.com/watch?v=06uB13C5pxwlist=PL$_iWQOsE6TfVmKkQHucjPAoRtIJYt8a5Aindex$

[12] https://en.wikipedia.org/wiki/Residual$_neural_network$

The variables for our deep Kalman filter is defined as following:
$w_k, Q_k \sim N(0, Q)$.

where w_k is the process noise and Q_k the covariance.

$\delta w_k, \delta Q_k$ is the difference between current and previous time step.

e is a learnable parameter.

t is our current observation.

s_{t+1} is our predicted underlying state.[13]

Recursive Bayesian estimation, also known as a Bayes filter, is a general probabilistic approach for estimating an unknown probability density function (PDF) recursively over time using incoming measurements and a mathematical process model

Algorithm 9: Deep Kalman filter

```
1:  Requires trajectory $T$
2:  Initialize parameters $\theta$ for w, Q, $\mu$, b and linear layer $L$
3:  Construct model(input, hidden, output)
4:  **for** $t \in T$ **do**:
5:      gather $wQ = w, Q$
6:      $\delta w, \delta Q = model(wQ)$
7:      $w = w + \delta w, Q = Q + \delta Q$
8:      $s_{t+1} = (w \bullet t) + (Q \bullet e)$
9:  **end for** $out = s_{t+1}[-1]$
10: Return $out = L(out)$
```

If we relate algorithm 9 to section 3.5. the model is the ODE and the Kalman filter itself is the modelling of the system dynamics as we have seen in section 3.4. In algorithm 10 our simpel ODE is seen. The theory of the neural network is found in subsection 3.3.2.

Algorithm 10: Parameter network for Kalman filter

```
**Require:** Input size $i$, hidden nodes $n$, output size $o$
  $l1 = Linear(i, n)$
  $l2 = Linear(n, o)$
  **for** x **do**
      $x = l1(x)$
      $out = l2(x)$
      Return parameter update
```

[13] https://imada.sdu.dk/ kandemir/rl-lecture10.pdf

3.7 Evaluation Metrics

This section provides the theory for how we evaluate the performance of the networks.

3.7.1 Expected calibration Error

Expected calibration Error, ECE, estimates how sure our models are in their predictions. This is done by comparing the neural network's output softmax probabilities to the model's accuracies. ECE takes a weighted average over the absolute difference between accuracy and confidence.[38]

$$ECE = \sum_{m=1}^{M} \frac{|B_m|}{n} |acc(B_m - conf(B_m)| \tag{3.37}$$

14

We want the model's ECE score to be as close to 0 as possible. The ECE is applied to our test data to see how well the models are calibrated before we deploy the models to the online environments.

3.7.2 Cumulative reward

The objective in RL is to maximize the cumulative reward.[49][33][30] Therefore, it is an essential performance indicator of the models in question. During the final test in the online environment, the reward will be accumulated for each time step $_t$.

[14]https://towardsdatascience.com/neural-network-calibration-using-pytorch-c44b7221a61

Chapter 4

Experiments

In this chapter we will cover the setup of the experiments and perform the experimental pipeline, see figure 3.1. and finally evaluate the results.

4.1 Experimental setup

4.1.1 Environments.

We work with following environments for control problems from Open AI gym: CartPole and Acrobot. Both environments have discrete action space and continuous state space.[9]

CartPole v0 operates in a 4 dimensional state space and a 2 dimensional action space $S \in \mathbb{R}^4, A \in \mathbb{R}^2$. In the CartPole environment a pole attached to an un-actuated joint at the cart moves frictionlessly on the track. The system is controlled through a force of +1 and -1 to the cart. The pole starts upright, and the goal is to prevent it from falling over. For every step the pole is upright, a reward of +1 is given. The episode ends when the pole is 15 degrees from vertical, or the cart moves more than 2.4 units from the center. In figure 4.2 an image of the CartPole environment is seen.[7]

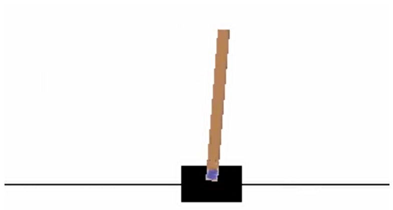


Figure 4.1: CartPole [7]

The data collected by the target policy is four tuple (State, Action, Next state, Reward).

$s \in S$ a state in State space in Cartpole is four-dimensional ndarray, which describes the angels of the pole, the position of the cart, and its velocities.

$a \in A$ action space is a ndarray too. It outputs discrete labels 1 or 0. 0 is a push to the left by a fixed force, and 1 is a push to the right by a fixed force.

R reward the system rewards the agent with +1 for every time step the pole is held upright. If the pole is not upright, -1 is given to the agent.[7]

AcroBot v1 operates in a 6 dimensional state space and a 3 dimensional action space $S \in \mathbb{R}^6, A \in \mathbb{R}^3$. AcroBot consists of two links connected by a joint in a chain-like form. One end is fixed to a joint that can move. In the initial state the "chain" is hanging straight down, and the goal is to reach to the black line above seen in figure 4.3.[8]

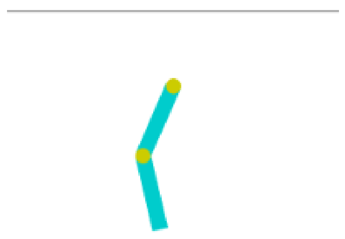


Figure 4.2: AcroBot [8]

The data collected by the target policy is four tuple (State, Action, Next state, Reward).

$s \in S$ a state in State space in AcroBot is six-dimensional ndarray, which describes the rotational joints angels and velocities.

$a \in A$ action space is a ndarray too. It outputs discrete labels 0, 1 or 2. It describes the torque applied on the joint between the two links. 0 is -1 torque, 1 is 0 torque and 2 is 1 torque.

R the system rewards the agent with -1 for every time step the height has not been reached. When the height is reached the agent is rewarded with 0.[8]

4.1.2 Policy collection.

Q-learning in algorithm 1 is used to solve for a target policy. To approximate our Q-values, the feedforward network in algorithm 12 subsection 4.1.3 is used. To find the suitable hyperparameters for solving a policy for the environments, brute force is applied. When a target policy is found it is collected by algorithm 11. CartPole is considered solved when the latest 100 consecutive time steps has an accumulated average reward of 195. AcroBot is considered solved when 1/5 actions reach the black line. When a policy is ready a four tuple of (s_t, a_t, r, s_{t+1}) per time step $_t$ is collected.
In table 4.1. an overview of the collected policies for training are provided.

Algorithm 11: Policy collection

```
Load trained policy $\pi$, set number of episodes to collect $e_n$.
Initialize map for collection $\mu$, Environment $y$
for $timestep \in episode$ do:
    $y \rightarrow s_t$
    while not done do
        $a_t \leftarrow \pi(s_t)$
        $s_{t+1}, r, done \leftarrow y(a_t)$
        $s_t = s_{t+1}$
        if done = True then
            $\mu = e_n : [timestep] : (s_t, a_t, r, s_{t+1})$
```

Episodes contain the total number of episodes in the policy. Max length is the maximum number of time steps an episode can contain, and cumulative reward is the total return over all episodes given the policy.

Environment	Training Episodes	Max Length	Cumulative reward	Avg. per episode
CartPole	800	200	155462	194.32
AcroBot	800	500	-138306	-172.88

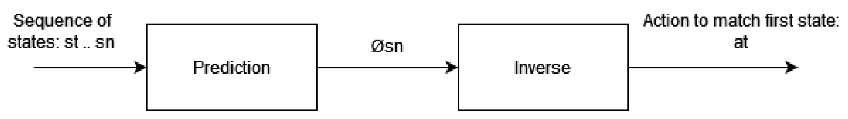
Table 4.1: Policy training data

Under same policy

An additional data set was collected for testing of the models to ensure no data leak had occurred during training. The size of the test data set is 400 episodes.

4.1.3 Models

This subsection gives an overview of the included models in the experiment, and the tuning and training of them. We want to make a script that can be used for multiple environments, so for tuning and training the Adam optimizer and cross entropy which was presented in section 3.3.2 is used for all the models. The overall goal for our models is to predict our action for current state to transition into a good next state. As seen in figure 4.3 we predict the next state through a sequence of states. We expect that the parameterized next state $\O sn$ contains all the information we need to know to map to the right action for our current state.



Inverse is wrong for FFN

Figure 4.3: High-level model

The prediction module in figure 4.3. is different for each model as seen in the following algorithms of this section. The inverse module is the same for all the models, a linear layer which maps our parameterized state to the action needed for the current state.

In algorithm 12 feedforward network, the network used in Deep Q-learning for policy collection and behavioral cloning is seen. The feedforward network (FF) is included as a baseline model. It provides an indication of how complex time series dynamics are. The architecture and hyperparameters have been chosen based on the models which solved the environments for a target policy.

Algorithm 12: Feedforward Network

```
Require: Input size $i$, hidden nodes $n$, output size $o$
1: $l1 = Linear(i, n)$
2: $l2 = Linear(n, o)$
3: for batch do
4:     $x = ReLu(l1(x))$
5:     $out = l2(x)$
6:     The predicted action is returned.
```

The predictive element in the FF network uses the theory of BC and supervised learning from section 3.3. Algorithm 13 contains the LSTM network. The algorithm also serve as a baseline model. The predictive module in this model is the LSTM layer which serves as an encoder and a non-linear state space model to include the underlying state of the dynamics to model for the right prediction. We include the theory from section 3.3 and 3.4 for the predictive module. The Deep Kalman Filter from algorithm 9 and 10 in section 3.6 uses theory from section 3.3, 3.4 and 3.5 to create the predictive module. This solves an ODE and filters for the latent state of the systems dynamics.

True and mini batching

I'm not sure solves ODE is the right expression. We just parameterize and take a step in time.

Algorithm 13: LSTM Network

```
Require: Input size $i$, hidden nodes $n$, output size $o$, layer dimension $dim$
    $Encoder = LSTM(i, n, dim)$
    $l2 = Linear(n, o)$
    $hidden_a = (dim, n)$
    $hidden_b = (dim, n)$
    $hidden = (hidden_a, hidden_b)$
    for sequence length do
        Initialize hidden()
        $out, self.hidden = Encoder(x, hidden)$
        $out = out[-1, ]$
        $out = l2(out)$
        Predicted action is returned.
```

Hyperparameter search and training

The hyperparameter search and training of models for the CartPole - and the Acrobot environment is done in two steps. First the pipeline for the CartPole environment is executed, and then the same pipeline is used for the AcroBot environment. The values needed for the search and training of the models for AcroBot environment start with the values found in the CartPole environment.

For the FF networks the same hyperparameters are used: Learning rate 0.001, batch size 32, hidden nodes 64, 2 layers, and ReLu as activation function. These hyperparameters are chosen because they solved the policies for both environments. In figure 4.3, the training of the FF baseline model for the CartPole environment is seen.

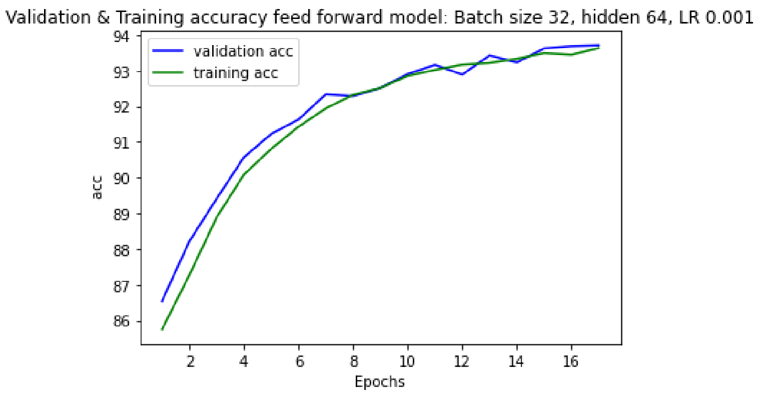


Figure 4.4: Training accuracy for CartPole feedforward network

The LSTM and the DKF are both used as state space models; therefore the same approach for hyperparameter tuning is used. The relevant hyperparameters for the models are identified: learning rate, hidden nodes, and sequence length. The hyperparameters are tested continuously in different combinations within five epochs. The model with the best validation accuracy and lowest loss is selected for further training. Algorithm 14 shows the process for hyperparameter tuning.

Algorithm 14: Hyperparameter tuning

```
Require: number of epoch $e$ and data
1: Initialize lists of learning rate ($\alpha$), hidden nodes ($\theta$) and sequence length ($l$)
2: for $i \in \alpha$ do:
3:     for $n \in \theta$ do:
4:         for $j \in l$ do:
5:             model($\theta, l$), optimizer($\alpha$)
6:         end for
7:         Performance matrices = run(e, model, optimizer, data)
8:     end for scoring = Map matrices: performance matrices
9: end for return sorted(max(scoring))
```

In table 4.2, the LSTM's three best models based on validation accuracy over five epochs is seen. The hyperparameter values should be read as learning rate, hidden nodes, and sequence length in the table.

Model	Validation accuracy	Validation Loss
0.001/256/4	0.9476	0.2022
0.001/256/16	0.9474	0.2006
0.001/16/4	0.9473	0.2001

Table 4.2: Hyperparametertuning of LSTM

The model with a learning rate of 0.001, 16 hidden nodes, and a sequence length of 4 is selected. The top 3 models have a similar accuracy score, so the model with the lowest loss is chosen.

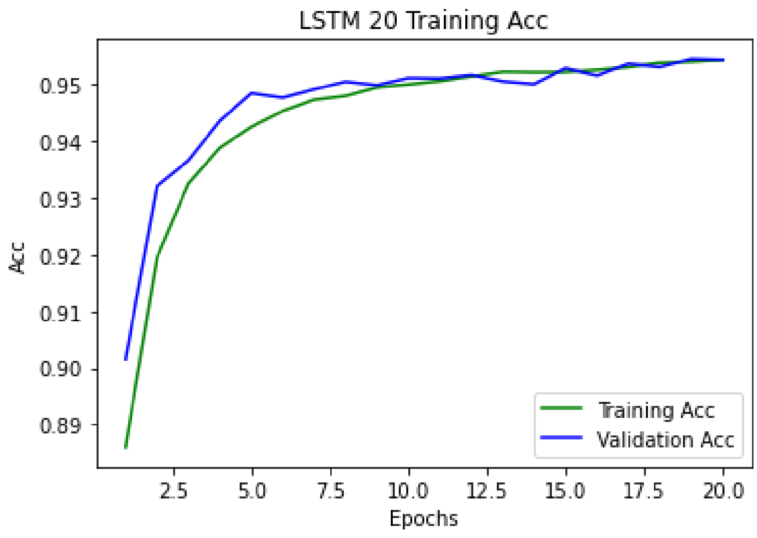


Figure 4.5: Accuracy for the CartPole LSTM model

In figure 4.5 the training of the CartPole LSTM model is seen. For the AcroBot LSTM environment model the same hyperparameters are used since they showed the same performance. In table 4.3 the DKFs three best models based on validation accuracy over five epochs are seen. The hyperparameter values should be read as learning rate, hidden nodes, and sequence length.
In the search for the hyperparameters the DKF model with the best validation

Model	Validation accuracy	Validation Loss
0.001/64/16	0.9197	0.3308
0.001/16/32	0.9186	0.3325
0.001/256/32	0.9173	0.3335

Table 4.3: Hyperparameter tuning of DKF

accuracy had following settings, learning rate: 0.001, hidden nodes: 64, sequence length: 16. But during training of the DKF it seemed like the model was already fitted after one to three epochs. Therefore further investigation by brute force was done. Figure 4.6 and 4.7 shows that the DKF with more hidden nodes have a higher validation accuracy then the DKF with fewer epochs. Out of curiosity a similar LSTM model with four hidden nodes was trained over four epochs.

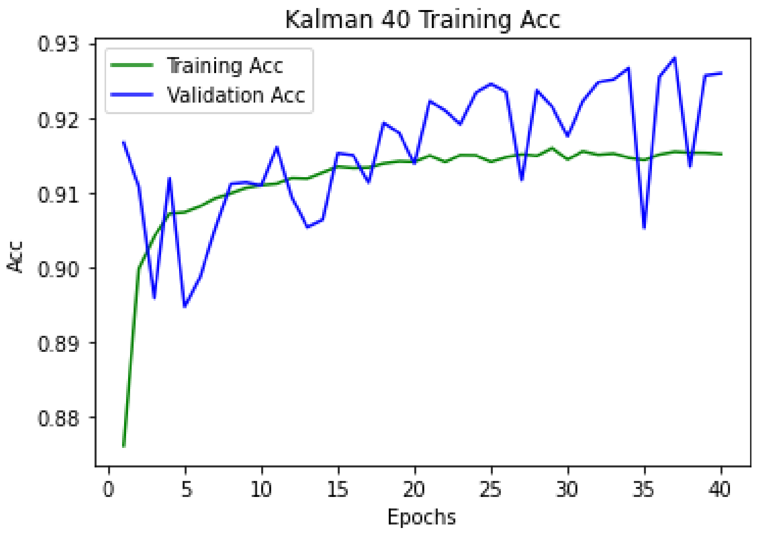


Figure 4.6: Accuracy for CartPole DKF with 64 hidden nodes over 40 epochs

For the DKF AcroBot environment model the same approach as the LSTM models is tried. However, the same hyperparameters can not be reused like for the LSTM models. Brute force is used to find a DKF model for the AcroBot environment. It is found difficult to push the validation accuracy above 88 percentage. In figure 4.8 the DKF with a 2 layer ODE and the highest validation accuracy is seen. The hyperparameters for this model is 10 hidden nodes and a sequence length of 32 observations. A 3 and 4 layered ODE with same hyperparameters

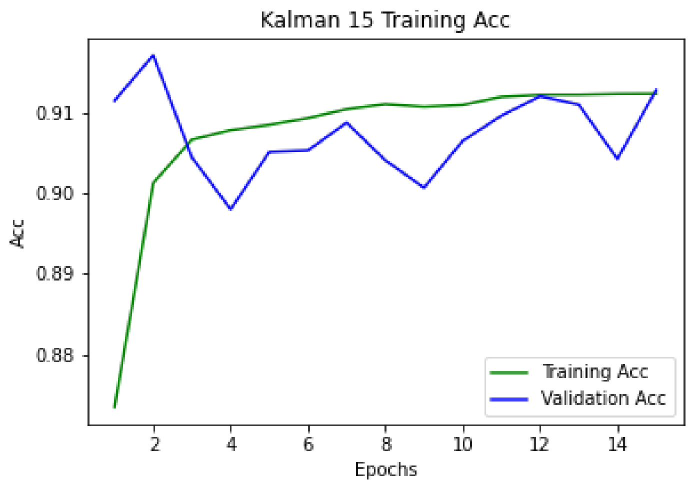


Figure 4.7: Accuracy for CartPole DKF with 4 hidden nodes over 15 epochs

was tested too. The DKF with a 3 layered ODE improved the validation accuracy to 90 percentage. Figure 4.8. displays the training of the DKF with the 3 ODE layers.

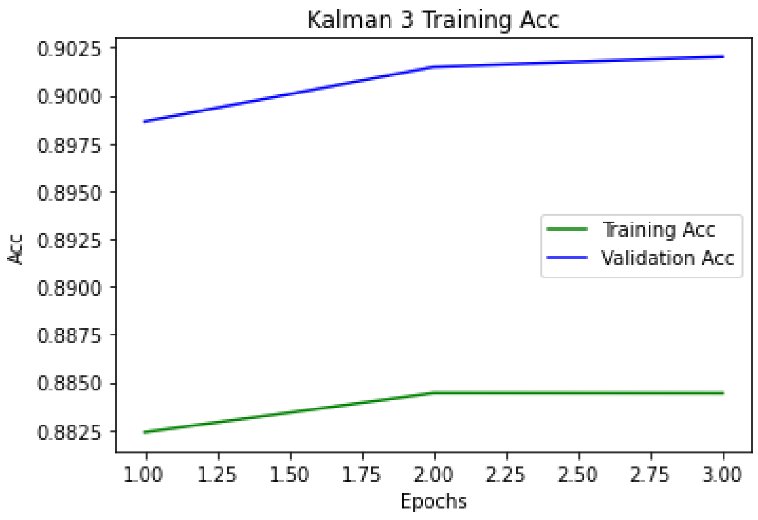


Figure 4.8: Accuracy for DKF AcroBot model with 3 layers, 10 hidden nodes and sequence length 32

Should have included graphs for the others in the repport

4.2 Results

In this section, we present and discuss the results of the feedforward Network (FF), LSTM models, and Deep Kalman Filters (DKF). The accuracy and the ECE are collected over 10 trials on a dedicated test set. The Cumulative Reward is collected online in the environments over 1000 episodes. In the CartPole environment we want to get as close as possible to a cumulative reward of 199999 over the 1000 episodes. For the AcroBot environment it is -99000 over the 1000 episodes.

CartPole				
Model	Accuracy	ECE	Cum. Reward	Each episode
FF	0.9516 ± 00	0.2640 ± 00	196460	196.46 ±0.135
LSTM	0.9392 ± 00	0.3521 ± 00	110530	110.53 ±1.32
DKF	0.9155 ± 00	0.2502 ± 00	196600	196.60 ±0.136
AcroBot				
FF	0.9441 ± 00	0.3970 ± 00	-164874	-164.87 ±1.092
LSTM	0.7034 ± 00	0.1959 ± 00	-191308	-191.30 ±1.149
DKF	0.8585 ± 00	0.3533 ± 00	-160444	-160.4 ±1.144

Maybe more data would have been good.

Table 4.4: Results: Accuracy, ECE and Cumulative reward with mean and standard error.

Given table 4.4, we see that the DKF provides a lower accuracy - and higher ECE score than the FF models and the LSTM models for both environments. During the experiment we saw that all models had a consistent performance in calibration and no standard error in accuracy. However, the ranking based on the accuracy and ECE did not transfer to the online environment. When testing the models online in the CartPole and the AcroBot environment, the DKF yielded a higher cumulative reward compared to the two other models. The standard error between the total reward per episode does differ in both environments. In the CartPole environment the DKF and the FF network have the lowest standard error in reward per episode. In the AcroBot environment the DKF places second in the standard error of the episodes.

The feedforward network did yield interesting results. We know that a simple feedforward network could approximate well in non-linear environments, but we did not expect it to consistently outperform the LSTM models, and be in close competition with the DKF in the online environments. The FF network had no standard error over the ten trials for the ECE score and the accuracy. When the model was deployed to the online environments the FF almost delivered the highest cumulative rewards. The LSTM model delivered steady results in the accuracy and the ECE score for both environments, but when deployed the models did not deliver the expected performance online. The DKF did provide conservative estimations for the accuracy compared to the two other models. It also delivered the lowest ECE and the highest cumulative reward when deployed online.

4.2.1 Discussion

The training of the FF - and LSTM models went smoothly. The training of the DFK needed more attention and brute force to find the best models. The more extensive networks trained quickly so additional experiments with fewer parameters of the 'optimal' models were conducted. When we changed the size of the LSTM and DKF to fewer parameters they both yielded a higher ECE score and a lower accuracy score. However, they did perform better in the online environments based on the cumulative reward. This observation could be explained by the distribution shifts between the collected data and online feedback. Also it could seem like that the models had already overfitted on the training data. Therefore a form of early stopping was used to prevent overfitting the models to the underlying dynamics of the training sets. It is of course always a trade-off between bias and variance of the models. The LSTM and DKF performed better with fewer hidden nodes and epochs in both environments. As the environment became more complex, more hidden nodes were added to the models, but few epochs were enough for training. With too much training, the models got too dependent on the underlying assumptions of the training distribution. Therefore hyperparameter tuning had to be done carefully. Other regularization techniques was not included in the experiment due to time restrictions. In training the Adam optimizer and the Cross-Entropy loss were used for all problems. This was done to make the training script generic. Therefore we do not know if a different optimizer or loss function would have yielded a better result for some models. As the complexity of the environments increased the DKF did require longer trajectories and more hidden nodes than the LSTM. The LSTM is also more parameter heavy and that could explain why the model did not need an increase in hidden nodes.

When comparing the scores of the test set and the actual online performance in the environment, it seems like the FF network and the DKF had the best calibrated predictions. The FF network's performance can be used as an indicator of how complex the dynamics of the time series trajectories are. Due to the performance of the FF network we would suspect a low complexity of the data sets. So this may be a case of "simple problems require simple solutions." It would be interesting to include more complex data sets to see if the DKF and LSTM would be significantly better than the FF network. If so, how complex should the dynamics of the data set be for the performance of the FF network to decline.

The DKF was a bit more "conservative" in the test accuracy. The expected calibration error was lower than the other models. When implemented it yields the best cumulative reward. The Deep Kalman filter has proven that it can be used for simple non-linear environments. It would be interesting to investigate if this tendency of conservative estimates would apply in other environments and if so, what would the bound for expected improvement in the online environments be. The standard error in the online environments where a bit higher than for the FF network. In case we would use the model for bootstrapping predictions, we could encounter compounded errors. However this most be investigated further and is often a risk when using behavioral cloning for modelling.

The policies collected for the CartPole environment was near-optimal with an average of 194.32 point per episode and sub-optimal for the AcroBot environment with an average of -172.88 per episode. The FF network and the DKF managed to improve their given policies for both environments, and at the end of the day, that is what we want to archive with offline reinforcement learning improved policies. The LSTM did struggle in both experiments. This can be explained by overfitting of the models and the models may benefit from regularization techniques like dropout. The technique used for hyperparameter search was a bit naive. Other techniques may have found better values for the models.

In this thesis, we used behavioral cloning to model the dynamics of the environment. It could be interesting to explore other offline reinforcement learning models with which the DKF could be combined. The models were also tested in relatively simple environments. It would be interesting to include experiments with more complex environments to conduct further testing. Time is often an issue when conducting projects and this project may have benefited from an author with more background knowledge.

Chapter 5

Conclusion and recommendations

In this thesis, we aimed to take a small step closer to applying offline reinforcement learning in critical real world settings. We have covered a broad area of reinforcement learning preliminaries, time series prediction, uncertainty calibration, deep learning, and state space model theories and literature. We utilized deep Q-learning to collect target policies from non-linear dynamic environments in AI Open Gym. We developed a model for environment dynamics from a POMDP, which utilized the Kalman filter filtering equation and an ODE net for predicting the right action given the state.

kalman prediction equation

During the project, we have tested different versions of the selected models: feedforward networks, Long-Short Term Memory models, and Deep Kalman Filter as potentials for our environment dynamics model. The DKFs outperformed the FF networks and LSTM models in expected calibration error and cumulative rewards. The DKF delivered conservative estimates of the accuracy of the test data, but when the model was deployed to the environments, it yielded the highest cumulative reward. One could argue that a conservative estimate is preferred over an optimistic one in a critical environment. It would be interesting to set a bound for the improvement we can expect when the DKF is deployed online. The FF network and DKF outperformed the policies their behavior was cloned from. It would be interesting to test if that property is transferable to other time series data with increased complexity.

Bayes filter -> estimating an unknown probability density function (PDF) recursively over time using incoming measurements and a mathematical process model

During the training of the models, we found that the DKF easily overfit. Therefore, hyperparameter tuning should be done carefully and include some kind of regularization technique to avoid overfitting. On a positive node, the DKF requires few hidden nodes compared to the FF network and it is trained over 3-4 epochs. Unfortunately, the LSTM models in the experiment overfitted so we can not conclude whether the DKF was better than the LSTM models. But overall the Deep Kalman filter displayed an exciting potential for further research.

Bibliography

[1] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning*, pages 104–114. PMLR, 2020.

[2] Nielsen Aileen. *Practical Time Series Analysis. Predictions with statistics and Machine Learning*, volume 1. 2019.

[3] François Auger, Mickael Hilairet, Josep M Guerrero, Eric Monmasson, Teresa Orlowska-Kowalska, and Seiichiro Katsura. Industrial applications of the kalman filter: A review. *IEEE Transactions on Industrial Electronics*, 60(12):5458–5471, 2013.

[4] Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PloS one*, 12(7):e0180944, 2017.

[5] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[6] D.P. Bertsekas and J.N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564 vol.1, 1995.

[7] Greg Brockman. Open AI Gym classic control, 2016.

[8] Greg Brockman. Open AI Gym classic control, 2016.

[9] Greg Brockman. Open AI Gym classic control, 2016.

[10] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[11] Jesse Clifton and Eric Laber. Q-learning: theory and applications. *Annual Review of Statistics and Its Application*, 7:279–301, 2020.

[12] Emmanuel de Bézenac, Syama Sundar Rangapuram, Konstantinos Benidis, Michael Bohlke-Schneider, Richard Kurle, Lorenzo Stella, Hilaf Hasson, Patrick Gallinari, and Tim Januschowski. Normalizing kalman filters for multivariate time series analysis. *Advances in Neural Information Processing Systems*, 33:2995–3007, 2020.

[13] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International Conference on Machine Learning*, pages 2052–2062. PMLR, 2019.

[14] Todd Hester Gabriel Dulac-Arnold, Daniel Mankowitz. Challenges of real-world reinforcement learning. *arXiv:1904.12901 [cs.LG]*, 1, 2019.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

[17] Jonathan Ho, Jayesh Gupta, and Stefano Ermon. Model-free imitation learning with policy optimization. In *International Conference on Machine Learning*, pages 2760–2769. PMLR, 2016.

[18] Jian Huang, Junyi Chai, and Stella Cho. Deep learning in finance and banking: A literature review and classification. *Frontiers of Business Research in China*, 14(1):1–24, 2020.

[19] Lee J. Dosovitskiy A. Bellicoso D. Tsounis V. Koltun V. Hwangbo, J. and M Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[20] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data mining and knowledge discovery*, 33(4):917–963, 2019.

[21] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34, 2021.

[22] Xue-Bo Jin, Wen-Tao Gong, Jian-Lei Kong, Yu-Ting Bai, and Ting-Li Su. Pfvae: A planar flow-based variational auto-encoder prediction model for time series data. *Mathematics*, 10(4), 2022.

[23] Nikolas Kantas, Arnaud Doucet, Sumeetpal S Singh, Jan Maciejowski, and Nicolas Chopin. On particle methods for parameter estimation in state-space models. *Statistical science*, 30(3):328–351, 2015.

[24] Jong Taek Kim. Application of machine and deep learning algorithms in intelligent clinical decision support systems in healthcare. *Journal of Health & Medical Informatics*, 9(05), 2018.

[25] Rahul G Krishnan, Uri Shalit, and David Sontag. Deep kalman filters. *arXiv preprint arXiv:1511.05121*, 2015.

[26] Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate uncertainties for deep learning using calibrated regression. In *ICML*, pages 2801–2809, 2018.

[27] Aviral Kumar, Joey Hong, Anikait Singh, and Sergey Levine. When should we prefer offline reinforcement learning over behavioral cloning? *arXiv preprint arXiv:2204.05618*, 2022.

[28] Max-Heinrich Laves, Sontje Ihler, Karl-Philipp Kortmann, and Tobias Ortmaier. Uncertainty calibration error: A new metric for multi-class classification, 2021.

[29] Vincent Le Guen and Nicolas Thome. Shape and time distortion loss for training deep time series forecasting models. *Advances in neural information processing systems*, 32, 2019.

[30] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv:2005.01643v3 [cs.LG]*, 3, 2020.

[31] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.

[32] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in Neural Information Processing Systems*, 32, 2019.

[33] Yuxi Li. Deep reinforcement learning: An overview. *arXiv:1701.07274v6 [cs.LG]*, 1, 2017.

[34] Richard J Meinhold and Nozer D Singpurwalla. Understanding the kalman filter. *The American Statistician*, 37(2):123–127, 1983.

[35] Sean P. Meyn and Richard L. Tweedie. *Markov Models*, pages 23–53. Springer London, London, 1993.

[36] Kavukcuoglu K. Silver D. et al Mnih, V. Human-level control through deep reinforcement learning. *Nature 518, 529–533 (2015)*, 1, 2015.

[37] Ashvin Nair, Murtaza Dalal, Abhishek Gupta, and Sergey Levine. Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*, 2020.

[38] Jeremy Nixon, Michael W Dusenberry, Linchuan Zhang, Ghassen Jerfel, and Dustin Tran. Measuring calibration in deep learning. In *CVPR Workshops*, volume 2, 2019.

[39] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97, 1991.

[40] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[41] Paria Rashidinejad, Banghua Zhu, Cong Ma, Jiantao Jiao, and Stuart Russell. Bridging offline reinforcement learning and imitation learning: A tale of pessimism. *Advances in Neural Information Processing Systems*, 34, 2021.

[42] Nicholas Rhinehart, Rowan McAllister, and Sergey Levine. Deep imitative models for flexible inference, planning, and control. *CoRR*, abs/1810.06544, 2018.

[43] Ullrika Sahlin, Inari Helle, and Dmytro Perepolkin. "this is what we don't know": Treating epistemic uncertainty in bayesian networks for risk assessment. *Integrated environmental assessment and management*, 17(1):221–232, 2021.

[44] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[45] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[46] Matthias W Seeger, David Salinas, and Valentin Flunkert. Bayesian intermittent demand forecasting for large inventories. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[47] Heckerman David Brafman Ronen I Boutilier Craig Shani, Guy. An mdp-based recommender system. *Journal of Machine Learning Research*, 6(9), 2005.

[48] Wen Sun, Arun Venkatraman, Geoffrey J Gordon, Byron Boots, and J Andrew Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. In *International conference on machine learning*, pages 3309–3318. PMLR, 2017.

[49] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[50] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Comparing model-free and model-based algorithms for offline reinforcement learning. *arXiv preprint arXiv:2201.05433*, 2022.

[51] Matthew E. Taylor, Nicholas K. Jong, and Peter Stone. Transferring instances for model-based reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, volume 5212 of *Lecture Notes in Artificial Intelligence*, pages 488–505, September 2008.

[52] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[53] Michel Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer, 2010.

[54] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[56] Zhikang T Wang and Masahito Ueda. A convergent and efficient deep q network algorithm. *arXiv preprint arXiv:2106.15419*, 2021.

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

[58] Alan Wu, AJ Piergiovanni, and Michael S Ryoo. Model-based behavioral cloning with future image similarity learning. In *Conference on Robot Learning*, pages 1062–1077. PMLR, 2020.

[59] A. Zai and B. Brown. *Deep Reinforcement Learning in Action*. Manning Publications, 2020.

[60] Guoqiang Zhang, B Eddy Patuwo, and Michael Y Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.