Imperial College London

Imperial College London

Department of Computing

Multi-agent Deep Reinforcement Learning for Anatomical Landmark Detection

Author:
Guy Leroy

Supervisor:
Dr. Amir Alansary

MEng Joint Mathematics & Computer Science

July 6, 2020

Abstract

Deep Reinforcement Learning (DRL) has proven to achieve state-of-the-art accuracy in medical imaging analysis. DRL methods can be leveraged to automatically find anatomical landmarks in 3D scanned images. Robust and fast landmark localisation is critical in multiple medical imaging analysis applications such as biometric measurements of anatomical structures, registration of 3D volumes and extraction of 2D clinical standard planes. Here, we explore more advanced approaches involving multiple cooperating agents with a focus on their communication in order to improve performances. The increase in accuracy could lead to a general adoption in clinical settings to reduce costs and human errors. We select three datasets comprising of brain and cardiac MRI scans as well as fetal brain ultrasounds to evaluate our proposed methods. Our results show that the CommNet architecture with communicating agents on a single landmark outperforms previous approaches. We can detect the anterior commissure landmark with an average distance error of 0.75mm. Our implementations also have greater accuracy than expert clinicians on the apex and mitral valve centre.

Acknowledgements

I would like to express my sincerest thanks and gratitude to my supervisor Dr. Amir Alansary, for his guidance, support, and for proposing this project.

I would also like to thank Dr. Ronald Clark for his valuable feedback on the project as well as insights on object localisation in computer vision and its link to landmark detection in medical imaging.

Special thanks to my family, friends and partner for their incredible support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Bringing the recent advances in Deep Reinforcement Learning (DRL) to clinical applications enables to enhance the current methodologies in healthcare and develop new ones [1]. Experts would not need to manually extract insights from medical data, which can be a tedious and error-prone task. Automation in this area may shorten the time to diagnosis and avoid the unnecessary burden on patients of further investigations and invasive tests.

In addition to contributing to the medical field, applying state-of-the-art DRL to real world problems brings new benchmarks and allows to test new advances. This progress can reverberate to other varied and seemingly unrelated applications of DRL [2].

1.2 Objectives

The main aim of this project is to investigate the application of multi-agent reinforcement learning (RL) approaches for the detection of multiple anatomical landmarks.

The main focus is on approaches where each agent looks for a different landmark in the same environment or image scan [3, 4]. As a result of the communication, the cumulative knowledge should perform better than single agent models looking separately for each landmark. We will also examine approaches where multiple agents look for the same landmark, and hybrids of both.

Automatic detection of anatomical landmarks trains agents to navigate in medical images and find target points [5]. This provides a mean to compute any standard measures that are used for multiple medical applications. The work aims to improve the accuracy on a state-of-the-art benchmark on three different medical datasets. One dataset is a collection of 455 cardiac magnetic resonance imaging (MRI) scans, which comprise the location of six different landmarks. Another dataset is composed of 832 adult brain MRI scans having annotations for fifteen landmarks. The third dataset is 72 fetal head ultrasounds with thirteen points labelled on each image.

The objectives are also open ended. We extend the published work[1] on anatomical landmarks detection using a single agent to trained multi-agents in a collaborative environment. A lot of efforts are spent to refactor the publicly available base code for a single agent, to enable further flexibility to the new multi-agent environments proposed in this project. These modifications are explained in detail in Chapter 3. To ensure a valid and fair comparison, we reproduce previously published results [5, 3] on the same aforementioned datasets.

1.3 Challenges

There are inherent challenges in DRL as shown by results of performance on Atari games played by different architectures [6]. Finding an optimal network architecture for achieving the best performances depends on many factors such as the environment and target landmarks chosen [5].

Also, as is customary in ML, the more data the merrier. Even though we already have access to three datasets having a total of 1359 medical scans, this is still on the low side compared to other benchmarked datasets such as the MNIST dataset of 70k examples [7]. More data would

[1]`https://github.com/amiralansary/rl-medical`

increase the performance of the medical agents. However, due to the specific expertise and time required to label 3D medical data, it is quite hard to acquire many labelled data points.

This work also builds on legacy code which adds a steep learning curve at the beginning where it is vital to test and understand multiple files of code, the overall architecture and design choices.

1.4 Contributions

The main contributions of this thesis can be summarised in:

- Re-implementation of the single agent approach in [5] giving similar results. In doing so we have laid a strong foundation of our software by refactoring the code using the powerful Pytorch library. The code is now easily scalable, maintainable and readable.
- Integration of the multi-agent approach, collab-DQN [3]. Enhancements have also been added to the original collab-DQN implementation to support more than two agents and better detection of agents stuck in loops.
- Implementation of the CommNet architecture [4]. This architecture learns a communication channel between agents through back-propagation.
- Design and implementation of multiple agents on a single landmark. Multiple agents look for the same landmark and their final positions are averaged. This allows for hand crafted or back-propagated communication and reduces "unlucky" starting points of agents.
- Evaluation of the methods listed above. We conclude multi-agents are superior to single agents. Amongst those, the CommNet performs better than the collab-DQN approach. We achieve superhuman accuracy on two landmarks. Finally, we show that our proposed method of multiple cooperative agents on a single landmark outperforms previous approaches.

The implementation of the code is currently publicly available on Github[2].

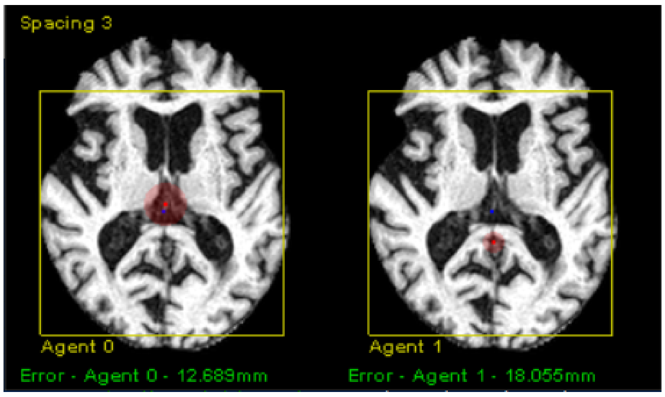


Figure 1.1: Two agents finding landmarks in an adult brain MRI scan.

[2]`https://github.com/gml16/rl-medical`

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is a sub-field of Machine Learning (ML) which lie under the bigger umbrella of Artificial Intelligence (AI). ML algorithms can be classified into different categories based on the amount and type of supervision during the model training. For example, supervised learning classifies or attributes values to data from a training set of labeled examples provided by an expert. While, unsupervised learning finds patterns within unlabeled data (two other major sub-fields of ML). RL differs from the two previous categories by training agents to correctly take actions within an environment.

RL draws inspiration from behavioral psychology and neuroscience [6], an artificial agent is trained by taking actions within an environment and receives an updated state with the associated reward similarly to an animal learning by trial and error. Policies in the environment are learnt directly from high-dimensional inputs.

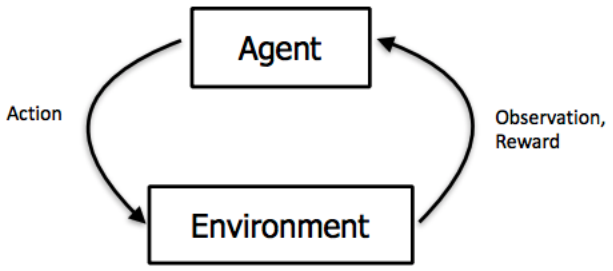


Figure 2.1: Reinforcement learning loop, the agent takes an action in the environment and receives the updated state and an associated reward [8].

RL methods are applicable in various scenarios. For example it can be used to manage an investment portfolio where actions are to buy a certain amount of a stock and the reward is the revenue generated. It can be used to play chess, the reward is to win the game and each agent is a player whose action is to move a pawn or piece as allowed per the rules. Another application is to teach a robot to solve a Rubik's cube, the reward being inversely proportional to the time taken. In this case actions are less obvious, they can be the continuous movements of the robot's hands.

2.1.1 Partially observable Markov decision process

The agent must form an understanding of its environment, but in most modern applications, the agent will not have total knowledge of its environment's state. This is referred to as a partially observable Markov decision process. Formally, it is defined as a tuple $\mathcal{S}$, $\mathcal{A}$, $\mathcal{O}$, $\mathcal{P}$, $\mathcal{R}$, $\mathcal{Z}$, γ where [9]:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.

- $\mathcal{O}$ is a finite set of observations.
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$.
- $\mathcal{R}$ is a reward function, $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$.
- $\mathcal{Z}$ is an observation function, $\mathcal{Z}^a_{s'o} = \mathbb{P}[O_{t+1} = o | S_{t+1} = s', A_t = a]$.
- $\gamma \in [0, 1]$ is a discount factor.

2.1.2 Tabular methods

Tabular methods were originally used to evaluate the expected return of actions with respect to a state. In general, the value of a state s under a policy π can be described as:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} | S_t = s], \quad \text{where } R_t \text{ is the return at step } t.$$

$$R_{t+1} = \sum_{i=0} \gamma^i r_{t+1+i}$$

We can now define a partial ordering over policies: $v_\pi(s) \geq v_{\pi'}(s)\ \forall s \in \mathcal{S} \implies \pi \geq \pi'$.

The agent decides which actions to take from its policy defined as the probability to take action a in state s at time step t: $\pi(a|s) = P[A_t = a | S_t = s]$.

The Bellman equality equation allows to compute for any state s its value $V(s)$ under an optimal policy π^*:

$$V^{\pi*}(s) = \max_a \{R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^{\pi*}(s')\}$$

Here, $R(s, a)$ defines the reward from taking action a in state s.

To find an optimal policy, we can start from an arbitrary policy, evaluate it, and improve it. This is the policy iteration algorithm which leverages dynamic programming [10]:

Algorithm 1: Policy iteration

```
Initialise $\mathbf{w} := 0$
repeat
    repeat
        $\Delta := 0$
        for each $s \in \mathcal{S}$ do
            $v := V(s)$
            $V(s) := \sum_{s'} \mathcal{P}^{\pi(s)}_{ss'} (\mathcal{R}^{\pi(s)}_{ss'} + \gamma V(s'))$
            $\Delta := max(\Delta, |v - V(s)|)$
        end
    until $\Delta$ is small;
    policy_stable := true
    for each $s \in \mathcal{S}$ do
        b := $\pi(s)$
        $\pi(s) := argmax_a \sum_{s'} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma V(s'))$
        if $b \neq \pi(s)$ then
            policy_stable := false
        end
    end
until policy_stable;
```

Model-free learning

One shortcoming of the policy iteration algorithm is it assumes the rewards and transition probability matrix are known. In modern applications, this is almost never the case. This is why we want to estimate the values of states without knowing the full model.

We present two algorithms that are model-free, Monte Carlo estimations and Temporal-difference learning. In both cases they learn V^π from traces. A trace τ is a sequence of tuple (action, reward) produced by the agent following a policy.

Algorithm 2: First Visit Monte Carlo estimation [10]

```
Initialise $V(s)$ as an arbitrary value for all $s \in \mathcal{S}$.
Initialise $Returns(s)$ is an empty list for all $s \in \mathcal{S}$.
repeat
    Get trace $\tau$ using $\pi$
    for all $s$ appearing in $\tau$ do
        $R$ := return from first appearance of $s$ in $\tau$
        Append $R$ to $Returns(s)$
        $V(s) := average(Returns(s))$
    end
until convergence;
```

Above is the First Visit Monte Carlo estimation. Another variant is the Every Visit Monte Carlo where instead of taking the return from the first appearance of s, the mean of the returns from all appearances of s are taken. To illustrate this, suppose $\gamma = 1$ and the following traces:

$$\tau_1 = (a = \text{"Forward"}, r = 4), (a = \text{"Right"}, r = 2), (a = \text{"Forward"}, r = -1), (a = \text{"Right"}, r = 4)$$

$$\tau_2 = (a = \text{"Right"}, r = 1), (a = \text{"Right"}, r = 2), (a = \text{"Forward"}, r = 0)$$

Using the FVMC estimation, we can compute $V(\text{"Forward"}) = \frac{1}{2}(9 + 0) = 4.5$. Using EVMC estimation, this gives $V(\text{"Forward"}) = \frac{1}{2}(\frac{1}{2}(9 + 3) + \frac{1}{1}(0)) = 3$.

While Monte Carlo estimations must wait for the trace to finish in a terminal state to compute the returns, TD learning can use incomplete sequences.

Algorithm 3: Temporal-difference estimation [10]

```
Initialise $V(s)$ as an arbitrary value for all $s \in \mathcal{S}$.
Choose learning rate $\alpha$
repeat
    Reset environment and observe $s$
    repeat
        Take action $a$ chosen from policy
        Observe reward $r$ and next state $s'$
        $\delta := r + \gamma V(s') - V(s)$
        $V(s) := V(s) + \alpha\delta$
        $s := s'$
    until $s$ is an absorbing state;
until convergence;
```

Differences between TD and MC estimations also arise in their bias-variance trade off. MC has high variance, zero bias while TD has low variance, some bias (since it bootstraps from previous estimates).

2.2 Deep Learning

Deep learning is also a sub-field of ML. Originally, the perceptron was invented in 1958 and performed linear classification [11].

Algorithm 4: Original perceptron algorithm

```
Initialise $\mathbf{w} := 0$
repeat
    Take example $\mathbf{x}$ and its label y (either -1 or 1) from dataset
    $\mathbf{w} := \mathbf{w} + y\mathbf{x}\mathbb{I}_{\text{sign}(\mathbf{w}^T\mathbf{x}) \neq y}$
until perceptron has converged;
```

The XOR problem showed in 1969 that the perceptron was unable to predict the "exclusive or" operation when given two binary inputs [12]. The multi layer perceptron (also called feedforward network) was created by adding hidden layers in between the input and output layers.

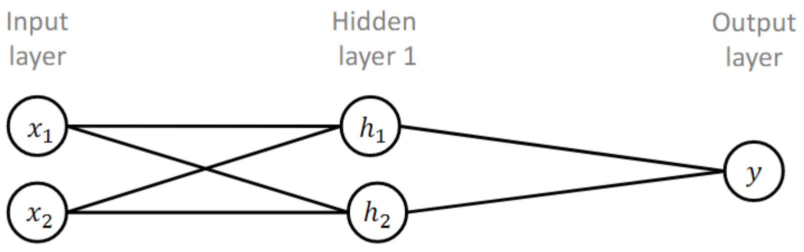


Figure 2.2: Single hidden layer feedforward network taking two inputs and outputting one scalar.

$$\mathbf{h} = g(\mathbf{w}^T\mathbf{x} + \mathbf{c}), \text{ where g is called the activation function}$$

$$\mathbf{y} = \mathbf{w}^T\mathbf{h} + \text{b}$$

We can now solve non linearly separable problems. To solve the XOR problem set the following weights and biases:

$$\mathbf{W_h} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \qquad \mathbf{W_{out}} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \qquad b = 0, \qquad \mathbf{c} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad g(x) = \max(0, x)$$

Feeding in the four possible combinations of two binary inputs we can see the output is similar to applying the XOR operation.

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \implies \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

K. Hornik showed that multi layer networks can arbitrarily accurately approximate any continuous function on a compact subset of $\mathbb{R}^n$ provided the activation function is bounded and non constant [13]. However, the width grows exponentially with dimensions, which makes single hidden layers impractical. Deeper networks reduced the layers' width and allowed to keep up with the size of the input but still required much computing power.

Rapid advances in processing powers as stated by Moore's law allowed to increase the number of hidden layers and decrease training times. Deep learning is now heavily used for many different tasks such as classification, regression, natural language processing, computer vision, denoising, density estimation, and reinforcement learning [14].

2.2.1 Optimising the network

In the XOR example, we gave weights and biases that one can check produce the expected results for the four possible combinations. However, for real world applications, not only is it hard to check whether given parameters produce accurate results due to the impossibility to observe every possible combination, but it is even harder to come up with great parameters in the first place.

One solution is to iteratively improve the parameters via gradient descent. Given a dataset, the network performs a forward pass and predicts a result for each sample. The error between the true and predicted output is then back-propagated through the network. This has been automated in recent ML frameworks such as TensorFlow and Pytorch via computational graphs, this is known as differential programming. Finally, the weights and biases are updated using a gradient descent algorithm to minimise the cost function **J**.

$$J(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} L(f(\mathbf{x};\theta), y)$$

L is the per-example loss function. $f(\mathbf{x};\theta)$ is the prediction of the network for input $\mathbf{x}$ with the parameters θ.

One method to optimise the cost function is stochastic gradient descent. There also exists extensions and variants such as AdaGrad, RMSProp and Adam [14].

Algorithm 5: Stochastic gradient descent

```
Choose weights $\mathbf{w}$
Choose learning rate $\alpha$
repeat
    Randomly shuffle $n$ examples in the training set
    for $i$ in $0,\ 1,\ ...,\ n$ do
        $\mathbf{w} := \mathbf{w} - \alpha\nabla J_i(\mathbf{w})$
    end
until an approximate minimum is obtained;
```

Weights initialisation

In the original perceptron algorithm and in the SGD algorithm the first line is about setting and choosing the weights respectively. In fact, the initial weights and biases in a network play an important role in its convergence speed. Weights can suffer from what is called vanishing or exploding gradients. When the gradient of an activation function is very small, the updates in the network are also small and the network takes a longer time to converge. Conversely, high gradients may lead to an unstable network as the updates are very large and likely overshooting the minimum of the cost function. This shows that if the mean of the activation functions are zero and the variance is constant the back-propagated gradients neither explode nor vanish over time. Another problem can arise if the weights are the same, the gradients will also be similar which cause the neurons to learn the same features symmetrically.

Recent ML frameworks allow users to choose their own weight initialisation method and often have a default mode for ease of use [15, 16]. For example, Pytorch uses by default the He initialisation [17] for convolutional layers. We present below some typical weights initialisation methods as well as some more recent ones.

- Constant initialisation. Originally, all weights would often be set to 0 or 1. As described above this can lead to poor learning.
- Random initialisation. One can randomly initialise weights following a uniform or normal distribution with mean zero. This can prevent symmetric learning as well as vanishing or exploding gradients.
- Xavier initialisation (also called Glorot) [18]. This encompasses two random initialisation methods. In the uniform case, weights follow a uniform distribution between $-\sqrt{\frac{6}{fan\_total}}$ and $\sqrt{\frac{6}{fan\_total}}$. In the normal case, weights follow a normal distribution with mean 0 and standard deviation $\sqrt{\frac{2}{fan\_total}}$. Fan_total is the total number of input and output units in the weight tensor. Xavier initialisation is the state-of-the art for networks with differentiable activation functions at zero such as Sigmoid [19].
- He initialisation [17]. This initialisation is very similar to Xavier's with a slight change in bounds. For the uniform case, the bound is set to $\sqrt{\frac{3}{fan\_single}}$. For the normal case, the mean is also zero but the standard deviation is $\sqrt{\frac{1}{fan\_single}}$. Here, fan_single is either the number of input or output units. This choice depends on whether one wants to preserve the magnitude of the variance of the weights in the forward pass (choose fan_input) or in the backward pass (choose fan_output). This initialisation is the state-of-the-art for non-differentiable activation functions at zero such as ReLU [19].

2.2.2 Capacity, underfitting, overfitting, and errors

One may think the best neural networks are the deepest ones. Not necessarily. One key objective of a machine learning model is to generalise well on new, unseen data. Underfitting is either caused from too little training, or is due to a model that has not enough capacity. On the other hand, overfitting is due to training on the same samples too many times and a large capacity. A model can learn the answer for each input in the training set if seen enough times without building a global model that generalises.

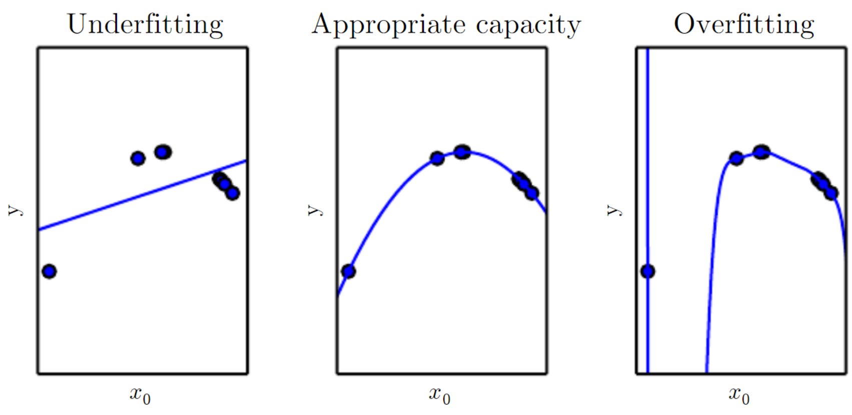


Figure 2.3: Models with different capacities fitting the data. The model on the left is a linear regression, in the middle it is a quadratic regression, on the right it is a polynomial of ninth degree. [14]

A dataset can be split into training and testing sets to measure the generalisation error (or test error), that is the expected value of the error on unseen data. The capacity correlates with the depth of the network. If it is too deep, the capacity is large and the network may overfit on the training data and not generalise well. Conversely, a shallow network has a smaller capacity and can underfit the data as it is not even capable of learning the training set.

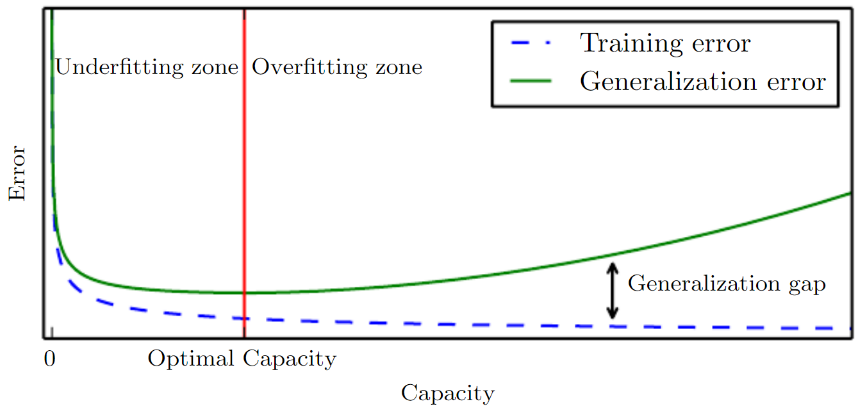


Figure 2.4: Typical relationship between capacity and error. [14]

To find the best model for the problem one must know about the different types of errors occurring in ML [20].

- Bayes error. This comes from the fact the output is most often not deterministically defined solely based on the input. For example, predicting house prices based on ZIP codes has some Bayes error since many other factors play a role.

- Approximation error. This error is due to a restrictive family of predictors chosen. This happens in the case of using a perceptron for the XOR problem, the capacity of the network is too small.

- Estimation error. This is due to having a small data set, or a data set that does not cover the whole probability spectrum of cases we may encounter.

- Optimisation error. This is when the model has not found an optimal minimiser, mostly because it has not run enough iterations. In the case of deep neural networks, this means the model is underfitting and we should let it run longer.

2.2.3 Convolutional neural networks

Convolutional neural networks were introduced in 1989 [21]. They have the benefit of reducing the computing complexity of high dimensional inputs. Let us take a 3D cube of length n, where each voxel (equivalent of a pixel in an image) is a scalar. If we were to pass each voxel as an input to a feedforward network having two hidden layers of size h that outputs a scalar, we would have $n^3h + h^2 + h$ weights. One can see the weights exponentially grow as the dimension of the input increases.

Convolutional layers leverage local connectivity. That is, each neuron from one layer is not connected to all the neurons in the next, but only to local ones. Shared weights also reduce the complexity as the kernel is the same across the whole input. This gives translational invariance properties to the network.

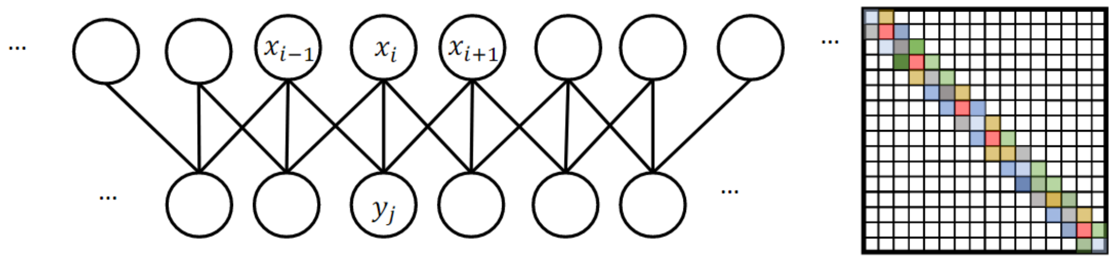


Figure 2.5: Local connectivity between two layers. We have $y_i = w_{j,i-1}x_{i-1} + w_{j,i}x_i + w_{j,i+1}x_{i+1}$.

The convolutional operation between an input x and a kernel w outputs a feature map and is defined as follows:

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In addition to convolutional layers, convolutional networks often use pooling layers. They have multiple advantages such as reducing the size of the input and contributing to permutation, shift and deformation invariances. A pooling layer of size $n \times n$ slides over the input and performs a function each time. The typical functions used are `max` and `mean`.

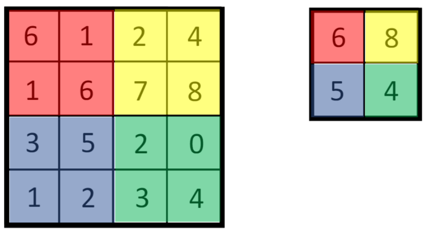


Figure 2.6: Example of a max pooling layer of size 2×2 over a 4×4 matrix [20].

For illustration purposes, we show below two examples of famous convolutional architectures: AlexNet [22] and LeNet-5 [23].

They both mix convolutional, pooling/subsampling and feedforward layers to classify images. AlexNet can classify amongst 1000 labels while LeNet-5 classifies images into 10 labels.

Table 2.1 shows that in both architectures the layers having the most parameters (and thus taking the most memory) are the dense layers. However, despite their low number of parameters, the convolutional layers are the ones with the highest FLOPS (thus increasing the training and testing time the most).

The convolutional and pooling layers presented are 2D, but one can design n-dimensional equivalents that have kernels of the corresponding dimensions used for n-dimensional inputs. 3D equivalents are popular for volumes such as MRI scans for example.

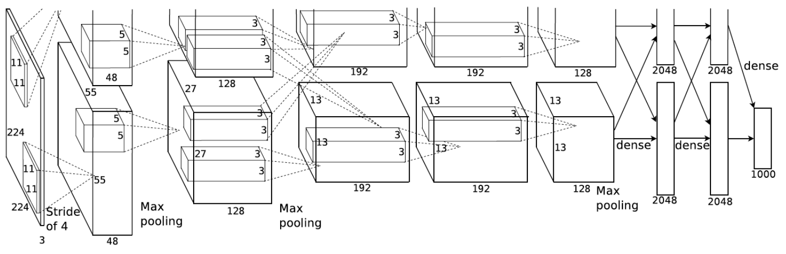


Figure 2.7: AlexNet architecture [22].

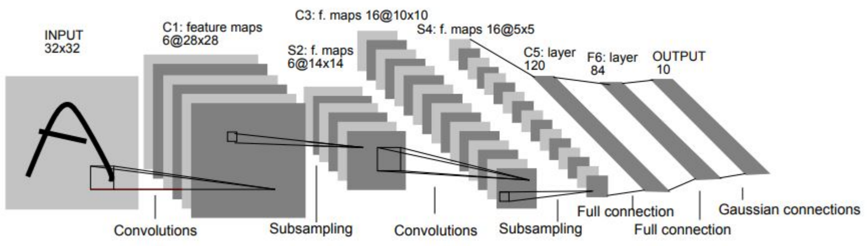


Figure 2.8: LeNet-5 architecture [23].

2.3 Deep Reinforcement Learning

As its name indicates, DRL is a mix between classical reinforcement learning and deep learning. Increasingly complex state and action spaces led to the use of deep Q-networks (DQN), hence the "deep" in deep reinforcement learning. The goal is to evaluate the expected Q-value of a state s when taking action a. It is defined as below:

$$Q^{\pi}(s,a) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}$$

γ is a discount factor that quantifies the uncertainty of rewards in the future. The reward the agent receives at transition i is denoted as r_i.

Having a network predict the reward of actions in a certain state allows to interpolate to nearby states, thus reducing the training time. Only the parameters of the network needs to be stored, instead of the state value for each state (for reference, a game of Go with board size 19×19 has about 10^{170} legal states [24]).

Algorithm 6: Deep Q-learning algorithm [10]

```
Initialise weights $\theta$ of $Q_\theta$
Initialise $\hat{Q}_{\hat{\theta}}$ with $\hat{\theta} = \theta$
Initialise empty replay buffer $\mathcal{D}$
for each episode do
    $S := S_{init}$
    for each step of episode do
        Choose A from S using policy derived from $Q_\theta$
        Take A, observe R and S'
        Store transition (S, A, R, S$'$) in $\mathcal{D}$
        Sample mini-batch $\mathcal{B} \subset \mathcal{D}$ of size $N$
        $\theta := \theta - \alpha \frac{1}{N} \sum_{(S,A,R,S') \sim \mathcal{B}} \nabla_\theta (R + \gamma \max_a \hat{Q}_{\hat{\theta}}(S', A) - Q_\theta(S, A))^2$
        Every K steps, set $\hat{\theta} := \theta$
        $S := S'$
    end
end
```

	Number of parameters		**FLOP**	
	AlexNet	**LeNet**	**AlexNet**	**LeNet**
Conv 1	35K	150	101M	1.2M
Conv 2	614K	2.4K	415M	2.4M
Conv 3-5	3M	-	445M	-
Dense 1	26M	0.48M	26M	0.48M
Dense 2	16M	0.1M	16M	0.1M
Total	46M	0.6M	1G	4M

Table 2.1: Comparison between AlexNet and LeNet architectures in terms of number of parameters and operations per seconds. [20]

Reinforcement learning often suffers from the exploration versus exploitation dilemma [25]. There is a trade off between the exploration of new states which may lead to higher rewards and exploitation of known states that have given high rewards. ϵ-greedy policy is a popular method which starts training by exploring and towards the end exploits high reward states with some noise, hoping to find even higher rewarding states nearby. This policy chooses the action with the highest Q-value with a probability ϵ and otherwise chooses any action randomly (with uniform distribution). The value of ϵ at the beginning of the training is 1 or close to 1 and decreases over time until it reaches a minimum close to zero (not exactly zero as otherwise the agent would never be able to visit new states and improve its strategy in a deterministic environment).

2.3.1 Target networks

Target networks $\hat{Q}$ were introduced to avoid having the Q-network bootstrapping from itself. At every predefined intervals, the weights θ of the Q-network would be copied to the target network. The temporal-difference error δ becomes:

$$\delta = \gamma \max_a \hat{Q}_{\hat{\theta}}(s', a) - Q_\theta(s, a)$$

2.3.2 Continuous action space

DQNs were first introduced with an input layer of the size of the space dimension to output a Q-value for each possible action.They have shown super human performances in various environments such as Atari games and the game of Go [6, 26]. A different architecture for the DQN that has improved performances uses a continuous action space [27]. The DQN has as many input neurons as the sum of the space dimension and action dimension and outputs a single Q-value. Continuous actions can be by definition much more precise than discrete actions and can thus arrive to the desired goal more quickly. For instance, in a 2D environment, actions such as move right, left, forward and backward can be replaced by one scalar representing the direction of the next step as an angle .

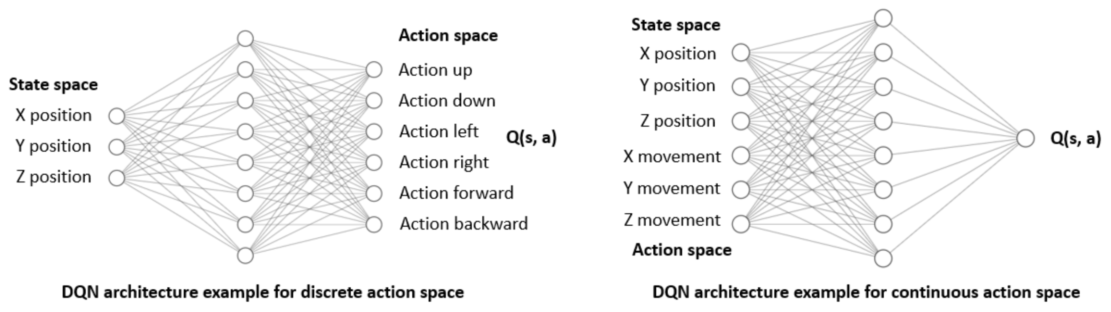


Figure 2.9: Architecture of a DQN for discrete actions and for continuous actions.

Continuous actions need to be sampled to predict the corresponding Q-value. This can be done either uniformly or using the cross-entropy method [28], which iteratively fits a Gaussian on the actions with the highest predicted Q-values. The action chosen is the mean of the Gaussian.

2.3.3 Prioritised experience replay

In online learning, each transition was trained on once. In addition to requiring many transitions, the training samples were highly correlated. An experience replay buffer solves this problem by storing past transition that are sampled uniformly to reduce correlation and to be able to train multiple times. However, this method is as likely to train on transitions that are not relevant anymore or transitions that are already very well predicted by the DQN. To mitigate this, prioritised experience replay buffers were introduced [29]. They are biased towards new and badly predicted transitions. Using a prioritised experience replay buffer, the probability of sampling transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \qquad p_i = |\delta_i| + \epsilon$$

ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. New transitions are assigned the same TD error as the highest among all transitions to incentivise their sampling. The exponent α determines the weight of prioritisation, $\alpha = 0$ corresponding to the uniform case.

2.3.4 Double Q-learning

The target Q-value needs to estimate the maximum Q-value over all actions. The problem is that the expected maximum Q-value is greater than the maximum expected Q-value. This can be referred as bootstrapping the network. To reduce the amount by which the maximum Q-value may be overestimated, double Q-learning uses the Q-network instead of the target network to reevaluate the Q-value of the action having the highest Q-value (predicted by the Q-network).

2.3.5 Dueling network architecture

A dueling network uses the hypothesis that most of the time the action taken does not matter, that Q-values are only important in key states.

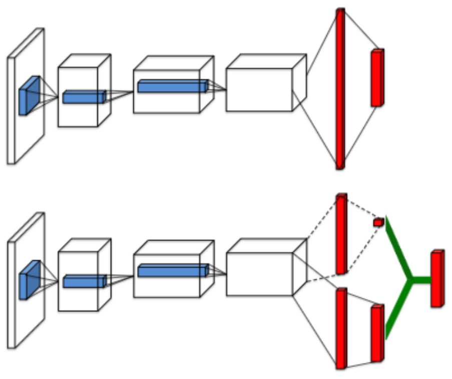


Figure 2.10: Example of a single sequence Q-network using a convolutional neural network (top) and of a dueling Q-network [30] (bottom).

The dueling network has two sequences of fully connected layers to separately estimate state-values and the advantages for each action as scalars. One sequence of fully-connected layers outputs the state-value $V(s; \theta, \beta)$, and the other sequence outputs an $|\mathcal{A}|$-dimensional vector $A(s, a; \theta, \beta)$ where θ, α and β represent the parameters of the convolutional layers common to both streams and the parameters of each fully-connected layers respectively. The Q-value is combined from the two streams using the following equation:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a, \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right)$$

2.3.6 Multiple agents

It has been showed that multiple agents cooperating on a task can yield better results and performances than independent agents [31, 32]. Such tasks can be naturally divided into multiple entities, for example in simulated robot soccer [33]. In resource management, even though resources can be managed by a central authority, distributing each one to a different agent can prove useful [34]. In these examples, agents learn independently and there is no communication.

Communication is essential to improve effectiveness between agents. Recent multi-agent methods have used handcrafted communication between agents, for example a more advanced version of robot soccer agents communicated by sending and receiving all players' position and distance to the ball [35].

CommNet model

A more recent work [36] has showed that, given a common communication channel holding a continuous communication vector, a possibly varying number of agents in the environment can learn to better communicate through back-propagation. This method allows to bypass the need to handcraft communication features. It also led to better results in many tasks compared to previous state-of-the-art approaches.

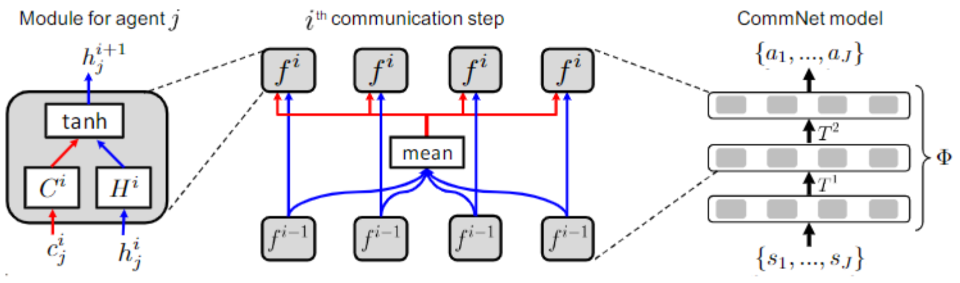


Figure 2.11: CommNet architecture [36].

In a model with J agents, let us consider the state-views $s = \{s_1, ..., s_J\}$ and a J-dimensional vector $a = \{a_1, ..., a_J\}$ denoting actions taken by all agents at time step t. The controller Φ is a mapping $a = \Phi(s)$. Φ encompasses the architectures of the controllers and communication for each agent. Multi layer neural networks f^i (with $0 < i < K$, K being the number of communication steps in the network) take as input the hidden state of each agent j h_j^i and its communication vector c_j^i. Each f^i outputs the vector input for the next communication step h_j^{i+1}.

Thus, the controller Φ computes:

$$h_j^{i+1} = f^i(h_j^i, c_j^i)$$

$$c_j^{i+1} = \frac{1}{N(J)} \sum_{j' \neq j} h_{j'}^{i+1},$$

where, $N(J)$ corresponds to the number of reachable agents. Φ takes into account the variable number of agents in the scene by scaling the communication vector relatively to $N(J)$, to avoid having over inflated communication inputs when many agents enter the episode. In a scenario where all agents can communicate to any other agent, $N(J) = J - 1$.

The CommNet model has been evaluated on four different tasks (lever pulling, traffic junction, team combat, and bAbI) against the following baseline models: independent controller, fully-connected, and discrete communication. CommNet performs significantly better on all tasks except on the bAbI one, where two methods using handcrafted communication (MemN2N and DMN+) perform best.

2.4 Medical applications

Clinical applications can leverage the accuracy and speed of reinforcement learning agents navigating through 3D medical images. Applications can range from finding standardised view planes such as the mid-sagittal and anterior-posterior commissure planes in brain MRI [37] and anatomical structures in scans acquired with a partial field-of-view [38] to finding landmarks in fetal heads ultrasounds, adult brain and cardiac MRI scans [5].

Using reinforcement learning agents over the traditional deep learning methods allows for a non-exhaustive search over the 3D volume while still improving state-of-the-art results.

2.4.1 Anatomical landmark detection

Single agent approaches

The landmark detection task can be described as a partially observable Markov decision process where the goal is to find an optimal policy for localising landmarks. Multiple deep reinforcement learning methods have been proposed to tackle this task.

Ghesu et al. first introduced a single agent navigating medical images by taking discrete steps [39]. Each state is defined as a Region of Interest (RoI) centered around the agent's 3D coordinates. They showed navigating in a localised environment is more accurate and much faster that perceiving the whole 3D environment.

Steps in the environment are stored in a replay buffer to train the DQN used to predict Q-values for each discrete action and state. The agent receives as input a 3D window around it. The input is fed first through a convolutional neural network and then through fully dense layers with six outputs at the end (one for each action).

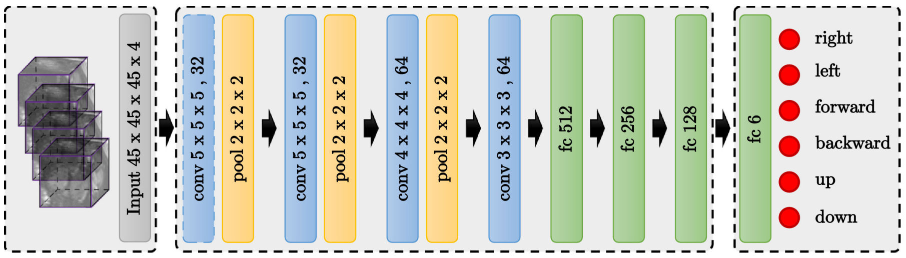


Figure 2.12: Architecture of the single agent's deep Q-network for landmark detection [5].

The reward is defined as the difference between the Euclidean distance of the previous agent's position to the target and of its current one to the target. An agent going out of the 3D environment is given a -1 penalty.

During training, the agent follows an ϵ-greedy policy. The terminal state is reached when the distance to the target landmark is less or equal than 1mm. During testing, the agent starts in the 80% inner region of the image and follows a full greedy policy. The episode ends when the agent oscillates or after 1500 steps.

They improve on the method by using a multi-scale strategy [40]. The agent starts the episode with a large window and large movement steps which reduce each time the agent oscillates between the same states (in the original implementation the step scales are 1mm, 2mm, and 3mm). This allows for a faster coarse to fine search which in general improved the accuracy. Hierarchical steps also sped up the searching process by a factor of 4–5 times.

In a later work [5], multiple DQN variants were evaluated: vanilla DQN, double DQN, duel DQN, duel double DQN. Table 2.2 shows the results of each experiment. We can see multi-scale generally outperforms fixed-scale, but it depends on the landmark. The best DQN variant is also dependent on the landmark.

It has also been shown that the multi-scale single agent is performant in incomplete volumetric data with arbitrary field of view [38]. The agent also learns to detect whether the landmark is missing from the environment suggesting a high-level anatomical understanding by the agent.

Multi-agent approach

The position of anatomical landmarks is interdependent and non-random within the human anatomy. A recent work [3] has extended the single agent version to multiple agents, as they theorise finding one landmark can help deduce the location of others. They use cooperative agents sharing convolutional layers for implicit communication in an architecture named collab-DQN.

	RC		LC		CSP	
Method	**FS**	**MS**	**FS**	**MS**	**FS**	**MS**
DQN	4.17 ± 2.32	3.37 ± 1.54	2.78 ± 2.01	3.25 ± 1.59	**4.95±3.09**	**3.66±2.11**
DDQN	3.44 ± 2.31	3.41 ± 1.54	2.85 ± 1.52	2.95 ± 1.00	5.01 ± 2.84	4.02 ± 2.20
Duel DQN	**2.37±0.86**	3.57 ± 2.23	**2.73±1.38**	**2.79±1.24**	6.29 ± 3.95	4.17 ± 2.62
Duel DDQN	3.85 ± 2.78	**3.05±1.51**	3.27 ± 1.89	3.50 ± 1.70	5.12 ± 3.15	4.02 ± 1.55

Table 2.2: Previous work's results of single agent landmark detection on the right cerebellum (RC), left cerebellum (LC), and cavum septum pellucidum (CSP) [5]. It compares fixed-scale (FC) and multi-scale (MC) approaches. Distance errors are in millimeters.

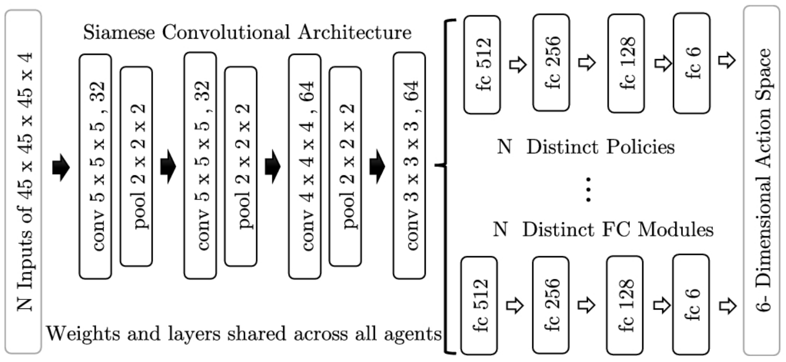


Figure 2.13: Architecture of the collab-DQN [3].

The collab-DQN approach is similar to the single agent, modeling the task as a decentralised partially observable Markov decision process with similar states, rewards and environment. The agents learn the policy using an adapted version of the Siamese architecture [41]. The convolutional layers are shared between all agents and each agent has a separate fully connected module. The shared weights of the CNN enable indirect communication of the state space between agents while each fully connected module learns specific features for each landmark. This reduces the time and memory required to train compared to independent agents.

This new architecture outperforms the accuracy of independent agents for all tested cases except on the cavum septum pellucidum (CSP) for the fetal brain as can be seen in table 2.3.

Method	**AC**	**PC**	**RC**	**LC**	**CSP**
Supervised CNN	–	–	–	–	5.47±4.23
DQN	2.46±1.44	2.05±1.14	3.37±1.54	3.25±1.59	**3.66±2.11**
Collab DQN	**0.93±0.18**	**1.05±0.25**	**2.52±2.25**	**2.41±1.52**	3.78±5.55

Table 2.3: Results of the collab-DQN on brain MRI scans and fetal brains (distance error in mm) [3].

The collab-DQN also has better memory performance, the DQN has 5% less parameters than two independent agents and 6% less in the case of three independent agents.

2.4.2 View planning

Similarly to the landmark detection single agent approach, the agent in view panning [37] navigates through a 3D medical image environment, but as a Cartesian plane defined using the equation $ax + by + cz + d = 0$. The state is the 3D RoI around the plane and there are eight discrete actions to modify the plane's parameters. The reward is either -1, 0 or 1, depending if the agent got further, stayed at the same distance or got closer to the target plane respectively. The episode ends once the agent oscillates. Similarly to the landmark detection, a multi-resolution approach is used with hierarchical action steps to increase speed and accuracy.

Training the agents took around 12–24 hours on the brain MRI data set and 2–4 days for the 4-chamber cardiac data set using an NVIDIA GTX 1080Ti GPU.

Model	**Mid-sagittal brain**		**ACPC brain**		**4-Chamber cardiac**	
	$e_d(mm)$	$e_\theta(^\circ)$	$e_d(mm)$	$e_\theta(^\circ)$	$e_d(mm)$	$e_\theta(^\circ)$
DQN	1.65±1.99	2.42±5.27	2.61±5.44	**3.23±6.03**	5.61±4.09	10.16±10.62
DDQN	2.08±2.58	3.44±7.46	**1.98±2.23**	4.48±14.00	5.79±4.58	11.20±14.86
Duel DQN	1.69±1.98	3.82±7.15	2.13±1.99	5.24±13.75	**4.84 ± 3.03**	8.86±12.42
Duel DDQN	**1.53±2.20**	**2.44±5.04**	5.30±11.19	5.25±12.64	5.07±3.33	**8.72±7.44**

Table 2.4: Results of view planning on brain and cardiac MRI scans (distance error in millimeters and degrees) [37].

Chapter 3

Single Agent Anatomical Landmark Detection

3.1 System requirements

In this project, we build on A. Alansary's work [5] presented in section 2.4.1 to find landmarks in 3D medical images as a partially observable Markov decision process. The three main features of the code are to train an agent to find landmarks in 3D volumes, evaluate the agent on labelled data and predict landmarks' coordinates on unlabelled data. The training reads from a NIfTI (`.nii.gz`) medical image and the landmarks' 3D coordinates from a text ot VTK file. The evaluation reports the distance error (in millimeters) between the predicted and true location. The testing only reports the predicted location as the true location is not known.

One first objective was to obtain similar results using the preexisting code at `https://github.com/amiralansary/rl-medical`. In addition to help understand the code logic, we could try to reproduce the results stated in the paper. The original code could also benefit from refactoring different parts of the code, in particular by upgrading its machine learning framework from Tensorpack[1] to a more recent and robust one and laying out a foundation for future extensions. It was key after this stage to rerun our previous experiments and make sure the results would at least be as good. This would be a sanity check that we did not introduce any bugs during our refactoring. We could then confidently explore more recent approaches to further improve the model. It is thus important that the software can be flexible to easily incorporate new ideas such as different architectures for the model.

3.1.1 From Tensorpack to Pytorch

The original code in Python used the machine learning framework Tensorpack. While this is a promising tool, it is not in a stable version yet and is built on top of TensorFlow 1.2. It is also meant as an abstraction layer to more easily run ML models. However, we are interested in having more control over our models to explore potential new architectures and tweak them more freely. TensorFlow's latest version is 2.1 which is more robust, includes more features and has a much more comprehensive documentation. This is also the case of other fast moving frameworks such as Pytorch.

Thus, we decided to migrate the code to a more convenient and powerful framework for our use case. We decided to go with Pytorch for the advantages listed above and also because I was more familiarised with it. In the long run, this improved the readability, maintainability and scalability of the code. In addition, rewriting in Pytorch helped me deeply understand the code. For reference, the original code to train a DQN in Tensorpack is shown in the appendix.

3.2 System implementation

The design after the Pytorch refactoring stayed close to the original design philosophy. The main file reads command line arguments, in particular the `task` to choose between training, evaluation

[1]https://github.com/tensorpack/tensorpack

and testing. It also reads the path to the training, validation and testing datasets (depending on the task chosen). It then runs the appropriate task.

Tensorpack being an abstract layer, most of the code had to be written from scratch. We were able to mostly reuse the environment and the viewer files which we cover below.

3.2.1 Environment

Perhaps to best understand the problem, understanding the environment in which the agent navigates is key. The environment is described in the `MedicalPlayer` class. It extends from the widely used OpenAI's `gym.Env` module which made it easy to understand and modify. Its main methods are `reset` and `step`.

The method `reset` instantiates a new episode where the agent is put in a random location within the middle 80% of the volume (this is to avoid spawning the agent outside or on the border of the volume which are more subject dependant and would lead to poorer results). The method `step` allows the agent to navigate within the volume by providing an action. The possible actions are to move in one of six directions (up, down, right, left, forward and backward). The step returns to the agent its current state and a reward.

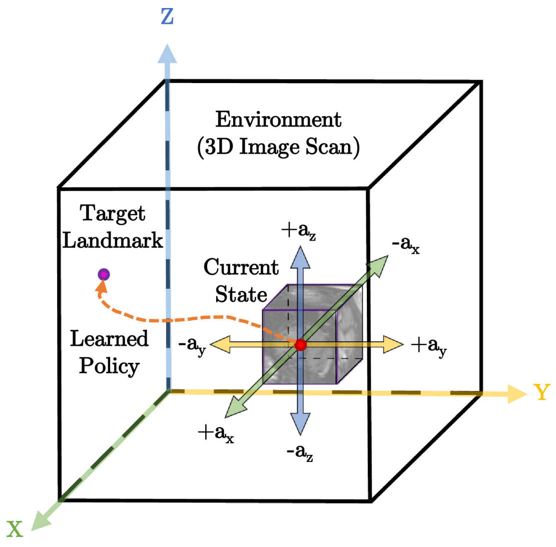


Figure 3.1: Single agent for landmark detection navigating in its environment, showing the six possible actions and its observable state.

States

An agent's Region of Interest (RoI) is a cube within the 3D image centered on its agent, of size $45 \times 45 \times 45$. A state is a frame history of $m = 4$ consecutive RoI. Whenever the agent takes a step, the new RoI replaces the oldest one. Those stacked frames allow the agents to get a sense of movement to know in which direction it was headed and improve the accuracy [6]. The algorithm is robust to different values of m, setting it to four is a nice balance between memory usage and giving enough information to the agent. A state is considered terminal if the agent is within 1 millimeter of the target landmark or if the maximum number of steps for an episode has been reached, the episode is then ended. Each voxel (point within the 3D space) in the state has one channel with value ranging from 0 to 255.

Intuitively, we may wonder why the agent is not given the whole 3D image at once. This implies using a 3D CNN in a supervised context instead of a reinforcement learning approach, as a 3D CNN would take as input the whole image and outputs the predicted landmark's position. The power of the RL approach is that it takes much less memory and computing power, and can be run on more modest GPU units, while still producing state-of-the-art results as we see later on.

Rewards

The reward is the Euclidean distance difference between the agent's previous distance to the landmark and its current one. In other words, it measures how closer to the goal the agent is. While this work does not focus on reward strategies, other types of rewards have been tried in the work of A. Vlontzos [3]. They are specific to multi-agent scenarios which we cover in the next chapter.

Oscillations

A feature handled by the environment severely speeding up training and testing is the multi hierarchical step. The agent's movements start at a fixed length of 3 units and each voxel is spaced by 3 units, meaning the observable state spans $135 \times 135 \times 135$ voxels. When the agent oscillates around a point, the step size is reduced to 2, and the observable space spans a $90 \times 90 \times 90$ cube. After a second oscillation the step size is 1 with no space between each voxel seen so the cube has size $45 \times 45 \times 45$. When the agent oscillates a final time, the episode ends. This method allows for fast movements at the beginning and a finer search near the target landmark.

Deciding when an agent oscillates is not an obvious task, especially considering policies that involve random actions such as the ϵ-greedy policy. For example, if an agent passes twice by the same location, it may not necessarily be oscillating. A random move may have made the agent take a step back once, while it may actually still be far from the landmark. Supposing an ϵ-greedy policy, an agent may leave and reenter the same voxel back and forth n times due to random moves with probability $(6\epsilon)^{-n} > 0$. This shows strict oscillation conditions may still trigger finer steps by random moves. It is also important to consider oscillations may happen as a bigger loop, that we may not spot by looking at a small state history. We can consider an oscillation memory of the past S states, and decide that if the same position happens N times within this memory, the agent's step size reduces. We have then N and S as additional hyperparameters. In our best model $S = 20$ and $N = 4$.

3.2.2 Training

When the user passes Training as mode argument we instantiate the environment and a class called Trainer which handles all the training logic.

Trainer class

The trainer class has a main method Train. It starts by filling an experience replay buffer with a random action policy to reduce the correlation of transitions sampled at the beginning. Then there is the typical DQN main loop which consists of running episodes as presented in the algorithm 9. Each episode begins by resetting the environment. The agent takes steps following an ϵ-greedy policy. The value for ϵ decreases by a small amount δ each step and is bounded below. If ϵ is near or at zero the learning can be severely impacted, this is the exploration vs exploitation dilemma. The threshold at which ϵ stops decreasing is set as a hyperparameter.

$$\epsilon_{n+1} = \max(\epsilon_n - \delta, \epsilon\_min)$$

$$\epsilon_0 = 0$$

The Trainer class has a method to choose an action following the ϵ-greedy policy. To do so, we can simply generate $u \sim U(0,1)$ and if $u < \epsilon$ then we generate a random action uniformly across the sample space of the six possible actions, otherwise we choose the action having the highest Q-value. To choose the best action, we can use double Q-learning by predicting the Q-value using the learning Q-network instead of the target network.

```
Algorithm 7: Select next action following $\epsilon$-greedy policy given state $s$
  Initialise $u \sim U(0,\ 1)$
  if $u < \epsilon$ then
  |  Pick $a$ randomly over the set of actions
  end
  else
  |  $a := \max_a Q_\theta(s, a)$
  end
```

Deep Q-network model

At the beginning of the project, we implemented a "dummy" multi-layer perceptron network to be able to run system tests and make sure the whole training workflow was working. This MLP would serve as a mock object and have the same inputs/outputs. Even though it led to very poor results, we were confident to move on to implementing a full DQN.

The DQN implemented in Pytorch has the same architecture than the previous one implemented with Tensorpack. The architecture is as shown in the background section 2.4.1. It takes as input a Pytorch tensor of size $batch\_size \times 1 \times 4 \times 45 \times 45 \times 45$. 1 is to easily scale it to multiple agents without changing number of dimensions, 4 is the frame history, and the 45s are the state dimensions. The input is a voxel value between 0 and 255 that we scaled between 0 and 1 which helps the network converge faster. The output of the model is a Pytorch tensor with dimensions $batch\_size \times 1 \times 6$. 1 is again to easily scale up to more agents and 6 is to predict the Q-value for each of the 6 possible actions. For the forward pass of the Q-network, we divide the input tensors by 255 to scale them between 0 and 1 for better convergence. The tensors are sent to the GPU memory (if the machine supports it) for faster training. The output tensor is copied to the CPU memory as Numpy functions use it and it is also stored in the experience replay. The GPU has much less memory than the CPU, and avoiding CUDA out of memory errors have been a regular challenge during the project.

When the network is trained on a mini-batch, it receives states as input and predict a Q-value for each action. We can then use the actual distance improvements (i.e. the rewards) with the Bellman equation to back-propagate the error. As in the original implementation, we use the Huber loss (also called Smooth L1 loss), double Q-learning and the reward is capped between -1 and 1.

```
Algorithm 8: Compute loss for our single agent DQN
  Sample transition from mini-batch and observe state $s$, action $a$, reward $r$ and next state $s'$
  $\hat{r} := \min(\max(r, -1),\ 1)$
  if $s$ is not terminal then
  |  $\hat{r} := \hat{r} + \gamma \max_{a'} \hat{Q}_{\hat{\theta}}(a',\ s')$
  end
  $loss = SmoothL1(Q_\theta(s,\ a),\ \hat{r})$
```

Experience replay

The experience replay is a class to store the tuples `(state, action, reward, next state, terminal)`. The class is instantiated with a maximum size, the shape of the states to store, the history length, i.e. the number of consecutive states the model trains on (which we set to 4), and the number of agents which in this chapter is always 1.

During an episode, at each step the agent takes, it would append the new tuple to the experience replay via its `append` method. The state and next state are the state the agent was in and the state it arrived upon taking the step respectively. The action is the one taken on said step, whether it was a random step or the best predicted step. The reward is a scalar given by the environment as explained above. Terminal state is a boolean indicating whether the episode ended on this step.

The experience replay also implements a `sample` method. The method generates an index at random from which it takes the last four transitions. It pads zero transitions if the index generated is before the fourth step of an episode.

I naively first implemented the experience replay storing each observation as a stack of the four last states directly. This meant it was easy to sample as we could simply select a random observation directly but we saved each state four times across consecutive observations. Changing to saving each state once and taking slices of four consecutive states when sampling reduced the memory used by a factor of four.

3.2.3 Evaluating

Evaluating a model is the second feature mentioned in the system requirements. Calling the script `DQN.py` with `eval` as `task` argument allows to evaluate the model which path is given in the `load` argument.

Evaluator class

The `evaluator` class handles the logic to evaluate a model. It calls a method named `play_one_episode` on all the files given in the `files` argument. The `play_one_episode` simply runs an episode in a similar fashion as is done in the training with the exception the policy is now fully greedy. At the end of the episode, the `logger` class records the metrics. We cover the `logger` class further down this chapter.

3.2.4 Testing

Testing the agent (not to confuse with testing the code, which we discuss later) is also done within the `evaluator` class. The difference with the evaluating mode is there is no labelled landmarks coordinates. The logger handles the results differently by merely outputting the coordinates of the agent at the end of the episode.

3.2.5 Utils

We have implemented a couple of utility classes. We present them below to have a complete overview of the code base and more fully understand some of its quirks.

Logger

The logger is our main interface to log our results. In particular, we use Tensorboard which is TensorFlow's visualisation toolkit via `torch.utils.tensorboard`.

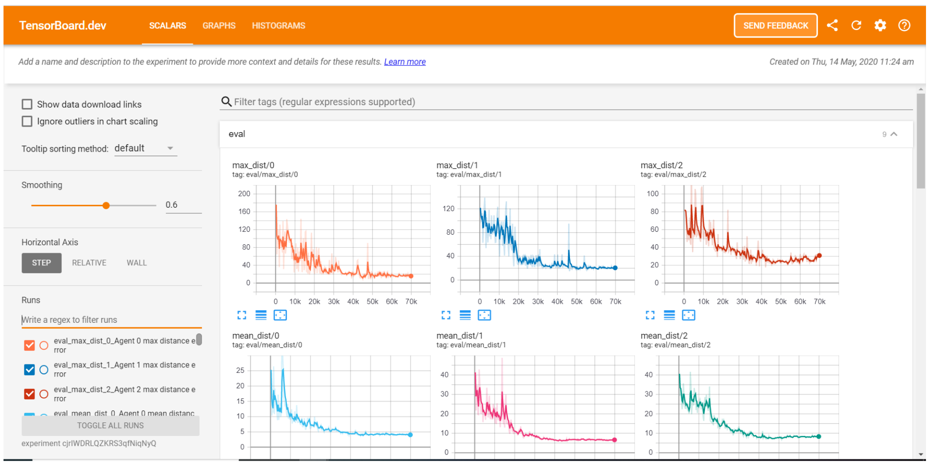

TensorBoard.dev
SCALARS
GRAPHS
HISTOGRAMS
SEND FEEDBACK
Add a name and description to the experiment to provide more context and details for these results. Learn more
Created on Thu, 14 May, 2020 11:24 am
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
eval_max_dist_0_Agent 0 max distance error
eval_max_dist_1_Agent 1 max distance error
eval_max_dist_2_Agent 2 max distance error
eval_mean_dist_0_Agent 0 mean distanc
TOGGLE ALL RUNS
experiment cjrIWDRLQZKRS3qfNiqNyQ
Filter tags (regular expressions supported)
eval
max_dist/0
tag: eval/max_dist/0
max_dist/1
tag: eval/max_dist/1
max_dist/2
tag: eval/max_dist/2
mean_dist/0
tag: eval/mean_dist/0
mean_dist/1
tag: eval/mean_dist/1
mean_dist/2
tag: eval/mean_dist/2

Figure 3.2: Example of a Tensorboard interface showing experiments.

Whenever a training is run, the logger creates a folder named as the date and stores the following files: `logs.txt`, `latest_dqn.pt`, `best_dqn.pt`, and an events.out.tfevents file. The `logs.txt` file stores all the prints shown on the terminal during the training. To do this, instead of calling the standard `print` function when needed, we call the `log` method which prints the text given as argument and stores in the the log file. The log message also saves the text in the events.out.tfevents file which can be viewed on the Tensorboard. The `.pt` files are how Pytorch handle saving models. At the end of each epoch, the Q-network is saved as `latest_dqn.pt`, and if the scores on the validation files were the best recorded so far, it is also saved as `best_dqn.pt`. The logger also has a method `write_to_board` which is used to create various graphs. We store the minimum, mean and maximum distances to the labelled landmark using a fully greedy policy every epoch on the validation files (if provided). We also store those three metrics using the current ϵ-greedy policy at each epoch on the training files (which are necessarily provided). The dashboard also displays the value of ϵ, the loss and the return (sum of the rewards) of the agent at each episode. This tool allows to store easily experiments and quickly spot if something goes wrong with the model such as a divergent loss or an overfit of the data.

Whenever we run the code in the evaluation or testing mode, a similar `logs.txt` file is generated and also a table named `results.csv`. This CSV file writes for each medical image in the dataset the 3D coordinates of the agent at the end of the episode. If it is in evaluation mode, it also writes the true 3D coordinates and the distance error.

Data reader

The data reader supports reading landmarks from `.txt` and `.vtk` files. Such file has one line per landmark and each line contains three scalars separated by a space which are respectively the x, y and z coordinates of the corresponding landmark. The method `sample_circular` can be called with the list of landmarks IDs needed and returns the landmark coordinates on the next medical image file.

Viewer

Viewer is a handy class which as its name indicates allows to visualise the medical image, the agent with its observable space and the distance error. This is especially useful for debugging, e.g. we can notice whether the agent gets stuck in an undetected oscillation. To easily draw a window we use the `pyglet` package[2].

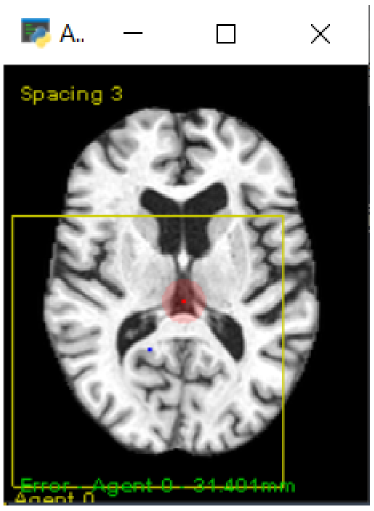


Figure 3.3: Single agent model being evaluated on the posterior commissure landmark in brain MRI scans. The blue dot represents the agent with its field of view being a yellow square around it. The red dot is the landmark, its red circle around represents its distance to the agent in the z dimension. The spacing 3 indicates the agent's steps are of size 3, and once it oscillates, the spacing is reduced to 2 then 1 and the yellow square shrinks accordingly. Finally, at the bottom one can see the distance error, if the previous step got the agent closer to the goal the text is green, otherwise it is red.

[2]https://github.com/pyglet/pyglet

3.2.6 Continuous Integration

Technical debt in machine learning application can accrue more quickly that in traditional software. Some anti-patterns are often present in ML applications such as glue code, pipeline jungles and dead experimental codepaths [42]. Following software engineering good practices such as continuous integration can be tricky as behaviour is often complex and harder to test.

The code is on Github at `https://github.com/gml16/rl-medical` which makes it easier to automatically run tests every time a push is made. This is configured in the file `pythonapp.yml` inside the folder `.github/workflows`. We focus on unit tests rather than system tests in a suite of automated tests using the package `pytest`. We also enforce a global consistent coding style using `Flake8`.

3.3 Summary

We built on the legacy code of the single agent implementation to lay a strong foundation for future extensions such as multiple agents following software engineering best practices. To do this, we changed from using the Tensorpack framework to Pytorch and re-implemented the main features: training, evaluation and testing. We model the task as a partially observable Markov decision process. This process is defined from the environment's states and transition probabilities and the agent's actions, observations, rewards and observation function. The discount factor is a hyperparameter of the model. While the medical image (i.e. environment) is immutable, the agent's observations are the regions of interest around it and its actions are to move in one of six directions. This gives a higher reward if the agent got closer to the goal and lower otherwise.

Chapter 4

Multi-agent Anatomical Landmark Detection

4.1 Design considerations

This work focuses on cooperative agents and leaves aside competitive agents as well as central authoritative entities [31]. Communicating agents searching different landmarks can improve accuracy over single agents. Finding one landmark may help looking for others under the assumption that landmarks' locations are interdependent.

There are different approaches to introduce multiple cooperative agents in this task. One way is to have multiple agents search for the same landmark, this for example allows to reduce the inherent randomness of their initial positions. Another method is to have each agents look for a different landmark, this could allow to search in parallel. A third hybrid approach is to have multiple agents look for multiple landmarks simultaneously.

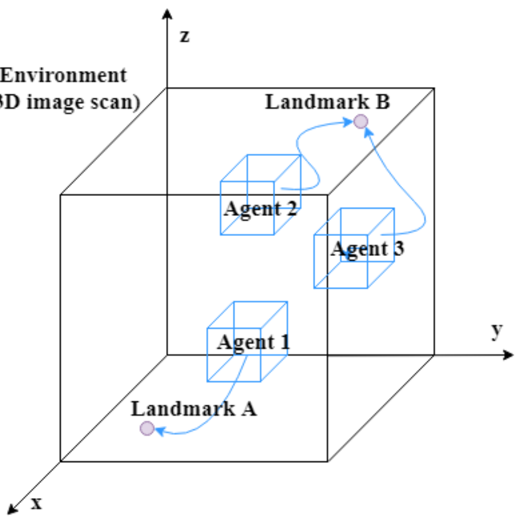


Figure 4.1: Diagram of our proposed multi-agent environment. In this scenario, we have an example of the hybrid approach. One can see three agents' state in blue where agent 1 learns to find landmark A and agents 2 and 3 look for landmark B. The blue arrows show the agents' learnt policies.

The goal is still the same, i.e. finding the best policy to localise landmarks using deep Q-learning. The setting is slightly different as we are now considering a concurrent partially observable Markov decision process as there are multiple agents. Contrary to tasks with central authoritative entities controlling all agents, each agent is independent and cooperatively learns its own policy.

4.2 System implementation

We built the multi-agent system on top of our single agent code. The user can choose the number of agents using the optional arguments. The deep Q-learning algorithm can be extended to take into account multiple agents.

Algorithm 9: Our proposed deep Q-learning algorithm for n agent(s)

```
Initialise weights $\theta$ of $Q_\theta$
Initialise $\hat{Q}_{\hat{\theta}}$ with $\hat{\theta} = \theta$
Initialise empty replay buffer $\mathcal{D}$
for each episode do
    $S := (s_{init,1}, ..., s_{init,n})$
    for each step of episode do
        Choose A = $(a_1, ..., a_n)$ from S using policy derived from $Q_\theta$
        for each agent i do
            Take $a_i$, observe $r_i$ and $s'_i$
        end
        Store transition $((s_1, ..., s_n), (a_1, ..., a_n), (r_1, ..., r_n), (s'_1, ...s'_n))$ in $\mathcal{D}$
        Sample mini-batch $\mathcal{B} \subset \mathcal{D}$ of size $N$
        Observe states $S$, actions $A$, rewards $R$ and next states $S'$ from $\mathcal{B}$
        for each agent i do
            $\hat{r_i} := \min(\max(r_i, -1),\ 1)$
            if $s_i$ is not terminal then
                $\hat{r_i} := \hat{r_i} + \gamma \max_{a'} \hat{Q}_{\hat{\theta}}(a',\ s'_i)$
            end
        end
        $loss = SmoothL1(Q_\theta(S,\ A),\ \hat{R})$
        Update $\theta$ with respect to loss using optimiser with learning rate $\alpha$
        Every $K_1$ epochs, set $\hat{\theta} := \theta$
        Every $K_2$ epochs, set $\alpha := K_3\alpha$
        $S := S'$
    end
end
```

We designed the data reader in such a way that we can give a list of landmark IDs as an argument and each agent receives its corresponding landmark to search for. For instance, if we give as landmark ID argument the list `[1, 2, 3, 4, 5]`, five agents learn with the first one searching for the first landmark, the second one for the second landmark, and so on. This has the enormous advantage of easily choosing whether we want different agents for each landmark, multi-agents on a single landmark or the hybrid: multi-agent per landmark, with multiple landmarks. So for the second case (multi-agents on a single landmark), one could just pass as landmark ID argument the list `[3, 3, 3, 3]`. This means four agents look for the third landmark. In the hybrid case, one could search for the second, fifth and sixth landmark with two agents for each with the following list: `[2, 2, 5, 5, 6, 6]`.

We had to make some changes to the original implementation to support multiple agents, even despite the fact we always kept in mind to write flexible code for future agents. For example the states are Pytorch tensors which had an extra dimension to stack multiple agents' RoI. We had also enabled to display multiple agents in the `viewer.py` file from the beginning. However, there were still many little details we had not taken into account when designing the software for single agents. Hyperparameters needed adjustments such as the initial memory size of the memory buffer because too many agents would lead to CUDA out of memory errors.

4.2.1 Oscillation detection in multi-agent setting

One other change we made to the original collab-DQN implementation was how oscillations were detected. In the original code, there was one unique state history for all agents and agents used to all reduce their step size whenever one agent was oscillating. In our proposed collab-DQN method, we introduced a unique state history for each agent and waited for all agents to oscillate before reducing the step size. Detecting more effectively agents stuck in an endless loop led to an improved accuracy.

Algorithm 10: Multi-agent oscillation detection

```
Initialise $step\_size$ some positive integer
Initialise episode
for $each\ agent\ i$ do
    Initialise empty state history $H_i$ of maximum size $M_i$
end
for $each\ step\ taken\ in\ the\ environment$ do
    $oscillating := True$
    for $each\ agent\ i$ do
        Add new state $s_i$ to $H_i$
        if $length(H_i) > M_i$ then
            Discard oldest state in $H_i$
        end
        if $s_i$ $occurs\ in$ $H_i$ $less\ than$ $T$ $times$ then
            $oscillating := False$
            Exit for loop
        end
    end
    if $oscillating$ then
        if $step\_size = 1$ then
            End episode
        end
        else
            $step\_size := step\_size - 1$
        end
    end
end
```

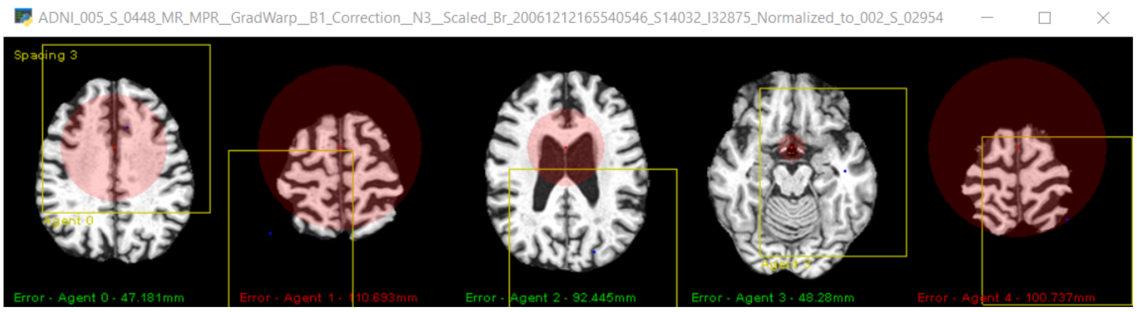
4.2.2 Parallel single agents

A naive implementation would be to run our single agent in parallel on each landmark. Once all agents reach their final position, one may take their average position or the location where the most agents land on. Even though this does reduce unfortunate starting locations of one agent, there is no communication which make it miss out on the full potential of multiple agents.

If we decide to run each single agent on a different landmark in parallel it does have the advantage of reducing the testing/evaluation time but that could also be the case of communicating agents.

This method has little to no advantage over cooperating agents. We thus look into those.

4.2.3 Collab-DQN improvements


ADNI_005_S_0448_MR_MPR__GradWarp__B1_Correction__N3__Scaled_Br_20061212165540546_S14032_I32875_Normalized_to_002_S_02954

Spacing 3
Agent 0
Agent 3
Error - Agent 0 - 47.181mm
Error - Agent 1 - 110.693mm
Error - Agent 2 - 92.445mm
Error - Agent 3 - 48.28mm
Error - Agent 4 - 100.737mm

Figure 4.2: 5 collab-DQN agents during training. They are simultaneously looking for the anterior commissure landmark in brain MRI scans. Screenshot has been taken on the agents' first step in the environment, right after they were spawned within the 80% starting zone.

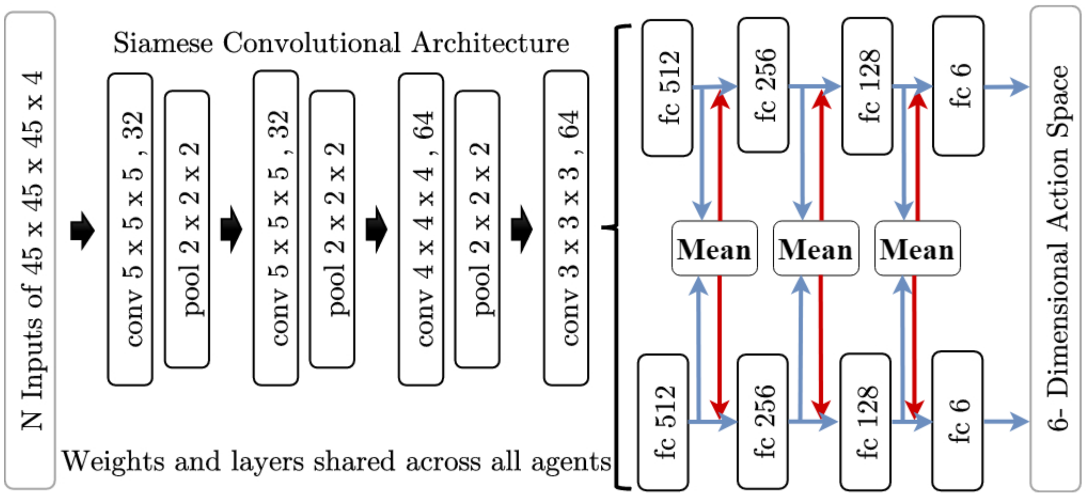


Figure 4.3: CommNet architecture integrated to landmark detection for 2 agents. This can be extended for any number of agent. The architecture is first composed of a Siamese convolutional network shared by all agents, followed by individual fully connected layers. Each FC layer sends its output to the next and also to a communication channel (blue arrows). The communication channel concatenates its average to the input on the next FC layer (red arrows).

Here, we integrated the collab-DQN approach shown in section 2.3.6 using Pytorch. The Siamese architecture with common convolutional layers allow the agents to cooperatively learn a representation of the medical images. Each agent has its own fully connected layers which allow it to learn its assigned landmark's position from the commonly learnt features representation. Even though this communication is implicit, it improves performances on several aspect in comparison to the single agent which we cover in the evaluation chapter.

Our code was written keeping in mind we would introduce multiple agents. Thus, it was fairly simple to add a new architecture. We extended the Pytorch class `torch.nn.Module` with our new class `Network3D` that we added in our `DQNModel.py` file. The forward pass takes as argument a Pytorch tensor of size $batch\_size \times num\_agents \times 4 \times 45 \times 45 \times 45$ and outputs a tensor of size $batch\_size \times num\_agents \times 6$. The input is also scaled down from 0-255 to 0-1 for better convergence.

4.2.4 CommNet implementation

We integrated one of the key ideas of the CommNet architecture presented in section 2.3.6. The shared convolutional layers efficiently learn an implicit communication between the agents with much less memory overhead [3]. We added new communication channels between the fully connected layers of the agents. In this case, the communication is learnt explicitly via back-propagation through the added channels.

Similarly as with our collab-DQN implementation, we extended the `torch.nn.Module` with our new class `CommNet` that we added in our `DQNModel.py` file. The inputs and outputs are Pytorch tensors of the same dimensions with the same scaling applied.

4.3 Summary

Cooperative multi-agents can have improved accuracy in anatomical landmark detection by communicating about their respective states as we assume landmarks are inter-dependent. We implemented the collab-DQN which shares common convolutional layers across agents. We then used the main idea of CommNet by adding communication channels between the fully connected layers of the collab-DQN for finer, learnt communication. Our proposed method also has the flexibility to handle a different number of agents for each landmark. This allows to have all agents search for one landmark, or each agent can look for a different landmark, or a mix of both.

Chapter 5

Evaluation

This chapter compares the accuracy of the single agent and multi-agent architectures as well as other metrics recorded during training such as the loss and min/average/max distances over epochs on our three datasets.

5.1 Datasets

We have randomly split the dataset into training, validation and testing sets with the proportions 70:15:15. The validation set was used to assess when the network was overfitting and thus to choose the epoch at which to save the optimal weights. The testing set was used to compute the accuracy scores that we discuss below.

One dataset is composed of 455 short-axis cardiac MRI scans of resolution $1.25 \times 1.25 \times 2$mm obtained from the UK Digital Heart project [43]. There are six landmarks' coordinates annotated by expert clinicians [44]. We focus our experiments on the following five landmarks to compare to previous works: the apex, centre of the mitral valve (MV), two right ventricle (RV) insert points (the intersection between the RV outer boundary and the LV epicardium), and the RV lateral wall turning point. As a baseline, we also have the inter-observer errors for the apex and MV centre which are 5.79 ± 3.28mm and 5.30 ± 2.98mm respectively.

The second dataset is from the ADNI database: 832 MRI isotropic 1mm brain scans [45]. We used the following landmarks out of the twenty available: anterior commissure (AC), posterior commissure (PC) as well as the outer aspect, inferior tip and inner aspect of the splenium of the corpus callosum (SCC).

Our third dataset is a collection of 72 fetal head ultrasound scans [5]. Amongst the thirteen annotated landmarks, we used the right and left cerebellum (AC and LC respectively) and cavum septum pellucidum (CSP).

5.2 Experimental setups

We used the departmental machines and GPU cluster with Slurm to run our experiments. Each experiment ran for about four days but would converge usually after one or two days. We used CUDA version 10.0.130, Torch v1.4, GPU cards had 12GB RAM and were either Nvidia Tesla or Nvidia GeForce GTX Titan Xp, 24-core/48 thread Intel Xeon CPUs with 256GB RAM.

5.3 Experiments

5.3.1 Single agent

The first experiments shown here are the ones we ran after the refactoring of the original single agent implementation from Tensorpack to Pytorch. If all went well we excepted similar results. They would not exactly be the same as some frameworks can train faster than others. Furthermore, our datasets have different splits, while we use 70:15:15, the original paper did not have validation files and had a 80:20 ratio, which makes the training set larger. Also some hyperparameters are slightly different. For instance, ϵ which dictates the proportion of greedy actions over random ones

Landmark	Original single agent	Our single agent
Apex	4.47±2.63	**4.38±2.49**
MV	5.73±4.16	**5.10±2.54**
AC	2.46±1.44	**1.14±0.53**
PC	2.05±1.14	**1.18±0.55**
CSP	**3.66±2.11**	9.90±3.13
RC	**3.37±1.54**	7.23±3.54
LC	**3.25±1.59**	4.37±1.45

Table 5.1: Single agent distance error (in mm). *Original single agent* are the results as reported in the original paper [5] while *our single agent* results come from our own experiments.

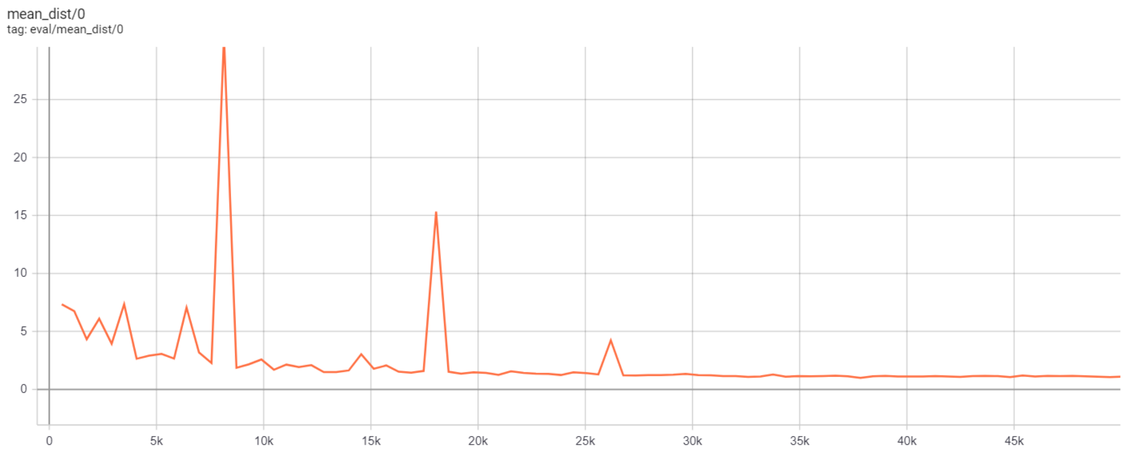


Figure 5.1: Average distance error on the validation set over the number of episodes. Single agent trains for an epoch on the posterior commissure landmark and is then evaluated on the validation set.

had a somewhat different interpolation function to determine its decrease over epochs. We were not after an exact replica of the original implementation but rather one which worked at least as well. These experiments can be considered a sanity check that no bugs were introduced.

Table 5.1 shows results from the reported implementation and our own. Interestingly, our implementation performs better on the cardiac and brain datasets but worse on the fetal one. Nevertheless, the single agent is actually learning and can find its way to the landmark under different types of medical images. In particular it performs very well on the AC where it more than halved the average error distance from 2.46 to 1.14. The single agents training on the CSP and RC landmarks performed quite poorly. However, this may be explained as this training set is the smallest. The original implementation did not have a validation dataset, this does not have much impact when the dataset is big enough but in the case of the fetal ultrasounds, each scan counts. Likewise, the testing set is comprised of only eleven scans, which gives a larger standard deviation when evaluating.

To have a better overview of our models, we looked at additional metrics. We include the corresponding graphs taken during the training of the agent on the posterior commissure landmark. We show the first 50,000 episodes of the training which took thirty-four hours. The rest of the training is not shown, the agent has converged and is relatively constant. The other metrics recorded on Tensorboard can also be seen online[1]. Tensorboards for other single agents on different landmarks are quite similar and their links are in the appendix.

In figures 5.1, 5.2, 5.3 and 5.4 we can see the mean distance error, maximum distance error, loss and score respectively. The average distance error is around 1.1 on the validation set. This is about similar with the reported result on the testing set. The maximum error towards the end of the training is around 2.4, which means that in the worst case scenario we can expect the agent

[1]`https://tensorboard.dev/experiment/S046BMkzQa2g72ulTKk75w`

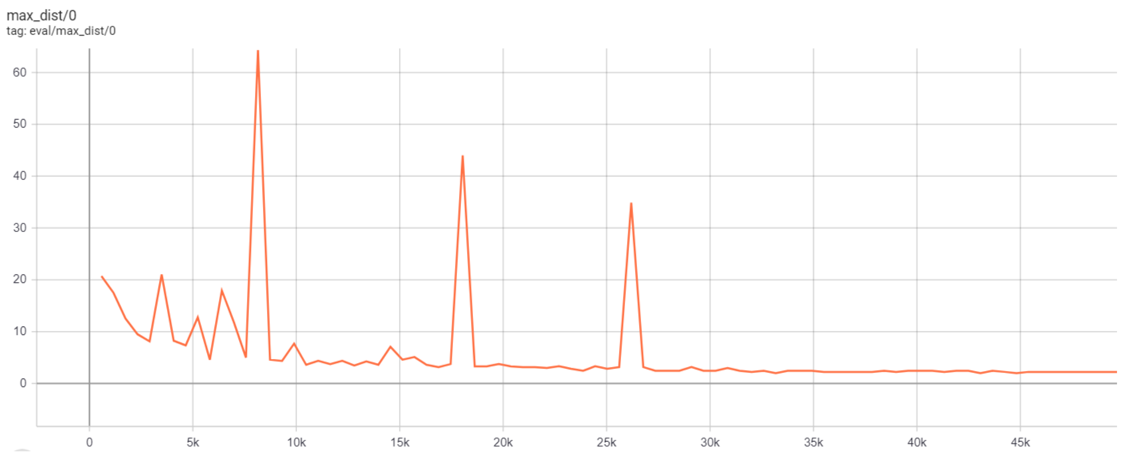


Figure 5.2: Maximum distance error on the validation set over the number of episodes. This shows worst cases are getting closer and closer to the correct landmark.

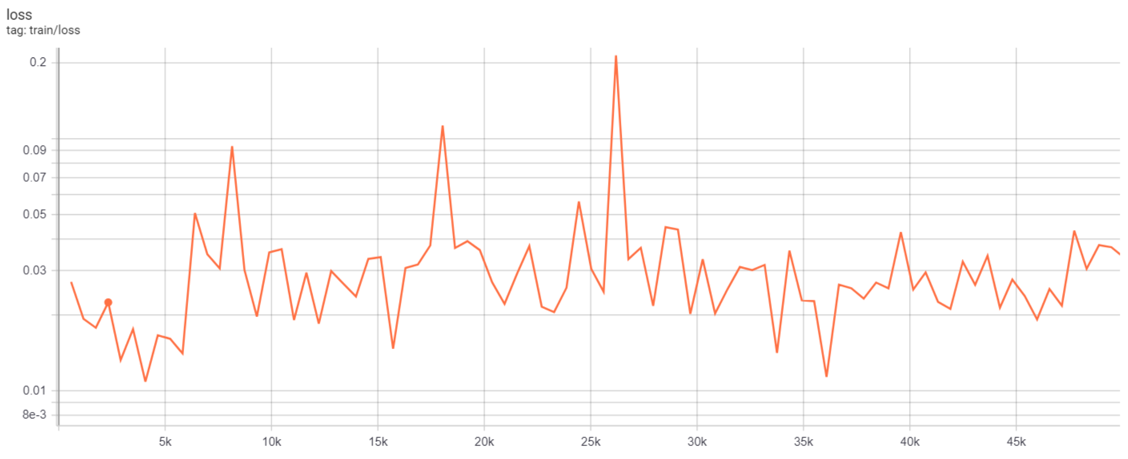


Figure 5.3: Huber loss of the single agent's DQN over the number of episodes (log-scale).

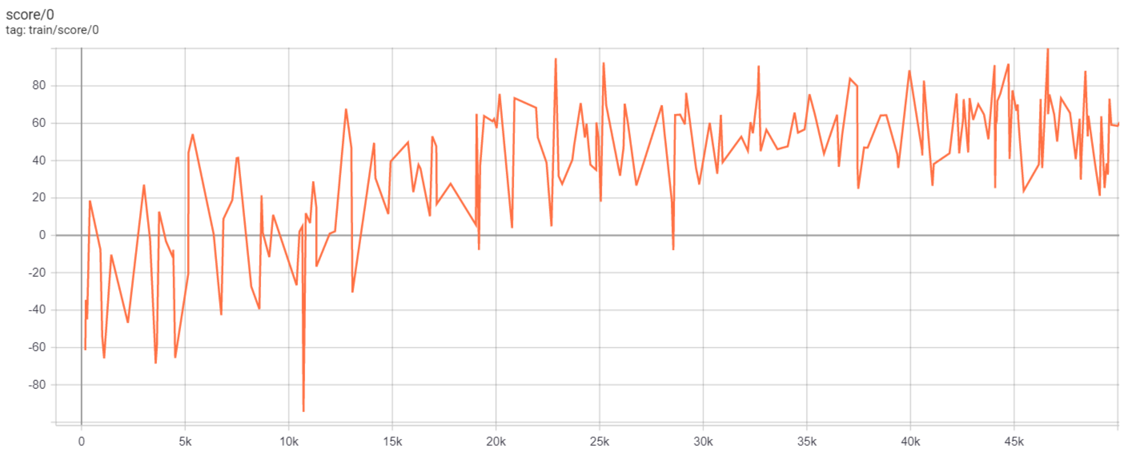


Figure 5.4: Score for each episode on the training set over the number of episodes.

	Original collab-DQN		Our collab-DQN	
Landmark	**3 agents**	**5 agents**	**3 agents**	**5 agents**
AC	**0.94±0.17**	0.98±0.25	1.16±0.59	1.13±0.64
PC	0.96±0.20	**0.90±0.18**	1.25±0.57	1.19±0.61
Outer SCC	1.45±0.51	1.39±0.45	**1.38±0.75**	1.51±0.77
Inferior SCC	-	1.42±0.90	-	**1.39±0.85**
Inner SCC	-	1.72±0.61	-	**1.53±0.97**

Table 5.2: Comparison between the original collab-DQN and our implementation for 3 and 5 agents on brain MRI scans.

	Single agent	**Collab-DQN**		**CommNet**	
Landmark		**3 agents**	**5 agents**	**3 agents**	**5 agents**
AC	1.14±0.53	1.16±0.59	1.13±0.64	**1.06±0.53**	1.12±0.65
PC	1.18±0.55	1.25±0.57	1.19±0.61	**1.10±0.60**	1.25±0.55
Outer SCC	1.47±0.64	**1.38±0.75**	1.51±0.77	1.43±0.65	1.62±0.79
Inferior SCC	2.40±1.13	-	**1.39±0.85**	-	1.50±0.89
Inner SCC	**1.46±0.73**	-	1.53±0.97	-	1.53±0.76

Table 5.3: Distance errors (in mm) in the brain MRI scans for our single agent, collab-DQN and CommNet implementations.

to be 2.4mm off the PC landmark. There are a few spikes during the training which may be explained from setting the target network's weights equal to the DQN's. The loss is around 0.01 and 0.1 during the training and does not diverge, even for the two days after convergence it was left running. We calculated the score by taking the sum of the rewards along an episode. A perfect agent would have a finite score which would fluctuate due to its random initial position in the environment. Our agent's score slowly increases during training and stabilises around 40 and 50. An agent taking random steps would have slightly negative score as, on average, a random step would increase the distance between the agent and the landmark in a Euclidean distance setting. These graphs confirm our agent is indeed learning during a stable training.

One final check is to look at the time and memory usage. The memory used is the highest during training because of the experience buffer. We set its size to 100,000 episodes, most of the space is taken by the states which are $45 \times 45 \times 45$ values of type `unsigned int8` which take up a total of $45 \times 45 \times 45 \times 10^5 = 9.1125 \times 10^9$ bytes. In total the memory taken is about 10^{10} bytes during training. After the training the only storage needed is the DQN's weights which take 2,206,723 bytes. We already mentioned the training takes thirty-four hours for 50,000 episodes, so that is about 2.45 seconds per episode. It is worth noting early episodes are faster because ϵ is close to one and most actions are random, no need to compute the Q-values. During testing, finding one landmark takes about 0.40 second which is faster than during training as expected.

5.3.2 Multiple agents

For our multi-agent methods, we also perform a sanity check that our collab-DQN improvements achieve at least as good results as the original collab-DQN implementations. This can be seen in table 5.2. We can see our collab-DQN performs slightly better than the original implementation in three landmarks out of five. We can also see three and five agents' performances are quite close, similarly to the original implementation. Our implementation is actually learning and seems to be free of any critical bug.

We then compare the different multi-agent methods and also place them in the context of the single agent architecture. Tables 5.3, 5.4 and 5.5 show the distance errors for our single agent, collab-DQN (three and five agents) and CommNet (three and five agents) for landmarks in the three datasets.

Table 5.3 shows the best model to find brain landmarks is usually CommNet with three agents. CommNet seems slightly better than collab-DQN. The single agent is the best in one landmark (Inner SCC) by a small margin. CommNet performs better with three than five agents, however it is hard to tell the optimal number of agents for collab-DQN. It is worth mentioning the three-agent version has been trained on the AC, PC and outer SCC. Different combinations of landmarks may

Landmark	Single agent	Collab-DQN		CommNet	
		3 agents	5 agents	3 agents	5 agents
Apex	4.38±2.49	4.34±2.41	4.61±2.89	4.94±2.57	**4.27±2.52**
MV centre	**5.10±2.54**	5.71±2.92	5.78±3.22	5.69±2.73	5.58±2.41
RV insert point 1	7.42±4.22	7.40±4.56	**5.00±3.87**	7.83±4.63	6.94±3.95
RV lateral wall turning point	15.67±8.59	-	14.68±8.03	-	**13.38±7.40**
RV insert point 2	9.73±5.56	-	9.17±5.37	-	**8.67±5.40**

Table 5.4: Distance errors (in mm) in the cardiac MRI scans for our single agent, collab-DQN and CommNet implementations.

Landmark	Single agent	Collab-DQN		CommNet	
		3 agents	5 agents	3 agents	5 agents
RC	7.23±3.54	**2.73±1.71**	4.20±3.76	6.53±4.21	4.86±2.31
LC	4.37±1.45	**4.20±2.87**	5.98±8.58	5.10±3.66	4.89±3.31
CSP	9.90±3.13	5.18±2.05	8.02±5.34	5.78±3.04	**5.15±4.36**
Fetal L0	29.43±17.83	-	**14.45±5.25**	-	16.23±8.10
Fetal L1	5.73±2.88	-	8.11±5.22	-	**5.13±3.24**

Table 5.5: Distance errors (in mm) in the fetal brain ultrasounds for our single agent, collab-DQN and CommNet implementations.

have different results since they are inter-dependent in a different way. The same remark applies to the two other datasets.

In table 5.4, cardiac landmarks are mostly best detected using CommNet with five agents (achieved best accuracy in three out of the five landmarks). We can also note that five agents CommNet always perform better than with three agents. For collab-DQN the number of agents has an impact in the accuracy, especially for RV insert point 1 (7.40mm for three agents against 5.00mm for five agents). However, the two other landmarks perform slightly better with only three agents. It thus suggests the right number of agents in collab-DQN depends on the landmark we are looking for. Looking at the landmarks for which we have the inter-observer errors, the apex and MV centre (these errors are 5.79±3.28mm and 5.30±2.98mm respectively), we see our best model has super human performances.

Table 5.5 shows fetal landmarks always perform better with multiple agents. With CommNet, five agents always perform better than three agents, however it is the opposite with collab-DQN. In this dataset, collab-DQN outperforms CommNet on three landmarks out of five.

Results show that no method is superior in all landmarks, but rather suggest the best architecture depends on the landmark. Nevertheless, we can note some general trends. For instance, multi-agent outperforms single agents on most landmarks. Collab-DQN is the best model for six landmarks whereas this number is seven for CommNet. Overall, we can note more agents tend to have slightly better accuracy scores. Thus, this suggests that when training agents on a new landmark for the first time, CommNet with five agents is a safer bet.

We now look further than the accuracy scores as we previously did with the single agent. We present below different metrics recorded on the Tensorboards. One can see this Tensorboard online[2]. Tables 5.5 and 5.6 respectively show the maximum and minimum distance error (in mm) for one agent (five in total) evaluated on the mitral valve centre during training over epochs. Table 5.7 show the Huber loss (which is common for all five agents). Two experiments using different architectures have been run, CommNet is in blue and collab-DQN in orange. CommNet has better accuracy than collab-DQN in most epochs using validation files on the MV centre which is consistent with the results. Alternatively, in some experiments on different landmarks, collab-DQN performed better during most of the training as shown by the accuracy measured on the validation files. In those cases, collab-DQN also had the best accuracy on the testing files for its selected model as one would expect. In none of the experiments we have run, had an architecture surpassed

[2]`https://tensorboard.dev/experiment/HrGbwY3aReilfVLRmAgxcQ`

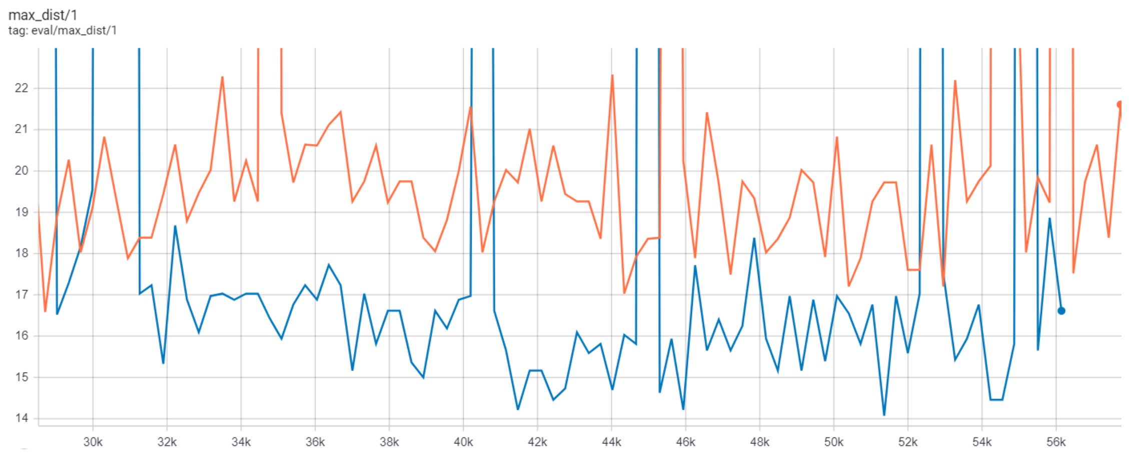


Figure 5.5: Maximum distance error for one validation epoch over the number of episodes on the MV centre in the cardiac dataset at the end of the training. CommNet architecture is in blue and collab-DQN in orange.

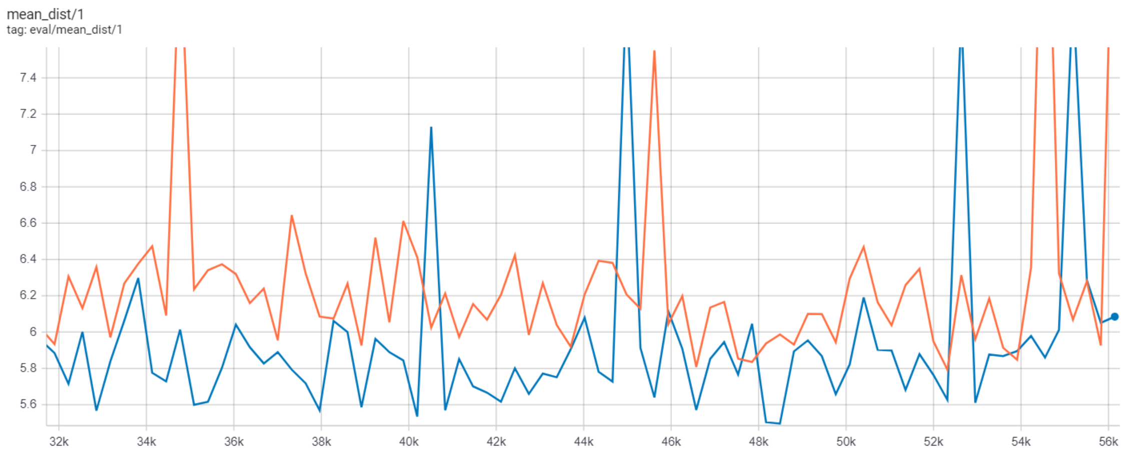


Figure 5.6: Average distance error for one validation epoch over the number of episodes on the MV centre in the cardiac dataset at the end of the training. CommNet architecture is in blue and collab-DQN in orange. The best model chosen is at the epoch with the lowest error. For CommNet it is around epoch 48.5k while for collab-DQN it is around epoch 52.5k.

the other once they both had stabilised mean distances. This shows some architectures are better for some landmarks and it can be detected early on in the training.

In four days, collab-DQN ran 30k episodes while CommNet only ran 20k episodes. CommNet is more time consuming as there are more weighs, they are used in the FC layers' communication channels. The memory space during training is mostly driven up by the memory buffer which we set to $\frac{100{,}000}{\#agents}$ episodes for multi-agent architectures so that it would take about 10GB as in the single agent case. As for the model's size, more agents take up more space and CommNet are bigger than collab-DQN. More precisely, a CommNet model size is 5,504,759 and 8,144,365 bytes for three and five agent respectively while for collab-DQN it is 3,529,451 and 4,852,185 bytes. For comparison, three single agents working independently have model size $2,206,723 \times 3 = 6,620,169$ bytes and for five single agents it is $2,206,723 \times 5 = 11,033,615$ bytes. This shows multi-agent models greatly reduce the models' trainable parameters. For the testing speed, CommNet takes about 2.5 and 4.9 seconds per episode for three and five agents respectively and those figures are 2.2 and 4.2 seconds for collab-DQN. Comparing to the 0.4 seconds per landmark per episode of the single agent, the testing is slower but still much quicker than done by expert clinicians.

All experiments presented thus far are in the scenario where each agent has its own landmark

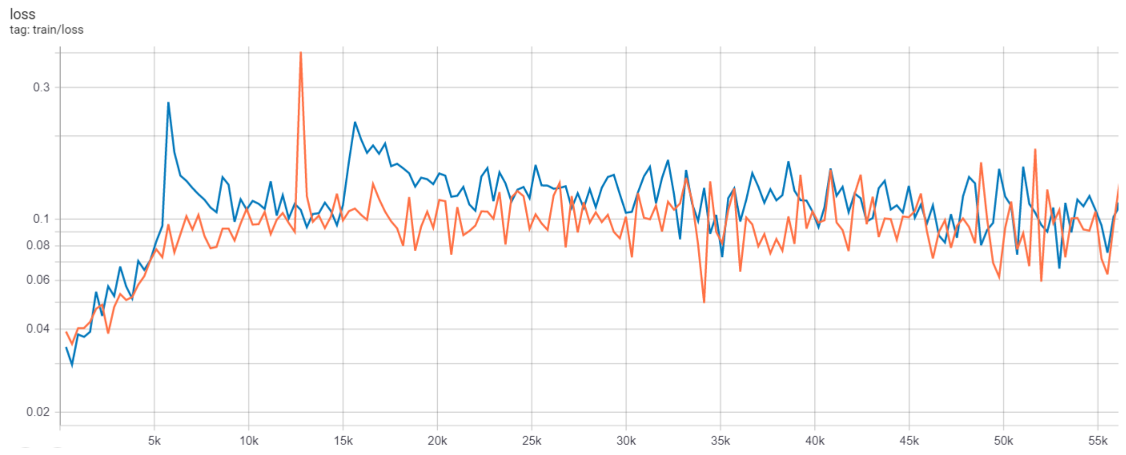


Figure 5.7: Huber loss over the number of episodes during training five agents on the cardiac dataset. CommNet architecture is in blue and collab-DQN in orange.

Landmarks	**Single agents**	**Collab-DQN**	**CommNet**
AC	0.97±0.40	0.81±0.36	**0.75±0.34**
CSP	10.43±4.28	6.66±4.19	**5.10±4.25**
Apex	4.71±4.11	4.06±2.35	**3.94±2.22**

Table 5.6: Distance error (in mm) with five agents looking for the same landmark.

assigned. We proceed to evaluate the accuracy values for the multi-agent variant with five agents on one landmark. The final location of the agents are averaged at the end on an episode. To give a baseline, we included a column for five single agents looking for the same landmark in parallel. We also chose one landmark in each dataset to account for the different scan types. The memory and time complexity is the same as the one-agent-per-landmark variant since the architectures and number of weights do no change.

Table 5.6 shows CommNet results are much better than in any of the previous methods. We can also see that the parallel single agents are not significantly better than the results with only one agent. This is because the mean location of an increasing number of single agents looking for the same landmark simultaneously does not converge towards the true landmark location since their policy are not unbiased.

We present in table 5.7 results for an experiment with four agents and two landmarks. Again, to give a baseline, we show the results for four non communicating agents. CommNet agents are better than the baseline but not better than the all-agents-on-one-landmark version despite the communication between two closely related landmarks: the posterior and anterior commissure. This may suggest that while communication is key to achieving better results, the inter-dependent structure of the anatomy is not the primary reason for it.

Landmarks	**Single agents**	**CommNet**
AC	1.17±0.61	**0.95±0.43**
PC	1.12±0.55	**0.97±0.46**

Table 5.7: Distance error (in mm) with two pairs of agent looking for two landmarks (four agents in total).

5.4 Summary

Evaluation of our proposed models having one agent per landmark showed no absolute superior accuracy across all datasets and landmarks for any one of them. Rather some landmarks have more suited models, for example to find the MV center it is best to use the single agent method while CommNet five agents is better suited for the apex. We have also shown the best model for a given landmark is apparent early on in the training.

We do note general trends. While having less agents is faster to train and test, more agents generally achieve better results. CommNet is also heavier than collab-DQN but achieves the best results for more landmarks than collab-DQN does given a similar training time.

Experimenting with all agents on one landmark shed light on an important realisation: while communication is key to better localise landmarks, it may not be because the structure of the anatomy is inter-dependent. Multiple cooperative agents trained on the same landmark outperforms previous methods even though they are not communicating about multiple dependent landmarks but the same one.

Chapter 6

Conclusions and Future Work

We have presented, implemented and evaluated deep reinforcement learning methods to detect anatomical landmarks in medical images. These methods involve single agent and multi-agent deep Q-learning (DQN) in partially observable Markov decision process (MDP) settings. In particular, we have explored communication between multiple agents. We have found that depending on the landmark, the optimal architecture or number of agents may change. Nevertheless, we showed that identifying the best architecture can be detected early on in the training. We also found that, in general, having more agents achieves better results. CommNet, which has more communication channels, generally outperforms collab-DQN. The best results have been attained by focusing all agents on one landmark. Our implementations outperformed expert clinicians on the two landmarks with accessible inter-observer error data. Overall, cooperating agents improve results, especially when they communicate about the same landmark.

There are still many other ideas to explore. One may want to apply our findings to other DRL medical applications such as view planning described in the background section. There are multiple extensions to DRL we can use such as continuous actions (including actions to dynamically change the size of the agent's region of interest) and prioritised experience replay. It is also possible to work on different architectures and communication models for multi-agent DRL. One can also improve upon the architectures we implemented. For instance, we used the vanilla CommNet and there exists other variants, one that only lets nearby agents communicate, deeming communication between farther agents wasteful or even having a negative impact. Other variants use skip connections and/or temporal units in the communication channel. There is also room for improvement tuning hyperparameters. Outside of DRL there are promising recent works in computer vision which could be applied such as new 3D CNN architectures. Another way to better our proposed models and future ones would be to gather and label more data, this would certainly lead to a greater accuracy as our datasets are relatively small for deep learning methods. One could combine this with data augmentation. While some of those ideas may not increase performances, new attempts may lead to advances in other, seemingly unrelated areas of artificial intelligence or healthcare.

Appendix A

Tensorpack training

Below is part of the original Tensorpack code to train the single agent DQN.

```python

from tensorpack import (TrainConfig, ModelSaver, PeriodicTrigger, ObjAttrParam,
                        HumanHyperParamSetter, RunOp, SimpleTrainer,
                        launch_train_with_config)

def get_config(files_list):
    expreplay = ExpReplay(
        predictor_io_names=(['state'], ['Qvalue']),
        player=get_player(task='train', files_list=files_list),
        state_shape=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
        memory_size=MEMORY_SIZE,
        init_memory_size=INIT_MEMORY_SIZE,
        init_exploration=1.0,
        update_frequency=UPDATE_FREQ,
        history_len=FRAME_HISTORY
    )

    return TrainConfig(
        model=Model(),
        callbacks=[
            ModelSaver(),
            PeriodicTrigger(
                RunOp(DQNModel.update_target_param, verbose=True),
                # update target network every 10k steps
                every_k_steps=10000 // UPDATE_FREQ),
            expreplay,
            ScheduledHyperParamSetter('learning_rate',
                                      [(60, 4e-4), (100, 2e-4)]),
            ScheduledHyperParamSetter(
                ObjAttrParam(expreplay, 'exploration'),
                # 1->0.1 in the first million steps
                [(0, 1), (10, 0.1), (320, 0.01)],
                interp='linear'),
            PeriodicTrigger(
                Evaluator(nr_eval=EVAL_EPISODE, input_names=['state'],
                          output_names=['Qvalue'], files_list=files_list,
                          get_player_fn=get_player),
                every_k_epochs=EPOCHS_PER_EVAL),
            HumanHyperParamSetter('learning_rate'),
        ],
        steps_per_epoch=STEPS_PER_EPOCH,
        max_epoch=1000,
    )

config = get_config(args.files)
# Run the actual training
launch_train_with_config(config, SimpleTrainer())
```

Appendix B

Experiments' Tensorboards

Below is the list of the 37 experiments' tensorboards supporting our results. They can all be found at the same url[1]. For the brain dataset, the landmarks AC, PC, outer SCC, inferior SCC and inner SCC correspond to the numbers 13, 14, 0, 1 and 2 respectively. For the cardiac dataset, the apex, MV centre, RV insert point 1, RV lateral wall turning point and RV insert point 2 are respectively numbered 4, 5, 0, 1 and 2. Finally, in the fetal ultrasound dataset, RC, LC, CSP, fetal L0, fetal L1 correspond to the numbers 10, 11, 12, 0 and 1 respectively.

[1]`https://tensorboard.dev/experiment/rtjVVTdYTPyqS5S9bDk9qQ`

Name	**Dataset**	**Architecture**	**Landmarks**
May02_17-33-04	brain	Single	14
May06_10-59-26	brain	CommNet	14 0 1
May06_12-39-27	cardiac	Single	4
May06_12-43-37	cardiac	CommNet	4 5 0
May06_12-46-35	cardiac	Network3d	4 5 0
May06_13-20-55	fetal	Single	10
May14_23-20-13	brain	CommNet	13 14 0 1 2
May19_10-51-07	brain	Single	2
May21_18-20-01	brain	Single	1
May27_15-17-12	brain	Single	14
May27_15-17-47	brain	Single	0
Jun01_22-53-16	cardiac	CommNet	4 5 0 1 2
Jun01_23-05-46	cardiac	Network3d	4 5 0 1 2
Jun03_00-19-34	brain	Network3d	13 14 0 1 2
Jun03_00-21-46	cardiac	Single	0
Jun03_00-25-11	cardiac	Single	2
Jun03_00-26-25	cardiac	Single	5
Jun03_00-29-13	cardiac	Single	1
Jun03_00-31-17	cardiac	Network3d	4 5 0 1 2
Jun03_00-35-06	cardiac	CommNet	4 5 0 1 2
Jun06_10-39-26	brain	CommNet	13 13 13 13 13
Jun06_10-40-12	brain	CommNet	13 13 14 14
Jun10_11-22-02	brain	Network3d	13 14 0
Jun10_11-22-18	brain	CommNet	13 14 0
Jun12_02-08-31	brain	Single	13
Jun12_02-09-19	fetal	CommNet	12 12 12 12 12
Jun12_02-10-11	cardiac	CommNet	4 4 4 4 4
Jun12_02-11-08	fetal	Single	11
Jun12_02-11-41	fetal	Single	12
Jun12_02-12-57	fetal	Single	0
Jun12_02-14-28	fetal	Single	1
Jun12_02-26-48	fetal	Network3d	10 11 12
Jun12_03-38-53	fetal	CommNet	10 11 12 0 1
Jun12_03-40-08	fetal	Network3d	10 11 12 0 1
Jun12_03-44-44	fetal	CommNet	10 11 12
Jun14_11-09-34	fetal	Network3d	12 12 12 12 12
Jun14_11-10-03	cardiac	Network3d	4 4 4 4 4

Appendix C

Command line arguments

Output from running the command `python DQN.py -help`. This shows the different features and hyperparameters along with their default values available in our proposed landmark detection software.

```
usage: DQN.py [-h] [--load LOAD] [--task {play,eval,train}]
              [--file_type {brain,cardiac,fetal}] [--files FILES [FILES ...]]
              [--val_files VAL_FILES [VAL_FILES ...]] [--saveGif]
              [--saveVideo] [--logDir LOGDIR] [--agents AGENTS]
              [--landmarks [LANDMARKS [LANDMARKS ...]]]
              [--model_name {CommNet,Network3d}] [--batch_size BATCH_SIZE]
              [--memory_size MEMORY_SIZE]
              [--init_memory_size INIT_MEMORY_SIZE]
              [--max_episodes MAX_EPISODES]
              [--steps_per_episode STEPS_PER_EPISODE]
              [--target_update_freq TARGET_UPDATE_FREQ]
              [--save_freq SAVE_FREQ] [--delta DELTA] [--viz VIZ]
              [--multiscale] [--write] [--train_freq TRAIN_FREQ]

optional arguments:
  -h, --help            show this help message and exit
  --load LOAD           Path to the model to load (default: None)
  --task {play,eval,train}
                        task to perform, must load a pretrained model if task
                        is "play" or "eval" (default: train)
  --file_type {brain,cardiac,fetal}
                        Type of the training and validation files (default:
                        train)
  --files FILES [FILES ...]
                        Filepath to the text file that contains list of
                        images. Each line of this file is a full path to an
                        image scan. For (task == train or eval) there should
                        be two input files ['images', 'landmarks'] (default:
                        None)
  --val_files VAL_FILES [VAL_FILES ...]
                        Filepath to the text file that contains list of
                        validation images. Each line of this file is a full
                        path to an image scan. For (task == train or eval)
                        there should be two input files ['images',
                        'landmarks'] (default: None)
  --saveGif             Save gif image of the game (default: False)
  --saveVideo           Save video of the game (default: False)
  --logDir LOGDIR       Store logs in this directory during training (default:
                        runs)
  --agents AGENTS       Number of agents (default: 1)
  --landmarks [LANDMARKS [LANDMARKS ...]]
                        Landmarks to use in the images (default: [1])
  --model_name {CommNet,Network3d}
                        Models implemented are: Network3d, CommNet (default:
                        CommNet)
  --batch_size BATCH_SIZE
                        Size of each batch (default: 64)
  --memory_size MEMORY_SIZE
                        Number of transitions stored in exp replay buffer. If
                        too much is allocated training may abruptly stop.
                        (default: 100000.0)
  --init_memory_size INIT_MEMORY_SIZE
                        Number of transitions stored in exp replay before
                        training (default: 30000.0)
  --max_episodes MAX_EPISODES
                        "Number of episodes to train for" (default: 100000.0)
  --steps_per_episode STEPS_PER_EPISODE
                        Maximum steps per episode (default: 200)
  --target_update_freq TARGET_UPDATE_FREQ
                        Number of episodes between each target network update
                        (default: 10)
  --save_freq SAVE_FREQ
                        Saves network every save_freq steps (default: 1000)
  --delta DELTA         Amount to decreases epsilon each episode, for the
                        epsilon-greedy policy (default: 0.0001)
  --viz VIZ             Size of the window, None for no visualisation
                        (default: 0.01)
  --multiscale          Reduces size of voxel around the agent when it
                        oscillates (default: False)
  --write               Saves the training logs (default: False)
  --train_freq TRAIN_FREQ
                        Number of agent steps between each training step on
                        one mini-batch (default: 1)
```

Bibliography

[1] Chao Yu, Jiming Liu, and Shamim Nemati. Reinforcement Learning in Healthcare: A Survey. aug 2019.

[2] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[3] Athanasios Vlontzos, Amir Alansary, Konstantinos Kamnitsas, Daniel Rueckert, and Bernhard Kainz. Multiple landmark detection using multi-agent reinforcement learning. In Dinggang Shen, Tianming Liu, Terry M. Peters, Lawrence H. Staib, Caroline Essert, Sean Zhou, Pew-Thian Yap, and Ali Khan, editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*, pages 262–270, Cham, 2019. Springer International Publishing.

[4] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. *CoRR*, abs/1605.07736, 2016.

[5] Amir Alansary, Ozan Oktay, Yuanwei Li, Loic Le Folgoc, Benjamin Hou, Ghislain Vaillant, Konstantinos Kamnitsas, Athanasios Vlontzos, Ben Glocker, Bernhard Kainz, and Daniel Rueckert. Evaluating reinforcement learning agents for anatomical landmark detection. *Medical Image Analysis*, 53:156–164, apr 2019.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015.

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[8] Pieter Abbeel John Schulman. *CS 294: Deep Reinforcement Learning*. 2015.

[9] David Silver. Reinforcement learning.

[10] Edward Johns Aldo Faisal. Co424 reinforcement learning.

[11] Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

[12] Minsky Marvin and A Papert Seymour. Perceptrons, 1969.

[13] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

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

[15] torch.nn.init - pytorch documentation.

[16] Module: tf.keras.initializers - tensorflow core v2.2.0.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[19] Siddharth Krishna Kumar. On weight initialization in deep neural networks. *CoRR*, abs/1704.08863, 2017.

[20] Michael Bronstein, Stefanos Zafeiriou, and Bjoern Schuller. 460 deep learning, 2019-2020.

[21] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99. Springer, 2006.

[25] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[26] Thore Graepel. AlphaGo - Mastering the game of go with deep neural networks and tree search. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9852 LNAI, page XXI. Springer Verlag, 2016.

[27] Timothy P. Lillicrap, Jonathan J. Hunt, Alexand er Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, page arXiv:1509.02971, Sep 2015.

[28] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[29] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv e-prints*, page arXiv:1511.05952, Nov 2015.

[30] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[31] Lucian Bu, Robert Babu, Bart De Schutter, et al. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[32] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents.

[33] Peter Stone and Richard S Sutton. Scaling reinforcement learning toward robocup soccer. In *Icml*, volume 1, pages 537–544. Citeseer, 2001.

[34] Robert H Crites and Andrew G Barto. Elevator group control using multiple reinforcement learning agents. *Machine learning*, 33(2-3):235–262, 1998.

[35] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[36] Sainbayar Sukhbaatar, arthur szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2244–2252. Curran Associates, Inc., 2016.

[37] Amir Alansary, Loic Le Folgoc, Ghislain Vaillant, Ozan Oktay, Yuanwei Li, Wenjia Bai, Jonathan Passerat-Palmbach, Ricardo Guerrero, Konstantinos Kamnitsas, Benjamin Hou, Steven McDonagh, Ben Glocker, Bernhard Kainz, and Daniel Rueckert. Automatic view planning with multi-scale deep reinforcement learning agents. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11070 LNCS, pages 277–285. Springer Verlag, 2018.

[38] Florin C. Ghesu, Bogdan Georgescu, Sasa Grbic, Andreas Maier, Joachim Hornegger, and Dorin Comaniciu. Towards intelligent robust detection of anatomical structures in incomplete volumetric data. *Medical Image Analysis*, 48:203–213, aug 2018.

[39] Florin C Ghesu, Bogdan Georgescu, Tommaso Mansi, Dominik Neumann, Joachim Hornegger, and Dorin Comaniciu. An artificial agent for anatomical landmark detection in medical images. In *International conference on medical image computing and computer-assisted intervention*, pages 229–237. Springer, 2016.

[40] Florin-Cristian Ghesu, Bogdan Georgescu, Yefeng Zheng, Sasa Grbic, Andreas Maier, Joachim Hornegger, and Dorin Comaniciu. Multi-scale deep reinforcement learning for real-time 3d-landmark detection in ct scans. *IEEE transactions on pattern analysis and machine intelligence*, 41(1):176–189, 2017.

[41] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Advances in neural information processing systems*, pages 737–744, 1994.

[42] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015.

[43] Antonio de Marvao, Timothy JW Dawes, Wenzhe Shi, Christopher Minas, Niall G Keenan, Tamara Diamond, Giuliana Durighel, Giovanni Montana, Daniel Rueckert, Stuart A Cook, et al. Population-based studies of myocardial hypertrophy: high resolution cardiovascular magnetic resonance atlases improve statistical power. *Journal of cardiovascular magnetic resonance*, 16(1):16, 2014.

[44] Ozan Oktay, Wenjia Bai, Ricardo Guerrero, Martin Rajchl, Antonio de Marvao, Declan P O'Regan, Stuart A Cook, Mattias P Heinrich, Ben Glocker, and Daniel Rueckert. Stratified decision forests for accurate anatomical landmark localization in cardiac images. *IEEE transactions on medical imaging*, 36(1):332–342, 2016.

[45] Susanne G. Mueller, Michael W. Weiner, Leon J. Thal, Ronald C. Petersen, Clifford Jack, William Jagust, John Q. Trojanowski, Arthur W. Toga, and Laurel Beckett. The alzheimer's disease neuroimaging initiative. *Neuroimaging Clinics of North America*, 15(4):869 – 877, 2005. Alzheimer's Disease: 100 Years of Progress.