Character Animation Using Reinforcement Learning and Imitation Learning Algorithms

Tokey Tahmid, Mohammad Abu Lobabah, Muntasir Ahsan,
Raisa Zarin, Sabah Shahnoor Anis, and Faisal Bin Ashraf
Department of Computer Science And Engineering
BRAC University

***Abstract*—Real-time character animation for gaming and film industries is challenging and achieving production-ready quality requires a huge amount of time and resources. Animation through marker-based motion capture is quite a tiresome process that requires costly motion-capture suits, multiple cameras, and a large database. In this paper, we propose a model that aims to generate real-time character animation for biped locomotion in Unity ML(Machine Learning) agents using RL(Reinforcement learning) and IL(Imitation learning) algorithms. We first evaluate the training process with solely the state-of-the-art RL algorithm, PPO(Proximal Policy Optimization). Then we analyze the combination of IL algorithms BC(Behavioral Cloning) and GAIL(Generative Adversarial Imitation Learning) in conjunction with PPO. We further discuss the comparison between the two training results and show that our model can generate animations in real-time avoiding all the tedious work and large databases. We demonstrate that our approach is effortlessly easy to implement while maintaining the quality of the animation.**

***Index Terms*—Animation, Reinforcement Learning, Imitation Learning, PPO, BC, GAIL, Unity ML agents**

I. Introduction

IN major animation studios and gaming industries, character animation is done using a marker-based motion capture technique that requires a performer to wear a motion-capture suit. Multiple cameras are required in the room to capture the motion of the performer. After that, the motion capture data is applied to the 3D virtual character through the rigging process. This method of character animation requires a huge amount of time and resources. With the use of neural networks, it is possible to generate animation in real-time without the need for motion capture techniques. The neural network approaches can be divided into two learning approaches. One is supervised learning and the other is reinforcement learning. Many researchers applied the supervised learning approach to various character animation and control projects such as locomotion, kung fu motion, and sports activities as well as for motion retargeting and interpolating key-frame postures [1]. There is a risk that these poses might converge into average poses because of ambiguity. Later on, Holden et al. introduced a special architecture which was called the Phase-Functioned Neural Networks [1] for producing sharp movements. In recent times, there has been a surge in work using deep RL (DRL) to train models for simulated character animation [2]. We further explore this direction for character animation using deep RL (DRL) and IL. In this paper, we propose an approach that aims to generate real-time character animation in Unity ML agents using Reinforcement learning and Imitation learning algorithms. We believe that the combination of RL and IL is the key to solving the character animation problem. In reinforcement learning, the model learns the animation through trial and error experience. This experience is utilized to maximize the rewards which encourage high-level behaviors, making character animation easier to generate. On the other hand, with imitation learning the model learns from expert demonstrations which encourage behaviors to generate human-like animations. Going into this project, we aim to provide a system that generates real-time character animation by combining RL algorithm and IL algorithms. We intend to make use of the Unity ML-Agents Toolkit and its easy-to-implement features.

II. Related Work

The field of character animation has seen some remarkable breakthroughs [3], [4], [1], [2] in recent years due to deep RL methods. There has been a significant increase in simulated characters performing a wide range of challenging tasks [4], [5]. Many have chosen the policy gradient methods for these locomotion tasks [6].

At first, the DeepLoco method proposed a system to add an imitation term to the reward function [5]. Their proposed system demonstrates locomotion tasks by foot-placement goals which are computed by high-level and low-level controllers. They trained both levels of the policy using deep RL. They demonstrated the results on a simulated 3D biped. Later, DeepMimic improves on their work by proposing deep RL methods that can be used to make the agent imitate a good number of reference motion clips [4]. Their method can be applied for extremely dynamic actions as well as keyframed motions. They combined the imitation of motion objectives along with a task objective. Their method maintains the quality of motion capture methods while making it more convenient to use [4]. The Self-imitation paper proposes a self-imitation learning system that enables stable locomotion controllers for rapid learning [7]. Their approach mainly combines the two recent works of continuous control which are FDI-MCTS and DeepMimic [4]. However, their work aims to overcome the limitations of these two methods. These limitations include the high run-time cost of the FDI-MCTS method and the data dependency and sample complexity of the DeepMimic method [4]. In their proposed system, they first generate reference motion using an online tree search method which allows them to use reinforcement learning with RSI(Reference State Initialization) to find a neural network controller to imitate the reference motion [7]. Our approach, similarly to these researches combines deep RL and IL while making the implementation much simpler and easy to use.

III. Overview

Our system uses the Unity ML-Agents Toolkit for training an agent in a simulated environment. We were able to train our model using reinforcement learning and imitation learning with the help of a simple-to-use Python API(Application Programming Interface). The Unity ML agents workflow can be divided into three steps. Creating the environment, training the neural network, and then embedding the neural network in a different environment [8]. At first, we created the environment and the character model with the help of the walker example environment provided by the Unity ML agents. In ML agents, the environment consists of an academy, brain, and agents. The academy is what lets us use other training networks using Unity's python API [8]. The training was done in two different sessions. The first one was done with the state-of-the-art reinforcement learning algorithm, PPO. In the second training session, we used imitation learning algorithms(GAIL and BC) in conjunction with PPO. To better visualize the results in graphs, we used Tensorboard. After that, we analyzed the results of the two training runs.

IV. Algorithms

With the use of the ML agents toolkit, we have access to two state-of-the-art Reinforcement Learning algorithms and two Imitation Learning algorithms. After much research, we have decided to use the Reinforcement Learning algorithm, PPO in conjunction with the two Imitation Learning algorithms, GAIL and BC. In the next sections, we further explain our reasoning for choosing these algorithms and how they work.

A. RL: Proximal Policy Optimization (PPO)

In reinforcement learning, the agent discovers a policy that maximizes the reward [9]. Among numerous RL algorithms, Proximal Policy Optimization(PPO) has proven to be more stable and more general purpose. Hence, we decided to use PPO as our RL algorithm.
According to OpenAI, Proximal Policy Optimization (PPO) performs comparatively better than other state-of-the-art algorithms [10]. Aside from this fact, one of the other reasons why we chose this algorithm is that it is much simpler to implement and tune making it ease-of-use while maintaining good performance. PPO is what makes it possible for us to train the agent in a challenging environment where the agent tries to move to the destination (target object). In this process, it learns to walk and run. Even turn, and face toward the target. In other RL algorithms, a substantial amount of effort is put into tuning the hyperparameters to get a good result. This is where PPO makes it easy to implement. Tuning the hyperparameters couldn't be any easier than in PPO [10].

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{1}$$

Here, in Eq. 1, the algorithm [6], ϵ is a hyperparameter where ϵ = 0.2, is the policy parameter, $\hat{E}t$ is the empirical expectation over timesteps, rt is the ratio of the probability under the new and old policies, $\hat{A}t$ is the estimated advantage at time t. To do a Trust Region update, the objective finds a way to implement it. It is compatible with Stochastic Gradient Descent. Furthermore, by removing the KL penalty, the algorithm is simplified [10]. In various tests, the algorithm has performed quite well.

B. IL: Behavioral Cloning (BC)

In imitation learning, it is presumed that a set of expert datasets is obtainable. The dataset contains a set of trajectories and factors. By implementing these datasets a policy can be learned by direct mapping of the factor to trajectory. This process to form a policy can be put together by using an algorithm called BC (Behavioral Cloning) [11]. Behavioral Cloning is the most straightforward form of imitation learning. In this algorithm, it gathers a set of demonstrations done by an expert, which is D. This is presented as a set of trajectories. Thus for application, a policy presentation π_θ is needed. To compare the resemblance between the demonstrated action and the agent's policy - objective function L has to be selected. Lastly the policy parameters θ has to be returned [11]. This algorithm [11] is considered efficient as it is capable of imitating the expert without having direct contact with the environment. It also takes a short amount of time.

C. IL: Generative Adversarial Imitation Learning (GAIL)

Generative Adversarial Imitation Learning algorithm (GAIL) is a fast and direct approach to drawing an analogy between generative adversarial networks imitation learning (IL) targeted towards boosting performance over imitating complex behaviors in large, high-dimensional environments [12]. In the GAIL framework, a secondary neural network called the discriminator is used which distinguishes the observations/actions between the demonstration and the agent. The discriminator rewards the agent if the observations/actions of the agent get closer to the demonstration. With each training step, the agent tries to maximize the reward and the discriminator gets better at distinguishing between agent and demonstration [13]. In this way, the agent learns a policy that performs states and actions similar to the demonstration resulting in human-like behavior.
The GAIL algorithm is presented by the following expression:

$$E_\pi[\log{(D(s,a))}] \ + \ E_{\pi E}[\log 1 - D(s,a)] \ - \ \lambda H(\pi) \tag{2}$$

In Eq. 2, a parameterized policy π_θ is fitted with weights θ, and a discriminator network D$_\omega$: S$\times A \rightarrow (0, 1)$, with weights ω.

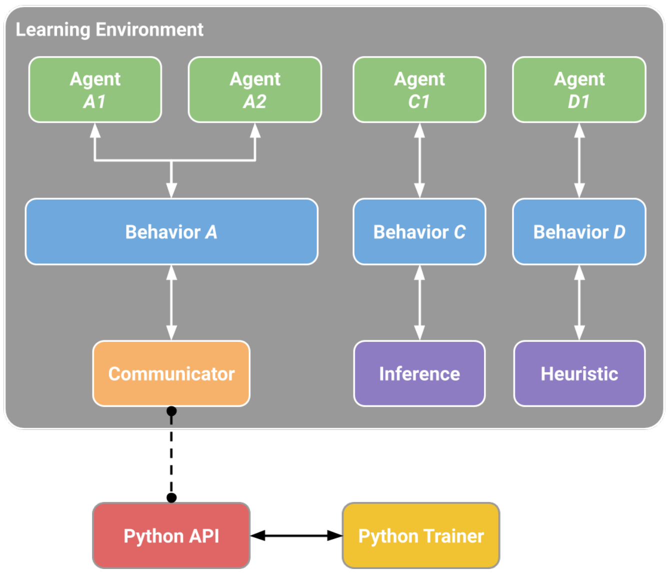

Fig. 1. Block Diagram of ML-Agents Toolkit [13]

V. ML-Agents Toolkit

A new module for the game engine, which is Unity ML-Agents, permits us to establish or utilize pre-made conditions to train the agents. The toolkit is free to use for researchers and developers. While building the simulated environments, researchers and developers make the best use of the Unity Editor with the help of Python API. A set of example environments is one of the main features of the toolkit. Several algorithms are also included in the features of the toolkit. For instance, RL and IL algorithms, PPO, BC, ICM(Intrinsic Curiosity Module), LSTM(Long Short-Term Memory) [8].
There are three key components in the ML-Agents SDK(Software Development Kit). They are the Brains, Agents, and an Academy. The Agent component is the Characters that are used in a scene. An agent has several features. It can collect observations, take actions and receive rewards. The agent will collect observations with the required sensors. A policy with a behavior name is a component of an agent [8].
The agents with the same behavior name will execute the same policy and share experience data during training. Not only there can be multiple agents in the same scenario, but also there can be multiple behavior names within a scene with multiple agents or single agents executing many different behavior types [8]. The block diagram of ML-Agents Toolkit is shown in Fig. 1

The Unity ML-Agents Toolkit contains several example environments. These environments contain examples of single and multi-agent scenarios, with agents using either vector or visual observations, taking either discrete or continuous actions, and receiving either dense or sparse rewards [8].
For building our model, we have followed the Walker example environment. It is a physics-based biped character. The character can freely move 26 parts of its joints. The objective of the agent is to move towards the target without falling over [8].

VI. Reward

ML agents toolkit defines the reward signals in a modular way. It provides two types of reward signals. The extrinsic reward type represents the rewards that are defined by the environment and correspond to reaching a goal. On the other hand, intrinsic reward signals are defined outside of the environment to encourage agent behavior in certain ways and help the learning of the extrinsic reward [13]. So, the reward function r_T is defined by the following:

$$r_T = r_E + r_I \tag{3}$$

Here in Eq. 3, r_T is the total instantaneous reward at any given time step t, which is the addition of extrinsic reward r_E and intrinsic reward r_I.

A. Extrinsic Reward Signal

The extrinsic reward represents the task objective reward for the agent. Here the agent receives positive rewards when it matches the target speed and looks towards the goal direction or touches the goal object and receives negative rewards for touching the ground. The reward signal is expressed by the following notation:

$$r_E = (r_s \times r_l) + r_t + r_p \tag{4}$$

In Eq. 4, the matchSpeedReward r_s calculates the normalized value of the difference of goal walking speed and average speed of the agent. The reward approaches 1 if the speed matches perfectly and approaches 0 as it deviates. The same goes for the lookAtTargetReward r_l where the reward approaches 1 if the agent faces the goal direction perfectly and approaches 0 as it deviates. The expressions for matchSpeedReward and lookAtTargetReward are as follows:

$$r_s = \left[1 - \left(\frac{v_{goal} - v_{avg}}{v_{max}}\right)^2\right]^2 \tag{5}$$

$$r_l = \left(\left(\sum_{i=1}^{n} C_i H_i\right) + 1\right) \times 0.5 \tag{6}$$

Here, in Eq. 5, v_{goal}, v_{avg}, and v_{max} represent goal walking speed, the actual speed of the agent, and max target walking speed respectively. In Eq. 6, C_i and H_i represent the components of the forward vectors of the orientation cube and head body part respectively.
The $\sum_{i=1}^{n} C_i H_i$ notation represents the dot product of the two forward vectors. Next, the touchedTargetReward and the groundContactPenalty are as follows,

$$r_t = 1, r_p = -1 \tag{7}$$

In Eq. 7, the groundContactPenalty is only given when body parts other than the feet are touching the ground.

B. Intrinsic Reward Signal

The intrinsic reward represents the imitation objective reward. For the intrinsic reward, we have used GAIL intrinsic reward. The GAIL reward introduces a survivor bias to the learning process with the method of Discriminator-Actor-Critic. The discriminator reward function is defined as follows [14]:

$$r_I = r(s_T, a_T) + \sum_{t=T+1}^{\infty} \gamma^{t-T} r(s_a, \cdot) \tag{8}$$

Here in Eq. 8, r(s_T,a_T)represents the reward for the final states. With the unbiasing reward functions of DAC, the learned reward r(s_a,·)is added to the returns for the final states instead of just r(s_T,a_T) [14]. For this, the GAIL discriminator can differentiate whether reaching a specific state is the desired behavior from the expert's point of view and give rewards accordingly.

VII. Training

Training of the networks was done in Unity using ML agents. The agents were trained in two different training sessions with two different training configurations. For the first sessions we trained the agent using the PPO algorithm only and in the second session, BC and GAIL were used in conjunction with PPO.
Training proceeds episodically where the agent is initialized at the start of each episode to a random orientation and the target walking speed is randomized for each episode as well. The episode is terminated either when the fixed time frame is over or the termination condition is met which is when the agent falls on the ground.
The hyperparameters for both training sessions are identical with a batch size of 2048, a buffer size of 20480, and a learning rate of 3×10^{-4}. The hyperparameters are shown in Table I. As for the networks, they are fully connected with 3 layers and 512 hidden units.
For the first training session, only extrinsic reward signal was used with a gamma setting of 0.995 and strength 1.0 where gamma is the discount factor for future rewards gained from the environment and strength being the factor by which to multiply the given reward. In the second training session, BC and GAIL intrinsic reward signals are added to the training configuration where both have low strength of 0.1 and the same set of demonstrations. The parameters for GAIL intrinsic reward are shown in Table II.

Hyperparameters	PPO
Batch Size	2048
Buffer Size	20480
Learning Rate	3×10^{-4}
Beta	5×10^{-3}
Epsilon	0.2
Lambda	0.95
Number of epoch	3

TABLE I
HYPERPARAMETERS FOR PPO

Parameters	GAIL
Strength	0.1
Gamma	0.99
Encoding Size	128
Learning Rate	$3.0\times e^{-4}$

TABLE II
PARAMETERS FOR GAIL

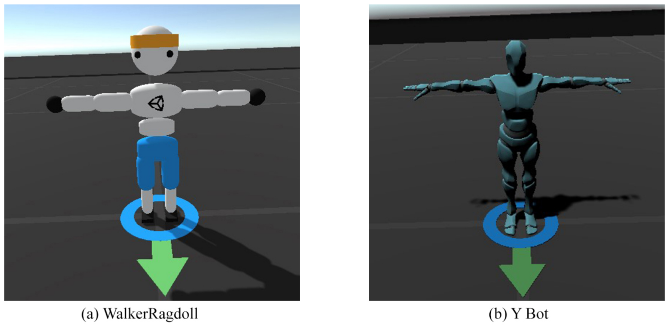
(a) WalkerRagdoll (b) Y Bot

Fig. 2. Character Models

VIII. Experiments

As we discussed before all our training and experiments were done in the Unity game engine. We have also imported character models and reference animations from Mixamo [15].

A. Character Models

We have used two character models for our training. One is Unity's WalkerRagdoll and the other one is Y Bot from Mixamo. Both models were humanoid as we were originally trying to achieve biped locomotion. So, it can be said that any humanoid character model can be used with our method. The portraits of the character models can be found in Fig. 2 as well as the characters performing the task in Fig. 3.

After importing the character model from Mixamo, the ragdoll for the character was set up using the ragdoll wizard of Unity. We had to manually configure some of the colliders and configurable joints to match with Unity's WalkerRagdoll. The rigid-body mass and drag were adjusted accordingly for all body parts. The properties of the character models are shown in Table III.

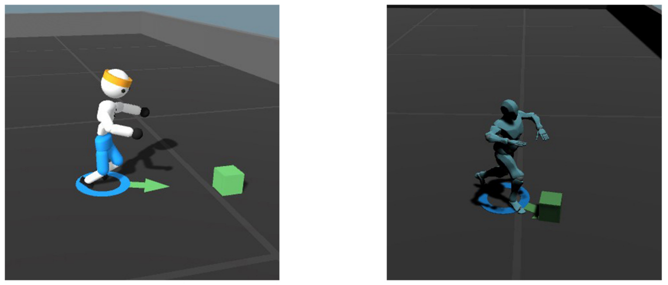
(a) WalkerRagdoll (b) Y Bot

Fig. 3. Character Performing Task

Property	WalkerRagdoll	Y Bot
Joints	16	16
Height (m)	3	3
Total Mass (kg)	100	109
Degrees of Freedom	26	26
Vector Observation Size	243	243
Vector Action Size	39	39

TABLE III
PROPERTIES OF THE CHARACTER MODELS

B. Environments

The training environment consists of a platform as the ground and a cube object as the target. The character model is placed at the center of the platform while the target object is randomly spawned on the platform. The walk speed randomizer randomizes the walking speed from 0.1 to 10 for each episode. This way the agent learns to walk at any chosen speed from 0.1 to 10. The agent along with the platform and target object is duplicated multiple times so that the environment contains 10 independent agents with the same behavior parameters. This speeds up the training process tremendously. We have used both of the character models in the same environment situations to perform the task.

C. Demonstrations

With the reference animations imported from Mixamo, we configured the heuristic function for the agent to record demonstrations. Demonstrations were recorded via Unity's demonstration recorder. We recorded the expert's demonstration for 5-7 mins for a total of 3534 steps. This demonstration was used as a reference for the GAIL and BC algorithm.

IX. Results

We analyzed our training results in TensorBoard, a visualization toolkit of TensorFlow. It is a suite of web applications for inspecting and understanding the training runs and graphs. TensorBoard is designed to run entirely offline on the local machine, without the need for the Internet [16]. TensorBoard provided us with the visualization and tooling needed for our training results. We could track and visualize the model graphs of cumulative reward, episode length, policy loss, value loss, and all policy graphs [17]. As there were two training sessions, we will be looking at two sets of results. On GitHub[1] source code and a supplemental video showing the final results can be found.

A. Training Session I

We only trained the Y Bot model in this session where only the PPO algorithm was used. Table IV shows the cumulative reward and episode length data for completed steps.

Here in Table IV, we can see that for max steps 30000000, the mean cumulative reward is 488.3 and 169.4 is the episode length.

[1]https://github.com/tokey-tahmid/Character-Animation

Steps	Cumulative Reward	Episode Length
15M	395.3	148.5
30M	488.3	169.4

TABLE IV
TRAINING SESSION(I) RESULTS DATA

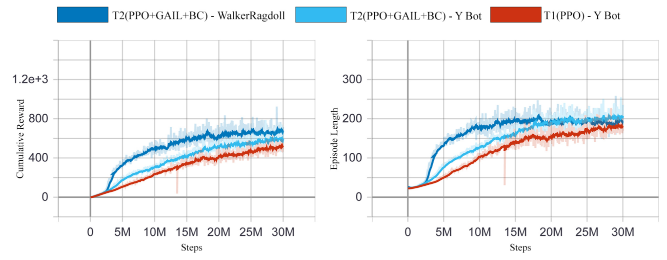

Fig. 4. Cumulative Reward Episode Length

B. Training Session II

Now let us look at Table V for the results of our 2nd training session where GAIL and BC were used in conjunction with PPO. For training session II we have used both character models.

Character Model	WalkerRagdoll	Y Bot
Steps	30M	30M
Cumulative Reward	635.8	678.8
Episode Length	191.3	235.5

TABLE V
TRAINING SESSION(II) RESULTS DATA

Here, we can see that after 30000000 max steps the mean cumulative reward for WalkerRagdoll is 635.8 and for Y Bot it is 678.8. The episode length for WalkerRagdoll is 191.3 and for Y Bot 235.5. Even though the episode length is higher in the 2nd training session, the cumulative reward is also higher than the first training session.

C. Graph Analysis

Now let us look at the difference between the two training sessions with the help of the Tensorboard graphs.

Fig. 4 shows the graphical representations of the mean cumulative reward and episode length. Here, the Cumulative Reward graph represents the mean cumulative episode reward of the two training sessions. The first training session(PPO)

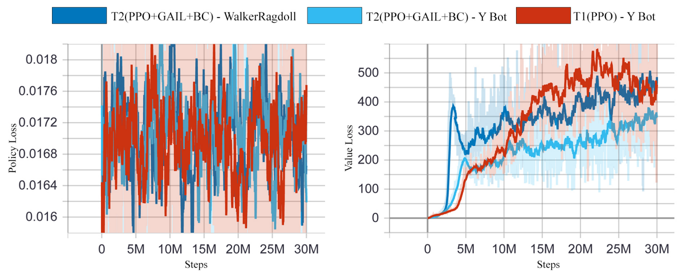

Fig. 5. Policy Loss Value Loss

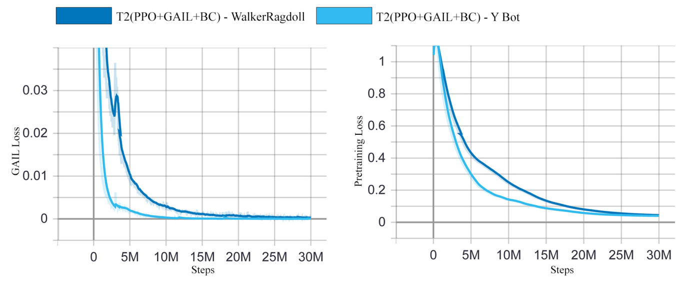

Fig. 6. GAIL Loss Pretraining Loss

is color-coded with red and the second training session(PPO + GAIL + BC) is color-coded with blue. During a successful training session, the graph should be increasing [18]. From the Cumulative reward graph, it can be seen that the second training session generates more mean rewards as it is more increasing. The Episode Length graph represents the mean length of each episode [18]. And here we can see that the episode length is also longer for the second training session.
Fig. 5 Fig. 6 illustrate the graphs for loss functions. The GAIL loss graph is generated only for the second training session as it was not used for the first training session. This graph shows the mean magnitude of the GAIL discriminator loss which tells us the performance of the model in imitating the demonstration data [18]. The mean magnitude of the policy loss function is represented by the policy loss graph which shows us how frequently the policy is changing. On a successful training session, it should be decreasing at the end and we can see that the 2nd training session is decreasing [18]. The pretraining loss graph is generated for BC which represents the mean magnitude of the behavioral cloning loss. And similarly to GAIL loss, this graph shows us how well the model imitates the demonstration data [18]. Lastly, the value loss graph representing the mean loss of the value function update should be increasing during learning and decrease when the reward stabilizes. The graph portrays the performance of the model in predicting the value of each state [18].

X. Discussion and Conclusion

We proposed an approach for character animation that combines Reinforcement Learning and Imitation Learning algorithms. First, we have trained our model with Reinforcement learning only. Although this training session was faster the results were not human-like animation. Then we trained the agents with the combination of RL and IL algorithms. The results this time were far more human-like even though the training time increased. Our method of character animation proves that it is possible to get human-like character animation without the need for costly motion-cap studios and large databases. However, we faced some challenges when it comes to setting up the ragdoll. Our method seems to work with characters that have similar humanoid bone structures and the ragdoll setup needs to be precisely similar as well. Due to time limitations, we also could not fix the arms to move like humans.
With the proper tuning of the RL algorithm PPO in combination with IL and the use of ML agents toolkit, generating character animation in real-time will be easier than ever reducing production time, cost and resources to a very minimum. For future work, we have a modification for the extrinsic reward that is yet to be tested. Finally, We believe that with further research and future developments our approach will perform even better animations making it comparable or better than the state-of-the-art methods.

Acknowledgment

Firstly, all praise to the Great Almighty Allah for whom our thesis have been completed without any major interruption.
Secondly, to our supervisor Mr. Faisal Bin Ashraf sir for his kind support and advice in our work. He helped us whenever we needed help.
And finally to our parents without their throughout support it may not be possible. With their kind support and prayer we are now on the verge of our graduation.

References

[1] S. Starke, H. Zhang, T. Komura, and J. Saito, "Neural state machine for character-scene interactions." *ACM Trans. Graph.*, vol. 38, no. 6, pp. 209–1, 2019.
[2] K. Bergamin, S. Clavet, D. Holden, and J. R. Forbes, "Drecon: data-driven responsive control of physics-based characters," *ACM Transactions On Graphics (TOG)*, vol. 38, no. 6, pp. 1–11, 2019.
[3] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.
[4] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne, "Deepmimic: Example-guided deep reinforcement learning of physics-based character skills," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, 2018.
[5] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–13, 2017.
[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[7] A. Babadi, K. Naderi, and P. Hämäläinen, "Self-imitation learning of locomotion movements through termination curriculum," in *Motion, Interaction and Games*, 2019, pp. 1–7.
[8] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, *et al.*, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2018.
[9] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.
[10] J. Schulman, "Proximal policy optimization," Sep 2020. [Online]. Available: https://openai.com/blog/openai-baselines-ppo/
[11] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters, "An algorithmic perspective on imitation learning," *arXiv preprint arXiv:1811.06711*, 2018.
[12] J. Ho and S. Ermon, "Generative adversarial imitation learning," *arXiv preprint arXiv:1606.03476*, 2016.
[13] Unity-Technologies, "Unity-technologies/ml-agents," Apr 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#imitation-learning
[14] I. Kostrikov, K. K. Agrawal, D. Dwibedi, S. Levine, and J. Tompson, "Discriminator-actor-critic: Addressing sample inefficiency and reward bias in adversarial imitation learning," *arXiv preprint arXiv:1809.02925*, 2018.
[15] [Online]. Available: https://www.mixamo.com/#/?page=1&query=Ybot&type=Character
[16] Tensorflow, "tensorflow/tensorboard." [Online]. Available: https://github.com/tensorflow/tensorboard
[17] "Tensorboard : Tensorflow." [Online]. Available: https://www.tensorflow.org/tensorboard

[18] Unity-Technologies, "Unity-technologies/ml-agents," Apr 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md