Deep Deterministic Policy Gradients (DDPG)

You can use one of many different algorithms to design your agent, as long as it works with continuous state and action spaces. One popular choice is **Deep Deterministic Policy Gradients** or **DDPG**. It is actually an actor-critic method, but the key idea is that the underlying policy function used is deterministic in nature, with some noise added in externally to produce the desired stochasticity in actions taken.

Let's develop an implementation of the algorithm presented in the original paper:

> Lillicrap, Timothy P., et al., 2015. **Continuous Control with Deep Reinforcement Learning**. [pdf]

The two main components of the algorithm, the actor and critic networks can be implemented using most modern deep learning libraries, such as Keras or TensorFlow.

DDPG: Actor (Policy) Model

Here is a very simple actor model defined using Keras.

DDPG: Actor

```
class Actor:
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, action_low, action_high):
        """Initialize parameters and build model.

        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            action_low (array): Min value of each action dimension
            action_high (array): Max value of each action dimension
        """
        self.state_size = state_size
        self.action_size = action_size
        self.action_low = action_low
        self.action_high = action_high
        self.action_range = self.action_high - self.action_low

        # Initialize any other variables here

        self.build_model()

    def build_model(self):
        """Build an actor (policy) network that maps states -> actions."""
        # Define input layer (states)
        states = layers.Input(shape=(self.state_size,), name='states')

        # Add hidden layers
        net = layers.Dense(units=32, activation='relu')(states)
        net = layers.Dense(units=64, activation='relu')(net)
        net = layers.Dense(units=32, activation='relu')(net)

        # Try different layer sizes, activations, add batch normalization, re
gularizers, etc.

        # Add final output layer with sigmoid activation
        raw_actions = layers.Dense(units=self.action_size, activation='sigmoi
d',
            name='raw_actions')(net)

        # Scale [0, 1] output for each action dimension to proper range
        actions = layers.Lambda(lambda x: (x * self.action_range) + self.acti
on_low,
            name='actions')(raw_actions)

        # Create Keras model
        self.model = models.Model(inputs=states, outputs=actions)

        # Define loss function using action value (Q value) gradients
        action_gradients = layers.Input(shape=(self.action_size,))
        loss = K.mean(-action_gradients * actions)

        # Incorporate any additional losses here (e.g. from regularizers)
```

DDPG: Actor

```
        updates_op = optimizer.get_updates(params=self.model.trainable_weight
s, loss=loss)
        self.train_fn = K.function(
            inputs=[self.model.input, action_gradients, K.learning_phase()],
            outputs=[],
            updates=updates_op)
```

Note that the raw actions produced by the output layer are in a `[0.0, 1.0]` range (using a sigmoid activation function). So, we add another layer that scales each output to the desired range for each action dimension. This produces a deterministic action for any given state vector. A noise will be added later to this action to produce some exploratory behavior.

Another thing to note is how the loss function is defined using action value (Q value) gradients:

```
# Define loss function using action value (Q value) gradients
action_gradients = layers.Input(shape=(self.action_size,))
loss = K.mean(-action_gradients * actions)
```

These gradients will need to be computed using the critic model, and fed in while training. Hence it is specified as part of the "inputs" used in the training function:

```
self.train_fn = K.function(
   inputs=[self.model.input, action_gradients, K.learning_phase()],
    outputs=[]
```

NEXT