Stochastic Optimization Assignment

Andrea Cognolato, s281940@studenti.polito.it

January 25, 2021

1 Problem 1: Static Optimization

The aim of this problem is threefold. Firstly, to develop a simulator of an industry and use it to evaluate the effect of various maintenance policies on its cumulative 60 day gain. Secondly, to build a Neural Network surrogate model, using values obtained from the simulator, to be able to approximate the gain function in a computationally efficient way. Thirdly to use the surrogate model to find the optimal parameters for maximizing the gain, using a stochastic gradient descent method.

Below, are three Julia codes[1] which implement these aforementioned objectives.

```
# Simulates a single realization for 60 days
function simulate_realization(M, s1, s2)
    # Number of days
    days = 60

    # State
    # number of worn machines
    W = zeros(Int, days)
    # number of broken machines
    B = zeros(Int, days)

    # probability of new -> worn
    p12 = (1/60) + (1/200 - 1/60) * sqrt(1 - ((10 - s1) / 10)^2)
    # probability of worn -> broken
    p23 = (1/30) + (1/100 - 1/30) * sqrt(1 - ((10 - s2) / 10)^2)

    # Simulate all days
    for i in 1:(days-1)
        B12 = rand(Binomial(M-W[i]-B[i], p12))
        B23 = rand(Binomial(W[i], p23))

        W[i+1] = W[i] + B12 - B23
        B[i+1] = B[i] +     + B23 - B[i]
    end

    # Calculate the number of new machines every day
    N = M * ones(days) - W - B

    N, W, B
end

# Calculate the cumulative gain over 60 days
```

[1] Full code available at https://github.com/mrandri19/polito-numerical-optimization

```
function cumulative_gain(M, B)
    total = 0.0
    for day in 1:60
        total += (1 * (M - B[day]) - 90 * B[day] - s1 - s2)
    end
    return total
end

# Performs a simulation experiment that calculates the gain, by
# averaging 10,000 realizations
function simulate(M, s1, s2)
    gain = 0.0

    realizations = 10_000
    for _ in 1:realizations
        N, W, B = simulate_realization(M, s1, s2)
        gain += cumulative_gain(M, B)
    end

    return gain / realizations
end

# Total number of machines
M = 2_000

X = []
y = []
@time for j=2:2:10, k=2:2:10
    gain = simulate(M, j, k)
    push!(X, [j, k])
    push!(y, [gain])
    println("($j, $k) = $(@sprintf("%3d" ,gain))")
end
```

The code's output:

```
(2, 2) = 83609
(2, 4) = 90274
(2, 6) = 94651
(2, 8) = 97224
(2, 10) = 98051
(4, 2) = 91288
(4, 4) = 96495
(4, 6) = 99957
(4, 8) = 101922
(4, 10) = 102568
(6, 2) = 96030
(6, 4) = 100338
(6, 6) = 103174
(6, 8) = 104825
(6, 10) = 105386
(8, 2) = 98693
(8, 4) = 102497
(8, 6) = 105029
(8, 8) = 106480
```

```
(8, 10) = 106966
(10, 2) = 99553
(10, 4) = 103199
(10, 6) = 105586
(10, 8) = 107006
(10, 10) = 107477
6.485861 seconds (61.76 M allocations: 1.704 GiB, 6.53% gc time)
```

1.1 Neural Network

```
# Define the nonlinearity used in our network and its derivative
sigma(x) = 1 / (1 + exp(-x))
sigma_prime(x) = sigma(x) * (1 - sigma(x))

# Compute y's mean and standard deviation to perform
# standard scaling
y_mean = mean(y)[1]
y_std = std(y)[1]

function z_transform(yi)
    (yi - y_mean) / y_std
end

function inv_z_transform(yi)
    (yi * y_std) + y_mean
end

# Compute the forward pass of a 1-hidden-layer neural network
function nn(x, W1, b1, W2, b2)
    z1 = W1 * x + b1
    a1 = sigma.(z1)
    z2 = W2 * a1 + b2
    a2 = z2
    return a2[1]
end

# Compute the Mean Squared Error of the NN's predictions for all samples
function compute_loss(X, y, W1, b1, W2, b2)
    loss = 0.0
    for (x, y_true) in zip(X, y)
        y_true = z_transform.(y_true)
        y_hat = nn(x, W1, b1, W2, b2)

        loss += (y_hat[1] - y_true[1])^2
    end
    loss /= (2 * size(X, 1))
    return loss
end

Random.seed!(1234);

# Hyperparameters
# Learning Rate/Step Size
mu = 5e-3
```

```
# Parameters and weights/biases initialization
INPUT_SIZE = 2
HIDDEN_LAYER_SIZE = 100
W1 = randn(HIDDEN_LAYER_SIZE, INPUT_SIZE)
b1 = randn(HIDDEN_LAYER_SIZE, 1)

OUTPUT_LAYER_SIZE = 1
W2 = randn(OUTPUT_LAYER_SIZE, HIDDEN_LAYER_SIZE)
b2 = randn(OUTPUT_LAYER_SIZE, 1)

# Training loop
for it in 1:10_000
    # For every sample in the training dataset
    for (x, y_true) in zip(X, y)
        # z-transform the target before training the network with it
        y_true = z_transform.(y_true)

        # Forward step to evaluate intermediate results
        z1 = W1 * x + b1
        a1 = sigma.(z1)
        z2 = W2 * a1 + b2
        a2 = z2

        # Backward step to evaluate parameter derivatives w.r.t. loss
        # d(loss)/d(z2) is `dz2`
        dz2 = a2 - y_true
        dW2 = dz2 * a1'
        db2 = dz2

        dz1 = (W2' * dz2) .* sigma_prime.(z1)
        dW1 = dz1 * x'
        db1 = dz1

        # Steepest descent to find optimal parameters
        W2 -= mu * dW2
        b2 -= mu * db2

        W1 -= mu * dW1
        b1 -= mu * db1
    end

    # Learning rate's geometric decay
    mu *= 0.9995

    if it % 200 == 0
        println("[$it]:
        $(@sprintf("%.3E", compute_loss(X, y, W1, b1, W2, b2))),
        mu=$(@sprintf("%.3E", mu))")
    end
end
```

The code's output:

```
[200]: 1.148E-03, mu=4.524E-03
```

```
[400]: 6.184E-04, mu=4.093E-03
[600]: 4.721E-04, mu=3.704E-03
[800]: 4.019E-04, mu=3.351E-03
[1000]: 3.589E-04, mu=3.032E-03
[1200]: 3.290E-04, mu=2.744E-03
[1400]: 3.068E-04, mu=2.482E-03
[1600]: 2.898E-04, mu=2.246E-03
[1800]: 2.763E-04, mu=2.032E-03
[2000]: 2.655E-04, mu=1.839E-03
[2200]: 2.567E-04, mu=1.664E-03
[2400]: 2.494E-04, mu=1.506E-03
[2600]: 2.433E-04, mu=1.362E-03
[2800]: 2.382E-04, mu=1.233E-03
[3000]: 2.339E-04, mu=1.115E-03
[3200]: 2.303E-04, mu=1.009E-03
[3400]: 2.271E-04, mu=9.130E-04
[3600]: 2.244E-04, mu=8.261E-04
[3800]: 2.220E-04, mu=7.475E-04
[4000]: 2.200E-04, mu=6.763E-04
[4200]: 2.182E-04, mu=6.120E-04
[4400]: 2.166E-04, mu=5.537E-04
[4600]: 2.152E-04, mu=5.010E-04
[4800]: 2.140E-04, mu=4.533E-04
[5000]: 2.129E-04, mu=4.102E-04
[5200]: 2.120E-04, mu=3.711E-04
[5400]: 2.111E-04, mu=3.358E-04
[5600]: 2.104E-04, mu=3.038E-04
[5800]: 2.097E-04, mu=2.749E-04
[6000]: 2.091E-04, mu=2.487E-04
[6200]: 2.085E-04, mu=2.251E-04
[6400]: 2.081E-04, mu=2.036E-04
[6600]: 2.076E-04, mu=1.843E-04
[6800]: 2.072E-04, mu=1.667E-04
[7000]: 2.069E-04, mu=1.509E-04
[7200]: 2.066E-04, mu=1.365E-04
[7400]: 2.063E-04, mu=1.235E-04
[7600]: 2.061E-04, mu=1.117E-04
[7800]: 2.058E-04, mu=1.011E-04
[8000]: 2.056E-04, mu=9.149E-05
[8200]: 2.054E-04, mu=8.278E-05
[8400]: 2.053E-04, mu=7.490E-05
[8600]: 2.051E-04, mu=6.777E-05
[8800]: 2.050E-04, mu=6.132E-05
[9000]: 2.049E-04, mu=5.548E-05
[9200]: 2.048E-04, mu=5.020E-05
[9400]: 2.047E-04, mu=4.542E-05
[9600]: 2.046E-04, mu=4.110E-05
[9800]: 2.045E-04, mu=3.719E-05
[10000]: 2.044E-04, mu=3.365E-05
```

1.2 Optimization

```
# Find the maximum of the surrogate model using the spall method for
# stochastic gradient ascent
```

```
function maximize_gain(;verbose = false)
    # Iteration counter
    m = 1
    m_max = 1000

    # Random starting poing
    x = [rand(2:2:10), rand(2:2:10)]

    # Gradient ascent step length
    mu = 5e-2

    C = 0.1

    while m < m_max
        # Compute a random vector of +-1
        h = rand([1, -1], size(x, 1))

        # compute the finite difference step length
        c = C / m^0.2

        nn_pred = inv_z_transform(nn(x, W1, b1, W2, b2))
        println("
            x=$(round.(x, digits=2)),
            nn=$(round(nn_pred, digits=1)),
            c=$(round(c, digits=3))"
        )

        # compute the derivative of the neural network w.r.t input using Spall's method
        finite_diff =
            inv_z_transform(nn(x + c*h, W1, b1, W2, b2)) -
            inv_z_transform(nn(x - c*h, W1, b1, W2, b2))
        dx = [finite_diff / 2*c*h[i] for i in 1:size(x, 1)]

        # gradient ascent step
        x_new = x + mu * dx

        if (norm(x - x_new)) / norm(x_new) < 1e-4
            verbose && println("Hit termination condition at m=$m, x=$(round.(x, digits=2))")
            break
        end

        x = x_new

        m += 1
    end

    if m == m_max
        verbose && println("Maximum number of iterations reached, m=$m")
    end
end
maximize_gain(verbose = true)
```

The code's output:

```
x=[2.0, 4.0], nn=90205.9, c=0.1
```

```
x=[1.22, 4.78], nn=91533.9, c=0.087
x=[0.75, 5.25], nn=92082.2, c=0.08
x=[1.52, 6.01], nn=94086.2, c=0.076
x=[2.35, 6.84], nn=96710.1, c=0.072
x=[3.24, 7.73], nn=99789.5, c=0.07
x=[3.75, 7.22], nn=100617.4, c=0.068
x=[4.05, 6.91], nn=101058.0, c=0.066
x=[4.79, 7.65], nn=103201.6, c=0.064
x=[5.3, 8.16], nn=104317.7, c=0.063
x=[5.57, 7.89], nn=104549.3, c=0.062
x=[5.91, 8.22], nn=105086.7, c=0.061
x=[5.9, 8.24], nn=105087.3, c=0.06
x=[5.89, 8.24], nn=105087.3, c=0.059
x=[6.15, 8.49], nn=105437.8, c=0.058
x=[6.13, 8.52], nn=105439.7, c=0.057
x=[6.34, 8.73], nn=105699.7, c=0.057
x=[6.53, 8.91], nn=105904.3, c=0.056
x=[6.69, 9.08], nn=106070.9, c=0.055
x=[6.84, 9.23], nn=106210.2, c=0.055
x=[6.81, 9.26], nn=106216.9, c=0.054
x=[6.79, 9.28], nn=106218.6, c=0.054
x=[6.78, 9.29], nn=106219.0, c=0.053
x=[6.78, 9.29], nn=106219.1, c=0.053
x=[6.91, 9.42], nn=106332.8, c=0.053
x=[7.03, 9.54], nn=106432.0, c=0.052
x=[7.02, 9.55], nn=106432.6, c=0.052
x=[7.01, 9.55], nn=106432.7, c=0.051
x=[7.01, 9.55], nn=106432.8, c=0.051
x=[7.12, 9.66], nn=106517.3, c=0.051
x=[7.22, 9.76], nn=106593.0, c=0.05
x=[7.21, 9.77], nn=106593.2, c=0.05
x=[7.3, 9.86], nn=106661.0, c=0.05
x=[7.3, 9.86], nn=106661.2, c=0.049
x=[7.39, 9.95], nn=106722.3, c=0.049
x=[7.38, 9.95], nn=106722.5, c=0.049
x=[7.46, 10.04], nn=106778.1, c=0.049
x=[7.54, 10.11], nn=106829.6, c=0.048
x=[7.54, 10.12], nn=106829.7, c=0.048
x=[7.54, 10.12], nn=106829.7, c=0.048
x=[7.54, 10.12], nn=106829.7, c=0.048
x=[7.53, 10.12], nn=106829.8, c=0.047
x=[7.61, 10.19], nn=106875.8, c=0.047
x=[7.68, 10.26], nn=106918.9, c=0.047
x=[7.74, 10.33], nn=106959.5, c=0.047
Hit termination condition at m=45, x=[7.74, 10.33]
```

2 Discussion

2.1 Neural Network

The neural network architecture consists of 2 input neurons, 100 sigmoid hidden neurons, 1 linear output neuron. Both input and hidden layers have biases. The model quickly converges to a good approximation of the training data as we can see in figure 1.

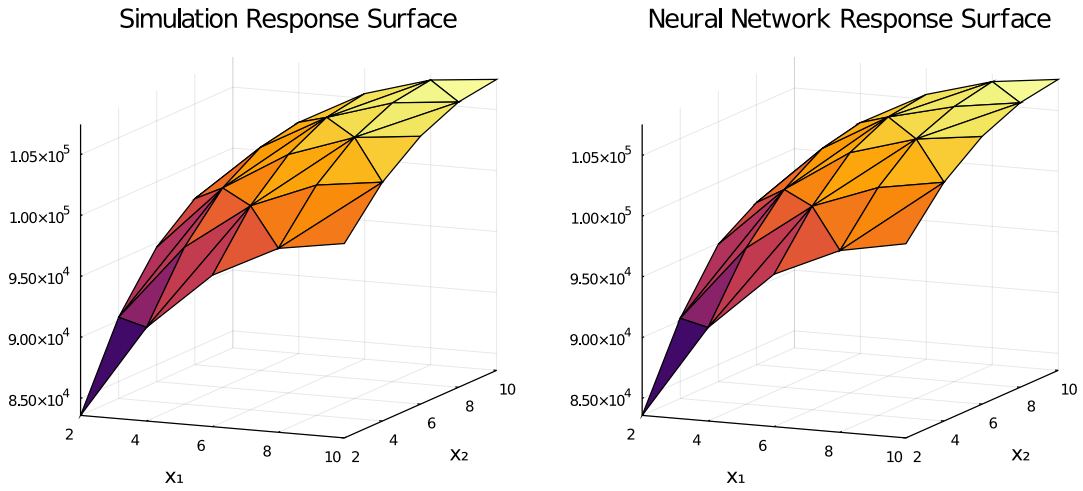

Figure 1: Comparison of simulation and neural network response surfaces

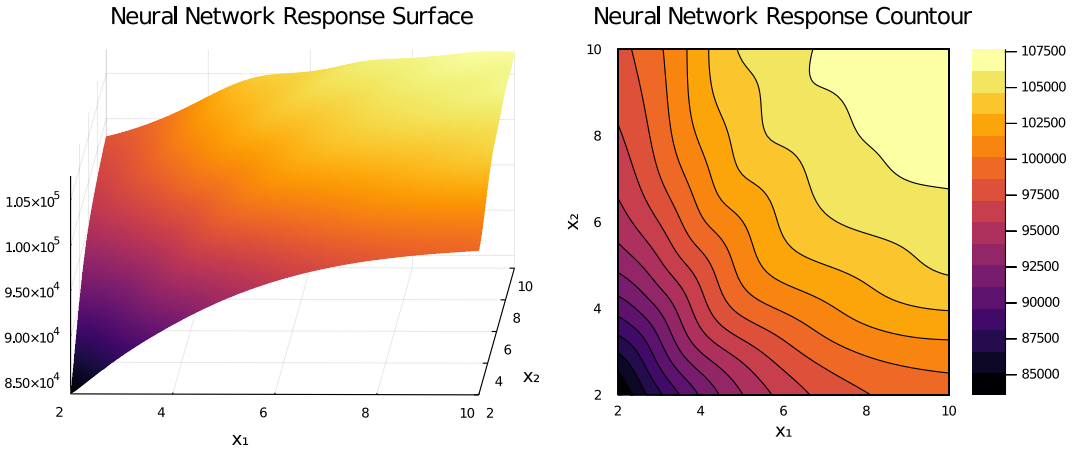

Figure 2: Neural network response surface over the training domain

To test the approximation's quality, the neural network is evaluated at three values outside the training domain. The neural network predictions for points outside the training domain are not very accurate as seen in the output below.

```
Gain NN: 86110, Gain Simulation: 58366
Gain NN: 82789, Gain Simulation: 63874
Gain NN: 83779, Gain Simulation: 59998
```

By comparing figure 2 and figure 3 it is possible to see how the network's behaviour changes rapidly as soon as the training domain is exited. These changes are also initialization-dependent, as we initialize our weight and biases with random values.

2.2 Optimization

Spall's method for Simultaneous Perturbation Stochastic Approximation (SPSA) is implemented and applied on the surrogate model to optimize the stochastic objective function.

A random initialization point is chosen, sampling uniformly inside the training domain. We use a fixed step size for the finite differences, of length 5×10^{-2}.

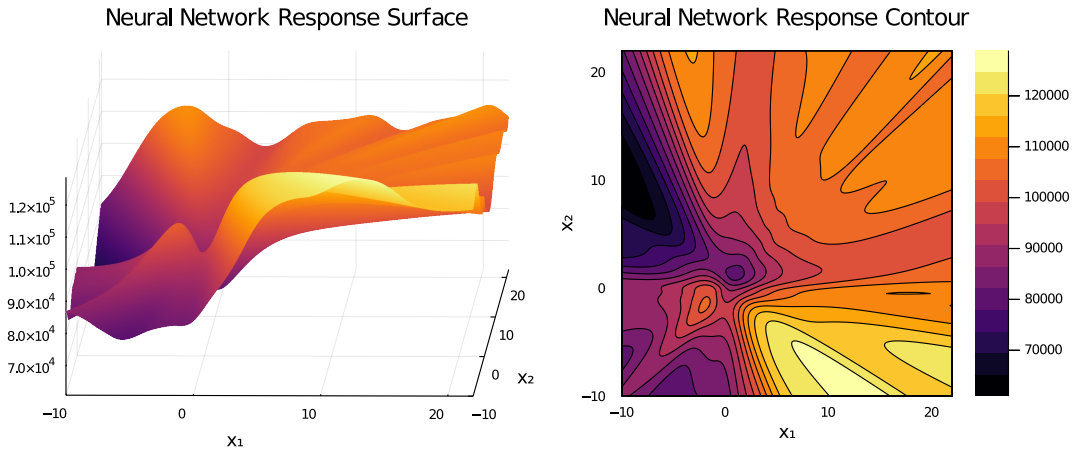

Figure 3: Neural network response surface over an extended domain

Because the neural surrogate model has a non-concave shape, the algorithm is not guaranteed to converge. The trajectories of the gradient ascent algorithm for different starting points can be seen in figure 4. We can see how, depending on the starting point, the optimization algorithms terminates in different points. From these results we cannot obtain a clear local optimum for neither our surrogate model, nor our simulation.

We believe that our simulation has a maximum, that is, an optimal configuration of parameters. This implies that the shape of our simulation is, at least in the neighborhood of the maximum, concave. But when plotting both the simulation and the neural network's surfaces, over the training domain, as seen in figure 1, we observe that this is clearly not the case.

Since the maximum is not in the training domain, this implies that the maximum must be outside. This hypothesis can be verified by analyzing more points with the simulator, in order to get a better idea of the response surface's shape. In figure 5 and 6 we can see a comparison of the actual surface and what the network predicts.

A possible solution would be to extend the domain of the training data. For example by examining all even points between $(0, 0)$ and $(20, 20)$. Then to perform the gradient ascent on the neural network trained on the extended training data.

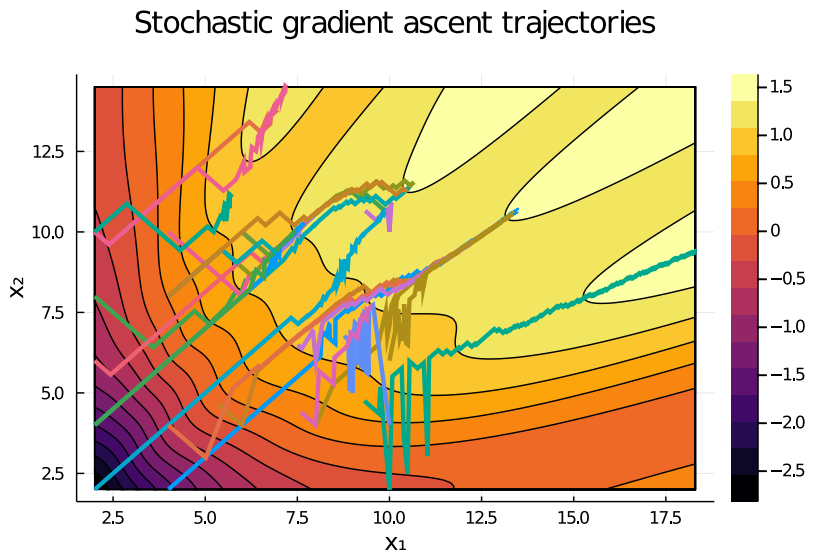

Figure 4: Trajectories of gradient ascent for various random starting points

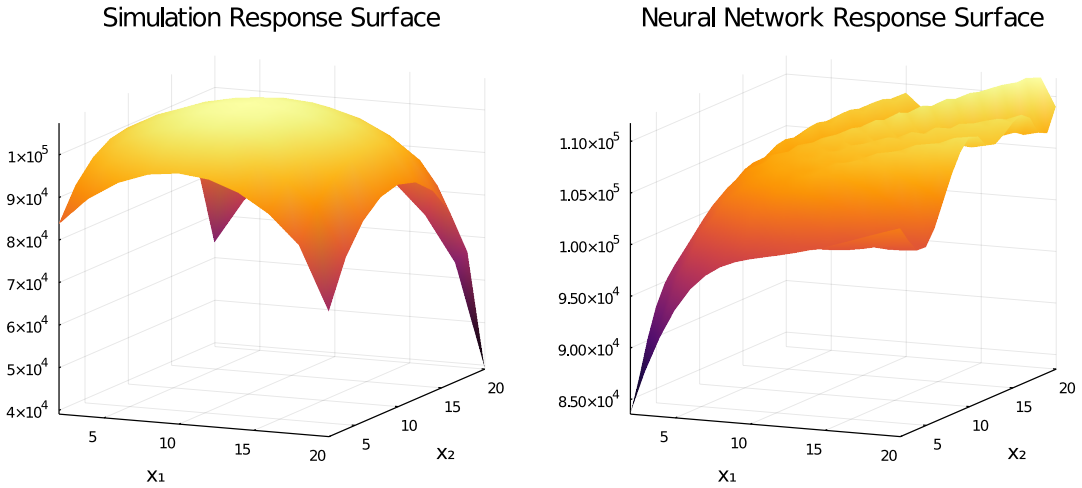

Figure 5: Response surfaces over an extended domain

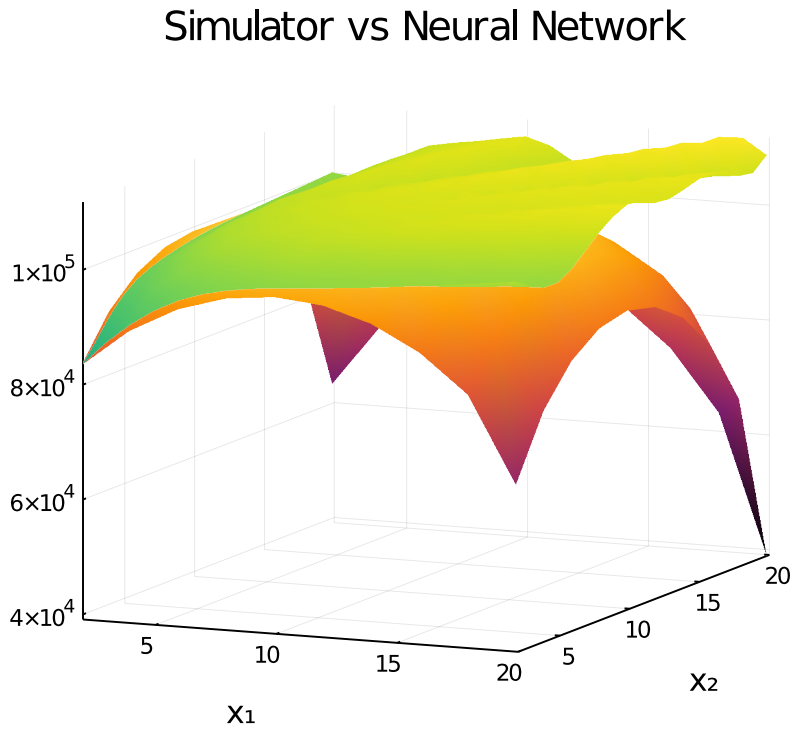

Figure 6: Superimposed response surfaces for the simulator and neural network