Investigating

Behavioural and Affective Cloning

via

Imitation and Reinforcement Learning

Kapenekakis Antheas

A thesis presented for the degree of:

Diploma of Electrical Engineering and Computer Engineering

Thesis Committee

Prof. Minos Garofalakis
Associate Prof. Michail G. Lagoudakis
Prof. Georgios N. Yannakakis (University of Malta)

School of Electrical Engineering and Computer Engineering

Technical University of Crete
Chania, Greece

June, 2021

Abstract

In recent years, studying affect in computer-user interactions, video games, and even live streams has become increasingly popular. Objectively measuring the emotional experience of an audience has important implications in revenue generation and user retention. What has been relatively unstudied is user modeling by affect, especially in the context of video games. In this thesis, an initial framework and proof-of-concept for creating an affective agent is presented, which leverages user provided annotations to change agent behavior towards displaying a specific emotion while trying to complete a behavioral objective.

To create this agent, a tool for sourcing annotations was created and used to form a dataset with affective annotations. Then, the dataset was tested for validity by running supervised learning experiments. Using a form of Deep Q Learning, along with the dataset, a set of agents was created with each having a different objective. A primary Reinforcement Learning agent focused on completing the environment and each of the rest focused on maximizing an emotion. Lastly, the set of agents was combined in a ratio to form composite agents that focused on both affect and behavior, with certain combinations being successful at both.

Acknowledgements

The completion of this thesis marks the end of and is the fruition of five years of effort in attaining my Diploma in Electrical Engineering and Computer Engineering.

First and foremost, I would like to thank my advisor, Prof. Georgios N. Yannakakis for his continued support throughout this year, constant advice, and insistence to push through in achieving the initial goals for this thesis, even as doubt crept in. I also thank Konstantinos Makantasis, for his advice in thesis related subjects and, especially in the beginning, for helping me create the foundation of this thesis.

I am appreciative of my friends who've been with me throughout this journey, in these five years of personal growth, and my Mom and sister who've supported me, increasingly so in this last year, as I've been working on completing this thesis.

Contents

Chapter 1

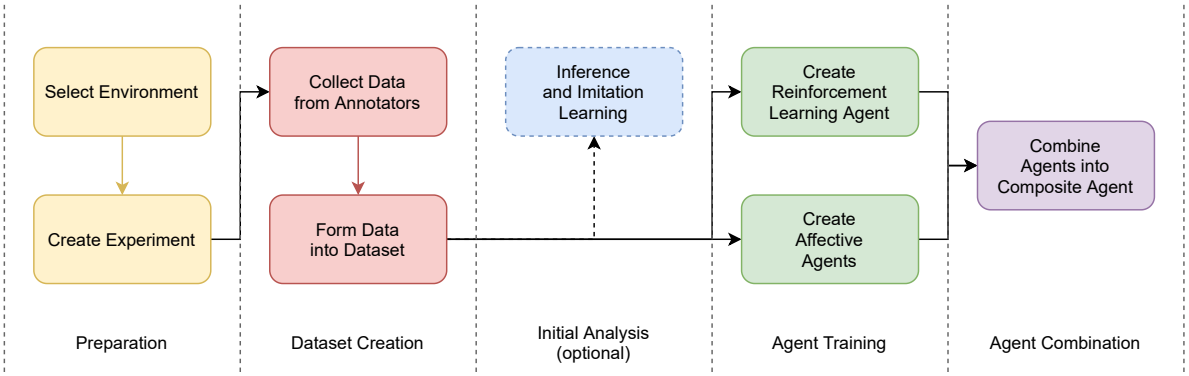
Introduction

This thesis was setup with an ambitious goal: creating an agent who considers emotion while completing a primary objective. In the field of Affective Computing, there have been efforts in combining Reinforcement Learning and Emotion, as well as the study of emotion on its own [12]. However, integrating emotion into an agent has mostly focused on improving performance. In this thesis, the objective will be on modeling the affect and the behavior of an agent via the Reinforcement Learning paradigm, while not compromising the agent's performance.

The subject of this thesis, being a novel pursuit, created a number of challenges which had to be overcome; from selecting the environment to creating objective metrics for measuring a subjective notion. Each step in creating the agent has been split into its own self-contained chapter, consisting of an introduction, methodology and results of experiments, if the chapter includes them. In total, 6 chapters comprise the body of this thesis, an overview of which will be covered in brief in the next section.

1 Thesis Contribution

The main contributions that stem from this thesis are an end-to-end process for creating affective agents and a set of tools for undergoing that process. The process for creating an agent is outlined below and consists of 8 steps, split into 5 parts that mirror the structure of the thesis. The initial steps focus on the creation of a dataset that contains affective annotations, using an annotation tool developed for this thesis. Following, there's an optional analysis step that tests the dataset using supervised learning to show its validity. The next steps use the dataset to assist in creating a conventional Deep Q Learning agent and to train a set of a affective agents. Lastly, the agents developed are combined in order to create an agent that considers both emotion and its primary objective.



2 Thesis Structure

Emotion is a human centric notion, that has to be sourced from people, who can take the form of annotators, experiencing the environment and relaying their emotions. While affective annotation datasets exist, there is no candidate dataset which combines affective annotations with a Reinforcement Learning friendly environment. Therefore, one will have to be created, presenting an opportunity to start from scratch and select the tools that will be well tailored for this pursuit. The first three content chapters cover the creation and study of this dataset, along with the technical choices that were made.

Chapter 2 explains the technology choices that were taken, as well as the tool that was created to source annotations for the affective dataset. OpenAI's Retro environment [13] was chosen as the environment platform and a desktop-based annotation app was created.

In the next chapter, Chapter 3, the simplistic dataset that was created for this thesis is presented, which is objective in nature, along with its qualitative data. It features the game Super Mario Bros (NES), due to its popularity, and consists of 54 first-party annotated 2 minute playthroughs that are each annotated according to the Pleasure Arousal Dominance (PAD) psychological model, with binary annotations. The playthroughs are limited to the first world of Super Mario Bros.

Lastly, in Chapter 4, an initial inference experiment is created and performed on the dataset. The experiment is comprised of state-action pair to affect inference and imitation learning. The goal of this experiment is two fold: to show the validity of the dataset and create an initial case study for the thesis while solving a simpler problem. Hyperparameter tuning is much simpler on a supervised learning problem, and the model that was created in this chapter, with its hyperparameters, is carried over to the next chapters as is.

The next three chapters focus on creating the agent. Deep Q Learning (DQN) was selected as the base algorithm, incorporating improvements from Deep Q Learning from Demonstrations [6] so the dataset can be used as demonstrations.

In Chapter 5, a base DQN agent is created that focuses on completing the first level of Super Mario Bros (NES). Performance varies between various runs and training steps, but once in a while the model weights are such that the agent completes the level from start to finish most of the time. The weights of that agent are carried over to the next chapters.

Chapter 6 creates 3 Affective (AF) agents (one for each emotion) that have as a goal to maximize the expression of their respective emotion. They use a modified form of Deep Q Learning from Demonstrations that works in an offline manner (using just demonstrations) while remaining compatible with the RL agent. Two objective metrics for affective learning are developed, in order to measure affective performance. Those are imitation percent and imitation action rank when there is an annotation. Transfer learning using the base model layers of the RL agent is utilized to improve and stabilize the Affective agent performance.

In the final chapter, Chapter 7, composite agents are created by combining the RL and AF agent's Q functions in a ratio. The composite agents for each emotion perform better than just the AF agent and produce visually different playstyles, which are presented using a tool developed for this chapter. The different playstyles are different in the ways that are expected for each emotion and are up to the expectations that were set for this thesis, rendering it a success.

A conclusion and future works chapter follows, Chapter 8, which closes this thesis. In it, the limitations of the dataset and training paradigm are discussed, as well as ideas and possible adaptations for future work.

Chapter 2

Annotation Tool

During the research for this topic it was observed that a candidate dataset for this problem does not exist. Furthermore, little work has been done in annotating actual Reinforcement Learning environments with any state, not just emotional. As such, the work of this thesis would have to begin with producing a tool and a workflow for collecting this type of data.

In this chapter, we'll discuss the process in which a tool for this purpose was developed, by going through the project requirements, the reasons behind the technologies chosen, the prior work it was based on, and the workflow by which a dataset can be created.

1 Objective

The purpose of this tool will be to produce one or more high quality datasets which consist of short gameplay vignettes, in which a human agent plays a segment of a game and annotates how he felt. Ideally, the data will be composed of the unchanged source state/action pairs and the highest quality traces for a handful of annotations. Furthermore, since the annotations will have a degree of subjectivity, multiple annotators should be used, so as to cancel out their subjectivity and reach a ground truth. Optionally, the source environment should be stimulating and entertaining, to motivate the annotator to produce varied, high-quality data and it should offer a large degree of freedom, so that the annotations can be then used to influence the decisions of an agent.

2 Requirements

Having listed the target objective for this tool, we can now use it to derive its requirements. The requirements will be split into functional requirements and nonfunctional requirements, where the former describe the core elements of the tool and the latter describe the way the tool will operate.

2.1 Functional Requirements

The functional requirements are the following:

Allow Human control of a traditional reinforcement learning environment A human agent or player should be able to take control of an environment and play through it as if he is a reinforcement learning agent, with intuitive and familiar controls.

Allow Annotation of Environment traces Starting from a collection of recordings, a human annotator should be able to go through them and add annotations. This is equivalent to third person annotations.

Allow for First Person Annotations By combining the two previous requirements, a human annotator should be able to perform first person annotations, by first playing a section of the environment and then replaying it while performing the annotation.

Allow For Fixing Mistakes The tool should be designed so as to allow the user to undo parts of his annotation and redo them without having to start over.

Allow For Playback After performing an annotation, both the annotator and the experiment coordinator should be able to play it back, to verify it was done correctly.

Allow for Defining Annotations The experiment coordinator should be able to define the annotations that will be used for the production of the dataset, by listing their name, type and description. The type of annotations will be analyzed in a later section.

Allow for Designing the Environment The experiment coordinator should be able to, without much effort, create one or more master traces of the environment, in which he plays a large section of it and record the states in it. Afterwards, by going through the traces, he should be able to dump initial states, which can then be shuffled and randomly chosen for the annotator playthroughs.

2.2 Nonfunctional Requirements

The nonfunctional requirements concern data quality, reliability, and ease of use.

Record Complete Source Data The data produced by the tool should be close to or equivalent to those a reinforcement learning agent would use to train on. This requirement is as much concerned with completeness (storing all available data such as actions, sensor activations) as with quality (avoiding the use of lossy compression).

Limit the Recording Space The annotation data should be able to be transmitted from the annotator's computer to a central server, assuming for a poor internet connection while at the same time taking into account the disk space that will be required in the annotators computer. As such, after a typical annotator has finished annotating, the tool shall take no more than 512 MB on his computer and require less than 128 MB of data to be uploaded.

Allow for Multiple Inputs The tool should conform to the user, and as such should allow for the use of a controller or the keyboard and contain alternative keybinds to conform to the needs of the annotator.

Allow for custom Keymaps The experiment coordinator should be able to define the mapping of key bindings to actions in the environment, per environment, and key bindings to annotation, per annotation type. Some annotation types may be best suited to using the scroll wheel, while others to key presses.

Support Custom Color Palettes Annotators may need to trace more than one annotation per session. If the annotations are only distinguished by their name, then the annotator will have to glance over and read it every time he forgets. Even worse, if by confusion the wrong emotion is annotated, the data will be invalid. A way for solving this would be to add customizable color palettes, so the current annotation is distinguishable at a glance, by color.

Window Size similar to Fovea The window size of the tool and all the information on it shall be around the size of the Fovea of the annotator (the part of the eye that is the clearest), which is around the size of the palm of his hand when held at arm's length. The annotator shouldn't redirect his gaze to perceive the current annotation state, thereby losing focus of the environment.

Portability The tool should be able to run on any computer, without the need of installation of itself or any other dependency, and after the annotation is done it should be removed without leaving a trace.

3 Technology Choices

In this section we'll analyze the technologies that were considered for this project, the ones that were chosen, and the reason behind those choices.

3.1 Backend Language

The natural choice of language of this tool is Python, since that is where most Reinforcement Learning research is performed on, and as such it has the largest number of environments with the greatest support. The alternatives, JavaScript, Java, and CPP, either support few reinforcement learning environments or offer no actual advantages over Python. An added benefit of Python is that the code that will produce the environment will be exactly the same as the one that will be used to train on.

3.2 Environment Technology

For the underlying technology, Unity and OpenAI's Retro Gym were considered, ultimately opting for OpenAI Retro.

3.2.0 A Case for Unity

Unity has become a strong contender in the research and indie development fields due to its flexible license model and its affordable asset marketplace. By using Unity, a sole developer can quickly and affordably produce a product that uses high quality assets.

Furthermore, Unity has made investments in the ML field by producing a library called ML-agents [7], which allows for connecting python ML algorithms, made with PyTorch and TensorFlow, to the Unity environment. The ML-agents toolkit allows a researcher to create custom environments and custom agents which can then pipe data into the python interpreter and be used to train a model. Afterwards, Unity features built-in support for Neural Networks, so the final weights can then be integrated into it and used to control an agent in an end product. On the other hand, Unity also has serious drawbacks, which is why it wasn't chosen.

Unity has a high resource requirement, so it would take up valuable resources from the training algorithm and would limit sample collection speed. That can somewhat be dealt with by training multiple agents in parallel, each in a little pen, but that would limit the richness of the environment. Similarly, due to its resource requirements it would be difficult to set up training in the cloud, so training would have to be done locally.

Moreover, Unity, being a modern engine, features a huge state space which would be impossible to input into an agent. As such, it would be impractical to feed the agent 4k renders or what is shown to the annotator. It's more likely that a collection of virtual sensors would be placed on the agent and they'd be the ones to feed him with data. This also raises questions of what the ideal storage format of the traces would be.

Therefore, the environments used for training would have to be handcrafted and feature different latent spaces as input for human agents, computer agents, and recording format, placing an undue burden to an experiment coordinator.

In conclusion, for the above reasons, the use of Unity does not meet this project's requirements.

3.2.0 A Case for OpenAI Retro

Other than Unity, OpenAI's Gym [2] interface has dominated the reinforcement learning space. It has done so in such a degree that a majority of environments provide a gym interface. Having excluded Unity, OpenAI

Gym becomes the frontier choice for the tool and a secondary choice arises: which Gym environment should then be chosen.

After weighing performance and variety in the project's requirements, OpenAI's Retro project was selected. Retro [13] is a gym environment built on top of RetroArch, a framework of emulators of vintage consoles, and provides bindings and reward functions for thousands of games, while enabling the addition of new ones in a simple manner. In addition, compared to similar projects, it's well supported and mature. As such, it provides the required variety for the project and features acceptable performance.

More importantly, Retro is deterministic and provides a data format for saving traces and states into files. Those files have a size of around 1 kB, so they can be committed into git repositories and sent through the internet with ease, including by email.

For performance considerations, Retro is a single threaded singleton instance that can run at around 1 thousand frames per second, or 15X real time. By using Python's subprocessing module it can then be duplicated so as to use all available processing power, if parallel learning can be opted for, leading to a performance of 1k frames per CPU core. The emulators feature SD resolutions, so they can be fed straight into a CNN without pre-processing if desired, even while being trained by consumer hardware.

3.2.0 Conclusion

By taking the above into account, the tool was designed to support the OpenAI gym interface, with a deeper integration with OpenAI Retro's features (rewinding, audio, and states) when that is the annotation environment.

3.3 Frontend Framework/Language

With Python being a versatile language, there are a variety of choices that were considered for the front-end. The front-end is defined as the technology with which the user interacts with. 3 options were under consideration: Pygame, Tk (with Matplotlib), and the Browser.

3.3.0 Tk

Tk is a legacy GUI framework developed for Unix in the 90s and is still widely used to this day. Python features first-party support for it and a wrapper for creating interfaces with it, which is why it was considered. Since the project has limited need for GUI elements and Tk doesn't feature gaming features such as joystick support, robust raster options and high framerates it wasn't chosen. Matplotlib had considerable performance issues and wasn't visually appealing.

3.3.0 Browser

Choosing the browser as the frontend was very appealing. After all, the volunteer annotator wouldn't have to download anything and his annotations would be securely stored in the cloud, preventing data loss. However, maintaining the use of the Retro environment would mean having to render the game in the server and then streaming the frames to the client. This would involve custom work integrating WebRTC into the client, if optimal performance is required, and even then the latency added would be considerable.

3.3.0 Pygame

Pygame is a simple 20 year old framework for creating games in Python. It features classes for raster drawing, audio, and key inputs, with acceptable performance. Using bit block transfer (BitBLT), a render of the gym environment can be printed to the screen. The built in classes provide support for joystick, mouse, and keyboard. Furthermore, the surface class can be used to draw a visually appealing annotator, based on David Melhart's work on PAGAN [9]. One notable downside of Pygame is the lack of antialiasing, so graphics produced by it are jagged. However, that is an acceptable compromise.

4 Annotations

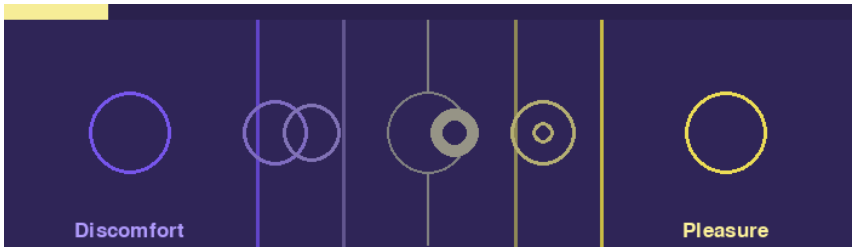
Building on the work done by Melhart et al. [9] on the tool PAGAN, the support for three different annotation types was added. Those are: RankTrace, BinaryTrace and GTrace, which are based on previous work in the affective computing field.

4.1 RankTrace

RankTrace allows for the unbounded annotation of change in emotion. The gradient of the annotation gives the moments in which there is a change in emotional charge. However, due to the unbounded nature of the annotation, the current level of an annotation can't be compared between annotators or within the same annotation trace, since due to its unbounded nature there will be constant drifting away from 0.

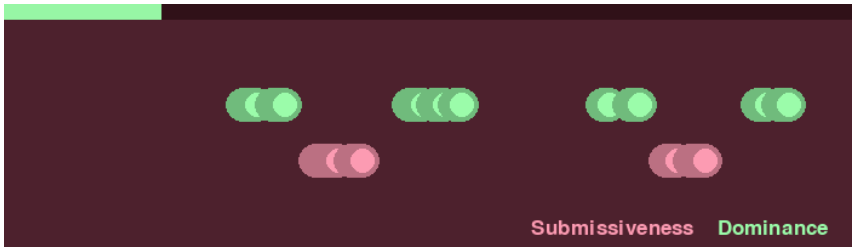
4.2 GTrace

GTrace is an ordinal annotation tool, in which the subject notes his emotional state as a percentage using his judgment. Since it is ordinal, it then becomes possible to compare the inter-rater agreement between two annotators. However, it requires more attention to use while annotating.



4.3 BinaryTrace

Binary trace allows the user to press buttons to indicate positive and negative reinforcement. For fast paced events where the user wouldn't have time to raise and lower RankTrace, BinaryTrace can be used as a good substitute.



5 Performing an Experiment

In this section, we'll go over the steps of undertaking an experiment using the annotation tool, from the view of the experiment coordinator (the individual coordinating the experiment).

5.1 Defining the Experiment

Starting off, the experiment coordinator should define the experiment parameters. Those include the Game or environment that will be tested, the length per trace that will be recorded and the annotations that will be collected. A trace is a short vignette that is performed by the annotator in one go and after it is done it is replayed for annotating. During a preliminary analysis of the problem, the experiment coordinator should also note the sections of the environment that will be annotated and how much time it's estimated it will take. Depending on the available annotators, this will determine the total time investment that will be required by each annotator and how many traces each one should complete.

5.2 Preparing the Environment

Having performed a preliminary analysis of the experiment, the experiment coordinator will move on to creating the archive that will be handed to each annotator for the purposes of annotating (from here on called the "distributable"). We'll assume a Retro environment will be used, since scripts have been created to aid in this process.

5.2.0 Setting up the Environment

The experiment coordinator should begin by procuring the ROM of the environment that is desired. If the ROM has been integrated into the Retro environment already, then its folder should be copied from Retro and modified as seen fit. Otherwise, by using OpenAI's retro integration tool an integration can be created. After finishing this step, the experiment coordinator should have placed the integration folder along with the ROM into the experiment folder. This will be important in packaging, since if the experimenter would opt to include integrations for all the games included in Retro in the executable, he would increase the executable by hundreds of megabytes.

5.2.0 Performing Playthroughs

Retro environments are deterministic. Thereby, by having a certain section of the environment in mind, the experimenter can devise a number of playthroughs that provide a high degree of coverage within the solution space of the environment. These playthroughs will provide the starting places for the annotators, so they should be thorough enough to allow access to the desired state space within the trace time limit, but not more, so as to intrude into the annotator's decision process as minimally as possible.

The experiment coordinator will now perform these playthroughs using the example scripts provided. Within the playthrough environment, the experimenter gains access to a rewind function, so he can sculpt the play trace completely as he sees fit. As an example, in Mario, the killing of enemies or dying can be avoided in order to ensure that the score isn't interfered with and that the starting lives are fixed. After performing the trace in a satisfactory manner and hitting a keybind, the experimenter will see a replay of the trace to verify it's correct. When the replay is finished, the play trace will be saved to the folder and with the prefix that were provided.

5.3 Creating a Pool of Starting States

Having a number of playthroughs in his possession, the experimenter will now dump the starting states for the annotators. By using a script similar to the example ones provided he will launch a replay of the playthrough for dumping states. In this mode, he gains access to fast forward and rewind functions. When the playthrough reaches a state which is desirable, the experimenter can, by pressing a button, dump it into

a file suffixed by the current frame. Attention should be paid for these states to be uniformly sampled since if they are not, by being selected at random, they will cause the dataset to contain an overrepresentation of a certain part of the state space.

5.4 Packaging

Having the states and the integration files in his possession, the experimenter can now move into packaging the experiment for distribution.

By referencing the spec and script files provided the experimenter will create an executable using pyinstaller. This executable is completely stand alone and doesn't leave a trace on the system after running. The only action required by the annotator for it to run is opening it.

A pdf file with instructions can be supplied alongside the executable to prepare the annotator. It would be impractical and tedious for those instructions to be included into the executable and it is one of the downsides of choosing this technology stack. However, the reader is encouraged to try if he wishes.

The executable with the pdf should then be placed into a folder and that folder should be zipped to produce the annotator's distributable. It is important for the zip to have an inner folder to increase the chance that it will be unzipped and, since the recording folder is created alongside the executable, it will reduce the probability of the recordings being lost.

In the time of writing, the distributable size is around 60 MB, with most of the space being taken up by numpy optimization binaries. They could possibly be removed by compiling numpy without them, which would halve the size of the distributable.

6 Recording a Trace

In the following section we'll accompany an annotator as he records and annotates a trace. The annotation will take place in the game Super Mario Bros (NES), using the distributable built for the next chapter.

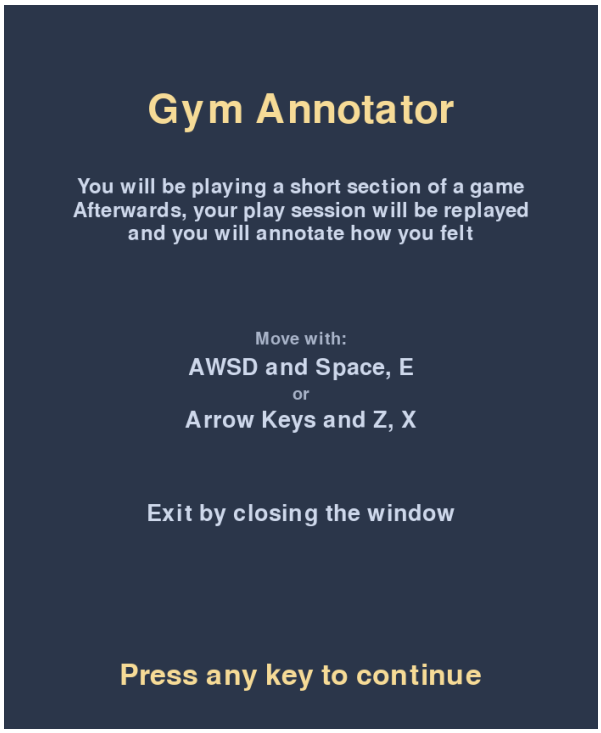
6.1 Opening the Executable

When the executable opens, the annotator is presented with a splash screen that lists the basic game controls and waits for a keypress. The annotator is given the option to play with the arrow keys or WASD. In addition, although undocumented in the distributable, there is support for XBOX 360 (and similar) controllers. It is assumed most annotators won't have access to a controller so its not included in the instructions.

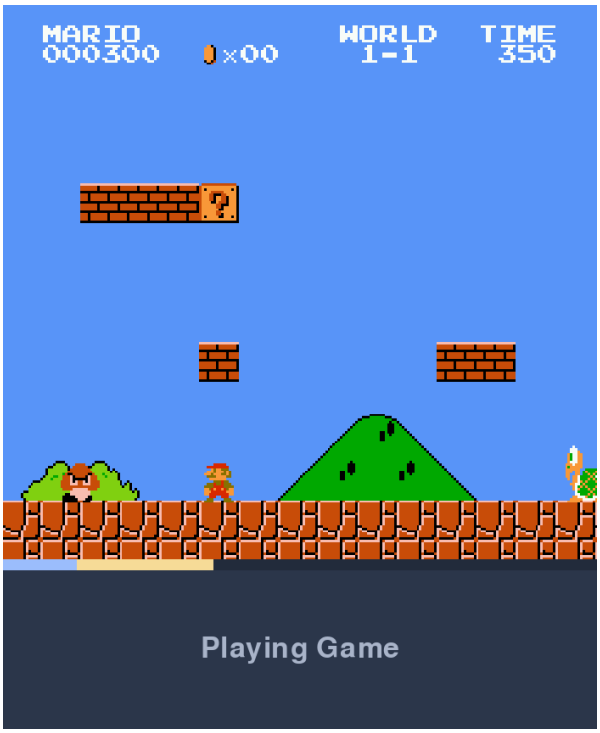
6.2 Starting to Play

After pressing any key (or button), the annotator begins the play trace. Rewind is disabled to capture candid gameplay, so the annotator is forced to go through unpleasant states.

There are two ways for the game to end: losing completely (all lives, in which case the gym environment sets done to true) and for the maximum trace length set during the design to elapse. Above the (disabled) annotator there is a two tone progress bar to represent the progression of time. For the case when the environment completes, and to avoid low quality recordings, the progress bar has a blue section in the beginning. If the game stops before the blue section completes, the recording is scrapped and the annotation part is skipped.



(a) Initial Splash Screen



(b) Playing the Game

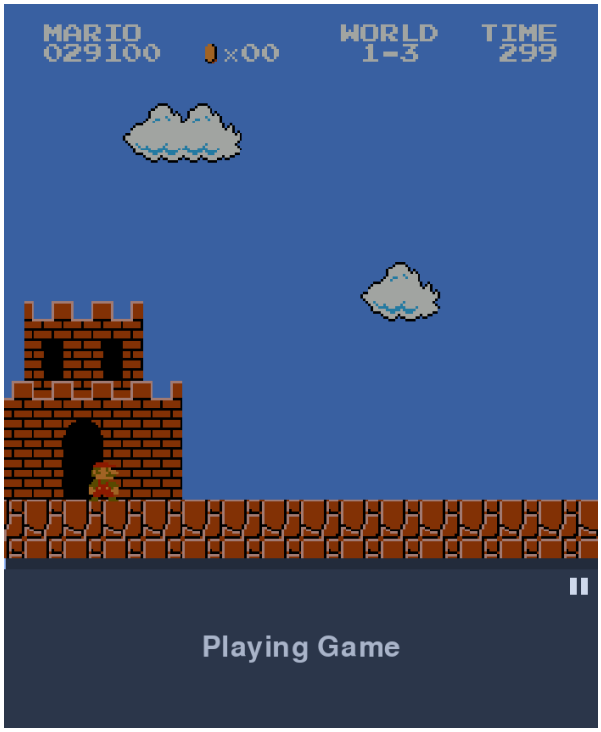
6.3 Pausing Feature

If the annotation tool loses focus or the annotator hits the pause keybind, the playthrough is paused as shown below. This function works both when annotating and playing and allows the annotator to take a break. If the annotator wants to stop, the annotation tool can be closed at any time by hitting the x on top of the program, and only the last halfway done annotation will be lost.

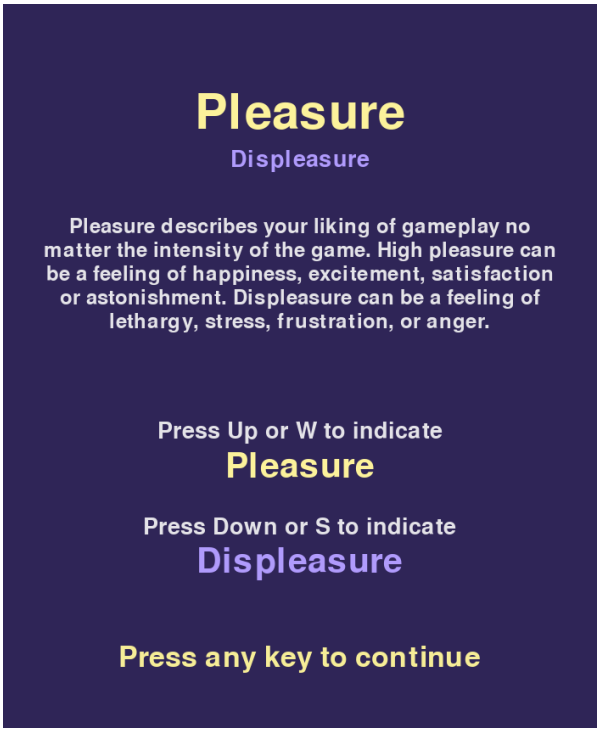
6.4 Annotation Introduction

After the play trace finishes, assuming it's longer than the minimum length, the annotator is presented with an introduction to the annotation he will be asked to perform. This introduction, supplemental to the one provided in the pdf, provides a simple description of the annotation and the controls that can be used for it. Depending on the annotation, the presented controls will change. For example, when annotating with RankTrace, the scroll wheel should be used, where with BinaryTrace it's more convenient to use key presses.

Since the annotations in the following section will be done according to the PAD model and there are three of them, three distinct palettes were created to represent them. For pleasure, yellow and purple were chosen as the main hues.



(a) Paused Game State



(b) Annotation Introduction

6.5 Performing an Annotation

After skipping the introduction, the trace which was recorded previously is replayed and the annotator is tasked with annotating it. The box that was previously reserved with the words "Playing Game" is now replaced with the annotator and the progress bar now represents the final length of the trace. The annotator is now given the ability to rewind and fast forward, so as to ensure the highest annotation quality, by allowing him to correct his mistakes.

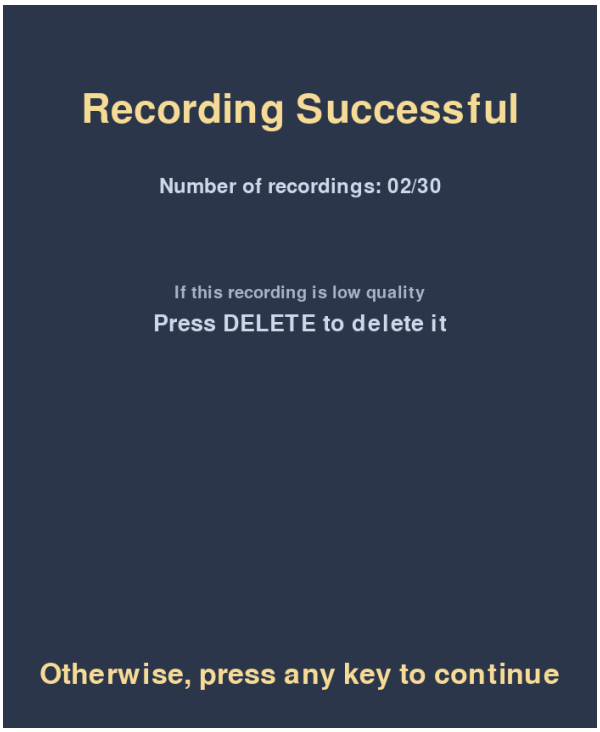
This process will repeat for each of the annotations that is desired. There are custom parameters within the code that allow for performing a random subsection of annotations for each trace. As such, we reduce the burden on the annotator for each trace, which is $(N + 1) * T$, where T is the desired time, and N the number of annotations per trace. For a 2 minute trace and three annotations that comes out to 8 minutes, reaching the limits of the annotator's attention span.

6.6 Saving Confirmation

When the annotator has finished the annotations, he is presented with a splash screen listing the number of traces that have been completed and optionally a target number. This target number is customizable and offers no functional value. It is to be used as a progress reference for the annotator, representing how much work there is left. The annotator can exceed that number if he so wishes.

The annotator is also presented with the option to "delete" the trace if he believes it wasn't of sufficient quality. Of course, the trace is already saved and won't be deleted, but a file ending in ".deleted" will be saved alongside it, allowing the experiment coordinator to include it after review.

(a) Performing an Annotation



(b) Confirming the Trace

7 Summary

The creation of this tool represents the first work to be produced during this thesis and is the pivotal step that will allow for the production of the data that will be used in the following chapters. The creation of the tool was successful and after testing it appears reliable and intuitive.

The data produced by the tool are complete and of the same format that will be used for training the agent, as the annotator plays by inputting and receiving data from the environment the exact same way an agent would. Also, by using the bk2 format of the Retro tool, each playthrough only takes up only 2 kB and the annotations, saved as JSON, only take up 10 kB per minute per annotation, without being compressed. As a result, they can be easily sent back by the annotator without needing a fast internet connection and can easily be shared with the research community. The challenges of converting those files into the actual data will be discussed in a later section.

In the following chapters, we will construct a proof-of-concept supervised learning experiment in which the annotator will associate certain actions in the game Super Mario Bros with emotions. Then, that data will be piped into an inference CNN which will try to predict the emotions of each state-action pair. By successfully completing those experiment we will have proved this tool's function end-to-end, which will allow for moving towards the creation of an agent.

Chapter 3

Dataset Creation

Following the creation of the tool, a dataset would have to be created in order to move to the process of learning. For this initial attempt, a simple dataset design was chosen, for it to be feasible to be created by one person and in a reasonable time frame. The design that was chosen was a collection of short two minute vignettes of the first world of Super Mario Bros. Those vignettes would then be first-person annotated with binary annotations that follow the PAD model (Pleasure, Arousal, Dominance) with objective annotations, to have reasonable inter-annotation agreement. In this section, we'll analyze the available options for datasets of this type and explain the reasoning behind the choices that were made, so as to open the door for future work.

1 Environment Selection

Since OpenAI Retro was chosen as the environment platform, every emulator and platform that exists for RetroArch is supported. Therefore, there's an extensive choice of available games to act as environments.

The requirements behind such an environment are the following: It should feature a high degree of freedom, as in the user should be able to choose different ways to complete an objective. It should be stimulating and entertaining, to motivate the production of varied, high quality annotations. It should have medium difficulty, since performance is a tertiary goal and it will have to be played by non-expert annotators. By considering the above, Super Mario Bros (NES) was chosen, perhaps being cliche, because of its low difficulty, entertainment value, and multiple ways to complete a level, which have an affective dimension.

2 The Reason behind Short Vignettes

The dataset consists of short, two minute vignettes of the first world of Super Mario Bros, where the annotator begins in a random level each time. The first world consists of four levels, each taking a minute or so to complete.

The length of two minutes was chosen based on the attention span of annotators. Drawing from the work of David Mehart, it's unreasonable for an annotator to remain focused for more than 10 minutes at a time. As such, by having 3 annotations and having to capture the replay at the start, each 2 minute vignette takes 8 minutes to capture, reaching that limit.

Starting randomly at the first three levels and playing for two minutes produces vignettes that contain on average a playthrough of two levels, allowing an even exploration of the dataset. Meaning that, since 54 vignettes where captured, there are $18 - 36$ playthroughs of each level.

3 Subjectivity in Emotions

The affective dimension is subjective and non-ordinal. That is, the emotional effect of an experience will be different between two people and if it's the same, it won't be directly comparable. However, there is an objective element to annotations which we can quantify using the tools below.

First versus Third Party Annotation There are two forms of annotating, first party and third party. In first party annotations, the content is annotated by the person who completed it. Whereas in third person, content is produced and then annotated by someone else. First annotations have the advantage of being more accurate because the annotator has experienced the content. Third party annotations, while not as accurate, have the advantage of being able to be completed multiple times, by different annotations. As such, they give us access to a range of analytics, such as Inter-rater agreement on exactly the same content, discussed below.

Inter-rater Agreement Inter-rater agreement is the extent to which different annotators agree about the annotations for a specific piece of content. High agreement confers that the annotations are objective and are based heavily on a ground truth. Low agreement, on the other hand, is indicative of subjective content, in which a ground truth may exist, but it's sparse.

If the inter-rater agreement is high, constructing a dataset can be done by as few as one annotator and can be useful with a modest size. When inter-rater agreement is low, however, multiple annotators are needed to remove bias and a large dataset size is required to weed through the noise, thereby revealing the ground truth in the data.

For the purposes of this thesis, as it is a first exploration, the dataset was designed to be objective, even if in the affective domain. This allowed for one first party annotator, lowered the required size of the dataset, and reduced the noise inherent in it. The action to annotation pairing was chosen to be according to the table below.

	Positive	Negative
Pleasure	Collecting Coins, Winning	Losing a life or a powerup
Arousal	Interacting with enemies, dangerous jumps	Staying still, getting stuck
Dominance	Beating Enemies	Losing a life, running away from enemies

Table 3.1: The objective definitions for emotions in the dataset

4 Overview of Annotation Types

As described in the tool section, there were multiple annotation types that were built into it. Those were RankTrace, BinaryTrace, and GTrace.

RankTrace RankTrace allows for the creation of unbounded annotations. As such, they are not directly comparable, but by taking the derivative, which shows the affect change, they can be compared to each other and used to train a model. They produce the most detailed annotations.

GTrace GTrace is an ordinal annotation tool. The annotator rates his feeling from $-100\,\%$ to $100\,\%$. As a result, the annotations are directly comparable, even from different annotators, but some detail is lost.

BinaryTrace BinaryTrace is the least detailed type, allowing the annotator to choose between a Negative and Positive emotion at any given time. This form was chosen for the dataset, since Super Mario Bros is fast paced and having multiple affect levels did not amount to much. In addition, the form of BinaryTrace $(-1, 0, 1)$ made it so that it could be piped into a Deep Q Learning Network directly as a reward, without any pre-processing.

5 Emotional Models

One of the core challenges of affective computing is quantifying emotion, which is fundamentally subjective. A solution to this problem is mapping independent (or orthogonal) elements of emotions to axis, to form a basis with which to classify emotions. A basis is a mathematical concept which describes a set of

orthogonal vectors whose linear combination can form any other vector. In the same manner, an emotional basis can describe an arbitrary emotion by varying the amount of each of its elements. In psychology, this concept is named Dimensional emotion theory [16], and usually involves Valence and Arousal [17].

5.1 Arousal

One of the core qualities of an emotion is the intensity or arousal with which it is expressed. Using this, we can form an one dimensional basis that describes an emotion as a function of its arousal. Happiness is high arousal, whereas Boredom is low arousal. Unsurprisingly, in works that study Affective Computing, classifying the intensity a participant feels while interacting with a piece of software or a game is the first target attempted [8].

5.2 Valence and Arousal

An expansion of the concept of arousal appends a second axis, which describes the Valence (or Pleasure) of an emotion. Using just arousal isn't enough to differentiate between happiness and anger, for example, but by adding Valence it becomes possible. It is commonly used in affective computing, because it can describe a breadth of emotion effectively.

5.3 PAD: Pleasure, Arousal, Dominance

Lastly, we can also append dominance to the above model, allowing differentiation between fear and anger. The PAD model [14] is a more physiological view of emotion, initially designed for environmental psychology, but has been successfully used for the creation of animated characters [1]. It is therefore well suited for the construction of an in-game agent, and that was the reason it was chosen for this dataset.

Parts of this section have been referenced from [12] and [3]

6 Dataset Contents

After creating the dataset, its contents were analyzed into summaries. Below is the composition of the dataset, in percentages, for actions and annotations. As can be noted, the dataset test and train splits are balanced, with each having similar percentages.

The annotation has a column for each of the annotations, with the first being the primary objective (going right). The numbers mean the following: -1 for a negative annotation, 1 for a positive annotation and 0 for a lack of annotation. The primary objective (going right) doesn't give negative rewards. The dataset is imbalanced in nature, with most annotation types appearing in less than 10 % of the frames, so the majority class (being the absence of an annotation) makes up 93 % of the dataset.

		Reward	Pleasure	Arousal	Dominance	Overall
Train	-1	0.00	1.42	0.26	1.36	1.01
	0	48.82	94.60	87.91	95.78	92.76
	1	51.18	3.98	11.82	2.87	6.22
Test	-1	0.00	1.67	0.13	1.75	1.19
	0	48.17	94.54	88.76	95.21	92.83
	1	51.83	3.79	11.11	3.04	5.98

Table 3.2: Annotation constitution of the dataset.

Actions were simplified into those shown in the table. The ones not mentioned are grouped into the "Unkn" tag and constitute only 1.5 % of the dataset. The largest move, by percentage, is going right at 40 %, followed by doing nothing at 30 %, and lastly jumping right at 15 %.

	Train	Test
Unkn	1.30	1.51
None	31.73	32.50
B	0.22	0.26
L	8.68	7.75
R	39.61	39.51
A	1.27	1.61
L+A	1.37	1.10
L+B	0.03	0.05
L+AB	0.00	0.00
R+A	15.65	15.59
R+B	0.14	0.11
R+AB	0.00	0.00

Table 3.3: Action constitution of the dataset.

To supplement the chapter on Affect, the following table was created. It lists, by column, the spread of actions that was performed depending on the annotation on the train set. The first column acts as a control and shows the overall percentage, while the last one shows the spread for when there were no annotations.

	Control	Pleasure	Arousal	Dominance	Absence
Unkn	1.30	1.73	1.56	1.52	1.24
None	31.73	30.82	12.65	18.78	34.40
B	0.22	0.00	0.92	2.34	0.11
L	8.68	13.42	16.53	28.35	7.24
R	39.61	27.64	33.50	34.73	41.06
A	1.27	4.76	1.39	1.17	1.04
L+A	1.37	5.35	2.53	2.05	1.05
L+B	0.03	0.04	0.15	0.35	0.01
L+AB	0.00	0.00	0.00	0.00	0.00
R+A	15.65	16.24	30.44	9.89	13.75
R+B	0.14	0.00	0.33	0.82	0.11
R+AB	0.00	0.00	0.00	0.00	0.00

Table 3.4: Action constitution, by annotation, of the dataset.

7 Summary

In sum, for the creation of this dataset, 54 2 minute vignettes of the first world of Super Mario Bros (NES) were created. Each of those vignettes started from a random level, and due to its length, in average, allowed the completion of 2 levels. At the end, each of those vignettes were first-party annotated by one annotator according to the PAD model. In this first attempt, objective annotations were deployed to instill a ground truth to the dataset and lower the noise due to inter-annotation disagreement.

Chapter 4

Inference

Having the tool and dataset in hand, the problem shifts into creating a pipeline to process this data. Questions arise concerning performance, model type, hyperparameter tuning, and data quality. Surely, those are questions that should be answered before delving into Reinforcement Learning. As such, for a first study into this data, affective inference and imitation learning were chosen. Both are supervised learning methods, which naturally converge faster than reinforcement learning, allowing for more convenient hyperparameter tuning and model selection.

1 Compilation

After initial experiments, it became apparent that compiling the dataset into batches, with the current hyperparameters, would be much faster than simulating the environment with OpenAI Retro against the whole dataset every epoch. Once the compilation process is explained, we'll move into describing the learning process.

1.1 Initial File Format

The dataset is composed of 54 two minute vignettes, or episodes. Each episode is made up of two files, an annotation file and a Retro Movie or backup file. The backup file contains the starting state and following it a series of recorded actions. Each action is a binary vector of 9 buttons, which can be either 0 or 1. The annotation file is made up of 3 vectors, one for each emotion, which contain an annotation value for each action (which can be $-1, 0, 1$).

To replay an episode, the initial state is fed into a Retro environment. Then, the recorded actions are piped into the environment, producing sequential obs, rew, done, info environment output pairs. The outputs, when paired with the annotations by index, represent the full information of the episode for that frame. That information can then be used for learning and is the same as the one an agent would have when playing a level.

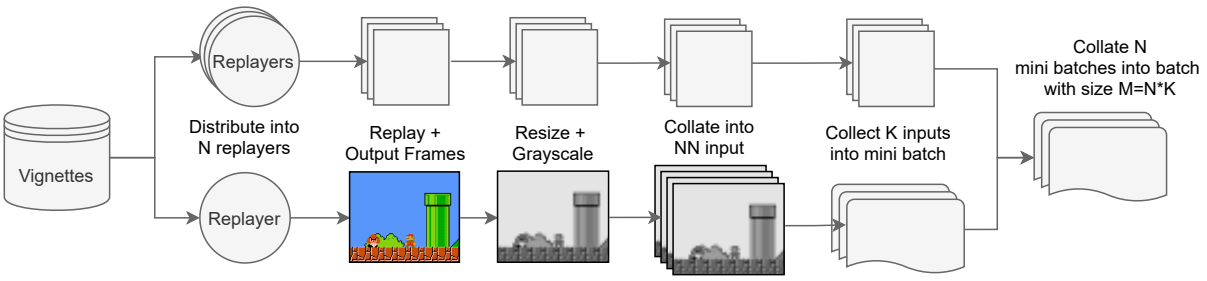


Figure 4.1: Compilation process

1.2 Hyperparameters

The goal of the compilation process is to take the initial backup files and convert them into batch files that can then be directly fed into the model, without any processing. Since parts of processing are hyperparameter specific (such as input frame resolution and number), the hyperparameters will be burned into the output files. As a result, each configuration set will have its own compiled data.

1.3 Parallel Processing

Batches should contain content from multiple episodes. If they don't, the similar frames contained in a batch will produce an effect akin to stochastic gradient descent even though batches are used. A solution to this could be compiling the dataset into a memory buffer and shuffling the data. However, this is a messy and slow solution. Moreover, OpenAI Retro is single threaded and a singleton (at most one per process).

Therefore, to utilize multiple cores and to shuffle batches to an extent, multiple Retro environments (which we'll refer to as Replayers) will run at the same time using the python subprocessing module. The frames from each Replayer will then be combined into a mini batch, and a set of mini batches from each Replayer will be formed into a batch. The mini-batches were made to be complete, insofar as processing is concerned, because any processing that runs in the Replayer is on a separate thread, making it hyper-threaded.

1.4 Frame Processing

Once the Replayers start, they are allocated a portion of episodes and begin to run them. With every step, one observation is produced. The observation is then resized and converted to grayscale. Resizing is done using OpenCV to a resolution, after hyperparameter tuning, that is 50×50 for CNN models and 18×12 for Fully Connected models. To convert to grayscale, picking the green channel or using an OpenCV grayscale function were both found satisfactory.

1.5 Collation

Following earlier work [4], the models utilize both frame skipping and multiple frame inputs. That is, only one every N frames is recorded, and the model is fed the last M recorded frames. As such, the Replayer skips $N-1$ frames and then saves a frame to a M sized queue. If the queue is full, those M frames, along with rewards and annotations, are collated into an input and output pair each time there is a new recorded frame. When enough pairs have been collected, they are formed into a mini batch and are sent to the main process.

Since frame skipping is implemented, there comes a question of which annotation and action is kept as an input. The action that is kept is the one that would have been taken right after the frame that was recorded. Whereas, for the annotations and rewards of the last N frames, the $N-1$ that were skipped and the N^{th} that wasn't are summed and normalized so that they are in the space $[-1, 1]$. For supervised learning, when other than 0 they are assumed to be -1, 1, depending on the sign, and for DQN they are used as is, since DQN is traditionally performed using decimals within $[-1, 1]$.

1.6 Compression

When a number of mini-batches equal to the Replayers is collected, it is formed into a batch and saved using LZ4 compression, with the blosc library. This results in files that are $10-50$ times smaller (3 MB per batch, 200 batches) that are loaded adequately fast so as to make learning GPU bottlenecked.

2 Learning

2.1 Actions

The actions are in the format of binary vectors of 9 boolean numbers. Those can't be directly used to learn on. As such, 12 boolean combinations that constitute the majority of those used were chosen as the model actions, and the other ones were ignored. As the Replayer runs, the boolean actions are converted into the index of the combination they matched. When running the model, a softmax layer is used, indicating the index of the action that the model would have chosen.

2.2 Annotations

Annotations can have fractional values from $[-1, 1]$. If an annotation is zero, it gets classified as no annotation in that frame. If it's lower than zero, it is assumed to be Negative and if it's greater than zero it is assumed to be Positive. Those 3 states are used with a softmax layer, so that the model predicts whether an annotation would have been chosen at that moment, and whether it would have been positive or negative. Since there are 3 annotation types (Pleasure, Arousal, and Dominance), 3 softmax layers of size 3 are used.

3 Model Selection

In this section, we'll talk about the types of networks that were tried and the hyperparameters inherent in the learning process.

3.1 Fully Connected Network

To form a baseline, a number of fully connected networks was tried. The fully connected networks varied in input size, frame number, layer size, and layer number. At the end, the most complex fully connected network tested, which performed better than its peers, did not compare favorably to a CNN, so they were abandoned in the following chapters.

3.2 Convolutional Neural Network

Moving on from Fully Connected networks, a large number of CNNs was tried. Here, performance also became an issue. Feeding the raw resolution RGB image is over $10-20$ times slower than using a smaller compressed image. Therefore, the complexity of the CNN became a balance of performance to accuracy. In the end, it was found that four 50×50 grayscale frames, along with a modest CNN, provided most of the accuracy of the full resolution large CNN, so this model configuration was used from then on until the end of the thesis.

3.3 Hyperparameters

For Inference Learning, the main hyperparameters of concern were those that affected performance and model architecture.

In both types of networks, their input shape had to be determined, concerning resolution and how many consecutive frames it would include. Furthermore, the layer number and the size of each one had to be determined, as well as learning rate and L2 decay. For CNNs, the convolutional layer number and size was also important.

Performance wise, batch size was important, as it determined VRAM use and performance. A suitable number is large enough so as to saturate the GPU performance without being too large to fit in VRAM or to pipeline by PyTorch.

4 Results

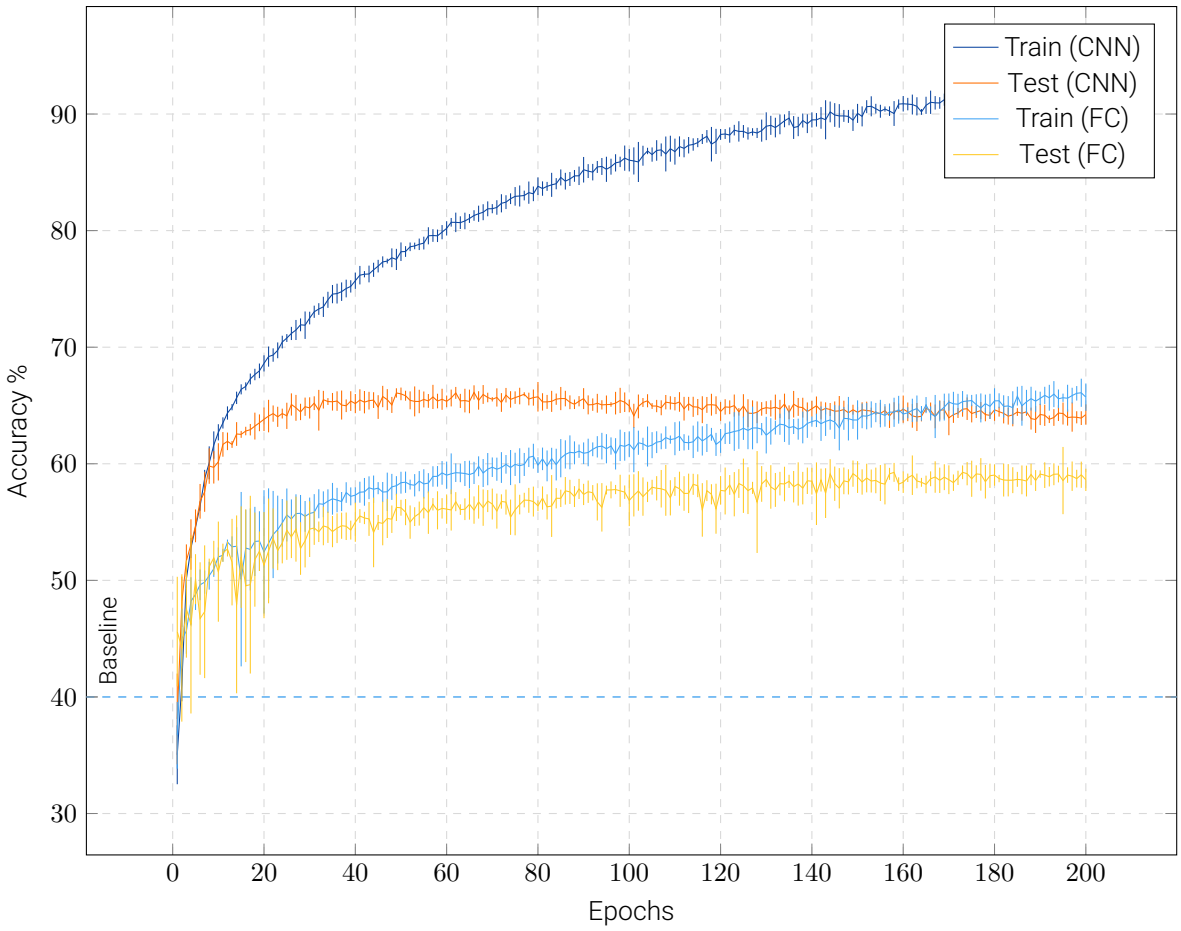
The dataset was split into train and test sets, and the best CNN, FC architectures were ran 5 times. The graphs following are the mean of those 5 runs, with a confidence interval of 95 %. The confusion matrices were the best ones that appeared in those 5 runs (early stopping). Their rows sum to 100 %. Both action and annotation models feature a variance problem, due to the small dataset, and slight overfitting (performance drops after 50 epochs).

4.1 Imitation Learning

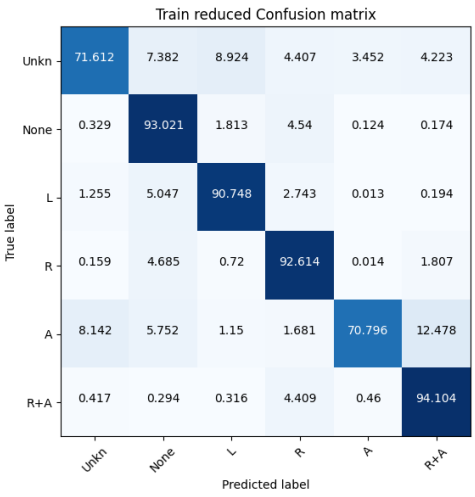
Imitation learning performance is satisfactory. The CNN manages to predict 65 % of actions on the test set and 90 % on the train set, with great reliability. It has a small confidence interval. On the other hand, the fully connected network isn't complex enough to model the problem and has inconsistent results between executions.

The model was trained for predicting 12 different button combinations and the rest were delegated to the first row, named Unknown. Since some of those 12 combinations are also rare, they were merged into the Unknown row in post-processing. Finally, the confusion matrices feature that reduced set of actions, which constitute 97 % of the actions in the dataset. We see that the model has trouble predicting jumps (A), confusing them for going right (R) and jumping right (R+A), as well as going left (L), mispredicting it to staying still and going right (R).

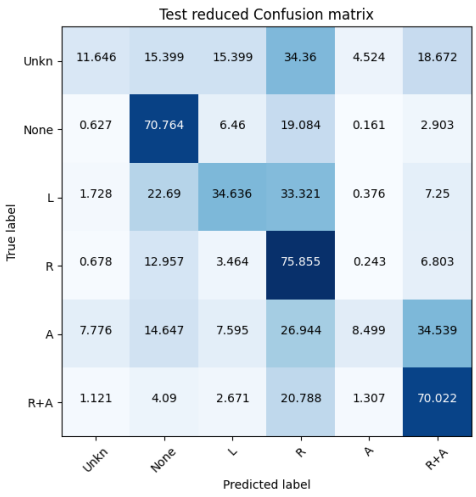
The baseline we pick for this work is defined through the majority class. According to the contents of the dataset, shown in Table 3.3, the majority class is going right, with a probability of 40 %.



(a) Accuracy Graph of Action Prediction, mean of 5 runs



(b) Best Train Confusion Matrix (CNN)



(c) Best Test Confusion Matrix (CNN), with early stopping

Figure 4.2: Action Accuracy Results

4.2 Affective Learning

Since annotations are relatively rare (appear around 10 % of the time), overall accuracy isn't all that helpful. As such, it is combined with another type of graph, which counts the percent of the time in which there was an annotation and the model predicted it correctly ($H\% = \frac{cm_{11}+cm_{33}}{\sum cm_{1i}+cm_{3i}}$).

Here, we see the model fits correctly to the train set, but it fails to do so in a large degree on the test set, due to the dataset size. After epoch 50, overfitting starts to occur. If we consider the fully connected network as a baseline, the CNN performs much better, predicting correctly 25 % of annotations. However, this causes a drop in overall accuracy. This is because the model overpredicts annotations in places where they don't happen, or predicts the wrong annotation in places they do. This is acceptable, up to a degree. For this model, for each one annotation predicted correctly in the dataset, another one is predicted where there is none.

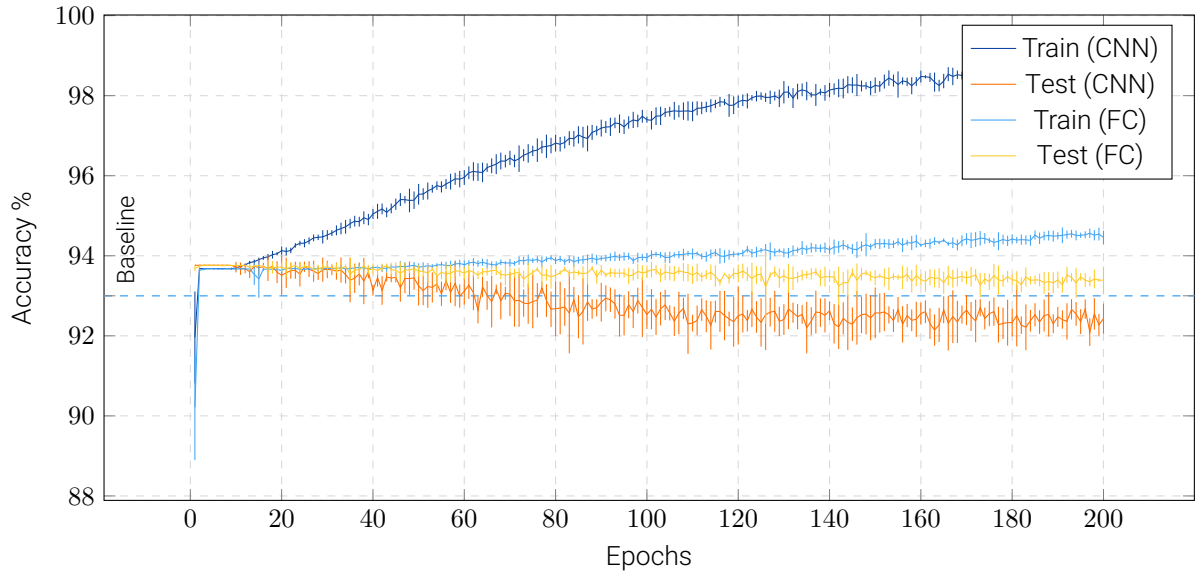
To combat the low prediction of annotations, due to the imbalance inherent in the dataset, class weights were tried, giving a higher loss to wrong annotations. This increased the prediction accuracy. However, it also increased false positives from a ratio of 1 wrong annotation to 1 correct, to 5 or more wrong to 1 correct. While this may be acceptable for a credit card company and fraudulent transactions (dealing with 5 false positives for each fraudulent transaction), it is not for this application. The dataset may be imbalanced in the sense that annotations appear in a minority of the time, but it is balanced when it comes to the real distribution of annotations. Meaning, data collection isn't skewed in some way. So, by influencing class weights we can trade higher false positives for better prediction accuracy, but not increase overall accuracy, as we could if data collection was skewed.

The baseline for each of the graphs was set by assuming a model that chooses the majority class for each of the annotations, which is zero. The percentage of that class can be found in Table 3.2. Since the output of the baseline model is always zero, it doesn't predict any annotations, so it is marked with 0 % on the second graphs.

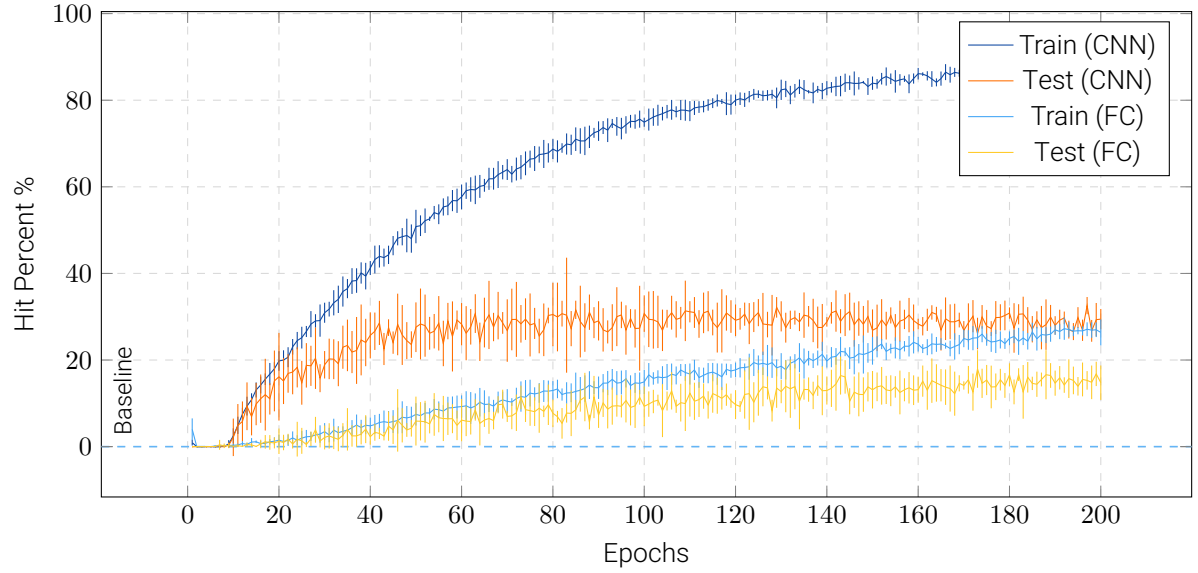
4.2.0 Overall Performance

Overall performance is calculated by merging the three different annotation type data into a new type, and then performing the same calculations as they were on each type, as opposed to just taking the mean. Therefore, annotation types contribute proportionally according to their occurrence.

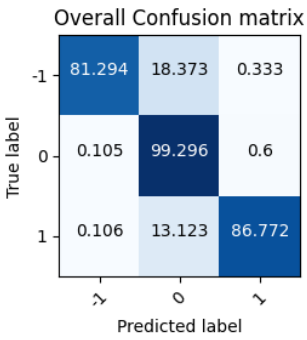
The model manages to predict 45 % of positive annotations correctly, and 10 % of negative annotations. Negative annotations were much rarer and not much focus was paid on them in the following chapters, so their performance isn't important. For the size of the dataset, 45 % for positive annotations is acceptable.



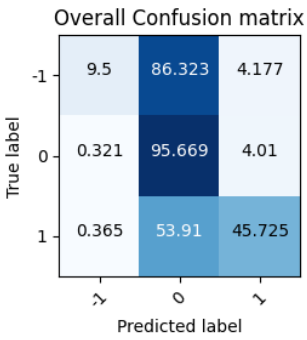
(a) Overall Accuracy



(b) Percent of Correct Non-Zero Annotations



(c) Best Train Confusion Matrix (CNN)



(d) Best Test Confusion Matrix (CNN)

Figure 4.3: Overall Annotation Accuracy Results

4.2.0 Type Specific Performance

The objective truth behind which the annotations were created is explained at Table 3.1

Pleasure The performance on pleasure is similar to overall performance. Negative annotations mainly appear when losing, which is rare, so the middling performance is expected.

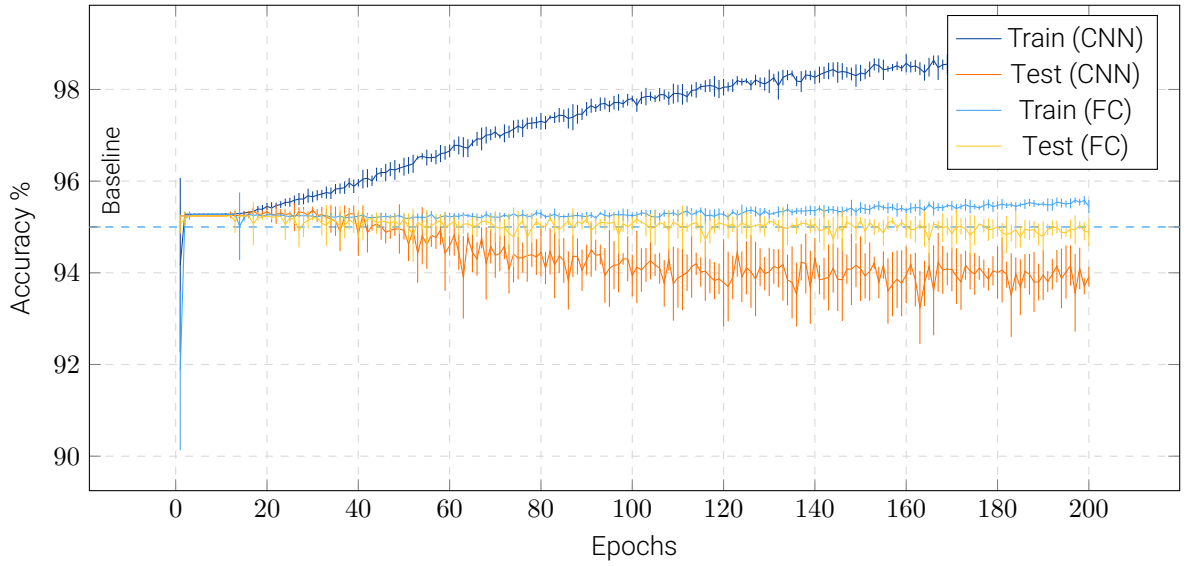
Arousal Arousal is the most common annotation type, as it would make sense for a fast paced platformer. Negative annotations for this type are almost non-existent, and in hindsight there wasn't enough information provided by them to justify including them

Dominance Dominance is a noisy annotation type, as Mario fighting enemies and winning or losing requires a longer history than the one provided to the model for it to be objective. In addition, the ways of approaching enemies are much more varied, and so require a larger amount of data.

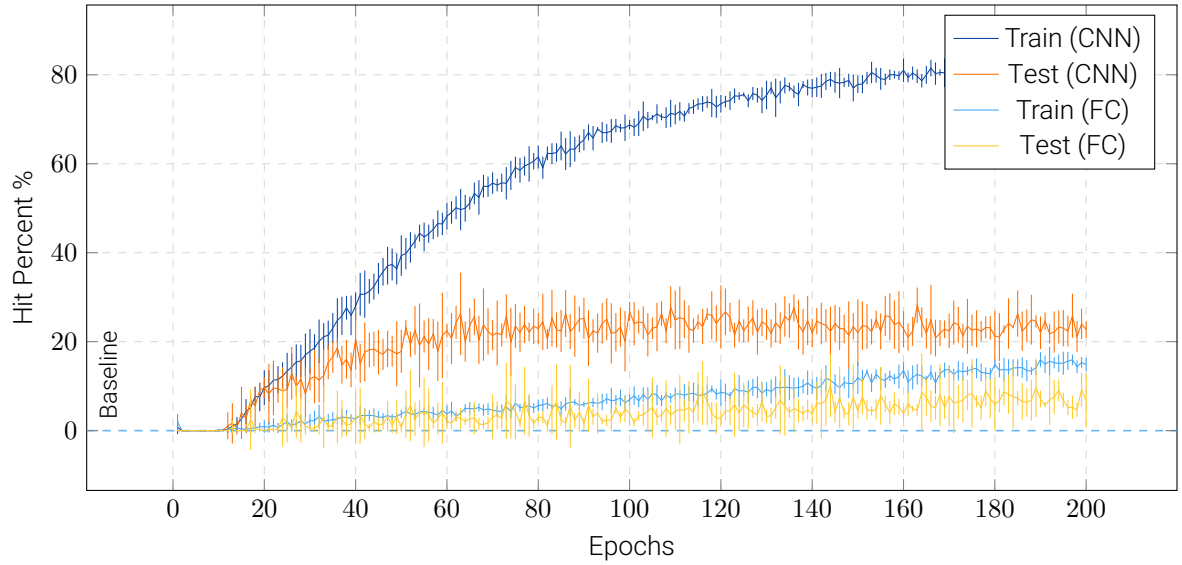
5 Summary

This experiment constitutes a first look and application of the tool and dataset created in the previous chapters. Imitation and affective learning were implemented. Imitation learning had adequate accuracy, while affective learning had low accuracy due to dataset size. Since the purpose of the dataset is to train an agent, the results are acceptable.

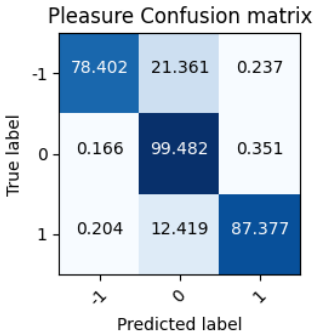
This experiment sets the foundation for the following chapters. The code base, the compilation process, the hyperparameters, and the model architecture will be carried over to the next chapters without extensive modifications.



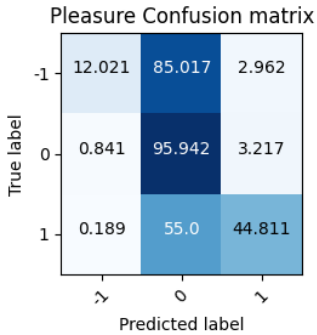
(a) Overall Accuracy



(b) Percent of Correct Non-Zero Annotations

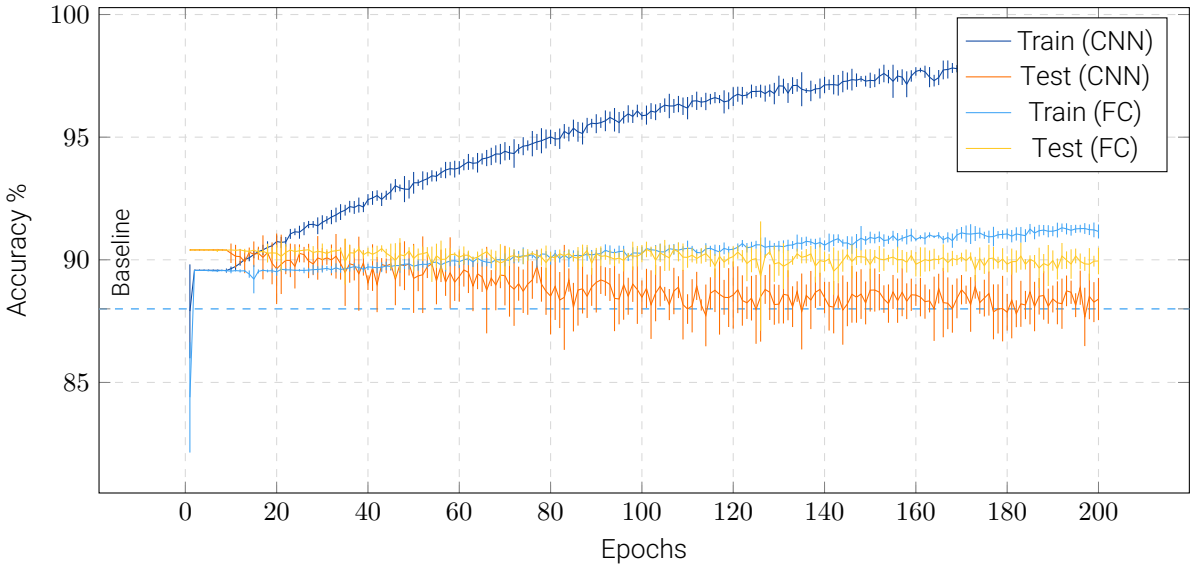


(c) Best Train Confusion Matrix (CNN)

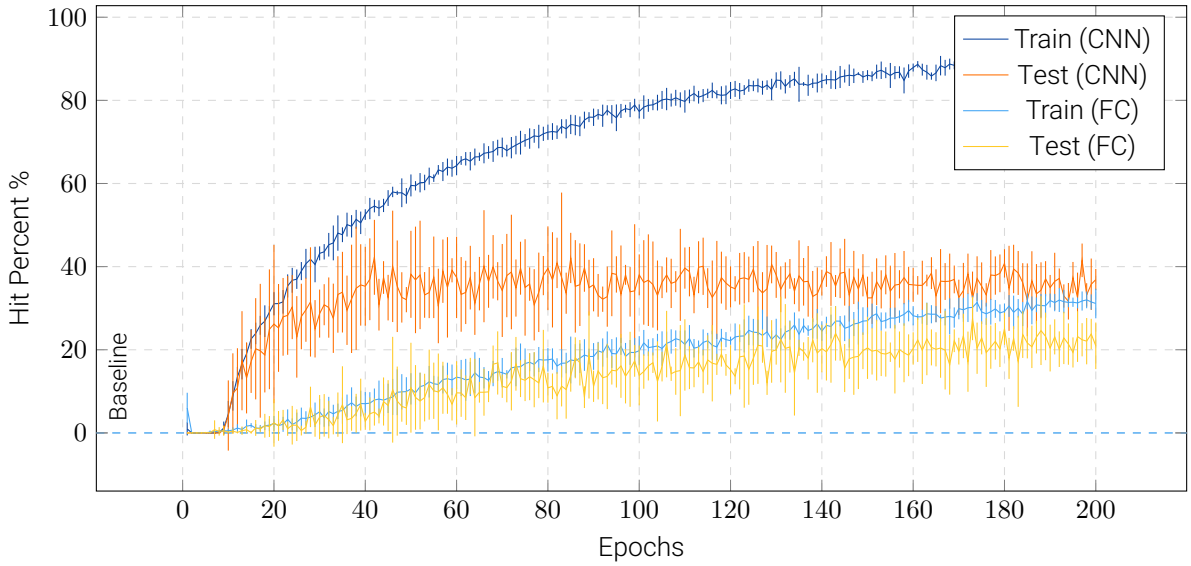


(d) Best Test Confusion Matrix (CNN)

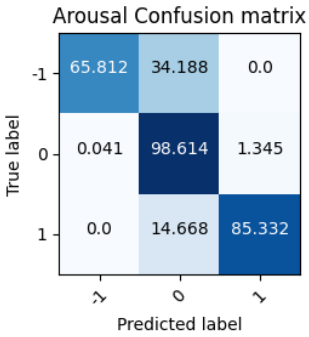
Figure 4.4: Pleasure Annotation Accuracy Results



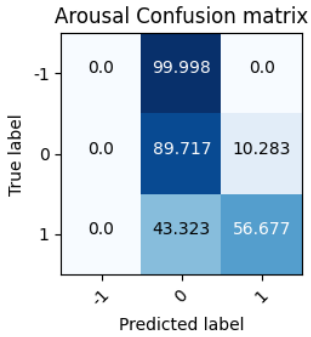
(a) Overall Accuracy



(b) Percent of Correct Non-Zero Annotations

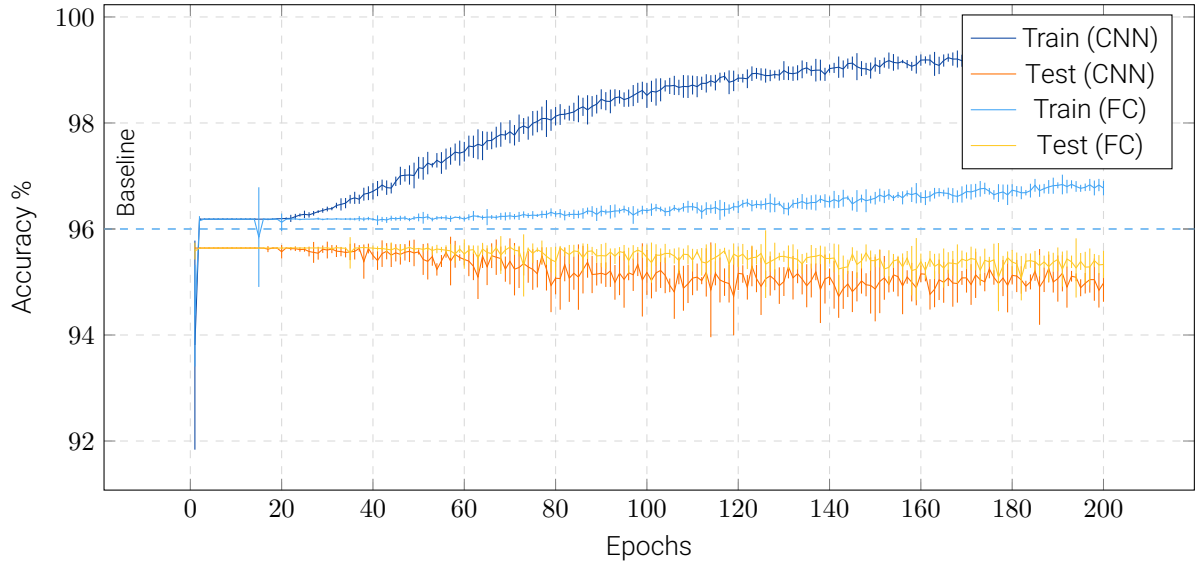


(c) Best Train Confusion Matrix (CNN)

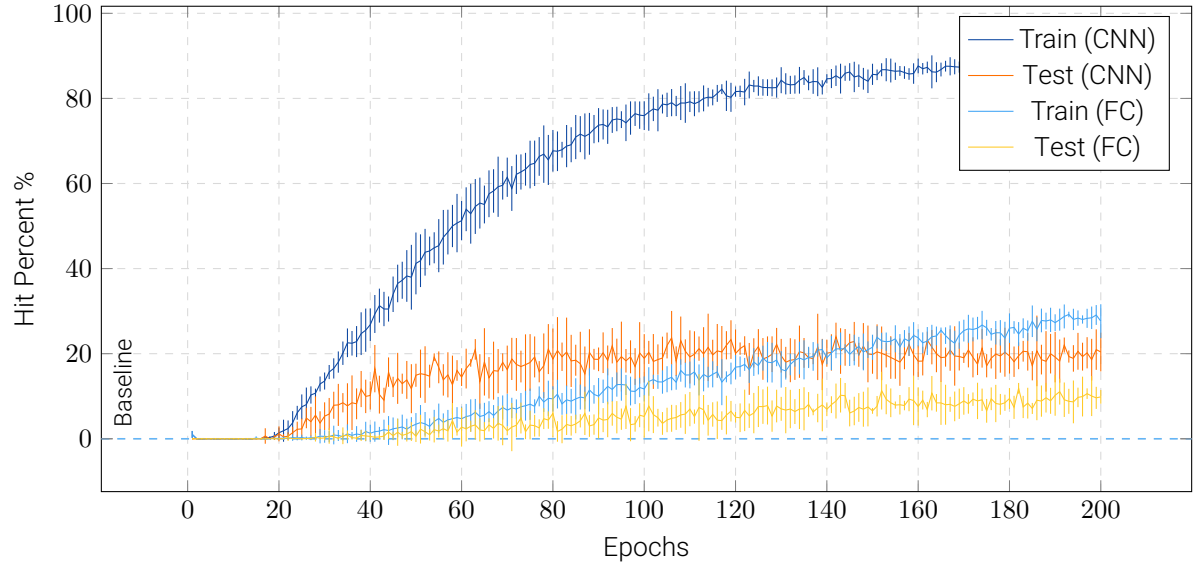


(d) Best Test Confusion Matrix (CNN)

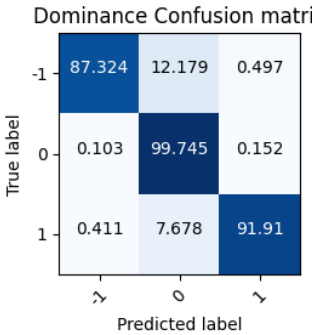
Figure 4.5: Arousal Annotation Accuracy Results



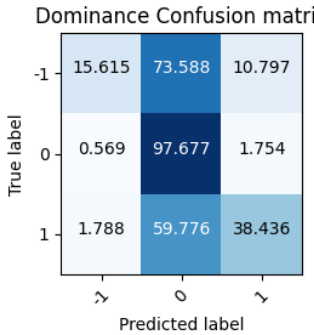
(a) Overall Accuracy



(b) Percent of Correct Non-Zero Annotations



(c) Best Train Confusion Matrix (CNN)



(d) Best Test Confusion Matrix (CNN)

Figure 4.6: Dominance Annotation Accuracy Results

Chapter 5

Reinforcement Learning

At this stage in the Thesis, we have a dataset of playthroughs and a pipeline, including a model, with which to process them and perform supervised learning. The next step is to use this work to train an agent using Reinforcement Learning to play the game, focusing only on the primary objective, which is going right. This chapter starts with an overview of reinforcement learning, moves on to an explanation of the algorithm chosen (DQfD) and the reasons behind it, and finishes by showcasing the results.

1 Overview

Reinforcement learning is a subsection of Machine Learning that focuses on training an agent to complete a task. It is comprised by mathematical theorems that describe the world, and algorithms that utilize them to learn to behave in an environment. In the last decade, driven by increased processing power, Neural Networks have been incorporated into the algorithms, acting as function approximators.

For example, if an environment can be described as a Markov Decision process (which is a mathematical model), Q Learning can be used to create a policy (the best actions to take given a state to complete an objective). Q Learning discretizes states by grouping similar states into a table and scoring them. The grouping function is pre-decided and could be, if the state is a approximated to be a game frame, resizing the frame into a smaller one and limiting the color depth. The Nintendo Entertainment System features $256 \times 240 \times 2^{15}$ distinct frames at full resolution, when including color. A grouping function could resize it into $12 \times 10 \times 5$, giving a total of $12 * 10 * 5 = 600$ states, from 2 billion ones. Deep Q learning extends Q learning by using a neural network in the place of the table, such that the initial input state can be used as is to produce an approximated score.

1.1 Types of Algorithms

Reinforcement learning algorithms can be grouped into three categories, depending on how they receive experience. Those are On-Policy, Off-Policy, and Offline. The fundamental difference between them is that On-Policy algorithms associate the states they receive when they explore with the policy they had at that moment, whereas Off-Policy algorithms don't. As a result, On-Policy algorithms can only learn from *their* experience, where Off-Policy algorithms can use their own experience, experience provided by demonstrators, or even other algorithms. Offline algorithms are a special type of Off-Policy algorithm, where the algorithm doesn't have access to the environment and has to learn from stored experience alone. Offline algorithms are more akin to supervised learning.

1.2 Considerations

The goal of this thesis is to produce an agent that completes an objective while considering the affective dimension, which is the affective annotations that are collected in the dataset. We'll define the primary objective as winning the level and the secondary objective as completing a high emotional target. For the primary objective, we have access to a reward function (going right) that is in the dataset and that the

agent can produce more of using the environment. For the secondary objective, the affective annotations in the dataset can act as a reward function, meaning the agent will have access to a pool of experience with affective rewards. However, replaying the environment will not produce additional annotations.

For the primary objective, both On-Policy algorithms and Off-Policy algorithms can be used. Given that the focus of this thesis isn't performance, we should use the dataset, which features playthroughs of an expert, to teach the algorithm, if it's convenient. However, On-Policy algorithms wouldn't be able to use that, because the states won't have an associated policy. For the same reason, it will be harder to merge the primary objective with the secondary to produce a composite agent. As such, Off-Policy algorithms would be preferable. Since we have access to the environment, we won't consider Offline algorithms. The secondary objective is strictly an Offline problem, since additional data can no longer be produced.

1.3 Deep Q Learning from Demonstrations

By taking into the account the above considerations, Deep Q Learning from Demonstrations (DQfD) was chosen. DQfD is an algorithm by Todd Hester et al. [6] that's between offline learning and Off-Policy learning. It is an extension of Deep Q Learning that features improvements from recent papers and supports learning in tandem from demonstrations and experience. The policy is created by demonstrations and then fine tuned by experience. As such, the algorithm is purely exploitative, with an epsilon of 0.01. DQfD was chosen because it can be used both for the primary objective and the secondary objective. After training, the Q functions of the objective policies can be summed in a ratio to form a composite policy.

2 Methodology

2.1 Carryover from Inference

In the previous chapter, we created a pipeline and a compilation process with which to ingest play traces and perform supervised learning. It featured great performance, so the majority of the codebase will be moved forward.

The compilation process that was created will be carried over in two places. It will be used as is to create a file based replay buffer for demonstrations, that will be run once and shared between different configurations. For experience, parts of it will be carried into a new module, named Explorer, which will produce batches in parallel in a similar fashion. For action selection, it will receive actions from the training model and feature a configurable epsilon policy. In addition, it will record statistics about the played episodes, such as the mean reward, cumulative reward, and episode length. Those statistics will then be graphed into histograms, mean curves, and highest of epoch curves.

For model choice, in the inference problem, a CNN was found to be the best performer. The same model structure will be carried over, since the input format will remain the same. The last layer will be replaced by N Q functions, each of which will be made up of M linear activations. Where N will be one in Reinforcement Learning and three in Affective Learning, one for each emotion. M will be 12, one for each action recorded from the dataset. In total, the last layer will feature $N * M$ activations.

2.2 Deep Q Learning Overview

DQfD is a variation of Deep Q Learning (DQN) [11], which is an extension of Q Learning, an algorithm that creates policies for an environment described by a Markov Decision Process (MDP).

A Markov Decision Process is defined by a set of 5 parameters (S, A, R, T, γ), which consists of a collection of states S, a set of actions A, a reward function $R(s, a)$, a transition function $T(s, a, s') = P(s'|s, a)$, and a discount factor γ. The agent is given a state $s \in S$, to which he responds with an action $\alpha \in A$, and receives reward $r = R(s, \alpha)$ and next state $s' = P(s'|s, a)$. Future rewards are discounted by being multiplied by γ. A policy π is a mapping of states to actions, in which the discounted reward is maximized. We define function $Q^{\pi}(s, \alpha)$ as the estimate of the future discounted rewards that will be accrued by taking

action α from state s, with $Q^\star(s,\alpha)$ being the optimal policy determined by solving the Bellman Equation:

$$Q^\star(s,a) = E\left[R(s,\alpha) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^\star(s',\alpha')\right]$$

The optimal policy π is then $\pi(s) = \arg\max_{\alpha \in A} Q^\star(s',\alpha')$.

Deep Q learning uses a neural network to approximate the $Q(s,a)$ function. The neural network outputs a set of values $\{Q(s,\alpha;\theta)|\alpha \in A\}$ for a given state s, where $Q(s,\alpha;\theta)$ is the expected reward for taking action α and θ are the current network weights. When training, a copy of the network, named the target, is used to provide an approximation of the next state's Q for calculating the loss. The target network is updated infrequently to stabilize the Q function. Without it, the Q function would increase every iteration and ultimately destabilize into a worthless policy. The double DQN loss [5], which incorporates target, is defined as:

$$J_{DQ}(Q) = \left[Q(s_{t+1}, \alpha_{t+1}^{max};\theta) - \left(R(s,\alpha) + \gamma Q(s_{t+1}, \alpha_{t+1}^{max};\theta')\right)\right]$$

where θ are the main network weights and θ' are the target network weights, synced every τ steps. To prevent biased state selection during training, due to the current policy, mini-batches of experience are sampled randomly when training using a replay (ring) buffer, which stores the experience that is accumulated from the environment. After reaching its maximum size, the replay buffer begins to expire old data.

2.3 Deep Q Learning from Demonstrations Overview

DQfD incorporates a pre-training phase, in which the network is fed data solely from demonstration. From then on, it starts to accumulate experience from the environment in its replay buffer D^{replay}, and excludes demonstration data from expiration.

The demonstration data incorporates a margin classification loss [15], which penalizes the network for producing greater or equal Q values for actions not chosen by the demonstrator:

$$J_E(Q) = \max_{\alpha \in A}\left[Q(s,\alpha) + l(\alpha_E,\alpha)\right] - Q(s,\alpha_E)$$

where α_E is the action of the expert at that state, and l is a penalizing function which is 0 when $\alpha = \alpha_E$ and a positive value when not, causing values of $Q(s,\alpha)$ to be at least l lower than $Q(s,\alpha_E)$.

The n-step return loss, with $n = 10$ is added to both experience and demonstration data, leading to faster learning, similar to A3C [10]:

$$Q_n(s_t,\alpha) = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_{\alpha} Q(s_{t+n},\alpha_{t+n})$$
$$J_n = \left[Q(s,\alpha) - Q_n(s_t,\alpha)\right]^2$$

Lastly, an L2 regularization loss is applied to the weights to prevent over-fitting. The overall loss is a weighted sum of the above losses, with λ_2 being 0 for non-demonstration data:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q)$$

The original algorithm also features prioritized experience replay [18], increasing the probability of selecting transitions that had a high loss previously.

2.4 Improvements

Traditional Q learning is typically composed of a model pair, a ring in-memory replay buffer, and one environment. The loss is calculated using tuples of (state, next_state, action, reward), excluding n-step. A learning step is performed by first forming one or more mini-batches of tuples, learning from them, and then having the agent play an episode in the environment, which gets stored in the replay buffer. Each of those steps introduces performance limitations into the process.

Specifically, since the replay buffer is stored in memory, it is constrained by the available RAM. Let's assume that the model receives the 4 last observations to produce a policy. A simple solution would be

to store the complete (state, next_state, action, reward) tuple in RAM for each training frame. However, that would require 5 or 8 frames of storage, 4 for state and 1 or 4 for next_state, since state and next_state have 3 common frames. To reduce RAM use, allowing for a larger buffer, we would store consecutive (obs, action, rew) tuples into the buffer instead, and to form the (state, next_state, action, reward) tuple we would merge 5 consecutive observations into state and next_state. That would produce a replay buffer that is 5–8 times smaller in size, or 9–12 times when including the n-step observation. At 1 frame stored per step (consequtive format), if each frame is $50 \times 50 \times 3 = 7.5$ kB, a 10 GB in-memory buffer would allow for 18 hours of memories, whereas for the duplicate format it would allow for 2 hours of memories (insufficient).

Unsurprisingly, the consecutive design creates performance limitations. Since the buffer stores a consecutive line of observations, in order to construct the loss tuple the CPU has to perform memory copy operations to unpack the observations into the 3 required states. This introduces a CPU bottleneck. Furthermore, since the buffer is consecutive, it isn't shuffled and can not be shuffled (observations will seize to be consecutive). By selecting states in order, the batches formed would have slices that are too similar and produce a result similar to stochastic decent. Therefore, the indices of states have to be randomly chosen and selected for every batch. The above result in a hard CPU limit in training speed, prompting the creation of band-aid fixes, such as prioritized experience replay, where the priority is based on loss and not score, so that the model can learn faster, not smarter.

The consecutive design also limits the number of environments that can be used to 1. The buffer is consecutive so it can not be fed the output of multiple environments. If the environment is single-threaded, this results in a slowdown proportional to the number of cores in the workstation. A band-aid solution could be to have N replay buffers, one for each environment, so that N environments can be utilized.

The solution chosen for this thesis was to create a batch file based ring replay buffer and to use multiple environments to form each batch. Each batch is represented by a file, which contains preformed (state, next_state, action, reward) tuples. The file is passed through a high performance compressor that can unload it into memory faster than a traditional memory copy operation. The compressor is efficient enough such that the overhead of storing the same frame 10–12 times is negligible, since it compresses the tuples $100\times$, a magnitude of efficiency higher than required.

To mitigate the problem of stochastic descent, the input of N environments (8 were used) is used to form each batch. Since each environment contains a different state, there's N times the variance in the batches, compared to a single environment. The use of multiple environments also utilizes multiple cores effectively.

In sum, the use of preformed compressed batches ensures that the gradient descent completely utilizes GPU performance, without being bottlenecked by CPU, and environment exploration completely utilizes CPU performance, by being parallelized. A summary of the modified algorithm is shown below.

Algorithm 1: Batched Deep Q Learning from Demonstrations

Input: E_{demon}: demonstration episodes, r_{exp}: ratio of experience batches to demonstration batches, e_{total}: number of total epochs, $e_{pretrain}$: pretrain initial epochs, $n_{batches}$: number of batches per epoch, n_{save}: number of batches to create per epoch, $n_{episodes}$: minimum number of episodes per epoch, ρ: model and process hyperparameters, τ: target update frequency

Initialize D_{demon} buffer by compiling E_{demon}, taking into account ρ;
for $e \leftarrow 1$ **to** e_{total} **do**
  $B \leftarrow$ select $n_{batches}$ from D_{exp} and D_{demon} with ratios r_{exp}, $(1 - r_{exp})$;
  Shuffle B;
  for b *in* B **do**
    Calculate loss $J(Q)$ using target for batch b (ommit J_E if $b \in D_{exp}$);
    Perform a gradient descent step to update θ;
  end
  if $e \mod \tau = 0$ **then**
    Sync target θ' with θ
  end
  $c_{ep} \leftarrow 0, c_{batch} \leftarrow 0$; /* Number of completed episodes, saved batches */
  Reinitialize agents to a random state.;
  while $c_{ep} \leq n_{episodes}$ *or* $c_{batch} \leq n_{batches}$ **do**
    Run agents in parallel to produce a varied batch, completing i episodes;
    $c_{ep} \leftarrow c_{ep} + i$; /* Run at least a minimum number of episodes, batches */
    $c_{batch} \leftarrow c_{batch} + 1$;
    if $e > e_{pretrain}$ **then**
      Save batch ; /* Batches in pretrain are only for measurements */
    end
  end
end

2.5 Environment Exploration

The target of this thesis is not to create an algorithm with the best playing performance, or with the best learning ability. It is sufficient for the agent to be able to complete a level, even if there's a helping hand in the training process. In most Reinforcement Learning algorithms, the agent begins exploring the environment at the same initial state each time, without any expert knowledge. One form of expert knowledge that was utilized and is previously discussed is learning from observations. The two other forms that will be used are smarter state management, and a tailored epsilon policy.

Traditionally, the agent starts in the same state every time the environment resets. To accelerate learning and to sample more of the environment in a uniform fashion, the level was split into 10 equidistant states. The agent begins the environment in one of those 10 randomized states. Furthermore, a strict time limit is placed on the agent. If two neighbouring states are 20 seconds apart, the episode should expire at most after 40 seconds, so as to prevent collecting null data when the agent becomes stuck.

One additional improvement that is derived from randomized states is higher quality statistics. When the agent always starts at the beginning of a level, if a policy error occurs at the beginning 20 % of the level, then that will cause up to an 80 % drop in cumulative reward, even though the agent has a similar performance to before. By integrating over the state space for starting positions, the effect of a fault in policy is reduced, since the states from 20 % forward won't have their cumulative reward reduced.

A traditional epsilon policy overrides the agent's action ϵ percent of the time and chooses a random one in an equal fashion. This creates a probability tree with one correct path, whose probability gets exponentially smaller with each training frame that needs to be correct. In Mario, this presents the following problem: in order for a jump to be successful Mario has to jump for at least 10 consecutive training frames. Let's assume Mario's agent decides not to jump when its appropriate and can choose from 4 actions (each action having the same 25 % probability). Even if $\epsilon = 0.6$, the chance of completing a successful jump is $\pi = (0.6 * 0.25)^{10} = 5 * 10^{-7}\% = 0$, so a successful policy can not be created by exploration.

One of the solutions for this problem was to use a similar composition of random actions to those in

the dataset, Table 3.3: going right 50 %, jumping right 33 %, and staying still 16 %. The other was to use a sticky epsilon policy. As in when a random action is chosen, it becomes sticky and overrides the model for N training frames. A new ϵ' has to be calculated such that random actions constitute an ϵ portion of actions:

The problem of ϵ''s transformation can be modeled with a geometric distribution. Let's consider a slice of events that occurs right after a sticky move finishes until a sticky move begins again. The agent will choose the model output with $(1-\epsilon')$ probability or begin another sticky action with ϵ' probability. If the model output is chosen, then the cycle repeats. If a sticky action is selected then that action will lock for N moves, with 100 % probability after which point the cycle will repeat. The moment up to selecting a sticky action can be defined as a geometric distribution, and the number of model output actions chosen as the failures, with choosing a sticky action being the success. If with a success N actions are completed randomly, then beforehand there need to be M actions on average, such that:

$$\frac{N}{M} = \frac{\epsilon}{1-\epsilon} \Rightarrow M = \frac{1-\epsilon}{\epsilon} N$$

Where M is the mean of the geometric distribution, which is defined as follows:

$$\mu = \frac{1-p}{p} \Rightarrow M = \frac{1-\epsilon'}{\epsilon'} \Rightarrow \epsilon' = \frac{1}{M+1}$$

Solving for ϵ':

$$\epsilon' = \frac{1}{\frac{1-\epsilon}{\epsilon} N + 1} \Rightarrow \epsilon' = \frac{\epsilon}{(1-\epsilon)N + \epsilon} \overset{\epsilon \neq 0}{\Rightarrow} \epsilon' = \frac{1}{\left(\frac{1}{\epsilon} - 1\right) N + 1}$$

For $\epsilon = 60\,\%$, $N = 10$:

$$\epsilon' = \frac{1}{\left(\frac{1}{0.6} - 1\right) \cdot 10 + 1} = 13.0\,\%$$

So, instead of the probability of a successful jump being $\pi = 5 * 10^{-7}\%$ at any given moment, it's $\pi = 0.13 * 0.33 = 4.3\,\%$, which is a considerable improvement.

One further improvement in exploration vs exploitation was the use of a hybrid epsilon policy. Epsilon is set to a high amount and decays exponentially in the following manner:

$$\epsilon_{exp} = \frac{1}{I} e^{\frac{s}{d}}$$

where s is the epoch, I is a dividing constant that makes epsilon start lower than 1, and d is the constant that sets the decay, according to how long the agent will train. As the exponential epsilon becomes lower, a second function takes effect, which uses the mean reward of the last epoch as a signal to boost exploration.

$$\epsilon_{rew} = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * \min\left(\frac{r_{max} - r_{curr}}{r_{max}}, 0\right)$$

where ϵ_{max}, ϵ_{min} define the range of epsilon required and r_{curr} is the current reward and r_{max} the typical maximum reward.
The epsilon policies are combined to form the following composite policy:

$$\epsilon_{comp} = \max\left(\epsilon_{exp}, \epsilon_{rew}\right)$$

Typical values for ϵ_{rew} would be in the range of 1 % to 15 %, so as the exponential epsilon decays to a lower number, ϵ_{rew} can increase the exploration if there's poor performance.

3 Results

The setup described above was implemented using the first level of Mario. Then, two learning approaches were followed, one with the modified epsilon policy and one that uses demonstrations with a minimal epsilon. For each set of parameters multiple runs were executed. A rolling mean window was applied on each run and they were combined to calculate the mean and confidence interval of the parameter set.

The highest score achievable for the first level of Mario, starting from the beginning, is 3100. 10 equidistant starting points were used in training for the level, with each point having a maximum score proportional to its distance from the end of the level. As such, the maximum expected mean reward for an agent is around half of the maximum, or 1500 points.

The performance of each model changed considerably every epoch and between each execution, while showing a trend, so after applying the window and finding the mean the performance of the algorithm will appear lower than it is. By selecting the best model out of the best execution, the mean score will be much closer to the maximum.

3.1 Demonstration vs Exploration

In this first set of executions, we will compare the performance of learning by demonstration (low epsilon) and learning by an epsilon policy. The baseline will be doing neither: using a low epsilon policy with no demonstrations, which featured a maximum score of 500 and a typical score of 280.

Demonstration varied in performance based on the ratio of experience and demonstration, with the best being 30 % and having a maximum mean score of 1600. Following an epsilon policy, on the other hand, had a more predictable trend. While it may seem the exploration agent is not learning, the epsilon policy starts to have positive performance spikes after epoch 80 that exceed score 800 and reach 1000 multiple times. They do not appear in the confidence interval because a rolling window is applied to the executions to help show a clearer trend.

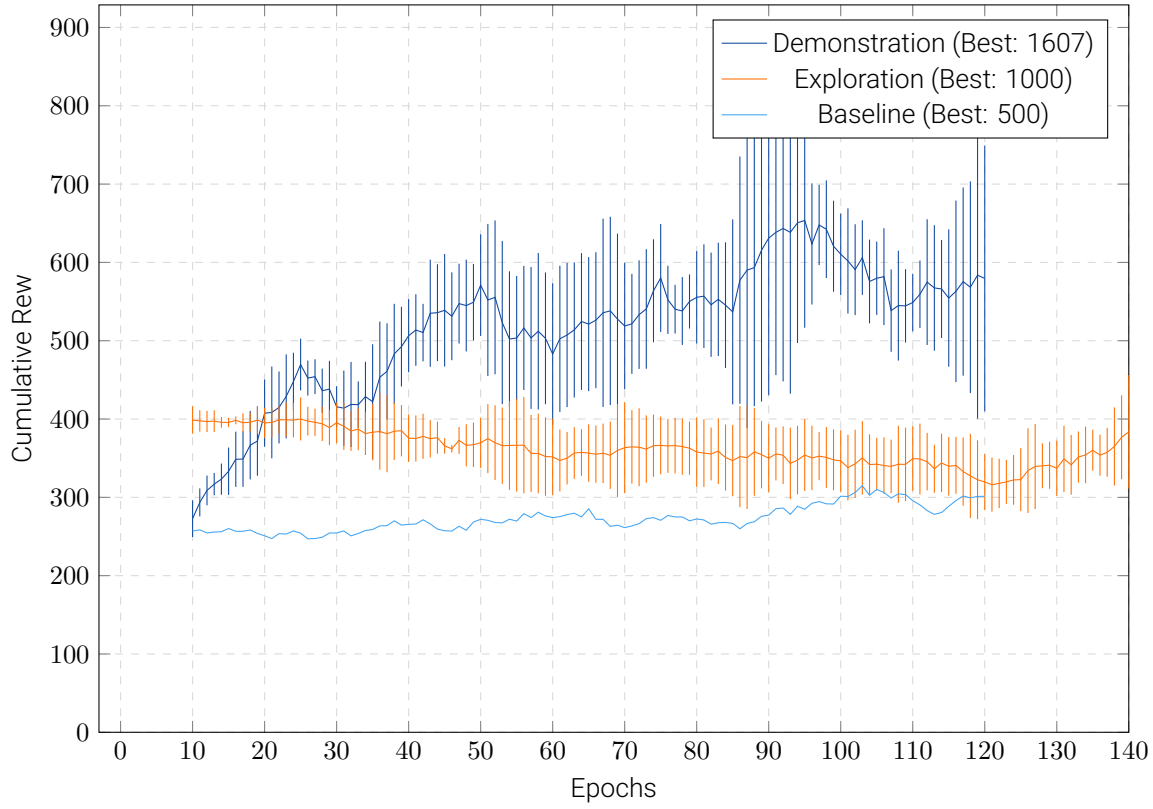


Figure 5.1: Demonstration vs Exploration

3.2 Demonstration Ratios

One important hyper-parameter in learning was the ratio of demonstration batches to experience batches. In the beginning of each epoch, a number of batches is selected in a predetermined ratio from the experience and demonstration replay buffers. Then they are shuffled together, leading in the model alternating training from demonstration and experience.

In the chart below, there is a range of experience ratios from 0 % to 100 %. Having a 40 % experience ratio would mean 40 % of batches are from the experience buffer and 40 % are from the demonstration buffer. From ratio 50 % and beyond, the performance starts to heavily decline, showing that it is preferable to use a majority of demonstration batches when training. The best performance is seen at around 30 %, with 25 % having great performance and 0 % being surprisingly good, considering the environment is not used at all.

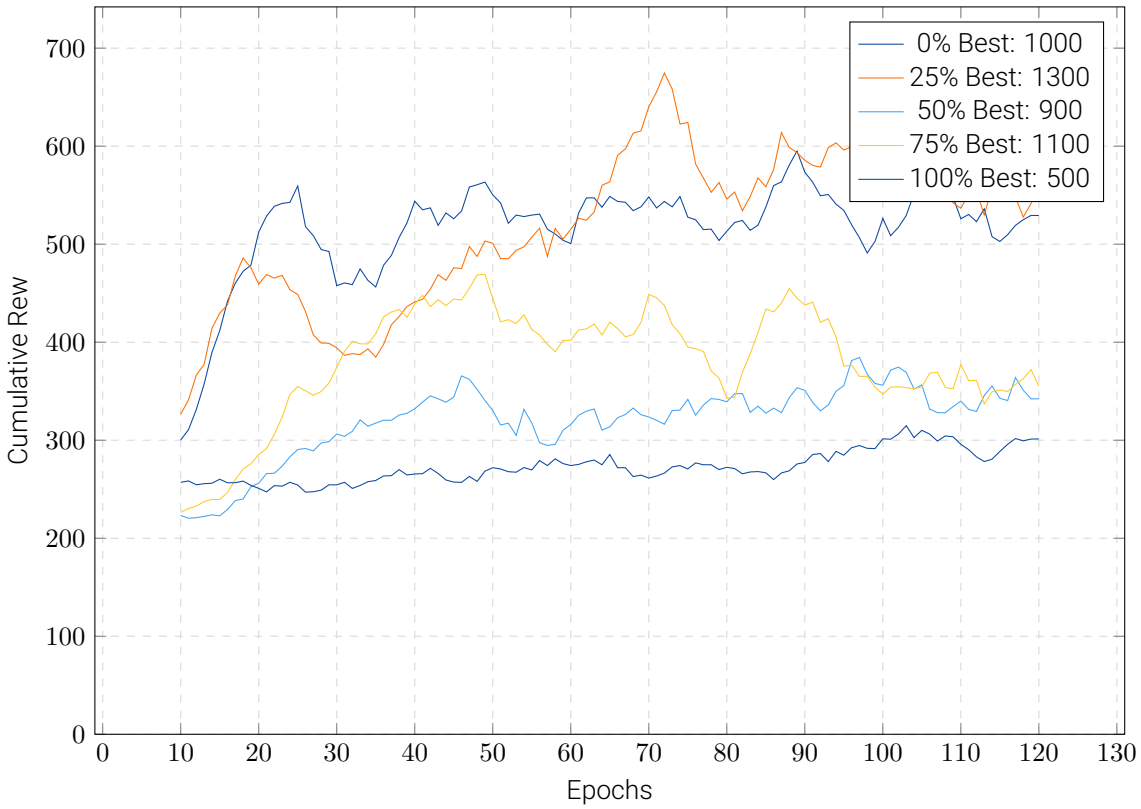
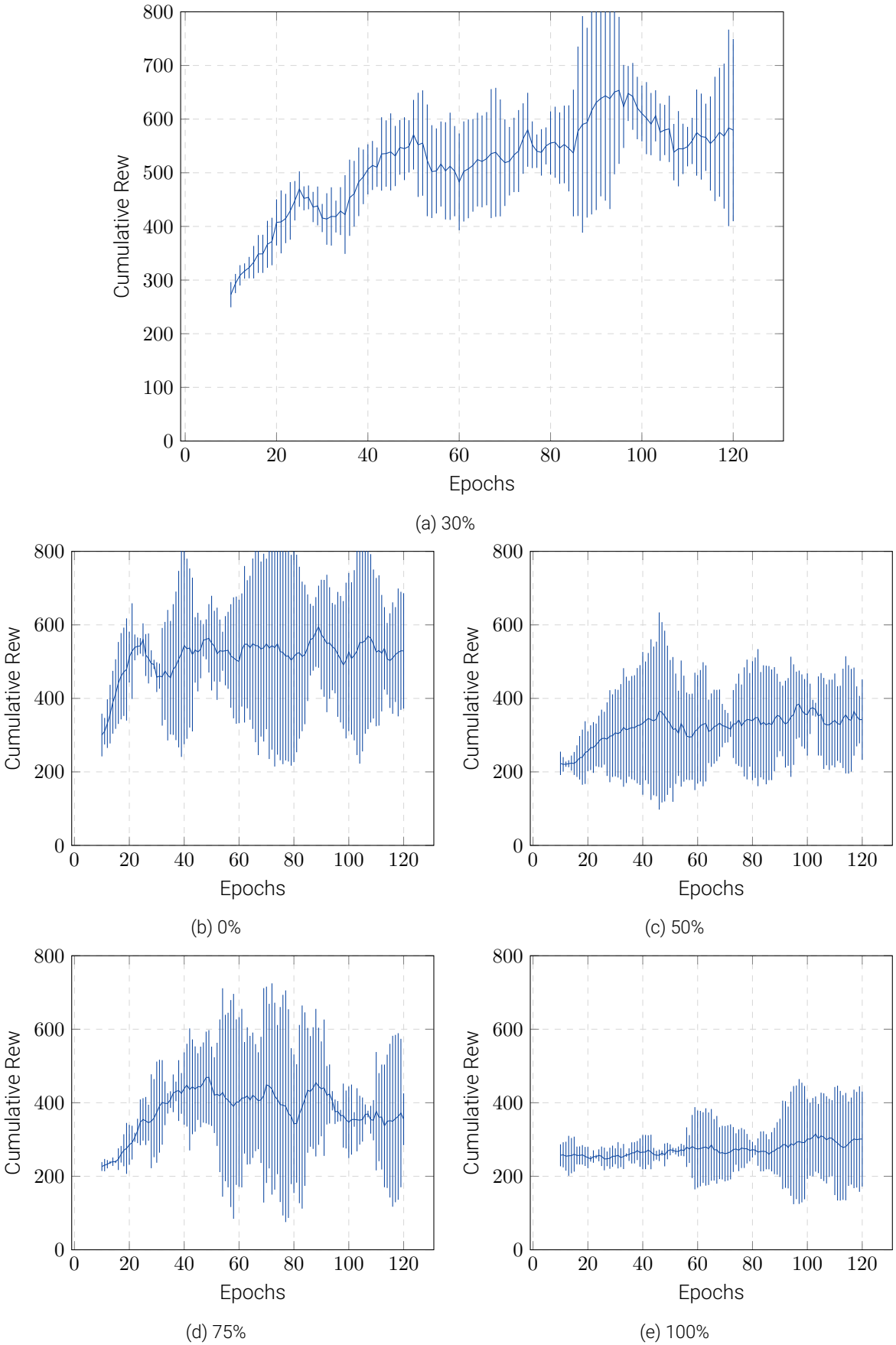


Figure 5.2: Experience to Demonstration ratio, plotted together

In the next page, the confidence intervals of each execution can be found. 0 % is the least stable of the executions, with 100 % being the most stable (showing little learning).

4 Summary

In this chapter, an agent for completing the first level of Super Mario Bros was created, using a modified form of Deep Q Learning. The agent is robust and can complete the level most of the time. Various combinations of hyper-parameters and models were tried, with using a demonstration heavy replay buffer with a low epsilon policy being the best. In the next chapter, 3 affective agents that focus on emotion will be created and then combined with this one to form a composite agent.



(a) 30%

(b) 0%

(c) 50%

(d) 75%

(e) 100%

Figure 5.3: Experience to Demonstration ratio, with 80% confidence interval

Chapter 6

Affective Learning

In the previous chapter, we created an agent that completes the first level of Mario by having its sole focus on the objective. In this chapter, we'll create a set of affective agents, who try to maximize a specific emotion without caring for completing the level. Alongside the agents, we'll design objective metrics for emotion that, by using a train and test split, can be used to measure performance.

The dataset created for this thesis is based on the P.A.D. model. It contains 3 emotions, Pleasure, Arousal, and Dominance, which will be used to create 3 Agents. Those agents will use a modified version of Deep Q Learning from Demonstrations, so their Q functions will be able to be combined afterwards to form a composite agent.

1 Methodology

The main limitation of training an agent to use emotions is that, unlike a traditional reward function, the agent can't explore the environment to receive new data. Environment exploration can't be utilized to train the agent. Moreover, unlike Offline learning, where access to the environment is artificially limited in training, the environment can't be used in benchmarking the agent either. It is logical, therefore, to model the problem using supervised learning, with the caveat that instead of a traditional softmax model, Deep Q learning will be used.

1.1 Algorithm

The modified Deep Q Learning algorithm described in Section 5.2.4 will be utilized. Since exploring the environment can't be used to generate new data for an exploration replay buffer, only the demonstration replay buffer will be used.

During testing it was found that using a blanket margin classification loss for all state action pairs is problematic. The margin classification loss penalizes the agent for giving high Q values to actions not chosen by the agent. This is required to balance the Q functions of actions that are not chosen in the demonstration. When using traditional Q learning, a bad action receiving high Q value would be balanced by having the agent explore it and add the low reward to the replay buffer. Without the exploration component, actions not chosen by the demonstration would receive inflated Q values and have those wrong Q values propagated to previous actions, leading to an invalid policy.

It also has an imitation learning aftereffect, in which actions of the demonstrator, who's considered an expert, are reinforced, regardless of their performance. This imitation learning bias is enough to strip the effect of the affective annotations. The agent blindly imitates the expert, regardless of emotion. A balance was found in the use of margin classification loss, by having it only apply when the agent receives an affective reward:

$$J_E(Q) = |R(s, \alpha_E)| \left[\max_{\alpha \in A} [Q(s, \alpha) + l(\alpha_E, \alpha)] - Q(s, \alpha_E) \right]$$

where α_E is the action of the expert at that state, $R(s, \alpha)$ is the reward function for s, α pairs, and l is a penalizing function which is 0 when $\alpha = \alpha_E$ and a positive value when not, causing values of $Q(s, \alpha)$ to be

at least l lower than $Q(s, \alpha_E)$, but now only when there is a reward.

This leads to a drop in performance, since a blanket margin classification loss allows the agent to learn from the whole dataset, not only the minority which features rewards. To amend this, transfer learning was utilized by having the model start using the weights of the best Reinforcement Learning model that was produced in the last chapter.

1.2 Metrics

The goal of this chapter could be best described as a conditional imitation problem. The agent has to imitate the expert if and only if it will be emotionally rewarded. The metrics that will be used will be focused on that imitation learning component. There is another element of maximizing affect that wasn't studied, and that is how the agent navigates the environment to reach highly emotional states, by following the Q function.

The first metric is imitation accuracy when there is an annotation. Mathematically, it is defined as the number of actions chosen that match the demonstrator when the reward is non-zero, divided by the total non-zero annotations.

$$\alpha_i\% = \frac{\sum_{(s,a)\in D_{replay}} |R(s,a)| \cdot [\arg\max(Q(s,a)) = a]}{\sum_{(s,a)\in D_{replay}} |R(s,a)|}$$

Where D_{replay} is either the train or test set experience replay buffer.

The second metric that was used is average rank annotation accuracy, where rank is the position of the action, when sorted by Q value. Mathematically, it is defined as the average of the ranks of the actions chosen when the reward is non-zero, divided by the total non-zero annotations.

$$\alpha_{i,rank} = \frac{\sum_{(s,a)\in D_{replay}} |R(s,a)| \cdot \arg\max(Q(s,a))}{\sum_{(s,a)\in D_{replay}} |R(s,a)|}$$

Broadly speaking, both metrics convey the same information. As it will be shown in the results section, there was a strong inverse correlation between the two metrics, that would render using both unnecessary. The correlation is inverse because with a better performing model, the average rank is lower while the prediction accuracy is higher.

2 Results

The dataset was split into a train and test set, and two approaches were tried: transfer learning and random initialization. Transfer learning was performed by copying the weights of the layers of an RL network, since both architectures are the same, sans the last layer, which is different. Statistics were compiled for each emotion separately, and then combined to form an overall metric for emotion, weighed by each emotion's occurrence.

Two baseline metrics are presented alongside the accuracy graphs. The baseline metric is based on the majority class action of that emotion, according to Table 3.4. The Imitation metric is the performance from the imitation learning figure 4.2. Since this model is based on Q learning, the action taken by the demonstrator when there was an annotation is not necessarily the best for maximizing reward, so the performance of imitation on the whole dataset is not necessarily indicative.

Imitation performance when there's an annotation exceeds the baseline in accuracy. The best accuracy was around epoch 25–50, after which point the model goes on to over-fit. There is a soft L2 weight decay applied to the model, which was enough to make over-fitting negligible in the inference chapter, while not lowering accuracy. Since the model stops learning after that point, it is fair to assume that over-fitting occurs due to lack of data, both on test and train sets, and not on model architecture.

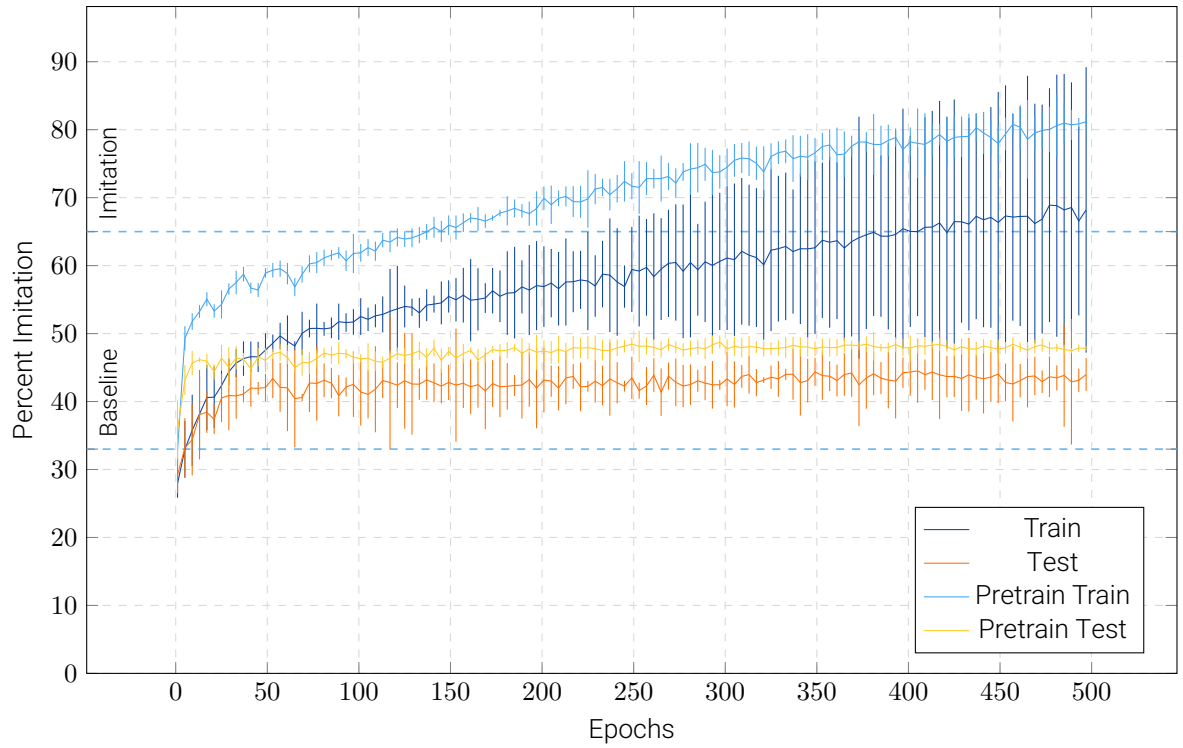
The model that is initialized randomly has a larger variance in-between executions than the one that utilizes transfer learning (pretrain). This is logical due to the small percent of annotations in the data, which would make learning from scratch difficult.

The annotation mean rank displays the model performance in a little more depth. For example, over-fitting is much more visible after epoch 50, due to a rise in average ranks, where it is barely visible in accuracy. However, unlike accuracy with the majority class, there doesn't exist a simple way to calculate baselines for it.

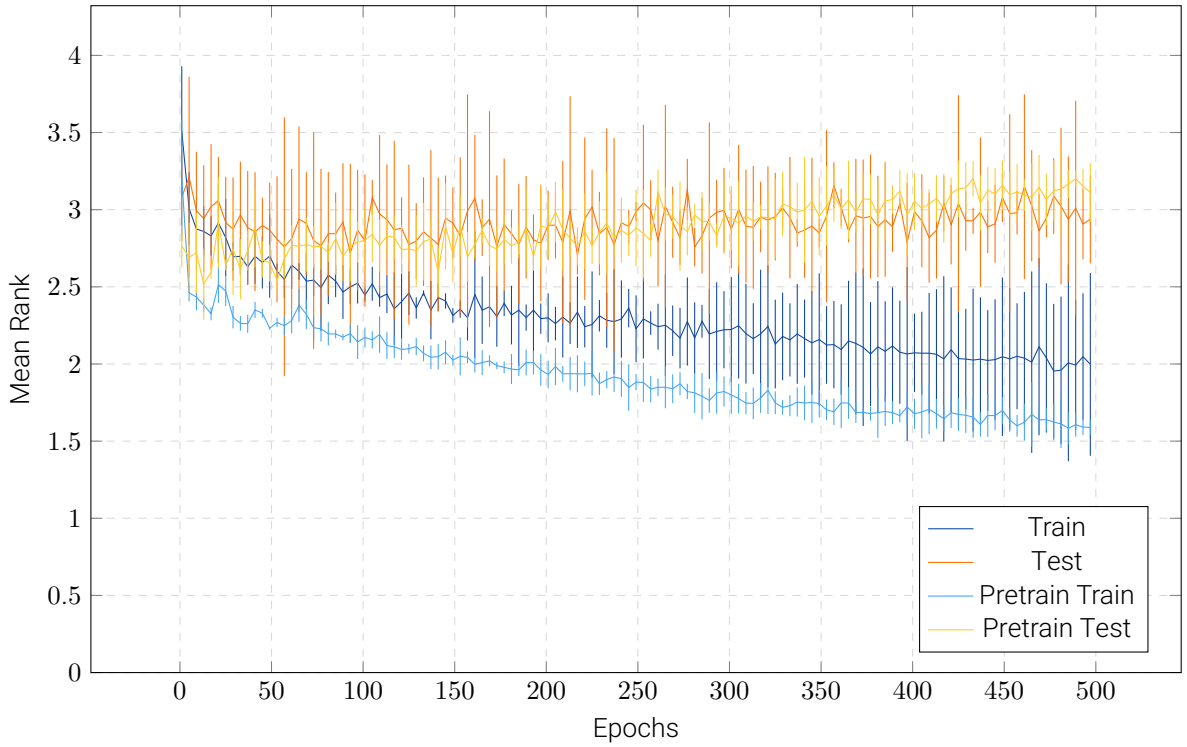
Pleasure and Arousal feature similar performance to overall, with arousal outperforming pleasure. Since arousal is the most common annotation of the three, this is expected. Dominance hovers around the baseline, both due to being a more complex emotion and due to having a large negative percentage, which was largely ignored in the construction of the metrics. However, as it will be shown in Chapter 7, even though the metrics for dominance aren't promising, the agent does learn a useful policy from it.

3 Summary

In this chapter, 3 affective agents were developed and benchmarked using a form of supervised learning. Testing them using the environment was ommited due to the expected low ingame performance of the agents, when trained using affect alone, which will be shown in the next chapter. There is significant variance in the model test results, but that is expected considering the dataset size. In the next chapter, the 3 affective agents developed will be combined with the reinforcement learning agent to form a composite agent that considers both affect and the primary objective.

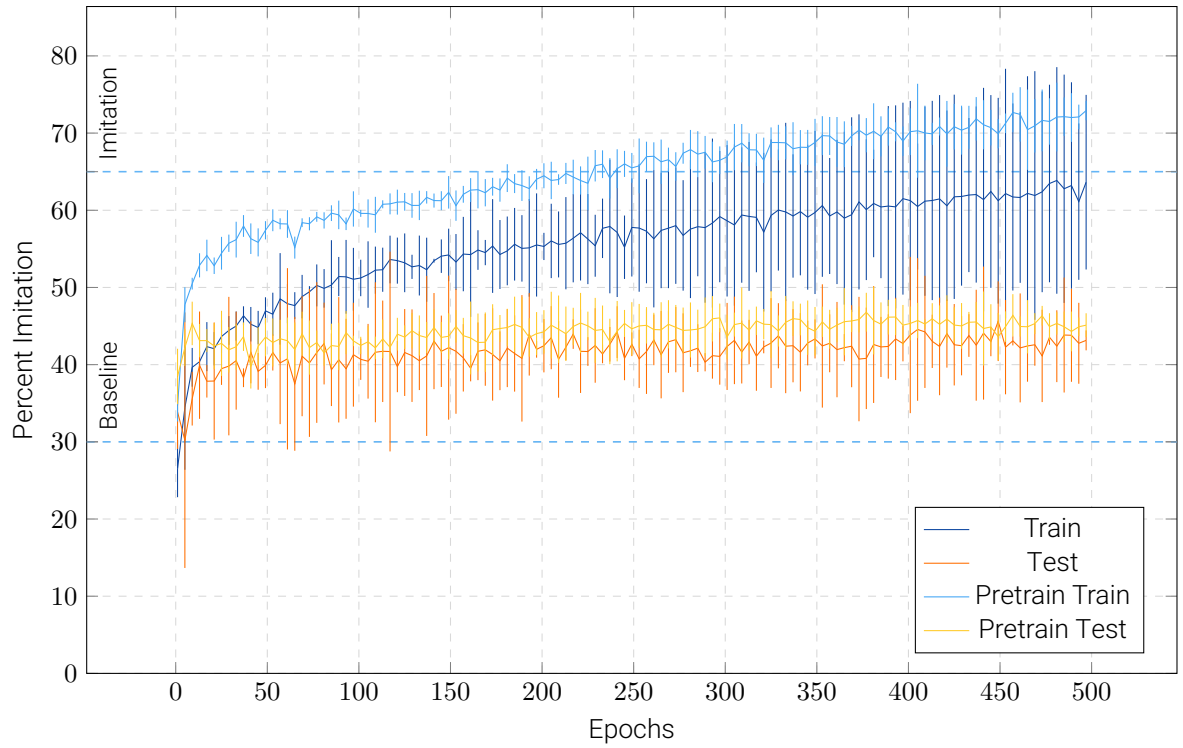


(a) Accuracy

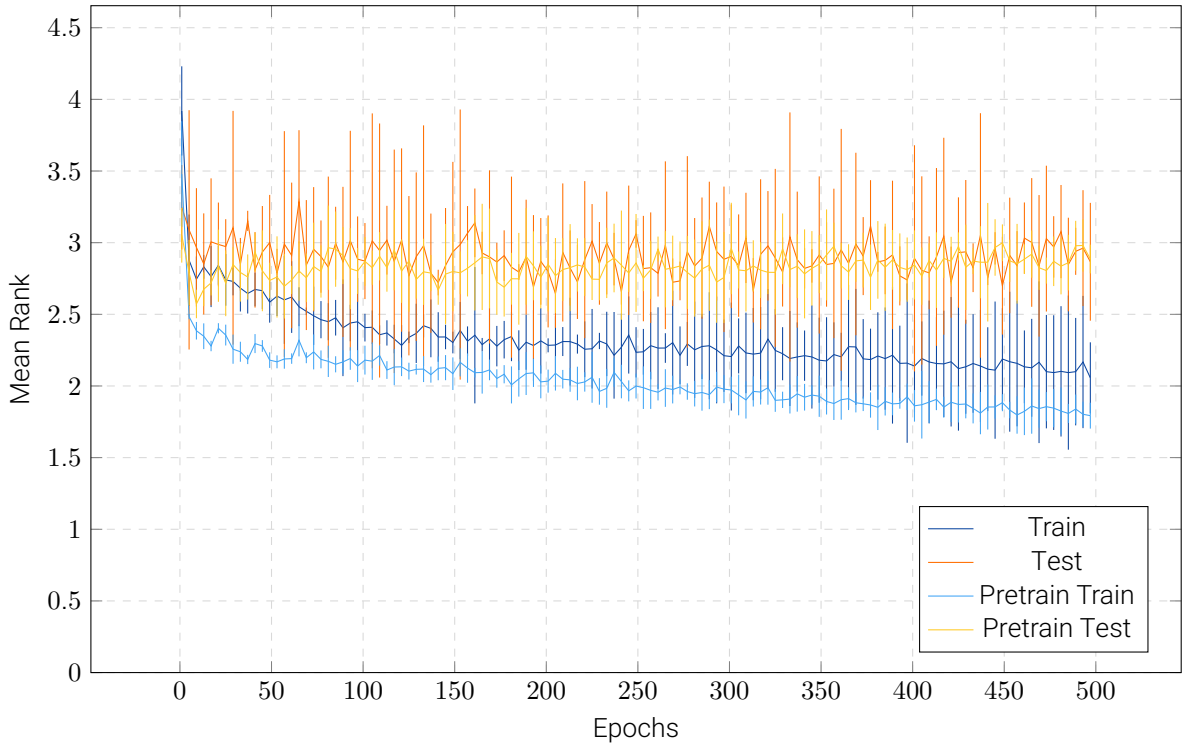


(b) Rank

Figure 6.1: Overall Performance

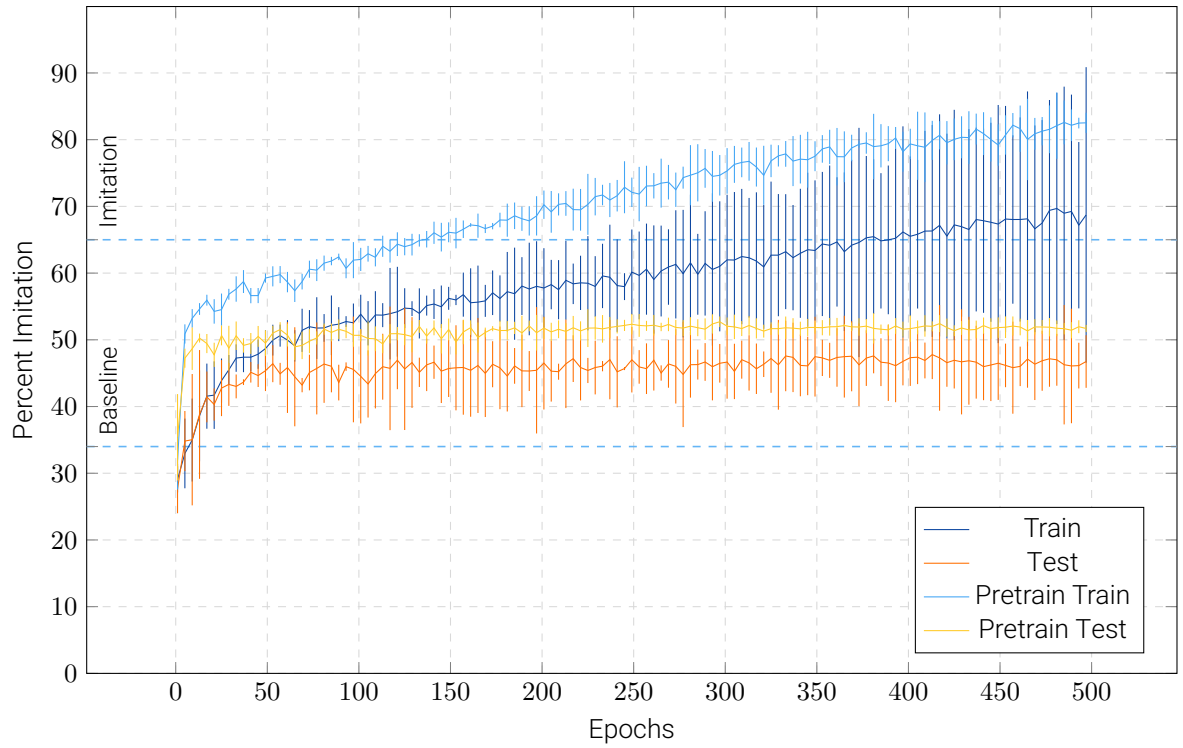


(a) Accuracy

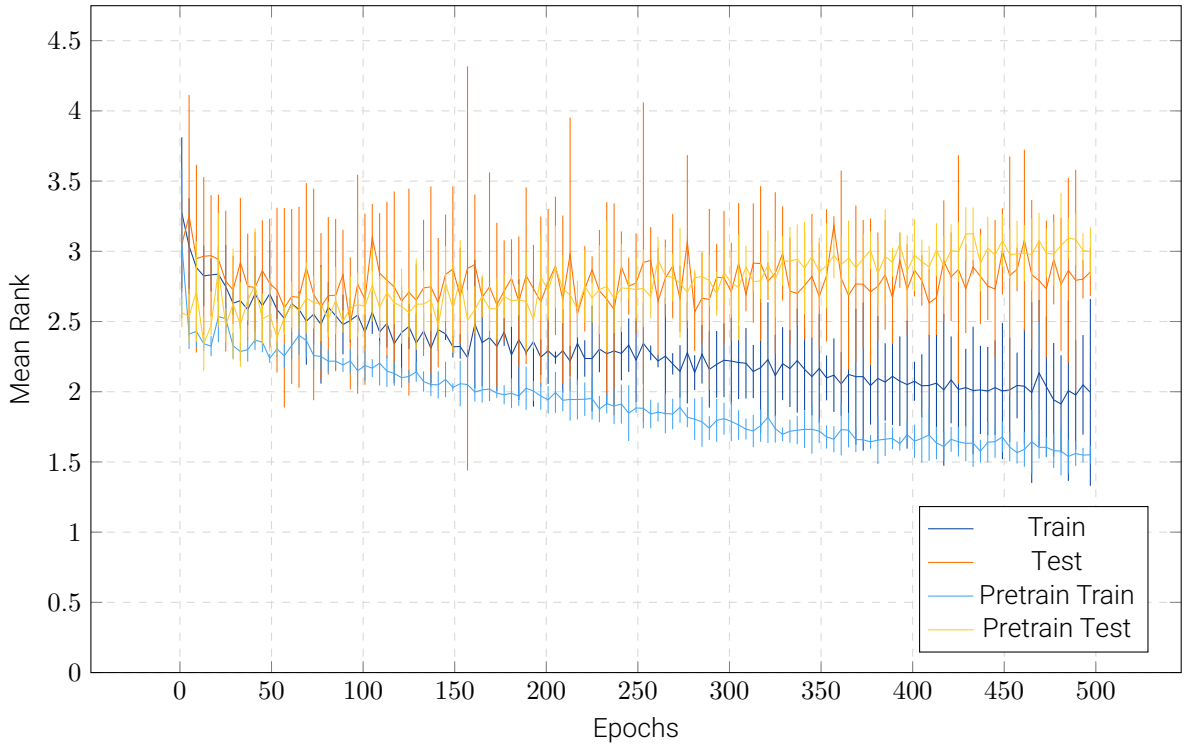


(b) Rank

Figure 6.2: Pleasure Performance

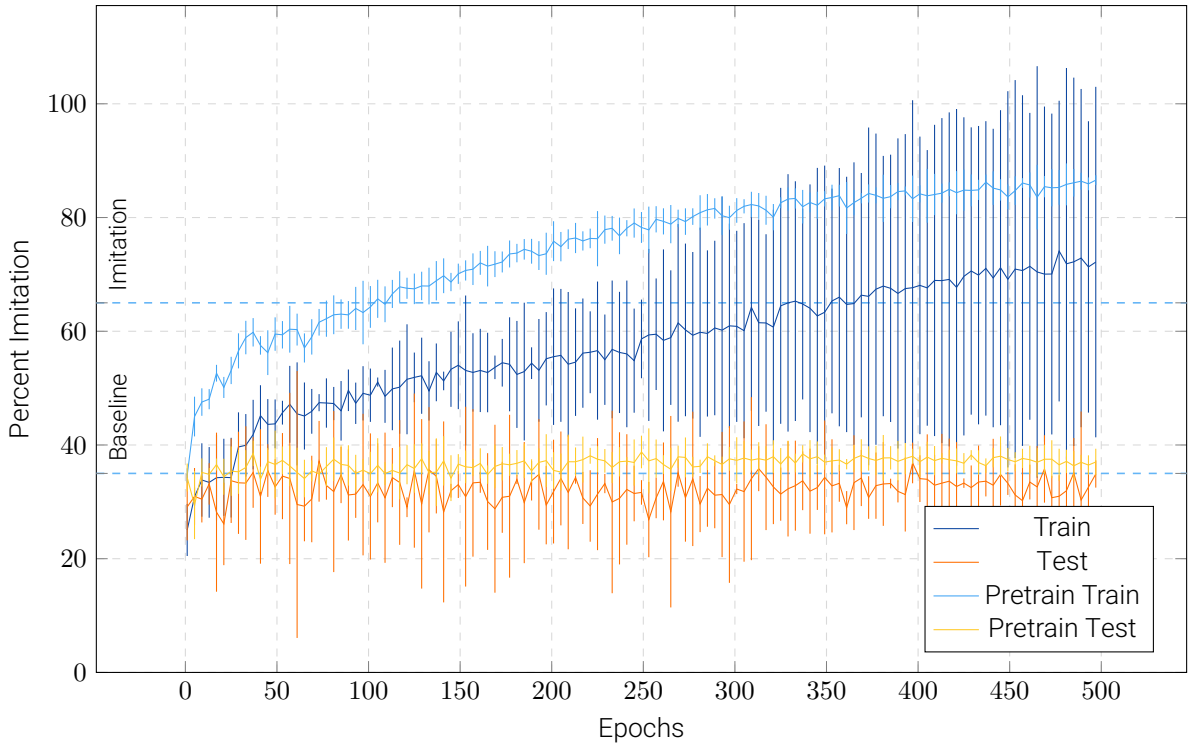


(a) Accuracy

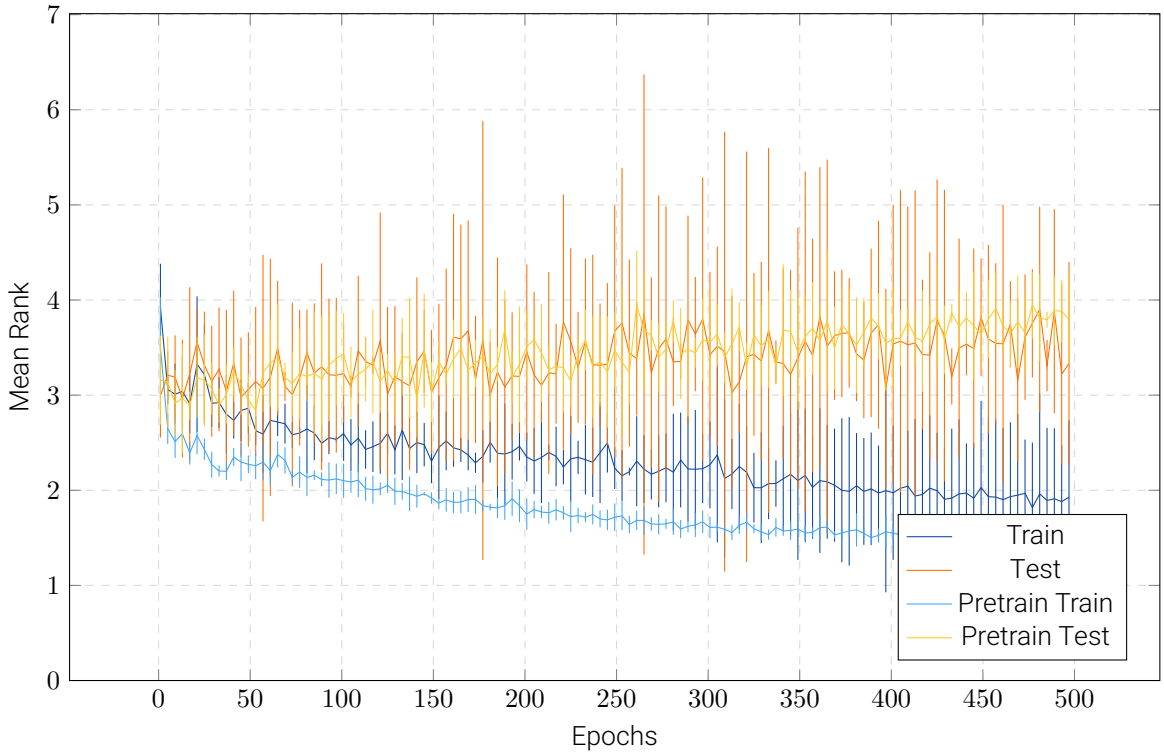


(b) Rank

Figure 6.3: Arousal Performance



(a) Accuracy



(b) Rank

Figure 6.4: Dominance Performance

Chapter 7

Composite Learning

In the previous chapters, a dataset containing emotional annotations in the game Mario was created, alongside 4 agents. The first agent, discussed in Chapter 5, focused entirely on completing the primary objective, which was the level. The other three, discussed in Chapter 6, tried to maximize the emotional charge of the agent in the three emotions contained in the dataset. All 4 agents were trained using a variant of Deep Q learning and are based on a Q function. In this chapter, we will form composites of those 4 agents by combining their Q functions in different ratios and discuss the results.

1 Methodology

The four agents that were developed in the previous sections are carried over. They are based on Deep Q Learning with the Q function obtained by solving the following Bellman Equation:

$$Q^{\star}(s,a) = E\left[R(s,\alpha) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^{\star}(s',\alpha')\right]$$

The reinforcement learning agent uses as a reward whether Mario is going right, which is the primary objective, while the other agents consider the affective annotation directly as their reward (It is a binary annotation with values $[-1, 0, 1]$), which are the secondary objectives.

The Q function defines the expected reward, in a relative sense. As in a Q value of 54.6 doesn't tie into the environment directly, but an action having it would be preferable to one with 30. If we want a target that considers multiple objectives equally, say completing the level and being happy, then we should add their Q functions before choosing an action:

$$Q_{comp}(s,a) = Q_{rl}(s,a) + Q_{af,pleasure}(s,a)$$

Since Q values are relative, we can add them in a ratio to influence the weight placed on each objective. Let's define weight w and Q function Q vectors such that:

$$w_{af} = \begin{bmatrix} w_{pleasure} \\ w_{arousal} \\ w_{dominance} \end{bmatrix} \quad w = \begin{bmatrix} w_{rl} \\ w_{af,1} \\ w_{af,2} \\ w_{af,3} \end{bmatrix}, \quad \sum_{i \in N} |w_i| = 1, \; w_i \in [-1, 1]$$

$$Q_{af} = \begin{bmatrix} Q_{pleasure} \\ Q_{arousal} \\ Q_{dominance} \end{bmatrix} \quad Q = \begin{bmatrix} Q_{rl} \\ Q_{af,1} \\ Q_{af,2} \\ Q_{af,3} \end{bmatrix}$$

Therefore:

$$Q_{comp} = w^T \cdot Q$$

The action a for state s is then chosen by:

$$a = \arg\max_{\alpha \in A} Q_{comp}(s,\alpha)$$

2 Results

Having created the agents, it's relatively simple to use the math provided above to create composite agents and test them. For each composite agent, we sample from the Q of reinforcement learning agent and the Q of an affective agent in a ratio.

We modulate that affective ratio from 0 to 1, with 0 creating an agent purely focused on completing the game and 1 creating an agent purely focused on maximizing affect. This ratio, as a percent, defines the X axis. Then, using each ratio, the agent is placed in the environment and plays 50–150 episodes, using the same environment as in training (scores are comparable). The agent is then pitted against the affective test set, measuring imitation performance when an annotation exists. Those two metrics define the dual Y axis of the following graphs.

2.1 One Emotion

The first set of graphs create composite agents that focus on one emotion each. The orange line and left Y axis measure the agent's mean cumulative reward, while the blue line and right Y axis measure the imitation performance as a percentage on the test set. The composite Q function is calculated as follows:

$$Q_{comp,i} = (1 - r_{af})Q_{rl} + r_{af}Q_{af,i}$$

At 0 %, environment performance is indicative of the reinforcement learning agent, while at 100 % affective performance is indicative of the affective agent. Both curves feature a steep drop-off for all three emotions. At either end, we create an agent that's good only at either objective. This is expected, since we only use one of the agents.

The golden ratio appears to be at 25 % and 75 %. At 25 %, the environment performance drop is small, while there's a considerable gain in imitation. At 75 %, most of the imitation gain has been achieved, so there can be some gain in performance. In fact, for pleasure and arousal, using a small percentage of the RL agent creates a better affective agent! Most of the tests in the next section will hover in 20 %–30 % for the aforementioned reasons.

2.2 Negative Imitation

As a second experiment, the affective Q function was flipped, to cause the agent to counter imitate the demonstrator when there's an annotation. In principle, this should create an affective agent that minimizes the his respective emotion. The composite Q function is calculated as follows:

$$Q_{comp,i} = (1 - r_{af})Q_{rl} - r_{af}Q_{af,i}$$

As expected, imitation performance starts at the baseline of using just the RL agent and drops to zero as the affect ratio is increased. There is a steep drop in environment performance as well. This is expected, as the actions that maximize affect overlap with those that constitute a good strategy. Therefore, to minimize affect, the agent must lose.

2.3 Multiple Emotions

As a last experiment, multiple emotions were combined. The Q functions of two emotion each time were combined in a 50 % ratio.

$$Q_{comp,i} = (1 - r_{af})Q_{rl} - r_{af}\left(0.5Q_{af,i} + 0.5Q_{af,j}\right)$$

The results are similar to the previous experiments. The imitation performance of both emotions rises.

Figure 7.1: Creating composite Q functions by summing Affect and Objective Qs. Different Ratios Plotted.

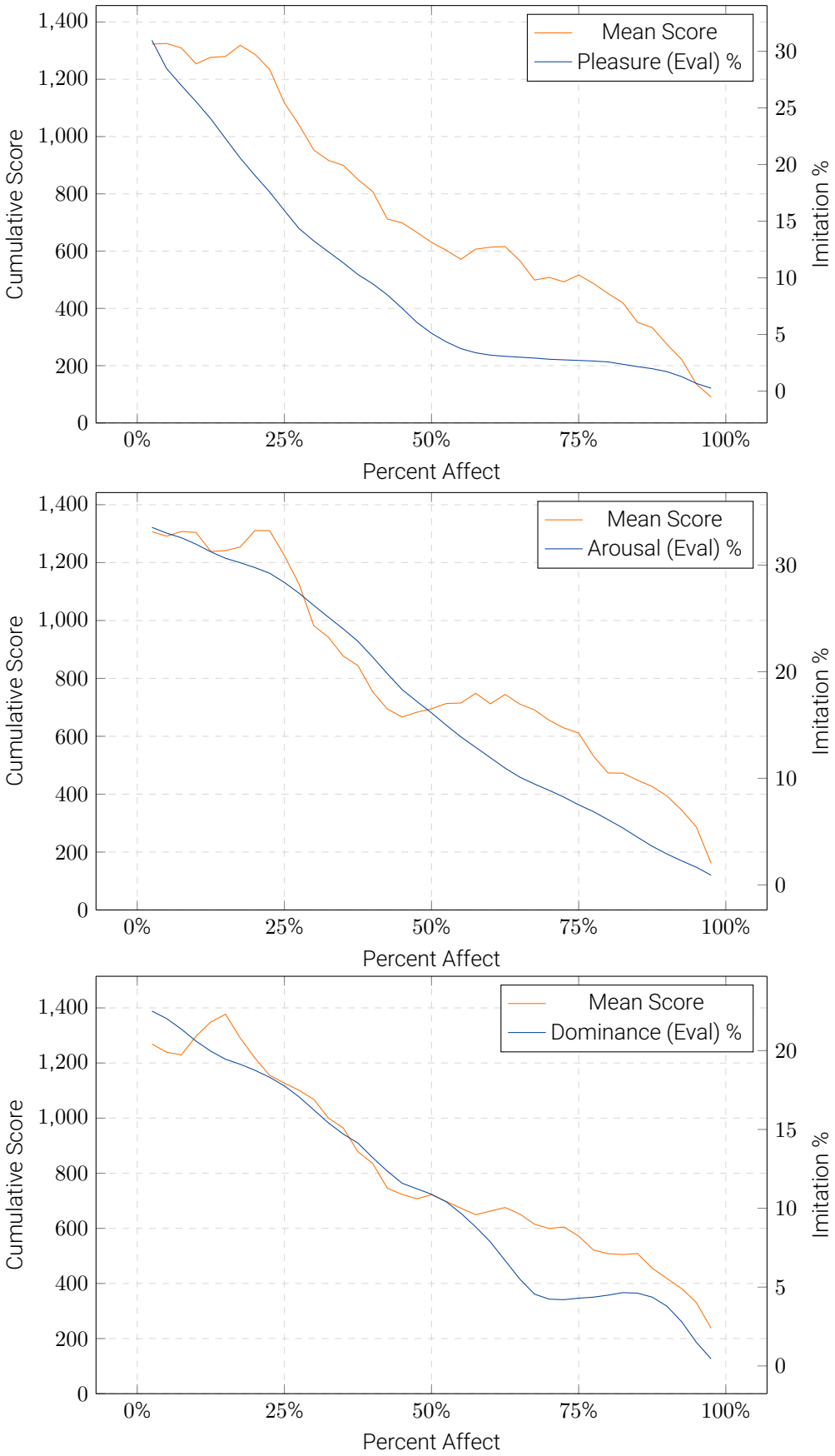


Figure 7.2: Composite Q functions, with the affective Q sign flipped to counter-imitate.

Figure 7.3: Composite emotions. Combining Pairs of Q affect functions in a 50% ratio instead of using 1.

3 Trails

To add more flair to the thesis, and visually demonstrate the work, two tools to create playtrace trails of the agents were developed. The training environment was copied into a new tool. This tool receives a starting state and weights for each of the Q functions. Then, it generates N episodes, starting from the provided state and using the provided weights, listing them by score.

Afterwards, one of those episodes can be input into another tool. That tool generates the frames of the episode and then splices them together by their distance from the start, using their difference to display Mario's trail. Additionally, the normalized Q function is graphed on top at each moment, displaying Mario's expected returns.

3.1 Base RL Agent

The base RL agent leaves good a margin between enemies when jumping and avoids breaking blocks. It has a conservative strategy that will complete the level, while minimizing the chance of loss.

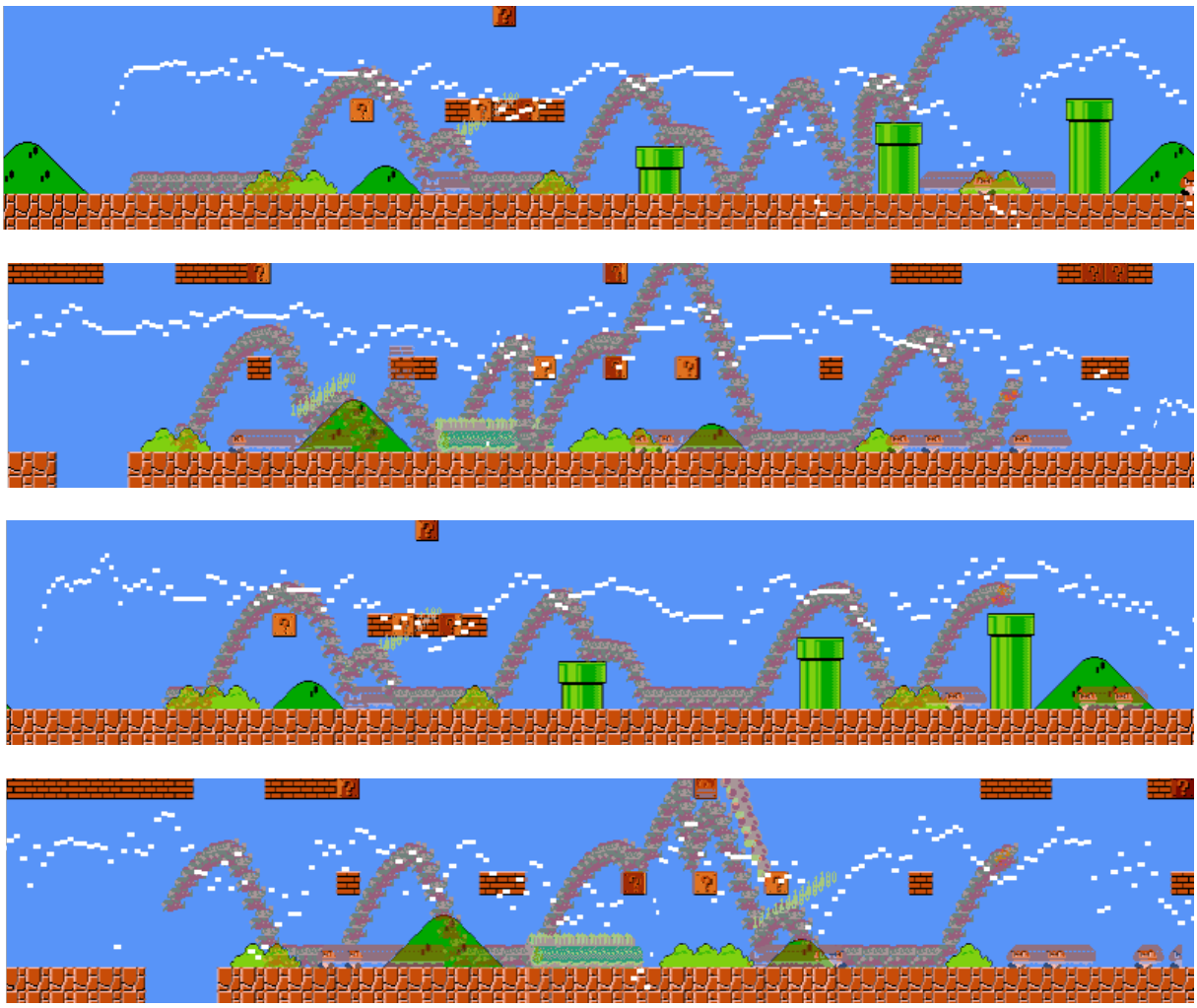


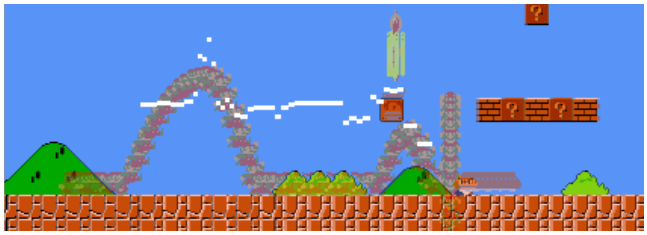
Figure 7.4: Base RL agent playing through the first level of Mario

3.2 Affective Agent: Pleasure

The RL agent Q function was combined to the Pleasure Q function using a 70 % to 30 % ratio, which retained a large part of the playing performance. In the following images, the Pleasure Q function is graphed on its own, not the composite one.

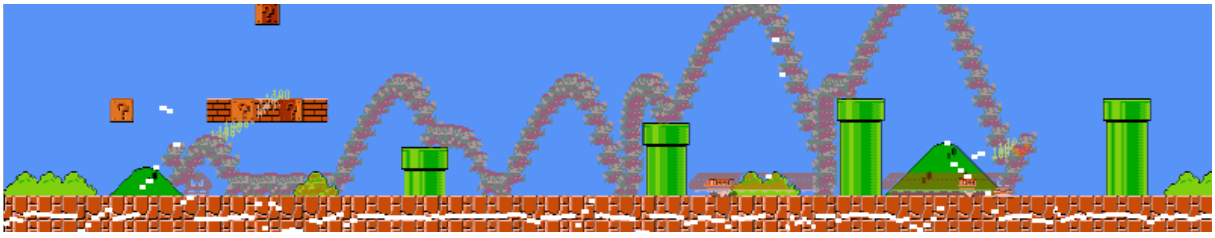
When starting from the initial position, the agent tries to break the first block and dies (inferred pleasure objective). Because it is impossible to break that block without losing without going backwards, afterwards which it has not learned.

When, due to the slight epsilon policy, it avoids the block, it jumps on top of the platform and breaks the other one. In both cases, the Q function spikes as Mario is approaching the block, and reduces after hitting it (as is expected).



3.3 Affective Agent: Arousal

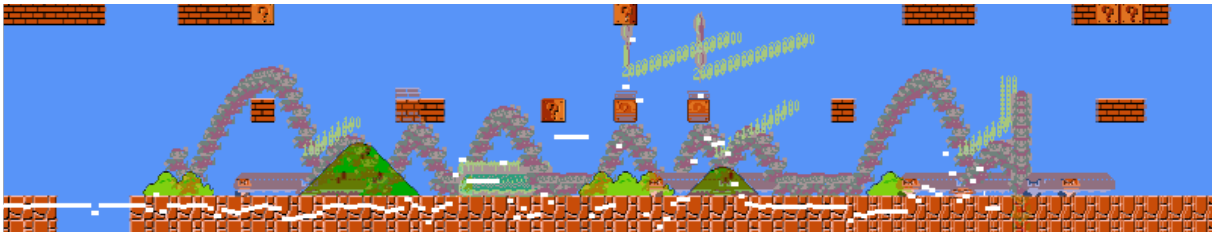
The agent that is focused on arousal, using a 80 % RL to 20 % affect ratio, comes into closer contact with the enemies it approaches and kills some of them. In the first image, it kills 2 of the initial enemies of the level. And in the second it kills 2 enemies as well, while getting closer to enemies than the RL agent would on its own. When it's close to enemies, the Arousal Q value becomes higher.



3.4 Affective Agent: Dominance

The dominant agent, using a 80 % RL to 20 % affect ratio, tries to kill as many enemies as possible. In one episode, the agent kills 4 enemies.

In the first image, the first enemy of the level is killed, and the Q function spikes beforehand. In the second image, the agent kills two enemies, having a heavy spike when killing the first enemy and a smaller spike on the second one. In the last image, the agent kills 2 other enemies, before meeting his demise on the 5th one, due to a missed jump. The Q function spikes correctly before all of the kills, but not when the agent is about to die. This is surprisingly good, since it shows the agent knows it's about to die and not kill an enemy, even though the jump was missed only by a couple of pixels.



4 Summary

In this last section showcasing the work of this thesis, the agents that were developed in the previous section are combined into a composite agent. Then, they are evaluated using objective metrics. Lastly, a visualization tool is used to show what the agent is thinking while playing the level and what path it is selecting.

While the results shown from the objective metrics might have been thought of as middling, the visualization tool shows that the agent thinks correctly when playing the level and exceeds the expectations set in this thesis. All three of the emotions that were annotated achieved visible differences in behavior for the agent, in the ways that are supported by the objective annotation definitions (Table 3.1). Therefore, the workflow and results showcased in this thesis show promise for use in future works.

Chapter 8

Conclusions & Future Work

1 Conclusions

The work presented in this thesis culminated into creating a multifaceted agent that, once trained, can have its behavior modified by a simple change of weights. If used in production, that means that one set of models (RL and AF) can be shipped with a product and agent behavior can then be modified on command, without a performance penalty or retraining, while presenting a rich range of actions.

The objective of this thesis was to create a proof of concept and lay the foundation for future works. Playing the first level of Super Mario Bros successfully has been achieved multiple times and in various ways, including using DQNs, after all. And for those purposes, it was successful. The final agent presents a breadth of modalities and the process that was developed can, with little modification, be used to model agents in thousands of games and levels within those games. Most of the effort required would be put into creating a new affective dataset for that game.

2 Limitations

The Tool that was created focused on OpenAI's Retro environment, which features support for most RetroArch emulators. While this allows for a large breadth of games, it still limits the environments that can be studied, since parts of the tooling rely on Retro environment's state loading behavior and would have to be adapted to be used with other environments. Also, since the annotator uses the OpenAI Retro environment internally it needs to run locally on a computer, and can't be run on a browser. This is somewhat offset by the ability to create a self-contained executable but it's nonetheless a platform-centric limitation.

The Dataset that was created was, in purpose, limited and when creating the agent it was limited furthermore. The dataset was first party annotated, with a single annotator, and emotions were explicitly associated with actions beforehand (ex. collecting coins equals pleasure). In that way, the agent's behavior was dictated by those associations and not by affect. These decisions were taken to reduce noise, produced due to low inter-rater agreement and a ill-posed ground truth, so that the dataset could be smaller, simpler to create, and have a higher likelihood of success. The dataset was also limited in scope to one game, Super Mario Bros, and in that game on one world (a collection of 4 levels). In the RL and composite learning chapters, only the first level of Mario was studied (due to time and compute constraints), leading to three quarters of the dataset being unused for the agent.

Deep Q Learning is a staple algorithm in Reinforcement Learning but in recent years has been surpassed by other on-policy algorithms in their ability to learn. It was chosen due to its off-policy nature making it easier to learn from the dataset directly. Traditionally, it is also slow to converge. Due to optimizations done in this thesis this was not the case, with each model taking about half an hour to reach a good policy on a GTX970 and quad core CPU pair (using demonstrations).

3 Future Work

On the creation of a Dataset future work should focus on developing datasets that are sourced from multiple annotators offering affect labels of subjective nature. While such a dataset would require a lot more resources to create, it would also produce an agent that would truly mimic human emotions. On a simpler note, the other three levels of Mario that were included in the dataset can also be studied.

On selecting games different games with a more emotionally charged content can be selected and different types of annotations can be used for each game. One can think of annotating a horror game, where the annotator notes how scared he is, and the agent that is produced can be timid or bold.

On studying algorithms the next endeavor should be on trying to integrate state-of-the-art On-Policy algorithms, such as Go Explore [4], so that the affective agent features performance comparable to those. The difficulty in integrating them is due to the affective demonstrations being essentially a replay buffer that was collected using a different policy (that of the annotator) and having an inability to collect more. One part that might help is that, unlike in this thesis, the final agent produced by training can have an emotional state baked into it (such as being a scared agent) without being customizable after the fact. Without this limitation, it could be easier to integrate newer algorithms.

On measuring performance the metrics presented in the affective Chapter 6 only measured one part of affective performance, imitation, and not how the agent will try to move towards states that exhibit arousal or how much arousal it will achieve. More innovative metrics should be developed, which are better suited to deal with this fact. The ideal metric to have would be the equivalent of Cumulative Reward in RL.

Bibliography

[1] Christian Becker, Stefan Kopp, and Ipke Wachsmuth. *Why emotions should be integrated into conversational agents*. Wiley, 2007.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Contributors to Wikimedia projects. PAD emotional state model - Wikipedia, Mar 2021. [Online; accessed 23. May 2021].

[4] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems, 2019.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-Learning. In *AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press, Feb 2016.

[6] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations, 2017.

[7] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents, 2020.

[8] David Melhart, Daniele Gravina, and Georgios N. Yannakakis. Moment-to-moment Engagement Prediction through the Eyes of the Observer: PUBG Streaming on Twitch. *arXiv*, Aug 2020.

[9] David Melhart, Antonios Liapis, and Georgios N. Yannakakis. PAGAN: Platform for Audiovisual General-purpose ANnotation. In *2019 8th International Conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW)*, pages 75–76. IEEE, Sep 2019.

[10] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937. PMLR, Jun 2016.

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

[12] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Emotion in Reinforcement Learning Agents and Robots: A Survey. *arXiv*, May 2017.

[13] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

[14] Charles Egerton Osgood, George J Suci, and Percy H Tannenbaum. *The measurement of meaning*. University of Illinois press, 1957.

[15] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Boosted Bellman Residual Minimization Handling Expert Demonstrations. In *Machine Learning and Knowledge Discovery in Databases*, pages 549–564. Springer, Berlin, Germany, Sep 2014.

[16] James A Russell. Evidence of convergent validity on the dimensions of affect. *Journal of personality and social psychology*, 36(10):1152, 1978.

[17] James A. Russell and Lisa Feldman Barrett. Core affect, prototypical emotional episodes, and other things called emotion: Dissecting the elephant. *J. Pers. Soc. Psychol.*, 76(5):805–19, Jun 1999.

[18] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv*, Nov 2015.