DDPG: Critic (Value) Model

Here's what a corresponding critic model may look like:

DDPG: Critic

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

        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
        """
        self.state_size = state_size
        self.action_size = action_size

        # Initialize any other variables here

        self.build_model()

    def build_model(self):
        """Build a critic (value) network that maps (state, action) pairs ->
 Q-values."""
        # Define input layers
        states = layers.Input(shape=(self.state_size,), name='states')
        actions = layers.Input(shape=(self.action_size,), name='actions')

        # Add hidden layer(s) for state pathway
        net_states = layers.Dense(units=32, activation='relu')(states)
        net_states = layers.Dense(units=64, activation='relu')(net_states)

        # Add hidden layer(s) for action pathway
        net_actions = layers.Dense(units=32, activation='relu')(actions)
        net_actions = layers.Dense(units=64, activation='relu')(net_actions)

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

        # Combine state and action pathways
        net = layers.Add()([net_states, net_actions])
        net = layers.Activation('relu')(net)

        # Add more layers to the combined network if needed

        # Add final output layer to prduce action values (Q values)
        Q_values = layers.Dense(units=1, name='q_values')(net)

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

        # Define optimizer and compile model for training with built-in loss
 function
        optimizer = optimizers.Adam()
        self.model.compile(optimizer=optimizer, loss='mse')

        # Compute action gradients (derivative of Q values w.r.t. to actions)
        action_gradients = K.gradients(Q_values, actions)

        # Define an additional function to fetch action gradients (to be used
```

```
                outputs=action_gradients)
```

It is simpler than the actor model in some ways, but there some things worth noting. Firstly, while the actor model is meant to map states to actions, the critic model needs to map (state, action) pairs to their Q-values. This is reflected in the input layers.

```
# Define input layers
states = layers.Input(shape=(self.state_size,), name='states')
actions = layers.Input(shape=(self.action_size,), name='actions')
```

These two layers can first be processed via separate "pathways" (mini sub-networks), but eventually need to be combined. This can be achieved, for instance, using the `Add` layer type in Keras (see **Merge Layers**):

```
# Combine state and action pathways
net = layers.Add()([net_states, net_actions])
```

The final output of this model is the Q-value for any given (state, action) pair. However, we also need to compute the gradient of this Q-value with respect to the corresponding action vector, needed for training the actor model. This step needs to be performed explicitly, and a separate function needs to be defined to provide access to these gradients:

```
# Compute action gradients (derivative of Q values w.r.t. to actions)
action_gradients = K.gradients(Q_values, actions)

# Define an additional function to fetch action gradients (to be used by acto
r model)
self.get_action_gradients = K.function(
    inputs=[*self.model.input, K.learning_phase()],
```

NEXT