DISTRIBUTIONAL META-GRADIENT REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Meta-gradient reinforcement learning (RL) algorithms have substantially boosted the performance of RL agents by learning an adaptive return. All the existing algorithms adhere to the same reward learning principle, where the adaptive return is simply formulated in the form of *expected* cumulative rewards, upon which the policy and critic update rules are specified under well-adopted distance metrics. In this paper, we present a novel algorithm that builds on the success of meta-gradient RL algorithms and effectively improves such algorithms by following a simple recipe, i.e., going beyond the *expected* return to formulate and learn the return in a more expressive form, value distributions. To this end, we first formulate a distributional return that could effectively capture bootstrapping and discounting behaviors over distributions, to form an informative distributional return target in value update. Then we derive an efficient meta update rule to learn the adaptive distributional return with meta-gradients. For empirical evaluation, we first present an illustrative example on a toy two-color grid-world domain, which validates the benefit of learning distributional return over *expectation*; then we conduct extensive comparisons on a large-scale RL benchmark Atari 2600, where we confirm that our proposed method with distributional return works seamlessly well with the actor-critic framework and leads to state-of-the-art median human normalized score among meta-gradient RL literature.

1 INTRODUCTION

Meta-gradient reinforcement learning (MGRL) (Xu et al., 2018b; Sutton, 2022) is a family of algorithms that leverage the gradient of the gradient descent update to learn better objectives for reinforcement learning (RL). The MGRL paradigm has achieved substantial performance breakthroughs and dominated the state-of-the-art model-free RL algorithms in various domains, such as Atari (Xu et al., 2018b; Zahavy et al., 2020; Flennerhag et al., 2022), DMLab (Zahavy et al., 2020), and DeepMind Control (Zahavy et al., 2020).

The major algorithmic research on MGRL has been developing towards more and more ambitious targets. For example, it starts from learning two fundamental hyperparameters (i.e., discount factor γ and bootstrapping factor λ) (Xu et al., 2018b), and then goes to self-tuning 20+ hyperparameters in a high-performing RL agent (Zahavy et al., 2020). As a powerful tool for discovering RL semantics from the data, MGRL has been utilized to discover intrinsic rewards (Zheng et al., 2018; 2020), auxiliary tasks (Veeriah et al., 2019), options (Veeriah et al., 2021), etc. Recently, it has even been used to tackle the more flexible form of parameterizing the objective, i.e. using black-box neural networks to learn the objectives from the environment interactions and learning context (Xu et al., 2020; Oh et al., 2020). However, MGRL research has been limited to scalar form, where either meta-parameters are scalar ones (Xu et al., 2018b; Zahavy et al., 2020), or outputs of the meta networks are in scalar form (Xu et al., 2020). This is in line with the traditional definition of returns (Sutton & Barto, 1998), which is an expected sum of the discounted rewards. However, note that MGRL is not limited to providing RL objectives, and it is possible to meta-learn a broader concept of knowledge, e.g, learning rate (Baik et al., 2020) and exploration (Xu et al., 2018a; Gupta et al., 2018).

In this work, we investigate an essential step of extending the algorithmic scope of MGRL to distributional methods, where RL algorithms take away the *expectations* in Bellman updates and consider a richer formulation of *value distributions* (Bellemare et al., 2017). This is essential for RL updates

and even more critical for MGRL which enjoy multiple satisfying properties. Sparse signals are hard to learn in RL, and bi-level optimization nature of the MGRL makes it extremely difficult to propagate helpful signals to learn meta-parameters. By modeling value distributions, the RL algorithm can naturally produce a rich set of predictions, which provides dense and informative signals for meta optimization. Moreover, some distributional RL methods come with good optimization properties in the loss form (not available in conventional non-distributional RL methods), e.g., with KL divergence between discrete distributions (Bellemare et al., 2017) or quantile regression (Dabney et al., 2018b). Despite the great accomplishment of distributional RL for non-meta policy learning, the attempt to learn a distributional meta objective has not yet been made by any prior MGRL methods.

This paper aims to develop a novel distributional meta-gradient RL algorithm that can discover meaningful value distributions online. Furthermore, we aim to both methodologically establish a novel distributional meta-gradient framework and find out the intriguing effect of applying distributional meta-gradient techniques on real-world problems, for which we conduct extensive evaluations with both motivating toy domains and large-scale end-to-end policy training problems. The evaluation results well demonstrate the effectiveness of our method.

Overall, the contributions of our paper are as follows:

(i) We present a novel meta-gradient variant of the RL method, which considers approximating the distributional return in its value update.

(ii) Our proposed distributional meta-gradient algorithm is general and can be compatible with almost all the existing meta-gradient RL approaches.

(iii) We conduct an extensive empirical study on a toy control domain and the large-scale benchmark Atari 2600. Our results have demonstrated substantial improvements by our method over strong baseline methods on the major evaluation domains.

2 Related Work

In recent decades, meta-learning, or *learning-to-learn*, has been explored extensively within the machine learning community (Hospedales et al., 2022; Sutton, 2022), e.g., learning a neural network update rule (Bengio et al., 1990), adapting the learning rate (Sutton, 1992; Schraudolph & Giannakopoulos, 1999), and transferring domain-invariant knowledge (Thrun & Mitchell, 1995). Notably, it has also driven many advances in the RL problems. One direction for such attempts is to learn *meta*-optimizers (Andrychowicz et al., 2016; Wichrowska et al., 2017) or *meta*- policies (Duan et al., 2016; Wang et al., 2017) parameterized by recurrent neural networks, which capture the high-level time-dependent information as meta knowledge. Another direction is gradient-based meta-learning, introduced in Model-Agnostic Meta-Learning (MAML) (Finn et al., 2017a), as well as its followup works (Finn & Levine, 2018; Finn et al., 2017b; Grant et al., 2018; Al-Shedivat et al., 2018). Those works focus on learning a good initialization of the model for one or few-shot multi-task learning with meta-gradients.

In this paper, we consider solving meta-gradient RL problem from a fresh new angle, i.e., optimizing the adaptive *return* as distributions. Policy learning with distributional *return* has been considered in existing RL literature for off-policy learning only. All the existing methods fall into the strand of DQN variants which replace the n-step return in DQN with distributions during Q-learning. In (Bellemare et al., 2017), the *return* is formulated as a categorical distribution with fixed values for the base, where Bellman operator shifts the base values of the distribution and the target distribution is contracted and projected upon the fixed base. In (Dabney et al., 2018b), the *return* is formulated as quantile distributions, where the algorithm tries to learn the values corresponding to each quantile, alleviating the need for distribution projection or value range approximation. In (Dabney et al., 2018a), the quantiles are approximated by mapping from values drawn from the random distribution (proposals) into quantiles, resulting in an unlimited number of implicit distributions. In (Wang et al., 2019), the proposals in implicit quantiles are replaced by the output of a learned proposal network. In contrast to the aforementioned off-policy distributional RL methods, we tackle a novel problem of learning *adaptive* distributional *return* in on-policy actor-critic algorithms with meta-gradient.

Compared to existing meta-gradient RL methods with scalar *return*, our method applies a distributional return, which is advantageous as (1) it can offer a more informative target to approximate, (2) it naturally provides a richer set of hyperparameters to be tuned with white-box meta-learning,

which offers more feedback in the meta-gradient, and (3) approximating such a distributional return naturally results in meaningful auxiliary tasks to facilitate more efficient policy and value training.

3 PRELIMINARY

Quantile Distributional Value Update In this work, we consider learning the *return* as a quantile distribution. One reason for us to choose quantile distribution rather than any other type of distribution is, the Bellman operator for quantile distribution does not need to perform projection (lose precision) or manually define the *return* boundaries, like C51 (Bellemare et al., 2017) does. Also, our proposed method employs a deterministic prediction function for quantiles which could be conveniently trained, unlike the stochastic mappings employed by IQN (Dabney et al., 2018a), where the stochasticity would bring noise to the meta gradient and confuse the training of adaptive *return*. We denote the distributional *return* function over each state s as $Z_\theta(s)$, where θ is its parameter set. With quantile, the distributional *return* can be approximated by a uniform mixture of N Dirac functions, i.e., $Z_\theta(s) \coloneqq \frac{1}{N}\sum_{q=1}^{N}\delta_{\theta_q}(s)$, where θ_q is the parameter for the q-th Dirac function which can be trained by quantile value update. For a given quantile distribution Z and a quantile τ, the value of the quantile function is denoted as $F_\theta^{-1}(\tau)$. The quantile distribution can be trained by quantile regression loss $\mathcal{L}_\tau^{qr}(\cdot)$ which penalizes overestimation errors with weight τ and underestimation errors with weight $1-\tau$, where $\tau \in [0,1]$:

$$\mathcal{L}_\tau^{qr}(\cdot) = \mathbb{E}_{\hat{Z}\sim Z}\big[\rho_\tau(\hat{Z}-\theta)\big], \qquad \text{where } \rho_\tau(u) = u\,(\tau - \delta_{\{u<0\}}), \quad \forall u \in \mathbb{R}. \tag{1}$$

To smooth the quantile loss at zero, we could apply Huber loss over it, i.e.

$$\mathcal{L}_k^{\text{huber}}(u) = \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \le k; \\ k\,(|u| - \frac{1}{2}k), & \text{otherwise.} \end{cases} \tag{2}$$

Thus the Quantile Huber Loss can be derived as follows,

$$\mathcal{J}_{(\tau,k)}^{qr} = |\tau - \delta_{\{u<0\}}| \cdot \mathcal{L}_k^{\text{huber}}(u). \tag{3}$$

White-box Meta-Gradient RL White-box meta-gradient RL methods use a bi-level optimization scheme to update important high-level parameters to characterize the *return* and the composed loss function by meta gradient. We denote the high-level parameter set as η. In the prior work from (Xu et al., 2018b), $\eta = \{\gamma, \lambda\}$; later, loss weights and importance sampling threshold have been added to η. We denote the RL loss function as $\mathcal{J}(\tau,\theta,\eta)$. During bi-level optimization, the meta-gradient methods first update the model parameter θ with the gradient $f(\cdot)$ w.r.t θ:

$$f(\tau,\theta,\eta) = \zeta\,\frac{\partial \mathcal{J}(\tau,\theta,\eta)}{\partial\theta}, \qquad \frac{\partial f(\tau,\theta,\eta)}{\partial\eta} = \zeta\,\frac{\partial^2 \mathcal{J}(\tau,\theta,\eta)}{\partial\theta\,\partial\eta}, \tag{4}$$

where ζ is the learning rate and the first-order gradient function $f(\cdot)$ is differentiable w.r.t η. Then, an alternative batch of transitions τ' is sampled to optimize the model in the second update stage, where outer objective $\mathcal{J}'$ computed from the updated model parameters θ' and a reference hyperparameters set η' is optimized (e.g., default values for η') w.r.t η:

$$\frac{\partial J'(\tau',\theta',\eta')}{\partial\eta} = \frac{\partial J'(\tau',\theta',\eta')}{\partial\theta'} \cdot \frac{\partial\theta'}{\partial\eta}. \tag{5}$$

Since θ' is differentiable w.r.t η, the meta gradient is computed as the gradient of the outer objective $\mathcal{J}'(\tau',\theta',\eta')$ w.r.t η. Thus meta-gradient RL enables the effective high-level parameters in *learning-to-learn* to compute the adaptive *return* and/or the importance of loss function components.

4 DISTRIBUTIONAL META-GRADIENT RL ALGORITHM

Actor-Critic with Distributional λ-Return The principle of RL is to optimize the agent's cumulative rewards, also known as the *return* (denoted as G), from the sequence of experiences collected from the environment, where the experience at time t id denoted as a tuple $\tau_t = (S_t, A_t, R_t, S_{t+1})$ with S_t, A_t and R_t denoting the state, action, and reward for the state-action pair at step t, respectively. In practice, the *return* can be formulated in various forms. One well-adopted form of *return*

is via *bootstrapping*. Let T denote the terminal step and γ the discount factor. An example *n*-step *return* (Sutton & Barto, 1998) is defined as:

$$G^{\text{n-step}}(\tau_t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ..., +\gamma^{n-1} v(S_{t+n}), \tag{6}$$

$$G^{\lambda}(\tau_t) = R_{t+1} + \gamma(1-\lambda)v(S_{t+1}) + \gamma\lambda G^{\lambda}(\tau_{t+1}), \tag{7}$$

where the *return* is the sum of n step rewards and the bootstrapped value for S_{t+n}. To better leverage the *return* G_t derived over multiple steps, there is also a λ-*return*, or TD(λ) *return*, which measures the *return* as the weighted TD error over all time steps (Sutton, 1988), where there is another high-level parameter λ introduced to characterize the λ-*return* function. When $\lambda = 0$, it terminates the bootstrapping from further steps, and when $\lambda = 1$, it promotes pure bootstrapping. White-box meta-gradient RL methods attempt to meta-learn the crucial hyperparameters characterizing the λ-*return* with a bi-level optimization regime.

We consider the training of actor-critic algorithms with the critic approximating a quantile distributional *return* predicted by a value function $Z_\theta(S)$, where S is an input state and θ is the parameter set of the value function. The function $Z_\theta(S)$ follows the *distributional Bellman optimality operator* (Bellemare et al., 2017),

$$\mathcal{T}Z(S) \overset{D}{:=} R(S,A) + \gamma\, \mathbb{E}_{S'\sim P(\cdot|S,A)} Z(S'). \tag{8}$$

The critic is updated towards a canonical *distributional* λ-return denoted as Γ_t^λ following a *distributional TD(λ)* algorithm,

$$\begin{aligned}\Gamma_t^\lambda &\overset{D}{:=} R_{t+1} + \gamma_{t+1}(1-\lambda_{t+1})\, Z(S_{t+1}) + \gamma_{t+1}\lambda_{t+1}\Gamma_{t+1}^\lambda, \\ &\overset{D}{:=} Z(S_t) + \sum_{k=0}^{\infty}\Big(\prod_{j=1}^{k}\gamma_{t+j}\lambda_{t+j}\Big)\underbrace{\Big(R_{t+k+1} + \gamma_{t+k+1}Z(S_{t+k+1}) - Z(S_{t+k})\Big)}_{\delta_{t+k}},\end{aligned} \tag{9}$$

where Γ_t^λ is essentially the sum of the predicted *return* distribution $Z(S_t)$ and the backpropagated TD error over all the future steps $\delta_{t+1:\infty}$, where the later could be computed iteratively and weighted by the product of discounting factor and trace in their exponential forms. When training the actor-critic with Γ_t^λ under a distributed regime like IMPALA (Espeholt et al., 2018) for efficient learning, the training data would become off-policy and we need to perform off-policy correction to get the unbiased gradient. Hence distributional λ-*return* with off-policy correction in the form of *v-trace* could be defined iteratively as follows,

$$\Gamma_t^\lambda \overset{D}{:=} Z(S_t) + \sum_{k=0}^{\infty} c_{t+k}\Big(\prod_{j=1}^{k}\gamma_{t+j}c_{t+j}\Big)\delta_{t+k}, \quad \text{where } c_m = \lambda_m\Big(\min\big(\bar{c}, \frac{\pi(A_m|S_m)}{\mu(A_m|S_m)}\big)\Big), \tag{10}$$

where c_m is the truncated importance sampling (IS) ratio with $\bar{c}$ being the clipping threshold, and $\pi(\cdot)$, $\mu(\cdot)$ are the present and previous behavior policies, respectively. To update the distributional actor-critic with off-policy correction, we optimize the following objective function:

$$\begin{aligned}&\mathcal{L}_Z = \mathcal{J}^{qr}_{(\tau,k)}(\text{sg}\big(\Gamma^\lambda(S)\big), Z_\theta(S)), && \mathcal{L}_\mathcal{H} = \mathcal{H}\big(\pi_\theta(\cdot|S)\big), \\ &\mathcal{A}_Z = \frac{1}{N}\sum_{q=1}^{N}\text{sg}\big(\Gamma^\lambda_{(q)}(S) - Z_{\theta_q}(\tau)\big)\cdot\Big(\min\big(\bar{c}, \frac{\pi(A|S)}{\mu(A|S)}\big)\Big), && \mathcal{L}_\pi = -\mathcal{A}_Z(\tau)\cdot\log\pi_\theta(A|S), \\ &\mathcal{L}_\theta = w_\pi\cdot\mathcal{L}_\pi + w_\mathcal{H}\cdot\mathcal{L}_\mathcal{H} + w_Z\cdot\mathcal{L}_Z,\end{aligned} \tag{11}$$

where $\mathcal{A}_Z$ denotes the advantage computed as the sum over quantile error of the distributional *return*, $\Gamma^\lambda_{(q)}(S)$ denotes the q-th quantile in the distributional λ-*return*, $\mathcal{L}_Z/\mathcal{L}_\pi/\mathcal{L}_\mathcal{H}$ and $w_Z/w_\pi/w_\mathcal{H}$ are functions and weights for the value/policy/entropy losses respectively, sg$(\cdot)$ is the *stop-gradient* function and the first term in $\mathcal{L}_\pi$ corresponds to the advantage.

Distributional Meta-Gradient Update Our proposed distributional Meta-Gradient RL algorithm adopts a white-box bi-level optimization scheme to learn adaptive distributional *return* Γ_t^η. We denote the meta parameters characterizing the adaptive *return* Γ_t^λ and important weights in the actor-critic loss function $\mathcal{L}_\theta$ as a set η, where $\eta = \{\lambda, \gamma, \alpha, w_\mathcal{H}, w_\pi, w_Z\}$, where α is the weight for the

leak-vtrace in STACX (Zahavy et al., 2020) to control the off-policy correction ratio. Formally, the adaptive return can be formulated as follows,

$$\Gamma_t^\eta \overset{D}{:=} R_{t+1} + \gamma(1-\lambda)\, Z(S_{t+1}) + \gamma\lambda \underbrace{\Big\{ Z(S_{t+1}) + \sum_{k=1}^{\infty} \Big(\prod_{j=1}^{k} \gamma\lambda \Big) \delta_{t+k} \Big\}}_{\Gamma_{t+1}^\eta},$$

$$\overset{D}{:=} R_{t+1} + \gamma Z(S_{t+1}) + \gamma\lambda \sum_{k=1}^{\infty} \Big(\prod_{j=1}^{k} \gamma\lambda \Big) \delta_{t+k}.$$

Our proposed distributional adaptive *return* can naturally result in a larger set of self-tuning hyperparameters. In other words, we can consider a quantile-specific discounting and bootstrapping operator over the distributions and vectorize the λ and γ in the set η. Furthermore, we can also specify a vectorial quantile loss weight w_Z. Thus, without employing auxiliary tasks, our method results in a much larger hyperparameter set compared with **STACX**, i.e., $|\eta|$ for our method, **STACX** and **MG** are $3 \times n_Z + 3$, $7 \times n_{blk}$ and 2, where n_Z and n_{blk} represents the dimension of quantiles and number of multi-head blocks respectively.

At the inner loop, the inner loss $\mathcal{L}_\theta^{\text{in}}(\tau, \theta, \eta)$ is first computed to update θ towards θ' using the gradient $\partial \mathcal{L}_\theta^{\text{in}}(\tau, \theta, \eta)/\partial\theta$, where τ is a batch of data sampled to update the inner loss and η is drawn from the meta parameters.

$$f(\tau; \theta; \eta) = -\zeta \frac{\partial \mathcal{L}_\theta^{\text{in}}(\tau, \theta, \eta)}{\partial\theta} = -\zeta w_Z \frac{\partial \mathcal{L}_Z}{\partial\theta} - \zeta w_\pi \frac{\partial \mathcal{L}_\pi}{\partial\theta} - \zeta w_{\mathcal{H}} \frac{\partial \mathcal{L}_{\mathcal{H}}}{\partial\theta}, \tag{12}$$

where the gradient of the distributional value loss is derived as follows,

$$\frac{\partial \mathcal{L}_Z(\tau, \theta, \eta)}{\partial\theta} = \begin{cases} w_Z \left(\Gamma^\eta(S) - Z_\theta(S)\right) \frac{\partial Z_\theta(S)}{\partial\theta}, & \text{if } |\Gamma^\eta(S) - Z_\theta(S)| \leq k; \\ w_Z \frac{\left(\Gamma_\eta(\tau) - Z_\theta(S)\right)}{|\Gamma_\eta(\tau) - Z_\theta(S)|} \frac{\partial Z_\theta(S)}{\partial\theta}, & \text{otherwise.} \end{cases} \tag{13}$$

At the outer loop, we evaluate the high-level parameters η by sampling an alternative batch of transitions τ' and computing the outer loss $\mathcal{L}_\eta^{\text{out}}(\tau', \theta', \eta')$, where η' corresponds to some reference hyperparameters for η, e.g., the default values predefined like those in other algorithms. The meta gradient to update the meta parameter η is defined as follows,

$$f'(\tau', \theta', \eta') = \zeta' \frac{\partial \mathcal{L}_\eta^{\text{out}}(\tau', \theta', \eta')}{\partial\eta} = \zeta' \frac{\partial \mathcal{L}_\eta^{\text{out}}(\tau', \theta', \eta')}{\partial\theta'} \frac{\partial\theta'}{\partial\eta} = f(\tau'; \theta'; \eta') \frac{\partial^2 \mathcal{L}_\theta^{\text{in}}(\tau, \theta, \eta)}{\partial\theta\, \partial\eta}, \tag{14}$$

where ζ' is the learning rate for the outer loss. The meta gradient evaluated at the outer loss interacts with meta parameters through $\frac{\partial\theta'}{\partial\eta}$, which could be further expanded as $\frac{\partial^2 \mathcal{L}_\theta^{\text{in}}(\tau,\theta,\eta)}{\partial\theta\, \partial\eta}$ based on the chain rule. With the definition of the distributional return, meta-learning variants become learning the bootstrapping or termination condition over the distributions, rather than scalars, which would bring informative information for return approximation with a greater degree of adaptability.

5 Experiments

We denote our proposed Distributional *return* Meta-Gradient algorithm as **DrMG**.

5.1 Two Colors Grid-World

Settings We consider a 7×7 grid-world, where the green-colored agent is assigned to collect the key to open the door at the yellow-colored goal. There are two types of keys; collecting one of them would get a reward of 0.7 and the other 0.3. The agent receives a reward $+1$ upon successful navigation. After an

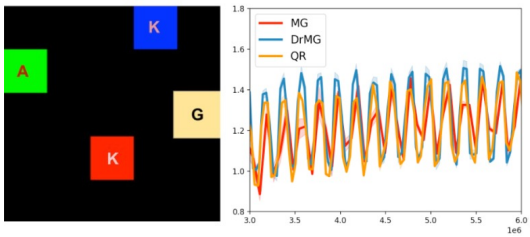


Figure 1: Two colors grid-world environment and the reward curve for **DrMG**, **MG** and a vanilla distributional method **QR** without meta-gradient.

episode terminates, we randomly generate locations for both keys and the goal. We toggle the reward at every 1e5 steps to investigate the agent's adaptation ability. Thus a key-agnostic optimal agent could collect $+1.5 - r_{pen} * T$ expected return, where r_{pen} and T are the step-wise penalty and episode horizon. We assume that the agent approximating the distributional return should be able to encode the dynamics in its *return* so that it can adapt better compared to the baseline agent approximating the *scalar return*.

Results We compare our proposed **DrMG** agent with a meta-gradient baseline algorithm **MG** as well as a vanilla distributional algorithm **QR**, where **QR** is an IMPALA variant with distributional return. The results are shown in Figure 1. We configure **DrMG**, **MG**, and **QR** agents by almost identical hyperparameter sets. Since the RL agents are unaware of the reward toggling during policy training, their adaptation ability is characterized by their efficiency in modeling the *return* during the training process. We notice that after the reward toggle, **DrMG** achieves peaks with substantially higher *return* compared to that for **MG** and **QR**. Furthermore, **QR** slightly outperforms **MG**, which shows that approximating distributional *return* benefits the task adaptation.

5.2 ATARI 2600 200M

Settings We testify our method on Atari 2600 Benchmark (Bellemare et al., 2013) under the 200M setting, where we try best to align the setting with prior works (Xu et al., 2018b; Zahavy et al., 2020). Our agent is developed upon IMPALA (Espeholt et al., 2018), an efficient distributed actor-critic framework. The actor-critic module employs a deep ResNet (He et al., 2016) as backbone, which is a feed-forward model without LSTM. The network takes 4 grayscale images with size 84×84 as input. The backbone is connected to a policy-value head where the value head outputs distributional quantiles with a dimension of N and the policy head outputs policy logits with a dimension of $|\mathcal{A}|$. The meta module is a joint set of scalars and vectors, with each vector entry representing a distributional hyperparameter in η.

For training, we strictly follow the 200M regime without extra data or experience reuse. During training, we sample two batches with identical batch size and use them to form two combinations of inner data and outer data to compute the meta-gradient update loop twice. Each task runs with identical hyperparameters, and device configurations for hardware and software. We highlight that DrMG brings very little computational overhead; namely, DrMG only brings $|N - 1| * 256$ more parameters compared to MG and it is more computationally efficient than STACX, since STACX introduces multiple policy-value heads as its auxiliary task components. All the games employ the same wrapper, for which the action repeat is 4, stick action probability is 0, episodic life is *false*, and maximum episode length is 108,000. All the tasks adopt the 'no-ops starts' protocol, where at the start of each episode, the agent randomly samples a period to take a dummy action '0' up to 30 steps. We present the environment wrapper setting in Sec. A.1.2, and the detailed model/experiment configurations in Sec. A.2. For comparison, we evaluate the performance on each game in terms of human normalized score (HNS) defined as $\text{HNS} = \frac{R_{\text{ours}} - R_{\text{rand}}}{R_{\text{human}} - R_{\text{rand}}}$, where $R_{\text{ours}}/R_{\text{rand}}/R_{\text{human}}$ are the episode rewards for our/random/human expert policies.

Results For evaluation over the 57-task benchmark, we compute the *median HNS* for each method. We consider the following five most related baselines: **C51**, **QR-DQN**, Implicit Quantile Network (**IQN**), Self-Tuning Actor-Critic with auXiliary tasks (**STACX**) and **MG**. The first three are off-policy value-based methods which are DQN-variants, while **STACX** and **MG** are white-box meta-gradient RL baselines that are IMPALA-variants, the same as **DrMG**. There is another recently published meta-grad RL work Bootstrapped Meta-Gradient (**BMG**), which is excluded here for comparison because it alters some important experimental settings for 200M evaluation to boost the performance, e.g., a much larger ResNet and RGB

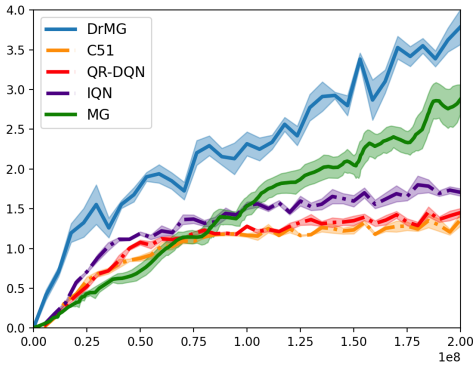
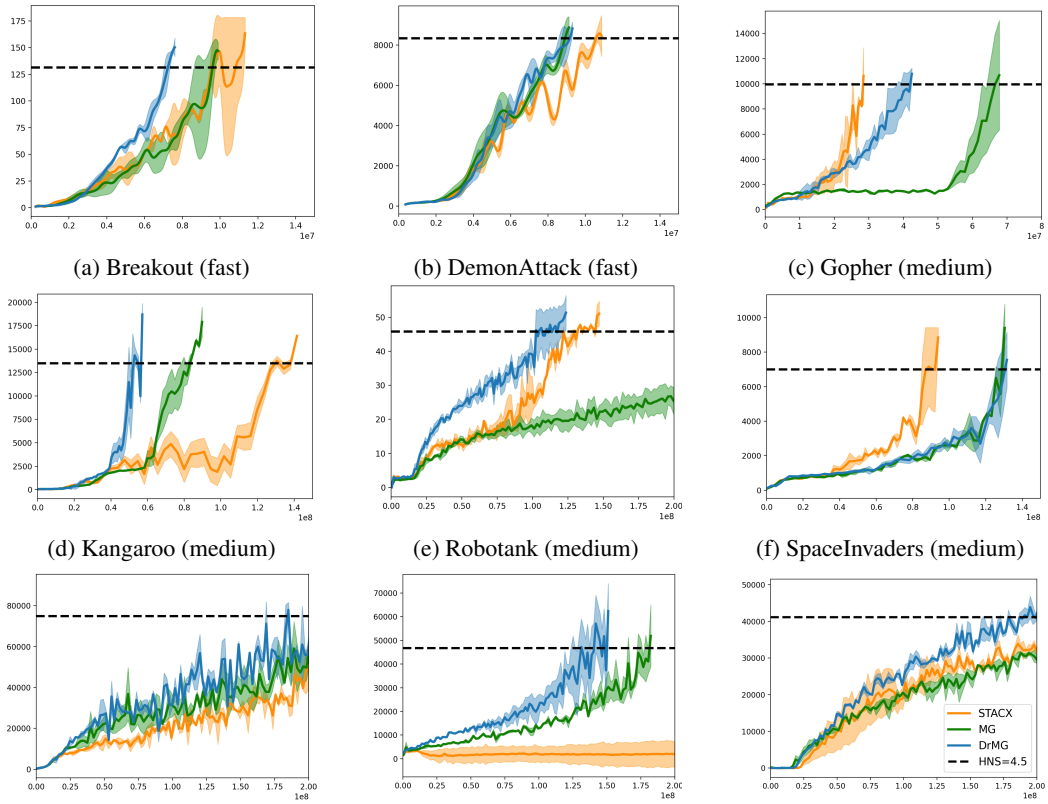


Figure 2: Learning curve in terms of median HNS evaluated on DrMG and its baselines.



(a) Breakout (fast) (b) DemonAttack (fast) (c) Gopher (medium)

(d) Kangaroo (medium) (e) Robotank (medium) (f) SpaceInvaders (medium)

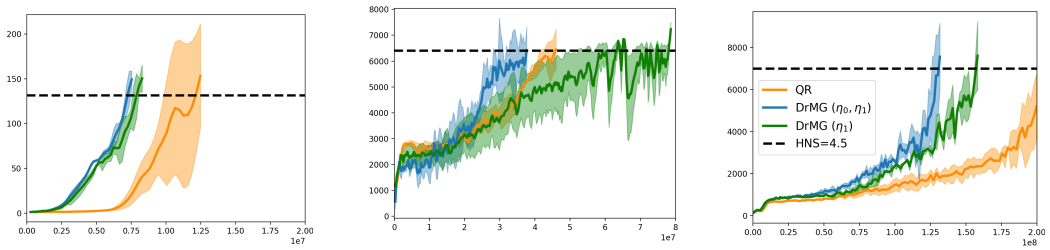
(g) BeamRider (slow) (h) Centipede (slow) (i) Zaxxon (slow)

Figure 3: Reward learning curves for our method and meta-gradient RL baselines **MG** and **STACX** evaluated on a representative set of Atari games. Overall, **DrMG** results in stable and efficient progress in reward learning across all the *fast* / *medium* / *slow* groups. Its median HNS score on Atari 2600 benchmark is 396%.

frames for inference, thus making it unfair to be directly compared with the other IMPALA variants. We show the learning curves for median HNS in Figure 2. We do not present the curve for **STACX**, because we find the reported standard for **STACX** is rather hard to be reproduced. When carefully reimplementing it with an identical framework, hyperparameters and model architectures, the gap between the reproduced and reported median HNS is $> 50\%$. Nevertheless, the learning curves for **STACX** on representative games can be found in Figure 3. Such a reproduction issue is notorious for Atari 2600 200M evaluation due to the missing official releases of the baselines.

From the median HNS curves shown in Figure 2, we can first observe that **DrMG** performs substantially better than the conventional distributional RL methods **C51**, **QR-DQN** and **IQN**, which demonstrates updating the distributional *return* adaptively with meta-gradient and the actor-critic framework is more effective than updating the distributional returns by off-policy DQN variants. We also find that **DrMG** brings noticeable performance improvement over the strong white-box meta-gradient RL method **MG**. The median HNS for **DrMG** is 396% and that reported in **MG** is 287%. These results show that optimizing the distributional *return* in our proposed λ-distributional form is more efficient than learning the scalar *return*.

We present the learning curves of our method, **MG** and **STACX** on a set of representative Atari games in Figure 3. For computational efficiency, the training adopts an *early stopping* policy, where the training terminates after the agent surpasses a fixed HNS standard (450% in our evaluation) for ¿2M steps. We classify the games into three strands $\{$*fast*, *medium*, *slow*$\}$, based on the behavior of the RL agents to hit the designated HNS standard, where *fast*=$\{$*Breakout*, *DemonAttack*$\}$, *medium*=$\{$*SpaceInvaders*, *Robotank*, *Gopher*, *Kangaroo*$\}$ and *slow* = $\{$*BeamRider*, *Zaxxon*, *Centipede*$\}$. Games in *fast*, *medium* and *slow* groups consume $< 10\%$, $30\% - 60\%$ and $\sim$200M frames to hit the HNS, respectively. We examine the *fast* group to show that learning the distributional



(a) Breakout (fast) (b) Krull (medium) (c) SpaceInvaders (medium)

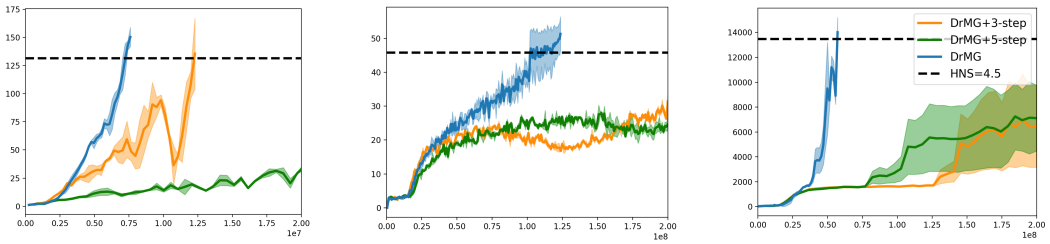
Figure 4: Ablation study results on comparing **DrMG** with a non-adaptive distributional actor-critic baseline **QR**. Baselines learning with non-adaptive distributional *return* performs substantially worse than **DrMG**.

return in **DrMG** would not slow down the learning process compared to its counterparts learning scalar *return*. On the classic Atari game *Breakout*, **DrMG** converges noticeably faster than **MG** and **STACX**, and on *DemonAttack*, **DrMG** learns as fast as **MG**. It is also desirable to evaluate learning efficiency for the games from the *medium* group since different methods behave very differently in making progress in those tasks. From the results, we can see **DrMG** progresses significantly faster than the other two methods in *Robotank* and *Kangaroo*. In *Kangaroo*, our method saves almost 1/3 and 1/2 frames compared to **MG** and **STACX**. In *SpaceInvaders*, the progress of **DrMG** is comparable to the others. In the *slow* group, the RL algorithms strive to progress with slow learning paces. All the three games are challenging ones for our method as well as the other baselines to achieve high HNS. So, the performance on them is crucial to derive the median HNS score, i.e., pulling back any one could potentially lower the median HNS. We can observe that **DrMG** leads to fairly stable performance among those in *slow* group. In *BeamRider*, *Centipede* and *Zaxxon*, **DrMG** achieve better asymptotic rewards than the baselines. It is also shown in Figure 3 that the performance of **STACX** is not ideal, e.g., irregular performance in *DemonAttack* and *Kangaroo*.

5.3 Ablation Study

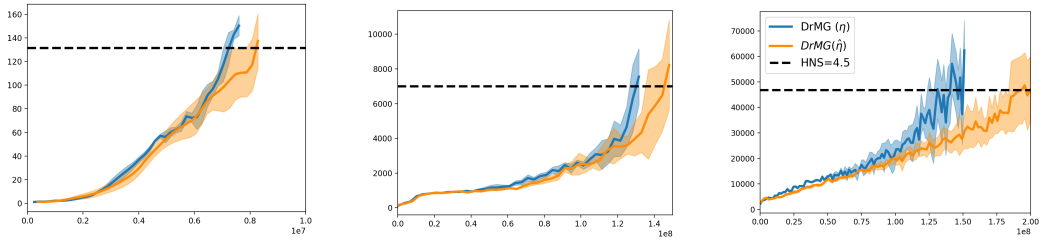
We perform ablation studies to investigate the effects of the following designs in our method: (i) the *adaptive* distributional *return* vs. *non-adaptive* distributional *return*; (ii) λ-distributional return vs. n-step distributional return; (iii) vector hyperparameter η vs. scalar η; (iv) adopting BMG's bootstrapped inner steps on top of our method. Due to space limit, we present the results of ablation study (iv) in appendix A.3.

Adaptive* vs. *non-adaptive* distributional *return We investigate the effects of adaptive/non-adaptive distributional return by comparing **DrMG** with two ablated baselines. The meta parameter set is split into two groups $\eta = \{\eta_0, \eta_1\}$, where $\eta_0 = \{\lambda, \gamma, \alpha\}$ and $\eta_1 = \{w_{\mathcal{H}}, w_\pi, w_Z\}$. The first baseline only meta-update η_1, denoted as **DrMG**(η_1) (complete version is denoted as **DrMG**(η_0, η_1)). The second baseline is **QR**, a non-meta method equivalent to **DrMG**($\{\}$). We



(a) Breakout (fast) (b) Robotank (medium) (c) Kangaroo (medium)

Figure 5: Ablation study results on comparing the λ-return used in **DrMG** with an ablated baseline **DrMG+n-step** which replaces the λ-distributional return with an n-step distributional return, where we consider two cases $n = 3$ and $n = 5$.



(a) Breakout (fast) (b) SpaceInvaders (medium) (c) Centipede (slow)

Figure 6: Ablation study results for comparing our proposed vector hyperparameter set η (**DrMG**(η)) with a scalar set hyperparameters $\bar{\eta}$ (**DrMG**($\bar{\eta}$)). **DrMG**($\hat{\eta}$) performs comparable to **DrMG**(η) the *fast* and *medium* games and underperforms **DrMG**(η) in the *slow* game.

compare them in *Breakout*, *Krull*, and *SpaceInvaders*. The results are shown in Figure 4. We first notice that both versions of *non-adaptive* distributional λ-*return* work considerably well with the actor-critic IMPALA framework (outperform their corresponding DQN variants by large margins). Using **QR** can learn as smoothly as **DrMG** in *Krull* but for the other two domains, it underperforms **DrMG**, which shows the effectiveness of using meta-gradient to optimize distributional *return*. **DrMG**(η_1) outperforms **QR** in 2/3 games, but it underperforms **DrMG**(η_0, η_1), which demonstrates the superiority of meta-learning adaptive *return* in distributional meta-gradient algorithms.

λ-distributional *return* vs. n-step distributional *return* We investigate the effects of λ-/n-step distributional return by comparing **DrMG** with a meta-gradient RL baseline that replaces the λ-distributional *return* by an n-step *return* computed over the *return* distributions, denoted as **DrMG+n-step**. We consider two values for n in evaluation, where $n = 3$ and $n = 5$, i.e., resulting in baselines **DrMG+3-step** and **DrMG+5-step**. We adopt three domains, *DemonAttack*, *SpaceInvaders*, and *Centipede*. The results are shown in Figure 5. We can observe that replacing λ-return with n-step return substantially pulls down the rewards even in the *fast* game *Breakout*. This further confirms that backpropagating the distributional TD errors from λ-return is highly efficient to formulate the *adaptive* distributional *return* target.

Vector hyperparameters η vs. scalar hyperparameters $\bar{\eta}$ We then investigate the effects of vector hyperparameters η vs. scalar hyperparameters $\bar{\eta}$ by training **DrMG** with two kinds of adaptive distributional *return* parameterizations. When using scalar hyperparameters $\bar{\eta}$, the *return* for each quantile defined in Γ^{λ} shares the same discounting factor γ, trace λ, as well as cost for the quantile regression loss. When using a vector form of η, each quantile head enjoys a quantile-specific parameterization for the adaptive *return*. We evaluate **DrMG**(η) and **DrMG**($\bar{\eta}$) in *Breakout*, *SpaceInvaders* and *Centipede*, where the baselines are most distinguishable with **DrMG**($\bar{\eta}$). The results are presented in Figure 6. We observe that **DrMG**($\hat{\eta}$) performs fairly well in *Breakout* as well as *SpaceInvaders*, where other ablated baselines easily fall in the latter case. However, for the *slow* game *Centipede*, **DrMG**($\bar{\eta}$) noticeably underperforms its vector version **DrMG**($\bar{\eta}$).

6 Conclusion

We present a meta-gradient algorithm with distributional return named **DrMG**, which employs a critic that learns adaptive distributional return with parameters to be trained by meta-gradient derived from a white-box bi-level optimization scheme. We demonstrate that the quantile distributional *return* can seamlessly work with distributed actor-critic algorithms and learning such a distributional adaptive distributional *return* can offer more meaningful signals to update the meta parameters. We show **DrMG** is able to outperform its most related non-meta-learning baseline IMPALA and several decent RL baselines on a motivating Two-Colors Grid-World domain and the challenging video game benchmark Atari 2600. Our method is also very general and the distributional *return* we introduce hereby can be adapted to work with alternative meta-gradient RL methods, with the quantile distributional *return* being able to be flexibly replaced with other forms of distributions. We leave the work of integrating more variants of actor-critic algorithms or distributional *returns* in **DrMG** as our future work. It is also worth considering applying **DrMG** on more practical RL domains, such as pure value-based RL, offline R, and continuous control domains.

ETHICS AND REPRODUCIBILITY STATEMENT

Our work tackles synthetic digital game playing applications and therefore we believe our work is free from ethical problems. We present details on the distributional framework, Atari wrapper settings, and model hyperparameters in Appendix. We also include the code for the two important algorithmic components, networks, and two-level meta-update step function in our work.

REFERENCES

Maruan Al-Shedivat, Trapit Bansal, Yura Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. Continuous adaptation via meta-learning in nonstationary and competitive environments. In *6th International Conference on Learning Representations (ICLR)*, 2018.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems (NIPS)*, pp. 3981–3989, 2016.

Sungyong Baik, Myungsub Choi, Janghoon Choi, Heewon Kim, and Kyoung Mu Lee. Meta-learning with adaptive hyperparameters. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS), 2020, virtual*, 2020.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, volume 70, pp. 449–458, 2017.

Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80, pp. 1104–1113, 2018a.

Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pp. 2892–2901, 2018b.

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rlˆ2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80, pp. 1406–1415, 2018.

Chelsea Finn and Sergey Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. In *6th International Conference on Learning Representations (ICLR)*, 2018.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 1126–1135, 2017a.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. In *1st Annual Conference on Robot Learning (CoRL)*, volume 78 of *Proceedings of Machine Learning Research*, pp. 357–368, 2017b.

Sebastian Flennerhag, Yannick Schroecker, Tom Zahavy, Hado van Hasselt, David Silver, and Satinder Singh. Bootstrapped meta-learning. In *The 10th International Conference on Learning Representations (ICLR)*, 2022.

Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, and Thomas L. Griffiths. Recasting gradient-based meta-learning as hierarchical bayes. In *6th International Conference on Learning Representations (ICLR)*, 2018.

Abhishek Gupta, Russell Mendonca, Yuxuan Liu, Pieter Abbeel, and Sergey Levine. Meta-reinforcement learning of structured exploration strategies. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, (NeurIPS), Montréal, Canada*, pp. 5307–5316, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 770–778, 2016.

Timothy M. Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J. Storkey. Meta-learning in neural networks: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(9):5149–5169, 2022.

Junhyuk Oh, Matteo Hessel, Wojciech M. Czarnecki, Zhongwen Xu, Hado van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 1060–1070, 2020.

Nicol N. Schraudolph and Xavier Giannakopoulos. Online independent component analysis with local learning rate adaptation. In *Advances in Neural Information Processing Systems 12: Annual Conference on Neural Information Processing Systems (NIPS)*, pp. 789–795, 1999.

Richard Sutton. A history of meta-gradient: Gradient methods for meta-learning. *CoRR*, abs/2202.09701, 2022.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Richard S. Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pp. 171–176, 1992.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

Sebastian Thrun and Tom M. Mitchell. Learning one more thing. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1217–1225, 1995.

Vivek Veeriah, Matteo Hessel, Zhongwen Xu, Janarthanan Rajendran, Richard L. Lewis, Junhyuk Oh, Hado van Hasselt, David Silver, and Satinder Singh. Discovery of useful questions as auxiliary tasks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*, pp. 9306–9317, 2019.

Vivek Veeriah, Tom Zahavy, Matteo Hessel, Zhongwen Xu, Junhyuk Oh, Iurii Kemaev, Hado P van Hasselt, David Silver, and Satinder Singh. Discovery of options via meta-learned subgoals. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems (NeurIPS)*, volume 34, pp. 29861–29873, 2021.

Jane Wang, Zeb Kurth-Nelson, Hubert Soyer, Joel Z. Leibo, Dhruva Tirumala, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matt M. Botvinick. Learning to reinforcement learn. In *Proceedings of the 39th Annual Meeting of the Cognitive Science Society, CogSci*, 2017.

Yufei Wang, Qiwei Ye, and Tie-Yan Liu. Beyond exponentially discounted sum: Automatic learning of return function. *CoRR*, abs/1905.11591, 2019.

Olga Wichrowska, Niru Maheswaranathan, Matthew W. Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, volume 70, pp. 3751–3760, 2017.

Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore via meta-policy gradient. In *Proceedings of the 35th International Conference on Machine Learning, (ICML), Stockholmsmässan*, volume 80, pp. 5459–5468. PMLR, 2018a.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems (NeurIPS)*, volume 31, pp. 2396–2407, 2018b.

Zhongwen Xu, Hado Philip van Hasselt, Matteo Hessel, Junhyuk Oh, Satinder Singh, and David Silver. Meta-gradient reinforcement learning with an objective discovered online. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 15254–15264, 2020.

Tom Zahavy, Zhongwen Xu, Vivek Veeriah, Matteo Hessel, Junhyuk Oh, Hado van Hasselt, David Silver, and Satinder Singh. A self-tuning actor-critic algorithm. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 20913–20924, 2020.

Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems (NeurIPS)*, pp. 4649–4659, 2018.

Zeyu Zheng, Junhyuk Oh, Matteo Hessel, Zhongwen Xu, Manuel Kroiss, Hado van Hasselt, David Silver, and Satinder Singh. What can learned intrinsic rewards capture? In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119, pp. 11436–11446, 2020.

A APPENDIX

A.1 EXPERIMENTAL DOMAINS

A.1.1 TWO-COLORS GRID-WORLD

The two-colors grid-world is a 7×7 maze navigation problem. The agent is tasked to first collect a key and then navigate to the goal location with the key to open the door with the key. The agent is represented by green color, the two keys are represented by red and blue color, and the goal location is represented by light yellow color in Figure 1.

Reward there are three types of rewards issued to the agent: (ii) key collection rewards for both keys where one is 0.7 and the other is 0.3; (ii) goal reaching result, which is +1 when the agent has collected a key and -1 when the agent come to the goal prior to key collection; (iii) step-wise life penalty 0.01 to promote shortest navigation paths.

Reward toggling the key collection reward values toggle between the two keys after every $1e5$ environment steps, so that the agent needs to learn to identify keys for maximizing the reward, rather than converging to a trivial policy where the agent just learn to collect one fixed type of key without considering another, for learning sub-optimal policy.

Terminal condition the agent will enter a terminal state under the following four scenarios: (i) arrive goal with a key; (ii) arrive goal without collecting a key; (iii) have collected one key and attempt to collect the other key; (iv) have collected one key but attempt to collect the same key again.

State an eight-dimensional vector [*agent-y - goal-y, agent-x - goal-x, agent-y - redkey-y, agent-x - redkey-x, agent-y - bluekey-y, agent-x - bluekey-x, blue-collected, red-collected*]. The agent needs to be unaware of whether when or how the key rewards are toggled or specified.

Action set consist of four moving directions: $\{$left, right, up, down$\}$. When the agent attempts to transit to out-of-boundary locations in the maze, the game does not terminate but keep the agent unmoved.

Analysis the optimal policy is the one which could always identify the correct key rewards association and thus always the collect the higher rewarded key for opening the door. Thus the optimal episodic *return* is $1 + 1 = 2$. A key-agnostic agent is the one that randomly collects one of the key or just stick to one key for collection before reaching the goal location. Thus the expected episodic *return* for such key-agnostic agent is $1 + (0.7 + 0.3)/2 = 1.5$. The aim of this experiment how fast/well the RL agent behaves within each reward toggling interval.

A.1.2 ATARI 2600

We describe how we implement the wrapper for Atari environment. We create a base environment from *gym.wrappers.atari_preprocessing.AtariPreprocessing*, then embed it into a self-implemented wrapper class to perform the standardized operations like *frame skip* and *no-op start*. Upon reset, the wrapper first selects a random number between 0 and 30 to determine the number of *no-op* steps. Then it performs *no-op* reset by repeatedly taking action 0. In the *step* $(\cdot)$ function, we receive an action determined by the RL agent and then perform *frame skipping* and *frame stacking* to generate observations. The agent first performs frame skipping by repeat the action by 4 times. If the episode terminates within the repeats, the agent jumps out of the loop immediately. After *frame skipping*, the agent checks life loss and return next observation, cumulative reward within the repeated 4 steps, episode terminal flag and life loss. Note that an important issue here is no frame pooling is performed. We found that with frame pooling, the learning curve for the agent hardly progress to above reported MG's standards in quite a few *slow* learning games like *Centipede*, *BeamRider* and *Alien*. We use the same wrapper for the meta-gradient RL baselines.

A.2 CONFIGURATIONS

The versions for sensitive packages are as follows:

Package	**Version**
jax	0.2.21
rlax	0.1.1
optax	0.1.2
gym	0.21.0
moolib	0.0.9
opencv-python	4.6.0.66
python	3.8.10

The hyperparameter configurations for our method is as follows:

Parameter	**Value**
Atari Version	NoFrameskip-v4
Repeat Action Probability	0
Image Size	(84, 84)
Grayscale	Yes
Num. Action Repeats	4
Num. Frame Stacks	4
Full Action Space	No
End of Episode When Life Lost	No
Set 0 Discounting When Life Lost	Yes
Num. Training Frames	200M
Num. Batch Actors	4
Num. Batch Actor Env Size	32
Random No-ops	30
Seq-length	20
Batch size	32
Network	IMPALA deep
Use LSTM	No
Reward Clipping	[-1, 1]
Learning Rate	6e-4
Learning Rate Schedule	Anneal linearly to 0
Meta Learning Rate	6e-4
Meta Learning Rate Schedule	No
base RmsProp momentum	0.99
base RmsProp ϵ	1e-6
base grad norm clipping	1e4
meta grad norm clipping	1e4
meta Adam β_1	0.9
meta Adam β_2	0.999
meta Adam ϵ	1e-8
Distributional num quantiles	51
Distributional huber param	1
Distributional use vectorial λ, γ, w_Z	Yes
Distributional γ, λ, init	0.995
Distributional $\alpha, w_{\mathcal{H}}, w_\pi, w_Z$ init	1
Baseline cost	0.25
Policy cost	1.0
Entropy cost	0.01

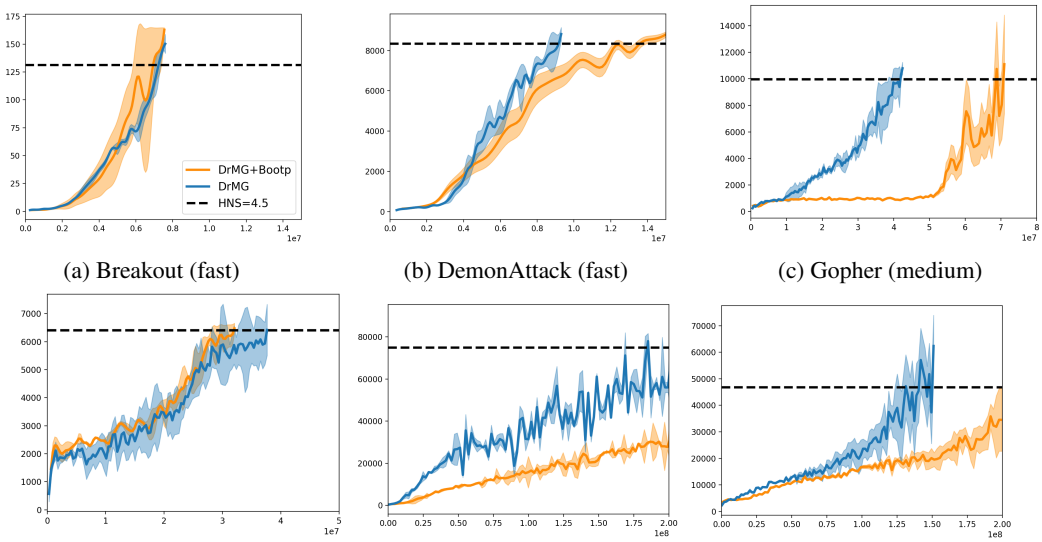
For distributional computation, we adopt *ray* for communication. We adopt batch environment wrapper from an open-source distributed reinforcement learning framework *moolib* to accelerate actor computation. The detailed configuration of the distributional framework is as follows:

Parameter	**Value**
Num. CPUs for moolib	25
Num. CPUs for Evaluator	2
Num. GPUs for Evaluator	0.1
Num. CPUs for Actor	3
Num. GPUs for Actor	0.2
Num. CPU for Learner	5
Num. CPU for Learner	1.0
Num. train itr to push params	1
Num. rollout step to pull params	1

A.3 Additional Ablation Study Results

Adapting Bootstrapped Meta Learning (BMG) into DrMG We additionally investigate effects of combining the bootstrapping techniques introduced in the state-of-the-art meta-gradient work **BMG** (Flennerhag et al., 2022) with the vanilla version of our method **DrMG** without bootstrapping. We denote our method with bootstrapping as **DrMG+Bootp**.

Note that the main difference between **DrMG+Bootp** and vanilla **DrMG** is that **DrMG+Bootp** considers a less frequent meta-gradient update that considers feedback from a long horizon of optimizer steps, whereas **DrMG** considers a more frequent meta-gradient update that adapts the meta parameters based on per optimizer step upon per mini-batch of transitions. For example, when we employ bootstrapping with 3 inner steps, the **DrMG+Bootp** optimizes η with meta-gradient once with every 3 mini-batches of data (to compute the bootstrapped inner steps with horizon 3), whereas **DrMG** optimizers η once upon each mini-batch of data (since the inner step updates the optimizer for a single iteration).



(a) Breakout (fast) (b) DemonAttack (fast) (c) Gopher (medium)

(d) Krull (medium) (e) BeamRider (slow) (f) Centipede (slow)

Figure 7: Ablation study results on combining bootstrapping with **DrMG** (denoted as **DrMG+Bootp**), where we consider a bootstrapping inner steps of 3 following the BMG paper.

We evaluate **DrMG** and **DrMG+Bootp** on six representative games. We observe that **DrMG** ($\hat{\eta}$) performs fairly well sometimes, e.g., on *Breakout*, *DemonAttack* and *Krull*, It fails on partial games,

e.g., *Gopher* and *Centipede*. We also observed the training instability when using bootstrapping. To conclude, bootstrapping can be helpful on partial games, but it is also not straightforward to tune/adapt to DrMG. We leave fully reproducing **BMG** on Atari 2600 as future work.

A.4 Implementation

A.4.1 Implementation of Quantile Distributional Loss

We present the pseudocode for two important functions in our algorithm as follows.

```
import jax
from jax import lax
from jax import numpy as jnp
from typing import Any, Tuple, Optional
from collections import namedtuple
VTraceOutput = collections.namedtuple('vtrace_output', ['errors', 'pg_advantage'])


def compute_loss(
    dist_src: Array,
    taus: Array,
    dist_target: Array,
    huber_param: float,
    stop_target_gradients: bool,
) -> Array:
  """ Compute huber quantile regression loss loss between
  two discrete quantile-valued distributions."""

  delta = dist_target[:, None, :] - dist_src[:, :, None]
  delta_neg = (delta < 0.).astype(jnp.float32)
  delta_neg = lax.select(stop_target_gradients,
                         sg(delta_neg), delta_neg)
  weight = jnp.abs(taus[None, :, None] - delta_neg)
  loss = clipping.huber_loss(delta, huber_param)
  loss *= weight

  return jnp.mean(loss, axis=-1)


def quantile_regression_td_learning(
        support: Array,
        z_logits_tm1: Array,
        r_t: Array,
        discount_t: Array,
        z_logits_t: Array,
        rho_tm1: Array,
        eta: namedtuple,
        stop_target_gradients: bool,
        huber_param: float,
) -> VTraceOutput:
    """Distributional TD(lambda) learning."""

    dist_target = r_t[:, None] + eta.gamma * discount_t[:, None] \
                * lax.stop_gradient(z_logits_t)
    dist_target = lax.select(stop_target_gradients,
                             sg(dist_target), dist_target)

    clipped_rhos_tm1 = sg(jnp.minimum(1.0, rho_tm1)) * eta.alpha + \
                (1 - eta.alpha) * lax.stop_gradient(rho_tm1)

    errors = []
    err_r = 0
    td_errors = clipped_rhos_tm1[:, None] * (dist_target -
                lax.stop_gradient(z_logits_tm1))
```

```
    for i in reversed(range(z_logits_t.shape[0])):
        err_r = td_errors[i] + discount_t[i] * clipped_rhos_tm1[i] * \
                eta.lambda_ * eta.gamma * err_r
        errors.insert(0, err_r)

    errors_r = jnp.array(errors_r)
    target_dist = errors_r + lax.stop_gradient(z_logits_tm1)

    target_dist = jax.lax.select(stop_target_gradients,
                                 lax.stop_gradient(target_dist), target_dist)
    quantile_loss = compute_loss(z_logits_tm1, support, target_dist,
                                 huber_param, stop_target_gradients)

    dist_bootstrap = jnp.concatenate([
        eta.lambda_ * target_dist[1:] + (1 - eta.lambda_) * \
        lax.stop_gradient(z_logits_tm1[1:]),
        lax.stop_gradient(z_logits_t[-1:]),
    ], axis=0)

    dist_estimate = r_t[:, None] + eta.gama * discount_t[:, None] * dist_bootstrap
    dist_estimate = jax.lax.select(stop_target_gradients,
                lax.stop_gradient(dist_estimate),  dist_estimate)
    pg_advantages = jnp.mean(dist_estimate, axis=-1) - \
                    lax.stop_gradient(jnp.mean(z_logits_tm1, axis=-1))
    pg_advantages = pg_advantages * clipped_rhos_tm1
    return VTraceOutput(errors=quantile_loss, pg_advantage=pg_advantages)
```

A.4.2 Implementation of Two-level Meta-Gradient Update

```
import jax
from jax import lax
import optax
from jax import numpy as jnp
from typing import Any, Tuple, Optional
from collections import namedtuple
EtaOutput = collections.namedtuple('eta_output', ['gamma', 'lambda_', 'alpha'])


class DRMGAgent:
    '''Base class for DrMG Agents.'''
    def __init__(self, ...):
        raise NotImplementedError

    def actor_step(self, ...):
        raise NotImplementedError

    def two_steps_update(self,
         params: Params,
         meta_params: Params,
         learner_state: OptimLearnerState,
         meta_learner_state: OptimLearnerState,
         batch_base: Any,
         batch_val: Any,
         rng_key: RngKey,
         meta_rng_key: RngKey,
         network: Network,
         meta_network: Network,
         optimizer: Optimizer,
         meta_optimizer: Optimizer,
         ):
        # Clip the gradients according to global max norm.
        (unused_outer_loss, (new_params, new_opt_state, logs)), meta_grads = \
            jax.value_and_grad(
```

```
            self._inner_update_and_outer_loss, has_aux=True
        )(meta_params, params, rng_key, meta_rng_key,
          learner_state, batch_base, batch_val, network,
          meta_network, optimizer)
    meta_grads = jax.lax.pmean(meta_grads, axis_name=_PMAP_AXIS_NAME)

    meta_updates, new_meta_opt_state = meta_optimizer.update(
        meta_grads, meta_learner_state.opt_state, meta_params
    )

    new_meta_params = optax.apply_updates(meta_params, meta_updates)
    logs.update({
        'meta_grad_norm_unclipped': optax.global_norm(meta_grads),
        'meta_grad_norm_applied': optax.global_norm(meta_updates),
    })
    new_learner_state = OptimLearnerState(
        opt_state=new_opt_state,
        opt_count=learner_state.opt_count + 1)
    new_meta_learner_state = OptimLearnerState(
        opt_state=new_meta_opt_state,
        opt_count=meta_learner_state.opt_count + 1)
    return new_params, new_meta_params, new_learner_state, \
           new_meta_learner_state, logs

def _inner_update_and_outer_loss(self,
     meta_params: Params,
     params: Params,
     rng_key: RngKey,
     learner_state: OptimLearnerState,
     batch_base: Transition,
     batch_val: Transition,
     network: Network,
     meta_network: Network,
     optimizer: Optimizer,
     ):
    ######### inner update
    (_, (logs,)), grads = jax.value_and_grad(
        self._drmg_loss, has_aux=True
    )(params, batch_base, rng_key, network,
      meta_network, True, meta_params,  )

    grads = jax.lax.pmean(grads, axis_name=_PMAP_AXIS_NAME)
    grad_norm = jnp.sqrt(
        jax.tree_util.tree_reduce(
            lambda x, y: x + jnp.sum(y ** 2), grads, initializer=0))
    mult = jnp.minimum(1, FLAGS.grad_clip / (1e-7 + grad_norm))
    grads = jax.tree_util.tree_map(lambda z: mult * z, grads)

    # Gather gradients from all devices.
    updates, new_opt_state = optimizer.update(
        grads, learner_state.opt_state, params)
    new_params = optax.apply_updates(params, updates)

    logs.update(
        {'inner_grad_norm_unclipped': optax.global_norm(grads),
         'inner_grad_norm_applied': optax.global_norm(updates),
         })
    ######### outer updates
    outer_loss, (_, meta_logs, _, _) = self._drmg_loss(
        new_params, batch_val, rng_key, network,
        meta_network, False, None)
    logs.update(meta_logs)

    return outer_loss, (new_params, new_opt_state, logs)
```

```
def _drmg_loss(self,
        params: Params,
        batch: Transition,
        rng_key: RngKey,
        network: Network,
        meta_network: Network,
        is_inner: bool,
        meta_params: Any = None,):
    _, net_outputs = network.forward(
        rng_key=rng_key,
        params=params,
        state=batch.state,
        for_training=True
    )
    if is_inner:
        eta = meta_network.forward(
            meta_params=meta_params,
            state=batch.reward,
        )
        gamma, lambda_ = eta.discount, eta.lambda_
        policy_cost =  eta.policy_cost * FLAGS.pg_cost
        value_cost  = eta.value_cost * FLAGS.value_cost
        entropy_cost = eta.value_cost * FLAGS.entropy_cost
        eta = EtaOutput(
          gamma=gamma, lambda_=lambda_,
          alpha=eta.alpha,
        )
    else:
        gamma, lambda_ = FLAGS.discounting, 1.0
        policy_cost, value_cost, entropy_cost = 1.0, 1.0, 1.0
        eta = EtaOutput(
            gamma=gamma, lambda_=lambda_,
            alpha=1,
        )
    # type annotation
    rewards_t = batch.reward[1:]
    rewards_t = jnp.clip(
        rewards_t, -FLAGS.reward_clip, FLAGS.reward_clip
    )
    v_t = net_outputs.value[1:]
    v_tm1 = net_outputs.value[:-1]
    discounts_t = (1 - batch.is_done.astype(jnp.float32))[1:]

    # Remove bootstrap timestep
    actor_outputs_tm1 = jax.tree_map(lambda t: t[:-1], batch.agent_output)
    learner_outputs_tm1 = jax.tree_map(lambda t: t[:-1], net_outputs)
    # Handful shortcuts
    pi_logits_tm1 = learner_outputs_tm1.policy_logits
    mu_logits_tm1 = actor_outputs_tm1.policy_logits
    action_tm1 = jax.tree_map(lambda t: t[:-n_step], batch.action)

    rhos_tm1 = rlax.categorical_importance_sampling_ratios(
        pi_logits_tm1, mu_logits_tm1, action_tm1)
    # vmap vtrace_td_error_and_advantage to take/return [T, B, ...].
    distributional_td_func = jax.vmap(
        quantile_regression_td_learning,
        in_axes=[None] + [1] * 5 + [None] * 3,
        out_axes=1)
    support = (jnp.arange(0, FLAGS.distributional.n_atoms) + 0.5
                    ) / float(FLAGS.distributional.n_atoms)

    quantile_return = distributional_td_func(
        support, v_tm1, rewards_t, discounts_t, v_t, rhos_tm1,
        eta, not is_inner, 1.0)
    pg_advs_tm1 = quantile_return.pg_advantage
```

```
        pg_loss = jax.vmap(
            policy_gradient_loss, in_axes=[1] * 3
        )(pi_logits_tm1, action_tm1, pg_advs_tm1)  # [B, T]

        pg_loss = jnp.sum(pg_loss)
        value_loss = jnp.sum(quantile_return.errors)
        entropy_loss = jax.vmap(
            entropy_loss_fn, in_axes=1
        )(pi_logits_tm1)
        entropy_loss = jnp.sum(entropy_loss)  # []

        # Weight the losses
        total_loss = pg_loss * policy_cost
        total_loss += value_loss * value_cost
        total_loss += entropy_loss * entropy_cost

        logs = {
            prefix + 'pg_loss': pg_loss * policy_cost,
            prefix + 'value_loss': value_loss * value_cost,
            prefix + 'entropy_loss': entropy_loss * entropy_cost,
        return  total_loss, (logs)
```

A.4.3 Implementation of Base and Meta Networks

Atari Network

```
import jax
import rlax
import haiku as hk
from jax import numpy as jnp
from typing import Any, Tuple, Optional, Union
from collections import namedtuple
from functools import partial
RngKey = jnp.ndarray
Array = Union[np.ndarray, jnp.ndarray]
PVOutput = collections.namedtuple('PVOutput', ['policy_logits', 'value'])
Action = Any


class AtariPVHead(hk.Module):
  def __init__(
    self,
    num_actions: int,
    fc_size: int,
    orth_linear: bool = False,
    num_value_heads: Optional[int] = 1,
    name: Optional[str] = None,
  ):
    super().__init__(name=name)
    self._num_actions = num_actions
    self._fc_size = fc_size
    self._orth_linear = orth_linear
    self._num_value_heads = num_value_heads

  def __call__(self, torso_output: Array) -> Tuple[Array, Array]:
    Linear = partial(
      hk.Linear,
      w_init=hk.initializers.Orthogonal(),
      b_init=hk.initializers.Constant(0)
    ) if self._orth_linear else hk.Linear

    torso_output = Linear(self._fc_size)(torso_output)
    torso_output = jax.nn.relu(torso_output)
```

```
    policy_logits = Linear(self._num_actions)(torso_output)
    value = Linear(self._num_value_heads)(torso_output)
    if self._num_value_heads == 1:
      value = jnp.squeeze(value, axis=-1)
    return policy_logits, value


class ResidualBlock(hk.Module):
  def __init__(
    self,
    num_channels: int,
    data_format: str = 'NCHW',
    name: Optional[str] = None
  ):
    super().__init__(name=name)
    self._num_channels = num_channels
    self._data_format = data_format

  def __call__(self, x: Array) -> Array:
    main_branch = hk.Sequential(
      [
        jax.nn.relu,
        hk.Conv2D(
          self._num_channels,
          kernel_shape=[3, 3],
          stride=[1, 1],
          padding='SAME',
          data_format=self._data_format, with_bias=with_bias,
        ),
        jax.nn.relu,
        hk.Conv2D(
          self._num_channels,
          kernel_shape=[3, 3],
          stride=[1, 1],
          padding='SAME',
          data_format=self._data_format, with_bias=with_bias,
        ),
      ]
    )
    return main_branch(x) + x


class AtariDeepTorso(hk.Module):
  def __init__(
    self,
    block_config: Any = [(16, 2), (32, 2), (32, 2)],
    data_format: str = 'NCHW',
    orth_linear: bool = False,
    name: Optional[str] = None
  ):
    super().__init__(name=name)
    self._data_format = data_format
    self._orth_linear = orth_linear
    self.block_config = block_config

  def __call__(self, x: Array) -> Array:
    torso_out = x
    for i, (num_channels, num_blocks) in enumerate(self.block_config):
      conv = hk.Conv2D(
        num_channels,
        kernel_shape=[3, 3],
        stride=[1, 1],
        padding='SAME',
        data_format=self._data_format,  with_bias=with_bias,
      )
```

```
      torso_out = conv(torso_out)
      if self._data_format == 'NHWC':
        torso_out = hk.max_pool(
          torso_out,
          window_shape=[1, 3, 3, 1],
          strides=[1, 2, 2, 1],
          padding='SAME',
        )
      elif self._data_format == 'NCHW':
        torso_out = hk.max_pool(
          torso_out,
          window_shape=[1, 1, 3, 3],
          strides=[1, 1, 2, 2],
          padding='SAME',
        )
      for j in range(num_blocks):
        block = ResidualBlock(
          num_channels, self._data_format, name='residual_{}_{}'.format(i, j)
        )
        torso_out = block(torso_out)

    torso_out = jax.nn.relu(torso_out)
    torso_out = hk.Flatten()(torso_out)
    return torso_out


class AtariHeadModule(hk.Module):
  def __init__(
    self,
    num_actions: int,
    data_format: str = 'NCHW',
    fc_size: int = 256,
    obs_scaling: float = 1. / 255,
    orth_linear: bool = False,
    num_value_heads: Optional[int] = 1,
    name: Optional[str] = None,
  ):
    super().__init__(name=name)
    self._num_actions = num_actions
    self._data_format = data_format
    self._use_resnet = use_resnet
    self._obs_scaling = obs_scaling
    self._orth_linear = orth_linear
    self._return_torso = return_torso
    self._is_rnd_head = is_rnd_head
    self._torso_module = AtariDeepTorso(
      data_format=self._data_format,
      orth_linear=orth_linear)
    self._head= AtariPVHead(
        self._num_actions,
        fc_size=fc_size,
        orth_linear=orth_linear,
        num_value_heads=num_value_heads)

  def __call__(self, obs: Array)-> Tuple[Array, Array]:
    """Compute the policy logits and value from the obs."""
    obs = obs * self._obs_scaling  # type: Array
    if obs.ndim == 4:
      # obs in shape [B, ...]
      torso_out = self._torso_module(obs)  # type: ignore
      out= self._head(torso_out)
    elif obs.ndim == 5:
      # obs in shape [T, B, ...]
      torso_out = hk.BatchApply(self._torso_module)(obs)
      out = self._head(torso_out)
```

```
    return out


class BaseNet(Network):
  def __init__(
    self,
    num_actions: Any,
    data_format: str = 'NCHW',
    fc_size: int = 256,
    obs_scaling: float = 1. / 255,
    orth_linear: bool = False,
    num_value_heads: Optional[int] = 1,
    name: Optional[str] = None
  ):
    self.num_actions = num_actions
    self._data_format = data_format
    module = functools.partial(
        AtariHeadModule,
        num_value_heads=num_value_heads,
        data_format=self._data_format,
        fc_size=fc_size,
        obs_scaling=obs_scaling,
        orth_linear=orth_linear,
        name=name,
      )
    self._net = hk.without_apply_rng(
      hk.transform(lambda x: module(num_actions)(x))
    )

  def init_params(self, rng_key: RngKey) -> Params:
    dummy_inputs = jnp.zeros((1, 1, 4, 84, 84))
    params = self._net.init(rng_key, dummy_inputs)
    return params

  @functools.partial(jax.jit, static_argnums=0)
  def forward(
    self,
    rng_key: RngKey,
    params: Params,
    state: State,
    **kwargs: Any
  ) -> Tuple[Action, PVOutput]:
    policy_logits, value = self._net.apply(params, state)
    distribution = rlax.softmax()
    action = distribution.sample(rng_key, policy_logits)
    return action, PVOutput(policy_logits=policy_logits, value=value)
```

Meta Network

```
import jax
import rlax
import haiku as hk
from jax import numpy as jnp
from typing import Any, Tuple, Optional, Union
from collections import namedtuple
from functools import partial
RngKey = jnp.ndarray
Array = Union[np.ndarray, jnp.ndarray]
MetaOutput = collections.namedtuple('MetaOutput',
                                    ['gamma', 'lambda_', 'alpha',
                                     'value_cost', 'entropy_cost',
                                      'policy_cost',])
Action = Any
```

```python
class MetaNetwork(Network):
  def __init__(
      self,
      learning_rate: float,
      entropy_cost: float,
      value_cost: float,
      policy_cost: float,
      alpha: float,
      discounting_default: float=0.99,
      trace_default: float=1.0,
      return_shape: list=[],
      loss_shape: list=[],
      name: Optional[str] = None
  ):
    module = functools.partial(
      MetaScalarModule_,
      learning_rate=learning_rate,
      entropy_cost=entropy_cost,
      value_cost=value_cost,
      policy_cost=policy_cost,
      alpha=alpha,
      discounting_default=discounting_default,
      trace_default=trace_default,
      return_shape=return_shape,
      loss_shape=loss_shape,
    )
    self._net = hk.without_apply_rng(
      hk.transform(lambda x: module()(x))
    )

  def init_params(self, rng_key: RngKey) -> StacxOutput:
    dummy_input = jnp.zeros((1, 2, 4, 84, 84,))
    params = self._net.init(rng_key, dummy_input)
    return params

  @functools.partial(jax.jit, static_argnums=0)
  def forward(
    self,
    meta_params: Params,
    state: Any,
    **kwargs: Any
  ) -> StacxOutput:
    stacx_output = self._net.apply(meta_params, state)
    return stacx_output


class MetaScalarModule_(hk.Module):
  def __init__(
    self,
    learning_rate: float,
    entropy_cost: float,
    value_cost: float,
    policy_cost: float,
    alpha: float = 1.0,
    discounting_default: float = 0.99,
    trace_default: float = 1.0,
    return_shape: list=[],
    loss_shape: list = [],
    name: Optional[str] = None
  ):
    super().__init__(name=name)
    self.discounting = inverse_sigmoid(discounting_default)
    self.lambda_ = inverse_sigmoid(trace_default)
    self.learning_rate = inverse_sigmoid(learning_rate)
```

```
    self.entropy_cost = inverse_sigmoid(entropy_cost)
    self.value_cost = inverse_sigmoid(value_cost)
    self.policy_cost = inverse_sigmoid(policy_cost)
    self.alpha = inverse_sigmoid(alpha)
    self.loss_shape = loss_shape
    self.return_shape = return_shape

  def __call__(self, state) -> Tuple[Array, Array]:
    discount = hk.get_parameter("discount", self.return_shape,
                   init=hk.initializers.Constant(self.discounting))
    lambda_ = hk.get_parameter("lambda", self.return_shape,
                   init=hk.initializers.Constant(self.lambda_))
    entropy_cost = hk.get_parameter("entropy_cost", self.loss_shape,
                   init=hk.initializers.Constant(self.entropy_cost))
    value_cost = hk.get_parameter("value_cost", self.loss_shape,
                   init=hk.initializers.Constant(self.value_cost))
    policy_cost = hk.get_parameter("policy_cost", self.loss_shape,
                   init=hk.initializers.Constant(self.policy_cost))
    alpha = hk.get_parameter("alpha", self.return_shape,
                   init=hk.initializers.Constant(self.alpha))
    return MetaOutput(
      value_cost=jax.nn.sigmoid(value_cost),
      entropy_cost=jax.nn.sigmoid(entropy_cost),
      policy_cost=jax.nn.sigmoid(policy_cost),
      discount=jax.nn.sigmoid(discount),
      lambda_=jax.nn.sigmoid(lambda_),
      alpha=jax.nn.sigmoid(alpha), )
```