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 comes 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 at the goal with a key; (ii) arrive at the 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 keeps the agent unmoved.

Analysis the optimal policy is the one which could always identify the correct key rewards association and thus always collect the higher rewarded key for opening the door. Thus the optimal episodic *return* is $1 + 1 = 2$. A key-agnostic agent is one that randomly collects one of the keys or just sticks 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

Wrapper We describe how we implement the wrapper for the 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 repeating 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 returns the 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.

Discussion on why BMG is excluded from our MGRL baseline We hereby explain why **BMG** is not considered as our comparison baseline from empirical and methodological perspectives.

First, from the empirical perspective, we wish to highlight that the effort of adjusting the setting of BMG to add it as a baseline for a fair comparison with our work is non-trivial, mainly due to the following factors:

1. BMG uses a different ALE wrapper, which makes the task different from the one adopted by our work. The main change is changing the grayscale input (84×84) into RGB input (160×210×3).
2. BMG uses a much larger ResNet, where the channel depth changed from (16, 32, 32) to (64,128,128,64).
3. BMG uses replay, wherein in each batch, the replay to online ratio is 2:1, whereas MG/STACX/DRMG do not employ replay. BMG with replay is also trained under a smaller batch size than MG/STACX/DRMG, where the batch sizes for the former and latter algorithms are 18 and 32, respectively.

The last point is somewhat more troublesome than the first two because it requires heavy engineering effort to fully integrate a reply buffer into the code base of MG/STACX/DrMG when BMG does not have open-source code. Meanwhile, running the BMG experiment with experience replay and a small batch size would significantly increase the run time for the method. It also remains unclear whether an IMPALA variant with replay could be considered a fair baseline to compare with the other meta-gradient methods in our work (MG/STACX/DrMG) that do not perform data reuse.

Second, from the methodological point of view, BMG and our work tackle orthogonal directions in meta-gradient RL problem, i.e., BMG is innovated on the solver level, focusing on how to better optimize the bi-level optimization with bootstrapping, whereas our work focuses on modeling the adaptive distributional return which is solved by a standard meta-gradient solver (with two-step inner-outer update). Nevertheless, we adapt the main bootstrapping techniques of BMG into our work and present comparison results between the bootstrapped DrMG baseline and our method in Appendix A.3.

A.2 CONFIGURATIONS

For distributional computation, we adopt *ray* for communication. We adopt a 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

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

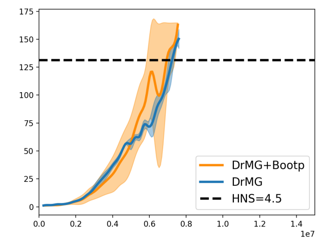
The versions for sensitive packages are as follows:

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

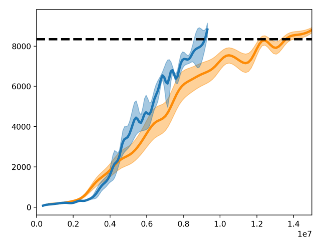
A.3 Additional Ablation Study Results

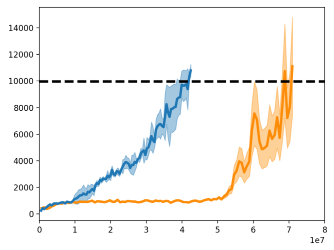
Adapting Bootstrapped Meta-Learning (BMG) into DrMG We additionally investigate the effects of combining the bootstrapping techniques introduced in the state-of-the-art meta-gradient work **BMG** (Flennerhag et al., 2022) with the vanilla version of our method **DrMG** without bootstrapping. We denote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

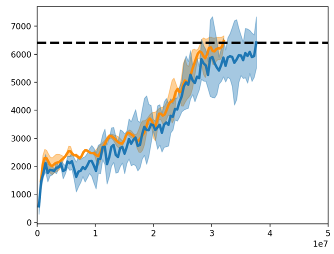
Note that the main difference between **DrMG+Bootp** and vanilla **DrMG** is that **DrMG+Bootp** considers a less frequent meta-gradient update that considers feedback from a long horizon of optimizer steps, whereas **DrMG** considers a more frequent meta-gradient update that adapts the meta parameters 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).

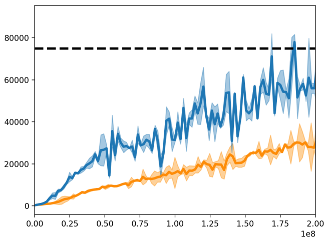


(a) Breakout (fast)


(b) DemonAttack (fast)


(c) Gopher (medium)


(d) Krull (medium)


(e) BeamRider (slow)

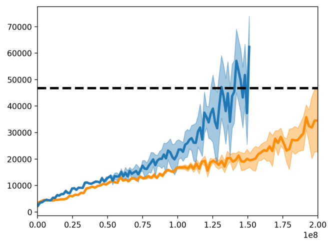

(f) Centipede (slow)

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 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,
         ):
        (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, 'inner_', meta_params,   )

        grads = jax.lax.pmean(grads, axis_name=_PMAP_AXIS_NAME)

        # 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 update
        outer_loss, (meta_logs) = self._drmg_loss(
            new_params, batch_val, rng_key, network,
            meta_network, False, 'outer_', 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,
            prefix: str,
            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 = FLAGS.pg_cost, \
                FLAGS.value_cost, FLAGS.entropy_cost
            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,
```

```python
          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


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), )
```