Decision Making under Uncertainty
ASEN 6519-007
Maggie Jacoby **Homework 4** March 17, 2020

Conceptual Questions

Problem 1:

Using the deep learning library of your choice, fit a neural network to approximate the function $f(x) = cos(20x^2)$ for the range $x \in [0, 1]$. Plot a set of 100 data points fed through the trained model and plot the learning curve.

Solution
By training a neural network with Flux.jl, I was able to achieve a fairly close match to the function, as seen in Figure 1. The mean squared error (MSE) seen during training is plotted as a function of the number of epochs used for training, in Figure 2. As expected, the error seems to decrease monotonically to 0 as the number of epochs used in training is increased. The improvement in error seems to level out around 500 epochs. Figure 3 shows a zoomed in version of the same graph for 500 to 1,000 training epochs. This shows that after about 800 epochs, the error increases some and seems to oscillate. It surprising that this happens, but for this low number of points (it was trained on 500 points), this oscillation could be due to a couple of points being incorrectly predicted. As the model improves it might alternately change the prediction of a small number of points, sometimes getting slightly better, sometimes slightly worse, leading to the observed oscillation.

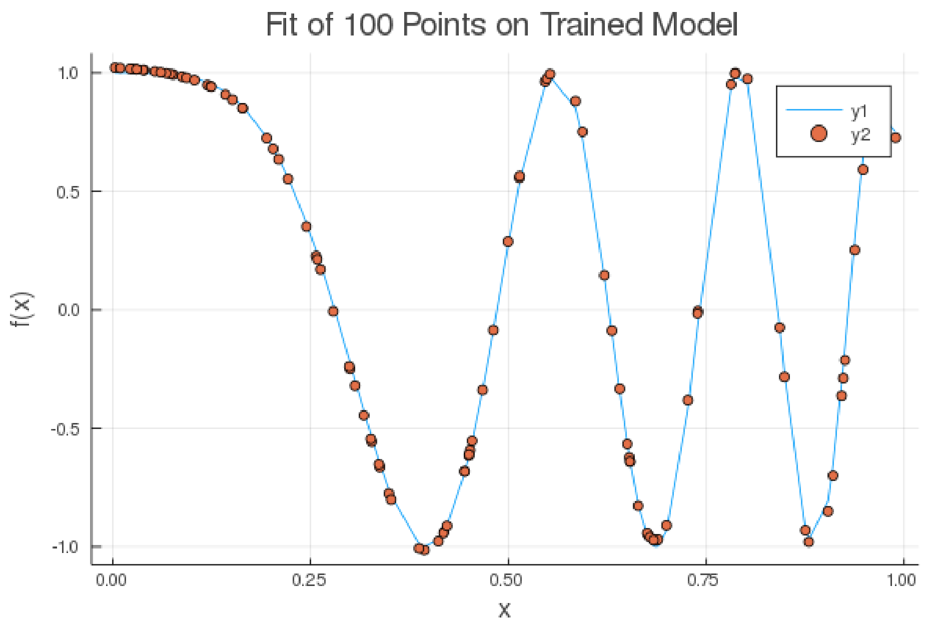

Figure 1: Expected y value for 100 randomly selected points as predicted by the trained neural network.

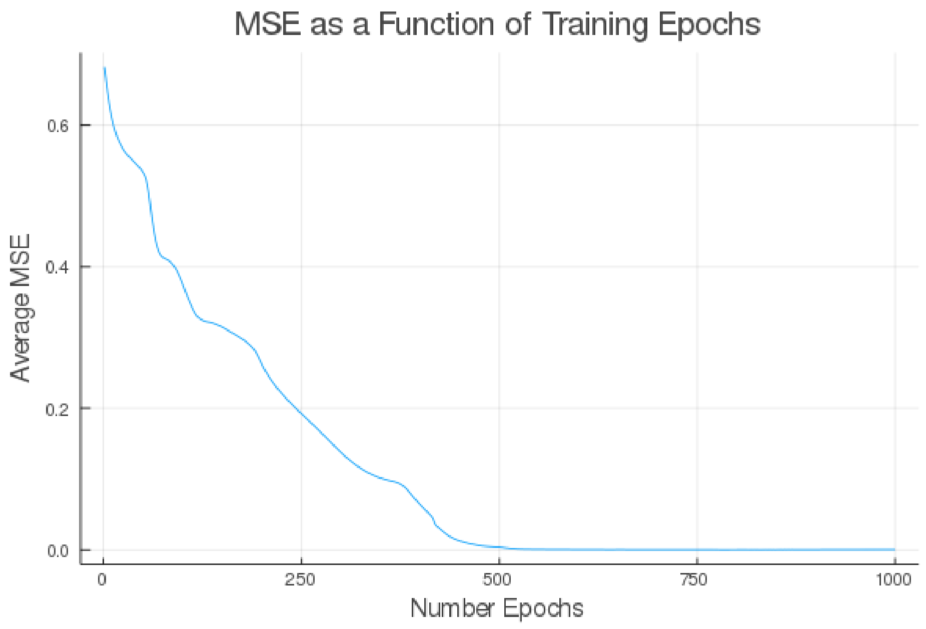

Figure 2: The average mean squared error for the fit of points to a line plotted with respect to the number of epochs used to train the function.

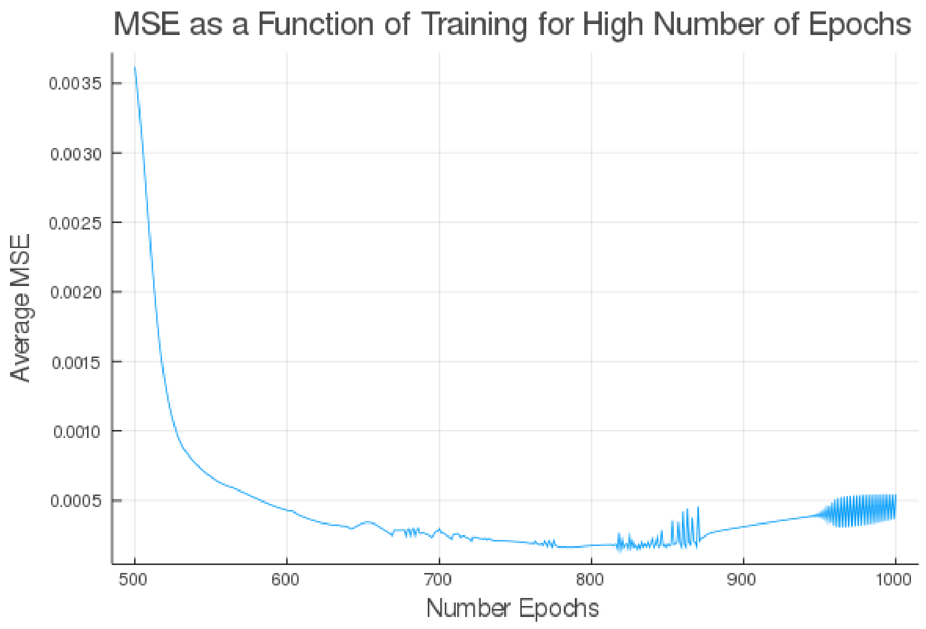

Figure 3: A zoomed in view of the MSE (same as above) focusing on 500 to 1000 epochs.

Code for Question 1

```
using Plots
using Flux
using StaticArrays
using Statistics


function MyData(n)
    x = rand(n)
    y = cos.(20 * x.^2)
    return ([(SVector(x[i]), SVector(y[i])) for i in 1:length(x)], x)
end

function train_model(data_train, dx, data_test, dy, N_epochs)
    m = Chain(Dense(1,50,σ), Dense(50,50,σ), Dense(50,1))
    loss(x, y) = Flux.mse(m(x), y)
    ps = Flux.params(m)

    x_train = [x[1] for x in data_train]
    y_train = [y[2] for y in data_train]
    x_test = [x[1] for x in data_test]
    y_test = [y[2] for y in data_test]
    Losses = Array{Any}(undef, N_epochs)

    for i in 1:N_epochs
        Flux.train!(loss, ps, data_train, ADAM())
        L = loss.(x_test, y_test)
        Losses[i] = mean(L)
    end
    p = plot(sort(dy), x->(cos(20*x^2)), title="Actual Function")
    scatter_trained = scatter!(p, dy, [first(y) for y in m.(x_test)])
    plot(scatter_trained, xlabel="x", ylabel="f(x)",
        title="Fit of 100 Points on Trained Model")
    savefig("DMU_HW4_Q1_fit2")
    return Losses
end

D_train = MyData(500)
train = D_train[1]
dx = D_train[2]

D_test = MyData(100)
test = D_test[1]
dy = D_test[2]

N = 1000

Losses = train_model(train, dx, test, dy, 1000)

### Plot and save the error plots
plot_error = plot(1:N, Losses, xlabel="Number Epochs", ylabel="Average MSE",
    title="MSE as a Function of Training Epochs", legend=false)
savefig("DMU_HW4_Q1_error")

plot_error_zoom = plot(500:N, Losses[500:N], xlabel="Number Epochs", ylabel="Average MSE",
    title="MSE as a Function of Training for High Number of Epochs", legend=false)
savefig("DMU_HW4_Q1_error_zoom")
```

Problem 2:

Implement **two** different traditional or deep learning algorithms to learn a policy for the `DMUStudent.HW4.gw` grid world environment. Use a discount factor of $\gamma = 0.95$ to encourage the agent to reach goals more quickly. Plot a learning curve for each and comment on why one performs better than the other.

Solution
I compared double Q-learning with vanilla Q-learning and Sarsa. Both double-Q and sarsa performed somewhat better than single-Q both on average, and when looking at the best performing example of all three. I was expecting double-Q to perform at least somewhat better than single-Q, though I was surprised that double-Q and sarsa had such similar performance.

What is especially surprising is that the performance of all three seems independent of the number of epochs used in training. This must because one epoch was generally able to cover the whole space, meaning that not much new information was gained in additional episodes. I tried lowering the number of steps per episode, but didn't see an overall change.

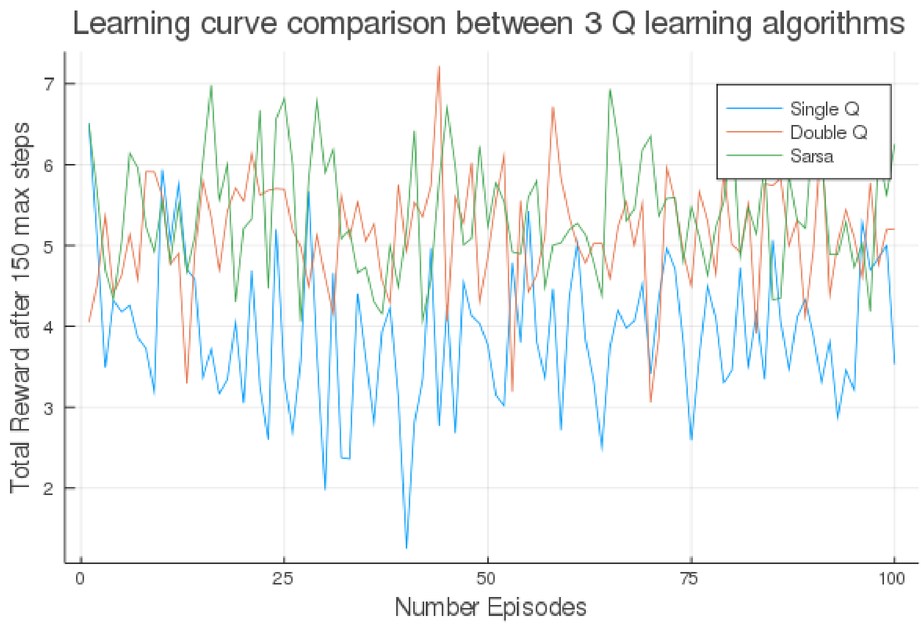

Figure 4: Comparison of the performance of three different Q-learning algorithms.

Code for Question 2

```
using POMDPs
using POMDPModels
using TabularTDLearning
using POMDPPolicies
using RLInterface
using POMDPModelTools
using DMUStudent
using ElectronDisplay
using Statistics
using Plots

## Map states and actions to integers
a_ind = Dict(:right => 1, :down => 2, :left => 3, :up => 4)
s_ind = Dict([i,j] => 10*(i-1)+j for i in 1:10 for j in 1:10 )
s_ind[[-1,-1]] = 101

### Single and double Q learning algorithms
function Q_learn(m, Q, alph, eps, max_t, g = 0.95)
    A = actions(m)
    s = reset!(m)
    done = false
    rsum = 0.0
    t = 1
    while t <= max_t
        si = s_ind[s]

        if rand() < eps
            a = rand(A)
        else
            a = A[argmax(Q[si, :])]
        end

        ai = a_ind[a]
        sp, r, done, info = step!(m, a)
        spi = s_ind[sp]
        Q[si,ai] += alph * (r + g*argmax(Q[spi, :]) - Q[si,ai])

        if done
            return Q
        end
        s = sp
        t += 1
    end
    return Q
end

###
function double_Q_learn(m, Q1, Q2, alph, eps, max_t, g = 0.95)
    A = actions(m)
    s = reset!(m)
    done = false
    rsum = 0.0
    t = 1
    while t <= max_t
        si = s_ind[s]
        Q = (Q1+Q2)/2

        if rand() < eps
            a = rand(A)
        else
            a = A[argmax(Q[si, :])]
        end

        ai = a_ind[a]
        sp, r, done, info = step!(m, a)
        spi = s_ind[sp]

        if rand() < 0.5
            api = argmax(Q1[spi, :])
            Q1[si,ai] += alph * (r + g*Q2[spi,api] - Q1[si,ai])
        else
            api = argmax(Q2[spi, :])
            Q2[si,ai] += alph * (r + g*Q1[spi,api] - Q2[si,ai])
        end

        if done
            return Q1, Q2
        end
        s = sp
        t += 1
    end
    return Q1, Q2
end

### Sarsa algorithm

function sarsa_Q_learn(m, Q, alph, eps, max_t, g = 0.95)
    A = actions(m)
    s = reset!(m)
    si = s_ind[s]
    done = false
    rsum = 0.0
    t = 1

    if rand() < eps
        a = rand(A)
    else
        a = A[argmax(Q[si, :])]
    end

    while t <= max_t
        si = s_ind[s]
        ai = a_ind[a]
        sp, r, done, info = step!(m, a)
        spi = s_ind[sp]

        if rand() < eps
            ap = rand(A)
        else
            ap = A[argmax(Q[si, :])]
        end

        api = a_ind[ap]
        Q[si,ai] += alph * (r + g*Q[spi,api] - Q[si,ai])

        if done
            return Q
        end
        s = sp
        a = ap
        t += 1
    end
    return Q
end

### Training function (use the Q learning functions above)
function train_Q(m, n, a, e, s, DQ, sarsa)
    Q1 = zeros(101,4)
    Q2 = zeros(101,4)
    Q = zeros(101,4)

    if DQ
        for i in 1:n
            Q1, Q2 = double_Q_learn(m, Q1, Q2, a, e, s)
        end
        return (Q1+Q2)

    elseif sarsa
        for i in 1:n
            Q = sarsa_Q_learn(m, Q, a, e, s)
        end
        return Q

    else
        for i in 1:n
            Q = Q_learn(m, Q, a, e, s)
        end
        return Q
    end
end

### test the trained Q functions (works for any Q alg)
function test_policy(m, Q, Num_runs, max_t)
    all_R = []
    for i in 1:Num_runs
        A = actions(m)
        s = reset!(m)
        done = false
        r_sum = 0
        t = 1
        while t <= max_t
            si = s_ind[s]
            a = A[argmax(Q[si, :])]
            sp, r, done, info = step!(m, a)
            r_sum += r
            if done
                break
                println("breaking, $(r_sum)")
            end
        end
        push!(all_R, r_sum)
    end
    return mean(all_R)
end

### Run all train/test algorithms
function train_test(m, a, e, n, DQ, sarsa)#, al, e, n_steps)
    println("starting to train and test algorithms...")
    N = 100        # Number of runs to average during testing
    T = 50         # Number of max steps to take per test run
    LC = []

    for episodes in 1:100
        Q = train_Q(m, episodes, a, e, n, DQ, sarsa)
        p = test_policy(m, Q, N, T)
```

```julia
        push!(LC, (episodes, p))
    end
    return LC
end

###
a = 0.2
e = 0.2
n = 100
my_gw = HW4.gw

println("a $a, e $e, t $n")
LC_single = train_test(my_gw, a, e, n, false, false)
R_Q = [x[2] for x in LC_single]
LC_double = train_test(my_gw, a, e, n, true, false)
R_DQ = [x[2] for x in LC_double]

LC_sarsa = train_test(my_gw, a, e, n, false, true)
R_sarsa = [x[2] for x in LC_sarsa]

print("Q $(maximum(R_Q)), QQ $(maximum(R_DQ)), sarsa $(maximum(R_sarsa))")

### Plot Learning Curve
R_x = [x[1] for x in LC_single]

plot(R_x, R_Q, xlabel="Number Episodes",
ylabel="Total Reward after 150 max steps",  label = "Single Q",
title="Learning curve comparison between 3 Q learning algorithms")#, legend=false)
plot!(R_x, R_DQ, label = "Double Q")
plot!(R_x, R_sarsa, label = "Sarsa")

savefig("DMU_HW4_Q2_compare3")

### Small Parametric Analysis
a_range = [0.2, 0.5, 0.8]
e_range = [0.2, 0.5, 0.8]
step_range = [20, 50, 100]

for a in a_range
    for e in e_range
        for s in step_range
            println("alpha $a, epsilon $e, max steps: $s")
            LC_single = train_test(my_gw, a, e, n, false)
            R_single = [x[2] for x in LC_single]
            LC_double = train_test(my_gw, a, e, n, true)
            R_double = [x[2] for x in LC_double]
            println("Q $(maximum(R_single)), QQ $(maximum(R_double))")
        end
    end
end
```

Challenge Problem

Problem 3:

Learn a policy for the mountain car environment `DMUStudent.HW4.mc`. You may use *any* libraries.

Solution
I had a significantly harder time with this problem. I initially tried to use a Deep-Q-Learning, though was never able to get a score greater than 0. I then tried traditional Q methods, using both sarsa and double-Q. LONG training time got me scores greater than 0, though not much. The long training meant I wasn't able to experiment with parameters very much in this case. I finally tried a purely heuristic method, telling the car to first go left to some point, then right. Through this I was able to do somewhat better, though still not great. I suspect that the lack of any good results for thee traditional Q methods means I am making some mistake in implementation of the algorithms, though I wasn't able to figure out what.

Solution submitted to leaderboard.

Code for Question 3

```
using DMUStudent
using RLInterface
using POMDPs
using POMDPModels
using ElectronDisplay
ElectronDisplay.CONFIG.single_window = true
using DeepQLearning
using Flux
using POMDPPolicies
using POMDPSimulators
using Random

### Deep Q Learning attempt
mc = HW4.mc;
mtnCar = RLInterface.MDPEnvironment(mc.problem, Vector{Float32})

actionsA = [-1.0, -0.5, 0.0, 0.5, 1.0]
RLInterface.actions(::typeof(mtnCar)) = actionsA
RLInterface.action(::typeof(RandomPolicy(mtnCar.problem)), ::typeof(reset!(mtnCar))) =
rand(actionsA)
model = Chain(Dense(2,64), Dense(64,64),  Dense(64,5));

solver = DeepQLearningSolver(qnetwork=model, max_steps=2000000,
                             learning_rate=0.001, prioritized_replay=true,
                             exploration_policy = max_energy, buffer_size = 50000,
                             train_start = 1000, max_episode_length = 500)
policy = solve(solver, mtnCar)

function max_energy(policy, env, obs, global_step, rng)
    s1, s2 = env.state
    if global_step < 1200000
        if s2 < 0.0
            return (-1.0, global_step)
        else
            return (1.0, global_step)
        end
    else
        return (policy(env.state), global_step)
    end
end
################

### Traditional methods
## discreetize x and xdot
function get_x(x)
    if x < -1.2
        x = -1.2
    elseif x > 0.5
        x = 0.5
    end
    return round(Int, x*30)+37
end

function get_v(v)
    return round(Int, v*150)+16
end

### max energy policy for exploring
function max_energy_trad(env) #policy, env, obs, global_step, rng)
    x, v = env
    if v < 0.0
        return -1.0
    else
        return 1.0
    end
end

## Heeuristic policy - go left until high enough, then go right
## this got my highest score. It doesn't involve any Q learning :(
function max_direction_policy(env)
    x, v = env
    if x > -.98 && v < 0.0
        return = -1.0
    else
        return = 1.0
    end
end

actionsA = [-1.0, -0.5, 0.0, 0.5, 1.0]
actions_ind = Dict(-1.0=>1, -0.5 => 2, 0.0 => 3, 0.5 => 4, 1 => 5)

### Use traditional Q learning methods - not included, other iterations of Q
function heuristic_sarsa(m, Q, alph, max_t, g = 0.95)
    A = actionsA
    s = reset!(m)
    done = false
    rsum = 0.0
    t = 1
    a = rand(A)
    min_x = 0.0
    while t <= max_t
        x = get_x(s[1])
        v = get_v(s[2])
        ai = actions_ind[a]

        sp, r, done, info = step!(m, a)
        xp = get_x(sp[1])
        vp = get_v(sp[2])

        if rand() < 0.1
            ap = rand(A)
        else
            if xp > -1.1 && min_x > -0.95 && vp < 0.0
                ap = -1.0
            else
                ap = 1.0
            end
        end
        api = actions_ind[ap]
        if min_x < xp
            min_x = xp
        end
        Q[x, v, ai] += alph * (r + (g^t)*Q[xp, vp, api] - Q[x, v, ai])
        if done
            return Q
        end
        s = sp
        a = ap
        t += 1
    end
    return Q
end


function train_Q(m, n, a, max_t)
    display("training Q....")
    Q = zeros(52,30,5)
    for i in 1:n
        Q = heuristic_sarsa(m, Q, a, max_t)
    end
    return Q
end


a = 0.2 #learning rate, alpha
n = 500000 #number of episodes for training
max_t = 500

r_sarsa = train_Q(MC, n, a, max_t)

function sarsa_policy(s, Q=r_sarsa, A=actionsA)
    x = get_x(s[1])
    v = get_v(s[2])
    return A[argmax(Q[x, v, :])]
end


### View a state
function view_state(m)
    s = reset!(m)
    done = false
    min_x = 0
    for t in 1:200
        if s[1] > -1.2 && min_x > -0.95 && s[2] < 0.0
            a = -1.0
        else
```

```
            a = 1.0
        end

        if s[1] < min_x
            min_x = s[1]
        end
        if done
            display("min x: $(min_x)")
            display("time step: $(t)")
            break
        end
        sp, r, done, info = step!(m, a)
        electrondisplay(RLInterface.render(m))
        s = sp
        t +=1
    end
end

MC = HW4.mc;
view_state(MC)
```