ASEN 5264 - Decision Making under Uncertainty - Homework 5

Aritra Chakrabarty

Problem 1

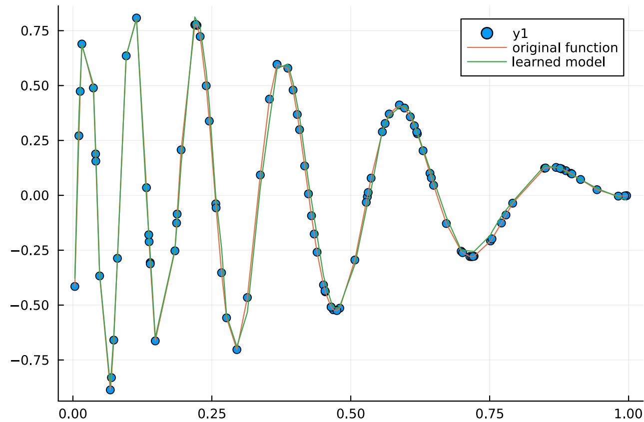
In [ ]:
```
using QuickPOMDPs: QuickPOMDP
using POMDPTools: Deterministic, Uniform, SparseCat, FunctionPolicy, RolloutSimulator
using Statistics: mean
import POMDPs

mammography = QuickPOMDP(
    states = [:healthy, :in_situ_cancer, :invasive_cancer, :death],
    actions = [:wait, :test, :treat],
    observations = [:pos, :neg],

    transition = function (s,a)
        if s == :healthy
            return SparseCat([:healthy, :in_situ_cancer],[0.98,0.02])
        elseif s == :in_situ_cancer
            if a == :treat
                return SparseCat([:in_situ_cancer, :healthy],[0.4,0.6])
            else #not treat
                return SparseCat([:in_situ_cancer, :invasive_cancer],[0.9,0.1])
            end
        elseif s == :invasive_cancer
            if a == :treat
                return SparseCat([:invasive_cancer, :healthy, :death], [0.6,0.2,0.2])
            else
                return SparseCat([:invasive_cancer, :death], [0.4,0.6])
            end
        else
            #terminal state, not sure if I should include this or not
            return Deterministic(:death)
        end
    end,

    observation = function (a, sp)
        if a == :test
            if sp == :healthy
                return SparseCat([:pos, :neg], [0.05, 0.95])
            elseif sp == :in_situ_cancer
                return SparseCat([:pos, :neg], [0.8, 0.2])
            else #invasive_cancer, don't want to leave room for modle doubt
                return Deterministic(:pos)
            end
        elseif a == :treat
            if sp in [:in_situ_cancer, :invasive_cancer]
                return Deterministic(:pos)
            else
                return Deterministic(:neg)
            end
        else
            return Deterministic(:neg)
        end
    end,

    reward = function (s,a)
        if s == :death
            return 0.0
        else
            if a == :wait
                return 1.0
            elseif a == :test
                return 0.8
            else #treat
                return 0.1
            end
        end
    end,

    initialstate = Deterministic(:healthy),

    discount = 0.99
)

policy = FunctionPolicy(o->:wait)
sim = RolloutSimulator(max_steps=1000)
mean(POMDPs.simulate(sim, mammography, policy) for _ in 1:10_000)
```

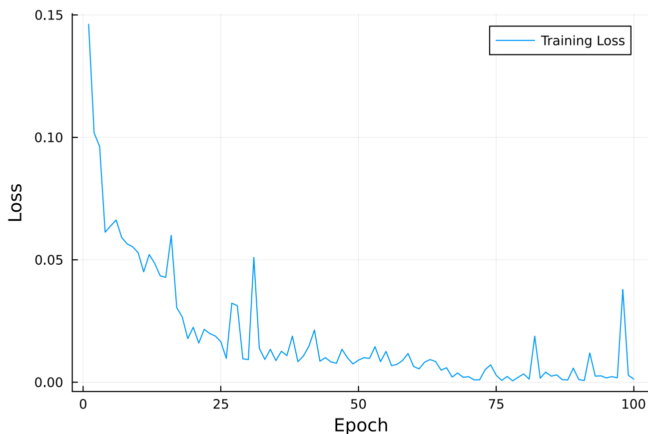
```
40.826162050467644
```

Problem 2

In [ ]:
```
using Plots
using Flux
using StaticArrays
using Random
using Statistics

f(x) = (1-x)*sin(20*log(0.2+x))
n = 100
dx = rand(Float32, n)
dy = convert.(Float32, (1 .- dx) .* sin.(20*log.(0.2 .+ dx)))


data = [(SVector(dx[i]), SVector(dy[i])) for i in 1:length(dx)]
m = Chain(Dense(1=>50, relu), Dense(50=>50, relu), Dense(50=>50, relu), Dense(50=>50, relu), Dense(50=>1))

loss(x,y) = Flux.Losses.mse(m(x),y)
#loss(x, y) = sum((m(x)-y).^2)

models = [deepcopy(m)]
losses = []
```

```julia
epochs = 100

for ep in 1:epochs
    Flux.train!(loss, Flux.params(m), repeat(data,50), Adam());
    push!(models, deepcopy(m))
    push!(losses, mean([loss(d...) for d in data]))
end

p = scatter(dx,dy)
plot!(p, sort(dx),x->f(x), label="original function")
plot!(p, sort(dx), first.(last(models).(SVector.(sort(dx)))), label="learned model")

display(p)

lp = plot(1:epochs, losses, label="Training Loss", xlabel="Epoch", ylabel="Loss")
display(lp)
```





Problem 3

Since a Deep Q Network was the suggested method for this problem to solve a MountainCar environment, I decided to use DQN and implement it in Julia as it would be easiest to get debugging support. The starter code also has suggested methods of interacting with the environment and the autograder, so an implementation in Julia seemed best.

I decided to use the algorithm from the original Deepmind paper that implemented the Deep Q Network: Human-level control through deep reinforcement learning. I had to however change some portions of the algorithm to fit our use case better where we can obtain the state directly. The algorithm I have implemented is shown below.

1. Initialise replay memory D to capacity N
2. Initialise $Q(s, a)$ with random weights θ
3. $\hat{Q}(s, a) \leftarrow Q(s, a)$
4. **For** episodes 1 to M:
   - a. Initialise sequence with s_1
   - b. **For** (timesteps 1 to T):
     - i. **IF** rand() < ϵ, a = random action **else** choose the best action from Q (the one that maximizes Q value)
     - ii. Execute action a, to obtain experience tuple $(s, a, r, s', terminal)$
     - iii. Set $s = s'$
     - iv. Store experience tuple $(s, a, r, s', terminal)$ in replay memory D
     - v. **IF** enough experience tuples in memory D

       A. Sample random minibatch of *experience tuples* from replay memory D

       B. **For** each sampled transition, calculate loss function $l(s, a, r, s') = (r + \gamma \max_{a'} \hat{Q}_\theta(s', a') - Q_\theta(s, a))^2$
     - vi. Minimize loss function using ADAM()
     - vii. After every C steps, $\hat{Q}(s, a) \leftarrow Q(s, a)$

I have also added some minor changes to the algorithm into the code below to both improve performance and show plots. I did some hyper-parameter tuning to get good exploration results and increase consistency. I run the code longer in part 3b, to get better rewards, but it is pretty much the same. I'm not evaluating the code in the notebook, so that line is commented out.

In [ ]:
```julia
using CommonRLInterface
using Flux
using CommonRLInterface.Wrappers: QuickWrapper
using DMUStudent.HW5
using Plots
using Statistics: mean
using DataStructures: CircularBuffer

function dqn(env, N, M, T, γ, ϵ_start, ϵ_end, ϵ_decay, C, batch_size)
    # Initialise replay memory D to capacity N
```

```julia
    # using a circular buffer because it automatically limits length
    buffer = CircularBuffer{Tuple}(N)

    # Initialise Q(s,a) with random weights θ
    #       Same network as starter
    # Q̂(s,a) ← Q(s,a)
    Q = Chain(Dense(2, 128, relu), Dense(128, length(actions(env))))
    Q̂ = deepcopy(Q) #Q_target
    Q_highest_reward = deepcopy(Q) #best_Q to return at the end

    #initialize optimiser for later use, kept it same as starter
    optimizer = Flux.setup(ADAM(0.0005), Q) #add dynamic learning rate α?

    #set up trackers for plotting learnign curve and keep track of best Q
    cumulative_rewards = []
    highest_reward = Float64(0)

    #need this for "freezing" Q, i.e Q̂(s,a) ← Q(s,a) updates
    step_count = 0

    #loss function, following algorithm above, target changes depending on s′
    function loss(Q, s, a_ind, r, s′, done)
        if done == true
            target = r
        else
            #Don't need to add a′ because maximum is taking care of it
            target = r + γ * maximum(Q̂(s′))
        end

        return Flux.Losses.mse(Q(s)[a_ind], target)
    end

    # a function to obtain random sample from buffer depending on requested batch size
    function sample_minibatch(buffer, batch_size)
        return [buffer[rand(1:length(buffer))] for _ in 1:batch_size]
    end

    # custom evaluation function to see how well a Q function perfoms
    # need this to essentially keep track of best Q
    #= even if cumulative reward is randomly very high in an episode, it does not mean
        that the Q function will consistently perform well =#
    function evaluate_current_Q(env, Q, n_episodes, max_steps, γ)
        total_rewards = []

        for episode in 1:n_episodes
            reset!(env)
            s = observe(env)
            episode_reward = 0.0
            t = 0

            while !terminated(env) && t < max_steps
                action = argmax(Q(s))
                r = act!(env, actions(env)[action])
                episode_reward += γ^t * r
                s = observe(env)
                t += 1
            end

            push!(total_rewards, episode_reward)
        end

        mean_reward = mean(total_rewards)
        return mean_reward
    end

    #For M episodes
    for episode in 1:M
        #get state and set up episodic reward
        s = observe(env)
        episode_reward = Float64(0)

        #dynamic ϵ starting at 0.5 and decatying to 0.05 over half the episodes
        ϵ = max(ϵ_end, ϵ_start - (episode - 1) * (ϵ_start - ϵ_end) / ϵ_decay)

        #steps in environment, either terminal or max T
        for t in 1:T
            #Sampling phase ======================================================================
            #simply interacting with environment and adding things to the buffer
            a_ind = rand() < ϵ ? rand(1:length(actions(env))) : argmax(Q(s))

            step_count += 1 #adding this for Q̂ updates

            r = act!(env, actions(env)[a_ind])
            episode_reward += (γ^(t-1))*r

            s′ = observe(env)
            done = terminated(env)

            push!(buffer, (s,a_ind,r,s′, done))

            s = s′

            #training phase ======================================================================
            #Training after buffer is at least 20% full and after every 4 steps
            if length(buffer) >= N*0.2 && t%4==0
                data = sample_minibatch(buffer, batch_size)
                Flux.Optimise.train!(loss, Q, data, optimizer)
            end

            #Freeze Q̂ ============================================================================
            #need separate counter because t can reach terminal before 200 steps
            if step_count % C == 0
                Q̂ = deepcopy(Q)
            end

            #Evaluate Q ==========================================================================
            #=if the agent reaches a terminal state, it is a good evaluate the Q, I also thought of
                trying it every time we optimise Q, but that will have to wait depending on available time=#
            if done == true
                current_performance = evaluate_current_Q(env, Q, 100, T, γ)
```

```
                if current_performance > highest_reward
                    highest_reward = current_performance
                    Q_highest_reward = deepcopy(Q) #keep the best Q
                end
                #end environment interaction if terminal state
                break
            end
        end
        #store best reward to plot
        push!(cumulative_rewards, episode_reward)
        reset!(env) #resetting at end of episode
    end
    #plot discounted cumulative reward achieved over many episodes, just to gauge performance
    display(plot(cumulative_rewards, title="Rewards Over Training", xlabel="Training Episode", ylabel="Cumulative Reward", legend=false))
    #return the best performing Q
    return Q_highest_reward
end

env = QuickWrapper(HW5.mc, actions=[-1.0, -0.5, 0.0, 0.5, 1.0], observe=mc->observe(mc)[1:2])

N = 50_000 #buffer size
M = 1000 #num of episodes #6250 episodes to get near deepmind ratio
T = 400 #max num of steps #standard for mountain car is usually 200 steps
γ = 0.99 #discount factor
ϵ_start = 0.5 #exploration, currently ceiling
ϵ_end = 0.05 #floor
ϵ_decay = M/2 #number of episodes over which to decay
C = 200 #target network update frequency
batch_size = 32

Q = dqn(env, N, M, T, γ, ϵ_start, ϵ_end, ϵ_decay, C, batch_size)

xs = -3.0f0:0.1f0:3.0f0
vs = -0.3f0:0.01f0:0.3f0
heatmap(xs, vs, (x, v) -> maximum(Q([x, v])), xlabel="Position (x)", ylabel="Velocity (v)", title="Max Q Value")
```