# NOISY NETWORKS FOR EXPLORATION

**Anonymous authors**
Paper under double-blind review

## 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 $\epsilon$-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) (Kearns & Singh, 2002; Jaksch et al., 2010), most exploration heuristics rely on random perturbations of the agent's policy, such as $\epsilon$-greedy (Sutton & Barto, 1998) or entropy regularisation (Williams, 1992), 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 (Osband et al., 2017).

*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 (Azar et al., 2017; Lattimore et al., 2013; Jaksch et al., 2010; Auer & Ortner, 2007; Kearns & Singh, 2002). 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 (except from work by (Geist & Pietquin, 2010a;b) but it doesn't come with convergence guarantees). A more structured approach to exploration is to augment the environment's reward signal with an additional *intrinsic motivation* term (Singh et al., 2004) that explicitly rewards novel discoveries. Many such terms have been proposed, including learning progress (Oudeyer & Kaplan, 2007), compression progress (Schmidhuber, 2010), variational information maximisation (Houthooft et al., 2016) and prediction gain (Bellemare et al., 2016). 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 (Ostrovski et al., 2017). Exploration in the policy space itself, for example, with evolutionary or black box algorithms (Moriarty et al., 1999; Fix & Geist, 2012; Salimans et al., 2017), usually requires many prolonged interactions with the environment. Although these algorithms are quite 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 $\epsilon$-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 (Hinton & Van Camp, 1993; Bishop, 1995) and flat minima search (Hochreiter & Schmidhuber, 1997) 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 (Graves, 2011; Blundell et al., 2015) or Thompson sampling (Thompson, 1933; Lipton et al., 2016) 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 exploration via randomised value functions can be provably-efficient with suitable linear basis (Osband et al., 2014). Previous attempts to extend this approach to deep neural networks required many duplicates of sections of the network (Osband et al., 2016). 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 (Mnih et al., 2016). Most recently (and independently of our work), Plappert et al. (Plappert et al., 2017) 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. We need to emphasise that the idea of injecting noise to improve the optimisation process have been thoroughly studied in the literature of supervised learning and optimisation under different names (e.g., Neural deffusion process (Mobahi, 2016) and graduated optimisation (Hazan et al., 2016)). Though these methods often rely on a non-trainable noise of vanishing size as opposed to NoisyNet which tunes the parameter of noise by gradient descent.

NoisyNet can also be adapted to any deep RL algorithm and we demonstrate this versatility by providing NoisyNet versions of DQN (Mnih et al., 2015), Dueling (Wang et al., 2016) and A3C (Mnih et al., 2016) algorithms. Experiments on 57 Atari games show that NoisyNet-DQN and NoisyNet-Dueling achieve striking gains when compared to the baseline algorithms without significant extra computational cost, and with less hyper parameters to tune. Also the noisy version of A3C provides some improvement over the baseline.

## 2 BACKGROUND

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

### 2.1 MARKOV DECISION PROCESSES AND REINFORCEMENT LEARNING

MDPs model stochastic, discrete-time and finite action space control problems (Bellman & Kalaba, 1965; Bertsekas, 1995; Puterman, 1994). 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 $\pi$ 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 $\pi$ is assessed by the action-value function $Q^\pi$ 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 $\pi$ starting from $x_0 = x$ and $a_0 = a$. Therefore, the quantity $Q^\pi(x, a)$ represents the expected $\gamma$-discounted cumulative reward collected by executing the policy $\pi$ 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 $V^\pi$ for a policy is defined as $V^\pi(x) = \mathbb{E}_{a \sim \pi(\cdot|x)}[Q^\pi(x, a)]$, and represents the expected $\gamma$-discounted return collected by executing the policy $\pi$ 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) (Mnih et al., 2015), Dueling networks (Wang et al., 2016), Asynchronous

Advantage Actor-Critic (A3C) (Mnih et al., 2016), Trust Region Policy Optimisation (Schulman et al., 2015), Deep Deterministic Policy Gradient (Lillicrap et al., 2015) and distributional RL (c51) (Bellemare et al., 2017) are examples of such algorithms. They frame the RL problem as the minimisation of a loss function $L(\theta)$, where $\theta$ represents the parameters of the network. In our experiments we shall consider the DQN and A3C algorithms.

DQN (Mnih et al., 2015) 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 $\theta$:

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

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 $\theta^-$ represents the parameters of a fixed and separate target network which is updated ($\theta^- \leftarrow \theta$) regularly to stabilise the learning. An $\epsilon$-greedy policy is used to pick actions greedily according to the action-value function $Q$ or, with probability $\epsilon$, a random action is taken.

The Dueling DQN (Wang et al., 2016) is an extension of the DQN architecture. The main difference is in using Dueling network architecture as opposed to the Q network in DQN. Dueling network estimates the action-value function using two parallel sub-networks, the value and advantage sub-network, sharing a convolutional layer. Let $\theta_{\text{conv}}$, $\theta_V$, and $\theta_A$ be, respectively, the parameters of the convolutional encoder $f$, of the value network $V$, and of the advantage network $A$; and $\theta = \{\theta_{\text{conv}}, \theta_V, \theta_A\}$ is their concatenation. The output of these two networks are combined as follows for every $(x, a) \in \mathcal{X} \times \mathcal{A}$:

$$Q(x, a; \theta) = V(f(x; \theta_{\text{conv}}), \theta_V) + A(f(x; \theta_{\text{conv}}); \theta_A) - \frac{\sum_b A(f(x; \theta_{\text{conv}}), b; \theta_A)}{N_{\text{actions}}}. \tag{3}$$

The Dueling algorithm then makes use of the the double-DQN update rule (van Hasselt et al., 2016) to optimise $\theta$:

$$L(\theta) = \mathbb{E}_{(x,a,r,y)\sim D}\left[\left(r + \gamma Q(y, b^*(y); \theta^-) - Q(x, a; \theta)\right)^2\right], \tag{4}$$

$$\text{s.t.} \qquad b^*(y) = \arg\max_{b\in\mathcal{A}} Q(y, b; \theta), \tag{5}$$

where the definition distribution $D$ and the target network parameter set $\theta^-$ is identical to DQN.

In contrast to DQN and Dueling, A3C (Mnih et al., 2016) is a policy gradient algorithm. A3C's network directly learns a policy $\pi$ 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:

$$\nabla_\theta L^\pi(\theta) = -\mathbb{E}^\pi\left[\sum_{i=0}^k \nabla_\theta \log(\pi(a_{t+i}|x_{t+i}; \theta))A(x_{t+i}, a_{t+i}; \theta) + \beta \sum_{i=0}^k \nabla_\theta H(\pi(\cdot|x_{t+i}; \theta))\right]. \tag{6}$$

$H[\pi(\cdot|x_t; \theta)]$ denotes the entropy of the policy $\pi$ and $\beta$ 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: $A(x_{t+i}, a_{t+i}; \theta) = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta) - V(x_{t+i}; \theta)$, $r_{t+j}$ 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 $\pi$ starting in state $x_{t+i}$: $\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta)$. The overall A3C loss is then $L(\theta) = L^\pi(\theta) + \lambda L^V(\theta)$ where $\lambda$ 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 $\theta$ which takes the input $x$ and outputs $y$. We represent the noisy parameters $\theta$ as $\theta \stackrel{\text{def}}{=} \mu + \Sigma \odot \varepsilon$, where $\zeta \stackrel{\text{def}}{=} (\mu, \Sigma)$ is a set of vectors of learnable parameters, $\varepsilon$ 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 $\varepsilon$: $\bar{L}(\zeta) \stackrel{\text{def}}{=} \mathbb{E}\left[L(\theta)\right]$. Optimisation now occurs with respect to the set of parameters $\zeta$.

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

$$y = wx + b, \tag{7}$$

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:

$$y \stackrel{\text{def}}{=} (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + \sigma^b \odot \varepsilon^b, \tag{8}$$

where $\mu^w + \sigma^w \odot \varepsilon^w$ and $\mu^b + \sigma^b \odot \varepsilon^b$ replace $w$ and $b$ in Eq. (7), 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 $\varepsilon^b_j$) of the random matrix $\varepsilon^w$ (respectively of the random vector $\varepsilon^b$) 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 $\varepsilon_i$ for noise of the inputs and and $q$ unit Gaussian variables $\varepsilon_j$ for noise of the outputs (thus $p + q$ unit Gaussian variables in total). Each $\varepsilon^w_{i,j}$ and $\varepsilon^b_j$ can then be written as:

$$\varepsilon^w_{i,j} = f(\varepsilon_i)f(\varepsilon_j), \tag{9}$$

$$\varepsilon^b_j = f(\varepsilon_j), \tag{10}$$

where $f$ is a real-valued function. 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:

$$\nabla\bar{L}(\zeta) = \nabla\mathbb{E}\left[L(\theta)\right] = \mathbb{E}\left[\nabla_{\mu,\Sigma}L(\mu + \Sigma \odot \varepsilon)\right]. \tag{11}$$

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

$$\nabla\bar{L}(\zeta) \approx \nabla_{\mu,\Sigma}L(\mu + \Sigma \odot \xi). \tag{12}$$

### 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) and Dueling.** We apply the following modifications to both DQN and Dueling: first, $\varepsilon$-greedy is no longer used, but instead the policy greedily optimises the (randomized) action-value function. Secondly, the fully connected layers of the value network 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 across the batch. Since DQN and Dueling take one step of optimisation for every action step, the noisy network parameters are re-sampled before every action. We call the new adaptations of DQN and Dueling, NoisyNet-DQN and NoisyNet-Dueling, respectively.

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:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b \in A} Q(y, b, \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right], \tag{13}$$

where the outer expectation is with respect to distribution of the noise variables $\varepsilon$ for the noisy value function $Q(x, a, \varepsilon; \zeta)$ and the noise variable $\varepsilon'$ for 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 online network. Concerning the action choice, we sample from the online network and we act greedily with respect to the output action-value function. The algorithm is provided in Sec. C.1.

Similarly the loss function for NoisyNet-Dueling is defined as:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma Q(y, b^*(y), \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right] \tag{14}$$

$$\text{s.t.} \qquad b^*(y) = \arg\max_{b \in \mathcal{A}} Q(y, b^*(y), \varepsilon'; \zeta). \tag{15}$$

The algorithm is provided in Sec. C.2.

**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 $\epsilon$-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.

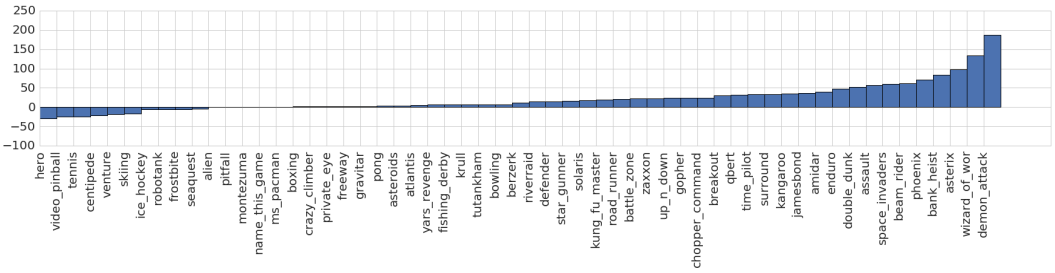
Indeed, when replacing the linear layers by noisy linear layers (the parameters of the noisy network are now noted $\zeta$), we obtain the following estimation of the return via a roll-out of size $k$:

$$\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \zeta, \varepsilon_i). \tag{16}$$

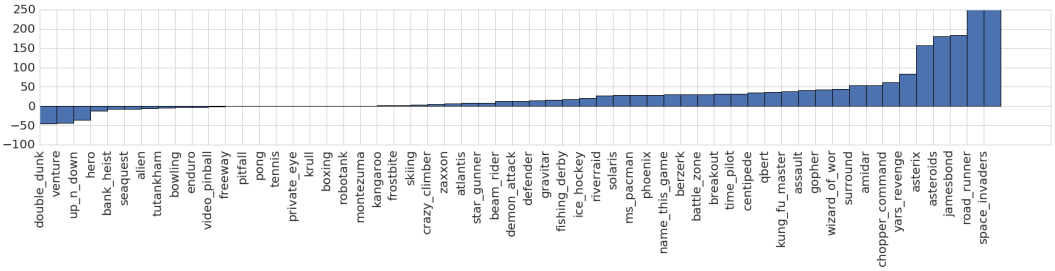
As A3C is an on-policy algorithm the gradients are unbiased when noise of the network is consistent for the whole roll-out. Consistency among action value functions $\hat{Q}_i$ is ensured by letting letting the noise be the *same* throughout each rollout, i.e., $\forall i, \varepsilon_i = \varepsilon$. Additional details are provided in the Appendix Sec. A.2 and the algorithm is given in Sec. C.3.

## 4 Results

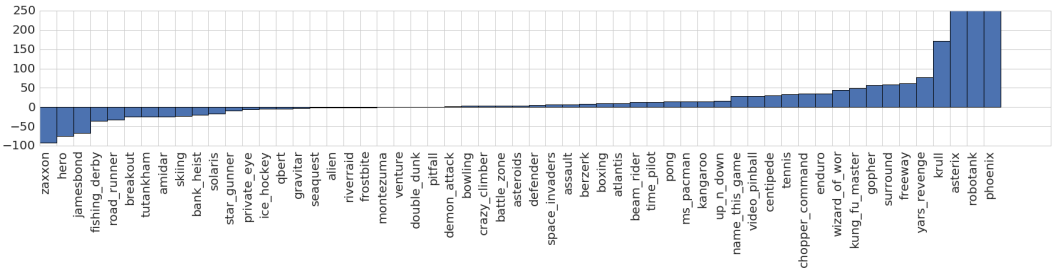
We evaluated the performance of noisy network agents on 57 Atari games (Bellemare et al., 2015) and compared to baselines that, without noisy networks, rely upon the original exploration methods




(a) Improvement in percentage of NoisyNet-DQN over DQN (Mnih et al., 2015)




(b) Improvement in percentage of NoisyNet-Dueling over Dueling (Wang et al., 2016)




(c) Improvement in percentage of NoisyNet-A3C over A3C (Mnih et al., 2016)

Figure 1: Comparison of NoisyNet agent versus the baseline according to Eq. (18). The maximum score is truncated at 250%.

($\varepsilon$-greedy and entropy bonus). We used the random start no-ops scheme for training and evaluation as described the original DQN paper (Mnih et al., 2015). The mode of evaluation is identical to those of Mnih et al. (2016) where randomised restarts of the games are used for evaluation after training has happened. The raw average scores of the agents are evaluated during training, every 1M frames in the environment, by suspending learning and evaluating the latest agent for 500K frames. Episodes are truncated at 108K frames (or 30 minutes of simulated play) (van Hasselt et al., 2016).

We consider three baseline agents: DQN (Mnih et al., 2015), duel clip variant of Dueling algorithm (Wang et al., 2016) and A3C (Mnih et al., 2016). 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 used the same hyper parameters as in the respective original paper for the baseline.

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

$$100 \times \frac{\text{Score}_{\text{agent}} - \text{Score}_{\text{Random}}}{\text{Score}_{\text{Human}} - \text{Score}_{\text{Random}}}, \tag{17}$$

where human and random scores are the same as those in Wang et al. (2016). Note that the human normalised score is zero for a random agent and 100 for human level performance. Per-game maximum scores are computed by taking the maximum raw scores of the agent and then averaging over three seeds. However, for computing the human normalised scores in figure 2, the raw scores

are evaluated every 1M frames and averaged over three seeds. The overall agent performance is measured by both mean and median of the human normalised score across all 57 Atari games.

The aggregated results across all 57 Atari games are reprted 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 18 (in the case of A3C) and at most 48 (in the case of DQN) percentage points to the median human normalised score. The mean human normalised score is also significantly improved for all agents. Interestingly the Dueling case, which relies on multiple modifications of DQN, 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

| | Baseline | | NoisyNet | | Improvement |
|---|---|---|---|---|---|
| | Mean | Median | Mean | Median | (On median) |
| DQN | 319 | 83 | **379** | **123** | 48 |
| Dueling | 524 | 132 | **633** | **172** | 30 |
| A3C | 293 | 80 | **347** | **94** | 18 |

Table 1: Comparison between the baseline DQN, Dueling and A3C and their NoisyNet version in terms of median and mean human-normalised scores defined in Eq. (17). We report on the last column the percentage improvement on the baseline in terms of median human-normalised score.

without noisy networks:

$$100 \times \frac{\text{Score}_{\text{NoisyNet}} - \text{Score}_{\text{Baseline}}}{\max(\text{Score}_{\text{Human}}, \text{Score}_{\text{Baseline}}) - \text{Score}_{\text{Random}}}. \tag{18}$$

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. In some 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 6, 4 and 5, for A3C, DQN and Dueling respectively.

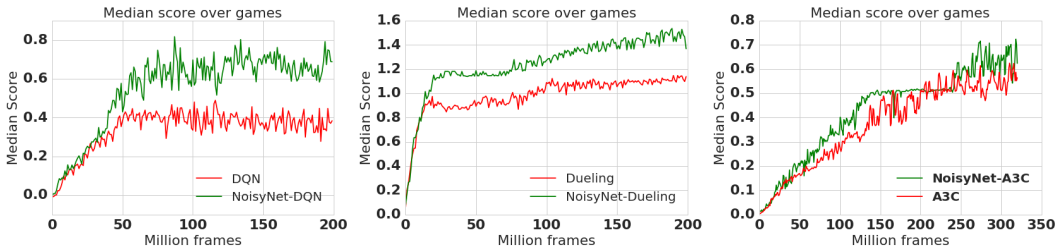



Figure 2: Comparison of NoisyNet agent versus the baseline according to the median human normalised score.

### 4.1 Analysis of Learning in Noisy Layers

In this subsection, we try to provide some insight on how noisy networks affect the learning process and the exploratory behaviour of the agent. In particular, we focus on analysing the evolution of the noise weights $\sigma^w$ and $\sigma^b$ throughout the learning process. We first note that, as $L(\zeta)$ is a positive and continuous function of $\zeta$, there always exists a *deterministic* optimiser for the loss $L(\zeta)$ (defined in Eq. (13)). Therefore, one may expect that, to obtain the deterministic optimal solution, the neural network may learn to discard the noise entries by eventually pushing $\sigma^w$s and $\sigma^b$ towards 0.

To test this hypothesis we track the changes in $\sigma^w$s throughout the learning process. Let $\sigma_i^w$ denote the $i^{\text{th}}$ weight of a noisy layer. We then define $\bar{\Sigma}$, the mean-absolute of the $\sigma_i^w$s of a noisy layer, as

$$\bar{\Sigma} = \frac{1}{\mathrm{N}_{\text{weights}}} \sum_i |\sigma_i^w|. \tag{19}$$

Intuitively speaking $\bar{\Sigma}$ provides some measure of the stochasticness of the Noisy layers. We report the learning curves of $\bar{\Sigma}$ in Fig. 3 for a selection of Atari games in NoisyNet-DQN agent. We observe that $\bar{\Sigma}$ of the last layer of the network decreases as the learning proceeds in all cases, whereas in the case of the penultimate layer this only happens for 2 games out of 5 (Pong and Beam rider) and in the remaining 3 games $\bar{\Sigma}$ in fact increases. This shows that in the case of NoisyNet-DQN the agent does not necessarily evolve towards a deterministic solution as one might have expected. Another interesting observation is that the way $\bar{\Sigma}$ evolves significantly differs from one game to another. This suggests that NoisyNet produce a problem-specific exploration strategy as opposed to fixed exploration strategy used in standard DQN.

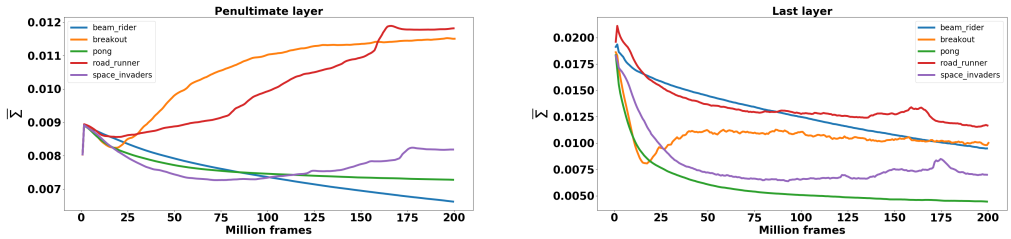



Figure 3: Comparison of the learning curves of the average noise parameter $\bar{\Sigma}$ across five Atari games. The results are averaged across 3 seeds.

## 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, Astroid and Freeway that the standard DQN, Dueling and A3C perform poorly compared with the human player, NoisyNet-DQN and NoisyNet-A3C achieve super human performance. Our method eliminates the need for $\epsilon$-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. Another advantage of NoisyNet is that the amount of noise injected in the network is tuned automatically by the RL algorithm. This alleviates the need for any hyper parameter tuning (required with standard entropy bonus and $\epsilon$-greedy types of exploration). This is also 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 (Fortunato et al., 2017) 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 (Lillicrap et al., 2015), TRPO (Schulman et al., 2015) or distributional RL (c51) (Bellemare et al., 2017). As such we believe this work is a step towards the goal of developing a universal exploration strategy.

## REFERENCES


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

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

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.

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*, pp. 1471–1479, 2016.

Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pp. 449–458, 2017.

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

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

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

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

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

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

Matthieu Geist and Olivier Pietquin. Kalman temporal differences. *Journal of artificial intelligence research*, 39:483–532, 2010a.

Matthieu Geist and Olivier Pietquin. Managing uncertainty within value function approximation in reinforcement learning. In *Active Learning and Experimental Design workshop (collocated with AISTATS 2010), Sardinia, Italy*, volume 92, 2010b.

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

Elad Hazan, Kfir Yehuda Levy, and Shai Shalev-Shwartz. On graduated optimization for stochastic non-convex problems. In *International Conference on Machine Learning*, pp. 1833–1841, 2016.

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*, pp. 5–13. ACM, 1993.

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

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*, pp. 1109–1117, 2016.

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

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

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

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.

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.

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.

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*, pp. 1928–1937, 2016.

Hossein Mobahi. Training recurrent neural networks by diffusion. *arXiv preprint arXiv:1601.04114*, 2016.

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

Ian Osband, Benjamin Van Roy, and Zheng Wen. Generalization and exploration via randomized value functions. *arXiv preprint arXiv:1402.0635*, 2014.

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

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

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.

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

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.

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

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

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

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

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

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

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, pp. 1057–1063, 1999.

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.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proc. of AAAI*, pp. 2094–2100, 2016.

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*, pp. 1995–2003, 2016.

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 (Mnih et al., 2015) is an approximate value iteration technique that uses a deep convolutional neural network to represent the Q-function as $Q(x, a; \theta)$, where $\theta$ are the parameters of the network. We used the network architecture from (Mnih et al., 2015) and also the same set up for interacting with the Atari games. We note, however, that we did not use $\epsilon$-greedy technique, instead relied solely upon NoisyNet for exploration. In DQN, the neural network is trained by minimising the following loss:

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

where $D$ is a distribution over transitions $e = (x, a, r = R(x, a), y \sim P(\cdot|x, a))$. Here $\theta^-$ 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 $\zeta$) 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:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b \in A} Q(y, b, \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right]. \tag{20}$$

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

## A.2 NOISYNET-A3C IMPLEMENTATION DETAILS

In contrast with value-based algorithms, policy-based methods such as A3C (Mnih et al., 2016) parameterise the policy $\pi(a|x; \theta_\pi)$ directly and update the parameters $\theta_\pi$ 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) (Sutton et al., 1999). A3C uses a deep neural network with weights $\theta = \theta_\pi \cup \theta_V$ to parameterise the policy $\pi$ 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 $\theta_\pi$ (resp. $\theta_V$) 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 $\epsilon$-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.3 or to the original A3C paper (Mnih et al., 2016). In order to train the policy-head, an approximation of the policy-gradient is computed for each state of the roll-out $(x_{t+i}, a_{t+i} \sim \pi(\cdot|x_{t+i}; \theta_\pi), r_{t+i})_{i=0}^k$:

$$\nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i}; \theta_\pi))[\hat{Q}_i - V(x_{t+i}; \theta_V)], \tag{21}$$

where $\hat{Q}_i$ is an estimation of the return $\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \theta_V)$. The gradients are then added to obtain the cumulative gradient of the roll-out:

$$\sum_{i=0}^{k} \nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i}; \theta_\pi))[\hat{Q}_i - V(x_{t+i}; \theta_V)]. \tag{22}$$

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 $(\theta_\pi, \theta_V)$ are updated after each

roll-out as follows:

$$\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \sum_{i=0}^{k} \nabla_{\theta_\pi} \log(\pi(a_{t+i}|x_{t+i};\theta_\pi))[\hat{Q}_i - V(x_{t+i};\theta_V)], \tag{23}$$

$$\theta_V \leftarrow \theta_V - \alpha_V \sum_{i=0}^{k} \nabla_{\theta_V} [\hat{Q}_i - V(x_{t+i};\theta_V)]^2, \tag{24}$$

where $(\alpha_\pi, \alpha_V)$ 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 $\zeta_\pi$ and $\zeta_V$), we obtain the following estimation of the return via a roll-out of size $k$:

$$\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k};\zeta_V,\varepsilon_i). \tag{25}$$

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 $(\zeta_\pi, \zeta_V)$ is done after each roll-out with the noise of the whole network held fixed for the duration of the roll-out:

$$\zeta_\pi \leftarrow \zeta_\pi + \alpha_\pi \sum_{i=0}^{k} \nabla_{\zeta_\pi} \log(\pi(a_{t+i}|x_{t+i};\zeta_\pi,\varepsilon))[\hat{Q}_i - V(x_{t+i};\zeta_V,\varepsilon)], \tag{26}$$

$$\zeta_V \leftarrow \zeta_V - \alpha_V \sum_{i=0}^{k} \nabla_{\zeta_V} [\hat{Q}_i - V(x_{t+i};\zeta_V,\varepsilon)]^2. \tag{27}$$

## B INITIALISATION OF NOISY NETWORKS

In the case of an unfactorised noisy networks, the parameters $\mu$ and $\sigma$ 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 $\sigma_0$ is set to $0.5$.

# C ALGORITHMS

## C.1 NOISYNET-DQN

**Algorithm 1:** NoisyNet-DQN

**Input** : $Env$ Environment; $N_f$ maximum length of list $x$; $\varepsilon$ set of random variables of the network
**Input** : $B$ empty replay buffer; $\zeta$ initial network parameters; $\zeta^-$ 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

**for** *episode* $e \in \{1, \dots, M\}$ **do**
  /* $x$ is a list of states */
  $x \leftarrow [\,]$
  Initialise state sequence $x_0 \sim Env$
  $x[0] \leftarrow x_0$
  **for** $t \in \{1, \dots\}$ **do**
    /* $l[-1]$ is the last element of the list $l$ */
    Set $x \leftarrow x[-1]$
    Sample a noisy network $\xi \sim \varepsilon$
    Select an action $a \leftarrow \text{argmax}_{b \in A} Q(x, b, \xi; \zeta)$
    Sample next state $y \sim P(\cdot|x, a)$, receive reward $r \leftarrow R(x, a)$ and append $x$: $x[-1] \leftarrow y$
    **if** $|x| > N_f$ **then**
      Delete oldest element from $x$
    **end**
    Add transition $(x, a, r, y)$ to the replay buffer $B[-1] \leftarrow (x, a, r, y)$
    **if** $|B| > N_B$ **then**
      Delete oldest transition from $B$
    **end**
    /* $D$ is a distribution over the replay, it can be uniform or implementing prioritised replay */
    Sample a minibatch of $N_T$ transitions $((x_j, a_j, r_j, y_j) \sim D)_{j=1}^{N_T}$
    /* Construction of the target values. */
    **for** $j \in \{1, \dots, N_T\}$ **do**
      Sample a noisy network $\xi_j \sim \varepsilon$
      Sample a noisy target network $\xi'_j \sim \varepsilon$
      **if** $y_j$ *is a terminal state* **then**
        $\widehat{Q} \leftarrow r_j$
      **else**
        $\widehat{Q} \leftarrow r_j + \gamma \max_{b \in A} Q(y_j, b, \xi'_j; \zeta^-)$
      Do a gradient step with loss $(\widehat{Q} - Q(x_j, a_j, \xi_j; \zeta))^2$
    **end**
    **if** $t \equiv 0 \pmod{N^-}$ **then**
      Update the target network: $\zeta^- \leftarrow \zeta$
    **end**
  **end**
**end**

### C.2 NOISYNET-DUELING

**Algorithm 2:** NoisyNet-Dueling

**Input** : $Env$ Environment; $N_f$ maximum length of list $x$; $\varepsilon$ set of random variables of the network
**Input** : $B$ empty replay buffer; $\zeta$ initial network parameters; $\zeta^-$ 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

**for** *episode* $e \in \{1, \dots, M\}$ **do**
  /* $x$ is a list of states */
  $x \leftarrow [\,]$
  Initialise state sequence $x_0 \sim Env$
  $x[0] \leftarrow x_0$
  **for** $t \in \{1, \dots\}$ **do**
    /* $l[-1]$ is the last element of the list $l$ */
    Set $x \leftarrow x[-1]$
    Sample a noisy network $\xi \sim \varepsilon$
    Select an action $a \leftarrow \text{argmax}_{b \in A} Q(x, b, \xi; \zeta)$
    Sample next state $y \sim P(\cdot|x, a)$, receive reward $r \leftarrow R(x, a)$ and append $x$: $x[-1] \leftarrow y$
    **if** $|x| > N_f$ **then**
      Delete oldest element from $x$
    **end**
    Add transition $(x, a, r, y)$ to the replay buffer $B[-1] \leftarrow (x, a, r, y)$
    **if** $|B| > N_B$ **then**
      Delete oldest transition from $B$
    **end**
    /* $D$ is a distribution over the replay, it can be uniform or implementing prioritised replay */
    Sample a minibatch of $N_T$ transitions $((x_j, a_j, r_j, y_j) \sim D)_{j=1}^{N_T}$
    /* Construction of the target values. */
    **for** $j \in \{1, \dots, N_T\}$ **do**
      Sample a noisy network $\xi_j \sim \varepsilon$
      Sample a noisy target network $\xi'_j \sim \varepsilon$
      **if** $y_j$ *is a terminal state* **then**
        $\widehat{Q} \leftarrow r_j$
      **else**
        $b^*(y_j) = \arg\max_{b \in \mathcal{A}} Q(y_j, b, \xi'_j; \zeta)$
        $\widehat{Q} \leftarrow r_j + \gamma Q(y_j, b^*(y_j), \xi'_j; \zeta^-)$
      Do a gradient step with loss $(\widehat{Q} - Q(x_j, a_j, \xi_j; \zeta))^2$
    **end**
    **if** $t \equiv 0 \pmod{N^-}$ **then**
      Update the target network: $\zeta^- \leftarrow \zeta$
    **end**
  **end**
**end**

### C.3 NOISYNET-A3C

**Algorithm 3:** NoisyNet-A3C for each actor-learner thread

**Input :** Environment $Env$, Global shared parameters $(\zeta_\pi, \zeta_V)$, global shared counter $T$ and maximal time $T_{max}$.
**Input :** Thread-specific parameters $(\zeta'_\pi, \zeta'_V)$, Set of random variables $\varepsilon$, 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.

Initial thread counter $t \leftarrow 1$
**repeat**
  Reset cumulative gradients: $d\zeta_\pi \leftarrow 0$ and $d\zeta_V \leftarrow 0$.
  Synchronise thread-specific parameters: $\zeta'_\pi \leftarrow \zeta_\pi$ and $\zeta'_V \leftarrow \zeta_V$.
  $counter \leftarrow 0$.
  Get state $x_t$ from $Env$
  Choice of the noise: $\xi \sim \varepsilon$
  `/* r is a list of rewards */`
  $r \leftarrow [\,]$
  `/* a is a list of actions */`
  $a \leftarrow [\,]$
  `/* x is a list of states */`
  $x \leftarrow [\,]$ and $x[0] \leftarrow x_t$
  **repeat**
    Policy choice: $a_t \sim \pi(\cdot|x_t; \zeta_\pi; \xi)$
    $a[-1] \leftarrow a_t$
    Receive reward $r_t$ and new state $x_{t+1}$
    $r[-1] \leftarrow r_t$ and $x[-1] \leftarrow x_{t+1}$
    $t \leftarrow t + 1$ and $T \leftarrow T + 1$
    $counter = counter + 1$
  **until** $x_t$ *terminal or* $counter == t_{max} + 1$
  **if** $x_t$ *is a terminal state* **then**
    $Q = 0$
  **else**
    $Q = V(x_t; \zeta_V, \xi)$
  **for** $i \in \{counter - 1, \ldots, 0\}$ **do**
    Update $Q$: $Q \leftarrow r[i] + \gamma Q$.
    Accumulate policy-gradient: $d\zeta_\pi \leftarrow d\zeta_\pi + \nabla_{\zeta'_\pi} \log(\pi(a[i]|x[i]; \zeta'_\pi, \xi))[Q - V(x[i]; \zeta'_V, \xi)]$.
    Accumulate value-gradient: $d\zeta_V \leftarrow d\zeta_V + \nabla_{\zeta'_V}[Q - V(x[i]; \zeta'_V, \xi)]^2$.
  **end**
  Perform asynchronous update of $\zeta_\pi$: $\zeta_\pi \leftarrow \zeta_\pi + \alpha_\pi d\zeta_\pi$
  Perform asynchronous update of $\zeta_V$: $\zeta_V \leftarrow \zeta_V - \alpha_V d\zeta_V$
**until** $T > T_{max}$

| Games | Human | Random | DQN | NoisyNet-DQN | A3C | NoisyNet-A3C | Dueling | NoisyNet-Dueling |
|---|---|---|---|---|---|---|---|---|
| alien | **7127.70** | 227.80 | 2404.38 | 2402.66 | 2026.59 | 1898.79 | 5933.93 | 5777.88 |
| amidar | 1719.50 | 5.80 | 923.85 | 1610.47 | 904.49 | 490.55 | 2118.39 | **3537.32** |
| assault | 742.00 | 222.40 | 3594.81 | 5510.49 | 2878.91 | 3060.40 | 7010.75 | **11231.06** |
| asterix | 8503.30 | 210.00 | 6253.26 | 14327.85 | 6822.22 | **32477.78** | 9932.47 | 28349.78 |
| asteroids | 47388.70 | 719.10 | 1823.74 | 3455.14 | 2544.01 | 4541.31 | 2220.17 | **86700.31** |
| atlantis | 29028.10 | 12580.00 | 876000.00 | 923733.33 | 422700.00 | 465700.00 | 902741.67 | **972175.00** |
| bank heist | 753.10 | 14.20 | 454.95 | 1067.89 | 1295.56 | 1033.41 | **1417.82** | 1318.42 |
| battle zone | 37187.50 | 2360.00 | 28980.78 | 36785.57 | 16410.71 | 17871.43 | 40481.15 | **52261.55** |
| beam rider | 16926.50 | 363.90 | 10564.08 | **20792.79** | 9214.00 | 11237.33 | 15561.57 | 18500.60 |
| berzerk | **2630.40** | 123.70 | 634.26 | 905.30 | 1022.15 | 1235.41 | 1121.83 | 1896.23 |
| bowling | **160.70** | 23.10 | 61.86 | 71.46 | 36.73 | 42.14 | 71.97 | 68.46 |
| boxing | 12.10 | 0.10 | 87.34 | 88.77 | 91.31 | **99.75** | 98.56 | 99.66 |
| breakout | 30.50 | 1.70 | 396.01 | **515.55** | 495.89 | 373.74 | 190.84 | 263.29 |
| centipede | **12017.00** | 2090.90 | 6439.62 | 4268.56 | 5350.46 | 8282.01 | 4166.43 | 7595.75 |
| chopper command | 7387.80 | 811.00 | 7271.46 | 8893.07 | 5284.57 | 7560.51 | 7346.01 | **11477.20** |
| crazy climber | 35829.40 | 10780.50 | 116479.82 | 118305.30 | 134783.33 | 139950.00 | 163334.56 | **171171.06** |
| defender | 18688.90 | 2874.50 | 18303.00 | 20525.47 | 52916.67 | **55491.67** | 35236.63 | 42252.75 |
| demon attack | 1971.00 | 152.10 | 12696.35 | 36149.69 | 37085.00 | 37880.00 | 56382.22 | **69311.25** |
| double dunk | -16.40 | -18.60 | -5.57 | 1.33 | 2.67 | 2.67 | **16.93** | 1.00 |
| enduro | 860.50 | 0.00 | 835.13 | 1239.58 | 0.00 | 300.00 | **2064.43** | 2012.75 |
| fishing derby | -38.70 | -91.70 | 4.15 | 10.51 | -7.30 | -37.92 | 34.20 | **56.84** |
| freeway | 29.60 | 0.00 | 31.21 | 32.00 | 0.11 | 18.11 | **33.99** | 33.98 |
| frostbite | **4334.70** | 65.20 | 999.80 | 753.08 | 288.21 | 260.56 | 2807.41 | 2923.44 |
| gopher | 2412.50 | 257.60 | 11825.18 | 14573.73 | 7991.67 | 12439.29 | 22328.36 | **38908.72** |
| gravitar | **3351.40** | 173.00 | 366.29 | 447.44 | 378.53 | 313.82 | 1630.96 | 2208.70 |
| hero | 30826.40 | 1027.00 | 15176.39 | 6246.10 | 30791.28 | 8471.06 | **35866.91** | 31532.73 |
| ice hockey | 0.90 | -11.20 | -1.97 | -2.76 | -2.25 | -2.75 | -0.06 | **2.50** |
| jamesbond | 302.80 | 29.00 | 908.53 | 1235.28 | 508.93 | 188.25 | 1666.70 | **4682.10** |
| kangaroo | 3035.00 | 52.00 | 8166.25 | 10944.22 | 1166.37 | 1603.70 | 14847.31 | **15226.88** |
| krull | 2665.50 | 1598.00 | 8343.45 | 8804.65 | 9422.08 | **22848.70** | 10732.98 | 10753.76 |
| kung fu master | 22736.30 | 258.50 | 30443.56 | 36309.99 | 37422.22 | **55790.00** | 29130.95 | 41671.58 |
| montezuma revenge | **4753.30** | 0.00 | 2.42 | 3.03 | 14.44 | 3.55 | 0.00 | 57.43 |
| ms pacman | **6951.60** | 307.30 | 2674.37 | 2721.74 | 2435.81 | 3400.80 | 3649.81 | 5545.57 |
| name this game | 8049.00 | 2292.30 | 8179.15 | 8181.27 | 7168.33 | 8798.33 | 9918.70 | **12210.96** |
| phoenix | 7242.60 | 761.40 | 9703.78 | 16028.18 | 9475.56 | **50338.33** | 8214.84 | 10378.86 |
| pitfall | **6463.70** | -229.40 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| pong | 14.60 | -20.70 | 19.65 | **21.00** | 7.39 | 12.50 | 21.00 | 21.00 |
| private eye | **69571.30** | 24.90 | 2361.08 | 3711.62 | 3780.58 | 100.00 | 227.07 | 278.64 |
| qbert | 13455.00 | 163.90 | 11241.37 | 15545.29 | 18585.56 | 17895.83 | 19819.19 | **27120.72** |
| riverraid | 17118.00 | 1338.50 | 7240.70 | 9425.32 | 8135.24 | 7877.64 | 18009.62 | **23133.61** |
| road runner | 7845.00 | 11.50 | 37909.81 | 45993.30 | 45315.38 | 30454.25 | 62878.71 | **234352.03** |
| robotank | 11.90 | 2.20 | 54.58 | 51.44 | 5.78 | 36.33 | 60.06 | **63.75** |
| seaquest | **42054.70** | 68.40 | 4163.48 | 2282.06 | 1744.00 | 942.67 | 17756.02 | 16753.99 |
| skiing | **-4336.90** | -17098.10 | -12630.41 | -14762.60 | -12972.25 | -15970.28 | -7988.88 | -7549.53 |
| solaris | 12326.70 | 1263.30 | 4055.22 | 6087.56 | **12380.00** | 10426.67 | 3423.08 | 6521.88 |
| space invaders | 1668.70 | 148.00 | 1282.57 | 2185.52 | 1034.27 | 1126.40 | 1157.19 | **5909.03** |
| star gunner | 10250.00 | 664.00 | 40934.15 | 47133.45 | 49155.56 | 45008.33 | 68340.88 | **75867.17** |
| surround | 6.50 | -10.00 | -6.11 | -0.52 | -8.28 | 1.33 | 0.90 | **9.66** |
| tennis | -8.30 | -23.80 | **7.58** | 0.00 | -6.00 | 0.00 | 0.00 | 0.00 |
| time pilot | 5229.20 | 3568.00 | 6166.77 | 7035.02 | 10293.65 | 11124.17 | 13845.90 | **17300.64** |
| tutankham | 167.60 | 11.40 | 217.81 | 232.01 | 213.08 | 164.33 | **279.62** | 268.58 |
| up n down | 11693.20 | 533.40 | 11652.21 | 14254.52 | 89066.67 | **103556.67** | 90265.42 | 61326.33 |
| venture | 1187.50 | 0.00 | 319.47 | 96.55 | 0.00 | 0.00 | **1433.31** | 814.78 |
| video pinball | 17667.90 | 16256.90 | 429936.09 | 322507.40 | 229402.17 | 294724.33 | 800657.23 | **870953.52** |
| wizard of wor | 4756.50 | 563.50 | 3600.57 | 9197.92 | 8953.33 | **12723.33** | 6461.22 | 9148.82 |
| yars revenge | 54576.90 | 3092.90 | 20648.09 | 23915.30 | 21596.32 | 61754.60 | 27148.06 | **86100.96** |
| zaxxon | 9173.30 | 32.50 | 4805.64 | 6920.22 | **16544.44** | 1324.36 | 13461.85 | 14874.04 |

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

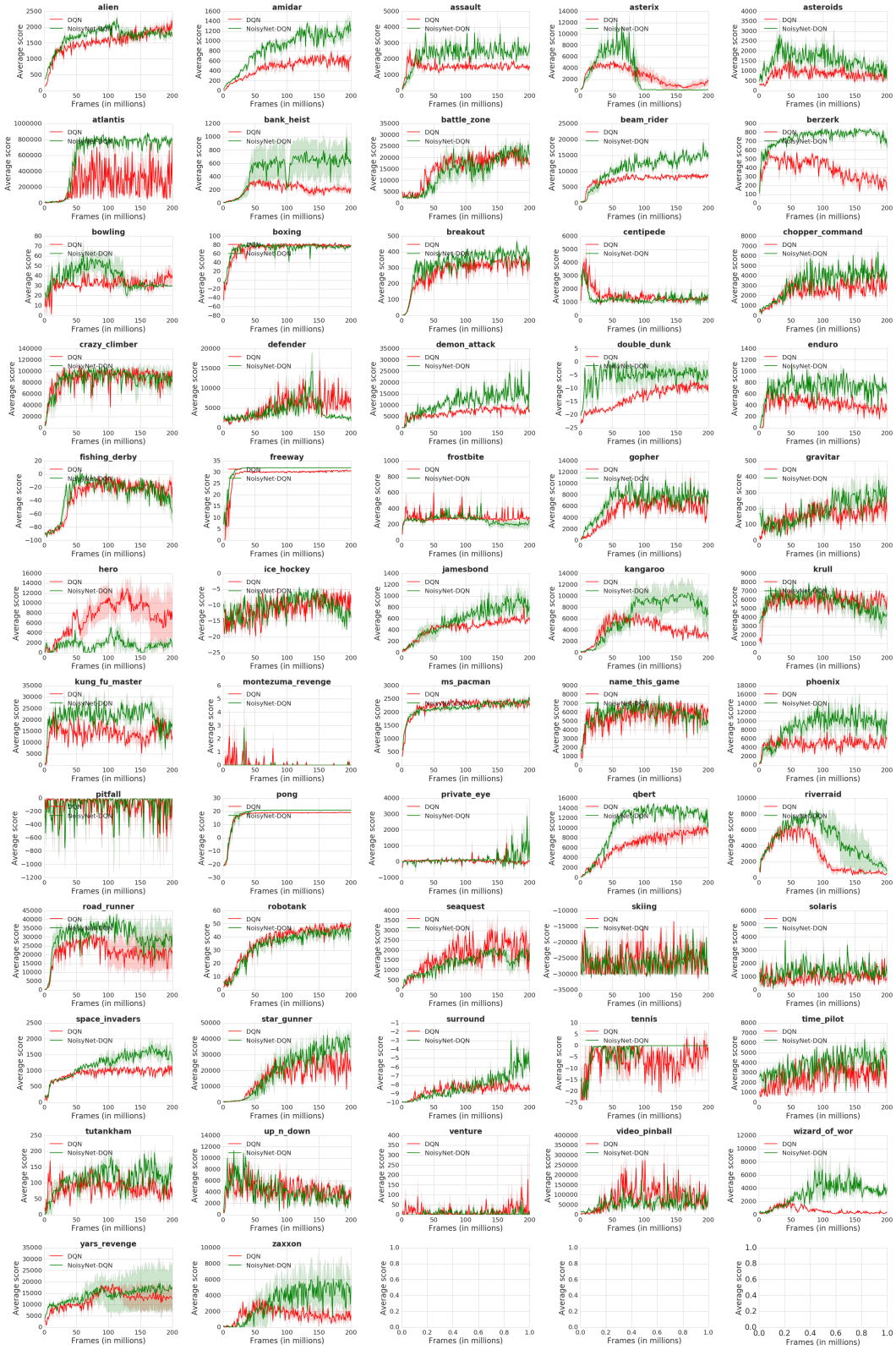



Figure 4: Training curves for all Atari games comparing DQN and NoisyNet-DQN.

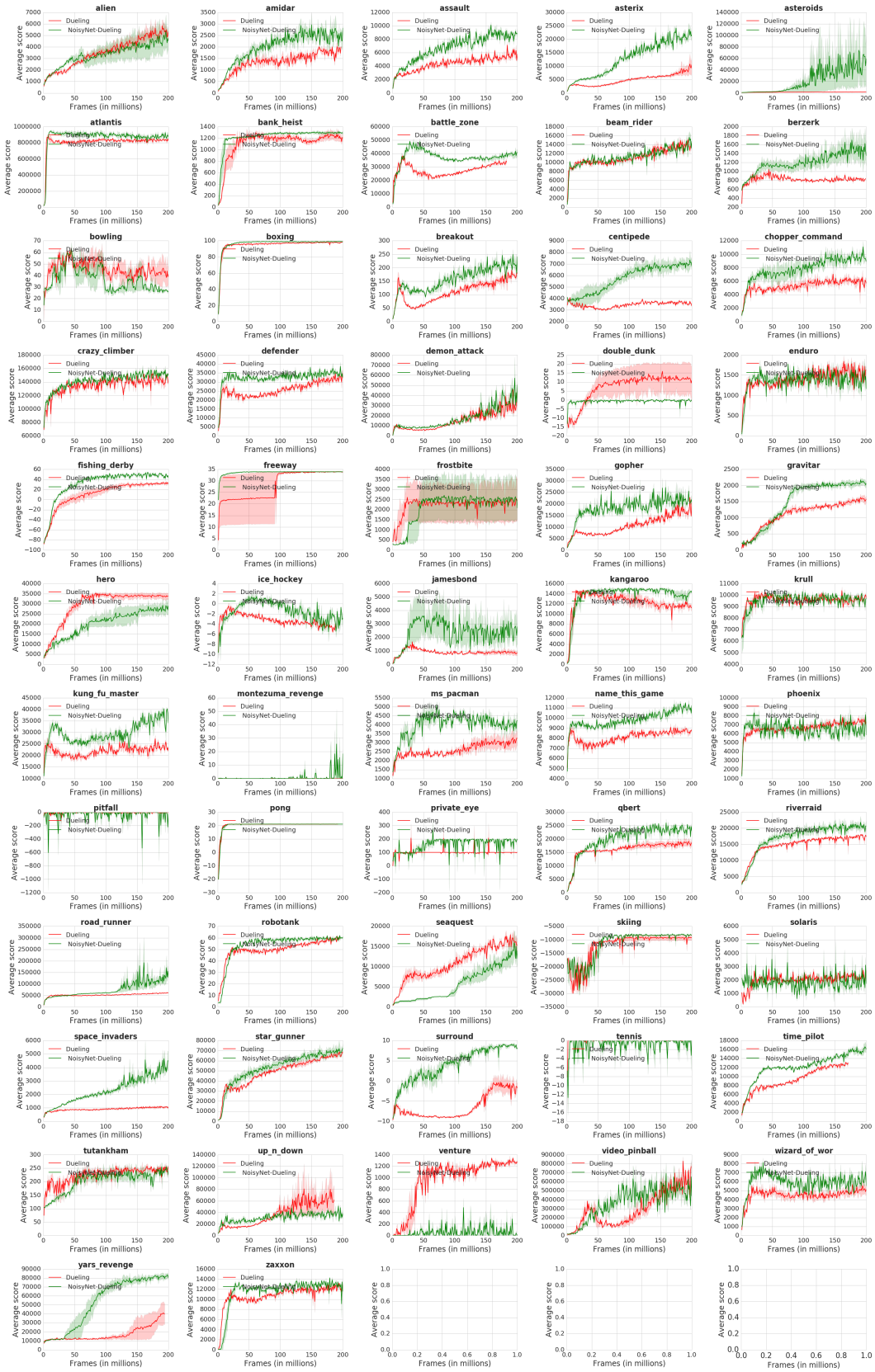



Figure 5: Training curves for all Atari games comparing Duelling and NoisyNet-Dueling.

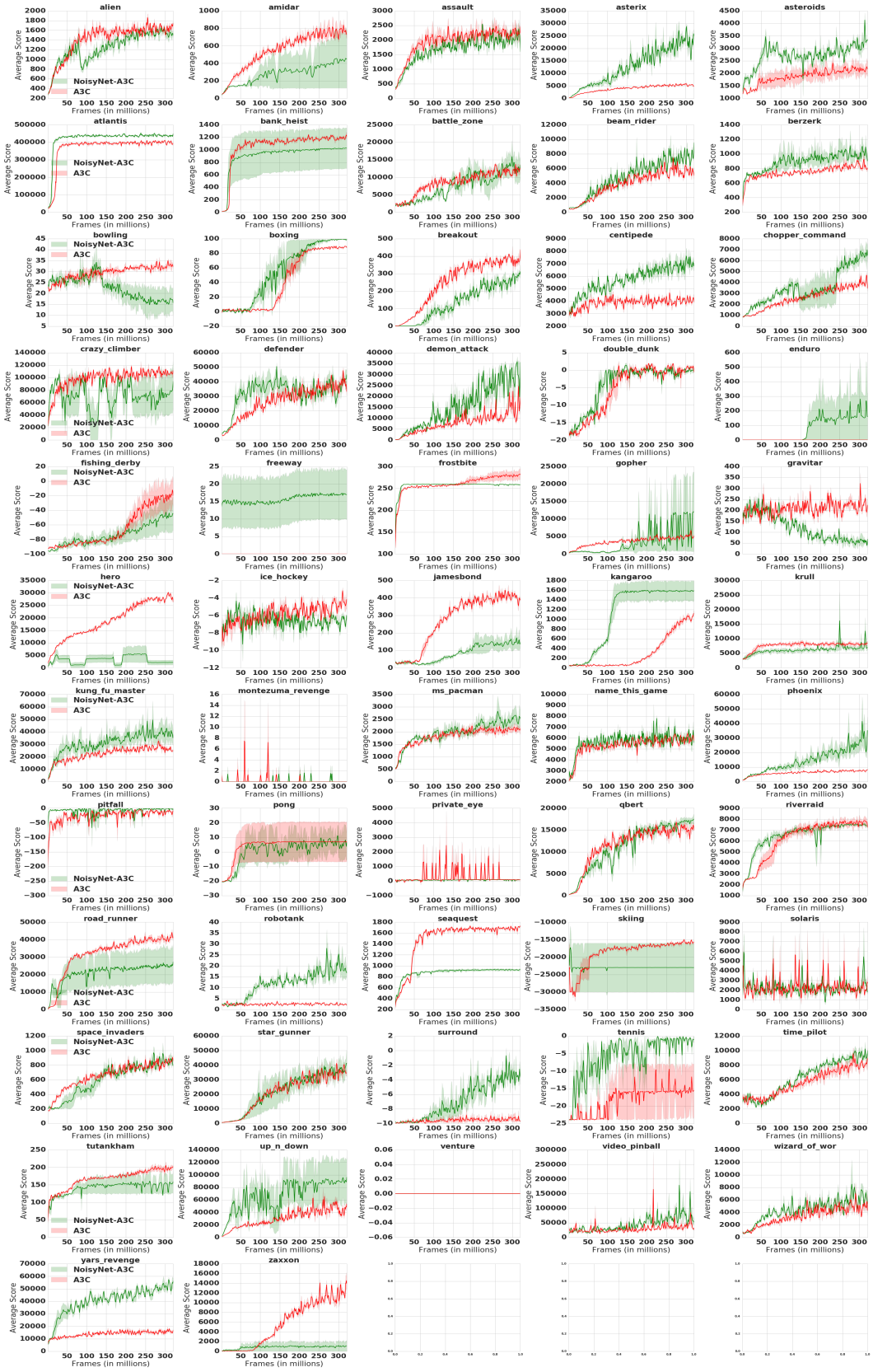



Figure 6: Training curves for all Atari games comparing A3C and NoisyNet-A3C.