Training a fly to walk in *silico* using reinforcement learning.

Timo Achard
under the supervision of
Sibo Wang and Gizem Ozdil

January 13, 2023

Abstract

Physics simulations, which are primordial for reinforcement learning tasks, tend to consume a lot of memory, computing power and time, due to the huge amount of calculations required. *Nvidia* has created an end-to-end GPU accelerated simulator called *IsaacGym* where both physics simulation and neural network policy training reside on GPU, making training time significantly shorter. Prior to this work, we didn't know whether it was possible to create an environment using this new tool to train a realistic model of an adult Drosophila melanogaster, called *NeuroMechFly*, to walk.

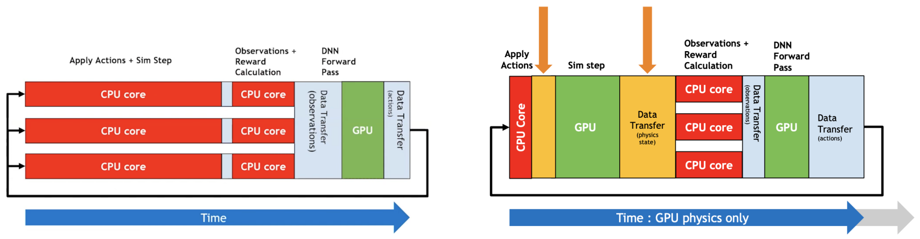
Here, I show how I was able create such working physics environment and make the model stand and exhibit walking behaviour, although not yet biologically realistic. I discuss the problems encountered, and particularly one concerning units used in the simulation, and how it could be ameliorated upon. The result is a proof of concept that the work done in other RL environments using *NeuroMechFly* can be translated in this faster and computationally less expensive environment. The code that I produced can be built upon to make an RL agent learn to walk realistically and then use such NN models to better understand the workings of neural circuits that can control these movements.

Contents

1 Introduction

Neural networks (NNs) have been for some time now often run on graphics processing units (GPU) as they contain many small, lightweight processors that can work in parallel to perform the matrix multiplications and other operations involved in training and running NNs. This allows to perform these computations much faster than on traditional central processing units (CPUs) and to significantly reduce the training and running times of the network. Unfortunately, the physic simulation used to reproduce the environment was still running on CPUs [Fig.1a]. One needed huge computers to be able to run thousands of environments at the same time until *Nvidia* tackled this problem by applying the same strategy used for the NN: running everything on GPUs. At the beginning, NN and physic simulations were running on the GPU while the reward function was still calculated on CPU, implying big amounts of data to transfer between the CPU and the GPU and therefore time losses [Fig.1b]. *Nvidia* was finally able to completely optimize the process [Fig.2] and created an end-to-end GPU accelerated simulator for robotics and AI research called *IsaacGym* [1].

The goal of this bachelor project is to design and implement a NN that can control the movement of a modelled fly using reinforcement learning (RL) and *IsaacGym*. This involves creating a simulation environment and developing a NN architecture that can take in sensory input, make decisions based on that input, and output the desired positions for the fly's legs degrees of freedom in order to generate a walking behavior. This project is linked to a larger project which aims to understand the principles of locomotion control of an adult Drosophila Melanogaster and demonstrate the potential for NNs using a physics-based simulation environment to replicate these behaviors in order to extract contact forces and joint torques from the model and better understand neuromuscular dynamics. The present work will therefore serve as a proof of concept that the work done on other physic engines can be translated to *IsaacGym* to have faster and maybe more accurate simulations [2].



(a) Normal RL environment (b) Semi-accelerated RL environment

Figure 1: RL compute timelines [3]

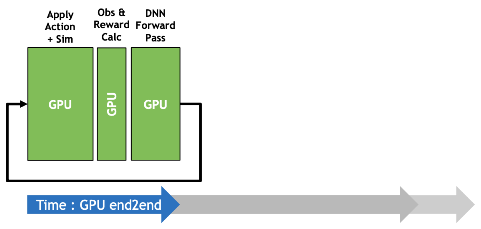


Figure 2: Fully accelerated Environment [3]

2 State-of-the-art

I didn't start this project totally in the unknown: the *Nvidia* team released along with *IsaacGym* a repository called *IsaacGymEnvs* [4]. With the sole purpose of helping developer use *IsaacGym* by giving them examples, *IsaacGymEnvs* has 14 environment completely implemented and working flawlessly. The main issue with those examples is the lack of documentation and the complexity of the structure of the code, particularly the interface between the environment and the NN. Making them really difficult to use at the start of the project, but pretty handy afterward, once more used to the API.

Therefore, to learn how these libraries work with a simple example, I have used *Minimal-Isaac-Gym* [5]. It has only 3 important files, the relation between them is simple to understand and it works pretty flawlessly. It is not perfect but it is one of the best ways to start a project for someone with no prior experience with *IsaacGym*.

3 Methods

3.1 Structure of the code

I started by forking [6] the repository *Minimal-Isaac-Gym* and building on top of that, so the structure remained exactly the same. First, `trainer.py` is the file to be called to launch the experiment, it parses the arguments and holds the main loop. The main loop just calls `ppo.py` the second important file which holds the neural network and the run function called by the main loop. This function takes the observations, decides on the actions to take and steps the environment. If enough steps have been made it calls the update functions, which updates the NN. `Fly.py` is the last important file that defines the environment and is responsible to step and apply the actions when required. This file is the heart of my project as it is the file that uses *IsaacGym* to simulate the environment on the GPU.

3.2 Implementing the environment

Implementing the environment has been the most difficult part of this project and I think it is still not perfect to this day. As said before, I started by forking a repository called *Minimal-Isacc-Gym* which was implementing the classical Cartpole environment. My first task was then to modify the code in order to load the asset of the fly and change the different variables so that every tensor has the correct size. This last part wasn't easy as I had trouble finding the numbers of degrees of freedom (DoF) of my asset, how many of them were controllable by the NN and in which order they were in the tensors. Once done, I had a fly standing in the environment, doing nothing. On top of that base, I have build all the functions described further below in order to control the environment so that the NN can correctly learn the model to walk.

3.2.1 The asset

The fly model I used for my environment is a very accurate model [Fig.3] written in the `Urdf` language and developed by the Ramdya Lab at EPFL [7]. It is composed of 92 parts and 42 degrees of freedom. To simplify the problem I used only 18 of them: 3 per legs [Appendix A]. The fly has a base position and the angles (in degree) of all the DoFs in this base position are stored in the `file pose_default.yaml`. In order to only use 18 of the DoFs I created a mask and, via the method

```
self.gym.set_dof_position_target_tensor(self.sim, positions)
```

I was setting the base position masked with the action given. ie.

```
initial_position[mask] = action_given_by_NN
```

But this meant that the position controller had to also handle the DoFs we were not using at all. This created a lot of dampening as the position controller isn't perfect. To correct that I created a small script (`script_change_urdf.py`) which changed in the urdf file the DoFs into fixed joint with the right angle. This greatly reduced the dampening.

3.2.2 Reward engineering

At first I did a really simple reward function to do some testing: flies gain rewards if they go away from the start position in any direction and reset after 500 time steps. As expected this was too simple and I quickly built upon it when my simulation was better. I made the flies reset more cleverly and gave them rewards for particular behaviours. For example, as standing was an issue, if a fly was too close to the ground for 100 steps then it would reset and have a penalty. Oppositely, the more a fly was up in the air the

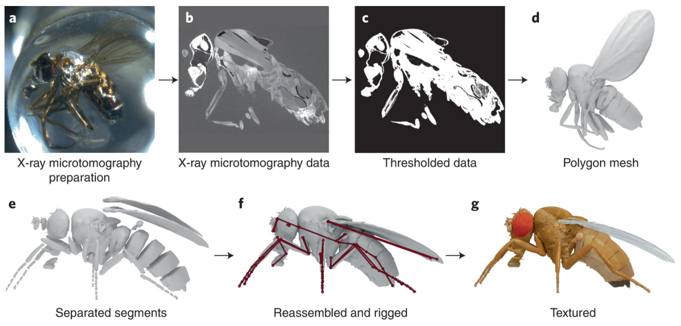


Figure 3: The process of making the NeuroMechFly [2]

more it would earn a reward. Also, the flies only gained reward for going away from the starting position in the x direction, hoping that they will learn to head in a particular direction not randomly. Unfortunately, this was not sufficient. So, I looked at the ant example given by the *IsaacGymEnv*, which is the closest example from what I was trying to implement, and I adapted its reward function to my code. It wasn't very different from my own reward function but it was better implemented. Upon that, I also added some conditions of my own. All the different rewards are explained below. The big difference between the ant example and my implementation is that the ant model can easily keep upright and parallel to the ground, which makes things a lot easier. Sadly, my flies were not able to learn both to stand and to walk at the same time.

I therefore partitioned the problem in two smaller problems: I first taught them to stand up without anything else than the legs touching the ground and I then used the weights of this trained neural network as the initialization weights of a new simulation where the flies where incentivised to walk. And this worked pretty well. Reward for the standing up part (most of the reward are scaled down):

```
total_reward = alive_reward + up_reward * orient_reward
- energy_cost_scale * electricity_cost - dof_at_limit_cost
* joints_at_limit_cost_scale + leg_ground_reward
```

Reward for the walking part (most of the reward are scaled down):

```
total_reward = progress_reward * 2 + alive_reward
+ up_reward * orient_reward + heading_reward - actions_cost_scale
* actions_cost - energy_cost_scale * electricity_cost
- dof_at_limit_cost * joints_at_limit_cost_scale + leg_ground_reward
```

The different rewards are:

- The heading reward. We want the flies to be able to go to a specific target so a bigger reward is given if they are oriented in the right direction.

- The progress reward. If the flies makes progress toward the target they also get a reward.
- The up reward. We want the flies to be walking realistically, not crawling. So a reward is given if their torso is above a certain height from the ground.
- The action cost. Actions have a bigger cost the further away they are from 0.
- The electricity cost. We don't want the flies to do erratic movements, so the greater the difference between the new and the previous action, the bigger the penalty.
- The DoF at limit cost. The flies are supposed to use their limbs in a physically "correct" way. A DoF at its limit is not normal, this is also penalised.
- The alive reward. We want the flies to stay "alive" -i.e. not reset- for as long as possible, so we reward it for staying alive.
- The orientation reward. We want the flies to be the most parallel possible to the ground plane. They had a tendency to pitch upward.
- The leg reward. Each leg touching the ground gets a reward. This was mainly used for the fly to be able to stand, it had a tendency to not put all its legs on the ground,

There are a lot of parameters and different rewards, it could be argued that some of them may be useless or redundant. In a future version, more testing should be done to assess the impact of each of them on the desired behaviour.

What could also be added is a reward for a balance gait pattern, as it would maybe increase the realism of the walk. But due to the time it took to achieve the first goal and complexity of trying to be more realistic, this problem remains untackled.

3.2.3 Reset engineering

One of the challenges more complicated than expected was implementing the reset function, which puts back the fly and the degrees of freedom at their starting position. This was where I encountered my first serious bug: when resetting all the flies, the observation tensors coming from the simulation would return NAN values and, when rendered, the fly would disappear when resetting. To be short, this was caused by two factors:

First, I was updating the positions of the torso and then fetching the new observation tensors without stepping the environment in between. The *IsaacGym* documentation does not recommend this.

Second, the copy of the tensor which describes the position of the flies was done before the simulation starts so all the values were not initialized, and when passing it to the simulation the flies would disappear but the simulation would not crash. So it took some time to find out what was really happening.

The reset function is responsible to reset the environments but doesn't decide when to do so. This is the job of the reward function and is done using the following rules:

- The flies are reset if they are too low, we don't want them crawling on the ground.
- The flies are reset if their abdomen touches the ground, they had a tendency to use it to be able to stand up.
- They are also reset if they are too high, we don't want them to launch themselves in the air.
- We also reset them if they are upside down, they were doing it too often and going back on track after a flip was be way too hard to learn and inefficient.
- And finally, we reset them after a certain time, just to have fresh data and not having one fly move in circle or be soft locked, for example.

3.2.4 Interfacing between the NN and the environment

The first results of the NN after the interfacing were, as expected, bad. The first problem was the fact that my neural network was outputting values between -1 and 1 but my degrees of freedom have very different max and min values. Some DoFs go from -4.1 to -0.6 and others from -3.1 to 6.9. A single mapping throughout every variable wasn't possible so I created a unique mapping for each DoF, resulting in two mapping tensors, one for scaling and one for shifting, called every time an action is applied to the environment. I later discovered that a cleaner version of what I did was already coded in the *IsaacGymEnv* repository, and switched to that one.

3.2.5 Additional features

I implemented three additional features: firstly, a way to pause the rendering so that the code runs faster, as the rendering of the physic simulation is really slow; and secondly a way to reset all the environments at once at any moment. Both of these are really useful for testing purposes and should be implemented by anyone working with *IsaacGym*, it is really simple to get the key events from the API. Lastly, I implemented a way to print something on the shell at any given time, also very useful for testing: you can know what values certain tensor contain at certain time without printing everything.

3.2.6 Possible different approach

I wrote my code on top of an existing simple code, but one could start from a blank page using the tools from *IsaacGymEnvs*. Indeed, a class named `vecttask` already implements some functions required by *IsaacGym's* API as well as other useful functions such as a random noise generator or rendering functions. In fact, all of their examples inherit from this class and some of those environments are way more complicated than what I did. The main problem is to understand the interfacing between the environment and the NN as there are a lot of different files involved. Some of them are just observers and hold objects defined in other files, some files calls utils functions defined in libraries, etc. The documentation isn't great and most of the parameters are defined in a `.cfg` file so you always have to look it up. It gets really messy quickly so coping the structure is not an easy task. The second reason is that if you decide to keep the interfacing and only modify the used NN, you find out quickly that *IsaacGymEnvs* doesn't use *PyTorch* or *Tensorflow*, but *rl_games* [8] which is open source but doesn't have a lot of documentation either.

Anyway, if one has the time to understand *IsaacGymEnvs's* code, I think it is a perfect tool to do some prototypes because much less code need to be written to create a working environment.

3.3 Implementing the Neural Network

A Proximal Policy Optimization (PPO) and a Deep Q Learning (DQN) agent already existed in the repository *Minimal-Isaac-Gym* that I forked. As the PPO file was easier to understand and to adapt I decided to chose it over the DQN agent and adapted it to my environment. I realized later that all the environments in *IsaacGymEnvs* were using PPO to learn, hence reinforcing my initial choice. After some experimentation with my environment I found it really slow – way slower than the examples of *IsaacGymEnvs* for example. By looking deeper into the code, I noticed that python lists were created. These could not be used in the GPU and induced a lot of time loss. I therefore replaced all those python lists with immutable tensors which was pretty tedious and I took the opportunity to make the code tidier. Unfortunately this didn't improve the speed. After some debugging, I found that the line

```
dist = MultivariateNormal(mu, cov_mat)
```

was up to 33 slower on GPU than on CPU [9]. Replacing this line by

```
scale_tril = torch.cholesky(cov_mat)
dist = MultivariateNormal(mu, scale_tril=scale_tril)
```

solved the problem. As of today I cannot assess if my previous changes were helpful or not.

The `action_var` (action variance) parameter, which decides on how much the actions applied to the environment should differ from what the NN is outputting, is also an important hyperparameter that I didn't really get right for a long time. It was often too small, not allowing the NN to explore the whole set of possible action. Therefore my NN was often stuck in a local minima quickly after the beginning of the simulation.

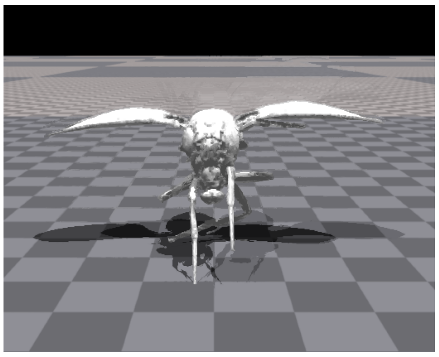
3.3.1 Additional features

I implemented a way to easily save the weights of the NN each and every 100 optimisation steps – it is useful if something happens in the simulation as it allows to go back to an already partly-trained NN to continue to train it or to do some tests. The E key also allows to quit safely by saving the weights in a file. I also implemented a way to load the weights of the neural network to continue some training done earlier.

3.4 Hyperparameters and the correctness of the simulation

A major problem I encountered, was that the flies weren't able to stand alone, even without any input from the NN. To solve this issue, I have set the value of `Stiffness` of the DoF to 1 000 000. But this introduced a bug that I found a long time afterwards. Indeed, I discovered that my flies DoFs weren't resetting properly: I gave them the correct values but the positioning of the DoF seemed to be random [Fig.4a]. When I put it back to 10 000 everything seemed to worked correctly again, I thought this was a bug from *IsaacGymEnvs*.

I looked at this "bug" again at the end of my project but I am not sure about myself anymore. I realised that those weird values of the DoFs could maybe just be the effect of the actions before this reset. Indeed, when resetting I use the function `gym.set_actor_root_state_tensor` which bypasses the physic simulation and hard resets the DoFs at the place given by the tensor. But my actions are set using `gym.set_dof_position_target_tensor` which gives a target to the position controller. The position controller will then transform these positions into forces and pass them to the simulation. When using the hard reset I don't know if the position controller will still hold in memory the actions given and apply them during the next simulation step. This could cause the reset to look visually weird but be a normal behaviour.


(a) Reset of a fly with a high stiffness

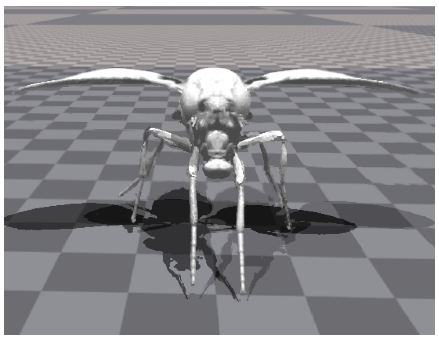

(b) Reset of a fly with a low stiffness

Figure 4: Reset of the fly not working as intended

But why would my environment react like this? To test it out I tried 2 different orders of applying the actions and resetting:

Setting the actions then resetting then simulating then rendering.

```
# Sets the target position dicted by the actions
self.gym.set_dof_position_target_tensor(self.sim, positions)

# Reset the environements
self.reset()

# Simulate
self.simulate()

# Renders
if not self.args.headless :
    self.render()
```

And it still visually doesn't look right even though the reset occurs after setting the actions. Is `gym.simulate()` applying the actions after resetting?

Setting the actions then simulating, then reseting (technically my reset has 1 simulation lag) then rendering.

```
# Sets the target position dicted by the actions
self.gym.set_dof_position_target_tensor(self.sim, positions)

# Simulate
self.simulate()
```

```python
# Reset the environements
self.reset()

# Renders
if not self.args.headless :
    self.render()
```

So my steps of simulation should be done and the reset should occur after it and we then render. We should then see a fly in the good position, but it still doesn't work. Does `gym.set_actor_root_state_tensor` needs a call to `gym.simulate()` to work even though it normally bypasses the position controller? And why does this way of doing works in the examples given by *IsaacGymEnvs* but not in my simulation?

Indeed, *IsaacGymEnvs* does exactly the second method and they appear to have a normal behaviour. It is really hard to tell visually because they introduce some randomness in the position and the velocity during the reset, but their reset position looks reasonable. It may also be because they are directly passing the force and not using the position controller or maybe because their stiffness is in the range [0.5 1.5] not 70. The only observation that I did that reaffirms that everything works as normal is the fact that If I pass the normal resting position as a target for the controller at every step, the reset has no problem [Fig.4b].

Beyond the reset function, another issue that arises at the end of my project is the fact that the stiffness value I use may be way too low. The fly can easily learn to stand because my gravity is artificially very low and therefore this low stiffness is enough to withstand the fly's weight. But looking at the actions given to the environment by the neural network, they seem sometimes way too sparse one from another however visually the fly doesn't move much. This is surely caused by the fact that the stiffness controls the force the position controller has to put to move the joints at the right angle. Without enough force the position controller cannot reach the target before the next actions are applied.

As expected, hyperparemeters have been a big struggle during this project especially towards the end. In particular hyperparameters controlling the DoFs like stiffness, damping, effort and velocity. The 4 of them control the degrees of freedom but we have very few information about how they really work. The documentation only says "The controller will apply DOF forces that are proportional to `posError * stiffness + velError * damping`", with no clear indication of the order of magnitude they should be. They need a lot of testing to get right, and I always thought that if too high they would break the position controller.

Another important hyperparameter that one need to be careful about is the number of environment we want to have. When I go with a high number of environment (4096) everything should work correctly, as it does when working with the examples given by *IsaacGymEnvs*. But in my case the position controller seems again to not work properly anymore and wrong values will be applied to the DoFs. That is really annoying because when testing with 10 environments you cannot always see the defaults you will have with 4096 of them. It is then necessary to stick to a lower number of training environment (~ 300) and it is slower. Maybe I am wrong again and everything works fine, I did not have the time to investigate more deeply.

As said before my flies couldn't hold their weights even without moving [video8, Video11]. So instead of changing the stiffness of the DoFs which I thought was creating problems, I reduced the gravity. *IsaacGym* doesn't care about units but is coherent in the way it applies them, in other words it relies on the user to give values that are the same scale. So if one uses millimeters instead of meters everything would work the same. That is exactly what we did. Indeed, my model fly is scaled up by 1000 to prevent mantissa underflow and not loose precision, so 1 m becomes 1000 mm and the values are not $2.34e^{-12}m$ but $2.34e^{-9}mm$ for example. And we do exactly the same for the mass: we use grams instead of kilograms. So we have to convert the gravity in mm/s^2 to keep the model coherent, this means the gravity is now 9810 mm/s^2. But as said before the simulation doesn't care about units: the gravity is 9810 [something] and the mass is 1000 times bigger so the force ($F = m * g$) applied on the fly is 1 000 000 times what it would be if we didn't change the units. Hence, the gravity becomes an enormous force and requires the joints to be very stiff in order to be able to hold the fly. *IsaacGymEnvs* examples have a stiffness of 0.5 to 1.5, my fly then has to have a stiffness of 50 to 100 which, as seen before, seems to make the position controller bug. Furthermore I don't know if the stiffness scales up the same way that the rest does. It is maybe not linear or it may has incidents over other stuff that should stay the same even if the fly is 1000 time bigger.

This huge gravity force was also requiring the steps of the simulation to be at least 1/500 s, otherwise the fly would clip in the ground. This was problematic because the neural network would then give the fly 500 actions per seconds. When the NN is not trained and outputs random values it will cause the DoFs to always hover around the middle of their range. Indeed, as the NN had 1 in 2 chances to output a value over this middle value and 1 in 2 chances to output a value below and the simulation didn't have the time to finish before an other value would be given, it causes the DoFs to always hover around the middle.

So I reduced the gravity back to 9.81 mm/s^2 which is less realistic but resolved both problems at the same time, it was like the flies were experiencing a gravity of 0.00981 m/s^2, which is about 165 times smaller than if they where on the moon.

You also have a lot of other hyperparameters that I didn't change at all since I had very few informations about them or about the values they should take - for example `sim_params.physx.rest_offset`. So I used the value I saw on other people's code. One should do more testing to really understand how they affect the simulation and thus the learning. A lot of them can really mess up the simulation

4 Results

4.1 Current results

The results were underwhelming at first for 4 main reasons:

1. My environment was still not perfect and I didn't really know if everything was working correctly. I had uncertainties about the correctness of the simulation: Was the observation buffer giving good values to the NN ? Were the actions given by the NN applied correctly ? Was the mapping from [-1 1] to [lim_lower lim_upper] working ? Etc. All of those questions are now answered and all should be correct, but even though I am now confident in my simulation, I am not sure that my changes did not affect something else somewhere which could not be expected and found easily. For example a high stiffness would make position controller not work correctly but this is not easily visible.

2. My action space wasn't diverse enough, I wasn't playing enough with the variation of the outputs of the neural network so it was easily stuck in a local minima

3. I was trying to teach the flies to walk directly as the team at *IsaacGymEnvs* did with the ants. But my model is more complex and is not able to stand as easily. So I had to partition the goal into two subgoals: being able to stand upright and then being able to walk in a direction. I will reuse the weights of the flies that are able to stand alone and then try to learn them how to walk, because I feel like it is too difficult for them to learn both at the same time.

4. My hyperparameters were not well tuned. My first tests were before I understood all I explained in 3.4. So the simulation wasn't good and it was impossible for the NN to learn something relevant.

Despite all these difficulties, I obtained some promising behaviours.

I tried first with 12 DoFs (2 per legs). I didn't use the coxa joint for all the legs. The fly was easily able to stand up right on its 6 legs [video1].

I then changed the reward function while keeping the same weights so that it can try to learn to walk. I got not walking flies but jumping flies [see video2]. This is mostly due to the fact that my reward function for walking was still giving rewards for the legs touching the ground so the flies adopted this technique of jumping. All their legs were therefore touching the ground for the longest amount of time. I tried again without this reward hoping that they would do something else, but they would do the same or be totally unable to do something depending on the amount of variance I was putting:

- If the variance was too high, they would start to do weird things and the NN would unlearn to stand without the reward for the legs.
- If the variance was too low, they would not loose this trait of keeping the legs on the ground.

More testing should be done to maybe find an in between.

I did exactly the same with the 18 DoFs thinking that maybe the coxa joint would help for a more realistic movement after the second training, even though it would be harder for them to learn to stand in the first place. But the results where pretty similar [video3]. It wasn't much harder to learn to stand and they in the end did it as good as with 12 DoFs. The jumps they did afterward where much higher and they were going quicker so there was some improvement [video4]. The small gravity is really affecting the way they are learning and we can see that they don't need to move a lot to attain this height. It is also interesting to note that they will be reset if they pass 6 units of heights, which is a bit more than three times their height, they are really close to this margin and it would also be interesting to see when we make it bigger if they would jump higher or if they are happy with the ratio distance/height.

I did try to train it again without putting the reward for the legs touching the ground and it was more chaotic but way less jumpy [video5, video6, video7]. Maybe we could have some very interesting results with more training and some rewards enticing for a balance gait pattern.

In terms of time, it takes at least 10 to 20 minutes to train the model. It is longer than the examples given by *IsaacGymEnvs* for similar and even bigger tasks. This is mainly due to the small amount of environment I am using (~ 300) because of irregularities of the position controller as explained in 3.4.

Globally, a lot of progress has been made and we have some interesting results.

First, once resolved the previously mentioned problems, the flies are walking, although not in a perfectly realistic way.

Secondly, some experiments with “normal” gravity and a high enough stiffness triggers new behaviours that would deserve further studies:

- For example if we only have the 3 degrees of freedom associated with the two hinge legs, all the flies turn in the same direction at the start of the simulation and walk approximatively in the same direction. However, they are really slow since a lot of their movements are totally useless and the ones that do make a difference are tiny.
- When the NN can use all 18 degrees of freedom, the result is pretty chaotic. They go in the good direction but without a lot of control [video9]. And If you add too much constraints (like the abdomen reset) then the flies are not able to learn anything.
- When you take out the middle legs, the flies are somehow stuck with the middle legs higher than the other ones, preventing a real move.

4.2 Next Steps

From this point there are two routes one could take. The hardest but the most relevant one is resolving the gravity problem we discussed in 3.4. I see two possible approaches:

- One should code the NN so that it doesn't give target positions at every steps of simulations but every 10 steps for example. To do so, one could keep the dt to $1/60s$ but put the subsets to about 10-20. You will then have about $60 * 20 = 1200$ steps in each seconds, which is enough to be able to put the gravity back to 9810 without the flies to clip in the ground. You then have to adapt the stiffness values. But maybe can someone find what is not happening correctly with what I discussed in 3.4 and just apply a stiffness of about 70. If it works then the problem would be solved as every units would then be coherent and we could have some very interesting results.
- Otherwise one could scale back down the fly by 1000. I think that this is the hardest method as you have to change the mass and size of the model and you have to recalculate the matrices of inertia. But if you are able to do so, you could then keep pretty much the same hyperparameters I used and have some interesting results without doing a lot of testing like the other method. Everything should be more easy in *isaacGym* on this point on.

The second route could be to make the actual NN learn better on the actual simulation setup, maybe changing some of the reward function, introduce for example a rewards enticing for a balance gait pattern, maybe change a bit the structure of the NN, number and size of the layers, etc. The problem of this route is that you will never be truly correct, your flies will always be on a planet that has 1/165 the gravity of the moon, so the results are not interesting if you want to really study the walk of the fly and how forces are applied.

5 References

[1] Viktor Makoviychuk et al. *Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning.* Aug. 25, 2021. DOI: 10.48550/arXiv.2108.10470. arXiv: 2108.10470[cs]. URL: http://arxiv.org/abs/2108.10470 (visited on 01/11/2023).

[2] Victor Lobato-Rios et al. "NeuroMechFly, a neuromechanical model of adult Drosophila melanogaster". In: *Nature Methods* 19.5 (May 2022). Number: 5 Publisher: Nature Publishing Group, pp. 620–627. ISSN: 1548-7105. DOI: 10.1038/s41592-022-01466-7. URL: https://www.nature.com/articles/s41592-022-01466-7 (visited on 01/11/2023).

[3] *Isaac Gym Part 1: Introduction and Getting Started.* URL: https://www.youtube.com/watch?v=nleDq-oJjGk (visited on 01/11/2023).

[4] *Isaac Gym Benchmark Environments.* original-date: 2021-08-27T02:36:51Z. Jan. 11, 2023. URL: https://github.com/NVIDIA-Omniverse/IsaacGymEnvs (visited on 01/11/2023).

[5] Shikun Liu. *Minimal Isaac Gym.* original-date: 2022-03-14T11:54:23Z. Nov. 23, 2022. URL: https://github.com/lorenmt/minimal-isaac-gym (visited on 01/11/2023).

[6] petim0. *The Project.* original-date: 2022-11-27T13:56:48Z. Nov. 27, 2022. URL: https://github.com/petim0/fly_bProject (visited on 01/11/2023).

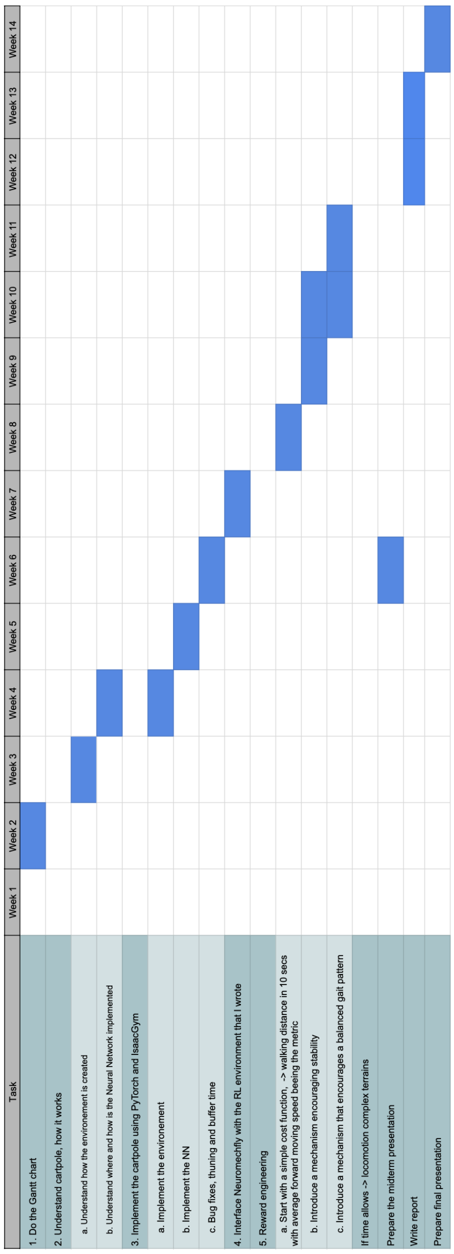
[7] *RAMDYA.* EPFL. URL: https://www.epfl.ch/labs/ramdya-lab/ (visited on 01/11/2023).

[8] Denys Makoviichuk. *RL Games: High performance RL library.* original-date: 2019-01-13T05:35:44Z. Jan. 10, 2023. URL: https://github.com/Denys88/rl_games (visited on 01/11/2023).

[9] *Construction of MultivariateNormal much slower on GPU than CPU · Issue #23780 · pytorch/pytorch.* GitHub. URL: https://github.com/pytorch/pytorch/issues/23780 (visited on 01/11/2023).

6 Original Gannt chart

I didn't follow my Gantt chart in it's entirety. Indeed, after finding the repository *Minimal-Isaac-Gym*, I was able to skip “Implement the Cartpole using *PyTorch* and *IsaacGym*” and save time on that part. On the opposite, as discussed earlier, I wasn't able to implement a reward function that incentives a balance gait pattern because I had a lot of problems with the gravity which took a lot of time to investigate.



Task	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14
1. Do the Gantt chart		🟦												
2. Understand cartpole, how it works														
a. Understand how the environement is created			🟦											
b. Understand where and how is the Neural Network implemented				🟦										
3. Implement the cartpole using PyTorch and IsaacGym														
a. Implement the environement				🟦										
b. Implement the NN					🟦									
c. Bug fixes, thuning and buffer time						🟦								
4. Interface Neuromechfly with the RL environment that I wrote							🟦							
5. Reward engineering														
a. Start with a simple cost function, -> walking distance in 10 secs with average forward moving speed beeing the metric								🟦						
b. Introduce a mechanism encouraging stability									🟦	🟦				
c. Introduce a mechanism that encourages a balanced gait pattern										🟦	🟦			
If time allows -> locomotion complex terrains														
Prepare the midterm presentation						🟦								
Write report												🟦	🟦	
Prepare final presentation														🟦

Figure 5: Impulse Responses

7 Appendix A

The DoFs used were the same as in [2]: ThC pitch for the front legs, ThC roll for the middle and hind legs, and CTr pitch and FTi pitch for all legs

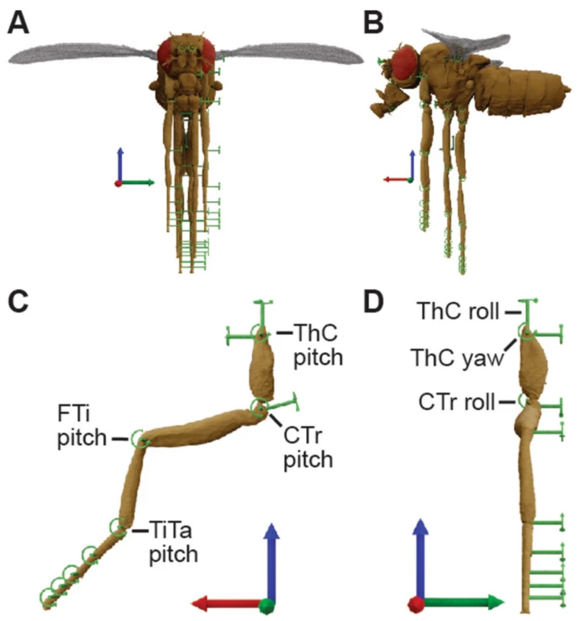


Figure 6: The 'zero pose' of *NeuroMechFly* and its leg joint DoFs [2]