Atari Agent Trajectories for State-to-Action Mappings

Carlos Avelar M. Sousa
168605
c168605@dac.unicamp.br

Lucas Baganha Galante
182364
l182364@dac.unicamp.br

Tiago Loureiro Chaves
187690
t187690@dac.unicamp.br

Tiago Petená Veraldi
187700
t187700@dac.unicamp.br

***Abstract*—In this project we have collected and prepared a dataset of state-to-action mappings for eight classic Atari games: Beam Rider, Breakout, Enduro, Ms Pacman, Pong, Qbert, Seaquest, and Space Invaders.**

For each game, we ran an expert agent trained with Proximal Policy Optimization (PPO) for 1000 steps, storing the actions taken given the state observations ($84 \times 84 \times 4$ images), for 100 of these trajectories. Thus, our dataset consists of 800,000 state-to-action pairs, which can be downloaded at: drive.google.com/open?id=1OtrZycTx-VItUxzCEVo9lKaBdGk4CyIC (the compressed file takes 1.7GB)

Now, with a good collection of state-to-action mappings on Atari games, this dataset can be used as a base for tackling the unsolved problem proposed by OpenAI in 2018 of fitting a generative model to all but one of the trajectories produced, and then to quantify the benefit of pre-training on the other games, to discover whether or not similar results could be achieved by using considerably less data when fine-tuning to the last game (compared to the amount used to fit to the other games).

If this proves itself feasible (i.e. if there is a measurable benefit from pre-training), then the generative model could be effectively used as a policy network for new different Atari games, in transfer learning fashion.

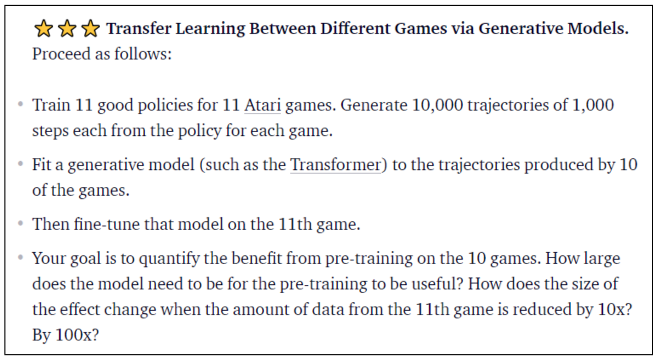
I. Introduction

A. Project Overview

The group choose one of the seven unsolved problems, published in "Request For Research 2.0" by OpenAI [1] as a goal to be worked on: "Transfer Learning Between Different Games via Generative Models" (Figure 1). The aim of this problem was to study how well generative models could be used as policy networks for Atari environments, quantifying the benefit of transfer learning after pre-training on different games.

In the original proposal, published by OpenAI in 2018, we would need to get good policies for 11 different Atari games, then fit a generative model to the trajectories of 10 of them, and finally fine-tune it on the last game, comparing the effectiveness of giving the same amount of data of the 11th game, as the amount used for the other ones during pre-training, to fine-tuning only with 10 or 100 times less information.

We were able to train good models for the easiest Atari games, but due to lack of computing power and time we decided to use pre-trained policies from the repository `rl-baselines-zoo` [2], as they compared well to the benchmarks published by OpenAI on `baselines` [3].



☆☆☆ **Transfer Learning Between Different Games via Generative Models.**
Proceed as follows:

- Train 11 good policies for 11 Atari games. Generate 10,000 trajectories of 1,000 steps each from the policy for each game.
- Fit a generative model (such as the Transformer) to the trajectories produced by 10 of the games.
- Then fine-tune that model on the 11th game.
- Your goal is to quantify the benefit from pre-training on the 10 games. How large does the model need to be for the pre-training to be useful? How does the size of the effect change when the amount of data from the 11th game is reduced by 10x? By 100x?

Figure 1: Problem published by OpenAI in "Requests for Research 2.0" [1]

With the good policies, we started to adapt the Transformer network for our problem, however, due to problems in the look-ahead masking we have not been able to actually fit it to the generated dataset.

B. Recent Events in Reinforcement Learning

Reinforcement Learning in the last few years has taken on very difficult problems, and enjoyed a wide variety of successes. In 2014, a team from DeepMind released a paper introducing Deep Q-Learning to play Atari games using only the frames from the screen as input (see Figure 2), achieving results never seen before and even super-human performance in a few of the games [4], [5].

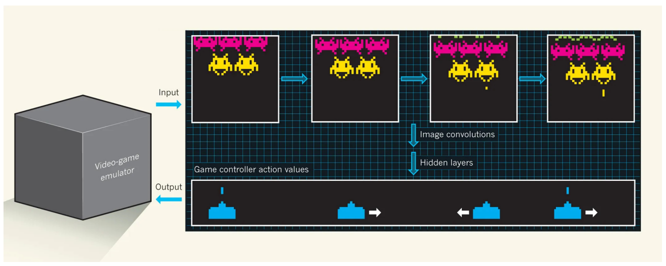


Figure 2: Illustration of the DQN algorithm on Atari games

Then, in 2015, DeepMind presented AlphaGo, a Go player trained with Reinforcement Learning techniques that was able to beat the best Go players in the world [6]. And, more recently, we've had OpenAI defeating the world champions of Dota 2 in April 2019 [7].

C. Training Algorithm

The Algorithm used to train the policies is PPO (Proximal Policy Optimization), developed by OpenAI in late 2017 [8].

It uses ideas from various previous algorithms, and is capable of achieving state-of-the-art results while being much simpler to implement and tune, making it generally the best policy based algorithm nowadays.

The PPO algorithm has become the main algorithm inside OpenAI and has been successfully used in some big projects like the aforementioned OpenAI Five [9], which was able to win a game against 5 professional Dota 2 players in 2019.

II. Background

A. Reinforcement Learning

Reinforcement Learning (RL) is one of the three base paradigms of machine learning, alongside supervised learning and unsupervised learning.

Unlike unsupervised learning where we are usually interested in extracting features or patterns from input data without any explicit instructions, the aim of an RL algorithm is to find a good behavior (strategy) in a framework where we have an agent and an environment — interacting as illustrated by Figure 3 —, such that it balances exploration (of uncharted territory) and exploitation (of current knowledge).

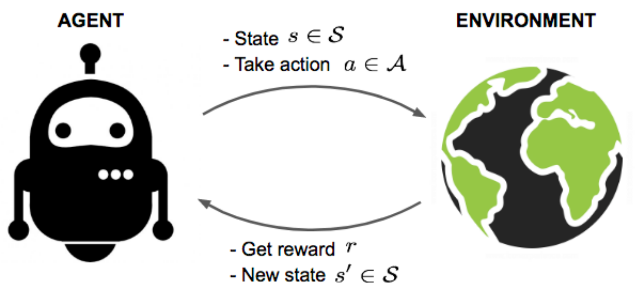

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 3: An agent interacts with the environment, trying to take smart actions to maximize cumulative rewards [1]

However, different from supervised learning, we do not possess labeled data — a set of observed states mapped to actions that should be taken, as pairs to fit a network model on —, thus, we can see reinforcement learning as a more general problem than supervised learning, as we don't just aim to make sense of data, but also to "extrapolate" actions from the information received (see Figure 4).

Using the same notation as in Figure 3, we introduce some of the common terminology that will be used later on:

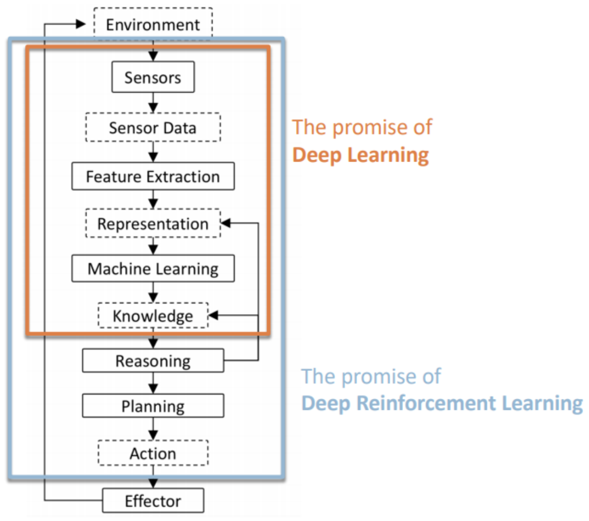


Figure 4: Aim difference between Deep Learning and Deep Reinforcement Learning [2]

- **Policy $\pi(s)$:** provides the "guideline" on what is the optimal action to take at state s, i.e. defines the agent's behavior strategy.
- **Value function $V_\pi(s)$:** predicts the expected amount of future rewards the agent is able to achieve from state s, acting by $\pi(s)$.
- **Model:** defines r and the transition probabilities between states, i.e. describes the environment.

There are two main approaches to creating RL agents, there are policy-based and value-based algorithms. The policy-based algorithms (such as PPO) try to learn directly how to act based on the observation, on the opposite side, the value-based (such as DQN and the classic Q-learning [10]) try to create a value function V_π for the possible states — which only implicitly contains the policy π —, consulting its values up before acting to maximize the obtained reward. Also, based on whether or not we have knowledge about the environment's model, we can categorize our agents as model-based or model-free (in which case we could try to learn the model explicitly).

For a more in-depth explanation of the concepts involved in reinforcement learning, and their mathematical representations, refer to: [11], [12].

As a last point, it's interesting to note that the validity of reinforcement learning is based on the reward hypothesis, that "all goals can be described by the maximization of expected cumulative reward".

B. PPO - Proximal Policy Optimization

The PPO algorithm tries to be balanced where most of its alternatives failed. One of the biggest problems with policy gradient methods is that if the hyperparameters aren't near perfect, you either have an agent that learns impractically slow or an agent that adjusts so violently to its last input that it doesn't learn at all [8].

[1]Image taken from Lilian Weng's blog post: "A (Long) Peek into Reinforcement Learning": lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html

[2]Image taken from Esther Colombini's class slides: "Aula 17: Aprendizado por Reforço": ic.unicamp.br/ esther/teaching/2019s1/mc906

Some earlier methods have succeeded in being more stable in that sense, however, at the cost of computation, making them hard to implement and also slow, and that's where PPO enters. PPO doesn't have necessarily the best performance in all benchmarks, but it has achieved very convincing results and done so with a simple implementation, while keeping the extra computation in check.

PPO has two main variations, PPO-Penalty and PPO-Clip, here we will focus on the Clip one, as it has become the standard. The PPO-Clip, put simply, removes incentives for updates that take the new policy too far from the current policy, and does so by clipping the objective function. So, by taking the minimum between the update and the clipping function, we limit how far the policy can go [13].

III. Proposed Solution

A. Atari Games

We have used OpenAI's Gym toolkit [14], which is the *de facto* standard for training RL agents on Atari games, since it provides a unified environment interface (`Env`) that provides the following primary properties and methods:

- **action_space:** Corresponds to valid actions.
- **observation_space:** Corresponds to valid observations.
- **reset(self):** Reset the environment's state. Returns observation.
- **step(self, action):** Step the environment by one timestep. Returns `observation`, `reward`, `done`, `info`:
    - **observation:** Agent's environment observation
    - **reward:** Reward returned after previous action
    - **done:** Whether the episode has ended
    - **info:** Auxiliary diagnostic information

In our case, the `action_space` ranges from 0 to 17, consisting of the 18 available actions that can be taken across all Atari games (e.g. `NOOP, FIRE, LEFT, RIGHT`, etc.), and the `observation_space` is a `(84, 84, 4)` tuple, as for each observation we have a stack of 4 frames — which are grayscaled and downscaled from 210×160 to 84×84 —, mimicking DeepMind's Nature implementation [15].

B. PPO

While OpenAI does provide a set of implementations of reinforcement learning algorithms, namely Baselines [3], it's poorly documented and doesn't officially support running on IPython notebooks (such as Google Colab). Therefore, we've decided to use the PPO algorithm implementation made by Stable Baselines [16], which is a cleaner and better documented fork of OpenAI's Baselines.

We have also used the pre-trained agents from RL Baselines Zoo [2], a collection of pre-trained RL agents using Stable Baselines implementations, that achieves similar results to the ones published by OpenAI in their Baseline's benchmark.

Thus, we have generated trajectories for all of the eight agents trained with PPO on Atari environments in RL Zoo Baselines, as shown in Table I.

Game	Gym Env
Beam Rider	BeamRiderNoFrameskip-v4
Breakout	BreakoutNoFrameskip-v4
Enduro	EnduroNoFrameskip-v4
Ms Pacman	MsPacmanNoFrameskip-v4
Pong	PongNoFrameskip-v4
Qbert	QbertNoFrameskip-v4
Seaquest	SeaquestNoFrameskip-v4
Space Invaders	SpaceInvadersNoFrameskip-v4

Table I: Atari games that were used, and their corresponding Gym environments

C. Transformer

We used TensorFlow's tutorial implementation of the Transformer network as a basis for our own modified version of it [17]. First, we understood more about the Transformer running it as a translation network in our project baseline, and then tried to adapt it to our problem — which could also be thought of as a translation problem, as it consists of mapping observations to actions (rather than one language to another).

We started by making the Transformer receive an image in its encoder, changed the embedding layer to a feature extraction layer using MobileNetV2, then used a global average pooling layer to transform the $5 \times 5 \times 2048$ tensor into a 2048 sized vector.

Next, we moved to the decoder and adapted the input to receive a 18 sized vector (the full size of the Atari action space). However, when we got to the second decoder layer some shape problems started to arise due to the shape of the input from the encoder.

What happens is that in our problem there's a big asymmetry in the sizes of the input and output, the input is 100 times bigger than the output. In this case it would be extremely inefficient to just pad the smaller to the size of the largest, so another solution was required.

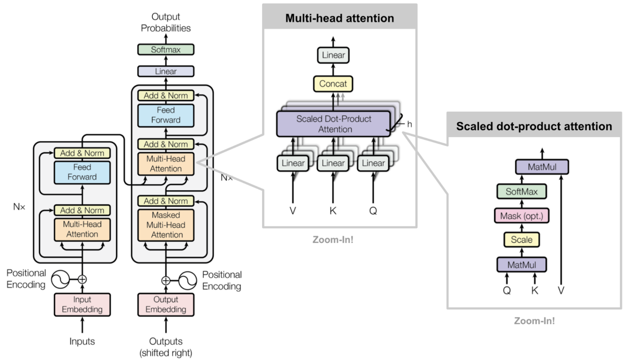


Figure 5: Transformer model architecture [3]

What we found out is that we needed to use different d_{model} sizes for the encoder and decoder, even though the writers of the original paper have said: "To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$.".

[3]Image taken from Lilian Weng's blog post: "Attention? Attention!": lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

After setting the d_{model} for the encoder as the size of the input (1280 after passing the images through MobileNetV2) and the d_{model} for the decoder as the size of the output (18), the network seemed to be good to go.

Unfortunately, after all layers worked individually, we went on to test the whole thing and ran into errors due to the masks shapes, which we couldn't figure out in the available time.

D. Dataset structure

After generating 100 trajectories of 1,000 steps each for the eight Atari games (see Table I), we organized our dataset with the following folder structure:

```
atari8
├─BeamRiderNoFrameskip-v4_PPO2
├─BreakoutNoFrameskip-v4_PPO2
├─EnduroNoFrameskip-v4_PPO2
├─MsPacmanNoFrameskip-v4_PPO2
├─PongNoFrameskip-v4_PPO2
├─QbertNoFrameskip-v4_PPO2
├─SeaquestNoFrameskip-v4_PPO2
└─SpaceInvadersNoFrameskip-v4_PPO2
   ├─00
   ├─...
   ├─99
   │  └─trajectory.npz
   └─images
      ├─00
      ├─...
      └─99
         ├─000.png
         ├─...
         └─999.png
```

We have a folder for each environment, and inside of it there are 100 folders corresponding to the generated trajectories (recorded in NumPy compressed files named `trajectory.npz`) and an `images/` folder.

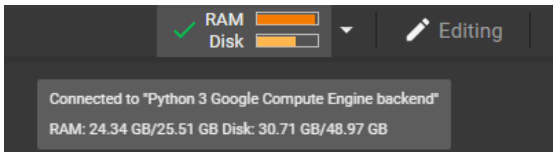
Each `trajectory.npz` file can be loaded as a set of four NumPy arrays, queried by the keys: `actions`, `observations`, `rewards` and `episode_starts`.

The elements of the `observations` array store the path to the PNG image (inside the `images/` folder) that represented the environment state that led to the action in the corresponding element position of the `actions` array. These are our state-to-action mappings.

The two other arrays contain information about the reward received at each time step, and also of whether or not the environment returned `done` as true — since we run every trajectory for 1000 steps, there may be more than one episode per trajectory, as an agent could lose the game and have the environment reset itself before 1000 steps (see Table II).

The compressed file (`atari8.zip`) takes 1.7GB, and can be downloaded at: drive.google.com/open?id=1OtrZycTx-VItUxzCEVo9lKaBdGk4CyIC.

Although we used Google Colab for running all our code, as it provides enough disk space to work with a dataset that has around 600 trajectories per game, once we unzipped the file



Connected to "Python 3 Google Compute Engine backend"
RAM: 24.34 GB/25.51 GB Disk: 30.71 GB/48.97 GB

Figure 6: Used RAM space in Google Colab, after reading all observation images into NumPy arrays with OpenCV

and read the images into arrays with OpenCV to manipulate them, the available RAM space would be exceeded, so we decided to record only 100 trajectories because it almost reached the maximum RAM available, as shown in Figure 6.

IV. Experiments and Discussion

A. Atari Games

After the project baseline, where we tested Monte-Carlo Policy Gradient (REINFORCE [18]), DQN and the classic Q-learning algorithm [10], we read about the benefits of Proximal Policy Optimization, and how it's become OpenAI's go to algorithm, so we ended up using it for all following experiments.

All agents were trained on Google Colab's K80 GPUs, using a GPU optimized implementation of PPO from Stable Baselines: `PPO2`.

We initially tried training the default MLP policy network architecture of Stable Baselines (2 layers of 64 neurons [19]), however we realized that the agent had no way of knowing what happened before as we were using a single frame from the game as the observation ($210 \times 160 \times 3$ image).

This way, the agent could not play a simple game like Breakout, because there was no way of even knowing where the ball was going without the temporal information, thus, not following the needed Markov property [20].

After noticing this, we tried using a layer normalized LSTM network with a CNN for feature extraction, but still with single image frames. The model was learning, but it was too slow.

So, we resorted to the same architecture that was used by DeepMind in their Nature paper [5, page 6], a simple CNN with three convolutional layers followed by a dense one, all with ReLU activation, and then an output dense linear layer with 18 neurons (one per action) — but with the caveat of using 4 stacked frames (downscaled to 84×84 and converted from RGB to grayscale) as the state observation. With it, we were finally able to get good results in reasonable time. Figure 7 shows a schematic illustration of the neural network.

On the Breakout game with 4 stacked frames using PPO2 with a CNN, the agent after 4 million epochs was playing really well, easily winning the game. However, on harder games like Space Invaders, our agents still performed very poorly after some days of training on Colab's K80 GPUs.

Because of that, we decided to use pre-trained agents when generating our dataset, so that it consisted of expert behavior strategies. Therefore, as mentioned in Section III-B, we downloaded the models from RL Baselines Zoo, as they

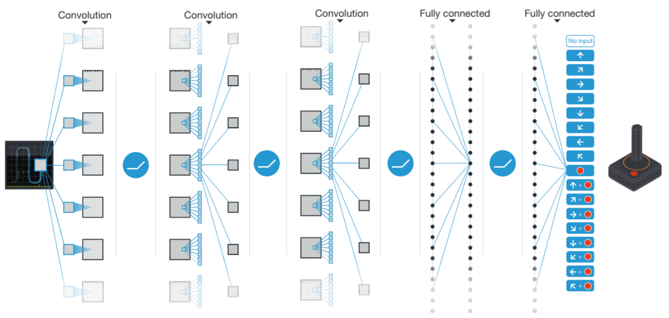


Figure 7: DeepMind's policy network architecture

were also pre-trained using Stable Baseline's `PPO2` algorithm we had been experimenting with.

Measurements of the cumulative sum of rewards received by the agents in the trajectories of our `atari8` dataset are shown in Figure 8.

B. Dataset Stats

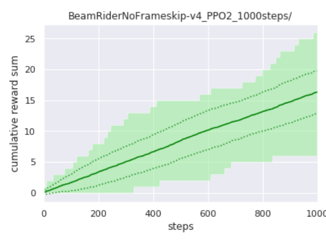
The `atari8` dataset we composed contains four different 1000 elements-sized information arrays, for each of the 100 trajectories of each game, namely: `actions`, `observations`, `rewards` and `episode_starts`, as described in Section III-D. Besides that, all of the $84 \times 84 \times 4$ observations used as input for the CNN policy network were saved, so that we could have a direct mapping of observed state to action taken.

One important aspect to note is that, as we're always collecting trajectories of 1000 steps, the agent might lose the game (or a life) in earlier steps, causing the environment to be reset. Therefore, the generated trajectories may contain more than one episode on harder games, as shown on Table II by the average and maximum episode lengths for each game (calculated from the `trajectory.npz` files of each trajectory).

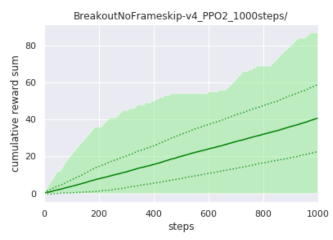
	Episode length		
Game	**Mean**	**Std. dev.**	**Max.**
Beam Rider	522.72	358.19	1000
Breakout	362.92	394.96	1000
Enduro	929.54	202.06	1000
Ms Pacman	199.84	165.73	603
Pong	765.43	326.97	1000
Qbert	298.03	281.21	1000
Seaquest	270.64	159.27	566
Space Invaders	262.89	216.86	906

Table II: Mean, standard deviation and maximum values for the episode lengths of trajectories collected (i.e. steps until the environment returns `done` as true, and we reset it)

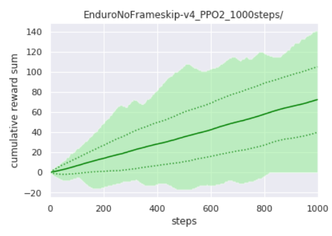
Also, as the images used have 4 channels — since four grayscaled frames are stacked to include temporal information in each state — we have two main choices when trying to fit a generative model to them: we could either directly pass $84 \times 84 \times 4$ images through an input convolutional layer, or we'd have to first convert them to RGB by constricting to 3 channels



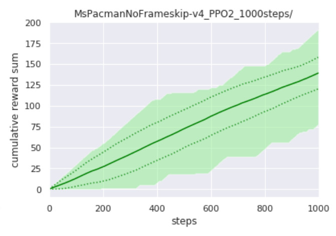
(a) Beam Rider



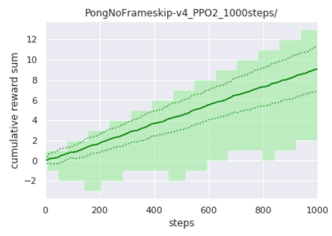
(b) Breakout



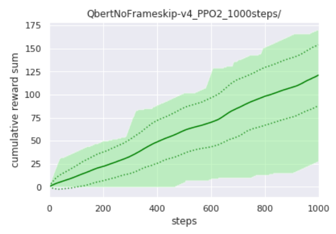
(c) Enduro



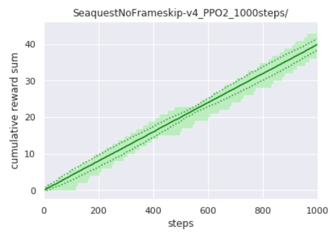
(d) Ms Pacman



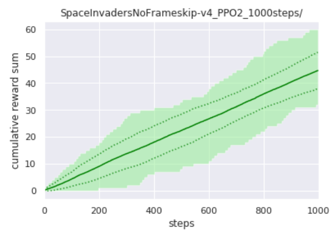
(e) Pong



(f) Qbert



(g) Seaquest



(h) Space Invaders

Figure 8: Cumulative reward sum for the collected trajectories (solid line shows the mean value, dotted lines show ± one standard deviation, and the colored space is defined by the minimum and maximum values)

to later be able to use a model pre-trained on ImageNet as a feature extractor.

One way we could make RGB images while retaining more information than simply dropping a channel would be to take the average of the 4 stacked frames pairwise, which results in 3 values.

C. Transformer

While we couldn't get our modified Transformer implementation to work in the available time, we attempted many experimentation's with it, so we believe that documenting our tests and mistakes with it is a viable contribution to future investigations on this problem.

The main steps taken to adapt the TensorFlow implementation of the Transformer network [17] were already described in Section III-C, so here our objective is to explain a bit more of the choices we made.

The TensorFlow tutorial was the best example of the Transformer we encountered, and the code was simple enough for us to quickly get a nice understanding of it. On the other hand, we should emphasize that the original article, "Attention is All You Need" [21], does a really poor job in conveying even the idea of how Query, Key and Value work together. For a good introduction to the concepts used, the following references are good resources to which we resorted to: [22]–[24].

Based on another TensorFlow tutorial, that did image captioning with additive (or concat) attention [25] — instead of scaled dot-product attention, as was used in the Transformer —, we decided to bet on the second approach mentioned at the end of the previous Section IV-C: to use a pre-trained network as a feature extractor for the observation images.

We have chosen the MobileNetV2 as a substitute for the embedding layer that's used for word tokens, as our input did not vary in size, opposite to the original network. However, as the dimension of the input was still high, we decided to pass the extracted feature array through a global average layer. Another feature extractor candidate was the InceptionV3 network, however, its output was larger and it took more time to pre-process the images.

About the d_{model}, we choose value of 1280 for the encoder and 18 for the decoder based on the input and output sizes. Defining d_{model} for the encoder and decoder was our solution for the dimension difference problem of our project.

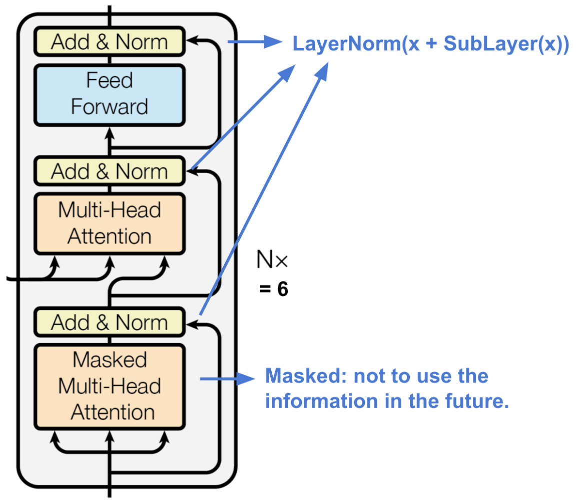

Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
LayerNorm(x + SubLayer(x))
N×
= 6
Masked: not to use the information in the future.

Figure 9: The Transformer's decoder[4]

At last, the look-ahead masking which we couldn't fix in due time was necessary for a complete working network since, without it, the Transformer wouldn't be much of a sequence processing network and more of a simple attentional network (see Figure 9), as we would be training it with all observations (previous or future) for all steps.

We failed to adapt the Transformer in the available time, but have advanced very well, especially considering the big lack of documentation and examples for it, and barely no applications of Transformers on tasks similar to ours.

[4]Image taken from Lilian Weng's blog post: "Attention? Attention!": lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

V. Conclusions and Future Work

Reinforcement Learning is already getting mature in terms of documentation and availability, frameworks such as Gym and Stable Baselines, and learning resources like OpenAI's Spinning Up in Deep RL are of big help for beginners. So much so, that it was possible for the group, who knew very little about reinforcement learning before the project and had no prior experience with RL, to get a good grasp of it and to train several good policies for Atari games.

This is an interesting project that we believe can be accomplished with more time and resources, specially with use of the Transformer network. And even though we weren't able to finish, there are a few things that if adapted could improve its performance, like for instance tuning the two d_{model} variables, and testing other feature extractors, like the InceptionV3 or a ResNet.

Another idea that we had to deal with the difficulty of adapting the Transformer would be to use a different generative model, like GANs or VAEs, even though they tend to be generally hard to properly train.

This was, in the end, quite an ambitious project, as it encompassed many different areas, some with just not enough documentation for a good understanding. So we consider it a success as members of the group were able to get a good grasp on the various areas and even able to get good results in a few of them.

Finally, we leave as contribution a dataset for validation with 100 trajectories for 8 Atari games, the code to generate datasets like this from trained models, and an unmasked Transformer adapted to the problem at hand. So anyone that would like to continue this work may already have a good jump start with this material.

The notebook with the source code can be seen on github.com/laurelkeys/machine-learning (inside the `assignment-4` folder).

References

[1] I. Sutskever, J. Schulman, T. Salimans, and D. Kingma, "Requests for Research 2.0," *https://openai.com/blog/*, 2018. [Online]. Available: https://openai.com/blog/requests-for-research-2/ 1

[2] "RL Baselines Zoo," https://github.com/araffin/rl-baselines-zoo, 2018. 1, 3

[3] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "OpenAI Baselines," https://github.com/openai/baselines, 2017. 1, 3

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602 1

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human Level Control Through Deep

Reinforcement Learning," *Nature*, vol. 518, pp. 529–33, 02 2015. [Online]. Available: https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf 1, 4

[6] DeepMind, "AlphaGo: The story so far." [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far 1

[7] OpenAI, "OpenAI Five," https://blog.openai.com/openai-five/, 2018. 2

[8] J. Schulman, O. Klimov, F. Wolski, P. Dhariwal, and A. Radford, "Proximal Policy Optimization," *https://openai.com/blog/*, 2017. [Online]. Available: https://openai.com/blog/openai-baselines-ppo/ 2

[9] OpenAI Five, "OpenAI Five Overview," https://openai.com/five/#overview, 2019. 2

[10] Wikipedia contributors, "Q-learning — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Q_learning 2, 4

[11] OpenAI Spinning up, "Key Concepts and Terminology," *https://spinningup.openai.com/*, 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#key-concepts-and-terminology 2

[12] L. Weng, "Policy gradient algorithms," *lilianweng.github.io/lil-log*, 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html 2

[13] OpenAI Spinning up, "Proximal Policy Optimization," *https://spinningup.openai.com/*, 2018. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html 3

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," https://github.com/openai/gym, 2016. 3

[15] D. Seita, "Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games," *https://danieltakeshi.github.io*, 2016. [Online]. Available: https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/ 3

[16] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable Baselines," https://github.com/hill-a/stable-baselines, 2018. 3

[17] TensorFlow, "Transformer model for language understanding," 2019. [Online]. Available: https://www.tensorflow.org/tutorials/text/transformer 3, 6

[18] R. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," *Adv. Neural Inf. Process. Syst*, vol. 12, 02 2000. 4

[19] Stable Baselines, "Policy Networks — Stable Baselines 2.9.0a0 documentation," https://stable-baselines.readthedocs.io/en/master/modules/policies.html, 2018-2019. 4

[20] Wikipedia contributors, "Markov property — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Markov_property 4

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762 6

[22] L. Weng, "Attention? Attention!" *lilianweng.github.io/lil-log*, 2018. [Online]. Available: http://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html 6

[23] TensorFlow, "Neural machine translation with attention," 2019. [Online]. Available: https://www.tensorflow.org/tutorials/text/nmt_with_attention 6

[24] A. Rush, G. Klein, and V. Nguyen, "The Annotated Transformer," *https://nlp.seas.harvard.edu/*, 2018. [Online]. Available: https://nlp.seas.harvard.edu/2018/04/03/attention.html 6

[25] TensorFlow, "Image captioning with visual attention," 2019. [Online]. Available: https://www.tensorflow.org/tutorials/text/image_captioning 6

Links accessed on December 2019.