university of groningen / faculty of science and engineering / artificial intelligence

On The Existence of Double Descent in Reinforcement Learning

Bachelor's Project Thesis

Richard Harnisch, s5238366, r.f.harnisch@student.rug.nl,
Supervisor: Dr. Matthia Sabatelli

Abstract: Double Descent (DD) is a test-performance phenomenon in which a deep model's performance on a test set worsens around the interpolation threshold (overfitting) but then improves again when increasing training episodes or model size. We implement an empirical experiment to test whether this phenomenon can appear in Reinforcement Learning. To this end, we train TRPO agents on a family of seeded grid-worlds with obstacles and evaluate the agent on held-out maps. Across model sizes and training duration, we observe a generalization gap appearing between performance on training and held-out testing maps. However, upon increasing the size of the model as well as the amount of training timesteps we do not observe a second descent regime in test performance. The results suggest that in this context, increased capacity and training do not recover generalization and motivate further experiments with different algorithms, architectures, and environment types.

1 Introduction

Double Descent is a well-studied phenomenon in supervised learning. When increasing model capacity, training time, or dataset size, the test risk of deep models begins to improve (decrease) with the train risk up until a "sweet spot" at which test risk is traditionally understood to be minimized. Past this point, increasing model capacity, training time, or dataset size leads to overfitting, a behavior where the model memorizes the training data and thus fails to generalize over unseen test data. In other words, a generalization gap grows between the test performance and the train performance. This behavior can be visualized as a U-shaped risk curve, as shown in Figure 1.1.

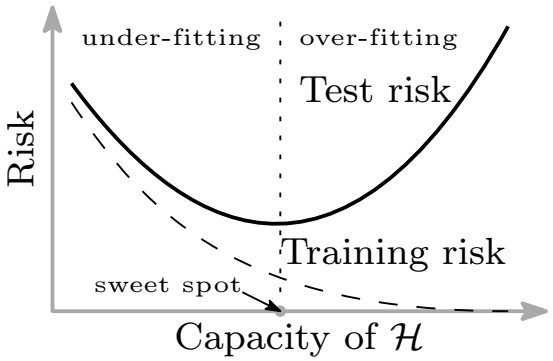


Figure 1.1: Traditional U-shaped risk curve. The training and test risk initially decrease together, until the model begins overfitting. Figure taken from Belkin et al. (2019).

However, work in the past years has shown that this traditional U-shaped risk curve is not the end of the story. Instead, after this initial overfitting phase, increasing model capacity, training time, or dataset size further leads to a second descent in test risk (Belkin et al., 2019; Nakkiran et al., 2019), as can be seen in Figure 1.2. This phenomenon has been coined "Double Descent" (DD) and has been observed across various architectures and datasets in supervised learning (Nakkiran et al., 2019).

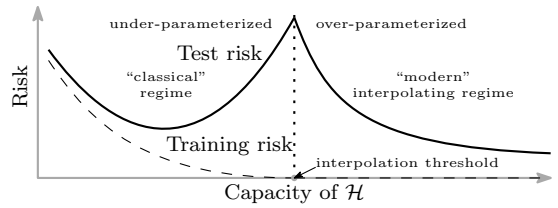


Figure 1.2: Double Descent risk curve. After the initial overfitting phase, increasing model capacity, training time, or dataset size further leads to a second descent in test risk. Figure taken from Belkin et al. (2019).

The reason for this behavior remains unsolved, but with a number of complementary hypotheses.

2 Methods

To replicate Double Descent as it occurs in Supervised Learning (SL) within Reinforcement Learning (RL), we need to define analogous concepts for test and training splits, and test and train risk. Since Double Descent is defined as a narrowing of the generalization gap between test and train risk, we need to be able to measure performance on both seen and unseen data. In SL, this is straightforward: the training split is the data the model is trained on, and the test split is held-out data the model has not seen during training. The train risk is the error (e.g., classification error, mean squared error) on the training split, and the test risk is the error on the test split. In RL, however, the concepts of training and test splits are less clear-cut, since the agent learns from interactions with an environment rather than from a fixed dataset.

To create train and test splits in Reinforcement Learning, we create a family of environments randomly generated based on a seed. Each environment in this family shares the same underlying structure (e.g., state and action space, transition dynamics, reward structure) but differs in specific map makeup (e.g., layout and obstacle placement). By training the RL agent on a subset of these environments (the training split) and evaluating its performance on a separate subset of unseen environments (the test split), we can emulate the train-test paradigm from SL. This approach allows us to assess the agent's ability to generalize its learned policy to new, unseen environments.

To plot a Double Descent curve, we further need to define an analogue to risk. In Supervised Learning, risk is typically defined as the expected loss (e.g., classification error, mean squared error) on a dataset. For our purposes, we use mean performance (i.e., average return) as a proxy for risk, where higher performance indicates lower risk. Additionally, we consider the trace of the Fisher Information Matrix (FIM) to indicate overfitting or memorizing in the model.

2.1 Environment

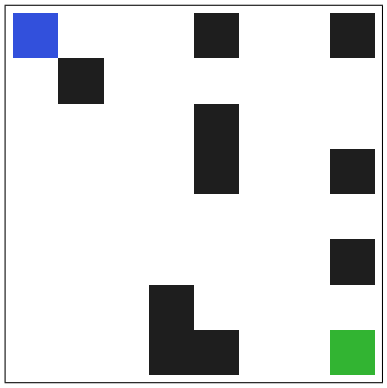
The environments used for this study are mazes. Each map is contained in an 8×8 grid, with the agent and the goal each taking up one tile. Tiles not occupied by the agent or the goal at game start are either empty or walls. Whether a tile is a wall is determined randomly during map generation, with a fixed probability of 0.2 for each tile to be a wall. This yields the possibility of generating maps that are unsolvable (i.e., there is no path from the agent to the goal). Such maps are discarded during generation and re-generated to ensure solvability. A selection of example maps can be seen in Figure 2.1.

The position of the goal and starting position of the agent can be randomized or set manually. We study both settings. In the manual setting, the goal is always placed in the bottom-right corner of the map (coordinates (7,7)) and the agent always starts in the top-left corner (coordinates (0,0)). In the randomized setting, both the agent's starting position and the goal position are randomized to one of the corners of the map at the start of each episode.

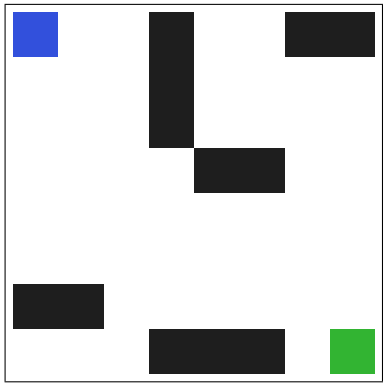
The environment's observation space consists of a binary vector, representing a flattened 8×8 grid where each tile is encoded using one-hot encoding to indicate whether it is empty, a wall, the agent's position, or the goal's position. The state space for this Markov Decision Process (MDP) is thus

$$\mathcal{S} = \{0, 1\}^{512} \tag{2.1}$$

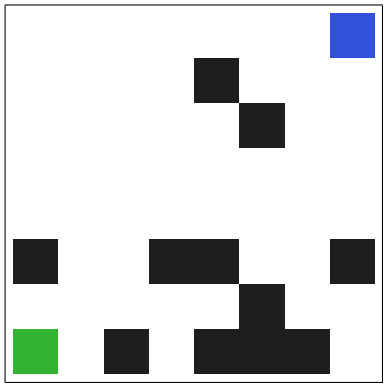
Each observation includes the current state of the grid as well as the previous state—frame-stacking. Each observation therefore encodes $8 \times 8 \times 4 \times 2 = 512$ bits. The action space is discrete and allows for moving in the four cardinal directions:



(a) Example environment in standard configuration. Walls are shown in black, the agent in blue, and the goal in green.



(b) Another example environment in standard configuration.



(c) Example environment in randomized configuration.

Figure 2.1: Example 8×8 maze environments used in this study.

$$\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\} \tag{2.2}$$

The agent receives a reward of $+1$ upon reaching the goal. In all other timesteps, a potential-based reward is applied. The potential function $\phi(s)$ is defined as one hundredth of the negative Euclidean distance from the agent's current position to the goal position. The reward at each timestep is then given by $r_t = \phi(s_t) - \phi(s_{t-1})$, encouraging the agent to move closer to the goal. Finally, the agent also receives a small time-penalty of -0.01 at each timestep to disincentivize standing still or taking suboptimal paths. Thus, the reward function is defined as follows:

$$r_t = \begin{cases} 1, & \text{if goal reached} \\ \phi(s_t) - \phi(s_{t-1}) - 0.01, & \text{otherwise} \end{cases} \tag{2.3}$$

with the potential function defined as:

$$\phi(s) = -\frac{1}{100} \cdot d_{\text{euclidean}}(\text{agent\_pos}, \text{goal\_pos}) \tag{2.4}$$

For example, if the agent moves a distance of exactly 1 unit closer to the goal in a timestep, it receives a reward of $0.01 - 0.01 = 0$. In all other cases the reward is negative, except when the goal is reached. Considering the agent starts in the opposite corner in the standard configuration and can thus move no farther away, the minimum return in the standard configuration is $G_{min} = -0.01 \times 64 = -0.64$. The minimum return in the randomized configuration, occurring when the agent does not start opposite of the goal, can be further lowered by the potential to increase distance to the goal of

$$G_{min} = \frac{1}{100} \times (\text{Max Distance} - \text{Starting Distance}) \tag{2.5}$$

$$= \frac{1}{100} \times (\sqrt{7^2 + 7^2} - 8) \tag{2.6}$$

$$\approx \frac{1}{100} \times (9.899 - 8) \tag{2.7}$$

$$= \frac{1}{100} \times 1.899 \tag{2.8}$$

$$= 0.01899 \tag{2.9}$$

Thus the minimum return becomes $-0.64 - 0.01899 \approx -0.65899$. The maximum return an agent can receive on a map depends on the map's

configuration, as some can be solved in fewer steps than others. However, we can compute the maximum return in the optimal case. Here, in the standard configuration, the agent needs at least 14 steps to reach the goal (moving 7 steps down and 7 steps right). In this case, the maximum return is:

$$G_{max} = 1 - (0.01 \times 13) - \phi(s_{end}) + \phi(s_{start}) \tag{2.10}$$

$$= 1 - 0.13 - (\frac{1}{100} \times 0) + (\frac{1}{100} \times \sqrt{7^2 + 6^2}) \tag{2.11}$$

$$\approx 0.87 + (\frac{1}{100} \times 9.21954446) \tag{2.12}$$

$$= 0.87 + 0.09210 \tag{2.13}$$

$$= 0.96210 \tag{2.14}$$

$$\tag{2.15}$$

Indeed, when computing the mean optimal return for all maps using seeds 0–9,999, we get 0.958585, slightly below this. This is expected as the frequency of maps in which the agent is forced to take more than 14 steps to reach the goal is very low—1.75% among these seeds, to be precise. In the randomized configuration, the maximum return is 1.0, as the agent can be positioned so that it can move a full unit closer to the goal in every step and thus gain a reward of zero in every step until reaching the goal, when it gains a reward of 1.

Therefore, the MDP under this environment can be formally defined as the tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$, where $\mathcal{S}$ is the state space as in (2.1), $\mathcal{A}$ is the action space as in (2.2), p is the state transition probability function defined by the environment dynamics, r is the reward function as in (2.3), and γ is the discount factor set to 1.

2.2 Metrics

For a y-axis representing train and test risk, we use mean return (i.e., average cumulative reward per episode) as a proxy for risk. Higher return indicates lower risk. We measure mean return on both the training environments (train risk) and the test environments (test risk) after every training epoch. To do this, we run an inference rollout on the set of training environments and the set of test environments. The set of test environments is equal in size to the set of training environments, up to a maximum of 100 test environments. Each test and train evaluation consists of running one episode on each environment in the respective set and taking the mean return across all episodes. We record this performance once every 10,000 training episodes. This is our most straightforward measure of a generalization gap between training and test performance, and in this context we would look for a "Double Ascent" as the test performance initially lags behind training performance before catching up again.

To further study generalization performance, we also measure the trace of the Fisher Information Matrix (FIM) after every training epoch. Instead of using the trajectory-based FIM, we use the state-action pair FIM, which is defined as:

$$F = \mathbb{E}_{(s,a)\sim\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s)^T\right] \tag{2.16}$$

where (s, a) are state-action pairs sampled from behavior under the policy π_θ. In other words, the FIM is the expected outer product of the gradient with respect to the model parameters θ of the score function, which is the log-probability of taking action a in state s. The FIM captures how much information about the parameters is contained in the actions taken by the policy in different states. Because of this definition, we can estimate the FIM using samples of state-action pairs collected during rollouts.

Let $g = \nabla_\theta \log \pi_\theta(a \mid s)$ be the gradient of the log-probability of action a given state s. Then the FIM is given by:

$$F = \mathbb{E}_{(s,a)\sim\pi_\theta}[gg^T] \tag{2.17}$$

The trace of the FIM, $\mathrm{Tr}(F)$, provides a measure of the sensitivity of the model's parameters to changes in the data distribution. A high trace value indicates that the model is highly sensitive to the training data, which can be a sign of overfitting or memorization. Conversely, a lower trace value suggests that the model is more robust and generalizes better to unseen data. By tracking the trace of the FIM on both training and test environments, we can gain insights into the model's generalization capabilities and its tendency to overfit as training progresses.

Using the definition of the FIM in Equation (2.17), we can derive an empirical estimator for the

trace of the FIM without forming the full matrix using the fact that the trace of a matrix is defined as the sum of its diagonal elements (e.g. $\sum_i F_{ii}$). We can thus derive:

$$\begin{aligned} \mathrm{Tr}(F) &= \mathrm{Tr}(\mathbb{E}_{(s,a)\sim\pi_\theta}[gg^T]) && (2.18)\\ &= \mathbb{E}_{(s,a)\sim\pi_\theta}[\mathrm{Tr}(gg^T)] && (2.19)\\ &= \mathbb{E}_{(s,a)\sim\pi_\theta}[g^T g] && (2.20) \end{aligned}$$

Therefore, to form a Monte Carlo estimator for the trace of the FIM, we can sample N state-action pairs (s_i, a_i) from rollouts under the policy π_θ and compute the score function gradients $g_i = \nabla_\theta \log \pi_\theta(a_i \mid s_i)$ for each pair. The empirical estimator for the trace of the FIM is then given by:

$$\widehat{\mathrm{Tr}(F)} = \frac{1}{N}\sum_{i=1}^{N} g_i^T g_i \tag{2.21}$$

where $g_i = \nabla_\theta \log \pi_\theta(a_i \mid s_i)$. This estimator allows us to compute the trace of the FIM efficiently without explicitly constructing the full matrix. Forming the full matrix would be prohibitively expensive for models with a large number of parameters (in the order of millions), as the FIM is of size $|F| \times |F|$ where $|F|$ is the number of parameters in the model.

Considering the trace of the FIM naturally scales with the size of the FIM, we can compute a trace per parameter by dividing the trace by the number of parameters in the model. This allows us to compare FIM trace values across models of different sizes. Because the trace of a matrix is also the sum of its eigenvalues, the trace per parameter can be interpreted as the average eigenvalue of the FIM. In other words, this metric can be described as the average Fisher information.

Considering we would expect higher average Fisher information when the model is overfitting or memorizing the training data, we can use this metric as a complementary measure of generalization performance alongside mean return. To support findings of Double Ascent in the mean return curves, we would expect to see the mean Fisher information to correlate with the size of the generalization gap between training and test performance. As the gap emerges, we would expect the mean Fisher information of the model to increase, which would remain until the gap narrows or closes.

2.3 Models

To conduct our experiment, we need to specify a model type. We run experiments both using a Deep Q-Network (DQN) (Mnih et al., 2013) and using a Policy Gradient method, specifically using Trust Region Policy Optimization (TRPO) agent (Schulman et al., 2017). Both models use a feedforward neural network as function approximator. The architecture of the neural network consists of an input layer matching the size of the observation space (512 units), followed by varying numbers of hidden layers of varying width with ReLU activations, and an output layer matching the size of the action space.

2.3.1 Deep Q-Networks

A Deep Q-Network (DQN) is a value-based Reinforcement Learning algorithm that approximates the optimal action-value function $Q^*(s, a)$ using a deep neural network. The action-value function estimates the expected cumulative reward for taking action a in state s and following the optimal policy thereafter. The DQN uses the Bellman equation as the foundation for its learning process, which states that the optimal action-value function satisfies:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{2.22}$$

where r is the immediate reward received after taking action a in state s, γ is the discount factor (in our case, $\gamma = 1$), and s' is the next state. The DQN approximates $Q^*(s, a)$ using a neural network parameterized by θ, denoted as $Q(s, a; \theta)$. The network is trained to minimize the difference between the predicted Q-values and the target Q-values derived from the Bellman equation. The loss function $L(\theta)$ used for training the DQN is defined as:

$$\mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{2.23}$$

where θ^- are the parameters of a target network that is periodically updated to stabilize training. The DQN employs experience replay, where transitions (s, a, r, s') are stored in a replay buffer and sampled randomly during training to break correlations between consecutive samples and improve learning stability.

We use a replay buffer size of 50,000 transitions and a target network update frequency of 5000 training steps. The DQN is trained using the Adam optimizer (Kingma & Ba, 2017) with a learning rate of 0.001 and a batch size of 64. The exploration-exploitation trade-off is managed using an epsilon-greedy strategy, where the exploration rate ϵ decays linearly from 1.0 to 0.05 over the first 30% of the episodes.

2.3.2 Trust Region Policy Optimizer

Trust Region Policy Optimization (TRPO) is a policy gradient method that aims to optimize the policy directly while ensuring stable and monotonic improvement. TRPO achieves this by constraining the step size of policy updates using a trust region approach, which prevents large, destabilizing updates to the policy. The core idea of TRPO is to maximize a surrogate objective function subject to a constraint on the Kullback-Leibler (KL) divergence between the old and new policies. The surrogate objective function is defined as:

$$L(\theta) = \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} A^{\pi_{\theta_{\text{old}}}}(s,a)\right] \quad (2.24)$$

where $\pi_{\theta_{\text{old}}}$ is the old policy, π_θ is the new policy parameterized by θ, and $A^{\pi_{\theta_{\text{old}}}}(s,a)$ is the advantage function estimating the relative value of action a in state s under the old policy. The KL divergence constraint is given by:

$$\mathbb{E}_{s\sim\pi_{\theta_{\text{old}}}}\left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \parallel \pi_\theta(\cdot \mid s))\right] \leq \delta \quad (2.25)$$

where δ is a predefined threshold that limits the size of the policy update. TRPO uses a conjugate gradient algorithm to solve the constrained optimization problem efficiently. The policy is updated iteratively by computing the natural gradient of the surrogate objective and scaling it to satisfy the KL divergence constraint.

We implement TRPO using the same feedforward neural network architecture as described in Section 2.3.1. The policy network outputs logits for each action, over which we use a softmax activation function to get a distribution over the action space. The value function is approximated using a separate value network with the same architecture but with a single output unit representing the state value. The advantage function is estimated using Generalized Advantage Estimation (GAE) to reduce variance in the policy gradient estimates.

The TRPO hyperparameters are a maximum KL divergence of $\delta = 10^{-2}$, conjugate-gradient iterations set to 10 with damping 0.1, backtracking coefficient 0.5 for 10 line-search steps, GAE $\lambda = 0.95$, and a batch size of 20 episodes per policy update. The value function is optimized for 5 iterations per update using Adam with learning rate 10^{-3}.

2.4 Training Runs

To search for Double Descent, we ran the models described above on various configurations of training environments and training durations. On the following configurations, we trained agents of varying depths between 3 and 8 and widths between 4 and 1024:

Type	Train Maps	Episodes	Start & Goal Position
TRPO	10	10,000	Standard*
DQN	50	20,000	Standard
DQN	100	50,000	Standard
TRPO	50	20,000	Randomized
TRPO	100	50,000	Randomized
TRPO	500	1,000,000	Standard
TRPO	750	1,000,000	Standard
TRPO	1,000	1,000,000	Standard

On the following configurations, we trained agents of depth 3 and widths 4, 16, 64, 256, and 1024 for unlimited time (in practice for about 1.8M episodes, until our high performance cluster ended the jobs):

Type	Train Maps	Start & Goal Position
TRPO	50	Standard
TRPO	100	Standard

3 Results

In this section, we present the results of our experiments searching for Double Descent in Reinforcement Learning. Unfortunately, across all configurations tested, we did not observe instances of Double Descent in the episoding or capacity regimes.

*Agent starts in top left, goal in bottom right

3.1 DQN Fails to Learn

Firstly, we examined performance using Deep Q-Networks (DQN) on our environment. Unfortunately, this architecture was unable to solve the environment altogether. We observe frequent catastrophic forgetting, with performance spikes being present but short-lived. Further examples of a DQN failing to learn the environment reliably can be found in the Appendix.

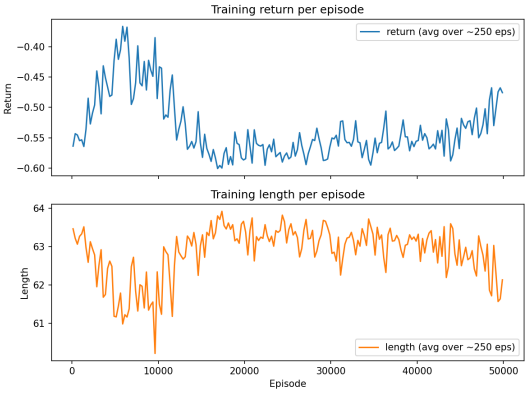
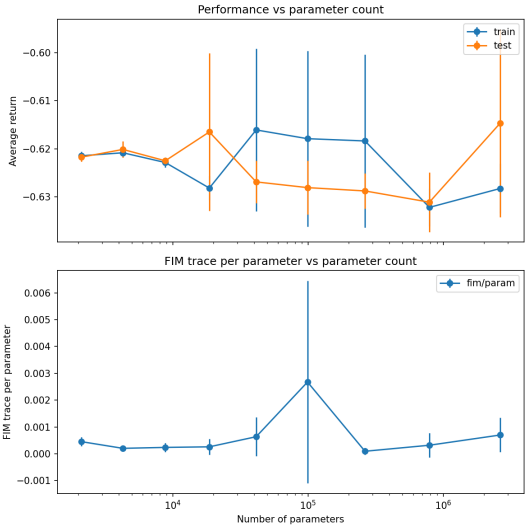


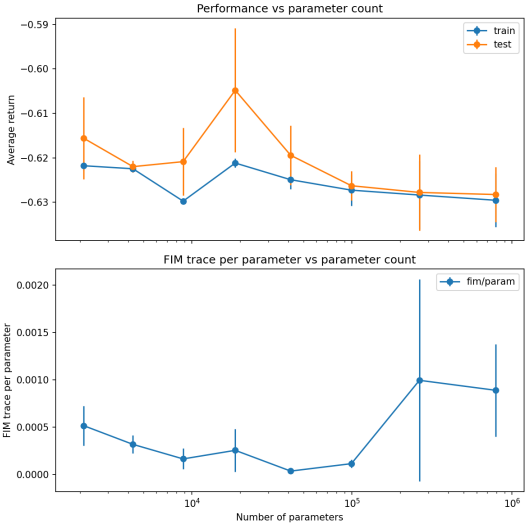
Figure 3.1: DQN performance using a model with width 512 and depth 3 over 50,000 episodes on 100 training maps. The model is unable to solve the environment, and reward (respective mean return over 200 batches for legibility) never becomes positive.

This occurs over a range of different model capacities in DQNs. Figure 3.2 shows metrics a range of model capacities, defined by varying width and depth of the neural network. No Double Descent is observed, as neither curve exhibits significant learning. Similarly, the FIM trace does not show interesting movement, staying extremely close to zero throughout different model capacities.

These results are not useful to our study of Double Descent, as the model fails to learn the environment in the first place. We cannot observe a second ascent of the test performance if there is no first ascent. Considering the poor performance of DQN on this environment, we decided to move to a different architecture that we hoped would perform better.



(a) Performance of DQN using 50 training seeds and 20,000 training episodes per model.



(b) Performance of DQN using 100 training seeds and 50,000 training episodes per model.

Figure 3.2: DQN training and test performance, and average Fisher information, as a function of model capacity (width and depth). Neither curve exhibits significant learning.

3.2 TRPO: No Double Descent

We next examined performance using Trust Region Policy Optimization (TRPO) on our environment. This architecture was able to learn the environment reliably across a range of model capacities. However, we were again unable to observe Double Descent in either the episoding or capacity regimes.

3.2.1 Capacity Regime

In the capacity regime, we conducted studies to heuristically search the available hyperparameter space. Our two hyperparameters to calibrate here are size of the training set (in seeds) and amount of training time (in episodes). We varied both hyperparameters across a range of values, as shown in Table 2.4.

First, we examined performance using 10 training seeds and 10,000 training episodes per model, with the standard environment configuration. Figure 3.3 shows the results of this experiment.

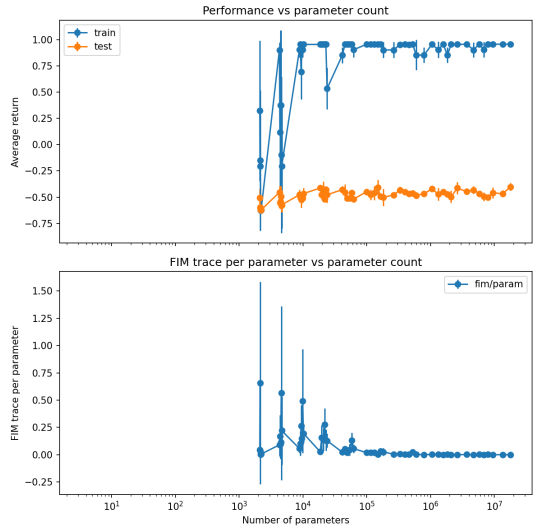


Figure 3.3: TRPO training and test performance, and average Fisher information, as a function of model capacity (width and depth), using 10 training seeds and 10,000 training episodes per model.

We can observe that both training and test performance are low when the model is at its smallest. As we increase model capacity, training performance increases rapidly until it hits the maximum reward

References

Belkin, M., Hsu, D., Ma, S., & Mandal, S. (2019, July). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, *116*(32), 15849–15854. Retrieved from `http://dx.doi.org/10.1073/pnas.1903070116` doi: 10.1073/pnas.1903070116

Kingma, D. P., & Ba, J. (2017). *Adam: A method for stochastic optimization.* Retrieved from `https://arxiv.org/abs/1412.6980`

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*. Retrieved from `https://arxiv.org/abs/1312.5602`

Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I. (2019). *Deep double descent: Where bigger models and more data hurt.* Retrieved from `https://arxiv.org/abs/1912.02292`

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017). Trust region policy optimization. *Proceedings of the 31st International Conference on Machine Learning*. Retrieved from `https://arxiv.org/abs/1502.05477`