Generalization in videogames

Marcos Torrejón, Carlos & Ortega Ochoa, Raúl & Jásdi, Miklós

Technical University of Denmark (DTU)
Github repository: https://github.com/raulorteg/deep-learning-project.git

Computer games offer an excellent benchmark to test various reinforcement learning (RL) practices and network architectures. In our project, we explored the possibilities of improving an RL agent's generalization performance in order to play different computer games with high success. We have tested different architectures and learning mechanisms to improve the agent's performance in a single game, then proceeded to assess the optimized model's performance over multiple games.

Introduction

Reinforcement learning is one of the fastest-developing areas within deep learning, in which agents have to carry out actions in an environment to maximize the cumulative reward associated with pre-defined goals. Agents are not supplied any prior training data, therefore it is their task to learn which actions lead to the highest reward, by carrying out actions in the environment.

As a layman's example, we can train an agent to play chess by supplying it the most essential rules (e.g. valid actions for a given piece) and a reward function (eliminating the pieces of the opponent), then making it play several games to discover the relation between the sequence of actions and the resulting reward. It is no surprise that reinforcement learning is being applied in many disciplines, as the ever-continuing improvements in computing technology let us train agents to handle complex problems.

In our project, we apply reinforcement learning in the field of computer games, as they offer convoluted environments and reward structures, making them an excellent benchmark for reinforcement learning algorithms. Our goal is to mimic a human's learning process by training an agent based solely on the screen output and reward (in-game score) - without the knowledge of the underlying game mechanics. In the meantime, we also aim for a general neural network architecture that can be trained to play vastly different games with high success.

We have centered our efforts around a neural network based on the actor-critic model, fed by data generated by the Procgen benchmark. We have also employed an encoder network in order to pre-process the high-dimensional screen pixel data. Our work was mainly focused on improving the actor critic model, and the aforementioned encoder.

Procgen Benchmark

Procgen Benchmark [1] consists of 16 procedurally-generated unique environments designed for evaluating sample efficiency and generalization in RL. Previous Benchmarks such as the Arcade Learning Environment (ALE) are simulators running a deterministic program where the only noise comes from

an initial random seed, and thus it has been shown [2] they are prone to over-fitting with the RL method exhausting the noise and memorizing the data, with poor generalization.

Procedural content generation benefits from a near-infinite supply of randomized content, avoiding in this manner relying on the fixed human-designed content that can be more easily over-fitted.

A subset of five environments (*figure 1*) was chosen from those available with Procgen Benchmark to represent the diversity of the benchmark due to limitations in available computer power. Procgen benchmark allows configuring the difficulty of the shown environments and the possibility of removing the backgrounds in games. In this report the difficulty was set to easy without backgrounds.

Figure 1: *Screenshots from the environments: Starpilot, Chaser, Jumper, BigFish, Coinrun.*

Actor-Critic

To train the agent we will use an Actor-Critic setup. Actor-Critic methods have a separate memory structure to explicitly represent the policy independent of the value function and each of these functions will be approximated by a neural network (Deep Reinforcement Learning).

In Reinforcement Learning the policy is defined as the distribution of actions (a) given states (s) of the evaluated environment $\pi_\theta(s,a) = \mathbb{P}[a \mid s, \theta]$ for a set of parameters θ of the policy function. In Policy-based Reinforcement Learning the learning process is purely based on policy function updates, but this process is typically too slow, as it requires finishing a full episode in the environment before doing a single update of the policy. This problem can be addressed by evaluating in parallel a function that estimates the goodness of the current states (how much reward will the agent get from that state ahead following the current policy) before the episode ends. This is what is known as the value function $V_{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[R_{t+1} + \gamma V_{\pi_\theta}\left(S_{t+1}\right) \mid S_t = s\right]$, where R_{t+1} is the

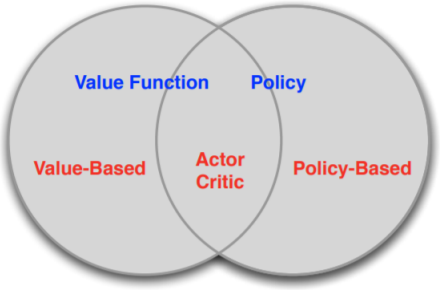


Figure 2: *The combination of Policy-based and Value-based Reinforcement Learning creates Actor-Critic methods.*

reward obtained in the next timestep t and γ is a future rewards discount factor. This way we speed up the convergence of the algorithm while getting more updates of the policy.

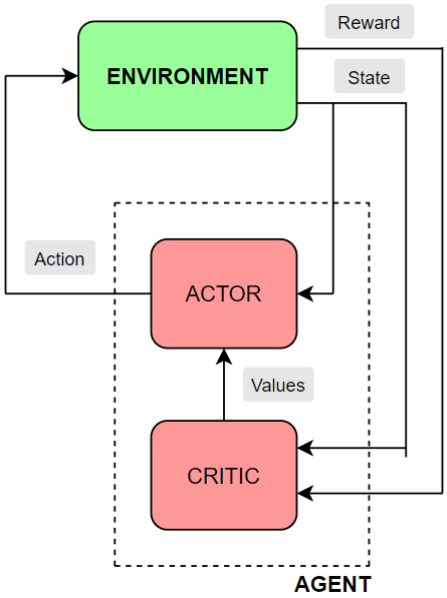


Figure 3: *Actor-Critic diagram*

The policy structure is then known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor.

Proximal Policy Optimization

In an Actor-Critic setup we will follow the guidelines of Proximal Policy Optimization (PPO) introduced by OpenAI [3]. In PPO we alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function (1) using stochastic gradient ascent. For the optimization task we will make use of collected sampled stored in a buffer. Therefore, to optimize policies, we al-

ternate between sampling data from the policy and performing several epochs of optimization on the sampled data.

$$L_t^{PF+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{PF}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t)] \tag{1}$$

The objective function (1) consists of three parts. The first refer to the clipped policy function loss (2)

$$L^{PF}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t)] \tag{2}$$

where $r_t(\theta)$ denotes the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, so $r_t(\theta_{old}) = 1$, c_1, c_2 are tunable hyperparameters and $\hat{A}_t$ is the advantage function. The advantage function is computed using the value function following expression (3)

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \quad where\ \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{3}$$

where t specifies the timestep index in [0, T=256] and uses collected samples from the buffer, *gamma* and *lambda* are also tunable hyperparameters.

The clip in the policy loss function allow us to constrain the policy function update, taking a more conservative approach. The utility of this approach is explained by the fact that policy updates are defined by the performance of the agent actions in the environment. That is, the policy function will receive a big update if the action taken by the agent is very "good" or "bad". While this can be seen useful at first glance, it is not if we consider that big leaps in the policy function can be harmful once we are orbiting near the global maxima, stepping out of the trust region.

The second part of the loss function (1) is the value function loss (5)

$$V_{CLIP}^{\pi_\theta}(s_t) = \text{clip}\left(V^{\pi_\theta}(s_t) - \hat{V}_t, -\epsilon, +\epsilon\right) \tag{4}$$

$$\mathcal{L}^{VF}(\theta) = \frac{1}{2}\mathbb{E}\left[\max\left(\left(V^{\pi_\theta}(s_t) - r_t\right)^2, \left(V_{CLIP}^{\pi_\theta}(s_t) - r_t\right)^2\right)\right] \tag{5}$$

where $V^{\pi_\theta}(s_t) - \hat{V}_t$ denotes the difference between the current value function output and the values of the collected samples for the update in the batch and r_t is the return at timestep t. The value loss function is also clipped but using the max, as we are considering the squared difference between the values and the returns.

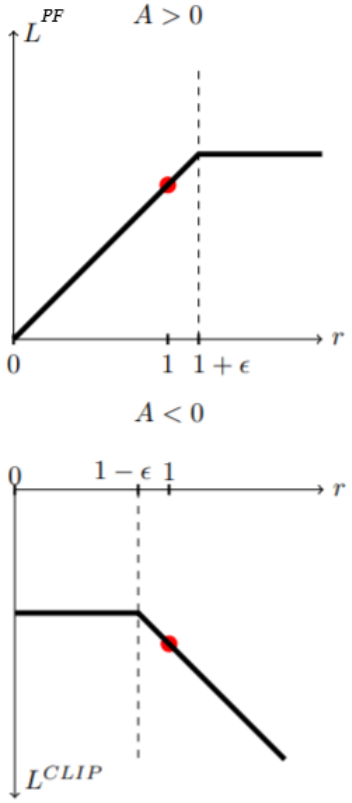


Figure 4: *Policy loss function update clip for different values of the advantage function.*

And the final part of the loss function corresponds to the entrophy term $S\left[\pi_\theta\right](s_t)$ which is a bonus to ensure sufficient exploration.

In our implementation of PPO each episode consists of 256 timesteps per environment. In each iteration, the actor collects 256 timesteps of data. Then, we construct the surrogate loss on these 256 timesteps of data, and optimize it with minibatch Adam, for 3 epochs.

Algorithm 1 PPO

```
for iteration=1,2,... do
  for timestep=1,...,256 do
    Run policy π_θold for 32 environments
    Store environment data
  end for
  for epoch=1,...,3 do
    Optimize surrogate L, with minibatch
    size 512 (< 32 × 256)
    θ_old ← θ
  end for
end for
```

Image encoding

Game environments are handed to the policy and value functions through an encoder. The encoder inputs $64x64$ pixels images from each environment and compresses the size of the input to be evaluated by the agent. The architecture of the encoder therefore plays a crucial role in the agent interpretation of the environments and directly affects the performance of the overall algorithm. We have tested three different encoders: Nature DQN, IMPALA and VGG to investigate how scaling model impacts sample efficiency and generalization.

Nature DQN [4] consists of 3 convolutional blocks that gradually compress the input image with convolutions of gradually smaller kernel sizes and strides (*Figure 5*) and a final fully connected dense layer. It is the smallest arquitecture among the three tested, and it was found that it struggled to train in the high diversity of the procedurally generated environments which aligns with results from (*Cobbe, Karl & Hesse et al. (2019)*), smaller architectures tend to struggle to generalize when faced with Procgen Benchmark.

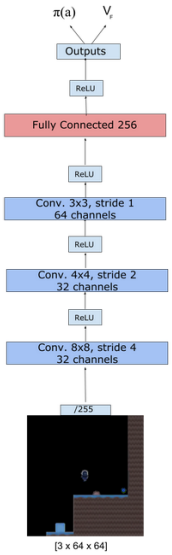


Figure 5: *DQN encoder diagram*

The **IMPALA** implementation used was a modified version of the original [5] where the LSTM cell is substituted by a dense layer as shown in *figure 6*. IMPALA consists of 3 major blocks consisting of a convolutional layer followed by a Max pooling layer and 2 residual blocks, each with 2 convolutional layers. In total this IMPALA implementation consists of 15 convolutions with 3 max pooling layers and a final Fully connected layer. The added complexity of the architecture, when compared to Nature DQN, benefited generalization achieving better results, but it is also more computationally expensive.

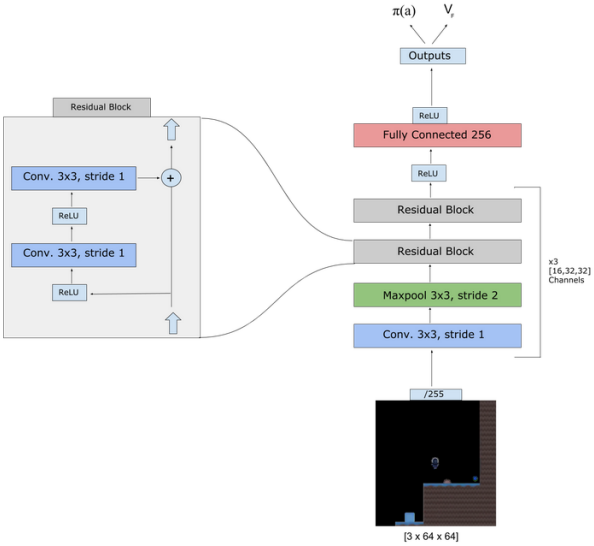


Figure 6: *IMPALA encoder diagram*

Finally **VGG** [6] is a Convolutional Neural Network consisting of 13 convolutional layers with 4 max pooling layers and 3 dense layers. It is the biggest architecture among the 3 that were tested. The structure of the VGG architecture used can be seen in *figure 7*. VGG was found to achieve better results than both Nature DQN and IMPALA, but at the expense of a significant increased training time (close to 12 hours runtime, compared to $\sim$ 8 hours runtime of IMPALA or $\sim$ 6 hours runtime of DQN).

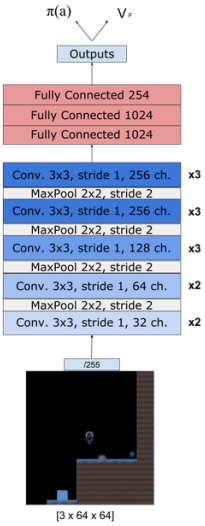


Figure 7: *VGG encoder diagram*

Results

Selection of the best model based on single-game performance

In order to come up with a model candidate that may likely perform well in multiple games, we have tested the generalization performance of the three aforementioned encoders over a single game, namely "starpilot". The outputs of the encoder were fed into a simple policy network consisting of one linear layer, with the following hyperparameters:

- Total timesteps = 8×10^6
- Number of environments = 32
- Number of levels = 10
- Timesteps per episode = 256
- Number of epochs = 3
- Batch size = 512
- PPO clip (ϵ) = 0.2
- Gradient clip = 0.5
- Learning rate = 5×10^{-4}
- Value coef (c_1) = 0.5
- Entropy coef (c_2) = 0.01
- $\gamma = 0.99$
- $\lambda = 0.95$

Our experiment has yielded the results marked by Figure8 below:

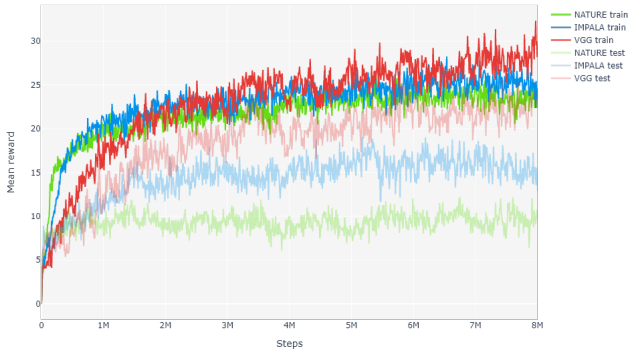


Figure 8: *Comparison of encoders*

In terms of test performance, overfitting is apparent in all three models, although to different extents. The policy equipped with the VGG (red) encoder has been found to be the best, tightly followed by our modified IMPALA (blue). The baseline Nature DQN's (green) performance was only a fraction of the other two.

However, there are other aspects to be taken into consideration, too. The encoders - given their complexities - required different amounts of computational resources and runtime to converge. For instance, VGG, that has by far the most convolutional layers, has converged in much more steps than the other two encoders. In addition, a single training step for the VGG-based network has taken significantly more runtime than for the other two.

Assessment of the selected model over multiple games

Considering the test performance and the required computational resources, we have determined that the IMPALA-based network offered the best compromise between performance and runtime. Therefore, we have selected the IMPALA encoder for our assessment of the generalization performance over multiple games.

Using Procgen, we have generated four different environments in addition to the original "starpilot", namely: "coinrun", "chaser", "bigfish" and "jumper", and trained our selected model architecture in them, using the same hyperparameters as in the single-game test. We have obtained the following mean rewards over 256 timesteps per environment outlined by *figure 9* below:

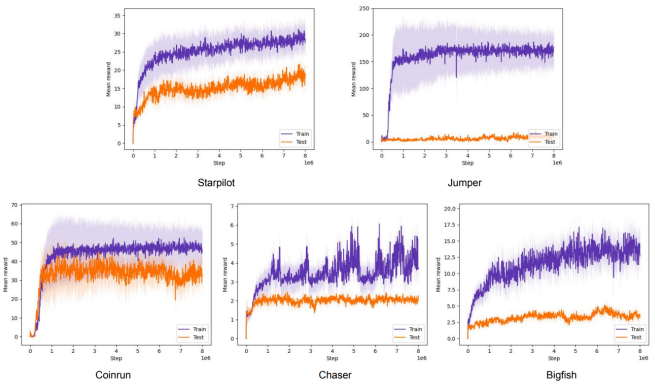


Figure 9: *Performance of the IMPALA-based network over different games*

We can see that the model performs well in some games, such as "starpilot" and "coinrun", while there are certain exceptions. Starpilot and "coinrun" are relatively simple games, in which there are many rewards scattered around in the environment (such as enemy ships in "starpilot", and coins scattered around the map in "coinrun"), that can be reached by performing simple actions.

On the other hand, games such as "jumper", "chaser" and "bigfish" have more complex rules which

might pose a challenge for the simple, one-layer policy network. The "jumper" game has proven to be particularly difficult, due to the phenomenon of "sparse rewards". In this game, the agent is only rewarded if it touches the coin at the very end of the map, which dramatically decreases the learning and generalization performance.

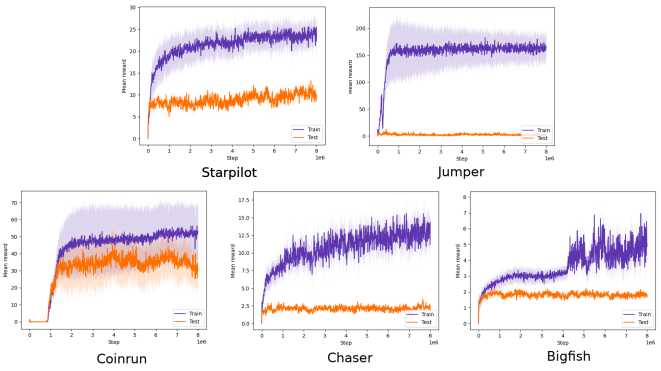
Conclusion

Generalization remains one of the most important challenges of Reinforcement Learning. Videogames have been proved really successful for testing RL algorithms as these environments often provide a easy way of assigning rewards and are not subject to real time but computer power, which allows to speed up the training process. However, Benchmarks such as the Arcade Learning Environment (ALE) or The Sonic benchmark [7] exhibit problems with generalization as they are Human designed deterministic programs where the only source of randomness comes from the seed of the game and thus are more easily over-fitted by the RL algorithm. In this report we have used a Benchmark with a variety of procedurally generated levels for a variety of videogames, which by its near-infinite supply of highly randomized content aims to help the community to contend with this challenge of over-fitting and generalization.

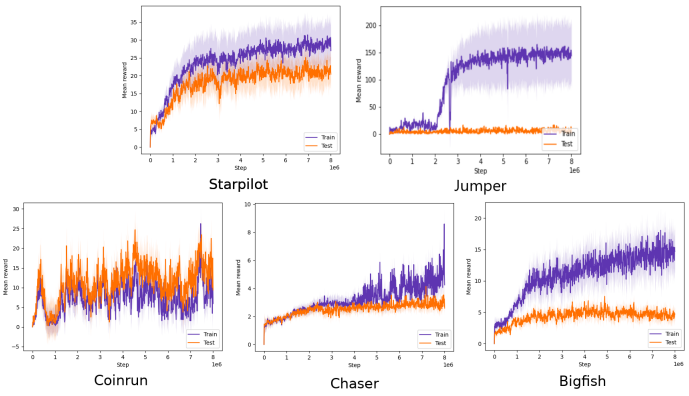
We have used Proximal Policy Optimization (PPO) Reinforcement Learning algorithm in a subset of the environments of Procgen Benchmark with 3 different encoders (DQN, IMPALA, VGG) to evaluate sample efficiency and generalization of the architectures. DQN, the smaller encoder, proved unsuccessful when faced with the high diversity of the levels within each environment, the more complex encoders IMPALA and VGG achieved better results, with VGG (the most complex encoder) achieving the best scores in test and train but at expense of significantly higher computer power cost. For that it was argued that the best balance of Generalization and efficiency was found with the IMPALA encoder.

Appendix

Mean rewards obtained with DQN and VGG encoder arquitectures.



Appendix 1: *Performance of the Nature DQN-based network over different games*



Appendix 2: *Performance of the VGG-based network over different games*

As we have commented in previous sections, DQN performs poorly compared to the rest of encoders. In contrast, VGG achieves better mean rewards - specially in starpilot and chaser, but with longer runtimes. An exception is the Coinrun environment. In this specific case it looks like a more complex architecture has not translated into better performance. In fact, the learning curve looks poor, resulting in the agent performing better in the test random seed than in the training random seed.

References

1. Cobbe, Karl & Hesse, Christopher & Hilton, Jacob & Schulman, John. (2019). Leveraging Procedural Generation to Benchmark Reinforcement Learning.

2. Zhang, N. Ballas, and J. Pineau. A dissection of overfitting and generalization in continuous reinforcement learning. CoRR, abs/1806.07937, 2018a.

3. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347v2.

4. Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Rusu, Andrei & Veness, Joel & Bellemare, Marc & Graves, Alex & Riedmiller, Martin & Fidjeland, Andreas & Ostrovski, Georg & Petersen, Stig & Beattie, Charles & Sadik, Amir & Antonoglou, Ioannis & King, Helen & Kumaran, Dharshan & Wierstra, Daan & Legg, Shane & Hassabis, Demis. (2015). Human-level control through deep reinforcement learning. Nature. 518. 529-33. 10.1038/nature14236.

5. Espeholt, Lasse & Soyer, Hubert & Munos, Remi & Simonyan, Karen & Mnih, Volodymir & Ward, Tom & Doron, Yotam & Firoiu, Vlad & Harley, Tim & Dunning, Iain & Legg, Shane & Kavukcuoglu, Koray. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.

6. Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.

7. Nichol, Alex Pfau, Vicki Hesse, Christopher Klimov, Oleg Schulman, John. (2018). Gotta Learn Fast: A New Benchmark for Generalization in RL.