Evaluating generalization in Deep Reinforcement Learning with Procedurally Generated Environments

Miguel Freire
Instituto Superior Técnico
University of Lisbon
Lisbon, Portugal
`miguel.freire@tecnico.ulisboa.pt`

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 options 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 for this project is publicly available on `https://github.com/MiguelFreire/rl-generalization`.

1 Introduction

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 [26] for board games such as Chess, Go, Shogi and Atari games, AlphaStar [1] for Starcraft II, OpenFive [21] for Dota 2 and DeepStack [19] 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. [14, 8].

1.1 Motivation

Classic Reinforcement Learning (RL) game benchmarks commonly promote training and testing in the same environment, but recent studies [3, 20, 32] 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 [22]. These challenges are also present in pixel-based reinforcement learning, where the agent learns from pixel-based observations, due to its high-dimensionality and partial-observability [7]. Bridging the gap between generalization and data-efficiency is pivotal not only to develop better and more robust agents for digital environments such as videogames but also for real-world applications such as robotics.

1.2 Proposal and Contributions

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.

The main contributions of this work are as follows:

- We extend a previous empirical study [3] 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 bottleneck in small neural-network architectures, while helping to bridge the gap between these and more robust models such as deeper Residual Networks.

2 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 [7, 32] identified that classic DRL algorithms such as DQN [18] 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 [16] 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 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 [20] 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 [13] is based on several video games from the General Video Game AI framework [25], 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 [33] 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 [17] and random starts [10], often fail to work in procedurally generated environments. The Procgen Benchmark [4], 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 [2] and provide enough diversity to measure generalization. A recent study [3] used CoinRun, an environment from the Procgen benchmark, to generate large training and test sets to better evaluate generalization and overtiffing. 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

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 exhibit low diversity between them (high correlation).

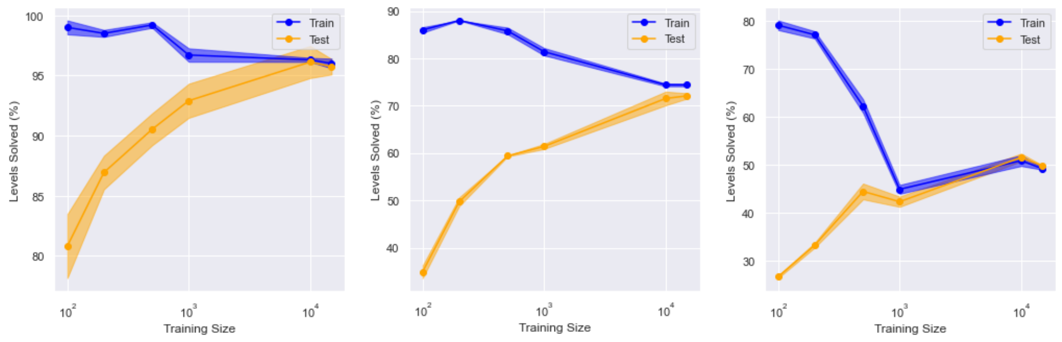
Convolutional Encoder: The convolutional encoder does not produce an invariant low-dimensional representation vector of the input state.

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

Policy stochasticity: Given 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 assess how each one of the sources presented above and their dependencies affect the generalization capability of the trained 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. In the third experiment we evaluate the impact of data augmentation in generalization and in the fourth and final experiment we evaluate how architectural decisions effect generalization.

In order to evaluate generalization we need to separate training and testing into two sets of levels. We use three environments from the Procgen benchmark (CoinRun, Maze and Jumper) following the same experimental setup as done in a previous work [3]. All the agents are trained using Proximal Policy Optimization (PPO) [28] with the advantage function being approximated using Generalized Advantage Estimation (GAE) [27]. We use the Average Discounted Return to track agent performance during training. We evaluate the percentage of levels solved both in the train and test phases to calculate the Generalization Gap (GG) [4] in order to measure overfitting. A description of each environment is present in Appendix A.1. The experimental setup and chosen hyper-parameters are presented in Appendix A.2.



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

Figure 1: Generalization Gap for several models trained with different set sizes

4 Results and Discussion

4.1 Evaluating Training Set Size

In this section we assess how the 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).

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 representation 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. Previous authors [4] reported an unusual trend in this same experiment: past a certain threshold, training performance improves as the training set increases. A larger training set can improve training performance if the agents learn, how to generalize across, abroad distribution of levels. Here we did not find this behavior. The only difference between the experiments is the architecture used. The previous authors used the IMPALA architecture [6], while we used a smaller one. Previous authors claimed that larger architectures largely 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.

For the remaining experiments, we set the training levels to 200 given that the results show a noticeable generalization gap with this set size across all environments. We present in appendix A.4, the learning curves for all experiments.

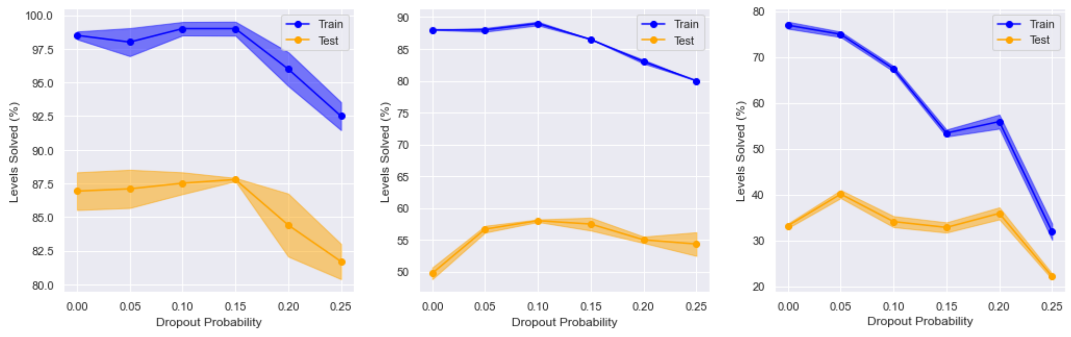
4.2 Evaluating Regularization Methods

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

4.2.1 Batch Normalization

In this sub-section, we assess the usage of Batch Normalization [12], a method that stabilizes layers input distributions by normalizing them within the training batch. Batch Normalization is a method that stabilizes learning by allowing higher learning rates but it also has a regularizing effect.

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.



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

Figure 2: Percentage of levels solved in train and test as a function of Dropout's probability p

Results The usage of Batch Normalization results in a higher percentage of levels solved in the training set in all three environments. In Jumper, Batch Normalization shows a regularizing effect by increasing the 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 and larger mini-batch sizes.

4.2.2 Dropout

In this section we evaluate the impact of Dropout [29] in training and testing. Dropout 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 networks and test on the complete larger network with smaller weights. 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 We report in Figure 2 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 a higher variance in training, which leads to the need for more timesteps to converge to the same average discounted reward as the baseline agent.

4.2.3 L2 Regularization

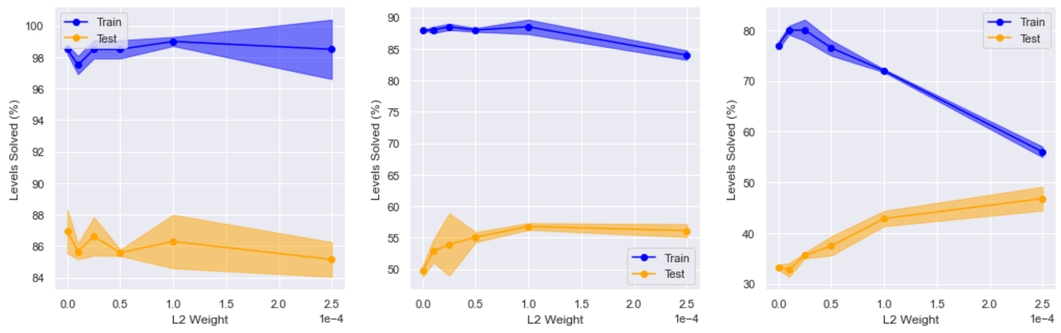
In this section we evaluate how L2 regularization impacts training and testing. We add a L2 penalizing term $w\|\theta\|_2^2$ to PPO's loss function, where θ are the model's parameters and w a parameter that controls the weight of the regularization. We vary the parameter w in the range $[0, 2.5 * 10^{-4}]$.

Results Results are presented in figure 3. 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.

4.2.4 Entropy Regularization

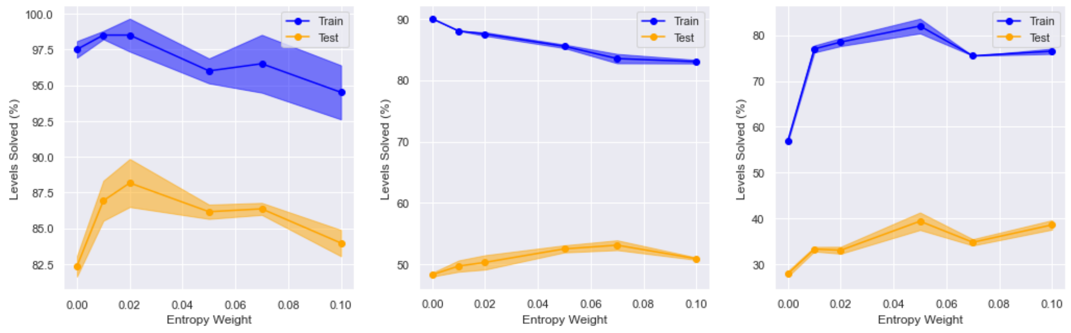
In this section we evaluate the impact of varying the weight k_w that controls the level of the policy's stochasticity. We vary the weight k_w between $[0, 0.1]$. Note that the baseline model has an entropy weight of 0.01.

Results The results for this experiment are presented in figure 4 showing that an increase in k_w improves generalization. CoinRun and Jumper success rate in training suggest that exploration is not a problem in this type of environments. On the other hand, the Maze's result show that higher values of k_w result in higher success rates both in training as in testing.



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

Figure 3: Percentage of levels solved in train and test as a function of L2's weight



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

Figure 4: Percentage of levels solved in train and test as a function of entropy's weight

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 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:

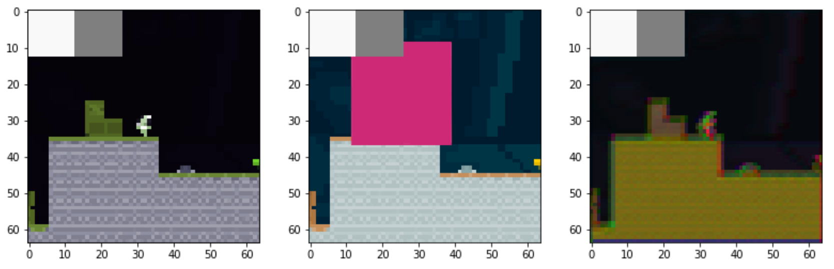
Color Jitter: This augmentation layer changes randomly the value of the input's color channel.

Cutout: This augmentation layer [5] 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 [31] 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 [9]. 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:

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

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 [15] for the Procgen Benchmark.



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

Figure 5: Example of all data augmentation methods applied to an observation from CoinRun

Table 1: Results for evaluating generalization with different data augmentation methods

	CoinRun			Jumper			Maze		
Augmentation	Train (%)	Test (%)	GG	Train (%)	Test (%)	GG	Train (%)	Test (%)	GG
None (Baseline)	98.50	86.93	11.57	88.00	49.73	38.27	77.00	33.30	43.70
Color Jitter	100.00	86.00	14.00	90.00	54.47	35.53	77.00	34.07	42.93
Cutout	88.00	82.00	6.00	77.00	59.67	17.00	61.50	53.80	7.70
Network Rand	91.50	77.40	14.10	75.50	55.47	20.03	56.00	51.20	4.80

Results The results, shown in table 1, 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.

4.4 Evaluating Neural Network Architectures

In our fourth experiment, we assess how different neural networks architectures affect generalization. Each experiment stated below is an independent experiment 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 N-CNN-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 N-CNN-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, N-CNN-Depth-A and N-CNN-Depth-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, N-CNN-HChannels and N-CNN-DChannels. 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 [11]. Skip connections are more effective if skipping 2 convolutional layers as reported by the methods'

Table 2: Results for evaluating generalization with different convolutional neural network architectures

	CoinRun			Jumper			Maze		
Architecture	Train (%)	Test (%)	GG	Train (%)	Test (%)	GG	Train (%)	Test (%)	GG
N-CNN (Baseline)	98.50	86.93	11.57	88.00	49.73	38.27	77.00	33.30	43.70
N-CNN MaxPool	99.00	87.10	11.90	87.50	52.53	34.97	76.00	36.13	39.87
N-CNN-HyperP	100.00	88.80	11.20	88.50	56.60	31.90	91.00	51.33	39.67
N-CNN-Depth-A	99.50	86.35	13.15	89.00	51.52	37.48	78.50	32.73	45.77
N-CNN-Depth-B	100.00	85.49	14.51	88.50	51.00	37.50	80.50	28.87	51.63
N-CNN-HChannels	97.50	83.40	14.10	89.00	54.20	34.80	73.00	35.00	38.00
N-CNN-DChannels	100.00	87.19	12.81	89.00	52.20	36.80	81.50	31.47	50.03
N-CNN-ResNet	99.50	88.73	10.77	88.00	53.80	34.20	82.00	32.00	50.00
N-CNN-FC-A	99.50	86.10	13.40	89.00	52.27	36.73	74.50	33.20	41.30
N-CNN-FC-B	100.00	85.10	14.90	88.50	51.53	36.97	81.00	32.47	48.53
N-CNN-MLP-A	99.00	85.45	13.55	88.50	50.47	38.03	74.50	30.60	43.90
N-CNN-MLP-B	98.50	87.32	11.27	88.50	50.47	38.03	78.50	31.00	47.50
N-CNN-IMPALA	100.00	93.53	6.47	88.50	56.60	31.90	91.00	51.30	39.70

original authors. 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 N-CNN-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 N-CNN-FC-A and the one with increased number of neurons is named N-CNN-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 N-CNN-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 N-CNN-MLP-B.

Finally, we evaluate how a larger residual model such as IMPALA [6] performs compared to our baseline. IMPALA is a much deeper network compared to NatureCNN, with 5x more convolutional layers and with skip-connections.

Results Results for all architectural experiments are shown in table 2. The use of Max Pooling layers (N-CNN-MaxPool) slightly improves generalization across all environments while having small to no effect in training.

Decreasing the number of channel and filters' size (N-CNN-HyperP), results in a major improvement in performance in all environments. In Maze, the use of smaller convolutional filters seems to lead to better sample efficiency during training that results in a much higher final discounted average reward and testing performance.

Varying the number of channels (N-CNN-HChannels, N-CNN-DChannels), increasing the depth of the network (N-CNN-Depth) or modifying the fully-connected layer (N-CNN-FC, N-CNN-MLP) results in no improvement in performance during testing and a small increase in the generalization gap. These results suggest that neither the final convolutional layers nor the fully-connected layer(s) impose a significant generalization bottleneck.

Using skip-connections (N-CNN-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 N-CNN-HyperP in Jumper and Maze.

Both IMPALA and N-CNN-HyperP results suggest that the generalization bottleneck is a result from the early convolutional layers. The 8x8 kernel in the first convolutional layer of the baseline model may be responsible for compressing too much the input resulting in information loss.

5 Conclusion

In this work, we provided an empirical study on the effects and dependencies of overfitting in Deep Reinforcement Learning. The main results were:

- In general, using regularization and data augmentation in reinforcement learning improves generalization. There is no rule of thumb for which method to use. Results showed that the effectiveness of each method varies between environments.
- 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.
- 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 results involving architectural modifications suggest that the generalization bottleneck in NatureCNN is present in the first layers of the convolutional encoder. Modifying, in isolation, the number of channels and the depth of the network has 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 (N-CNN-HyperP) we can achieve similar performance to IMPALA in Jumper and Maze.

References

[1] 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*, pages 314–315, 2019. doi: 10.1145/3319619.3321894.

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

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

[4] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning. dec 2019. URL `http://arxiv.org/abs/1912.01588`.

[5] T. DeVries and G. W. Taylor. Improved Regularization of Convolutional Neural Networks with Cutout. aug 2017. URL `http://arxiv.org/abs/1708.04552`.

[6] 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. URL `http://arxiv.org/abs/1802.01561`.

[7] J. Farebrother, M. C. Machado, and M. Bowling. Generalization and Regularization in DQN. sep 2018. URL http://arxiv.org/abs/1810.00123.

[8] 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*, 16(21):7085–96, nov 1996. ISSN 0270-6474. URL http://www.ncbi.nlm.nih.gov/pubmed/8824344http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6579263.

[9] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

[10] 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.

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

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

[13] 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. URL http://arxiv.org/abs/1806.10729.

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

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

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

[17] 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*, 61:523–562, mar 2018. ISSN 1076-9757. doi: 10.1613/jair.5699. URL https://jair.org/index.php/jair/article/view/11182.

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

[19] 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. doi: 10.1126/science.aam6960. URL http://arxiv.org/abs/1701.01724http://dx.doi.org/10.1126/science.aam6960.

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

[21] 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. URL http://arxiv.org/abs/1912.06680.

[22] 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. URL `http://arxiv.org/abs/1910.07113`.

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

[24] 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. URL `http://arxiv.org/abs/1912.01703`.

[25] 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. URL `http://arxiv.org/abs/1802.10363`.

[26] 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. doi: 10.1038/s41586-020-03051-4. URL `http://arxiv.org/abs/1911.08265http://dx.doi.org/10.1038/s41586-020-03051-4`.

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

[28] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. jul 2017. URL `http://arxiv.org/abs/1707.06347`.

[29] 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*, 15 (56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

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

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

[32] A. Zhang, N. Ballas, and J. Pineau. A Dissection of Overfitting and Generalization in Continuous Reinforcement Learning. jun 2018. URL `http://arxiv.org/abs/1806.07937`.

[33] C. Zhang, O. Vinyals, R. Munos, and S. Bengio. A Study on Overfitting in Deep Reinforcement Learning. apr 2018. URL `http://arxiv.org/abs/1804.06893`.

A Appendix

A.1 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 [4], a set of 16 procedurally 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 procedurally 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 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. Procedurally 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. Procedurally 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. Procedurally 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.

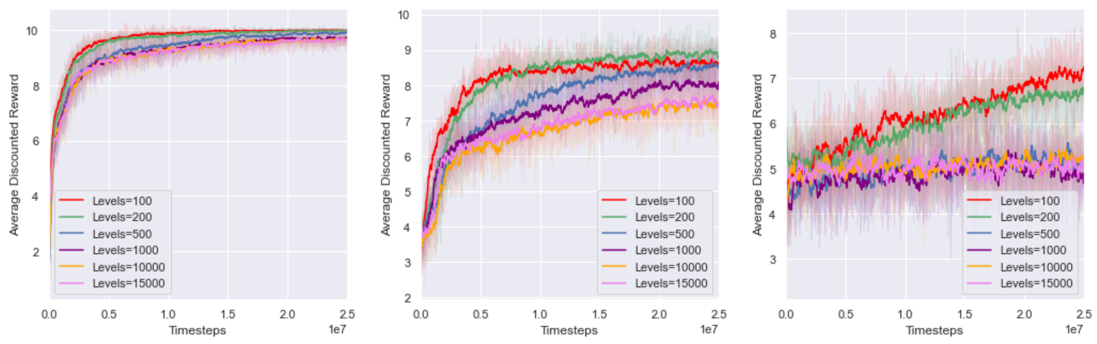
We chose these three environments based on the following requirements:

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.

A.2 Experimental Setup

Training Following the same experimental setup as done in a previous work[3], all the agents are trained using Proximal Policy Optimization (PPO) [28] with the advantage function being approximated using Generalized Advantage Estimation (GAE) [27] 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 [3], which guarantees an increasing learning progress over time. All the hyperparameters values can be found in table 3. 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. The critic uses the Advantage Function as the value-function, estimated using GAE with λ set to 0.95. We use the Adam Optimizer to optimize the parameters of the model. Our baseline model is the NatureCNN [18], 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



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

Figure 6: Training curves for several models trained with different set sizes

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. 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, 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 [24] 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.

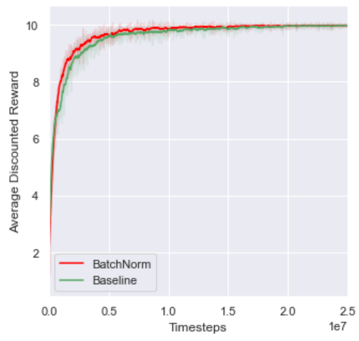
Table 3: Default hyperparameters' values for PPO Algorithm

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)

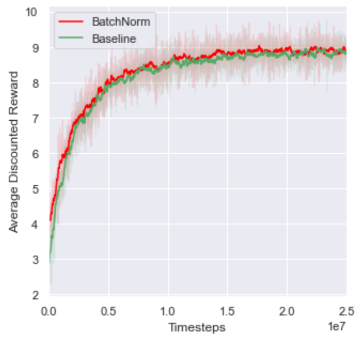
Testing During testing, we assess the zero-shot-performance from three different training seeds and average the results. The testing set has 500 different levels that were not seen during training. We consider that the agent completed the level if it could reach the goal without dying or before the episode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

A.3 Data Augmentation Methods

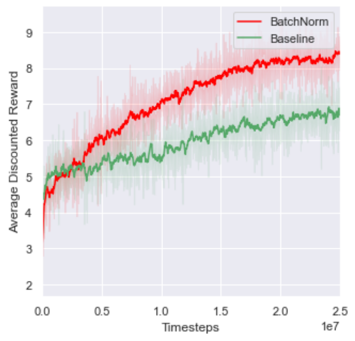
A.4 Learning Curves



(a) CoinRun

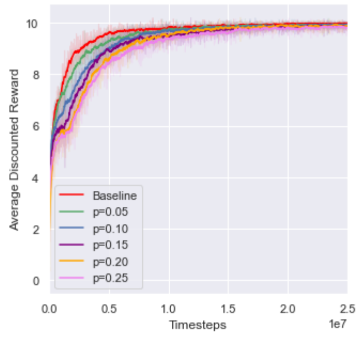


(b) Jumper

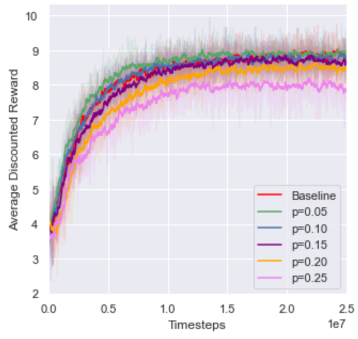


(c) Maze

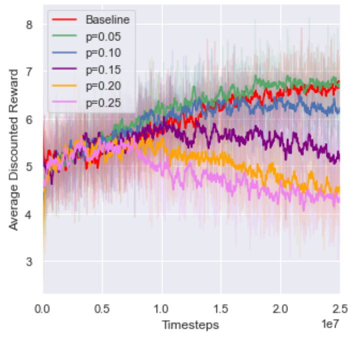
Figure 7: Training curves for several models trained with and without Batch Normalization



(a) CoinRun

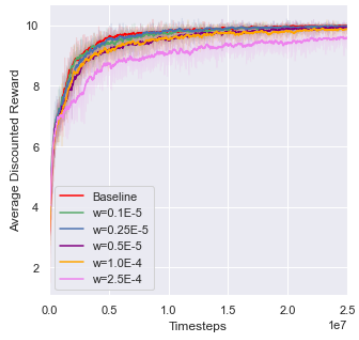


(b) Jumper

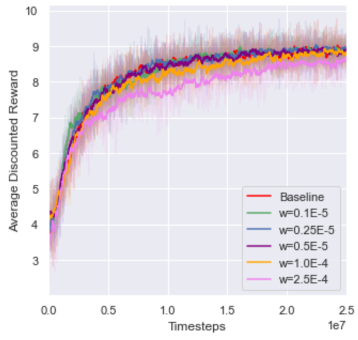


(c) Maze

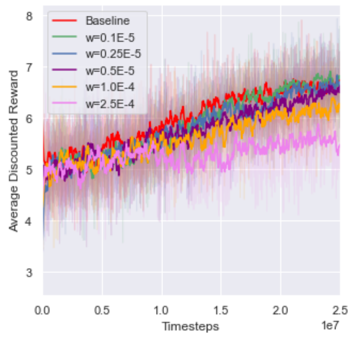
Figure 8: Training curves for several models with different Dropout's probability p



(a) CoinRun

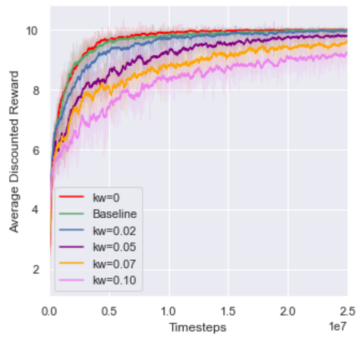


(b) Jumper

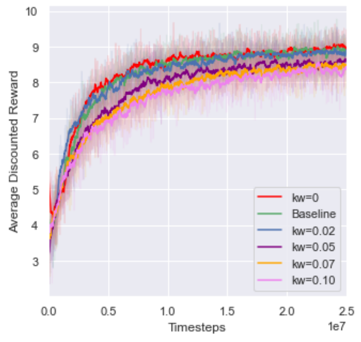


(c) Maze

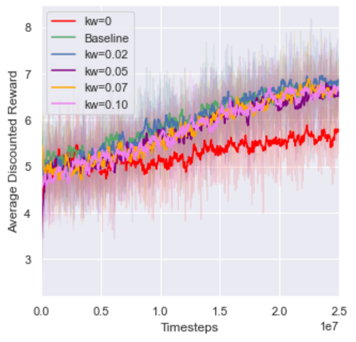
Figure 9: Training curves for several models trained with different L2's weight



(a) CoinRun

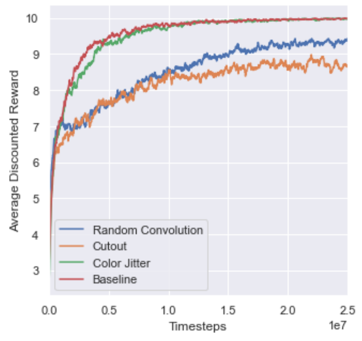


(b) Jumper

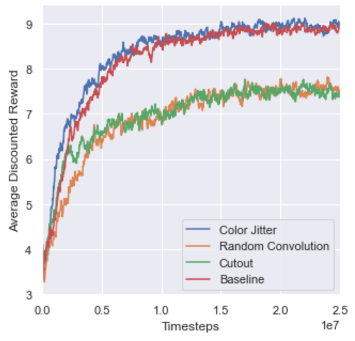


(c) Maze

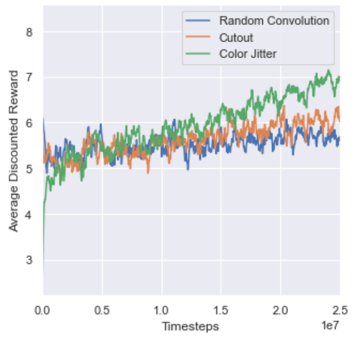
Figure 10: Training curves for several models trained with different entropy weight



(a) CoinRun

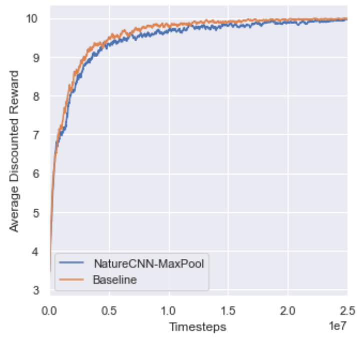


(b) Jumper

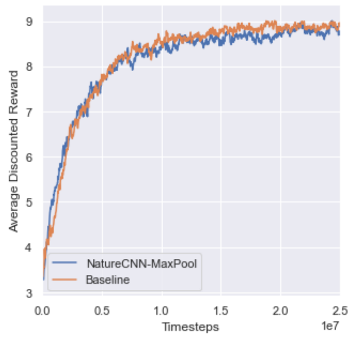


(c) Maze

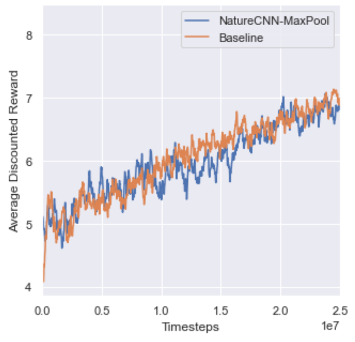
Figure 11: Training curves for models trained using data augmentation



(a) CoinRun



(b) Jumper

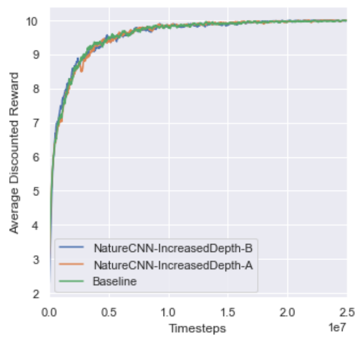


(c) Maze

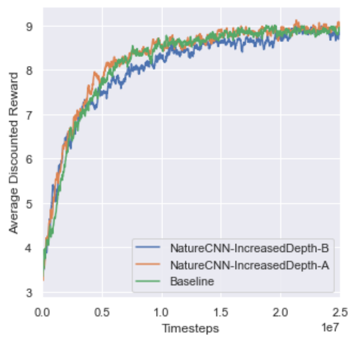
Figure 12: Training curves for models trained using max pooling layers

(a)(b)(c)
CoinJumperMaze
Run

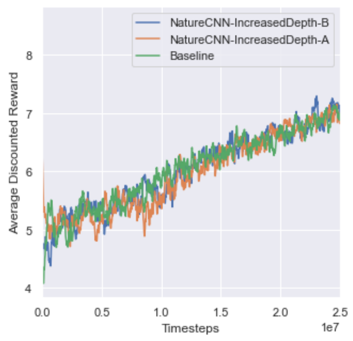
Figure 13: Training curves for models trained using smaller convolutional filters



(a) CoinRun

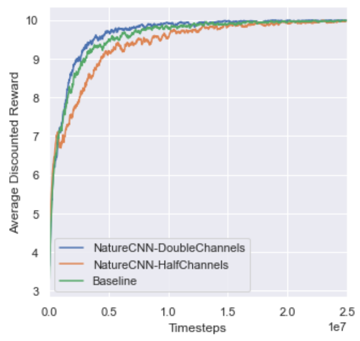


(b) Jumper

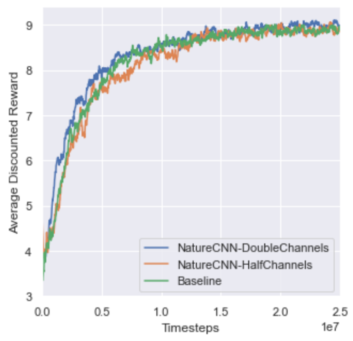


(c) Maze

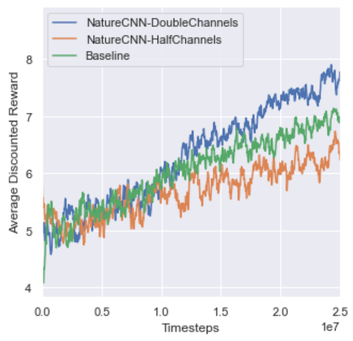
Figure 14: Training curves for models with different depth



(a) CoinRun

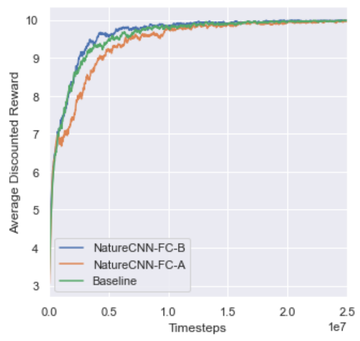


(b) Jumper

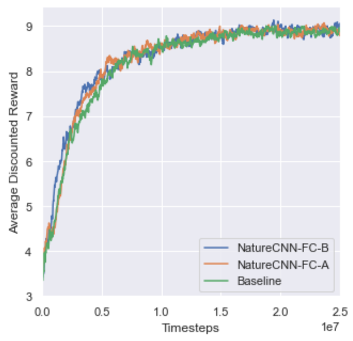


(c) Maze

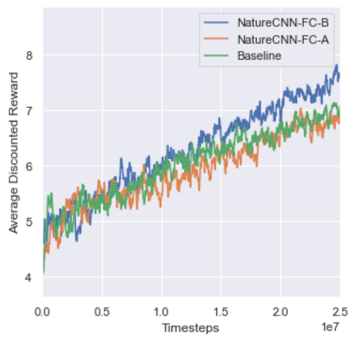
Figure 15: Training curves for models with different number of channels



(a) CoinRun

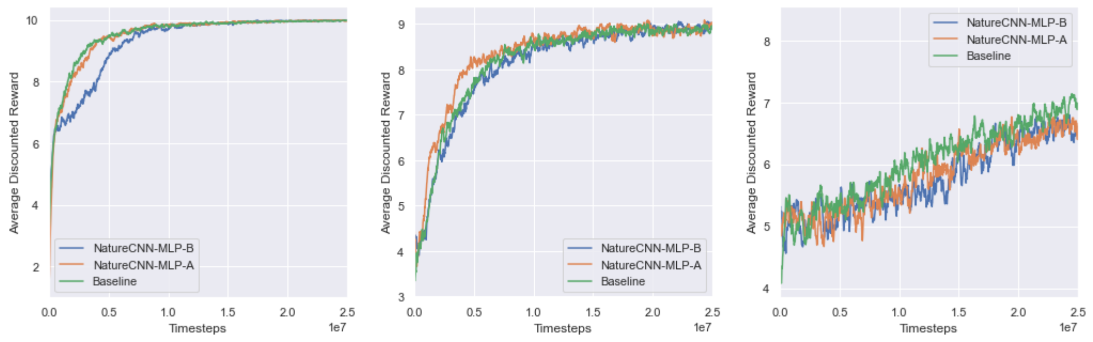


(b) Jumper



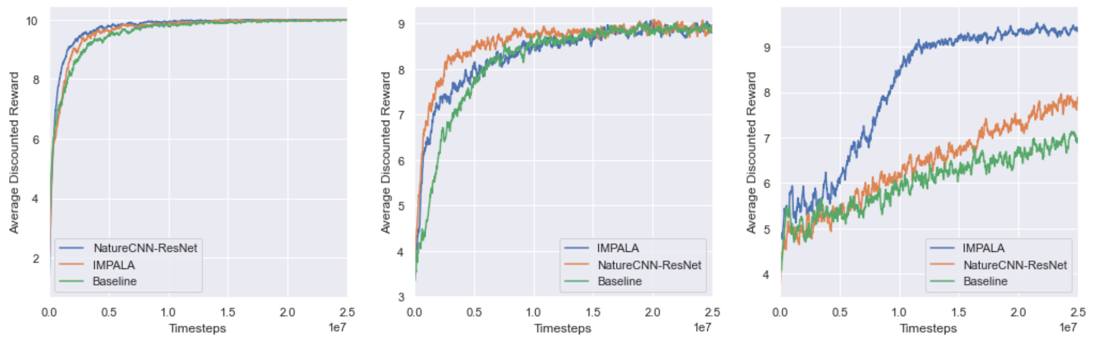
(c) Maze

Figure 16: Training curves for models trained using different number of neurons in the fully-connected layer



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

Figure 17: Training curves for models trained using a MLP instead of just a single fully-connected layer



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

Figure 18: Training curves for models trained with skip-connections