arXiv:1706.10295v1 [cs.LG] 30 Jun 2017

Noisy Networks for Exploration

Meire Fortunato* **Mohammad Gheshlaghi Azar*** **Bilal Piot ***
Jacob Menick **Ian Osband** **Alex Graves** **Vlad Mnih**
Remi Munos **Demis Hassabis** **Olivier Pietquin** **Charles Blundell**
Shane Legg
DeepMind
{meirefortunato,mazar,piot,
jmenick,iosband,gravesa,vmnih,
munos,dhcontact,pietquin,cblundell,
legg}@google.com

Abstract

We introduce NoisyNet, a deep reinforcement learning agent with parametric noise added to its weights, and show that the induced stochasticity of the agent's policy can be used to aid efficient exploration. The parameters of the noise are learned with gradient descent along with the remaining network weights. NoisyNet is straightforward to implement and adds little computational overhead. We find that replacing the conventional exploration heuristics for A3C, DQN and dueling agents (entropy reward and ϵ-greedy respectively) with NoisyNet yields substantially higher scores for a wide range of Atari games, in some cases advancing the agent from sub to super-human performance.

1 Introduction

Despite the wealth of research into efficient methods for exploration in Reinforcement Learning (RL) [16, 15], most exploration heuristics rely on random perturbations of the agent's policy, such as ϵ-greedy [33] or entropy regularisation [37], to induce novel behaviours. However such local 'dithering' perturbations are unlikely to lead to the large-scale behavioural patterns needed for efficient exploration in many environments [24].

Optimism in the face of uncertainty is a common exploration heuristic in reinforcement learning. Various forms of this heuristic often come with theoretical guarantees on agent performance [2, 17, 15, 1, 16].

However, these methods are often limited to small state-action spaces or to linear function approximations and are not easily applied with more complicated function approximators such as neural networks. A more structured approach to exploration is to augment the environment's reward signal with an additional *intrinsic motivation* term [32] that explicitly rewards novel discoveries. Many such terms have been proposed, including learning progress [26], compression progress [30], variational information maximisation [14] and prediction gain [4]. One problem is that these methods separate the mechanism of generalisation from that of exploration; the metric for intrinsic reward, and–importantly–its weighting relative to the environment reward, must be chosen by the experimenter, rather than learned from interaction with the environment. Without due care, the optimal policy can be altered or even completely obscured by the intrinsic rewards; furthermore, dithering perturbations are usually needed as well as intrinsic reward to ensure robust exploration [25]. Exploration in the policy space itself, for example, with evolutionary or black box algorithms [22, 9, 29], usually requires many prolonged interactions with the environment. Although these algorithms are quite

*Equal contribution.

generic and can apply to any type of parametric policies (including neural networks), they are usually not data efficient and require a simulator to allow many policy evaluations.

We propose a simple alternative approach, called NoisyNet, where learned perturbations of the network weights are used to drive exploration. The key insight is that a single change to the weight vector can induce a consistent, and potentially very complex, state-dependent change in policy over multiple timesteps – unlike dithering approaches where decorrelated (and, in the case of ϵ-greedy, state-independent) noise is added to the policy at every step. The perturbations are sampled from a noise distribution. The variance of the perturbation is a parameter that can be considered as the energy of the injected noise. These variance parameters are learned using gradients from the reinforcement learning loss function, along side the other parameters of the agent. The approach differs from parameter compression schemes such as variational inference [12, 7] and flat minima search [13] since we do not maintain an explicit distribution over weights during training but simply inject noise in the parameters and tune its intensity automatically. It also differs from variational inference [11, 8] or Thompson sampling [35, 19] as the distribution on the parameters of our agents does not necessarily converge to an approximation of a posterior distribution.

At a high level our algorithm induces a randomised network for exploration, with care this approach can be provably-efficient with suitable linear value functions [24]. Previous attempts to extend this approach to deep neural networks required many duplicates of sections of the network [23]. By contrast our NoisyNet approach requires only one extra parameter per weight and applies to policy gradient methods such as A3C out of the box [20]. Most recently (and independently of our work), Plappert et al. [27] presented a similar technique where constant Gaussian noise is added to the parameters of the network. Our method thus differs by the ability of the network to adapt the noise injection with time and it is not restricted to Gaussian noise distributions. It can also be adapted to any deep RL algorithm and we demonstrate this versatility by providing NoisyNet versions of DQN (value-based) [21] and A3C (policy based) [20] algorithms. Experiments on 57 Atari games show that they both achieve striking gains when compared to the baseline algorithms without significant extra computational cost.

2 Background

This section provides mathematical background for Markov Decision Processes (MDPs) and deep RL with Q-learning and actor-critic methods.

2.1 Markov Decision Processes and Reinforcement Learning

MDPs model stochastic, discrete-time and finite action-state space control problems [5, 6, 28]. An MDP is a tuple $M = (\mathcal{X}, \mathcal{A}, R, P, \gamma)$ where $\mathcal{X}$ is the state space, $\mathcal{A}$ the action space, R the reward function, $\gamma \in]0, 1[$ the discount factor and P a stochastic kernel modelling the one-step Markovian dynamics ($P(y|x, a)$ is the probability of transitioning to state y by choosing action a in state x). A stochastic policy π maps each state to a distribution over actions $\pi(\cdot|x)$ and gives the probability $\pi(a|x)$ of choosing action a in state x. The quality of a policy π is assessed by the action-value function Q^π defined as:

$$Q^\pi(x, a) = \mathbb{E}^\pi \left[\sum_{t=0}^{+\infty} \gamma^t R(x_t, a_t) \right], \tag{1}$$

where $\mathbb{E}^\pi$ is the expectation over the distribution of the admissible trajectories $(x_0, a_0, x_1, a_1, \dots)$ obtained by executing the policy π starting from $x_0 = x$ and $a_0 = a$. Therefore, the quantity $Q^\pi(x, a)$ represents the expected γ-discounted cumulative reward collected by executing the policy π starting from x and a. A policy is optimal if no other policy yields a higher return. The action-value function of the optimal policy is $Q^\star(x, a) = \arg\max_\pi Q^\pi(x, a)$.

The value function V^π for a policy is defined as

$$V^\pi(x) = \mathbb{E}_{a \sim \pi(\cdot|x)}[Q^\pi(x, a)], \tag{2}$$

and represents the expected γ-discounted return collected by executing the policy π starting from state x.

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning uses deep neural networks as function approximators for RL methods. Deep Q-Networks (DQN) [21], Asynchronous Advantage Actor-Critic (A3C) [20], Trust Region Policy Optimisation [31] and Deep Deterministic Policy Gradient [18] are examples of such algorithms. They frame the RL problem as the minimisation of a loss function $L(\theta)$, where θ represents the parameters of the network. In our experiments we shall consider the DQN and A3C algorithms.

DQN [21] uses a neural network as an approximator for the action-value function of the optimal policy $Q^{\star}(x, a)$. DQN's estimate of the optimal action-value function, $Q(x, a)$, is found by minimising the following loss with respect to the neural network parameters θ:

$$L(\theta) = \mathbb{E}_{(x,a,r,y)\sim D}\left[\left(r + \gamma \max_{b \in A} Q(y, b; \theta^{-}) - Q(x, a; \theta)\right)^2\right], \tag{3}$$

where D is a distribution over transitions $e = (x, a, r = R(x, a), y \sim P(\cdot|x, a))$ drawn from a replay buffer of previously observed transitions. Here θ^{-} represents the parameters of a fixed and separate target network which is updated ($\theta^{-} \leftarrow \theta$) regularly to stabilise the learning. An ϵ-greedy policy is used to pick actions greedily according to the action-value function Q or, with probability ϵ, a random action is taken.

In contrast, A3C [20] is a policy gradient algorithm. A3C's network directly learns a policy π and a value function V of its policy. The gradient of the loss on the A3C policy at step t for the roll-out $(x_{t+i}, a_{t+i} \sim \pi(\cdot|x_{t+i}; \theta), r_{t+i})_{i=0}^{k}$ is:

$$\nabla_\theta L^{\pi}(\theta) = -\mathbb{E}^{\pi}\left[\sum_{i=0}^{k} \nabla_\theta \log(\pi(a_{t+i}|x_{t+i}; \theta))A(x_{t+i}, a_{t+i}; \theta) + \beta \sum_{i=0}^{k} \nabla_\theta H(\pi(\cdot|x_{t+i}; \theta))\right]. \tag{4}$$

$H[\pi(\cdot|x_t; \theta)]$ denotes the entropy of the policy π and β is a hyperparameter that trades off between optimising the advantage function and the entropy of the policy. The advantage function $A(x_{t+i}, a_{t+i}; \theta)$ is the difference between observed returns and estimates of the return produced by A3C's value network: $A(x_{t+i}, a_{t+i}; \theta) = \sum_{j=i}^{k} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta) - V(x_{t+i}; \theta)$, r_{t+j} being the reward at step $t + j$ and $V(x; \theta)$ being the agent's estimate of value function of state x.

Parameters of the value function are found to match on-policy returns: $L^{V}(\theta) = \mathbb{E}^{\pi}\left[\sum_{i=0}^{k} (\hat{Q}_i - V(x_{t+i}; \theta))^2\right]$ where $\hat{Q}_i$ is the return obtained by executing policy π starting in state x_{t+i}: $\hat{Q}_i = \sum_{j=i}^{k} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta)$. The overall A3C loss is then $L(\theta) = L^{\pi}(\theta) + \lambda L^{V}(\theta)$ where λ balances optimising the policy loss relative to the baseline value function loss.

3 NoisyNets for Reinforcement Learning

NoisyNets are neural networks whose weights and biases are perturbed by a parametric function of the noise. These parameters are adapted with gradient descent. More precisely, let $y = f_\theta(x)$ be a neural network parameterised by the vector of *noisy* parameters θ which takes the input x and outputs y. We represent the noisy parameters θ as $\theta \stackrel{\text{def}}{=} \mu + \Sigma \odot \varepsilon$, where $\zeta \stackrel{\text{def}}{=} (\mu, \Sigma)$ is a set of vectors of learnable parameters, ε is a vector of zero-mean noise with fixed statistics and $\odot$ represents element-wise multiplication. The usual loss of the neural network is wrapped by expectation over the noise ε: $\bar{L}(\zeta) \stackrel{\text{def}}{=} \mathbb{E}\left[L(\theta)\right]$. Optimisation now occurs with respect to the set of parameters ζ.

Consider a linear layer of a neural network with p inputs and q outputs, represented by

$$y = wx + b \tag{5}$$

where $x \in \mathbb{R}^{p}$ is the layer input, $w \in \mathbb{R}^{q\times p}$ the weight matrix, and $b \in \mathbb{R}^{q}$ the bias. The corresponding noisy linear layer is defined as:

$$y \stackrel{\text{def}}{=} (\mu^{w} + \sigma^{w} \odot \varepsilon^{w})x + \mu^{b} + \sigma^{b} \odot \varepsilon^{b}, \tag{6}$$

where $\mu^{w} + \sigma^{w} \odot \varepsilon^{w}$ and $\mu^{b} + \sigma^{b} \odot \varepsilon^{b}$ replace w and b in Eq. (5), respectively. The parameters $\mu^{w} \in \mathbb{R}^{q\times p}$, $\mu^{b} \in \mathbb{R}^{q}$, $\sigma^{w} \in \mathbb{R}^{q\times p}$ and $\sigma^{b} \in \mathbb{R}^{q}$ are learnable whereas $\varepsilon^{w} \in \mathbb{R}^{q\times p}$ and $\varepsilon^{b} \in \mathbb{R}^{q}$ are noise random variables (the specific choices of this distribution are described below).

We now turn to explicit instances of the noise distributions for linear layers in a noisy network. We explore two options:

(a) Independent Gaussian noise: the noise applied to each weight and bias is independent, where each entry $\varepsilon^w_{i,j}$ (respectively each entry ε^b_j) of the random matrix ε^w (respectively of the random vector ε^b) is drawn from a unit Gaussian distribution. This means that for each noisy linear layer, there are $pq + q$ noise variables (for p inputs to the layer and q outputs).

(b) Factorised Gaussian noise reduces the number of random variables in the network from one per weight, to one per input and one per output of each noisy linear layer. By factorising $\varepsilon^w_{i,j}$, we can use p unit Gaussian variables ε_i for noise of the inputs and and q unit Gaussian variables ε_j for noise of the outputs (thus $p + q$ unit Gaussian variables in total). Each $\varepsilon^w_{i,j}$ and ε^b_j can then be written as:

$$\varepsilon^w_{i,j} = f(\varepsilon_i) f(\varepsilon_j) \tag{7}$$

$$\varepsilon^b_j = f(\varepsilon_j), \tag{8}$$

where f is a functions. In our experiments we used $f(x) = \mathrm{sgn}(x)\sqrt{|x|}$.

Since the loss of a noisy network, $\bar{L}(\zeta) = \mathbb{E}\left[L(\theta)\right]$, is an expectation over the noise, the gradients are straightforward to obtain:

$$\nabla \bar{L}(\zeta) = \nabla \mathbb{E}\left[L(\theta)\right] = \mathbb{E}\left[\nabla_{\mu,\Sigma} L(\mu + \Sigma \odot \varepsilon)\right]. \tag{9}$$

We use a Monte Carlo approximation to the above gradients, taking a single sample ξ at each step of optimisation:

$$\nabla \bar{L}(\zeta) \approx \nabla_{\mu,\Sigma} L(\mu + \Sigma \odot \xi). \tag{10}$$

3.1 Deep Reinforcement Learning with NoisyNets

We now turn to our application of noisy networks to exploration in deep reinforcement learning. Noise drives exploration in many methods for reinforcement learning, providing a source of stochasticity external to the agent and the RL task at hand. Either the scale of this noise is manually tuned across a wide range of tasks (as is the practice in general purpose agents such as DQN or A3C) or it can be manually scaled per task. Here we propose automatically tuning the level of noise added to an agent for exploration, using the noisy networks training to drive down (or up) the level of noise injected into the parameters of a neural network, as needed.

A noisy network agent samples a new set of parameters after every step of optimisation. Between optimisation steps, the agent acts according to a fixed set of parameters (weights and biases). This ensures that the agent always acts according to parameters that are drawn from the current noise distribution.

Deep Q-Networks (DQN) We modify DQN as follows: first, ε-greedy is no longer used, but instead the policy greedily optimises the value function. Secondly, the fully connected layers of the value function are parameterised as a noisy network, where the parameters are drawn from the noisy network parameter distribution after every replay step. We used factorised Gaussian noise as explained in (b) from Sec. 3. For replay, the current noisy network parameter sample is held fixed. Since DQN takes one step of optimisation for every action step, the noisy network parameters are re-sampled before every action. We call this adaptation of DQN, NoisyNet-DQN.

We now provide the details of the loss function that our variant of DQN is minimising. When replacing the linear layers by noisy layers in the network (respectively in the target network), the parameterised action-value function $Q(x, a, \varepsilon; \zeta)$ (respectively $Q(x, a, \varepsilon'; \zeta^-)$) can be seen as a random variable and the DQN loss becomes the NoisyNet-DQN loss:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b\in A} Q(y, b, \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right]. \tag{11}$$

where the outer expectation is with respect to distribution of the noise variables ε for the noisy value function $Q(x, a, \varepsilon; \zeta)$ and the noise variable ε' of the noisy target value function $Q(y, b, \varepsilon'; \zeta^-)$. Computing an unbiased estimate of the loss is straightforward as we only need to compute, for each transition in the replay buffer, one instance of the target network and one instance of the learning network. Concerning the action choice, we sample from the target network and we act greedily with respect to the output action-value function. The algorithm is provided in Sec. C.1.

Asynchronous Advantage Actor Critic (A3C) A3C is modified in a similar fashion to DQN: firstly, the entropy bonus of the policy loss is removed. Secondly, the fully connected layers of the policy network are parameterised as a noisy network. We used independent Gaussian noise as explained in (a) from Sec. 3. In A3C, there is no explicit exploratory action selection scheme (such as ϵ-greedy); and the chosen action is always drawn from the current policy. For this reason, an entropy bonus of the policy loss is often added to discourage updates leading to deterministic policies. However, when adding noisy weights to the network, sampling these parameters corresponds to choosing a different current policy which naturally favours exploration. As a consequence of direct exploration in the policy space, the artificial entropy loss on the policy can thus be omitted. New parameters of the policy network are sampled after each step of optimisation, and since A3C uses n step returns, optimisation occurs every n steps. We call this modification of A3C, NoisyNet-A3C.

Indeed, when replacing the linear layers by noisy linear layers (the parameters of the noisy network are now noted ζ), we obtain the following estimation of the return via a roll-out of size k:

$$\hat{Q}_i = \sum_{j=i}^{k} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \zeta, \varepsilon_i). \tag{12}$$

As A3C is an on-policy algorithm the gradients are unbiased when noise of the network is consistent for the whole roll-out. Consistency among action value functions $\hat{Q}_i$ is ensured by letting each action value function have its noise be the *same*, i.e., $\forall i, \varepsilon_i = \varepsilon$. Additional details are provided in the Appendix Sec. A.2 and the algorithm is given in Sec. C.2.

4 Results

We evaluated the performance of noisy network agents on 57 Atari games [3] and compared to baselines that, without noisy networks, rely upon the original exploration methods (ε-greedy and entropy bonus). We used the random start no-ops scheme for training and evaluation as described the original DQN paper [21]. The mode of evaluation is identical to those of [20] where randomised restarts of the games are used for evaluation after training has happened.

We consider three baseline agents: DQN [21], duel clip variant of Dueling algorithm [36], and A3C [20]. The NoisyNet variant of each of the agents was described in the previous section, except for the dueling case which is a straightforward adaptation of the DQN described in [36]. In each case, we used the neural network architecture from the corresponding original papers for both the baseline and NoisyNet variant. For the NoisyNet variants we searched over the same hyperparameters as in the respective original paper for the baseline.

We compared absolute performance of agents using the human normalised score:

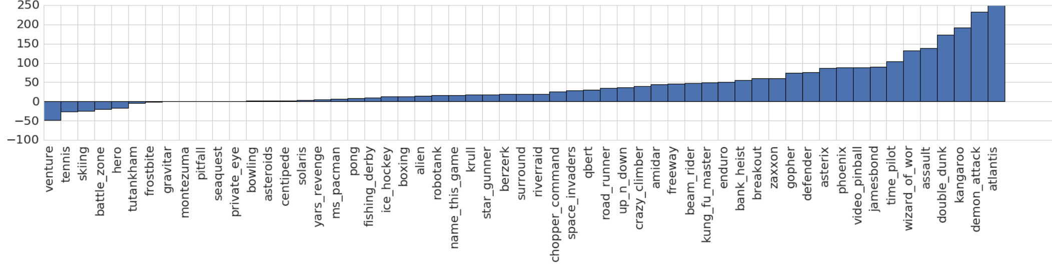
$$100 \times \frac{\text{Score}_{\text{agent}} - \text{Score}_{\text{Random}}}{\text{Score}_{\text{Human}} - \text{Score}_{\text{Random}}}, \tag{13}$$

where human and random scores are the same as those in [36]. Note that the absolute human normalised score is zero for a random agent and 100 for human level performance. Per-game scores are computed by taking the maximum performance for each game and then averaging over three seeds. The overall agent performance is measured by either the mean or median of the human normalised score across all 57 Atari games.

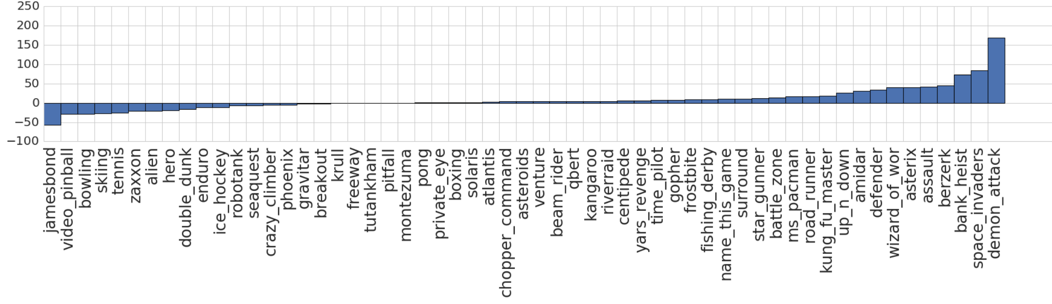
The aggregate results across all 57 Atari games are shown in Table 1, while the individual scores for each game are in Table 2 from the Appendix. The median human normalised score is improved in all agents by using NoisyNet, adding at least 20 percentage points to the median human normalised score for all agents. The mean human normalised score is also dramatically improved for DQN and A3C agents, but the mean score is worse for Dueling. Note that the mean human normalised score is often dominated by the performance of the agent on just a few games and so is not a robust estimator of overall agent performance. Interestingly the Dueling case demonstrates that NoisyNet is orthogonal to several other improvements made to DQN.

We also compared relative performance of NoisyNet agents to the respective baseline agent without noisy networks:

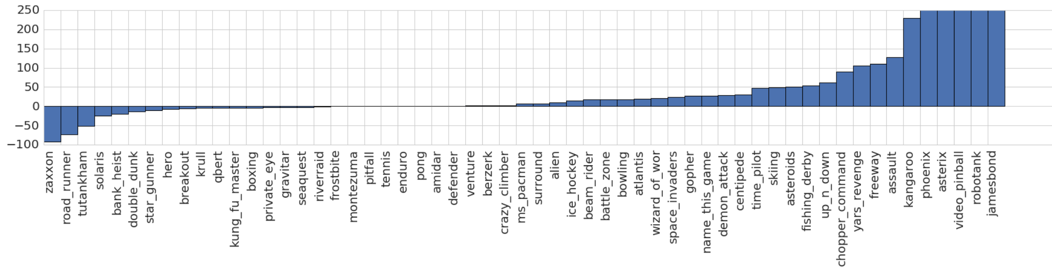
$$100 \times \frac{\text{Score}_{\text{NoisyNet}} - \text{Score}_{\text{Baseline}}}{\max(\text{Score}_{\text{Human}}, \text{Score}_{\text{Baseline}}) - \text{Score}_{\text{Random}}}, \tag{14}$$



(a) Improvement in percentage of NoisyNet-DQN over DQN [21]



(b) Improvement in percentage of NoisyNet-Dueling over Dueling [36]



(c) Improvement in percentage of NoisyNet-A3C over A3C [20]

Figure 1: Comparison of NoisyNet agent versus the baseline according to Eq. (14). The maximum score is truncated at 250%.

	Baseline		NoisyNet	
	Mean	Median	Mean	Median
DQN	213	47	**1210**	**89**
A3C	418	93	**1112**	**121**
Dueling	**2102**	126	1908	**154**

Table 1: Comparison between the baseline DQN [21], A3C [20] and Dueling [36] and their NoisyNet version in terms of median and mean human-normalised scores defined in Eq. (13).

As before, the per-game score is computed by taking the maximum performance for each game and then averaging over three seeds. The relative human normalised scores are shown in Figure 1. As can be seen, the performance of NoisyNet agents (DQN, Dueling and A3C) is better for the majority of games relative to the corresponding baseline, and in some cases by a considerable margin. This is especially evident in the case of NoisyNet-A3C, where the agent can solve some games that the baseline A3C previously could not solve (e.g., see the results for freeway and Kangaroo in Fig. 2). In some other games, NoisyNet agents provide an order of magnitude improvement on the performance of the vanilla agent; as can be seen in Table 2 in the Appendix with detailed breakdown of individual

game scores. More information is provided in the learning curves plots from Figs 3, 4 and 5, for A3C, DQN and dueling respectively.

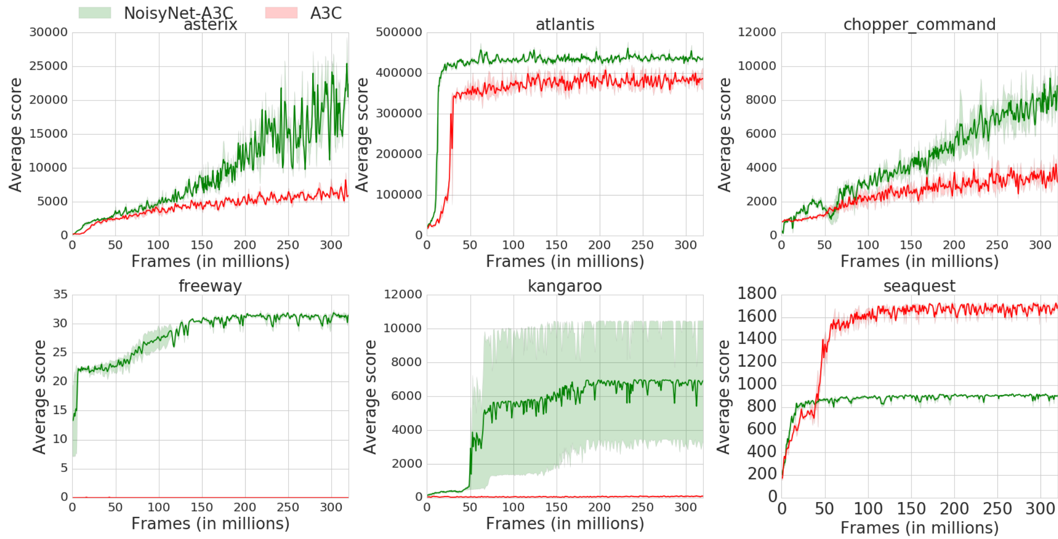


Figure 2: Training curves for selected Atari games comparing A3C and NoisyNet-A3C. Please refer to Fig. 3 in the Appendix for additional games.

5 Conclusion

We have presented a general method for exploration in deep reinforcement learning that shows significant performance improvements across many Atari games in three different agent architectures. In particular, we observe that in games such as Asterix and Freeway that the standard DQN and A3C perform poorly compared with the human player, NoisyNet-DQN and NoisyNet-A3C achieve super human performance. Our method eliminates the need for ϵ-greedy and the entropy bonus commonly used in Q-learning-style and policy gradient methods, respectively. Instead we show that better exploration is possible by relying on perturbations in weight space to drive exploration. This is in contrast to many other methods that add intrinsic motivation signals that may destabilise learning or change the optimal policy. Another interesting feature of the NoisyNet approach is that the degree of exploration is contextual and varies from state to state based upon per-weight variances. While more gradients are needed, the gradients on the mean and variance parameters are related to one another by a computationally efficient affine function, thus the computational overhead is marginal. Automatic differentiation makes implementation of our method a straightforward adaptation of many existing methods. A similar randomisation technique can also be applied to LSTM units [10] and is easily extended to reinforcement learning, we leave this as future work.

Note NoisyNet exploration strategy is not restricted to the baselines considered in this paper. In fact, this idea can be applied to any deep RL algorithms that can be trained with gradient descent, including DDPG [18] and TRPO [31]. As such we believe this work is a step towards the goal of developing a universal exploration strategy.

Acknowledgements We would like to thank Koray Kavukcuoglu, Oriol Vinyals, Daan Wierstra, Georg Ostrovski, Joseph Modayil, Simon Osindero, Chris Apps, Stephen Gaffney and many others at DeepMind for insightful discussions, comments and feedback on this work.

References

[1] Peter Auer and Ronald Ortner. Logarithmic online regret bounds for undiscounted reinforcement learning. *Advances in Neural Information Processing Systems*, 19:49, 2007.

[2] Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. Minimax regret bounds for reinforcement learning. *arXiv preprint arXiv:1703.05449*, 2017.

[3] Marc Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[4] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[5] Richard Bellman and Robert Kalaba. *Dynamic programming and modern control theory*. Academic Press New York, 1965.

[6] Dimitri Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific, Belmont, MA, 1995.

[7] Chris M Bishop. Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

[8] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 1613–1622, 2015.

[9] Jeremy Fix and Matthieu Geist. Monte-Carlo swarm policy search. In *Swarm and Evolutionary Computation*, pages 75–83. Springer, 2012.

[10] Meire Fortunato, Charles Blundell, and Oriol Vinyals. Bayesian recurrent neural networks. *arXiv preprint arXiv:1704.02798*, 2017.

[11] Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.

[12] Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 5–13. ACM, 1993.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[14] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. VIME: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[15] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11(Apr):1563–1600, 2010.

[16] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.

[17] Tor Lattimore, Marcus Hutter, and Peter Sunehag. The sample-complexity of general reinforcement learning. In *Proceedings of The 30th International Conference on Machine Learning*, pages 28–36, 2013.

[18] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[19] Zachary C Lipton, Jianfeng Gao, Lihong Li, Xiujun Li, Faisal Ahmed, and Li Deng. Efficient exploration for dialogue policy learning with BBQ networks & replay buffer spiking. *arXiv preprint arXiv:1608.05081*, 2016.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] David E Moriarty, Alan C Schultz, and John J Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999.

[23] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. In *Advances In Neural Information Processing Systems*, pages 4026–4034, 2016.

[24] Ian Osband, Daniel Russo, Zheng Wen, and Benjamin Van Roy. Deep exploration via randomized value functions. *arXiv preprint arXiv:1703.07608*, 2017.

[25] Georg Ostrovski, Marc G Bellemare, Aaron van den Oord, and Remi Munos. Count-based exploration with neural density models. *arXiv preprint arXiv:1703.01310*, 2017.

[26] Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? A typology of computational approaches. *Frontiers in neurorobotics*, 1, 2007.

[27] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[28] Martin Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

[29] Tim Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *ArXiv e-prints*, 2017.

[30] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010.

[31] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Proc. of ICML*, pages 1889–1897, 2015.

[32] Satinder P Singh, Andrew G Barto, and Nuttapong Chentanez. Intrinsically motivated reinforcement learning. In *NIPS*, volume 17, pages 1281–1288, 2004.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998.

[34] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proc. of NIPS*, volume 99, pages 1057–1063, 1999.

[35] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[36] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1995–2003, 2016.

[37] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

A Algorithm Descriptions

A.1 NoisyNet-DQN implementation details

DQN [21] is an approximate value iteration technique that uses a deep convolutional neural network to represent the Q-function as $Q(x, a; \theta)$, where θ are the parameters of the network. We used the network architecture from [21] and also the same set up for interacting with the Atari games. We note, however, that we did not use ϵ-greedy technique, instead relied solely upon NoisyNet for exploration. In DQN, the neural network is trained by minimising the following loss:

$$\begin{aligned} L(\theta) &= \mathbb{E}_{(x,a,r,y)\sim D}[y^{DQN} - Q(x, a; \theta)]^2, \\ y^{DQN} &= r + \gamma \max_{b \in A} Q(y, b; \theta^-) \end{aligned} \tag{15}$$

where D is a distribution over transitions $e = (x, a, r = R(x, a), y \sim P(\cdot|x, a))$. Here θ^- represents the parameters of the target network which is updated ($\theta^- \leftarrow \theta$) regularly and stabilises the learning. The distribution D is generated by constructing a replay buffer were training examples are uniformly sampled from previous experience tuples $e_t = (x_t, a_t, r_t, x_{t+1})$. We now provide the details of the loss function that our variant of DQN with noisy network is minimising. When replacing the linear layers by noisy layers (the parameters are now noted ζ) in the network (respectively in the target network), the parameterised action-value function $Q(x, a, \varepsilon; \zeta)$ (respectively $Q(x, a, \varepsilon'; \zeta^-)$) can be seen as a random variable and the DQN loss becomes the NoisyNet-DQN loss:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b \in A} Q(y, b, \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right]. \tag{16}$$

The goal of NoisyNet-DQN is to optimize the loss $\bar{L}(\zeta)$ in terms of ζ.

A.2 NoisyNet-A3C implementation details

In contrast with value-based algorithms, policy-based methods such as A3C [20] parameterise the policy $\pi(a|x; \theta_\pi)$ directly and update the parameters θ_π by performing a gradient ascent on the mean value-function $\mathbb{E}_{s\sim D}[V^{\pi(\cdot|\cdot;\theta_\pi)}(x)]$ (also called the expected return) [34]. A3C uses a deep neural network with weights $\theta = \theta_\pi \cup \theta_V$ to parameterise the policy π and the value V. The network has one softmax output for the policy-head $\pi(\cdot|\cdot; \theta_\pi)$ and one linear output for the value-head $V(\cdot; \theta_V)$, with all non-output layers shared. The parameters θ_π (resp. θ_V) are relative to the shared layers and the policy head (resp. the value head). A3C is an asynchronous and online algorithm that uses roll-outs of size $k + 1$ of the current policy to perform a policy improvement step.

In A3C, there is no explicit exploratory action selection scheme (such as ϵ-greedy); and the chosen action is always drawn from the current policy. For this reason, an entropy loss on the policy is often added to discourage updates leading to deterministic policies. However, when using noisy weights in the network, sampling corresponds to choosing a different current policy which naturally favours exploration. As a consequence of direct exploration in the policy space, the artificial entropy loss on the policy can thus be omitted.

For simplicity, here we present the A3C version with only one thread. For a multi-thread implementation, refer to the pseudo-code C.2 or to the original A3C paper [20]. In order to train the policy-head, an approximation of the policy-gradient is computed for each state of the roll-out $(x_{t+i}, a_{t+i} \sim \pi(\cdot|x_{t+i}; \theta_\pi), r_{t+i})_{i=0}^k$:

$$\nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i}; \theta_\pi))[\hat{Q}_i - V(x_{t+i}; \theta_V)], \tag{17}$$

where $\hat{Q}_i$ is an estimation of the return $\hat{Q}_i = \sum_{j=i}^{k} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta_V)$. The gradients are then added to obtain the cumulative gradient of the roll-out:

$$\sum_{i=0}^{k} \nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i}; \theta_\pi))[\hat{Q}_i - V(x_{t+i}; \theta_V)]. \tag{18}$$

A3C trains the value-head by minimising the error between the estimated return and the value $\sum_{i=0}^{k} (\hat{Q}_i - V(x_{t+i}; \theta_V))^2$. Therefore, the network parameters (θ_π, θ_V) are updated after each

roll-out as follows:

$$\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \sum_{i=0}^{k} \nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i};\theta_\pi))[\hat{Q}_i - V(x_{t+i};\theta_V)], \tag{19}$$

$$\theta_V \leftarrow \theta_V - \alpha_V \sum_{i=0}^{k} \nabla_{\theta_V} [\hat{Q}_i - V(x_{t+i};\theta_V)]^2, \tag{20}$$

where (α_π, α_V) are hyper-parameters. As mentioned previously, in the original A3C algorithm, it is recommended to add an entropy term $\beta \sum_{i=0}^{k} \nabla_{\theta_\pi} H(\pi(\cdot|x_{t+i};\theta_\pi))$ to the policy update, where $H(\pi(\cdot|x_{t+i};\theta_\pi)) = -\beta \sum_{a\in A} \pi(a|x_{t+i};\theta_\pi) \log(\pi(a|x_{t+i};\theta_\pi))$. Indeed, this term encourages exploration as it favours policies which are uniform over actions. When replacing the linear layers in the value and policy heads by noisy layers (the parameters of the noisy network are now ζ_π and ζ_V), we obtain the following estimation of the return via a roll-out of size k:

$$\hat{Q}_i = \sum_{j=i}^{k} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k};\zeta_V, \varepsilon_i). \tag{21}$$

We would like $\hat{Q}_i$ to be a consistent estimate of the return of the current policy. To do so, we should force $\forall i, \varepsilon_i = \varepsilon$. As A3C is an on-policy algorithm, this involves fixing the noise of the network for the whole roll-out so that the policy produced by the network is also fixed. Hence, each update of the parameters (ζ_π, ζ_V) is done after each roll-out with the noise of the whole network held fixed for the duration of the roll-out:

$$\zeta_\pi \leftarrow \zeta_\pi + \alpha_\pi \sum_{i=0}^{k} \nabla_{\zeta_\pi} \log(\pi(a_{t+i}|x_{t+i};\zeta_\pi, \varepsilon))[\hat{Q}_i - V(x_{t+i};\zeta_V, \varepsilon)], \tag{22}$$

$$\zeta_V \leftarrow \zeta_V - \alpha_V \sum_{i=0}^{k} \nabla_{\zeta_V} [\hat{Q}_i - V(x_{t+i};\zeta_V, \varepsilon)]^2. \tag{23}$$

B Initialisation of Noisy Networks

In the case of an unfactorised noisy networks, the parameters μ and σ are initialised as follows. Each element $\mu_{i,j}$ is sampled from independent uniform distributions $\mathcal{U}[-\sqrt{\frac{3}{p}}, +\sqrt{\frac{3}{p}}]$, where p is the number of inputs to the corresponding linear layer, and each element $\sigma_{i,j}$ is simply set to 0.017 for all parameters.

For factorised noisy networks, each element $\mu_{i,j}$ was initialised by a sample from an independent uniform distributions $\mathcal{U}[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}]$ and each element $\sigma_{i,j}$ was initialised to a constant $\frac{\sigma_0}{\sqrt{p}}$. The hyperparameter σ_0 is set to 0.4.

C Algorithms

C.1 NoisyNet-DQN

Algorithm 1: NoisyNet-DQN

Input : Env Environment; N_f maximum length of list x; ε set of random variables of the network
Input : B empty replay buffer; ζ initial network parameters; ζ^- initial target network parameters
Input : N_B replay buffer size; N_T training batch size; N^- target network replacement frequency
Output : $Q(., \varepsilon; \zeta)$ action-value function

```
1  for episode e ∈ {1, ..., M} do
       /* x is a list of states                                              */
2      x ← [ ]
3      Initialise state sequence x_0 ∼ Env
4      x[0] ← x_0
5      for t ∈ {1, ...} do
           /* l[−1] is the last element of the list l                        */
6          Set x ← x[−1]
7          Sample a noisy network ξ ∼ ε
8          Select an action a ← argmax_{b∈A} Q(x, b, ξ; ζ^−)
9          Sample next state y ∼ P(·|x, a), receive reward r ← R(x, a) and append x: x[−1] ← y
10         if |x| > N_f then
11             Delete oldest element from x
12         end
13         Add transition (x, a, r, y) to the replay buffer B[−1] ← (x, a, r, y)
14         if |B| > N_B then
15             Delete oldest transition from B
16         end
           /* D is a distribution over the replay, it can be uniform or
              implementing prioritised replay                                */
17         Sample a minibatch of N_T transitions ((x_j, a_j, r_j, y_j) ∼ D)_{j=1}^{N_T}
           /* Construction of the target values.                             */
18         for j ∈ {1, ..., N_T} do
19             Sample a noisy network ξ_j ∼ ε
20             Sample a noisy target network ξ'_j ∼ ε
21             if y is a terminal state then
22                 y_j ← r_j
23             else
24                 y_j ← r_j + max_{b∈A} Q(y_j, b, ξ'_j; ζ^−)
25             Do a gradient step with loss (y_j − Q(x_j, a_j, ξ_j; ζ))^2
26         end
27         if t ≡ 0 (mod N^−) then
28             Update the target network: ζ^− ← ζ
29         end
30     end
31 end
```

C.2 NoisyNet-A3C

Algorithm 2: NoisyNet-A3C for each actor-learner thread

Input : Environment Env, Global shared parameters (ζ_π, ζ_V), global shared counter T and maximal time T_{max}.
Input : Thread-specific parameters (ζ'_π, ζ'_V), Set of random variables ε, thread-specific counter t and roll-out size t_{max}.
Output : $\pi(\cdot; \zeta, \varepsilon)$ the policy and $V(\cdot; \zeta_V, \varepsilon)$ the value.

```
1  Initial thread counter t ← 1
2  repeat
3  |   Reset cumulative gradients: dζ_π ← 0 and dζ_V ← 0.
4  |   Synchronise thread-specific parameters: ζ'_π ← ζ_π and ζ'_V ← ζ_V.
5  |   counter ← 0.
6  |   Get state x_t from Env
7  |   Choice of the noise: ξ ∼ ε
   |   /* r is a list of rewards                                   */
8  |   r ← [ ]
   |   /* a is a list of actions                                   */
9  |   a ← [ ]
   |   /* x is a list of states                                    */
10 |   x ← [ ] and x[0] ← x_t
11 |   repeat
12 |   |   Policy choice: a_t ∼ π(·|x_t; ζ_π; ξ)
13 |   |   a[−1] ← a_t
14 |   |   Receive reward r_t and new state x_{t+1}
15 |   |   r[−1] ← r_t and x[−1] ← x_{t+1}
16 |   |   t ← t + 1 and T ← T + 1
17 |   |   counter = counter + 1
18 |   until x_t terminal or counter == t_max + 1
19 |   if x_t is a terminal state then
20 |   |   Q = 0
21 |   else
22 |   |   Q = V(x_t; ζ_V, ξ)
23 |   for i ∈ {counter − 1, ..., 0} do
24 |   |   Update Q: Q ← r[i] + γQ.
25 |   |   Accumulate policy-gradient: dζ_π ← dζ_π + ∇_{ζ'_π} log(π(a[i]|x[i]; ζ'_π, ξ))[Q − V(x[i]; ζ'_V, ξ)].
26 |   |   Accumulate value-gradient: dζ_V ← dζ_V + ∇_{ζ'_V} [Q − V(x[i]; ζ'_V, ξ)]^2.
27 |   end
28 |   Perform asynchronous update of ζ_π: ζ_π ← ζ_π + α_π dζ_π
29 |   Perform asynchronous update of ζ_V: ζ_V ← ζ_V − α_V dζ_V
30 until T > T_max
```

Game	Human	Random	DQN	NoisyNet-DQN	Dueling	NoisyNet-Dueling	A3C	NoisyNet-A3C
alien	**7128**	228	1860	2899	5899	4460	2148	2809
amidar	1720	6	783	1534	1690	**2215**	814	819
assault	742	222	1439	3126	2792	3876	3434	**7502**
asterix	8503	210	3807	11037	23091	32427	8756	**34011**
asteroids	**47389**	719	1336	2101	1506	3166	2576	26380
atlantis	29028	12580	60001	7910700	8546500	**8782433**	415000	491000
bank heist	753	14	235	646	656	1198	**1252**	999
battle zone	37188	2360	18810	11925	37763	**42767**	16146	22546
beam rider	16926	364	6850	14689	16397	**17092**	7746	10783
berzerk	**2630**	124	497	972	1168	2277	994	1032
bowling	**161**	23	47	49	71	32	34	60
boxing	12	0	78	89	94	**96**	92	89
breakout	30	2	240	382	428	421	**451**	422
centipede	**12017**	2091	3131	3352	5759	6276	5772	8704
chopper command	7388	811	3628	5343	5333	5565	4473	**10389**
crazy climber	35829	10780	89944	121229	**133065**	126772	127750	130400
defender	18689	2874	7923	19880	24725	32042	48604	**48883**
demon attack	1971	152	8142	26730	29101	**77881**	32772	42168
double dunk	-16	-19	-11	2	**5**	1	3	0
enduro	860	0	473	909	**1679**	1483	0	0
fishing derby	-39	-92	-14	-6	11	**19**	-22	16
freeway	30	0	19	32	**33**	33	0	33
frostbite	**4335**	65	418	355	986	1338	283	276
gopher	2412	258	6149	10516	22970	**24725**	7354	9271
gravitar	**3351**	173	258	255	380	324	405	343
hero	**30826**	1027	14706	9704	21722	16127	26829	24511
ice hockey	**1**	-11	-6	-5	-0	-2	-2	0
jamesbond	303	29	574	1061	1339	584	532	**4214**
kangaroo	3035	52	5088	**14763**	14005	14678	131	6967
krull	2666	1598	6263	7096	9264	9192	**9853**	9475
kung fu master	22736	258	16240	27324	35985	42526	**44625**	42683
montezuma	**4753**	0	9	5	1	5	15	10
ms pacman	**6952**	307	2372	2786	3101	4209	2445	2862
name this game	8049	2292	6183	7153	10770	**11603**	7655	9242
phoenix	7243	761	6798	12540	17932	17073	10844	**36839**
pitfall	**6464**	-229	0	-2	0	0	0	0
pong	15	-21	18	**21**	21	**21**	**21**	**21**
private eye	**69571**	25	1313	2015	173	911	1942	282
qbert	13455	164	7700	11797	15028	15662	**17971**	17169
riverraid	**17118**	1338	5717	8766	14081	14855	8355	8125
road runner	7845	12	25650	34459	47524	**55497**	39073	10298
robotank	12	2	47	54	**58**	54	7	45
seaquest	**42055**	68	2881	3302	12534	9941	1742	927
skiing	**-4337**	-17098	-14779	-17871	-11425	-14835	-15115	-8846
solaris	**12327**	1263	2522	2913	2177	2360	11830	9173
space invaders	1669	148	1081	1524	2516	**4490**	1008	1378
star gunner	10250	664	24582	28861	51787	**58290**	50933	45612
surround	**6**	-10	-9	-6	1	3	-8	-7
tennis	-8	-24	**9**	0	8	0	0	0
time pilot	5229	3568	4392	6116	10671	11192	8993	**11560**
tutankham	168	11	141	135	264	264	**321**	165
up n down	11693	533	5209	9364	14569	18234	90290	**145113**
venture	**1188**	0	586	8	8	51	0	17
video pinball	17668	16257	88092	152022	**768419**	548689	140863	630390
wizard of wor	4756	564	2045	7576	7715	10605	9858	**11852**
yars revenge	54577	3093	14027	16437	18917	21738	14715	**69408**
zaxxon	9173	32	4218	9703	12609	9954	**15567**	1367

Table 2: Raw scores across all games with random starts.

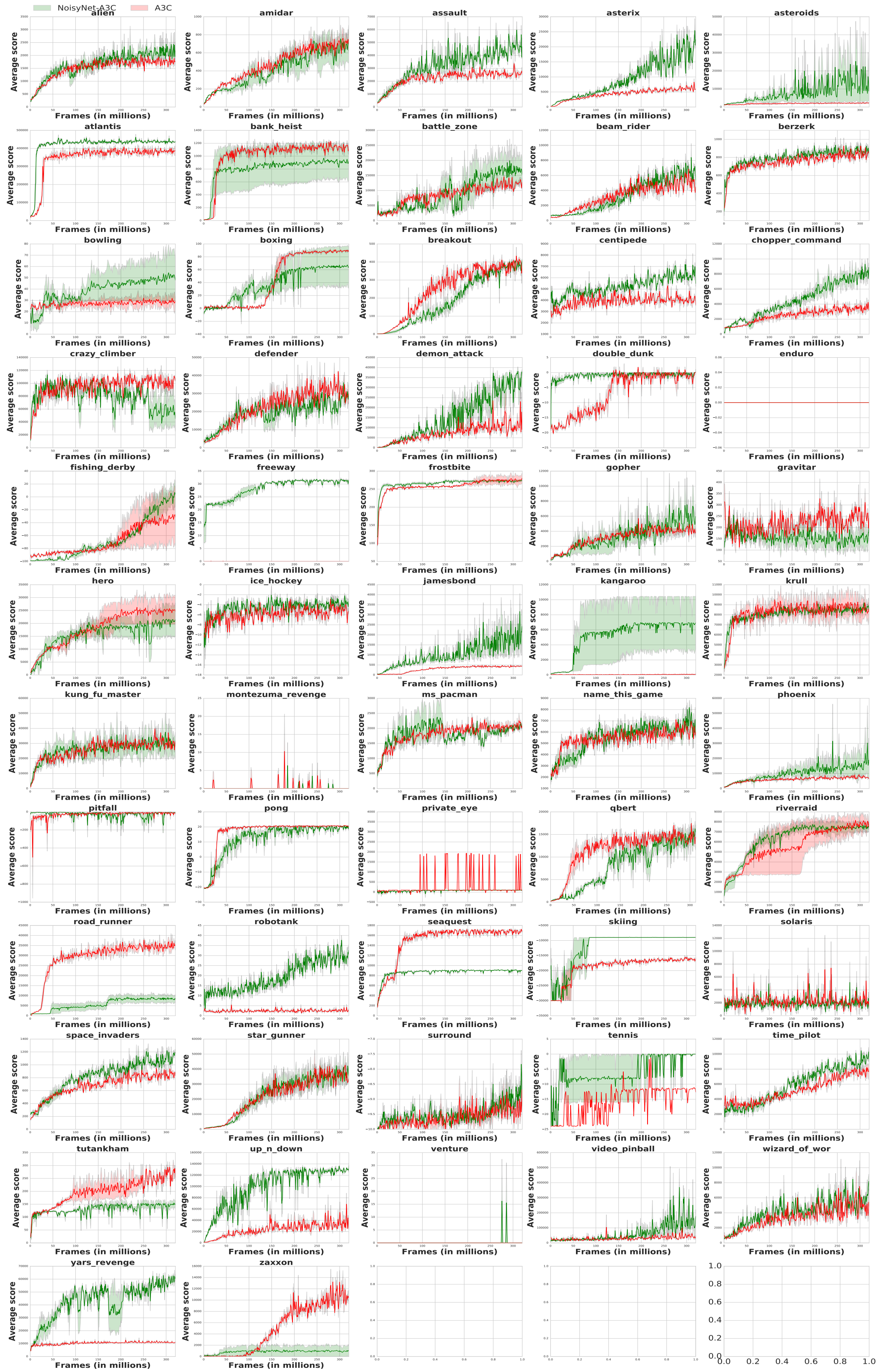


Figure 3: Training curves for all Atari games comparing A3C and NoisyNet-A3C.

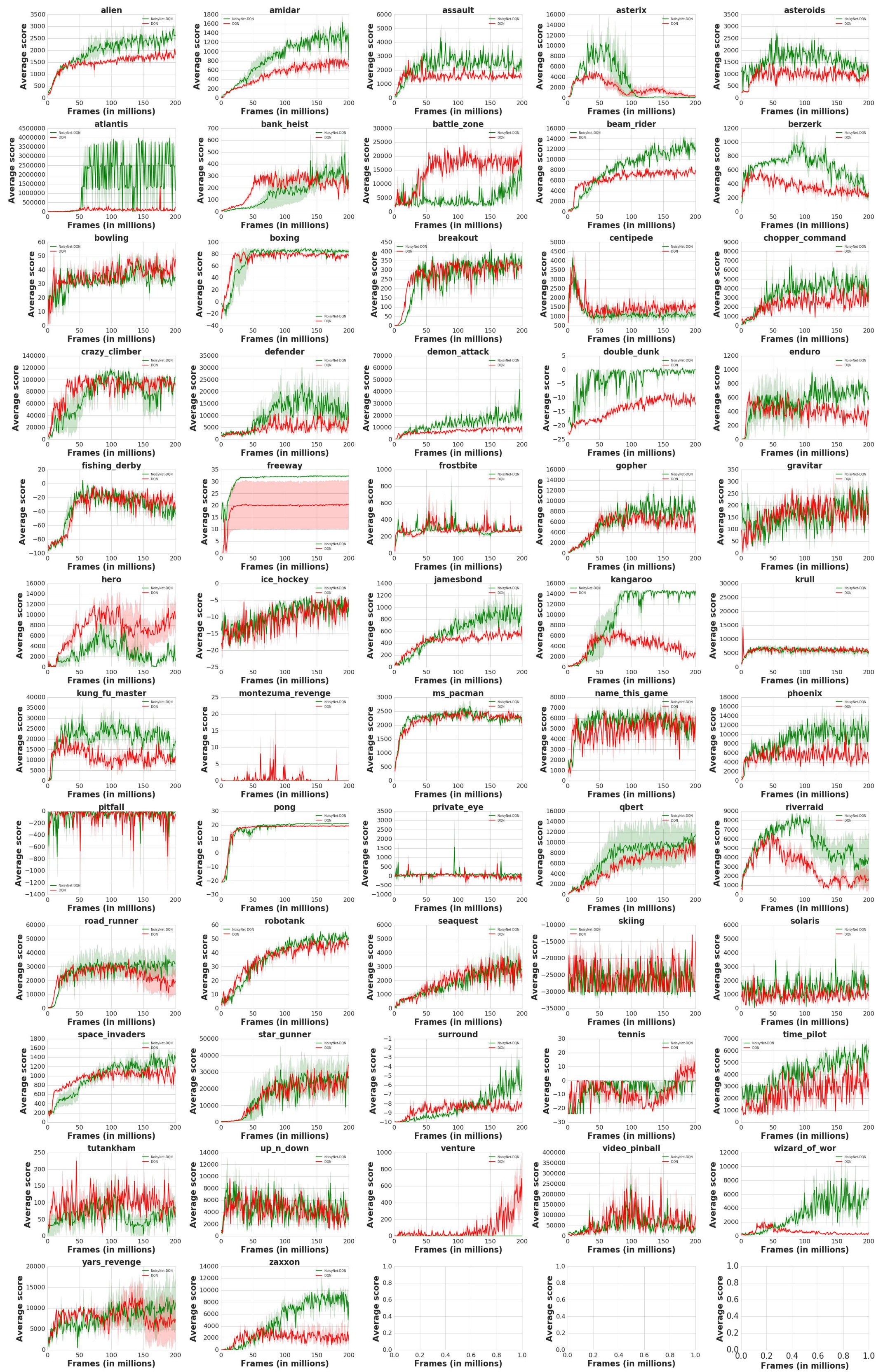


Figure 4: Training curves for all Atari games comparing DQN and NoisyNet-DQN.

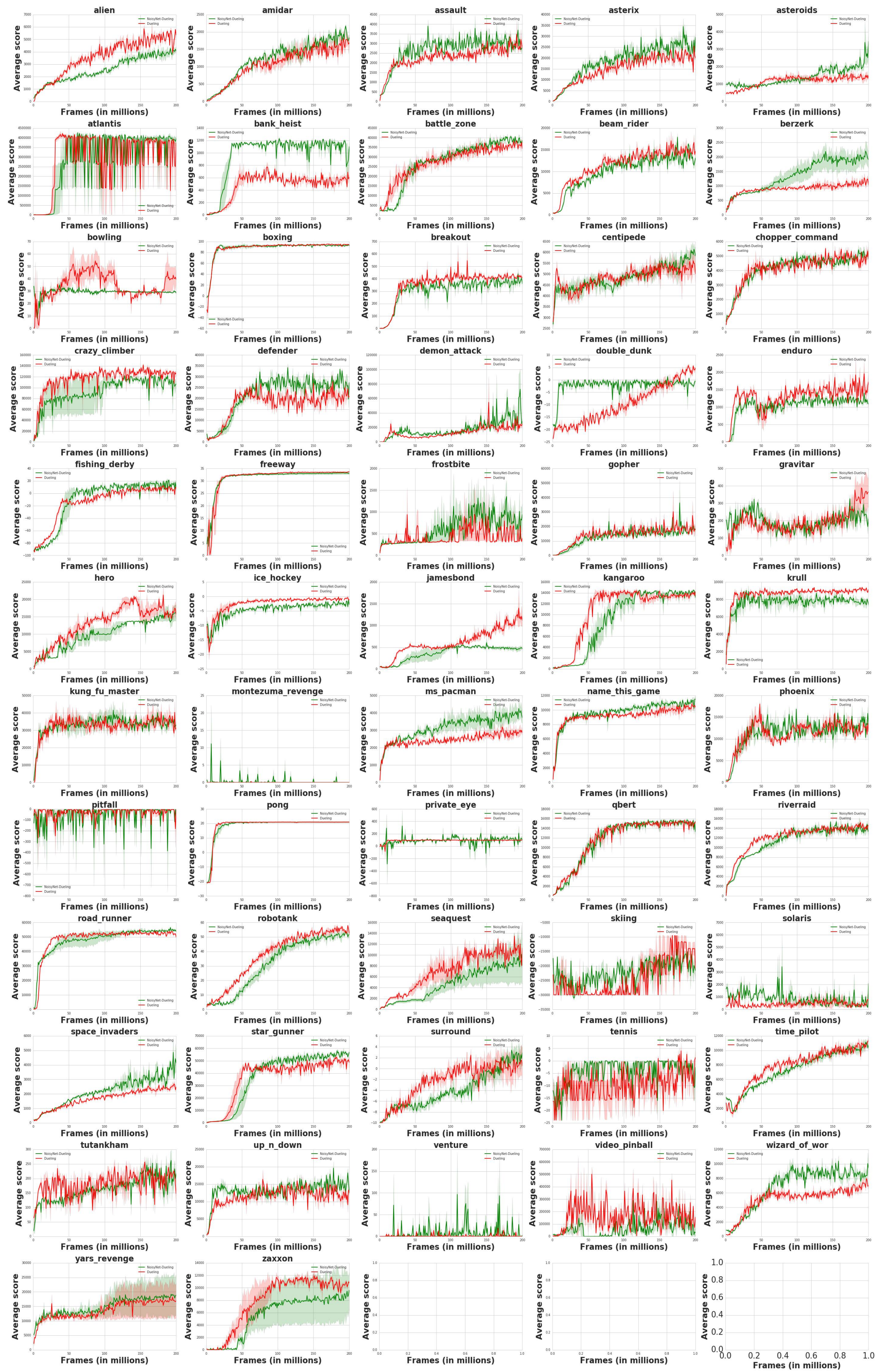


Figure 5: Training curves for all Atari games comparing Duelling and NoisyNet-Dueling.