```python
import argparse
import collections
import functools
import json
import os
import pathlib
import sys
import time

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['MUJOCO_GL'] = 'egl'

import numpy as np
import tensorflow as tf
from tensorflow.keras.mixed_precision import experimental as prec

tf.get_logger().setLevel('ERROR')

from tensorflow_probability import distributions as tfd

sys.path.append(str(pathlib.Path(__file__).parent))

import models
import tools
import wrappers


def define_config():
  config = tools.AttrDict()
  # General.
  config.logdir = pathlib.Path('.')
  config.seed = 0
  config.steps = 5e6
  config.eval_every = 1e4
  config.log_every = 1e3
  config.log_scalars = True
  config.log_images = True
  config.gpu_growth = True
  config.precision = 16
  # Environment.
  config.task = 'dmc_walker_walk'
  config.envs = 1
  config.parallel = 'none'
  config.action_repeat = 2
  config.time_limit = 1000
  config.prefill = 5000
  config.eval_noise = 0.0
  config.clip_rewards = 'none'
  # Model.
  config.deter_size = 200
  config.stoch_size = 30
  config.num_units = 400
  config.dense_act = 'elu'
  config.cnn_act = 'relu'
  config.cnn_depth = 32
  config.pcont = False
  config.free_nats = 3.0
  config.kl_scale = 1.0
  config.pcont_scale = 10.0
  config.weight_decay = 0.0
  config.weight_decay_pattern = r'.*'
  # Training.
  config.batch_size = 50
  config.batch_length = 50
  config.train_every = 1000
  config.train_steps = 100
```

```
  config.pretrain = 100
  config.model_lr = 6e-4
  config.value_lr = 8e-5
  config.actor_lr = 8e-5
  config.grad_clip = 100.0
  config.dataset_balance = False
  # Behavior.
  config.discount = 0.99
  config.disclam = 0.95
  config.horizon = 15
  config.action_dist = 'tanh_normal'
  config.action_init_std = 5.0
  config.expl = 'additive_gaussian'
  config.expl_amount = 0.3
  config.expl_decay = 0.0
  config.expl_min = 0.0
  return config


class Dreamer(tools.Module):

  def __init__(self, config, datadir, actspace, writer):
    self._c = config
    self._actspace = actspace
    self._actdim = actspace.n if hasattr(actspace, 'n') else actspace.shape[0]
    self._writer = writer
    self._random = np.random.RandomState(config.seed)
    with tf.device('cpu:0'):
      self._step = tf.Variable(count_steps(datadir, config), dtype=tf.int64)
    self._should_pretrain = tools.Once()
    self._should_train = tools.Every(config.train_every)
    self._should_log = tools.Every(config.log_every)
    self._last_log = None
    self._last_time = time.time()
    self._metrics = collections.defaultdict(tf.metrics.Mean)
    self._metrics['expl_amount']  # Create variable for checkpoint.
    self._float = prec.global_policy().compute_dtype
    self._strategy = tf.distribute.MirroredStrategy()
    with self._strategy.scope():
      self._dataset = iter(self._strategy.experimental_distribute_dataset(
          load_dataset(datadir, self._c)))
      self._build_model()

  def __call__(self, obs, reset, state=None, training=True):
    step = self._step.numpy().item()
    tf.summary.experimental.set_step(step)
    if state is not None and reset.any():
      mask = tf.cast(1 - reset, self._float)[:, None]
      state = tf.nest.map_structure(lambda x: x * mask, state)
    if self._should_train(step):
      log = self._should_log(step)
      n = self._c.pretrain if self._should_pretrain() else self._c.train_steps
      print(f'Training for {n} steps.')
      with self._strategy.scope():
        for train_step in range(n):
          log_images = self._c.log_images and log and train_step == 0
          self.train(next(self._dataset), log_images)
      if log:
        self._write_summaries()
    action, state = self.policy(obs, state, training)
    if training:
      self._step.assign_add(len(reset) * self._c.action_repeat)
    return action, state

  @tf.function
  def policy(self, obs, state, training):
```

This is part of imagining trajectories

imported as models.RSSM in build_model()

Recurrent State Space Model in Fig. 5

```
    if state is None:
      latent = self._dynamics.initial(len(obs['image']))
      action = tf.zeros((len(obs['image']), self._actdim), self._float)
    else:
      latent, action = state
    embed = self._encode(preprocess(obs, self._c))
    latent, _ = self._dynamics.obs_step(latent, action, embed)
    feat = self._dynamics.get_feat(latent)
    if training:
      action = self._actor(feat).sample()
    else:
      action = self._actor(feat).mode()
    action = self._exploration(action, training)
    state = (latent, action)
    return action, state

  def load(self, filename):
    super().load(filename)
    self._should_pretrain()

  @tf.function()
  def train(self, data, log_images=False):
    self._strategy.experimental_run_v2(self._train, args=(data, log_images))

  def _train(self, data, log_images):
    with tf.GradientTape() as model_tape:
      embed = self._encode(data)
      post, prior = self._dynamics.observe(embed, data['action'])
      feat = self._dynamics.get_feat(post)
      image_pred = self._decode(feat)
      reward_pred = self._reward(feat)
      likes = tools.AttrDict()
      likes.image = tf.reduce_mean(image_pred.log_prob(data['image']))
      likes.reward = tf.reduce_mean(reward_pred.log_prob(data['reward']))
      if self._c.pcont:
        pcont_pred = self._pcont(feat)
        pcont_target = self._c.discount * data['discount']
        likes.pcont = tf.reduce_mean(pcont_pred.log_prob(pcont_target))
        likes.pcont *= self._c.pcont_scale
      prior_dist = self._dynamics.get_dist(prior)
      post_dist = self._dynamics.get_dist(post)
      div = tf.reduce_mean(tfd.kl_divergence(post_dist, prior_dist))
      div = tf.maximum(div, self._c.free_nats)
      model_loss = self._c.kl_scale * div - sum(likes.values())
      model_loss /= float(self._strategy.num_replicas_in_sync)

    with tf.GradientTape() as actor_tape:
      imag_feat = self._imagine_ahead(post)
      reward = self._reward(imag_feat).mode()
      if self._c.pcont:
        pcont = self._pcont(imag_feat).mean()
      else:
        pcont = self._c.discount * tf.ones_like(reward)
      value = self._value(imag_feat).mode()
      returns = tools.lambda_return(
          reward[:-1], value[:-1], pcont[:-1],
          bootstrap=value[-1], lambda_=self._c.disclam, axis=0)
      discount = tf.stop_gradient(tf.math.cumprod(tf.concat(
          [tf.ones_like(pcont[:1]), pcont[:-2]], 0), 0))
      actor_loss = -tf.reduce_mean(discount * returns)
      actor_loss /= float(self._strategy.num_replicas_in_sync)

    with tf.GradientTape() as value_tape:
      value_pred = self._value(imag_feat)[:-1]
      target = tf.stop_gradient(returns)
      value_loss = -tf.reduce_mean(discount * value_pred.log_prob(target))
```

Representation model $p_\theta(s_t | \dots)$

previous state, previous action, embedding

Relate to eq (10) in paper

$\ln[q(o_t|s_t)]$ & $\ln[q(r_t|s_t)]$ contains

KL div

```
      value_loss /= float(self._strategy.num_replicas_in_sync)

    model_norm = self._model_opt(model_tape, model_loss)
    actor_norm = self._actor_opt(actor_tape, actor_loss)
    value_norm = self._value_opt(value_tape, value_loss)

    if tf.distribute.get_replica_context().replica_id_in_sync_group == 0:
      if self._c.log_scalars:
        self._scalar_summaries(
            data, feat, prior_dist, post_dist, likes, div,
            model_loss, value_loss, actor_loss, model_norm, value_norm,
            actor_norm)
      if tf.equal(log_images, True):
        self._image_summaries(data, embed, image_pred)

  def _build_model(self):
    acts = dict(
        elu=tf.nn.elu, relu=tf.nn.relu, swish=tf.nn.swish,
        leaky_relu=tf.nn.leaky_relu)
    cnn_act = acts[self._c.cnn_act]
    act = acts[self._c.dense_act]
    self._encode = models.ConvEncoder(self._c.cnn_depth, cnn_act)
    self._dynamics = models.RSSM(
        self._c.stoch_size, self._c.deter_size, self._c.deter_size)
    self._decode = models.ConvDecoder(self._c.cnn_depth, cnn_act)
    self._reward = models.DenseDecoder((), 2, self._c.num_units, act=act)
    if self._c.pcont:
      self._pcont = models.DenseDecoder(
          (), 3, self._c.num_units, 'binary', act=act)
    self._value = models.DenseDecoder((), 3, self._c.num_units, act=act)
    self._actor = models.ActionDecoder(
        self._actdim, 4, self._c.num_units, self._c.action_dist,
        init_std=self._c.action_init_std, act=act)
    model_modules = [self._encode, self._dynamics, self._decode, self._reward]
    if self._c.pcont:
      model_modules.append(self._pcont)
    Optimizer = functools.partial(
        tools.Adam, wd=self._c.weight_decay, clip=self._c.grad_clip,
        wdpattern=self._c.weight_decay_pattern)
    self._model_opt = Optimizer('model', model_modules, self._c.model_lr)
    self._value_opt = Optimizer('value', [self._value], self._c.value_lr)
    self._actor_opt = Optimizer('actor', [self._actor], self._c.actor_lr)
    # Do a train step to initialize all variables, including optimizer
    # statistics. Ideally, we would use batch size zero, but that doesn't work
    # in multi-GPU mode.
    self.train(next(self._dataset))

  def _exploration(self, action, training):
    if training:
      amount = self._c.expl_amount
      if self._c.expl_decay:
        amount *= 0.5 ** (tf.cast(self._step, tf.float32) / self._c.expl_decay)
      if self._c.expl_min:
        amount = tf.maximum(self._c.expl_min, amount)
      self._metrics['expl_amount'].update_state(amount)
    elif self._c.eval_noise:
      amount = self._c.eval_noise
    else:
      return action  (not training → no noise addition)
    if self._c.expl == 'additive_gaussian':
      return tf.clip_by_value(tfd.Normal(action, amount).sample(), -1, 1)
    if self._c.expl == 'completely_random':
      return tf.random.uniform(action.shape, -1, 1)
    if self._c.expl == 'epsilon_greedy':
      indices = tfd.Categorical(0 * action).sample()
      return tf.where(
```

```
          tf.random.uniform(action.shape[:1], 0, 1) < amount,
          tf.one_hot(indices, action.shape[-1], dtype=self._float),
          action)
    raise NotImplementedError(self._c.expl)

  def _imagine_ahead(self, post):
    if self._c.pcont:  # Last step could be terminal.
      post = {k: v[:, :-1] for k, v in post.items()}
    flatten = lambda x: tf.reshape(x, [-1] + list(x.shape[2:]))
    start = {k: flatten(v) for k, v in post.items()}
    policy = lambda state: self._actor(
        tf.stop_gradient(self._dynamics.get_feat(state))).sample()
    states = tools.static_scan(
        lambda prev, _: self._dynamics.img_step(prev, policy(prev)),
        tf.range(self._c.horizon), start)
    imag_feat = self._dynamics.get_feat(states)
    return imag_feat

  def _scalar_summaries(
      self, data, feat, prior_dist, post_dist, likes, div,
      model_loss, value_loss, actor_loss, model_norm, value_norm,
      actor_norm):
    self._metrics['model_grad_norm'].update_state(model_norm)
    self._metrics['value_grad_norm'].update_state(value_norm)
    self._metrics['actor_grad_norm'].update_state(actor_norm)
    self._metrics['prior_ent'].update_state(prior_dist.entropy())
    self._metrics['post_ent'].update_state(post_dist.entropy())
    for name, logprob in likes.items():
      self._metrics[name + '_loss'].update_state(-logprob)
    self._metrics['div'].update_state(div)
    self._metrics['model_loss'].update_state(model_loss)
    self._metrics['value_loss'].update_state(value_loss)
    self._metrics['actor_loss'].update_state(actor_loss)
    self._metrics['action_ent'].update_state(self._actor(feat).entropy())

  def _image_summaries(self, data, embed, image_pred):
    truth = data['image'][:6] + 0.5
    recon = image_pred.mode()[:6]
    init, _ = self._dynamics.observe(embed[:6, :5], data['action'][:6, :5])
    init = {k: v[:, -1] for k, v in init.items()}
    prior = self._dynamics.imagine(data['action'][:6, 5:], init)
    openl = self._decode(self._dynamics.get_feat(prior)).mode()
    model = tf.concat([recon[:, :5] + 0.5, openl + 0.5], 1)
    error = (model - truth + 1) / 2
    openl = tf.concat([truth, model, error], 2)
    tools.graph_summary(
        self._writer, tools.video_summary, 'agent/openl', openl)

  def _write_summaries(self):
    step = int(self._step.numpy())
    metrics = [(k, float(v.result())) for k, v in self._metrics.items()]
    if self._last_log is not None:
      duration = time.time() - self._last_time
      self._last_time += duration
      metrics.append(('fps', (step - self._last_log) / duration))
    self._last_log = step
    [m.reset_states() for m in self._metrics.values()]
    with (self._c.logdir / 'metrics.jsonl').open('a') as f:
      f.write(json.dumps({'step': step, **dict(metrics)}) + '\n')
    [tf.summary.scalar('agent/' + k, m) for k, m in metrics]
    print(f'[{step}]', ' / '.join(f'{k} {v:.1f}' for k, v in metrics))
    self._writer.flush()


def preprocess(obs, config):
  dtype = prec.global_policy().compute_dtype
```

predicted next feature in image form

```
  obs = obs.copy()
  with tf.device('cpu:0'):
    obs['image'] = tf.cast(obs['image'], dtype) / 255.0 - 0.5
    clip_rewards = dict(none=lambda x: x, tanh=tf.tanh)[config.clip_rewards]
    obs['reward'] = clip_rewards(obs['reward'])
  return obs


def count_steps(datadir, config):
  return tools.count_episodes(datadir)[1] * config.action_repeat


def load_dataset(directory, config):
  episode = next(tools.load_episodes(directory, 1))
  types = {k: v.dtype for k, v in episode.items()}
  shapes = {k: (None,) + v.shape[1:] for k, v in episode.items()}
  generator = lambda: tools.load_episodes(
      directory, config.train_steps, config.batch_length,
      config.dataset_balance)
  dataset = tf.data.Dataset.from_generator(generator, types, shapes)
  dataset = dataset.batch(config.batch_size, drop_remainder=True)
  dataset = dataset.map(functools.partial(preprocess, config=config))
  dataset = dataset.prefetch(10)
  return dataset


def summarize_episode(episode, config, datadir, writer, prefix):
  episodes, steps = tools.count_episodes(datadir)
  length = (len(episode['reward']) - 1) * config.action_repeat
  ret = episode['reward'].sum()
  print(f'{prefix.title()} episode of length {length} with return {ret:.1f}.')
  metrics = [
      (f'{prefix}/return', float(episode['reward'].sum())),
      (f'{prefix}/length', len(episode['reward']) - 1),
      (f'episodes', episodes)]
  step = count_steps(datadir, config)
  with (config.logdir / 'metrics.jsonl').open('a') as f:
    f.write(json.dumps(dict([('step', step)] + metrics)) + '\n')
  with writer.as_default():  # Env might run in a different thread.
    tf.summary.experimental.set_step(step)
    [tf.summary.scalar('sim/' + k, v) for k, v in metrics]
    if prefix == 'test':
      tools.video_summary(f'sim/{prefix}/video', episode['image'][None])


def make_env(config, writer, prefix, datadir, store):
  suite, task = config.task.split('_', 1)
  if suite == 'dmc':
    env = wrappers.DeepMindControl(task)
    env = wrappers.ActionRepeat(env, config.action_repeat)
    env = wrappers.NormalizeActions(env)
  elif suite == 'atari':
    env = wrappers.Atari(
        task, config.action_repeat, (64, 64), grayscale=False,
        life_done=True, sticky_actions=True)
    env = wrappers.OneHotAction(env)
  else:
    raise NotImplementedError(suite)
  env = wrappers.TimeLimit(env, config.time_limit / config.action_repeat)
  callbacks = []
  if store:
    callbacks.append(lambda ep: tools.save_episodes(datadir, [ep]))
  callbacks.append(
      lambda ep: summarize_episode(ep, config, datadir, writer, prefix))
  env = wrappers.Collect(env, callbacks, config.precision)
  env = wrappers.RewardObs(env)
```

```
  return env


def main(config):
  if config.gpu_growth:
    for gpu in tf.config.experimental.list_physical_devices('GPU'):
      tf.config.experimental.set_memory_growth(gpu, True)
  assert config.precision in (16, 32), config.precision
  if config.precision == 16:
    prec.set_policy(prec.Policy('mixed_float16'))
  config.steps = int(config.steps)
  config.logdir.mkdir(parents=True, exist_ok=True)
  print('Logdir', config.logdir)

  # Create environments.
  datadir = config.logdir / 'episodes'
  writer = tf.summary.create_file_writer(
      str(config.logdir), max_queue=1000, flush_millis=20000)
  writer.set_as_default()
  train_envs = [wrappers.Async(lambda: make_env(
      config, writer, 'train', datadir, store=True), config.parallel)
      for _ in range(config.envs)]
  test_envs = [wrappers.Async(lambda: make_env(
      config, writer, 'test', datadir, store=False), config.parallel)
      for _ in range(config.envs)]
  actspace = train_envs[0].action_space

  # Prefill dataset with random episodes.
  step = count_steps(datadir, config)
  prefill = max(0, config.prefill - step)
  print(f'Prefill dataset with {prefill} steps.')
  random_agent = lambda o, d, _: ([actspace.sample() for _ in d], None)
  tools.simulate(random_agent, train_envs, prefill / config.action_repeat)
  writer.flush()

  # Train and regularly evaluate the agent.
  step = count_steps(datadir, config)
  print(f'Simulating agent for {config.steps-step} steps.')
  agent = Dreamer(config, datadir, actspace, writer)
  if (config.logdir / 'variables.pkl').exists():
    print('Load checkpoint.')
    agent.load(config.logdir / 'variables.pkl')
  state = None
  while step < config.steps:
    print('Start evaluation.')
    tools.simulate(
        functools.partial(agent, training=False), test_envs, episodes=1)
    writer.flush()
    print('Start collection.')
    steps = config.eval_every // config.action_repeat
    state = tools.simulate(agent, train_envs, steps, state=state)
    step = count_steps(datadir, config)
    agent.save(config.logdir / 'variables.pkl')
  for env in train_envs + test_envs:
    env.close()


if __name__ == '__main__':
  try:
    import colored_traceback
    colored_traceback.add_hook()
  except ImportError:
    pass
  parser = argparse.ArgumentParser()
  for key, value in define_config().items():
    parser.add_argument(f'--{key}', type=tools.args_type(value), default=value)
```

Initial random exploration

run episodes using trained policy

saving new trajectories to update the model

```
main(parser.parse_args())
```