```python
import torch as torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F

def fanin_init(size, fanin=None):
    '''a helper function to initialize the weights of the model'''
    fanin = fanin or size[0]
    v = 1. / np.sqrt(fanin)
    return torch.Tensor(size).uniform_(-v, v)


class Actor(nn.Module):
    """Actor model for the DDPG algorithm.

    Layer 1: 400 units, ReLU activation, Fan-in weight initialization, ie each weight is
initialized with a uniform distribution in the range of -1/sqrt(fan_in) to 1/sqrt(fan_in)
    Layer 2: 300 units, ReLU activation, Fan-in weight initialization, ie each weight is
initialized with a uniform distribution in the range of -1/sqrt(fan_in) to 1/sqrt(fan_in)
    Layer 3: 1 unit, tanh activation, intialized with uniform weights in the range of -0.003 to
0.003

    """
    def __init__(self, input_size:tuple[int], action_size:int,CNN = None):
        """
        input: tuple[int]
            The input size, as a tuple of dimensions, for the DoubleInvertedPendulum
environment, of shape (11,)
        action_size: int
            The number of actions
        """
        super(Actor, self).__init__()
        # ========== YOUR CODE HERE ==========
        # TODO:
        # define the fully connected layers for the actor
        # ====================================
        self.CNN = CNN  # Currently unused

        in_dim = input_size[0]  # e.g., 11 for (11,)
        self.fc1 = nn.Linear(in_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, action_size)

        self.init_weights()  # Initialize all layers

        # ========== YOUR CODE ENDS ==========

    def init_weights(self,init_w=3e-3):
        """
        Args:
            init_w (float, optional): the onesided range of the uniform distribution for the
final layer. Defaults to 3e-3.
        """
        # ========== YOUR CODE HERE ==========
        # TODO:
        # initialize the weights of the model
        # ====================================
        # Fan-in initialization for fc1
        fan_in1 = self.fc1.weight.data.size()[1]  # input dim
        lim1 = 1. / np.sqrt(fan_in1)
        nn.init.uniform_(self.fc1.weight, -lim1, lim1)
        nn.init.uniform_(self.fc1.bias, -lim1, lim1)

        # Fan-in initialization for fc2
        fan_in2 = self.fc2.weight.data.size()[1]
        lim2 = 1. / np.sqrt(fan_in2)
        nn.init.uniform_(self.fc2.weight, -lim2, lim2)
```

```python
        nn.init.uniform_(self.fc2.bias, -lim2, lim2)

        # Small uniform initialization for output layer
        nn.init.uniform_(self.fc3.weight, -init_w, init_w)
        nn.init.uniform_(self.fc3.bias, -init_w, init_w)

        # ========== YOUR CODE ENDS ==========

    def forward(self, x:torch.Tensor)->torch.Tensor:
        # ========== YOUR CODE HERE ==========
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.tanh(self.fc3(x))  # Ensures output in [-1, 1]
        return x

        # ========== YOUR CODE ENDS ==========



class Critic(nn.Module):
    """Critic model for the DDPG algorithm.
    Layer 1: 400 units, ReLU activation, Fan-in weight initialization, ie each weight is
initialized with a uniform distribution in the range of -1/sqrt(fan_in) to 1/sqrt(fan_in)
    Layer 2: 300 units, ReLU activation, Fan-in weight initialization, ie each weight is
initialized with a uniform distribution in the range of -1/sqrt(fan_in) to 1/sqrt(fan_in).
Input is the concatenation of the 400 dimension embedding from the state, and the action taken.
    Layer 3: 1 unit, intialized with uniform weights in the range of -0.003 to 0.003
    """
    def __init__(self,input_size:tuple[int],action_size:int):
        """
        input: tuple[int]
            The input size, as a tuple of dimensions, for the DoubleInvertedPendulum
environment, of shape (11,)
        action_size: int
            The number of actions
        """
        super(Critic, self).__init__()
        # ========== YOUR CODE HERE ==========
        # TODO:
        # define the fully connected layers for the critic and initialize the weights
        # ====================================
        state_dim = input_size[0]

        self.fc1 = nn.Linear(state_dim, 400)
        self.fc2 = nn.Linear(400 + action_size, 300)
        self.fc3 = nn.Linear(300, 1)

        self.init_weights()

        # ========== YOUR CODE ENDS ==========

    def init_weights(self,init_w=3e-3):
        # ========== YOUR CODE HERE ==========
        # TODO:
        # initialize the weights of the model
        # ====================================
        fan_in1 = self.fc1.weight.data.size()[1]
        lim1 = 1. / np.sqrt(fan_in1)
        nn.init.uniform_(self.fc1.weight, -lim1, lim1)
        nn.init.uniform_(self.fc1.bias, -lim1, lim1)

        fan_in2 = self.fc2.weight.data.size()[1]
        lim2 = 1. / np.sqrt(fan_in2)
        nn.init.uniform_(self.fc2.weight, -lim2, lim2)
        nn.init.uniform_(self.fc2.bias, -lim2, lim2)

        nn.init.uniform_(self.fc3.weight, -init_w, init_w)
```

```python
        nn.init.uniform_(self.fc3.bias, -init_w, init_w)

        # ========== YOUR CODE ENDS ==========

    def forward(self, x:torch.Tensor, a:torch.Tensor)->torch.Tensor:
        # ========== YOUR CODE HERE ==========
        xs = F.relu(self.fc1(x))             # process state
        xsa = torch.cat([xs, a], dim=1)      # concatenate state embedding with action
        x = F.relu(self.fc2(xsa))
        q_value = self.fc3(x)                # no activation here; scalar output
        return q_value

        # ========== YOUR CODE ENDS ==========
```