Training a simple neural network

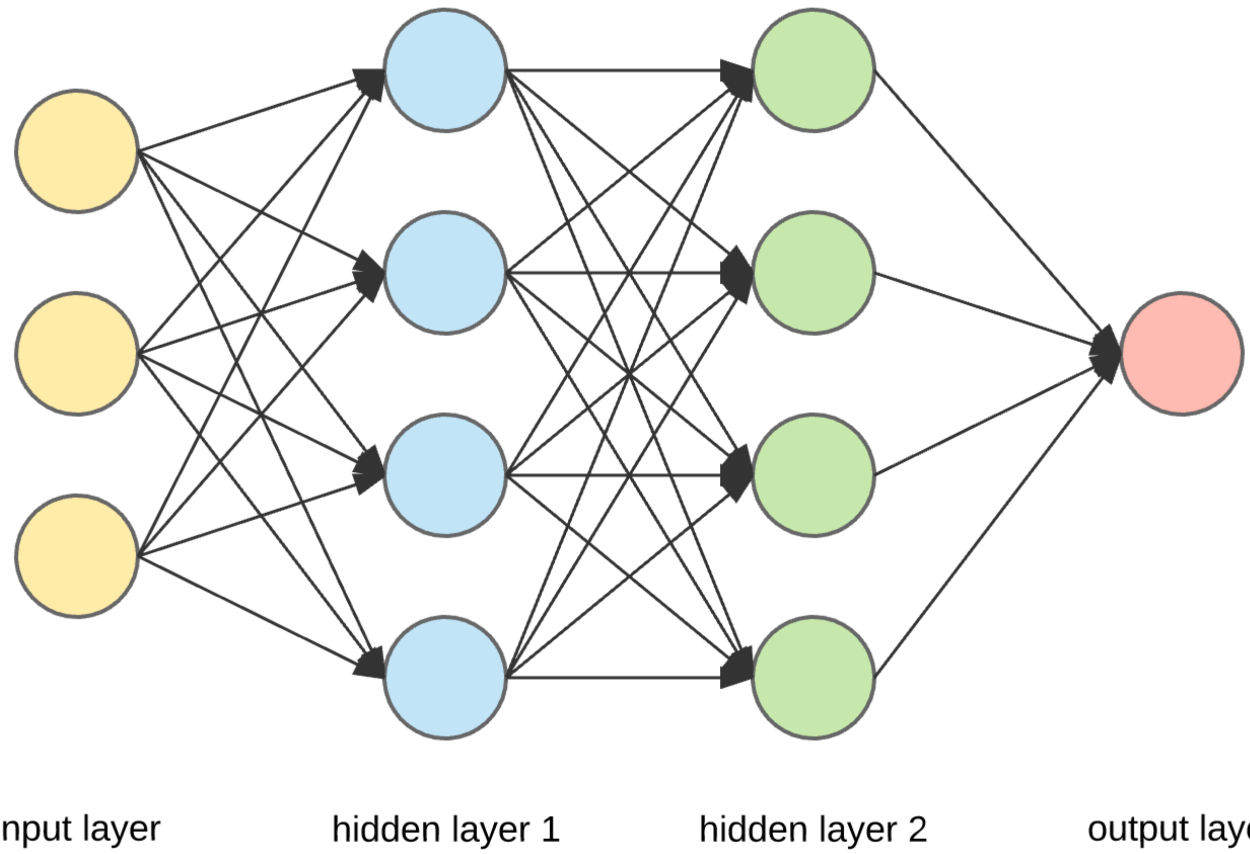


Table of Contents


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

```julia
begin
    using Plots

    # Packages for automatic differentiation and neural networks
    using Flux, Zygote
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 constructor, which takes the input and output dimensions of the layer as parameters along with an activation function (default is the identity function) and randomly initializes 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 network 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))`.

```
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
```

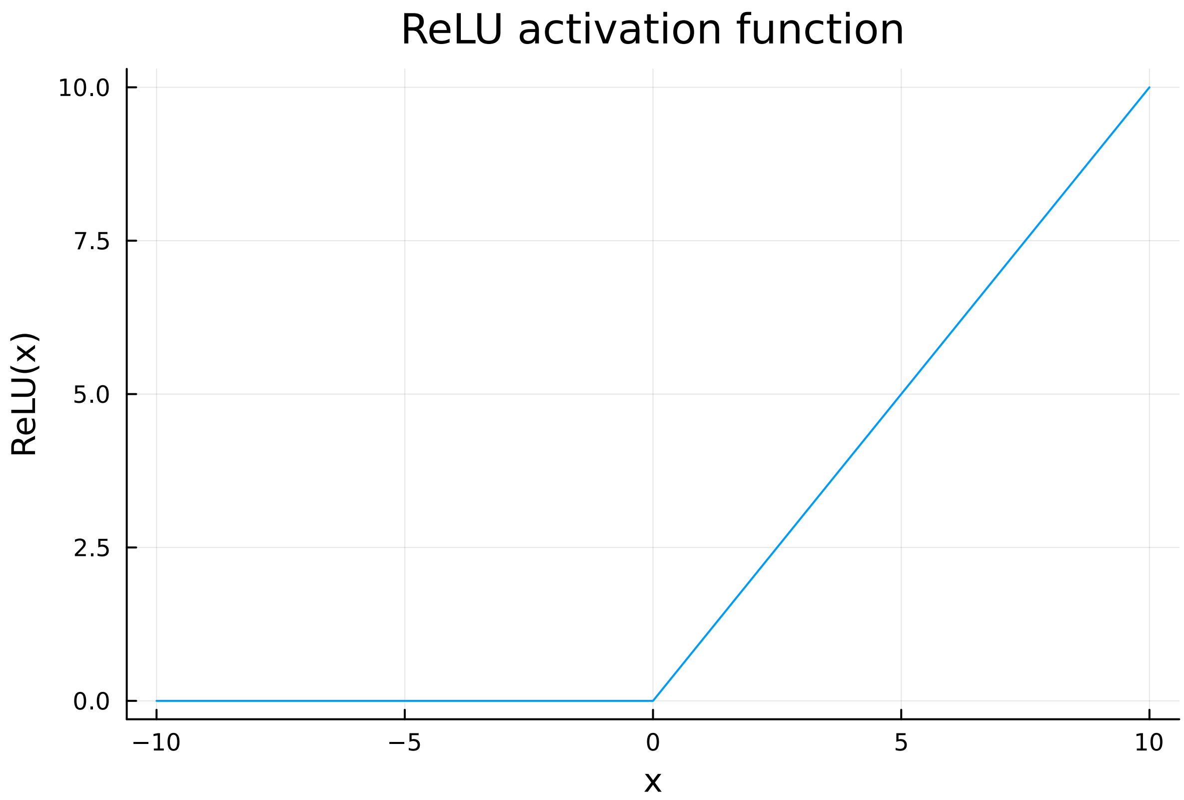
Create a two-layer network

Define a neural network with two hidden layers of 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(100×7 Matrix{Float32}:
              1.22684     -1.22644    -2.72411    -1.65359     -0.790929     1.12893    -
```

```julia
begin
    inputs = rand([-1, 1], (7, 10)) # 7-bit input, 10 examples
    targetOutput = rand([-1, 1], (10, 10)) # 10-bit output, 10 examples

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

    Random.seed!(54321) # for reproducibility

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



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

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

Train on random data

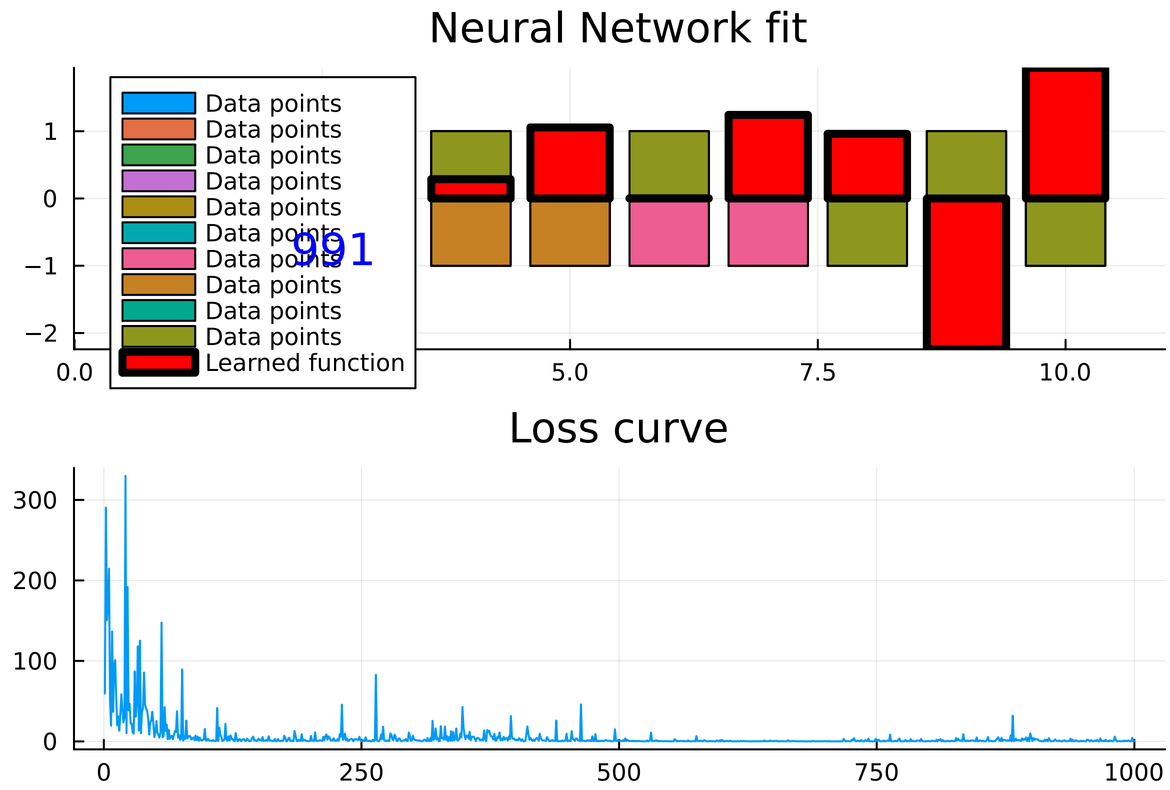
We use the Flux library functions to calculate the relevant gradients for the `Layer` and `Network` structs.

- 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.

```
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(0.05) # from Flux-library

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

    for i in 1:1000
        # Randomly select one of the 10 vectors for each training iteration
        random_index = rand(1:10)
        single_input = inputs[:, random_index]
        single_output = targetOutput[:, random_index]

        # Calculate the gradients for the network parameters
        gradients = Zygote.gradient(
            () -> mse(twoLayerNeuralNet(single_input), single_output),
            parameters
        )

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

        # Log the performance for later plotting
        actualOutput = twoLayerNeuralNet(single_input)
        push!(netOutput, actualOutput)
        push!(lossCurve, mse(actualOutput, single_output))
    end
end
```

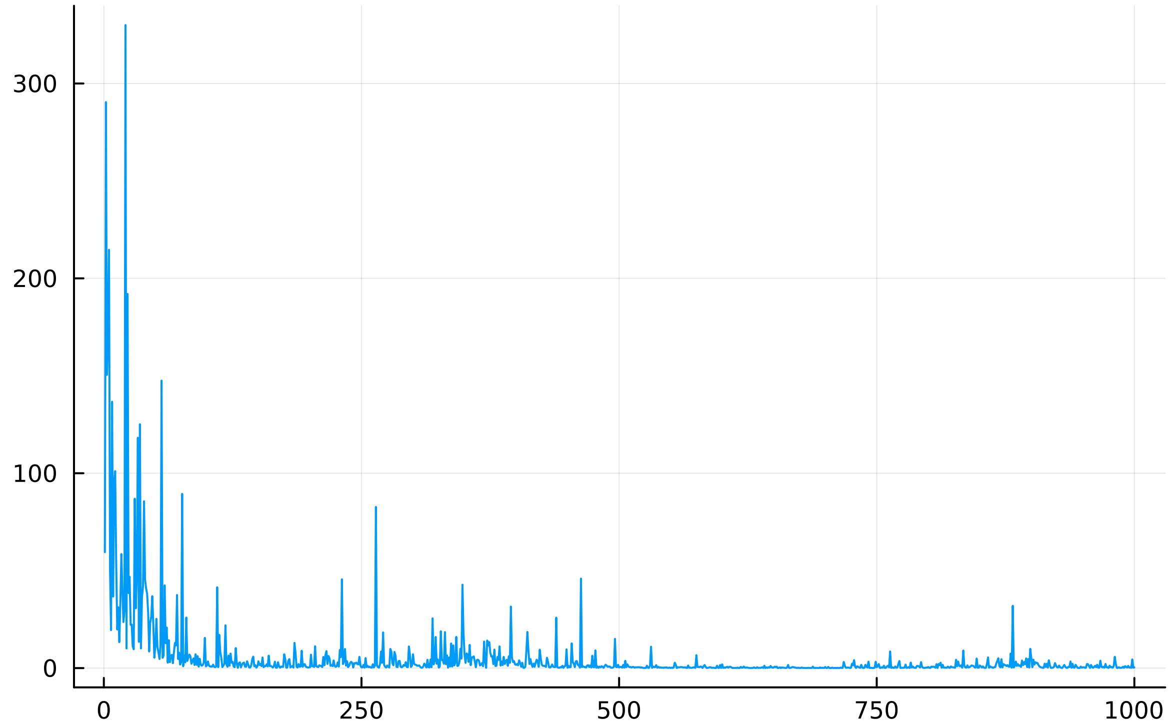
Visualize Network's Performance

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

```




```julia
begin
 #    outputPlot = scatter(1:10, targetOutput,
 #        title = "Neural Network fit", label = "Data points", legend=:topleft
 #    )

 #    plot!(outputPlot, 1:10, 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)

    # Bar plot for the data points
    outputPlot = bar(1:10, targetOutput,
        title = "Neural Network fit", label = "Data points", legend=:topleft
    )

    # Bar plot for the learned function
    bar!(outputPlot, 1:10, netOutput[plotIndex],
        label = "Learned function", lw = 4, color = :red
    )

    # Annotation
    annotate!(outputPlot, 3.0, -0.75, text(string(plotIndex), :blue, :right, 15))

    # Line plot for the loss curve
    lossCurvePlot = plot(lossCurve, title = "Loss curve", legend=:none)

    # Combine plots
    plot(outputPlot, lossCurvePlot, layout = (2, 1))
end
```

Loss curve




```
plot(lossCurvePlot)
```