UNIVERSITY
OF OSLO

Master's thesis

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

Magnus Dalen Kvalevåg

60 study points

Department of Informatics
The Faculty of Mathematics and Natural Sciences

Abstract

The thesis explores ways of formulating the problem of ED-/ES-frame detection as a reinforcement learning problem, and whether there are any benefits in doing so. Of special interest is the design of the reward function. The standard performance metric for the task of ED-/ES-frame detection is the Average Absolute Frame Difference (aaFD), defined as the average error between predicted and ground truth events. A Generalized aaFD (GaaFD) metric is designed such that it can be used when there is a different number of predicted and ground truth events. GaaFD is then used directly as the reward function R_{GaaFD}, defined such that the agent receives the reward at the end of an episode. ED-/ES-frame detection is also framed as a phase classification problem, whereby the agent predicts the phase of each frame in a video. The reward functions R_{simple} and $R_{proximity}$ is defined on this formulation of the problem, providing a less sparse reward signal than that of R_{GaaFD}. R_{simple} is designed to be simple, giving a reward of either 1 or -1 for correct and incorrect phase predictions, respectively. $R_{proximity}$ is designed to bring the reward function closer to the semantics of GaaFD by giving a penalty proportional to the distance to the nearest frame with the predicted phase. Additionally, two formulations of the environment are explored: Binary Classification Environment (BCE), designed to be a direct reformulation of a supervised binary classification task, and M-Mode Binary Classification Environment (MMBCE), designed to provide the agent with the ability to explore the environment using synthetic m-mode imaging. Because of time constraints, MMBCE was only preliminary explored, yet the results of which indicate that the problem is too complex for the current setup and requires more work before we can draw any conclusions on its feasiability.

Experiments with the different reward functions show that the agent is able to learn from all of them, even the very-sparsely defined R_{GaaFD}. The agent is able to predict the correct number of frame-events on 77% of the videos in the test-dataset for all reward functions. Of the videos with the current number of predicted frame-events the best agents scored 2.43, 1.69, and 1.71 for R_{GaaFD}, R_{simple}, and $R_{proximity}$, respectively.

It is concluded that there are multiple ways of formulating the problem of ED-/ES-frame detection as a reinforcement learning problem, but not all formulations are equal. Reward sparsity degrades the performance of the agent significantly, and by defining the reward on phase detection instead the results improve. There are indications that designing a reward function to be closer in semantics to the key performance metric, such as $R_{proximity}$,

may improve the performance of the agent on that metric, though the final results show that R_{simple} performs slightly better. By regarding exploration as a way to sample data to the network the results indicate that a very low amount of exploration has a regularizing effect on the network, resulting in less overfitting. It is believed that the same effect can be reproduced in a regular supervised learning setting and that reinforcement learning does not provide substantial value to the problem of ED-/ES-frame detection.

Contents

List of Figures

List of Tables

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 [**noauthor_cardiovascular_nodate**]. 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. It is also more flexible; 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 [**mordi_efficacy_2017**], 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 [**lane_multibeat_2021**]. 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 it can help reduce annotation 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 [**silver_mastering_2016**] or becoming among the 0.2% best players in the world in the video game Starcraft II [**vinyals_grandmaster_2019**]. However, RL can do more than just play games, and many medical imaging applications also show promising potential [**zhou_deep_2021**].

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.

We pose the following research questions:

- **Is it possible to use Reinforcement Learning for the task of ED-/ES-frame detection?**
- **How does the formulation of the problem as a Reinforcement Learning problem affect the performance of the model?**

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 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 **[mada_razvan_o_how_2015]**:

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).

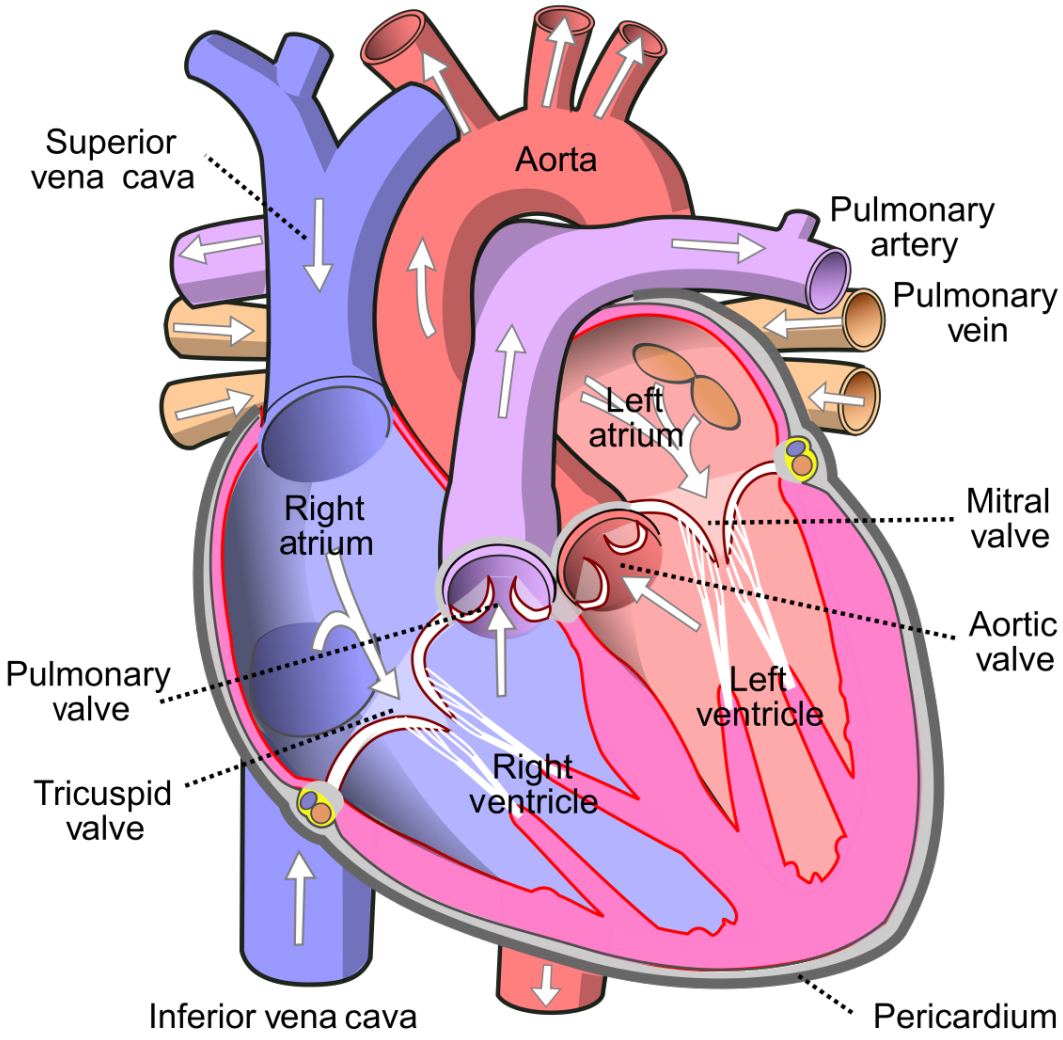


Figure 2.1: An illustration of the heart. The heart has two sides, each side having two chambers. Source: https://en.wikipedia.org/wiki/Atrium_(heart).

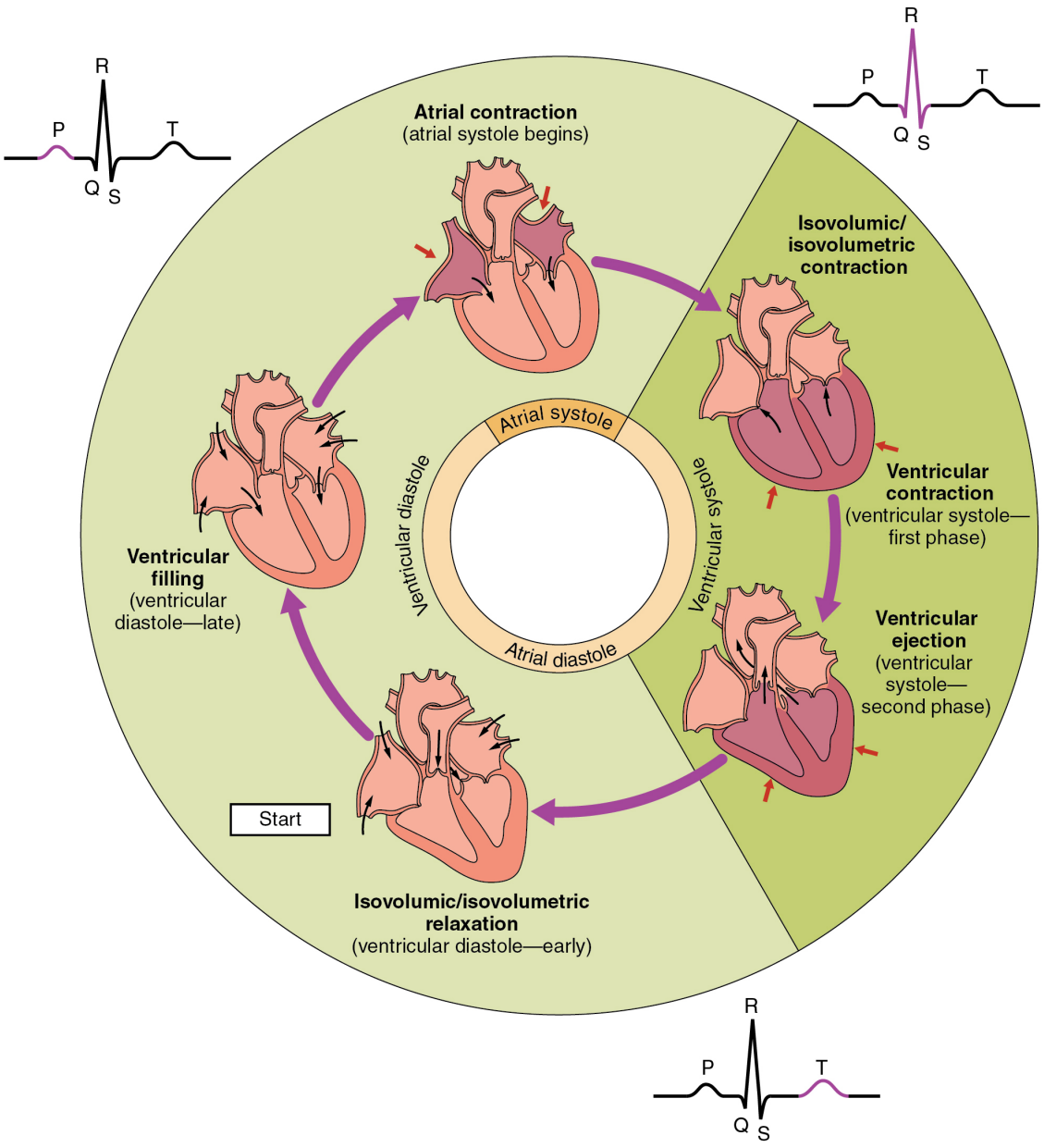


Figure 2.2: The cardiac cycle illustrated with the direction of blood flow and pressure from and into the atria and ventricles. Source: https://en.wikipedia.org/wiki/Heart.

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 [**mada_razvan_o_how_2015**]. Acquiring an ECG signal also requires applying electrodes to the patient, which is not ideal in emergency settings.

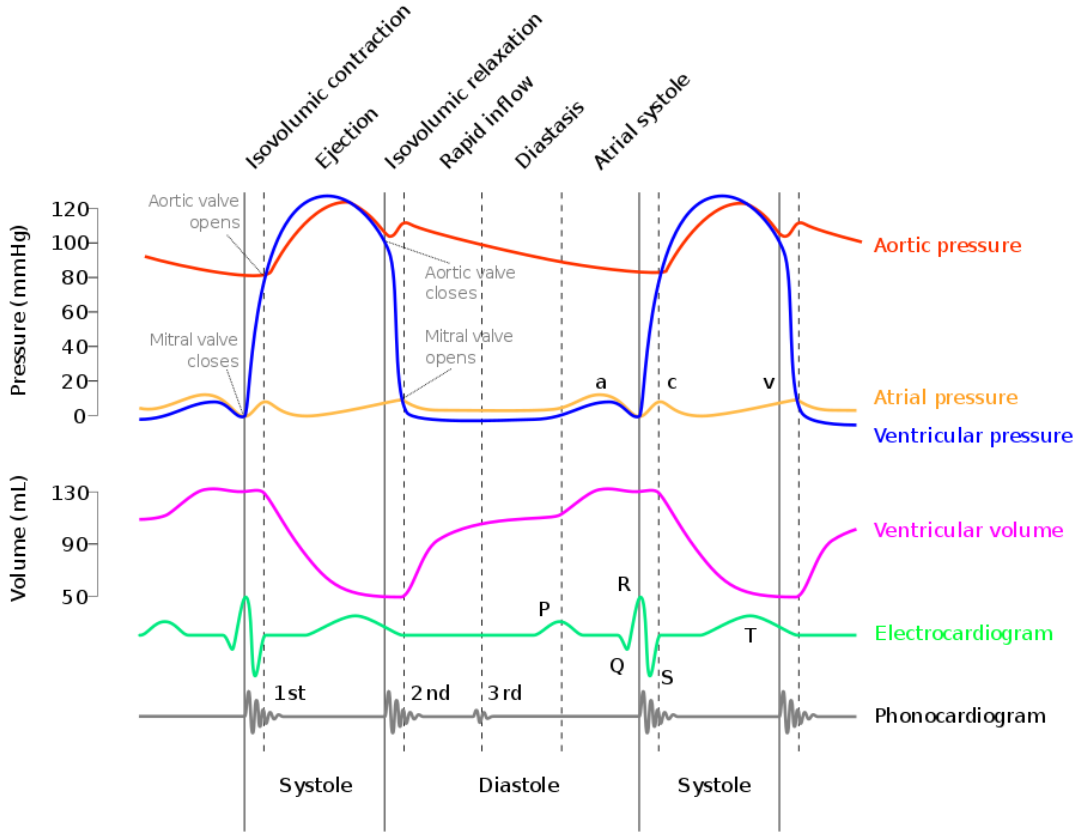


Figure 2.3: The Wiggers diagram desccribes the different phases of the cardiac cycle, as well as what they represent in different measurements. Source https://en.wikipedia.org/wiki/Wiggers_diagram.

2.2 What is Ultrasound?

What we as humans perceive as sound are simply vibrations of the particles that surround us. Sound waves push particles together, creating an increase in pressure. Particles in an area of high pressure moves to areas of lower

pressure, and this creates a chain reaction where a pressure field moves through particles, illustrated in figure 2.4. This is called wave propagation. Sound is simply waves of pressure propagating through a medium.

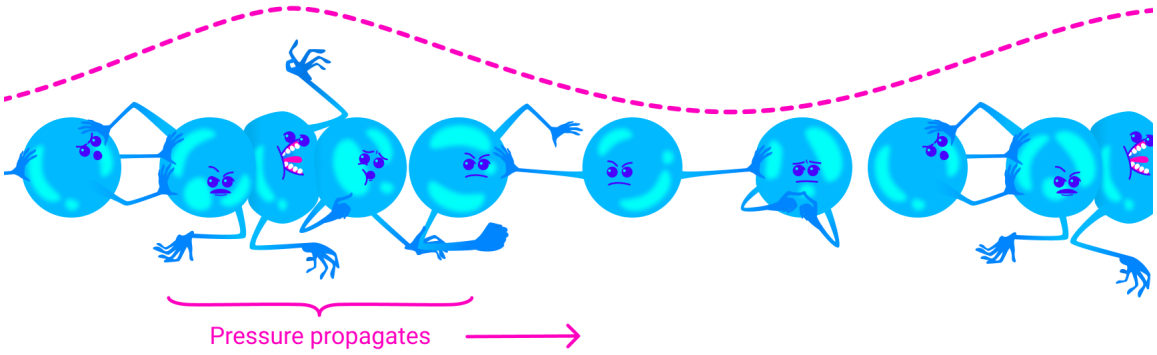


Figure 2.4: A pressure wave moves through a medium by pushing particles in a medium close together. The particles pushes back as the pressure increases, making the pressure field move further on. Warning: this image is just a representation of how particles interact — real particles don't look like this.

2.2.1 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, as visualized in figure 2.5. 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.

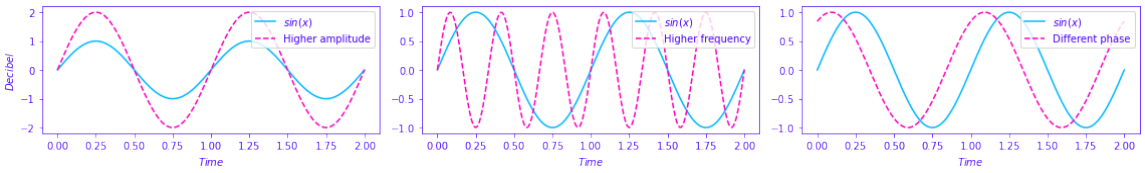


Figure 2.5: The left-most plot shows two basic waves where one has twice the amplitude. The middle plot shows two basic waves where one has a higher frequency. The right-most plot shows two basic waves that have different phases.

A basic wave means a sine wave in this context. Every sound can be represented as a sum of sine waves, and every sound can be transformed into its frequency spectrum. As seen in figure 2.6, the frequency spectrum of a sine wave is just a single spike. Adding together to sounds has the same effect as adding together their frequency spectrums.

Real world sounds often have much more complex frequency spectrums, as many more sine waves are needed to represent it. When a pi-

ano 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.

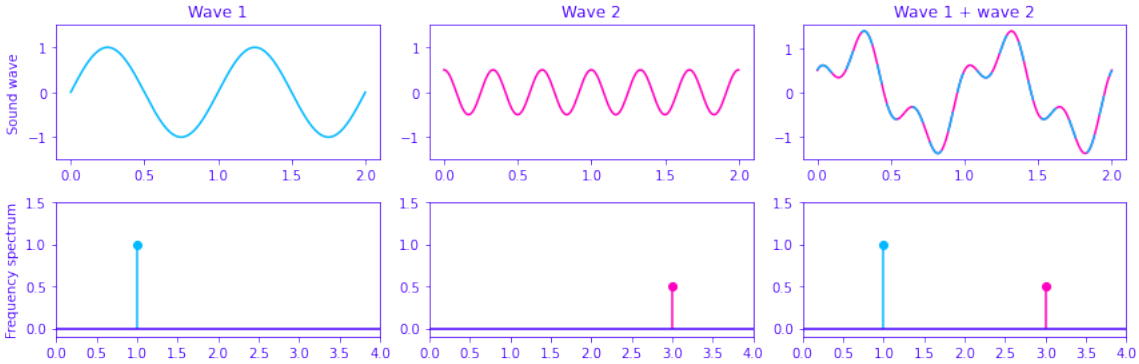


Figure 2.6: Adding two sounds together means that their frequency spectrums are also added together.

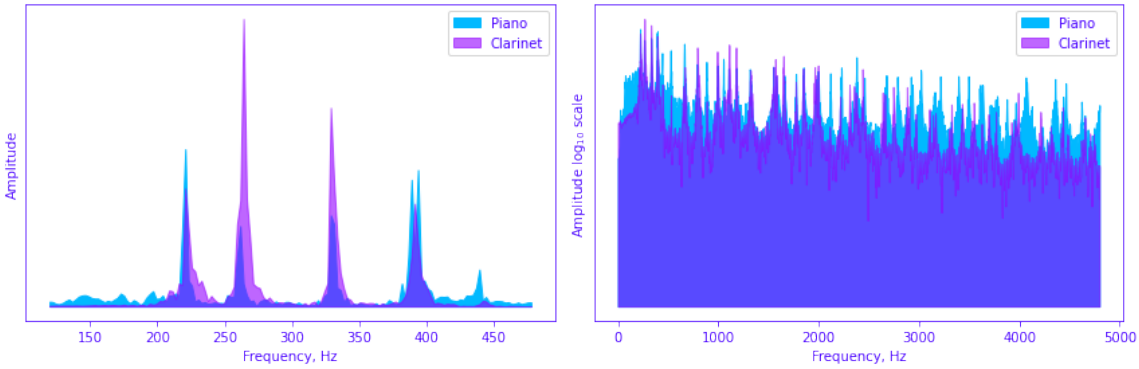


Figure 2.7: It's the overtones that makes two instruments sound different, even while they are playing the same notes. To the left is the frequency spectrum of a piano and a clarinet from 150 to 450 hertz. To the right is the same frequency spectrum from 0 to 5000 hertz, in $\log_{10}$ scale. Both instruments 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.

2.2.2 Attributes of the Medium

Another important aspect of sound is the medium in which it travels through. Properties such as the speed of sound, density, attenuation and non-linearity affect how a sound wave propagates through its medium. 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, as visualized in figure 2.8. 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. In water, pressure waves propagate faster in higher pressure. The pressure may be cause by the wave itself, in which case the shape of the wave may change, as visualized in figure 2.9.

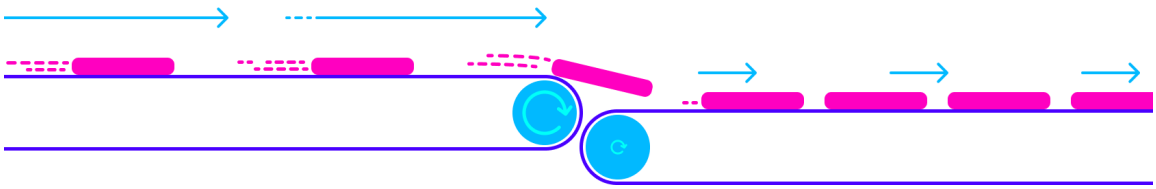


Figure 2.8: Even though the rate of packages per second stays the same, the distance between each package decreases when arriving on a slower conveyor belt. This is analogous to a sound wave propagating through a medium where the speed of sound changes. Even though the frequency is the same, the wavelength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

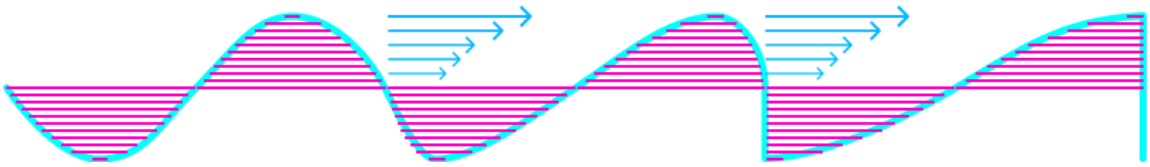


Figure 2.9: In a medium with nonlinearity the higher-pressure parts of a wave propagates faster than lower-pressure parts. Over time, the higher-pressure parts will "catch up" to the lower-pressure parts, and what started as a sine wave will start to resemble a sawtooth wave.

An important concept is "acoustic impedance" which is a measure of how much resistance the wave encounters while propagating through the medium. Acoustic impedance is a function of the speed of sound and density. When a wave propagates out of one medium and into another medium that has a different acoustic impedance, a fraction of the energy is reflected back. 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.

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

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

2.3 Echocardiography

Light is a signal that does not penetrate very far into the body, which is why we can't simply gaze into each other's hearts. We could, however, imagine a universe where light penetrates all the way, 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.

Table 2.1 lists the speed of sound and acoustic impedance Z of some substances. Notice how for example there is a large difference in acoustic impedance between air and soft tissue. This means that if there is air between the sound wave transmitter and the body then most of the energy will be reflected back by the skin. To reduce this effect, ultrasound gel, which has a very similar acoustic impedance to soft tissue, is applied between the body and the sound wave transmitter. Notice also the difference in acoustic impedance between bone and soft tissue. This has consequences for what we can image in the body, as bones such as the ribcage act as shields to the sound waves.

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, as visualized in figure 2.10. 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 and receiving 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 through interpolation. To create an image we could simply repeat this process for every point in the image.

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

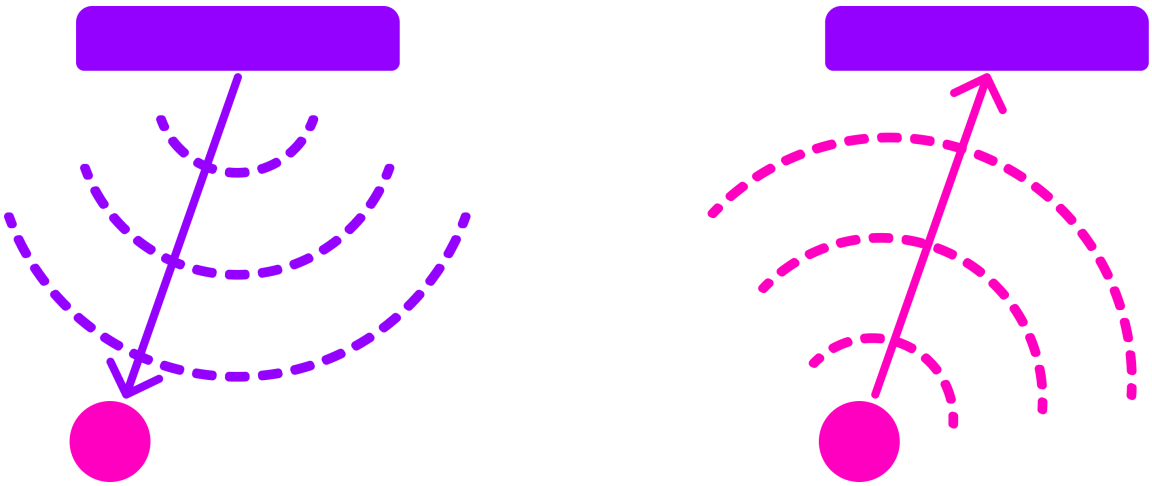


Figure 2.10: By measuring the time between sending a signal and receiving it back from a reflector we can approximate how far away the reflector is — given that we know the approximate speed of sound.

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 there will be a correlation between signals across receivers at the reflecting object.

By utilizing multiple sender elements that can send sound waves independently of each other, we are able to shape the wavefront as we wish. This lets 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 the Huygens-Fresnel principle 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. Time delays are to sound waves like a lens is to a magnifying glass. An example of this has been visualized in figure 2.11.

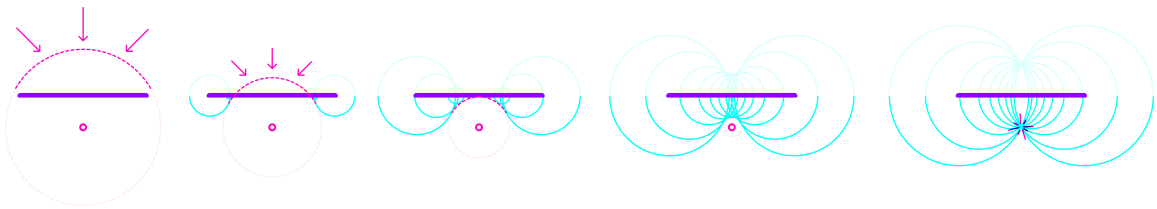


Figure 2.11: Because of the Huygens-Fresnel principle, we can create a desired wavefront by creating spherical waves at each sender element the moment the imagined wavefront would hit it. The dashed, pink curve represents the imagined desired wavefront as it approaches the sender elements marked by the 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 2.12. Another method is to transmit unfocused plane waves. A single transmit creates an unfocused image of the scatterers in the medium, but multiple transmits in different directions may be compounded to create an image of comparable quality to those of focused transmits [**montaldo_coherent_2009**].

B-mode imaging provide 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

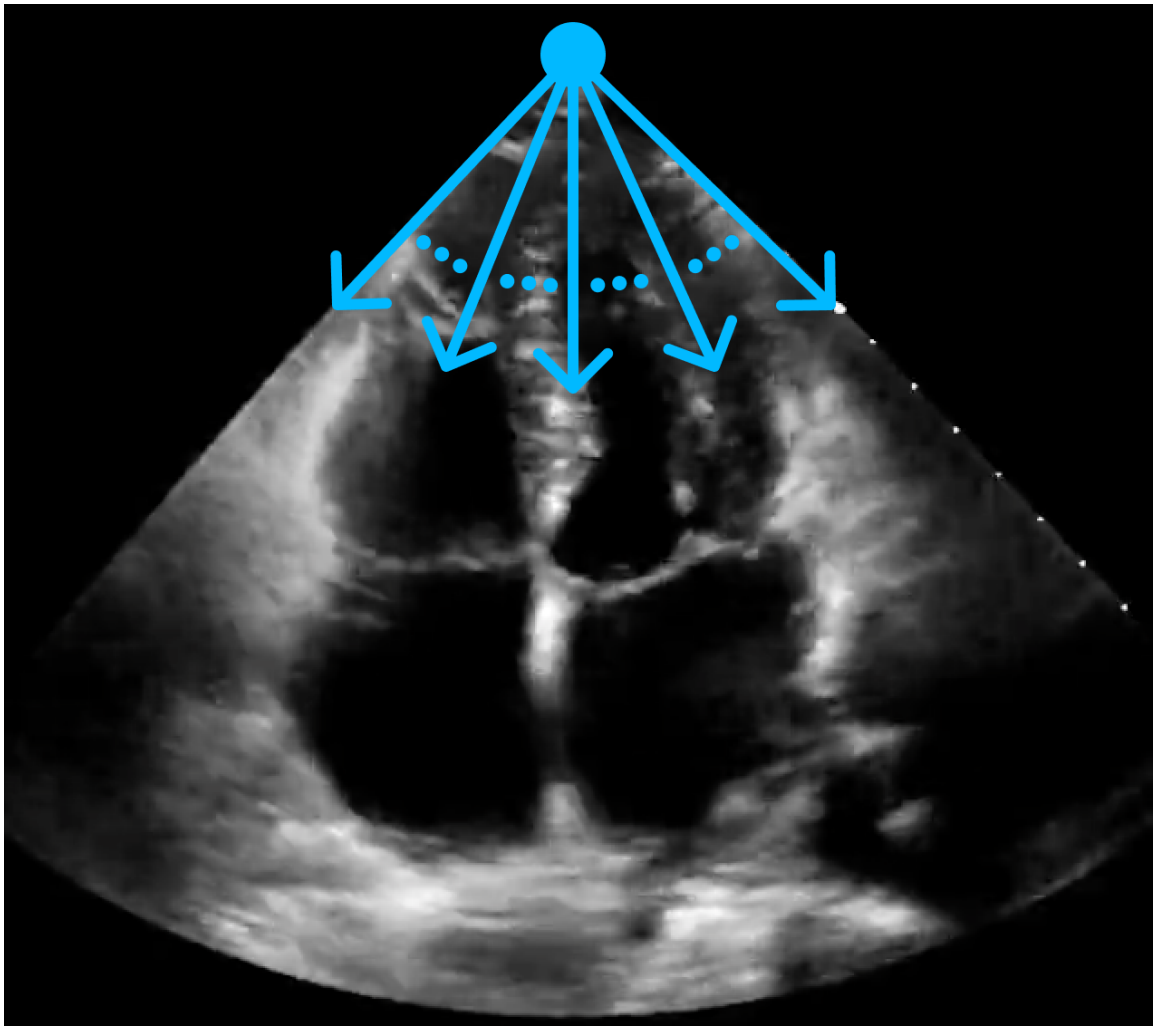


Figure 2.12: Imaging along different angles from a common starting point creates a sector scan.

can be made and received back in a short period of time, but it does have consequences for the temporal resolution.

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, giving it a higher temporal resolution compared to B-mode imaging, but at the cost of only focusing in a single direction. 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, as seen in figure 2.13.

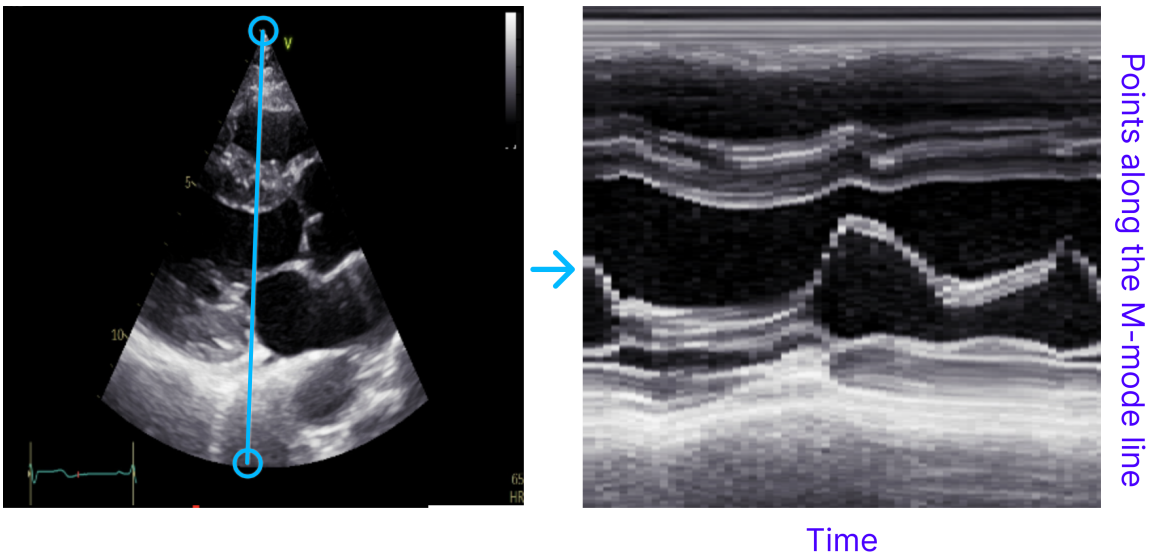


Figure 2.13: A still of a sector scan (left) and the M-mode (right) of the same video for the indicated blue line.

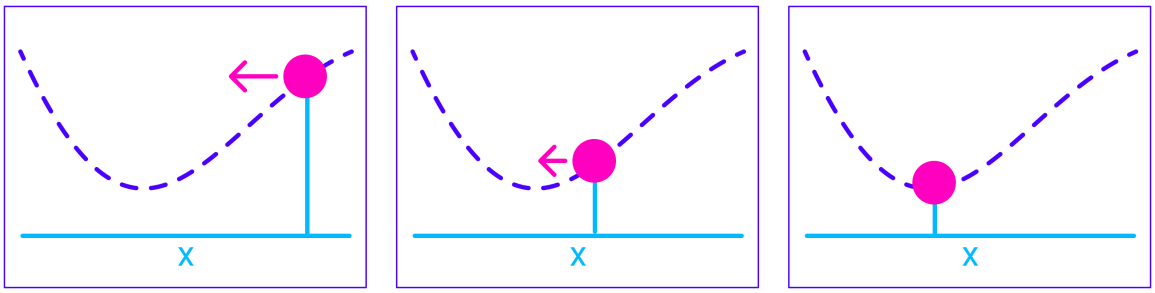
2.4 Deep Learning

2.4.1 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, if it is differentiable, then we can calculate its slope at a given point. The slope is also 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 2.4.1. 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 scales to 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 10 000 parameters. It is no longer possible to visualize this high-dimensional parameter space as we did in 2.4.1, but the principles still hold, and gradient descent still works the same way.

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 quantify the error of the model so that we can minimize it. As long as both the model and the loss function is differentiable 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. One important aspect is the shape of the gradient and whether it contains a lot of local minima.

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. We may consider the data set as part of the loss function, as we want to minimize the error of the model on these specific data. 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, perhaps surprisingly, it is often better at generalizing on the dataset than using gradient descent on the whole dataset at once.

Another aspect of great importance is to instill what's called inductive bias into the model; that is, implicit knowledge about the task at hand. Some models capture implicit knowledge about the problem at hand better than others, and some important features are explored in the next section.

2.4.2 Deep Neural Networks

Fully connected layers is just a matrix multiplication between the input x and a weigth matrix w, often with an additional bias b added, such that $\hat{y} = wx + b$. The shape of the matrix w determines the number of output neurons it produces.

Multiple fully connected layers can be stacked to create a more complex network. However, beneath the hood each layer is performing matrix operations, which are linear operations, and no matter how linear layers are stacked they can only represent linear relationships. To allow the network to represent more complex, non-linear relationships we need to additionally add non-linearity to the network. This is usually done using activation functions. Examples of activation functions are the sigmoid function, seen in equation 2.5, or ReLU, seen in 2.6.

$$S(X) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

$$ReLU = max(0, x) \tag{2.6}$$

Using just two fully connected layers separated by non-linear activation functions one could represent any arbitrary function, given that one includes enough neurons [TODO: Cite "Multilayer Feedforward Networks are Universal Approximators"]. That does mean that they are the right tool for every job.

Fully connected layers combine every input with every output. This does not take advantage of the spatial locality of images. In an image a given pixel is often more related to pixel who lie closer to it. Convolutional layers take advantage of this by applying filters to an image, with each filter only processing a small part of the image at a time. The filters are often small matrices that is applied everywhere in an image.

Each filter has a width and height which determine how many pixels are included in one application of the filter. One may also affect how the filters are applied to an image through stride, which determine how much the sliding window of the filter "skips" for each application. For example a stride of 2 means that the filter moves two pixels for each application. Another hyper-parameter is dilation, which affect the spacing between pixels for an individual application. A dilation of 2 means that every other pixel is ignored when applying the filter.

Another popular layer is the recurrent layer. The recurrent layer is a way of processing sequential data, such as temporal data. Recurrent layers have a unit with hidden state that is applied at every time step. One popular version of a recurrent layer is LSTM [TODO: citation].

2.4.3 Optimization Process

For each iteration of gradient descent we update the model's parameters a small step in the opposite direction in order to minimize it. It is important to only update them in a small step each time, otherwise they

may overshoot and in the worst case cause the model's performance to diverge. For standard SGD we choose how much to update the parameters using the hyper-parameter α which is often a low number between 0 and 1.

Using a low α means that we don't update the parameters too much when the gradient is steep, but it also means that the parameters are updated very little when the gradient is near-flat. In addition, we may encounter flat regions of the gradient which can be hard to move past regardless of the chosen value of α. For these reasons other optimizers have been developed, such as ADAM [TODO: cite].

TODO: Explain ADAM

2.4.4 Overfitting and Regularization

The goal of machine learning is to train models that generalize to data samples outside of the training set. When we optimize a model on a given data distribution we risk making the model specialize too much on that specific distribution. When the model performs significantly better on the data it has been trained on versus unseen data we say that it has overfit.

One way to reduce the chance of overfitting is to use regularizers. Regularizers either augment the training data or put additional constraints on the optimization process such that the the model is likely to overfit. One example is to use data augmentations: random transformations on the training data. In practice, this increases the training dataset as more data are added to it. One could also augment the loss function itself by adding the magnitude of all the parameters, thereby encouraging the network to be less dependent on a small number of features.

2.4.5 Supervised, Semi-Supervised, and Unsupervised Learning

One way of designing the loss function is to define it as the difference between the predicted values from the model and ground truths that were labeled beforehand. Learning methods that use these kinds of loss functions are generally called Supervised Learning, as if a "supervisor" tells the model what the right answer ought to have been.

If we don't have access to ground truth labels we can instead define the loss in other ways. TODO: semi-supervised and unsupervised

2.4.6 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 takes future decisions into account when deciding on the next action. 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) [**sutton_reinforcement_2018**], 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 a_t: $\pi(a|s) \in [0, 1]$.

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, as seen in equation 2.7. T marks the timestep where the episode ends.

$$G_t = r_{t+1} + r_{t+1} + r_{t+2} + \ldots + r_T \tag{2.7}$$

However, some tasks are not episodic, which means that they may run forever. For environments with potentially endless rewards, the returns G may become infinite, making the optimization problem intractable. To solve this problem we include *discounting* to the returns, as seen in equation 2.8. γ is the *discount rate* and is a number in the range $[0, 1]$. If $\gamma < 1$, then future rewards counts for less in the full returns, and as the number of steps into the future approaches inifity, the corresponding rewards approaches 0. Discounting guarantees that non-episodic tasks converges to optimal solutions, while also giving a mechanism for preferring more immediate rewards compared to future rewards.

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + \ldots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_t + \gamma G_{t+1} \end{aligned} \tag{2.8}$$

One way to select action is to try to predict the value of the following state after taking an aciton. 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 at subsequent steps. For this setup, 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.

TODO: Mention epsilon-greedy policy and UCB

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, as seen in equation 2.9. $(r_{t+1} + \gamma V(s_{t+1}))$ 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.

$$V(s_t) \leftarrow V(s_t) + \alpha[(r_{t+1} + \gamma V(s_{t+1})) - V(s_t)] \tag{2.9}$$

To be able to use $V(s)$ for making a decision, the agent needs knowledge about the transition function $P(s_{t+1}|s_t, a_t)$. 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. In equation 2.10 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.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[(r_{t+1} + \gamma max_a Q(s_{t+1}, a)) - Q(s_t, a_t)] \tag{2.10}$$

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 [**zhou_deep_2021**]. 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[**mnih_human-level_2015**]. 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 [**mnih_human-level_2015**]. 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.

We assume that there exists a way to approximate the value of states in a much more compressed way. This can be done through function approximation [**sutton_reinforcement_2018**], 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 may also allow 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 [**sutton_reinforcement_2018**]. Policy gradient methods may more naturally model continuous action spaces as it outputs a distribution of action probabilities instead of the values of a discrete set of actions. As seen in later chapters, the RL formulations used in this thesis all use discrete action spaces, and only action-value methods are considered for this thesis.

Deep Q-Network

The modified Q-learning algorithm was termed Deep Q-Network [**mnih_human-level_2015**] (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 [**liao_artificial_2016**]. 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 2.14.

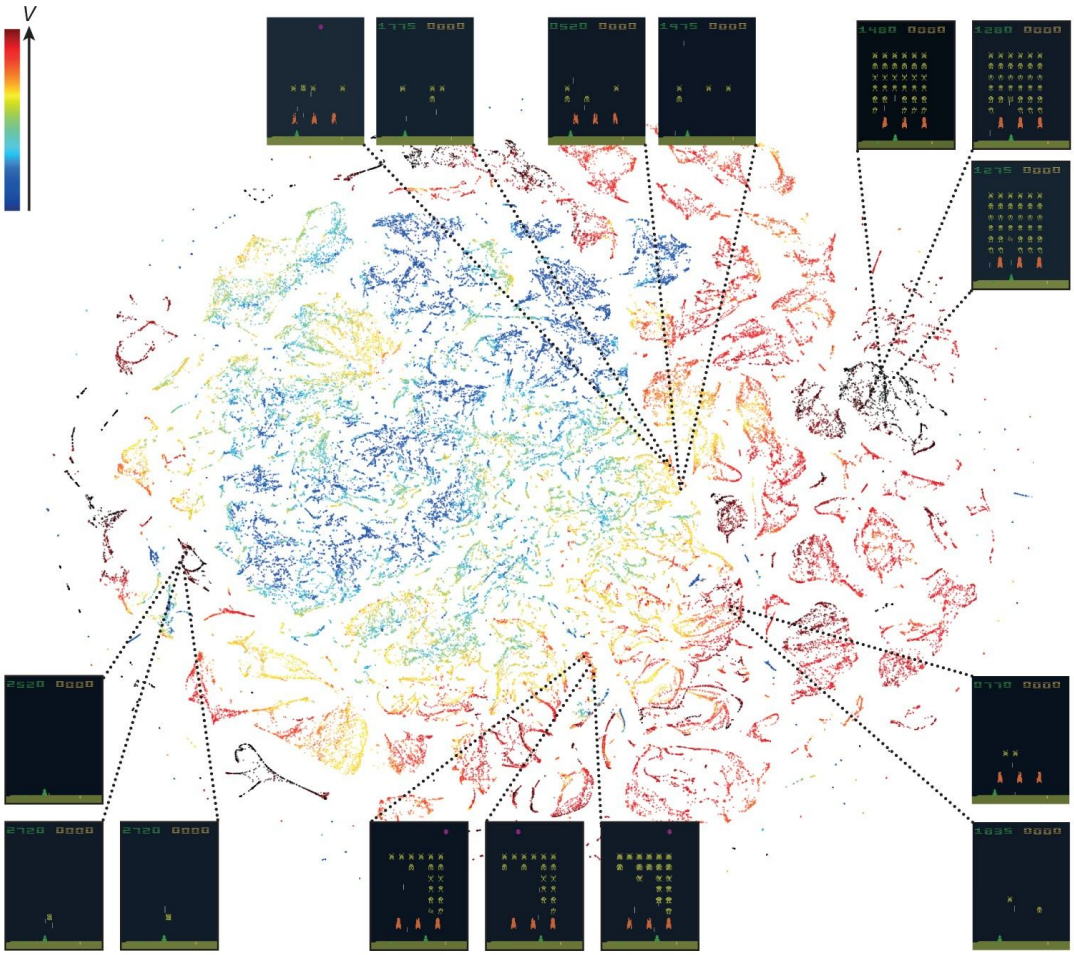


Figure 2.14: A figure from **[mnih_human-level_2015]** that shows a two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced while playing Space Invaders. The points are coloured according to the state values predicted by DQN for the corresponding game states. The states rendered in the top right, which are of almost full of enemy ships, and the states rendered in the bottom left, which are nearly empty, have similar predicted state values even though they are visually dissimilar, because the agent has learned that completing a screen leads to a new screen full of enemy ships.

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, and 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 value , 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 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 in the Q-learning update adds a delay between the time an update to Q is made and the time it affects the targets, 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[**h_p_van_hasselt_hado_double_2010**]. 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:

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

This can be rewritten as:

$$r_{t+1} + \gamma Q^A(s_{t+1}, argmax_a Q^B(s_{t+1}, a)) \tag{2.11}$$

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[**van_hasselt_deep_2015**].

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[**schaul_prioritized_2016**].

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[**wang_dueling_2016**]. The combination of the state-value estimate and the advantage yields the Q values:

$$Q(s,a) = V(s) + A(s,a) \tag{2.12}$$

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.

$$Q(s,a) = V(s) + A(s,a) - \frac{\sum_a A(s,a)}{N_{actions}} \tag{2.13}$$

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[**sutton_reinforcement_2018**].

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:

$$\sum_{k=0}^{N-1} \gamma^k r_{t+k+1} + \gamma^N max_a(Q(s_{t+N}, a)) \tag{2.14}$$

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.

$$r_{t+1} + \gamma max_a Q(s_{t+1}, a) = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n max_a(Q(s_{t+n}, a)), \text{ iff n=1} \tag{2.15}$$

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[**bellemare_distributional_2017**]. 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 non-stationary 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[**bellemare_unifying_2016**]. 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[**fortunato_noisy_2019**]. 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[**hessel_rainbow_2017**] algorithm combines six such improvements:

1. Double DQN[**van_hasselt_deep_2015**]
2. Prioritized replay[**schaul_prioritized_2016**]
3. Dual DQN[**wang_dueling_2016**]
4. Multi-step learning[**sutton_reinforcement_2018**]
5. Distributional RL[**bellemare_distributional_2017**]
6. Noisy DQN[**fortunato_noisy_2019**]

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.

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

2.5.1 ED-/ES-Detection

One early attempt for detecting the ED and ES frames took advantage of the rapid mitral valve opening during early diastole [**kachenoura_automatic_2007**]. 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

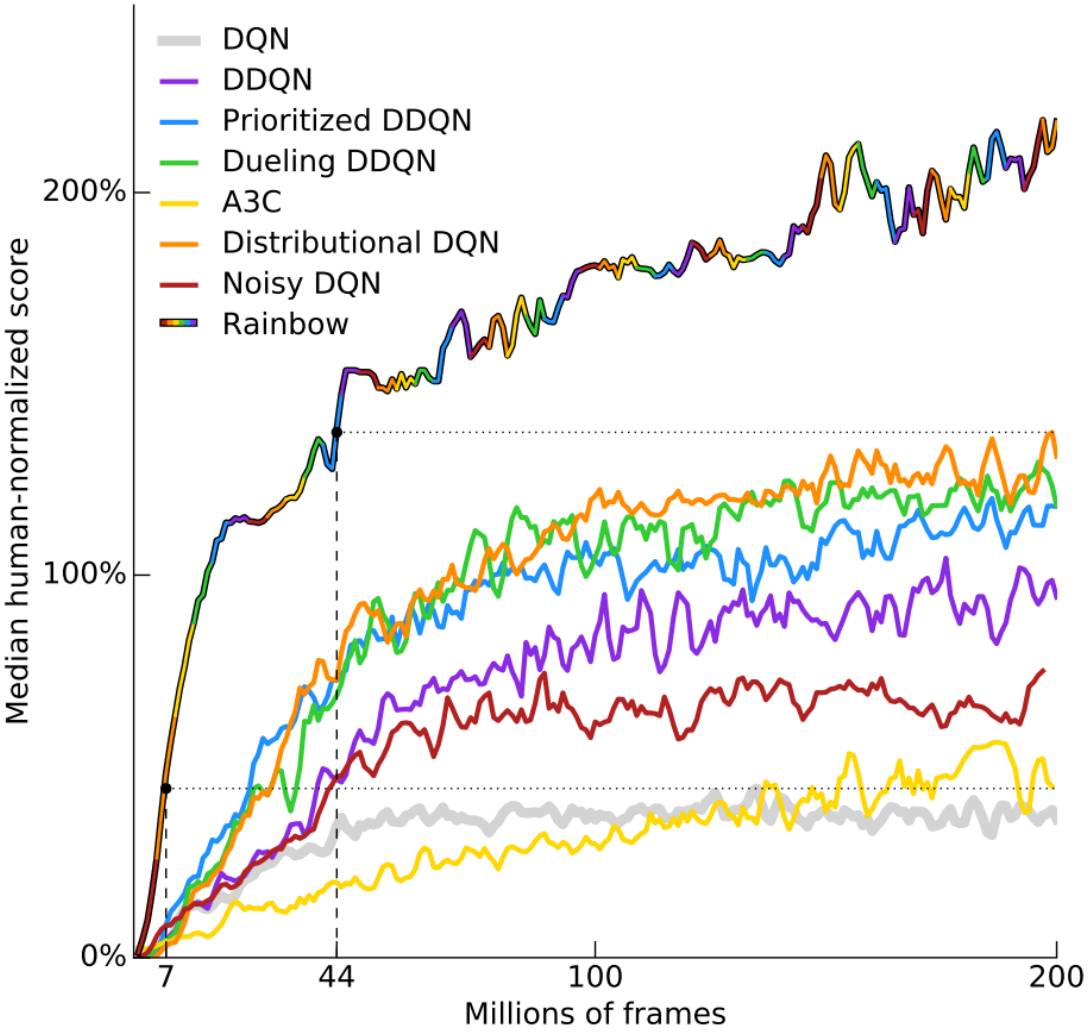


Figure 2.15: A figure from [**hessel_rainbow_2017**] showing the median performance of multiple modified DQN agents compared to human performance across 57 Atari games. After 200 million frames, all modifications show an improvement over regular DQN, but together (Rainbow), they perform significantly better than any one single improvement. Curves are smoothed with a moving average over 5 points.

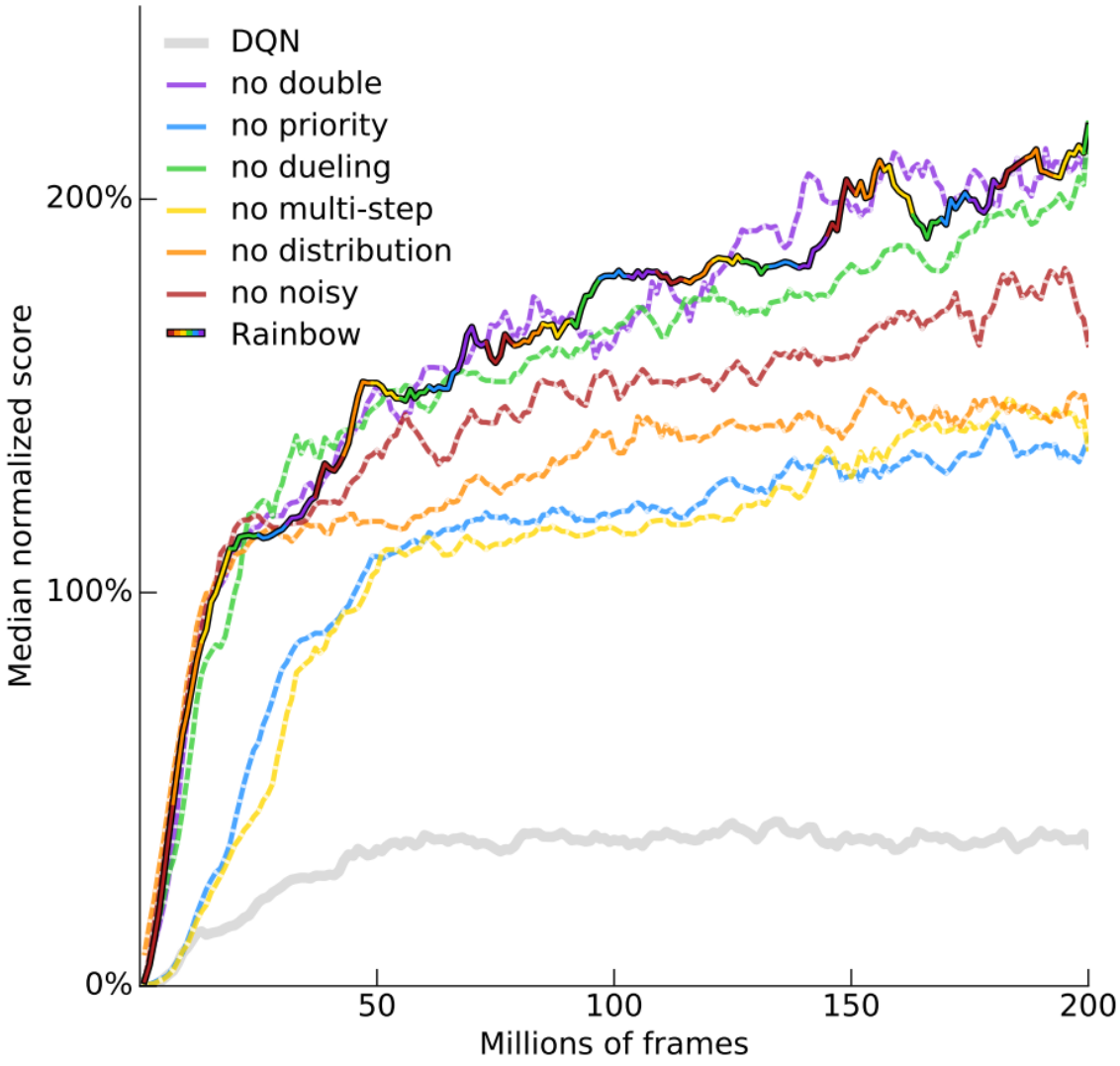


Figure 2.16: A figure from [**hessel_rainbow_2017**] visualizing an ablation study of the various DQN modifications (dashed lines). Dashed lines that are close to the rainbow line indicates that the corresponding DQN modification does not add much benefit to the overall agent, or is overshadowed by other modifications. The three most important modifications according to the ablation study are N-step bootstrapping (multi-step), distributional Q-learning, and prioritized replay.

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. 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 [**gifani_automatic_2010**]. 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 [**gifani_noise_2011**], 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 [**yuan_machine_2017**]. 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.

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 [**barcaro_automatic_2008**] [**darvishi_measuring_2013**] [**a_automatic_2015**]. 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

	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 2.17: Comparison between NMF, LLE and ISOMAP results for all 99 cases in the apical 4 view, taken from [**yuan_machine_2017**].

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 [**kong_recognizing_2016**]. The combined network was trained on cardiac MRI data, and it used a Zeiler-Fergus model [**zeiler_visualizing_2013**] for the CNN and an LSTM [**hochreiter_long_1997**] 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 [**dezaki_deep_2017**], and then again by swapping it out for a DenseNet [**taheri_dezaki_cardiac_2019**], 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 [**fiorito_detection_2018**]. 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 [**jahren_estimation_2020**]. 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 [**lane_multibeat_2021**]. 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 [**zhou_deep_2021**]. The next section will introduce RL, and it is followed by some examples of how it has been applied to medical imaging.

2.5.2 Reinforcement Learning in Medical Imaging

RL has seen many medical imaging applications in the last decade, especially in the last five years [**zhou_deep_2021**]. 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. [**ghesu_towards_2018**] [**ghesu_robust_2017**] [**ghesu_artificial_2016**]. 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 [**ghesu_towards_2018**].

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 [**ghesu_robust_2017**] and [**ghesu_artificial_2016**], 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 [**mnih_human-level_2015**]. 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 [**alansary_evaluating_2019**]. The formulation of the problem into state, actions, and reward function remains mostly the same as in [**ghesu_robust_2017**] and [**ghesu_artificial_2016**], 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 [**vlontzos_multiple_2019**]. 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 [**liao_artificial_2016**] [**krebs_robust_2017**], object/lesion localization and detection [**maicas_deep_2017**], and more [**zhou_deep_2021**].

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 [**liao_artificial_2016**]. 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 [**krebs_robust_2017**]. 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 [**maicas_deep_2017**]. 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.

This work serves as inspiration for this thesis and help guide our own RL formulations for the task of ED-/ES-frame detection.

Chapter 3

Datasets

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

3.1 Echonet-Dynamic Dataset

The Echonet-Dynamic Dataset[**ouyang_echonet-dynamic_2019**] 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

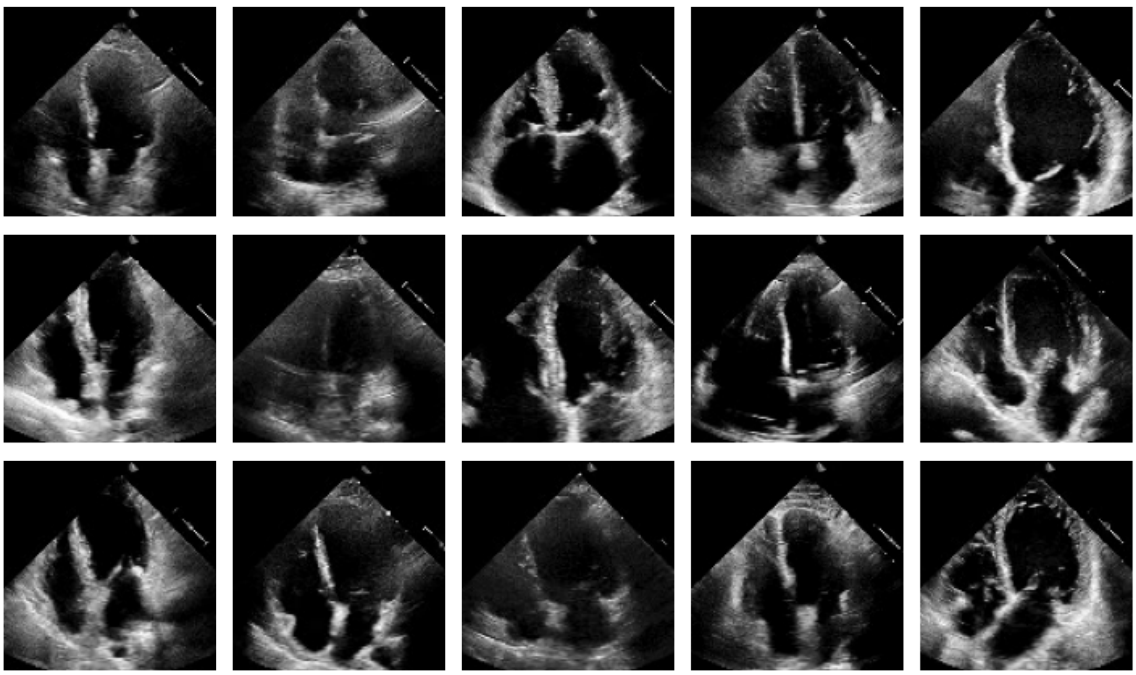


Figure 3.1: The first frames of 15 randomly sampled videos from the Echonet dataset.

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 that 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.

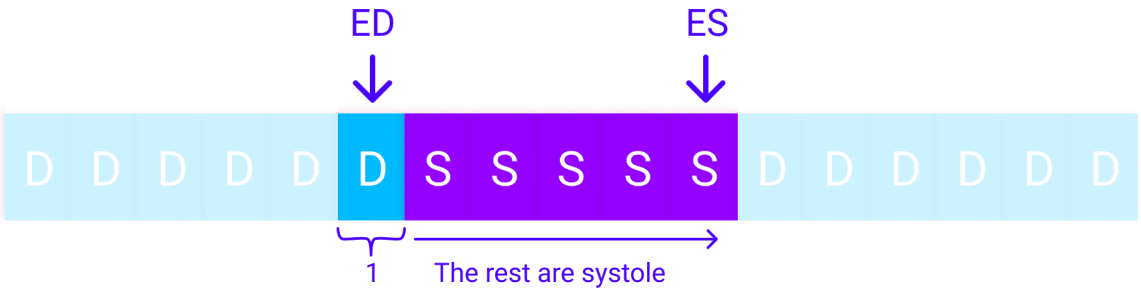


Figure 3.2: Class imbalance: only the first frame is marked with the phase of the first end-event (either ED or ES), all others are marked with the other phase.

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.

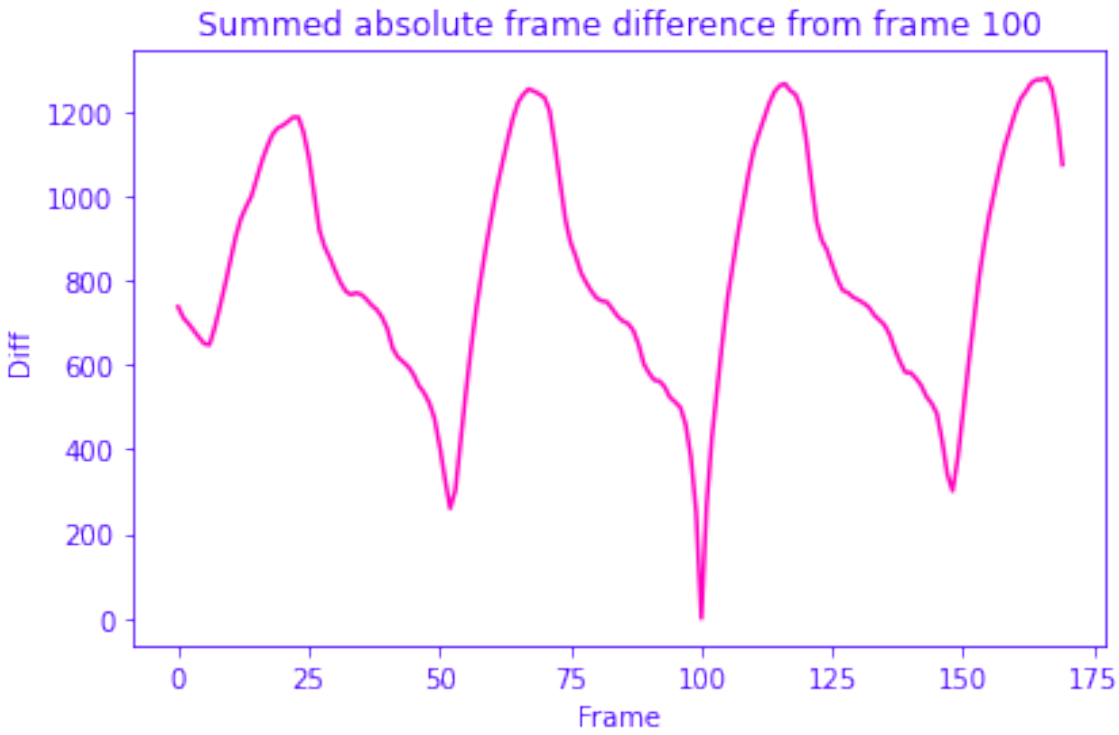


Figure 3.3: The absolute frame difference of all frames in a video compared to frame 100. Notice that the difference for frame 100 is 0 as it (of course) equals itself.

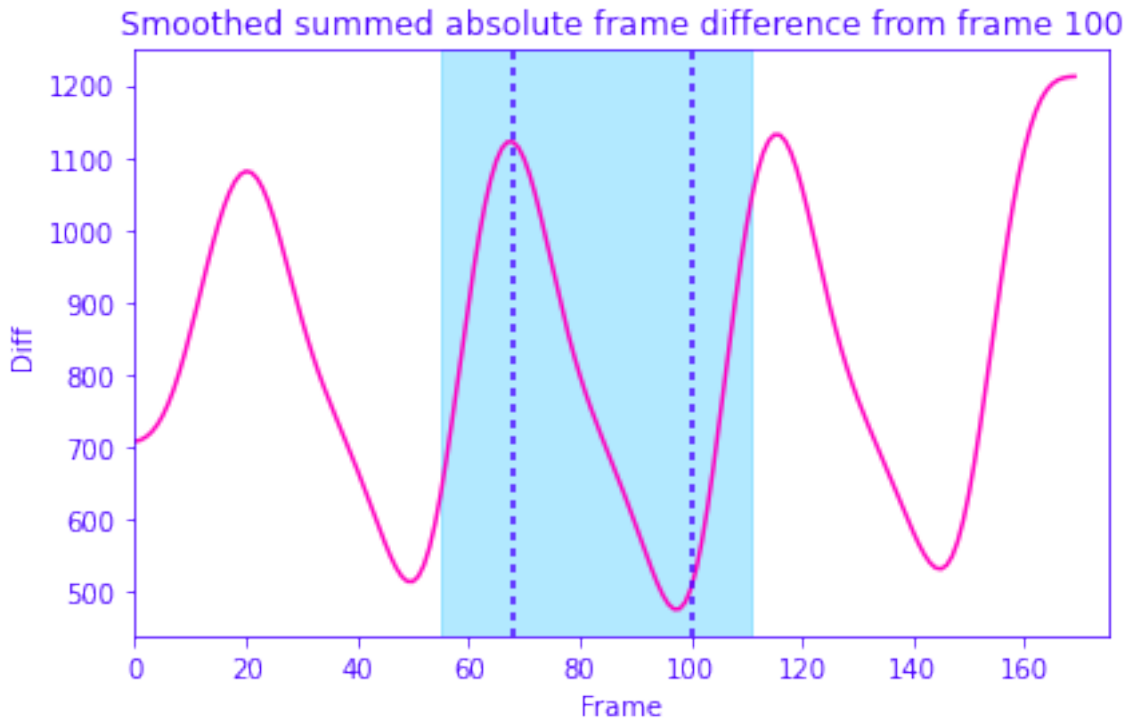


Figure 3.4: The same summed absolute frame difference plot as in figure 3.3, but smoothed using a gaussian blur with a kernel standard deviation of 5. The dashed lines represent phase-end events and the frames in the light blue area are frames with labeled phase. Notice how the labeled frames area only 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 Normalizing and 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 labeled 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.

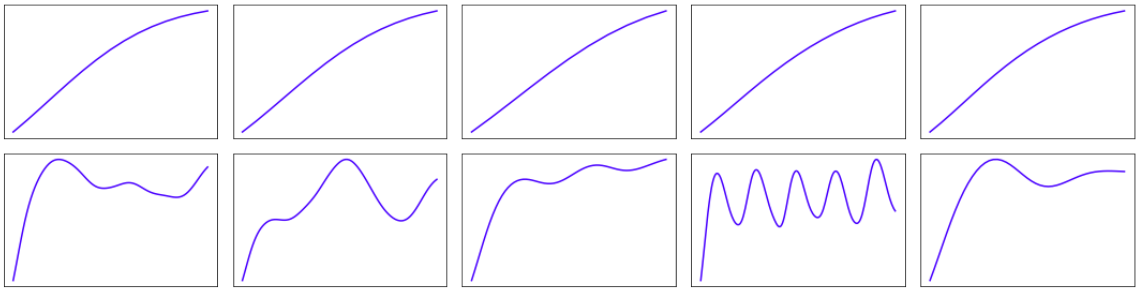


Figure 3.5: The summed absolute frame difference between first end-phase event and the frames up until the next end-phase event. This should only be a half cardiac cycle, so there should be at most one peak. The upper plots show videos where the end-phase labels only cover one half cardiac cycle, while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

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.

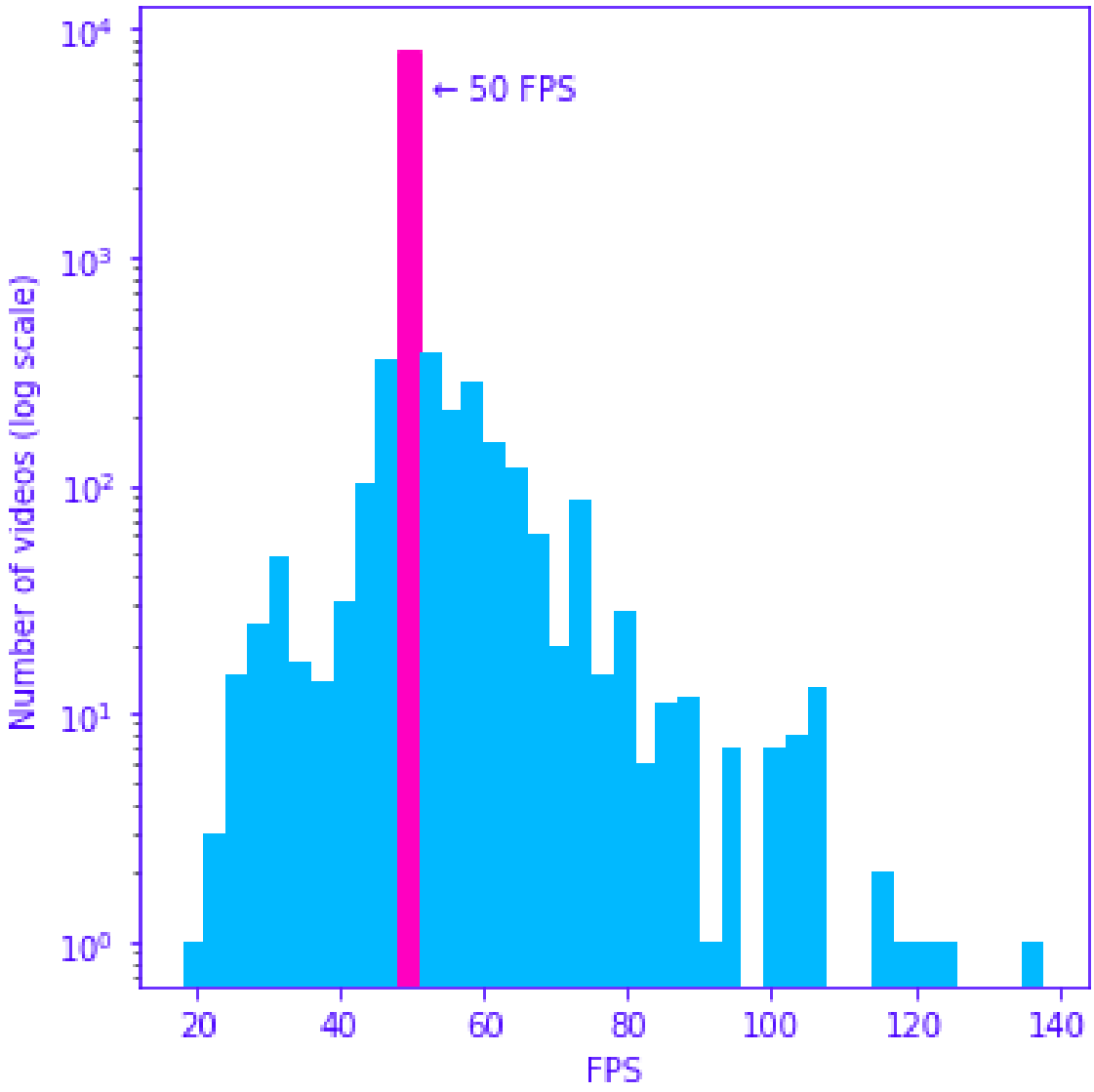


Figure 3.6: A histogram of the different FPS rates of the videos in the Echonet dataset. Note that the y-axis is in logarithmic scale — in fact, almost 80% of the videos have exactly 50 FPS.

3.1.4 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.

A full Echonet-Dynamic dataset pipeline is visualized in figure 3.7.

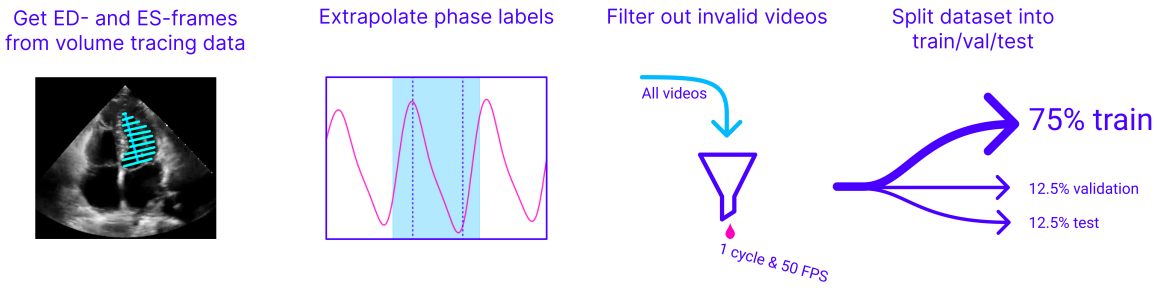


Figure 3.7: A visualization of the data processing pipeline for the Echonet-Dynamic dataset, as described in the previous subsections. First, the ED- and ES-frames from the video is extracted from the volume tracings data. The frame with the biggest volume is ED, the other one is ES. Next, more frame labels are extrapolated by looking at the absolute pixel differences between the ED- or ES-frame and the other frames of the video. Then, videos are filtered such that not more than one cardiac cycle is included in the labeled frames and all videos have 50 FPS. Finally, the videos are split randomly into 3 subsets: training, validation, and testing.

Chapter 4

Methodology

4.1 Environment Formulation

As described in section 2.4.6 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.

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 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.

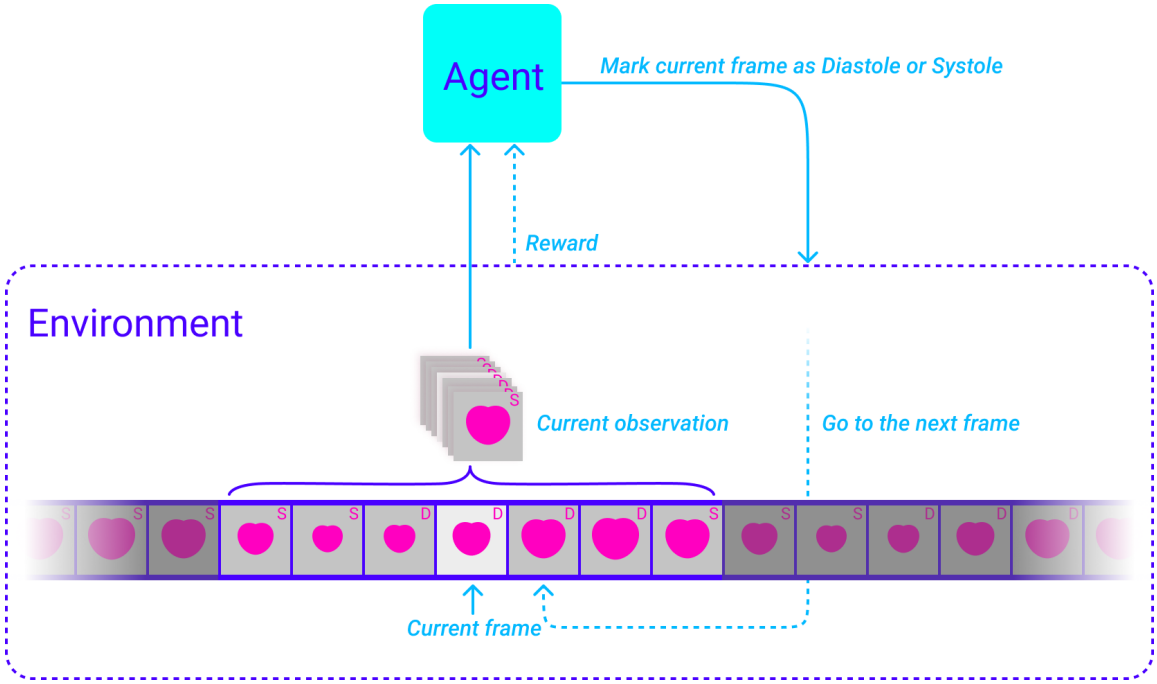


Figure 4.1: Visualization of the Binary Classification Environment loop. An agent sees the observation from the current frame and takes an action, either marking it as Diastole or as Systole, and gets back the reward and the observation for the next frame from the environment.

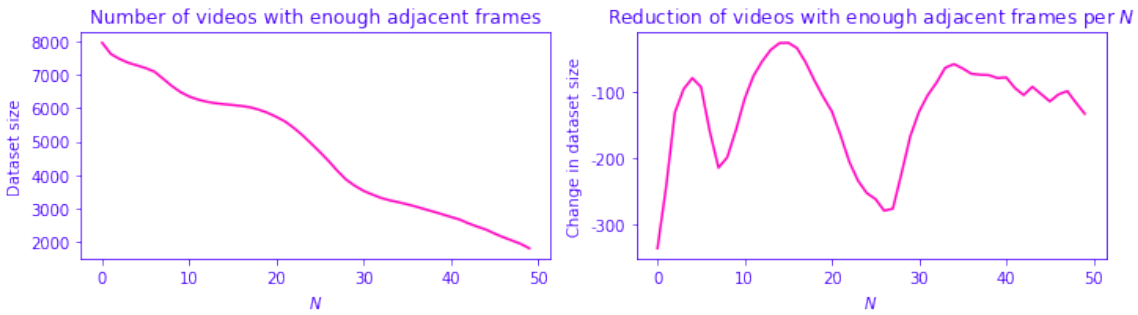


Figure 4.2: The effect of N on the size of the dataset. Left plot shows the number of valid videos (videos with at least N adjacent frames on either side) for the whole dataset. Right plot shows the change in the number of valid videos per N for the whole dataset.

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

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

We could instead base it on the true events and take the distance to the 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_{simple} 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_{simple} 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 in frames from the current state s to the nearest frame that has the predicted phase a.

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

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

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 [**bradbury_jax_2018**]. Other frameworks considered were Tensorflow [**abadi_tensorflow_2015**] and PyTorch [**paszke_pytorch_2019**]. A list of the most important ones can be found in table 4.1.

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

Library	Description
jax	Main data-processing framework. Provides autodifferentiation, vectorization, J
gym	An interface for defining RL environments [**brockman_openai_2016**]
dm-haiku	A neural network library for JAX [**hennigan_haiku_2020**]
optax	A gradient processing and optimization library for JAX [**hessel_optax_2020**]
rlax	Building blocks for building RL agents [**babuschkin_deepmind_2020**]
dm-acme	Distributed RL agent implementations and building blocks [**hoffman_acme_20**
dm-reverb	A database for storing and sampling experience replay [**cassirer_reverb_2021**]
dm-launchpad	A library for defining and creating distributed systems [**yang_launchpad_2021**
Scikit-learn	A collection of machine learning algorithms. In this project it is mostly used fo

4.3 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.

4.3.1 Neural Network

The neural network that approximates the Q-function is inspired by the original Atari DQN paper[**mnih_human-level_2015**]. 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. The network is visualized in figure 4.3.

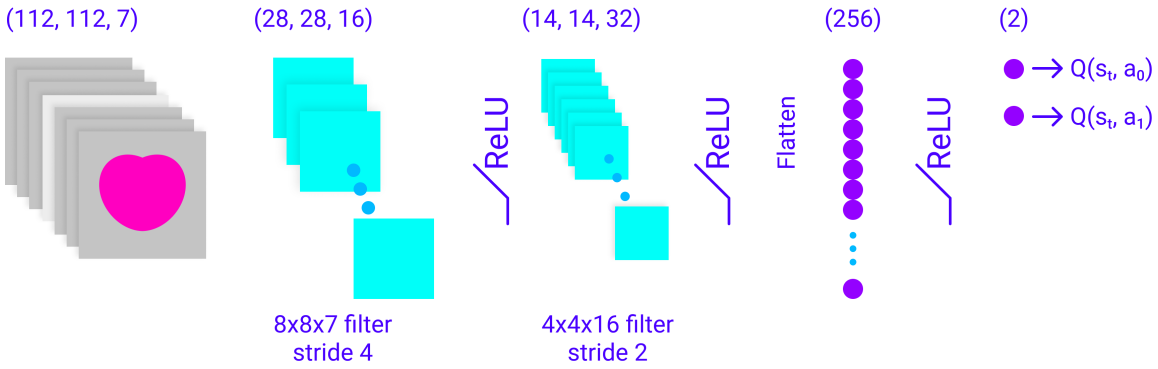


Figure 4.3: A visualization of the simple DQN-Atari paper inspired CNN.

4.3.2 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 prioriticed 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.

4.3.3 Distributed Training

As mentioned, DQN lends itself nicely to distributed training. In this project, this is achieved through a library called Acme[**hoffman_acme_2020**]. At the center of Acme is another library called Reverb[**cassirer_reverb_2021**]. 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 don'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 called Launchpad[**yang_launchpad_2021**].

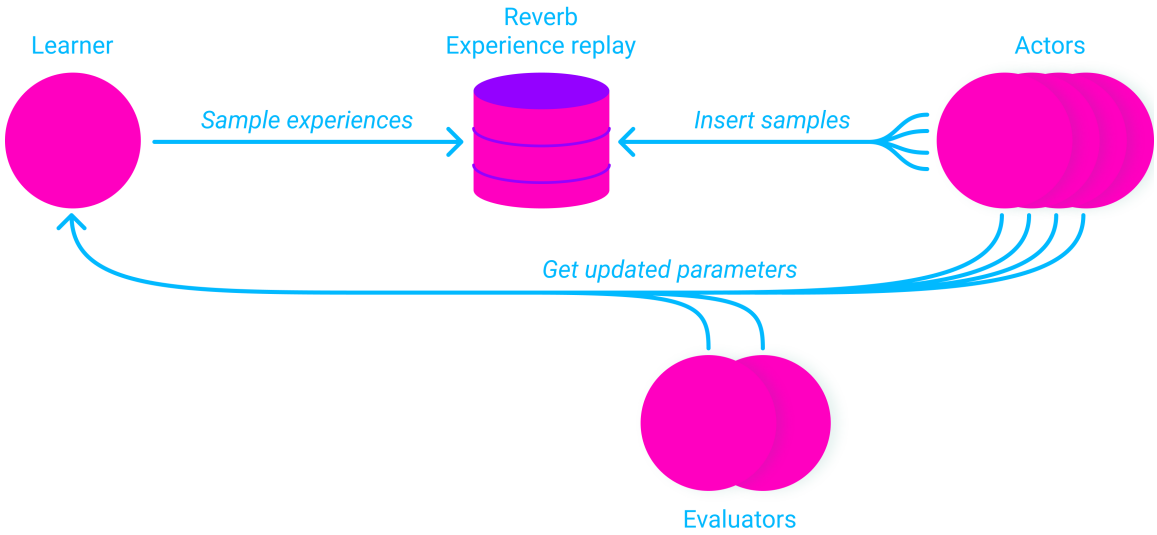


Figure 4.4: The distributed RL training system. Each pink node runs in a separate Python process, and each blue arrow is a inter-process function call facilitated by Launchpad.

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 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 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.

4.4 Evaluation

During training, the updated parameters of the model are continuously evaluated using GaaFD on 50 videos, randomly selected each time, from 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 receive 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. The density plots used are an approximation of the continuous distribution of GaaFD. A histogram may also be used, but these plots were easier to compare using density plots. They are created using Gaussian kernel Estimation (KDE) [**scott_multivariate_1992**]. 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, e.g. 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 on the CPU and GPU. Additionally, IO overhead is reported by comparing the average processing time when sending the data to GPU for each call versus pre-placing the data on the GPU. The average run time is calculated by taking the elapsed time, averaged over 1000 calls.

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

4.5 Selection of Hyper-Parameters

4.5.1 Generalized Average Absolute Frame Difference Reward Function

Using GaaFD directly as the reward function 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.

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 classification 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.

The expected returns are assumed to be very sensitive to the current policy as a correctly selected next action's returns may be jeopardized by future wrongly selected actions. This would make it hard for the agent to extract information about which action were wrong and which were correct. Because of this we opt to use very low values of the exploration parameter ϵ, as higher values of ϵ means that actions will be selected at random more often, making the expected returns harder to predict. 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 4.5.1 (most relevant ones are highlighted).

4.5.2 Simple- and Proximity-Based Reward Functions

Using reward functions that are based on each individual phase prediction gets around the reward sparsity problem of using GaaFD as the reward function. Two more reward functions are explored: a simple reward function R_{simple}, as defined in equation 4.5, and an proximity-based reward function $R_{proximity}$, 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

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

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 for experiments with reward functions R_{simple} and $R_{proximity}$ is listed in table 4.5.2 (most relevant ones are highlighted).

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

4.6 Incorporating Search

Although RL is designed to be able to perform 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?

4.6.1 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 [**alansary_evaluating_2019**], 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.

4.6.2 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.

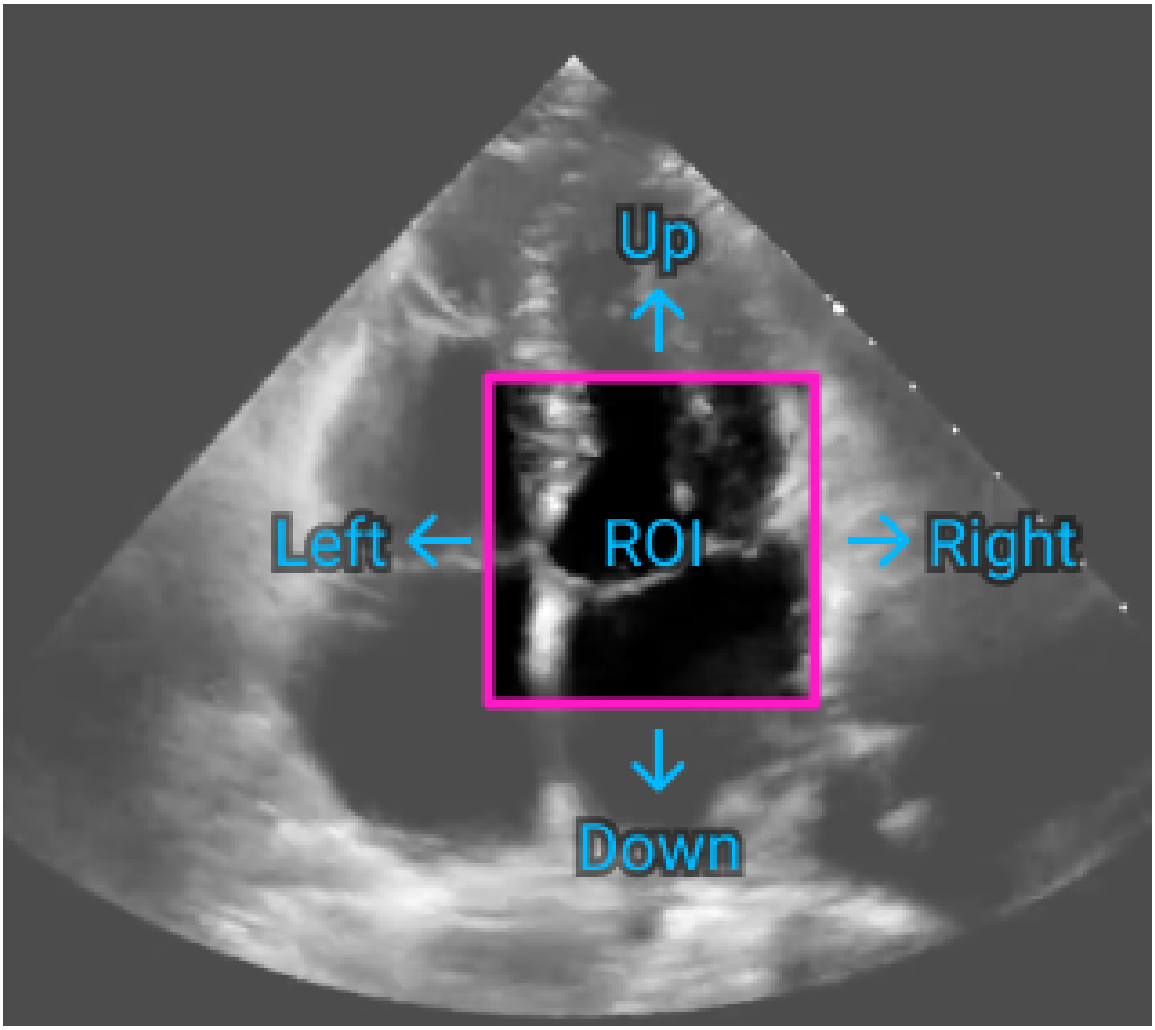


Figure 4.5: A Region Of Interest (ROI) is given to the agent which it can then move around in order to explore.

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. The m-mode can be seen as taking a 2D slice of the video, as seen in figure 4.6. 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.

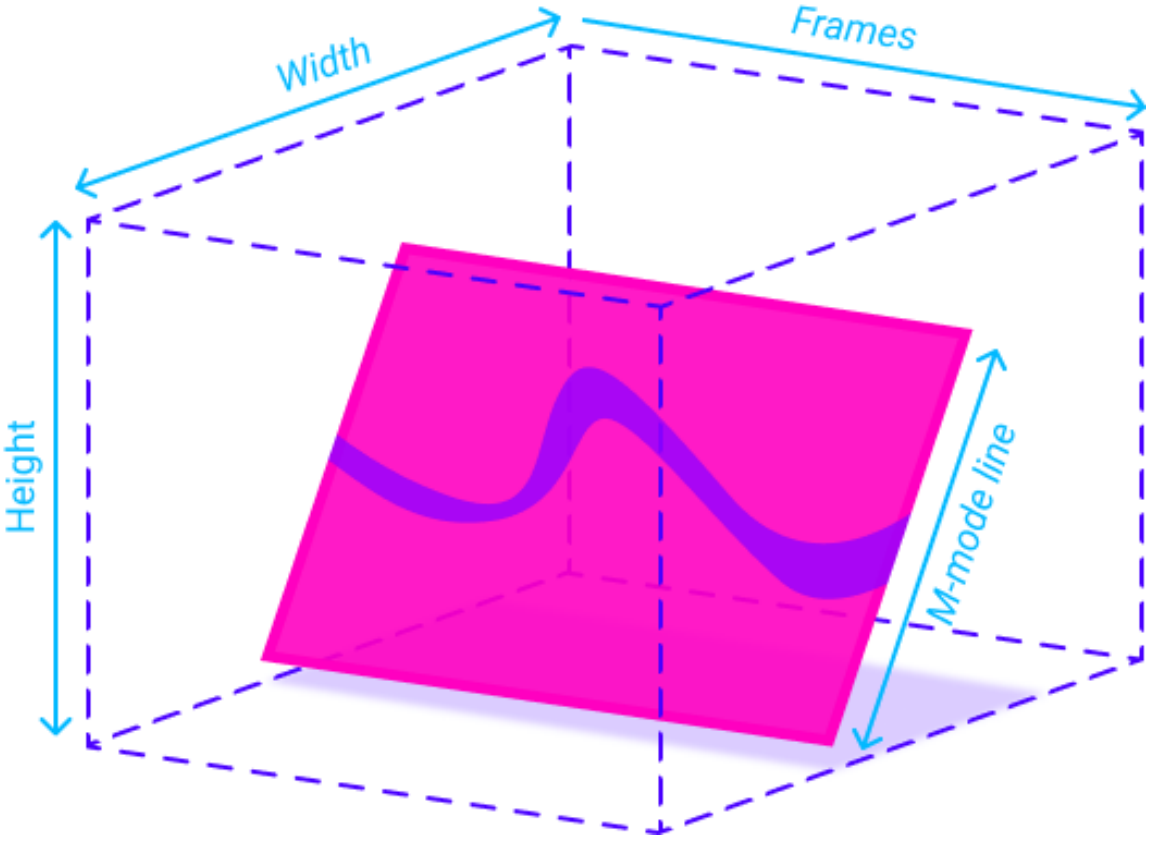


Figure 4.6: An m-mode image is an intersecting plane in 3D "video space".

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. Local and global translation is visualized in figure 4.7. 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, as seen in figure 4.8 — independent of rotation, it simply shifts the m-mode image down or up, respectively.

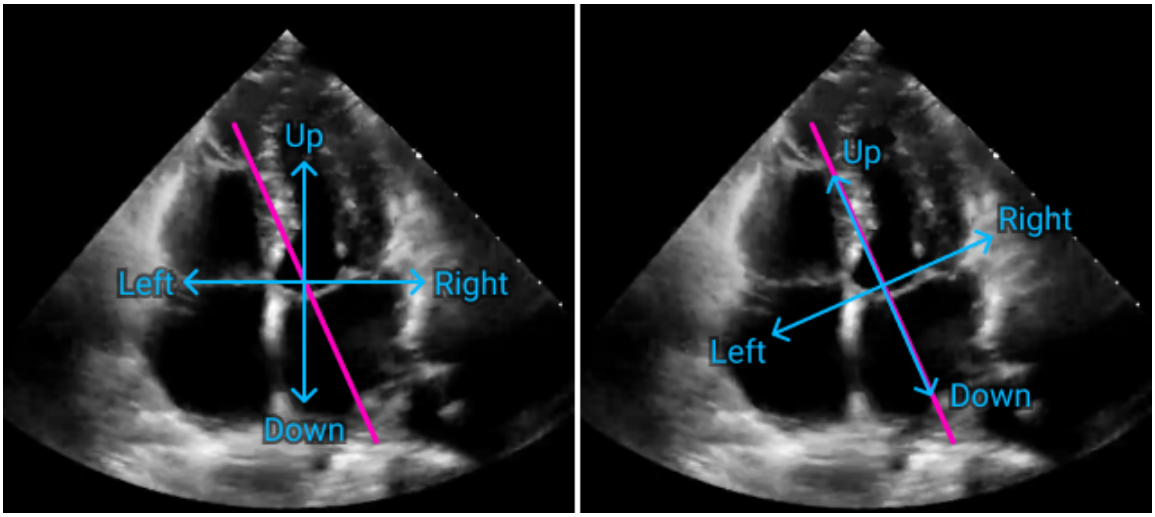


Figure 4.7: Global (to the left) versus local (to the right) translation. Local translation means that the movement depends on the direction of the m-mode line.

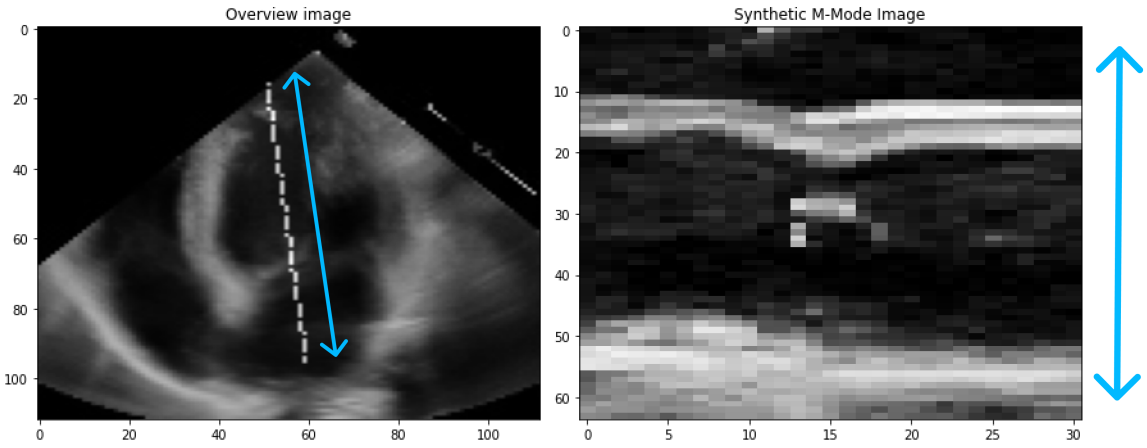


Figure 4.8: Moving the line in up or down using local translation changes the synthetic m-mode image very little — it simply translates the whole image up or down, as indicated by the blue arrows. To the left: an overview image of a video with the line added on top. To the right: the resulting synthetic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

4.7 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 as additional channels 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 within in a bounding box. 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 the bounds of the video, 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 cut-off number of 200 steps was chosen a bit arbitrary, but given that the average video length is 50 frames, 200 steps should give the agent ample time to find the best synthetic m-mode line position.

The reward function is the same as in the simple binary classification environment, but with some modifications. The agent receives a reward of

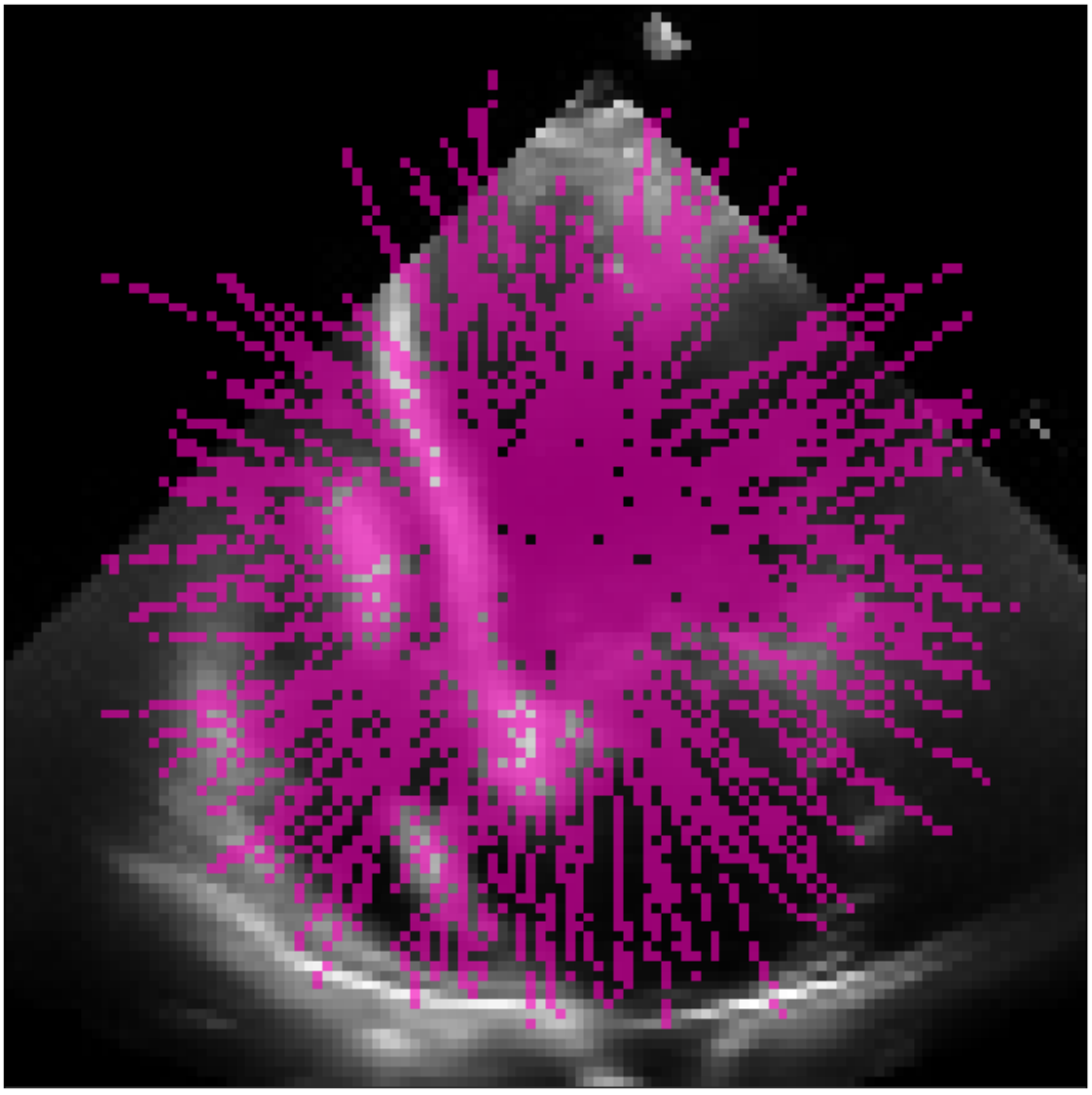


Figure 4.9: The union of 100 randomly sampled m-mode lines.

-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 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.

4.7.1 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 in order to accommodate concatenation. 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. The network is visualized in figure 4.10.

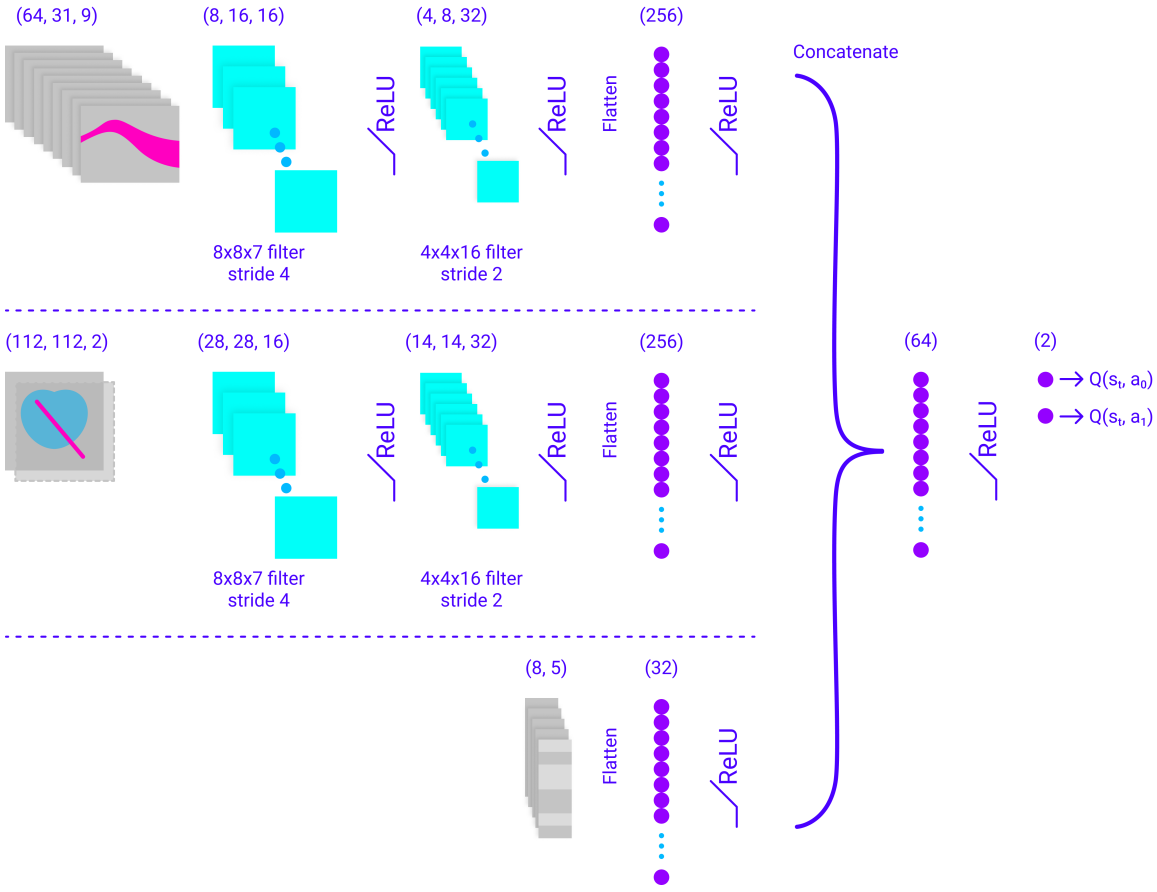


Figure 4.10: The network architecture of the m-mode agent. An observation consists of three parts. Each part is processed independently by a neural network before being concatenated and used to produce the approximated Q-values.

Chapter 5

Experiments and Results

5.1 Performance Metrics

The performance metrics is reported in the following four tables. Tables 5.1, 5.2 and 5.3 presents the performance of agents traines using R_{GaaFD}, R_{simple} and $R_{proximity}$, respectively. Lastly, table 5.4 presents a comparison between the best models for each reward function.

Out of all the trained models, the model trained using R_{simple} with a value of $\epsilon = 0.5$ performed the best.

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

	$\epsilon = 0.1$	$\epsilon = 0.01$	$\epsilon = 0.0$
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	63.71%	70.33%	76.63%
aaFD	3.51	2.71	2.43
Accuracy	0.82	0.86	0.88
Accuracy diastole	0.90	0.91	0.94
Accuracy systole	0.69	0.75	0.77
Balanced accuracy	0.58	0.67	0.70

5.2 The Impact of Epsilon on Average Absolute Frame Difference

Lower values of ϵ yield a better GaaFD score when using R_{GaaFD} as the reward function. This is best seen in figure 5.1. There is not a consistent difference between GaaFD on ED- or ES-frames individually, but there is less difference in performance on ES-frames between the training split and the test split, as seen in 5.2. Figure 5.3 shows that lower values of ϵ also reduces the mismatch between the number of predicted versus ground

Table 5.2: Performance of agents trained using R_{simple} 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	79.30%	80.26%	76.95%
aaFD	1.71	1.69	1.69
Accuracy	0.91	0.91	0.91
Accuracy diastole	0.93	0.93	0.93
Accuracy systole	0.87	0.88	0.88
Balanced accuracy	0.80	0.81	0.81

Table 5.3: Performance of agents trained using $R_{proximity}$ 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	78.87%	79.40%	76.95%
aaFD	1.74	1.80	1.71
Accuracy	0.91	0.91	0.91
Accuracy diastole	0.94	0.93	0.93
Accuracy systole	0.86	0.87	0.88
Balanced accuracy	0.80	0.80	0.81

Table 5.4: Performance of the best agent for each explored reward function on the test dataset. The best agent was selected by the best GaaFD score.

	R_{GaaFD}	R_{simple}	$R_{proximity}$
ϵ	$\epsilon = 0.0$	$\epsilon = 0.5$	$\epsilon = 0.1$
Best model SGD step	95 616	**10 960**	107 936
GaaFD	3.68	**2.46**	2.55
GaaFD ED	3.74	**2.43**	2.52
GaaFD ES	3.50	**2.41**	2.50
% valid aaFD	76.63%	**80.26**	78.87%
aaFD	2.43	**1.69**	1.74
Accuracy	0.88	**0.91**	**0.91**
Accuracy diastole	**0.94**	0.93	**0.94**
Accuracy systole	0.77	**0.88**	0.86
Balanced accuracy	0.70	**0.81**	0.80

truth events. This was also clearly seen in table 5.1, where a value of $\epsilon = 0.0$ predicted the correct number of events 77% of the time, while $\epsilon = 0.1$ and $\epsilon = 0.01$ yielded 64% and 70%, respectively. We also see a significant "bump" at 2 mismatched number of predictions for all 3 models.

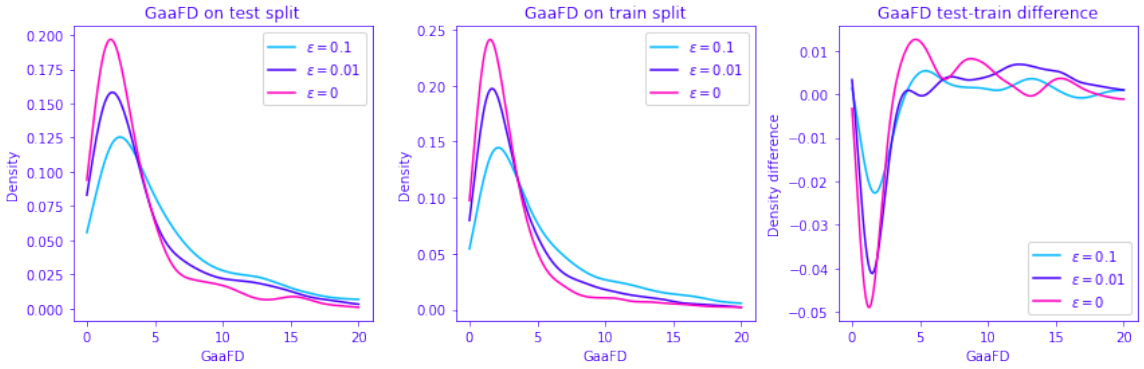


Figure 5.1: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using GaaFD as the reward function. The left plot compares all three models on the test-set. The middle plot compares all three models on the train-set. The right plot shows the difference between the two as a means to visualize model overfitting.

The choice of ϵ did not matter as much for agents trained using R_{simple} or $R_{proximity}$, compared to those trained using R_{GaaFD}, as seen in figures 5.4 and 5.6. As with R_{GaaFD}, R_{simple} and $R_{proximity}$ showed little consistent difference in performance between gaaFD on ED- or ES-frame individually, as seen in figures 5.5 and 5.7. Figure 5.8 further shows that there is little difference between values of ϵ for predicting the correct number of events as the number of ground truth events, both for R_{simple} and $R_{proximity}$.

5.3 The Impact of Reward Function and Epsilon on Accuracy

Accuracy and balanced accuracy are not the main metrics that we want to optimize for, but it is useful to report them to get better insight into how the model performs.

The accuracy of the agents trained using R_{GaaFD} show a similar pattern as with GaaFD, as lower values of ϵ give better scores. This can be seen in figure 5.9 as well as in table 5.1. Figure 5.9 also shows this when accounting for class imbalance through balanced accuracy. All 3 models performs perform better at classifying diastole frames compared to systole frames, as visualized in figure 5.10.

Again, the choice of ϵ did not matter as much for models trained using R_{simple} or $R_{proximity}$ with regards to accuracy and balanced accuracy, as seen in figure 5.11 and 5.13. These models also perform better at classifying diastole frames compared to systole frames.

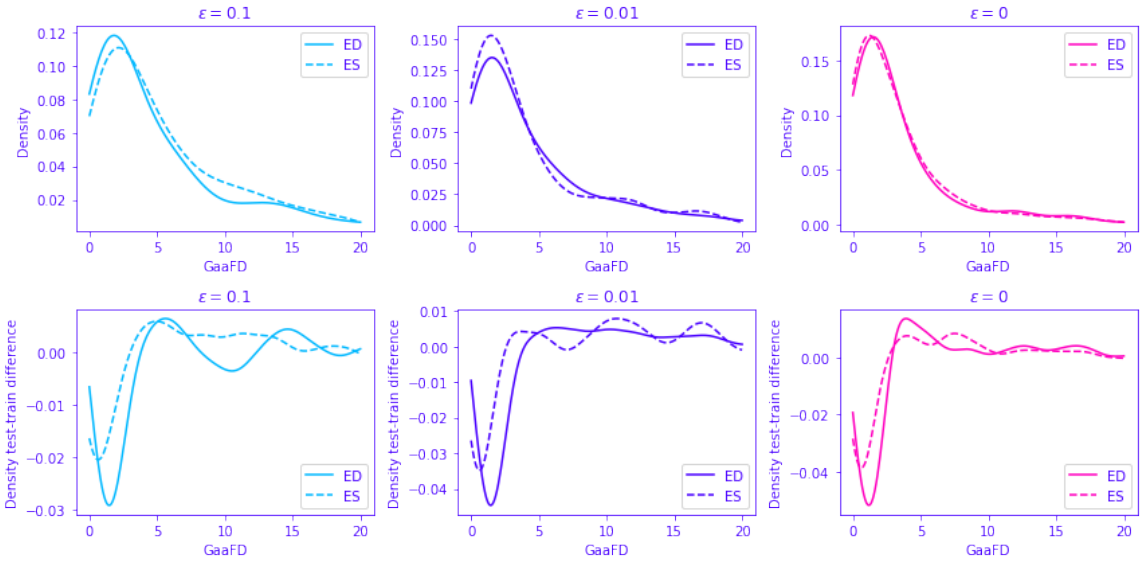


Figure 5.2: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using GaaFD as the reward function, only accounting for either ED- or ES-events individually. The upper row compares the performance on ED and ES for each model. The bottom row shows the difference in GaaFD-density on the test-set versus the train-set as a means to visualize model overfitting.

5.4 Learning Curves

For models trained using R_{GaaFD}, lower values of ϵ converge faster, as seen in the training curve in figure 5.15. There is no apparent degradation in performance over time that would indicate that the models start to overfit at some point, but the models trained using $\epsilon = 0.0$ or $\epsilon = 0.01$ perform slightly better on the training split than on the test split. This is not apparent for the model trained using $\epsilon = 0.1$. The loss curves for all models follow a peculiar pattern where it starts with sinking rapidly, before increasing again, followed by a slow decrease until it (presumably) converges, as seen in figure 5.16. Interestingly, the model trained using a value of $\epsilon = 0.01$ reaches a higher loss than the other two at its peak following the rapid sinking at the start. Also interesting, the model trained using the highest value of ϵ has a loss curve sitting between the other two.

Lower values of ϵ yields less overfitting both for models trained using R_{simple} and $R_{proximity}$, as seen in figures 5.17 and 5.18. In fact, the models are able to reach perfect accuracy on the training split, at the cost of worse performance on the test split, as seen in figure 5.19.

The loss curves for models trained using R_{simple} or $R_{proximity}$ does not show the same peculiar pattern as those trained using R_{GaaFD}. They all decrease quite rapidly at first before slowing down until convergence. The model trained with the lowest value of ϵ, $\epsilon = 0.1$, converge the fastest. The models trained using $R_{proximity}$ also converge faster overall.

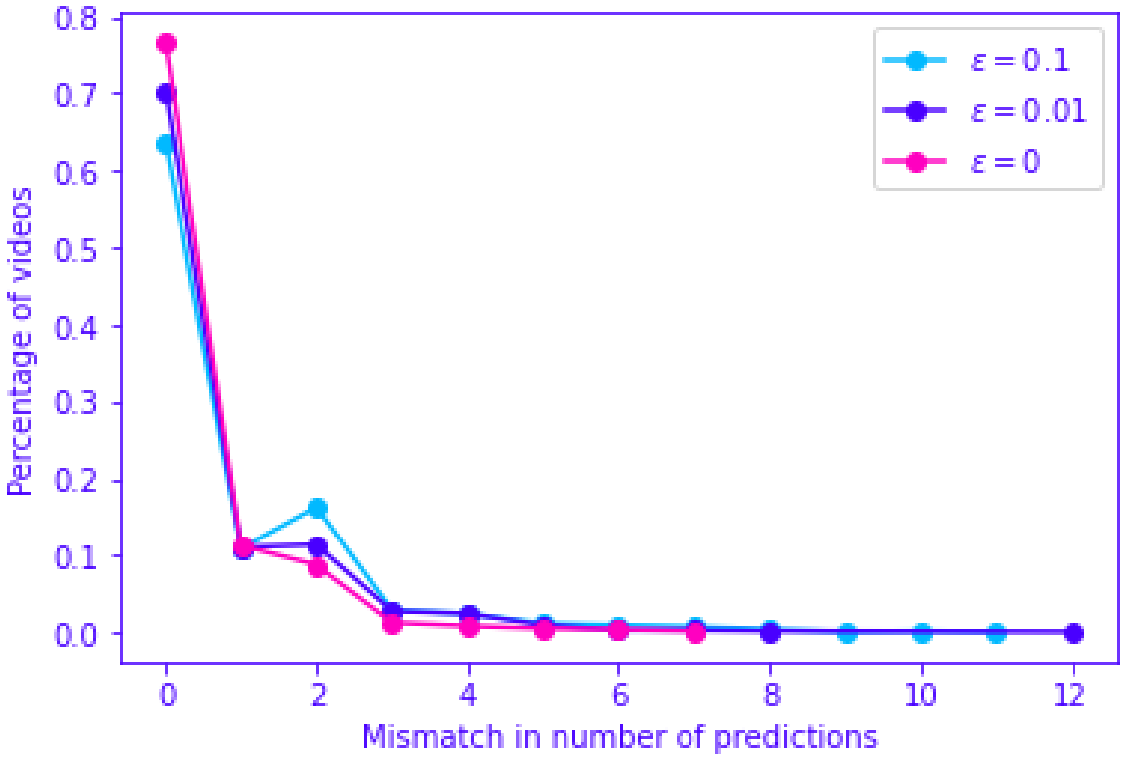


Figure 5.3: The difference between the number of predicted events and the number of ground truth events for each model when using GaaFD as the reward function. Most predictions produce the same number of predicted events as ground truth, e.g. the model with $\epsilon = 0$ produces the correct number of events 77% of the time, which can also be seen in table 5.1.

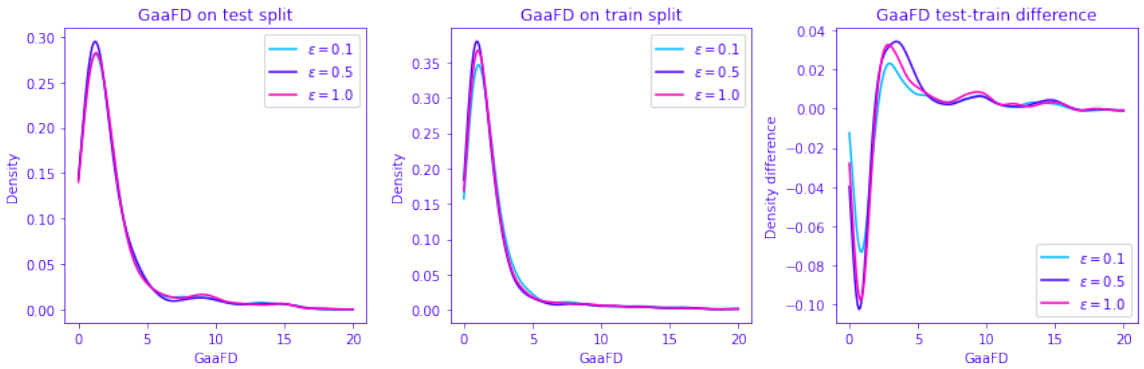


Figure 5.4: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.5$, and $\epsilon = 1.0$) when using R_{simple} as the reward function. The left plot compares all three models on the test-set. The middle plot compares all three models on the train-set. The right plot shows the difference between the two as a means to visualize model overfitting.

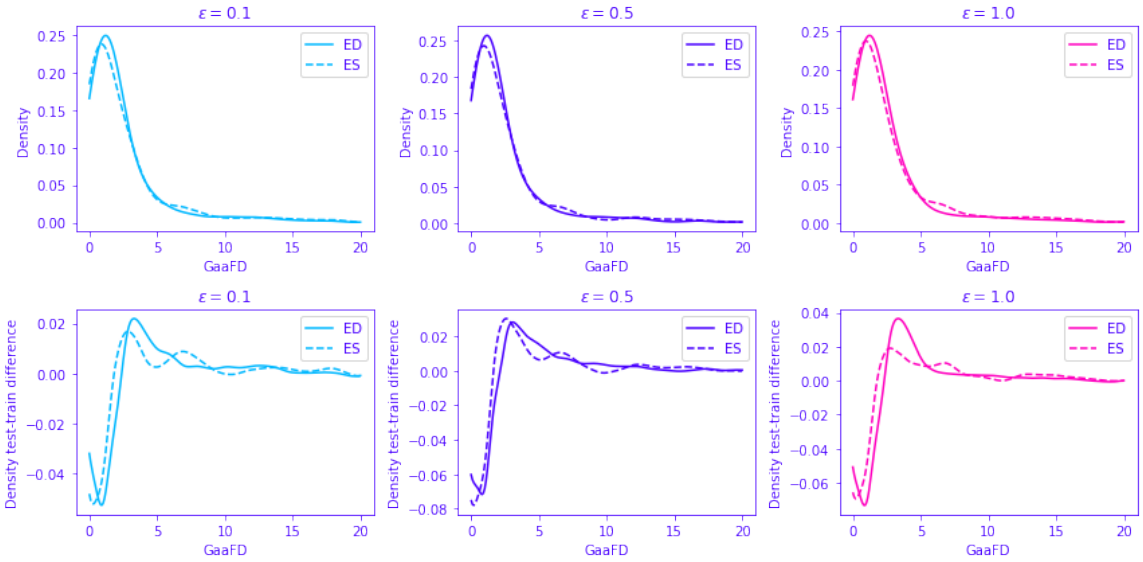


Figure 5.5: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using R_{simple} as the reward function, only accounting for either ED- or ES-events individually. The upper row compares the performance on ED and ES for each model. The bottom row shows the difference in GaaFD-density on the test-set versus the train-set as a means to visualize model overfitting.

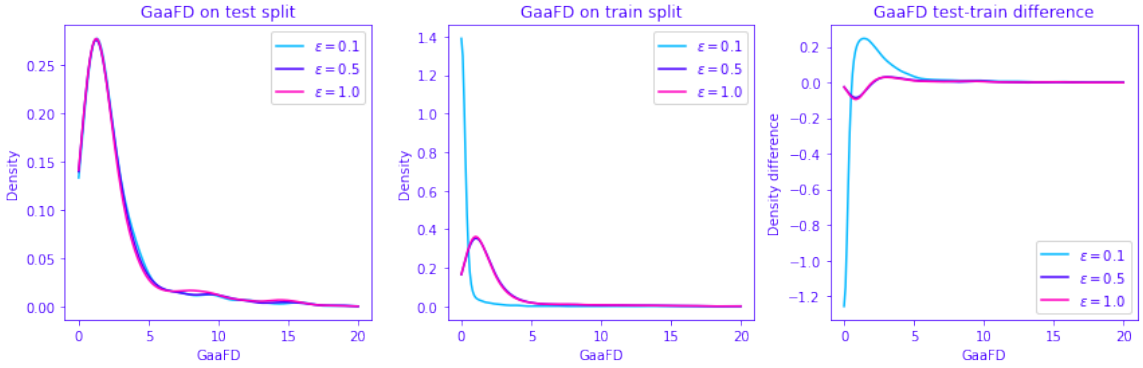


Figure 5.6: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.5$, and $\epsilon = 1.0$) when using $R_{proximity}$ as the reward function. The left plot compares all three models on the test-set. The middle plot compares all three models on the train-set. The right plot shows the difference between the two as a means to visualize model overfitting.

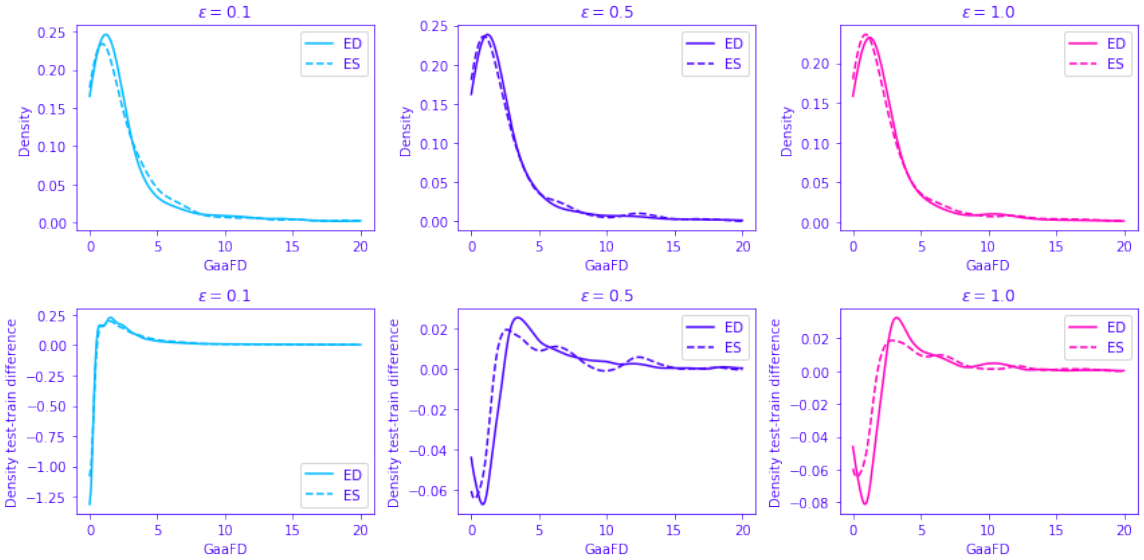


Figure 5.7: Gaussian KDE of the GaaFD-performance for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using $R_{proximity}$ as the reward function, only accounting for either ED- or ES-events individually. The upper row compares the performance on ED and ES for each model. The bottom row shows the difference in GaaFD-density on the test-set versus the train-set as a means to visualize model overfitting.

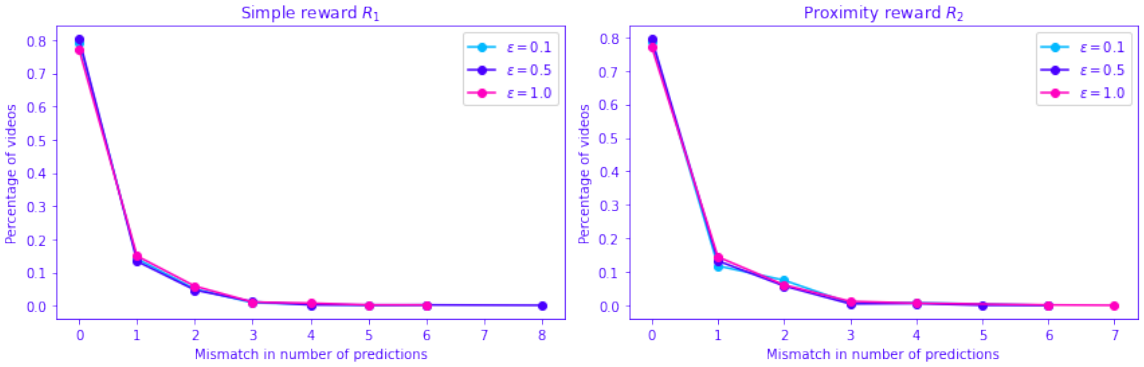


Figure 5.8: The difference between the number of predicted events and the number of ground truth events for each model when using R_{simple} (left) and $R_{proximity}$ (right) as the reward function. Most predictions produce the same number of predicted events as ground truth, e.g. the model with $\epsilon = 0.5$ and R_{simple} as the reward function produces the correct number of events 80% of the time, which can also be seen in table 5.2.

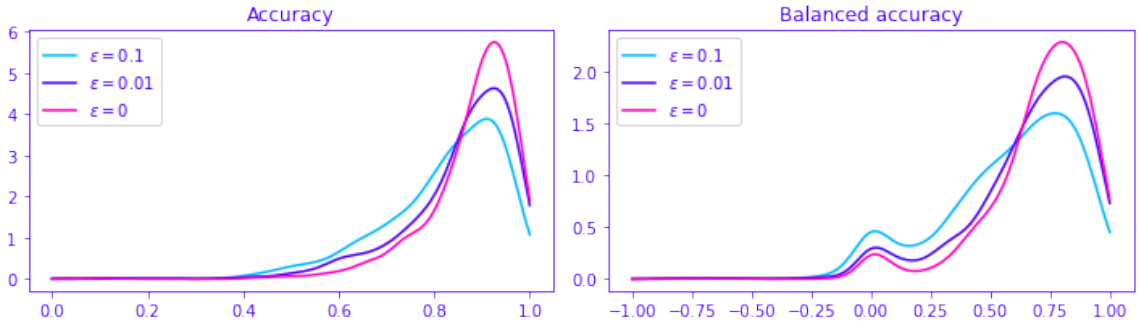


Figure 5.9: Gaussian KDE of the accuracy and balanced accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using GaaFD as the reward function. The left plot shows the accuracy. The right plot shows the balanced accuracy, which accounts more for class-imbalance.

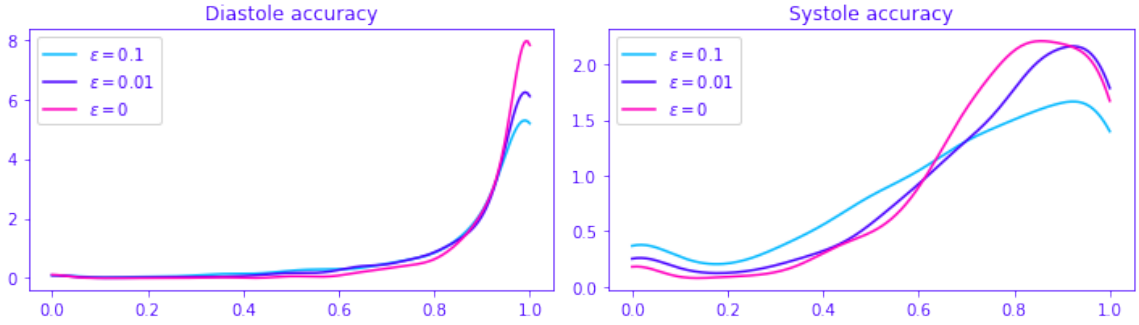


Figure 5.10: Gaussian KDE of the accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using GaaFD as the reward function for diastole or systole phase predictions individually. Left plot shows the accuracy for diastole frame predictions. The right plot shows the accuracy for systole frame predictions.

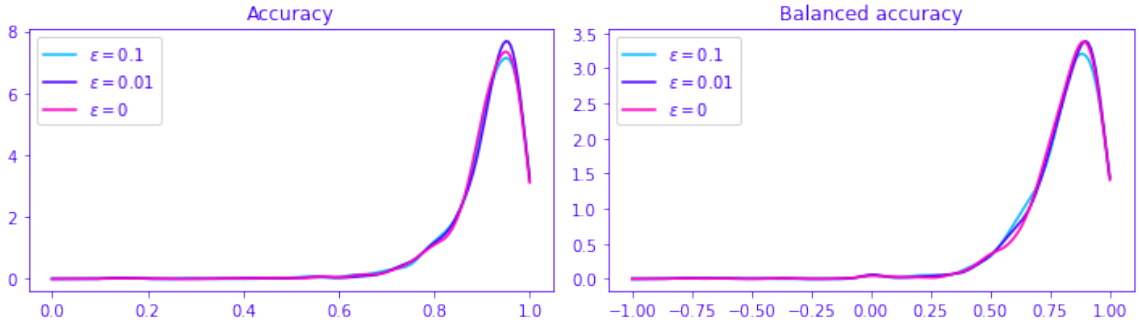


Figure 5.11: Gaussian KDE of the accuracy and balanced accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using R_{simple} as the reward function. The left plot shows the accuracy. The right plot shows the balanced accuracy, which accounts more for class-imbalance.

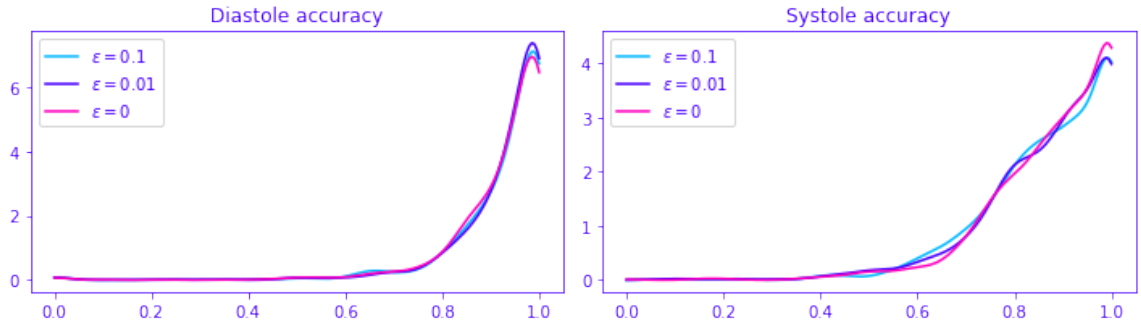


Figure 5.12: Gaussian KDE of the accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using R_{simple} as the reward function for diastole or systole phase predictions individually. Left plot shows the accuracy for diastole frame predictions. The right plot shows the accuracy for systole frame predictions.

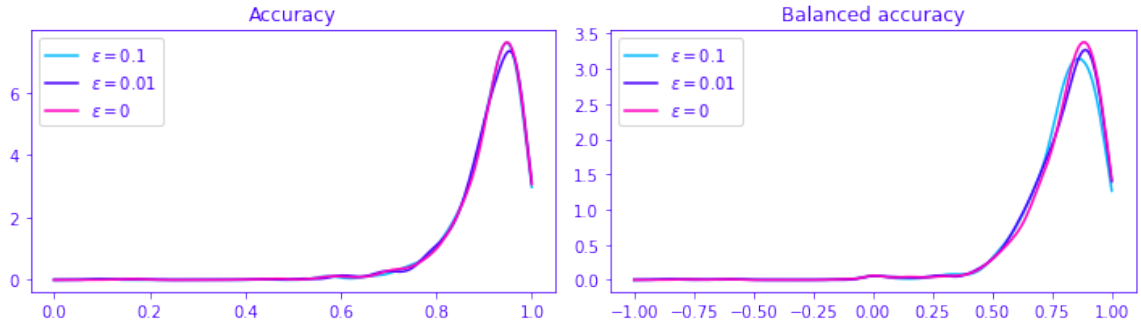


Figure 5.13: Gaussian KDE of the accuracy and balanced accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using $R_{proximity}$ as the reward function. The left plot shows the accuracy. The right plot shows the balanced accuracy, which accounts more for class-imbalance.

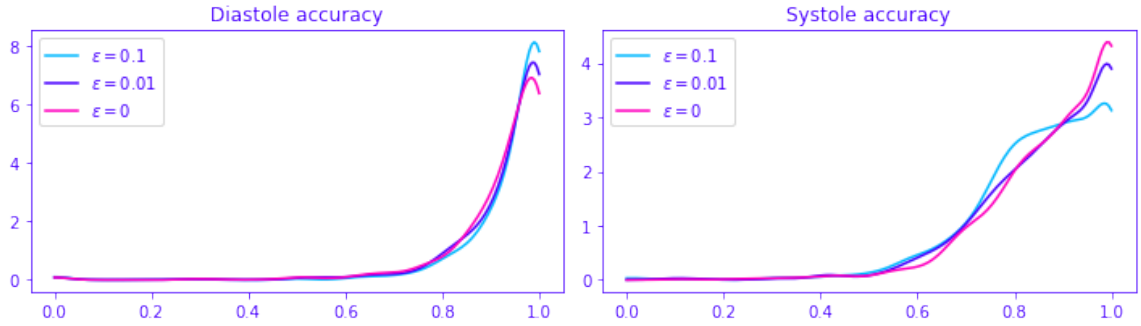


Figure 5.14: Gaussian KDE of the accuracy for each model ($\epsilon = 0.1$, $\epsilon = 0.01$, and $\epsilon = 0$) when using $R_{proximity}$ as the reward function for diastole or systole phase predictions individually. Left plot shows the accuracy for diastole frame predictions. The right plot shows the accuracy for systole frame predictions.

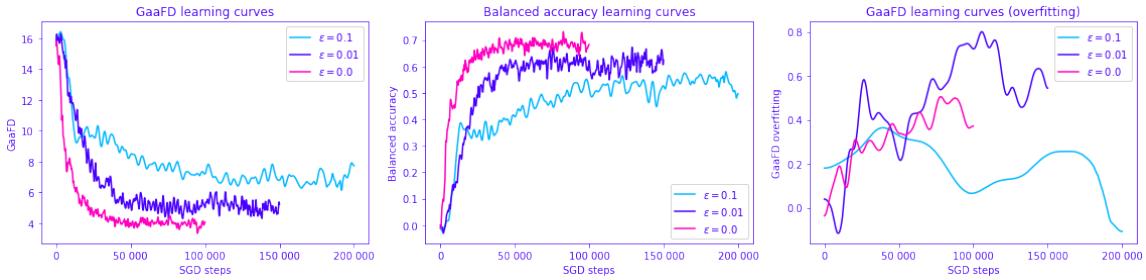


Figure 5.15: The learning curves of using GaaFD as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the training 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.

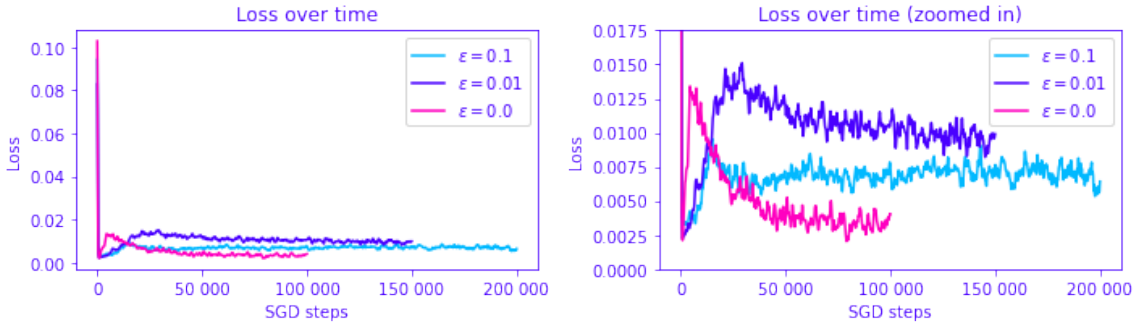


Figure 5.16: The training loss over time for different values of epsilon. The left plot shows the full y-axis, while the right plot shows the same plots but with a zoomed-in y-axis.

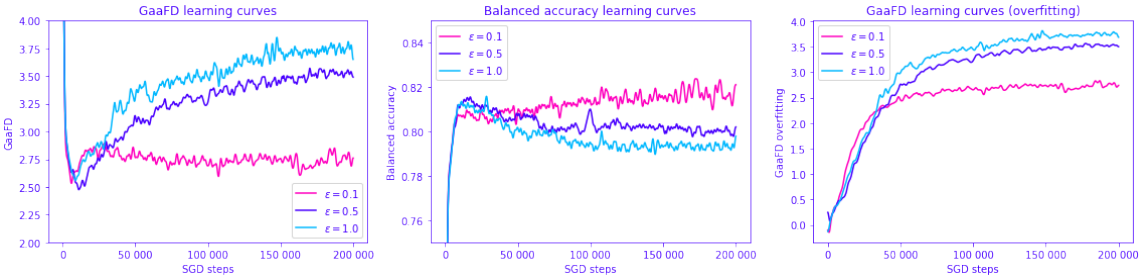


Figure 5.17: The training curves of using R_{simple} as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the training 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.

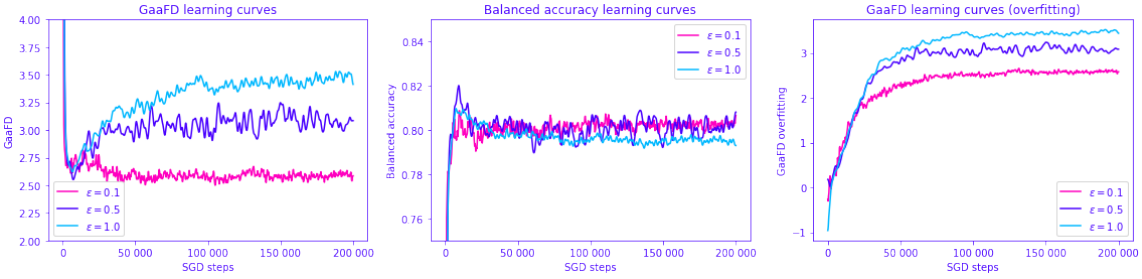


Figure 5.18: The training curves of using $R_{proximity}$ as the reward function for different values of the exploration parameter ϵ. Left: GaaFD over training time (gradient descent steps). Middle: Balanced accuracy over training time. Right: The difference in GaaFD between the validation set and the 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.

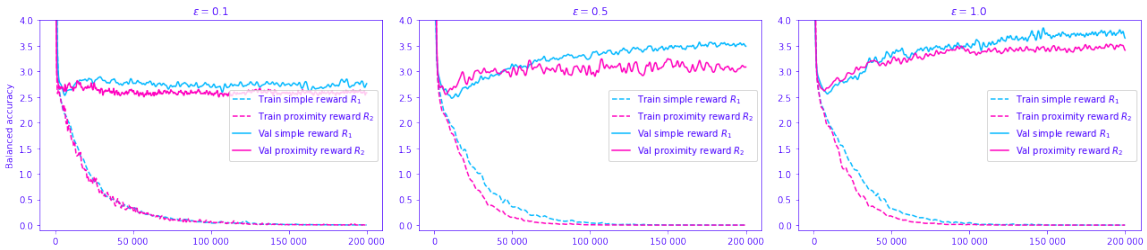


Figure 5.19: The GaaFD over training time (gradient descent steps) on the validation set (solid pink and blue line) and the training set (dashed pink and blue lines). The GaaFD on the training set reaches 0, meaning perfect predictions.

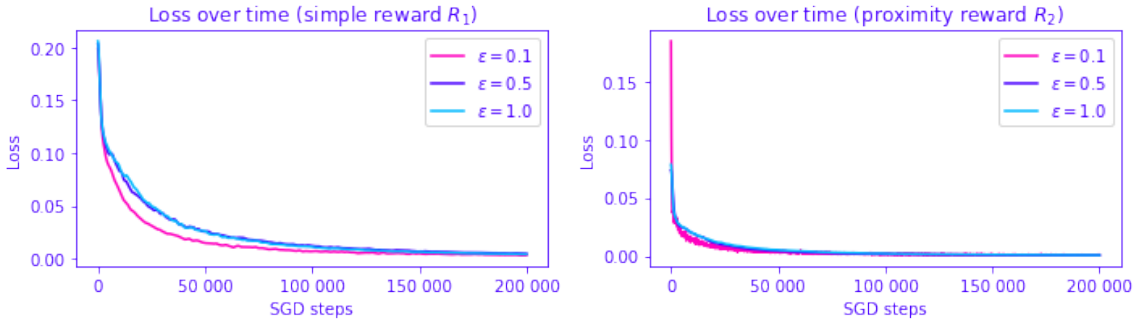


Figure 5.20: The training loss over time for different values of epsilon. Left: an agent trained using R_{simple}. Right: an agent trained using $R_{proximity}$.

Models trained using $R_{proximity}$ consistently perform better at the metric of GaaFD in later iterations of training, though this is most appararent long after the models have overfit on the training split. The models trained using R_{simple} have indications of performing better at the metric of balanced accuracy, though the difference is mostly apparent in the models trained using a value of $\epsilon = 0.1$. Even though $R_{proximity}$ performs better at the important metric of GaaFD after overfitting occur, the best model overall belongs to the model trained using R_{simple} with a value of $\epsilon = 0.5$.

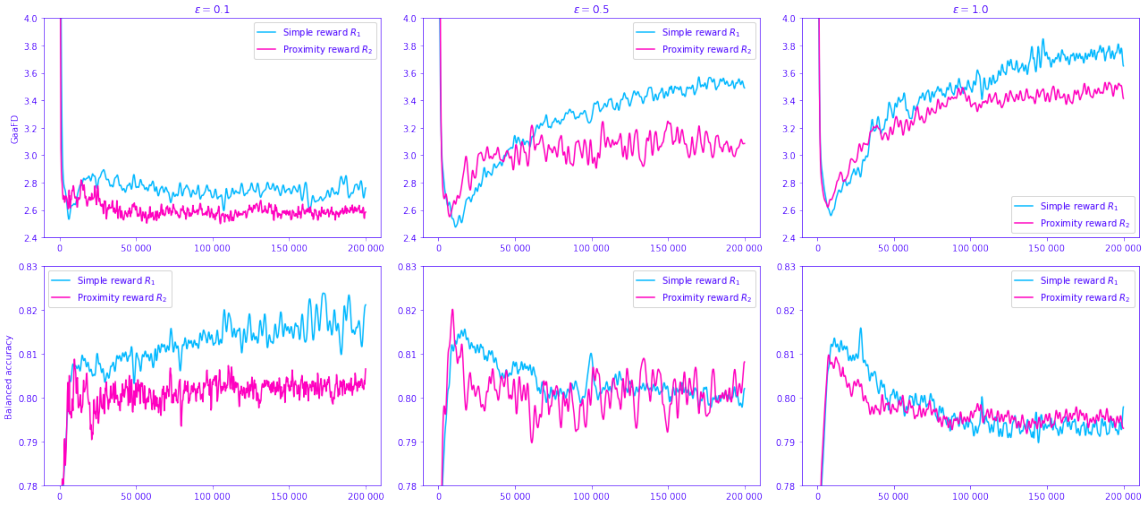


Figure 5.21: Comparison of the training curves using R_{simple} versus $R_{proximity}$ for different values of of the exploratio parameter ϵ. Top row shows the GaaFD over training time (gradient descent steps). Bottom row shows the balanced accuracy over training time. Each column correspond to one of the agents, $\epsilon = 0.1$, $\epsilon = 0.5$, and $\epsilon = 1.0$, respectively.

5.5 The Impact of Reward Function and Epsilon on Q-Values

For DQN, the Q-values dictate what actions are taken. This section plots the Q-values for the best and worst performing videos for each reward function and value of epsilon used for the reward function. These results are largely qualitative but shed light on how the models "reason" about the frames in a video.

Figures 5.22, 5.23, and 5.23 plot the Q-values of each frame in the 3 best performing videos for that model for models trained with R_{GaaFD}, R_{simple}, and $R_{proximity}$, respectively, and for each value of ϵ. Likewise, Figures 5.22, 5.23, and 5.23 plots the same, but for the 3 worst performing videos.

The effect of a higher value of ϵ of agents trained with R_{GaaFD} is that the values of marking a frame as diastole or as systole grows closer, as seen in figure 5.22. There is also a noticable positive spike for systole values in the middle of the diastole phase, mostly visible for lower values of ϵ.

The effect of using R_{simple} over $R_{proximity}$ seems to be that R_{simple} causes less noisy Q-values. Interestingly, the spikes in the middle of systole is also visible for agents trained with R_{simple} and $R_{proximity}$, though slightly less than for $thosetrainedwithR_{GaaFD}$.

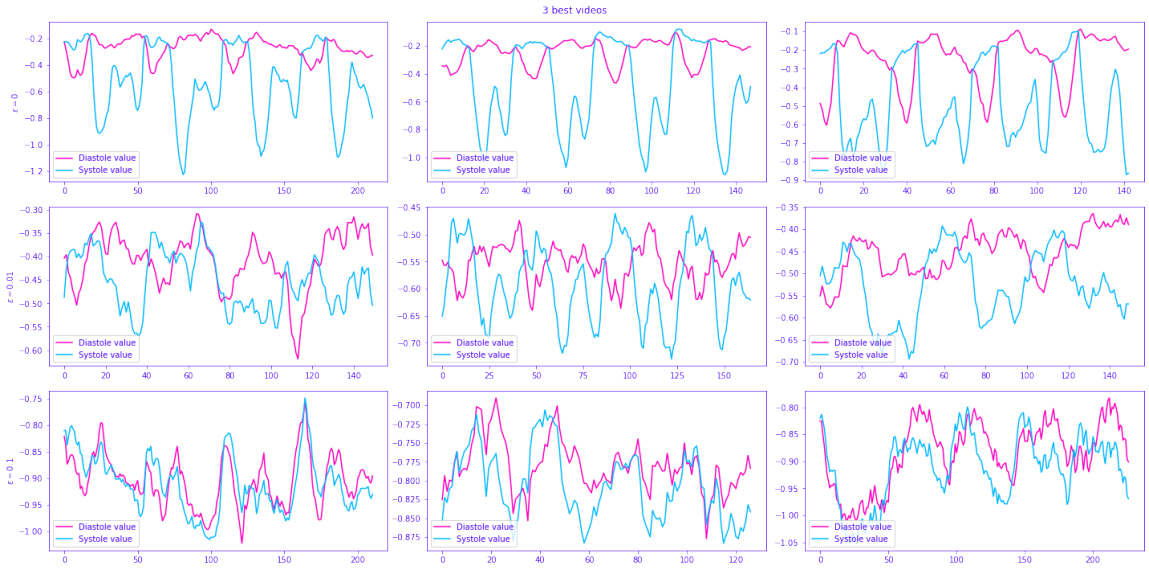


Figure 5.22: The Q-values for three of the best predicted videos for each model trained using R_{GaaFD}. Top row is the model with $\epsilon = 0$, middle row is the model with $\epsilon = 0.01$, and the bottom row is the model with $\epsilon = 0.1$. The x-axis represents time in the video.

5.6 Inference speed

The inference time is faster on the CPU than the GPU when we include IO roundtrip time. However, ignoring IO, the network performs extremely fast on the GPU; processing a batch of 128 frames takes just a little over a milliseconds, producing a framerate of over 125 000 FPS. This is merely considered a "fun-fact", however, and a realistic scenario would include IO roundtrip time. A single frame may be processed on the CPU in 0.80 milliseconds, meaning that it can likely be included as a step in a processing stream.

5.7 M-Mode Binary Classification Environment

A single experiment was run using the M-Mode Binary Classification Environment (MMBCE), the result of which can be seen in table 5.6. The percentage of episodes where the agent actively explores its environment by moving the synthetic m-mode line is reported in addition to the key metrics. The agent is said to have explored if at least one of the actions in the episode moved or rotated the line. The GaaFD is also reported

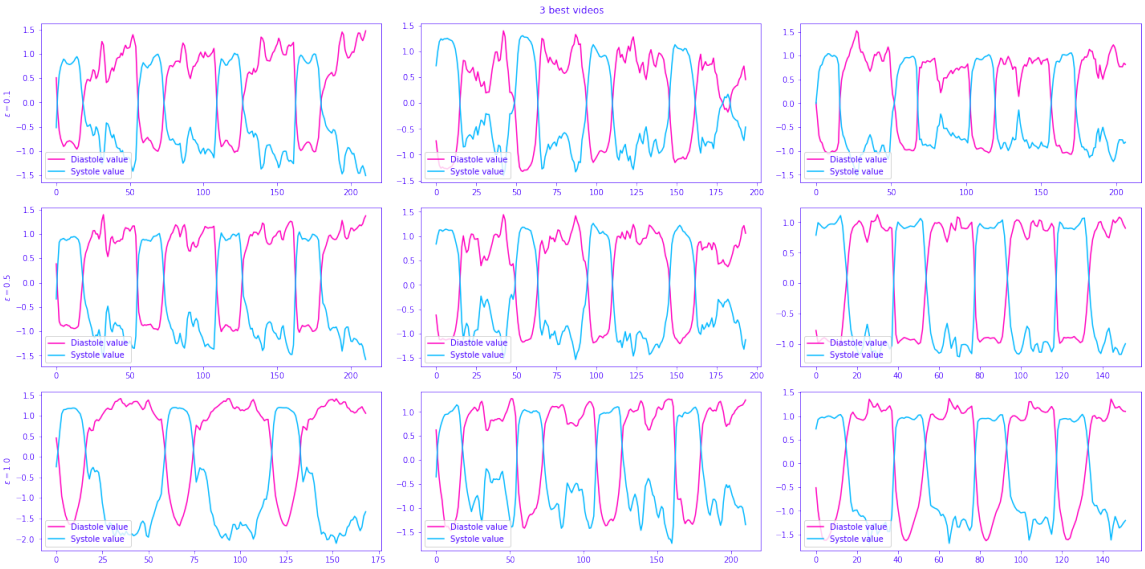


Figure 5.23: The Q-values for three of the best predicted videos for each model trained using R_{simple}. Top row is the model with $\epsilon = 0.1$, middle row is the model with $\epsilon = 0.5$, and the bottom row is the model with $\epsilon = 1.0$. The x-axis represents time in the video.

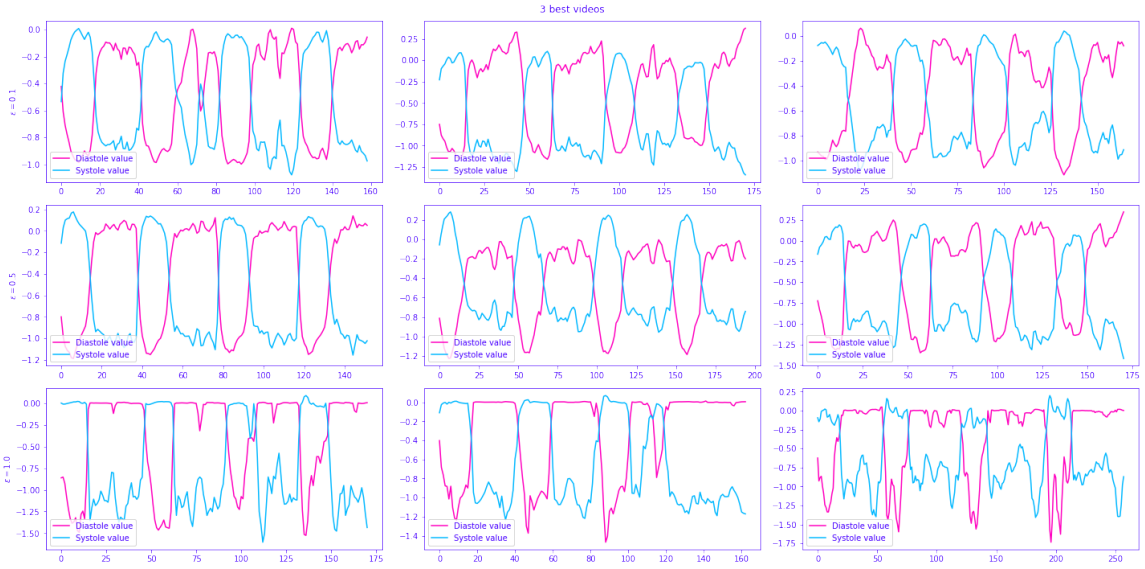


Figure 5.24: The Q-values for three of the best predicted videos for each model trained using $R_{proximity}$. Top row is the model with $\epsilon = 0.1$, middle row is the model with $\epsilon = 0.5$, and the bottom row is the model with $\epsilon = 1.0$. The x-axis represents time in the video.

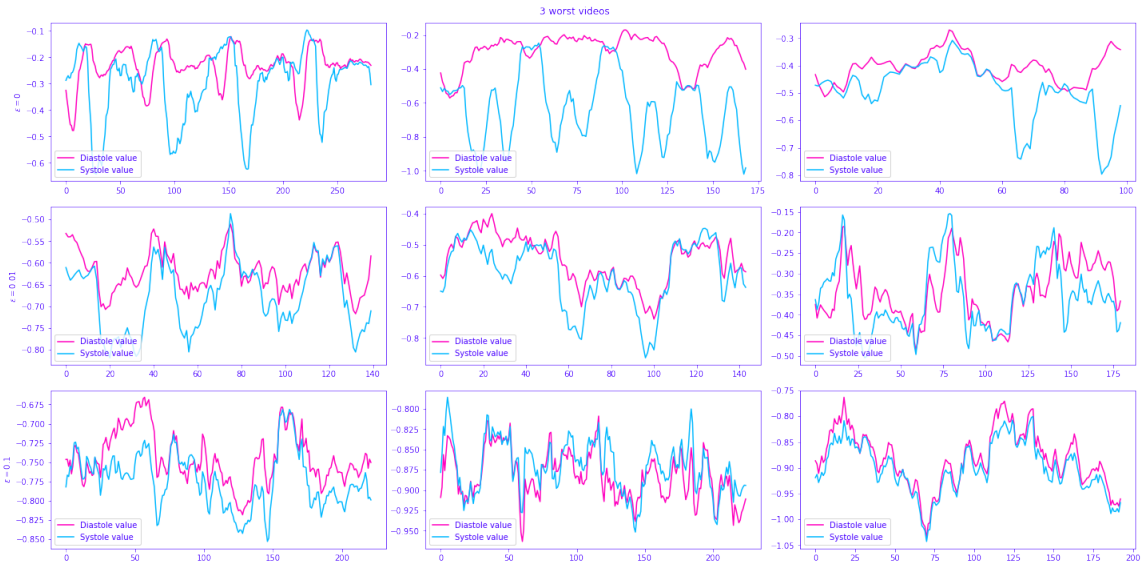


Figure 5.25: The Q-values for three of the worst predicted videos for each model trained using R_{GaaFD}. Top row is the model with $\epsilon = 0$, middle row is the model with $\epsilon = 0.01$, and the bottom row is the model with $\epsilon = 0.1$. The x-axis represents time in the video.

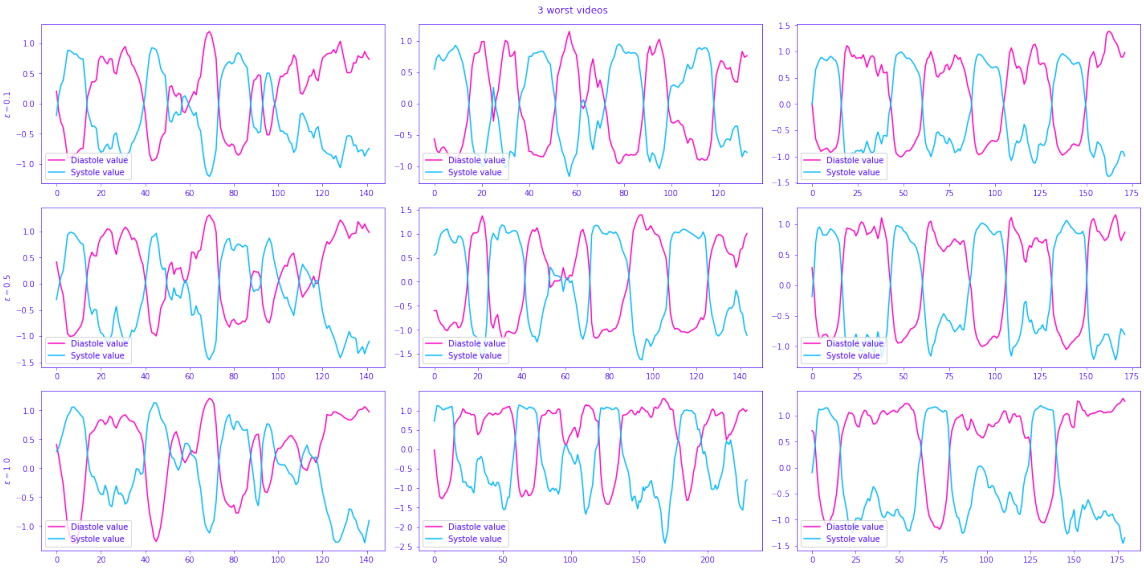


Figure 5.26: The Q-values for three of the worst predicted videos for each model trained using R_{simple}. Top row is the model with $\epsilon = 0.1$, middle row is the model with $\epsilon = 0.5$, and the bottom row is the model with $\epsilon = 1.0$. The x-axis represents time in the video.

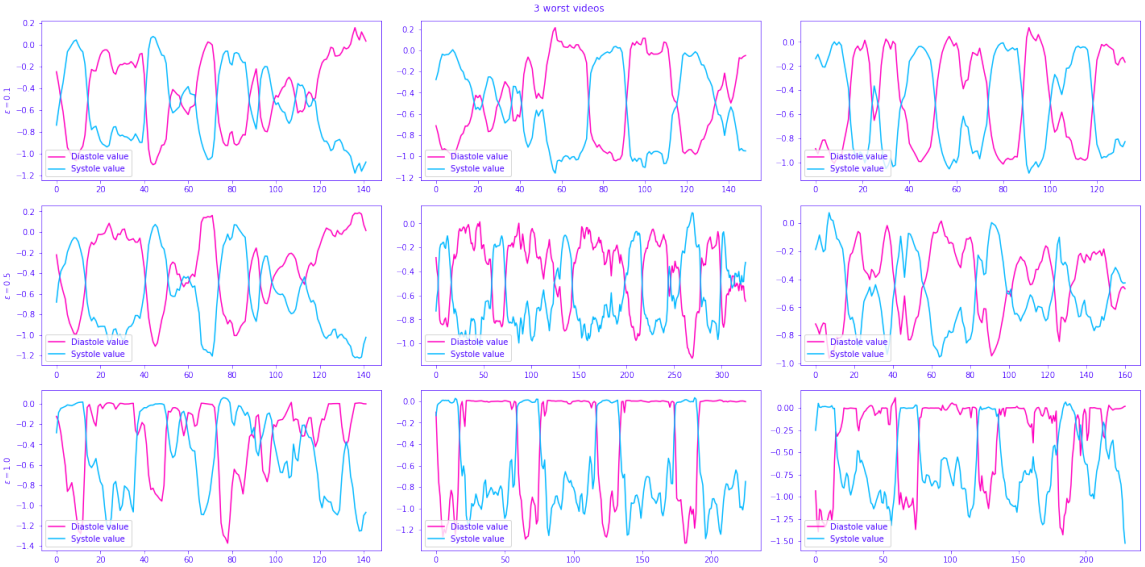


Figure 5.27: The Q-values for three of the worst predicted videos for each model trained using $R_{proximity}$. Top row is the model with $\epsilon = 0.1$, middle row is the model with $\epsilon = 0.5$, and the bottom row is the model with $\epsilon = 1.0$. The x-axis represents time in the video.

Table 5.5: The compilation time and average elapsed time over 1000 calls for the neural network, on the CPU and on the GPU, with or without IO overhead.

Device	# frames	Compilation time	Average run time
CPU	128 frames	273.95 ms	29.15 ms
	Single frame	205.93 ms	0.80 ms
GPU (including IO)	128 frames	2399.56 ms	34.43 ms
	Single frame	418.77 ms	2.88 ms
GPU (pre-placed data)	128 frames	251.10 ms	1.02 ms
	Single frame	285.56 ms	0.17 ms

for episodes where the agent performs some exploration and where it performs no exploration, individually.

The agent trained on the MMBCE performs worse than any agent trained on the BCE, as seen in table 5.6. It also performs very little exploration of the environment, where almost 35% of episodes contain no movement of the synthetic m-mode line at all. Figure 5.28 show the distribution of actions taken by the agent on the test split. Over 90% of actions were marking the current frame as either diastole or systole.

Furthermore, in the episodes where the agent *did* perform any type of exploration, the agent performed worse than in the ones it did not move the synthetic m-mode line at all, as visualized in 5.29. Table 5.6 report that the agent had an average GaaFD score of 5.47 for episodes where it performed exploration, versus 3.13 in episode where the line was still.

Table 5.6: Performance of agents trained on the m-mode binary classification environment.

Best model SGD step	23 080
GaaFD	4.66
GaaFD ED	4.85
GaaFD ES	4.37
% episodes with exploration	65.22%
% episodes without exploration	34.78%
GaaFD for episodes with exploration	5.47
GaaFD for episodes without exploration	3.13
% valid aaFD	59.29%
aaFD	2.22

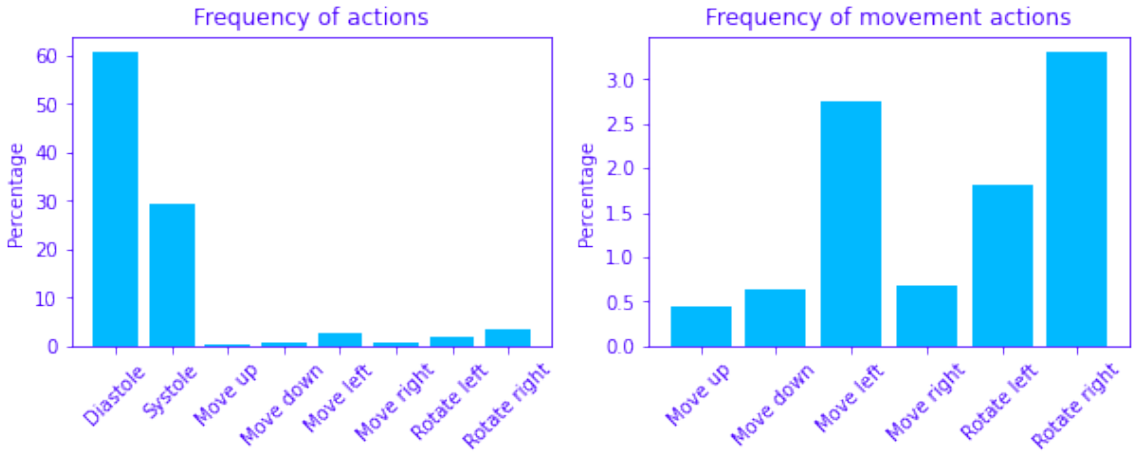


Figure 5.28: A bar chart showcasing the distribution of actions selected by the agent. To the left are all actions, while to the right are only movement actions, i.e. marking a frame as diastole or systole not included. The vast majority of actions are that of marking frames as diastole or systole.

And finally, the MMBCE agent is significantly slower at inference than

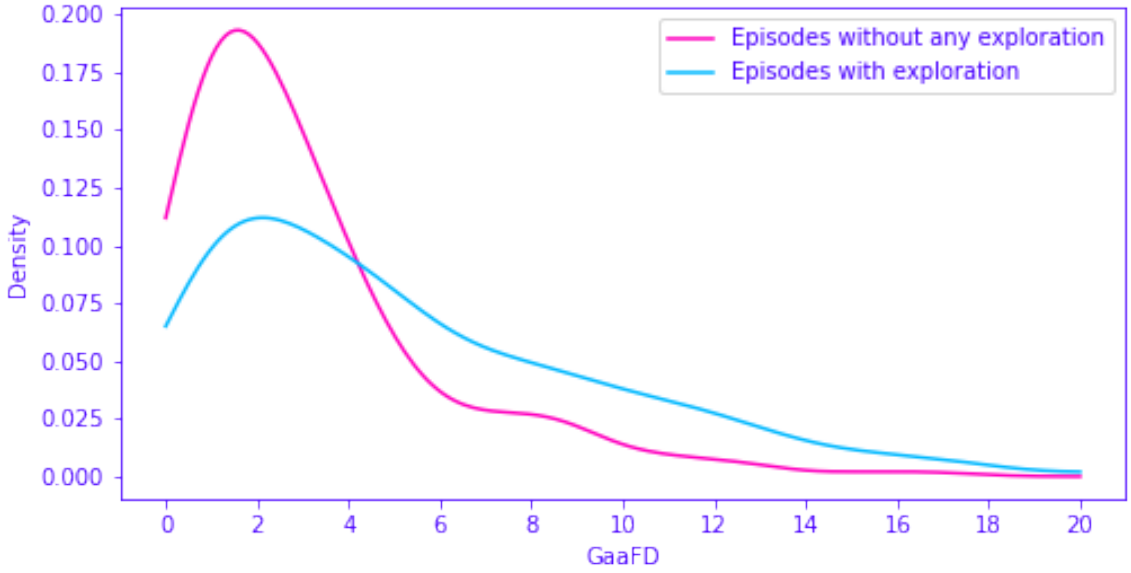


Figure 5.29: A density plot of GaaFD for episodes where the agent performed no other actions than marking frames as diastole or systole, i.e. no exploration, versus the density plot of GaaFD for episodes where the agent moved the synthetic m-mode line in any way at least once.

the BCE agents, taking multiple seconds to evaluate a full video of 128 frames. It also takes more than 3 times as long when running the agent on the GPU.

Table 5.7: The average compilation and run time for predicting the phase of 128 frames in a video (including IO overhead).

Device	Compilation time	Average run time for 128 steps
CPU	389.70 ms	3169.83 ms
GPU	2384.04 ms	9738.50 ms

Chapter 6

Discussion

6.1 Generalized Average Absolute Frame Difference

From the learning curves in figure 5.15 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 5.15. This is less apparent for 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 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 5.16. 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. 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.

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 5.3. 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 6.1.

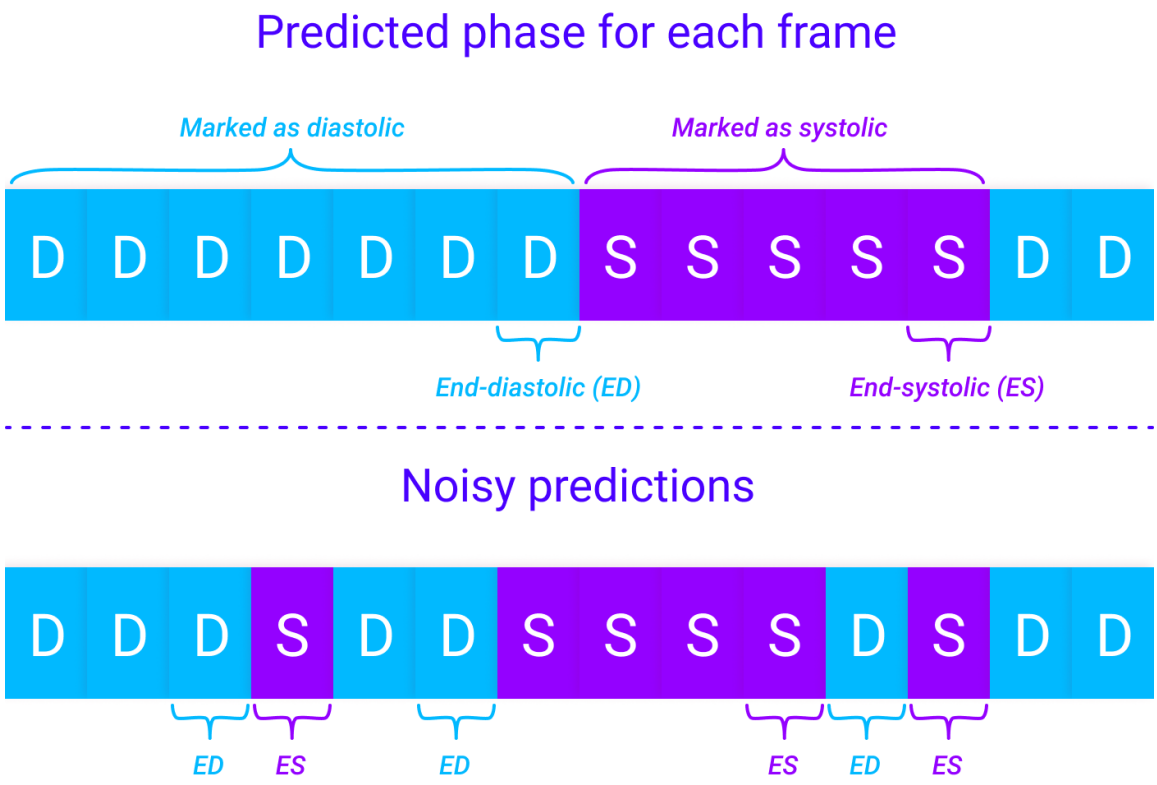


Figure 6.1: A single wrongly predicted phase that is corrected right after creates two incorrect events.

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 5.22 and 5.25. It is apparent that the model that uses $\epsilon = 0$ is better at differentiating the value of taking either action for a given state.

6.2 Simple- and Proximity-Based Reward Functions

From the learning curves in figure 5.17 and figure 5.18 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_{simple} and $R_{proximity}$ — R_{simple} and $R_{proximity}$ simply provides information more efficiently, and with less noise, to the learner.

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.

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.

However, despite the regularization, the agent is still able to achieve perfect accuracy and GaaFD on the training set. Figure 5.19 plots the learning curves of the agents on both the training set and the validation set.

The effect of the reward function is also visible in the learning curves if we plot them for agents trained with R_{simple} and agents trained with $R_{proximity}$ together, as in figure 5.21. The blue curves are the performance of agents trained using R_{simple} and the pink curves are the performance of agents trained using $R_{proximity}$. 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 $R_{proximity}$ 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_{simple} 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 $R_{proximity}$ reward makes more errors compared to an agent trained using R_{simple}, 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_{simple} outperforms $R_{proximity}$. This is likely due to $R_{proximity}$ being a more difficult function to estimate, as its values span multiple values, while R_{simple} can only be either 1 or -1.

Compared to the loss curves of agents trained using GaaFD as the reward, agents trained using R_{simple} and $R_{proximity}$ yield a more familiar-looking loss curve. The loss curves in figure 5.20 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_{simple} and $R_{proximity}$, 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.

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 5.6 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 $\epsilon = 1.0$.

6.3 Why Use Reinforcement Learning?

In the experiments that uses the reward function R_{GaaFD} we have seen that the RL agent is able to learn from a very sparse reward signal. This makes RL a very general tool that can be used when supervised learning methods are not applicable. However, there's no such thing as a free lunch, and bringing in the whole RL machinery for a classification task brings a lot of complexity.

The methods that have been used in this thesis suffer from low data efficiency compared to a supervised learning approach. Each sample given to the learner is of only one of the phases, as only one phase is predicted at every step. We also applied additional data sampling constraints on the learner by enforcing that a data item should be sampled 0.5 on average, i.e. half of the data is discarded. For the formulations where the agent's actions don't affect future actions, such as in BCE with either phase classification reward function R_{simple} and $R_{proximity}$, the sampling should arguably be at least 1. The value of 0.5 for these experiments was due to running the experiments with R_{GaaFD} first, and it was overlooked in subsequent experiments.

The use of the environment abstraction for sampling was also a big performance bottleneck for the BCE ennvironment. Every new sample had to be created by stepping through the environment which occured on the CPU. For the BCU environment, whose next states were completely independent on the action taken by the actors, this abstraction limited us from batching the inference on the GPU. Because the environment abstraction is such an important part of the RL ecosystem and libraries we still opted to use it.

A synthetic m-mode version of BCE was included to give RL a fair shot

at proving its usefulness. The search for an m-mode to base its decisions off of is something that would be very difficult to solve for using supervised learning, and here RL is assumed to actually be the best tool for the job. However, this was a very hard problem to solve, presumably too complex for the current setup, and the benefits are not all apparent. Even if it did work, its inference would have been much slower than that of BCE, since there would be no way of batching forward passes through the neural network.

However, even though RL may not be the best tool for the job of ED-/ES-frame detection, it is still a promosing techonology, especially for problems that require exploration.

6.4 Lack of Comparison Experiments

Only one other study was found to report their model's performance on the Echonet dataset [**lane_multibeat_2021**]. This is considered a weakness of this project and getting access to multiple datasets early in the project should have been a priority. This makes it hard to gain precise insight into the performance of the methods versus supervised learning methods.

In their paper the authors report an average aaFD of 2.30 and 3.49 for ED and ES events. Our best model can report an average aaFD of 1.69 overall, but this is only for 80% of the videos as 20% have an incorrect number of predicted events with respect to ground truth events. The authors does not report how many, if any, of the videos had an incorrect number of predicted events.

Chapter 7

Conclusion and Further Work

We are now ready to answer the original research questions:

Is it possible to use Reinforcement Learning for the task of ED-/ES-frame detection? Yes, there are multiple ways, but some are more efficient than others. Rl is very flexible and allows us to model the problem in many different ways, both in terms of reward functions and environment dynamics.

How does the formulation of the problem as a Reinforcement Learning problem affect the performance of the model? We have seen that the design of the reward function matters a lot. If the reward signal is too sparse then it will be more difficult for the agent to learn, which may result in poorer performance, as seen in our experiments. Our test results comparing the reward function R_{simple} and $R_{proximity}$ shows that R_{simple} performs better at aaFD, even though $R_{proximity}$ was specifically designed to be more similar to aaFD. This is despite the learning curves seen in figure 5.21 indicating that $R_{proximity}$ performs better at GaaFD after the models have begun to overfit, because R_{simple} is able to reach a better score just before overfitting occurs.

Furthermore, the value of ϵ has been seen to be an important hyper parameter for these tasks. For the sparse GaaFD reward function, using a value of $\epsilon = 0$ yielded the best results. For R_{simple} and $R_{proximity}$, the difference between different values of ϵ had little effect on the performance of the best model, but seemed to have a regularizing effect. It would be interesting to dive deeper into why this happens, and perhaps how we can take advantage of it inwhen sampling data for supervised learning.