In [13]:

```
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
env = gym.make('MountainCarContinuous-v0')
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
```

mountain car env.

$state = [x, v]$ where $x \in [-1.2, 0.5]$ and $v \in [-0.07, 0.07]$

$action \in [-1, 1]$

$reward = -a^2$ if $x < 0.5$, $100 - a^2$ if $x >= 0.5$

I have used following parameterized policies-

- $\pi_\theta(s, a) = \theta^T . z = h(z)$ where $\theta \in R^4$ and $z = [1, x, v, a]^T$, $x \to position$ $v \to velocity$ and $a \to action$,
- $\pi_\theta(s, a) = sin(h(z))$ where h(z) is the above function.
- $\pi_\theta(s, a) = tanh(h(z))$. and
- $\pi_\theta(a/s) = N(\mu, \sigma)$ where $\mu = \theta^T . z$ and $\sigma = 1$ as constant (also took other values)

  and tried reinforce algorithm to solve the mountain-car problem but this is not work. so I switched to parameterized policy with deep learning. and used actor-critic algorithm.

Actor-Critic Algorithm

- **Actor**

  policy function $\pi_\theta(a/s) = N(\mu, \sigma)$

  loss function $-log(N(a|\mu(s_t), \sigma(s_t)).(G_t - v(s_t, w))$
- **Critic**

  state value function $v(s, w)$

  loss function $(G_t - v(s_t, w))^2$

where θ and w are Deep-Neural-Network parameter

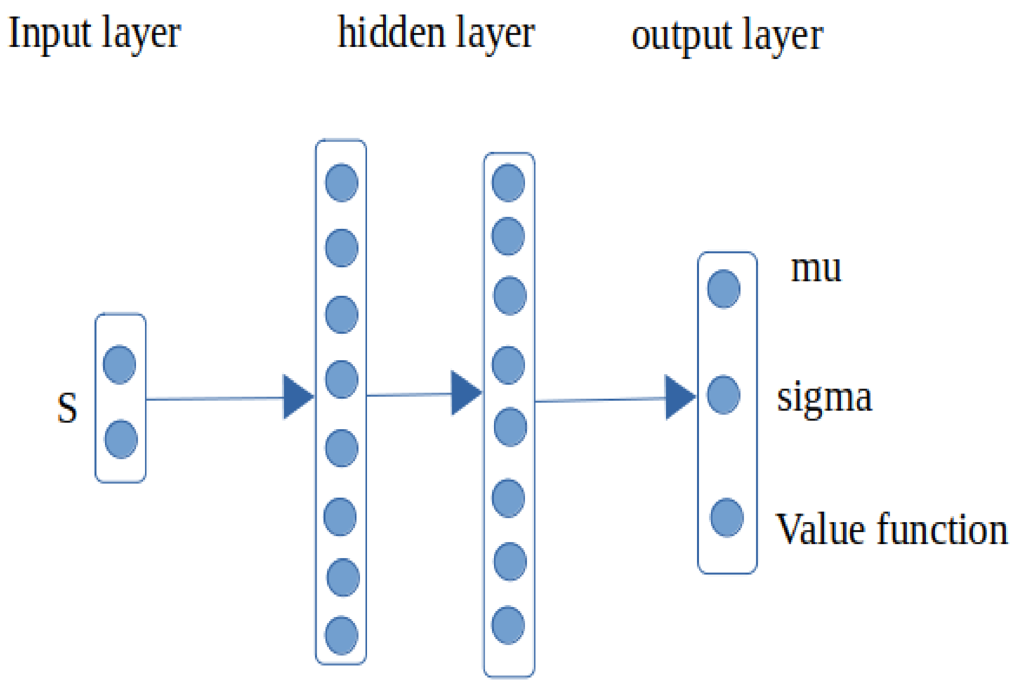


Figure 1 : Model-Architecture

ActorCriticModel(

(fc1): Linear(in_features=2, out_features=200, bias=True)

(fc2): Linear(in_features=200, out_features=200, bias=True)

(mu): Linear(in_features=200, out_features=1, bias=True)

(sigma): Linear(in_features=200, out_features=1, bias=True)

(value): Linear(in_features=200, out_features=1, bias=True))

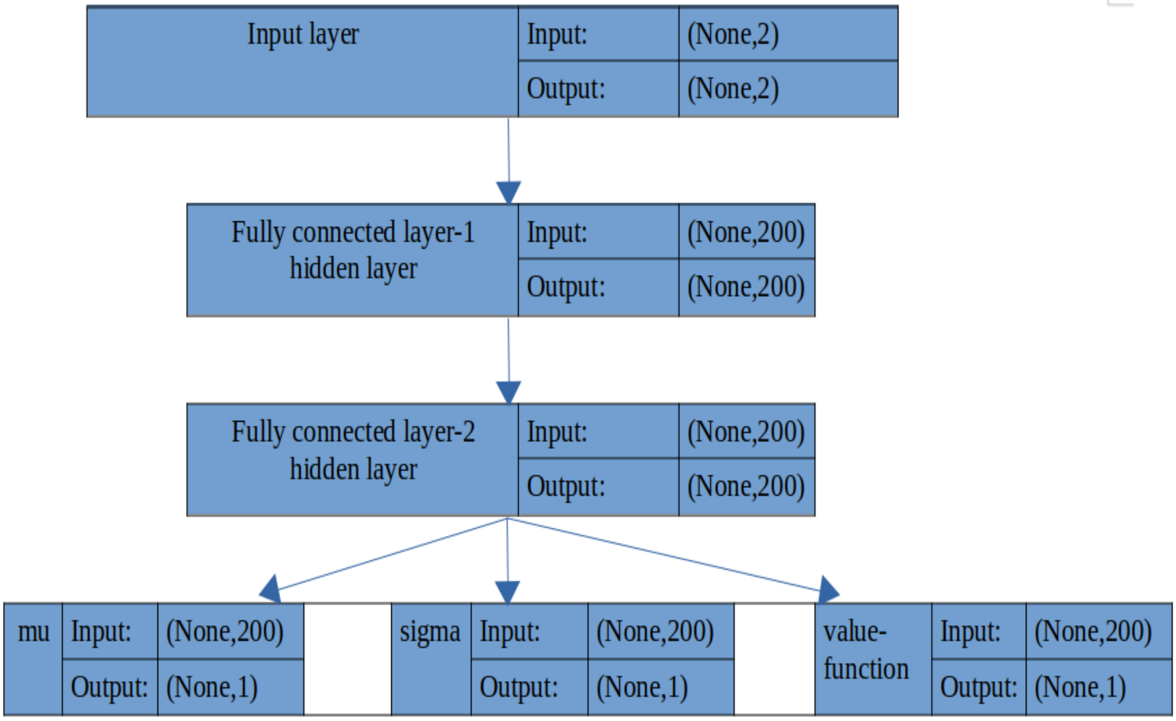


Figure 2 : Model-Architecture

below I have defined ActorCriticModel Class which follows the above neural-network architecture

In [2]:

```
class ActorCriticModel(nn.Module):
    def __init__(self, n_input, n_output, n_hidden1,n_hidden2):
        super(ActorCriticModel, self).__init__()
        self.fc1 = nn.Linear(n_input, n_hidden1)
        self.fc2 = nn.Linear(n_hidden1,n_hidden2)
        self.mu = nn.Linear(n_hidden2, n_output)
        self.sigma = nn.Linear(n_hidden2, n_output)
        self.value = nn.Linear(n_hidden2, 1)
        self.distribution = torch.distributions.Normal

    def forward(self, x):
        x = F.elu(self.fc1(x))
        x = F.elu(self.fc2(x))
        mu = 2 * torch.tanh(self.mu(x))
        sigma = F.softplus(self.sigma(x)) + 1e-5
        dist = self.distribution(mu.view(1, ).data, sigma.view(1, ).data)
        value = self.value(x)
        return dist, value
```

In [3]:

```
class PolicyNetwork():
    def __init__(self, n_state, n_action,n_hidden1=200,n_hidden2=200, lr=0.001):
        self.model = ActorCriticModel(n_state, n_action, n_hidden1,n_hidden2)
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
    def predict(self, s):
        """ Compute the output using the continuous Actor Critic model
        @param s: input state
        @return: Gaussian distribution, state_value """
        self.model.training = True
        return self.model(torch.Tensor(s))

    def update(self, returns, log_probs, state_values):
        """Update the weights of the Actor Critic network given the training sample
        @param returns: return (cumulative rewards) for each step in an episode
        @param log_probs: log probability for each step
        @param state_values: state-value for each step"""
        loss = 0
        for log_prob, value, Gt in zip(log_probs, state_values, returns):
            advantage = Gt - value.item()
            policy_loss = -log_prob * advantage
            value_loss = F.smooth_l1_loss(value, Gt)
            loss += policy_loss + value_loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def get_action(self, s):
        """Estimate the policy and sample an action,compute its log probability
        @param s: input state
        @return: the selected action, log probability,predicted state-value """
        dist, state_value = self.predict(s)
        action = dist.sample().numpy()
        log_prob = dist.log_prob(action[0])
        return action, log_prob,state_value
```

In below cell I have used sklearn libray to preprocess the state of env.

I have taken 10,000 random sample from env. obervation space and fitted with sklearn.preprocessing.StandardScaler() function to normalize the env state so mean is zero and std is 1

so in training loop I will call scale_state function which will normalize the state of env.

In [4]:

```
import sklearn.preprocessing
import numpy as np
state_space_samples = np.array([env.observation_space.sample() for x in range(10000
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(state_space_samples)

def scale_state(state):
    scaled = scaler.transform([state])
    return scaled[0]
```

Simulation Model Class

To get simulation model of the mountain car environment, I have used two type of Regression method

- polynomial regression (d=2) (parametric regrssion) (weight_updation function in simulation model class)
- Gaussian process regression (non-parametric regression) (Gaussian_Process_Regression in simulation model class)

At the time of training of the policy_network, I have included only GP simulation model in this code file because from polynomial regression simulation model goal is not achived.

Functions--

weight updation (polynomial regression (d=2))

```
fitting 2nd degree polynomial for position and velocity

input arguments-

@ X: Nx3 matrix where N is no. of instances
```

$$X^i = [x^i, v^i, a^i]$$

```
@ y: Nx2 matrix   (sucessor state)
```

$$y^i = [x^i, v^i]$$

```
PolynomialFeatures(2).fit_transform convert X matrix into X_design matrix(
 size=Nx10)
```

$$X^i_{design} = [1, x^i, v^i, a^i, x^i.v^i, x^i.a^i, v^i.a^i, (x^i)^2, (v^i)^2, (a^i)^2]$$

$$y^{pred} = X_{design}.w$$

```
where w is parameter matrix of size 10x2

cost function--
```

$$J = \frac{(y^{pred} - y)^2}{2.N}$$

```
parameter theta updation (SGD) --
```

$$w^{new} = w^{old} - \alpha.\nabla J(\theta)$$

```
return--
@ tuple of (position error, velocity error)
@ y_pred
```

kernel

```
GP squared exponential kernel

input arguments-

@ a, b vector or matrix like
```

$$k_{ij} = exp(-\frac{(a_i - b_j)^2}{2.l^2})$$

```
return-

@ K -covariance matrix
```

Gaussian_Process_Regression

```
inputs-
@ x_train,y_train,x_test
@ training -- takes boolean value  True -for training the model and false-
 to get predicted mean and variance at test point

return--
if training is false
@ mu -expected mean value at test points
@ var - variance at test points
```

GP-Regression

we observe a training set $D = \{(x_i, f_i), i = 1 : N\}$ where $f_i = f(x_i)$
Given a test set X_* of size N_*Xd,we want to predict the function f_*

$$\begin{pmatrix} f \\ f_* \end{pmatrix} = N\left(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix} \begin{bmatrix} K & K_* \\ K_*^T & K_{**} \end{bmatrix}\right)$$

where:
K=K(X,X) is NXN

$K_* = K(X, X_*)$ is NXN_*, and

$K_{**} = K(X_*, X_*) is N_*XN_*$

$$K(x, x') = \sigma^2 exp(-\frac{(x - x')^2}{2l^2})$$

$$P(f_*|X_*, X, f) = N(f_*|\mu_*, \Sigma_*)$$

$$\mu_* = \mu(X_*) + K_*^T K^{-1}(f - \mu(X))$$

$$\Sigma_* = K_{**} - K_*^T K^{-1} K_*$$

Figure 3 : GP-Regression

run_simulation

```
we simulate, simulation model to train the policy network (optimize the pol
icy)

inputs--

@ mode- if mode='poly_reg' then it simulate polynomial regression model

        if mode='GP' then it simulate GP regression model

@ estimator- Policy_network class object

@ x_train,y_train

return--

@ reward_sequence,log_probs,state_values
```

In [5]:

```
class Simulation_Model:

    def __init__(self,w,K,L,s):
        self.w=w
        self.K=K      # covariance matrix
        self.L=L      # cholesky decomposition of K=LL^T
        self.s=s      # error variance for regularization

    def get_weight(self):
        return self.w

    def set_weight(self,weight):
        self.w=weight

    def weight_updation(self,X,y):
        """ fitting both position and velocity using polynomial regression (degree=
        count=0
        alpha=0.2
        N=X.shape[0]
        poly = PolynomialFeatures(2)
        X_design=poly.fit_transform(X)
        Error_pos,Error_vel=[],[]
        while count<1000:
            Error=X_design.dot(self.w)-y
            # weight updation
            self.w=self.w-(alpha/N)*(X_design.T).dot(Error)
            error_pos=(Error[:,[0]].T).dot(Error[:,[0]])/N
            error_vel=(Error[:,[1]].T).dot(Error[:,[1]])/N
            Error_pos.append(error_pos)
            Error_vel.append(error_vel)
            count+=1
        return (np.array(Error_pos),np.array(Error_vel)),X_design.dot(self.w)-y

    def kernel(self,a, b):
        """ GP squared exponential kernel """
        kernelParameter = 0.1
        sqdist = np.sum(a**2,1).reshape(-1,1) + np.sum(b**2,1) - 2*np.dot(a, b.T)
        return np.exp(-.5 * (1/kernelParameter) * sqdist)

    def Gaussian_Process_Regression(self,x_train,y_train,x_test,training=None):
        ''' fitting both position and velocity using GP-regression'''
        if training:
            self.K=self.kernel(x_train,x_train)
            self.L=np.linalg.cholesky(self.K + self.s*np.eye(x_train.shape[0]))
        else:
            # compute the mean at test points.
            Lk = np.linalg.solve(self.L, self.kernel(x_train, x_test))
            mu = np.dot(Lk.T, np.linalg.solve(self.L, y_train))

            # compute the variance at test points.
            K_ = self.kernel(x_test, x_test)
            s2 = np.diag(K_) - np.sum(Lk**2, axis=0)
            var = np.sqrt(s2)
            return mu,var

    def run_simulation(self,x_train,y_train,estimator,mode):
        #print('run simulation')
        reward_sequence=[]
        action_sequence=[]
```

```
        state_sequence=[]
        state_values=[]
        log_probs=[]
        if mode=='poly_reg':
            for i in range(1000):
                if i==0:
                    # initial state x=[-0.6,-0.4] and v=0
                    x_t=(-0.4+0.6)*np.random.random_sample()-0.6
                    v_t=0.0
                state_sequence.append([x_t,v_t])
                action_sequence.append(a_t)
                s_t=scale_state([x_t,v_t])
                a_t, log_prob, state_value = estimator.get_action(s_t)
                a_t=action.clip(env.action_space.low[0],env.action_space.high[0])
                if x_t<0.5:
                    reward=-a_t**2
                else:
                    reward=100-a_t**2
                reward_sequence.append(reward)
                poly = PolynomialFeatures(2)
                x_test=np.array([[x_t,v_t,a_t]])
                X_design=poly.fit_transform(x_test)
                #a_t=agent.pi_sa(x_t,v_t,a_t)[0]
                x_t,v_t=X_design.dot(self.w)[0]
                log_probs.append(log_prob)
                state_values.append(state_value)
                if x_t>=0.5:
                    break
            return reward_sequence,log_probs,state_values

        elif mode=='GP':
            #print('i was here')
            for i in range(1000):
                if i==0:
                    # initial state x=[-0.6,-0.4] and v=0
                    x_t=(-0.4+0.6)*np.random.random_sample()-0.6
                    v_t=0.0
                state_sequence.append([x_t,v_t])
                s_t=scale_state([x_t,v_t])
                a_t, log_prob, state_value = estimator.get_action(s_t)
                a_t=action.clip(env.action_space.low[0],env.action_space.high[0])
                if x_t<0.5:
                    reward=-a_t**2
                else:
                    reward=100-a_t**2
                reward_sequence.append(reward)
                #print(f'{i}: a_t:{a_t} log_prob:{log_prob} V:{state_value} R:{rewa
                action_sequence.append(a_t)
                x_test=np.array([[x_t,v_t,a_t[0]]])
                #print(f'x_test:{x_test}')
                mu,var=self.Gaussian_Process_Regression(x_train,y_train,x_test,Fals
                x_t,v_t=mu[0,0],mu[0,1]
                #print(x_t,v_t)
                log_probs.append(log_prob)
                state_values.append(state_value)
                if x_t>=0.5:
                    break
            return reward_sequence,log_probs,state_values
```

In [6]:

```
w_initial=np.zeros((10,2))
np.random.seed(42)
mountain_car=Simulation_Model(w_initial,None,None,1e-4)
```

In [7]:

```
n_state = env.observation_space.shape[0]
n_action = 1
n_hidden1 = 200
n_hidden2=200
lr = 0.0003
policy_net = PolicyNetwork(n_state, n_action, n_hidden1,n_hidden2, lr)
```

In the below cell I have loaded trained parameters of the model so we don't have to train model every time.

In [8]:

```
policy_net.model.load_state_dict(torch.load('policy_model_parameter'))
```

Out[8]:

```
<All keys matched successfully>
```

for model learning collecting data points

episode=1

horizon=1000

action are randomly taken

to train the simulated model of the system, I have used only 1 episode because in this system within a episode by choosing random action we can explore most of the system states.

In [9]:

```
for i_episode in range(1):
    observation = env.reset()
    action_sequence,state_sequence=[],[observation]
    for t in range(1000):
        env.render()
        action = env.action_space.sample()
        action_sequence.append(action)
        observation, reward, done, info = env.step(action)
        state_sequence.append(observation)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

```
Episode finished after 999 timesteps
```

coverting data into in array form

In [10]:

```
previous_state=np.array(state_sequence[0:-1])
sucessor_state=np.array(state_sequence[1:])
action_sequence=np.array(action_sequence)
print(previous_state.shape,sucessor_state.shape,action_sequence.shape)

X=np.hstack((previous_state,action_sequence))
y=sucessor_state
print(X.shape,y.shape)
print('\n \n ')
print('X:')
print(X[0:5])
print('\n \n ')
print('y:')
print(y[0:5])
```

```
(999, 2) (999, 2) (999, 1)
(999, 3) (999, 2)



X:
[[-0.42535058  0.          -0.20847303]
 [-0.42638953 -0.00103895 -0.15474942]
 [-0.42837938 -0.00198985  0.55287111]
 [-0.4302444  -0.00186502  0.69936097]
 [-0.43175142 -0.00150702  0.87920189]]



y:
[[-0.42638953 -0.00103895]
 [-0.42837938 -0.00198985]
 [-0.4302444  -0.00186502]
 [-0.43175142 -0.00150702]
 [-0.4326198  -0.00086839]]
```

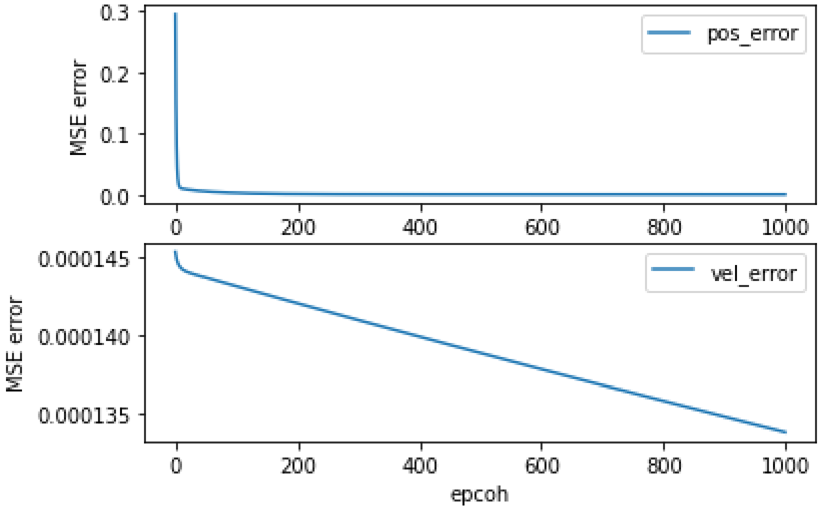
learning polynomial regression model

In [14]:

```
J,y_pred=mountain_car.weight_updation(X,y)
```

In [19]:

```
plt.subplot(2,1,1)
plt.plot(J[0].flatten(),label='pos_error')
plt.legend()
plt.xlabel('epcoh')
plt.ylabel('MSE error')
plt.subplot(2,1,2)
plt.plot(J[1].flatten(),label='vel_error')
plt.legend()
plt.xlabel('epcoh')
plt.ylabel('MSE error')
plt.show()
print(f"final position MSE error:{J[0][-1]}")
print(f"final position MSE error:{J[1][-1]}")
```




```
final position MSE error:[[0.00031054]]
final position MSE error:[[0.00013379]]
```

learning GP regression model

splitted dataset into training and testing data

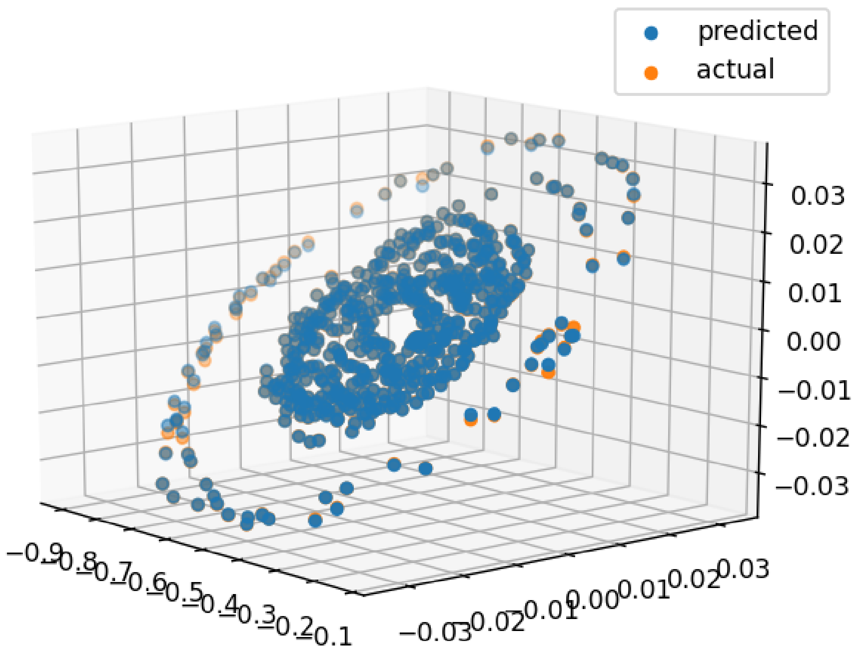
In [21]:

```
x_train,x_test,y_train,y_test=train_test_split(X,y,test_size=0.6,random_state=42)
x_train.shape,x_test.shape

mountain_car.Gaussian_Process_Regression(x_train,None,None,True)
mu,var=mountain_car.Gaussian_Process_Regression(x_train,y_train,x_test,False)
error=y_test-mu
error_pos=(error[:,[0]].T).dot(error[:,[0]])/error.shape[0]
error_vel=(error[:,[1]].T).dot(error[:,[1]])/error.shape[0]

print(f"final position MSE error:{error_pos}")
print(f"final position MSE error:{error_vel}")
```

```
final position MSE error:[[1.19301722e-05]]
final position MSE error:[[5.39717139e-08]]
```

In [22]:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

%matplotlib notebook
fig=plt.figure()
ax=fig.add_subplot(111,projection='3d')
ax.scatter(x_test[:,0],x_test[:,1],mu[:,1],label='predicted')
ax.scatter(x_test[:,0],x_test[:,1],y_test[:,1],label='actual')
plt.legend()
plt.show()
```



In [23]:

```
gamma=0.99
n_episode = 20
```

Training the policy network from GP-simulation model

I have already loaded trained parameters in the model so no need to run below cell.

In [24]:

```
# for episode in range(n_episode):
#     rewards,log_probs,state_values=mountain_car.run_simulation(x_train,y_train,es
#     returns = []
#     Gt = 0
#     pw = 0
#     for reward in rewards[::-1]:
#         Gt += gamma ** pw * reward
#         pw += 1
#         returns.append(Gt)
#     returns = returns[::-1]
#     returns = torch.tensor(returns)
#     returns = (returns - returns.mean()) / (returns.std() + 1e-9)
#     policy_net.update(returns, log_probs, state_values)
#     print(f'episode:{episode}')
```

Now applying the optimized policy on the environment

In [26]:

```
n_episode=20
total_reward_episode=[0]*n_episode
for episode in range(n_episode):
    state = env.reset()
    count=0
    while True:
        count+=1
        #print(count)
        env.render()
        state = scale_state(state)
        action, log_prob, state_value = policy_net.get_action(state)
        action = action.clip(env.action_space.low[0],env.action_space.high[0])
        next_state, reward, is_done, _ = env.step(action)
        total_reward_episode[episode] += reward
        if is_done:
            print(f'episode:{episode} no. of steps:{count} total_reward:{total_rewa
            break
        state=next_state
```

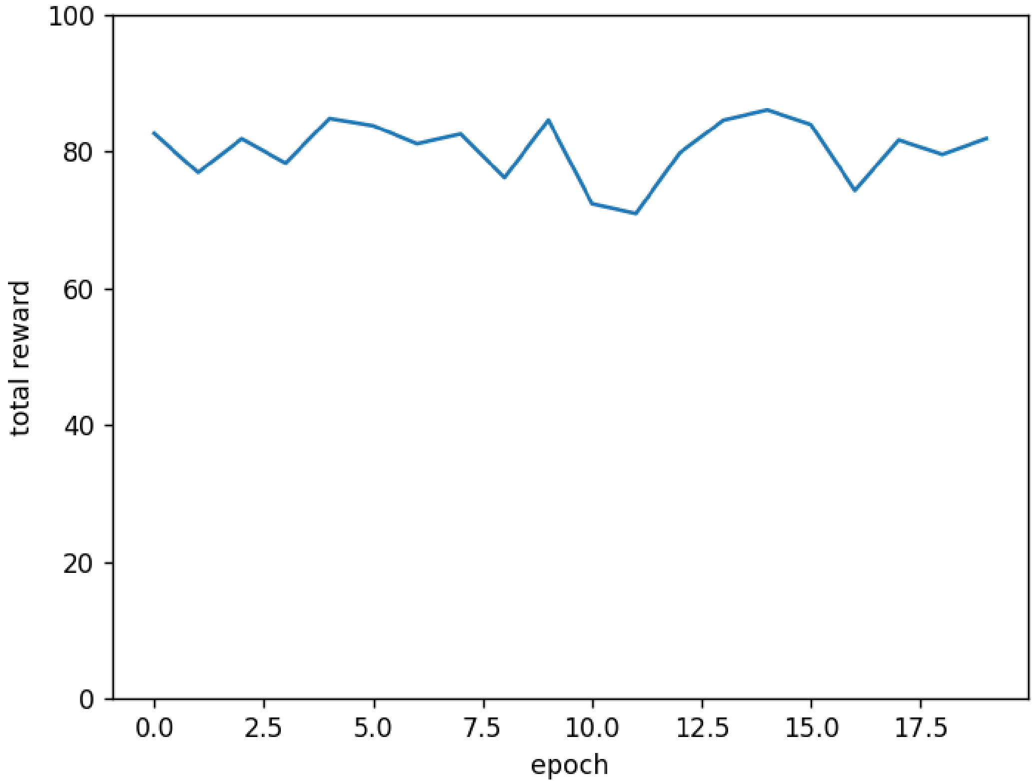
```
episode:0 no. of steps:450 total_reward:82.64970054431885
episode:1 no. of steps:644 total_reward:76.95082346556279
episode:2 no. of steps:465 total_reward:81.8763185356372
episode:3 no. of steps:558 total_reward:78.31845189494832
episode:4 no. of steps:382 total_reward:84.85109157632952
episode:5 no. of steps:441 total_reward:83.79773338978835
episode:6 no. of steps:458 total_reward:81.15541401395966
episode:7 no. of steps:440 total_reward:82.58436741701124
episode:8 no. of steps:632 total_reward:76.19145348504483
episode:9 no. of steps:420 total_reward:84.63487952183114
episode:10 no. of steps:727 total_reward:72.35290471261479
episode:11 no. of steps:802 total_reward:70.9353718820916
episode:12 no. of steps:501 total_reward:79.82817680731779
episode:13 no. of steps:395 total_reward:84.60360546811216
episode:14 no. of steps:380 total_reward:86.09554181609226
episode:15 no. of steps:429 total_reward:83.94839009969758
episode:16 no. of steps:654 total_reward:74.33744381211282
episode:17 no. of steps:482 total_reward:81.69122956283559
episode:18 no. of steps:562 total_reward:79.59758322820198
episode:19 no. of steps:445 total_reward:81.89989367926601
```

In [27]:

```
# torch.save(policy_net.model.state_dict(),'policy_model_parameter')
```

In [29]:

```
plt.plot(total_reward_episode)
plt.ylabel('total reward')
plt.xlabel('epoch')
plt.ylim(0,100)
plt.show()
```



In [ ]:

```
```