In [205]:
```
import gym
import pybulletgym
import pybulletgym.envs
import numpy as np
import math
import matplotlib.pyplot as plt
from numpy.linalg import pinv
import time
```

In [212]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
from torch.distributions import Categorical
```

In [207]:
```
env = gym.make("CartPole-v1")
env.reset()
```

Out[207]:
```
array([-0.03294539,  0.04493735,  0.00958779,  0.01345614])
```

In [217]:
```python
learning_rate = 0.01
gamma = 0.99

#4 states, 2 actions
num_states = env.observation_space.shape[0]
num_actions = env.action_space.n

class PolicyNetwork(nn.Module):
    def __init__(self):
        super(PolicyNetwork, self).__init__()
        self.state_num = num_states
        self.action_num = num_actions

        self.l1 = nn.Linear(self.state_num, 128, bias=False)
        self.l2 = nn.Linear(128, self.action_num, bias=False)

        self.gamma = gamma

        # Episode policy and reward history
        self.policy_episode = Variable(torch.Tensor())
        self.reward_episode = []   #a hash containing the reward of step
        # Overall return history
        self.return_history = []
        self.return_reward = []
        self.return_history_stepzero = []
        self.episodesPeriter = []




    def forward(self, x):
        model = torch.nn.Sequential(
                                    self.l1,
                                    nn.Dropout(p=0.6),
                                    #nn.ReLU(),
                                    nn.Tanh(),
                                    self.l2,
                                    nn.Softmax(dim=-1)
                                    )
        return model(x)
```

In [218]:
```
class CartPole ():



    def __init__(self, env, policy, part, totsteps, iterationsNo, learni
        self.env = env
        self.policy = policy
        self.part = part
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.optimizer = optim.Adam(self.policy.parameters(), lr=learnin
        self.iterationsNo = iterationsNo
        self.totsteps = totsteps

        self.iterations = []

        self.polHist_allepisodes = Variable(torch.Tensor())
        self.rewHist_allepisodes = Variable(torch.Tensor())
        self.rewHist_allepisodes_mod = Variable(torch.Tensor())

    def select_action(self, state):
        #Select an action (0 or 1) by running policy model and choosing
        state = torch.from_numpy(state).type(torch.FloatTensor)
        act_prob = self.policy(state)
        c = Categorical(act_prob)
        action = c.sample()

        if len(self.policy.policy_episode) > 0:
            self.policy.policy_episode = torch.cat([self.policy.policy_e
        else:
            self.policy.policy_episode = (c.log_prob(action).reshape(1)
        return action

    def rewardFunction(self, polHistory, rewHistory, part = 1):

        Rewards_tot = 0
        rewards = []
        for Reward in rewHistory[::-1]:
            Rewards_tot = Reward + self.policy.gamma * Rewards_tot
            rewards.insert(0, Rewards_tot)

        if(self.part == 1):
            #gainPerStep = rewards[0]
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            reward = rewards[0]*torch.sum(polHistory)



        if(self.part == 2):
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            rewards = torch.FloatTensor(rewards)
            reward = torch.sum(torch.mul(polHistory, rewards))

        if(self.part == 3):
```

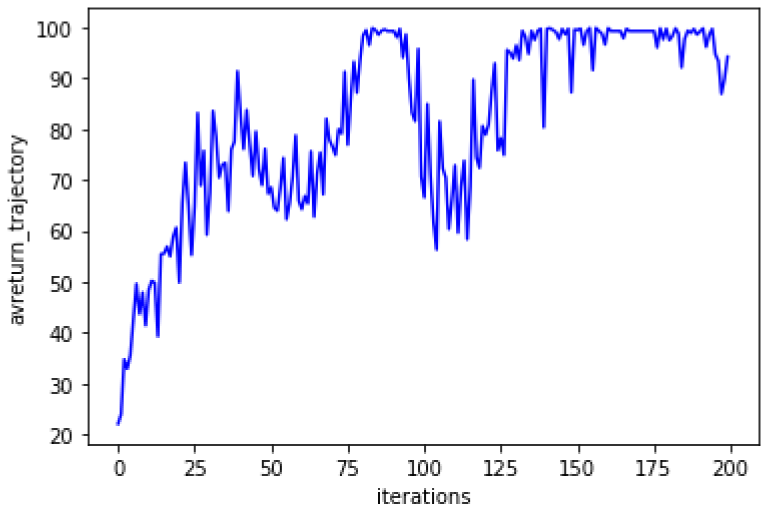
```
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            rewards = torch.FloatTensor(rewards)
            #rewards = (rewards - rewards.mean())/ (rewards.std())
            #reward = torch.sum(torch.mul(polHistory, rewards))

            #for i in range(0,len(polHistory)):
            #print(len(self.polHist_allepisodes))
            #print(len(self.rewHist_allepisodes))

            #if self.polHist_allepisodes.size(0) > 0:
            if (len(self.polHist_allepisodes) > 0):
                #print(self.polHist_allepisodes.shape)
                #print(type(polHistory))
                self.polHist_allepisodes = torch.cat([self.polHist_alle
                self.rewHist_allepisodes = torch.cat([self.rewHist_alle
            else:
                self.polHist_allepisodes = polHistory
                self.rewHist_allepisodes = rewards
            #rewards = (rewards - rewards.mean())/ (rewards.std())
            #reward = torch.sum(torch.mul(polHistory, rewards))

            #self.rewHist_allepisodes = (self.rewHist_allepisodes - sel
            self.rewHist_allepisodes_mod = (self.rewHist_allepisodes - s
            reward = torch.sum(torch.mul(self.polHist_allepisodes, self
            #print(reward)


        return reward, gainPerStep, gainZeroStep


    def update_policy(self, reward, retTraj_tot, retTraj_tot_stepzero, e
        #print("in update")
        # Update network weights
        self.optimizer.zero_grad()
        #print(reward)
        reward.backward()
        self.optimizer.step()
        self.policy.return_history.append(retTraj_tot)
        #self.policy.return_reward.append(reward)
        self.policy.return_history_stepzero.append(retTraj_tot_stepzero
        self.policy.episodesPeriter.append(episodes_iter)

    def reinforceAlgo(self):
    #running_reward = 10

        for iter in range(self.iterationsNo):
            self.polHist_allepisodes = Variable(torch.Tensor())
            self.rewHist_allepisodes = Variable(torch.Tensor())
            #print("iteration no",iter)
            steps = 0
            state = env.reset() # Reset environment, starting state rec
            done = False
            episodes = 0;
            rewardFunc = Variable(torch.FloatTensor([0]))
            rewardEpisode = Variable(torch.FloatTensor())
            retTraj_tot = 0
```

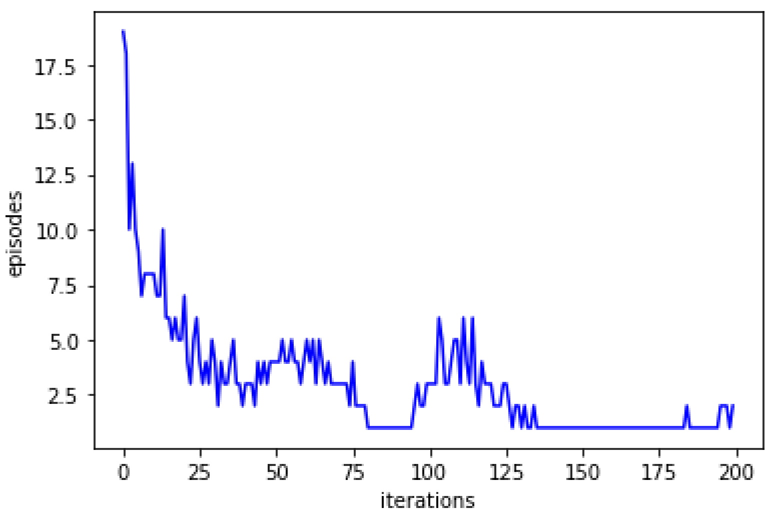
```
            retStepzero_tot = 0
            #optimizer.zero_grad()
            while(steps < self.totsteps):
                steps += 1;
                action = self.select_action(state)
                # Step through environment using chosen action
                state, reward, done, _ = env.step(action.item())
                #env.render()

            # Save reward
                self.policy.reward_episode.append(reward)
                if (done == True):
                    rewardEpisode, retTraj, retStepzero = self.rewardFu
                    if(self.part != 3):
                        rewardFunc += rewardEpisode
                    retTraj_tot += retTraj
                    retStepzero_tot += retStepzero
                    #reset the environment again
                    self.policy.policy_episode = Variable(torch.Tensor(
                    self.policy.reward_episode = []
                    state = env.reset()
                    done = False
                    episodes += 1

            if(self.part == 3):
                 rewardFunc += rewardEpisode
            if(episodes > 0):
                self.update_policy(-1 * rewardFunc/episodes, retTraj_to
            else:
                self.update_policy(-1 * rewardFunc, retTraj_tot, retSte
            self.iterations.append(iter)
```

##############################1.1 - Batch size = 500, iterations = 200 ##

In [219]:
```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 1, totsteps = 500
cartPole.reinforceAlgo()
```

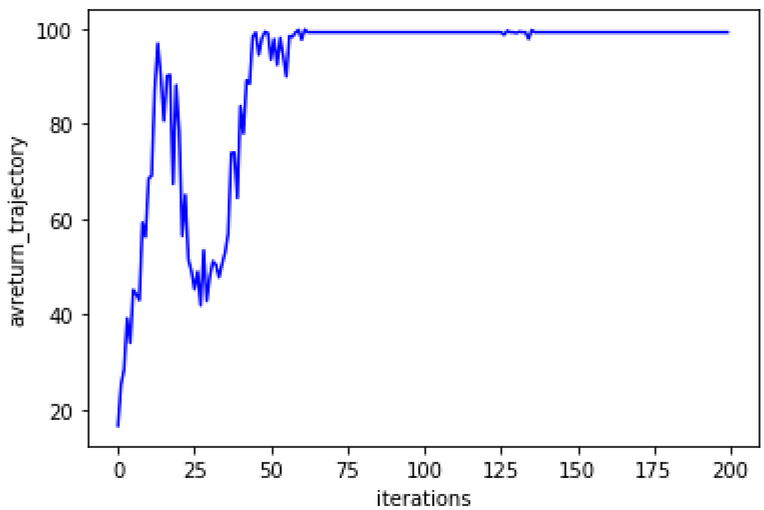
In [220]:
```
plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, co
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```

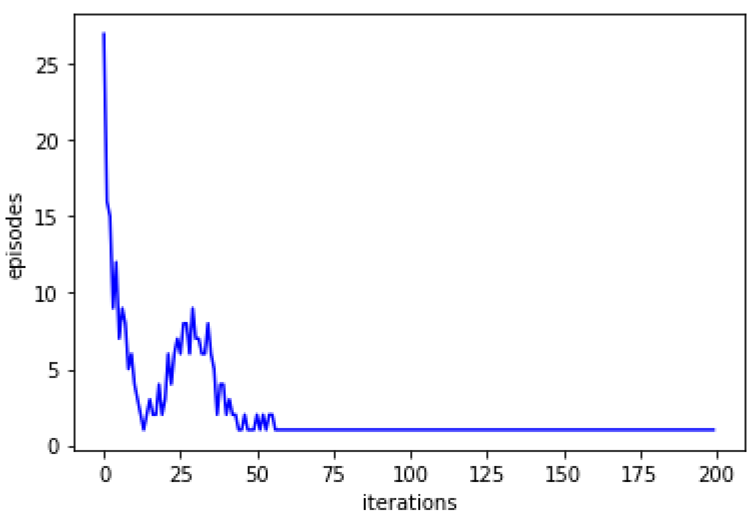




############################1.2 - Batch size = 500, iterations = 200
##

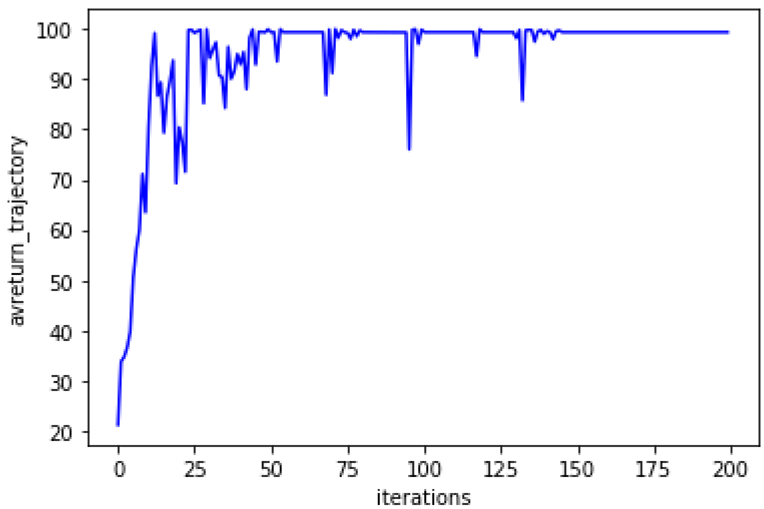
In [222]:
```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 2, totsteps = 500
cartPole.reinforceAlgo()



plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, co
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```

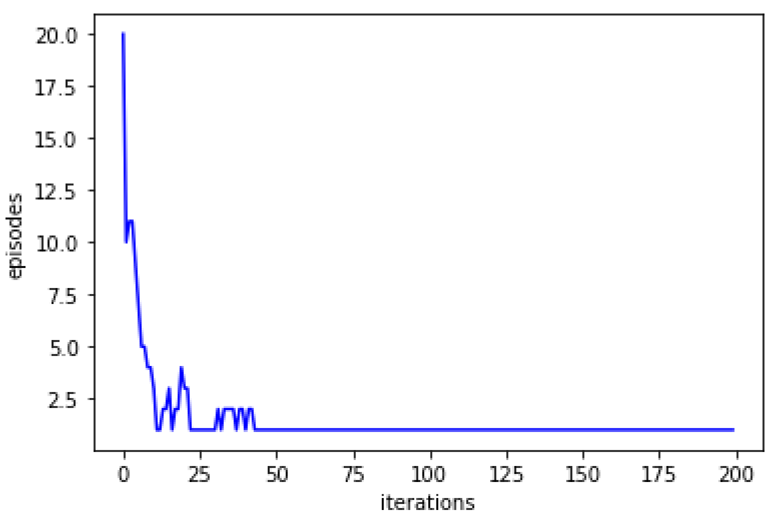





```
###########################1.3 - Batch size = 500, iterations = 200
########################################
```

In [224]:
```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 3, totsteps = 500
cartPole.reinforceAlgo()


plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, c
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```

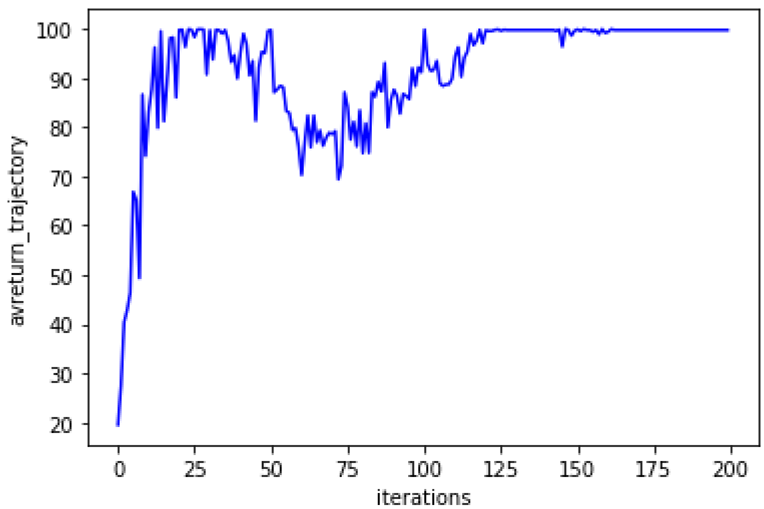


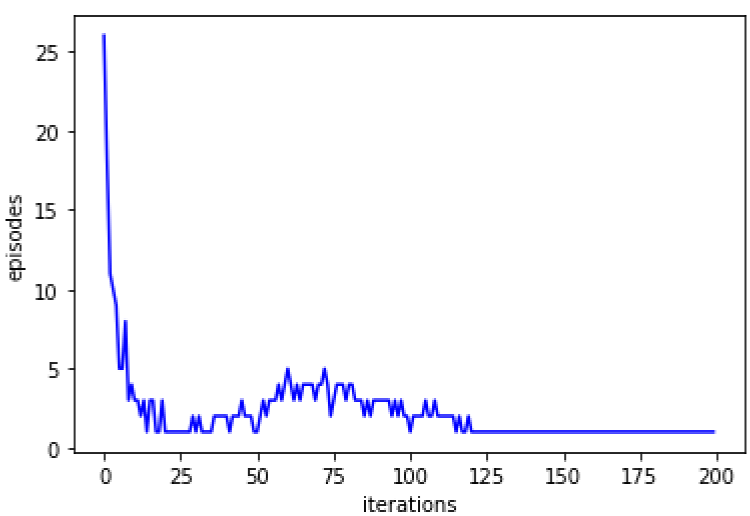


```
###########################1.3 - Batch size = 600, iterations = 200
########################################
```

In [225]:

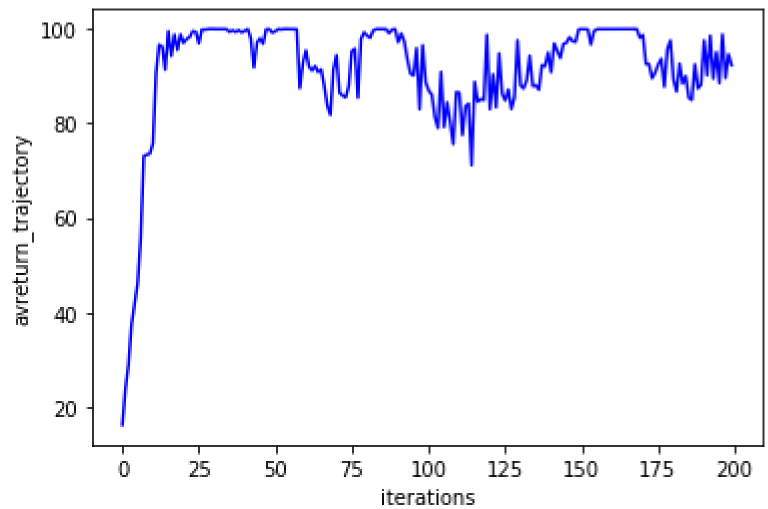
```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 3, totsteps = 600
cartPole.reinforceAlgo()



plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, c
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```

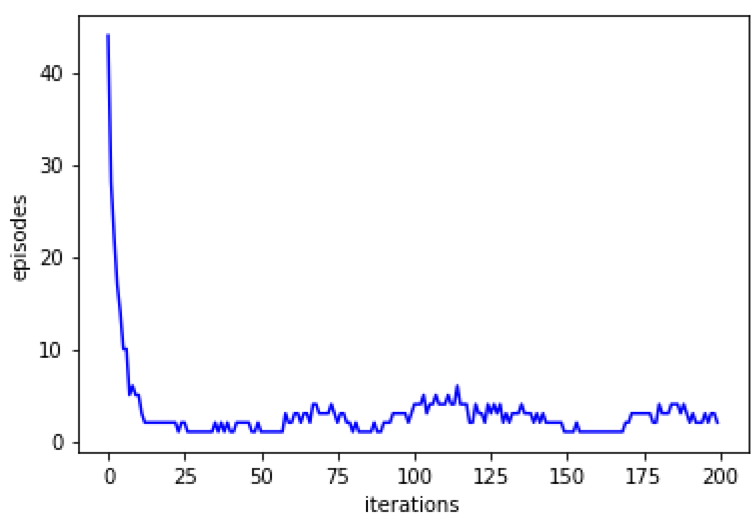




###########################1.3 - Batch size = 800, iterations = 200
##

In [228]:
```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 3, totsteps = 800
cartPole.reinforceAlgo()



plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, c
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```

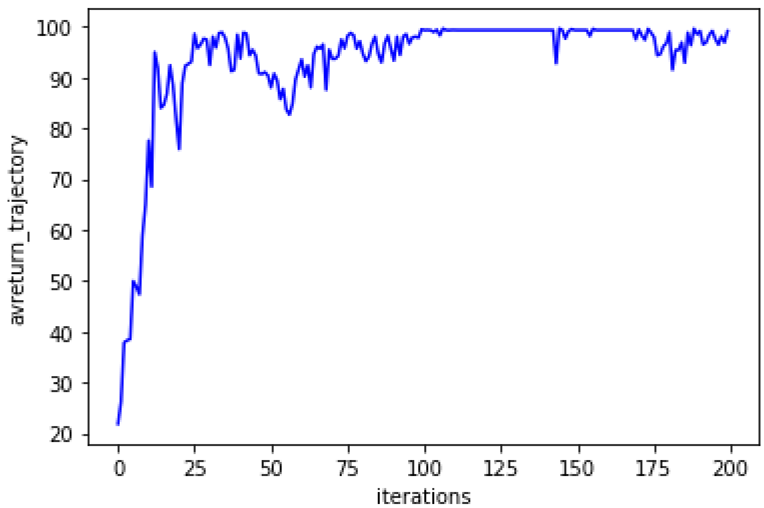


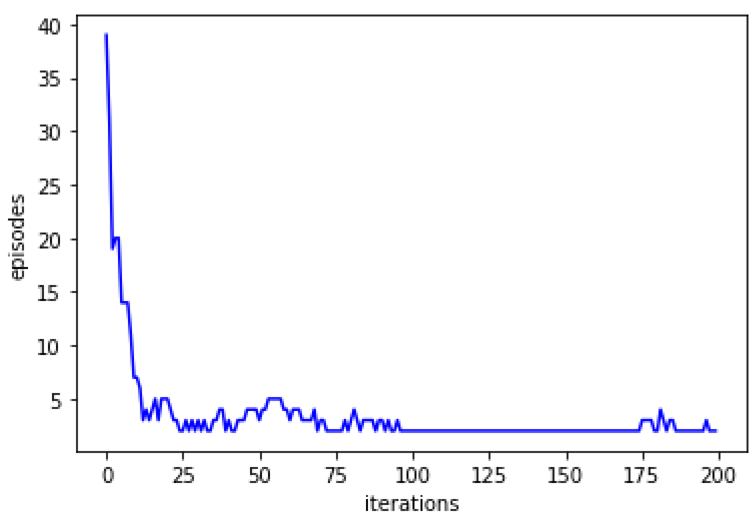



```
############################1.3 - Batch size = 1000, iterations = 200
########################################
```

In [230]:

```
policy = PolicyNetwork()
cartPole = CartPole(env= env, policy = policy, part = 3, totsteps = 100(
cartPole.reinforceAlgo()



plt.plot(cartPole.iterations,cartPole.policy.return_history_stepzero, c(
plt.xlabel("iterations")
plt.ylabel("avreturn_trajectory")
plt.show()


plt.plot(cartPole.iterations,cartPole.policy.episodesPeriter, color='b'
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```





Increasing the batch size ideally should give better training results, since we would be exploring more no of trajectories in a given iteration to determine the trajectory with the best result. However, in this case the average discounted reward per episode remains 100 in all the above four cases, meaning the training has achieved the result with batch size 500 only.

If we look at rewards obtained at each step the average max reward in the case of 500 steps would be 500, while in the case of 600,800 and 1000 steps it would be 300,400 and 500 respectively since the min no of episodes would be more than 1. Hence that cannot be given as the metric to determine the performance of training. Hence the average discounted measure would tell us the training performance, which is equivalent in all the above cases.

In [1]:
```
import gym
import pybulletgym
import pybulletgym.envs
import numpy as np
import math
import matplotlib.pyplot as plt
from numpy.linalg import pinv
import time
```

In [2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
from torch.distributions import Categorical
from torch.distributions import MultivariateNormal
```

In [3]:
```
env = gym.make("modified_gym_env:ReacherPyBulletEnv-v1", rand_init = Fa
#env.render()
env.reset()
```

```
current_dir=/home/apurba/.virtualenvs/276c_assgn/lib/python3.6/site-pa
ckages/pybullet_envs/bullet
options=
```

Out[3]:
```
array([ 0.3928371 ,  0.3928371 , -0.68091764,  0.26561381,  0.5
,
        0.        ,  0.08333333,  0.        ])
```

In [4]:
```
learning_rate = 0.01
gamma = 0.90

num_states = env.observation_space.shape[0]
print(num_states)
```

```
9
```

In [5]:
```
class PolicyNetwork(nn.Module):
    def __init__(self):
        super(PolicyNetwork, self).__init__()
        self.state_num = 8
        self.action_num = 2

        #self.l1 = nn.Linear(self.state_num, 64, bias=False)
        #self.l2 = nn.Linear(64, 64, bias=False)
        #self.l3 = nn.Linear(64, self.action_num, bias=False)
        self.l1 = nn.Linear(self.state_num, 128, bias=False)
        self.l2 = nn.Linear(128, self.action_num, bias=False)
        self.gamma = gamma

        # Episode policy and reward history
        self.policy_episode = Variable(torch.Tensor())
        self.reward_episode = []
        # Overall return history
        self.return_history = []
        self.return_reward = []
        self.return_history_stepzero = []

        self.policy_episode_eval = Variable(torch.Tensor())
        self.reward_episode_eval = []


        self.episodesPeriter = []
        self.sigmax = torch.nn.Parameter(torch.FloatTensor([0.1]))
        self.sigmay = torch.nn.Parameter(torch.FloatTensor([0.1]))
        #self.sigma = torch.nn.Parameter(torch.tensor([[0.1, 0], [0, 0..




    def forward(self, x):

        model = torch.nn.Sequential(
                                        self.l1,
                                        nn.Dropout(p=0.6),
                                        #nn.ReLU(),
                                        nn.Tanh(),
                                        self.l2,
                                        #nn.Dropout(p=0.6),
                                        #nn.Tanh(),
                                        #self.l3,
                                        nn.Tanh()
                                        )
        return model(x)
```

In [6]:
```
class modtwoLink ():



    def __init__(self, env, policy, part, totsteps, iterationsNo, learn
        self.env = env
        self.policy = policy
        self.part = part
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.optimizer = optim.Adam(self.policy.parameters(), lr=learni
        self.iterationsNo = iterationsNo
        self.totsteps = totsteps

        self.iterations = []

        self.polHist_allepisodes = Variable(torch.Tensor())
        self.rewHist_allepisodes = Variable(torch.Tensor())
        self.rewHist_allepisodes_mod = Variable(torch.Tensor())

    def select_action(self, state):


        #Select an action (0 or 1) by running policy model and choosing
        state = torch.from_numpy(state).type(torch.FloatTensor)
        probs = self.policy(state)#this will give the mean of x and y
        mux = policy.sigmax.reshape(1)
        muy = policy.sigmay.reshape(1)
        if(mux < 0.001):
            mux = mux + 0.001
        if(muy < 0.001):
            muy = muy + 0.001
        covariance = torch.cat([mux, muy])
        #print(covariance)
        covariance_tensor = torch.FloatTensor(covariance)
        c = MultivariateNormal(probs, torch.diag(torch.abs(covariance_te
        #print(self.policy.sigma)
        #c = MultivariateNormal(probs, self.policy.sigma)
        action = c.sample()

        if len(self.policy.policy_episode) > 0:
            self.policy.policy_episode = torch.cat([self.policy.policy_e
        else:
            self.policy.policy_episode = (c.log_prob(action).reshape(1)
        return action

    def rewardFunction(self, polHistory, rewHistory, part = 1):

        Rewards_tot = 0
        rewards = []
        for Reward in rewHistory[::-1]:
            Rewards_tot = Reward + self.policy.gamma * Rewards_tot
            rewards.insert(0, Rewards_tot)

        if(self.part == 1):
            #gainPerStep = rewards[0]
```

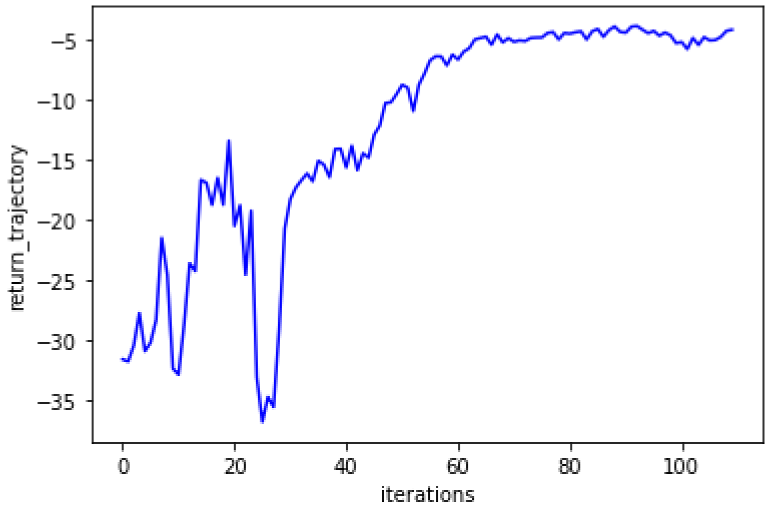
```
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            reward = rewards[0]*torch.sum(polHistory)



        if(self.part == 2):
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            rewards = torch.FloatTensor(rewards)
            reward = torch.sum(torch.mul(polHistory, rewards))

        if(self.part == 3):
            gainPerStep = sum(rewHistory)
            gainZeroStep = rewards[0]
            rewards = torch.FloatTensor(rewards)

            if (len(self.polHist_allepisodes) > 0):

                self.polHist_allepisodes = torch.cat([self.polHist_alle
                self.rewHist_allepisodes = torch.cat([self.rewHist_alle
            else:
                self.polHist_allepisodes = polHistory
                self.rewHist_allepisodes = rewards


            #self.rewHist_allepisodes = (self.rewHist_allepisodes - sel
            self.rewHist_allepisodes_mod = (self.rewHist_allepisodes - s
            reward = torch.sum(torch.mul(self.polHist_allepisodes, self
            #print(reward)


        return reward, gainPerStep, gainZeroStep


    def update_policy(self, reward, retTraj_tot, retTraj_tot_stepzero, e
        #print("in update")
        # Update network weights
        self.optimizer.zero_grad()
        #print(reward)
        reward.backward()
        self.optimizer.step()
        self.policy.return_history.append(retTraj_tot)
        #self.policy.return_reward.append(reward)
        self.policy.return_history_stepzero.append(retTraj_tot_stepzero
        self.policy.episodesPeriter.append(episodes_iter)

    def reinforceAlgo(self):
    #running_reward = 10

        for iter in range(self.iterationsNo):
            self.polHist_allepisodes = Variable(torch.Tensor())
            self.rewHist_allepisodes = Variable(torch.Tensor())
            #print("iteration no",iter)
            steps = 0
            state = env.reset() # Reset environment, starting state rec
            done = False
```

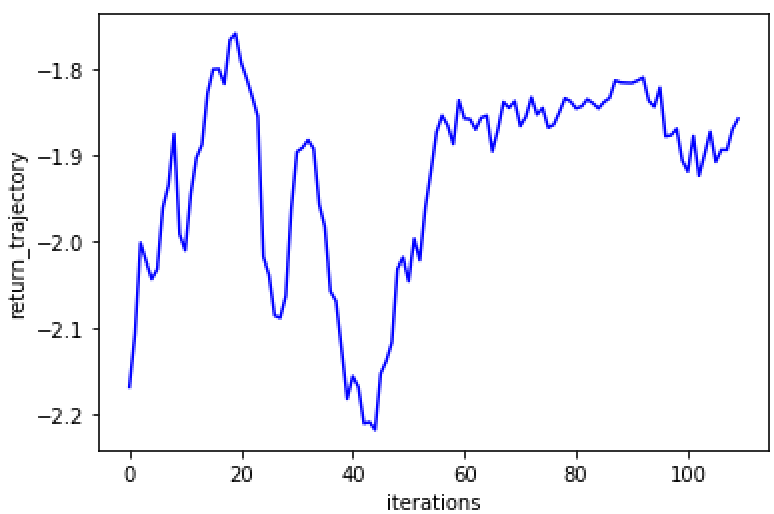
```
            episodes = 0;
            rewardFunc = Variable(torch.FloatTensor([0]))
            rewardEpisode = Variable(torch.FloatTensor())
            retTraj_tot = 0
            retStepzero_tot = 0
            #optimizer.zero_grad()
            while(steps < self.totsteps):
                steps += 1;
                action = self.select_action(state)
                # Step through environment using chosen action
                state, reward, done, _ = env.step(action)
                #env.render()

            # Save reward
                self.policy.reward_episode.append(reward)
                if (done == True):
                    rewardEpisode, retTraj, retStepzero = self.rewardFu
                    if(self.part != 3):
                        rewardFunc += rewardEpisode
                    retTraj_tot += retTraj
                    retStepzero_tot += retStepzero
                    #reset the environment again
                    self.policy.policy_episode = Variable(torch.Tensor(
                    self.policy.reward_episode = []
                    state = env.reset()
                    done = False
                    episodes += 1

            if(self.part == 3):
                 rewardFunc += rewardEpisode
            if(episodes > 0):
                self.update_policy(-1 * rewardFunc/episodes, retTraj_to
            else:
                self.update_policy(-1 * rewardFunc, retTraj_tot, retSte
            self.iterations.append(iter)
```

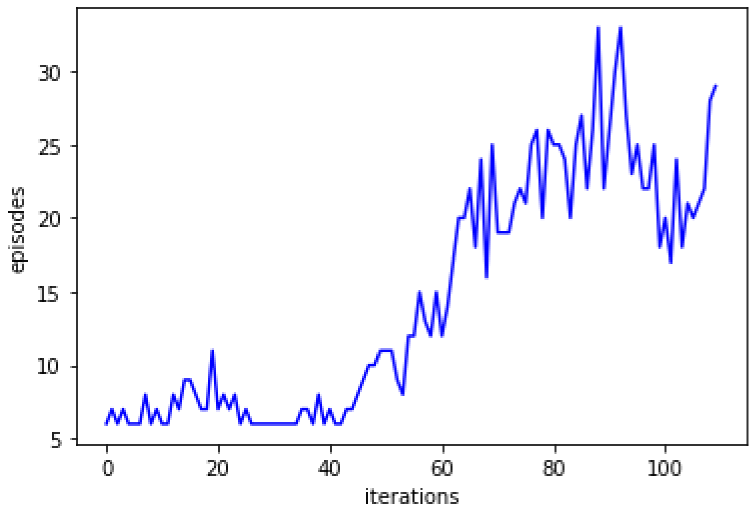
In [7]:
```
state = env.reset()
policy = PolicyNetwork()
twoLink = modtwoLink(env= env, policy = policy, part = 3, totsteps = 10
twoLink.reinforceAlgo()
```

The below graphs denote the average rewards, average discounted reward and the no of episodes varying with the iteration. The no of episodes and rewards should increase with no of iterations.

In [8]:
```
plt.plot(twoLink.iterations,twoLink.policy.return_history, color='b');
plt.xlabel("iterations")
plt.ylabel("return_trajectory")
plt.show()

plt.plot(twoLink.iterations,twoLink.policy.return_history_stepzero, col
plt.xlabel("iterations")
plt.ylabel("return_trajectory")
plt.show()


plt.plot(twoLink.iterations,twoLink.policy.episodesPeriter, color='b');
plt.xlabel("iterations")
plt.ylabel("episodes")
plt.show()
```







For evaluation, sampling the x, y values from the mean obtained from the network

In [9]:
```
def select_action_eval(state):
    #Select an action (0 or 1) by running policy model and choosing base
    state = torch.from_numpy(state).type(torch.FloatTensor)
    action = policy(state)#this will give the mean of x and y



    return action
```

In [10]:

```
env = gym.make("modified_gym_env:ReacherPyBulletEnv-v1", rand_init=Fals
steps = 0
env.render('human')
state = env.reset()
done = False
while (steps<300 and done == False):
        action = select_action_eval(state)
        action_np = action.detach().numpy()
        state, r, done, info = env.step(action_np)
        steps+=1
        env.render('human')
        time.sleep(0.1)
        print(done)

print(steps)
```

```
options=
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
```

```
False
False
False
False
False
False
False
False
True
49
```

In [ ]:
```
env.close()
```