Training neural network for Assignment 4, LED segments

Table of Contents


```julia
begin
    using Plots

    # Packages for automatic differentiation and neural networks
    using Flux, Zygote
end
```

Define a generic NN layer

No changes here from the lecture notes

```julia
begin
    struct Layer
        W::Matrix{Float32} # weight matrix - Float32 for faster gradients
        b::Vector{Float32} # bias vector
        activation::Function
        Layer(in::Int64,out::Int64,activation::Function=identityFunction) =
            new(randn(out,in),randn(out),activation) # constructor
    end

    (m::Layer)(x) = m.activation.(m.W*x .+ m.b) # feed-forward pass
end
```

Define some required activation functions. `ReLu` is a standard neural netwrok activation function.

```julia
begin
    ReLu(x) = max(0,x)
    identityFunction(x) = x
end;
```

Define a network as a concatenation of many layers

No changes here from notes

```julia
begin
    struct Network
        layers::Vector{Layer}
        Network(layers::Vararg{Layer}) = new(vcat(layers...))
            # constructor - allow arbitrarily many layers
    end

    (n::Network)(x) = reduce((left,right)->right∘left, n.layers)(x)
        # perform layer-wise operations over arbitrarily many layers
end
```

Create a two-layer network

Input is numerical encoded segment values, see **inputs** matrice

Output is one-hot-encoded digits from 0-9, see **outputs** matrice

```
mse (generic function with 1 method)
```

```julia
begin
    #inputs = collect(-3:0.1:3) # create training data
    #targetOutput = sin.(inputs)

    inputs = [
    1 1 1 1 1 1 0;  # 0
    0 1 1 0 0 0 0;  # 1
    1 1 0 1 1 0 1;  # 2
    1 1 1 1 0 0 1;  # 3
    0 1 1 0 0 1 1;  # 4
    1 0 1 1 0 1 1;  # 5
    1 0 1 1 1 1 1;  # 6
    1 1 1 0 0 0 0;  # 7
    1 1 1 1 1 1 1;  # 8
    1 1 1 1 0 1 1;  # 9
    ]

    targetOutput = [
    1 0 0 0 0 0 0 0 0 0;  # 0
    0 1 0 0 0 0 0 0 0 0;  # 1
    0 0 1 0 0 0 0 0 0 0;  # 2
    0 0 0 1 0 0 0 0 0 0;  # 3
    0 0 0 0 1 0 0 0 0 0;  # 4
    0 0 0 0 0 1 0 0 0 0;  # 5
    0 0 0 0 0 0 1 0 0 0;  # 6
    0 0 0 0 0 0 0 1 0 0;  # 7
    0 0 0 0 0 0 0 0 1 0;  # 8
    0 0 0 0 0 0 0 0 0 1;  # 9
    ]
    mse(x,y) = sum((x .- y).^2)/length(x) # MSE will be our loss function
end
```

Create Nueral Net with 2 layers

Input layer is 7 neurons, first layer is 100 neurons

Second layer is 100 neurons and output layer is 10 neurons

```
Network([Layer(100×7 Matrix{Float32}:
                 1.22684    -1.22644    -2.72411    -1.65359    -0.790929    1.12893    -1
```

```julia
begin
    using Random
    Random.seed!(54321) # for reproducibility

    twoLayerNeuralNet = Network(Layer(7,100 ,ReLu), Layer(100,10))
        # instantiate a two-layer network
end
```

Train on Input data

- This involes 2 loops, one nested inside another
- The outer loop iterates through 'epochs'

The inner loop iterates 10 times per epoch, for each loop we iterate over 10 random digit patterns, so 'i' is a random value between 1-10 (not-recurring) for each loop. This is done to ensure the input digit (vector) is always random.

The outer loop runs 800 times, 800 epochs, inner loop 10 times per epoch

Have the following variables

- input - LED vector, the LED number
- output - One-hot-encoded digit
- prediction - 10 digit vector, each value is confidence the corresponding index is the predicted output for the LED input

Traning remains same as in notes, adjusting parameters based on the gradients, calculated by differentiating the mean squared error with respect to the paramters

For graphing have the the following arrays

- epochCount - for reference y axis later
- epochAccuracy - store accuracy (correct predictions out of 10) for each epoch
- lossCurve - store mean squared error

Accuracy is calculated by checking if predicted digit is equal to target digit, if so increment a counter 'correct' (this is done inside the inner loop). The value of 'correct' is divided by 10 and pushed to the epochAccuracy array

```julia
begin

    Flux.@functor Layer     # set the Layer-struct as being differentiable
    Flux.@functor Network   # set the Network-struct as being differentiable

    parameters = Flux.params(twoLayerNeuralNet)
        # obtain the parameters of the layers (recurses through network)

    optimizer = ADAM(0.001) # from Flux-library

    epochCount = []
    epochAccuracy = []
    lossCurve = []
    epochs = 800

    for epoch in 1:epochs
    correct = 0
    loss_count = 0

        for j in shuffle(1:10)
            # Select input-output pair
            input = inputs[j, :]
            output = targetOutput[j, :]

            #run the network based on the input
            prediction = twoLayerNeuralNet(input)
            loss = mse(prediction, output)

            #calculate the gradient
            gradients = Zygote.gradient(() -> mse(twoLayerNeuralNet(input), output),
                parameters)

            #adjust the parameters based on the gradient
            Flux.Optimise.update!(optimizer, parameters, gradients)


            loss_count += loss


            out_label = findmax(output)[2] - 1
            pred_label = findmax(prediction)[2] - 1
            if out_label == pred_label
                correct += 1
            end
        end


        push!(epochAccuracy, correct / 10)
        push!(lossCurve, loss_count / 10)
        push!(epochCount, epoch)
    end


end
```

Check tolerance

To ensure the elements of the output vector can be within a 0.1 tolerance of the target value I use monte carlo method.

A random digit from 0-9 is inputed into the model to get a prediction vector, the prediction vector values are checked to ensure they are within the tolerance. This is done 800 times

- Predicting confidence - highest value in vector, what network predicts digit is
- Non-predicting confidence - all other values in vector beside predicting confidence

Over 800 epochs, the highest (max) and lowest (min) values are recorded for the

- predicting confidence - should be between 0.9 and 1.1
- Non-predicting confidence - should be between -0.1 and 0.1

checkNonPrediction (generic function with 1 method)

```
function checkNonPrediction(prediction, pred_value, max_nonPred_confidence, min_nonPred_confidence)
    for i in 1:10
        if i != pred_value+1
            if prediction[i] > max_nonPred_confidence
                max_nonPred_confidence = prediction[i]
            elseif prediction[i] < min_nonPred_confidence
                min_nonPred_confidence = prediction[i]
            end
    end
    end
return (max_nonPred_confidence, min_nonPred_confidence)
end
```

```
begin
max_pred_confidence = 0
min_pred_confidence = 100

max_nonPred_confidence = 0
min_nonPred_confidence = 100
for i in 1:500
    pred_confidence_count = 0
    for j in shuffle(1:10)
        input = inputs[j, :]
        output = targetOutput[j, :]
        prediction = twoLayerNeuralNet(input)

        input_label = join(inputs[j , :], ", ")
        out_value = findmax(output)[2] - 1
        pred_value = findmax(prediction)[2] - 1

        pred_confidence = findmax(prediction)[1]

        #Check if confidence is between 0.1 tolerance, so between 0.9 and 1.1
        if pred_confidence > max_pred_confidence
            max_pred_confidence = pred_confidence
        elseif(pred_confidence < min_pred_confidence)
            min_pred_confidence = pred_confidence
        end
        tuple = checkNonPrediction(prediction, pred_value, max_nonPred_confidence,min_nonPred_confidence )

        max_nonPred_confidence = tuple[1]
        min_nonPred_confidence = tuple[2]


    end
    prediction_tolerance = (pred_confidence_count/10)

end
    println("Max prediction confidence: $max_pred_confidence\nMin prediction confidence: $min_pred_confidence\n")

    println("Max Non Prediction confidence: $max_nonPred_confidence\nMin Non Prediction confidence: $min_nonPred_confidence\n")

    println("Is value prediction outside tolerance: ",  (max_pred_confidence > 1.1 || min_pred_confidence < 0.9))
    println("Is non-value prediction outside tolerance: ",  (max_nonPred_confidence > 0.1 || min_nonPred_confidence < -0.1))
end
```

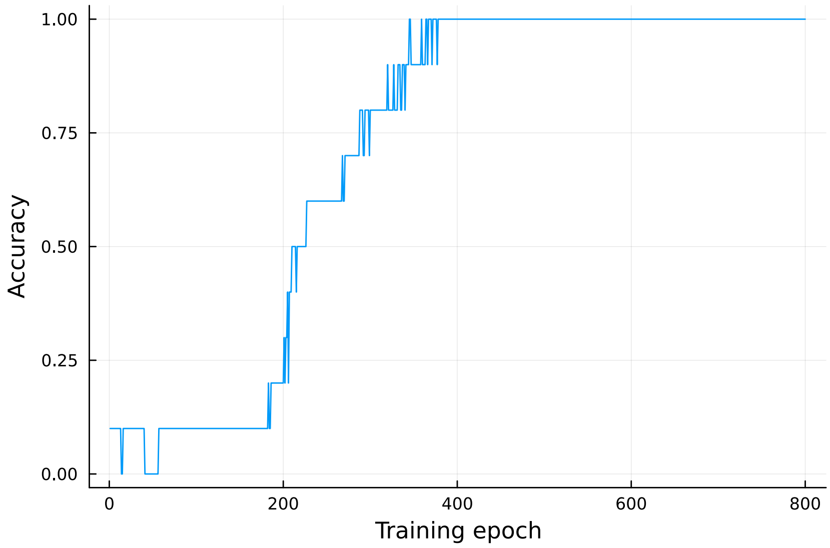
```
Max prediction confidence: 1.006978
Min prediction confidence: 0.9844361

Max Non Prediction confidence: 0.032113
Min Non Prediction confidence: -0.046803057

Is value prediction outside tolerance: false
Is non-value prediction outside tolerance: false
```

Plot the result

Accuracy over time

- X-axis - epochs, number of times trained model 10 times on random inputs 1-10, 800 times
- Y-axis - Accuracy, from 0-1

See how accuracy is stable at 1.0 at 400 epochs, around halfway, I think this ensures not under/over-confident predictions, hence stay within tolerance

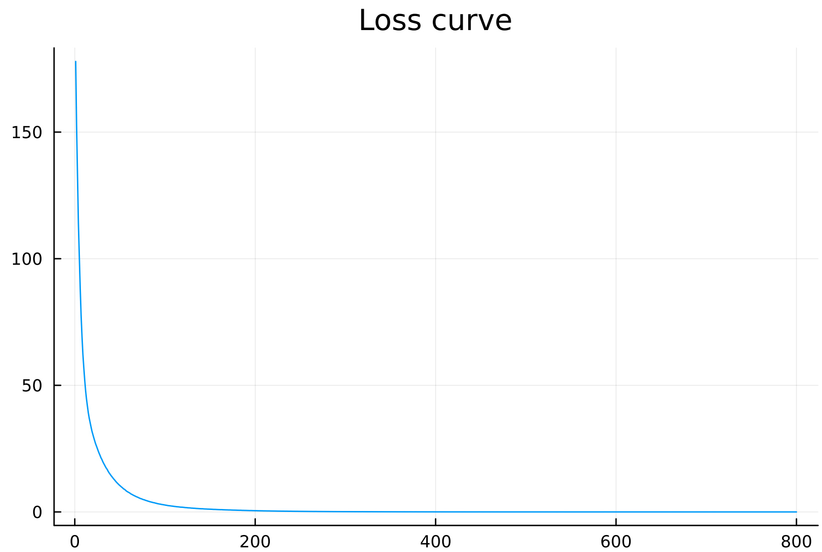



```
begin
    plot(epochCount, epochAccuracy, lab="")
    yaxis!("Accuracy")
    xaxis!("Training epoch")
end
```

Loss Curve

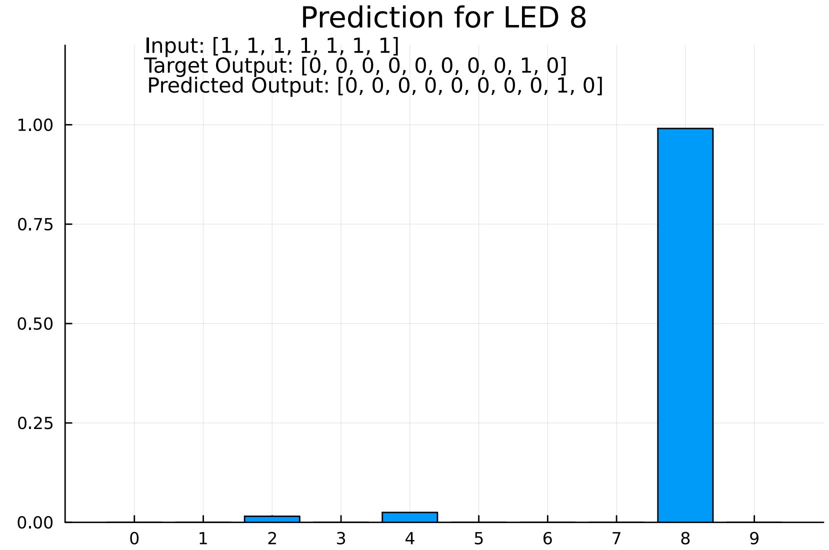
- X-axis - epochs
- Y-axis - Mean Squared Error

```
lossCurvePlot =
```




```
lossCurvePlot = plot(lossCurve, title = "Loss curve", legend=:none)
```

Bar Plot interaction




```julia
begin
    function get_prediction(digit)
        input = inputs[digit + 1, :]  # +1 why?? julia -1 indexed??
        return twoLayerNeuralNet(input)
    end

    function plot_prediction(digit)
        prediction = get_prediction(digit)

        input_str = join(inputs[digit + 1, :], ", ")
        target_output = join(targetOutput[digit + 1, :], ", ")
        predicted_output = join(targetOutput[findmax(prediction)[2], :], ", ")


        Plots.bar(0:9, prediction, title="Prediction for LED $digit",legend=false,
    xticks=0:9)
        plot!(ylim=(0, 1.2))
        annotate!(2, 1.2, text("Input: [$input_str]", 10))
        annotate!(3.2, 1.15, text("Target Output: [$target_output]", 10))
        annotate!(3.5, 1.1, text("Predicted Output: [$predicted_output]", 10))
        #why does x shift? 2 then 3.2 then 3.5???

    end

    plot_prediction(digit)
end
```

8

```julia
@bind digit Slider(0:9, show_value=true)
```