Getting on with Life(Steps): a simulation based on daily life decisions

Daniel Bernardi
Alma Mater Studiorum - Bologna, MSc in Computer Engineering
daniel.bernardi@studio.unibo.it - daniel_bernardi@outlook.it

Abstract

This paper describes the implementation of a Reinforcement Learning agent using the Proximal Policy Optimization algorithm [Schulman et al., 2017]. The agent interacts with LifeSteps, a customized Gymnasium Environment that simulates various human daily decisions and their impact on life. LifeSteps was developed specifically for this report, alongside the algorithm's implementation.

1 LifeSteps

LifeSteps is a single-player game where the player, or an agent, must make decisions on each timestep. Each action taken by the player has consequences that affect their overall progress in the game, requiring strategic thinking to avoid losing. The objective of the game is to keep the player alive until the end of the simulation, which goes on for a fixed number of timesteps.

LifeSteps offers two game modes: a simple *standard* mode and a more challenging mode called *monopoly*.

1.1 The standard gamemode

Since the *monopoly* gamemode is an extension of the *standard*, all of the details written here will be true also for the next section.

The **life** of the player is the main information contained in each observation. It encodes the quality of three parts of the player's life: it's *money*, it's *health* and it's *sociality*.

There is also another value, called **friends**, which tells if the player has at least one friend. This value doesn't have neither good or bad implications in this gamemode, but it will be important in *monopoly* episodes.

The **actions** that the player can perform are three: go to *work*, do some *sport*, enjoy a *social* occasion. Each action is linked to a part of the state. When performing an action, the corresponding life's value will receive a bonus. The other two components will decrease in a deterministic way. The difficulty of this simulation is on making the right choices to avoid death ($\Leftrightarrow$ game is lost), which can happen at any time, when money or health reach the value 0.

For example, doing sports for a timestep will give a 7 points bonus to the health value, but at the same time the money and social values will decrease of respectively 5 and 1 points.

The **reward is given to the player only at the end of the simulation**. The simulation has a fixed length. The reward is calculated using this formula *if the player didn't reach the end of the game*:

$$reward = max\ ts - current\ ts - difficulty$$

Preprint. Under review.

If the player accomplished to reach the end of the game, but didn't win, which means that at least one of the life's values is below the difficulty level, the reward is:

$$reward = min(life[i]) - difficulty, i = [money, health, sociality]$$

with $life[i]$ being the values about money, health and sociality, and *difficulty* being an initialization parameter of the environment. *If the player **wins** the game*, which means that all the life's components have a value higher than the difficulty parameter, the **reward is 1**.

1.2 The monopoly gamemode

In the Monopoly game mode of LifeSteps, similar to the popular table game, players may encounter chance cards that can have negative effects on the game's progression. In LifeSteps, these **chance cards** represent events that can happen with a certain probability p. After each timestep, the probability p increases, except when a chance card is drawn, in which case the probability is reset to zero. When an event occurs, a **points deficit** is applied on the player's health or money, complicating the game.

The possibility of encountering chance cards events is balanced by another game mechanic, which has no effects in the standard mode: the player's **friends** state. This information is represented by a boolean value, and it can change only in one direction, from False to True, when the *social* development of the player goes above 40 points.

Having a friend in this gamemode provides a significant reduction in the penalties imposed by chance cards events. This features adds variety to the game.

Summary of game settings and behaviour Life's components (money, health, social) are sampled in the range $[25, 35)$ at each environment's reset. Friend's starting value is always 0. The difficulty level is chosen by the user when creating the environment. In the monopoly gamemode, the penalty in life values in the chance of a troubled event is 10 if the player has no friends, otherwise is sampled in the range $[0, 4]$. The probability p is 0 at episode's start, then grows of 0.03 at each timestep, until an event comes up. The next life's state is always calculated adding 10 points to the value linked to the last action performed. Also, at each timestep, life's values decrease: money, health and sociality lose respectively 5, 3 and 1 points.

1.3 An example of a LifeSteps simulation

To better describe how the game works, in Figure 1 there is an example of the first steps of an episode rendered in text mode.

M, H and S are the three life statistics values: money, health, sociality. The F value describes if the player has some friends. The last column tells if some trouble happened during the last step, indicating also how many points were lost. If the player loses pennies, the deficit goes to the money statistic, if the player get's hurt, the deficit goes to its health.

Let's take into consideration the second and third line. The selected action is sociality, so:

- the M value decreases of 5 points ($\Leftrightarrow 0 - 5$);
- the H value decreases of 3 points ($\Leftrightarrow 0 - 3$);
- the S value increases of 9 points ($\Leftrightarrow 10 - 1$);

(0,0,10) is the increase on the state for that timestep (since sociality was selected), (-5,-3,-1) is the constant decrease of the state at each timestep.

In addition to that, the S value is now above the threshold value (40), so the player finally acquires a friend.

2 Implementation

This section will discuss the main parts related to the implementation of the architecture of the neural networks, the structure of the project files and the Jupyter Notebook.

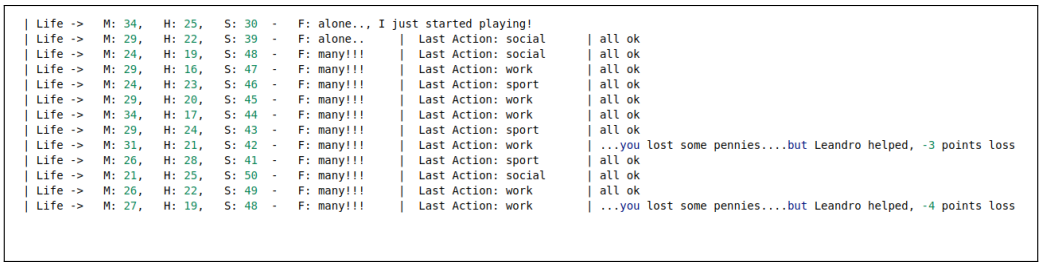

Figure 1: The start of an episode in monopoly mode

2.1 Project files

The project is divided in multiple files. The *agent.py and memory.py* files implement respectively the algorithm and the necessary memory structures. The notebook *final.ipynb* uses the previous files to perform different training steps and showing the results.

The *gym_projects* directory contains everything related to the custom environment creation. The code of LifeSteps can be found in the *lifesteps.py* file.

Since the notebook was executed three times with different seeds[1], each run was saved into a different file. These files can be found in the *notebook-runs* directory.

Tensorboard was used to log useful informations about the training. The logs directory is *resources/logs/scalars*. Additionally, the outputs of the evaluations' prints of the notebooks were extracted into txt files inside the *resources/logs/text* directory.

The *models* directory contains the main checkpoints of the deep learning models.

2.2 Notebook: final.ipynb

The notebook has been executed three times, each one with a different seed. At the beginning of the notebook, the same seed for each run is set for the Numpy library, with the purpose of collecting a fixed list of evaluation seeds. As a result, the environments on which to perform the evaluation steps are the same for each run. The notebook performs 4 main Steps:

- Step 1: Training and evaluation in the standard game mode with a difficulty level of 40.
- Step 2: Fine-tuning and evaluation of the models from the previous step in the monopoly game mode with a difficulty level of 40.
- Step 3: Training and evaluation of new models, using the settings from Step 2.
- Step 4: Training and evaluation of new models with more units in the hidden layers, using the settings from Step 2.

The code was executed on a laptop with an Intel® Core™ i7-8565U CPU and an Nvidia MX250 2GB GPU.

2.3 Deep learning architecture

The Actor-Critic architecture was implemented using two separate neural networks ($\rightarrow$ no weights sharing) with identical topology, except for the output layers. Each network consists of two hidden dense layer with 32 units each and hyperbolic tangent activations, followed by output heads with linear activations. In Step 4, the number of units was increased to 128 for both models. The notebook follows the fundamental characteristics of the original paper while incorporating some changes, which are described in comments.

[1] *final.ipynb* refers to the run with seed 14, and it's the only notebook with significant comments

Table 1: Training results

Model	GM	Units	LR	c_2	Seed	Updates	Evaluation
Step 1	std	32	2.5e-4	0	14	350	**1.0**
					77	350	0.92
					39	350	1.0
Step 2	mon	32	1e-4	5e-3	14	900	-1.14
					77	2343	-2.04
					39	2343	**0.66**
Step 3	mon	32	2.5e-4	5e-3	14	2300	**0.72**
					77	1150	0.66
					39	3500	0.40
Step 4	mon	128	2.5e-4	5e-4	14	1750	**0.86**
					77	2300	0.86
					39	1250	-0.96
random-bs	std				14		-122.6
random-bs	mon				14		-123.4

3 Results

Table 1 summarizes the results of the evaluations for each step and seed, specifying some of the hyperparameters used. The timesteps length of an episode was set to 100 and the difficulty level to 40. At each step, the training stopped after six consecutive evaluations that returned a cumulative reward higher than 0.

The number of Updates in the table refers to the lenght of the training at the time of the early stopping[2].

A note on the evaluation values The values in the Evaluation column are the mathematical mean between the cumulative rewards of 50 episodes. The unbalanced nature of the rewards in LifeSteps should be taken into consideration when discussing the models' performances.

4 Discussion

The PPO Agent solves easily the game in standard gamemode and difficulty 40, with only one lost game between the three runs.

Step 2's goal was to understand if, starting from pre-trained networks, the training time to play a similar game could be reduced. The results suggest that fine-tuning the neural networks in this particular case does not result in a good evaluation score, most of the time.[3]

This result isn't that much of a surprise. Since the state is encoded into a 4 values array of integers, the features that the networks need to learn are very simple. On the other hand, the policy to win a monopoly episode must be significantly different, and focus on boosting the sociality score from the start to reduce the penalties of chance cards.

Step 3 proves that the monopoly gamemode can be solved, even if the training time is longer than for the standard gamemode. The results are encouraging, the lost games usually where very near to be won by the agent. Another valuable information of this step is that a brand new untrained PPO Agent can perform better, with much less training, than a fine-tuned one taken from Step 2.

There is not a lot to say about Step 4, the results are positive but similar to Step 3, even with the significant increase in hidden layers' units.

[2]Step 2 is the only one that starts from pre-trained neural networks, so even the number of updates of Step 1 should be considered.

[3]even when fine-tuned models have a decent score, the training time is very long

Table 2: Agent's behaviour during the evaluation steps.

gamemode	seed	work (%)	sport (%)	sociality (%)
standard	14	54.8	32.8	12.4
monopoly	14	53.4	34.5	12.0

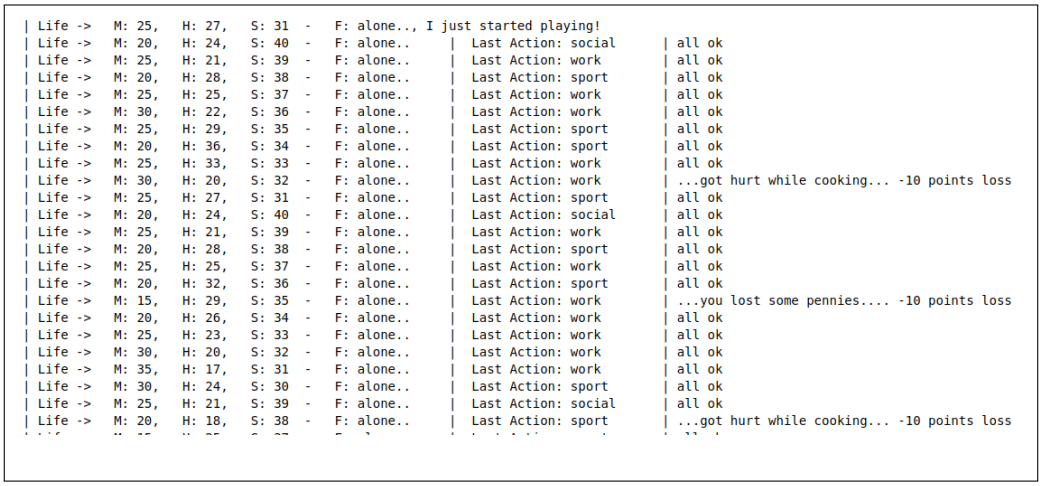

Figure 2: A failed monopoly game

4.1 A deeper analysis

During each evaluation step, some data about the model's behaviour was collected. To be more specific, it was considered useful to store the overall percentages of chosen actions. This data is described in Table 2 and the values are taken from the best performer of each gamemode.

This data is much informative because it's not fine grained, but in a very simple way it describes the overall behaviour of the agent. In the monopoly gamemode, there is need of a more balanced policy for the selection of work and sport actions, to avoid losing the game. Sociality doesn't change much between the two gamemodes because the difficulty level is set to 40, which coincidentally is the threshold to get a new friend.

A standard behaviour for the agent in monopoly mode is to improve the social score during the first timesteps, as seen in Figure 1. If this is not accomplished the game would be too hard to beat, as in Figure 2. In this example (step 4, seed 39, eval seed 40587), the agent fails to perform enough social actions in the first steps and acquire a friend, which would decrease the health and money losses after chance card events. Since the maximum deficit for a chance card while having a friend is 4, it's easy to see how, with social actions during the two first timesteps, the agent would have lost at least 18 points less.

Additionaly, the policy does something strange at the end of a monopoly episode. As can be seen in Figure 3 (step 4, seed 39, eval seed 75763), the player has a very high social score at the end of the simulation. The actual social score needed to win the game (in that difficulty level) was 40. On the contrary, the money score is 43, so it seems like the agent takes an unnecessary risk on developing sociality more than it's actually needed.

A more robust behaviour can be observed at the end of a standard episode, which at the end of the game results in a more balanced life score.

To understand more the behaviour of the agents it's advisable to read the text logs of the evaluations.

The Tensorboard logs will not be exposed in the report because, even if useful to find problems during the design and programming of the project, they don't give out much useful informations at the end.

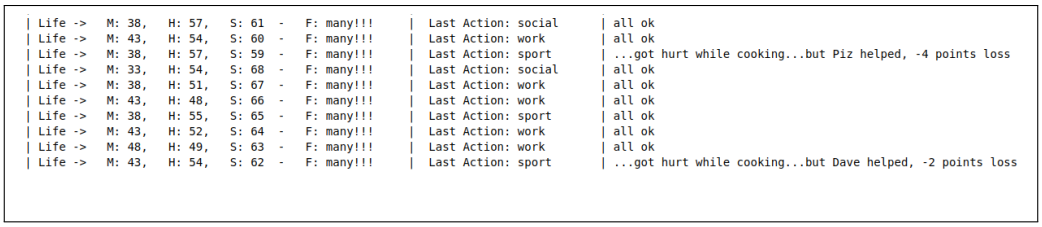

Figure 3: Unbalanced life scores at the end of a monopoly episode

5 Conclusions

The LifeSteps environment works correctly and the implemented agent is able to solve the game without many errors. When the game gets difficult, e.g. in monopoly episodes, the agent can still win games, but the behaviour seems to be risky and the strategy isn't optimal. Obviously the agents could've been trained more and maybe reach perfect scores even in monopoly gamemode, but that level of performance was not the goal of the project.

Using vectors of environments proved to be very useful to shorten the length of the training, along with the possibility to run many parallel *notebooks*. The agents for the monopoly gamemode would have been very hard to train with a single environment.

The agent was used (with very little code tweaks) with other environments (e.g. CartPole) to be sure that the algorithm worked well even in different conditions.

The performance of the trained models seem to be dependend on the seed. This may be for many reasons such as the randomness and difficulty of the game, or the lack of enough exploration to avoid getting stuck in a local minimum. An exhaustive search of optimal hyperparameters (e.g. the entropy bonus) may lead to better performance.

6 Useful links and references

Project code: github.com

The following arcticles were read during the debug process.

The 37 implementation details of PPO was an interesting read for understand details about why to use certain activations, weights network initialization, advantages and returns normalization: iclr-blog-track.github.io

The *CleanRL PPO implementation* was useful for checking the correctness of my code: github.com

This repository suggested the idea of dividing the code into more files, simplifying the debug and test activities: github.com

The report was written in english using Overleaf (overleaf.com) and the phrasing corrected using ChatGPT (openai.com)

References

J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL `http://arxiv.org/abs/1502.05477`.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

A Appendix

A.1 Background

A.1.1 Gymnasium

Gymnasium is an API for Reinforcement Learning that offers a wide range of environments for training and evaluating RL algorithms. It includes support for incorporating new custom environments into the local installation. This feature is valuable because once a custom environment is registered in the local database of environments, it becomes readily usable like any other environment. This guarantees standardized usage and allows for the utilization of wrappers and utilities provided by the Gymnasium library.

A.1.2 Proximal Policy Optimization

The implementation of the Proximal Policy Optimization (PPO) algorithm [Schulman et al., 2017] is a central component of this project, and this section aims to provide background information on it. Since some implementation details of the algorithm can vary, the explanations in this section will provide just enough background for discussing the project's specific implementation. Further informations can be found in the original paper.

PPO builds upon the concepts of Policy Gradient methods and Trust Region methods.

Policy Gradient methods involve computing the policy gradient, which can serve as input for a stochastic gradient ascent algorithm. Leveraging existing libraries that perform automatic differentiation, this gradient estimator can be obtained by differentiating an objective function using such software.

Trust Region Policy Optimization (TRPO), introduced by Schulman et al. [2015]), adds a constraint on the size of the policy update. Alternatively, instead of using a constraint, it is possible to incorporate a penalty into the objective function.

PPO implements the idea of a constrained policy update of TRPO in a simple way, based on the following formulas:

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

With $r_t(\theta)$ being the ratio between the new and the old policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)}$$

And $\hat{A}_t$ being the advantage function.

The advantage function implemented in PPO is a truncated version of **Generalized Advantage Estimation** [Schulman et al., 2018]:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$
$$where\ \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

In conclusion, the objective proposed by PPO is:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

with $L_t^{VF}(\theta)$ being the mean-squared error loss between the estimated and the target value function, $S[\pi_\theta](s_t)$ being the entropy of the policy on a given state, and c_1,c_2 two coefficients (hyperparameters).

The algorithm proceeds with a series of iterations decided by the designer. In each iteration, it collects data by running the policy in the environment and stores (a **batch** of) the observed states, taken actions, received rewards, and other relevant information. It computes advantages and returns

using the collected data. The policy is then updated by optimizing a surrogate objective function. This process is repeated for multiple epochs at each iteration to refine the policy.

This was a summarized review of how PPO works. A more detailed description of the algorithm can be found on the original paper. What has been written here will be basis for discussion in section A.2.1.

A.2 Learning

A.2.1 The training loop

Gymnasium provides the possibility to create a vector of multiple environments. This feature was utilized in the notebook by creating an AsyncVec (asynchronous vector) consisting of 32 LifeStep parallel environments. Even if not strictly necessary, this decision accellerated the training process in a considerable way, especially in the monopoly gamemode where many epochs of training were needed to approximate a satisfactory policy.

The algorithm works by optimizing the objective for each **minibatch**, which is a portion of the total **batch** of (batch_length * num_environments, in this case 128*32) observations. The training loop iterates for a specified number of **updates**, which depends on the number of parallel environments and the maximum duration of the training (measured in timesteps). For each update, 128 actions are selected for each environment (serialized process w.r.t. the timesteps, parallelized process w.r.t. the environments), forming a batch of $128 * 32$ sets of informations stored in Numpy NDArrays (observations, rewards, terminations, truncations, state-values). Then, the rewards are normalized and the advantages and returns are calculated for the whole batch.

The batch gets shuffled and subdivided in minibatches, for each of those we perform the optimization step for a specified number of epochs.

Some characteristic of the implementation:

- The learning rate and the ϵ clipping parameter of PPO are linearly annealed from their starting value to 0 at the end of the training.
- Orthogonal initialization of the weights in the neural network.
- Reward scaling to the $[-1, 1]$ range.
- Advantages normalization.

The **advantages** are calculated for all the steps in the batch. The **returns** are calculated as the discounted sum of the actual rewards, with the exception of the last value, which is calculated using the state-value given by the neural network at the first observation outside the scope of the current batch.

The optimizer used is Adam, same as the original PPO paper. No exhaustive search for the optimal training hyperparameters was performed. The parameters for the PPO algorithm are very similar to the one specified in the original paper.

A.2.2 Loss function

PPO is based on gradient ascent, but the popular machine learning frameworks that provides automatic differentiation are based on gradient descent methods. That's why the original PPO objective is changed and minimized for the **actor network**:

$$Loss_{actor} = -L_t^{CLIP}(\theta) - c_2 S[\pi_\theta](s_t)$$

The loss minimized on the **value network** is the mean-squared error between the returns and the approximated state-values.