ME5406

Deep Learning for Robotics

Project 2: Piano Playing Hand

Sheth Riya Nimish
A0176880R
e0235287@u.nus.edu

Department of Mechanical Engineering

National University of Singapore

Semester I, November 2020

Contents

1. Model-Free Reinforcement Learning

1.0.1 Team Members

Srishti Ganguly [A0170868R]

1.1 Introduction

1.1.1 Problem Statement

In this problem we consider a robot hand with four fingers of different lengths. The objective of the robot is to play the key of the piano specified by the user. There are four possible choices that the user can specify, Key C, D, E or F. Hence, this problem pertains to a single agent which can achieve multiple goals. The action-space is discrete and the agent has 16 possible actions. Each finger has two joints and each joint's angle can be increased or decreased by 1 °. Hence the actions per joint are 2, per finger are 4 and per hand are 16. However, the angles have a lower-bound of $30°$ and an upper-bound of $60°$ (both inclusive). Hence, the number of states is 31^8, represented in the form of a box with discrete values in an 8-dimension space. The agent would obtain a reward of 200 for playing the correct key, a penalty of -25 for playing a wrong key and a penalty of -10 for going out of bounds.

For the render function, we used a top view and a side view to understand the 3D construction in the 2D setting. We later translated this to a CAD model and used pybullet for the 3D physics simulation.

The entire environment was fully coded by us from scratch based on the OpenAI gym framework.

1.1.2 Potential

The increased dexterity of the robot hand through this project, could be applied to various other tasks besides playing the piano. Having a robot hand which has precise control over each finger analogous to muscle contractions in human arms could greatly benefit those with disabilities by making improved prosthetic equipment.

1.1.3 Existing Conventional Methods

Below are some of the learning techniques which have been used in the context of a robot hand:

1. Supervised learning algorithm: This method has advantages such as having absolute control over what the robot is learning. In our case, to humans it is obvious about which fingers would be needed to play which keys and hence a supervised learning model may learn more efficiently through labelled data while an unsupervised model may learn patterns which are redundant. However, the drawback of this method is that it is increasingly time-consuming to collect and label the data and the training needs a lot of computational

time as well [1]

2. Imitation learning algorithm: Imitation learning is practical and easy to understand. However, it can make the robot only as good as the human demonstrations given as input to the robot. Hence, deep reinforcement learning which allows the robot to explore various state-action pairs and predict values even for unexplored states can provide more optimum policies and better results [2].

1.2 Implementation

In the first part of the module, we were familiarised with Q-Learning, SARSA and Monte-Carlo reinforcement learning algorithms. However, considering our current problem we noticed the impracticality of the Q-table which would be of size $31^8 \times 16$. Moreover, we noticed that the above algorithms cannot predict the Q-values of states that it has not explored, hence the time needed to explore every state would be substantially large. Hence, to counter these drawbacks we tried an extension of Q-learning that is Deep Q Networks or DQN.

1.2.1 Deep Q Network

DQN is an off-policy which uses a deep neural network, that is a non-linear function approximator to estimate the Q values.

The neural network structure is fully connected with the input as states and the output as the Q values. The neural network structure that we used in the code is multi-layered dense; it is a 2-layered densely connected neural network. The neural networks were created using Keras, a deep learning API.

For more efficient use of previous experiences and better convergence behaviour, experience replay was used in the code. In experience replay we stored the agents experience at each time step. Then, a random sample of that memory was used to train the network.

The code used by us closely follows the pseudo-code here [3].

Training Process: On executing our agent we realised that the time taken for each episode was extremely high which made training unfeasible. As seen in Figure 1.1, the average time for each episode was about 0.56 hours with peaks of about 2 hours.

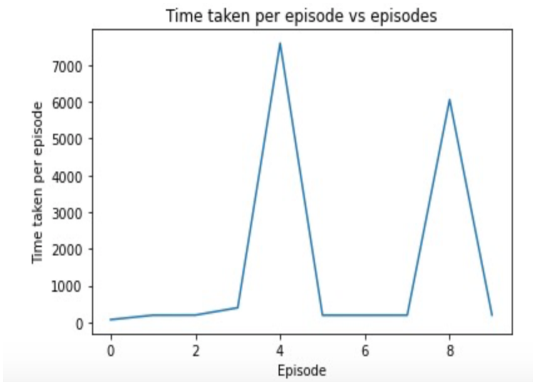

Figure 1.1: The time taken in seconds per episode vs episodes

After obtaining this graph in Figure 1.1, we realised that the huge state-space could be a cause for the slow performance. The experience replay requires a lot of memory which makes the process time consuming. Retrieving and propagating information from the memory irrespective of the batch size made the algorithm slower. Hence, we decided to overcome this difficulty by using a faster algorithm:the Actor-Critic Method.

1.2.2 Actor-Critic Method

The Actor-Critic Model (ACM), which is an on-policy method using a temporal difference (TD) algorithm, was used to train the agent. As seen in Figure 1.2, the actor contains the policy, Π_s and controls the actions taken by the agent while the critic contains the value function $Q(s, a)$ and critiques the action taken by the actor in the form of a TD error . The policy parameters in the code were initially updated by stochastic gradient descent (SGD).
The code used was adapted from the algorithm here [4].

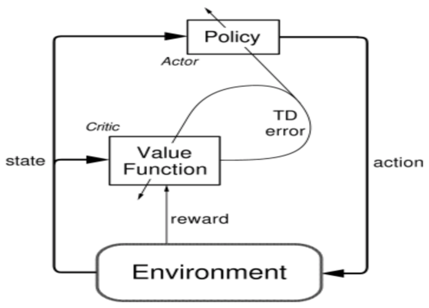

Figure 1.2: The time taken in seconds per episode vs episodes [5]

The neural network structure that was used here is a dense fully connected layer to implement actor and critic. The first layer is common to both actor and critic, while the second layer for both are different. This was created using Keras as well.

In the algorithm, we used GradientTape API in tensor flow because of the

automatic differentiation property which is useful for back-propagation, an efficient way to calculate the gradient of the loss function with respect to the weights. Since normalized vectors improve the performance of the back-propagation algorithm, we did so in our code.
After the implementation of ACM, we obtained results in a short period of time. However, we aimed to fine tune the parameters so that our agent trained better.

1.2.3 Optimization

Firstly, we tried to change the learning rate, which is the amount that the weights of the neural network are updated after each episode based on the approximated losses and errors. From literature review, we found out that the value of the learning parameter should be a small value ranging from 0.0 to 1.0 and also learnt the importance of choosing the correct value since a higher value may result in unstable training while lower values can fail to train the agent [6]. Hence, we tried out 5 different values (0.1, 0.2, 0.3, 0.4 and 0.5). A problem that we had been facing is the convergence of the training to a sub-optimal solution (going out of bounds and getting a reward of -10) which occurred when the training parameters was larger; hence, learning parameters from 0.5 to 1.0 are discarded in the analysis. From the Figure 1.3, the dark green line which was for the training parameter of 0.1 gave the lowest losses, hence that was chosen. The reward graphs fluctuated considerably hence it was not possible to make a definite conclusion from that graph.

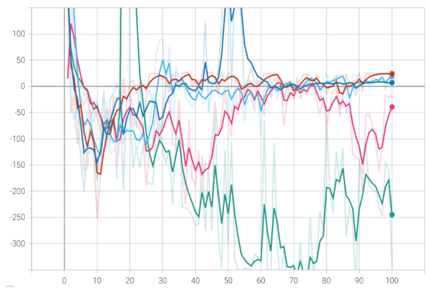

Figure 1.3: The epoch average loss for different learning parameters

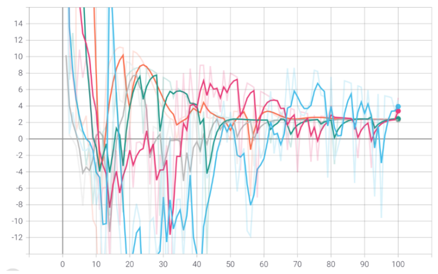

Figure 1.4: The epoch reward loss for different learning parameters

The next parameter we optimised was the loss function that is used to evaluate the loss in the current episode so that the weights can be changed to decrease the loss in the next episode. Keras provides a myriad of different loss functions. However, due to computational time constraints it was impossible to try all. We tried one loss from each of the following categories: Probabilistic Loss (KL Divergence), Regression Loss (Huber) and Hinge Loss (Square-Hinge Loss). The Huber-Loss function (orange line) in Figure 1.5 gave the best performance. The loss results seemed more consistent than the reward results and hence were used for the analysis.

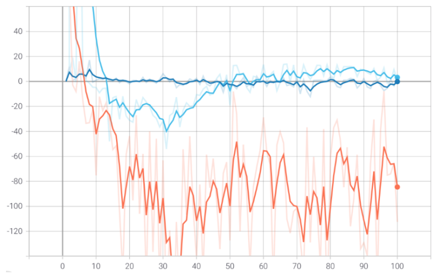

Figure 1.5: The epoch average loss for different loss functions

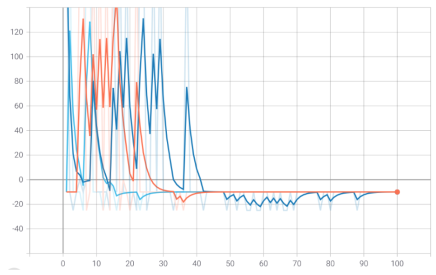

Figure 1.6: The epoch average reward for different loss functions

Keras provides a plethora of optimizers, which are the connection between the loss function and model parameters. We tried the following optimizers to check which one gave the best performance: Adam, SGD, Adagrad and RMSProp. The SGD optimizer is a stochastic gradient descent optimizer. The rest are adaptive learning optimizers which train the agent more efficiently and propel a faster convergence rate. Hence, the following optimizers were used for testing. The best results were obtained for RMSProp (light blue) in Figure 1.7.

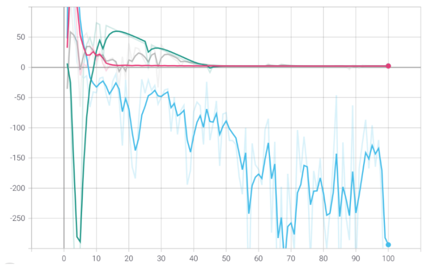

Figure 1.7: The epoch average loss for different optimizers

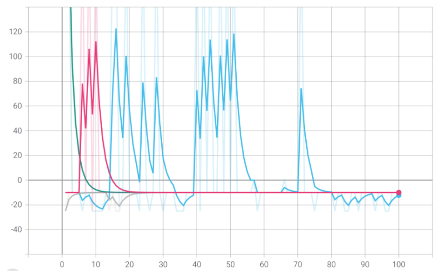

Figure 1.8: The epoch average reward for different optimizers

Next we decided to vary the number of units in hidden layer (2, 4, 6, 8, 12, 16). On further studies, we realised too few layers will lead to high training and errors due to under-fitting, while too many may lead to low training error but a high generalization error due to over-fitting [7]. Hence, we tried the above range and obtained the best result for 2 layers (dark green) as seen in Figure 1.9. We noticed that the results with higher number of hidden layers which is supposed to give a more accurate estimation worsened the problem of choosing a local optima over a global optima.

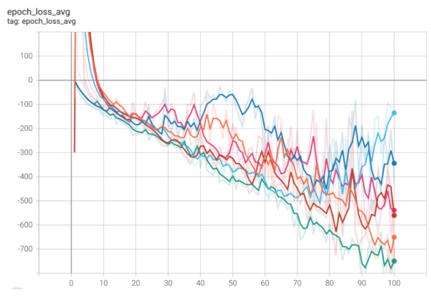

Figure 1.9: The epoch average loss for different hidden layers

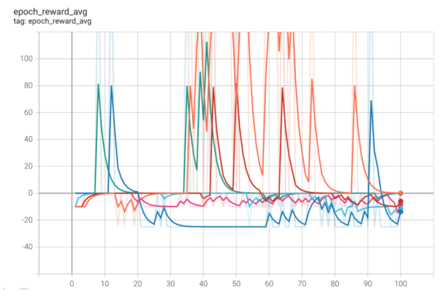

Figure 1.10: The epoch average rewards for different hidden layers

1.2.4 Results

The results below in Figure 1.11 and Figure 1.12 show a typical result of the average epoch loss and average reward loss for the Actor-Critic method using the optimization described in the section 1.2.3.

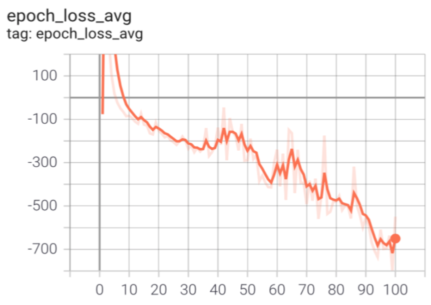

Figure 1.11: A typical average epoch loss for ACM

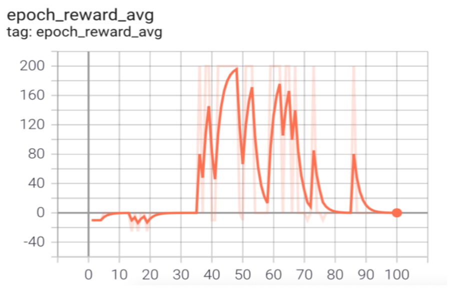

Figure 1.12: A typical average epoch reward for ACM

Note: In the above graphs, we see that the loss function is often negative. Although this initially worried us, we realised that this is normal behaviour for the actor critic method. Furthermore, the rewards graph showed a great degree of randomness and variance in the results.

1.3 Comparison with Conventional Algorithms

1.3.1 PID

To compare our algorithm's performance to classical control algorithms, we analysed the performance of the Proportional Integral Derivative Controller (PID Controller) which is a control-loop feedback mechanism. In the algorithm that we used in our code which has been adapted from this source [8]. Here, the sigmoid function was used to limit the variability of the coefficients of the PID control.
However, we realised that this method is unsuitable for application in our problem. We realised that the PID algorithm repeatedly choose the same action and not enough exploration was done as seen in the action distribution graph below, Figure 1.13. Consequently, the rewards generated by PID were not good either.

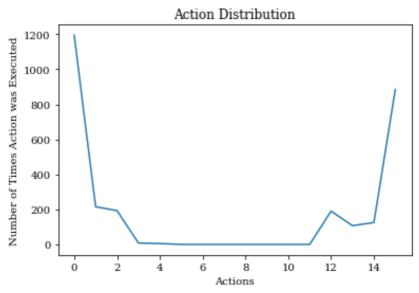

Figure 1.13: Action Distribution for PID

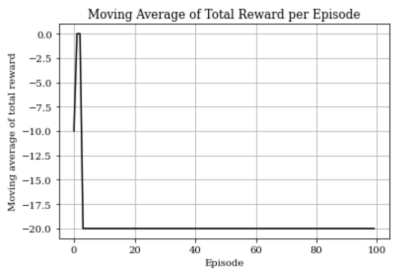

Figure 1.14: Moving Average of Rewards for PID

1.3.2 SARSA

The second conventional reinforcement learning algorithm that we compared our performance to was SARSA, a model-free on policy which uses temporal difference to approximate the action value function and obtain the best policy. The update rule for SARSA is: $Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$. The term $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ is the estimated target value. The code for SARSA has been based on the psuedocode provided by Prof. Chen. Since SARSA uses a Q-table and a Q-table of magnitude 31^8 is not possible, we decided to train only one finger (which makes the state-space 31^2)and test the results. The results, while subjected to a lot of random nature, often gave excellent results as seen in Figure 1.15.

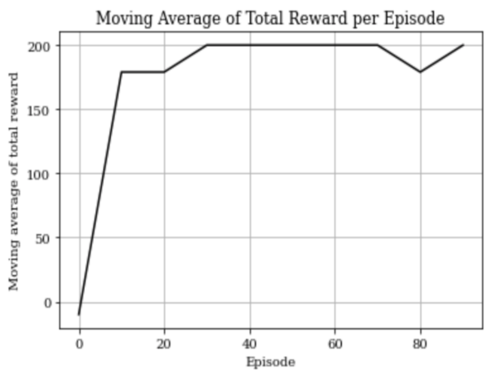

Figure 1.15: Moving average of rewards with one finger using SARSA

The question arises about why we did not train all four fingers with SARSA itself and get a model for each of the finger making it separate agents. However, because of the complexity of the implementations of multi-agent SARSA and the difficulties associated with it (credit assignment, moving target problem, global exploration and so forth) [9], we did not proceed with that idea.
Hence comparing our currently used method to the above state-of-the-art algorithms, we realised why deep learning was the wisest and most practical approach to solving the given problem.

1.4 Validation

Positives: We were able to get a validated model for goals C and D using the actor critic method. The actor critic method also did not take much time for computation.
Negatives: We were unable to obtain suitable weights for the goals E and F since keys E and F were intuitively harder to reach due to a combination of link movements required. One of the key difficulties we faced was the repeated choice of the same action in the weights for E and F. This was baffling since our policy was stochastic and not deterministic. However, with better training we hope to understand why this occurs and overcome the problem.

1.5 Challenges and Improvements

One of the major problems that we faced during this project is the convergence to a local maximum. Often the key would converge to a solution where it would always go out of bounds and obtain a penalty -10, because that is a local maximum compared to the case when the agent plays the wrong key which gives it a penalty of -25. We realised that gradient descent only assures minimization of the network output error but does not guarantee a global optimum solution. Hence, we would like to improve the convergence problem by creating more models and using an ensemble method or a dropout model [10].

The second problem that we faced was that the time taken to converge was very large. This was because our problem was very complicated with a lot of states and to train an agent the number of epochs needed to be quite large. Using a GPU, we hope to solve this difficulty.
Initially we had a state space of size 180^8 and a reward structure of $+1$ for playing the current key, and -1 for playing the wrong key and no penalty for going out of bounds but just keeping the link consistent(for instance if it is 30, and it tries to decrease the angle it remains 30). However, we faced major difficulties of our agent not being able to achieve good results and being unstable. One of the problems we detected was because our reward structure was very sparse which makes the model predict 0 most of the times and rightfully so. Hence, we decided to improve our reward structure to the current one. This showed a significant improvement in the results. Furthermore, we reduced the state space so that each epoch could terminate faster and train the agent quicker.
We were initially rendering in a 2D setting. However, when we extended it to a 3D scenario by using pybullet, we realised issues pertaining to the physical possibility of the model. In our current model, the tip of the finger needs to touch the piano to play the key, while the rest of the body is allowed to be within the piano key's rectangular 3D space. This does not make sense practically. Hence, in the future we would like to put additional constraints on the movement of the fingers such that the agent cannot proceed within the piano key's 3D space but just touch the surface. Furthermore, we were faced with challenges pertaining to the orientation of the keys and the robot hand in pybullet. However, with better cadding and understanding of pybullet those issues can be resolved.

We also faced the issue of our agent choosing the same actions over and over . Hence, adding an element of increased randomness would help us solve this issue. We would love to explore the technique of adding entropy to the loss to encourage exploration.

1.6 Future Work

The possibilities from this project seem endless. Firstly, we would like to implement better algorithms such as Advantage Actor Critic (A2C). In A2C, the agent learns the advantage values and not the Q values which creates a more stable model [11]. We would also like to try the Asynchronous Advantage Critic (A3C) by making multiple independent agents (each finger being a separate agent).
In the future we would like to extend our environment to give more degrees of freedom to the finger motion. Currently, each finger can only move in a 2D plane. Moreover, we would like to incorporate in the solution other physical parameters such as velocity and gravity to better simulate a real life situation. Increasing the number of joints would be another improvement we would have liked to make. Making the state-space continuous would thereby increase the smoothness in the performance; this would be another goal we

would like to accomplish. While we tried exploring starts by giving the agent random angle position of the finger to better model real life scenarios, it did not give a good result. However, in the future we would like to debug this and train our model better.
A very ambitious possibility would be the integration of audio sensors to be able to recognize the sounds played and then replicate them by playing it on the piano. Musical techniques such as stocatto could also be taught to the robot. In the distant future, having a neural interface that uses the brain impulses to control the robotic hand can be employed.

1.7 Self-Reflections

Through this course, my teammate and I have learnt to code an environment based on the OpenAI gym from scratch. We have learnt the importance of unit tests while making an environment because we realised the many tiny typos and errors we had made while coding the environment. Also, we appreciated the value of good variable naming conventions. In our code, our lower bound and upper bound was initially 0 to 180. However, after some consideration when we needed to change it to 30 and 60, we had to manually change it in all locations. Hence, this was a lesson to use better programming conventions.
We became better acquainted with various deep reinforcement learning algorithms as well as the advantages and disadvantages of these algorithms. Hence, in the future I feel better equipped to decide which algorithm would be suited for which problem.
In this course, I learnt the theory of the algorithms and got a gist and intuition of the proofs. More importantly, I have learnt the implementation of the theory. I feel better prepared to tackle other deep learning in robotics problems in the future.

Bibliography

[1] A. Nair, “Combining self-supervised learning and imitation for vision-based rope manipulation,” 2017. [Online]. Available: https://arxiv.org/pdf/1703.02018.pdf

[2] I. Fadelli, “An imitation learning approach to train robots without the need for real human demonstrations,” *Tech Xplore*, 2019. [Online]. Available: https://techxplore.com/news/2019-11-imitation-approach-robots-real-human.html

[3] Kung-Hiang, “Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg),” *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

[4] A. Nandan, “Actor critic method,” *Keras Example*, 2020. [Online]. Available: https://keras.io/examples/rl/actor_critic_cartpole/

[5] “Actor critic methods,” 2017. [Online]. Available: http://incompleteideas.net/book/first/ebook/node66.html

[6] J. B. Lee, “Understand the impact of learning rate on neural network performance,” *Machine Learning Mastery*, 2020. [Online]. Available: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

[7] “Model selection, underfitting, and overfitting,” *Dive into Deep learning*. [Online]. Available: https://d2l.ai/chapter_multilayer-perceptrons/underfit-overfit.html

[8] H. Jia, “Solving openai’s cartpole with a very simple pid controller in 35 lines,” 2020. [Online]. Available: https://gist.github.com/HenryJia/23db12d61546054aa43f8dc587d9dc2c

[9] Christina, “A brief summary of challenges in multi-agent rl,” 2020. [Online]. Available: https://christinakouridi.blog/2020/01/02/challenges-multiagent-rl/

[10] B. Blagojevic, “Reinforcement learning with sparse rewards,” *ml everything*, 2018. [Online]. Available: https://medium.com/ml-everything/reinforcement-learning-with-sparse-rewards-8f15b71d18bf

[11] S. Karagiannakos, “The idea behind actor-critics and how a2c and a3c improve them,” *AI Summer*, 2018. [Online]. Available: https://theaisummer.com/Actor_critics/