Bachelor's Thesis
Academic Year 2022

Modeling Head-Bobbing in Pigeon Locomotion using Reinforcement Learning

Faculty of Environment and Information Sciences,
Keio University

Mioto Takahashi

A Bachelor's Thesis
submitted to Faculty of Environment and Information Sciences, Keio University
in partial fulfillment of the requirements for the degree of
BACHELOR of Environment and Information Sciences

Mioto Takahashi

Thesis Committee:

Professor Tatsuya Hagino (Supervisor)
Professor Takashi Hattori (Co-Supervisor)

Abstract of Bachelor's Thesis of Academic Year 2022

Modeling Head-Bobbing in Pigeon Locomotion using Reinforcement Learning

Category: Science / Engineering

Summary

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In efficitur porta augue, at interdum nunc lobortis at. Morbi feugiat facilisis justo, vitae maximus dolor. Cras convallis at elit in porta. Fusce lobortis tortor nibh, quis imperdiet arcu luctus quis. Mauris imperdiet urna eu mauris aliquet, vitae tincidunt orci dapibus. Vestibulum convallis elit ut velit accumsan cursus. Pellentesque lacus lacus, blandit eu felis vitae, pellentesque dignissim est.

Keywords:

Reinforcement Learning, Biomimetic, Pigeon, Locomotion, OpenAI Gym

Faculty of Environment and Information Sciences, Keio University

Mioto Takahashi

Contents

List of Figures

Chapter 1

Introduction

Biomimetics is a research field that attempts to replicate biological phenomena, such as behaviors exhibited by animals, using methods seen in engineering. Models developed in biomimetics can be applied to developments of bio-inspired robotics, which can help solve difficult engineering problems similar to those solved by biological organisms. For example, engineers building control systems for bipedal locomotion robots can use proposed models of bipedal locomotion in bipedal organisms, since such organisms have already solved problems that would arise while attempting to build such control systems, such as preventing the body from falling over during the said locomotion.

Head-bobbing is a behavior unique to small birds, mainly pigeons, which consists of stabilizing their heads in a single location while occasionally altering it. Their abilities to lock their limbs may garner interest in the fields regarding camera stabilization due to their similarities in behavior. In such cases, physics-based models of pigeons that can reproduce such behavior would translate well to engineering.

Preliminary research have proposed possible functionalities behind such behavior, mainly consisting of gathering information of surrounding objects using the retina. However, to our knowledge, no physics-based model capable of replicating the behavior have been proposed. We attempt to make contribution in this aspect in our research.

Our goal in this research is to test such preliminary hypotheses in a physics engine and evaluate their sufficiency in reproducing the head-bobbing behavior.

We conduct such experiment by modeling control systems regarding head-bobbing in pigeons and their morphology, specifically their upper torso, neck, and head. We use reinforcement learning for modeling the controller of the pigeon model, because it allows us to construct control systems that represent preliminary hypotheses.

The following will be discussed in the succeeding chapters.

Chapter 2 Background knowledge of reinforcement learning necessary for understanding our research

Chapter 3 Preliminary research regarding the use of robotics for testing and proposing hypotheses in biology and the functionalities of head-bobbing in pigeons

Chapter 4 Recapitulation of our goals in this research and the methods that we use for conducting it

Chapter 5 Details regarding the experiments that we conducted

Chapter 6 Results of the aforementioned experiments

Chapter 7 Analysis of the results and proposals for improvement of our model in future research

Chapter 8 Summary of the thesis and conclusion to our research

Chapter 2

Background: Reinforcement Learning

Reinforcement learning is a type of control system that attempts to execute tasks described by manually set cost functions by minimizing them using optimization algorithms or learning algorithms. In the case of deep reinforcement learning, the controller is modeled using deep neural networks and the cost function is minimized using gradient descent algorithms.

Reinforcement learning divides control systems into an agent and an environment. The agent acts the controller of the system that inspects its environment's state and sends output signals, or actions, that affect the environment. This mutual interactions creates a feedback loop between the two modules.

In the context of a pigeon tasked to move forward, the pigeon's brain and its nervous system connected to each limb act as the agent, and its surroundings, such as the ground and arbitrary objects on it, act as the environment. The environment outputs a state, such as the global position of the pigeon, which is used as the input for the agent. Using the provided state, the agent calculates and outputs an action, such as the torque of each joint in the pigeon's body. The action alters the state of the environment, and the environment outputs a new state and a reward. The reward describes how well the pigeon was able to execute its task, such as the current position of the pigeon relative to its previous timestep. The agent is updated to output sequences of actions that maximizes the cumulative reward, or return. The return can be interpreted as the inverted

or negated cost function.

Deep reinforcement learning algorithms can train deep neural network controllers to maximize cumulative sums of arbitrarily-defined reward functions, as long as they are achievable within the given environment. Therefore, in application to biology, we can encode constraints and definitions of tasks, such as behaviors produced by biological organisms, into reward functions, train control systems to follow such constraints and tasks, and generate behaviors accordingly, while abiding by the laws of physics.

Notable deep reinforcement learning algorithms include proximal policy optimization (PPO) [12] and soft actor critic (SAC) [9]. PPO is used as baseline for many reinforcement learning experiments such as building agents to play games, particularly Pommerman [6], controlling automated vehicles [8], and controlling unmanned aircrafts [1]. On the other hand, SAC has been shown to perform better than PPO in benchmark tests and has been applied to biology-inspired robotics, particularly quadrupedal robotic control [10].

Chapter 3

Preliminary Research

3.1. Modeling Biological Phenomena using Robotics

Webb discusses the topic of utilizing robotics to aid research in biology [14], where he argues that robotic modeling can be used for proposing or testing hypotheses. This can be accomplished by depicting the hypotheses in the form of algorithms or hardware configurations in robots and observing their emergent behaviors. The generated behaviors can then be compared with their biological counterparts. In particular, robotic models implemented based on preliminary hypotheses can be used as null hypotheses, which can be validated after observing behaviors exhibited by them.

While Webb argues about the potential of using robotics for research in biology, he also highlights the limitations of such methods. Since robotic models naturally contain information on all of the assumptions made on the hypotheses of the biological phenomena they reference, discrepancies between the model and the phenomena must be carefully taken into consideration during experiments. For example, if a bipedal robot were to be used to model a pigeon's forward locomotion, it should account for noises and disturbances in the pigeon's sensory input signals and neurological output signals that stimulate their muscles. Additionally, environmental factors such as rough terrain that pigeons walk on should be simulated in the experiment.

From such perspective algorithmic models are inferior to models that utilize real robots since, as Webb states, they cannot confront problems seen in physical

environments, such as noises or disturbances, that are not seen in virtual environments, such as physics engines. As such, algorithms in biology are often used for modeling neurological functionalities. In the context of modeling pigeons' locomotion, algorithmic models can represent the neurological controllers for the pigeons. In this case, the pigeons' interaction between their brains, their morphology, and their surrounding environments can be seen as a control system.

Webb identifies incremental modeling as a method to tackle the discrepancies between complexity of biological phenomena and their model counterparts. Incremental modeling builds multiple models for the same hypotheses, where each model gets more complex with biological or physical details.

A biological model of a pigeon with high biological and physical details would consist of the musculoskeletal mechanism of pigeons, a spiking neural network to control all of the muscles, and a surrounding environment with complex details, such as bumpy terrain and colors. This would be too complicated of a problem to tackle in a single research paper, especially considering that the learning and control mechanism of spiking neural networks is currently unclear. Even if the musculoskeletal model were to be controlled by a fully connected deep neural network, the biological and environmental details, including the colors and the sheer number of controllable muscles, would heighten the input and output dimension of the neural network, making training it using deep reinforcement learning extremely difficult.

Our research specifically focuses on a simplified model of the pigeon, which includes simplified representations of limb control and retinal input.

3.2. Head-Bobbing in Pigeons

The "head-bob" behavior in pigeons consists of stabilizing the global location and orientation of the head and altering them periodically. Such are dubbed as the hold phase and the thrust phase, respectively [3].

Frost and Davies' have proposed hypotheses regarding the functionalities of such behavior have been proposed [5] [3]. Both proposals highlight the use of the hold phase as a means to stabilize vision and the use of the thrust phase as a means to detect motion parallax and determine the distance between objects.

3.2.1 Hold Phase

Frost's hypothesis links the functionality of the hold phase to the detection of backward motion within the eye [5]. Pigeons' heads, while flying or moving forward, would be detecting objects whose movements can be distinguished from the surrounding stationary objects. Since stationary objects would be moving backwards relative to the pigeons' eyes, desensitizing backward motion would be necessary for such distinctions to be detected. However, this desensitization would be detrimental to the pigeons' object recognition while the pigeons' heads are stationary. The hypothesis highlights the existence of "backward notch" cells which counteract the aforementioned desensitization. Such cells would be activated when the pigeons' vision is stabilized and allowing them to distinguish objects moving backward relative to stationary objects, hence the necessity of the hold phase during locomotion.

Davies' hypothesis challenges this notion and highlights the lack of necessary conditions in Frost's hypothesis to induce a hold phase by stating that "they would fail to detect objects moving backwards through the visual field at velocities similar to that of the bird, as their responses could not be discriminated from those caused by self-induced motion" [3]. Davies proposes the existence of cells that detect objects' movement relative to stationary backgrounds regardless of their directions.

In the context of our model, combining the two hypotheses leads to a mechanism that stabilizes the head of the pigeon relative to arbitrary stationary objects and activate cells that detect arbitrary motion during the hold phase.

3.2.2 Thrust Phase

Frost proposes a hypothesis which links the behavior of thrusting the head forward to depth perception [5]. He takes the two most common methods of depth perception in biology, stereopsis and motion parallax, into consideration, and points out that, upon examining the anatomy of birds' heads, the eyes are seen to be placed on the opposite sides of the skull instead of having both of them be placed on the frontal area. Such positioning of the eyes in pigeons implies that the overlapping areas in visions of both eyes are reduced compared to animals that

mainly use stereopsis for identifying depth of objects. Given such observation, he argues that pigeons are more likely to resort to utilizing motion parallax over the alternative, given the lateral eye placement in pigeons' heads.

Davies later elaborates on the hypothesis by modeling the retinal information of objects [3].

Davies models the depth perception caused by motion parallax as

$$\dot{\theta} = \frac{v \sin \theta}{x} \tag{3.1}$$

where θ is the angle of the object relative to the eye, v is the velocity of the pigeon's head, and x is the distance between the object and the head, or the depth of the object. The equation indicates that the depth of the object can be derived if the velocity of the head and the angular velocity of the object within the retina are given.

Davies further extends this model to account for detecting the difference in depth between objects. Davies differentiates the angular velocity of a single object and the relative angular velocity between an object at depth x and an object at depth y with the notation Δ.

$$\Delta\dot{\theta} = v \sin \theta \left(\frac{1}{x} - \frac{1}{y} \right) \tag{3.2}$$

Given the two equations, Davies argues that maximizing the speed of head would result in better differentiation between objects with different depths. By thrusting the head forward, the pigeon would be able to detect stationary objects in different depths, which cannot be detected during the hold phase. As an example Davies mentions the role of the eye in detection of food, such as the detection of bread crumbs lying on the ground close by as opposed to lampposts seen far away.

3.2.3 Regarding Kinematic Functionalities

When building our pigeon model, in addition to the hypotheses proposed regarding the hold phase and the thrust phase, we must take the effect of the torques of the neck joints and the movement of the head generated by them. Intuitively

such motion would alter the balance of the entire pigeon, leading to mutual adjustments between the bipedal walk cycle and the neck control for head positioning. Additionally, the head-bobbing motion could be hypothesized to function as a means to balance the pigeon's forward locomotion. However as Davies argues in his paper [3], since head-bobbing is not exhibited during fast forward locomotion, such as flying, it is unlikely that such behavior has kinematic purposes. Frost's findings [5] further support this idea by demonstrating that pigeons stabilize their head in one global coordinate regardless of the body's global velocity, it is likely that head-bobbing's functionalities are solely based on vision.

Chapter 4

Approach

Our research purpose is to verify whether preliminary hypotheses regarding the functionalities of head-bobbing, particularly visual stabilization and motion parallax, are sufficient to cause such behavior. We would like to examine whether there are possibilities of other causes or functionalities that may contribute to generating such particular movements.

4.1. Definition of the Pigeon Model

We define a simplified 2-dimensional model of pigeons based on incremental modeling. The pigeon model consists of 3 joints connecting one body representing the head, 2 bodies representing the neck, and one body representing the torso. The model's physics, mainly the collision and gravity, is simulated in a 2 dimensional physics engine. The torso's orientation and y-position is fixed while its x-position is incremented by a constant value. This represents forward locomotion at a constant speed.

Additionally, we build control systems for the model using deep reinforcement learning. By using deep reinforcement learning, we can train the controller to maximize reward functions that represent hypotheses or manually-defined trajectories for the bodies in the model to follow.

4.2. Baseline: Manually-Defined Head Trajectory

As the baseline for the model's control system, we attempt to recreate the head-bob movement by setting a target position for the head's position to match every timestep. The target position is first defined at a set location in front of the pigeon model T relative to the position of its torso. The target then acts as a static position in the global coordinate for the head to follow. If the distance between the target position and the torso's position goes below a set threshold value, the target is repositioned at the same location T relative to the torso's position.

4.3. Hypothesis Testing

For verifying the preliminary hypotheses we compare the behaviors of the pigeon model produced by the baseline control system to those generated by the control system that represents preliminary hypotheses.

Preliminary hypotheses for the functionalities of head-bobbing behavior can be depicted using two reward functions, each representing head stabilization $r_{head\_stabilize}$ and motion parallax $r_{motion\_parallax}$.

The reward function that represents the hypothesis is described as below.

$$r_{fifty\_fifty_t} = r_{head\_stabilizet} + r_{motion\_parallax_t} \tag{4.1}$$

where r_t is the reward at timestep t.

4.3.1 Head Stabilization

Davies' hypothesis indicate that static objects should be stabilized into one location in the retina for the pigeon to easily determine the moving objects' velocities during the hold phase. In application, the pigeon's head should move in a trajectory that minimizes retinal velocities of objects.

We define the reward function for head stabilization as,

$$r_{head\_stabilizet} = -\sum_{i}^{n} |\dot{\theta}_{it}| \tag{4.2}$$

where n is the number of objects in the environment and $\dot{\theta}$ is the angular velocity of each object.

4.3.2 Motion Parallax

Frost and Davies's hypotheses indicate that differences in distances between objects from the pigeon's retina can be emphasized by inducing motion parallax. The sum of angular velocities of objects relative to each other should be maximized.

We define the reward function for motion parallax as follows.

$$r_{motion\_parallax_t} = \sum_{i}^{n} \sum_{j \neq i}^{n} |\dot{\theta}_{it} - \dot{\theta}_{jt}| \tag{4.3}$$

4.4. Additional Notable Questions

Given our approach, we can see that combining the reward functions for head stabilization and motion parallax result in a tradeoff between having to lock the position of the head in one place and having to move the head around. Therefore, two more questions need to be addressed: Can hold and thrust phases emerge given such tradeoff? How would optimization algorithms solve this tradeoff?

If hold and thrust phases were not to emerge, we can deduce that factors aside from the two functionalities are essential to the generation of such behaviors. As mentioned before, we are comparing the differences between trajectories of heads in pigeon models whose controllers are trained by the baseline reward function and those trained by the reward function representing the preliminary hypotheses. By utilizing such analysis, we can differentiate functionalities missing in preliminary hypotheses and propose novel hypotheses that incorporate them. Additionally, by observing the trajectories of heads, bodies, and limbs in the pigeon models generated by controllers trained using $r_{fifty\_fifty_t}$, we can gain explanations regarding possible efficient solutions that maximize the usage of both head stabilization and motion parallax that are not seen in real pigeons.

Chapter 5

Experiments

We compare the trajectories of the bodies in the baseline control systems to those generated by control systems that represent preliminary hypotheses. By examining the similarities and differences between the trajectories, we can verify whether the preliminary hypotheses are sufficient for producing the head-bobbing behaviors seen in pigeons.

5.1. Dimensions of the Pigeon Model

Our pigeon model's dimensions and orientations are set at static values for all experiments, as shown in Figure 5.1.

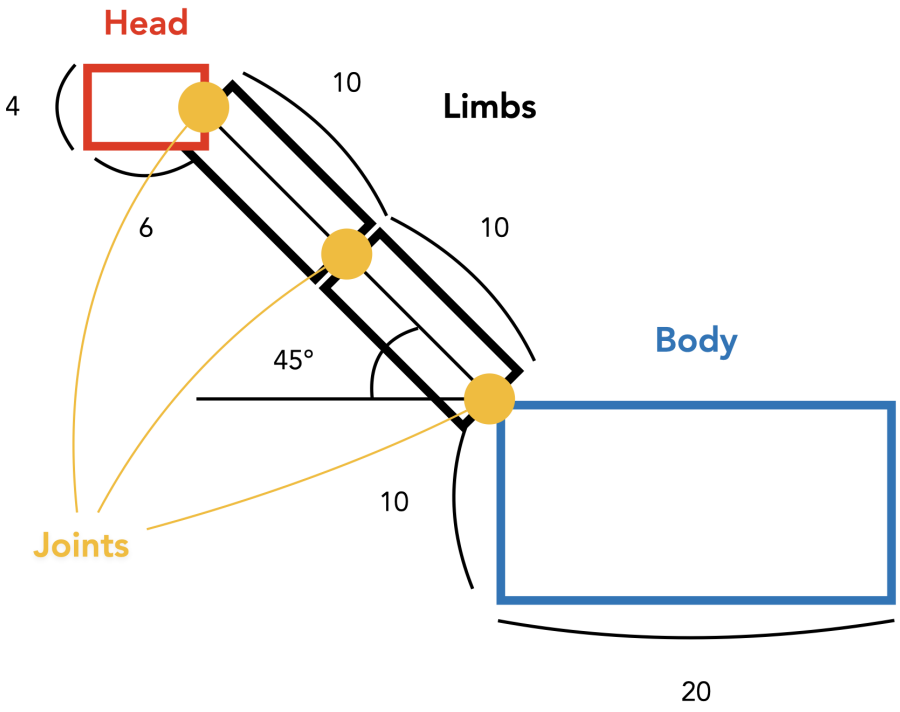

Figure 5.1: Diagram of the pigeon model

Additionally, the pigeon's head relative to the body is facing the negative direction relative to the x-axis. The widths and heights of each limb, head, and body are $(10, 4)$, $(6, 4)$, $(20, 10)$ respectively. The initial angles of each limb are oriented at 45 degrees relative to the x-axis, and both the head and the body are oriented parallel to the x axis. The body's initial position is at the origin, and is set to move at a constant speed in the negative direction along the x-axis.

5.2. Experiment Environments

We trained a total of 5 reinforcement learning agents to control the pigeon model: 1 agent for the case where the model is given the baseline task given an unmoving body, 2 agents for cases where the model is given the same task with the body speed of 1, 2 agents for cases where the model is tasked to move its

joints to maximize the reward function r_{fifty_fifty}, as defined in Chapter 4, under the body speed of 0 and 1 respectively.

Several additional variables are defined for the pigeon to execute the baseline task as shown in Figure 5.2.

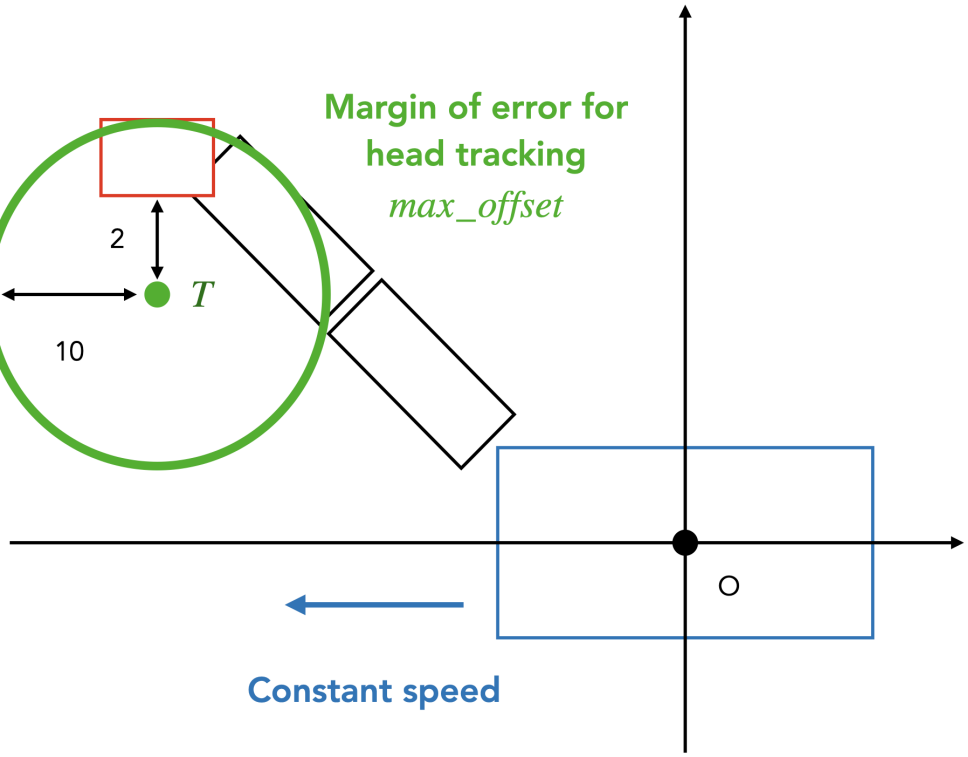

Figure 5.2: Head trajectory tracking

T is set at $(0, -2)$ relative to the initial position of the head. We set the threshold value for the distance between the target position and the torso's position to 10. As mentioned in Chapter 4 when the the said distance is below the threshold value, T is reset to be the same position relative to the body as its initial position. We set a value max_offset that represents the margin of error between T and the position of the head.

For the case with an unmoving body, we define a function that generates positive rewards only for timesteps where the distance between the head and T is

within $max\_offset = 0.5$. Each reward is bounded within $[0, 1]$ and scaled based on the level of alignment to the x-axis.

$$r_{head\_stable\_manual\_reposition\_strict\_angle_t} = \begin{cases} 1 - \frac{\alpha_t}{\pi} & \text{if } \alpha_t < \frac{\pi}{6} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where α_t is the angle of the head at timestep t.

For the speed of 1, unlike the case with the unmoving body, $max\_offset = 1$. Additionally, alongside $r_{head\_stable\_manual\_reposition\_strict\_angle}$, we define a looser reward function that generates positive rewards as long as the head is within the set margin of error around the target location. It is expected that this function would serve as an alternate less strict to the former reward function that produce similar behaviors.

$$r_{head\_stable\_manual\_reposition_t} = r_{head\_stable\_manual\_reposition\_strict\_angle_t} + \begin{cases} 1 - \frac{\delta_t}{max\_offset} & \delta_t < max\_offset \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

where δ_t is the distance between the head and T at timestep t.

Preliminary hypotheses regarding retinal stabilization and motion parallax are depicted as the reward function r_{fifty_fifty}. The reward function is used to train agents that represent behaviors derived from such hypotheses under the conditions of both speeds 0 and 1. For both cases, $max\_offset = 1$. 3 points and their positions are defined to represent 1 static and 2 dynamic objects placed on the surrounding environment of the pigeon. The static object's position is $(-30.0, 30.0)$, and the 2 dynamic objects' positions are $(-30.0, 60.0)$, $(-60.0, 30.0)$. The former dynamic object moves at speed 1 in the positive direction along the x-axis, while the latter moves at the same speed in the negative direction along the x-axis.

We constructed OpenAI Gym [2] environments `PigeonEnv3Joints` and `PigeonRetinalEnv` for conducting reinforcement learning based on the baseline and preliminary hypotheses, respectively. Details regarding the environments' code are in Appendix A.

5.3. Reinforcement Learning

We used SAC to conduct batch training on each deep neural network agent for 3000 epochs. Each deep neural network has one hidden layer containing 256 neurons. Details regarding more rigorous hyperparameter settings are in Appendix B.

PPO, despite being used often as baseline for many reinforcement learning experiments as we have stated beforehand, was not used for training any of the 5 aforementioned controllers. When we attempted to train deep neural network controllers for pigeon models with static bodies using PPO and SAC with $r_{head\_stable\_manual\_reposition}$, we found that the agent trained on SAC had a more stable learning curve and a faster convergence rate than the agent trained on PPO. Therefore, we determined that it would be more reliable to use SAC to obtain the desired results.

Chapter 6

Results

6.1. Summary

We summarize the behaviors exhibited by the pigeon models in each experiment, categorized by their body speeds and the reward functions the controllers were trained with.

- Body speed 0; $r_{head\_stable\_manual\_reposition\_strict\_angle}$; $max\_offset = 0.5$

  The pigeon model managed to keep its head stationary as expected.

- Body speed 1; $r_{head\_stable\_manual\_reposition\_strict\_angle}$; $max\_offset = 1.0$

  The pigeon model initially struggled to control its head in a specific pattern or position. However, after 5 seconds, it managed to align the head within the radius max_offset around the manually set position T, resulting in a generation of trajectory that depicts a pattern of thrust and hold phases. The final 5 seconds of the footage depicts the head and the limbs getting stuck on the topside of the body.

- Body speed 1; $r_{head\_stable\_manual\_reposition}$; $max\_offset = 1.0$

  The pigeon model produced similar results to the previous experiment, with the exception of the overall performance. Due to the less strict definition of the reward function, it generated a trajectory that resulted in the acquirement of larger return, and as a result, a better performance in producing the desired head-bobbing behavior. In particular, the timespan of the head following the manually-defined trajectory was longer than those seen in the previous experiment.

- Body speed 0; r_{fifty_fifty}

  The head of the model moved around in the vicinity of a single position, similar to the baseline counterpart.

- Body speed 1; r_{fifty_fifty}

  The head of the model gradually leaned down and backwards as if it were following the surrounding stationary object.

6.1.1 Learning Curves of Policies Controlling the Pigeon Models

We examine the learning curves of the returns acquired by the policies trained for each experiment. Figure 6.1, 6.2, 6.3, 6.4, and 6.5 present the exploration and evaluation returns, where the exploration returns are returns acquired by policies that output stochastic distributions of actions while the evaluation returns are such acquired by policies that output deterministic values for actions.

Examining at the figures, all policies, or in this case the controllers for the pigeon models, converge to a return in the span of 3000 epochs. Regarding the

controllers trained to follow manually-defined trajectories on a pigeon model with the body speed of 1, it is shown that more return are acquired for policy trained on $r_{head\_stable\_manual\_reposition}$ than $r_{head\_stable\_manual\_reposition\_strict\_angle}$. It can therefore be concluded that it is more reliable to use the trajectory produced by the controller trained on the former reward function than the latter reward function.

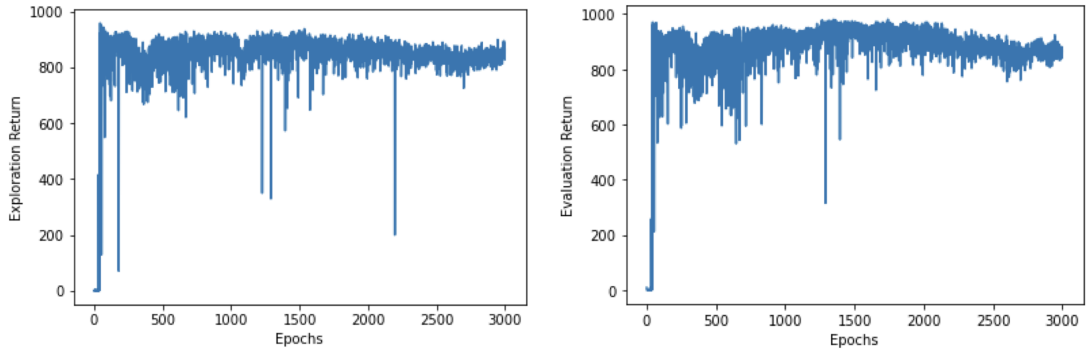

Figure 6.1: Learning curve for returns of the policy trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $body\_speed = 0$ and $max\_offset = 0.5$

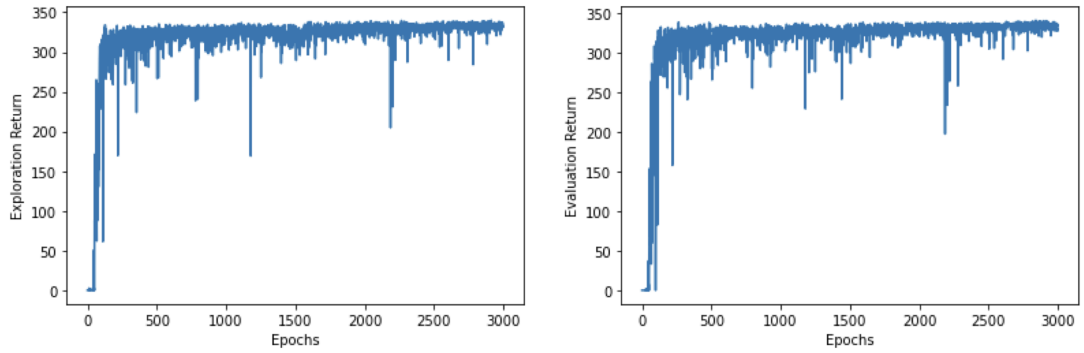

Figure 6.2: Learning curve for returns of the policy trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $body\_speed = 1$ and $max\_offset = 1.0$

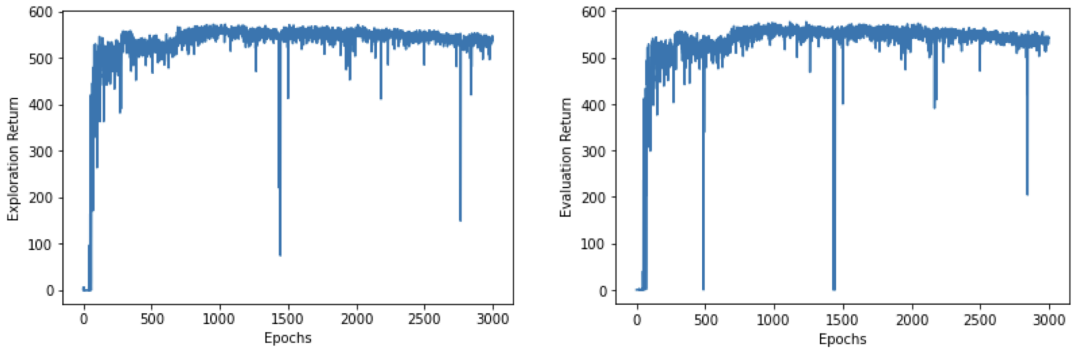

Figure 6.3: Learning curve for returns of the policy trained on $r_{head\_stable\_manual\_reposition}$ with $body\_speed = 1$ and $max\_offset = 1.0$

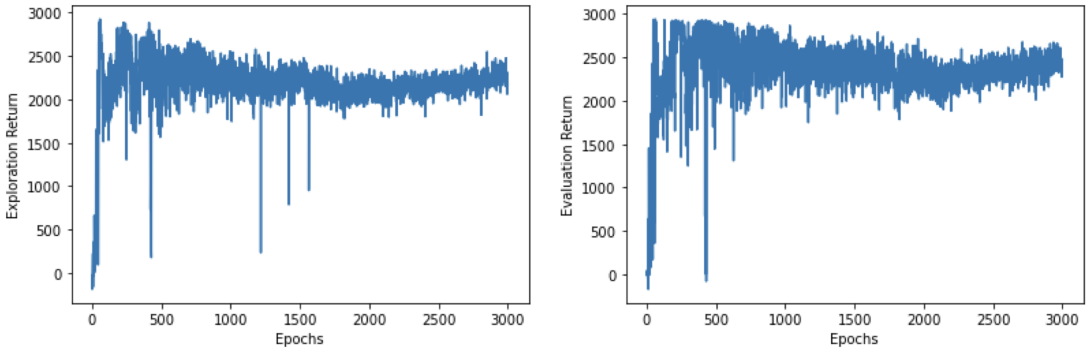

Figure 6.4: Learning curve for returns of the policy trained on r_{fifty_fifty} with $body\_speed = 0$

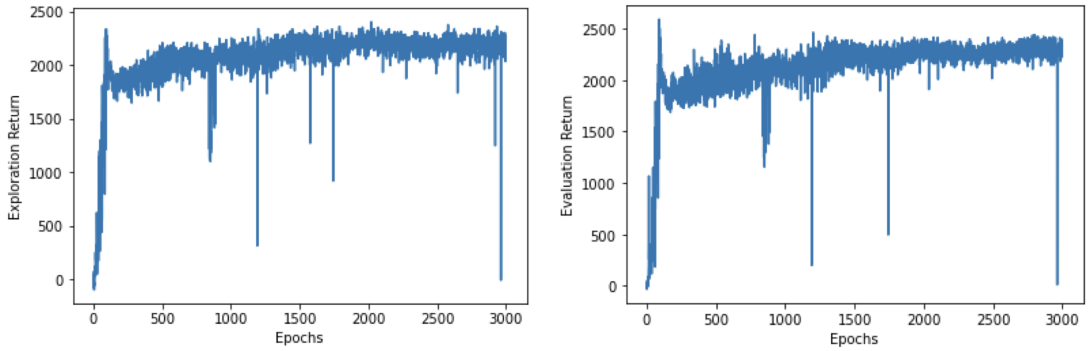

Figure 6.5: Learning curve for returns of the policy trained on r_{fifty_fifty} with $body\_speed = 1$

6.2. Behavior Renderings

We rendered the resulting behaviors produced by controllers trained on aforementioned reward functions and environments into images or frames. Combining the frames generated for each of the 1000 timesteps and setting as 60 frames per second resulted in 33.35 second videos. The time-lapses presented in Figure 6.6, 6.7, 6.8, 6.9, and 6.10 were created by sampling every 300 frames within the last 30 seconds of the video. The frames' sequential order is from the top left to the bottom right. The camera is locked to follow the pigeon's body.

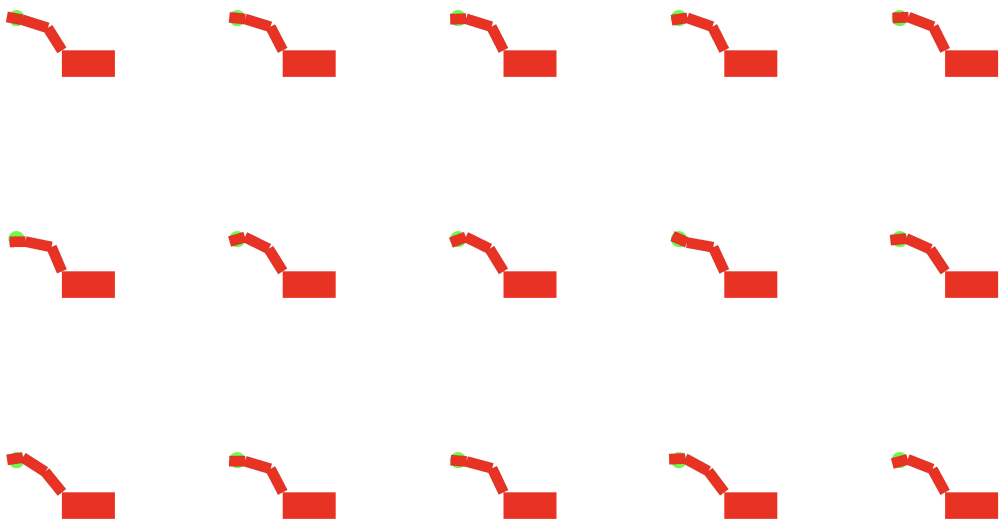

Figure 6.6: Control of a pigeon model with a static body trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $max\_offset = 0.5$. The green circle indicates the margin of error around the target head location defined by max_offset.

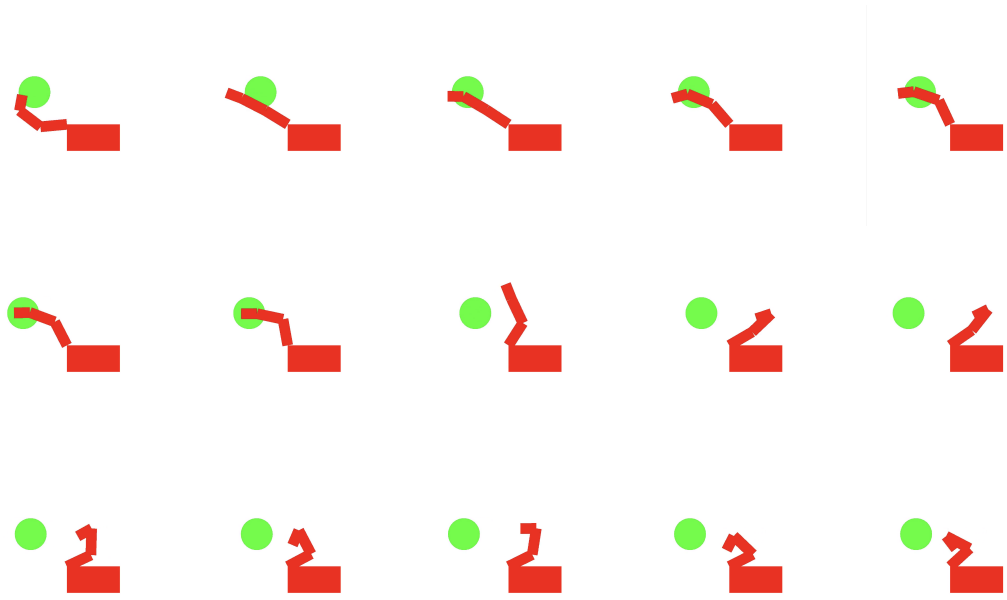

Figure 6.7: Control of a pigeon model with the body speed of 1 trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $max\_offset = 1.0$. The green circle indicates the margin of error around the target head location defined by max_offset.

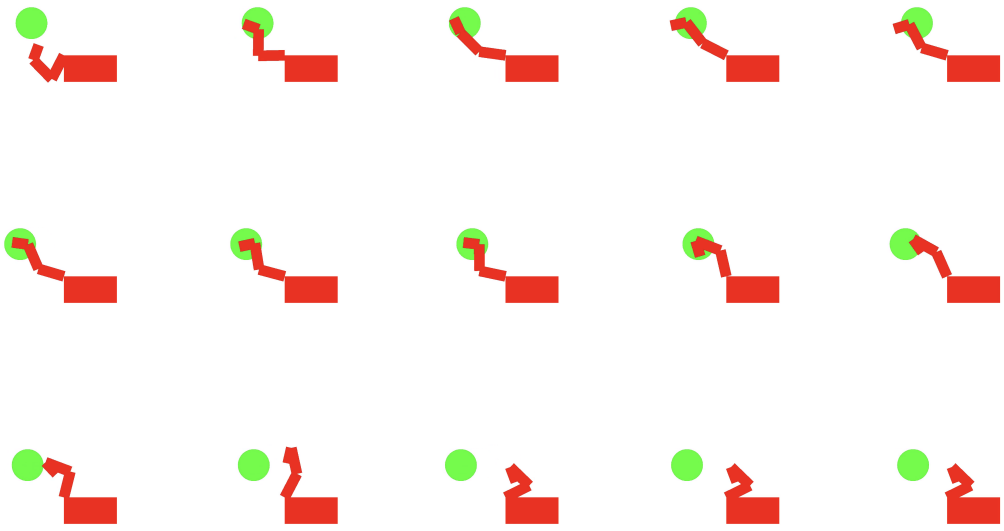

Figure 6.8: Control of a pigeon model with the body speed of 1 trained on $r_{head\_stable\_manual\_reposition}$ with $max\_offset = 1.0$. The green circle indicates the margin of error around the target head location defined by max_offset.

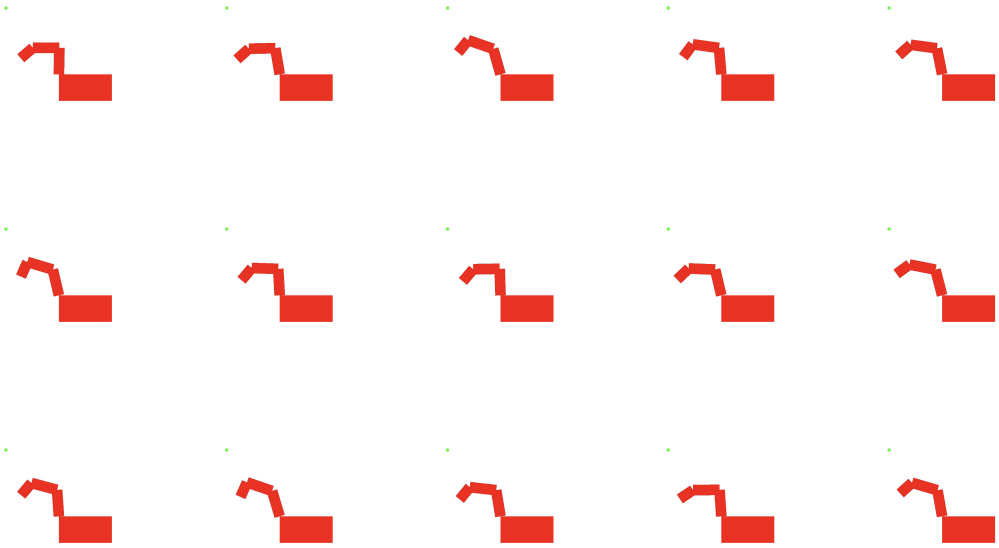

Figure 6.9: Control of a pigeon model with a static body trained on r_{fifty_fifty}

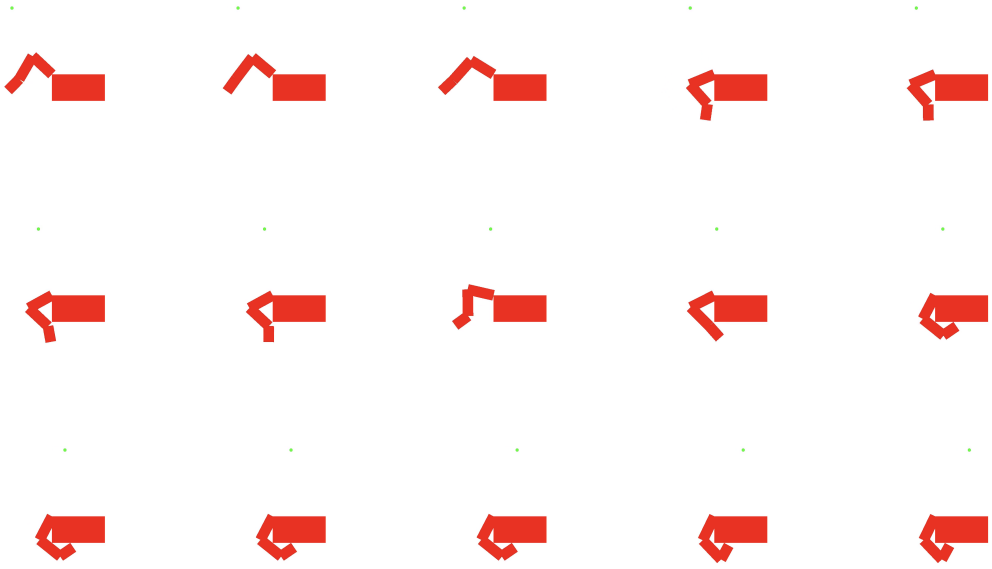

Figure 6.10: Control of a pigeon model with the body speed of 1 trained on r_{fifty_fifty}

6.3. Head Trajectories

We additionally visualized the trajectories of heads and bodies of each pigeon model. With this method, we can compare the resulting behaviors of the controlled pigeons between the reward functions they were trained on.

We coupled the trajectories produced by pigeons with the same body speeds in Figure 6.11 and 6.12.

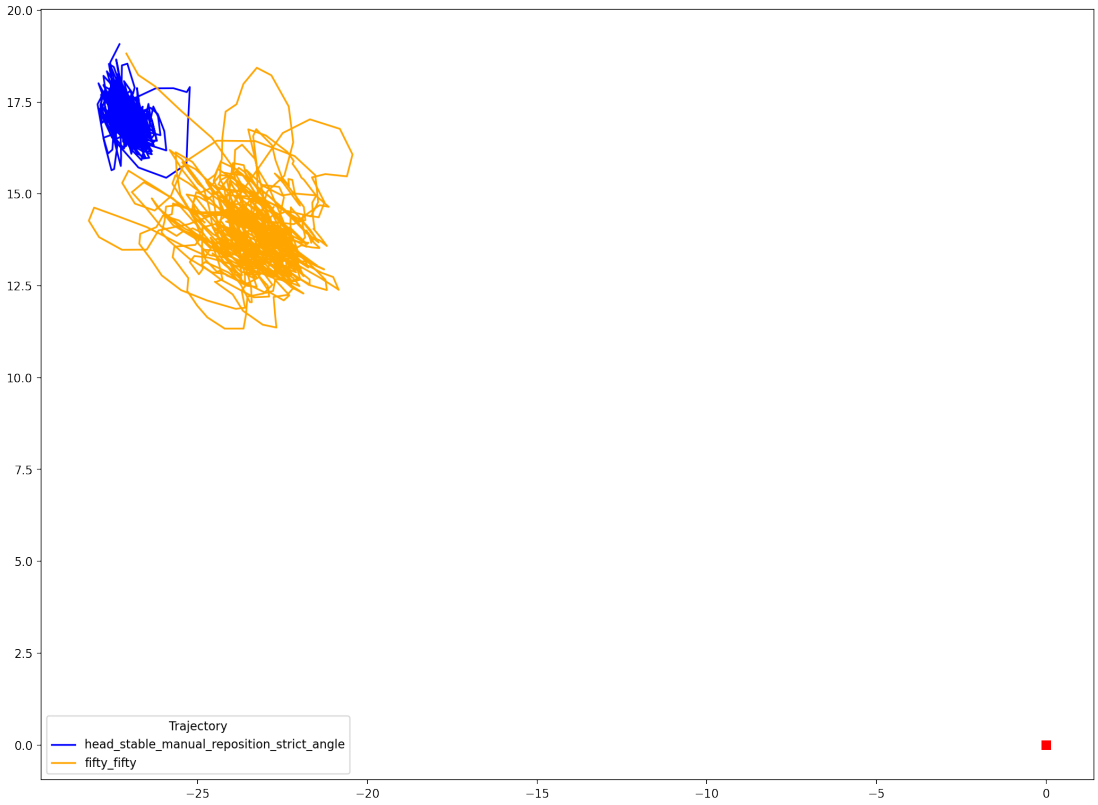

Figure 6.11: Trajectories of heads and bodies of pigeon models with static bodies trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $max\_offset = 0.5$ and r_{fifty_fifty}

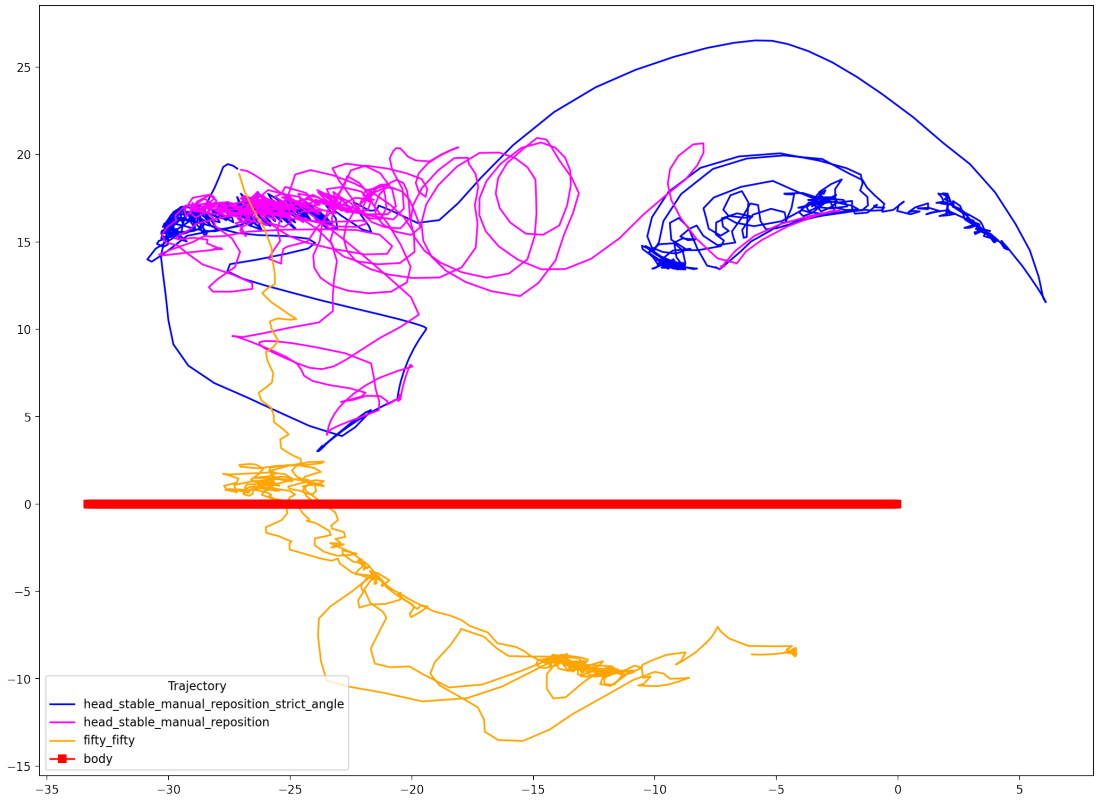

Figure 6.12: Trajectories of heads and bodies of pigeon models with body speeds of 1 trained on $r_{head\_stable\_manual\_reposition\_strict\_angle}$ with $max\_offset = 1.0$, $r_{head\_stable\_manual\_reposition}$ with $max\_offset = 1.0$, and r_{fifty_fifty}

For pigeon models with static bodies, the manually-defined trajectory set by $r_{head\_stable\_manual\_reposition\_strict\_angle}$ and the trajectory that reflects the preliminary hypotheses set by r_{fifty_fifty} are similar: they both attempt to stabilize the head in one location. The trajectory produced by the controller trained on r_{fifty_fifty}, albeit with more variance, has such trait as an emergent behavior.

For pigeon models with body speeds of 1, as mentioned beforehand, pigeon models whose controllers were trained using $r_{head\_stable\_manual\_reposition\_strict\_angle}$ and $r_{head\_stable\_manual\_reposition}$ struggle to maintain their heads to follow their target locations T. The manually-defined trajectories set by the two reward functions reflect such behaviors.

Taking these issues into consideration, we present an alternative visualization of the trajectories Figure 6.13, where the trajectory representing $r_{head\_stable\_manual\_reposition\_strict\_angle}$

is trimmed to only show from 250 to 500 timesteps and the trajectory representing $r_{head\_stable\_manual\_reposition}$ is trimmed to only show from 300 to 500 timesteps. The two manually-defined trajectories now share similar paths of head-bobbing; however even with such adjustments, it is evident that the trajectory produced by the controller trained on r_{fifty_fifty} is vastly different from the 2 prior trajectories.

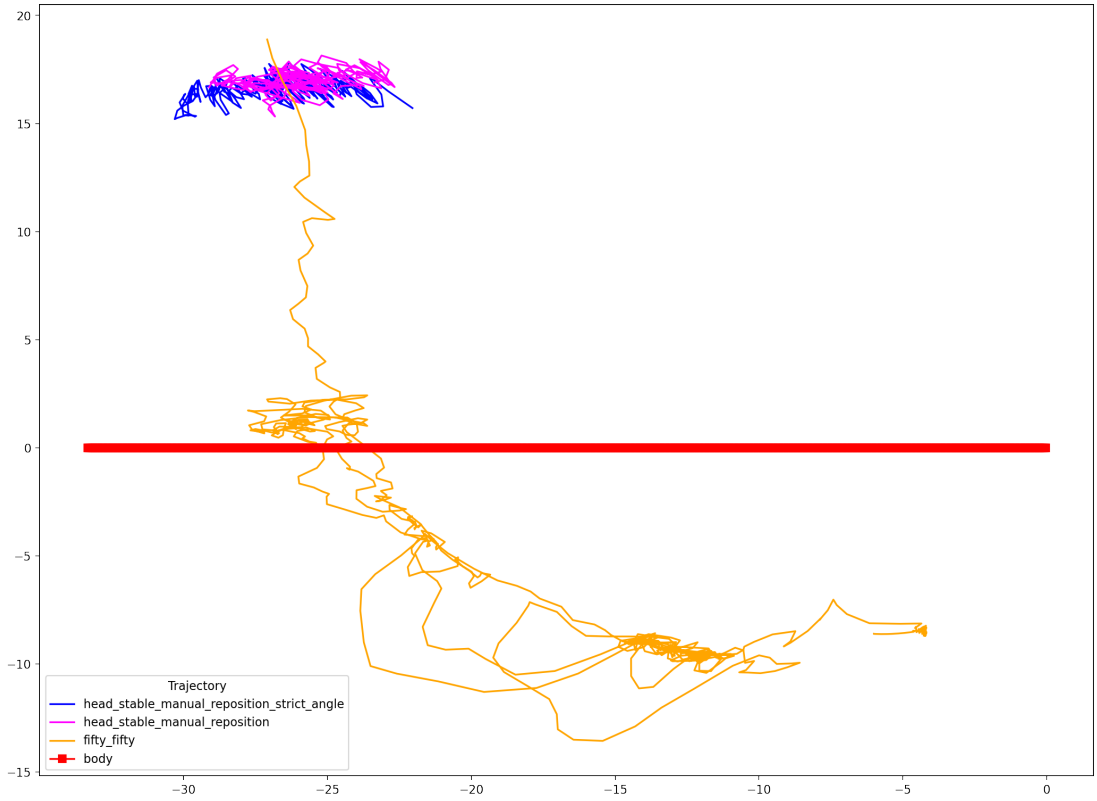

Figure 6.13: An altered Figure 6.12, where trajectories representing $r_{head\_stable\_manual\_reposition\_strict\_angle}$ and $r_{head\_stable\_manual\_reposition}$ are trimmed to only show their head-bobbing behaviors

Since we have already determined that the controller trained on $r_{head\_stable\_manual\_reposition}$ produces better results than the alternative, we can additionally refer to Figure 6.14 and reach the same conclusion.

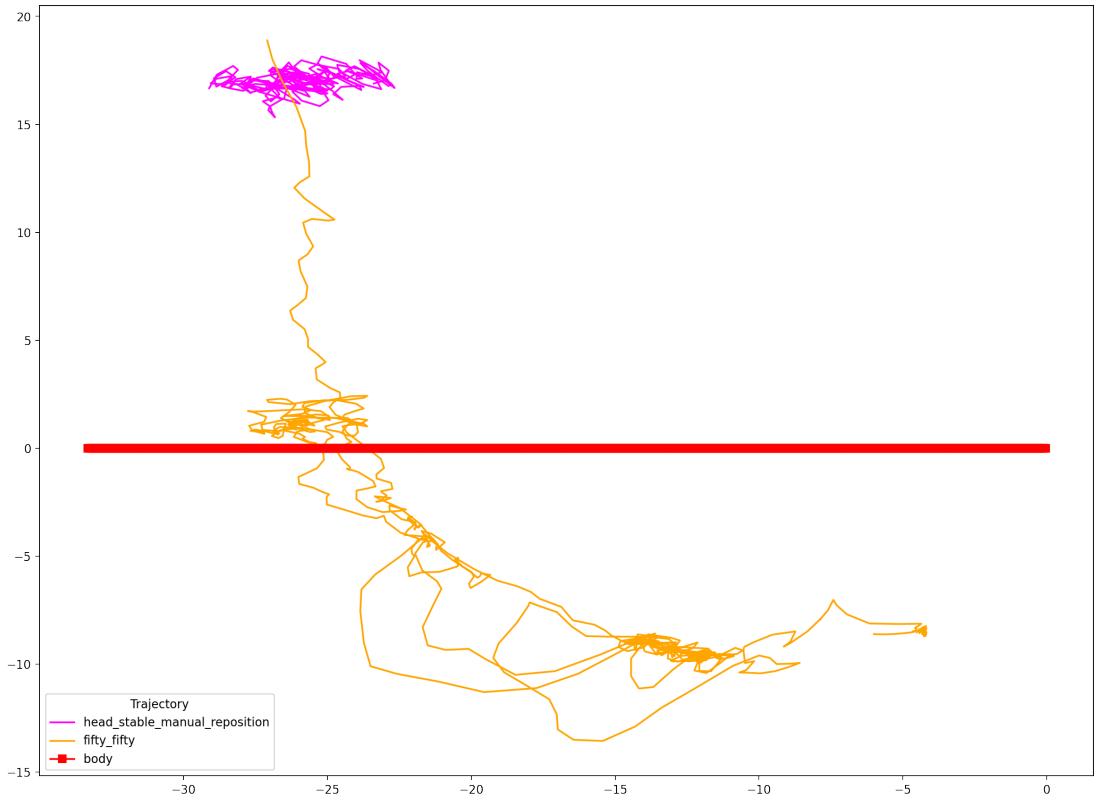

Figure 6.14: Same as Figure 6.13, except the trajectory representing $r_{head\_stable\_manual\_reposition\_strict\_angle}$ is removed to only show the trajectory representing $r_{head\_stable\_manual\_reposition}$ as baseline

Chapter 7

Discussion

Examining the trajectories of each of the pigeon models' heads as seen in Figure 6.11, the resulting behavior exhibited by the pigeon model with a fixed body, whose deep reinforcement learning controller was trained on r_{fifty_fifty}, indicate that the combination of visual stabilization and motion parallax are sufficient to generate a behavior that fixes the position of the head, resembling the hold phase in pigeons.

On the other hand, the resulting behavior exhibited by the pigeon model with body speed 1, whose deep reinforcement learning controller was trained on the same reward function as seen in Figures 6.13 and 6.14, indicate that the 2 functionalities are insufficient to produce head-bobbing behaviors in moving bodies during forward locomotion, as he pigeon model did not need to periodically thrust its head forward to maximize Davies' equation depicting motion parallax between objects.

Such results indicate that visual stabilization with retinal cells capable of detecting movement in all directions is enough to maximize the sum of external objects' angular velocities within the retina. Examining Davies' equation 3.2, it can be hypothesized that reinforcement learning controllers trained on reward function that reflect such function would only reproduce head-bobbing behaviors when all objects within the retina are globally static, since the equation does not account for the objects' velocities. Since the external objects in our experiment had objects moving forwards and backwards, such may have automatically increased the reward function depicting Davies' equation of motion parallax per

timestep.

One possible argument against our claim can be pointed to our decision to weigh $r_{head\_stabilize}$ and $r_{motion\_parallax}$ equally for the reward that the policy or controller representing preliminary hypotheses r_{fifty_fifty}. One could argue that by changing the weights, it may be possible to produce different behaviors than those shown in the results. Specifically, by assigning a larger weight value for $r_{head\_stabilize}$ than that for $r_{motion\_parallax}$, one could expect a pigeon model with the body speed of 1 to reflect the task of constricting the global position of its head and produce hold phases while occasionally moving the head to induce motion parallax. However, considering that the maximum return for $r_{head\_stabilize}$ is 0 and the pigeon whose controller was trained on r_{fifty_fifty} still managed to produce high positive returns (Figure 6.5) , it can be inferred that penalizing movement of the head would not change the resulting behavior. Therefore, factors other than static visual stabilization $r_{head\_stabilize}$ and motion parallax induction $r_{motion\_parallax}$ are most likely also contributing to the induction of head-bobbing behaviors.

We examine the biggest behavioral difference between pigeon models in the baseline experiments and their counterparts reflecting preliminary hypotheses, particularly, those trained on r_{fifty_fifty} with the body speed of 1. The latter example, unlike the former, presents a behavior where the head is moved downwards throughout the duration of the experiment (Figure 6.10). Despite the major differences between the two experiments, the latter does not suffer from an inability to acquire rewards (Figure 6.5). Considering that constantly bending one's neck downwards outside of its neutral posture would likely result in physical trauma for pigeons, it is possible to suspect that muscular strain may be a motivation to avoid reproducing the behavior seen in our experiment with r_{fifty_fifty} and adjust their neck posture to maintain an upright position. Additionally, muscular simulation and their placements within the physical model may lead to more stabilization in its movements, as seen in Geijtenbeek's simulations of models of bipedal virtual creatures with muscular simulations [7], where an optimization of placements of muscles using covariance matrix adaptation evolution strategy (CMA-ES) resulted in higher stability in the models' locomotion. Following such observation, we can hypothesize that a lack of muscular strain penalty in our pigeon model may be the cause in the discrepancies. As a progression in incremental modeling,

an addition of muscular physics would be appropriate.

The lack of emergent behaviors that resemble phases in pigeon models controlled by non-hierarchical reinforcement learning policies trained on r_{fifty_fifty}, indicates that a hierarchical control system may be embedded in pigeons' neurology that activates when they exhibit head-bobbing behaviors. An addition of a high-level state machine-like control mechanism composed of a "hold phase" state and a "thrust phase" state may be a viable approach for replicating such behaviors. The high-level controller can output values that indicate the appropriate phase for the pigeon model to execute, such as a binary digit where 1 represents the hold phase and 0 represents the thrust phase.

Hierarchical reinforcement learning could be used for modeling such control systems. Preliminary research have examples of successfully accomplishing complex tasks that require sequential execution of multiple smaller tasks. Diversity Is All You Need [4] can execute sequential tasks such as running, jumping over hurdles, and resuming running, by acquiring skills, or primitive tasks, in the low-level policy and executing them in order using the high-level policy. Similarly, FeUdal Networks [13] have shown to complete Montezuma's Revenge, an ATARI game that was deemed to be difficult to complete without using control systems that can execute sequential tasks.

Chapter 8

Conclusion

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In efficitur porta augue, at interdum nunc lobortis at. Morbi feugiat facilisis justo, vitae maximus dolor. Cras convallis at elit in porta. Fusce lobortis tortor nibh, quis imperdiet arcu luctus quis. Mauris imperdiet urna eu mauris aliquet, vitae tincidunt orci dapibus. Vestibulum convallis elit ut velit accumsan cursus. Pellentesque lacus lacus, blandit eu felis vitae, pellentesque dignissim est.

References

[1] Bøhn, E., Coates, E. M., Moe, S., and Johansen, T. A. Deep reinforcement learning attitude control of fixed-wing uavs using proximal policy optimization. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE (2019), 523–533.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] DAVIES, M. N., and GREEN, P. R. Head-bobbing during walking, running and flying: relative motion perception in the pigeon. *Journal of Experimental Biology 138*, 1 (1988), 71–91.

[4] Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070* (2018).

[5] Frost, B. The optokinetic basis of head-bobbing in the pigeon. *Journal of Experimental Biology 74*, 1 (1978), 187–195.

[6] Gao, C., Hernandez-Leal, P., Kartal, B., and Taylor, M. E. Skynet: A top deep rl agent in the inaugural pommerman team competition. *arXiv preprint arXiv:1905.01360* (2019).

[7] Geijtenbeek, T., Van De Panne, M., and Van Der Stappen, A. F. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics (TOG) 32*, 6 (2013), 1–11.

[8] Guan, Y., Ren, Y., Li, S. E., Sun, Q., Luo, L., and Li, K. Centralized cooperation for connected and automated vehicles at intersections by proximal policy optimization. *IEEE Transactions on Vehicular Technology 69*, 11 (2020), 12597–12608.

[9] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, PMLR (2018), 1861–1870.

[10] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905* (2018).

[11] OpenAI. Getting started with gym. `https://gym.openai.com/docs/`, 2016.

[12] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[13] Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, PMLR (2017), 3540–3549.

[14] Webb, B. What does robotics offer animal behaviour? *Animal behaviour 60*, 5 (2000), 545–558.

Appendix

A. OpenAI Gym for Simplified Model of Pigeons' Head Control

Our OpenAI Gym [2] environments `PigeonEnv3Joints` and `PigeonRetinalEnv` mentioned in Experiments are written in Python. Implementation for running and testing our environments should be referred to the documentation for OpenAI Gym [11].

A.1 Usage

```
PigeonEnv3Joints(self, body_speed = 0,
                reward_code = "head_stable_manual_reposition",
                max_offset = 0.5)
```

- `body_speed` indicates the speed in which the pigeon model's body moves.
- Each of the following `reward_code` are assigned to their respective reward functions.
  - `"head_stable_manual_reposition"`

    Implementation of $r_{head\_stable\_manual\_reposition}$
  - `"head_stable_manual_reposition_strict_angle"`

Implementation of $r_{head\_stable\_manual\_reposition\_strict\_angle}$

- `max_offset` indicates the max_offset for each reward function to reference.

```
PigeonRetinalEnv(self,
                body_speed = 0,
                reward_code = "motion_parallax")
```

Parallel to `PigeonEnv3Joints`, each of the following `reward_code` are assigned to their respective reward functions.

- `"retinal_stabilization"`
  - Implementation of $r_{head\_stabilize}$
  - Depicts the preliminary hypothesis regarding the functionality of retinal stabilization during the hold phase.
- `"motion_parallax"`
  - Implementation of $r_{motion\_parallax}$
  - Depicts the preliminary hypothesis regarding the functionality of motion parallax induced depth perception during the thrust phase.
- `"fifty_fifty"`
  - Implementation of $r_{head\_stable\_manual\_reposition\_strict\_angle}$
  - Sum of rewards produced by `"retinal_stabilization"` and `"motion_parallax"`

A.2 Dependencies (Anaconda YAML File)

The listed versions are recommendations and not strictly necessary for replication

```
name: pigeon-env
channels:
  - conda-forge
```

```
  - defaults
dependencies:
  - bzip2=1.0.8=h0d85af4_4
  - ca-certificates=2021.10.8=h033912b_0
  - certifi=2016.9.26=py36_0
  - ffmpeg=4.3.2=h4dad6da_0
  - freetype=2.10.4=h4cff582_1
  - future=0.18.2=py36h79c6626_3
  - gettext=0.19.8.1=h7937167_1005
  - gmp=6.2.1=h2e338ed_0
  - gnutls=3.6.13=h756fd2b_1
  - lame=3.100=h35c211d_1001
  - libcxx=12.0.0=h2f01273_0
  - libffi=3.3=hb1e8313_2
  - libiconv=1.16=haf1e3a3_0
  - libpng=1.6.37=h7cec526_2
  - ncurses=6.3=hca72f7f_2
  - nettle=3.6=hedd7734_0
  - openh264=2.1.1=hfd3ada9_0
  - openssl=1.1.1l=h0d85af4_0
  - pip=21.2.2=py36hecd8cb5_0
  - pybox2d=2.3.10=py36hefe7e0e_1
  - pyglet=1.5.16=py36h79c6626_0
  - python=3.6.13=h88f2d9e_0
  - python_abi=3.6=2_cp36m
  - readline=8.1.2=hca72f7f_1
  - setuptools=58.0.4=py36hecd8cb5_0
  - sqlite=3.37.0=h707629a_0
  - tk=8.6.11=h7bc2e8c_0
  - wheel=0.37.1=pyhd3eb1b0_0
  - x264=1!161.3030=h0d85af4_1
  - xz=5.2.5=h1de35cc_0
  - zlib=1.2.11=h4dc903c_4
  - pip:
    - cloudpickle==2.0.0
    - gym==0.21.0
```

```
    - importlib-metadata==4.8.3
    - numpy==1.19.5
    - typing-extensions==4.0.1
    - zipp==3.6.0
```

A.3 Manually Defined Head Trajectory (Baseline)

```
from Box2D import *
import gym
from gym import spaces

from math import sin, pi, sqrt
import numpy as np
from copy import copy, deepcopy

# anatomical variables ("macros")
BODY_WIDTH = 10
BODY_HEIGHT = 5

LIMB_WIDTH = 5
LIMB_HEIGHT = 2

HEAD_WIDTH = 3

ANGLE_FREEDOM = 0.6

# control variables/macros
MAX_JOINT_TORQUE = 200 #70
MAX_JOINT_SPEED = 5 #10
VELOCITY_WEIGHT = 1.0 #0.9
LIMB_DENSITY = 0.1 ** 3
LIMB_FRICTION = 5

VIEWPORT_SCALE = 6.0
FPS = 60
```

```
HEAD_OFFSET_X = 10
HEAD_OFFSET_Y = 2

class PigeonEnv3Joints(gym.Env):
    metadata = {"render.modes": ["human", "rgb_array"], "video.
        frames_per_second": FPS}

    def __init__(self,
                 body_speed = 0,
                 reward_code = "head_stable_manual_reposition",
                 max_offset = 0.5):
        """
        Action and Observation space
        """

        # 3-dim joints' torque ratios
        self.action_space = spaces.Box(
            np.array([-1.0] * 3).astype(np.float32),
            np.array([1.0] * 3).astype(np.float32),
        )
        # 2-dim head location;
        # 1-dim head angle;
        # 3x2-dim joint angle and angular velocity;
        # 1-dim x-axis of the body
        # [NEW] 2-dim target head location
        high = np.array([np.inf] * 12).astype(np.float32) # formally 10
        self.observation_space = spaces.Box(-high, high)

        """
        Box2D Pigeon Model Params and Initialization
        """
        self.world = b2World()                    # remove in Framework
        self.body = None
        self.joints = []
        self.head = None
```

```
    self.bodyRef = [] # for destruction
    self.body_speed = body_speed
    self._pigeon_model()

    """
    Box2D Simulation Params
    """
    self.timeStep = 1.0 / FPS
    self.vel_iters, self.pos_iters = 10, 10

    self.viewer = None

    """
    Assigning a Reward Function
    """
    self._assign_reward_func(reward_code, max_offset)

"""
Define Reward Function and Necessary Parameters
"""
def _assign_reward_func(self, reward_code, max_offset):
    if "head_stable_manual_reposition" in reward_code:
        self.max_offset = max_offset

        self.relative_repositioned_head_target_location = np.array(
            self.head.position) - np.array([0, HEAD_OFFSET_Y])
        self.head_target_location = self.
            relative_repositioned_head_target_location + np.array(
            self.body.position)
        self.head_target_angle = self.head.angle
        self.reward_function = self._head_stable_manual_reposition

        if "strict_angle" in reward_code:
            self.reward_function = self.
                _head_stable_manual_reposition_strict_angle
```

```
    else:
        raise ValueError("Unknown reward_code")

"""
Box2D Pigeon Model
"""
def _pigeon_model(self):
    # params
    body_anchor = np.array([float(-BODY_WIDTH), float(BODY_HEIGHT)])
    limb_width_cos = LIMB_WIDTH / sqrt(2)

    self.bodyRef = []
    # body definition
    self.body = self.world.CreateKinematicBody(
        position = (0, 0),
        shapes = b2PolygonShape(box = (BODY_WIDTH, BODY_HEIGHT)), #
            x2 in direct shapes def
        linearVelocity = (-self.body_speed, 0),
        angularVelocity = 0,
        )
    self.bodyRef.append(self.body)

    # neck as limbs + joints definition
    self.joints = []
    current_center = deepcopy(body_anchor)
    current_anchor = deepcopy(body_anchor)
    offset = np.array([-limb_width_cos, limb_width_cos])
    prev_limb_ref = self.body
    for i in range(2):
        if i == 0:
            current_center += offset

        else:
            current_center += offset * 2
            current_anchor += offset * 2
```

```
    tmp_limb = self.world.CreateDynamicBody(
        position = (current_center[0], current_center[1]),
        fixtures = b2FixtureDef(density = LIMB_DENSITY,
                                friction = LIMB_FRICTION,
                                restitution = 0.0,
                                shape = b2PolygonShape(
                                    box = (LIMB_WIDTH, LIMB_HEIGHT))
                                        ,
                                ),
        angle = -pi / 4
    )
    self.bodyRef.append(tmp_limb)

    tmp_joint = self.world.CreateRevoluteJoint(
        bodyA = prev_limb_ref,
        bodyB = tmp_limb,
        anchor = current_anchor,
        lowerAngle = -ANGLE_FREEDOM * b2_pi, # -90 degrees
        upperAngle = ANGLE_FREEDOM * b2_pi, # 90 degrees
        enableLimit = True,
        maxMotorTorque = MAX_JOINT_TORQUE,
        motorSpeed = 0.0,
        enableMotor = True,
    )

    self.joints.append(tmp_joint)
    prev_limb_ref = tmp_limb

# head def + joints
current_center += offset
current_anchor += offset * 2
self.head = self.world.CreateDynamicBody(
    position = (current_center[0] - HEAD_WIDTH, current_center
        [1]),
    fixtures = b2FixtureDef(density = LIMB_DENSITY,
                            friction = LIMB_FRICTION,
```

```
                                restitution = 0.0,
                                shape = b2PolygonShape(
                                    box = (HEAD_WIDTH, LIMB_HEIGHT)),
                                ),
        )
        self.bodyRef.append(self.head)

        head_joint = self.world.CreateRevoluteJoint(
            bodyA = prev_limb_ref,
            bodyB = self.head,
            anchor = current_anchor,
            lowerAngle = -ANGLE_FREEDOM * b2_pi, # -90 degrees
            upperAngle = ANGLE_FREEDOM * b2_pi, # 90 degrees
            enableLimit = True,
            maxMotorTorque = MAX_JOINT_TORQUE,
            motorSpeed = 0.0,
            enableMotor = True,
        )
        self.joints.append(head_joint)

        # head tracking
        self.head_prev_pos = np.array(self.head.position)
        self.head_prev_ang = self.head.angle

    def _destroy(self):
        for body in self.bodyRef:
            # all associated joints are destroyed implicitly
            self.world.DestroyBody(body)

    def _get_obs(self):
        # (self.head{relative}, self.joints -> obs) operation
        obs = np.array(self.head.position) - np.array(self.body.position)
        obs = np.concatenate((obs, self.head.angle), axis = None)
        for i in range(len(self.joints)):
            obs = np.concatenate((obs, self.joints[i].angle), axis = None
                )
```

```
        obs = np.concatenate((obs, self.joints[i].speed), axis = None
            )
    obs = np.concatenate((obs, self.body.position[0]), axis = None)

    # complement a target position
    obs = np.concatenate((obs, self.head_target_location - np.array(
        self.body.position)),
                         axis = None)

    obs = np.float32(obs)
    assert self.observation_space.contains(obs)
    return obs

def reset(self):
    self._destroy()
    self._pigeon_model()
    return self._get_obs()

def _head_target_reposition_mechanism(self):
    # detect whether the target head position is behind the body edge
         or not
    if self.head_target_location[0] > self.body.position[0] - float(
        BODY_WIDTH + HEAD_OFFSET_X):
        self.head_target_location = np.array(self.body.position) + \
            self.relative_repositioned_head_target_location

    head_dif_loc = np.linalg.norm(np.array(self.head.position) - \
            self.head_target_location)
    head_dif_ang = abs(self.head.angle - self.head_target_angle)
    return head_dif_loc, head_dif_ang

"""
Modular Reward Functions
"""
def _head_stable_manual_reposition(self):
    # This method is separated from step(), since there are variables
```

```
        used
    # that are only defined in with this strain of reward functions
    head_dif_loc, head_dif_ang = self.
        _head_target_reposition_mechanism()

    reward = 0
    # threshold reward function with static offset
    if head_dif_loc < self.max_offset:
        reward += 1 - head_dif_loc/self.max_offset

        if head_dif_ang < np.pi / 6: # 30 deg
            reward += 1 - head_dif_ang/ np.pi

    return reward

def _head_stable_manual_reposition_strict_angle(self):
    head_dif_loc, head_dif_ang = self.
        _head_target_reposition_mechanism()

    reward = 0
    # threshold reward function with static offset
    if head_dif_loc < self.max_offset:
        if head_dif_ang < np.pi / 6: # 30 deg
            reward += 1 - head_dif_ang/ np.pi

    return reward

def step(self, action):
    assert self.action_space.contains(action)
    # self.world.Step(self.timeStep, self.vel_iters, self.pos_iters)
    # Framework handles this differently
    # Referenced bipedal_walker
    # self.world.Step(1.0 / 50, 6 * 30, 2 * 30)
    self.world.Step(1.0 / FPS, self.vel_iters, self.pos_iters)
    obs = self._get_obs()
```

```
        # MOTOR CONTROL
        for i in range(len(self.joints)):
            # Copied from bipedal_walker
            self.joints[i].motorSpeed = float(MAX_JOINT_SPEED * (
                VELOCITY_WEIGHT ** i) * np.sign(action[i]))
            self.joints[i].maxMotorTorque = float(
                MAX_JOINT_TORQUE * np.clip(np.abs(action[i]), 0, 1)
            )

        reward = self.reward_function()

        done = False
        info = {}
        return obs, reward, done, info

    def render(self, mode = "human"):
        from gym.envs.classic_control import rendering
        if self.viewer is None:
            self.viewer = rendering.Viewer(500, 500)

            # Set ORIGIN POINT relative to camera
            self.camera_trans = b2Vec2(-250, -200) \
            + VIEWPORT_SCALE * self.bodyRef[0].position # camera moves
                with body

            ## Needs head_stable_manual_reposition reward function to
                execute
            try:
                # init visualize max_offset
                render_target_area = rendering.make_circle( \
                    radius=VIEWPORT_SCALE * self.max_offset,
                    res=30,
                    filled=True)
                target_translate = rendering.Transform(
                    translation = VIEWPORT_SCALE * self.
                        head_target_location - self.camera_trans,
```

```
            rotation = 0.0,
            scale = VIEWPORT_SCALE * np.ones(2)
        )
        render_target_area.add_attr(self.target_translate)
        render_target_area.set_color(0.0, 1.0, 0.0)
        self.viewer.add_geom(render_target_area)
    except:
        pass

    # init translation and rotation for each limb
    self.render_polygon_list = []
    self.render_polygon_rotate_list = []
    self.render_polygon_translate_list = []
    for body in self.bodyRef:
        polygon = rendering.FilledPolygon(
            body.fixtures[0].shape.vertices
        )
        rotate = rendering.Transform(
            translation = (0.0, 0.0),
            rotation = body.angle,
        )
        translate = rendering.Transform(
            translation = VIEWPORT_SCALE * body.position - self.
                camera_trans,
            rotation = 0.0,
            scale = VIEWPORT_SCALE * np.ones(2)
        )
        polygon.set_color(1.0, 0.0, 0.0)
        polygon.add_attr(rotate)
        polygon.add_attr(translate)
        self.render_polygon_list.append(polygon)
        self.render_polygon_rotate_list.append(rotate)
        self.render_polygon_translate_list.append(translate)
        self.viewer.add_geom(polygon)

# Update ORIGIN POINT relative to camera
```

```
        self.camera_trans = b2Vec2(-250, -200) \
        + VIEWPORT_SCALE * self.bodyRef[0].position # camera moves with
            body

        ## Needs head_stable_manual_reposition reward function to execute
        try:
            # update max_offset shape translation
            new_target_translate = VIEWPORT_SCALE * self.
                head_target_location - self.camera_trans
            self.target_translate.set_translation(new_target_translate
                [0], new_target_translate[1])
        except:
            pass

        # update body rotation and translation
        for i, body in enumerate(self.bodyRef):
            self.render_polygon_rotate_list[i].set_rotation(body.angle)
            new_body_translate = VIEWPORT_SCALE * body.position - self.
                camera_trans
            self.render_polygon_translate_list[i].set_translation(
                new_body_translate[0], new_body_translate[1])

        return self.viewer.render(return_rgb_array = mode == "rgb_array")

    def close(self):
        # self._destroy()
        # self.world = None

        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

A.4 Pigeons' Head Control Based on Retinal Inputs

```
import PigeonEnv3Joints, VIEWPORT_SCALE
import numpy as np
import gym
from gym import spaces

class PigeonRetinalEnv(PigeonEnv3Joints):

    def __init__(self,
                 body_speed = 0,
                 reward_code = "motion_parallax"):

        """
        Object Location Init (2D Tensor)
        """
        self.objects_position = np.array([[-30.0, 30.0],
                                          [-30.0, 60.0],
                                          [-60.0, 30.0],])
        self.objects_velocity = np.array([[0.0, 0.0],
                                          [1.0, 0.0],
                                          [-1.0, 0.0],])

        """
        Init based on superclass
        Reward function is defined here
        """
        super().__init__(body_speed, reward_code)

        """
        Redefining Observation space
        """
        # 2-dim head location;
        # 1-dim head angle;
        # 3x2-dim joint angle and angular velocity;
        # 1-dim x-axis of the body
```

```
    high = np.array([np.inf] * 10).astype(np.float32) # formally 10
    self.observation_space = spaces.Box(-high, high)


"""
Retinal coords (angles); Within [-np.pi, np.pi]
"""
def _get_retinal(self, object_position):
    # normalized direction of object from head
    object_direction = object_position - np.array(self.head.position)
    object_direction = object_direction / np.linalg.norm(
        object_direction)

    sign = np.ones(object_direction.shape[0])
    for i in range(sign.size):
        # is the object above or below the head?
        if object_direction[i][1] < 0:
            sign[i] = -1

    # calculate COSINE angle of object relative to head (positive if
        above, negative if below)
    # cosine_angle is of size [num_objects,]
    cosine_angle = sign * np.arccos( \
        np.dot(object_direction, np.array([-1.0, 0.0])))

    # differnce in angle between the head angle and sine_angle of
        head
    relative_angle = cosine_angle + self.head.angle

    # relative_angle should be within [-np.pi, np.pi]
    for i in range(relative_angle.shape[0]):
        if relative_angle[i] < -np.pi:
            k = 1
            while relative_angle[i] < (k + 1) * -np.pi:
                k += 1
            relative_angle[i] = relative_angle[i] + 2 * np.pi * ((k +
```

```
                1) // 2)

        elif relative_angle[i] > np.pi:
            k = 1
            while relative_angle[i] > (k + 1) * np.pi:
                k += 1
            relative_angle[i] = relative_angle[i] - 2 * np.pi * ((k +
                1) // 2)

    return relative_angle

def _get_angular_velocity(self, prev_ang, current_ang):
    angle_velocity = current_ang - prev_ang
    angle_speed = np.absolute(angle_velocity)
    for i in range(angle_velocity.size):
        if angle_speed[i] > np.pi:
            angle_velocity[i] = 2 * np.pi - angle_velocity[i]
        elif angle_speed[i] < -np.pi:
            angle_velocity[i] = 2 * np.pi + angle_velocity[i]
        else:
            pass
    return angle_velocity

"""
Defining Reward Functions
"""
def _assign_reward_func(self, reward_code, max_offset = None):
    self.prev_angle = self._get_retinal(self.objects_position)
    if "motion_parallax" in reward_code:
        self.reward_function = self._motion_parallax
    elif "retinal_stabilization" in reward_code:
        self.reward_function = self._retinal_stabilization
    elif "fifty_fifty" in reward_code:
        self.reward_function = self._fifty_fifty
    else:
        raise ValueError("Unknown reward_code")
```

```
def _motion_parallax(self):
    current_angle = self._get_retinal(self.objects_position)

    parallax_velocities = \
        self._get_angular_velocity(current_angle, self.prev_angle)

    reward = 0
    # sum of motion parallax magnitudes
    for i in range(parallax_velocities.size):
        for j in range(i, parallax_velocities.size):
            reward += np.abs(parallax_velocities[i] -
                parallax_velocities[j])
        # reward += parallax_velocities[i]
    return reward

def _retinal_stabilization(self):
    reward = 0
    current_angle = self._get_retinal(self.objects_position)
    relative_speeds = \
        np.absolute(self._get_angular_velocity(current_angle, self.
            prev_angle))
    reward -= np.sum(relative_speeds)
    return reward

def _fifty_fifty(self):
    reward = 0
    reward += self._retinal_stabilization()
    reward += self._motion_parallax()
    return reward

def _get_obs(self):
    # (self.head{relative}, self.joints -> obs) operation
    obs = np.array(self.head.position) - np.array(self.body.position)
    obs = np.concatenate((obs, self.head.angle), axis = None)
    for i in range(len(self.joints)):
```

```
            obs = np.concatenate((obs, self.joints[i].angle), axis = None
                )
            obs = np.concatenate((obs, self.joints[i].speed), axis = None
                )
        obs = np.concatenate((obs, self.body.position[0]), axis = None)
        obs = np.float32(obs)
        assert self.observation_space.contains(obs)
        return obs

    def step(self, action):
        self.prev_angle = self._get_retinal(self.objects_position)
        # alter object
        self.objects_position += self.objects_velocity
        return super().step(action)

    def render(self, mode = "human"):
        from gym.envs.classic_control import rendering
        if self.viewer is None:
            self.render_objects_list = None
            self.render_objects_translate_list = None

        super().render(mode)
        # initialize object rendering pointers
        if self.render_objects_list is None:
            self.render_objects_list = []
            self.render_objects_translate_list = []
            for i in range(self.objects_position.shape[0]):
                object_render_instance = rendering.make_circle( \
                    radius=0.6,
                    res=30,
                    filled=True)
                object_render_instance_translate = rendering.Transform(
                    translation = VIEWPORT_SCALE * \
                        (self.objects_position[i] - self.camera_trans),
                    rotation = 0.0,
                    scale = VIEWPORT_SCALE * np.ones(2)
```

```
        )
        object_render_instance.add_attr(
            object_render_instance_translate)
        object_render_instance.set_color(0.0, 1.0, 0.0)
        self.render_objects_list.append(object_render_instance)
        self.render_objects_translate_list.append(
            object_render_instance_translate)
        self.viewer.add_geom(object_render_instance)

# update object translation
new_object_translate = VIEWPORT_SCALE * self.objects_position -
    self.camera_trans
for i in range(self.objects_position.shape[0]):
    self.render_objects_translate_list[i].set_translation( \
        new_object_translate[i][0], new_object_translate[i][1])

return self.viewer.render(return_rgb_array = mode == "rgb_array")
```

B. Hyperparameters for Soft Actor Critic Training

In the paper which proposed the soft actor critic algorithm [9], a workflow is presented as Figure 8.1.

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

Figure 8.1: Soft Actor Critic algorithm as seen in [9]

The soft actor critic algorithm consists of 3 deep neural networks: a state value function V, a state-action value function Q, and a policy π. In 8.1, the parameters of the functions, or in this case their weight values, are set as ψ, θ, and ϕ respectively. Each deep neural network has one hidden layer that consists of 256 neurons.

The learning rates of each deep neural network are set as the following.

- $\lambda_V = 3.0 \times 10^{-4}$
- $\lambda_Q = 3.0 \times 10^{-4}$
- $\lambda_\pi = 3.0 \times 10^{-4}$
- $\tau = 5.0 \times 10^{-3}$

We trained our policies using batch training, where we preserve a set size of trajectories, or replay buffer size, of states and actions taken by the policy per iteration. Parameters regarding batch training are set as the following.

- Iterations (epochs): 3000
- Environment steps per iteration: 1000
- Gradient steps per iteration: 1000
- Batch size: 256
- Replay buffer size: 1.0×10^6

Trajectories of states and actions totaling 5000 steps per epoch for determining the average return for evaluation.

Returns R are calculated as the following,

$$R = \sum_{t=0}^{n} \gamma^t r_t \tag{8.1}$$

where r_t is the reward gained during timestep t, n is the total number of timesteps taken to calculate R, and γ is the discount. For all of our experiments, we set $\gamma = 0.99$.