CIS600 HW1

Yuchen Wang
905508464

1.Citions

Most of my codes are modified from https://towardsdatascience.com/how-to-build-an-artificial-neural-network-from-scratch-in-julia-c839219b3ef8.

2. Data Set

The data set I chose is about sales data of mobile phones of various companies. The data contains battery_power(Total energy a battery can store in one time measured in mAh); blue(Has bluetooth or not); clock_speed(speed at which microprocessor executes instructions); dual_sim(Has dual sim support or not); fc(Front Camera megapixels); four_g(Has 4G or not); int_memory(Internal Memory in Gigabytes); m_dep (Mobile Depth in cm); mobile_wt(Weight of mobile phone) ….. and price_range(This is the target variable with value of 0(low cost), 1(medium cost), 2(high cost) and 3(very high cost))

2,000 rows × 21 columns (omitted printing of 13 columns)

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep
	Int64	Int64	Float64	Int64	Int64	Int64	Int64	Float64
1	842	0	2.2	0	1	0	7	0.6
2	1021	1	0.5	1	0	1	53	0.7
3	563	1	0.5	1	2	1	41	0.9
4	615	1	2.5	0	0	0	10	0.8
5	1821	1	1.2	0	13	1	44	0.6
6	1859	0	0.5	1	3	0	22	0.7
7	1821	0	1.7	0	4	1	10	0.8
8	1954	0	0.5	1	0	0	24	0.8
9	1445	1	0.5	0	0	0	53	0.7
10	509	1	0.6	1	2	1	9	0.1
11	769	1	2.9	1	0	0	9	0.1
12	1520	1	2.2	0	5	1	33	0.5
13	1815	0	2.8	0	2	0	33	0.6
14	803	1	2.1	0	7	0	17	1.0
15	1866	0	0.5	0	13	1	52	0.7
16	775	0	1.0	0	3	0	46	0.7
17	838	0	0.5	0	1	1	13	0.1

Therefore, the data can be treated as a 2-class classification problem with first class of low to medium price range (0 and 1 price range) and second class of high to very high cost (2 and 3 price range).

```
df = file |> Tables.matrix

mat_0 = df[df[:,21] .<= 1, :]
mat_1 = df[df[:,21] .>= 2, :]
```

I use 70% of the data to train the model and use the rest to do the test.

```
train_data = randsubseq(1:1000, 0.7)
train_df = vcat(mat_0[train_data, :], mat_1[train_data, :])

test_data = [i for i in 1:1000 if isempty(searchsorted(train_data, i))]
test_df = vcat(mat_0[test_data, :], mat_1[test_data, :])
```

3. Neural Network

My model use sigmoidal node functions as the activation function

```
function sigmoid(Z)
    A = 1 ./ (1 .+ exp.(.-Z))
    return (A = A, Z = Z)
end

function linear_forward(A, W, b)
    Z = (W * A) .+ b
    cache = (A, W, b)

    @assert size(Z) == (size(W, 1), size(A, 2))

    return (Z = Z, cache = cache)
end

function forward_activation(A_prev, W, b, activation_function="sigmoid")
    Z, linear_cache = linear_forward(A_prev, W, b)

    if activation_function == "sigmoid"
        A, activation_cache = sigmoid(Z)
    end

    cache = (linear_step_cache=linear_cache, activation_step_cache=activation_cache)

    @assert size(A) == (size(W, 1), size(A_prev, 2))

    return A, cache
end
```

Start with randomly initialized weights and bias

```
function initialise_model_weights(layer_dims, seed = 7)
    params = Dict()

    for l=2:length(layer_dims)
        params[string("W_", (l-1))] = rand(StableRNG(seed), layer_dims[l], layer_dims[l-1]) * sqrt(2 / layer_dims[l-1])
        params[string("b_", (l-1))] = zeros(layer_dims[l], 1)
    end

    return params
end
```

Using back-propagation to improve the model. sigmoidal node functions is also the activation function for back-propagation.

```
function sigmoid_backwards(∂A, activated_cache)
    s = sigmoid(activated_cache).A
    ∂Z = ∂A .* s .* (1 .- s)

    @assert (size(∂Z) == size(activated_cache))
    return ∂Z
end

function linear_backward(∂Z, cache)
    # Unpack cache
    A_prev , W , b = cache
    m = size(A_prev, 2)

    # Partial derivates of each of the components
    ∂W = ∂Z * (A_prev') / m
    ∂b = sum(∂Z, dims = 2) / m
    ∂A_prev = (W') * ∂Z

    @assert (size(∂A_prev) == size(A_prev))
    @assert (size(∂W) == size(W))
    @assert (size(∂b) == size(b))

    return ∂W , ∂b , ∂A_prev
end

function activation_backward(∂A, cache, activation_function="sigmiod")
    @assert activation_function ∈ ("sigmoid", "relu")

    linear_cache , cache_activation = cache

    if (activation_function == "relu")

        ∂Z = relu_backwards(∂A , cache_activation)
        ∂W , ∂b , ∂A_prev = linear_backward(∂Z , linear_cache)

    elseif (activation_function == "sigmoid")

        ∂Z = sigmoid_backwards(∂A , cache_activation)
        ∂W , ∂b , ∂A_prev = linear_backward(∂Z , linear_cache)

    end
```

4. Training

Train the model by repeating the experiment ten times, each time starting with a different set of randomly initialized weights;

```
X, y = make_blobs(10_000, 3; centers=2, as_table=false, rng=2020);
X = Matrix(X');
y = reshape(y, (1, size(X, 2)));
f(x) =  x == 2 ? 0 : x
y2 = f.(y);

# Input dimensions
input_dim = size(X, 1);

# Train the model
nn_results = train_network([input_dim, 5, 3, 1], X, y2; η=0.01, epochs=10, seed=1, verbose=true);

# Plot accuracy per iteration
p1 = plot(nn_results.accuracy,
        xlabel="Log",
        ylabel="Accuracy",
        title="Development of accuracy at each iteration");

# Combine accuracy and cost plots
plot(p1, layout = (2, 1), size = (800, 600))
```

✓ 15.1s

```
function train_network(layer_dims , DMatrix, Y;  η=0.001, epochs=10, seed=2020, verbose=true)
    # Initiate an empty container for cost, iterations, and accuracy at each iteration
    costs = []
    iters = []
    accuracy = []

    # Initialise random weights for the network
    params = initialise_model_weights(layer_dims, seed)

    # Train the network
    for i = 1:epochs

        Ŷ , caches  = forward_propagate_model_weights(DMatrix, params)
        cost = calculate_cost(Ŷ, Y)
        acc = assess_accuracy(Ŷ, Y)
        ∇  = back_propagate_model_weights(Ŷ, Y, caches)
        #each time starting with a different set of randomly initialized weights
        params = initialise_model_weights(layer_dims, seed)
        # params = update_model_weights(params, ∇, η)

        if verbose
            println("Iteration -> $i, Cost -> $cost, Accuracy -> $acc")
        end

        # Update containers for cost, iterations, and accuracy at the current iteration (epoch)
        push!(iters , i)
        push!(costs , cost)
        push!(accuracy , acc)
    end
        return (cost = costs, iterations = iters, accuracy = accuracy, parameters = params)
end
```

✓ 0.8s

5 Outcome

The training outcome is 50%. Seems not right…..