Fachhochschule Südwestfalen
University of Applied Sciences

Systems Engineering

Motion Control of Peristaltic Sorting Machine by the Application of Reinforcement Learning Method of Actor-Critic
(Assignment - 2)

Submitted to:

Prof. Dr. Andreas Schwung

Supervised by:

Mr. Fabian Westbrink

Submitted by:

Murtaza Khuzema Basuwala (10062949),

Nutakki Pradeep Chakravarthi (10062965)

SEEM MSc Program

WS-2018/19

Table of Contents

1. Introduction 3

2. Implementation with Keras 3

2.1 Implementation 3

2.2 Results and Evaluation 6

3. Implementation with Pytorch 10

3.1 Entropy Loss Function 10

3.2 Loss Function 11

3.3 Implementation 11

3.4 Results and Evaluation 16

4. Conclusion 20

References 28

1. Introduction

The task of this assignment is to implement the actor-critic algorithm to control the motion of the peristaltic sorting machine in order to reach the parcel position in the most efficient way. To implement the actor-critic methodology two different approaches were used. The first implementation was done using Keras library where there was no exploration employed and the second implementation was made with Pytorch library where entropy was employed in order to explore the state space and try out all the possible actions to get the optimal policy and achieve the maximum cumulative reward. From the given environment of the sorting machine the state, action, and rewards are described as follows,

Table 1: Given State, Action and Rewards structure

State	State = [Actuator position / Packet Position / Actuator Velocity]
	State space of nine elements where the first three elements give information about the actuator position, the next three elements tells about the parcel position and the last three elements give the velocity of the actuators in x, y and z axis respectively.
Action	An array of three elements with possible values of 1,0 and -1
	Discrete action space with 27 ($3 \times 3 \times 3$) possible action combinations.
	Each action represents the direction of acceleration for the actuator in x,y and z axis.
Rewards	An action is performed, and the target reaches the parcel position, then R = 1
	An action is performed, and the target does not reach the parcel then, R -0.1
	An action is performed such that the target should reach the parcel position but it lands in a different position then, R = -10

As discussed that the actor-critic uses two neural networks, the actor defines the current policy and therefore it takes an action based on the current policy. The critic on the other hand, estimates the value for each state based on the current policy defined by the actor. The actual reward is then compared to the value estimated by the critic which gives the error. After certain number of steps the, agent updates its policy and value by bootstrapping in order to improve its predictions in the future.

2. Implementation with Keras

2.1 Implementation

a) Libraries

Figure 1 shows the different relevant libraries that are imported for the execution of the algorithm

```
import sys
import environment_seminare
import pylab
import numpy as np
import itertools
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import Adam
```

Figure 1: Libraries imported

b) Hyper Parameters

Figure 2 shows the hyperparameters which are chosen based on trial and error method

```
class A2CAgent:
 def __init__(self, state_size, action_size):

 # get size of state and action
  self.state_size = state_size
  self.action_size = action_size
  self.value_size = 1
 # These are hyper parameters for the Policy Grad
  self.discount_factor = 0.99
  self.actor_lr = 0.001
  self.critic_lr = 0.002
 # create model for policy network
  self.actor = self.build_actor()
  self.critic = self.build_critic()
```

Figure 2: Hyper Parameters

c) Actor :

The actor neural network as shown below in figure 3 is created using the sequential model to create a linear stack of three layers.

```
def build_actor(self):
 actor = Sequential()
 actor.add(Dense(32, input_dim=9, activation='relu', kernel_initializer='he_uniform'))
 actor.add(Dense(32, activation='relu'))
 actor.add(Dense(27, activation='softmax', kernel_initializer='he_uniform'))
 actor.summary()
 actor.compile(loss='categorical_crossentropy',
 optimizer=Adam(lr=self.actor_lr))
 return actor
```

Figure 3: Actor Neural Network

The first layer is a dense layer of 32 neurons(random selection) with an input dimension of nine since there are nine elements in the states. Rectified Linear Unit (RELU) is chosen as the activation function which gives the output as zero if the input is less than zero and output equals to input if the input is greater than zero. RELU also gives the benefit that if there are many neurons defined where almost half of the inputs have negative values then due to the characteristics of RELU (output is zero if the input is negative) only those neurons are fired which carry a positive value. The second layer is a dense hidden layer with 32 neurons and with the same activation function RELU. The third layer is the output layer which has 27 neurons because of the 27 possible action combinations as discussed in table 1.

The softmax activation function is chosen for the output layer which gives the highest probability of a particular action from the 27 possible actions based on the current policy. The actor takes the action a and enters a new state s_{t+1} while gaining a reward r along the way. The categorical cross entropy loss function is chosen to train the neural network based on the loss between the computed output and target output. Finally, Adam optimizer algorithm is chosen to update the network weights based on the given learning rate for the actor.

d) Critic

Similarly like actor, the critic neural network is also created with a sequential model of three layers. Figure 4 shows the critic neural network.

```
def build_critic(self):
 critic = Sequential()
 critic.add(Dense(32, input_dim=9, activation='relu',
 kernel_initializer='he_uniform'))
 critic.add(Dense(32, activation='relu'))
 critic.add(Dense(self.value_size, activation='linear', kernel_initializer='he_uniform'))
 critic.summary()
 critic.compile(loss="mse", optimizer=Adam(lr=self.critic_lr))
 return critic
```

Figure 4: Critic Neural Network

The neural network for the critic is similar to that of the actor but the only difference is in the output layer which has only one neuron which outputs the value function of the current state. Moreover, the linear activation function is used in the output layer as it outputs only one value i.e. the action value function of the state. Based on the action a taken by the actor the agent lands up in a new state s_{t+1} and achieves a new reward r. The critic estimates the value of the state based on the current policy which is defined by the actor.

e) Training the Network

The variable 'target' and 'advantages' are used in training the network. Using these variables as baseline the neural network estimates the loss and updates the weights of the network. The 'advantages' variable is used by the actor-network to update itself while the critic network is updated by estimating its loss using the 'target' variable. Figure-5 shows how the network is trained.

```
def train_model(self, next_state, reward, done, final_state ,place, Error):
 target = np.zeros((1, self.value_size))
 advantages = np.zeros((1,27))
 value = self.critic.predict(state)[0]
 next_value = self.critic.predict(next_state)[0]
 if done:
  advantages[0][action] = reward - value
  target[0][0] = reward
 else:
  advantages[0][action] = reward + self.discount_factor * (next_value) - value
  target[0][0] = reward + self.discount_factor * next_value
 actor_loss = self.actor.fit(state, advantages, epochs=1, verbose=0)
 critic_loss = self.critic.fit(state, target, epochs=1, verbose=0)
 return actor_loss.history['loss'], critic_loss.history['loss']
```

Figure 5: Training the Network

f) Working the Network

Figure 6 shows how the algorithm is connected to the give environment in order to initiate the training of the network.

```
if __name__ == "__main__":
 env = environment_seminare.environment()
 # get size of state and action from environment
 state = env.reset()
 state_size = 9
 action_size = 27
 action_index = np.array([p for p in itertools.product([-1,0,1], repeat=3)])
 # make A2C agent
 agent = A2CAgent(state_size, action_size)
 scores, episodes, rewards = [], [], []
 for e in range(EPISODES):
  done = False
  score = 0
  state = env.reset()
  state = np.reshape(state, [1, state_size])
  while not done:
   ind = np.array(agent.get_action(state),dtype='int32')
   action = action_index[ind][:]
   #print("action\n",action)
   next_state, reward, done, final_state ,place, Error = env.Step(state,action[0])
   next_state = np.reshape(next_state, [1, state_size])
 # if an action make the episode end, then gives penalty of -100
   agent.train_model(next_state, reward, done, final_state ,place, Error)
   reward = reward if not done or score == 49 else -10
   score += reward
   state = next_state
  if done:
 # every episode, plot the play time
   score = score if score == 50.0 else score + 10
   scores.append(score)
   episodes.append(e)
```

Figure 6: Working of the network

As discussed before that the action is an array of three elements with 27 possible combinations, therefore to work these arrays with the neural network, indexing method is used. A list of numbers ranging from 0 to 27 is given as the input to the output layer of the actor-network. The actor based on the current policy chooses the highest probable index number from this list which is estimated by the 'softmax' function. The variable 'action_index' carries all the 27 possible combinations of the action arrays. Based on the index number obtained from the actor-network the action is selected by just looking for the corresponding action array in the 'action_index' for that index number.

For every episode in the while statement, the reward from the environment is collected and stored in the variable 'score' which sums the total reward collected for every episode. A negative reward of 10 is given if the total score of the collected reward is less than 49 or if the condition becomes true. On the other hand, if all the episodes are completed then the total score collected by accumulating all the rewards are stored if the score is 50 else +10 is added to the score.

Note: The full code is attached in Appendix-1.

2.2 Results and Evaluation

There was three simulations performed based on a different number of episodes which are shown below. All simulation is performed on four hidden layers and 25 neurons.

- **Simulation - 1**

Episodes = 800

The set of hyperparameters chosen for this simulation were,

Actor learning rate = 0.0001

Critic learning rate = 0.05

Rewards (in the environment):

R = 1; R = -0.1; R = -0.2

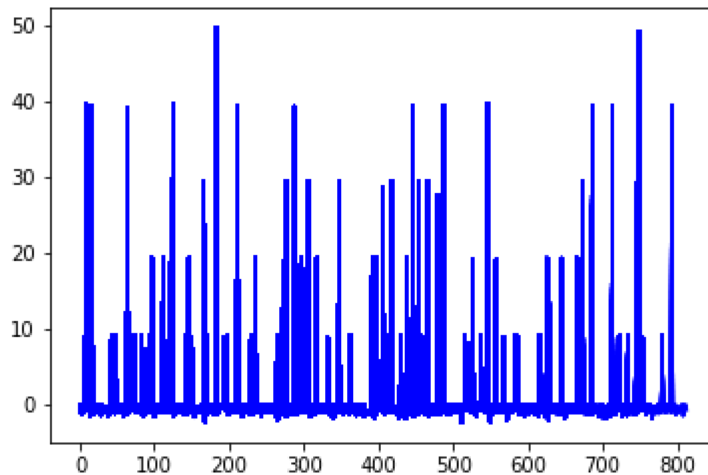

Figure 7: Simulation-1 (Keras) [*]

From the above figure, it's concluded that the agent starts exploring and takes actions which gets him high rewards. The highest score achieved by the agent is 50. Moreover, the agent learns something but the mean of the rewards stay near zero.

❖ Simulation - 2

Episode - 2500

The set of hyper parameters chosen for this simulation were,

Actor learning rate = 0.0001

Critic learning rate = 0.005

Rewards (in the environment):

R = 1; R = -0.1; R = -0.2

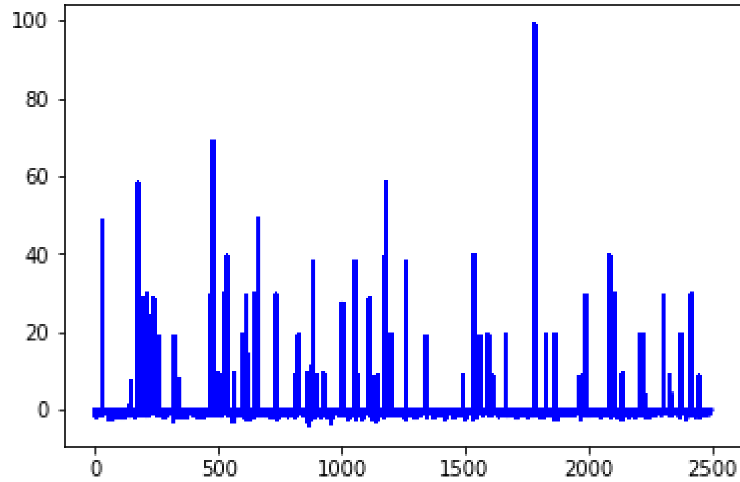

Figure 8: Simulation-2 (Keras)[*]

From figure 8, The results are the same as before, but the mean of the reward is between -1 to 1. The highest score achieved by the agent is 5.

- **Simulation - 3**

Episode - 10000

The set of hyper parameters chosen for this simulation were,

Actor learning rate = 0.0001

Critic learning rate = 0.0002

Rewards (in the environment):

R = 1; R = -0.1; R = -0.2

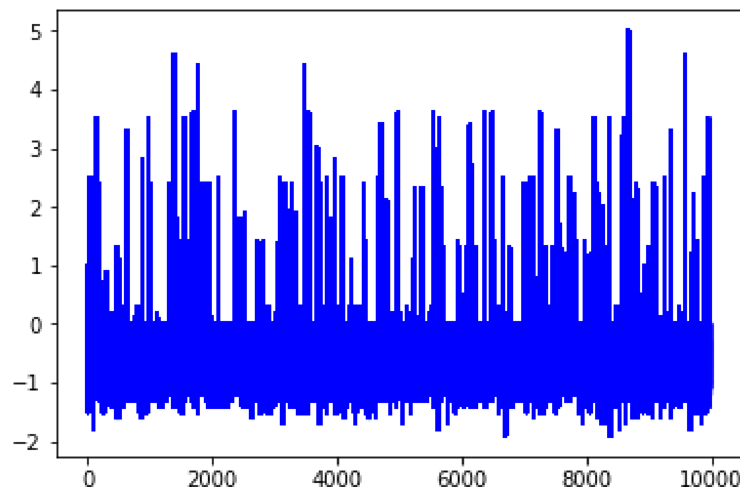

Figure 9: Simulation-3 (Keras) [*]

From figure 8, The results are the same as before, but the mean of the reward is between -1 to 1. The highest score achieved by the agent is 5.

❖ Keras – Batch Normalization

In order to optimize the learning batch normalization was implemented in the actor and critic network which gets the input from the input layer calculates its mean and variance and then applies normalization. This normalized output is then sent to the activation function. Theoretically, it is proved that batch normalization helps in achieving faster convergence. Figure 10 shows the implementation of batch normalization in the networks [1].

```
def build_actor(self):
 actor = Sequential()
 actor.add(Dense(25, input_dim=9))
 actor.add(BatchNormalization())
 actor.add(Activation("relu"))
 actor.add(Dense(25, activation='relu'))
 actor.add(Dropout(0.2))
 actor.add(Dense(27, activation='softmax', kernel_initializer='he_uniform'))
 actor.summary()
 actor.compile(loss='categorical_crossentropy', optimizer=Adam(lr=self.actor_lr))
 return actor
# critic: state is input and value of state is output of model
def build_critic(self):
 critic = Sequential()
 critic.add(Dense(25, input_dim=9))
 critic.add(BatchNormalization())
 critic.add(Activation("relu"))
 critic.add(Dense(25, activation='relu'))
 critic.add(Dropout(0.2))
 critic.add(Dense(self.value_size, activation='linear', kernel_initializer='he_uniform'))
 critic.summary()
 critic.compile(loss="mse", optimizer=Adam(lr=self.critic_lr))
 return critic
```

Figure 10: Actor-Critic Batch Normalization

❖ Simulation - 4

Episode – 2500

The set of hyperparameters chosen for this simulation were,

Actor learning rate = 0.001

Critic learning rate = 0.05

Rewards (in the environment):

$R = 1; R = -0.1; R = -0.2$

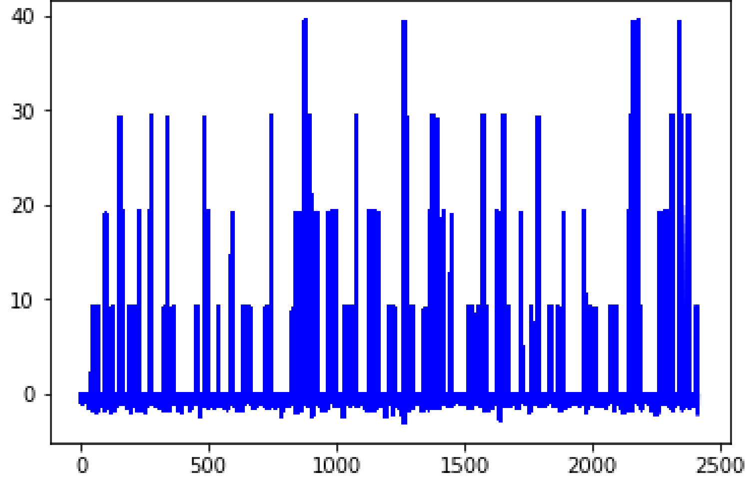

Figure 11: Simulation-4 [*]

From the above simulations, it was concluded that one of the reasons for no gradual increase in the mean rewards may be due to insufficient exploration. So to implement exploration

3. Implementation with Pytorch

3.1 Entropy Loss Function

In order to encourage exploration, entropy is introduced into the loss function which avoids premature convergence and finds the most suitable policy. Entropy works by bringing all the states to the same probability because the gain in information is higher when all the states are at the same level of probability and less if one states dominate the other. Bringing all the states to the same probability makes the network work very hard in choosing a particular action and this makes the network to choose any random actions which encourages exploration [2]. The equation for entropy is given as,

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i)$$

A high probability action is penalized by increasing the loss through the entropy term which brings all the actions to almost the same probability. The entropy loss is given by just taking the above entropy equation and plugging in the policy instead of the random value x. A scaling factor β is used to scale the entropy loss and a relatively low value is chosen so that it does not dominate the loss that is received from the loss function [2]. The entropy loss function is given by,

$$H\big(\pi(a_t|s_{t,}\theta_\pi)\big) = -\sum_{t} \pi(a_t|s_{t,}\theta_\pi) \log \pi\,(a_t|s_{t,}\theta_\pi)$$

3.2 Loss Function

The above entropy loss function is subtracted from the standard policy gradient loss function i.e.

$$L_\pi = -\log\left(\pi\left(a_t|s_{t,}\theta_\pi\right)\right)\delta_t - \beta H\left(\pi(a_t|s_{t,}\theta_\pi)\right)$$

The term δ_t is the TD error at step t which is in product with the log of the probability of taking a certain action a at time t which makes up the overall policy gradient loss function. The policy gradient loss function is used to maximize the expected future log rewards. Now, this loss function is added with the entropy loss function which adds more cost to the actions that quickly dominate over other actions and doing this initiates exploration. Since in Pytorch it's not possible to maximize a function therefore the negative of these functions is minimized which is mathematically equivalent to maximization [2]. The value loss is also added in this loss function which is the mean square error (MSE) of the predicted value of the state and the discounted rewards i.e.

$$L_v = \left(v\left(s_{t,}\right) - G_{t,}\right)^2$$

To initiate learning a scaling factor ς is used for scaling the value function gradient which brings the gradient to the same magnitude so that they don't out pass each other. The final loss function is given by [2],

$$min\ L_\pi = -\log\left(\pi\left(a_t|s_{t,}\theta_\pi\right)\right)\delta_t - \beta H\pi\left(a_t|s_{t,}\theta_\pi\right) + \varsigma\left(v\left(s_{t,}\right) - G_{t,}\right)^2 \quad \text{(A)}$$

Figure 9 below shows the visual representation of the above equation

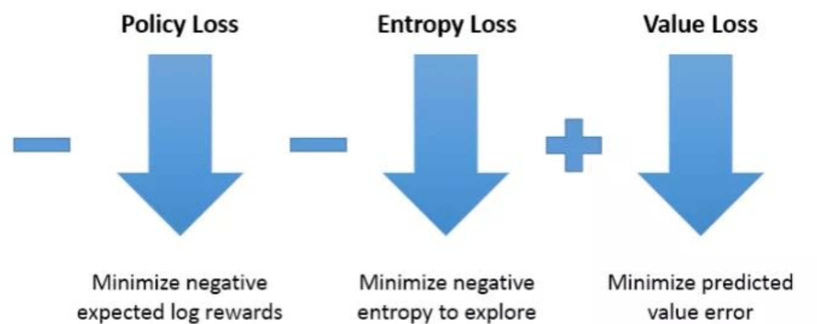

Figure 12: Loss function [2]

3.3 Implementation

a) Importing Libraries

Figure 11 shows the relevant libraries that are imported for the execution of the algorithm

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import itertools
from collections import OrderedDict
import environment_seminare
import matplotlib.pyplot as plt
from matplotlib import gridspec
```

Figure 13: Libraries Imported

b) Network Architecture

Instead of using two different neural networks which are used in the case of Keras where one predicts the policy and other predicts the value, a single network is constructed here from which both the value and policy are estimated. This construction is known as two-headed A2C Network [2]. Figure 12 shows how a single network is headed to estimate both policy and head.

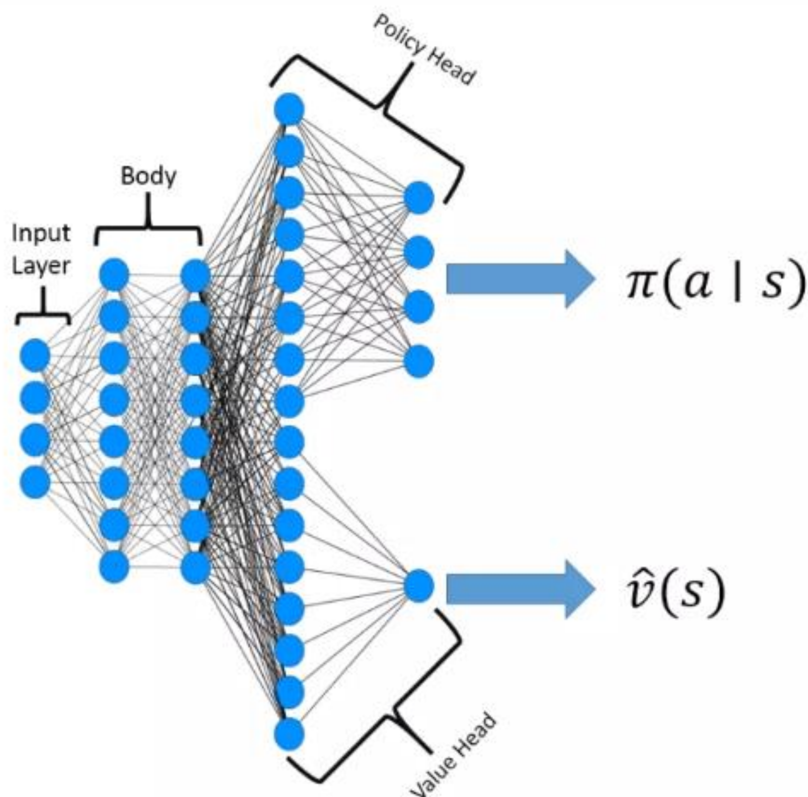

Figure 14: Two-Headed A2C Network [2]

```python
class actorCriticNet(nn.Module):
    def __init__(self, env, n_hidden_layers=4, n_hidden_nodes=32,
                 learning_rate=0.01, bias=False, device='cpu'):
        super(actorCriticNet, self).__init__()

        self.device = device
        self.n_inputs = 9
        self.n_outputs = 27
        self.n_hidden_nodes = n_hidden_nodes
        self.n_hidden_layers = n_hidden_layers
        self.learning_rate = learning_rate
        self.bias = bias
        self.action_space = np.array(self.n_outputs)

        # Generate network according to hidden layer and node settings
        self.layers = OrderedDict()
        self.n_layers = 2 * self.n_hidden_layers
        for i in range(self.n_layers + 1):
            # Define single linear layer
            if self.n_hidden_layers == 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_inputs,
                    self.n_outputs,
                    bias=self.bias)
            # Define input layer for multi-layer network
            elif i % 2 == 0 and i == 0 and self.n_hidden_layers != 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_inputs,
                    self.n_hidden_nodes,
                    bias=self.bias)
            # Define intermediate hidden layers
            elif i % 2 == 0 and i != 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_hidden_nodes,
                    self.n_hidden_nodes,
                    bias=self.bias)
            else:
                self.layers[str(i)] = nn.ReLU()

        self.body = nn.Sequential(self.layers)

        # Define policy head
        self.policy = nn.Sequential(
            nn.Linear(self.n_hidden_nodes,
                      self.n_hidden_nodes,
                      bias=self.bias),
            nn.ReLU(),
            nn.Linear(self.n_hidden_nodes,
                      self.n_outputs,
                      bias=self.bias))
        # Define value head
        self.value = nn.Sequential(
            nn.Linear(self.n_hidden_nodes,
                      self.n_hidden_nodes,
                      bias=self.bias),
            nn.ReLU(),
            nn.Linear(self.n_hidden_nodes,
                      1,
                      bias=self.bias))

        if self.device == 'cuda':
            self.net.cuda()

        self.optimizer = torch.optim.Adam(self.parameters(),
                                          lr=self.learning_rate)
```

Figure 15: Network Architecture

The network architecture as shown in figure 15 is constructed by using 'OrderedDict' where every even layer is a linear layer and every odd layer is an activation function. The 'linear' layer is equivalent to the 'Dense' layer in Keras. There are four hidden layers chosen for each actor and critic neural network. The learning rate defined here is the same for both the actor and critic since they both share the same network. The policy head and value head defined in the figure corresponds to the respective actor and critic network. Once again, Adam optimizer is chosen to update the network weight based on the learning rate.

c) Estimating various parameters

Figure 16 shows the various function used to estimate different parameters. The function 'def predict' is used to predict the probability distribution for the action and the value for each state. The function 'get_body_output' converts the states into float tensors which are better computable. The next function 'get_action' chooses the index of the action based on the probability estimated by the 'softmax' function. Finally, the 'get_log_probs' function estimates the logarithmic of the probability estimated by the 'softmax' function which will be used the calculate the loss function.

```
def predict(self, state):
    body_output = self.get_body_output(state)
    probs = F.softmax(self.policy(body_output), dim=-1)
    #print(probs)
    return probs, self.value(body_output)

def get_body_output(self, state):
    state_t = torch.FloatTensor(state).to(device=self.device)
    return self.body(state_t)

def get_action(self, state):
    probs = self.predict(state)[0].detach().numpy().flatten()
    a = np.random.choice(27, p=probs)
    return a
    #print("Its me", action2)

def get_log_probs(self, state):
    body_output = self.get_body_output(state)
    logprobs = F.log_softmax(self.policy(body_output), dim=-1)
    return logprobs
```

Figure 16: Predicting the probability, value, action based on probability and the logarithmic probability for the loss

d) Computing the Losses

The entropy loss, policy loss and the value loss which are discussed in the equation (A) are computed as shown in figure 17. These losses are the important pillars of the algorithm as they are used to ensure that the agent is not captivated in a local minimum rather it explores all the possible states. The policy loss and the entropy loss are maximized which ensures taking good reward action in the future and increase exploration while the value loss

is minimized which reduces the error in the predicted value of the state and the discounted rewards.

```
def calc_loss(self, states, actions, rewards, advantages, beta=-0.001):
    actions = torch.LongTensor(actions).to(self.network.device)
    rewards_t = torch.FloatTensor(rewards).to(self.network.device)
    advantages_t = torch.FloatTensor(advantages).to(self.network.device)
    log_probs = self.network.get_log_probs(states).view(-1,27)
    log_prob_actions = advantages_t * log_probs[range(len(actions)), actions]
    policy_loss = -log_prob_actions.mean()
    action_probs, values = self.network.predict(states)
    entropy_loss = -self.beta * (action_probs * log_probs).sum(dim=1).mean()
    value_loss = self.zeta * nn.MSELoss()(values.squeeze(-1), rewards_t)
    self.policy_loss.append(policy_loss)
    self.value_loss.append(value_loss)
    self.entropy_loss.append(entropy_loss)
    return policy_loss, entropy_loss, value_loss
```

Figure 17: Computing Losses

e) Updating the Losses

The losses are updated by first computing their gradients using the 'backward()' function and then it is updated. The total policy loss is calculated by measuring the difference between the policy loss and the entropy loss. The overall loss is computed by summing up the policy loss, entropy loss, and value loss. The network is then updated by the computed gradient using the 'optimizer' function.

```
def update(self, states, actions, rewards, advantages):
    self.network.optimizer.zero_grad()
    policy_loss, entropy_loss, value_loss = self.calc_loss(states, actions, rewards, advantages)
    total_policy_loss = policy_loss - entropy_loss
    self.total_policy_loss.append(total_policy_loss)
    total_policy_loss.backward(retain_graph=True)

    value_loss.backward()

    total_loss = policy_loss + value_loss + entropy_loss
    self.total_loss.append(total_loss)
    self.network.optimizer.step()
```

Figure 18: Computing Losses

By updating the weights of the network using these losses push the gradient of the policy in the right direction to choose the best action for a particular sate to gain maximum cumulative reward.

Note: The code is attached in Appendix-2

3.4 Results and Evaluation

The results of the simulation which are performed for different episodes are discussed below.

- **Simulation-1 (Episodes-5000; Epoch-10)**

Hyperparameters:

- Learning rate = $1.3 \times e^{-3}$
- $\beta = 0.01$
- $\varsigma = 0.1$
- No of hidden layers = 2
- No of neurons = 25
- Reward (Environment) = +10, -0.1, -0.2

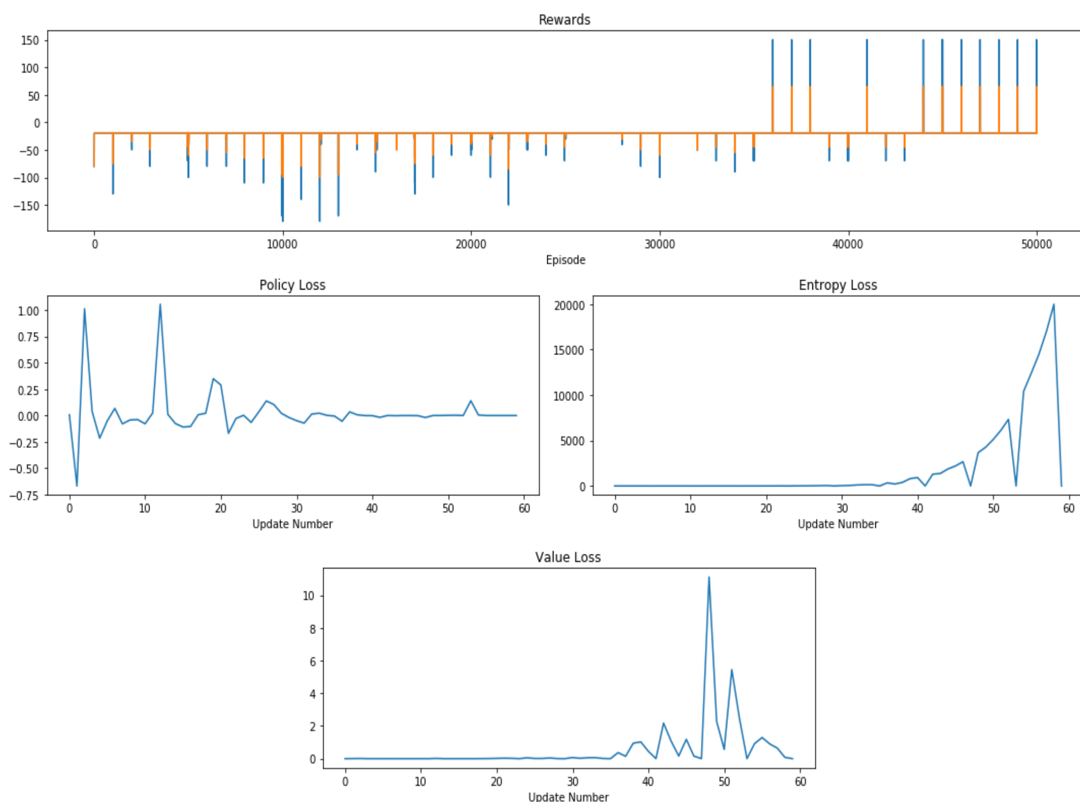

Figure 19: Simulation-1 [*]

From figure 14, it is observed that the agent initially takes actions which give negative rewards. Approximately after 35000 episodes, the agent takes actions which give high rewards and then continues it consistently to take positive until the end of the end of the episode. From the policy loss graph, it is seen that the policy changes significantly until it is updated for at least 25 times and after that, it stabilizes following a single policy. The value loss remains to be zero until it has been updated for approximately 36 times but after that, there is a large difference in the value estimated and the discounted rewards. The entropy loss

is zero initially but after updating for about 40 times the exploration increases gradually reaching maximum at the end of the episode.

- **Simulation-2 (Episodes-5000; Epoch-10)**

Hyperparameters:

- Learning rate = $1.7 \times e^{-4}$
- $\beta = 0.005$
- $\varsigma = 0.01$
- No of hidden layers = 4
- No of neurons = 25
- Reward (Environment) = +10, -0.1, -0.2

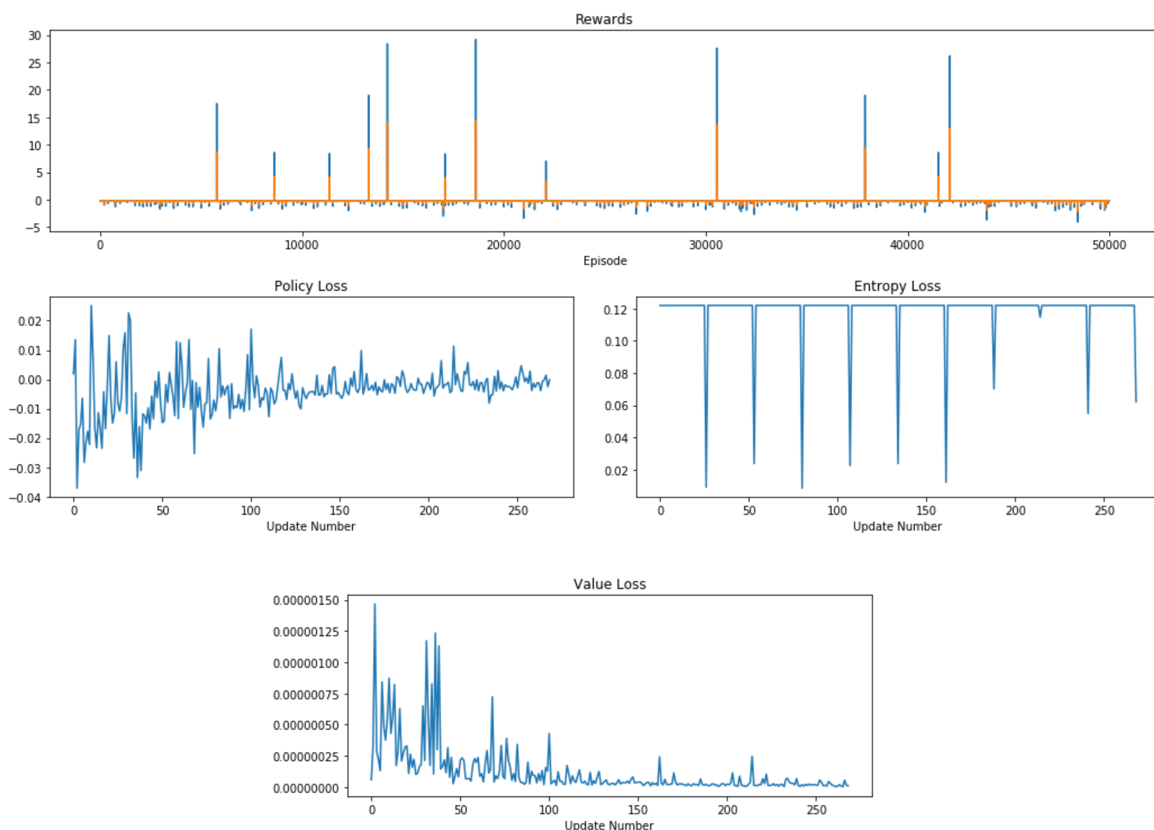

Figure 20: Simulation-2 [*]

The agent manages to achieve a few positive rewards but there was no learning observed. From the loss graph, there is considerable change in the policy and value till an update of 100 after which it gets stabilized.

- **Batch Normalization**

In order to improve the learning of the agent a layer of batch-normalization was implemented in the neural net which calculates the mean and the variance of the inputs, normalizes them and sends as an output to the next layer. Figure 16 shows the implementation of batch normalization in the network.

```
        else:
            self.layers[str(i)] = nn.ReLU(nn.BatchNorm1d(9))

    self.body = nn.Sequential(self.layers)
```

Figure 21: Batch Normalization [*]

❖ Simulation-3 (Episodes-50000; Epoch-10)

Hyperparameters:

- Learning rate = $1.4 \times e^{-4}$
- $\beta = 0.001$
- $\varsigma = 0.1$
- No of hidden layers = 4
- No of neurons = 25
- Reward (Environment) = +10, -0.1, -0.2

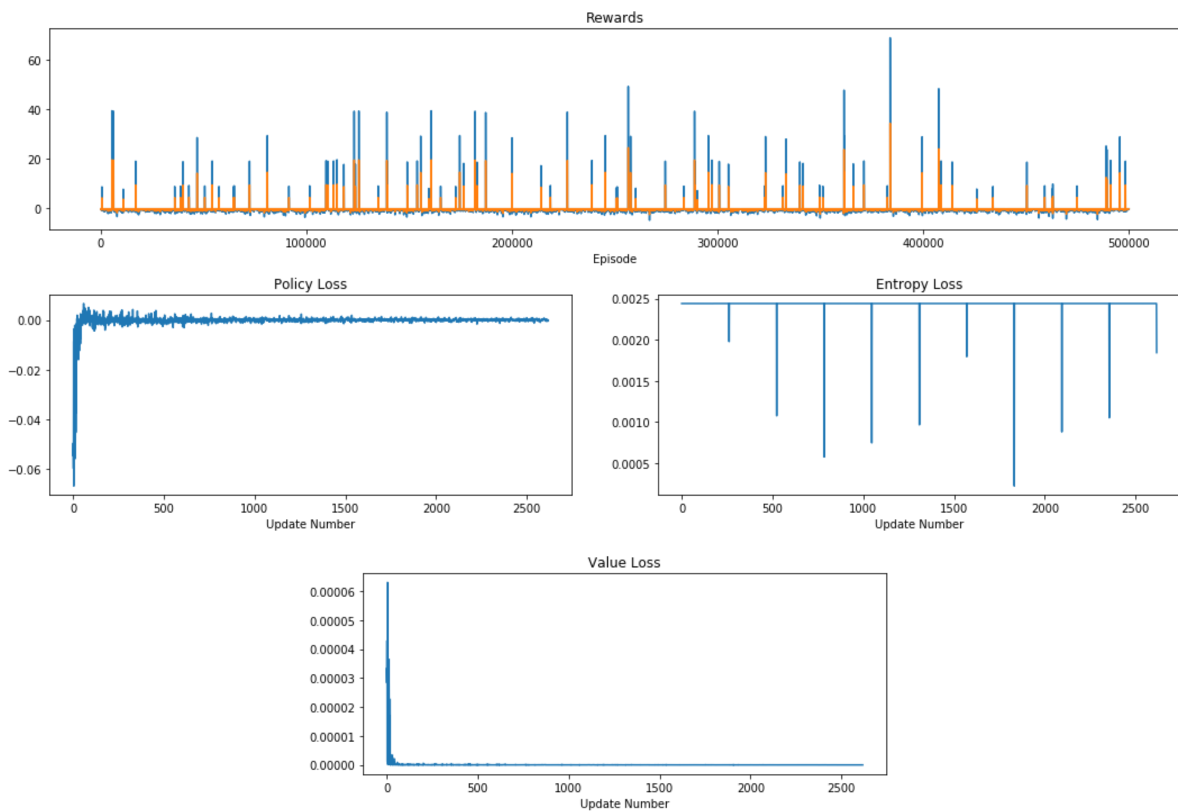

Figure 22: Simulation-3 [*]

From figure, it is illustrated that the agent initially started taking actions which gave fewer rewards and improved slowly in gaining more rewards later. The agent did learn something but the learning process was not enough. The policy loss and value loss are on the same scale having an overshoot in the respective losses initially, and then being stable throughout all the updates.

❖ Simulation- 4 (Episodes-10000; Epoch-10)

Hyperparameters:

- Learning rate = $5 \times e^{-4}$
- $\beta = 0.001$
- $\varsigma = 0.001$
- No of hidden layers = 4
- No of neurons = 32

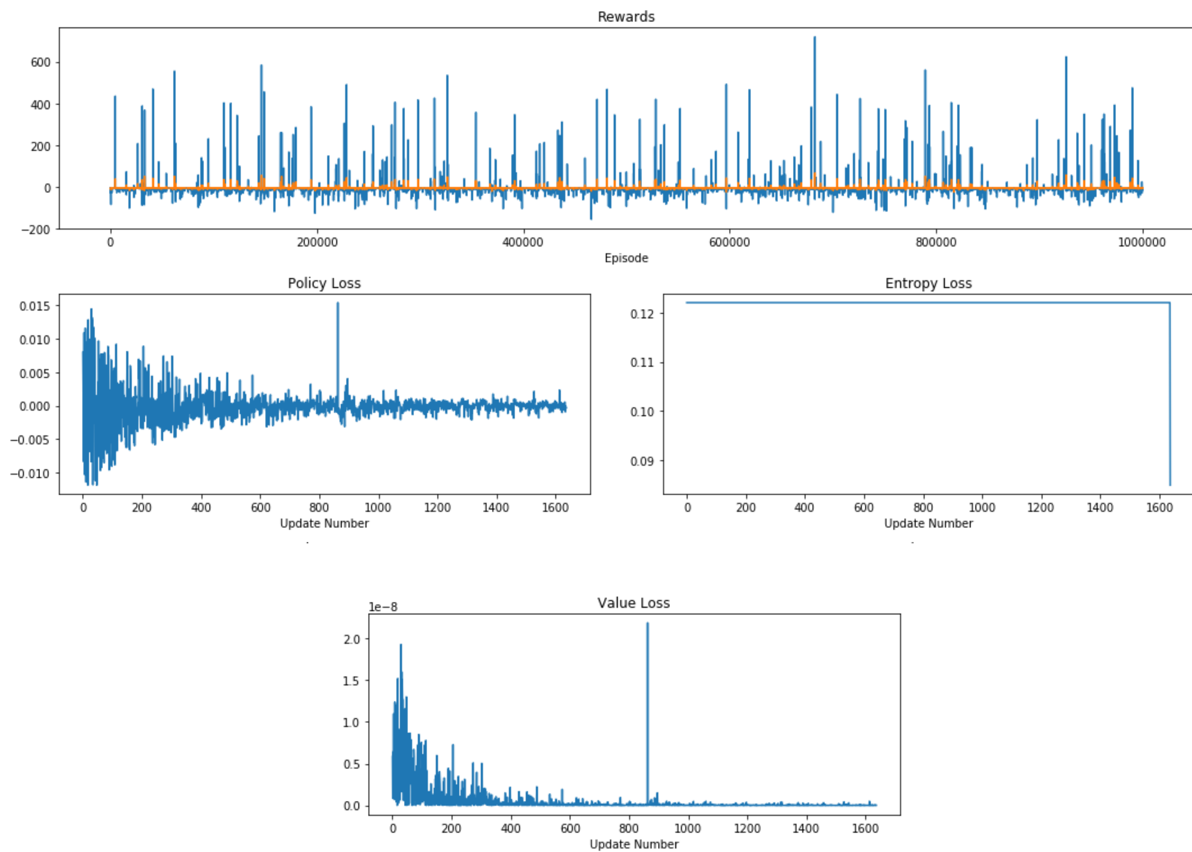

Figure 23: Simulation-4 [*]

❖ Simulation- 5 (Episodes-50000; Epoch-10)

Hyperparameters:

- Learning rate = $5.5 \times e^{-4}$
- $\beta = 0.0001$
- $\varsigma = 0.01$
- No of hidden layers = 4
- No of neurons = 25

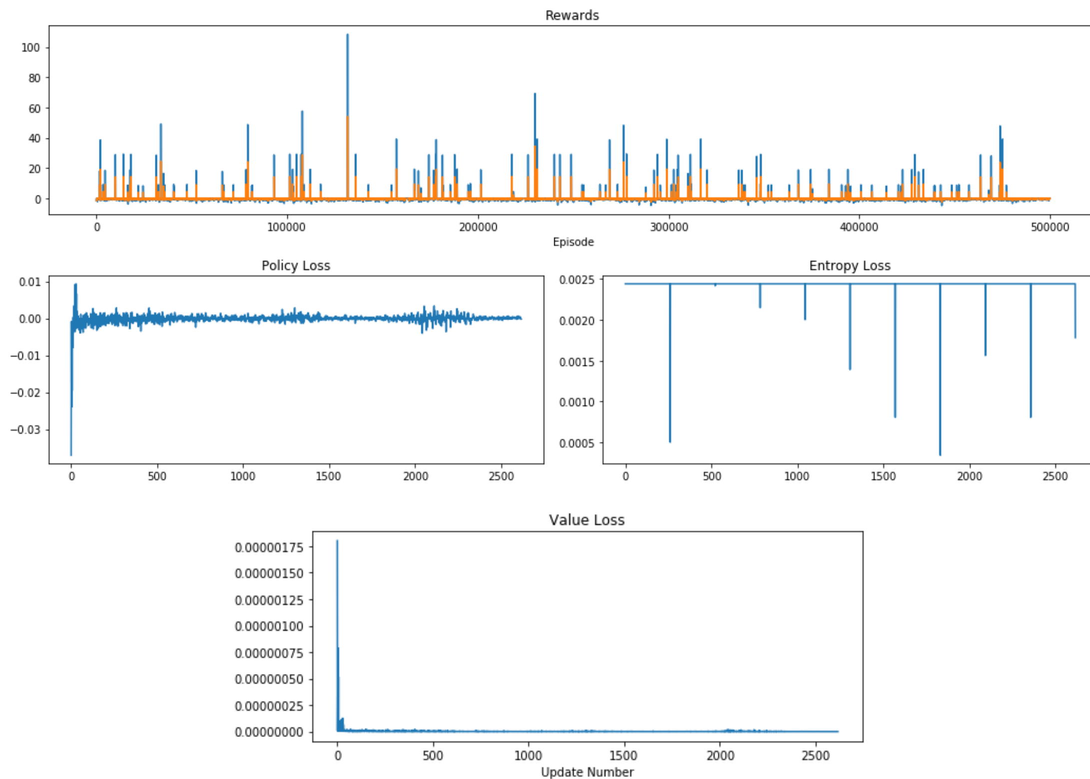

Figure 24: Simulaion-5[*]

4. Conclusion

The results from both the algorithm (Keras and Pytorch) reveals that the agent learns something throughout the episode. Initially, due to lack of exploration, the agent was thought of being constrained in a local minimum which allowed it to take only a few sets of actions from the 27 different possibilities. To deal with this issue entropy was implemented to encourage exploration and also the policy and value loss were back propagated in order to update the gradients of the network. The main challenge faced during this assignment was to set the right hyperparameters for both the algorithms. Many simulations were performed by observing the graph obtained and then increasing or decreasing the learning rate and other hyperparameters in suitable steps. The agent performed learning at some places but it was not consistent throughout. Therefore, it is concluded that with the right set of hyperparameters the agent can converge to an optimum result. Hence, more tuning of these hyperparameters is needed to achieve good learning.

❖ Appendix-1 [3]:

Coding with Keras

```
"""
ALGORITHM - 1
System Engineering Assignment-2
@author: Murtaza Khuzema Basuwala, Pradeep Nutakki Chakravarthi


The algorithm is designed for PSM (Peristaltic Sorting Machine) environment

This code is referenced from  'https://github.com/rlcode/reinforcement-learning'
"""
import sys
import environment_seminare
import pylab
import numpy as np
import itertools
from keras.layers import Dense, Dropout, BatchNormalization, Activation
from keras.models import Sequential
from keras.optimizers import Adam

EPISODES = 2500
# A2C(Advantage Actor-Critic) agent for the PSM_Sorting_Machine
class A2CAgent:
 def __init__(self, state_size, action_size):
  self.render = False
  self.load_model = False
 # get size of state and action
  self.state_size = state_size
  self.action_size = action_size
  self.value_size = 1
 # Hyper-parameters
  self.discount_factor = 0.99
  self.actor_lr = 1.3e-3
  self.critic_lr = 0.05
 # create model for policy network
  self.actor = self.build_actor()
  self.critic = self.build_critic()
  if self.load_model:
   self.actor.load_weights("D:\PSM_actor.h5")
   self.critic.load_weights("D:\PSM_critic.h5")
 # approximate policy and value using Neural Network
 # actor: state is input and probability of each action is output of model
 def build_actor(self):
  actor = Sequential()
  actor.add(Dense(25, input_dim=9))
  actor.add(BatchNormalization())
  actor.add(Activation("relu"))
  actor.add(Dense(25, activation='relu'))
  actor.add(Dropout(0.2))
  actor.add(Dense(27, activation='softmax', kernel_initializer='he_uniform'))
  actor.summary()
  actor.compile(loss='categorical_crossentropy', optimizer=Adam(lr=self.actor_lr))
  return actor
 # critic: state is input and value of state is output of model
 def build_critic(self):
  critic = Sequential()
```

```
  critic.add(Dense(25, input_dim=9))
  critic.add(BatchNormalization())
  critic.add(Activation("relu"))
  critic.add(Dense(25, activation='relu'))
  critic.add(Dropout(0.2))
  critic.add(Dense(self.value_size, activation='linear', kernel_initializer='he_uniform'))
  critic.summary()
  critic.compile(loss="mse", optimizer=Adam(lr=self.critic_lr))
  return critic
 # using the output of policy network, pick action stochastically
 def get_action(self, state):
  policy = self.actor.predict(state, batch_size=1).flatten()
  return np.random.choice(27, 1, p=policy)

 # update policy network every episode
 def train_model(self, next_state, reward, done, final_state ,place, Error):
  target = np.zeros((1, self.value_size))
  advantages = np.zeros((1,27))
  value = self.critic.predict(state)[0]
  next_value = np.argmax(self.critic.predict(next_state)[0])
  if done:
   advantages[0][action] = reward - value
   target[0][0] = reward
  else:
   advantages[0][action] = reward + self.discount_factor * (next_value) - value
   target[0][0] = reward + self.discount_factor * next_value
  actor_loss = self.actor.fit(state, advantages, epochs=1, verbose=0)
  critic_loss = self.critic.fit(state, target, epochs=1, verbose=0)
  return actor_loss.history['loss'], critic_loss.history['loss']

if __name__ == "__main__":
 env = environment_seminare.environment()
 # get size of state and action from environment
 state = env.reset()
 state_size = 9
 action_size = 27
 action_index = np.array([p for p in itertools.product([-1,0,1], repeat=3)])
 # make A2C agent
 agent = A2CAgent(state_size, action_size)
 scores, episodes, rewards = [], [], []
 for e in range(EPISODES):
  done = False
  score = 0
  state = env.reset()
  state = np.reshape(state, [1, state_size])
  while not done:
   ind = np.array(agent.get_action(state),dtype='int32')
   action = action_index[ind][:]
   #print("action\n",action)
   next_state, reward, done, final_state ,place, Error = env.Step(state,action[0])
   next_state = np.reshape(next_state, [1, state_size])
 # if an action make the episode end, then gives penalty of -100
   agent.train_model(next_state, reward, done, final_state ,place, Error)
   reward = reward if not done or score == 49 else -10
   score += reward
   state = next_state
  if done:
 # every episode, plot the play time
   score = score if score == 50.0 else score + 10
   scores.append(score)
```

```
  episodes.append(e)
 #loss.append(loss)
  pylab.plot(episodes, scores, 'b')
  #print("episode:", e, " / SCORE:", score, " / REWARD:", reward, " / ERROR:", Error)

 # if the mean of scores of last 10 episode is bigger than 49
 # stop training
  if np.mean(scores[-min(10, len(scores)):]) > 49:
   sys.exit()
 # save the model
 if e % 50 == 0:
  agent.actor.save_weights("D:\psm_actor.h5")
  agent.critic.save_weights("D:\psm_critic.h5")
```

❖ Appendix-2 [2]:

Coding with Pytorch

```
"""
ALGORITHM - 2
System Engineering Assignment-2
@author: Murtaza Khuzema Basuwala, Pradeep Nutakki Chakravarthi


The algorithm is designed for PSM (Peristaltic Sorting Machine) environment

This code is referenced from 'https://www.datahubbs.com/two-headed-a2c-network-in-pytorch/'
"""


import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import itertools
from collections import OrderedDict
import environment_seminare
import matplotlib.pyplot as plt
from matplotlib import gridspec
#%matplotlib inline

class actorCriticNet(nn.Module):
    def __init__(self, env, n_hidden_layers=4, n_hidden_nodes=25,
             learning_rate=5e-4, bias=False, device='cpu'):
        super(actorCriticNet, self).__init__()

        self.device = device
        self.n_inputs = 9
        self.n_outputs = 27
        self.n_hidden_nodes = n_hidden_nodes
        self.n_hidden_layers = n_hidden_layers
        self.learning_rate = learning_rate
        self.bias = bias
        self.action_space = np.array(self.n_outputs)

        # Generate network according to hidden layer and node settings
        self.layers = OrderedDict()
        self.n_layers = 2 * self.n_hidden_layers
```

```
        for i in range(self.n_layers + 1):
            # Define single linear layer
            if self.n_hidden_layers == 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_inputs,
                    self.n_outputs,
                    bias=self.bias)
            # Define input layer for multi-layer network
            elif i % 2 == 0 and i == 0 and self.n_hidden_layers != 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_inputs,
                    self.n_hidden_nodes,
                    bias=self.bias)
            # Define intermediate hidden layers
            elif i % 2 == 0 and i != 0:
                self.layers[str(i)] = nn.Linear(
                    self.n_hidden_nodes,
                    self.n_hidden_nodes,
                    bias=self.bias)
            else:
                self.layers[str(i)] = nn.ReLU(nn.BatchNorm1d(9))

        self.body = nn.Sequential(self.layers)

        # Define policy head
        self.policy = nn.Sequential(
            nn.Linear(self.n_hidden_nodes,
                      self.n_hidden_nodes,
                      bias=self.bias),
            nn.ReLU(),
            nn.Linear(self.n_hidden_nodes,
                      self.n_outputs,
                      bias=self.bias))
        # Define value head
        self.value = nn.Sequential(
            nn.Linear(self.n_hidden_nodes,
                      self.n_hidden_nodes,
                      bias=self.bias),
            nn.ReLU(),
            nn.Linear(self.n_hidden_nodes,
                      1,
                      bias=self.bias))

        if self.device == 'cuda':
            self.net.cuda()

        self.optimizer = torch.optim.Adam(self.parameters(),
                                          lr=self.learning_rate)

    def predict(self, state):
        body_output = self.get_body_output(state)
        probs = F.softmax(self.policy(body_output), dim=-1)
        #print("PROBS", probs)
        return probs, self.value(body_output)

    def get_body_output(self, state):
        state_t = torch.FloatTensor(state).to(device=self.device)
        return self.body(state_t)

    def get_action(self, state):
```

```
        probs = self.predict(state)[0].detach().numpy().flatten()
        #print("PROBS", probs)
        a = np.random.choice(27, p=probs)
        #print("Index", a)
        return a
        #print("Its me", action2)

    def get_log_probs(self, state):
        body_output = self.get_body_output(state)
        logprobs = F.log_softmax(self.policy(body_output), dim=-1)
        #print("LOGPROBS", logprobs)
        return logprobs

class A2C():
    def __init__(self, env, network):

        self.env = env
        self.network = network
        self.ep_rewards = []
        self.kl_div = []
        self.policy_loss = []
        self.value_loss = []
        self.entropy_loss = []
        self.total_policy_loss = []
        self.total_loss = []

        #self.action_space = np.arange(27)
    def generate_episode(self):
        states, actions, next_states, rewards, dones, final_states, places, Errors = [], [], [], [], [], [], [], []
        counter = 0
        state = env.reset()
        state = np.reshape(state,[1,9])
        #print("STATE", state)
        action_index =  np.array([p for p in itertools.product([-1,0,1], repeat=3)])
        total_count = self.batch_size * self.n_steps
        while counter < total_count:
            done = False
            while done == False:
                ind = np.array(self.network.get_action(state),dtype='int32')
                #print(ind)
                action = action_index[ind][:]
                #print(action_index)
                #print("ation in generate episode", action)
                next_state, reward, done, final_state, place, Error = env.Step(state,action)
                next_state = np.reshape(next_state, [1, 9])
                self.reward += reward
                states.append(state)
                actions.append(ind.tolist())
                final_states.append(final_state)
                next_states.append(next_state)
                places.append(place)
                rewards.append(reward)
                dones.append(done)
                Errors.append(Error)
                state = next_state
                #print("STATES", states)

                if done:
                    self.ep_rewards.append(self.reward)
                    self.state = self.env.reset()
```

```
            self.reward = 0
            self.ep_counter += 1
            if self.ep_counter >= self.num_episodes:
                counter = total_count
                break

        counter += 1
        if counter >= total_count:
            break
    return states, actions, final_states, rewards, places, dones, next_states, Errors

def calc_rewards(self, batch):
    states, actions, rewards, final_states, places, dones, next_states, Errors = batch
    rewards = np.array(rewards)
    total_steps = len(rewards)

    state_values = self.network.predict(states)[1]
    next_state_values = self.network.predict(next_states)[1]
    done_mask = torch.ByteTensor(dones).to(self.network.device)
    next_state_values[done_mask] = 0.0
    state_values = state_values.detach().numpy().flatten()
    next_state_values = next_state_values.detach().numpy().flatten()

    G = np.zeros_like(rewards, dtype=np.float32)
    td_delta = np.zeros_like(rewards, dtype=np.float32)
    dones = np.array(dones)

    for t in range(total_steps):
        last_step = min(self.n_steps, total_steps - t)

        # Look for end of episode
        check_episode_completion = dones[t:t+last_step]
        if check_episode_completion.size > 0:
            if True in check_episode_completion:
                next_ep_completion = np.where(check_episode_completion == True)[0][0]
                last_step = next_ep_completion

        # Sum and discount rewards
        G[t] = sum([rewards[t+n:t+n+1] * self.gamma ** n for
                n in range(last_step)])

    if total_steps > self.n_steps:
        G[:total_steps - self.n_steps] += next_state_values[self.n_steps:] \
            * self.gamma ** self.n_steps
    td_delta = G - state_values
    return G, td_delta

def train(self, n_steps=50, batch_size=2, num_episodes=2000,
        gamma=0.90, beta=1e-3, zeta=1e-3):
    self.n_steps = n_steps
    self.gamma = gamma
    self.num_episodes = num_episodes
    self.beta = beta
    self.zeta = zeta
    self.batch_size = batch_size
    self.state = self.env.reset()
    self.reward = 0
    self.ep_counter = 0
    while self.ep_counter < num_episodes:
        batch = self.generate_episode()
```

```
        G, td_delta = self.calc_rewards(batch)
        states = batch[0]
        #print(states)
        actions = batch[1]
        self.update(states, actions, G, td_delta)
        print("\rMean Rewards: {:.2f} Episode: {:d}    ".format(
            np.mean(self.ep_rewards[-10:]), self.ep_counter), end="")
def plot_results(self):
    avg_rewards = [np.mean(self.ep_rewards[i:i + self.batch_size])
                   if i > self.batch_size
        else np.mean(self.ep_rewards[:i + 1]) for i in range(len(self.ep_rewards))]

    plt.figure(figsize=(15,10))
    gs = gridspec.GridSpec(3, 2)
    ax0 = plt.subplot(gs[0,:])
    ax0.plot(self.ep_rewards)
    ax0.plot(avg_rewards)
    ax0.set_xlabel('Episode')
    plt.title('Rewards')

    ax1 = plt.subplot(gs[1, 0])
    ax1.plot(self.policy_loss)
    plt.title('Policy Loss')
    plt.xlabel('Update Number')

    ax2 = plt.subplot(gs[1, 1])
    ax2.plot(self.entropy_loss)
    plt.title('Entropy Loss')
    plt.xlabel('Update Number')

    ax3 = plt.subplot(gs[2, 0])
    ax3.plot(self.value_loss)
    plt.title('Value Loss')
    plt.xlabel('Update Number')
    plt.tight_layout()
    plt.show()

def calc_loss(self, states, actions, rewards, advantages, beta=-0.001):
    actions = torch.LongTensor(actions).to(self.network.device)
    rewards_t = torch.FloatTensor(rewards).to(self.network.device)
    advantages_t = torch.FloatTensor(advantages).to(self.network.device)
    log_probs = self.network.get_log_probs(states).view(-1,27)
    log_prob_actions = advantages_t * log_probs[range(len(actions)), actions]
    policy_loss = -log_prob_actions.mean()
    action_probs, values = self.network.predict(states)
    entropy_loss = -self.beta * (action_probs * log_probs).sum(dim=1).mean()
    value_loss = self.zeta * nn.MSELoss()(values.squeeze(-1), rewards_t)
    self.policy_loss.append(policy_loss)
    self.value_loss.append(value_loss)
    self.entropy_loss.append(entropy_loss)
    return policy_loss, entropy_loss, value_loss

def update(self, states, actions, rewards, advantages):
    self.network.optimizer.zero_grad()
    policy_loss, entropy_loss, value_loss = self.calc_loss(states, actions, rewards, advantages)
    total_policy_loss = policy_loss - entropy_loss
    self.total_policy_loss.append(total_policy_loss)
    total_policy_loss.backward(retain_graph=True)
    value_loss.backward()
    total_loss = policy_loss + value_loss + entropy_loss
```

```
        self.total_loss.append(total_loss)
        self.network.optimizer.step()

env = environment_seminare.environment()
net = actorCriticNet(env, learning_rate= 5e-4, n_hidden_layers=4, n_hidden_nodes=25)
a2c = A2C(env, net)
for i in range(10):
    a2c.train(n_steps=100, num_episodes=50000, beta=0.0001, zeta=.01)
a2c.plot_results()
```

References

[1] Peccia F. *Batch normalization: theory and how to use it with Tensorflow*. Available at: https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad; 2018 [accessed 25.01.2019].

[2] https://www.facebook.com/christian.hubbs1. *Two-Headed A2C Network in PyTorch - DataHubbs*. Available at: https://www.datahubbs.com/two-headed-a2c-network-in-pytorch/; 2018 [accessed 23.01.2019].

[3] *rlcode/reinforcement-learning*. Available at: https://github.com/rlcode/reinforcement-learning [accessed 27.11.2018].