```julia
using LinearAlgebra
```

```julia
using Random
```

```julia
using Noise
```

inputs = []

```julia
# Define XOR points...
inputs = Any[]
```

outputs = []

```julia
# ... and corresponding labels
outputs = Any[]
```

n = 500000

```julia
# Define number of data points
n = 500000
```

```julia
for _ = 1 : n

    # define XOR input point
    xor_point = bitrand(2)

    # define the two cases in which XOR outputs 0, otherwise 1
    if (xor_point[1] == 1 && xor_point[2] == 1) || (xor_point[1] == 0 && xor_point[2] == 0)

        label = 0
    else

        label = 1
    end

    # add small gaussian noise
    dp = add_gauss(Float64.(xor_point), 0.05)

    # add to dataset
    push!(inputs, [dp; 1.0]) # absorb bias into datapoint
    push!(outputs, label)

end
```

sigmoid (generic function with 1 method)

```julia
# Activation function sigmoid
function sigmoid(z)
    return 1 / (1 + exp(-z))
end
```

relu (generic function with 1 method)

```
# Activation function ReLU
function relu(z)
    if z > 0
        return z
    else
        return 0
    end
end
```

MSEloss (generic function with 1 method)

```
# Mean-Squared Error loss function
function MSEloss(y_tilde, y)
    return (y_tilde - y)^2
end
```

feedforward (generic function with 1 method)

```
# Takes one data point, produces prediction based on current state of weights
function feedforward(x, W1, W2)

    # Repackage in desired way (make x the operator on vector w)
    X = [x' zeros(6)'; zeros(3)' x' zeros(3)'; zeros(6)' x']
    w1 = [W1[1, 1:3] ; W1[2, 1:3]; W1[3, 1:3]]

    # Feedforward starts here
    z1 = X * w1

    r = map(relu, z1)

    x2 = [r; 1]

    z2 = W2 * x2

    y_tilde = map(sigmoid, z2)

    return w1, X, z1, x2, z2, y_tilde

end
```

d_sigmoid (generic function with 1 method)

```
# Derivative of sigmoid activation function
function d_sigmoid(z)
    return sigmoid(z) * (1 - sigmoid(z))
end
```

d_relu (generic function with 1 method)

```
# Derivative of ReLU activation function
function d_relu(z)
    if z > 1
        return 1
    else
        return 0
    end
end
```

d_MSEloss (generic function with 1 method)

```
# Derivative of Mean-Squared Error loss function
function d_MSEloss(y, y_tilde)
    return 2 * (y_tilde - y)
end
```

backprop (generic function with 1 method)

```
function backprop(w1, W2, X, z1, x2, z2, y_tilde, y)

    dL_dz2 = d_MSEloss(y, y_tilde) * map(d_sigmoid, z2)

    # Compute update of W2:
    dL_dW2 = dL_dz2 * x2'

    # Compute update of W1:
    dL_dw1 = dL_dz2 * W2 * [I ; [0 0 0]] * diagm(map(d_relu, z1)) * X
    dL_dW1 = [dL_dw1[1:3]' ; dL_dw1[4:6]'; dL_dw1[7:9]']

    return dL_dW1, dL_dW2

end
```

```
let

    # Number of epochs (times to iterate over dataset)
    num_epochs = 10

    # Learning rate
    α = 0.05

    # Initialize weights
    W1 = add_gauss(zeros(3, 3), 0.5)
    W2 = add_gauss(zeros(1, 4), 0.5)

    # Record loss
    training_loss = Any[]

    function train()

        for _ = 1 : num_epochs
            for i = 1 : n

                dp = inputs[i]
                y = outputs[i]

                w1, X, z1, x2, z2, y_tilde = feedforward(dp, W1, W2)

                dL_dW1, dL_dW2 = backprop(w1, W2, X, z1, x2, z2, y_tilde[1], y)

                W1 -= α * dL_dW1
                W2 -= α * dL_dW2

                loss = MSEloss(y, y_tilde[1])
                push!(training_loss, loss)

            end
        end
    end

    function predict(x)
        _, _, _, _, _, pred = feedforward([x ; 1], W1, W2)

        if pred[1] >= 0.5
            return pred, 1
        else
            return pred, 0
        end
    end

    train()

    # Predict actual XOR points
    pred1, label1 = predict([0; 0])
    pred2, label2 = predict([1; 0])
    pred3, label3 = predict([0; 1])
    pred4, label4 = predict([1; 1])
```

```julia

    println(pred1, label1)
    println(pred2, label2)
    println(pred3, label3)
    println(pred4, label4)

```

```
[0.010927996838302154]0
[0.9898135233251393]1
[0.9998436152871761]1
[0.0003025395204761653]0
```