Ctrl + S

Training a simple neural network

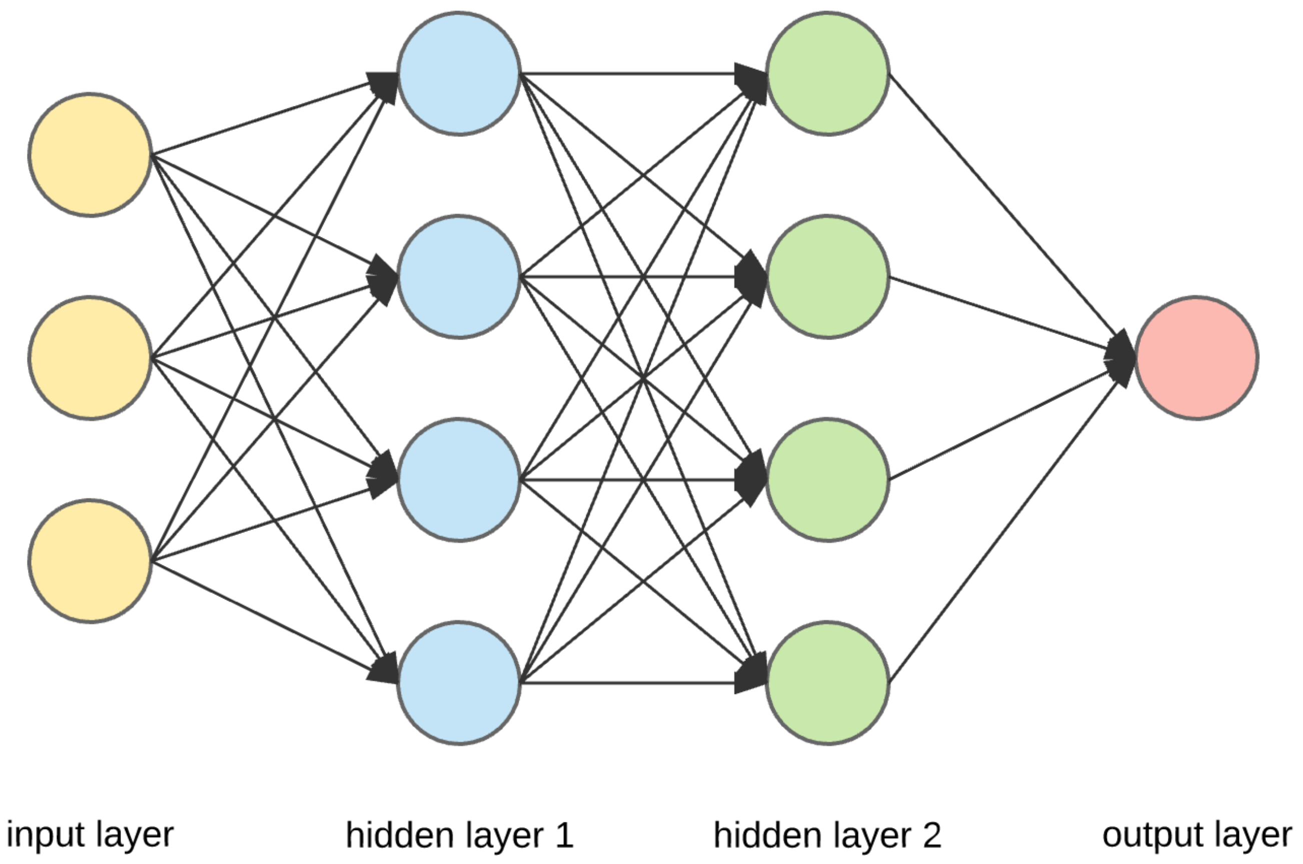


Table of Contents

```
begin
    using PlutoUI
    using Latexify
    TableOfContents()
end
```

```
Multiple definitions for Flux and Zygote
Combine all definitions into a single reactive cell using a `begin ... end` block.
```

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

Define a generic NN layer

Use a struct to define a generic layer of a neural network. The struct fields comprise:

- `W`: a weight matrix of floats connecting the layer's input to its output
- `b`: a vector of float biases which serve to modulate the default output of each neuron
- `activation`: an activation function that maps the layer's input to its output
- a contructor, which takes the input and output dimensions of the layer as parameters along with an activation function (default is the identity function) and randomly initialises the weights and biases

The final expression in the block below calls the `Layer` struct as a function with an input vector as an argument and returns an output vector, effectively it implements the feedforward step for the layer.

```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

Again, we use a struct to define a network comprising an arbitrary number of layers. We need just one field, `layers`, which is a vector of type `Layer`. The constructor takes a variable number of `Layer` arguments and assigns them to the `layers` field.

The final function definition in the block serves to propagate its vector argument `x` through the entire network. For clarity, the expression `reduce((left,right)->right∘left, m.layers)(x)` can be broken down into a number of elements:

- `right∘left` represents the composition of the function `right` and `left`. So `(right∘left)(x)` is equivalent to `right(left(x))`.
- `(left, right)->right∘left` is an anonymous function taking two arguments: `left` and `right`, which executes a composition of its functional arguments
- `reduce` is a function that applies its first argument, a function, to the elements of a collection. In this case the layers of the network, which is equivalent to `right(left(x))`.

```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
```

```julia
reduce((x,y)->x*y, [1,2,3,4]);
```

```julia
begin
    f(x) = x^2
    g(x) = x+1

    (f∘g)(3), f(g(3))
end;
```

Create a two-layer network

Define a neural network with two hidden layers of a 100 neurons each, and one input and one output unit.

The 1st hidden layer uses a rectified linear unit (ReLU) activation function, and the 2nd uses the default identity function.

```
Network([Layer(100×1 Matrix{Float32}:, [   more], ReLu), Layer(1×100 Matrix{Float32}:
                 1.22684                                        -0.683266  1.58826  -0.3
```

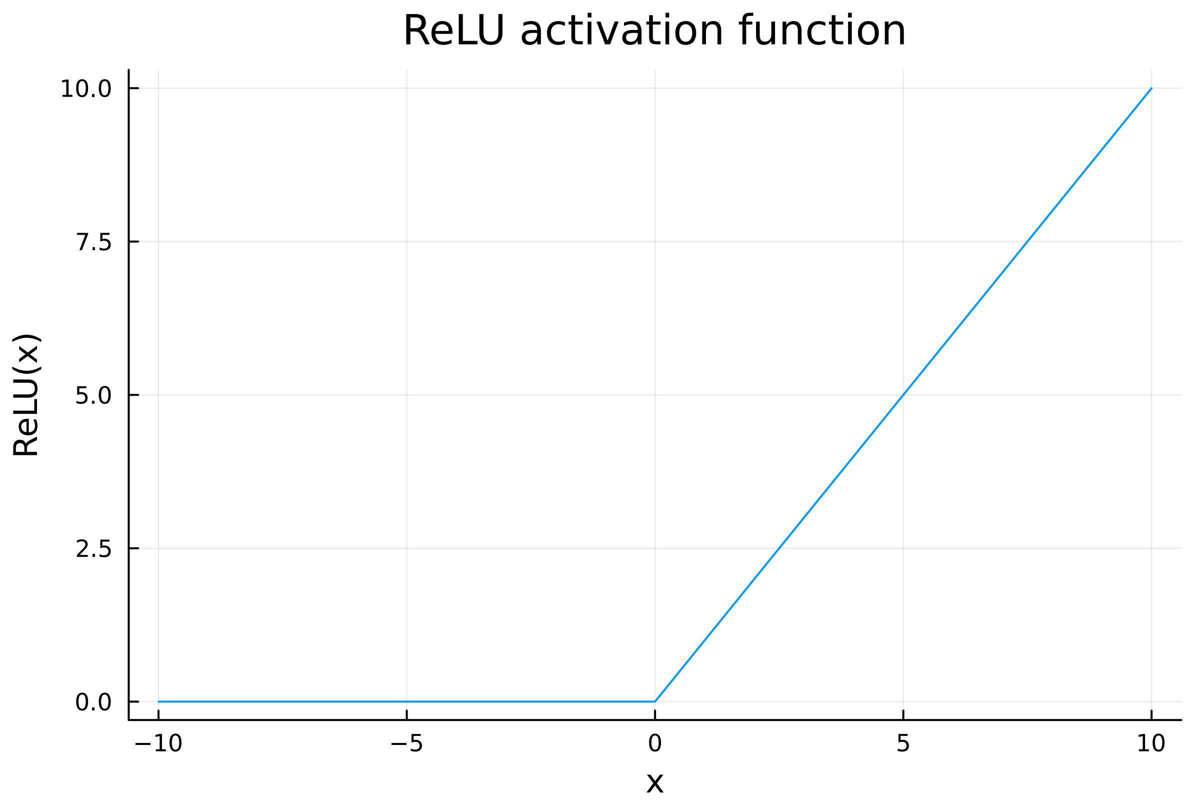
```
begin
    inputs = [
        1 1 1 1 1 1 0;
        0 1 1 0 0 0 0;
        1 1 0 1 1 0 1;
        1 1 1 1 0 0 1;
        0 1 1 0 0 1 1;
        1 0 1 1 0 1 1;
        1 0 1 1 1 1 1;
        1 1 1 0 0 0 0;
        1 1 1 1 1 1 1;
        1 1 1 1 0 1 1] #Training Data

    targetOutput = [
        1 0 0 0 0 0 0 0 0 0;
        0 1 0 0 0 0 0 0 0 0;
        0 0 1 0 0 0 0 0 0 0;
        0 0 0 1 0 0 0 0 0 0;
        0 0 0 0 1 0 0 0 0 0;
        0 0 0 0 0 1 0 0 0 0;
        0 0 0 0 0 0 1 0 0 0;
        0 0 0 0 0 0 0 1 0 0;
        0 0 0 0 0 0 0 0 1 0;
        0 0 0 0 0 0 0 0 0 1]

    mse(x,y) = sum((x .- y).^2)/length(x) # MSE will be our loss function

    using Random
    Random.seed!(54321) # for reproducibility

    twoLayerNeuralNet = Network(Layer(7,100,ReLu), Layer(100,10))
        # instantiate a two-layer network, takes in 7's gives 0-10 vals
end
```



The error (or loss) function is the mean squared difference between the actual output and target output.

$$\mathrm{mse}\left(x, y\right) = \frac{\sum\left(x - y\right)^{2}}{\mathrm{length}\left(x\right)}$$

Train on 7-segment digit representations

The task

-The 10 digits 0 to 9 can be represented on an LED using the above pattern of segments. -Your task is to train a neural network to take a binary representation of the display as input and produce the number it represents as output. For example, the number "1" can be represented as the seven-element integer vector [0 1 1 0 0 0 0], "2" as [1 1 0 1 1 0 1], and so on. -The output classification can be represented as a 10-element vector, such that the input representing "2" will produce the output vector [0 0 1 0 0 0 0] and a "3" input will give an output of [0 0 0 1 0 0 0]. -Note that the elements of the output vector can be within a 0.1 tolerance of the target value.

You will need to:

- (1) Modify the network architecture to allow a 7-bit input and a 10-bit output;
- (2) Modify the training loop so that the 10 vectors representing each LCD display are randomly presented during a given training iteration;
- (3) Note that the shape of both the input and output data is different from the sine example used in the original notebook, so the training code will need to be modified to deal with this;
- (4) Create a visualisation of the network's performance using the HTML controls

feature.

Have to randomly sample the inputs

We use the Flux library functions to calculate the relevant gradients for the Layer and Network structs.

- We first extract the trainable parameters (weights and biases) from the network
- We assign an optimiser, ADAM, which adjusts the rate at which we change these parameters
- We set up vectors to log the training performance
- Then we iterate over the training set, adjusting the weights after each iteration with repeated calls to Zygote.gradient followed by weight updates

Multiple definitions for Flux and Zygote

Combine all definitions into a single reactive cell using a `begin ... end` block.

```julia
begin

    using Flux, Zygote

    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 the Flux-Library

    ##Added netOutput

    netOutput = [] #store output for plotting
    lossCurve = [] #store loss for plotting

    for i in 1:500
        # Randomly iterate over the 10 digit patterns
        for j in shuffle(1:10)
            # Calculate the grandients for the network parameters
            gradients = Zygote.gradient(()
            -> mse(twoLayerNeuralNet(inputs[j, :], targetOutput[j, :]), parameters))

            #Update the parameters using the gradients and optimiser settings
            Flux.Optimise.update!(optimizer, parameters, gradients)

            #Log the performance for later plotting
            actualOutput = twoLayerNeuralNet(inputs[j, :])[:]
            push!(netOutput, actualOutput)
            push!(lossCurve, mse(actualOutput, targetOutput[j, :]))
        end
    end
end
```

Plot the result

```
UndefVarError: `inputs` not defined
```

```julia
begin
    #using Plots

    outputPlot = scatter(inputs, targetOutput,
        title = "Neural Network fit", label = "Data points", legend=:topleft
    )

    plot!(outputPlot, inputs, netOutput[plotIndex],
        label = "Learned function", lw = 4, color = :red
    )

    annotate!(3.0, -0.75, text(plotIndex, :blue, :right, 15))

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

    plot(outputPlot)
end
```

Move the slider to observe training progress.

```julia
@bind plotIndex Slider(1:10:1000, default=1000)
```

```
UndefVarError: `lossCurvePlot` not defined
```

```julia
plot(lossCurvePlot)
```