TÉCNICO LISBOA

Evaluating generalization in Deep Reinforcement Learning with Procedurally Generated Environments

Miguel Borges Freire

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Arlindo Manuel Limede de Oliveira

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Arlindo Manuel Limede de Oliveira
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

December 2021

Acknowledgments

First of all, I'd like to thank my supervisor Professor Arlindo Oliveira for his guidance, friendship, and all the trust placed in me.

My humble thank you to Alexandre and André for all the support, discussions, and friendship.

A special thanks to João, Henrique C., Henrique P., and Guilherme for all the adventures and companionship over the last six years.

A big thank you to all my friends and family for the constant support and love.

Finally, I'd like to thank my parents for every opportunity they gave me; for all the love and care. It's been a privilege being your son, and I'm sure you are proud of the man I've become.

Abstract

Deep Reinforcement Learning agents, mainly those who learn from visual observations, often fail to transfer their knowledge to unseen environments. In games, standard Deep Reinforcement Learning protocols commonly promote testing in the same set of levels used in training. This practice leads an agent to easily overfit a given training set, failing to transfer its knowledge to out of distribution levels. To overcome this problem, we construct two separate training and test sets using procedurally generated environments from the Procgen Benchmark. We use this benchmark to measure the extent of overfitting and systematically study the effects of using regularization and data augmentation methods on the capacity of the agent to generalize. We found that, in general, using regularization and data augmentation improves generalization, with an efficacy that is dependent on the environment's dynamics. Furthermore, we study how network architectural decisions such as the depth and the width of the convolutional network, the usage of pooling layers, skip-connections, and modifications of the classification layer affect generalization. Finally, we empirically demonstrate that convolutional neural networks with small kernels in the early convolutional layers can accomplish the same generalization level as a deeper residual model. The code used for this dissertation is publicly available on https://github.com/MiguelFreire/rl-generalization.

Keywords

Deep Reinforcement Learning; Generalization; Overfitting; Procedural Generated Content.

Resumo

Agentes de aprendizagem profunda por reforço, principalmente aqueles que aprendem de observações visuais tendem a falhar a transferência do seu conhecimento para ambientes nunca antes vistos. Em jogos, é comum protocolos de aprendizagem profunda por reforço promoverem o teste no mesmo conjunto de níveis usado durante o treino. Esta prática leva o agente a dar overfit no conjunto de treino, não conseguindo transferir o seu conhecimento para níveis fora da distribuição. Para ultrapassarmos este problema, construímos dois conjuntos separados de treino e de teste usando ambientes gerados processualmente do Procgen Benchmark. Usamos este *benchmark* para medir a extensão do *overfitting* e estudar sistematicamente os efeitos da regularização e de métodos de augmentação de dados, frequentemente usados em aprendizagem supervisionada, na capacidade de generalização do agente . Descobrimos que, em geral, usar regularização e augmentação de dados melhora a generalização, e que a sua eficácia está dependente das dinâmicas do ambiente. Além disso, estudámos como as decisões sobre a arquitetura neuronal, tais como a profundidade e largura da rede, o uso de camadas de *pooling*, *skip-connections*, e como modificações à camada de classificação, afetam a generalização. Finalmente, demonstramos empiricamente que uma rede convolucional com filtros pequenos nas primeiras camadas convolucionais consegue atingir o mesmo nível de generalização que modelos residuais mais profundos. O código usado nesta dissertação está disponível publicamente em https://github.com/MiguelFreire/rl-generalization.

Palavras Chave

Aprendizagem Profunda por Reforço; Generalização; Overfitting; Conteúdo gerado por procedimentos.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ALE Arcade Learning Environment

ANN Artificial Neural Network

CNN Convolutional Neural Network

DQN Deep-Q Network

DL Deep Learning

DRL Deep Reinforcement Learning

FC Fully Connected

GAE Generalized Advantage Estimation

GG Generalization Gap

MDP Markov Decision Process

MLP Multilayer Perceptron

PPO Proximal Policy Optimisation

RL Reinforcement Learning

1

Introduction

Contents

1.1 Motivation

Deep Reinforcement Learning (DRL) has been broadly used in recent years to train agents capable of playing games and solving tasks at expert level. Some examples of these agents are MuZero [1] for board games such as Chess, Go, Shogi and Atari games, AlphaStar [2] for Starcraft II, OpenFive [3] for Dota 2 and DeepStack [4] for Poker.

Despite their broad success deep reinforcement learning algorithms require millions or even billions of data points and huge computational resources to train agents to achieve master performance. Even though these agents can perform at the highest level in the environment they were trained on, they fail, sometimes catastrophically, to transfer their knowledge to unseen environments. Biological agents, however, are able to learn quickly and to generalize to a number of different tasks and environments, using methods that are still not fully understood. [5, 6].

Classic Reinforcement Learning (RL) game benchmarks commonly promote training and testing in the same environment, but recent studies [7–9] have shown that the capacity to generalize to unseen environments can be used to optimize data efficiency and develop more robust and capable agents. For example, in real-world applications of reinforcement learning like robotics the agent must be trained in a simulated environment with a large collection of data and then have its knowledge transferred to a real-world environment [10]. These challenges are also present in pixel-based reinforcement learning, where the agent learns from pixel-based observations, due to its high-dimesionality and partial-observability [11]. Bridging the gap between generalization and data-efficiency is pivotal not only to develop better and robust agents for digital environments such as videogames but also for real-world applications such as robotics.

1.2 Thesis Proposal

In this work, we provide a systematic empirical study on how different algorithmic and architectural decisions impact generalization in pixel-based reinforcement learning in games. We use the power of procedural generation to evaluate the level of overfitting by using two different sets of levels, one for training and another one for testing.

1.3 Contributions

The main contributions of this dissertation are as follows:

- We extend a previous empirical study [7] on how different neural network architecture modules, regularization methods and data augmentation techniques impact generalization by applying it to

different environments and by using a smaller neural-network architecture.

- We identify the source of generalization bottlenecks in small neural-network architectures helping to bridge the gap between these and more robust models such as deeper Residual Networks.
- We employ a visual method (GradCAM) to understand the effects of different algorithmic and neural architectural decisions on the learned vector produced by the convolutional encoder.

1.4 Structure of the document

This dissertation is structured as follows. Chapter 2 introduces fundamental concepts and background for DRL as well as related work on the topic of Generalization in pixel-based DRL. Chapter 3 describes the problem at hand, the proposed solution, and the experimental setup. Chapter 5 describes all experiments, presents all results and associated discussions. Finally, Chapter 6 presents the main findings and provides possible directions for future work.

2

Background

Contents

This chapter presents all background, fundamental concepts and related work referred in this dissertation, allowing for readers without a background in Deep Reinforcement Learning to be introduced to the topic. In Section 2.1 we introduce the learning framework of Reinforcement Learning. In Section 2.2 we introduce deep learning and regularization methods. In the following section, 2.3, we introduce Deep Reinforcement Learning that allows the usage of Neural Networks as function approximators for Reinforcement Learning algorithms. Finally, in section 2.4 we cover the work related to the topic.

2.1 Reinforcement Learning

Reinforcement Learning is a learning framework where an agent learns to solve a sequential decision problem through trial-and-error by interacting with an environment. This framework can be mathematically described as a Markov Decision Process (MDP). A MDP is described by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of actions the agent can execute, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition probability of transitioning from one state to another given an action, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function that gives a real value for a given action and state, and $0 < \gamma \leq 1$ the discount factor which decreases the value of rewards received in the future. The interaction between the agent and the environment can be done in a continuing setting (infinite horizon task) or in a episodic setting (finite horizon task). In this work we are only concerned with the episodic setting. In this setting each episode is further broken into individual timesteps t. At each timestep t the agent observes a state s_t executes an action a_t, transitions to a new state s_{t+1} and receives a reward r_t. Let $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ be a policy that determines the probability of executing a particular action for a given state and π_θ a policy parameterized by a weight vector θ. The goal of the agent is to find a policy π_θ parameterized by θ that maximizes an objective function J_θ.

$$J_{\pi_\theta} = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]. \tag{2.1}$$

The most common objective functions in the episodic setting are the undiscounted reward-to-go (Eq. 2.12 with $\gamma = 1$) and the discounted reward-to-go (Eq. 2.12 with $\gamma < 1$). In policy based methods, which we will explain in section 2.1.2, it is often used the average per-step reward as an objective function:

$$J_{\pi_\theta} = \lim_{T \rightarrow \infty} \frac{1}{T} \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]. \tag{2.2}$$

2.1.1 Value functions

Value functions are functions that describe how good it is for an agent to be in a given state or executing a given action based on expected future rewards. The value of a state s following policy π is defined as

the expected return when starting in state s and following π thereafter. It's formally defined as follows:

$$V_\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_t = s]. \tag{2.3}$$

We can define a similar function for each state-action pair which describes how good is for an agent to be in a given state and executing a given action based on expected future rewards. The value of a state s and executing action a following policy π is defined as the expected return when starting in state s, executing action a and following policy π thereafter. It's formally defined as follows:

$$Q_\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_t = s, a_t = a]. \tag{2.4}$$

We call $V_\pi(s)$ the state-value function and $Q_\pi(s, a)$ the action-value function.

Another value function that can be defined is the advantage function, which is the difference between the action-value function $Q_\pi(s, a)$ and the state value function $V_\pi(s)$ and measures how good it is to take an action a in state s compared to taking the average action at state s. It is formally defined as follows:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.5}$$

2.1.2 Solving the MDP

Solving a reinforcement learning problem, that is solving the MDP, means finding a policy that maximizes some objective function in the long run. A policy π is said to be better or equal to a policy π' if its expected return is greater or equal to the one of π' which can be defined formally as $\pi \leq \pi'$ if and only if $V_\pi(s) \leq V_{\pi'}(s)\ \forall s \in \mathcal{S}$. An optimal policy π^* is defined as a policy that is equal or better than all other policies: $V_{\pi^*}(s) \leq V_\pi(s)\ \forall \pi, \forall s \in \mathcal{S}$. Similarly, we can define optimal value functions as:

$$V^*(s) = \max_\pi V_\pi(s), \forall s \in \mathcal{S}, \tag{2.6}$$

$$Q^*(s, a) = \max_\pi Q_\pi(s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \tag{2.7}$$

The optimal policy can than be extracted directly from the optimal action-value function:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a). \tag{2.8}$$

We can rewrite equation 2.3 as a recursive definition:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[r_t + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_t = s], \\ &= \mathbb{E}[r_t + \gamma V_\pi(s_{t+1}) \mid s_t = s]. \end{aligned} \tag{2.9}$$

The same can be done to the action-value function in Equation 2.4:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a], \\ &= \mathbb{E}[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})) \mid s_t = s, a_t = a]. \end{aligned} \tag{2.10}$$

The two equations above are called Bellman Equations.

For tasks with finite MDPs (i.e processes with a finite set of states and actions), we have guarantees of the existence of a solution which is independent of the policy and is a fixed point of the Bellman Equation. The recursive relation in the Bellman equation can thus be used to compute the optimal policy.

There are several methods to compute the optimal policy or a near optimal one. If the environment's transition dynamics are known or can be estimated, one may use Dynamic Programming to compute the optimal policy. The methods where the environment's dynamics are known or can at least be approximated are known as model-based methods. Another kind of methods that exploit the recursive relationship on the Bellman equation are value-based methods which directly compute or approximate value functions using Monte Carlo methods from complete episode's trajectories or Temporal Difference Learning using incomplete episode trajectories. The third and final method to compute the solution is to directly optimize a parametrized policy using collected data and the policy gradient theorem (equation 2.11).

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\pi \ln \pi_\theta(a|s) \Psi_t]. \tag{2.11}$$

All methods that exploit the policy gradient theorem are called policy-based and play an important role in Deep Reinforcement Learning. There are several different functions Ψ_t that can be used in equation 2.11, for example the action-value function (2.4), undiscounted total reward 2.12 or the advantage function (2.5).

When using temporal difference methods or policy-based methods, one typically optimizes the policy that the agent is currently following using data collected using the same policy. This type of learning is called on-policy learning. If the agent is instead optimizing a policy using data collected using a different policy this type of learning is called off-policy learning.

Policy-based methods usually have better convergence guarantees, are effective in high-dimensional state and action spaces, can learn stochastic policies but are subject to local optima and are sample

inefficient (high variance). On the other hand, value-based methods are more sample efficient but are less stable and can only model deterministic policies.

We can combine policy-based methods and value-based methods by decoupling the policy optimization and value function learning in the policy gradient. These types of methods are called actor-critic and have two components: the actor who executes a policy π_θ and the critic, which is typically a value-function, evaluates a policy.

Note that for small state and action spaces, it is possible to use matrices to represent any of the value-functions and the policy, but when the state and action space starts to grow the computational memory grows quadratically to each state-action pair so we need to find compact representations (approximations) of these functions. Furthermore, the state space is, in many cases, exponential or the dimension of the problem.

2.1.3 Exploration-Exploitation Dilemma

One of the most famous problems in Reinforcement Learning is the trade-off between exploration and exploitation. Should the agent explore and try new actions and perhaps find better policies or exploit the knowledge that it has to earn high reward while probably missing the optimal policy? This dilemma exists because the agent cannot exclusively choose one of the two strategies without failing. The agent must try several actions and favor the ones that bring higher reward. The control between the two strategies is key to develop agents that can master tasks. There are several strategies to tackle this problem. Here, we present two. The first is the ϵ-greedy strategy where the agent as an ϵ probability of choosing a random action instead of the one chose by its current policy, decreasing ϵ over time to favor exploitation. In policy-based methods, the policy can also be optimized to favor long-term entropy which provides early exploration, i.e unpredictability when taking actions. The entropy term can be added to the objective function: 2.12:

$$J_{\pi_\theta} = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t + k\mathcal{H}(\pi_\theta(s_t))], \tag{2.12}$$

where k is a weight that controls the strength of the entropy's term.

2.2 Deep Learning

Deep Learning (DL) is a class of machine learning methods based on deep Artificial Neural Network (ANN), that can model compact representations or complex relationships through a sequence of non-linear transformations. Artificial neural networks are represented by a composition of layers that transform the input through non-linear operations and the sequence of these operations can represent different levels of abstraction. Deep Learning's objective is to find a function $f : \mathbf{X} \rightarrow \mathbf{Y}$ parametrized by

a vector $\theta \in \mathbb{R}^n (n \in \mathbb{N})$:

$$y = f(x, \theta), \tag{2.13}$$

whose parameters can be directly learned by minimizing a loss function that evaluates how good is the neural network's prediction. The most famous method to optimize the parameters of a neural network is based on gradient descent using the backpropagation algorithm [12]. In its simplest form, the parameters are updated using the gradient of the loss function L with respect to parameters θ, given a learning rate α:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta L(\theta_t), \tag{2.14}$$

which indicates how much the weights will change at each update. This rate can be fixed or adaptive i.e varies during training. Another way to update the parameters of the model is by using mini-batch gradient descent where it approximates the loss' gradient at each step by averaging the loss' gradient over a batch of datapoints:

$$\nabla_\theta L(\theta_t) \approx \frac{1}{N} \sum_{n=0}^{N} \nabla_\theta L(\hat{y}_n, y_n), \tag{2.15}$$

resulting in a trade between learning efficiency and oscillations towards the solution.

There are several variants of Neural Networks, each one with its own applications and advantages. The simplest type of Neural Network is the Multilayer Perceptron (MLP) or Fully Connected (FC). The MLP is defined as a succession of layers where information is fed forward, each layer has an arbitrary number of units called neurons, each neuron from layer n^{th} is connected to every neuron in layer $(n+1)^{th}$ where a non-linear function σ is applied to a weighted average of its inputs plus a bias term:

$$h_n = \sigma(W_n x_n + b_n), \tag{2.16}$$

where x_n is the input of size n_{x_n}, W_n a learnable matrix of size $n_{x_n} \times n_{h_n}$, b_n a learnable vector of size n_{x_h} and σ a differentiable non-linear function. Figure 2.1 shows an example of a MLP network.

Convolutional Neural Network (CNN), are a type of neural networks that use convolutional layers to capture spatial relationships and patterns within data. A convolutional layer is based on how the human's visual cortex and its receptive field perceives and responds to visual patterns. In a convolutional layer, a learnable kernel or receptive filter of size $F \times F$ is applied using a convolution operation to an input of size $W \times H \times C$ and generates a lower-dimensional feature map W_n that retains some of its input information and passes it to the next layer. A example of a convolution layer is shown in figure **??**. The first layers of the network, which have higher resolution, are responsible for capturing low-level patterns such as edges and colors while deeper layers are responsible for capturing high-level patterns such as curves, figures and textures. Finally, the feature map from the last convolutional layer is flattened and fed to a fully-connected layer in order to produce a low-dimensional vector that can be used for classification

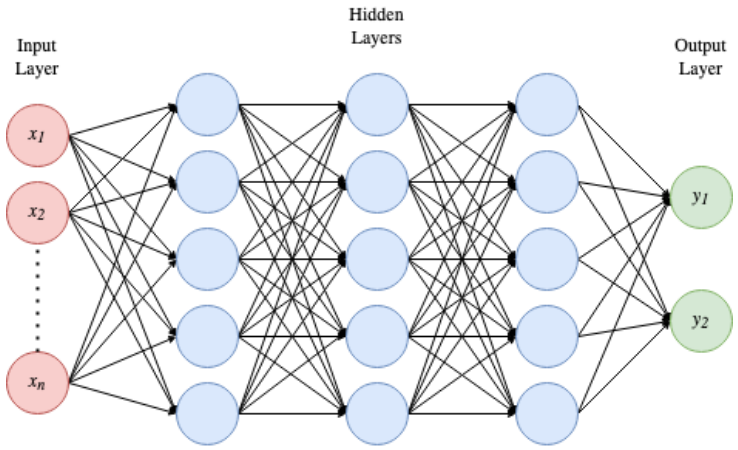


Figure 2.1: An example of a MLP network with n inputs, 3 hidden layers, and 2 outputs. Each neuron in the hidden layer applies a non-linearity to the weighted sum of its inputs

and regression tasks.

After each convolutional layer it is common to insert a pooling layer, which is responsible to reduce the spatial size of the feature map thus reducing the number of parameters and computation operations. Pooling layers can also work as regularizers as they prevent small changes in the input to have a major effect in the produced feature map. The most common pooling layer is the Max Pooling which applies a max operation channel-wise over an input using a $K \times K$ filter. An example of a max pooling operation is shown in figure 2.3.

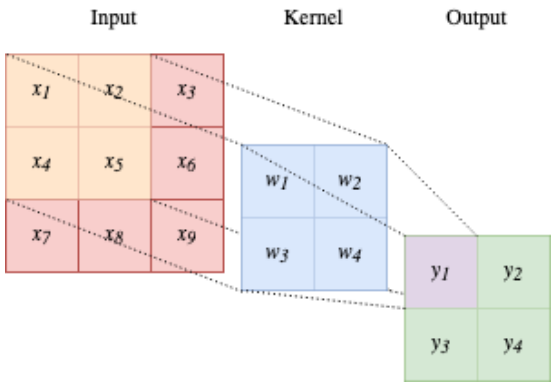


Figure 2.2: An example of a Convolutional Layer where a 2×2 kernel is applied to a $3 \times 3 \times 1$ input which generates a 2×2 feature map

Residual Networks, are a special type of CNN that use skip-connections to improve the flow of information to deeper layers. These type of layers allow to combine low-level and high-level features in order to create richer feature maps while also preventing the vanishing of gradients during backpropagation. A Residual layer can be formaly defined by:

$$y_n = \sigma(W_n x_n) + x_n, \tag{2.17}$$

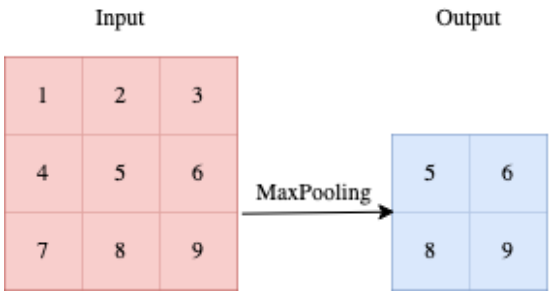


Figure 2.3: An example of a MaxPooling Layer where a 2×2 kernel is applied to a $3 \times 3 \times 1$ input which generates a 2×2 output

where x_n is the input of size n_{x_n}, W_n a learnable matrix of size $n_{x_n} \times n_{h_n}$, and σ a differentiable non-linear function. Since the dimensions of the non-linear output σ and x_n must be the same, a linear projection W_l is often applied to the input to match both dimensions:

$$y_n = \sigma(W_n x_n) + W_l x_n. \tag{2.18}$$

2.2.1 Preventing Overfitting in Deep Learning

Deep Learning uses models with a large number of parameters that can model complex relationships within data. These type of models are often prone to overfitting. Overfitting occurs when a machine learning model captures relationships and noise within the training data but fails to generalize to unseen data points. In this section we present the regularization methods used in this work to prevent overfitting in deep neural networks.

The simplest regularization method that doesn't require changing the network architecture is to impose a constraint in the loss function that is proportional to the size of the parameters. These type of methods encourage the weights of the model to be kept small. In this work we use the sum of L2-norm of the parameters to constraint their size:

$$L^{regularized} = L^{original} + w \sum_{k=0}^{K} \theta_k^2, \tag{2.19}$$

where $w \in \mathbb{R}$ controls the regularizing effect. Other regularization methods that prevent overfitting but require changing the neural network architecture are Dropout and Batch Normalization.

Dropout [13] is a technique where some neurons and their connections are dropped with a probability p during training. It makes it possible to train several different pruned smaller networks and test on the complete larger network with smaller weights.

Batch Normalization [14] is a method that stabilizes layers input distributions by normalizing them within the training batch. While it is a method to stabilize learning by allowing higher learning rates it also has

a regularizing effect. For each batch, the input is normalized following:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{Var[x] - \epsilon}} * \gamma + \beta, \tag{2.20}$$

where the mean and standard-deviation are calculated per-dimension over the mini-batch, β and γ are learnable parameters and ϵ a small positive constant for numerical stability. During training this layer keeps the running estimates of its computed mean and variance, which is then used for normalization during testing.

2.3 Deep Reinforcement Learning

In this section we introduce the deep reinforcement learning algorithms and methods used in this work. As we are working with pixel-based reinforcement learning it is unfeasible to model policies and value-functions as matrices. We make use of deep neural networks as function approximators to model compact representations of policies and value-functions that can be trained end-to-end using deep reinforcement learning algorithms. The neural architectures used in pixel-based reinforcement learning are based on convolutional networks, where the input is the environments screen represented by a 3D tensor, where the first dimensions corresponds to the RGB channel, the second dimension to the screen's width and the third to the screen's height.

In actor-critic methods, one can use two separate neural networks to approximate the actor (policy) and the critic (value function) but in literature is standard just to use one network with a shared convolutional encoder and two output heads corresponding to the policy and value function.

2.3.1 Proximal Policy Optimisation

Model-free policy gradient poses some major challenges such as large policy changes that can cause training to diverge, improper learning rates that can cause vanishing or exploding gradients and sample inefficiency. The main idea behind constrained policy optimization is to enforce a constraint on the update of the policy at each iteration.

Proximal Policy Optimisation (PPO) [15] is an on-policy first-order method that updates the current policy with the biggest step possible without stepping to far to cause the learning to collapse. There are two versions of PPO: PPO-Penalty which adds a KL-divergence term to the objective function and PPO-Clip that clips the policy update in order to maintain the updated policy close to the previous one. Here we'll present the clipped version which is the one used in this work.

The objective function that is maximized by gradient ascent at each iteration t for PPO is:

$$L_t(\theta) = \mathbb{E}[L^{\pi} - k_v L^{VF} + k_w \mathcal{H}(\pi_\theta(s_t))], \tag{2.21}$$

where k_v, k_w are two coefficients that control the weight of each loss term, L^{π} is the policy objective, L^{VF} the loss for the value function and $\mathcal{H}$ denotes the entropy bonus to control the Exploration-Exploitation trade-off as mentioned in section 2.1.3.

Let $r_t(\theta)$ denote the probability ratio between the old and the updated policy $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$. The policy objective is given by:

$$L_t^{\pi} = \min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right), \tag{2.22}$$

where ϵ is a hyperparameter and $\hat{A}_t$ an estimator for the Advantage function. The estimator used in this work is Generalized Advantage Estimation (GAE) [16] and is presented in the next chapter.

The motivation for this objective is to take the minimum update step between $r_t(\theta)\hat{A}_t$ and its clipped version $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$. The clipped version clips the probability ratio, which removes the incentive to move the probability ratio r_t outside $[1-\epsilon, 1+\epsilon]$, thus having a lower-bond on the unclipped version. If a given state-action pair (a_t, s_t) has negative advantage $\hat{A}(a_t, s_t) < 0$ the optimizer should make $\pi_\theta(a_t|s_t)$ smaller, but there is no motivation to make it smaller than $(1-\epsilon)\pi_{\theta_{old}}(a_t|s_t)$. If, on the other hand, a given state-action pair (a_t, s_t) has positive advantage $\hat{A}(a_t, s_t) > 0$ the optimizer should make $\pi_\theta(a_t|s_t)$ bigger, but there is no motivation to make it bigger than $(1+\epsilon)\pi_{\theta_{old}}(a_t|s_t)$.

Finally the value-function loss is given by the squared-loss:

$$L^{VF} = \frac{1}{2}(V_\theta(s_t) - R_t)^2, \tag{2.23}$$

where $V_\theta(s_t)$ is the estimator of the state-value function for state s_t and R_t the sum of discounted rewards.

Let $V_\phi(s)$ and $\pi_\theta(s|a)$ be approximated by neural networks and parametrized by vectors ϕ and θ respectively. At each iteration, N actors collect T timesteps of data following policy π_{θ_k}, the neural networks are then optimized for NT steps using the ADAM optimizer [17] and updated following:

$$\theta_{k+1} = \arg\max \frac{1}{T}\sum_{t=0}^{T} \min\left(r_t(\theta_k)\hat{A}_t, \text{clip}(r_t(\theta_k), 1-\epsilon, 1+\epsilon)\hat{A}_t\right), \tag{2.24}$$

$$\phi_{k+1} = \arg\min \frac{1}{T}\sum_{t=0}^{T} \frac{1}{2}(V_{\phi_k}(s_t) - R_t)^2, \tag{2.25}$$

for K epochs. The full algorithm is presented below:

Algorithm 2.1: Actor-Critic PPO Clipped

Input: Initial policy parameters θ, initial value-function parameters ϕ
for K epochs **do**
  foreach actor **do**
    Run policy π_{θ_k} and collect trajectories $\tau = (s_t, a_t, r_t, s_{t+1})$ with T timesteps
    Compute rewards-to-go $R_1, ..., R_T$
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end
  Update policy parameters following:
  $\theta_{k+1} = \arg\max \frac{1}{NT} \sum_{t=0}^{T} \min\left(r_t(\theta_k)\hat{A}_t, \text{clip}(r_t(\theta_k), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$
  Update state-value function parameters following:
  $\phi_{k+1} = \arg\min \frac{1}{NT} \sum_{t=0}^{T} \frac{1}{2}(V_{\phi_k}(s_t) - R_t)^2$
end

2.3.2 Generalized Advantage Estimation

As we explained in the previous section, to optimize the policy using PPO we need to estimate advantages at each timestep. Most techniques developed to compute a low-variance estimator for the Advantage function make use of a learned state-value function such as a Neural Network. [18] uses an n-step TD-residual of the state-value function with discount γ:

$$\hat{A}_t^{(n)} = \sum_{i=0}^{n-1} \gamma^l \delta_{t+i}^V, \tag{2.26}$$

where $\delta_t^V = r_t + V(s_{t+1}) - V(s_t)$. While the estimator bias decreases with the increase in the number of steps, since the terms become exponentially more discounted with $n \to \infty$, the variance increases. GAE is a method that uses a exponentially-weighted average of n-step (2.26) estimators [16] which controls the tradeoff between the estimator's bias and variance using a parameter $0 < \lambda < 1$. GAE is defined as:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{i=0}^{\infty} (\gamma\lambda)^l \delta_{t+i}^V. \tag{2.27}$$

If $\lambda = 0$, we recover the advantage estimator using single-step state-value TD-residual which has high bias:

$$\hat{A}_t^{GAE(\gamma,0)} = \delta_t^V = r_t + V(s_{t+1}) - V(s_t). \tag{2.28}$$

By the other hand, if we set $\lambda = 1$ we recover the empirical return minus a baseline which has high variance:

$$\hat{A}_t^{GAE(\gamma,1)} = \hat{A}_t^{(\infty)} = \sum_{i=0}^{\infty} \gamma^l r_{t+i} - V(s_t). \tag{2.29}$$

2.4 Related Work

Deep Reinforcement Learning methods used in pixel-based reinforcement learning, commonly perform training and testing in the same environment. Recent studies were focused on measuring the capacity of trained agents to generalize to unseen levels.

Several studies [9, 11] identified that classic DRL algorithms such as Deep-Q Network (DQN) [19] and policy-based methods struggle to generalize to remarkably similar environments. They propose a protocol to evaluate the generalization capabilities of DQNs agents by using different modes of Atari games. They also show that regularization methods, widely used in the supervised setting, such as L2-regularization and dropout improves DQNs agents' generalization and performance in continuous control tasks. Other authors [20] measured the level of overfitting and capacity of generalization in continuous domains, for a system trained in a simulated environment and tested in a real-world application. They also noted that DRL algorithms generalize well if there is enough diversity in the training data.

In practice, evaluation protocols and Reinforcement Learning (RL) setups need to be designed in order to be able to detect overfitting. Several benchmarks were introduced to evaluate the capacity of the agents to generalize to out-of-distribuition levels or domains. The benchmark based on the classic game Sonic The Hedgehog [8] was designed to assess generalization by separating levels of the game into training and test sets as is standard in the supervised setting. They found that the agent struggles to generalize between sets while having difficulties to measure progress and overfitting. A similar benchmark [21] is based on several video games from the General Video Game AI framework [22], where the training and test sets are also generated using different game levels. They also found that agents regularly overfit to particular training distributions. Other authors proposed a different benchmark to measure generalization by using a modified version of classic control RL environments [23], which are controlled by a set of parameters. The generalization capability is quantified by training and testing in environments with different parameter ranges.

The aforementioned studies generated separate sets for training and testing by using different game levels, different difficulties or different parameters but this can cause issues when measuring generalization, given that the separated sets can have different dynamics. To overcome this issue, several proposals [24] evaluated overfitting in RL agents using procedurally generated gridworld mazes. They found that agents have a trend to memorize specific levels in a training set, and that techniques that add stochasticity and help prevent overfitting, such as sticky actions [25] and random starts [26], often fail to work in procedurally generated environments. The Procgen Benchmark [27], a set of 16 game-like procedurally generated environments were particularly designed to measure both generalization and sample-efficiency. These environments are based on the Arcade Learning Environment [28] and provide enough diversity to measure generalization. A recent study [7] used CoinRun, an environment from the Procgen benchmark, to generate large training and test sets to better evaluate generalization and over-

tiffing. It introduced a new metric, the Generalization Gap, to measure the level of overfitting between training and test sets. The results showed how regularization methods such as Dropout, BatchNorm, L2-regularization and data augmentation techniques can help improve generalization.

3

Evaluating Generalization in Deep Reinforcement Learning

Contents

This chapter provides a detailed description on the systematic empirical study performed in order to evaluate generalization of Reinforcement Learning agents trained in Procedural Generated Environments following the structure employed by previous works [7]. Section 3.1 provides details of the problem, section 3.2 details the environments used, section 3.3 introduces all the metrics and finally section 3.4 details the experimental setup.

3.1 Overfitting in Deep Reinforcement Learning and Generalization

In Deep Reinforcement Learning, in particular when using the model-free actor-critic framework, the value-function and the policy are approximated using a neural network. In the pixel-state setting, which is usually applied to videogames, the neural network architecture is usually a Convolutional Neural Network followed by a MLP, which outputs an approximation of a value-function and of a policy given an input state which is represented by the raw-pixels image of the game's screen. In the classic RL setting, the learning is made in a continuous setting without explicitly separate training and testing sets, where the objective is to maximize a discounted-cumulative reward over time. The usage of high-capacity models such as Neural Networks, within the classic RL training setting and with long-training times could lead to overfitting. Overfitting may be the consequence of a fundamental trade-off in Machine Learning. Controlling and regularizing training is key to the development of robust agents that learn a relevant skill and don't just memorize specific trajectories. In this setting, agents often perform at expert level in seen environments but fail to transfer experience to unseen ones, overfitting the ones seen during training. The capacity of an agent to perform at expert-level in unseen environments is called generalization.
The lack of generalization, overfitting, can happen by several reasons:

Data Size: The training samples are few and/or exists low diversity between them (high correlation)

Convolutional Encoder: The convolutional encoder does not produce an invariant low-dimensional representation vector of the input state.

MLP acting over latent vector: The MLP which acts over the latent vector produced by the Convolutional Encoder is not able to output a good value function or policy.

Policy stochasticity: Following the Exploration vs. Exploitation dilemma, policy stochasticity controls how much the agent wants to exploit what it knows or to explore new paths. If policy stochasticity is small the agent will keep following the same trajectories resulting in overfitting.

In the next sections we introduce the experimental setup used in this work to assess each one of the sources presented above and their dependencies affect the generalization capability of the agents. In our first experiment we assess how the size of the training set effects generalization. In our second experiment we evaluate if regularization methods, often used in Supervised Learning such as Dropout,

Batch Normalization and L2 Regularization, lead to an improvement in generalization in the RL setting. We also evaluate Entropy Regularization, which is often used in Policy-Based Actor-Critic methods to control the Exploration vs Exploitation Dilemma, as explained in chapter 2.1.3. In the third experiment we evaluate the impact of data augmentation in generalization and in on the fourth and final experiment we evaluate how architectural decisions effects generalization.

3.2 Procedurally Generated Environments

In order to evaluate generalization we need to separate training and testing into two sets. We use the power of procedural content generation, the algorithmic creation of a near-infinite supply of highly randomized content, to accomplish this. We use the Procgen benchmark [27], a set of 16 procedural generated games designed to benchmark generalization and sample efficiency. Procedural generation logic controls level layout, the spawning location and time of game entities, a selection of game assets and other details. To master any one of these environments, an agent must learn a robust policy that takes into account all variables related to the environment dynamics and not just overfit to a subset of fixed levels.

All the environments were designed to satisfy a condition of high diversity since there are various degrees of freedom for procedural generated content subject to very basic constraints. The benchmark also presents tunable difficulty with guarantees of very high percentage of solvable levels (99%). It is also optimized for fast evaluation. The set of environments have a shared observation and action space, with a focus on visual recognition and motor control based in other environments such as the Arcade Learning Environment (ALE).

In this work we chose three environments from the Procgen Benchmark:

CoinRun: A very simple platform game, where the objective is to collect a coin in the far right of the level while the player spawns at the far left. The player must overpass a series of obstacles such as stationary saws, moving enemies and crates. Procedural generated logic controls the location of crates, platforms, obstacles and enemies. The game also presents the option to add a colorful box at the top left corner of the screen where the color is proportional to the velocity of the player. If the player collects the coin, the episode (level) ends, giving the player a reward of 10. Otherwise, if the player falls into a crate or collides against an enemy, it receives a reward of 0 and the episode ends.

Jumper: A platform game, where the objective is to collect a carrot that spawned somewhere in an open-world map. The player must explore the world to find the carrot, using a ”double-jump” ability to reach higher platforms while dodging spike obstacles. A compass is presented in the top right corner of the screen that shows the direction and distance to the carrot. If the player collects the coin, the episode ends giving the player a reward of 10. Otherwise, if the player collides against a

spike obstacle, the episode also ends giving the player a reward of 0. Procedural generated logic controls the map layout through the use of cellular-automata.

Maze: A labyrinth game, where the player must find the sole piece of cheese to earn a reward of 10. Procedural generated logic controls the map size and layout by generating mazes using Kruskal Algorithm.

The three environments above give a reward of 0 at every time-step except the goal. The episode times out and ends without reaching the goal after 500 steps in Maze and 1000 steps in CoinRun and Jumper.

We chose these three environments based on the following requirements:

1. The baseline model can learn a policy that can solve more than half the levels in the training set.

2. The generalization gap between training and testing, for the baseline model, is noticeable i.e $GG > 10\%$

The first requirement guarantees that the baseline model can learn a policy that can solve a majority of levels in the training set. The second requirement guarantees that the generalization gap is big enough to notice the regularizing effect of our experiments.

3.3 Generalization Gap and Evaluation Metrics

In this section we present the evaluation metrics used in this study. The Generalization Gap (GG) [27] measures how successfully agents generalize from a training set of levels to an unseen testing set. It is defined as the difference between the percentage of levels solved in the training set and the percentage of levels solved in the testing set. We consider that a level is solved if the agent reaches the goal without dying or before the episode times out. The Generalization Gap also determines the extent of overfitting as training and test levels are drawn from the same distribution. If the percentage of levels in testing is higher than in testing there is no overfitting so we set a generalization gap of 0%. We use the Average Discounted Return to track agent performance during training. We also employ a visual gradient-based method called GradCAM [29] that highlights the locations of the screen the agent focuses in order to make a decision. The spatial attention maps are extracted by flowing the gradients from action taken (policy output), and into the first convolutional layer of the agent's Neural Network. We expect that a general convolutional encoder in order to produce invariant feature vectors will pay more attention to game deciding elements such as the agent's position, the objective, traps and enemies.

3.4 Training and Testing

The usage of procedural generated environment enables us to define two different sets of levels. The training of the agent is done in the training set and the evaluation is made on the testing set. There is no cross-over between the sets, the agent never sees the same level both in testing and training.

3.4.1 Training

Following the same experimental setup as done in a previous work [7], all the agents are trained using Proximal Policy Optimization with the advantage function being approximated using Generalized Advantage Estimation in order to control the bias-variance tradeoff. The choice of this training algorithm is justified by its stability and monotonic increase in performance. In terms of hyperparameters, we use the ones suggested by a previous work [7], which guarantees an increasing learning progress over time. All the hyperparameters values can be found in table 3.1. We don't change the hyperparameters values between experiences unless stated otherwise. We set the difficulty to all environments to "easy" and train using PPO for 25M steps in CoinRun and Jumper and 50M steps in Maze. We use 64 actors and 1 critic. Each actor collects trajectories with a max of 256 timesteps and sends them to the critic. We set PPO's clip value to be 0.2 and the entropy bonus to be 0.01. The actor uses 8 mini-batches of 256 timesteps for 3 epochs to optimize the policy following equation 2.24. The critic uses the Advantage Function as the value-function, estimated using GAE with λ set to 0.95 and optimized following equation 2.25. We use the Adam Optimizer to optimize the parameters of the model. Our baseline model is the NatureCNN [19], the architecture that consists of 3 convolutional layers and one fully-connected layer with two shared heads: one for policy and the other for the value-function. The first convolutional layer has 32 kernels of size 8x8 with stride 4 and padding 0, the second has 64 kernels of size 4x4 with stride 3 and padding 0, the third has 64 kernels of size 3x3 with stride 1 and padding 1. The convolutional encoder is connected to a fully-connected layer of 512 neurons which outputs a vector of size 15 that represents the policy π and a value that represents the value-function for a given input. The ReLU activation function is applied at the output of each layer. The input for the network is the game's screen image with size 3x64x64 for all environments. The input is normalized to be between 0 and 1. The neural network's architecture is shown in figure 3.1.

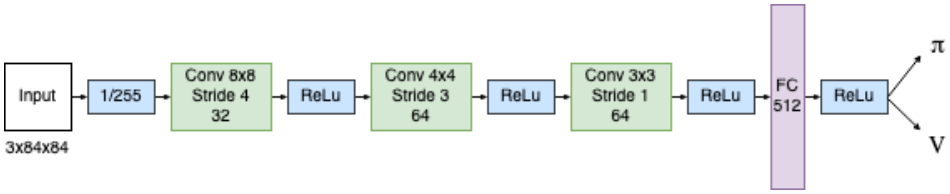


Figure 3.1: Nature CNN Architecture

Hyperparameter	Value
λ	.999
γ	.95
# timesteps per rollout	256
epochs per rollout	3
# minibatches per epoch	8
Entropy Bonus	0.01
Learning Rate	5×10^{-4}
# environments per worker	64
Workers	8
Total timesteps	25M (CoinRun, Jumper) 50M (Maze)

Table 3.1: Default hyperparameters' values for PPO Algorithm

We set our training size to be 200 levels in all experiments except for the first one where we vary the number of training levels in order to evaluate how generalization depends on training set size. We use this particular value because is where the Generalization Gap is still big between training and testing (as we show in the results Chapter), and allows us to evaluate which methods and architectures are effective in increasing generalization.

All agents were trained in Google Cloud's E2 Compute instances with 8 virtual CPUs, 32 GB of RAM and one NVIDIA's T4 Graphics Card. We use the rlpyt framework [30], a set of optimized algorithms and unified infrastructure for Reinforcement Learning that uses PyTorch [31] for all deep learning related operations to train our models. In each experiment, during training, we track the discounted average reward as a signal for how learning is progressing.

3.4.2 Testing

During testing, we assess the zero-shot-performance from three different training seeds and average the results. The testing set has 500 different levels that were not seen during training. We consider that the agent completed the level if it could reach the goal without dying or before the episode timed out. We also assess the zero-shot-performance in the training set to be able to calculate the generalization gap between both sets.

4

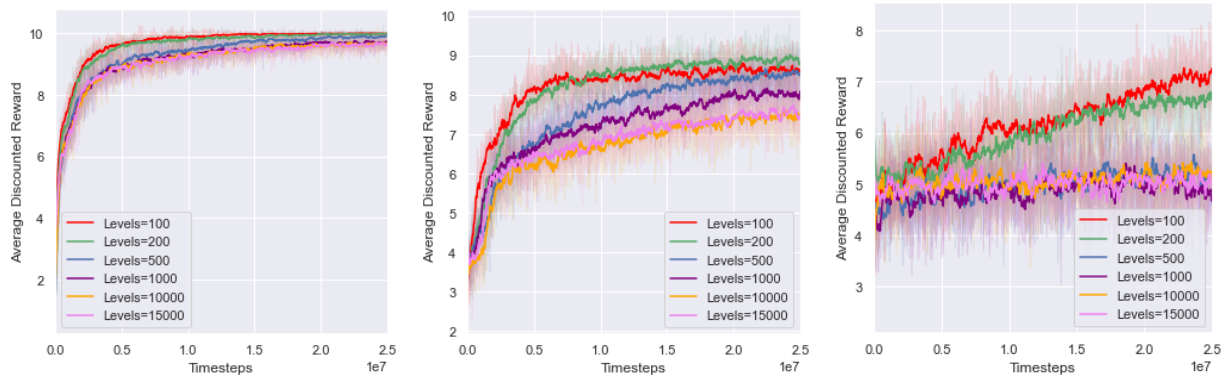
Results and Discussion

Contents

In this chapter, we present the experimental setup, related results and discussion. This experimental setup is based on a previous work [7]. We try to replicate the original authors' results while using a smaller neural network model (NatureCNN). We extend their study, by trying to identify the source of bottleneck in generalization capability of these types of architectures. In section 4.1 we analyse how the size of the training set can impact generalization. In section 4.2 and 4.3 we analyse how regularization and data augmentation methods impact generalization and learning. Finally, in section 4.4, we evaluate several architectures and different types of layers.

4.1 Evaluating Training Set Size

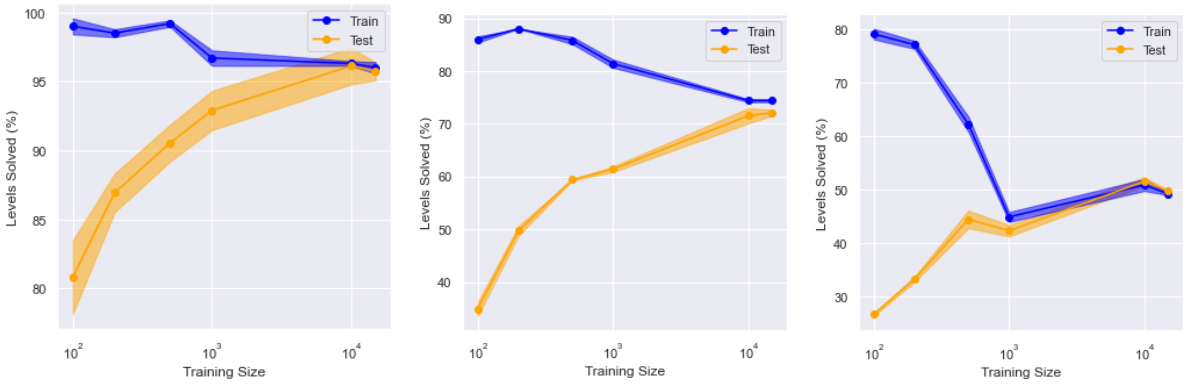
In this section we assess how training set size impacts training and generalization. We vary the training set size in the range $[100, 15000]$. With the access to a broader set of levels in training we expect the agent to be able to generalize better, thus decreasing the generalization gap (overffiting).



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.1: Training curves for several models trained with different set sizes

Results In this experiment we find that the agent overfits for small training sets in all environments. The increase of training set size allows the agent to build a curriculum able to generalize to unseen levels thus decreasing the generalization gap. For Maze, we also see a decrease in the generalization gap but a big drop in training performance with the increase of training size. This might be related to the nature of the environment. Compared to CoinRun and Jumper, Maze is an environment that requires more exploration and backtracking. This is expected to happen given the large variability of levels and the need for higher exploration capabilities. Original authors reported an unusual trend in this same experiment: past a certain threshold, training performance improves as the training set increases.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.2: Percentage of levels solved in train and test as a function of the training set size

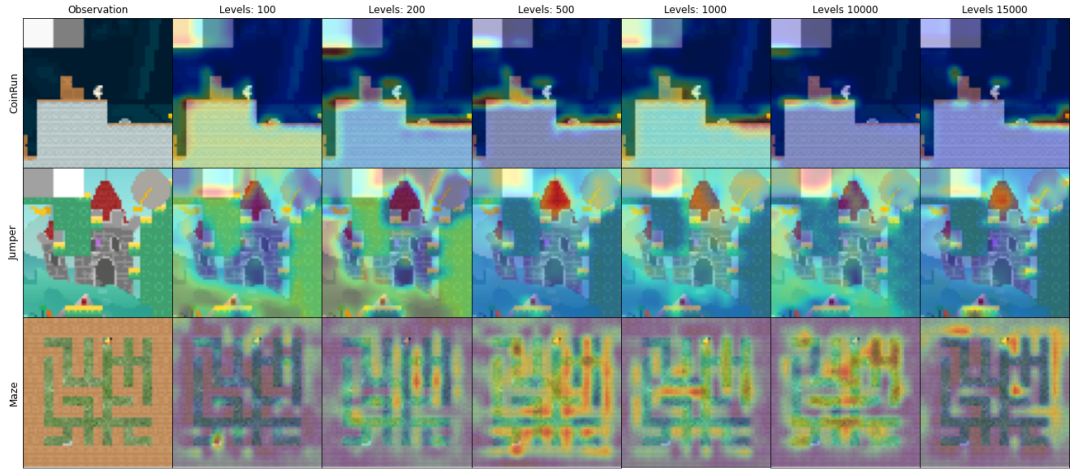


Figure 4.3: Attention Maps for several models trained with different training set sizes

As stated by the original authors, a larger training set can improve training performance if the agents learn how to generalize across a broad distribution of levels. Here we did not find this behavior. The only difference between the experiments is the model used. The original authors used the IMPALA model [32], while we used a smaller one, as claimed by previous authors: larger architectures improve sample efficiency and generalization. We present the results and discussion of using larger models and other architectures in section 4.4. We chose to use a smaller model given the limited computational resources and the large quantity of proposed experiments.

Figure 4.3 shows the attention maps extracted using GradCAM. The increase in number of levels of training shifts the agent's attention to more smaller and game-deciding elements such as: the objective, traps and its position.

For the remaining experiments, we set the training levels to 200 given that the results show a notice-

Environment	Training Set Size	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	100	99.00%	80.81%	18.19%
	200	98.50%	86.93%	11.57%
	500	99.20%	90.53%	8.67%
	1000	96.70%	92.90%	3.80%
	10000	96.30%	96.12%	0.18%
	15000	96.00%	95.73%	0.27%
Jumper	100	86.00%	34.93%	51.07%
	200	88.00%	49.73%	38.27%
	500	85.80%	59.40%	26.40%
	1000	81.40%	61.40%	20.00%
	10000	74.39%	71.53%	2.86%
	15000	74.38%	72.07%	2.31%
Maze	100	79.00%	26.73%	52.27%
	200	77.00%	33.30%	43.70%
	500	62.20%	44.47%	17.73%
	1000	44.90%	42.33%	2.57%
	10000	50.86%	51.53%	0.00%
	15000	49.17%	49.80%	0.00%

Table 4.1: Summary results for training set size experiment

able generalization gap with this set size across all environments.

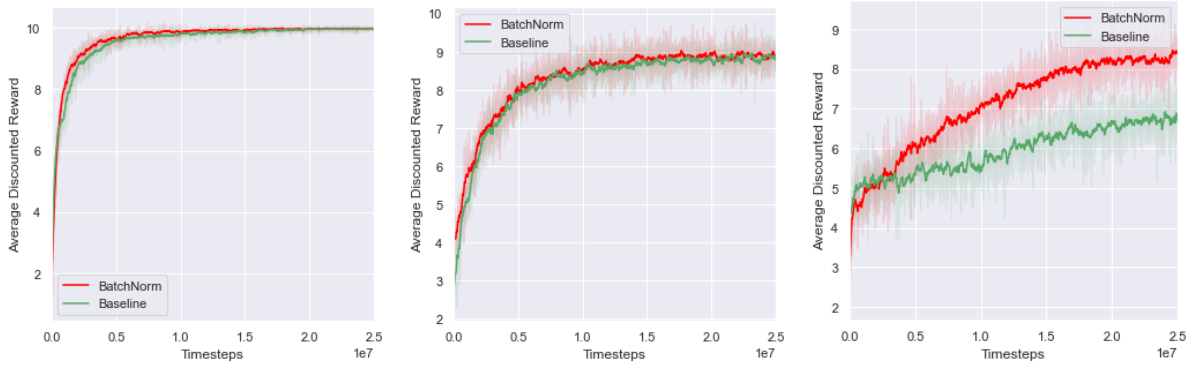
4.2 Evaluating Regularization Methods

In this section we show the results and discuss how regularization methods impact training and generalization.

4.2.1 Batch Normalization

In this subsection, we assess the usage of Batch Normalization. The baseline architecture NatureCNN is augmented with Batch Normalization after every convolutional layer. All running estimates of its input's computed mean and variance are saved during training and then the moving average is used for normalization during testing.

Results The usage of Batch Normalization results in a higher percentage of levels solved in the training set in all three environments. Figure 4.4 shows an increase in sample-efficiency as the agent earns a higher average discounted reward in less timesteps which also results in an increase in the percentage of levels solved in the training set. In Jumper, Batch Normalization shows a regularizing effect by increasing both percentage of levels solved in both sets while decreasing the generalization gap. On the other hand it shows the opposite effect in CoinRun and Maze. The regularizing effect of Batch Normalization seems to be environment-dependent and may have a stronger effect when working with larger training sets with higher data variability.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.4: Training curves for several models trained with and without Batch Normalization

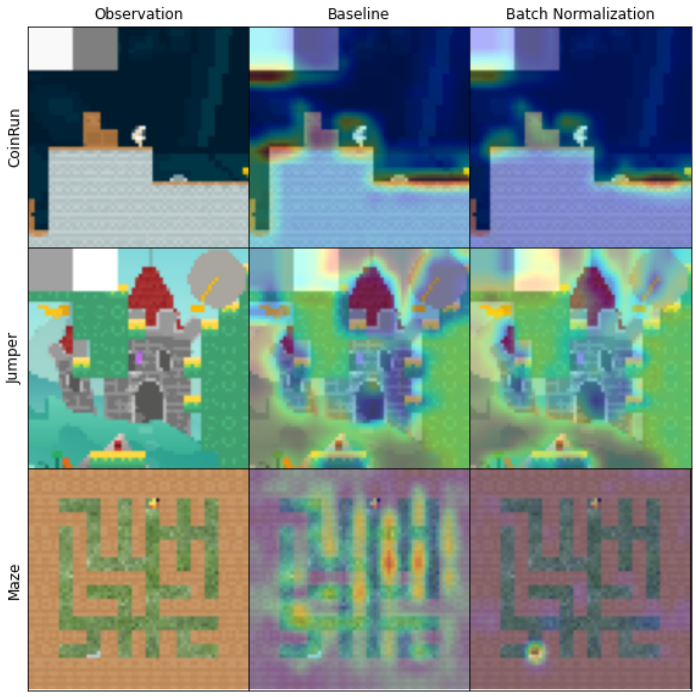


Figure 4.5: Attention Maps for models trained with batch normalization

4.2.2 Dropout

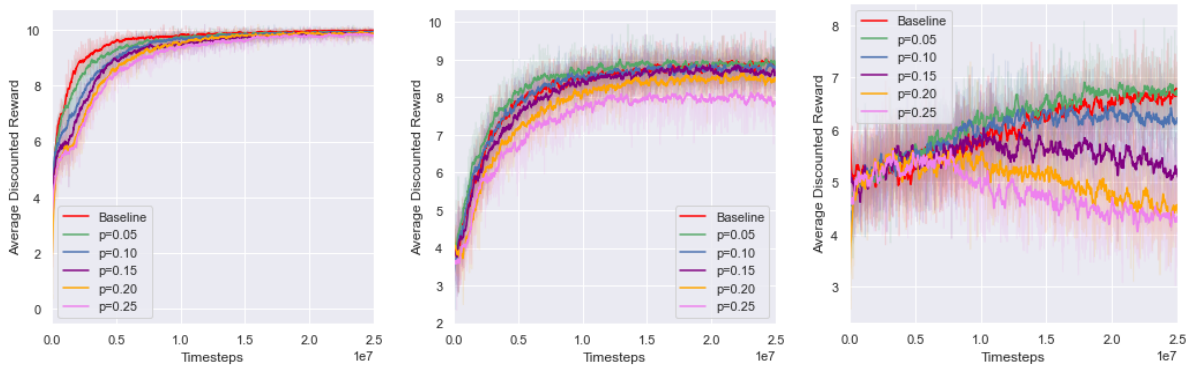
In this subsection we evaluate the impact of Dropout in training and testing. Each convolutional layer of the baseline architecture NatureCNN is augmented with Dropout with probability p . We vary p in the range $[0, 0.25]$. During testing Dropout's probability p is set to 0.

Results The average discounted reward for different values of p, shown in Figure 4.6, shows the decrease in sample efficiency when increasing the value of p. For higher values of p, sample efficiency is significantly reduced, needing more time-steps to converge to the same final average discounted reward as the baseline in all three environments.

We also report in Figure 4.7 and Table 4.3, the percentage of levels solved and the generalization gap showing a clear decrease in overfitting with the increase of p in all environments. Dropout is a method that induces stochasticity in the network nodes, making training noisy and adding variance to the process. Higher values of p induce an even higher variance in training, resulting in needing many more timesteps to converge to the same average discounted reward as the baseline agent.

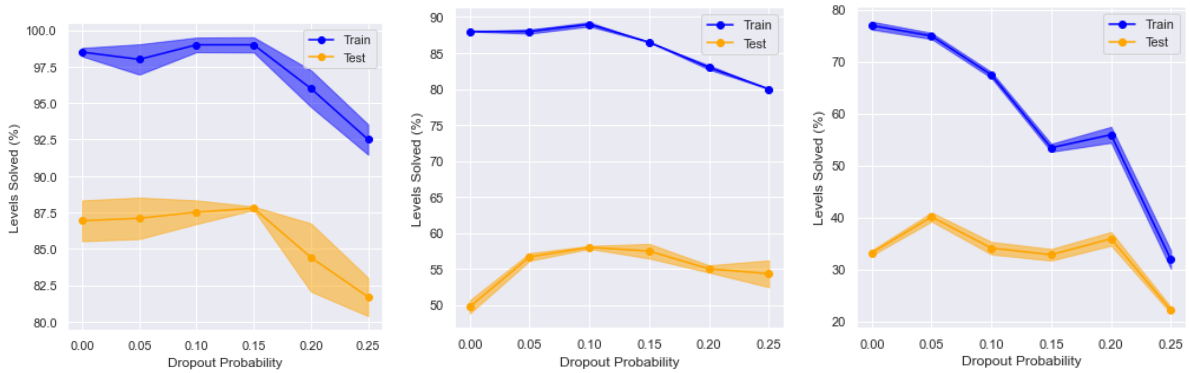
Environment	Model	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	Baseline	98.50%	86.93%	11.57%
	Batch Normalization	99.50%	87.26%	12.24%
Jumper	Baseline	88.00%	49.73%	38.27%
	Batch Normalization	89.50%	53.67%	35.83%
Maze	Baseline	77.00%	33.30%	43.70%
	Batch Normalization	83.00%	31.73%	51.27%

Table 4.2: Summary results for batch normalization experiment



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.6: Training curves for several models with different Dropout's probability p



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.7: Percentage of levels solved in train and test as a function of Dropout's probability p

Table 4.3: Summary results for dropout experiment

Environment	Dropout Probability	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	0.00 (Baseline)	98.50%	86.93%	11.57%
	0.05	98.00%	87.10%	10.90%
	0.10	99.00%	87.52%	11.48%
	0.15	99.00%	87.79%	11.21%
	0.20	96.00%	84.41%	11.59%
	0.25	92.50%	81.70%	10.80%
Jumper	0.00 (Baseline)	88.00%	49.73%	38.27%
	0.05	88.00%	56.67%	31.33%
	0.10	89.00%	58.00%	31.00%
	0.15	86.50%	57.47%	29.03%
	0.20	83.00%	55.00%	28.00%
	0.25	80.00%	54.33%	25.67%
Maze	0.00 (Baseline)	77.00%	33.30%	43.70%
	0.05	75.00%	40.20%	34.80%
	0.10	67.50%	34.20%	33.30%
	0.15	53.50%	32.93%	20.57%
	0.20	56.00%	36.00%	20.00%
	0.25	32.00%	22.27%	9.73%

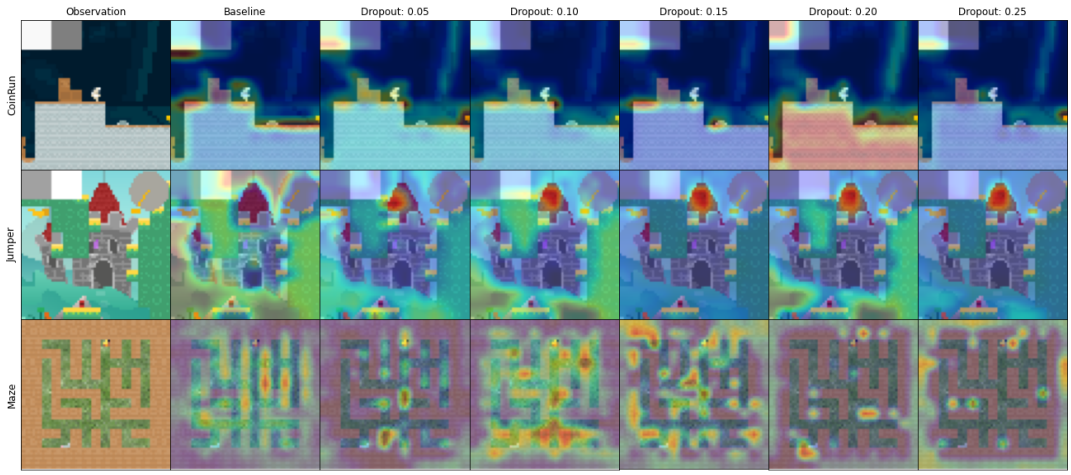
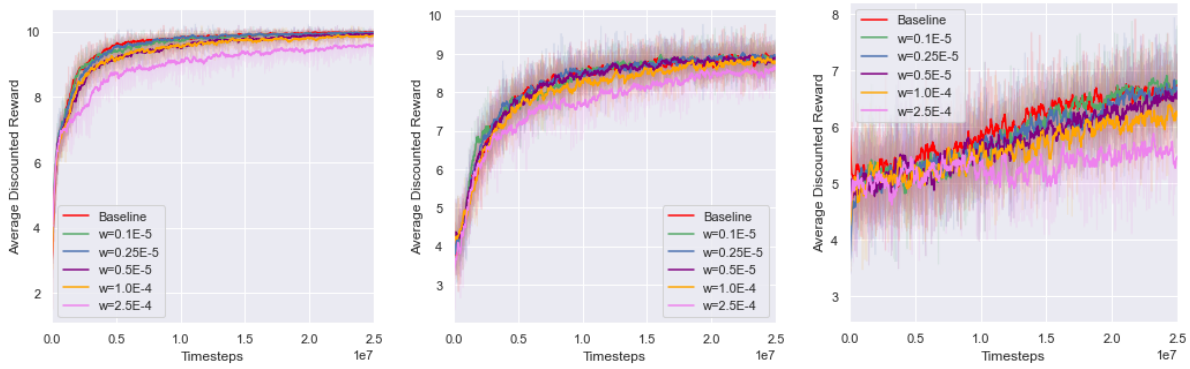


Figure 4.8: Attention Maps for several models trained with different dropout probability p

4.2.3 L2 Regularization

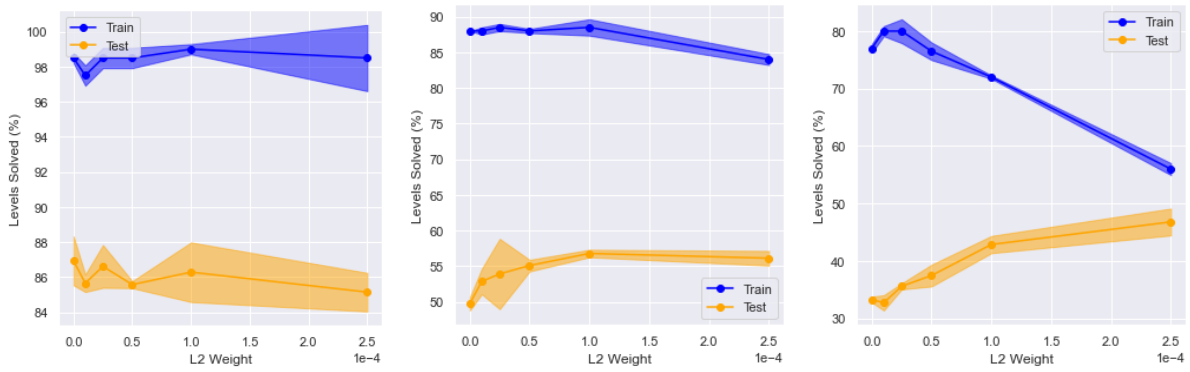
In this subsection we evaluate how L2 regularization impacts training and testing. We add a L2 penalizing term $w\|\theta\|_2^2$ to the loss function, where θ are the model's parameters and w a parameter that controls the weight of the regularization. We vary the parameter w in the range $[0, 2.5 * 10^{-4}]$.

Results Results are presented in table 4.4 and 4.10. Figure 4.9 shows the learning curves for all three environments. Higher values of L2 regularization weight cause a drop in sample efficiency and final discounted average reward, with a higher impact in Maze. L2's regularization weight seems to have a very small effect in CoinRun. On the other hand, in Jumper there is a 10% decrease in the generalization gap compared to the baseline while maintaining a constant successful rate in the training set. Maze presents the largest decrease in overfitting from all three environments for higher values of w.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.9: Training curves for several models trained with different L2's weight



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.10: Percentage of levels solved in train and test as a function of L2's weight

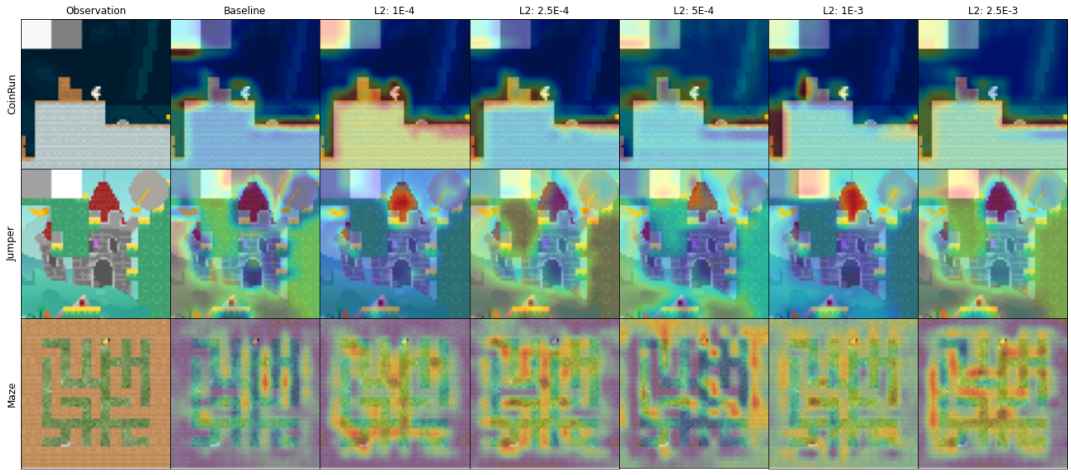


Figure 4.11: Attention Maps for several models trained with different values of L2's weight

Figure 4.11 shows the attention maps produced by GradCAM for models trained with different values of L2 weight. L2 regularization seems to cause a small-effect in the produced attention maps, as they all share similar attention patterns.

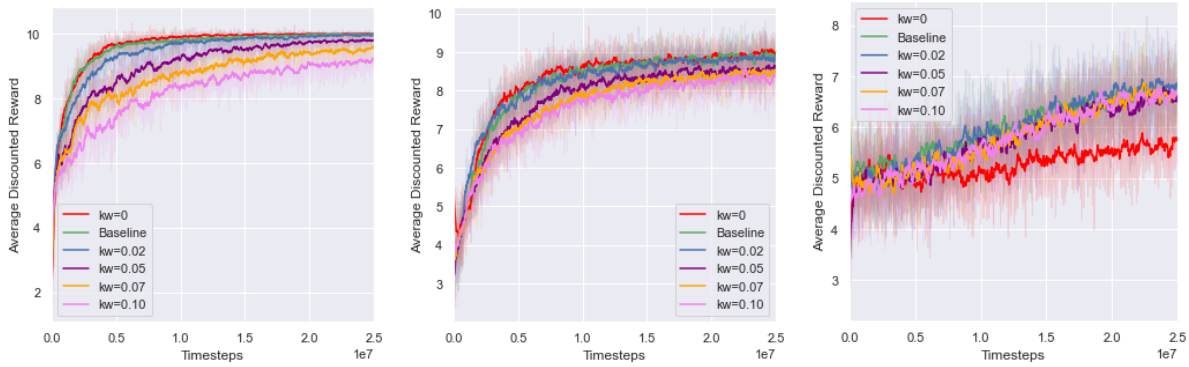
4.2.4 Entropy Regularization

In this section we evaluate the impact of varying the weight k_w that controls the level of the policy's stochasticity. We vary the weight k_w between $[0, 0.1]$. Note that the baseline model has an entropy weight of 0.01.

Table 4.4: Summary results for L2 regularization experiment

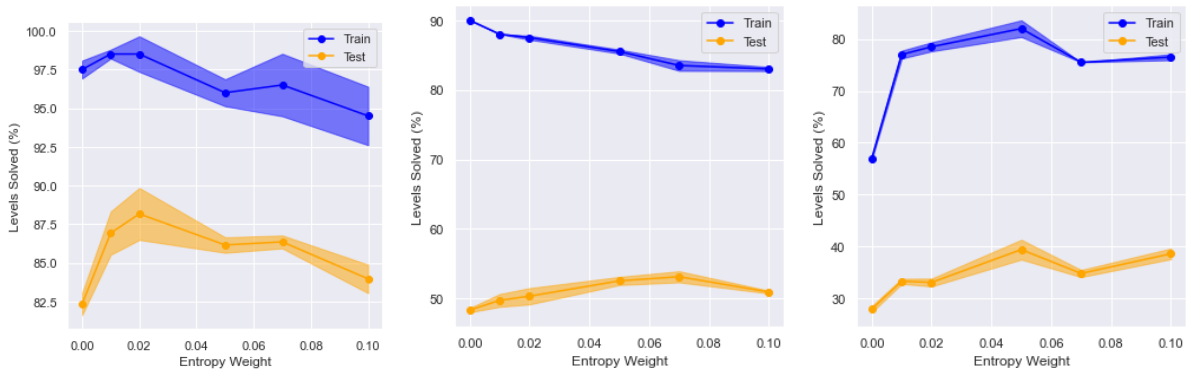
Environment	L2 Weight	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	0.00 (Baseline)	98.50%	86.93%	11.57%
	1.0E-5	97.50%	85.65%	11.85%
	2.5E-5	98.50%	86.61%	11.89%
	5.0E-5	98.50%	85.57%	12.93%
	1.0E-4	99.00%	86.28%	12.72%
	2.5E-4	98.50%	85.14%	13.36%
Jumper	0.00 (Baseline)	88.00%	49.73%	38.27%
	1.0E-5	88.00%	52.87%	35.13%
	2.5E-5	88.50%	53.93%	34.57%
	5.0E-5	88.00%	55.10%	32.90%
	1.0E-4	88.50%	56.80%	31.70%
	2.5E-4	84.00%	56.13%	27.87%
Maze	0.00 (Baseline)	77.00%	32.30%	43.70%
	1.0E-5	80.00%	32.73%	47.27%
	2.5E-5	80.00%	35.60%	44.40%
	5.0E-5	76.50%	37.48%	39.02%
	1.0E-4	72.00%	42.87%	29.13%
	2.5E-4	56.00%	46.80%	9.20%

Results The test results for this experiment are presented in 4.5 and figure 4.13 showing that an increase in k_w improves generalization. It is expected that higher values of k_w should result in higher early exploration which results in smaller discounted average reward early in training due to the high randomeness when taking an action. This trend is illustrated in figure 4.12(a) and 4.12(b). CoinRun and Jumper learning curves suggest that exploration is not a problem in this type of environments, given the inverse trend between final discounted average reward and k_w. On the other hand, Maze's learning curve shows a higher final discounted average reward when trained with higher values of k_w. The attention maps produced by GradCAM are presented in figure 4.14. The attention patterns for CoinRun and Jumper are very similar for different values of k_w. In Maze, the resulting attention maps are scattered around the observation.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.12: Training curves for several models trained with different entropy weight



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.13: Percentage of levels solved in train and test as a function of entropy's weight

Table 4.5: Summary results for entropy regularization experiment

Environment	Entropy Weight	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	0.00	97.50%	82.35%	15.15%
	0.01 (Baseline)	98.50%	85.65%	11.57%
	0.02	98.50%	88.17%	10.33%
	0.05	96.00%	86.16%	9.84%
	0.07	96.50%	86.36%	10.14%
	0.10	94.50%	83.97%	10.53%
Jumper	0.00	90.00%	48.33%	41.67%
	0.01 (Baseline)	88.00%	49.73%	38.27%
	0.02	87.50%	50.33%	37.17%
	0.05	85.50%	52.53%	32.97%
	0.07	83.50%	53.13%	30.37%
	0.10	83.00%	50.93%	32.07%
Maze	0.00	57.00%	27.93%	29.07%
	0.01 (Baseline)	77.00%	32.30%	43.70%
	0.02	78.50%	33.07%	45.43%
	0.05	82.00%	39.40%	42.60%
	0.07	75.50%	34.80%	40.70%
	0.10	76.50%	38.60%	37.90%

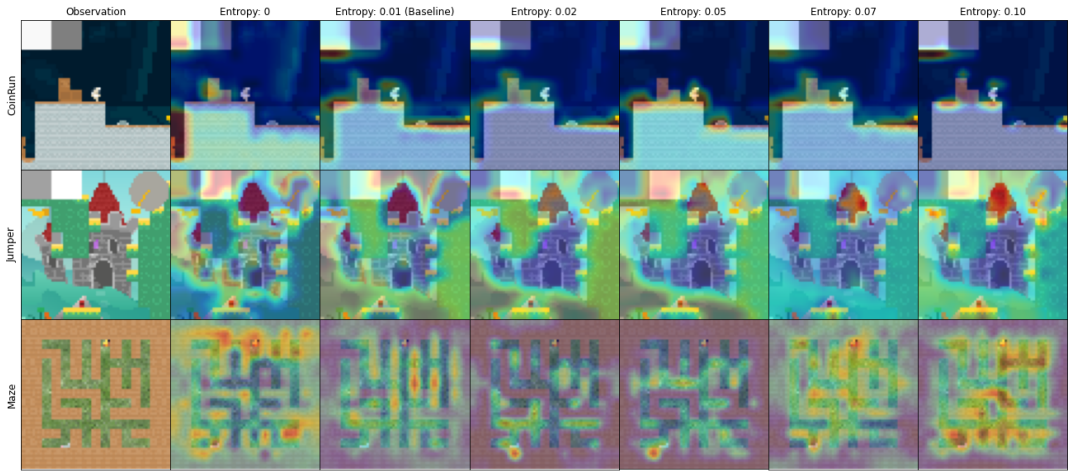
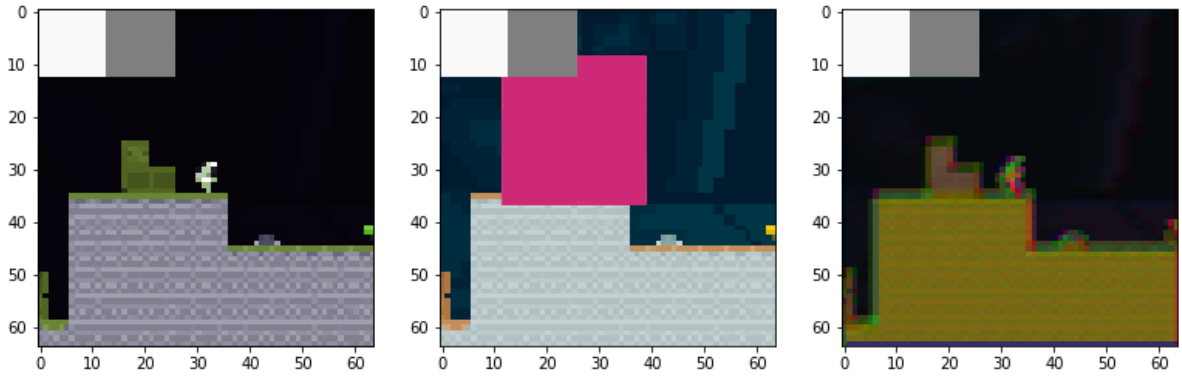


Figure 4.14: Attention Maps for several models trained with different entropy's bonus value

4.3 Evaluating Data Augmentation

In the third experiment we evaluate how data augmentation methods impact generalization. Data Augmentation is often effective in the supervised setting to reduce overfitting. We augment the input of our



(a) Color Jitter **(b)** Cutout **(c)** Network Randomization

Figure 4.15: Example of all data augmentation methods applied to an observation from CoinRun

baseline model with a Data Augmentation Layer that augments the input screen. The augmentation is applied sample-wise within a minibatch both during training and testing. We evaluate the following data augmentation methods:

Color Jitter: This augmentation layer changes randomly the value of the input's color channel.

Cutout: This augmentation layer, based on the method by [33], cuts out a rectangular area of the screen and fills it with a random color. If the game contains areas of the screen with useful information for learning such as the velocity information rectangle in CoinRun or the compass in Jumper, those

areas are masked and are never cut out.

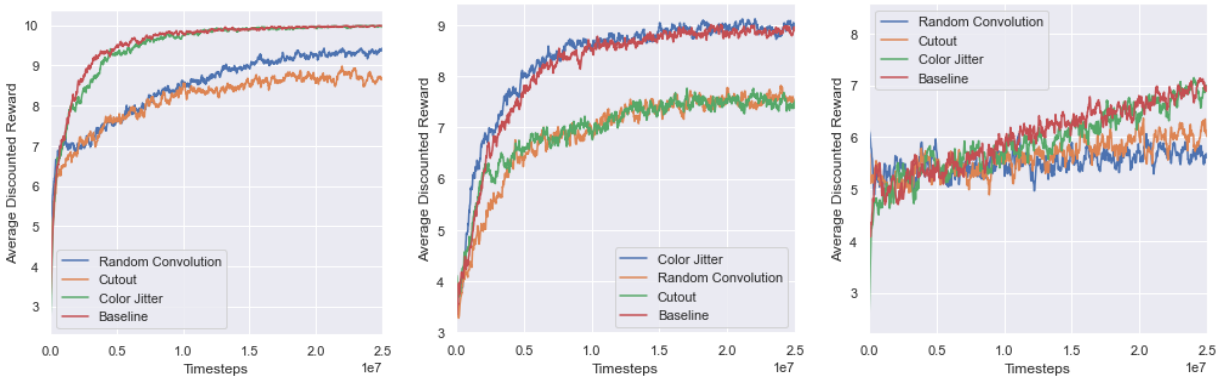
Network Randomization: This augmentation layer, introduced by [34], augments the baseline model with a random convolutional layer that perturbs the input image. This layer has 3 output channels, a 3x3 kernel with stride 1 and padding 1. The weights are not learnable and they get reset at each forward pass sample-wise using Xavier Normal Distribution [35]. During evaluation we use a Monte-Carlo approximation that stabilizes performance by reducing variance. This method generates M random inputs for each observation which are than aggregated as follows:

$$\pi(a|s;\theta) \approx \frac{1}{M}\sum_{m=1}^{M}\pi(a|f(s;\phi_m);\theta), \tag{4.1}$$

where f is a convolutional layer whose parameters ϕ are sampled from $\mathcal{N}(0, 1/18)$ We use $M = 10$ as suggested by a previous study [36] for the Procgen Benchmark.

Figure 4.15 shows a CoinRun observation augmented with each one of the data augmentation methods.

Results The results, shown in table 4.6, show that the effectiveness of data augmentation varies between environments. Models trained using data augmentation in CoinRun seem to perform worse compared to the baseline. In Jumper and Maze, every data augmentation method performed better than the baseline, with Cutout being the most effective. Figure 4.16 shows that Network Randomization and Cutout in comparison with Color Jitter and the baseline introduce more noise in the training process thus reducing sample efficiency. Figure 4.17 shows similar attention patterns, where the agent focus its attention in important game elements.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.16: Training curves for models trained using data augmentation

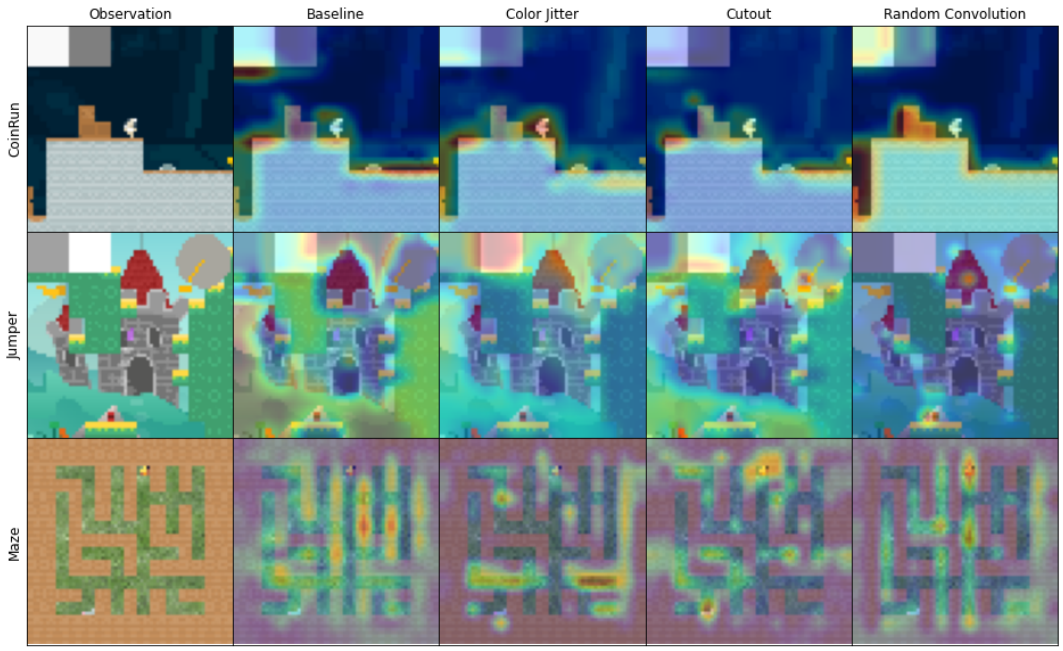


Figure 4.17: Attention Maps for several models trained with different augmentation methods

4.4 Evaluating Neural Network Architectures

In our fourth experiment, we assess how different neural networks architectures affect generalization. Each experiment stated below is independent and the modifications are not combined unless stated otherwise. The first batch of experiments are a direct modification of our baseline model's convolutional encoder and are as follows:

Max Pooling: A Max Pooling Layer is added after each Convolutional Layer. The Max Pooling layer has the same kernel size and padding as the previous convolutional layer. The name for this architecture is NatureCNN-MaxPool.

Convolutional Layer Hyperparameters: The baseline model's convolutional channels, kernel size, strading and padding are modified. The first convolutional layer has 16 kernels of size 4x4 with stride 2 and 0 paddings. The second layer has 32 kernels of size 3x3 with stride 2 and 0 paddings. The third and final layer has 32 kernels of size 3x3 with stride 2 and 1 padding. A Max Pooling layer is added after every convolutional layer due to the increase of the last convolutional layer output size. This architecture is named NatureCNN-HyperP. We want to assess the impact in generalization of using smaller-filters in early layers in order to capture more local information.

Depth: The depth of the network is increased by adding 1 or 2 convolutional layers with 128 channels, 2x2 kernel sizes, 1 stride and 0 padding. The name for this architecture with 1 and 2 additional convolutional layers is, respectively, NatureCNN-IncreasedDepth-A and NatureCNN-IncreasedDepth-

Table 4.6: Summary results for data augmentation experiments

Environment	Augmentation	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	None (Baseline)	98.50%	86.93%	11.57%
	Color Jitter	100.00%	86.00%	14.00%
	Cutout	87.50%	82.00%	5.50%
	Network Rand	91.50%	77.40%	14.10%
Jumper	None (Baseline)	88.00%	49.73%	38.27%
	Color Jitter	89.50%	54.47%	35.03%
	Cutout	76.50%	59.67%	16.83%
	Network Rand	75.00%	55.47%	19.53%
Maze	None (Baseline)	77.00%	33.30%	43.70%
	Color Jitter	77.00%	34.07%	42.93%
	Cutout	61.50%	53.80%	7.70%
	Network Rand	56.00%	51.20%	4.80%

B. We want to assess if by increasing depth, the network produces a more general agent.

Number of channels: The number of channels in each convolutional layer is doubled or halved. The name for this architecture with half and double channels is, respectively, NatureCNN-HalfChannels and NatureCNN-DoubleChannels. We want to assess if there is a relation between the number of channels and generalization.

Skip Connections: The baseline model is augmented with residual connections. [37] reported that skip-connections are more effective if skipping 2 convolutional layers. As the baseline architecture only has 3 convolutional layers, we double each convolutional layer and short-cut the odd layers. The identity function is used as short-cut if the input's and output's shape are the same, otherwise a 1x1 convolutional layer is applied in order to match both dimensions. This architecture is named NatureCNN-ResNet. We want to assess if skip-connections, by combining high-level and low-level features, improve generalization.

The second batch of experiments are modifications to the fully-connected layer of our baseline model. We want to assess if the number of neurons and depth of the fully-connected layer, which acts over the latent vector produced by the convolutional encoder, has impact in generalization.

Variation of the number of neurons: The number of neurons in the fully-connected layer is decreased to 256 or increased to 1024. The architecture with decreased number of neurons is named NatureCNN-FC-A and the one with increased number of neurons is named NatureCNN-FC-B.

Multilayer Perceptron The fully connected layer is replaced by a deeper feedforward network. The

first architecture has a MLP with 2 hidden layers. The first layer has 512 neurons and the second 256. This architecture is named NatureCNN-MLP-A. The second architecture has a MLP with 3 hidden layers. The first layer has 512 neurons, the second 256 and the third 128. This architecture is named NatureCNN-MLP-B.

Finally, we evaluate how a larger residual model such as IMPALA [32] performs compared to our baseline. IMPALA is a much deeper network compared to NatureCNN, with 5x more convolutional layers and with skip-connections. Figure 4.18 shows IMPALA's architecture.

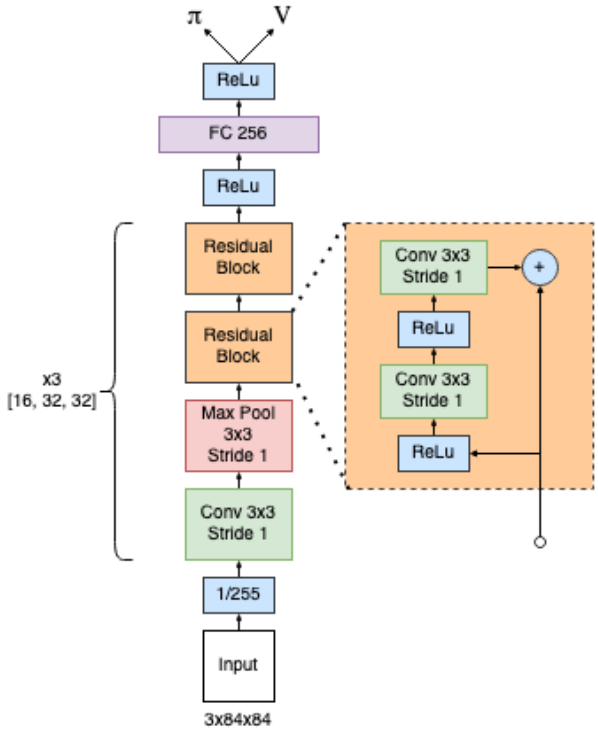
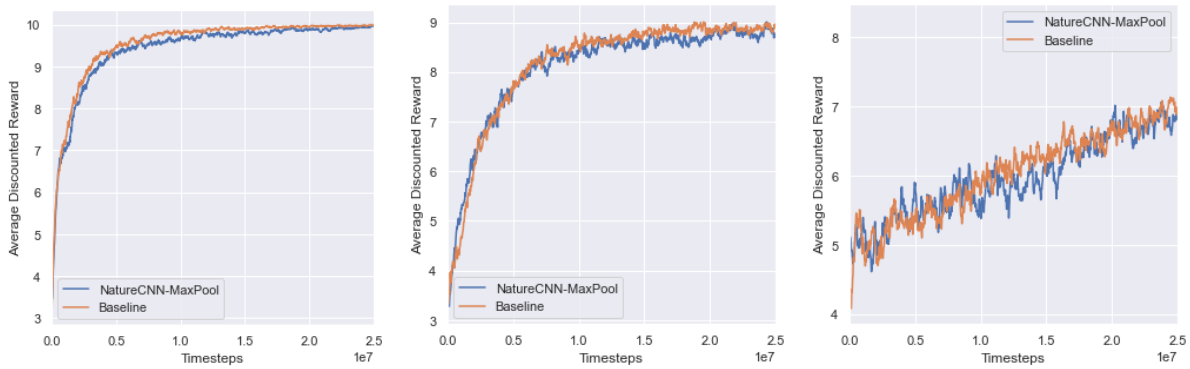


Figure 4.18: IMPALA Architecture

Results Results for all architectural experiments are shown in table 4.7. Figure 4.26, 4.27 and 4.28 provides the attention maps produced by GradCAM for the various modified models. The use of Max Pooling layers (NatureCNN-MaxPool) seems to slightly improve generalization across all environments while having no effect in training (figure 4.19) The attention maps in 4.26, show that Max Pooling reduces the magnitude of attention in CoinRun and Jumper, while scattering attention in small separated chunks in Maze.

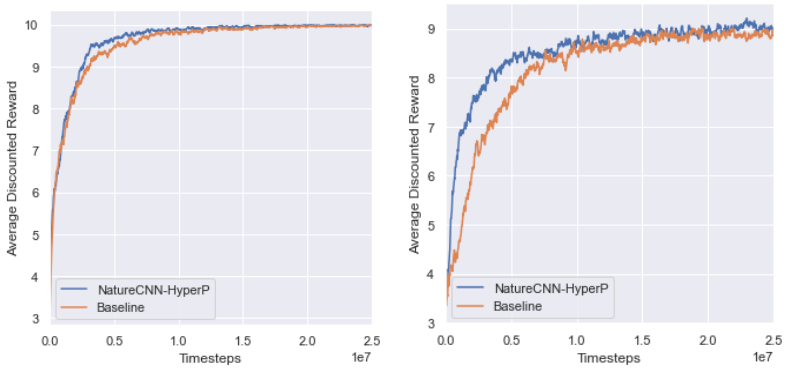
Decreasing the number of channel and filters' size (NatureCNN-HyperP), results in a major improvement in performance in all environments. The resulting attention map is presented in figure 4.26 and shows the attention patterns concentrated around important game elements. Figure 4.20 shows a small improvement in training sample efficiency in CoinRun and Jumper. In Maze, the use of smaller convolu-



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.19: Training curves for models trained using max pooling layers

tional filters seems to unlock a better sample efficiency during training that results in a much higher final discounted average reward.

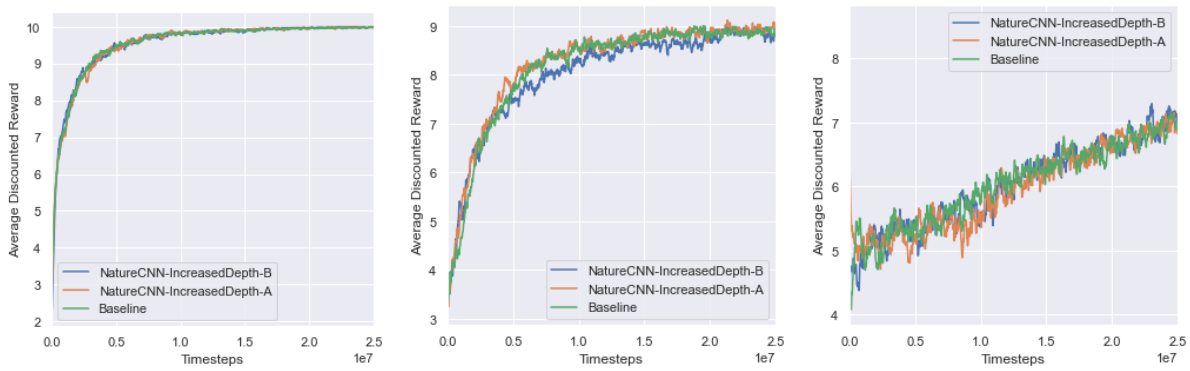


(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.20: Training curves for models trained using smaller convolutional filters

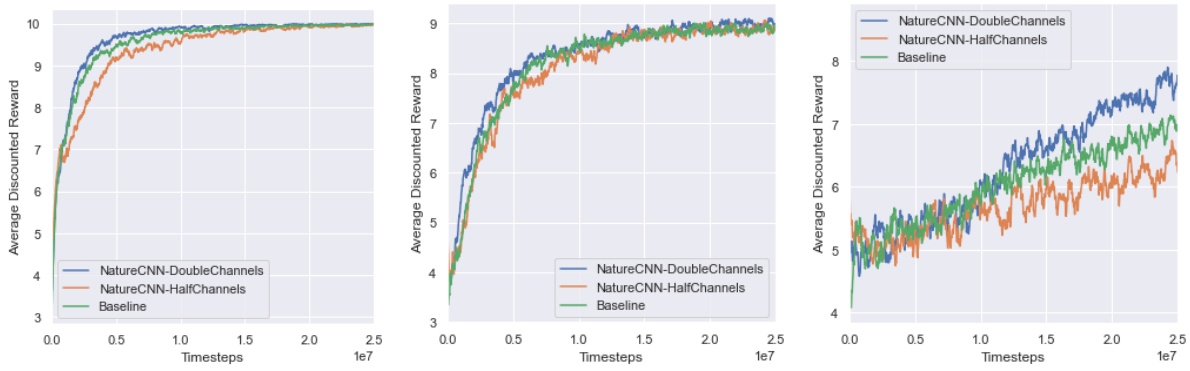
Varying the number of channels (NatureCNN-HalfChannels, NatureCNN-DoubleChannels) or increasing the depth of the network (NatureCNN-IncreasedDepth) results in no improvement in performance during testing. Increasing the number of channels improves learning in Maze while maintaining the same learning performance in CoinRun and Jumper (figure 4.22. Increasing the depth has no effect to the training process (figure 4.21). The attention maps in figure 4.26 for this experiment show similar attention patterns within the same environment.

Modifications to the fully-connected layer (NatureCNN-FC, NatureCNN-MLP) also result in no improvement in testing performance, suggesting that there is no source of bottleneck in the fully-connected layer. Figure 4.27 shows similar attention patterns across all environments and figures 4.23 4.24 show



(a) CoinRun **(b)** Jumper **(c)** Maze

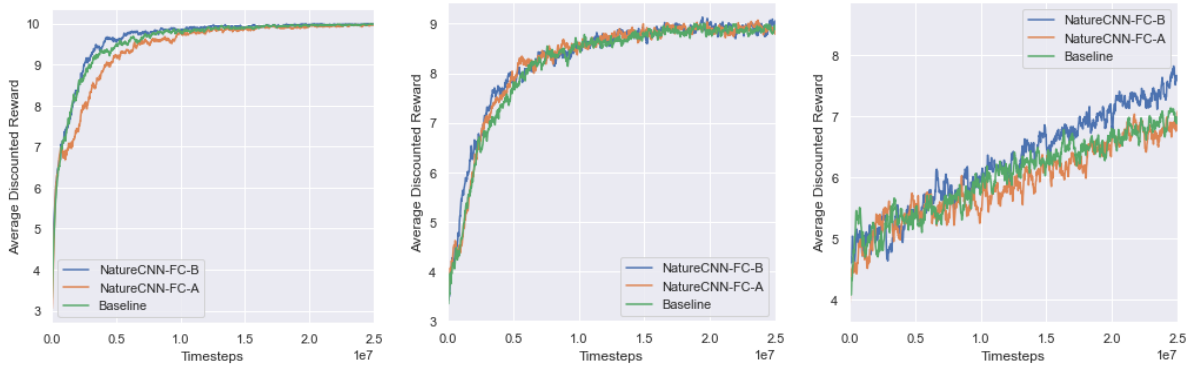
Figure 4.21: Training curves for models trained using more convolutional layers



(a) CoinRun **(b)** Jumper **(c)** Maze

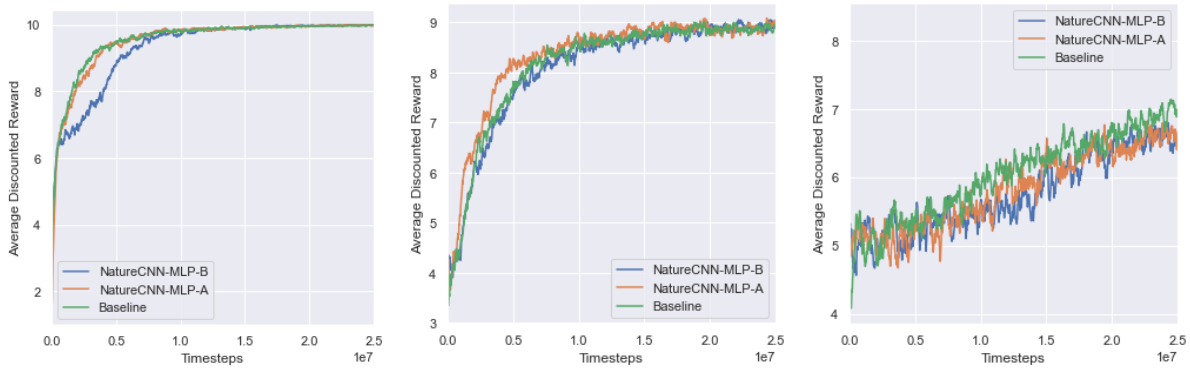
Figure 4.22: Training curves for models trained using different numbers of convolutional channels

similar training performances across all environments.



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.23: Training curves for models trained using different number of neurons in the fully-connected layer



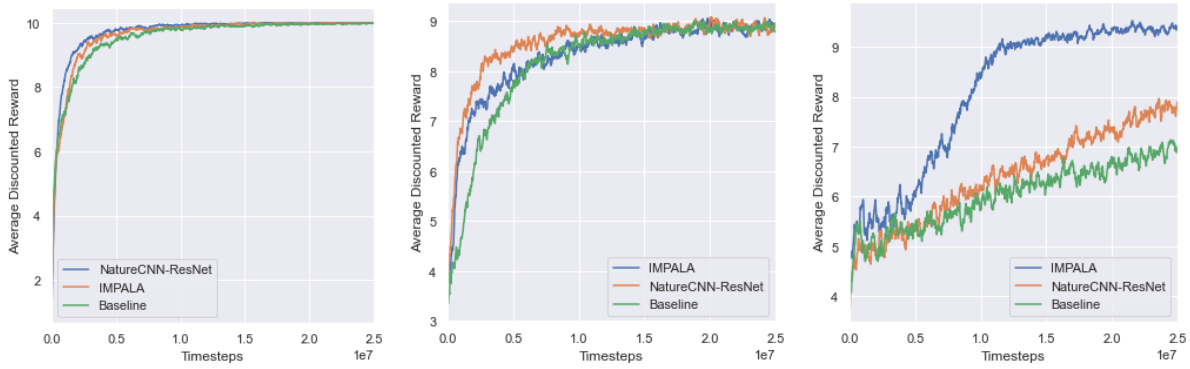
(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.24: Training curves for models trained using a MLP instead of just a single fully-connected layer

Using skipped-connections (NatureCNN-ResNet) results in a small performance improvement during testing in Jumper and CoinRun, but this can be due to the increase in the number of convolutional layers. The IMPALA model surpasses most of all other models, only having a similar performance with NatureCNN-HyperP in Jumper and Maze. During training (figure 4.25), NatureCNN-ResNet shows a better sample efficiency in CoinRun and Jumper but is completely surpassed both in sample efficiency and final discounted average reward in Maze. The attention maps for these models are shown in figure 4.28, where the attention is heavily concentrated in deciding game elements.

Table 4.7: Summary results for architecture experiments

Environment	Architecture	% Levels Solved - Train	% Levels Solved - Test	GG
CoinRun	NatureCNN (Baseline)	98.50%	86.93%	11.57%
	NatureCNN-MaxPool	99.00%	87.10%	11.90%
	NatureCNN-HyperP	100.00%	88.80%	11.20%
	NatureCNN-IncreasedDepth-A	99.50%	86.35%	13.15%
	NatureCNN-IncreasedDepth-B	100.00%	85.49%	14.51%
	NatureCNN-HalfChannels	97.50%	83.40%	14.10%
	NatureCNN-DoubleChannels	100.00%	87.19%	12.81%
	NatureCNN-ResNet	99.50%	88.73%	10.77%
	NatureCNN-FC-A	99.50%	86.10%	13.40%
	NatureCNN-FC-B	100.00%	85.10%	14.90%
	NatureCNN-MLP-A	99.00%	85.45%	13.55%
	NatureCNN-MLP-B	98.50%	87.23%	11.27%
	IMPALA	100.00%	93.53%	6.47%
Jumper	NatureCNN (Baseline)	88.00%	49.73%	38.27%
	NatureCNN-MaxPool	87.50%	52.53%	34.97%
	NatureCNN-HyperP	88.50%	56.60%	31.90%
	NatureCNN-IncreasedDepth-A	89.00%	51.52%	37.48%
	NatureCNN-IncreasedDepth-B	88.50%	51.00%	37.50%
	NatureCNN-HalfChannels	89.00%	54.20%	34.80%
	NatureCNN-DoubleChannels	89.00%	52.20%	36.80%
	NatureCNN-ResNet	88.00%	53.80%	34.20%
	NatureCNN-FC-A	89.00%	52.27%	36.73%
	NatureCNN-FC-B	88.50%	51.53%	36.97%
	NatureCNN-MLP-A	88.50%	51.73%	36.77%
	NatureCNN-MLP-B	88.50%	50.47%	38.03%
	IMPALA	88.50%	56.60%	31.90%
Maze	NatureCNN (Baseline)	77.00%	33.30%	43.70%
	NatureCNN-MaxPool	76.00%	36.13%	40.00%
	NatureCNN-HyperP	91.00%	51.33%	39.67%
	NatureCNN-IncreasedDepth-A	78.50%	32.73%	45.77%
	NatureCNN-IncreasedDepth-B	80.50%	28.87%	51.63%
	NatureCNN-HalfChannels	73.00%	35.00%	38.00%
	NatureCNN-DoubleChannels	81.50%	31.47%	50.03%
	NatureCNN-ResNet	82.00%	32.00%	50.00%
	NatureCNN-FC-A	74.50%	33.20%	41.30%
	NatureCNN-FC-B	81.00%	32.47%	48.53%
	NatureCNN-MLP-A	74.50%	30.60%	43.90%
	NatureCNN-MLP-B	78.50%	31.00%	47.50%
	IMPALA	91.00%	51.30%	39.70%



(a) CoinRun **(b)** Jumper **(c)** Maze

Figure 4.25: Training curves for models trained using skip-connections

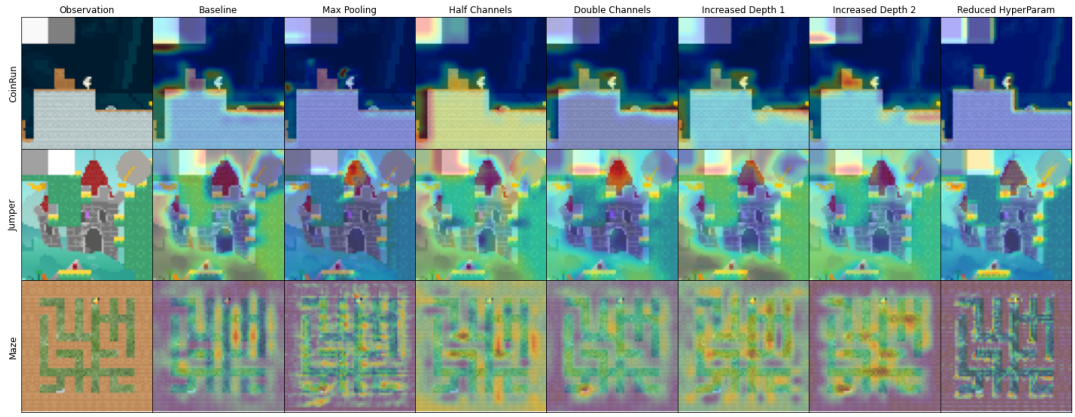


Figure 4.26: Attention Maps for architectures with modified convolutional encoder

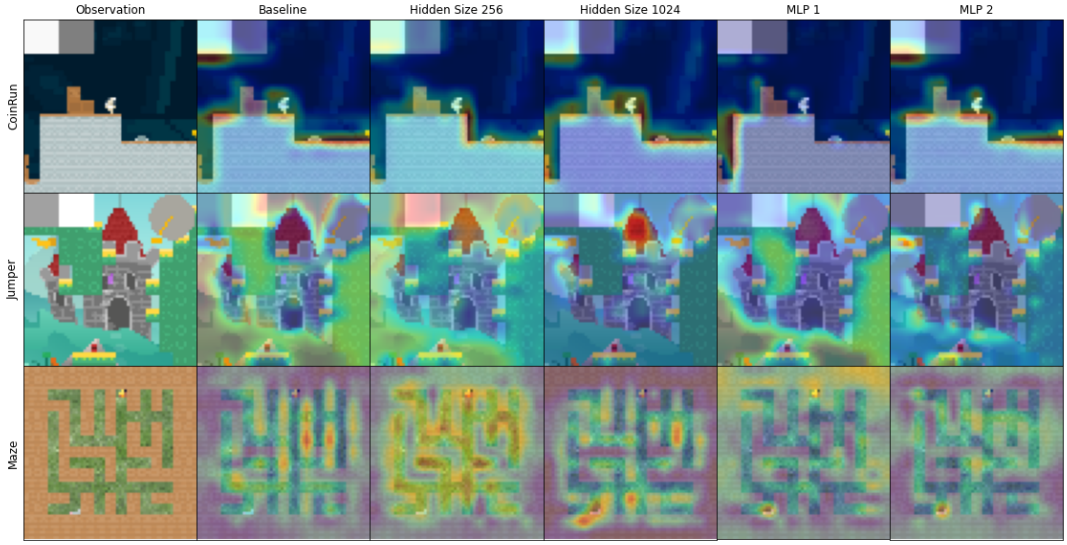


Figure 4.27: Attention Maps for architectures with modified fully-connected layer

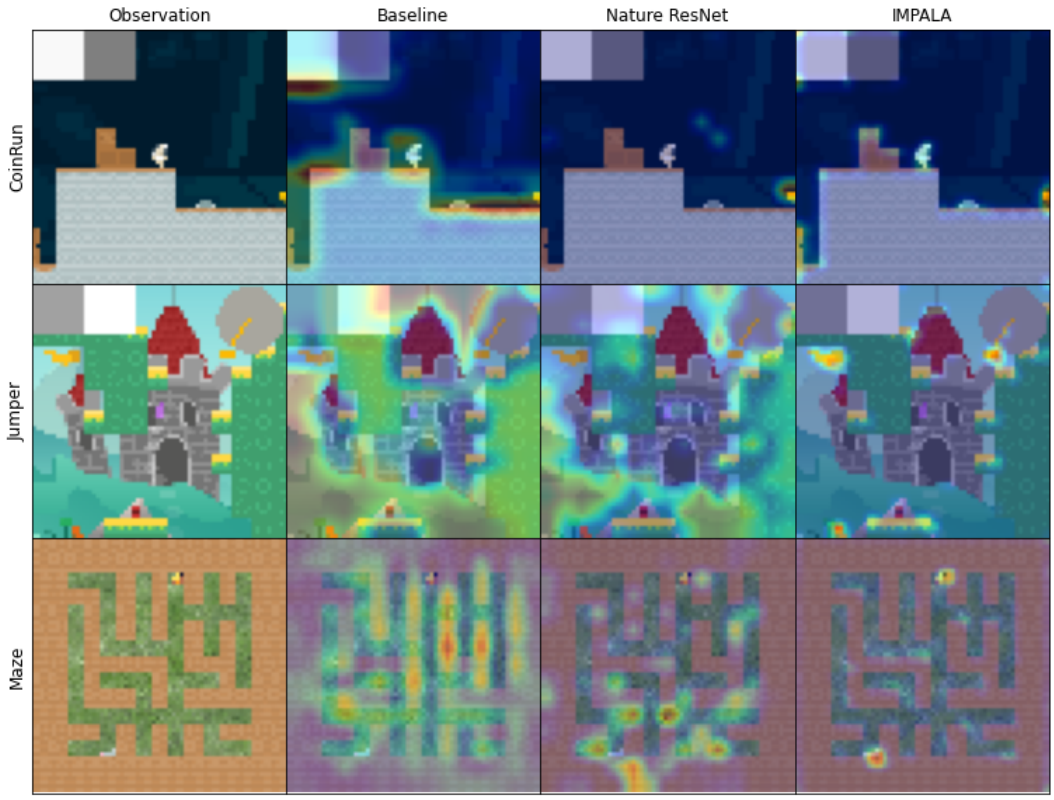


Figure 4.28: Attention Maps for architectures with residual layers

5

Conclusions and Future Work

Contents

In this work, we provided an empirical study on the effects and dependencies of overfitting in Deep Reinforcement Learning. This chapter presents a clear overview of the obtained results, and highlights possible directions for future research.

5.1 Overview on the Results

An overview of the results are as follows:

- In general, using regularization and data augmentation in reinforcement learning improves generalization. There is no golden rule for which method to use. Results showed that the effectiveness of each method varies between environments.

- Data variability seems to be key in Reinforcement Learning. Scaling the training set size enables the agent to learn in a more general way, which enables it to perform well in unseen levels.

- The different results between environments suggest that improving generalization by inducing stochasticity such as in the agent's decision (policy's entropy), network nodes (dropout) or the input (data augmentation) are deeply dependent on the environment dynamics.

- GradCAM attention maps confirms that the majority of the agents, when deciding what action to take, they pay attention to deciding game elements such as traps, objectives, agent's position, objective's direction (Jumper's compass), velocity information (CoinRun's velocity rectangle). Attention maps from Nature-ResNet and IMPALA suggest that deeper residual networks produce attention maps with smaller and isolated attention patterns that focus deciding game elements.

- The results involving architectural modifications suggest that the generalization bottleneck in NatureCNN is presented in the first layers of the convolutional encoder. Modifying, in isolation, the number of channels and the depth of the network seems to have a very small effect in performance and generalization. The number of neurons and depth of the fully-connected layer seems also not to have impact in generalization and training performance. There is no type of layer (BatchNorm, MaxPooling, Residual) that used in isolation causes a major improvement in both performance and generalization. Networks with max pooling and residual connections make it possible to train deeper models that produce more capable and general agents as it is suggested by the results using IMPALA. However by using smaller early convolutional filters (NatureCNN-HyperP) we can achieve similar performance to IMPALA in Jumper and Maze.

5.2 Future Work

We leave two possible directions for future work. First, it is standard in Deep Reinforcement Learning to train using one neural network with shared weights for two different objectives (policy and value function). Investigate to what extent there is some interference between objectives during training and if using two separate training phases [38] improves testing performance and generalization. Furthermore, assess if there is any advantage in using two separated neural network instead of a shared one. Second, investigate if using more advanced architectures and mechanisms such as Dense Networks [39] or Transformers (attention) [40] improves generalization.

Bibliography

[1] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model," nov 2019. [Online]. Available: http://arxiv.org/abs/1911.08265http://dx.doi.org/10.1038/s41586-020-03051-4

[2] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: An evolutionary computation perspective," *GECCO 2019 Companion - Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion*, pp. 314–315, 2019.

[3] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," dec 2019. [Online]. Available: http://arxiv.org/abs/1912.06680

[4] M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker," jan 2017. [Online]. Available: http://arxiv.org/abs/1701.01724http://dx.doi.org/10.1126/science.aam6960

[5] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building Machines That Learn and Think Like People," apr 2016. [Online]. Available: http://arxiv.org/abs/1604.00289

[6] Z. Ghahramani, D. M. Wolpert, and M. I. Jordan, "Generalization to local remappings of the visuomotor coordinate transformation." *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 16, no. 21, pp. 7085–96, nov 1996. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/8824344http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6579263

[7] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying Generalization in Reinforcement Learning," dec 2018. [Online]. Available: http://arxiv.org/abs/1812.02341

[8] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, “Gotta Learn Fast: A New Benchmark for Generalization in RL,” apr 2018. [Online]. Available: http://arxiv.org/abs/1804.03720

[9] A. Zhang, N. Ballas, and J. Pineau, “A Dissection of Overfitting and Generalization in Continuous Reinforcement Learning,” jun 2018. [Online]. Available: http://arxiv.org/abs/1806.07937

[10] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, “Solving Rubik’s Cube with a Robot Hand,” oct 2019. [Online]. Available: http://arxiv.org/abs/1910.07113

[11] J. Farebrother, M. C. Machado, and M. Bowling, “Generalization and Regularization in DQN,” sep 2018. [Online]. Available: http://arxiv.org/abs/1810.00123

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, oct 1986. [Online]. Available: http://www.nature.com/articles/323533a0

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[14] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, feb 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” jul 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[16] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” jun 2015. [Online]. Available: http://arxiv.org/abs/1506.02438

[17] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” dec 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning,” feb 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” dec 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[20] Z. Liu, X. Li, B. Kang, and T. Darrell, “Regularization Matters in Policy Optimization – An Empirical Study on Continuous Control,” oct 2019. [Online]. Available: http://arxiv.org/abs/1910.09191

[21] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, “Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation,” jun 2018. [Online]. Available: http://arxiv.org/abs/1806.10729

[22] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms,” feb 2018. [Online]. Available: http://arxiv.org/abs/1802.10363

[23] C. Packer, K. Gao, J. Kos, P. Krähenbühl, V. Koltun, and D. Song, “Assessing Generalization in Deep Reinforcement Learning,” oct 2018. [Online]. Available: http://arxiv.org/abs/1810.12282

[24] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, “A Study on Overfitting in Deep Reinforcement Learning,” apr 2018. [Online]. Available: http://arxiv.org/abs/1804.06893

[25] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents,” *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, mar 2018. [Online]. Available: https://jair.org/index.php/jair/article/view/11182

[26] M. Hausknecht and P. Stone, “The Impact of Determinism on Learning Atari 2600 Games,” in *AAAI Workshop on Learning for General Competency in Video Games*, jan 2015.

[27] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, “Leveraging Procedural Generation to Benchmark Reinforcement Learning,” dec 2019. [Online]. Available: http://arxiv.org/abs/1912.01588

[28] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The Arcade Learning Environment: An Evaluation Platform for General Agents,” jul 2012. [Online]. Available: http://arxiv.org/abs/1207.4708http://dx.doi.org/10.1613/jair.3912

[29] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization,” oct 2016. [Online]. Available: http://arxiv.org/abs/1610.02391http://dx.doi.org/10.1007/s11263-019-01228-7

[30] A. Stooke and P. Abbeel, “rlpyt: A Research Code Base for Deep Reinforcement Learning in PyTorch,” sep 2019. [Online]. Available: http://arxiv.org/abs/1909.01500

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy,

B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” dec 2019. [Online]. Available: http://arxiv.org/abs/1912.01703

[32] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, “IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures,” feb 2018. [Online]. Available: http://arxiv.org/abs/1802.01561

[33] T. DeVries and G. W. Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout,” aug 2017. [Online]. Available: http://arxiv.org/abs/1708.04552

[34] Z. Xu, D. Liu, J. Yang, C. Raffel, and M. Niethammer, “Robust and Generalizable Visual Representation Learning via Random Convolutions,” jul 2020. [Online]. Available: http://arxiv.org/abs/2007.13003

[35] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[36] K. Lee, K. Lee, J. Shin, and H. Lee, “Network Randomization: A Simple Technique for Generalization in Deep Reinforcement Learning,” oct 2019. [Online]. Available: http://arxiv.org/abs/1910.05396

[37] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” dec 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[38] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman, “Phasic Policy Gradient,” sep 2020. [Online]. Available: http://arxiv.org/abs/2009.04416

[39] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” aug 2016. [Online]. Available: http://arxiv.org/abs/1608.06993

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” jun 2017. [Online]. Available: http://arxiv.org/abs/1706.03762