UNIVERSITY
OF OSLO

Master's thesis

Exploring Reinforcement Learning for End-Diastolic and End-Systolic Frame Detection

or: How I Learned to Stop Worrying and Love the Bomb

Magnus Dalen Kvalevåg

60 study points

Department of Informatics
The Faculty of Mathematics and Natural Sciences

Abstract

To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take Arms against a Sea of troubles, And by opposing end them: to die, to sleep No more; and by a sleep, to say we end The heart-ache, and the thousand natural shocks That Flesh is heir to? 'Tis a consummation Devoutly to be wished. To die, to sleep, To sleep, perchance to Dream; aye, there's the rub, For in that sleep of death, what dreams may come, When we have shuffled off this mortal coil, Must give us pause. There's the respect That makes Calamity of so long life: For who would bear the Whips and Scorns of time, The Oppressor's wrong, the proud man's Contumely, The pangs of dispised Love, the Law's delay, The insolence of Office, and the spurns That patient merit of th'unworthy takes, When he himself might his Quietus make With a bare Bodkin? Who would Fardels bear, [F: these Fardels] To grunt and sweat under a weary life, But that the dread of something after death, The undiscovered country, from whose bourn No traveller returns, puzzles the will, And makes us rather bear those ills we have, Than fly to others that we know not of? Thus conscience does make cowards of us all, And thus the native hue of Resolution Is sicklied o'er, with the pale cast of Thought, And enterprises of great pitch and moment, [F: pith] With this regard their Currents turn awry, [F: away] And lose the name of Action. Soft you now, The fair Ophelia? Nymph, in thy Orisons Be all my sins remember'd.

Contents

List of Figures

List of Tables

Preface

Chapter 1

Introduction

1.1 Motivation

Cardiovascular disease is the number one cause of death globally, taking an estimated 17.9 million lives each year [6]. It is important to make a timely diagnosis so that patients receive early treatment risk assessment. One standard tool used for diagnosis is cardiac imaging; non-invasive imaging of the heart.

In order to obtain images of the heart, clinicians use tools such as Magnetic Resonance Imaging (MRI), Computerized Tomography (CT) scans, or ultrasound. MRI and CT are less routinely used due to being expensive, having limited availability and a prolonged acquisition time, and using radiation for CT scans. Furthermore, both MRI and CT scans can not be performed if the patient has any metal in their body, such as a pacemaker or metal implants. Ultrasound, on the other hand, is comparatively inexpensive. There even exists handheld devices that can be carried by hand and brought on-site. Ultrasound does have a lower imaging quality compared to, for example, MRI [31], and the images can be difficult to interpret due to ultrasound-specific artifacts. Despite this, it is still preferable in many cases because of the aforementioned reasons.

Many heart measurements depend on two key events in the cardiac cycle: End-Diastole (ED) and End-Systole (ES). Roughly speaking, ED is when the heart is the most relaxed, and ES is when it is the most contracted. Left ventricular ejection fraction is an example of an important measurement that is calculated from ED and ES frames of the cardiac cycle.

A recent study has reported that the average time taken for manually annotating ED and ES frames from visual cues from a video of 1 to 3 heartbeats is 26 seconds, with a standard deviation of ± 11 seconds [26]. Furthermore, because there is not much movement around these frames, the predicted ED and ES frames may differ between different operators. It may even differ for the same operator predicting on the same video at different times. For these reasons, automating ED/ES frame detection is desirable because to reduce time and create a more robust and deterministic result.

Machine learning methods show promising results on several tasks

within medical imaging, as is explored in the following chapter. For ED/ES frame detection, most recent methods revolve around the use of Supervised Deep Learning, a family of methods in which a computer program is shown examples of correct predictions and over time learns to make the correct predictions itself. Reinforcement Learning (RL) is another family of methods that has as of yet not been explored for the problem of ED/ES frame detection. RL is able to outperform humans in complex tasks, such as mastering the board game Go in 2016 [34] or becoming among the 0.2% best players in the world in the video game Starcraft II [37]. However, RL can do more than just play games, and many medical imaging applications also show promising potential [44].

1.2 Goal and Research Question

The goal of this Master's project is to explore the use of RL for automatically detecting the ED and ES frames from an ultrasound video. From a healthcare perspective it is interesting because it may open the doors for better automated tools. Yet, it is arguably more interesting from a research perspective because RL is not an obvious choice for this task. RL is built for tasks that require strategic reasoning, but ED/ES frame detection is fundamentally a classification problem. Of importance to all types of machine learning is formulating the problem in a way that makes it easier to learn for the computer. That is, optimizing the *inductive bias* by incorporating human knowledge into the algorithm itself. Using RL for ED/ES frame detection may open up possibilities of seeing the problem from a new perspective, allowing us the add the right set of inductive bias.

1.3 Limitations of the Work

1.4 Thesis Structure

What are in each chapter...

Chapter 2

Background

2.1 Creating Images of the Heart Using Sound

2.1.1 The Cardiac Cycle

The human heart is situated in the middle compartement of the chest, between the lungs. Blood is used for transporting oxygen and essential nutrients throughout the body and carry metabolic waste such as carbon dioxide to the lungs. The heart is responsible for keeping the blood flowing by acting as a pump.

The heart consists of two halves, the left heart and the right heart, as illustrated in figure 2.1. The left heart pumps newly oxinated blood from the lungs out to the rest of the body and the right heart pumps oxygen-depleted blood back to the lungs. Each side has two chambers, the atrium and the ventricle, for a total of four chambers. The upper chambers, the atria, is where the blood first enters the heart, and the lower chambers, the ventricles is where the blood exits the heart. Each chamber also have valves which are opened and closed during a cardiac cycle to help keep the blood flowing in one direction.

During a cardiac cycle the different chambers are filled at different times. At the start of a new cycle, the left and right ventricles relax and are filled with blood coming from their respective atria. As the ventricles are filled with blood, the pressure increases which causes the valves from the atria to close. After this, the ventricles start contracting, pushing blood out from the heart. As the ventricle pressure decreases and the pressure in the aorta increases, the valve going out of the ventricle is closed. Blood flows into the atria before the cycle starts over. This is illustrated in figure 2.2.

There are multiple ways of finding the ED and ES frames in a cardiac cycle [28]:

1. Finding the frame with the maximum left ventricle volume (for ED) and the frame with the minimum left ventricle volume (for ES).

2. Finding the first frame following the closure of the mitral valve (for ED) and the first frame following the closure of the aortic valve (for ES).

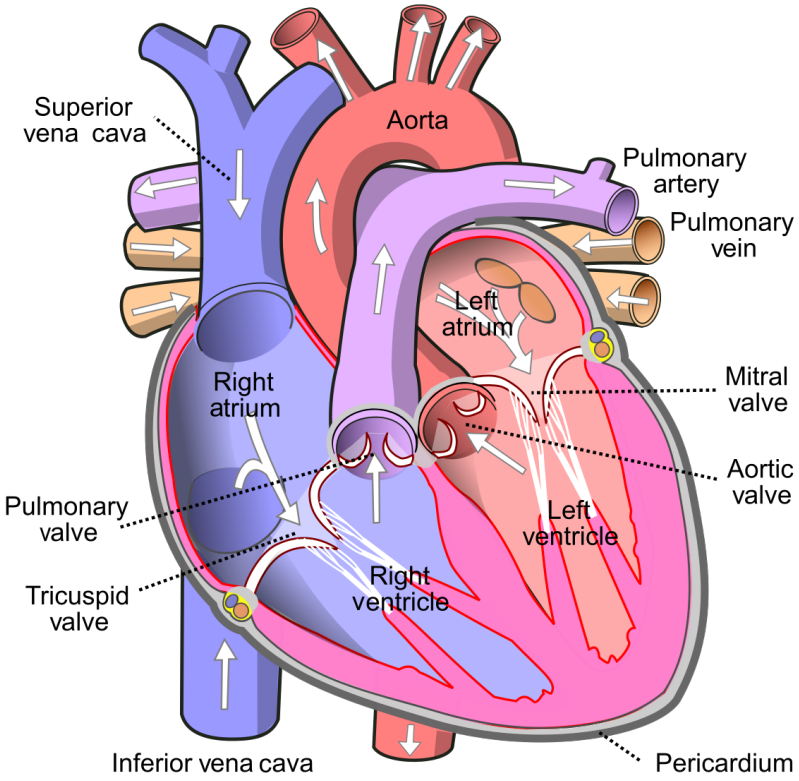


Figure 2.1: An illustration of the heart. The heart has two sides, each side having two chambers. Source: https://en.wikipedia.org/wiki/Atrium_(heart).

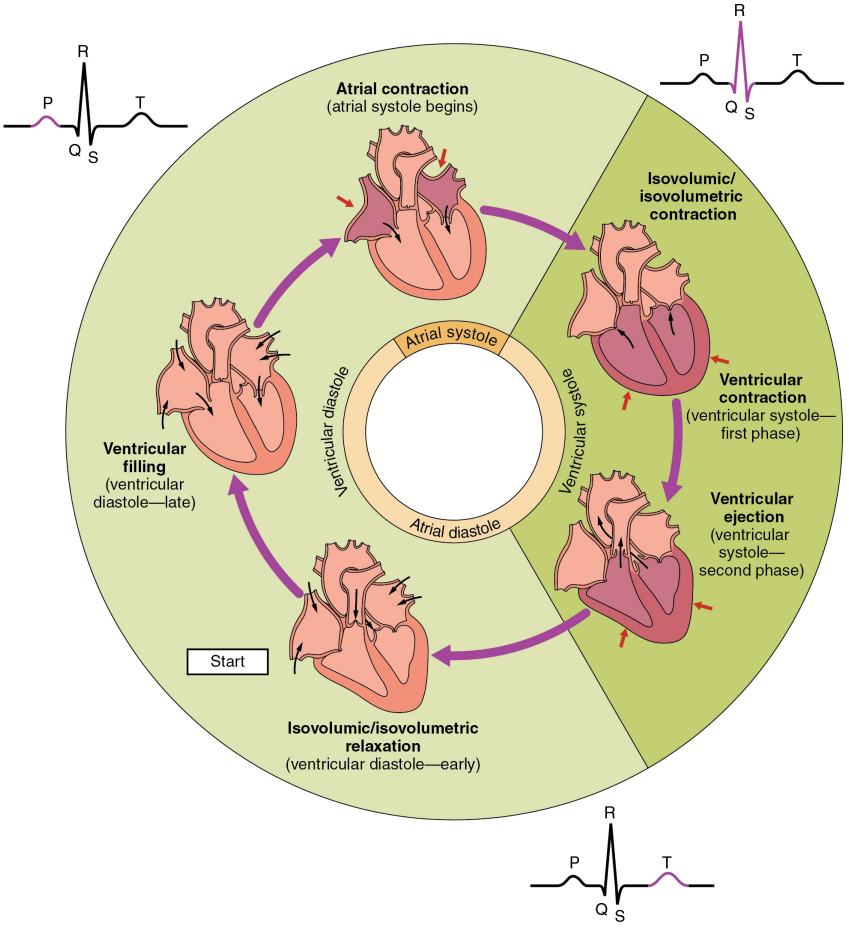


Figure 2.2: The cardiac cycle illustrated with the direction of blood flow and pressure from and into the atria and ventricles. Source: https://en.wikipedia.org/wiki/Heart.

3. Analyzing a simultaneously acquired Electrocardiogram (ECG) signal.

These methods can be visualized in the Wiggers diagram, as seen in figure 2.3, which plots several key events in the cardiac cycle and the corresponding values of various measurements.

Out of these three, using the ECG signal is the least preferable. This is because the methods for detecting the ED and ES frame may become unreliable when given an unconventional ECG signal, such as from patients with cardiomyopathy or regional wall motion abnormalities [28]. Acquiring an ECG signal also requires applying electrodes to the patient, which is not ideal in emergency settings.

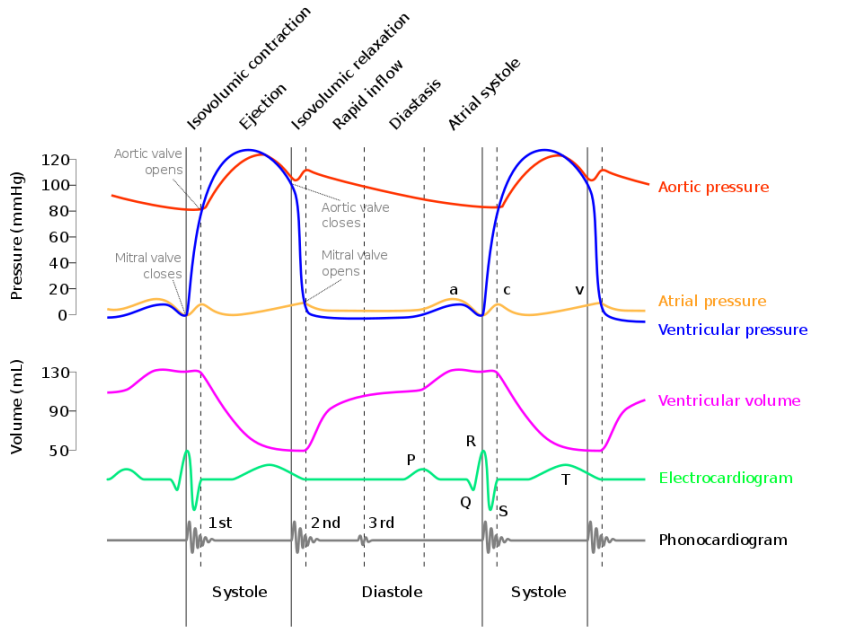


Figure 2.3: The Wiggers diagram desccribes the different phases of the cardiac cycle, as well as what they represent in different measurements. Source https://en.wikipedia.org/wiki/Wiggers_diagram.

2.1.2 What is Sound?

What we as humans perceive as sound are simply vibrations of the particles that surround us. When particles are disturbed, such as what happens to air particles when we clap our hands together, they interact by pushing into each other. As the atoms and molecules that make up the air bump into each other, they also repell, creating an increase in pressure and causing a chain reaction where the pertubation moves from particle to particle, illustrated in figure 2.4. This is called wave propagation. Sound is simply waves of pressure propagating through a medium.

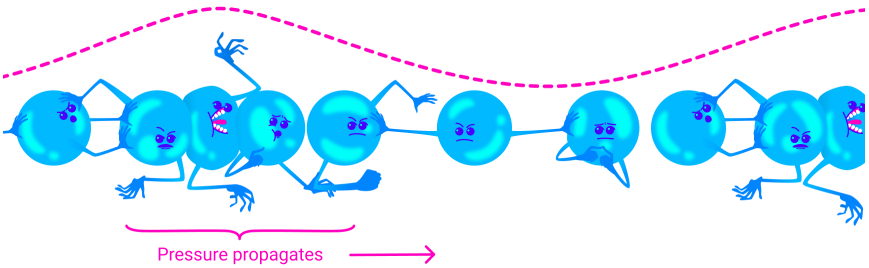


Figure 2.4: A pressure wave moves through a medium by pushing particles in a medium close together. The particles pushes back as the pressure increases, making the pressure field move further on. Warning: this image is just a representation of how particles interact — real particles don't look like this.

Attributes of a Sine Wave

A basic wave has three important attributes: frequency, how fast it vibrates, amplitude, by how much it vibrates, and phase, where in its cycle a wave is at a given time. Our ears have evolved to sense frequency and amplitude, where frequency determines the pitch of a sound and amplitude determines the loudness. Phase can not be sensed by human ears on its own, but can affect the sound in relation with other sound waves.

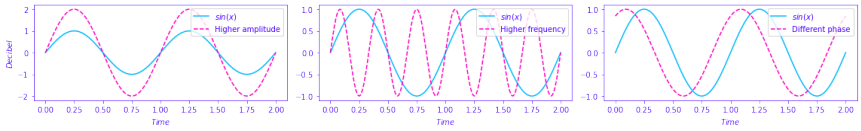


Figure 2.5: The left-most plot shows two basic waves where one has twice the amplitude. The middle plot shows two basic waves where one has a higher frequency. The right-most plot shows two basic waves that have different phases.

A basic wave means a sine wave in this context. Every sound can be represented as a sum of sine waves, and every sound has a unique frequency spectrum. Finding the frequency spectrum is the same as decomposing a sound into its sine waves. We can also take the frequency spectrum and convert it back to its original sound. These operations are called the Fourier Transform and the inverse Fourier Transform, respectively. As seen in figure 2.6, the frequency spectrum after adding two sine waves together is fairly simple as well, but real world sounds often have much more complex frequency spectrums, as many more sine waves are needed to represent it. When a piano and a clarinet play the same note, what we are really saying is that the frequencies with the highest amplitudes are generally the same for both sounds. Musicians speak of overtones — it's the overtones that are different for different instruments playing the same notes. What they are referring to are the additional frequencies that can be seen in the frequency spectrum.

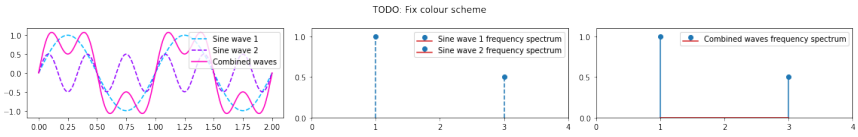


Figure 2.6: Adding two sounds together means that their frequency spectrums are also added together.

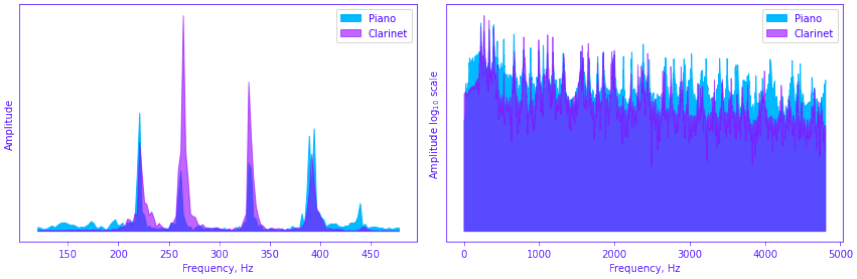


Figure 2.7: It's the overtones that makes two instruments sound different, even while they are playing the same notes. To the left is the frequency spectrum of a piano and a clarinet from 150 to 450 hertz. To the right is the same frequency spectrum from 0 to 5000 hertz, in $\log_{10}$ scale. Both instruments are playing the Am7 chord which consists of four notes. You can see the notes clearly in the left image, all having relatively high amplitudes for both instruments.

Attributes of the Medium

Another important point about sound is the medium in which it travels through. Medium properties such as speed of sound, density, attenuation and non-linearity affect how the wave propagates through it. Speed of sound is how fast a wave propagates through the medium. Because the frequency will stay the same, if the speed of sound is lower then the wavelength will be smaller. Density is how tightly packed the particles are in the medium when at rest. Attenuation is a fancy word for absorption, how much energy the wave loses as it propagates through the medium. Non-linearity is the property where the speed of sound at a point depends on the pressure at that point. For example, in water, waves propagate faster the higher the pressure — pressure for example caused by the wave itself.

An important concept is "acoustic impedance" which is a measure of how much resistance the wave encounters while propagating through the medium, and is a function of the speed of sound and density. When a wave goes from one medium and into another medium that has a different acoustic impedance, a part of the energy is reflected back, the amplitude being reduced for both resulting waves. So when one hears a sound being reflected back from a wall it is because the air that the wave travels through and the wall has different acoustic impedance. Equation 2.1 shows the relationship between acoustic impedance, density and speed of sound, where Z is the acoustic impedance, while ρ and c are the density and

speed of sound of the medium, respectively. Equation 2.2 is the reflection factor and determines how much of the energy is reflected back, where Z_1 is the acoustic impedance of the original medium and Z_2 is the acoustic impedance of the second medium. When Z_1 and Z_2 are equal, no sound is reflected back, which is what we expect — after all we usually don't hear an echo while speaking when there is only air in front of us. However, when there is a difference it doesn't matter which medium has the highest or lowest acoustic impedance — the same amount of energy is reflected either way. The only thing that changes is the sign og the reflection factor, but the magnitude of the wave stays the same whether $Z_1 > Z_2$ or $Z_1 < Z_2$. This means that the amount of echo would be the same if you were talking in the second medium, into the first one, or the other way around.

$$Z = \rho c \tag{2.1}$$

$$RF = \frac{Z_2 - Z_1}{Z_2 + Z_1} \tag{2.2}$$

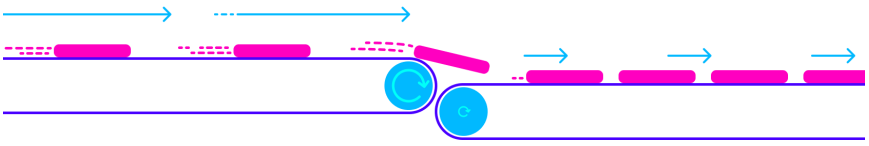


Figure 2.8: Even though the rate of packages per second stays the same, the distance between each package decreases when arriving on a slower conveyor belt. This is analogous to a sound wave propagating through a medium where the speed of sound changes. Even though the frequency is the same, the wavelength (the length between each top) decreases when it encounters a lower speed of sound.

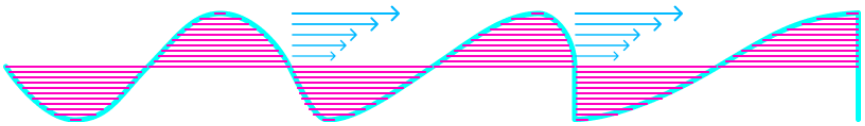


Figure 2.9: In a medium with nonlinearity the higher-pressure parts of a wave propagates faster than lower-pressure parts. Over time, the higher-pressure parts will "catch up" to the lower-pressure parts, and what started as a sine wave will start to resemble a sawtooth wave.

2.1.3 Echocardiography

Light is a signal that does not penetrate very far into the body, which is why we are unable to simply gaze into each other's hearts. We could, however, imagine a universe where the light penetrates too much, giving off no reflections at all. In this universe we would not be able to see the heart either — in fact we would not be able to see any body at all! To be able to

look *inside* something based on reflections alone requires a sweetspot where the signal is able to penetrate tissue with enough energy while at the same time being reflected back with enough energy so that we can measure it. Arguably, we are quite lucky with our universe, at least in terms of cardiac imaging, because sound is such a signal.

Table 2.1: Values of the acoustic wave velocity c and acoustic impedance Z of some substances from [**suetens_fundamentals_2017**].

Substance	c (m/s)	Z=ρ c ($10^{6\text{kg}}/\text{m}^{2\text{s}}$)
Air (25)	346	0.000410
Fat	1450	1.38
Water (25)	1493	1.48
Soft tissue	1530	1.63
Liver	1550	1.64
Blood (37)	1570	1.67
Bone	4000	3.8 to 7.4
Aluminium	6320	17.0

- Why we use ultrasound gel explained by the above table 2.1.

How can we use sound reflections to create images? If we send out a sound signal and measure the time it takes for a reflection to come back we can get information about the relative distance to various reflectors in the medium from the sound source. If we know the speed of sound, and assume that the speed of sound is homogeneous in the medium, then we can approximate the distance that the wave has travelled by multiplying the delay between sending the sound signal and receiving back an echo by the speed of sound (equation 2.3). This makes the assumption that waves always travel in straight lines, which is not always true, but the effect is often neglible in medical ultrasound usecases.

$$\text{distance} = \text{delay} \times c \tag{2.3}$$

Likewise, if want to know what the reflected signal is for a given distance away from the sender and receiver we can calculate the corresponding delay of a signal traveling that distance and back by dividing the total distance by the speed of sound (equation 2.4). When we know the corresponding delay we can simply look up its value in the signal. If we repeat this for every point in an area that we want to image we would end up with an image.

$$\text{delay} = \frac{\text{distance}}{c} \tag{2.4}$$

When we only have a single receiver that measures the reflected sound waves we can not know the exact location of a given reflector, only the distance. By utilizing more receivers spread over some area we get more information about where the signal orginated from as the distance will match across receivers for an actual reflector object.

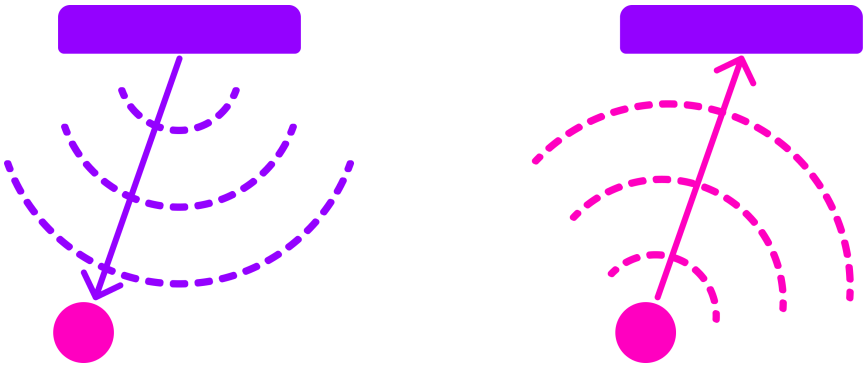


Figure 2.10: By measuring the time between sending a signal and receiving it back from a reflector we can approximate how far away the reflector is — given that we know the approximate speed of sound.

By utilizing multiple sender elements as well, tha can send sound waves independently of each other we are able to shape the wavefront as we wish. This let's us for example focus the energy of the sound wave in a specific area, or shape the wave front to be planar. The Huygens-Fresnel principle states that every point of a wavefront is the source of a new spherical wavefront. We can simulate this behavior by imagining a desired wavefront passing through the sender elements, activating each element at the moment the wave hits it. Each sender element on its own creates a spherical wavefront, but together they make up the desired imagined wavefront. An example of this has been visualized in figure 2.11. Time delays are to sound waves like a lens is to a magnifying glass.

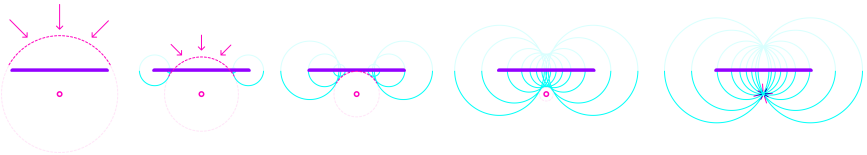


Figure 2.11: Because of the Huygens-Fresnel principle, we can create a desired wavefront by creating spherical waves at each sender element the moment the imagined wavefront would hit it. The dashed, pink curve represents the imagined desired wavefront as it approaches the sender elements marked by the purple rectangle. Each sender element is activated the moment the imagined wavefront passes through it, creating new spherical waves, represented by the cyan semi circles. The generated spherical waves converge on the same point that the imagined wavefront would have converged.

In reality, an ultrasound probe consists of many elements which act both as transmitters and as receivers. This is done using a piezoelectric material, a material that is both able to produce vibrations if given electric current, and produce electric current when exposed to vibrations. With a transducer, we can apply an electric current to each element independently to create sound waves with given wave front characteristics, and read off

the electric current generated by reflected pressure waves.

There are multiple modes of ultrasound imaging. The two most important modes to this project is B-mode imaging and M-mode imaging.

In B-mode (as in "Brightness"-mode) imaging an image is created by visualizing the amplitude of the reflected signal as the brightness for a given point. This imaging mode often sends out individual focused transmits in multiple directions, creating a sector scan — a fan-like image, as seen in [TODO: sector scan image with lines indicating different angles]. Another method is to transmit unfocused plane waves in multiple directions, each transmit creating an unfocused image of the scatterers, but together they form an image of equal (TODO: equal or near-equal?) quality to focused-transmit scans. B-mode images provides images of the whole area of interest, but because they require multiple transmits to do so they also take longer to acquire, as we have to fire each transmit after the other. In extreme cases this could pose a problem given that the heart is an organ that moves quite rapidly and if we are transmitting too slow then the heart may have a noticably different phase on one side of the sector scan compared to the other. This is not a big problem for 2D-images as even multiple transmits can be made and received back in a short period of time, but it does have consequences for the temporal resolution if we are creating many images in a video.

In M-mode (as in "Motion"-mode) imaging only one direction is imaged over time, instead of a whole sector. This means that it only requires one transmit per frame meaning that it has a higher temporal resolution compared to B-mode imaging, at the cost of only focusing on a small part of the heart. M-mode imaging lets us see the motion of a focused part of the heart in single image as the columns can be concatenated into an image where the y-axis represent the amplitudes at different depths and the x-axis represent time. [TODO: M-mode image example with lines with B-mode image indicating which line it is looking at.].

2.2 Data Processing Section (NAME TBD)

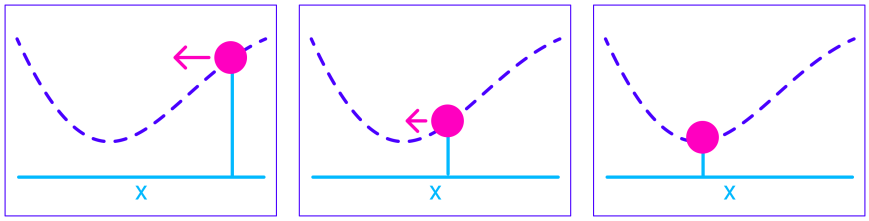
2.2.1 Deep Learning

Gradient Descent

The most significant deep learning innovations have all used a technique called gradient descent.

Gradient descent is based on calculus. It takes advantage of the fact that even if we don't know the true nature of some function, given that it is differentiable, we can calculate its slope at a given point. This is called the gradient and it gives us information about how to update its parameters in order to maximize or minimize the result. This is easily visualized when we have a differentiable function that takes a single parameter x, as in figure 2.2.1. Even though we may now know the true shape of the function, as represented by the dashed line, we can calculate its slope. If we nudge x in the opposite direction of the slope, i.e. reduce x if the slope

tends upwards and vice-versa, and repeat this multiple times, then we will eventually reach a minimum where the slope becomes 0. This iterative process of calculating the gradient at a point and updating the parameters in the opposite direction is what's called gradient descent.



Gradient descent also scales to functions that take multiple parameters, so instead of just taking x it may take an arbitrary number of parameters. This lets us optimize complex models that take a lot of parameters. One example could be that of a model that performs some operation on an image. If we want to process each pixel individually in some parameterizable way then the number of parameters is at least equal to the number of pixels in the image. If the image is 100-by-100 pixels big then the model would take at least 10000 parameters. It is no longer possible to visualize this high-dimensional parameter space as we did in 2.2.1, but the principles still hold, and gradient descent still works.

We may want to optimize some parameters working on a set of images, for example when training a model to classify pictures as those of cats or of dogs. Because of either memory or computational constraints, there may be too many pictures in the dataset for the model to try to optimize for at once. In this case it is common to apply gradient descent on just a subset of the full dataset at once, chosen randomly at each iteration. This is called Stochastic Gradient Descent (SGD) and it is often better at generalizing on the dataset than regular gradient descent.

The function that we optimize using SGD consists of two parts: a model and a loss function. The job of the model is to perform the task at hand, and the job of the loss function is to enable the model to be optimized using SGD. As long as both the model and the loss function is differentiable and convex then we can optimize it using SGD. Not all models and not all loss functions are equally good, however. Some models may better represent the problem at hand than others and some loss functions may produce gradients that are easier to optimize for than others. Of great importance is to instill what's called inductive bias into the model — that is, implicit knowledge about the task at hand. How to do this is still an ongoing research topic, but some of the most popular approaches are explored in the next section.

Deep Neural Networks

Activation functions Fully connected layers Convolutional layers Batch normalization (++ more used in Mobilenet?)

Optimization Process

- SGD
- Optimizers like ADAM
- Overfitting and regularizers.

Supervised and Unsupervised Learning

Basically two families of loss functions

2.2.2 Reinforcement Learning

RL allows an agent to learn a strategy, called a *policy*, that maximizes the total reward received through interacting with an environment. RL can leverage time in a way that neither supervised nor unsupervised learning is able to because it can reason about future decisions. An RL agent can make a decision now that has no immediate benefit, but that will lead to a better result in the future.

At the core of RL are Markov Decision Processes (MDP) [35], which can be described using four elements:

- The state space S
- The action space A
- The transition function $P(s_{t+1}|s_t, a_t)$
- The reward function $R(s_t, a_t)$

An RL agent is faced with a sequence of decisions. At each step it is presented with the current state $s_t \in S$ of the environment, and must take an action $a_t \in A$. In an episodic task, the agent's goal is to maximize the total amount of reward r it receives during its lifetime, called an episode. The environment may change after the agent takes an action in a given state, and how it changes, i.e. what the next state s_{t+1} will be, is determined by the transition function $P(s_{t+1}|s_t, a_t)$. How much reward the agent receives after taking an action in a given state is determined by the reward function $R(s_t, a_t)$. The goal of RL is to find a policy π, a strategy that, if followed, will yield the most amount of total reward during the lifetime of the agent. In practice, the policy is simply a function that takes in the current state s_t and returns the probability of taking an action at: $\pi(a|s) \in [0, 1]$.

The agent's goal is not to maximize the immediate reward r but rather the expected return. The return is denoted as G_t, and is in its simplest form a sum of all the future rewards:

$$G_t = r_{t+1} + r_{t+1} + r_{t+2} + \ldots + r_T$$

where T marks the timestep where the episode ends. However, some tasks are not episodic, which means they can, in theory, run forever. For

this reason, we apply discounting to the return, giving greater weight to more immediate rewards and less weight to rewards in the far future:

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_t + \gamma G_{t+1} \end{aligned}$$

where γ is the discounting factor. Discounting ensures that the return, which we are trying to maximize, can not be infinite, even when, in theory, the agent could go on forever.

To select an appropriate next action, the policy needs to know the value of states and actions. For this we could use the state value function $V_\pi(S_t)$ which estimates the expected return G_t of being in state s_t, while following the policy π. Alternatively, we could use the state-action value function $Q_\pi(s_t, a_t)$ which estimates the expected return of taking action a_t in state s_t, while following the policy π. Both value functions depend on the policy being followed because the policy decides what actions to take in the future, which again has consequences for what rewards the agent expects to receive. The "learning" part of RL could be considered to be updating a value function towards the "optimal value function", defined as the value function that uses the optimal policy when estimating returns. The optimal policy π^* is one *(of the possibly many policies)* that yield the maximum amount of total reward if followed.

One algorithm for updating the state value function is called Temporal Difference learning (TD). In TD, the state value function $V(s_t)$ is updated after every step, by comparing the value it expected to see, with a value that takes the newly observed reward r_{t+1} into consideration:

$$V(s_t) \leftarrow V(s_t) + \alpha[(r_{t+1} + \gamma V(s_{t+1})) - V(s_t)]$$

$(r_{t+1} + \gamma V(s_{t+1}))$ is called the TD-target, and because it incorporates the actual observed reward r_{t+1}, it can be considered as a more up-to-date version of the state value function. $(r_{t+1} + \gamma V(s_{t+1})) - V(s_t)$ is called the TD-error. The lower the TD-error is, the better the RL agent is able to reason the value of states, and as such, we want to minimize it. We do this by updating the state value by nudging it slightly towards the TD-target. How far it is nudged at each update is determined by α.

To be able to use $V(s)$ for making a decision, the agent needs knowledge about the transition function. This is because it needs to know what the next state will be in order to select the best action to take. $Q(s, a)$ does not need knowledge about the transition function because it learns the value of taking an action in a state directly. TD can be modified to use the state-action value function instead of the value function, in which case it is called Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[(r_{t+1} + \gamma max_a Q(s_{t+1}, a)) - Q(s_t, a_t)]$$

Here the target, the Q-target, is defined as the immediate reward of taking action a_t, plus the discounted value of taking the best action in the following state.

In TD-learning, as the agent explores the environment and encounters new states, it has to store those states and their associated values. The same is true for Q-learning, but it also has to take state-action pairs into account, meaning that it has to store up to a number of $\|S\| \times \|A\|$ entries. That is fine when the state space and the action space are small but becomes infeasible when they are too big.

The described way of storing and updating the values is called tabular methods because we treat the states, or state-action pairs, as entries in a table. Tabular methods break down when the state space or the action space becomes very large or even continuous. Creating RL algorithms that can handle very large or continuous action spaces is challenging [44]. However, there exist methods that can scale RL to handle very large or continuous state spaces.

Deep Reinforcement Learning

A modified Q-learning algorithm has been shown to be able to play Atari games simply by looking at the raw pixel values[30]. The state space thus consists of the pixel values of the current game screen. A simple Atari game has $210 \times 160 = 33600$ pixels, and each pixel can be one of 128 colors [30]. In theory there are $128^{33600} \approx 10^{70803}$ different states. If a computer were able to process $1\,000\,000\,000$ such states every second, it would still take more than 10^{70785} years to process all of them. In practice, the vast majority of pixel permutations are not used, so we could ignore them, but the number of possible states would still be too high to explore exhaustively.

The values of even such a large state space can be represented in much less data without losing much relevant information. This can be done through function approximation [35], where instead of storing and updating the value estimates in a table, such as with tabular methods, they are approximated using a neural network. This allows the agent to generalize state value or state-action value functions to new not-before-seen states.

A lot of today's research into RL goes into scaling it up to a larger state space. Methods that scale RL by modifying the Q-learning algorithm are called "action-value methods", but they are not the only ones to do so. Policy gradient is another popular set of methods that is able to learn a parameterized policy directly, without consulting a value function [35]. TODO: Quick explaination of Policy-gradient methods

Deep Q-Network

The modified Q-learning algorithm was termed Deep Q-Network [30] (DQN) for its ability to take advantage of recent deep learning advances and deep neural networks.

The original DQN algorithm takes the raw pixel values from an Atari game as input, followed by three convolutional layers and two fully connected layers. The final fully connected layer outputs one value for each possible action, approximating the expected value of taking each action given the state, i.e., $Q(s, a)$. An ϵ -greedy policy then chooses either the action with the highest approximated value with probability $1 - \epsilon$ or a random action with probability ϵ.

The authors showed how the network is able to reduce the state space by applying a technique called "t-SNE" to the DQNs' internal state representation. t-SNE is an unsupervised learning algorithm that maps high-dimensional data to points in a 2D or 3D map [27]. As expected, the t-SNE algorithm tends to map the DQN representation of perceptually similar states to nearby points. Interestingly, it also maps representations that are perceptually dissimilar, but that are close in terms of expected rewards, to nearby points. This indicates that the network is able to learn a higher-level, but lower-dimensional, representation of the states in terms of expected reward. This is visualized in figure 2.12.

TODO:Rewrite this caption as it is a direct copy

Using function approximation does have its problems. Naively training the network by inputting state and returns pairs as they are generated by the agent can result in the algorithm becoming unstable. There is a strong correlation between consecutive samples, which leads to variance in the network updates. If a neural network receives a batch of very similar input, it might overwrite previously learned knowledge. Furthermore, an update that increases $Q(s, a)$ often also increases $Q(s + 1, a)$ and therefore also increases the target y_j , possibly leading to oscillations or divergence of the policy. These problems are mitigated by using experience replay and by using a separate network for generating the targets y_j in the Q-learning update.

In experience replay, the agent's experiences over multiple episodes are stored in a data set called the replay memory. Each experience item is a tuple consisting of the previous state, selected action, returned reward, and new state: (s_t, a_t, r_t, s_{t+1}). During training, randomly sampled batches from the replay memory are used to train the Q-network.

Using a separate network for generating the targets y_j in the Q-learning update adds a delay between the time an update to Q is made and the time it affects the targets y_j, making the algorithm more stable and reducing the chance of oscillations or divergence.

Double Deep Q-Network

Several improvements have been made to DQN over the years. Q-learning has been shown to produce overly optimistic action values as a result of using the maximum action value as approximation for the maximum expected action value[17]. Double Q-learning attempts to reduce this overestimation by decomposing the target into an action selector and an action value estimator. The regular Q-learning target is written as:

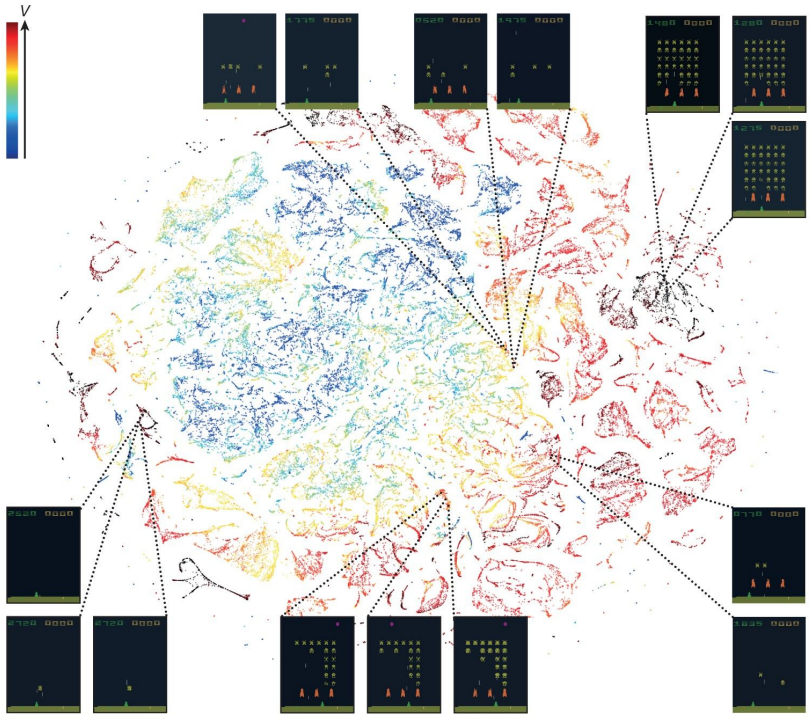


Figure 2.12: From [30]: "Two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced while playing Space Invaders. The plot was generated by letting the DQN agent play for 2 h of real game time and running the t-SNE algorithm on the last hidden layer representations assigned by DQN to each experienced game state. The points are coloured according to the state values (V, maximum expected reward of a state) predicted by DQN for the corresponding game states (ranging from dark red (highest V) to dark blue (lowest V)). The screenshots corresponding to a selected number of points are shown. The DQN agent predicts high state values for both full (top right screenshots) and nearly complete screens (bottom left screenshots) because it has learned that completing a screen leads to a new screen full of enemy ships. Partially completed screens (bottom screenshots) are assigned lower state values because less immediate reward is available. The screens shown on the bottom right and top left and middle are less perceptually similar than the other examples but are still mapped to nearby representations and similar values because the orange bunkers do not carry great significance near the end of a level. With permission from Square Enix Limited."

$$r_{t+1} + \gamma max_a Q(s_{t+1}, a)$$

This can be rewritten as:

$$r_{t+1} + \gamma Q^A(s_{t+1}, argmax_a Q^B(s_{t+1}, a)) \tag{2.5}$$

Where Q^A acts as an action value estimator and Q^B acts as an action selector. If $Q^A = Q^B$ then this is just the regular Q-learning target. If we only update the action selector at each update, and randomly choose which of the two Q-functions should be used as the action selector at each update, then the overestimation is reduced. This also applies to DQN, and it has been shown that using a double DQN results in better policies than using a regular DQN[18].

Prioritized Replay

Using experience replay, an agent isn't forced to process transitions in the exact order that they are experienced. However, because we are sampling the transitions uniformly from the replay memory, all transitions are given equal priority. We might benefit from prioritizing transitions that have a high TD-error magnitude, which acts as a proxy-measure of how "surprising" a transition is to the agent[33].

Prioritizing experience by the magnitude of the TD-error may introduce a lack of diversity. One of the reasons for this is that an experience that initially had a low TD-error, but that later becomes large as the network is trained, will continue to be down prioritized because the TD-error is only updated when the transition is revisited — and because of its low prioritization, the probability that it will be visited again soon is low. To overcome this challenge, a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling is introduced.

Another problem with prioritized experience replay is that DQN optimizes for minimizing the expected TD-error squared, with respect to the network parameters θ, assuming that the samples in the replay buffer corresponds to the same distribution as seen while exploring. Prioritized experience replay breaks this assumption, introducing a bias in the calculated gradient. This is fixed by using importance sampling, such that the less-sampled experiences are compensated for in the gradient. As the unbiased nature of the updates is most important near convergence at the end of training, the importance sampling is gradually added towards the end of training, with less importance sampling included at the start of training.

Prioritized replay is found to speed up an agent's ability to learn by a factor of 2.

Dual Deep Q-Network

In the dueling architecture, or Dual DQN, the network that approximates the Q-function is split into two parts: one for estimating the value of the

current state, and one for measuring the so-called advantage of taking an action in this state[39]. The combination of the state-value estimate and the advantage yields the Q values:

$$Q(s,a) = V(s) + A(s,a) \tag{2.6}$$

But because the state value function $V(s)$ can be expressed in terms of the state-action value function $Q(s,a)$ by taking the mean of $Q(s,a)$ over all actions, then it means that the mean of the advantage function $A(s,a)$ over all actions equals zero. This is not necessarily the case because the networks are simply approximations. To fix this the authors also subtract the mean advantage from the equation. This change loses the original semantics of $V(s)$ and $A(s,a)$, but results in a more stable algorithm.

$$Q(s,a) = V(s) + A(s,a) - \frac{\sum_a A(s,a)}{N_{actions}} \tag{2.7}$$

The dueling architecture lets the network train the state-value function and the advantage function separately.

Multi-Step Learning

We look only one step ahead when constructing the target in the Q-learning update, but this isn't a requirement. We could extend it to look N steps ahead if we wanted to, in which it is called N-step learning, or multi-step learning[35].

To use multi-step learning we must look at N consecutive experiences for every update, and sum the appropriately discounted rewards and add it to an appropriately discounted value estimation of the final state in the sequence. The N-step target for a given state s_t is given as:

$$\sum_{k=0}^{N-1} \gamma^k r_{t+k+1} + \gamma^N max_a(Q(s_{t+N}, a)) \tag{2.8}$$

If we set N to be 1, then the algorithm would be equal to the regular Q-learning algorithm. As we increase N, the algorithm would become more and more similar to the Monte Carlo method, which looks all the way until the agent hits a terminal state.

$$r_{t+1} + \gamma max_a Q(s_{t+1}, a) = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n max_a(Q(s_{t+n}, a)), \text{ iff n=1} \tag{2.9}$$

The best choice of N usually lies somewhere between 1 and the length of an episode. This is because bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. Another intuition for why it is better is that when we look further ahead into the future we depend less on our own estimates of the future.

Distributional Reinforcement Learning

The Q-function is an approximation of the *expected* returns, but it is also possible to approximate the *distribution* of returns instead[4]. It makes sense to think about the returns as a distribution, even when the environment has deterministic rewards, because stochasticity is still introduced while training through various sources. Firstly, state aliasing, the conflation of two or more states into one representation, may cause different amounts of rewards to be observed even though the agent "sees" the same state. Secondly, because of bootstrapping, target values are nonstationary while training, and the return will seem to take on different values over time. Lastly, because we are approximating the Q-function, approximation errors will make the returns seem stochastic.

Approximating the distribution of returns instead of the expected returns results in more stable learning targets.

Noisy Deep Q-Network

Exploration of the environment is often enabled by using an ϵ -greedy policy, where ϵ is gradually reduced. For particularly hard problems, like the Atari game "Montezuma's Revenge", this technique become insufficient for exploration[5]. ϵ -greedy explores with a fixed probability that is the same for every state. An alternative could be to let the network itself learn when it should explore, and for what states.

NoisyNet-DQN does this by applying learnable parameterized noise to the value network parameters[11]. This does not only enable it to change the amount of exploration itself, alleviating the need for hyper parameter tuning, but also to apply different amounts of exploration to different states.

Rainbow Deep Q-Network

Many of the improvements that has been made to DQN may be complementary and could be combined into a single algorithm. The Rainbow[19] algorithm combines six such extensions:

1. Double DQN[18]
2. Prioritized replay[33]
3. Dual DQN[39]
4. Multi-step learning[35]
5. Distributional RL[4]
6. Noisy DQN[11]

The authors are able to show that the combined algorithm performs much better than each extension alone, in terms of both learning speed and overall performance.

They also performed an ablation study on the Rainbow algorithm to see how much each extension contributes to its overall performance. The study conclude that prioritized replay and multi-step learning contribute the most to the overall performance, as removing them from the algorithm reduces its performance the most. Distributional Q-learning ranked directly below, followed by Noisy DQN, and then Dual DQN. The benefit of using a Double DQN is not apparent, as removing it from the algorithm does not reduce its performance.

TODO: Rephrase

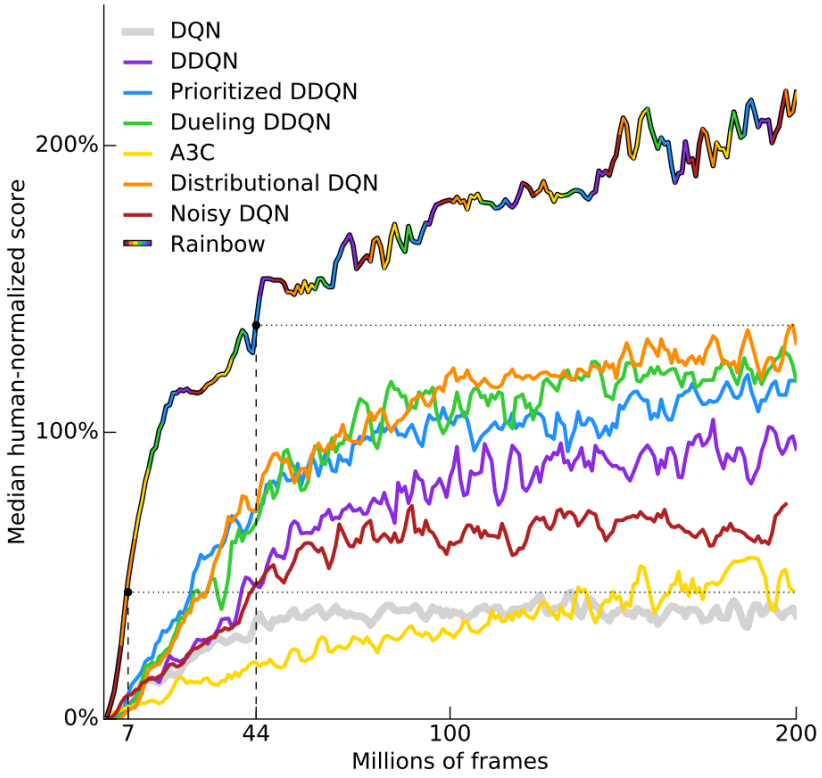


Figure 2.13: From [19], figure 1: Median human-normalized performance across 57 Atari games. We compare our integrated agent (rainbowcolored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

TODO: Rephrase

TODO: Comments from Krissy

2.3 Related Work (State-of-the-art Section (TBD))

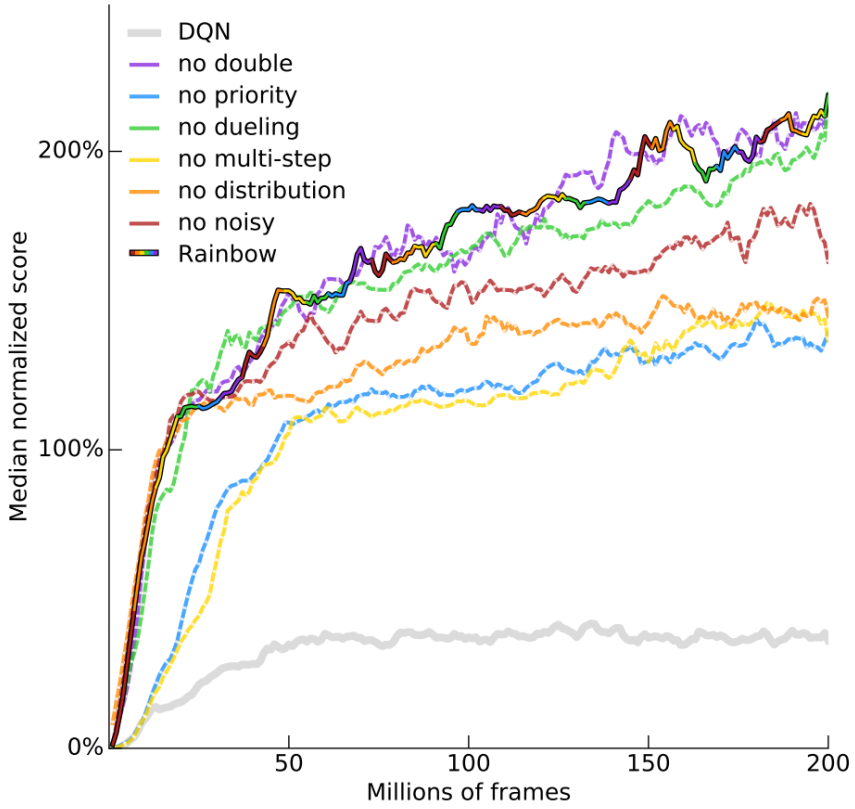


Figure 2.14: From [19], figure 3: Median human-normalized performance across 57 Atari games, as a function of time. We compare our integrated agent (rainbow-colored) to DQN (gray) and to six different ablations (dashed lines). Curves are smoothed with a moving average over 5 points.

2.3.1 ED-/ES-Detection

TODO: This is copied from the essay.

- ☐ Read through to confirm that it still makes sense
- ☐ Re-add references
- ☐ Double-check for plagiarization

One early attempt for detecting the ED and ES frames took advantage of the rapid mitral valve opening during early diastole [23]. By measuring the mean intensity variation over time in a small region of interest, one could capture the mitral valve opening and define the frame corresponding to peak intensity as ES. This signal was in some cases disturbed by early longitudinal motion of the heart, which led to falsely labeling frames as ES. The authors introduced another method in the same paper that took advantage of the left ventricle deformation during the cardiac cycle. With this method, ES was defined as the frame which had the lowest correlation with the ED frame. Because of little movement around systole, the correlation curve would flatten out, making the predictions more uncertain. The best results were achieved when using a combination of both methods. A small time window was selected around ES using the correlation method, and the mean intensity variation method was used to determine the final ES frame prediction.

The main disadvantage with this approach is that it is only semi-automated. The first method requires the clinician to select multiple landmarks in order to define the correct region of interest around the mitral valve, and the second method assumed that the ED frame has already been found in order to compute the correlation between it and the other frames.

It has become more common to apply end-to-end Machine Learning (ML) for fully automating tasks like this in recent times. ML is the study and development of algorithms that can learn from experience. If given enough examples, ML can approximate any mapping between input and output data [43]. ML is generally divided into three categories:

1. Supervised learning, where the algorithm learns a mapping between input and ground truth labels.
2. Unsupervised learning, where the algorithm learns to recognize patterns in the input data without any explicit ground truth labels.
3. Reinforcement learning, where the algorithm learns a strategy for solving a sequential task, given a reward signal.

Gifani et al. (2010) employed manifold learning, an unsupervised learning algorithm that is used to map high-dimensional data onto a lower-dimensional manifold. Manifold learning tries to ensure that points that are similar in the high-dimensional space are projected close together in the low-dimensional space. The authors reduced the dimensionality of each frame down to two dimensions, followed by analyzing the density

between the projected points to determine the ED and ES frames [15]. This method is based on the fact that there is no prominent change in ventricular volume during the three cardiac phases: isovolumetric contraction, isovolumetric relaxation, and reduced filling. Frames that lay close together, i.e., frames that lay in dense regions, are considered to be part of one of these three phases. The projected points move very little in these dense regions, and the three points that had the least movement were selected as representative of three phases. The ED and ES frames were then found by finding the pair of said frames with the minimum correlation. The manifold learning algorithm that the authors used is called Locally Linear Embedding (LLE). In a follow-up paper, they used Isomap instead [16], which yielded better results. When using Isomap, they defined the ED and ES frames as the projected points with the greatest distance between them.

Non-negative Matrix Factorization (NMF) is another unsupervised learning method that has been employed to reduce the dimensionality of ultrasound videos [41]. In this work, rank-2 NMF was used to generate two end-members from a cardiac ultrasound video. The end-members turn out to be quite similar to the ED and ES frames, and the end-member coefficient peaks can be used to find ED and ES. NMF was found to give predictions with less error than LLE and Isomap manifold learning.

	ES Difference[1]		ED Difference[2]	
	mean	variance	mean	variance
NMF	0.93	1.72	0.93	1.17
LLE	2.76	5.92	2.22	8.79
ISOMAP	1.93	3.94	1.90	8.75

1. Difference between the frame number of extracted ES and Ground truth ES.
2. Difference between the frame number of extracted ED and Ground truth ED.

Figure 2.15: Comparison between NMF, LLE and ISOMAP results for all 99 cases in the apical 4 view, taken from [41].

Other methods use either image segmentation or speckle tracking to track the changes to the left ventricle volume, taking advantage of the fact that it is most expanded during ED and most contracted during ES [3] [8] [1]. However, these methods are prone to significant errors due to noise inherent in cardiac ultrasound or discontinuous edges.

The most successful approaches to the task of ED and ES frame detection so far have been to use supervised learning methods. A 2D video consists of a sequence of 2D images and thus has two spatial dimensions and one temporal dimension. Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) are both supervised learning models that can extract spatial and temporal features, respectively. A basic CNN consists of one or more convolutional layers that each consists of a set of filters. The filters act as pattern-matchers and are applied to every part of the input image, and each subsequent convolutional layer can capture more high-level features of the image. A basic RNN consists of one or more processing units that are repeatedly applied to the items in the input sequence and can build up a memory of previous items.

Due to the increase in computing power in the form of GPUs, it is possible to train CNNs with many convolutional layers, or RNNs with stacked processing units, creating deep networks that can learn increasingly complex image features. This architecture gives rise to the term "Deep Learning" and is what has made some supervised learning methods so successful.

A CNN and an RNN were combined to do spatial and temporal feature extraction to detect the ED and ES frames by Kong et al. in 2016 [24]. The combined network was trained on cardiac MRI data, and it used a Zeiler-Fergus model [42] for the CNN and an LSTM [20] for the RNN. The problem was treated as a regression problem for a monotonically decreasing function during diastole and monotonically increasing during systole. Thus, the function being regressed is a latent space representation of the left ventricle volume as it expands and contracts, and the ED and ES frames can be found by finding the maximum peaks and minimum valleys of the DL model's output. This approach was later improved by swapping out the CNN with a ResNet [9], and then again by swapping it out for a DenseNet [36], while different choices for the RNN did not significantly improve the performance of the model.

Instead of treating the model's output as a function regression, it has also been treated as a binary classification of either ED or ES [10]. The authors of this paper argued that treating it as a regression problem forced the model to learn a function that was not present in the data because the regressed function does not represent the actual left ventricle volume. They argued further that, in some cases of pathology, such as in the event of post-systolic contraction, the volume might not be smallest at the time of ES. Their model also uses a 3D CNN with a sliding window that does both spatial and temporal feature extraction on the data before being passed into an LSTM. A similar architecture has been used for finding the ED frames in cardiac spectral Doppler imaging [22]. Spectral Doppler is a technique that outputs a spectrogram representing the blood velocity over time. It thus has one spatial dimension and one temporal dimension. A CNN with a sliding window was used to extract spatial and temporal features, followed by a bidirectional GRU that further connects said features temporally. For each patch in the sliding window, the model predicts whether it contains an ED frame and which frame in the patch it is.

The latest model iteration in this sequence of papers reverts back to a regression-based approach, countering the anti-regression argument by stating that a simple binary classification ignores high-level spatial and temporally related markers [26]. The authors explore multiple different architectures, but a ResNet50 followed by two layers of LSTM yielded the best results and is the current state-of-the-art. Lastly, they also provided a method for benchmarking different architectures by providing their patient dataset and models to the public and including performance reports on an independent external dataset.

RL has not yet been applied to the problem of ED and ES detection, even though it has seen a similar increase in capabilities as supervising learning has in the last decade. RL has produced even better results than

supervised learning methods for many tasks, including medical imaging tasks [44]. The next section will introduce RL, and it is followed by some examples of how it has been applied to medical imaging.

2.3.2 Reinforcement Learning in Medical Imaging

RL has seen many medical imaging applications in the last decade, especially in the last five years [44]. One of the main challenges of applying RL is formulating the problem to fit into the RL framework of states, actions, and transition and reward function. Out of these four elements, the reward function is usually the most difficult to get right.

One way to formulate the problem is as a search through parameter space. Here, the actions are defined as taking a single step along one of the parameter dimensions. The reward function could be how much closer the agent got to the optimal solution after taking a step (the state and transition function definitions vary depending on the problem). This formulation has been applied to many different medical imaging problems, including that of landmark detection.

The goal of landmark detection is to find a point in an image that represents a medical landmark. In a 2D image, it can thus be defined by the parameters $[x, y]$, where the goal is to find the x and y values that correspond to a given landmark. The state presented to the RL agent will thus be defined in terms of these parameters, such as a smaller section of the image centered around the current point. The action space is defined as a change to the parameters, for example, by increasing or decreasing one of them by some value δ:

$$A = \pm\delta x, \pm\delta y$$

The reward signal could be to look at the change of distance to the ground truth landmark after taking an action, which incentivizes the agent to take steps that take it closer to the landmark:

$$R(s_t, s_{t-1}, a) = D(x_{t-1}, y_{t-1}) - D(x_t, y_t)$$

where $D(x, y)$ returns the distance from the point (x, y) to the ground truth landmark. If the distance were 10 in the previous state and 8 at the new current state, then the reward would be $10 - 8 = 2$. If the distance were 4 in the previous state and 7 in the new current state, then the reward, or penalty in this case, would be $4 - 7 = -3$.

This formulation was used for landmark detection in 2D and 3D CT images in a series of papers by Ghesu et al. [14] [13] [12]. Compared to other state-of-the-art methods at the time, which performed an exhaustive search across the input image, an RL agent only have to follow a simple path, which in the first paper of the series was reported to speed up the detection by 80 times for 2D data and 3100 times for 3D data [14].

The agent traverses the space by taking a step in one direction, up, down, left, right, forward, and back for 3D images, until it converges around a point that is then considered landmark prediction. Convergence

occurs when the agent starts showing oscillating behavior. In the follow-up papers [13] and [12], a multi-scale approach was used, wherein the agent searches for the landmark at increasingly fine levels. The first and largest field of view ensures that the agent has access to sufficient global context. When the agent converges, the next scale level is used, and the agent continues searching on this finer scale. A final prediction is made when the agent converges on the finest scale level.

Q-learning is used with a deep CNN as a function approximator, making it a DQN, similar to the model used in [30]. A different model is trained at each scale.

In addition to a strong speed-up and ability to detect landmarks perfectly from the authors' validation data, the agent can also detect when a landmark is outside of the present scan. In this case, the agent will attempt to leave the image space.

Different versions of DQN and landmark detection problem formulation have been explored. Inspired by the work by Ghesu et al., Alansary et al. explore using a DQN, a Double DQN, a Duel DQN, and a Double Dual DQN for landmark detection in 3D ultrasound and MRI [2]. The formulation of the problem into state, actions, and reward function remains mostly the same as in [13] and [12], except that the state also has a buffer of the last three previously visited states. Including a small history buffer of previous states increases stability and prevents the agent from getting stuck in repeating cycles. Both fixed and multi-scale searching strategies are compared, but the same DQN is shared across all levels in the multi-scale case. They conclude that a multi-scale search strategy improves the performance, especially for large or noisy images, while also speeding up the search process by 4-5 times, but that the choice of deep RL architecture depends on the environment.

A medical image may consist of multiple different landmarks. Vlontzos et al. extend the DQN to a collaborative model where multiple agents share a common CNN but look for different landmarks [38]. This is done using a shared CNN, followed by K different sets of fully connected layers, where K equals the number of agents. The fully connected layers learn to find their respective landmarks, while the CNN is trained on data from all the agents at once. This collaborative framework acts as an implicit layer regularization to the network and provides indirect knowledge transfer between agents.

The formulation for treating RL as a search through parameter space has been applied to other tasks as well, such as image registration [27] [25], object/lesion localization and detection [29], and more [44].

Image Registration is about aligning two or more images, transforming them into the same coordinate system, and allowing them to provide complementary information in combination. If the transformations can be assumed to be rigid, the set of parameters could consist of simply translation and rotation, making a total of 6 parameters, or 12 actions, for 3D images [27]. If the transformations have to be non-rigid, then free form deformations can be used on the image to be registered, such as in the work by Krebs et al. in 2017 [25]. In their paper, to reduce the number of actions,

they use the first m modes of the PCA as the parameter vector, making a total of $m \times 2$ actions.

Object/lesion localization and detection is the application of object localization to medical imaging. The goal of the algorithm is to find a bounding box around certain objects in the image. For lesion detection in 3D breast scans, Maicas et al. (2017) used a parameter space consisting of translation and scale [29]. The agent can take a step along any of the three spatial dimensions or change the scale of the bounding box, making a total of eight actions. Additionally, a ninth action was added that acted as a trigger for when the agent has found a lesion, instead of relying on an agent's oscillating behavior around the target.

Not all problems fit into this formulation, however. Video summarization is the task of reducing the length of a video while keeping as much useful information as possible. Liu et al. (2020) use RL for summarizing 15 to 65 minutes long fetal ultrasound videos. It is difficult to formulate this problem as a search in parameter space, and therefore the aforementioned reward function based on distance can not be used. Instead, the authors design a reward function that tries to encapsulate what it means to have a good video summarization. The reward function is a sum of three parts:

- $\mathcal{R}_{det}$: the likelihood that a selected frame is of a standard diagnostic plane.
- $\mathcal{R}_{rep}$: the temporal cohesiveness of the selected frames, incentivizing selecting continuous video sections.
- $\mathcal{R}_{div}$: the diversity of the frames, incentivizing selecting frames that are different from each other such that the summarization will be more representative of the whole session.

The action space consists of only two actions: include the current frame or do not include the current frame in the video summary. By using this very simple action-space formulation, and a set of high-level rewards, the agent is still able to achieve good performance. The agent's predicted summary scores 62.08 in precision and 64.54 in recall compared to a user annotated summary.

Chapter 3

Datasets

Overview of the chapter. Short description of the different datasets used.

3.1 Echonet-Dynamic Dataset

The Echonet-Dynamic Dataset[32] is an openly available collection of 10,030, 112-by-112 pixels echocardiography videos for studying cardiac motion and chamber volumes. Each video has been cropped and masked to exclude text, ECG- and Respirometer-information, and downsampled from their original size into 112-by-112 pixels using cubic interpolation. All videos are of the apical-4-chamber view and each video is from unique individuals who underwent imaging between 2016 and 2018 as part of routine clinical care at Stanford University Hospital. Images were acquired by skilled sonographers using iE33, Sonos, Acuson SC2000, Epiq 5G, or Epiq 7C ultrasound machines. Each video has been labeled by a registered sonographer and verified by a level 3 echocardiographer in the standard clinical workflow.

The dataset consists of three parts: *FileList.csv* contains general information about each video, its variables are listed in table 3.1. *VolumeTracings.csv* contains the volume tracings and ED/ES frame index of each video, its variables are listed in table 3.2. And finally *Videos*, containing all the ultrasound videos in `.avi` format. Video frame samples can be seen in figure 3.1.

3.1.1 Getting ED/ES Frame Information

To get the ED and ES frames we have to look at the volume tracings, whose variables are listed in table 3.2. The volume tracings is a list of line segments that together define the volume of the heart at a given frame. For each video there are two sets of line segments, one for ED and one for ES, but which one is which is not given explicitly. We can find this information by calculating the volume from the line segments for both frames and comparing them — the one with the largest volume is ED and the other one is ES.

Table 3.1: Echonet video general information variables.

Variable	Description
FileName	Hashed file name used to link videos, labels, and annotations
EF	Ejection fraction calculated by ratio of ESV and EDV
ESV	End systolic volume calculated by method of discs
EDV	End diastolic volume calculated by method of discs
FrameHeight	Video Height
FrameWidth	Video Width
FPS	Frames Per Second
NumberOfFrames	Number of Frames in whole video
Split	Classification of train/validation/test sets used for benchmarking

Table 3.2: Echonet video volume tracing variables

Variable	Description
FileName	Hashed file name used to link videos, labels, and annotations
X1	X coordinate of left most point of line segment
Y1	Y coordinate of left most point of line segment
X2	X coordinate of right most point of line segment
Y2	Y coordinate of right most point of line segment
Frame	Frame number of video on which tracing was performed

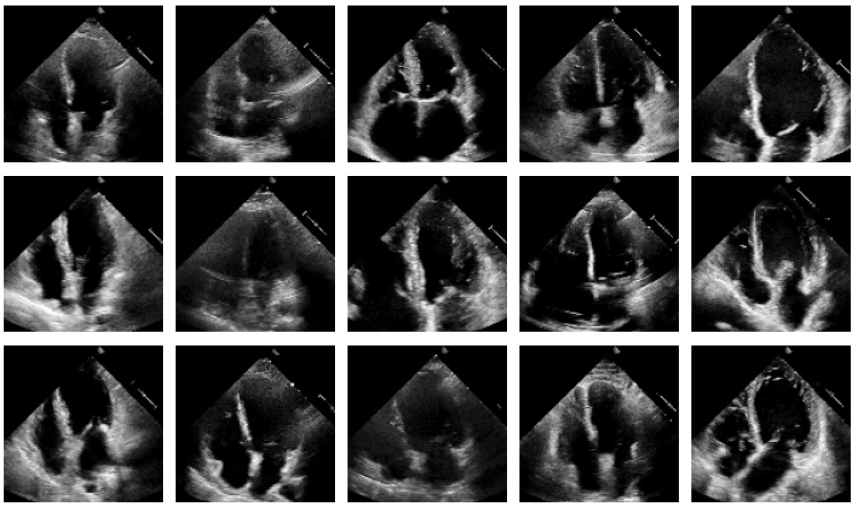


Figure 3.1: The first frames of 15 randomly sampled videos from the Echonet dataset.

3.1.2 Extrapolating Diastole and Systole Labels

As is explored in later chapters, we would also like to label the phase of each frame in the video, not just the frame which ends each phase. When we only have access to the end-frames of each phase the first phase will only have one labeled frame. For example, if the ED frame comes first then only the first frame will be labeled diastole as the rest will be systole, as visualized in figure 3.2.

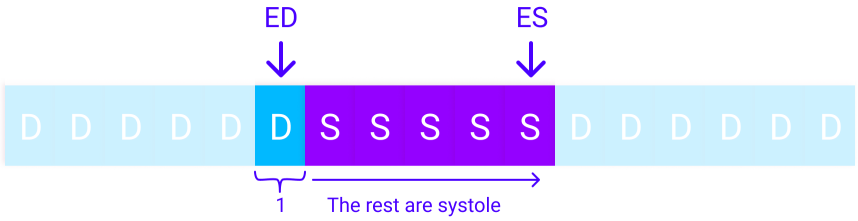


Figure 3.2: Class imbalance: only the first frame is marked with the phase of the first end-event (either ED or ES), all others are marked with the other phase.

We can extract more frames before and after the labeled frames by exploiting the periodicity of the cardiac cycle. As the heart goes from one phase-end to another the difference between the current frame and the first phase-end becomes more and more different until around the point when the opposite end-phase is reached. For example, the next frame with the biggest difference from the ED frame is likely to be close to the ES frame. This periodic effect can be seen if we plot the absolute difference between a frame and the rest of the video, as seen in figure 3.3.

An optimistic approach would be to label all the frames until the previous or next peak difference. For example, if the first event is ED then we could label all previous frames up until the next peak difference as diastole. Likewise, if the final event is ES then we could label all following frames up until the next peak difference as diastole. The peak can be found by finding the first frame whose difference is less than the one preceding it, i.e. when the difference is no longer increasing. This risks labeling too few frames if there is a local peak due to noise, but this problem can be mitigated by smoothing the summed absolute difference values. A gaussian blur with a kernel standard deviation of 5 was used to smooth the values.

We also risk labeling too many frames, adding wrongly labeled frames, because there are no guarantees that the peaks directly coincide with the change of phase. This problem can be mitigated by only including a certain percentage of frames leading up to the peak. We elect to include 75% of the frames leading up to the peaks.

An example of a smoothed absolute-difference curve with 75% of extrapolated frames highlighted is plotted in figure 3.4.

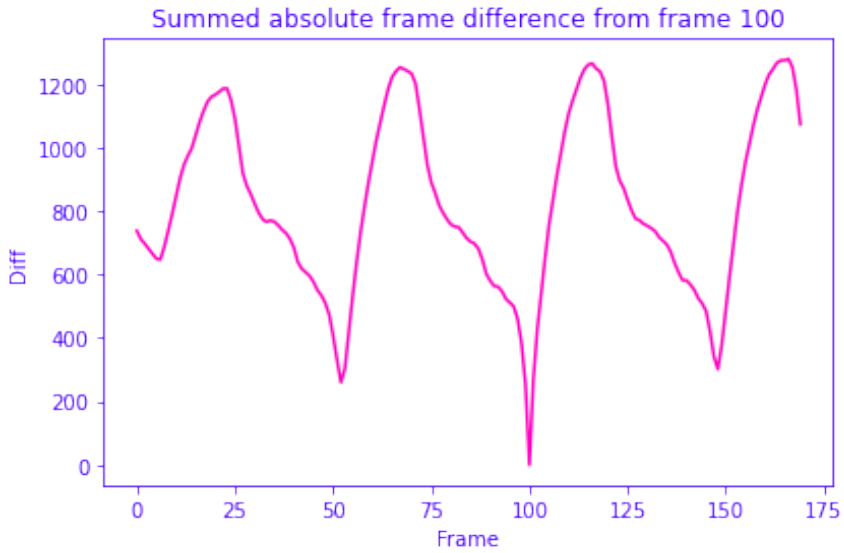


Figure 3.3: The absolute frame difference of all frames in a video compared to frame 100. Notice that the difference for frame 100 is 0 as it (of course) equals itself.

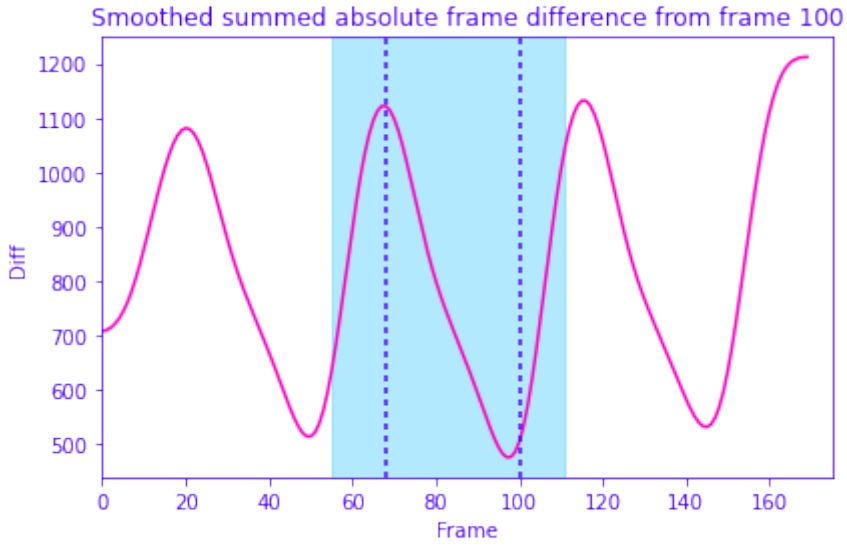


Figure 3.4: The same summed absolute frame difference plot as in figure 3.3, but smoothed using a gaussian blur with a kernel standard deviation of 5. The dashed lines represent phase-end events and the frames in the light blue area are frames with labeled phase. Notice how the labeled frames area only extend 75% towards the peak on the fight side. Also note that the gaussian blur causes the summed absolute frame difference for frame 100 to no longer be 0.

3.1.3 Removing Invalid Videos

An assumption made when labeling the frames is that both events occur within the same cardiac cycle, though this is not always the case in the dataset. To filter out videos where the annotated end-phase events goes beyond a single cycle we again analyze the periodicity using a similar method to the one used in the previous section.

The summed absolute frame difference should at most have one peak if the frames are from the same cardiac cycle. If it has two or more peaks then it suggests that the video contains more than one heartbeat and thus can not be properly labeled. There are 19 of such videos in total, and these are filtered out. A set of good and bad video label examples are visualized in figure 3.5.

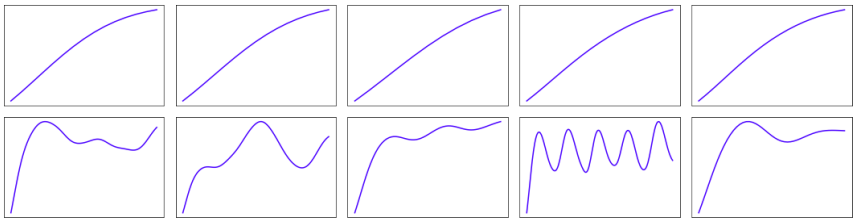


Figure 3.5: The summed absolute frame difference between first end-phase event and the frames up until the next end-phase event. This should only be a half cardiac cycle, so there should be at most one peak. The upper plots show videos where the end-phase labels only cover one half cardiac cycle, while the bottom plots show videos with more than one cardiac cycle, and thus have incorrect labels.

3.1.4 Normalizing Videos

The videos all already have the same size of 112-by-112, but the FPS differ. Luckily, most videos in the dataset have the same FPS — almost 80% of the videos have exactly 50 FPS. The smallest FPS is 18 and the higest FPS is 138. See figure 3.6 for a histogram (logarithmic scale on the y-axis) of the different FPS values.

To normalize the videos with a much smaller FPS than 50 we would have to add information to them by inserting new frames. This may add unwanted bias to the data however, and it is not obvious how to label the interpolated frames when the video goes from one phase to another. To normalize the videos with a much higher FPS we would have to remove frames. Unless the FPS is a multiple of 50, we risk introducing varying FPS to the video which may confuse the model. For example, if a video has 75 FPS we could opt to remove every third frame to make it 50 FPS, but this would make it seem like the heart moves slightly faster every third frame.

Because the Echonet dataset is so large, we opt to simply filter out all videos that have an FPS other than 50. Thus, we filter out another 2071 videos, leaving us with a total of 7946 videos.

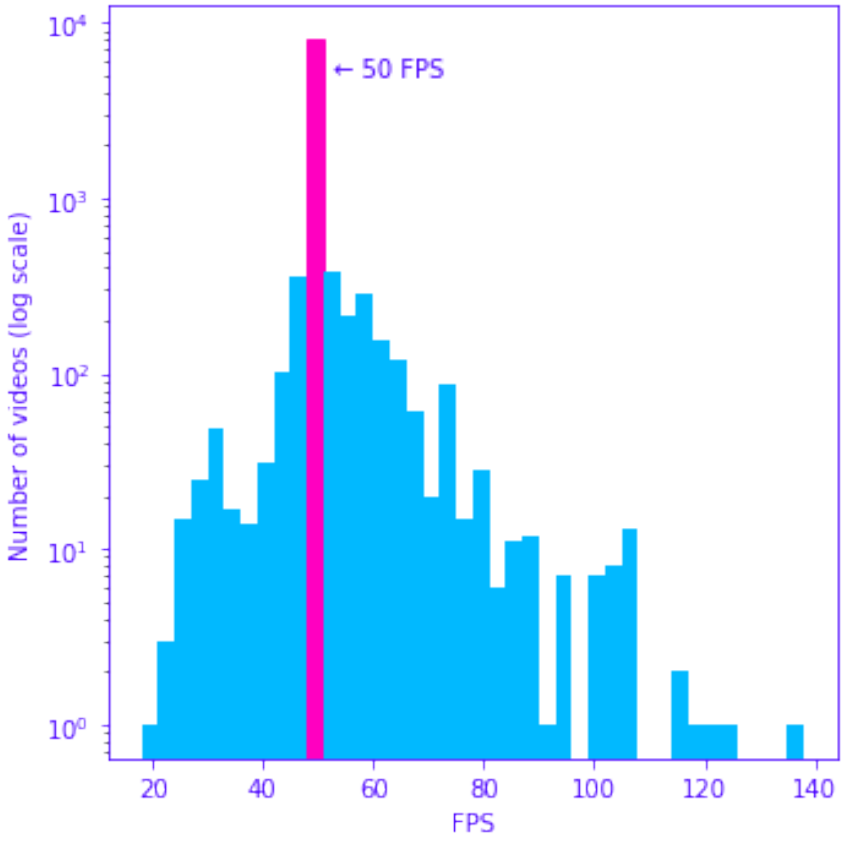


Figure 3.6: A histogram of the different FPS rates of the videos in the Echonet dataset. Note that the y-axis is in logarithmic scale — in fact, almost 80% of the videos have exactly 50 FPS.

3.1.5 Training, Validation, Test Split

The dataset has already been split into three parts: one part for training the algorithm, one part for validation, and one for testing (i.e. presenting results). The percentage split is approximately 75% for training, 12.5% for the validation, and 12.5% for testing. These split ratios remains approximately the same after filtering out videos as explained in the previous two sections. We opt to also use this split in this project.

TODO: A figure showing the pre-processing steps

Chapter 4

Methodology

4.1 Environment Formulation

As described in section 2.2.2 a Markov Decision Process (MDP), which is at the core of RL, can be described using four elements: the state space, the action space, the transition function and the reward function. The states and actions dictate what information the agent receives from the environment and how it can in turn interact with the environment. The transition function defines the effect of actions on the environment. The reward function defines the goal of the agent.

The experiments in the next chapter use an environment inspired by supervised learning approaches and is named Binary Classification Environment (BCE).

4.1.1 Binary Classification Environment

BCE is visualized in figure 4.1. The agent, after observing the current and adjacent frames, takes an action predicting that the current frame is either of Diastole or Systole phase, and receives a reward dependent on its prediction before the environment moves the current frame one frame forwards.

More formally, the observation o_t at time t is the current frame in the video prepended by the N previous frames and the N next frames. The shape of an observation is thus $(W, H, 2N + 1)$. The agent takes the observation as-is and takes one of two actions: *Mark current frame as Diastole* or *Mark current frame as Systole*. After taking an action a_t, the agent receives a reward r_{t+1} and is presented with the next observation o_{t+1}. The current frame is moved one frame forwards after each action taken and the episode ends when there are no more labeled frames to decide on.

Given that videos from the dataset are 112-by-112, the only two hyper-parameters for this setup are N and the choice of reward function. Increasing N means that the agent has access to more temporal information but at the cost of increased computational and memory requirements and a decrease in the number of videos with enough adjacent frames on either side. The number of valid videos for a given N, as well as the change in

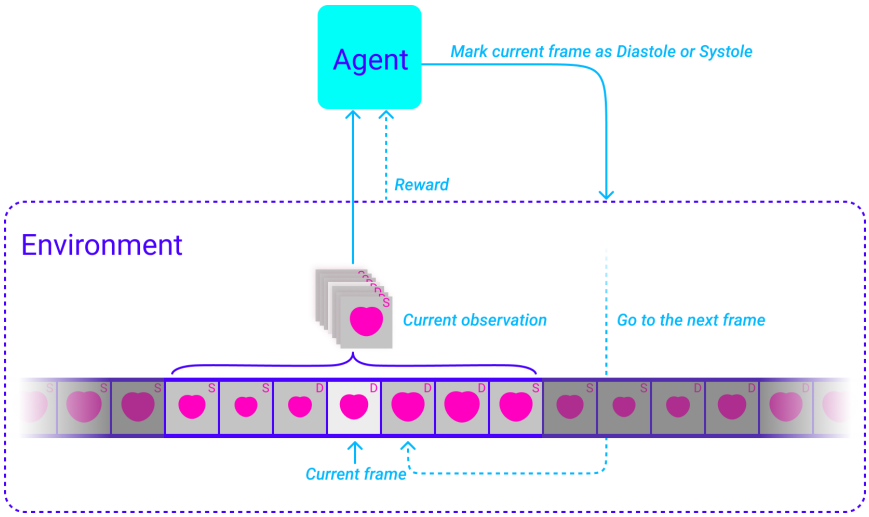


Figure 4.1: Visualization of the Binary Classification Environment loop. An agent sees the observation from the current frame and takes an action, either marking it as Diastole or as Systole, and gets back the reward and the observation for the next frame from the environment.

number of valid videos, is plotted in figure 4.2. As a starting point, N was selected rather arbitrarily to be 3. This means that an observation has the shape $(112, 112, 7)$, having $2 \times 3 + 1 = 7$ channels.

4.1.2 Reward Function Design

The standard metric for this task is the Average Absolute Frame Difference (aaFD), as defined in equation 4.1. aaFD measures the precision and accuracy of predictions by measuring the frame difference between each ground truth event y_t and the corresponding prediction $\hat{y}_t$ generated by the model — a lower aaFD meaning that the model is making fewer errors. t is the index of a specific event, of which there are N in total.

$$aaFD = \frac{1}{N} \sum_{t=1}^{N} |y_t - \hat{y}_t| \tag{4.1}$$

One weakness of aaFD is that it is only defined when there are an equal number of predicted events as there are ground truth events. This is not always the case as an imperfect model may predict more or fewer events. A generalized aaFD ($GaaFD_1$) was considered for a metric instead, calculated as the average frame difference between each predicted event and its nearest ground truth event as in equation 4.2, having the property that it converges towards the true aaFD as the model becomes better. In equation 4.2 $\hat{N}$ is the number of predicted events and $\mathcal{C}(y, \hat{y})$ is the frame difference between the predicted event to the *closest* ground truth event of the same type. For cases where there are more predicted events than there are ground truth events $GaaFD_1$ would, as is rational, give a worse score. But for cases where there are fewer predicted events than there are ground

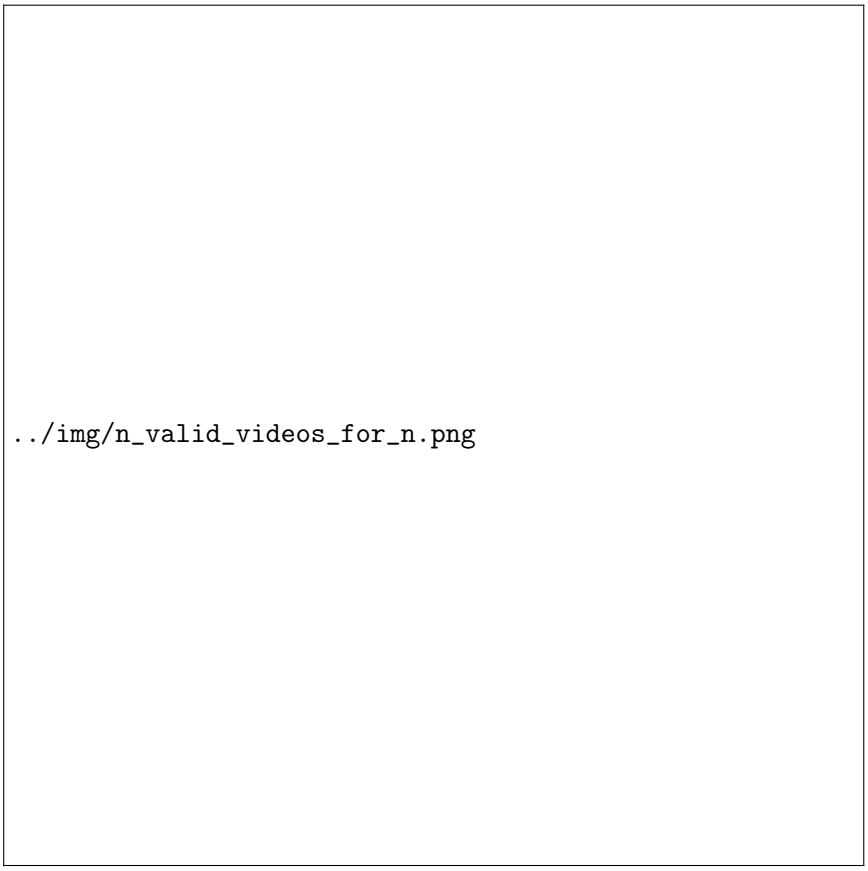


Figure 4.2: The effect of N on the size of the dataset. Left plot shows the number of valid videos (videos with at least N adjacent frames on either side) for the whole dataset. Right plot shows the change in the number of valid videos per N for the whole dataset.

truth events $GaaFD_1$ would give a score that does not reflect its inability to predict all events.

$$GaaFD_1 = \frac{1}{\hat{N}} \sum_{t=1}^{\hat{N}} |\mathcal{C}(y, \hat{y}_t) - \hat{y}_t| \tag{4.2}$$

If we instead calculate the average frame difference between each ground truth event and its nearest predicted event, $GaaFD_2$, as in equation 4.3, we get the opposite problem — too many predicted events are not reflected negatively in the score.

$$GaaFD_2 = \frac{1}{N} \sum_{t=1}^{N} |y_t - \mathcal{C}(y_t, \hat{y})| \tag{4.3}$$

By combining $GaaFD_1$ and $GaaFD_2$ as in equation 4.4 we mitigate these problems while maintaining the convergence property.

$$GaaFD = \frac{1}{N + \hat{N}} (\sum_{t=1}^{N} |y_t - \mathcal{C}(y_t, \hat{y})| + \sum_{t=1}^{\hat{N}} |\mathcal{C}(y, \hat{y}_t) - \hat{y}_t|) \tag{4.4}$$

Using negative GaaFD (negative because we wish to minimize it) as a reward function for RL means that we are optimizing the agent directly for our main metric aaFD. It does have one final flaw, however: it is only defined on whole episodes. This means that the agent has to run an entire episode before getting a reward, making the reward signal sparse.

We could instead frame the problem as a simple classification problem where the agent must classify individual frames as either ED, ES, or neither. This allows us to give a reward at each step depending on whether the prediction was correct or not. One problem with this approach is that there is a heavy class imbalance because most frames are neither ED nor ES. A solution to this is to instead predict the phase, either Diastole or Systole, as it is trivial to find ED and ES from the phase by finding the frames where it transitions from one to the other.

From this we can define a simple reward function R_1 that gives a reward of 1 if the predicted phase was correct and -1 if it was incorrect, as seen in equation 4.5. The information that the agent receives from the reward signal R_1 is slightly different from the one defined through GaaFD, as GaaFD penalizes predictions that are more wrong heavier than those that are close to the ground truth. We can make the reward signal more similar to GaaFD by defining it in terms of the distance to the nearest predicted phase, as seen in equation 4.6, where $d(s, a)$ is the distance from the current phase s to the nearest predicted phase a.

$$R_1(s, a) \triangleq \left\{ \begin{array}{ll} 1 & \text{if } s = a \\ -1 & \text{if } s \neq a \end{array} \right\} \tag{4.5}$$

$$R_2(s, a) \triangleq -d(s, a) \tag{4.6}$$

4.2 Frameworks and Libraries

The code to train and run the agent is written in Python because of its ML and data-processing ecosystem. The main framework for data-processing is JAX [**jax2018github**], selected because of its usage at Deepmind for multiple state-of-the-art projects [TODO: citation, alphago, alphafold, etc]. Many of the main libraries used are also developed by Deepmind; a list of the most important ones can be found in table 4.1. Other frameworks considered were Tensorflow [**Abadi_TensorFlow_Large-scale_machine_2015**] and PyTorch [**NEURIPS2019_9015**].

Table 4.1: A collection of the most important libraries used in the project.

Library	Description
jax	Main data-processing framework. Provides autodifferentiation, vectorization, Just-
gym	An interface for defining RL environments [**brockman_openai_2016**]
dm-haiku	A neural network library for JAX [**haiku2020github**]
optax	A gradient processing and optimization library for JAX [**optax2020github**]
rlax	Building blocks for building RL agents [**deepmind2020jax**]
dm-acme	Distributed RL agent implementations and building blocks [21]
dm-reverb	A database for storing and sampling experience replay [7]
dm-launchpad	A library for defining and creating distributed systems [40]

TODO: Add SciPy (used for calculating metrics)

4.2.1 Agent Architecture

Deep Q-Network was selected for the RL agent architecture. DQN is a well-established method for scaling up RL by approximating the expected returns of taking an action in a given state using a (deep) neural network. It is also simple to train distributedly as it is off-policy, enabling us to separate the algorithm into a learner and multiple agents, as explained in the next sub-section.

We take advantage of a few additions to the original DQN algorithm, namely: Prioritized Replay, N-step returns, and Double Q-Learning. For facilitating exploration, an ϵ greedy policy is used.

Neural Networks

Two neural networks are explored — one simple baseline CNN and a more complex CNN with more parameters.

The first neural network is relatively simple, and is inspired by the original Atari DQN paper[30]. It has two convolutional layers and two fully connected layers, each layer except for the last one is followed by a ReLU activation layer. The first convolutional layer has 16 output channels, a kernel size of 8-by-8, and a stride of 4. The second has 32 output channels, a kernel size of 4-by-4, and a stride of 2. Before the fully connected layers the data is flattened. The first fully connected layer has an output size of 256, and the final layer has two outputs, each representing the estimated

value of taking one of the actions, given the input state. In total there are 1 621 810 parameters.

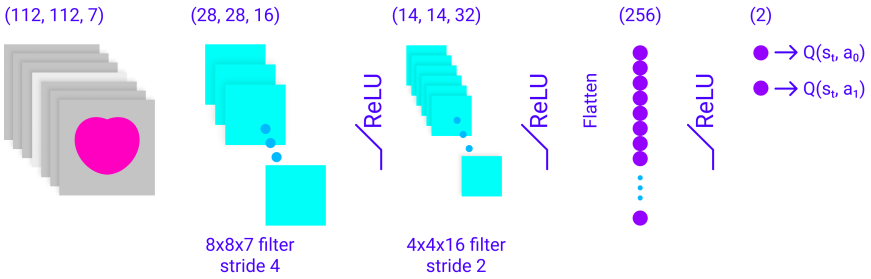


Figure 4.3: A visualization of the simple DQN-Atari paper inspired CNN.

TODO: Describe MobileNet

Loss Function and Optimizer

The loss function is the Double Q-Learning loss where the TD-error is calculated with respect to another Q-network. Because of this we have to keep track of two sets of network parameters: one for the selector Q-network and one for the estimator Q-network. Huber loss[**huber_robust_1964**] is applied to the TD-error such that the L2 loss becomes linear after a certain threshold. In addition, the loss is weighted with respect to the prioritixed replay importance weights.

The Adam optimizer[**kingma_adam_2017**] is used to update the selector parameters and the target network parameters are updated to equal the selector parameters every 100 gradient descent steps.

Distributed Training

As mentioned, DQN lends itself nicely to distributed training. In this project, this is achieved through Deepmind's library Acme[21]. At the center of Acme is another library by Deepmind called Reverb[7]. Reverb is a database for storing experience replay samples that lets us insert and sample experiences independently. If we separate the learning step and the acting step om the algorithm Reverb can be used as the communication point between the two. In this way one or more actors, possibly on different machines, can generate experience samples and insert them into the Reverb experience replay database and a learner, also possibly on a different machine, can sample from it to perform gradient descent. The actors and the learner doesn't need to know about each other, except when an actor needs to update its parameters, in which case it needs to query the learner for the latest trained parameters. It is also trivial to add one or more evaluators that can run in parallel and that only need to query the learner for the latest trained parameters. Inter-process communication is facilitated by a third library, also by Deepmind, called Launchpad[40].

There is a balance to be made between how fast experience samples should be added to the experience replay and how fast they should be

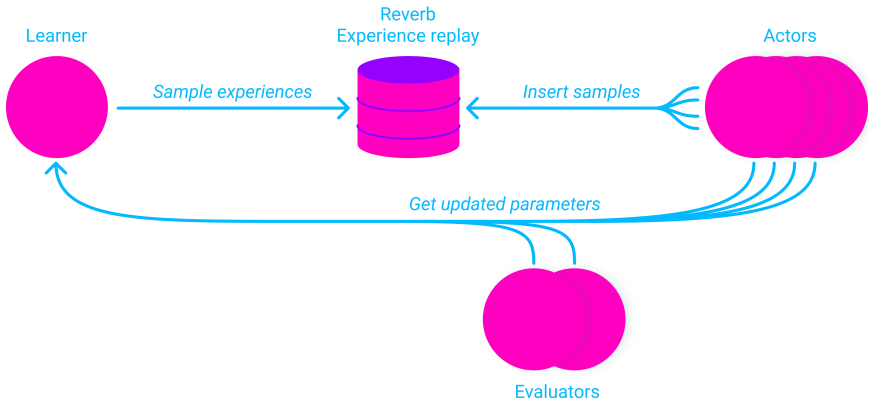


Figure 4.4: The distributed RL training system. Each pink node runs in a separate Python process, and each blue arrow is a inter-process function call facilitated by Launchpad.

sampled by the learner. If the learner samples faster than the actors are able to generate new samples then the network will be trained using trajectories generated from outdated policies. If the actors generate new samples much faster than the learner is able to sample then we are arguably wasting computer resources.

Reverb helps maintain this balance through rate limiters. We use a rate limiter that tries to maintain a specified ratio between insertions and samples, blocking either the actors from inserting new samples or the learner from sampling if the ratio starts to differ too much. For example, using a samples-per-insert ratio of 2 means that, on average, each insertion made by an actor will be sampled twice. A ratio of 0.5 means that, on average, each insertion will be sampled one half time — i.e.: there are twice as many insertions as there are samples.

4.3 Evaluation

During training, the updated parameters of the model are continuously evaluated using GaaFD on 50 random videos in the validation set. Smoothing is applied to the learning curves using a Gaussian filter with a kernel standard deviation of 10 in order to compensate for the low sample size for each point. The best parameters are selected by finding the parameters that produces the lowest GaaFD during training for the smoothed GaaFD learning curve.

The main evaluation metric for the trained model is aaFD. However, some videos may not have the same number of predicted events as there are ground truth events, and as such aaFD is undefined. Because of this the percentage of videos that have a valid aaFD is presented and aaFD is calculated using only those videos. The corresponding ground truth event to each predicted event is chosen to be the one that is closest to it and we can therefore use GaaFD, as defined in equation 4.4, for calculating aaFD.

It may also be interesting to see the density plots of GaaFD for all

videos and compare the performance of the agent on ED- and ES-frames individually. These density are created using Gaussian kernel Estimation (KDE) [TODO: citation]. The kernel bandwidth is automatically selected using Scott's rule, which is the default selection method for SciPy's KDE implementation.

Because the RL problem formulation is so similar to a regular binary classification problem, accuracy and balanced accuracy is also reported. Accuracy and balanced accuracy are defined on frame phase predictions instead of on end-phase events. Accuracy is simply the percentage of correctly labeled frames, as defined in equation 4.7, where $1(y = \hat{y})$ is the indicator function. Given that there is a class-imbalance between diastole and systole frames, balanced accuracy gives a score that is more representative of the actual model performance. Balanced accuracy weights systole frames accuracy higher than diastole frames and is defined in equation 4.8. TP, FP, TN, and FN stands for True Positives, False Positives, True Negatives, and False Negatives, respectively. It is also defined as the average between the sensitivity and the specificity. The balanced accuracy score is also rescaled such that it gives a score in the range $[-1, 1]$, where 0 means that the model's predictions are random, and -1 and 1 means that the predictions are all incorrect or all correct, respectively.

$$\text{accuracy}(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} 1(\hat{y}_i = y_i) \tag{4.7}$$

$$\text{balanced-accuracy}(y, \hat{y}) = \frac{1}{2}\left(\frac{TP(y, \hat{y})}{TP(y, \hat{y}) + FN(y, \hat{y})} + \frac{TN(y, \hat{y})}{TN(y, \hat{y}) + FP(y, \hat{y}}\right. \tag{4.8}$$

Models are also evaluated on their inference time — how long it takes to make predictions for a video. To use a trained model, one can use the Q-network directly, without instantiating a gym environment or using an ϵ -greedy policy. The Q-network outputs the expected returns of taking either action so picking the action with the highest output is the same as following a greedy policy. The Q-network can be evaluated on individual frames or on the video as a whole, where all the frames are combined into a single batch. Evaluating each frame individually enables incorporating the model into a pipeline of streaming frames, of which one step is predicting the current cardiac phase. Evaluating the whole video as a batch is generally faster as it gets away with less IO overhead of sending data back and forth between the CPU and the GPU.

Batching the frames of a video may require more JIT-compilation with JAX. This is because, in order to speed the network up significantly, it is JIT-compiled to XLA, but JIT-compiled functions requires that the shape of the data remain the same. If the shape of the data is not the same, f.ex. if we are evaluating two videos with a different number of frames, as two different batches, the function will be recompiled, adding overhead. This could be solved by fixing the batch-size to a constant number. For videos

with fewer frames than the batch-size, or with a number of frames that can not be split into equal chunks of the batch-size, frames filled with zeros can be added as needed. These extra frames creates needless work on the GPU, but doesn't require recompilation.

Inference time is evaluated using single frame-inference and batched-frames inference with a batch-size of 128. Compilation time, the time it took to run the function the first time, is also included.

Finally, models are evaluated on how long it took to train them, in terms of clock time and the number of SGD steps performed.

4.4 Selection of Hyper-Parameters

4.5 Incorporating Search

This is not a RL problem, really. RL is designed to search through an unknown state space. In the previous setup there is no exploration as previous actions do not affect future actions. There is therefore no reason to believe that RL will outperform a carefully designed Supervised learning approach. By transforming the problem to one that requires search we will have a problem that is not trivially solved by supervised learning but where RL can shine. Though this may seem like straightening a screw to make it work with a hammer there may be unforeseen benefits. Of great importance to ML is to represent the problem space in a way that is easy to learn from. Perhaps there is an optimal representation of the problem of ED-/ES-detection that also happens to require search?

4.5.1 Temporal Search

We could formulate the problem as a search in time where the agent must learn to move the current frame towards the end-phase event. The agent sees the current frame and some number of previous and following frames and can either move the current frame backwards or forwards. The agent can be rewarded with 1 if it moves a step closer to the nearest end-phase frame and -1 if it moves away from it.

There are a handful of issues with this approach. **Issue 1**: we'd have to train two different agents: one for ED and one for ES. **Issue 2**: there is no terminal state and the episodes can run forever. **Issue 3**: there will be ambiguity in what frame the agent truly predicts as end-phase because it will likely show oscillating behavior around the predicted frame. **Issue 4**: we'd have to run multiple agents at different points in the video in order to find all end-phase events and it is not obvious how to do so.

Issue 2 and **issue 3** can be partially solved by including a third action for marking the current frame and ending the episode, though this may still lead to the agent getting stuck in an endless loop of going back and forth. We could also keep just the two actions, but terminate the episode once the agent starts showing oscillating behavior, as in [TODO: citation], as this indicates that it has found the predicted frame. The problem with

this is that the final predicted frame would be ambiguous as we don't know which of the two frames that the agent oscillates between is the true predicted frame. If we're using DQN, we could however peek at the Q-values and pick the frame where expected reward of taking the action with the maximum expected reward is the lowest. **Issue 4** may be solved by starting an agent from each frame, though this would increase the computational requirements of the algorithm.

4.5.2 Spatial Search

Instead of searching through the frames of the video, we could let the agent search spatially in the video. In this formulation, the agent only has access to a part of the images while making a prediction. Similar to landmark detection tasks it can move its focus around in the image, the hope being that it is able to discover parts of the video which makes it easier to identify the correct phase. In a way this can be seen as reducing the space and memory requirements at the cost of speed, as the agent has to process a smaller part of the image, but may explore for multiple steps before making a prediction.

One option is to look at a Region Of Interest (ROI) around a point that the agent can move. If we build upon the simple binary classification environment described in previous sections, this would add 4 new actions: move up, move down, move left, and move right. This is visualized in figure 4.5. Because we reduce the size of the observations we could either trade it for reduced memory usage or for including more temporal information in terms of included adjacent frames.

Another option is to take inspiration from m-mode imaging used in ultrasound. We can define a synthetic m-mode image in terms of a line in the video. The synthetic m-mode image shows how the pixels along this line changes over time. A video can be seen as a 3D data cube, consisting of width, height, and time, but using the synthetic m-mode technique width and height are replaced by the line, effectively removing one spacial dimension while keeping the temporal dimension intact. Compared to the region of interest exploration scheme, synthetic m-mode exploration allows us to keep more temporal data. This synthetic m-mode exploration formulation adds 6 new actions: move up, move down, move left, move right, rotate left, and rotate right. M-mode imaging is also a well established imaging mode in clinical settings, so this is the method that we want to explore further.

When moving the line up, down, left, or right, it is done relative to its own rotation. We call this local translation, which is different from global translation where the movement is independent of the rotation of the line. Using local translation is presumed to add some rotational invariance, as the rotation of the video itself can be counteracted by the m-mode line without changing the perceived m-mode effects of translation. This also makes the effects of the up- and down-translations trivial, independent of rotation — it simply shifts the m-mode image down or up, respectively.

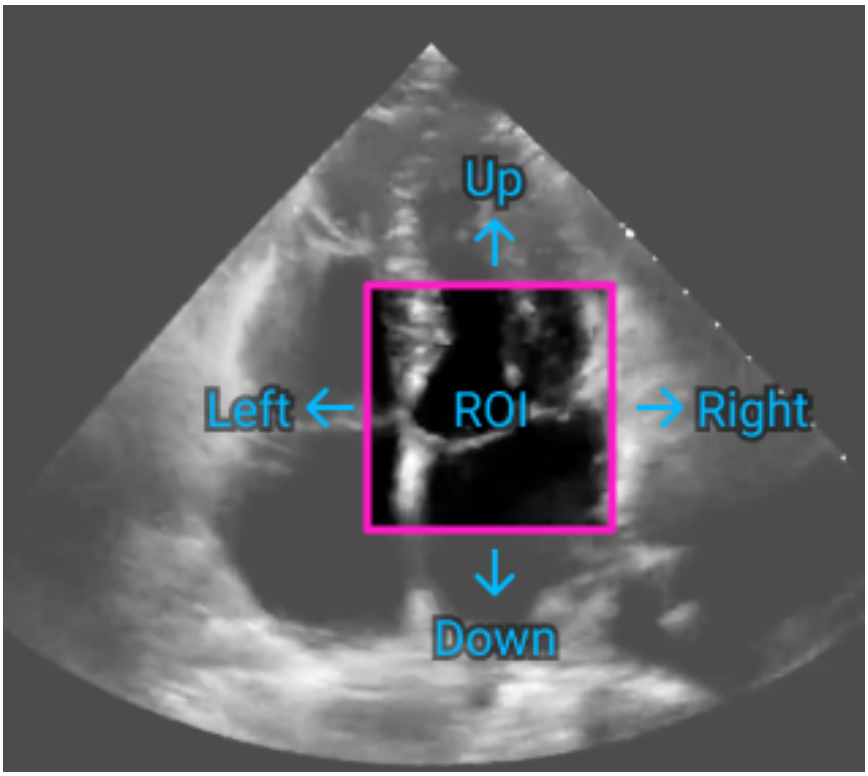


Figure 4.5: A Region Of Interest (ROI) is given to the agent which it can then move around in order to explore.

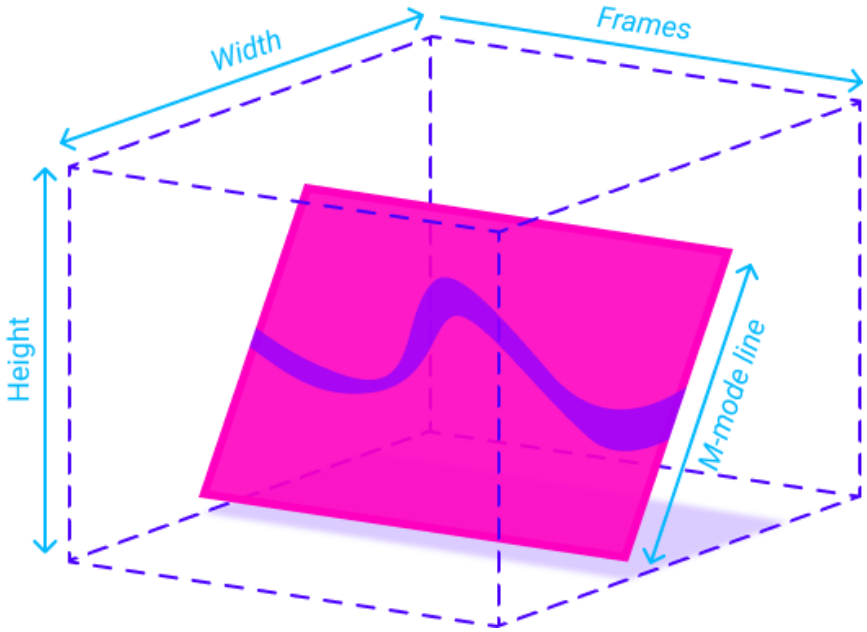


Figure 4.6: An m-mode image is an intersecting plane in 3D "video space".

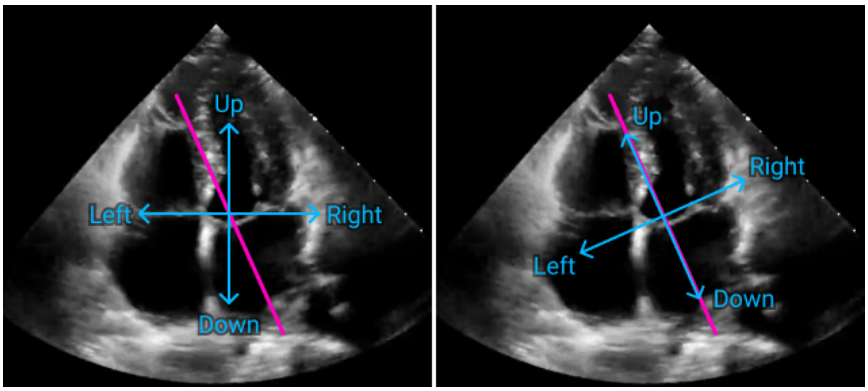


Figure 4.7: Global (to the left) versus local (to the right) translation. Local translation means that the movement depends on the direction of the m-mode line.

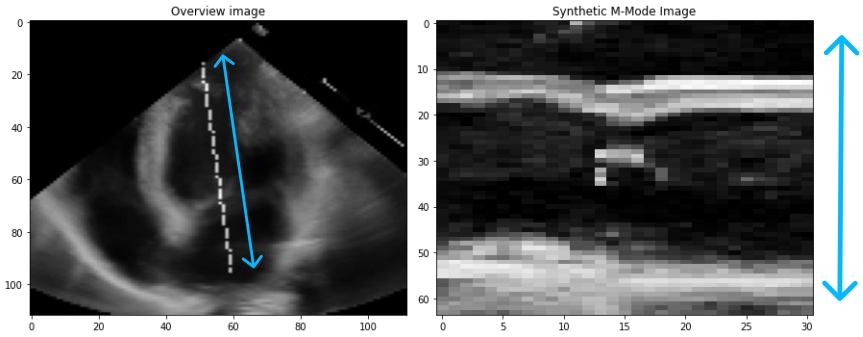


Figure 4.8: Moving the line in up or down using local translation changes the synthetic m-mode image very little — it simply translates the whole image up or down, as indicated by the blue arrows. To the left: an overview image of a video with the line added on top. To the right: the resulting synthetic m-mode image.

4.6 M-Mode Binary Classification Environment

Using a synthetic m-mode search space scheme, we formulate the M-Mode Binary Classification environment. The agent can make one of 8 actions: *Mark current frame as Diastole*, *Mark current frame as Systole*, *Rotate line* (left or right), *Move line along its pointing direction* (up or down), and *Move line perpendicular to its pointing direction* (left or right). The observation includes the synthetic m-mode image of the current line position, but we also want to give the agent explicit information about what it would look like if it moved or rotated the line, and a history of the latest actions.

The observation therefore consists of the synthetic m-mode image for three different rotations (rotated left, not rotated at all, and rotated right) and for three different perpendicular movements (moved to the left, not moved at all, moved to the right), for a total of 9 synthetic m-mode images. The synthetic m-mode image is created by interpolating the line across the video using nearest neighbour. Up and down line movements are not included because they do not provide as much information to the agent, as is visualized in figure 4.8. An overview image consisting of the average of the first 50 frames is also included in the observation, as well as the current position of the line overlapping it in a different channel, also visualied in figure 4.8. Lastly, we include the last 5 actions taken as a one-hot encoded array of shape $(5, 8)$ — 8 being the number of possible actions. Observations are thus a tuple of an "overview" image of shape $(W, H, 2)$, a synthetic m-mode image of shape $(T, L, 9)$, and an action history array of shape $(5, 8)$, where W and H are the width and height of the video respectively, and T and L are the number of frames (amount of temporal information) and length of the line respectively.

At the start of an episode the line is placed at a random position. This is in order to force the agent to learn to explore instead of learning to predict the phase from a common starting position. The random position is selected by first placing the line, centered, facing upwards, before translating it in the direction it's facing by a random amount sampled uniformly from the interval $[-0.1H, 0.1H]$, and in the perpendicular direction by a random amount sampled uniformly from the interval $[-0.1W, 0.1W]$, and rotated by an angle sampled uniformly from the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ radians. The starting positions are asserted to be within bounds and if a line somehow is generated outside of bounds a new line is generated instead. The random starting positions were verified to be reasonable through visual inspection of a sample of 1 000 lines as seen in figure 4.9. An episode ends once the phase of every frame have been predicted by the agent, or until the agent has taken 200 steps. We have to cut the episode off at 200 steps because the agent may now move indefinitely.

The reward function is the same as in the simple binary classification environment, but with some modifications. The agent receives a reward of -1 if it moves the line such that it goes out of bounds of the video. The agent will also receives a reward of -1 if it becomes stuck in a loop. The agent is stuck in a loop if it has already visited the current position at an

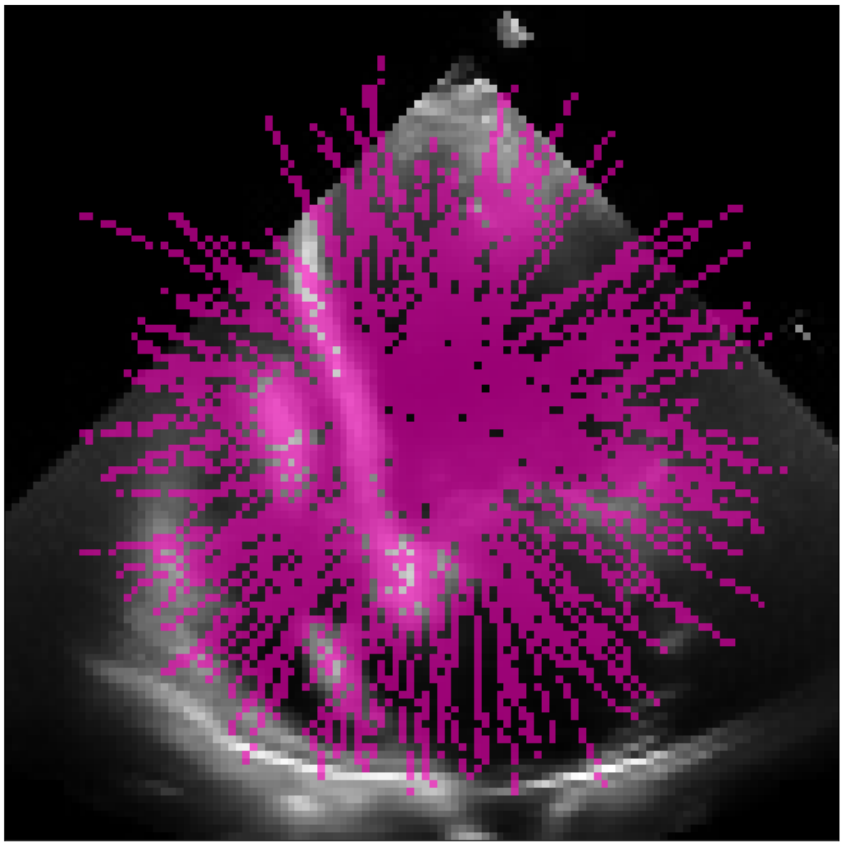


Figure 4.9: The union of 100 randomly sampled m-mode lines.

earlier time without marking the frame as diastole or systole. If either of these things happen the line is moved to a new random position.

4.6.1 Agent Architecture

We keep the same base architectures as in the simple binary classification environment, but we need to also accomodate the overview image and action history array. This is done by passing all three through their own neural network before concatenating the result and passing it through a couple more fully connected layers. Both the synthetic m-mode image and the overview image is passed through the Atari DQN-paper inspired CNN that is visualized in figure 4.3, but with the final output layer removed. The action-history array is flattened before being passed into a fully connected layer with 32 outputs followed by a ReLU activation layer. After concatenating the three results they are passed through yet another fully connected layer of 64 outputs and a ReLU activation layer, before being passed through a final fully connected layer with 2 outputs.

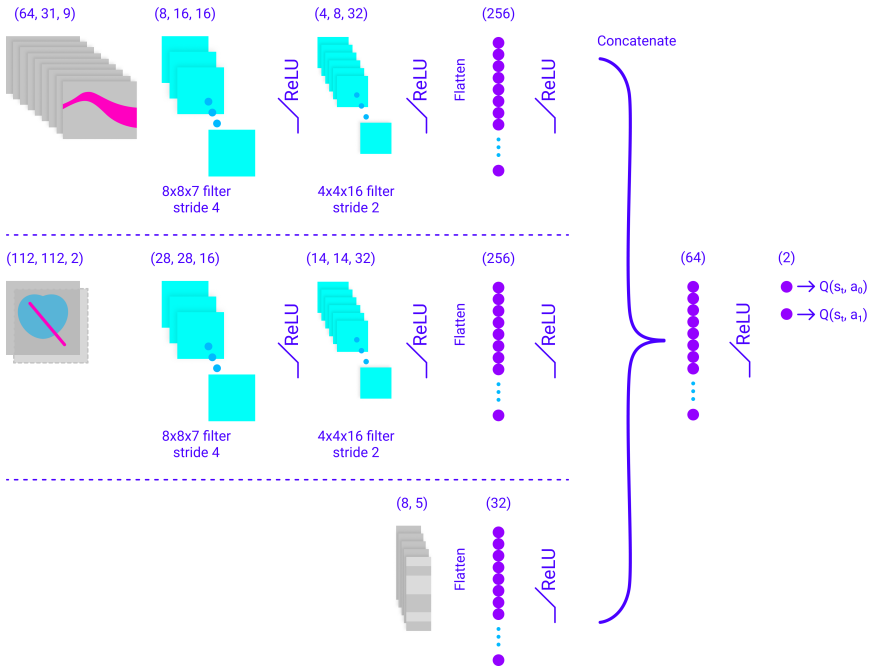


Figure 4.10: The network architecture of the m-mode agent. An observation consists of three parts. Each part is processed independently by a neural network before being concatenated and used to produce the approximated Q-values.

For the more complex network architecture, we simply swap out the CNN for the synthetic m-mode images with MobileNet-v1.

TODO: Write about why we chose DQN, what alternatives we considered, etc.

TODO Discussion

Under the hood, the DQN algorithm is solving a regression problem. Given a state, the model predicts the expected future returns after taking a given action.

TODO: BCE is using RL for a job that asks for Supervised Learning. There is no exploration, but we still use exploration mechanisms like greedy-epsilon. Using epsilon of 1.0 (100% random decisions while training) is a sign that something is off. It is like an inefficient supervised learning training loop.

- How is this similar to regular supervised learning classification problem?
  - DQN predicts expected future returns of taking an action. We can set up a supervised learning regression problem that predicts the same thing
- We use epsilon=1 and discount=0 — implications?
- Write about how DQN is simply a regression problem
- Future work could be using Policy Gradient methods

4.6.2 TODO Discussion

- Sparse reward signal may make the results worse. Can be counteracted by: what? n-step? Less dicsount (gamma closer to 1.0)? Using "advantage" for Q-function? Actor-critic network agent?

Chapter 5

Experiments and Results

This chapter is dedicated to exploring the performance of methods described in the previous chapter through experiments. It is separated into two main parts, one for each environment formulation. The three reward functions are explored: Generalized Average Absolute Frame Difference (GaaFD), a simple binary phase prediction reward R_1, and the proximity-based phase prediction reward R_2.

TODO: How fast is it at inference?

Device	Run	Compilation time	Average run time
GPU (including IO)	168 frames	2706.17 ms	48.96 ms
	Single frame	248.20 ms	2.64 ms
GPU (pre-placed arrays)	168 frames	2385.17 ms	1.52 ms
	Single frame	395.50 ms	0.17 ms
CPU	168 frames	224.09 ms	36.10 ms
	Single frame	118.42 ms	0.90 ms

5.1 Simple Binary Classification Environment

5.1.1 Generalized Average Absolute Frame Difference Reward Function

We start by exploring the use of GaaFD as defined in equation 4.4 as the reward function. This has the benefit that we are directly optimizing the agent for the key performance metric, which is aaFD as defined in equation 4.1. However, a weakness, as discussed in section 4.1.2 is that it is only defined at the end of an episode, making the reward signal very sparse. The agent will only get a reward at the last step of an episode, which on average lasts for 50 steps.

For a random agent the GaaFD score is approximately in the range of 10 to 20. We scale this down by a factor of 10 to keep the gradients from exploding too much when there is a large discrepancy between predicted returns and actual returns.

To solve for reward-sparsity we use multistep bootstrapping with a value of N that is greater than or equal to the number of steps in an episode.

This will in practice mean that the agent is trained using the Monte Carlo method. We do this by setting $N = 200$ because we automatically stop an episode once it reaches 200 steps (though in the case of the simple binary classificaiton environment this will never happen because no video have this many frames).

We also set the discount value $\gamma = 1.0$ which means that an agent tries to maximize all future rewards. Having $\gamma < 1.0$ means that the calculated returns will be more noisy and harder to predict because the discounted returns calculated for steps earlier in an episode would have a lower value than those calculated closer to the end.

Three values are tested for the exploration hyper-parameter ϵ: $\epsilon = 0.0$, $\epsilon = 0.01$, and $\epsilon = 0.1$. The agents were allowed to train until they visually reach a plateau. A full list of the hyper-parameters used is listed in table 5.1.1 (most relevant ones are highlighted).

Hyper parameter	Value
Epsilon	$\{0.0, 0.01, 0.1\}$
Discount	1.0
N (N-step bootstrapping)	∞
Target update period	100
Importance sampling exponent	0.2
Priority exponent	0.6
Number of actors	8
Min replay size	10 000
Max replay size	250 000
Samples per insert ratio	0.5
Optimizer	Adam with default parameters
Huber loss parameter	1.0
Learning rate	1^{-4}
Gradient descent steps	$\{100\,000, 150\,000, 200\,000\}$
Batch-size	128

From the learning curves in figure 5.1 we see that the agent does indeed over time learn to make correct predictions. Interestingly, the best agent is the one who performs no exploration and "explores" greedily at every step. It also converges faster on a solution, though they are all arguably very slow, taking tens of thousands of SGD steps to converge. Even a value of $\epsilon = 0.01$, meaning that only 1 percent of actions are random, significantly reduces performance and convergence speed.

Not surprisingly, over time the agent will perform better on the training data compared to the validation data, as is visualized in the right-most plot in figure 5.1. This is less apparent the agent that uses a value of $\epsilon = 0.1$ as the overfitting curve, whose positive values indicate worse performance on the validation set compared to the training set, remain closer to 0. For neither agent the increased relative performance on the training set appears to have a negative impact on the validation performance, so overfitting is not a big issue.

The reason behind these results is likely due to the learner having access

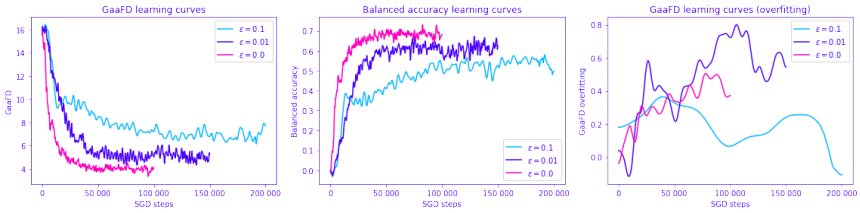


Figure 5.1: The training curves of using GaaFD as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the training set over training time, positive values indicating overfitting on the training set. Each point in the curve is calculated on 50 random videos in the validation (or training) set. The curves have been smoothed using a gaussian filter with a kernel standard deviation of 4 to reduce noise due to the low sample size of each data point. The overfitting (right) plot has additionally been smoothed using a gaussian filter with a kernel standard deviation of 50 to make sure that overall trend is visible.

to noisier signals the higher the value of ϵ. Recall that the agent only receives a reward at the very end of the episode which on average lasts for 50 steps. Any mistake in those 50 steps will be penalized and the agent has no way of knowing whether it was penalized for an action taken under its policy or an action taken randomly.

Further evidence of this can be found in the loss curves, as generally the models with $\epsilon \in \{0.01, 0.1\}$ have a greater loss at the end of training, as seen in 5.2. This indicates that the data that it has trained on "surprises" the model which could be explained through the fact that when it makes a mistake through random exploration the model will not know which action in the episode was the true culprit.

Another interesting feature of the loss curves is the valleys in the beginning of training. In the beginning of training the model has no knowledge about the data and any prediction will be random. This leads to actors taking random actions and the GaaFD, i.e. the reward, staying pretty uniformly bad (TODO: back up with data?). As the actors learn which action to pick the sample data distribution changes, reflecting the new policies. This in turn creates a change in loss as the learner "catches up" to the new policy. As the model approaches a good estimate of the true Q-value Q^* it will make less mistakes and in turn the loss will decrease (TODO: this can probably be explained through information theory and Shannon entropy also? The amount of new information gained follows a bell curve depending on the accuracy of predictions. When every action is random, how do we know what to update? When 50% of actions are randomly incorrect we have maybe more information? When only 5% of actions are randomly incorrect, we again have little information. **OR** maybe it could simply be explained by that an agent needs time to fill the replay buffer with samples from the new distribution. But why does the

curves have different shapes then?). (TODO: what does it mean that the blue curve has lower loss than the dark-blue/purple one?)

(TODO: perhaps interesting to see how the Q-values change over training time for a video? An image where x-axis are the estimated value of marking each frame as Diastole and y-axis is SGD-steps.)

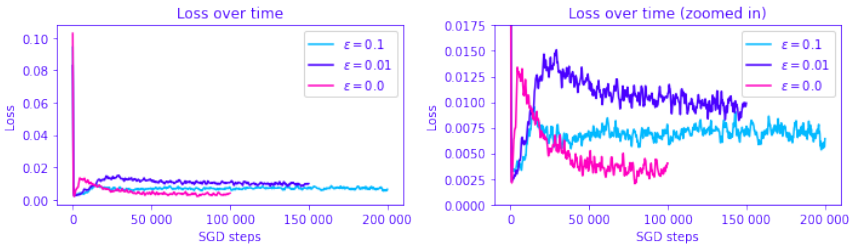


Figure 5.2: The training loss over time for different values of epsilon. The left plot shows the full y-axis, while the right plot shows the same plots but with a zoomed-in y-axis.

For all three models we select the parameters at the step where GaaFD was lowest for the validation set to find the best one. Because the GaaFD is only calculated on a subset of the validation data, 50 videos to be exact, we opt to smooth the curves first using a Gaussian filter with a kernel standard deviation of 20. If we don't do this we risk selecting a model that by chance was evaluated on 50 very easy videos instead of being the actual best model. Smoothing the curves this way may lead us to not pick the absolute best model parameters, but in general it should be among the best. The SGD step of the selected model parameters and the overall performance of that model are listed in table 5.1.

Table 5.1: Performance of agents trained using GaaFD as the reward function on the test dataset.

	$\epsilon = 0.1$	$\epsilon = 0.01$	$\epsilon = 0$
Best model SGD step	167 336	136 996	95 616
GaaFD	5.84	4.59	3.68
GaaFD ED	5.69	4.84	3.74
GaaFD ES	5.83	4.20	3.50
% valid aaFD	0.64	0.70	0.77
aaFD	3.51	2.71	2.43

TODO: Explain how ES performance is better than ED performance. Hypothesis: systole phase is shorter therefore mistakes are closer to ES on average.

TODO: Explain how (for $\epsilon = 0.1$) GaaFD is higher (worse) than both the GaaFD for only ED and only ES. It's because when we filter out events sometimes there are no events to predict.

A GaaFD-density plot comparing the performance on both the test-data and the train-data was created for each model using Gaussian Kernel Density Estimation (KDE), as seen in figure 5.3. The kernel bandwidth is

selected using Scott's rule which is the default selection method for SciPy's KDE implementation. (TODO: discuss plot)

Similarly, a GaaFD-density plot was created for ED- and ES-events individually, as seen in figure 5.4. (TODO: discuss plot)

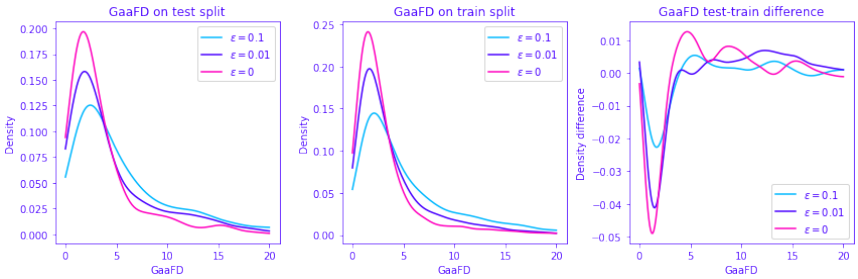


Figure 5.3: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$). The left plot compares all three models on the test-set. The middle plot compares all three models on the train-set. The right plot shows the difference between the two as a means to visualize model overfitting.

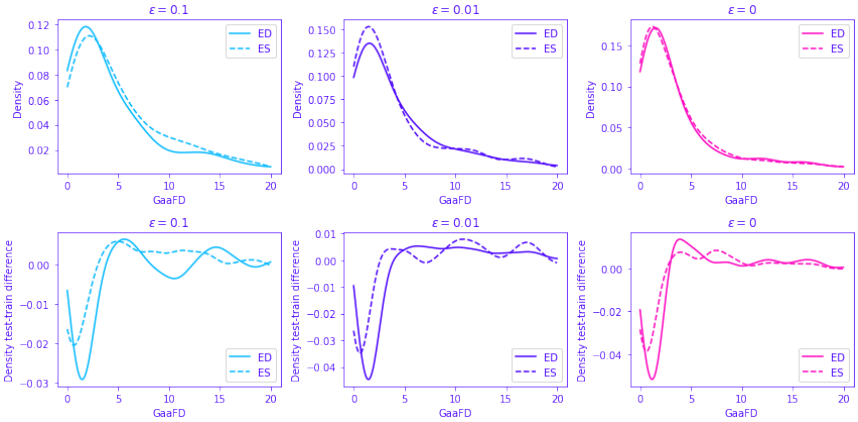


Figure 5.4: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$), only accounting for either ED- or ES-events individually. The upper row compares the performance on ED and ES for each model. The bottom row shows the difference in GaaFD-density on the test-set versus the train-set as a means to visualize model overfitting.

To compare against other state-of-the-art models we use aaFD, but as discussed in earlier sections aaFD is not defined on videos whose number of predictions does not equal the number of ground truth events. As seen in table 5.1, the best model of the three generates the correct number of predictions in 77% of the cases, the worst model 64%. Disregarding these low numbers, and/or assuming there are methods of cleaning up the predictions such that aaFD becomes valid, we can filter out the invalid predictions for each model and compare their aaFD. This is seen in 5.5. Unsurprisingly, the model with $\epsilon = 0$ who performs the best on GaaFD

also makes the fewest mistakes in number of predictions. We also see a small bump at 2 mismatched number of predictions. This may be due to the fact that the model sometimes predicts rogue frames with wrong labels, perhaps due to noise, who are quickly fixed in the following frames. This creates two events in rapid succession, as visualized in figure 5.6.

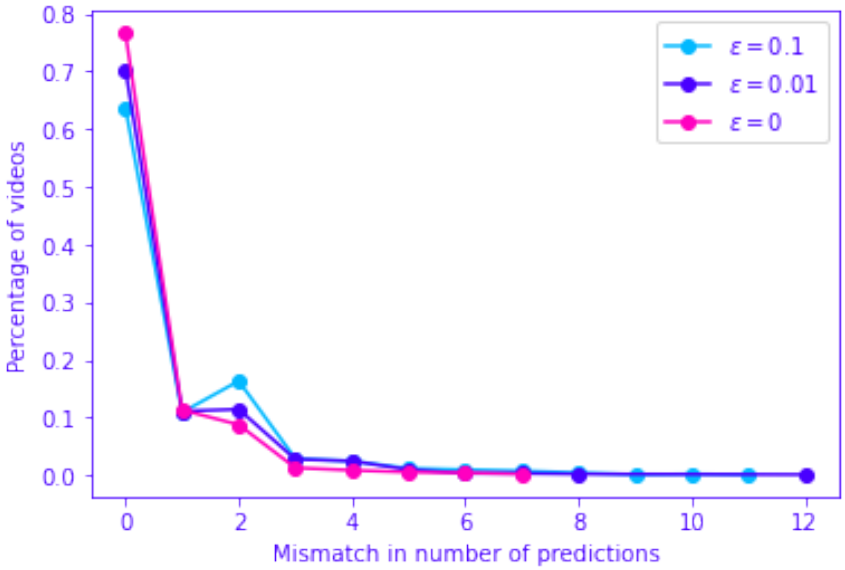


Figure 5.5: The difference between the number of predicted events and the number of ground truth events for each model. Most predictions produce the same number of predicted events as ground truth, f.ex. the model with $\epsilon = 0$ produces the correct number of events 77% of the time, which can also be seen in table 5.1.

Peeking inside the machinery of the DQN-agent, we see a potential cause of the performance discrepancy between the three models. For every frame the agent predicts the future returns of marking a frame as diastole or marking it as systole. These predictions are plotted in 5.7 and 5.8. It is apparent that the model that uses $\epsilon = 0$ is better at differentiating the value of taking either action for a given state.

TODO: Also show example of frames from video of good video and bad video with corresponding q-values.

TODO: What does the in-the-middle spikes represent in the $\text{gaafd}_{\text{bestvideosq}}$ plot?

TODO: Plot the best/worst videos as per each model's performance. Now each column is the same video, selected as the best for one of the models.

TODO: How fast is it at inference?

5.1.2 Simple- and Proximity-Based Reward Functions

The next set of experiments explore the phase detection reward functions which are able to provide a reward on every step, not only at the end of

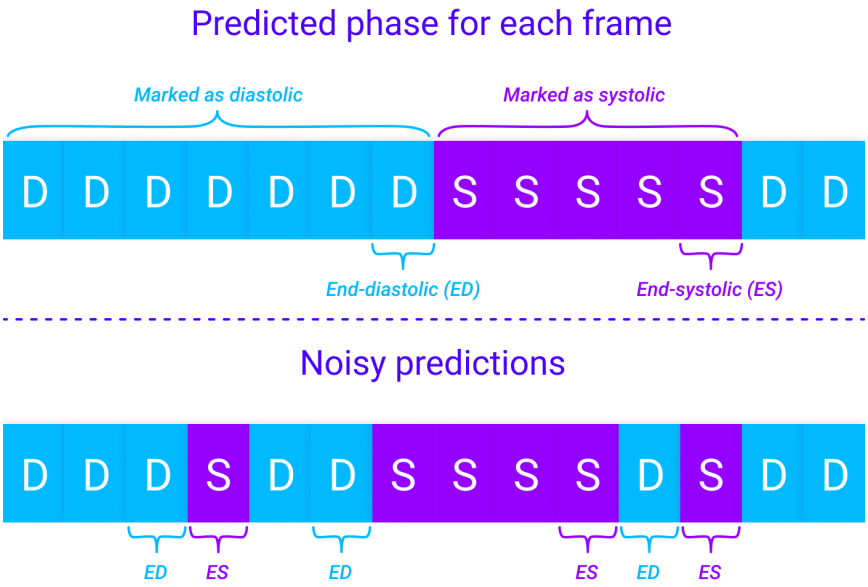


Figure 5.6: A single wrongly predicted phase that is corrected right after creates two incorrect events.

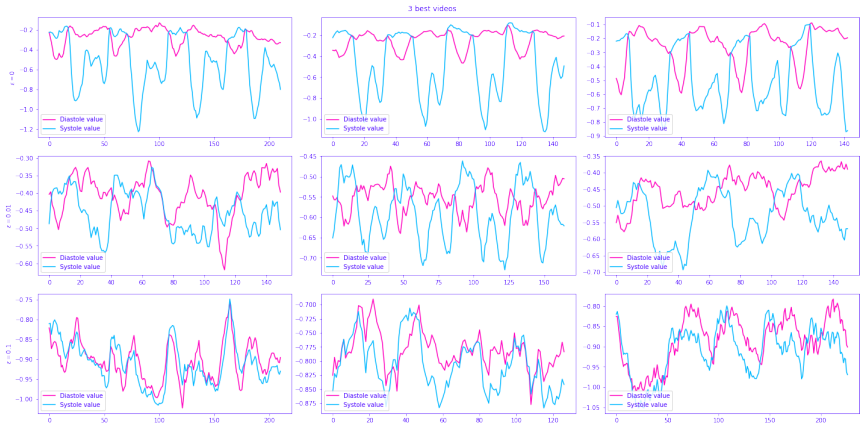


Figure 5.7: The Q-values for three of the best predicted videos for each model. Top row is the model with $\epsilon = 0$, middle row is the model with $\epsilon = 0.01$, and the bottom row is the model with $\epsilon = 0.1$. The x-axis represents tiime in the video.

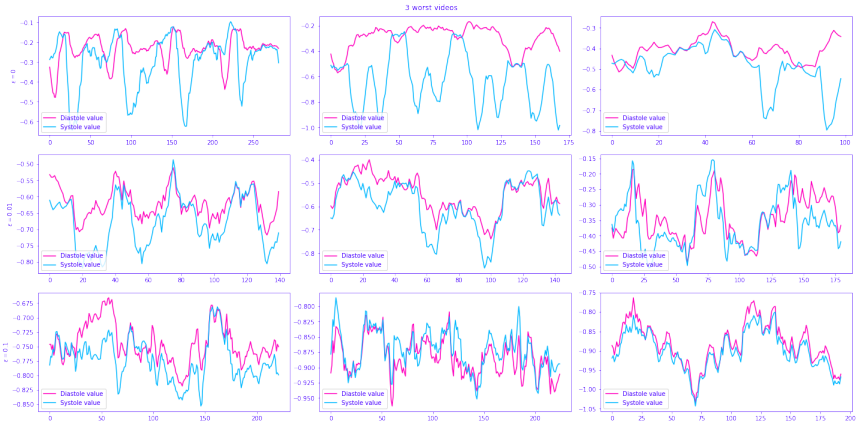


Figure 5.8: The Q-values for three of the worst predicted videos for each model. Top row is the model with $\epsilon = 0$, middle row is the model with $\epsilon = 0.01$, and the bottom row is the model with $\epsilon = 0.1$. The x-axis represents tiime in the video.

the episode as with GaaFD. Two reward functions are explored: a simple reward function R_1, as defined in equation 4.5, and an error-based reward function R_2, as defined in equation 4.6.

This makes it quite similar to a supervised regression problem where we want to learn the Q-values given an observation and an action. This is because the returns only depend on the current action and not on all the actions in an episode as we saw with the GaaFD reward function. As a result, it is assumed that the optimal discounting factor is $\gamma = 0.0$, meaning that the returns is calculated using only the immediate reward. A discount value of $\gamma > 0.0$ would make expected future returns predictions depend more on the current policy, adding noise to the target values until the policy converges.

Unless discounting is not zero there will be no need for bootstrapping and we can ignore N-step bootstrapping for these reward functions by setting $N = 1$.

Since an action does not affect future states, exploration is not as important. Instead, we can view the exploration variable ϵ as affecting how input/label pairs are sampled. An exploration value of $\epsilon = 1.0$ means that actions are sampled uniformly and a value of $\epsilon = 0.0$ means that actions are sampled based on how good it is assumed to be.

Three values are tested for the exploration hyper-parameter ϵ: $\epsilon = 0.1$, $\epsilon = 0.5$, and $\epsilon = 1.0$. The agents were trained for 200 000 SGD steps. A full list of the hyper-parameters used is listed in table 5.1.2 (most relevant ones are highlighted).

From the learning curves in figure 5.9 and figure 5.10 we see that the agent is able to learn to make correct predictions much faster than when using GaaFD as the reward function. This is very likely due to the GaaFD reward signal being much sparser than R_1 and R_2 — R_1 and R_2 simply provides information more efficiently, and with less noise, to the learner.

Hyper parameter	Value
Epsilon	$\{0.1, 0.5, 1.0\}$
Discount	0.0
N (N-step bootstrapping)	1
Target update period	100
Importance sampling exponent	0.2
Priority exponent	0.6
Number of actors	8
Min replay size	10 000
Max replay size	250 000
Samples per insert ratio	0.5
Optimizer	Adam with default parameters
Huber loss parameter	1.0
Learning rate	1^{-4}
Gradient descent steps	200 000
Batch-size	128

Compared to GaaFD, these models also reach their best performance much faster, though the performance on the validation set visibly degrades as the model starts to overfit. The exception to this is the model of the agent that has been trained using $\epsilon = 0.1$, i.e. the agent that most often greedily takes actions, which seemingly doesn't overfit that much.

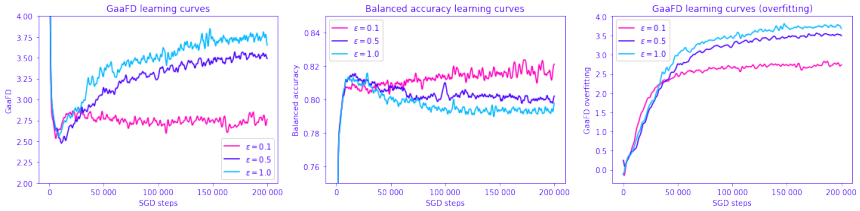


Figure 5.9: The training curves of using R_1 (simple reward) as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the training set over training time, positive values indicating overfitting on the training set. Each point in the curve is calculated on 50 random videos in the validation (or training) set. The curves have been smoothed using a gaussian filter with a kernel standard deviation of 4 to reduce noise due to the low sample size of each data point. The overfitting (right) plot has additionally been smoothed using a gaussian filter with a kernel standard deviation of 50 to make sure that overall trend is visible.

To explain the fact that the least explorative agent is the one that overfits the least on the training set it is useful to think of exploration in this case as a means of sampling the training data. No action affects future states in this environment, so the agent is not exploring to discover long-term strategies. It instead provides the learner only with samples of which it believes are the best and this, perhaps surprisingly, seems to have a regularizing effect.

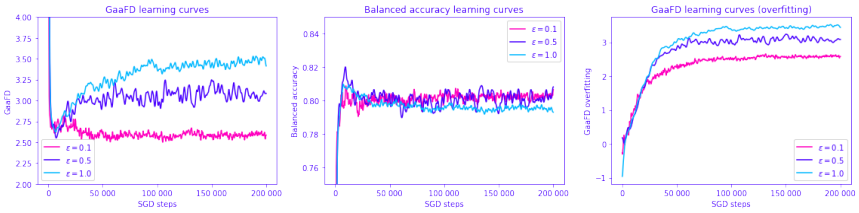


Figure 5.10: The training curves of using R_2 (proximity reward) as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the training set over training time, positive values indicating overfitting on the training set. Each point in the curve is calculated on 50 random videos in the validation (or training) set. The curves have been smoothed using a gaussian filter with a kernel standard deviation of 4 to reduce noise due to the low sample size of each data point. The overfitting (right) plot has additionally been smoothed using a gaussian filter with a kernel standard deviation of 50 to make sure that overall trend is visible.

However, despite the regularization, the agent is still able to achieve perfect accuracy and GaaFD on the training set. Figure 5.11 plots the learning curves of the agents on both the training set and the validation set.

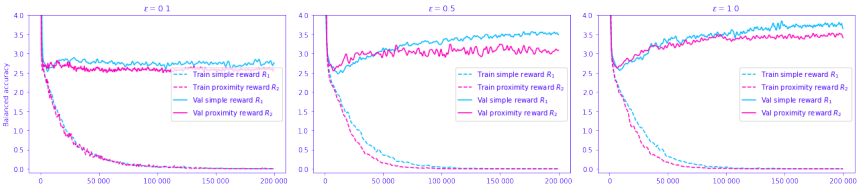


Figure 5.11: The GaaFD over training time (gradient descent steps) on the validation set (solid pink and blue line) and the training set (dashed pink and blue lines). The GaaFD on the training set reaches 0, meaning perfect predictions.

The effect of the reward function is also visible in the learning curves if we plot them for agents trained with R_1 and agents trained with R_2 together, as in figure 5.12. The blue curves are the performance of agents trained using R_1 and the pink curves are the performance of agents trained using R_2. Looking only on the right side of the plots, i.e. as the agent approaches 200 000 SGD steps, we see a clear pattern. The upper plots show GaaFD, in which lower values are better, and therefore R_2 is the better reward function, at least for agents with $\epsilon = 0.1$. The lower plots show balanced accuracy, in which higher values are better, and in this case R_1 is definitely the better reward function. So, it would seem that the reward function that was deliberately designed to be more similar to GaaFD turned out to get a better GaaFD score, even though it actually performs worse on

a metric based on accuracy. This indicates that an agent trained using R_2 reward makes more errors compared to an agent trained using R_1, but the errors are less often severe. However, this is only when looking at the right side of the plot, where the agent is already fitted to the training set. If we look at the curve at the point of the lowest GaaFD score the simple reward function R_1 outperforms R_2. This is likely due to R_2 being a more difficult function to estimate, as its values span multiple values, while R_1 can only be either 1 or -1.

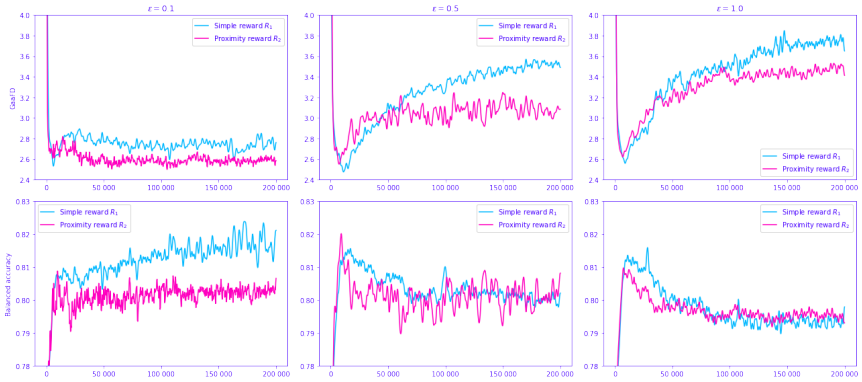


Figure 5.12: Comparison of the training curves using R_1 (simple reward) versus R_2 (proximity reward) for different values of of the exploratio parameter ϵ. Top row shows the GaaFD over training time (gradient descent steps). Bottom row shows the balanced accuracy over training time. Each column correspond to one of the agents, $\epsilon = 0.1$, $\epsilon = 0.5$, and $\epsilon = 1.0$, respectively.

Compared to the loss curves of agents trained using GaaFD as the reward, agents trained using R_1 and R_2 yield a more familiar-looking loss curve. The loss curves in figure 5.13 is seen dropping sharply in the beginning before slowly approaching some minimum. This is likely because in this case where discounting is 0.0 the value of taking an action does not depend on the current policy, and thus the distribution of returns-estimations doesn't change as the policy changes as it did when using GaaFD as the reward function.

Again, the agent with the lowest amount of exploration, plotted as the pink curve, stands out from the other models, its loss seeming to decrease faster in the beginning.

Again, we select the parameters at the step where GaaFD was lowest on the validation set for each of the three ϵ values and each of R_1 and R_2, giving us 6 trained models. Before doing so we smooth the curves using a Gaussian filter with a kernel standard deviation of 20 such that the selected parameters are not selected due to randomly sampling 50 easy videos. The SGD step of the selected model parameters and the overall performance of that model are listed in table 5.2 and table 5.3. TODO: Rewrite this, and maybe merge with the *exact* same paragraph as in GaaFD reward?

There is a lot of duplicate text here if I explain the plots once again... Maybe this information should be presented in another way.

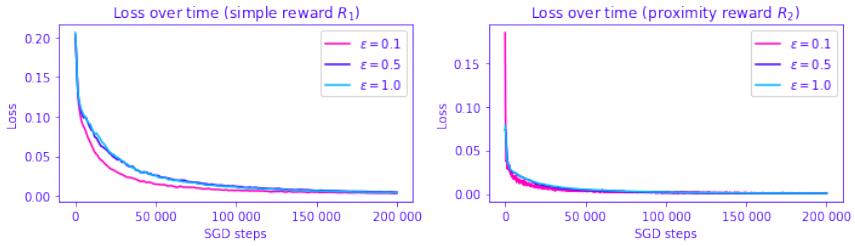


Figure 5.13: The training loss over time for different values of epsilon. Left: an agent trained using R_1 (simple reward). Right: an agent trained using R_2 (proximity reward).

Table 5.2: Performance of agents trained using R_1 as the reward function on the test dataset.

	$\epsilon = 0.1$	$\epsilon = 0.5$	$\epsilon = 1$
Best model SGD step	6 220	10 960	8 964
GaaFD	2.52	2.46	2.57
GaaFD ED	2.48	2.43	2.52
GaaFD ES	2.47	2.41	2.55
% valid aaFD	0.79	0.80	0.77
aaFD	1.71	1.69	1.69

Table 5.3: Performance of agents trained using R_2 as the reward function on the test dataset.

	$\epsilon = 0.1$	$\epsilon = 0.5$	$\epsilon = 1$
Best model SGD step	107 936	6 880	7 096
GaaFD	2.55	2.56	2.63
GaaFD ED	2.52	2.56	2.67
GaaFD ES	2.50	2.48	2.52
% valid aaFD	0.79	0.79	0.77
aaFD	1.74	1.80	1.71

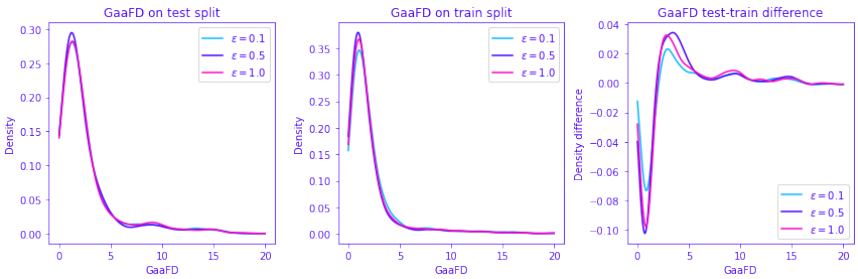


Figure 5.14: TODO

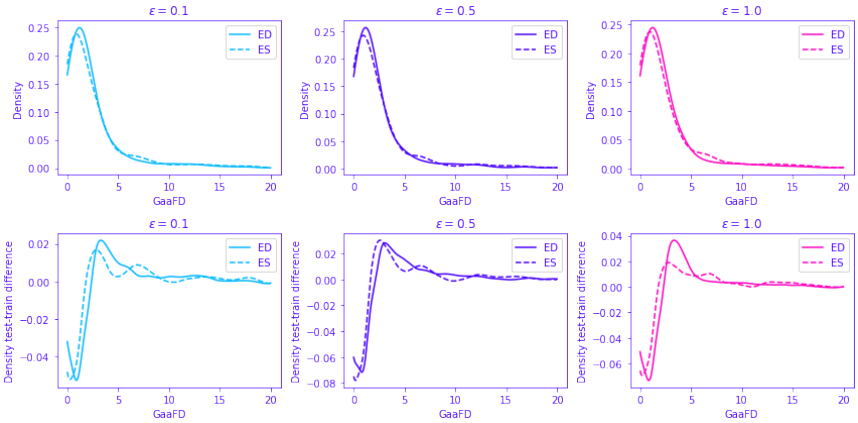


Figure 5.15: TODO

One consequence of selecting the model with $\epsilon = 0.1$ at a late time-step (107 936, table 5.3) is that the model have a much longer time to overfit on the training set. We can indeed see in figure 5.16 that the model performs incredibly well on the training set, yet there is no visible degrading on the test set compared to models with $\epsilon = 0.5$ or $\epsilon = 1.0$.

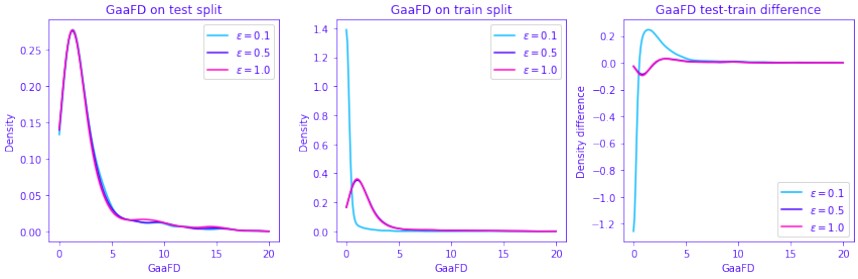


Figure 5.16: TODO

Q-values

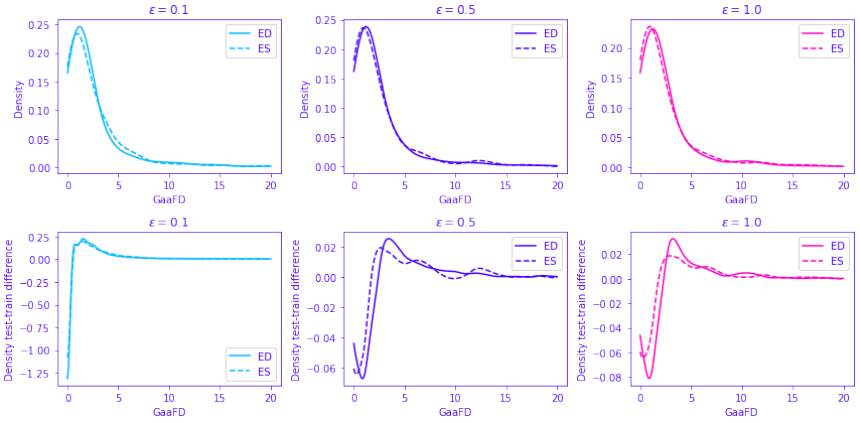


Figure 5.17: TODO

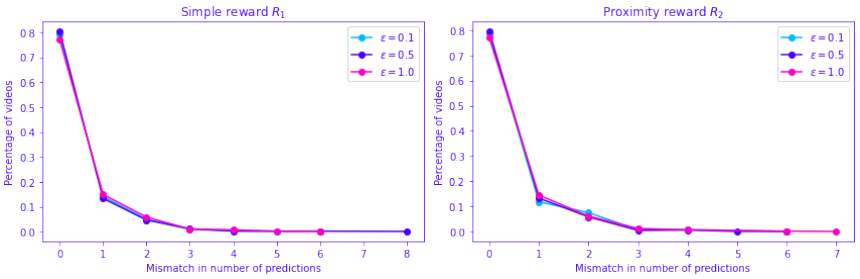


Figure 5.18: TODO

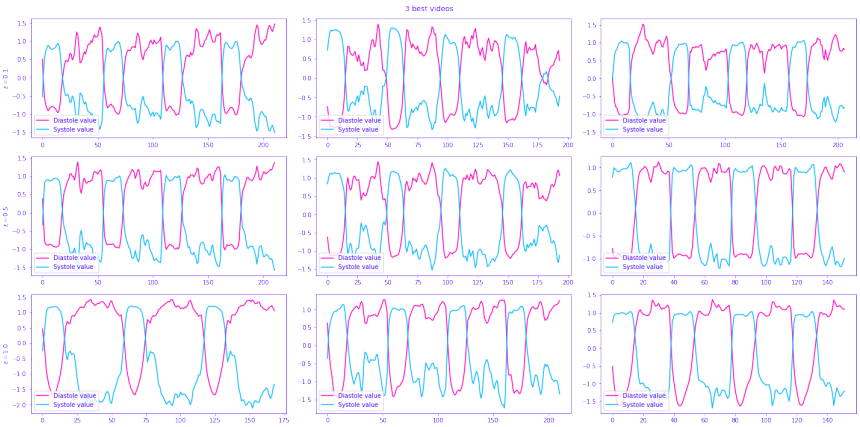


Figure 5.19: TODO

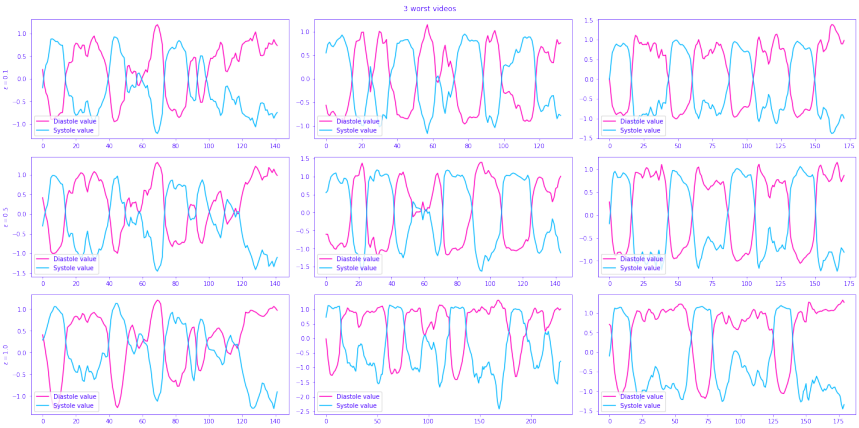


Figure 5.20: TODO

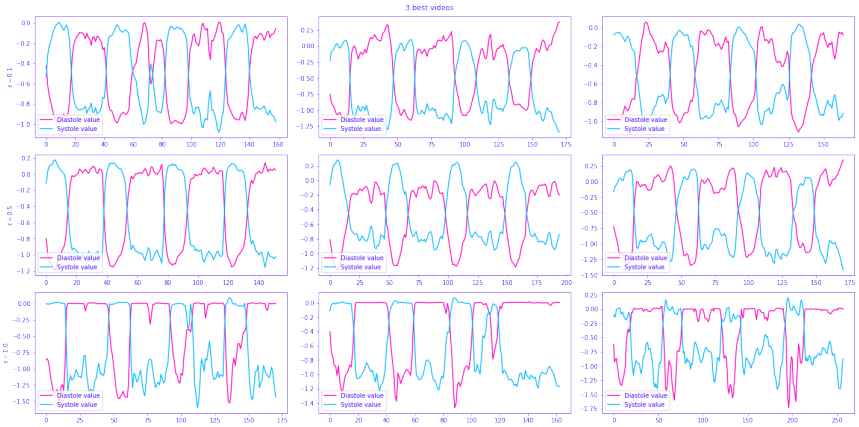


Figure 5.21: TODO

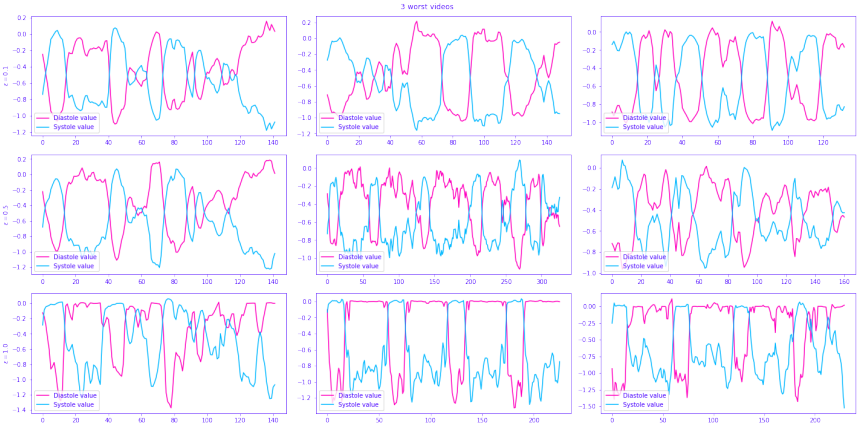


Figure 5.22: TODO

5.2 Hey, Reader! You can disregard the following sections...

TODO: Mention the deadly triad in Q-learning with function approximators.

5.3 M-Mode Binary Classification Environment

5.4 Discussion

5.5 Conclusion and Further Work

Bibliography

[1] Anas A. et al. 'Automatic Detection of the End-Diastolic and End-Systolic from 4D Echocardiographic Images'. In: *Journal of Computer Science* 11 (Jan. 2015), pp. 230–240. DOI: 10.3844/jcssp.2015.230.240.

[2] Amir Alansary et al. 'Evaluating reinforcement learning agents for anatomical landmark detection'. en. In: *Medical Image Analysis* 53 (Apr. 2019), pp. 156–164. ISSN: 1361-8415. DOI: 10.1016/j.media.2019.02.007. URL: https://www.sciencedirect.com/science/article/pii/S1361841518306121 (visited on 11/05/2021).

[3] U. Barcaro, D. Moroni and O. Salvetti. 'Automatic computation of left ventricle ejection fraction from dynamic ultrasound images'. en. In: *Pattern Recognition and Image Analysis* 18.2 (June 2008), p. 351. ISSN: 1555-6212. DOI: 10.1134/S1054661808020247. URL: https://doi.org/10.1134/S1054661808020247 (visited on 31/05/2021).

[4] Marc G. Bellemare, Will Dabney and Rémi Munos. 'A Distributional Perspective on Reinforcement Learning'. In: *arXiv:1707.06887 [cs, stat]* (July 2017). arXiv: 1707.06887. URL: http://arxiv.org/abs/1707.06887 (visited on 18/05/2021).

[5] Marc G. Bellemare et al. 'Unifying Count-Based Exploration and Intrinsic Motivation'. In: *arXiv:1606.01868 [cs, stat]* (Nov. 2016). arXiv: 1606.01868. URL: http://arxiv.org/abs/1606.01868 (visited on 19/05/2021).

[6] *Cardiovascular diseases*. en. URL: https://www.who.int/westernpacific/health-topics/cardiovascular-diseases (visited on 12/05/2021).

[7] Albin Cassirer et al. 'Reverb: A Framework For Experience Replay'. In: *arXiv:2102.04736 [cs]* (Feb. 2021). arXiv: 2102.04736. URL: http://arxiv.org/abs/2102.04736 (visited on 29/03/2022).

[8] Saeed Darvishi et al. 'Measuring Left Ventricular Volumes in Two-Dimensional Echocardiography Image Sequence Using Level-set Method for Automatic Detection of End-Diastole and End-systole Frames'. In: *Research in Cardiovascular Medicine* 2.1 (Feb. 2013), pp. 39–45. ISSN: 2251-9572. DOI: 10.5812/cardiovascmed.6397. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4253755/ (visited on 31/05/2021).

[9] Fatemeh Taheri Dezaki et al. 'Deep Residual Recurrent Neural Networks for Characterisation of Cardiac Cycle Phase from Echocardiograms'. en. In: *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*. Ed. by M. Jorge Cardoso et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 100–108. ISBN: 978-3-319-67558-9. DOI: 10.1007/978-3-319-67558-9_12.

[10] Adrian Meidell Fiorito et al. 'Detection of Cardiac Events in Echocardiography Using 3D Convolutional Recurrent Neural Networks'. In: *2018 IEEE International Ultrasonics Symposium (IUS)*. ISSN: 1948-5727. Oct. 2018, pp. 1–4. DOI: 10.1109/ULTSYM.2018.8580137.

[11] Meire Fortunato et al. 'Noisy Networks for Exploration'. In: *arXiv:1706.10295 [cs, stat]* (July 2019). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295 (visited on 18/05/2021).

[12] Florin C. Ghesu et al. 'An Artificial Agent for Anatomical Landmark Detection in Medical Images'. en. In: *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*. Ed. by Sebastien Ourselin et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 229–237. ISBN: 978-3-319-46726-9. DOI: 10.1007/978-3-319-46726-9_27.

[13] Florin C. Ghesu et al. 'Robust Multi-scale Anatomical Landmark Detection in Incomplete 3D-CT Data'. en. In: *Medical Image Computing and Computer Assisted Intervention MICCAI 2017*. Ed. by Maxime Descoteaux et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 194–202. ISBN: 978-3-319-66182-7. DOI: 10.1007/978-3-319-66182-7_23.

[14] Florin C. Ghesu et al. 'Towards intelligent robust detection of anatomical structures in incomplete volumetric data'. en. In: *Medical Image Analysis* 48 (Aug. 2018), pp. 203–213. ISSN: 1361-8415. DOI: 10.1016/j.media.2018.06.007. URL: https://www.sciencedirect.com/science/article/pii/S1361841518304092 (visited on 10/05/2021).

[15] Parisa Gifani et al. 'Automatic detection of end-diastole and end-systole from echocardiography images using manifold learning'. eng. In: *Physiological Measurement* 31.9 (Sept. 2010), pp. 1091–1103. ISSN: 1361-6579. DOI: 10.1088/0967-3334/31/9/002.

[16] Parisa Gifani et al. 'Noise reduction of echocardiography images using Isomap algorithm'. In: *2011 1st Middle East Conference on Biomedical Engineering*. ISSN: 1558-2531. Feb. 2011, pp. 150–153. DOI: 10.1109/MECBME.2011.5752087.

[17] H. P. van Hasselt (Hado) and Intelligent and autonomous systems. 'Double Q-learning'. en. In: *Advances in Neural Information Processing Systems*. The MIT Press, Dec. 2010. URL: https://ir.cwi.nl/pub/16889 (visited on 15/05/2021).

[18] Hado van Hasselt, Arthur Guez and David Silver. 'Deep Reinforcement Learning with Double Q-learning'. In: *arXiv:1509.06461 [cs]* (Dec. 2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461 (visited on 15/05/2021).

[19] Matteo Hessel et al. 'Rainbow: Combining Improvements in Deep Reinforcement Learning'. In: *arXiv:1710.02298 [cs]* (Oct. 2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298 (visited on 16/05/2021).

[20] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-term Memory'. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[21] Matt Hoffman et al. 'Acme: A Research Framework for Distributed Reinforcement Learning'. In: *arXiv:2006.00979 [cs]* (June 2020). arXiv: 2006.00979. URL: http://arxiv.org/abs/2006.00979 (visited on 29/03/2022).

[22] Tollef Struksnes Jahren et al. 'Estimation of End-Diastole in Cardiac Spectral Doppler Using Deep Learning'. In: *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control* 67.12 (Dec. 2020). Conference Name: IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control, pp. 2605–2614. ISSN: 1525-8955. DOI: 10.1109/TUFFC.2020.2995118.

[23] Nadjia Kachenoura et al. 'Automatic detection of end systole within a sequence of left ventricular echocardiographic images using autocorrelation and mitral valve motion detection'. eng. In: *Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual International Conference* 2007 (2007), pp. 4504–4507. ISSN: 2375-7477. DOI: 10.1109/IEMBS.2007.4353340.

[24] Bin Kong et al. 'Recognizing End-Diastole and End-Systole Frames via Deep Temporal Regression Network'. en. In: *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*. Ed. by Sebastien Ourselin et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 264–272. ISBN: 978-3-319-46726-9. DOI: 10.1007/978-3-319-46726-9_31.

[25] Julian Krebs et al. 'Robust Non-rigid Registration Through Agent-Based Action Learning'. en. In: *Medical Image Computing and Computer Assisted Intervention MICCAI 2017*. Ed. by Maxime Descoteaux et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 344–352. ISBN: 978-3-319-66182-7. DOI: 10.1007/978-3-319-66182-7_40.

[26] Elisabeth S. Lane et al. 'Multibeat echocardiographic phase detection using deep neural networks'. en. In: *Computers in Biology and Medicine* 133 (June 2021), p. 104373. ISSN: 0010-4825. DOI: 10.1016/j.compbiomed.2021.104373. URL: https://www.sciencedirect.com/science/article/pii/S0010482521001670 (visited on 12/05/2021).

[27] Rui Liao et al. 'An Artificial Agent for Robust Image Registration'. In: *arXiv:1611.10336 [cs]* (Nov. 2016). arXiv: 1611.10336. URL: http://arxiv.org/abs/1611.10336 (visited on 12/05/2021).

[28] Mada Razvan O. et al. 'How to Define End-Diastole and End-Systole?' In: *JACC: Cardiovascular Imaging* 8.2 (Feb. 2015). Publisher: American College of Cardiology Foundation, pp. 148–157. DOI: 10.1016/j.jcmg.2014.10.010. URL: https://www.jacc.org/doi/full/10.1016/j.jcmg.2014.10.010 (visited on 15/05/2021).

[29] Gabriel Maicas et al. 'Deep Reinforcement Learning for Active Breast Lesion Detection from DCE-MRI'. en. In: *Medical Image Computing and Computer Assisted Intervention MICCAI 2017*. Ed. by Maxime Descoteaux et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 665–673. ISBN: 978-3-319-66179-7. DOI: 10.1007/978-3-319-66179-7_76.

[30] Volodymyr Mnih et al. 'Human-level control through deep reinforcement learning'. en. In: *Nature* 518.7540 (Feb. 2015). Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236 (visited on 11/05/2021).

[31] Ify R Mordi et al. 'Efficacy of noninvasive cardiac imaging tests in diagnosis and management of stable coronary artery disease'. In: *Vascular Health and Risk Management* 13 (Nov. 2017), pp. 427–437. ISSN: 1176-6344. DOI: 10.2147/VHRM.S106838. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5701553/ (visited on 12/05/2021).

[32] David Ouyang et al. *EchoNet-Dynamic: a Large New Cardiac Motion Video Data Resource for Medical Machine Learning*. en. 2019. URL: https://www.semanticscholar.org/paper/EchoNet-Dynamic%3A-a-Large-New-Cardiac-Motion-Video-Ouyang-He/44bfcf2409c0826584c7c409b6a2fcf8c9910c88 (visited on 04/03/2022).

[33] Tom Schaul et al. 'Prioritized Experience Replay'. In: *arXiv:1511.05952 [cs]* (Feb. 2016). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952 (visited on 16/05/2021).

[34] David Silver et al. 'Mastering the game of Go with deep neural networks and tree search'. en. In: *Nature* 529.7587 (Jan. 2016). Number: 7587 Publisher: Nature Publishing Group, pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: http://www.nature.com/articles/nature16961 (visited on 21/05/2021).

[35] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. en. Google-Books-ID: uWV0DwAAQBAJ. MIT Press, Nov. 2018. ISBN: 978-0-262-35270-3.

[36] Fatemeh Taheri Dezaki et al. 'Cardiac Phase Detection in Echocardiograms With Densely Gated Recurrent Neural Networks and Global Extrema Loss'. In: *IEEE Transactions on Medical Imaging* 38.8 (Aug. 2019). Conference Name: IEEE Transactions on Medical Imaging, pp. 1821–1832. ISSN: 1558-254X. DOI: 10.1109/TMI.2018.2888807.

[37] Oriol Vinyals et al. 'Grandmaster level in StarCraft II using multi-agent reinforcement learning'. en. In: *Nature* 575.7782 (Nov. 2019). Number: 7782 Publisher: Nature Publishing Group, pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://www.nature.com/articles/s41586-019-1724-z (visited on 21/05/2021).

[38] Athanasios Vlontzos et al. 'Multiple Landmark Detection using Multi-Agent Reinforcement Learning'. In: *arXiv:1907.00318 [cs]* (July 2019). arXiv: 1907.00318. URL: http://arxiv.org/abs/1907.00318 (visited on 11/05/2021).

[39] Ziyu Wang et al. 'Dueling Network Architectures for Deep Reinforcement Learning'. In: *arXiv:1511.06581 [cs]* (Apr. 2016). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581 (visited on 16/05/2021).

[40] Fan Yang et al. 'Launchpad: A Programming Model for Distributed Machine Learning Research'. In: *arXiv:2106.04516 [cs]* (June 2021). arXiv: 2106.04516. URL: http://arxiv.org/abs/2106.04516 (visited on 29/03/2022).

[41] Baichuan Yuan et al. 'Machine learning for cardiac ultrasound time series data'. In: *Medical Imaging 2017: Biomedical Applications in Molecular, Structural, and Functional Imaging*. Vol. 10137. International Society for Optics and Photonics, Mar. 2017, p. 101372D. DOI: 10.1117/12.2254704. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10137/101372D/Machine-learning-for-cardiac-ultrasound-time-series-data/10.1117/12.2254704.short (visited on 31/05/2021).

[42] Matthew D. Zeiler and Rob Fergus. 'Visualizing and Understanding Convolutional Networks'. In: *arXiv:1311.2901 [cs]* (Nov. 2013). arXiv: 1311.2901. URL: http://arxiv.org/abs/1311.2901 (visited on 31/05/2021).

[43] Aston Zhang et al. *Dive into Deep Learning*. 2020.

[44] S. Kevin Zhou et al. 'Deep reinforcement learning in medical imaging: A literature review'. In: *arXiv:2103.05115 [cs, eess]* (Mar. 2021). arXiv: 2103.05115. URL: http://arxiv.org/abs/2103.05115 (visited on 10/05/2021).