CMPUT 607-W17 Robot Module 4 - Actor Critic

David Quail

Abstract:

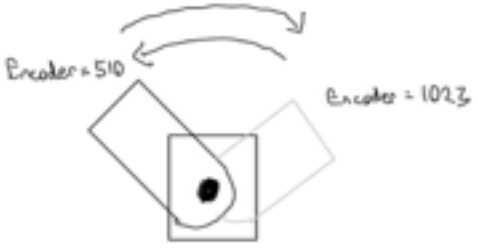
Thus far, we have explored reinforcement learning systems that make predictions (GVF's) about the future using a stream of sensorimotor readings. We have not however, used this stream of data to control the actions our robot was taking, in an attempt to maximize some sort of reward signal. In this paper, we look to have our robot attempt to maximize some reward, by changing the robots policy directly. This policy parameterization is very different than other reinforcement learning algorithms which consult an action value function in order to "pick" optimal actions. We look at two different ways to parameterize the policy, in both cases, in an effort to maximize some reward.

1. Using discrete actions. In our case, the robot actuator can choose to move "left" or right." And
2. Using continuous actions. In our case, the robot actuator can choose actions which correlate to a desired encoder value.

Experiment 1: Discrete action selection:

Experiment Setup:

We used a single actuator whose physical range of encoder positions was between 510 and 1023. The possible actions that the robot could take were {moveLeft, moveRight}. The reward function returned 1 any time the actuator moved to the extreme left position (encoder value 510). All other moves were given a reward of 0. Thus, it was as if our robot was being rewarded for banging it's head against a wall.



Algorithm:

We used an actor critic algorithm where the critic learned the value function for states observed by updating a value weight vector; and the actor used this to change the policy weight vectors. We used eligibility traces and function approximation for state and action representations. A boltzman softmax was used for action selection.

number of Tiles	8
number of Tilings	8
state representation	{encoder, speed}
reward	1 when moving to the extreme left (encoder = 510). 0 otherwise
step size policy (actor)	0.1
step size value (critic)	0.1

step size (average reward)	0.01
lambda	0.35
update frequency	0.2 seconds per update

Action selections were made using the boltzman softmax of each action (moving left or moving right). ie. Each action was assigned a probability of being taken using the estimated action value as it's feature vector multiplied by the policy weight vector.

Boltzman Softmax

$$\pi(a|s,\theta) = \frac{\exp[h(a,s,\theta)]}{\sum_b \exp[h(b,s,\theta)]}$$

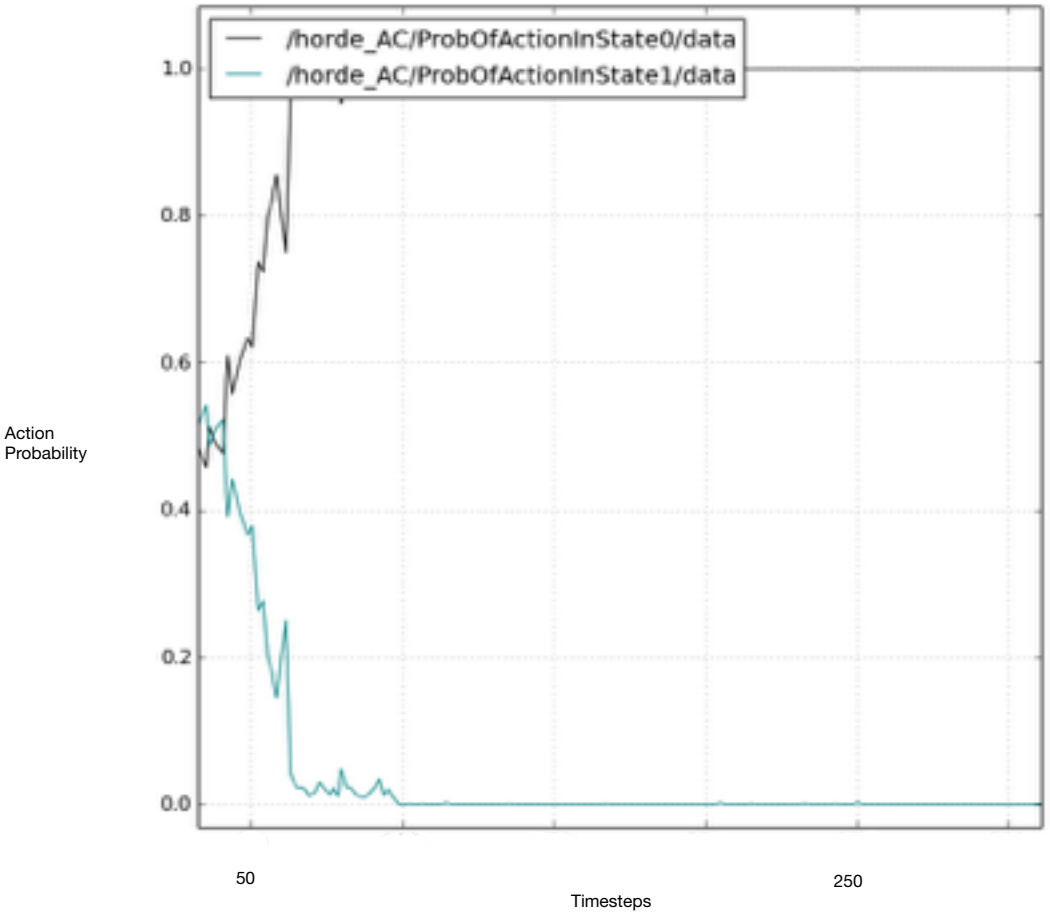
linear - $h(a,s,\theta) = \theta^T \phi(s,a)$ = action value

Given this as our policy, the gradient can be calculated and the general pseudocode of the update function at each time step can be seen as:

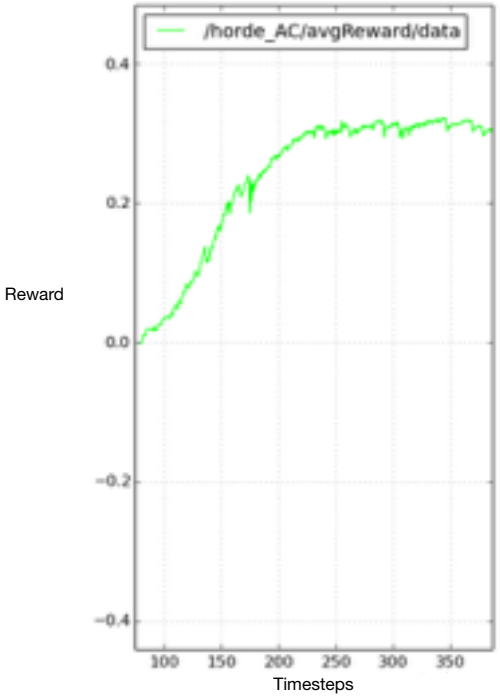
$e^\theta \leftarrow 0$
$e^w \leftarrow 0$
$\bar{R} \leftarrow 0$
$\theta \leftarrow 0$
$w \leftarrow 0$
$\alpha, \beta, \eta > 0$

Repeat:
- $A = \pi(\cdot|S,\theta)$
- Take A, observe S', R
- $\delta = R - \bar{R} + \phi_{S'}^T w - \phi_S^T w$
- $\bar{R} = \bar{R} + \eta\delta$
- $e^w = \lambda^w e^w + \phi_S$
- $e^\theta = \lambda^\theta e^\theta + \phi(S,A) - \sum_b \pi(b|S,\theta)\phi(S,b)$
- $w = w + \beta\delta e^w$
- $\theta = \theta + \alpha\delta e^\theta$
- $S = S'$

Results:
For our given setup, it is fairly obvious what the robot should do to maximize it's reward. Remember that a reward of 1 is given when it moves to the extreme left position, and 0 elsewhere. Therefore, it should learn to move to the extreme left position, move back one, move to the extreme left position, and repeat forever. Below are plots that demonstrate this.



This graph shows the probability distribution of the state just to the right of the extreme left. As you can see, to begin with, it's not certain which way to move, but very quickly learns to move left with near 100% probability.



This graph shows the reward increasing as the robot learns to move to the extreme left

Experiment 2: Continuous Action Selection

Experiment Setup:

We used the same single actuator setup as above - whose physical range of encoder positions was between 510 and 1023. However, the actions were no longer the discrete options of {left, right}, but rather were the continuous encoder values between 510 and 1023. The reward function was also altered slightly to be the distance away from the right extreme / 100. ie. (1023 - encoderPosition) / 100. Thus, the maximum reward (in the extreme left) was (1023-510)/100 = 5.13, and the minimum reward was 1023-1032/100 = 0. Clearly, the agent should learn to move to the extreme left and stay there.

Algorithm:
The algorithm used for discrete is very similar to the one used for continuous. The big change is that we now parameterize the policy by using a mean, and a variance.

The action taken above is drawn from the normal distribution with a mean and variance which is estimated using their respective weight vectors.

$$\text{action} = \text{sampleFromNormalDist}(\text{mean}, \text{variance})$$
$$= N(\theta_\mu^T x(s), e^{\theta_\sigma^T x(s)})$$

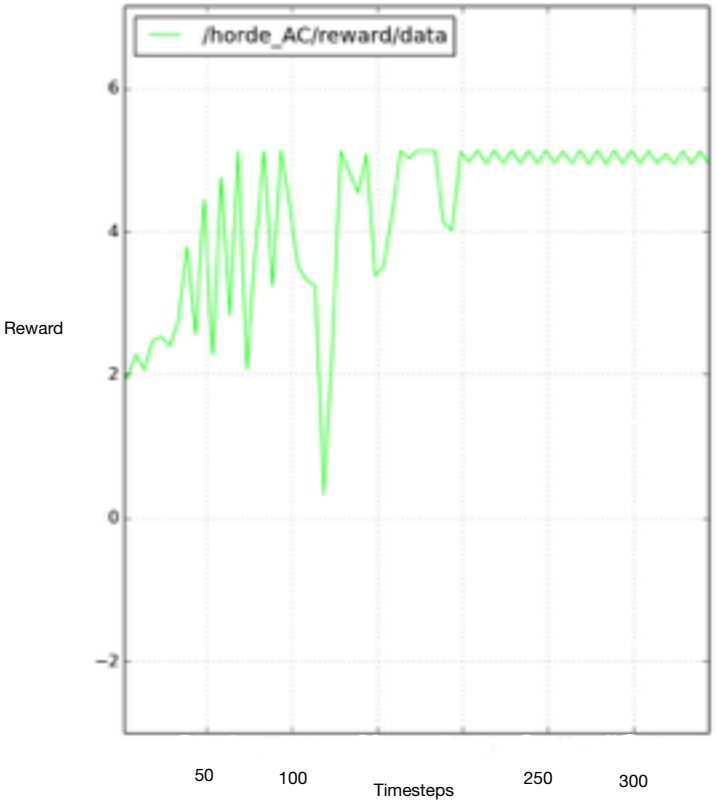
These whose values are learned. Thus we introduce eligibility traces, weight vectors, and learning rates for the mean and variance, rather than for just the policy. Since a normal distribution is used for action selection, the gradient is now different, and thus the weight update for these is also different.

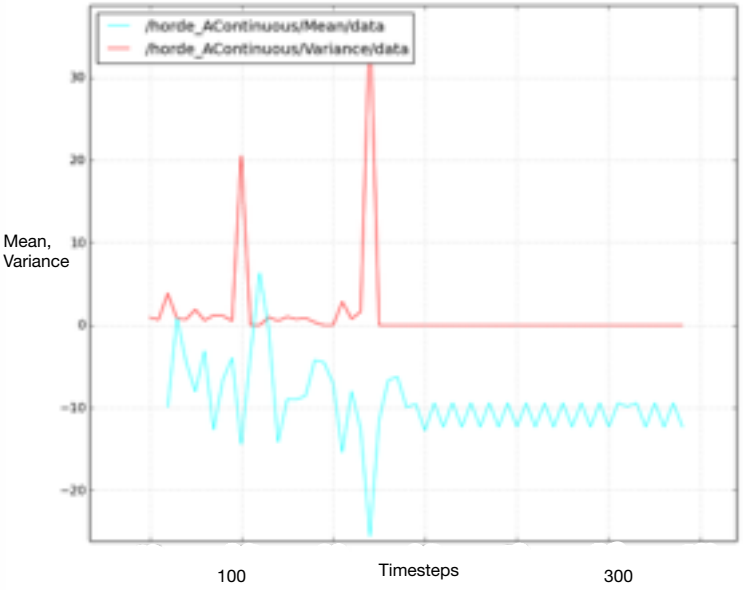
$$e_{\theta_\mu} = \lambda e_{\theta_\mu} + (a-\mu)x(s)$$
$$\theta_\mu = \theta_\mu + \alpha_{\theta_\mu} \delta e_{\theta_\mu}$$
$$e_{\theta_\sigma} = \lambda e_{\theta_\sigma} + ((a-\mu)^2 - \sigma^2)x(s)$$
$$\theta_\sigma = \theta_\sigma + \alpha_{\theta_\sigma} \delta e_{\theta_\sigma}$$

Experiment 2a: Large step sizes

For the first run of experiments, very aggressive step sizes were used. When this worked, it proved to work incredibly well, in the sense that the optimal action was very quickly learned. However, it proved to be incredibly fragile to occasional observations. In these situations, the weights would become polluted and the eligibility traces would grow exponentially, thus causing the algorithm to diverge. That said, below is a plot of the algorithm getting lucky and learning the optimal action quickly without any such explosion in weights.

number of Tiles	8
number of Tilings	8
state representation	{encoder, speed}
reward	(1023 - encoder) / 100. where 0<=reward<=5.3
step size value (value)	0.1
step size mean	0.1
step size variance	0.01
step size (average reward)	0.1
lambda	0.35
update frequency	0.2 seconds per update

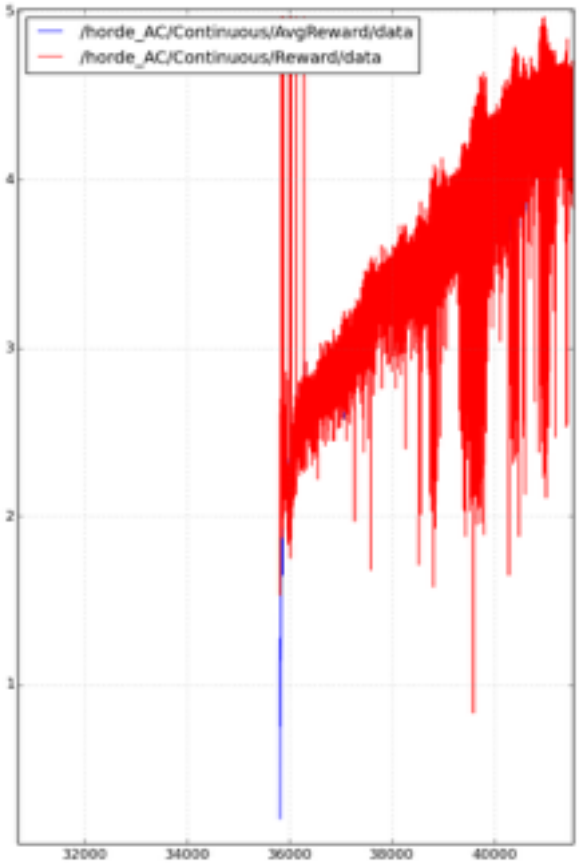




As you can see, the agent quickly learned to take the optimal action. After doing so, the mean action approached -10 (correlated to going to the extreme left. while the variance approached zero.

Experiment 2b: Smaller step sizes
In an effort to stabilize our algorithm, we decreased the step sizes for all parameters. This did indeed prevent us from seeing any explosions and divergences, but the learning was incredibly slow several orders of magnitude slower than with the previous parameter values. There seems to be a clear trade off between stability and speed, at least for this experimental configuration.

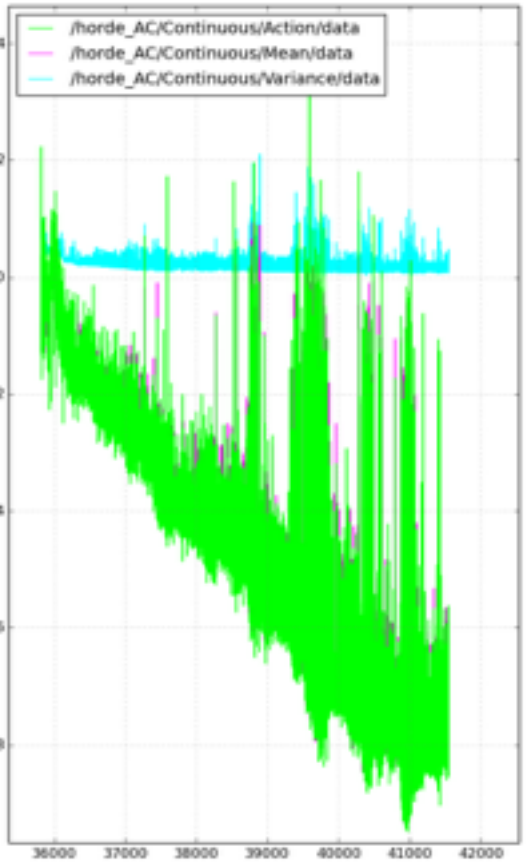
number of Tiles	8
number of Tilings	8
state representation	{encoder, speed}
reward	(1023 - encoder) / 100. where 0<=reward<=5.3
step size value (value)	0.05
step size mean	0.005
step size variance	0.005
step size (average reward)	0.0005
lambda	0.35
update frequency	0.2 seconds per update



Reward

The reward steadily increases towards the maximum value of 5.3.

Timesteps

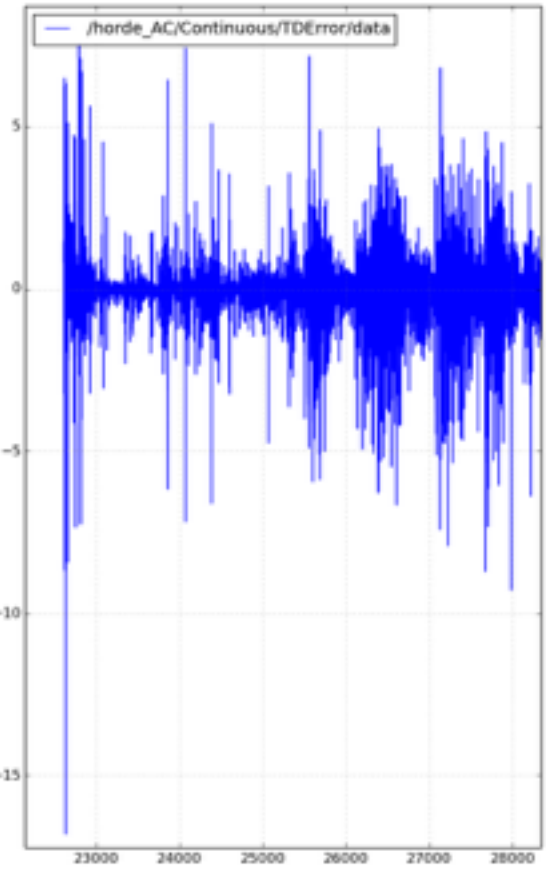


Action, Mean, Variance

Timesteps

The variance starts out high but decreases over time.

The smaller the action value, the larger the reward, so you can see the mean value decrease over time.



Td error

Timesteps

Future Work:
Because of the stochastic nature of a normal distribution, action selection is not bound within the range of possible values accepted by the robot actuator. In these situations, the action value was truncated to fall within the allowable region. The mean value was also truncated, otherwise, the mean would drift further and further away from -10 (the action correlating to the extreme left). Thus, we weren't truly following the gradient of the policy. Furthermore, for variance, a minimum and maximum value were set. The minimum was set to allow for continual exploration. And the maximum was set to prevent explosion of the variance. In all of these instances, the algorithm was being manipulated in such a way to slightly invalidate the gradient of the action selection.
One such workaround would be to create a distribution such that a maximum and minimum value could be selected corresponding to the max and min of the robots allowable actions.

Code:
All code can be found at https://github.com/dquail/RobotPerception/tree/RobotModule4/
and
https://github.com/dquail/RobotPerception/tree/RobotModule4Continuous

```
import numpy
import rospy
from std_msgs.msg import Float64
from horde.msg import StateRepresentation
from TileCoder import *

class ActorCriticContinuous():
    def __init__(self):
        self.maxAction = 1023.0
        self.minAction = 510.0
        self.numberOfFeatures = TileCoder.numberOfTilings * TileCoder.numberOfTiles * TileCoder.numberOfTiles

        #Eligibility traces and weights
        #Value / Critic
        self.elibibilityTraceValue = numpy.zeros(self.numberOfFeatures)
        self.valueWeights = numpy.zeros(self.numberOfFeatures)

        #Mean
        self.elibibilityTraceMean = numpy.zeros(self.numberOfFeatures)
        self.policyWeightsMean = numpy.zeros(self.numberOfFeatures)

        #Deviation
        self.elibibilityTraceVariance = numpy.zeros(self.numberOfFeatures)
        self.policyWeightsVariance = numpy.zeros(self.numberOfFeatures)

        self.lambdaPolicy = 0.35
        self.lambdaValue = 0.35
        self.averageReward = 0.0

        self.isRewardGoingLeft = True
       self.stepSizeValue = 0.01
        self.stepSizeVariance = 0.01
        self.stepSizeMean = 0.1
        self.rewardStep = 0.005

        self.i = 0

    def mean(self, state):
        m = numpy.inner(self.policyWeightsMean, state.X)
        if m > 10.0:
```

```
        m = 10
    if m < -10.0:
        m = -10.0
    return m

def variance(self, state):
    v = numpy.exp(numpy.inner(self.policyWeightsVariance, state.X))
    #on occasion, variance can be massive. so we bound it here
    if numpy.isinf(v):
        v = 5.0
    elif v < 0.001:
        v = 0.001
    return v

def pickActionForState(self, state):

    print("******* pickActionForState ***********")
    m = self.mean(state)
    v = self.variance(state) + 0.0000001 #to prevent against 0 variance
    pubMean = rospy.Publisher('horde_AC/Continuous/Mean', Float64, queue_size=10)
    pubMean.publish(m)

    pubVariance = rospy.Publisher('horde_AC/Continuous/Variance', Float64, queue_size=10)
    pubVariance.publish(v)

    print("mean: " + str(m) + ", variance: " + str(v))
    action = numpy.random.normal(m, v)
    #generally between -10 and 10. If greater or less than these values can cause overflow and underflow
    if ((state.encoder > 620) & (state.encoder < 700) & (state.speed <=0)):
        pubMeanSpecial = rospy.Publisher('horde_AC/Continuous/MeanInState', Float64, queue_size=10)
        pubMeanSpecial.publish(m)
        pubVarianceSpecial = rospy.Publisher('horde_AC/Continuous/VarianceInState', Float64, queue_size=10)
        pubVarianceSpecial.publish(v)

    if action < -10.0:
        action = -10.0 + 0.1*random.randint(1,4)

    if action > 10.0:
        action = 10.0 - 0.1*random.randint(1,4)

    print("action: " + str(action))

    pubAction = rospy.Publisher('horde_AC/Continuous/Action', Float64, queue_size=10)
    pubAction.publish(action)

    return action

def reward(self, previousState, action, newState):

    #Higher reward, the closer you are to 550

    rewardGoingLeft = (1023.0 - newState.encoder) / 100.0
    rewardGoingRight = (newState.encoder - 510.0)/10.0
    print("---- reward iteration: " + str(self.i))
    self.i = self.i + 1

    return rewardGoingLeft


def learn(self, previousState, action, newState):
    print("============= In Continuous actor critic learn =========")

    reward = self.reward(previousState, action, newState)
```

```
        print("previous encoder: " + str(previousState.encoder) + ", speed: " + str(previousState.speed) + ", new encoder: " +
str(newState.encoder) + " speed: " + str(newState.speed) +  ", action: " + str(action) + ", reward: " + str(reward))

        #Critic update
        tdError = reward - self.averageReward + numpy.inner(newState.X, self.valueWeights) - numpy.inner(previousState.X,
self.valueWeights)
        print("tdError: " + str(tdError))
        self.averageReward = self.averageReward + self.rewardStep * tdError
        print("Average reward: " + str(self.averageReward))
        self.elibibilityTraceValue = self.lambdaValue * self.elibibilityTraceValue + previousState.X
        self.valueWeights = self.valueWeights + self.stepSizeValue * tdError *                 self.elibibilityTraceValue

        m = self.mean(previousState)
        v = self.variance(previousState)

        #Mean Update
        self.elibibilityTraceMean = self.lambdaPolicy * self.elibibilityTraceMean + ((action - m) * previousState.X)
        self.policyWeightsMean = self.policyWeightsMean + self.stepSizeMean * tdError * self.elibibilityTraceMean

        #Variance Update
        logPie = (numpy.power(action - m, 2) - numpy.power(v, 2)) * previousState.X
        self.elibibilityTraceVariance = self.lambdaPolicy * self.elibibilityTraceVariance + logPie
        self.policyWeightsVariance = self.policyWeightsVariance + self.stepSizeVariance * tdError * self.elibibilityTraceVariance

        pubReward = rospy.Publisher('horde_AC/Continuous/Reward', Float64, queue_size=10)
        pubReward.publish(reward)

        pubTD = rospy.Publisher('horde_AC/Continuous/TDError', Float64, queue_size=10)
        pubTD.publish(tdError)

        pubEncoder = rospy.Publisher('horde_AC/Continuous/Encoder', Float64, queue_size=10)
        pubEncoder.publish(newState.encoder / 100.0)

        pubAvgReward = rospy.Publisher('horde_AC/Continuous/AvgReward', Float64, queue_size=10)
        pubAvgReward.publish(self.averageReward)
        print("============= End continuous actor critic learn =============")
        print("-")
```