E90 Project Proposal:
Autonomously Learning Walking Policy Using Deep Neural Networks

Christian Vik

Advisor:
Professor Matt Zucker
Table of Contents:

Abstract: p.2

Introduction: p.3

Technical Discussion p.4-8

Results: p.9-15

Discussion: p.16-20

References: p.

Appendix: p.
Abstract:

Training agents to learn policies on complex problems has been computationally too expensive for a long time. Even after the advent of the neural network, the computational requirements exceeded the abilities of our current hardware. Now that our hardware is able to handle the larger computational loads of deep neural networks, we are able to leverage their power against harder, more complex problems.

The problem of autonomous walking is a well-defined and explore one in computer science and engineering. The famous DARPA grand challenge charged teams to build robots to accomplish a series of complex tasks aimed to push the boundaries of what has been previously accomplished in the field. The Google DeepMind team famously taught an AlphaGo (and later AlphaGo Zero) to beat the best human player in the world at the one of most complex games we have. Not long afterwards Google DeepMind showed off learned policies for a number of agents with different bodies to finish complex obstacle ridden courses by attacking the problems using Deep Reinforcement Learning. The power of the deep neural network is shown in the variety and complexity of tasks it has accomplished, consistently outperforming human adversaries in many fields.

In this project we will attempt to similarly learn a use deep reinforcement learning to allow a quadruped agent to walk. By applying different deep learning techniques and varying network architecture we hope to solve the autonomous walking problem for our agent.
Introduction:

The primary goal of this project is to have a quadruped agent autonomously learn a walking policy by using Deep Q-Learning. The secondary goal of this project is to find the methods that lead to learning the best walking policy through exploring different training methods and using different network architectures, as well as tuning the parameters and hyperparameters of these networks.

This report is organized into technical discussion, results, and conclusion. The technical discussion will cover background information on traditional Reinforcement Learning, specifically Q-learning, and the motivation and high level information for the Actor-Critic method which we used for this project. We will discuss the theory behind it and its limitations when faced with problems like the one which we are considering for this project. We will discuss deep Q-learning and how it could overcome the limitations of traditional learning. We will also discuss the reasoning behind using the actor-critic method to solve this problem.

The results section will include both quantitative and qualitative assessments of the performance of agents on the OpenAI Roboschool Ant problem as well as two toy problems which were used to test our agent before deployment on harder problems: OpenAI Mountain Car Continuous and OpenAI Pendulum.

The conclusion section will include discussion over which methods worked or didn’t work and why, as well as possible changes and future work. We will also discuss potential limitations to the approaches we used as well as the OpenAI Roboschool Ant problem in general including its multiple iterations and different implementations within OpenAI.
Technical Discussion:

In artificial intelligence, learning is the ability of an agent to develop a policy or plan of action to solve a given problem. Specifically in this project we will talking about a subset of learning called Reinforcement Learning. Reinforcement Learning involves an agent learning through experience. When exploring an environment, agents will encounter positive and negative rewards corresponding to actions from certain states. The policy of the agent, or the way in which the agent will select actions given states, will change according to its past experience in order to maximize reward.

There are generally two types of Reinforcement Learning: Value Learning and Policy Learning. In Value Learning the aim is to learn some function that gives values to possible actions. Then by inspecting the values we have mapped to actions we can decide which action to take. The decision of which action to take is called the Policy and is generally denoted by $\pi$. Policy Learning, uses a level of abstraction to remove the value of actions from consideration and directly learns a policy. This policy we learn is a direct mapping from states to actions. Both methods of learning accomplish the same goal in the end, allowing the agent to select actions, but the way in which our policy formed differs.

Q-Learning is an example of Value Learning and will allow us to describe some fundamental ideas in Value Learning. In Q-Learning, the agent will learn some mapping that gives a quality value (Q) to an action (a) taken from a state (s). The agent will record the rewards obtained at each time an action is taken from a given state (hereafter referred to as a state-action pair). The rewards from these state-action pairs can be stored in a table and updated as the agent explores the state space. As the agent explores the state space and accumulates rewards, the rewards will be propagated throughout the table.
Rewards are updated in the table using value iteration, weighing new and old experiences to come up with a current expected value for each state-action pair. Value iteration allows the rewards from actions taken in the future to affect expected reward for the current action. The SARSA (state-action-reward-state-action) equation governs this update rule for the table and is expressed as:

$$Q^{\text{new}}(s_t, a_t) \leftarrow (1-\alpha)Q^{\text{old}}(s_t, a_t) + \alpha (r_t + \gamma \max Q(s_{t+1}, a))$$

Where the new value is a weighted sum of the current table value times one minus alpha (our learning rate), plus alpha times the the quantity of the current reward plus the product of gamma (the discount factor) and the maximum possible value to achieve from the next state at given and possible action we make. Alpha is a constant called the learning rate which determines how much new information will be considered compared to old information, and is typically less than 1. For large alpha (0.5<\alpha), the agent considers the new information more than old information. Changing alpha allows the agent to decide how “willing” the agent is to change its policy based on new information. Gamma is a constant called the discount factor which determines how much the future experience will affect the current quality. In this way, expected future rewards can be discounted and tell the agent that this action will lead to more reward in the future if not now.

We must also consider our policy during training (i.e. before we have reliable values in the Q-table) as well. We want to find the best policy possible, which requires trying many options, while also making sure rewards from good decisions are propagated backwards in our Q-table. This is known as the explore-exploit tradeoff- do we choose to explore as much of the table as possible? or do we pursue paths that have proven to give higher rewards? This problem is generally addressed by incorporating a stochastic method into our policy (e.g.
\( \epsilon \)-greedy exploration where the agent takes a random action with probability \( 1-\epsilon \) and otherwise takes the action with highest quality (Q)).

With smaller state spaces it is feasible to explore the entire state space to ensure optimality, however as the state space increases with the complexity with the problem, this is no longer feasible. In the problem we are considering- learning a motion policy for a quadruped robot- the state space is much too large to feasibly explore entirely. The size of a state space is defined by \( N^d \) where \( N \) is the number of states per dimension, and \( d \) is the number of dimensions. In the problem we are considering, the state is defined by 28 continuous variables in the range \([-\text{inf}, \text{inf}]\). Now for the sake of theory if we were to confine these variables for the sake of argument to the integers between \([-10, 10]\), we would have \( 20^{28} \approx 2.7 \times 10^{36} \) possible states. This state space is far too large to be able to explore to any reasonable “fullness” to guarantee finding a policy that solves the problem. In order to tackle the problem of the intractable state space, we can instead use a neural network to learn our value function rather than using a table to hold each value individually.

Deep neural networks are being used more and more frequently to tackle complex computing problems due to their versatility. Deep neural networks are incredibly powerful tools against extraordinarily complex problems able to learn arbitrarily complex functions. A Q-network will learn the function that given a state, will output a Q value for each of the possible discrete actions. With a Q-network, we still need to use the returned Q-values to select actions ourselves from each state. We will then similarly update the values in our network based on the expected future reward of the next state. The value used will still be the maximum Q-value over all actions in the next state, but will also be supplied by our network of our network taking the next state as input.
When there exists a set of discrete actions to take Value Learning can assign a value to each possible action even if there are many states trading the Q-table for a Q-network. What if rather than a set of discrete actions, we had a continuous range of actions from which to choose? We now have an infinite number of actions and Value Learning methods become intractable. There are too many possible actions to calculate expected rewards for each of them and then choose the maximum.

Policy Learning can be leveraged to tackle continuous action spaces. Rather than learning a value for each action we can take from a state, Policy Learning will learn a mapping directly from state to action regardless of how many possible actions we have. By using a neural network to learn the policy function, it is possible to use policy learning to learn a mapping from a continuous set of states to a continuous set of actions. The agent still learns by trial and error, but the way in which the policy is updated is less intuitive than in the case of simple Q-Learning.

Policy Gradient Search is a method of Policy Learning that uses a neural network to learn the policy function. In Policy Gradient search we first model the actions taken at any given state by a probability distribution (either discrete or continuous). We update the weights of the neural network after taking action $a$ from state $s$ and receiving a discounted reward $R(\tau)$ according to the following rule:

$$J(\theta) = \alpha \cdot \left( \nabla_{\theta} \log(\pi(s,a,\theta)) \right) \cdot R(\tau)$$

We take the gradient of the log probability of taking action $a$ from state $s$ according to our policy $\pi$ times the discounted reward received. We discount the reward to take into account the future rewards the agent received using this same policy over a full episode. This quantity is multiplied by the learning rate alpha to control how much we change the weights according to the new information.
Value and Policy Learning can be combined to complement each other and create an Actor-Critic Agent. The agent will learn a policy function (actor) which supplies actions to be tried in the environment. The agent will also learn a modified value function (critic) which evaluates the actor’s policy compared to the expected reward from a given state. The difference between the discounted reward received from trying action a from state s as supplied by the actor and the value supplied by the critic is our error which we will train both networks to minimize. The idea is that the critic’s value function will output the best expected value of a state based on the rewards of taking many different actions. It is important to note that the critic only takes as input a state, so it is action agnostic. The actor’s policy function should yield actions that give high rewards. If the action supplied by the actor (plus the discounted value of the next state) resulted in less reward than the critic expected, then the probability we take this action should be decreased and the value function should take into account this reward when evaluating this state. If the action supplied by the actor resulted in more reward than the critic expected, then the probability we take this action should be increased and the critic should update the expected value of this state to take this action into account.
Results:

Note: videos of the Actor-Critic Agent solving Mountain Car and Pendulum are available on the Github repository referenced in Appendix 1, the full link is available below:

https://github.com/cvik1/swat_e90/tree/master/videos

Figure 1: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.001 and Critic learning rate=.01
Figure 2: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.0001 and Critic learning rate=.01

Figure 3: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.00001 and Critic learning rate=.01
Figure 4: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.000001 and Critic learning rate=.01

![Actor-Critic with Learning Rates .000001, .01](image)

Figure 5: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.001 and Critic learning rate=.001

![Actor-Critic with Learning Rates .001, .001](image)
Figure 6: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.001 and Critic learning rate=.0001

Figure 7: Actor-Critic Agent Rewards over 1000 Episodes with Actor learning rate=.001 and Critic learning rate=.00001
**Figure 9:** Actor-Critic Agent Rewards over 7000 Episodes with Actor learning rate=.001 and Critic learning rate=.00001

![Actor-Critic with Learning Rates .001, .00001](image)

**Figure 10:** Actor-Critic Agent Rewards over 6000 Episodes with Actor learning rate=.001 and Critic learning rate=.001

![Actor-Critic with Learning Rates .001, .001](image)
Figure 11: Actor-Critic Agent Rewards over 6000 Episodes with Actor learning rate=.0001 and Critic learning rate=.01

Figure 12: The network architecture of the actor within the Actor-Critic Agent
**Figure 13:** The network architecture of the critic within the Actor-Critic Agent

**Figure 14:** This figure shows the body orientation of Ant
Conclusion:

The Actor-Critic agent easily and quickly solved the two proposed toy problems: Mountain Car Continuous and Pendulum. As shown in the videos referenced in the results section, both problems were solved within 1000 episodes. The solutions to those problems served as a proof of concept for our implementation of the Actor-Critic. By solving smaller, easier problems, we can say with certainty that our agent can solve a problem in the first place and prove the correctness of our implementation. By solving problems of different scales, we hoped to say that our approach was scalable to problems of different difficulties and larger state spaces.

To solve the toy problems we used an actor learning rate of .00001 and a critic learning rate of .01. As shown in Figure 11, when the our Actor-Critic agent was run on Ant with these hyperparameters and the network sizes shown in Figures 12 and 13, the agent did not perform well at all. This was not necessarily surprising as Ant is a vastly different problem than both Mountain Car and Pendulum. The action and state spaces dwarf those of the toy problems. Although we cannot know, the policy function to solve Ant is likely much more complicated than that of either toy problem.

We refined our approach and tested different combinations of learning rates over a 1000 episode period to see what yielded good results on Ant. We tested 7 different combinations, first changing the actor learning rate and keeping the critic learning rate at .01 as shown in Figures 1 through 4. As shown in Figure 2, the hyperparameters we first attempted to solve Ant with did not perform very well at all even over the 1000 episode time span. The best results showing increases in rewards, on average, over time were found using an actor learning rate of .001 as shown in Figure 1.
We then used the actor learning rate of .001 as the control while we tested different critic learning rates over the 1000 episode time span. As shown in Figures 5, 6, and 7, we tested three new learning rates for the critic, not having to test the previously tested learning rate of .01 with the actor learning rate of .001. From these results we saw the best plotted rewards over time from the combination of actor and critic learning rates to be .001 for both the actor and the critic, and .001 for the actor and .00001 for the critic.

Taking our most promising results from the 1000 episode time horizon, we ran the agent with each set of hyperparameters with a time horizon of 100,000 episodes. While the agents were training, we monitored their progress to be able to stop them before the 100,000 episodes in case they were not performing well. These results are shown in Figures 9 and 10.

In Figure 9, the agent does not seem to be learning a policy at all. The constant rewards with some noise around approximately 250 tell us that the agents are not being updated to reflect the current information as much as they should be. This leads to an agent that stagnates and does not improve over time as it should. Although the results were promising over the smaller time horizon, when looking at the larger learning trends over time we can see that the agent is not learning as it should be.

In Figure 10, the agent does steadily better up until approximately episode 1000. At episode 1000 the agent’s reward falls to near 0 and stays there for the remaining 5000 episodes. These results are very similar to the results when using the same hyperparameters as used on the toy problems (actor learning rate = .0001m critic learning rate = .01). The reproducibility of these results points to a potentially larger problem our methods than just the hyperparameters. Two different sets of learning rates producing very similar failures must mean that there is a larger problem with either the implementation or the Actor-Critic method itself.
The actor critic method is unique in that it learns both a policy and a value function, and additionally trains both networks on the error of the value function. Training both networks on the error of a single network can be problematic if the network whose error we are using makes a mistake. The Actor-Critic method is unstable and does not always converge to a maximum.

Consider the following example:

Given some state $s_i$, we ask our actor for an action $a_i$ according to our current policy. We also ask our critic to give a value $v_s$ for the state $s_i$ according to the current function. If for whatever reason, either the action or the value are not well calibrated to the values each network expects of the other, then our error will be very high. Even if we do much better than expected it can pose problems. If our error is the difference between the reward we recieved and the value we expected, then in either case we will have very large error. The larger the magnitude of the error we are using, the more significant the changes to the weights will be in order to attempt to learn from this experience. If large changes are made in either direction then we will destabilize our networks by drastically changing the weights they have learned from other experiences. Thus effectively unlearning our policy and inducing bad performance.

The above example is what we suspect happened in Figure 10 and 11, but a similar example can explain what is happening in Figure 9 as well:

Given some state $s_i$, we ask our actor for an action $a_i$ according to our current policy. We also ask our critic to give a value $v_s$ for the state $s_i$ according to the current function. The actor is trained to give actions whose rewards match the value $v_s$. If the action matches $v_s$ closely enough, then there is little to no error to use for training. When this is the case the agent will continue to produce these results and no appreciable learning will occur.
We can explain all three failures of our model with the natural instability of the Actor-Critic method.

While this instability is inherent in the Actor-Critic method, that does not mean that is cannot be overcome. While we did not have the opportunity to test more combinations of learning rates our agent, some combination of learning rates should be able to solve Ant using our agent. We suspect that due to the very complex nature of the value function and the subtle differences between similar actions, these values will be hard to find as it is a very difficult problem to solve.

Since we could not solve the problem ourselves, we then looked to the solutions of those who had solved a similar problem: OpenAI Mujoco Ant. The difference between the Mujoco Ant and the Roboschool Ant is namely that the central body on the Roboschool Ant is heavier (see Figure 14) is made heavier. Other differences are in the backend supporting Ant. Mujoco Ant uses the Mujoco physics engine which requires an expensive license whereas Mujoco Ant uses the free Bullet physics engine. In order to test whether or not the new implementation of Ant was significantly harder, we ran the code of current leader on OpenAI’s website for Mujoco Ant on Roboschool Ant. The code for this implementation can be found at https://github.com/pat-coady/trpo. This implementation is one of two implementations that have solved Mujoco Ant and submitted their results the OpenAI leaderboard and solved the problem (averaged over 6000 reward over 100 episodes) in just under 70,000 episodes.

We ran the same code on the Roboscool Ant on 100,000 episodes which took just over two days to complete. Unfortunately the method he used to graph his results was not compatible with our AWS server we were using to train the agent and we were unable to produce graphs. The output of the program was saved to a file “pat-coady.out” on the github repository for this project (linked in Appendix 1). Through inspection of this file we can see that
the agent first seemed to be learning well, increasing as a steady rate as time went on. Over the full 100,000 episodes however, it was barely able to break 3000 reward. While this is significantly better than we were able to do with any of our methods, it is still only 50% of the goal to solve the problem.

From the results of this test, we conclude that the Roboschool Ant is a harder problem to solve than the Mujoco Ant and the heavier body leads to an appreciable increase in the difficulty of solving the problem. While this problem is inherently more difficult than the original Mujoco Ant, our approach still should have yielded better results. In order to solve this problem, we must explore the possible values of all hyperparameters more fully, not just the learning rates. It is also possible that our network architecture is ill-suited to the Ant problem. While the networks worked for the smaller problems, it is possible that networks need to be larger, or have different overall shapes in order to better map our 28x1 state input to 8x1 actions or 1x1 values.
References:


Appendix:

Appendix 1: Github repository where the code is contained
https://github.com/cvik1/swat_e90

Appendix 2: Select code from the Actor-Critic class
class A2CAgent_v2_tf(MLAgent):
    
    our second attempt at writing a actor critic agent to solve
    continuous action space problems
    This implementation is written in raw tensorflow
    
    def __init__(self, alpha, gamma, epsilon, env):
        super().__init__(alpha[0], gamma, epsilon, env)
        self.sess = tf.Session()  # session to use for training
        backend.set_session(self.sess)
        self.memory = deque(maxlen=10000)  # array to hold experiences for training

        # define required placeholders
        self.state_placeholder = tf.placeholder(tf.float32, [None,
                                                   self.env.observation_space.shape[0]])
        self.action_placeholder = tf.placeholder(tf.float32)
        self.delta_placeholder = tf.placeholder(tf.float32)
        self.target_placeholder = tf.placeholder(tf.float32)

        self.action_tf_var, self.norm_dist = self.buildActor()
        self.V = self.buildCritic()

        # define actor (policy) loss function
        lr_actor = alpha[0]
        self.loss_actor = -tf.log(self.norm_dist.prob(self.action_placeholder) + 1e-5) * self.delta_placeholder
        self.training_op_actor = tf.train.AdamOptimizer(lr_actor, name='actor_optimizer').minimize(self.loss_actor)

        # define critic (state-value) loss function
        lr_critic = alpha[1]
        self.loss_critic = tf.reduce_mean(tf.squared_difference(tf.squeeze(self.V), self.target_placeholder))
        self.training_op_critic = tf.train.AdamOptimizer(lr_critic, name='critic_optimizer').minimize(self.loss_critic)
# create our scaler for normalizing the inputs
# state_space_samples = np.array([self.env.observation_space.sample() for x in
    # range(10000)])
# state_space_samples = np.array([np.random.uniform(low = -500.0, high = 500.0,
    # size=28) for x in range(10000)])

self.saver = tf.train.Saver()

self.sess.run(tf.global_variables_initializer())

def getAction(self, state):
    ""
    greedily returns an action from the actor model given the state
    ""
    action = self.sess.run(self.action_tf_var, feed_dict={
        self.state_placeholder: self.scaleState(state)})
    return action.reshape(-1,1)

def buildActor(self):
    ""
    builds the actor model
    ""
    n_h1 = 40
    n_h2 = 40
    n_out = self.env.action_space.shape[0]
    with tf.variable_scope("policy_network"):  
        init_xavier = tf.contrib.layers.xavier_initializer()
        h1 = tf.layers.dense(self.state_placeholder, n_h1, tf.nn.elu, init_xavier)
        h2 = tf.layers.dense(h1, n_h2, tf.nn.elu, init_xavier)
        mu = tf.layers.dense(h2, n_out, None, init_xavier)
        sigma = tf.layers.dense(h2, n_out, None, init_xavier)
        sigma = tf.nn.softplus(sigma) + 1e-5
        norm_dist = tf.contrib.distributions.Normal(mu, sigma)
        action_tf_var = tf.squeeze(norm_dist.sample(1), axis=0)
        action_tf_var = tf.clip_by_value(action_tf_var, -1,1)
        return action_tf_var, norm_dist

def buildCritic(self):
    ""
    builds the critic model
n_h1 = 400
n_h2 = 400
n_out = 1

with tf.variable_scope("value_network"):
    init_xavier = tf.contrib.layers.xavier_initializer()

    h1 = tf.layers.dense(self.state_placeholder, n_h1, tf.nn.elu, init_xavier)
    h2 = tf.layers.dense(h1, n_h2, tf.nn.elu, init_xavier)
    V = tf.layers.dense(h2, n_out, None, init_xavier)
    return V

def trainModel(self):
    # trains the model using batches from past experiences

    # decay epsilon
    self.epsilon *= .9995

    batch_size = 32
    if len(self.memory) < batch_size:
        # if we dont have enough experiences to train
        return

        # sample a batch of experiences randomly from memory
    samples = random.sample(self.memory, batch_size)

    for sample in samples:
        state, action, reward, next_state, done = sample

        if not done:
            #V_of_next_state.shape=(1,1)
            target_reward = self.sess.run(self.V, feed_dict =
                {self.state_placeholder: next_state})
            #Set TD Target
            #target = r + gamma * V(next_state)
            target = reward + self.gamma * np.squeeze(target_reward)

            # td_error = target - V(s)
            # needed to feed delta_placeholder in actor training
            td_error = target - np.squeeze(self.sess.run(self.V, feed_dict =
                {self.state_placeholder: state}))

    return td_error
#Update actor by minimizing loss (Actor training)
_, self.loss_actor_val = self.sess.run(
    [self.training_op_actor, self.loss_actor],
    feed_dict={self.action_placeholder: np.squeeze(action),
               self.state_placeholder: state,
               self.delta_placeholder: td_error})

#Update critic by minimizing loss (Critic training)
_, self.loss_critic_val = self.sess.run(
    [self.training_op_critic, self.loss_critic],
    feed_dict={self.state_placeholder: state,
               self.target_placeholder: target})

def online(self, state, action, reward, next_state, done):
    
    implements online learning
    
    # scale the states
    state = self.scaleState(state)
    next_state = self.scaleState(next_state)

    if not done:
        #V_of_next_state.shape=(1,1)
        target_reward = self.sess.run(self.V, feed_dict =
                                      {self.state_placeholder: next_state})
        #Set TD Target
        #target = r + gamma * V(next_state)
        target = reward + self.gamma * np.squeeze(target_reward)

        # td_error = target - V(s)
        #needed to feed delta_placeholder in actor training
        td_error = target - np.squeeze(self.sess.run(self.V, feed_dict =
                                                {self.state_placeholder: state}))

        #Update actor by minimizing loss (Actor training)
        _, self.loss_actor_val = self.sess.run(
            [self.training_op_actor, self.loss_actor],
            feed_dict={self.action_placeholder: np.squeeze(action),
                       self.state_placeholder: state,
                       self.delta_placeholder: td_error})

        #Update critic by minimizing loss (Critic training)
        _, self.loss_critic_val = self.sess.run(
            [self.training_op_critic, self.loss_critic],
            feed_dict={self.state_placeholder: state,
Appendix 3: The training loop used to train our agent (agent) on the environment (env)

while training:

    # train the agent
    for episode in range(training_episodes):
        # print("Episode: {}".format(episode))
        # initialize environment variables
        state = env.reset()
        # reshape the state array
        state = state.reshape(1,-1)
        sum_reward = 0
        steps = 0
        done = False
        action = None
        # to record the episode
        test_record = np.empty((1000,36))
        # while not done:
        while not done:
            # get an action from the exploration policy
            action = agent.explore(state)

            # apply the action to the env
            # we must reshape the action before stepping for compatability with
            # rendering/recording the video
            next_state, reward, done, info = env.step(action.reshape(1,-1)[0])

            # reshape the action for input use in training
            action = action.reshape(1, -1)
            # reshape the next_state array
            next_state = next_state.reshape(1,-1)
            # place the action in the array
            # test_record[steps,:8] = action.reshape(1,-1)[0].flatten()
            # # place the state in the array
            # test_record[steps,8:] = state.flatten()
            # add this experience to memory for training use later
            agent.remember(state, action, reward, next_state, done)
            # update the current state
            state = next_state
            # update the steps and sum reward for bookkeeping purposes
            steps +=1
sum_reward += reward

# online training
agent.online(state, action, reward, next_state, done)

# train the model after every iteration
# agent.trainModel()