The _dynamics variable in main dreamer code is RSSM in this code


```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers as tfkl
from tensorflow_probability import distributions as tfd
from tensorflow.keras.mixed_precision import experimental as prec

import tools


class RSSM(tools.Module):

  def __init__(self, stoch=30, deter=200, hidden=200, act=tf.nn.elu):
    super().__init__()
    self._activation = act
    self._stoch_size = stoch
    self._deter_size = deter
    self._hidden_size = hidden
    self._cell = tfkl.GRUCell(self._deter_size)

  def initial(self, batch_size):
    dtype = prec.global_policy().compute_dtype
    return dict(
        mean=tf.zeros([batch_size, self._stoch_size], dtype),
        std=tf.zeros([batch_size, self._stoch_size], dtype),
        stoch=tf.zeros([batch_size, self._stoch_size], dtype),
        deter=self._cell.get_initial_state(None, batch_size, dtype))

  @tf.function
  def observe(self, embed, action, state=None):
    if state is None:
      state = self.initial(tf.shape(action)[0])
    embed = tf.transpose(embed, [1, 0, 2])
    action = tf.transpose(action, [1, 0, 2])
    post, prior = tools.static_scan(
        lambda prev, inputs: self.obs_step(prev[0], *inputs),
        (action, embed), (state, state))
    post = {k: tf.transpose(v, [1, 0, 2]) for k, v in post.items()}
    prior = {k: tf.transpose(v, [1, 0, 2]) for k, v in prior.items()}
    return post, prior

  @tf.function
  def imagine(self, action, state=None):
    if state is None:
      state = self.initial(tf.shape(action)[0])
    assert isinstance(state, dict), state
    action = tf.transpose(action, [1, 0, 2])
    prior = tools.static_scan(self.img_step, action, state)
    prior = {k: tf.transpose(v, [1, 0, 2]) for k, v in prior.items()}
    return prior

  def get_feat(self, state):
    return tf.concat([state['stoch'], state['deter']], -1)

  def get_dist(self, state):
    return tfd.MultivariateNormalDiag(state['mean'], state['std'])

  @tf.function
  def obs_step(self, prev_state, prev_action, embed):
    prior = self.img_step(prev_state, prev_action)
    x = tf.concat([prior['deter'], embed], -1)
    x = self.get('obs1', tfkl.Dense, self._hidden_size, self._activation)(x)
    x = self.get('obs2', tfkl.Dense, 2 * self._stoch_size, None)(x)
    mean, std = tf.split(x, 2, -1)
    std = tf.nn.softplus(std) + 0.1
    stoch = self.get_dist({'mean': mean, 'std': std}).sample()
    post = {'mean': mean, 'std': std, 'stoch': stoch, 'deter': prior['deter']}
```

```
    return post, prior

  @tf.function
  def img_step(self, prev_state, prev_action):
    x = tf.concat([prev_state['stoch'], prev_action], -1)
    x = self.get('img1', tfkl.Dense, self._hidden_size, self._activation)(x)
    x, deter = self._cell(x, [prev_state['deter']])
    deter = deter[0]  # Keras wraps the state in a list.
    x = self.get('img2', tfkl.Dense, self._hidden_size, self._activation)(x)
    x = self.get('img3', tfkl.Dense, 2 * self._stoch_size, None)(x)
    mean, std = tf.split(x, 2, -1)
    std = tf.nn.softplus(std) + 0.1
    stoch = self.get_dist({'mean': mean, 'std': std}).sample()
    prior = {'mean': mean, 'std': std, 'stoch': stoch, 'deter': deter}
    return prior


class ConvEncoder(tools.Module):

  def __init__(self, depth=32, act=tf.nn.relu):
    self._act = act
    self._depth = depth

  def __call__(self, obs):
    kwargs = dict(strides=2, activation=self._act)
    x = tf.reshape(obs['image'], (-1,) + tuple(obs['image'].shape[-3:]))
    x = self.get('h1', tfkl.Conv2D, 1 * self._depth, 4, **kwargs)(x)
    x = self.get('h2', tfkl.Conv2D, 2 * self._depth, 4, **kwargs)(x)
    x = self.get('h3', tfkl.Conv2D, 4 * self._depth, 4, **kwargs)(x)
    x = self.get('h4', tfkl.Conv2D, 8 * self._depth, 4, **kwargs)(x)
    shape = tf.concat([tf.shape(obs['image'])[:-3], [32 * self._depth]], 0)
    return tf.reshape(x, shape)


class ConvDecoder(tools.Module):

  def __init__(self, depth=32, act=tf.nn.relu, shape=(64, 64, 3)):
    self._act = act
    self._depth = depth
    self._shape = shape

  def __call__(self, features):
    kwargs = dict(strides=2, activation=self._act)
    x = self.get('h1', tfkl.Dense, 32 * self._depth, None)(features)
    x = tf.reshape(x, [-1, 1, 1, 32 * self._depth])
    x = self.get('h2', tfkl.Conv2DTranspose, 4 * self._depth, 5, **kwargs)(x)
    x = self.get('h3', tfkl.Conv2DTranspose, 2 * self._depth, 5, **kwargs)(x)
    x = self.get('h4', tfkl.Conv2DTranspose, 1 * self._depth, 6, **kwargs)(x)
    x = self.get('h5', tfkl.Conv2DTranspose, self._shape[-1], 6, strides=2)(x)
    mean = tf.reshape(x, tf.concat([tf.shape(features)[:-1], self._shape], 0))
    return tfd.Independent(tfd.Normal(mean, 1), len(self._shape))


class DenseDecoder(tools.Module):

  def __init__(self, shape, layers, units, dist='normal', act=tf.nn.elu):
    self._shape = shape
    self._layers = layers
    self._units = units
    self._dist = dist
    self._act = act

  def __call__(self, features):
    x = features
    for index in range(self._layers):
      x = self.get(f'h{index}', tfkl.Dense, self._units, self._act)(x)
```

```
    x = self.get(f'hout', tfkl.Dense, np.prod(self._shape))(x)
    x = tf.reshape(x, tf.concat([tf.shape(features)[:-1], self._shape], 0))
    if self._dist == 'normal':
      return tfd.Independent(tfd.Normal(x, 1), len(self._shape))
    if self._dist == 'binary':
      return tfd.Independent(tfd.Bernoulli(x), len(self._shape))
    raise NotImplementedError(self._dist)


class ActionDecoder(tools.Module):

  def __init__(
      self, size, layers, units, dist='tanh_normal', act=tf.nn.elu,
      min_std=1e-4, init_std=5, mean_scale=5):
    self._size = size
    self._layers = layers
    self._units = units
    self._dist = dist
    self._act = act
    self._min_std = min_std
    self._init_std = init_std
    self._mean_scale = mean_scale

  def __call__(self, features):
    raw_init_std = np.log(np.exp(self._init_std) - 1)
    x = features
    for index in range(self._layers):
      x = self.get(f'h{index}', tfkl.Dense, self._units, self._act)(x)
    if self._dist == 'tanh_normal':
      # https://www.desmos.com/calculator/rcmcf5jwe7
      x = self.get(f'hout', tfkl.Dense, 2 * self._size)(x)
      mean, std = tf.split(x, 2, -1)
      mean = self._mean_scale * tf.tanh(mean / self._mean_scale)
      std = tf.nn.softplus(std + raw_init_std) + self._min_std
      dist = tfd.Normal(mean, std)
      dist = tfd.TransformedDistribution(dist, tools.TanhBijector())
      dist = tfd.Independent(dist, 1)
      dist = tools.SampleDist(dist)
    elif self._dist == 'onehot':
      x = self.get(f'hout', tfkl.Dense, self._size)(x)
      dist = tools.OneHotDist(x)
    else:
      raise NotImplementedError(dist)
    return dist
```