Backpropagation in Practice: Doing even more explorations

Here we will do some exploring based on **our third lecture**

Getting all the Packges we will need

```
begin
    using StatsFuns
    using Plots
    using Random
    using LinearAlgebra
    using PlutoUI
    using BenchmarkTools
    using Flux
end
```

Code to Build the NN

We have made a type called Weights which is a mutable struct. This struct is going to store all the weights of our NN.

To generalize this solution we can define a function to instantiate the Weights (constructor) object where the user can define the number of desired layers.

For this assignment, we are going to hard code the number of weight vectors assuming an NN made of 3 layers (two hidden layers and an output layer)

```
mutable struct Weights
    W::Array{Float64}
    V::Array{Float64}
    U::Array{Float64}
end
```

We defined a function to instantiate the weights (a constructor). the constructor takes two partameters, the number of first hidden layer nodes and the second hidden layer nodes

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

```
init_weights(d_h1, d_h2) = Weights(randn(d_h1, 2), randn(d_h2, d_h1 + 1), randn(1, d_h2 + 1))
```

We recylce the logistic function for our notation

```
σ = logistic (generic function with 2 methods)
```

```
σ=StatsFuns.logistic
```

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

```
σ̇(x)=σ.(x) .* (1 .- σ.(x))
```

We modified the forwardProp function to move the loop that iterates over the vector z (or x in this document) out of the function.

This choice was made to allow the author to do some structural changes that they thought made the code more modular and easy to understand

```
forwardProp (generic function with 2 methods)
```

```
# Forward propagation
function forwardProp(z::Float64, weights::Weights)

    x̄ = weights.W * [1,z...]
    x = σ.(x̄)
    pushfirst!(x, 1.)
    ȳ = weights.V * x
    y = σ.(ȳ)
    pushfirst!(y, 1.)
    o = weights.U * y

    return o, y, x, ȳ, x̄
end
```

We are going to utilize multiple dispatch to define another forwardProp function that takes a vector of floats for z instead of a float. This function is going to be used to facilitate evaluation later on.

```
forwardProp (generic function with 2 methods)
```

```
function forwardProp(z::Vector{Float64}, weights::Weights)
    o = []

    for z_ in z
        o_, _, _, _, _ = forwardProp(z_, weights)
        push!(o, o_[1])
    end

    return o
end
```

Our first implementation of $backProp$ is based on matrix operations instead of using loops to compute the gradients

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

```
function backProp(z::Float64, o::Array{Float64}, t::Float64,
        y::Array{Float64}, x::Array{Float64}, ȳ::Array{Float64},
        x̄::Array{Float64}, weights::Weights)

    δₒ = o - [t]
    ∂E_∂U = reshape(δₒ, length(δₒ), 1) * transpose(y)

    δₕ₂  = Diagonal(ȯ(ȳ)) * (transpose(weights.U) * δₒ)[2:end]
    ∂E_∂V = reshape(δₕ₂, length(δₕ₂), 1) * transpose(x)

    δₕ₁ = Diagonal(ȯ(x̄)) * (transpose(weights.V) * δₕ₂)[2:end]
    ∂E_∂W = reshape(δₕ₁, length(δₕ₁), 1) * transpose([1,z...])

    return ∂E_∂W, ∂E_∂V, ∂E_∂U
end
```

Our second implementation of $backProp$ is based on the same matrix operations method but it utilizes julia's $broadcast$ capabilities.

We will call this implementation $backPropBroadcast$

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

```
function backPropBroadcast(z::Float64, o::Array{Float64}, t::Float64,
        y::Array{Float64}, x::Array{Float64}, ȳ::Array{Float64},
        x̄::Array{Float64}, weights::Weights)

    δₒ = o - [t]
    ∂E_∂U = δₒ .* transpose(y)

    δₕ₂  = Diagonal(ȯ(ȳ)) * (transpose(weights.U) * δₒ)[2:end]
    ∂E_∂V = δₕ₂ .* transpose(x)

    δₕ₁ = Diagonal(ȯ(x̄)) * (transpose(weights.V) * δₕ₂)[2:end]
    ∂E_∂W = δₕ₁ .* transpose([1,z...])

    return ∂E_∂W, ∂E_∂V, ∂E_∂U
end
```

Our third implementation of $backProp$ uses loops to compute the gradients with no matrix operations at all.

We will call this implementation $backPropLong$

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

```
function backPropLong(z::Float64, o::Array{Float64}, t::Float64,
        y::Array{Float64}, x::Array{Float64}, ȳ::Array{Float64},
        x̄::Array{Float64}, weights::Weights)

    δₒ, ∂E_∂U = o - [t], zeros(size(weights.U))
    δₕ₂, ∂E_∂V = zeros(size(weights.V, 1)), zeros(size(weights.V))
    δₕ₁, ∂E_∂W = zeros(size(weights.W, 1)), zeros(size(weights.W))

    for i in 1:size(weights.U,1), j in 1:size(weights.U,2)
```

```
            ∂E_∂U[i,j] += δₒ[i]*y[j]
        end

        _temp_arg = 0
        for i in 1:size(weights.V,1)
            for j in 1:length(δₒ)
                _temp_arg += δₒ[j]*weights.U[j, i+1]
            end
            δₕ₂[i] += σ̇(ȳ[i])*_temp_arg
        end

        for i in 1:size(weights.V,1), j in 1:size(weights.V,2)
            ∂E_∂V[i,j] += δₕ₂[i]*x[j]
        end

        _temp_arg = 0
        for i in 1:size(weights.V,1)
            for j in 1:length(δₕ₂)
                _temp_arg += δₕ₂[j]*weights.V[j, i+1]
            end
            δₕ₁[i] += σ̇(x̄[i])*_temp_arg
        end

        for i in 1:size(weights.W,1), j in 1:size(weights.W,2)
            ∂E_∂W[i,j] += δₕ₁[i]*[1,z...][j]
        end

        return ∂E_∂W, ∂E_∂V, ∂E_∂U
    end
```

Code to Conduct Training and Evaluation

We are going to implement batch learning through our $batchLearn$ function. We should recall that our weights constructor handled the number of nodes in our two hidden layers. in this function we are going to infer these values from the sizes of our weights, which are parameters to this function.

This function also takes z (training dataset), t (target dataset), bp_fun (our choice of $backProp$ implementations), and η (learning rate) as parameters.

In this function, we can observe the implications of our choice to modify $forwardProp$ earlier. It made it possible for us to make this for loop which is going to simply call $forwardProp$ and then bp_fun consecutively on each data point in the batch. and then update the wieghts once for the batch.

The change allowed us to encapsulate both the forward and back propagation steps in the same for loop

The function calculates the cost which is chosen to be MSE for this application. Taking into

consideration that calculating $RMSE$ is more computationally expensive and both are equivalent for our application in practice.

The function returns the updated $weights$ after the batch is done and also the $cost$ accumulated during the training process.

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

```
MSE(x, y) = (y- x)^2
```

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

```
function batchLearn(z::Array{Float64}, t::Array{Float64},
        bp_fun::Function, η::Float64, weights::Weights)

    cost = 0
    ∂E_∂W = zeros(size(weights.W,1), size(weights.W,2))
    ∂E_∂V = zeros(size(weights.V,1), size(weights.V,2))
    ∂E_∂U = zeros(size(weights.U,1), size(weights.U,2))
    lenBatch = length(z)

    for i in 1:lenBatch
        oₜₑₘₚ, yₜₑₘₚ, xₜₑₘₚ, ȳₜₑₘₚ, x̄ₜₑₘₚ = forwardProp(z[i], weights)
        ∂E_∂Wₜₑₘₚ, ∂E_∂Vₜₑₘₚ, ∂E_∂Uₜₑₘₚ = bp_fun(z[i], oₜₑₘₚ, t[i], yₜₑₘₚ, xₜₑₘₚ,
ȳₜₑₘₚ, x̄ₜₑₘₚ, weights)
        ∂E_∂W .+= ∂E_∂Wₜₑₘₚ
        ∂E_∂V .+= ∂E_∂Vₜₑₘₚ
        ∂E_∂U .+= ∂E_∂Uₜₑₘₚ
        cost += MSE(oₜₑₘₚ[1], t[i])
    end

    weights.W -= η .* ∂E_∂W
    weights.V -= η .* ∂E_∂V
    weights.U -= η .* ∂E_∂U

    return weights, cost
end
```

Next, we defined $trainOneEpoch$ to iteratively call $batchLearn$, after specifying a certain batch size, until the training dataset is exhausted. Therefore, completing an epoch

This function takes z (training dataset), t (target dataset), $batchSize$ (batch size), bp_fun (our choice of $backProp$ implementations), η (learning rate) and $weights$ as parameters. all of these parameters, except $batchSize$ are passed `as is` to $batchLearn$

the function returns the updated $weights$ after the epoch is done and also the $cost$ accumulated during the training process.

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

```
function trainOneEpoch(z::Array{Float64}, t::Array{Float64},
        batchSize::Int64, bp_fun::Function, η::Float64,
        weights::Weights)

    cost = 0
```

```
    lenData = length(z)
    curBatchStart = 1

    curBatchSize = batchSize
    while curBatchStart < lenData
        batchInputs = z[curBatchStart:curBatchStart + curBatchSize]
        batchTargets = t[curBatchStart:curBatchStart + curBatchSize]
        weights, cost = batchLearn(batchInputs, batchTargets, bp_fun, η, weights)
        cost += cost
        curBatchStart += curBatchSize + 1

        if (curBatchStart + batchSize) > lenData
            curBatchSize = lenData - curBatchStart
        end

    end

    cost /= lenData

    return weights, cost
end
```

Finally, we have defined the $train$ function which instantiates the weights randomly using our previously defined constructor, Then calls $trainOneEpoch$ for the defined number of epochs.

This function takes z (training dataset), t (target dataset), $noEpochs$ (number of epochs), $batchSize$ (batch size), bp_fun (our choice of $backProp$ implementations), η (learning rate), d_h1 (number of first hidden layer nodes) and d_h2 (number of second hidden layer nodes)

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

```
function train(z::Array{Float64}, t::Array{Float64},
        noEpochs::Int64, batchSize::Int64, bp_fun::Function,
        η::Float64, d_h1::Int64, d_h2::Int64)

    weights::Weights=init_weights(d_h1, d_h2)
    cost_hist = zeros(noEpochs)

    for epochNo in 1:noEpochs
        weights, cost = trainOneEpoch(z, t, batchSize, bp_fun, η, weights)
        cost_hist[epochNo] = cost
        # println("Epoch Number: $epochNo, Total Error: $cost")
    end

    return weights, cost_hist
end
```

We again utilize julia's multiple dispatch concept to define two implementations of the $predict$ function, to predict the corresponding value for either a single datapoint (float) or a dataset (array of floats) depending on the type of evaluation needed.

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

```
function predict(z::Vector{Float64}, weights::Weights)

    o = forwardProp(z, weights)
```

```
    return o
end
```

```
predict (generic function with 2 methods)
```

```
function predict(z::Float64, weights::Weights)

    o, _, _, _, _ = forwardProp(z, weights)
    return o[1]
end
```

Code to Plot the Results!

We defined the $addplot!$ which modifies an existing plot in-place and adds a linear plot based on an NN's predictions. Displaying a given label.

```
addplot! (generic function with 1 method)
```

```
function addplot!(testSet::Vector{Float64}, weights::Weights, label::String)
    plot!(testSet, predict(testSet, weights), label=label)
end
```

We defined the $initPlot$ Which creates the base plot showing the underlying function and a sample of noisy input that was used for training.

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

```
function initPlot(f::Function, x::Vector{Float64}, t::Vector{Float64}, title::String)
    graph = plot(f, label="Original Function", title=title)
    scatter!(x, t, label="Noisy Input")
    return graph
end
```

We defined $comparisonPlots!$, a function that iteratively calls $addplot!$ for a number of NNs and displays them with appropriate labels.

```
comparisonPlots! (generic function with 1 method)
```

```
function comparisonPlots!(f::Function, x::Vector{Float64}, t::Vector{Float64},
        testSet::Vector{Float64}, weightsArray::Vector{Weights},
        labelsArray::Vector{String}, title::String)
    graph = initPlot(f, x, t, title)
    for i in 1:length(weightsArray)
        weights, label = weightsArray[i], labelsArray[i]
        plot!(testSet, predict(testSet, weights), label=label)
    end
    return graph
end
```

We also defined the $learningPlots$ to plot the learning curves for any number of NNs to compare them.

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

```
function learningPlots(costHistArray::Vector{Vector{Float64}},
        labelsArray::Vector{String}, title::String)
    graph = plot(title=title, xlabel = "Epoch", ylabel = "MSE")
    for i in 1:length(costHistArray)
        costHist, label = costHistArray[i], labelsArray[i]
        plot!(1:length(costHist), costHist, label=label)
    end
    return graph
end
```

Assignment Questions

Question 1

Implement `backprop!`, `backprop_long!`, `backprop_broadcast!`, and `train!`. Show that they work propelry via a plot the showing how they generalize.

This is the code from the original assignment document to initiate the underlying function the noisy t dataset.

```
begin
    Random.seed!(132)
    x = rand(-5:0.01:5,20)
    t = f.(x) + randn(length(x))
end;  #The semicolum here just supresses the output
```

We define a simple quadratic function $f(x)$ that we will use to sample from to train our neural network

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

```
f(x) = x^2 + 2x +1
```

We generate 12 random samples over the range (-5,5), for each sample we get the output of our $f(x)$ and add a little bit of Gaussian noise

Below we are going to plot the results of three NNs, trained with our three implementations of $backProp$. These NNs are all going to use the following parameters:

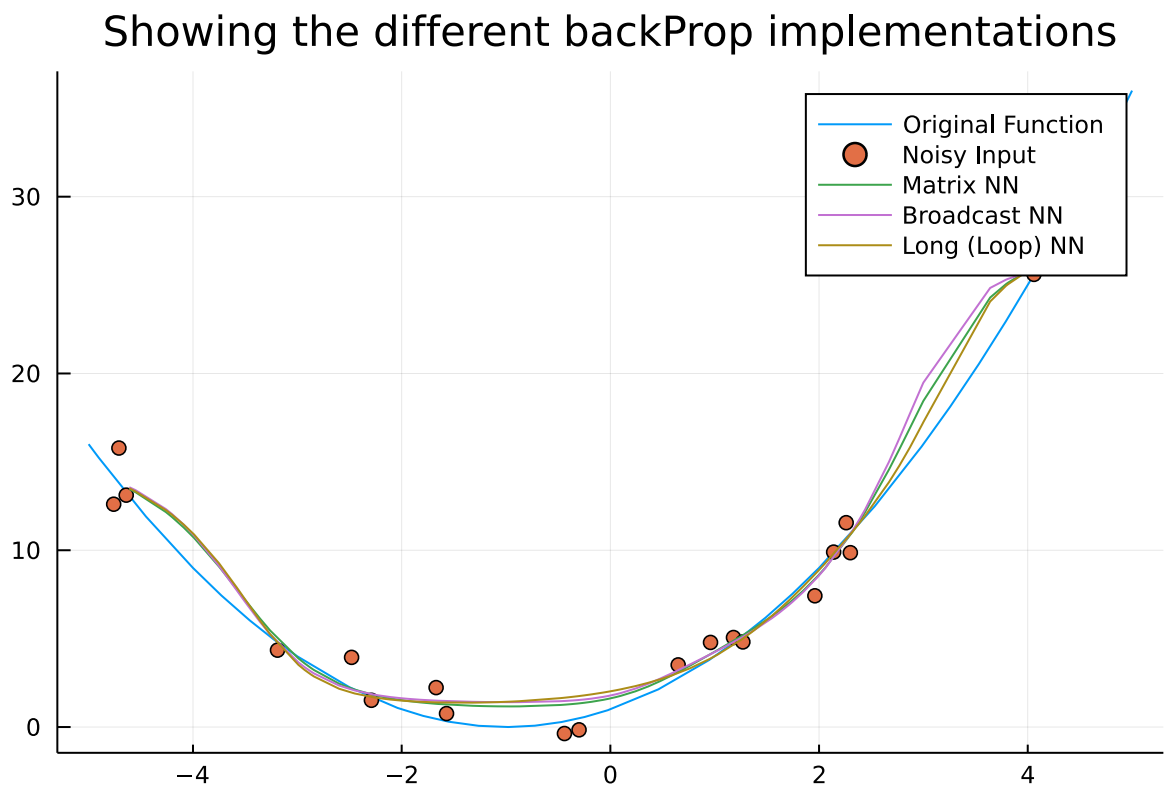
No. of Epochs = 1500

Batch Size = 3

Learning Rate = 0.001

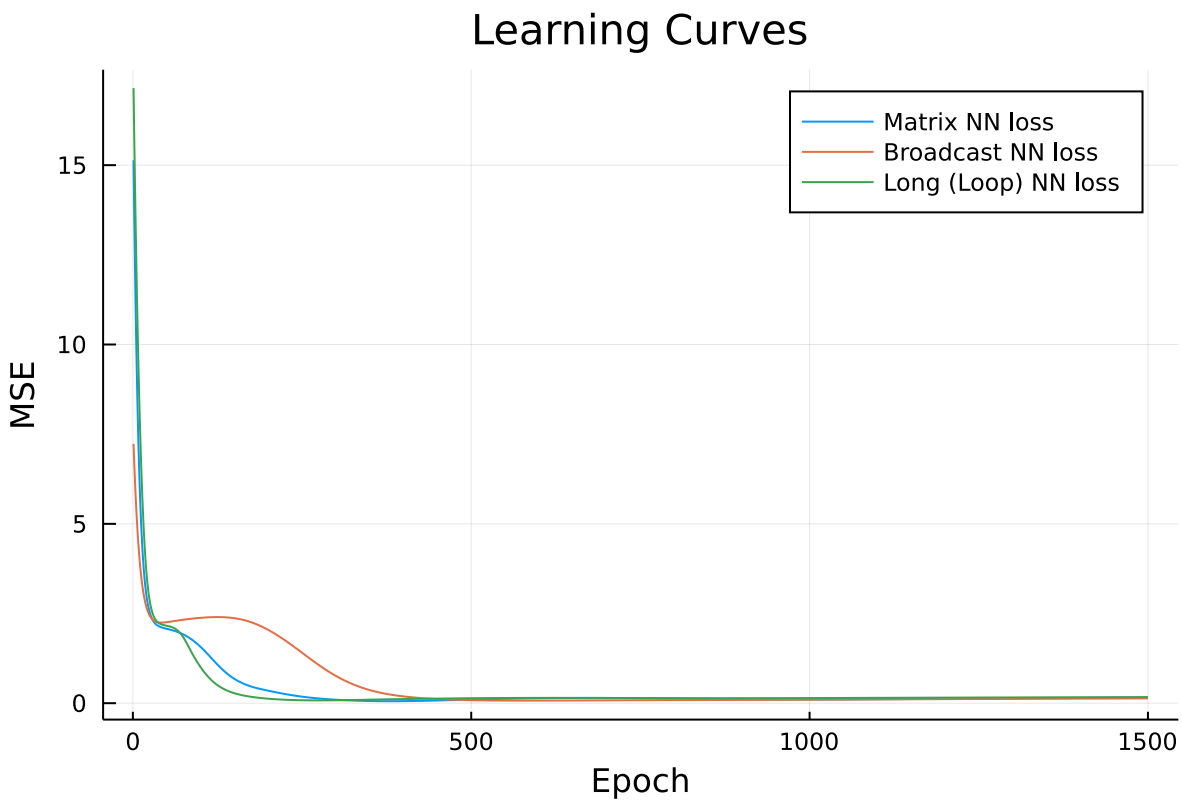
No. of 1st hidden layer nodes = 6

No. of 2nd hidden layer nodes = 3




```
begin
    x_test_ = sort(rand(-5:0.01:5,100))
    weights_matrix_test, cost_history_matrix_test = train(x, t, 1500, 5,
            backProp, .001, 6, 3)
    weights_broadcast_test, cost_history_broadcast_test = train(x, t, 1500, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_test, cost_history_long_test = train(x, t, 1500, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_test, weights_broadcast_test, weights_long_test],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "Showing the different
backProp implementations")
end
```

Below we can see the learning curves are very similar. This proves that our implementation of the three $backProp$ variants was correct




```
learningPlots([cost_history_matrix_test, cost_history_broadcast_test, cost_history_long_test],
              ["Matrix NN loss", "Broadcast NN loss", "Long (Loop) NN loss"], "Learning Curves")
```

Question 2

Compare the accuracy `backprop!`, `backprop_long!`, `backprop_broadcast!` on *test data* as

1. The number of layer 2 (dh_1) hidden node
2. The learning rate changes
3. The number of epocsh increase

To explore the effect of multiple parameters on how our NN learns we have made sliders for each of them. The sliders immediately change the plot below so we can see the result of the experiment.

Hidden layer one nodes:

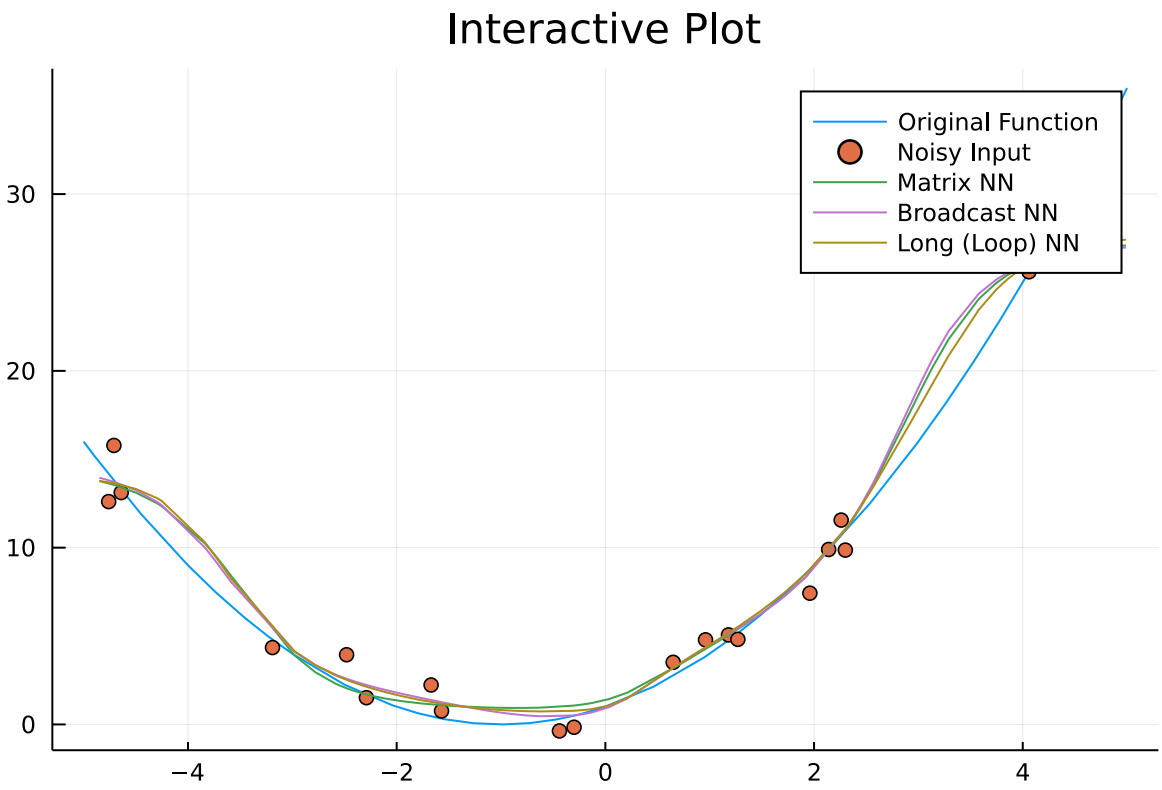
Hidden layer two nodes:

No of Epochs:

Batch Size:

value for η:

```
("d_h1:6", "d_h2:4", "noEpochs:500", "batchSize:5", "η:0.005")
```



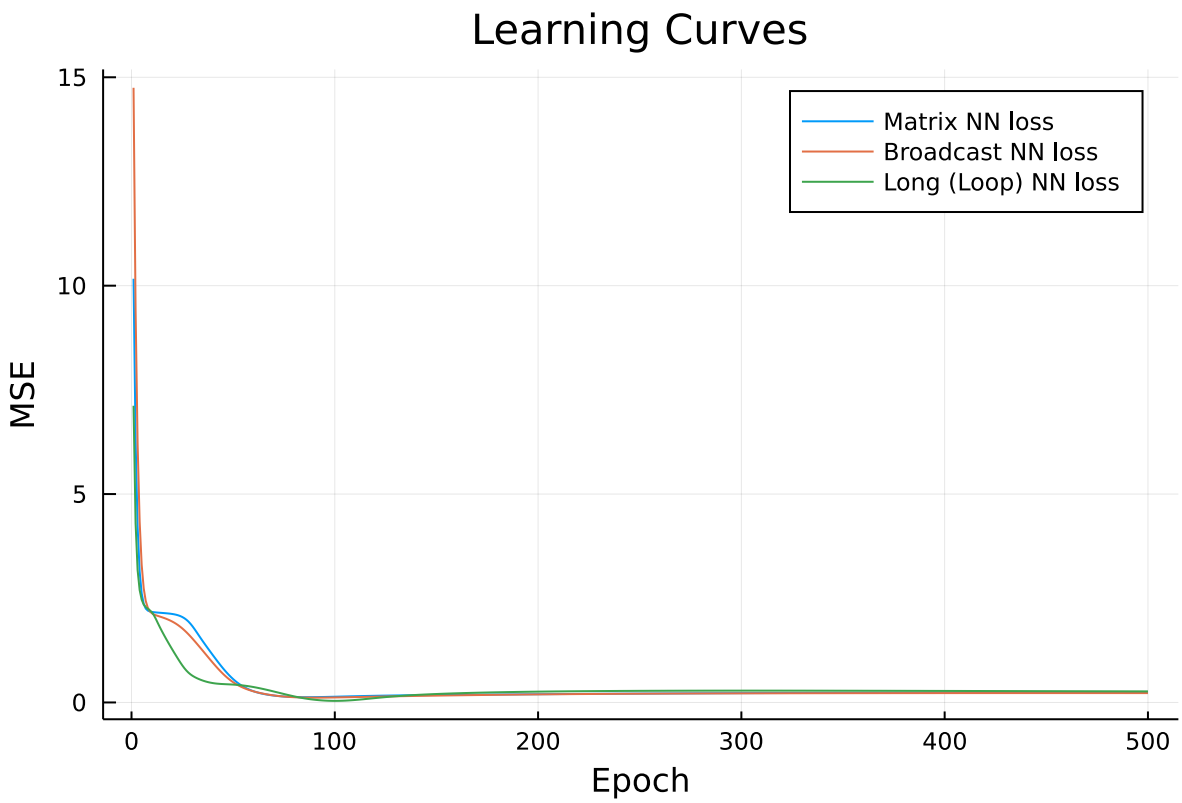

```
begin
    x_test  = sort(rand(-5:0.01:5,100))
    weights_matrix, cost_hist_matrix = train(x, t, noEpochs, batchSize,
            backProp, η, d_h1, d_h2)
    weights_broadcast, cost_hist_broadcast = train(x, t, noEpochs, batchSize,
            backPropBroadcast, η, d_h1, d_h2)
    weights_long, cost_hist_long = train(x, t, noEpochs, batchSize,
            backPropLong, η, d_h1, d_h2)
    comparisonPlots!(f, x, t, x_test,
            [weights_matrix, weights_broadcast, weights_long],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "Interactive Plot")
end
```

Calculating and printing MSE (our measure of accuracy) for each of our three networks

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

```
function calcAccuracy(testSet, weights, target)
    error = 0
    for testPoint in testSet
        error += MSE(testPoint, predict(testPoint, weights))
    end
    return error/length(testSet)
end
```

```
("NN_matrix_accuracy = 119.89", "NN_broadcast_accuracy = 121.46", "NN_long_accuracy =
```




```
learningPlots([cost_hist_matrix, cost_hist_broadcast, cost_hist_long],
            ["Matrix NN loss", "Broadcast NN loss", "Long (Loop) NN loss"], "Learning
Curves")
```

Question 3

How can you explain the results observed in light of all you learned in the course so far.

To answer this question we are going to perform three experiments. We are going to start with a base configuration for an NN as follows:

- No of Epochs = 1000
- batch size = 5
- learning rate = 0.001
- number of first hidden layer nodes = 6

- number of second hidden layer nodes = 3

In each experiment, We are going to change only one of the No. of Epochs, the learning rate or the number of second hidden layer nodes parameters, and observe the effect of that change on the fit using the plots that are generated

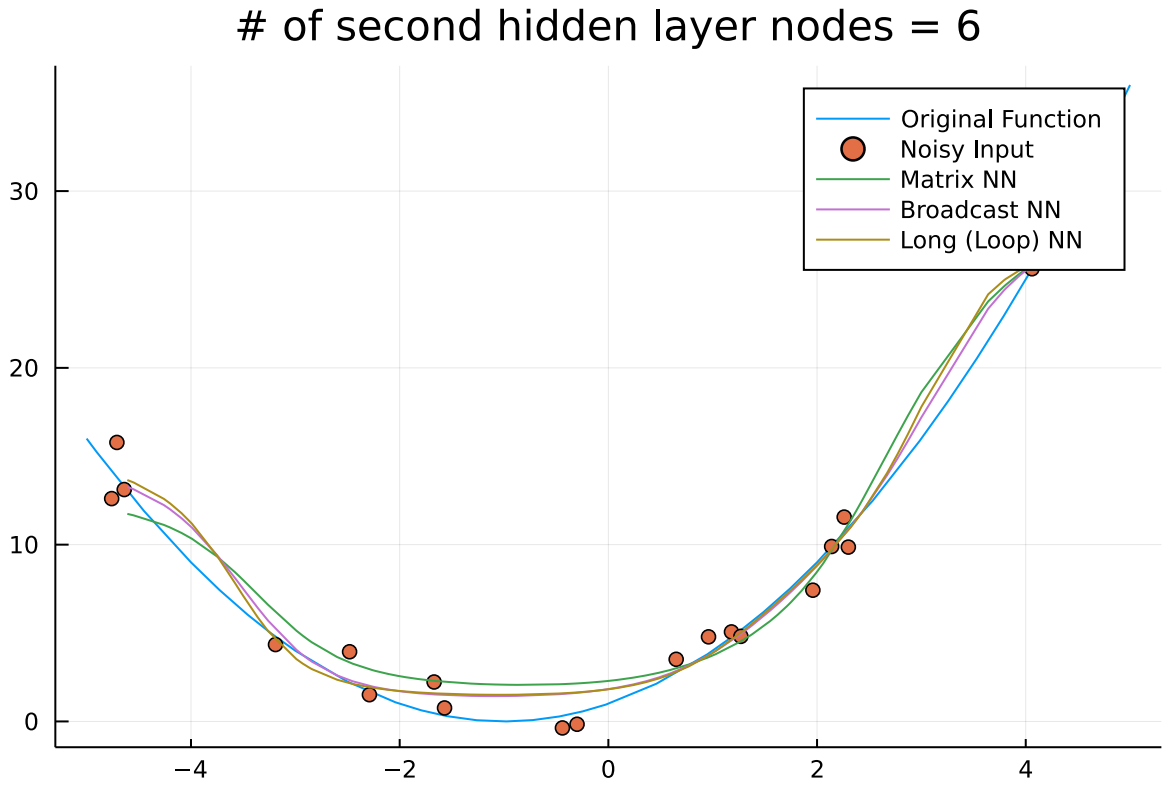
We will also compare the learning curves for each experiment. but we are only going to show the learning curves for the matrix implementation. Since all implementations are similar and we are only interested in studying the effect of change in some hyperparameters as opposed to the NN $backProp$ **implementation**

Effect of Changes in the Number Second Hidden Layer Nodes on the Fit

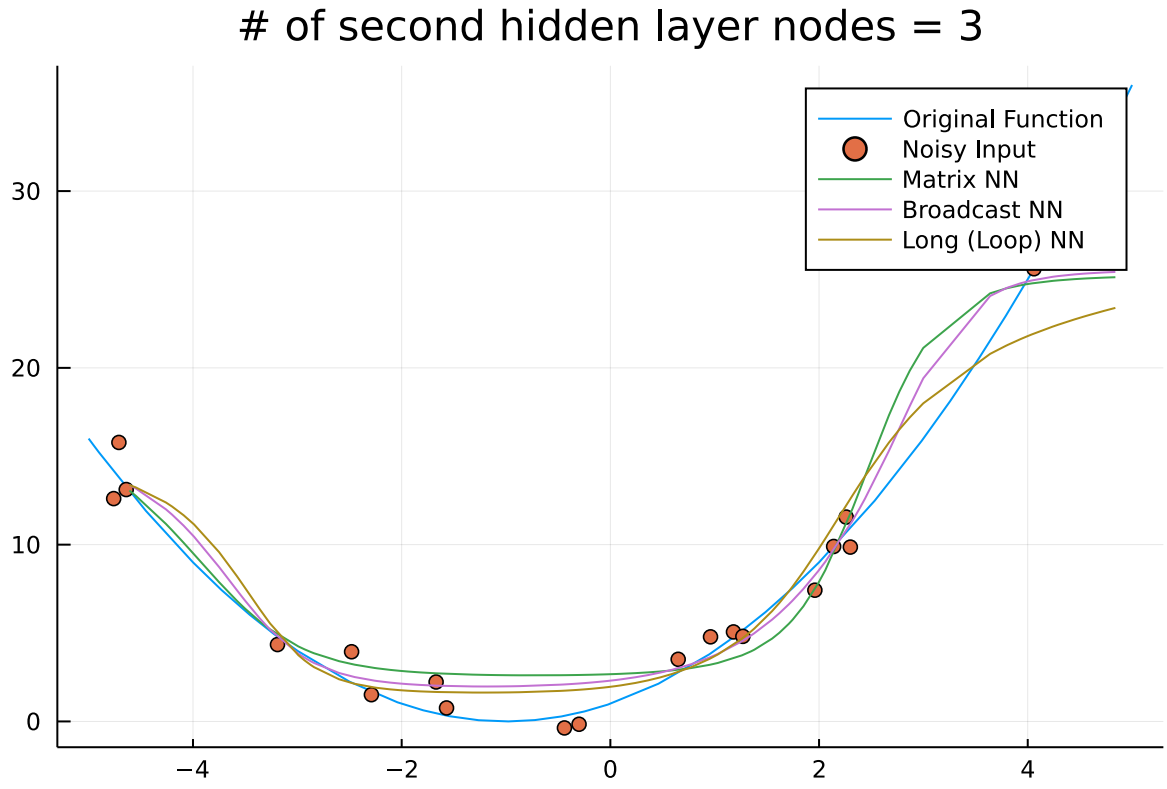
During this experiment, we observe that with increasing dh_2, the resulting curve fits the noisy sample more and more. This is to be expected, because with increased model complexity, the model is able distinguish more nuanced features and combinations there of.

An additional explaination to why the fit was best in the first instance of this experiment is that it is considered best practice to have the same number of nodes across layers in shallow NNs. This condition happened to be true in the first instance.

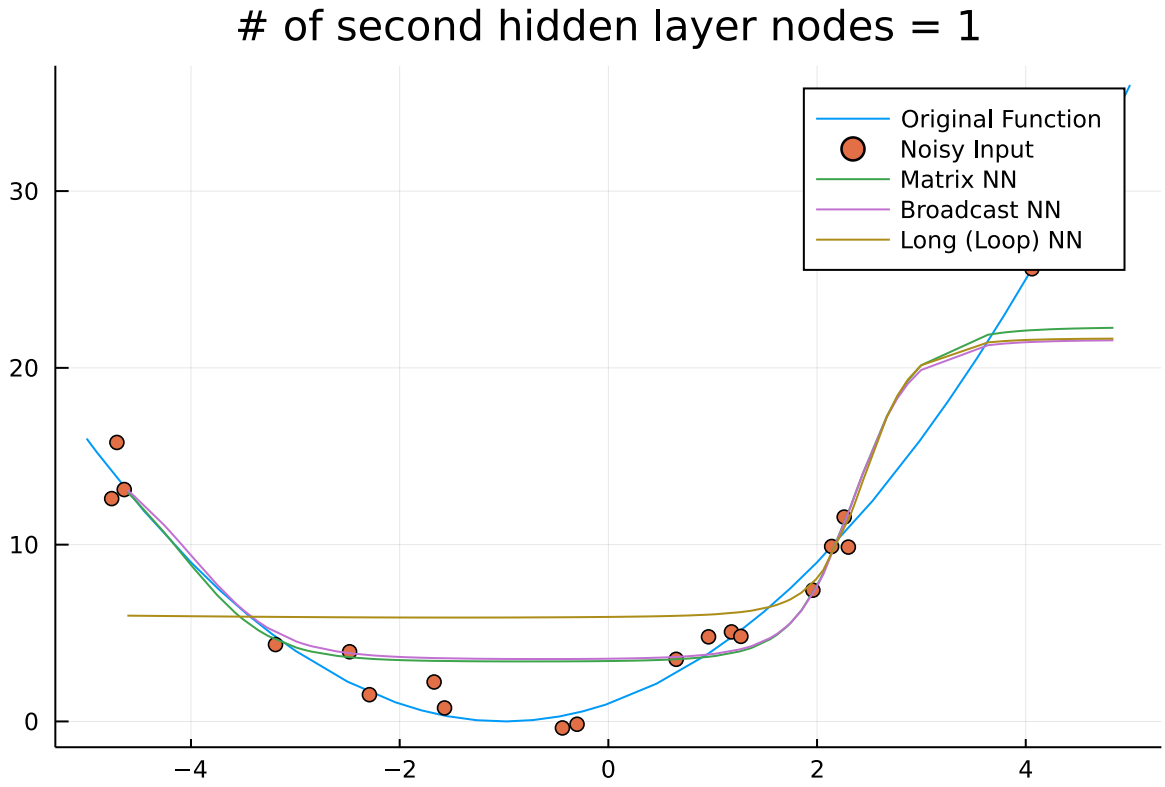
We also notice that the model with the highest number of nodes was overfitting the noisy sample, showing less ability to generalize.



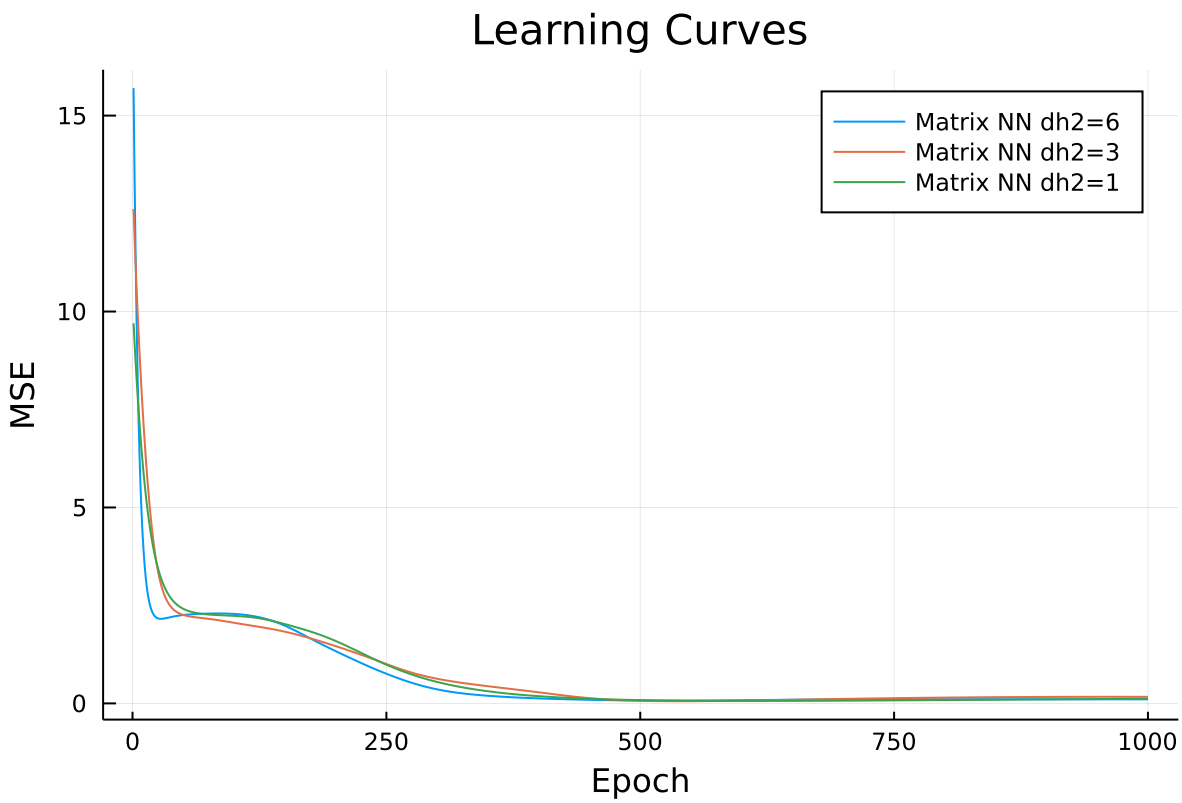

```
begin
    weights_matrix_dh2_1, cost_matrix_dh2_1 = train(x, t, 1000, 5,
            backProp, .001, 6, 6)
    weights_broadcast_dh2_1, _ = train(x, t, 1000, 5,
            backPropBroadcast, .001, 6, 6)
    weights_long_dh2_1, _ = train(x, t, 1000, 5,
            backPropLong, .001, 6, 6)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_dh2_1, weights_broadcast_dh2_1, weights_long_dh2_1],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of second hidden
layer nodes = 6")
end
```




```
begin
    weights_matrix_dh2_2, cost_matrix_dh2_2 = train(x, t, 1000, 5,
            backProp, .001, 6, 3)
    weights_broadcast_dh2_2, _ = train(x, t, 1000, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_dh2_2, _ = train(x, t, 1000, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_dh2_2, weights_broadcast_dh2_2, weights_long_dh2_2],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of second hidden
layer nodes = 3")
end
```




```
begin
    weights_matrix_dh2_3, cost_matrix_dh2_3 = train(x, t, 1000, 5,
            backProp, .001, 6, 1)
    weights_broadcast_dh2_3, _ = train(x, t, 1000, 5,
            backPropBroadcast, .001, 6, 1)
    weights_long_dh2_3, _ = train(x, t, 1000, 5,
            backPropLong, .001, 6, 1)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_dh2_3, weights_broadcast_dh2_3, weights_long_dh2_3],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of second hidden
layer nodes = 1")
end
```



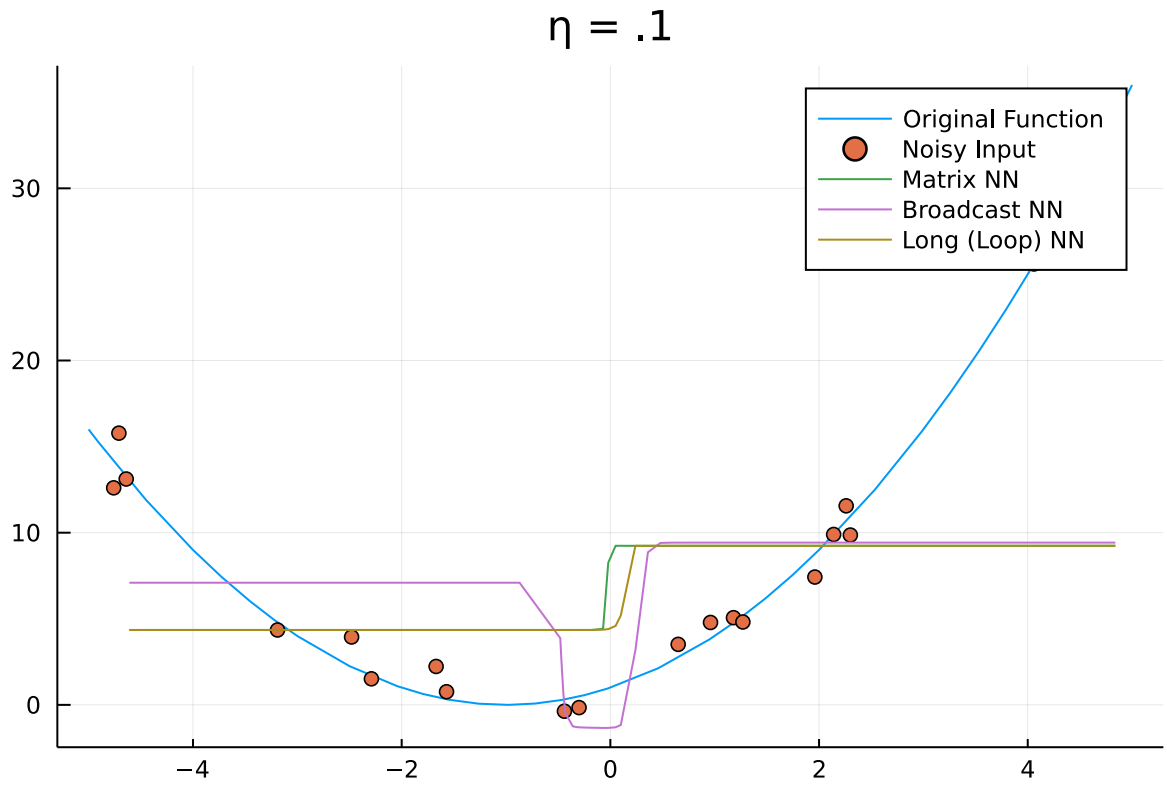

```
learningPlots([cost_matrix_dh2_1, cost_matrix_dh2_2, cost_matrix_dh2_3],
            ["Matrix NN dh2=6", "Matrix NN dh2=3", "Matrix NN dh2=1"], "Learning Curves")
```

Effect of Learning Rate Changes on the Fit

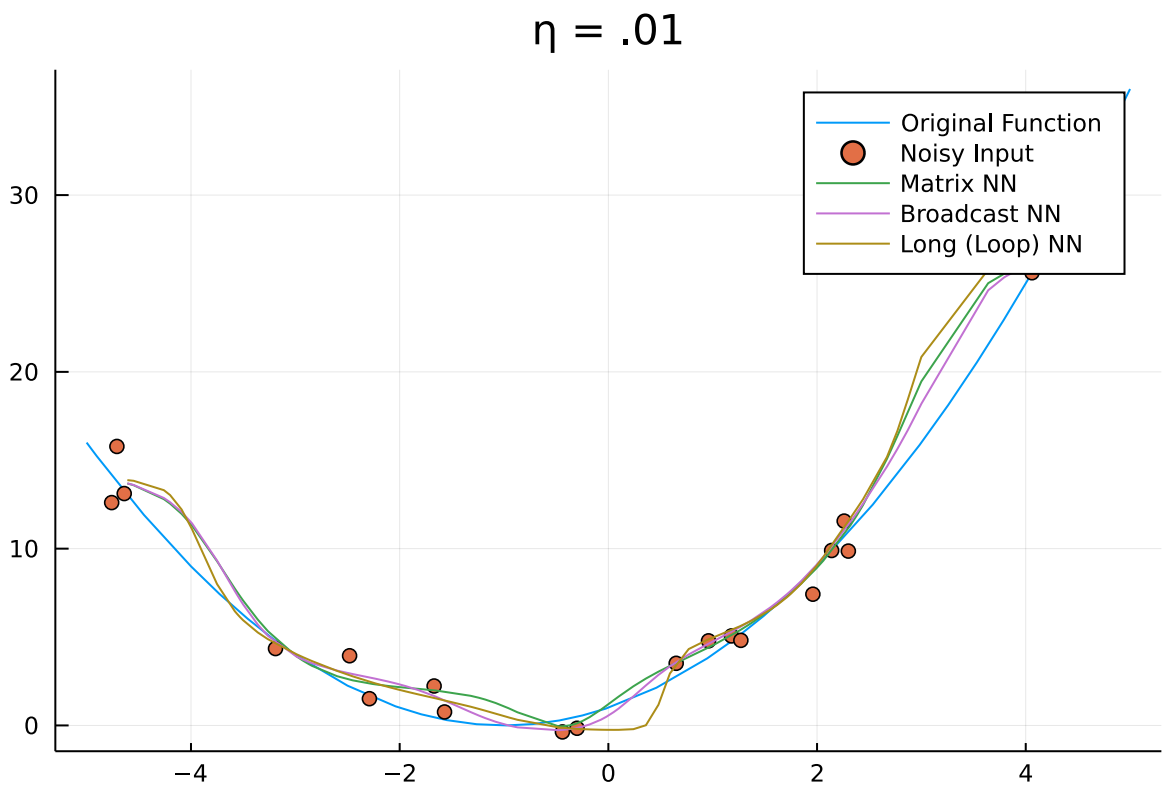
The learning rate is fundamently important to the NNs learning performance as the learning rate dictates the magnitude of update (size of step) that affects the weights after each batch, in the case of batch training.

If the learning rate is too high the updates will be too large that they will overshoot the minimas and fail to converge. If the learning rate is too small, the NN will take a very long time to converge and might be stuck in a local minima. This risk is especially dangerous here as we are using non-adaptive learning rates.

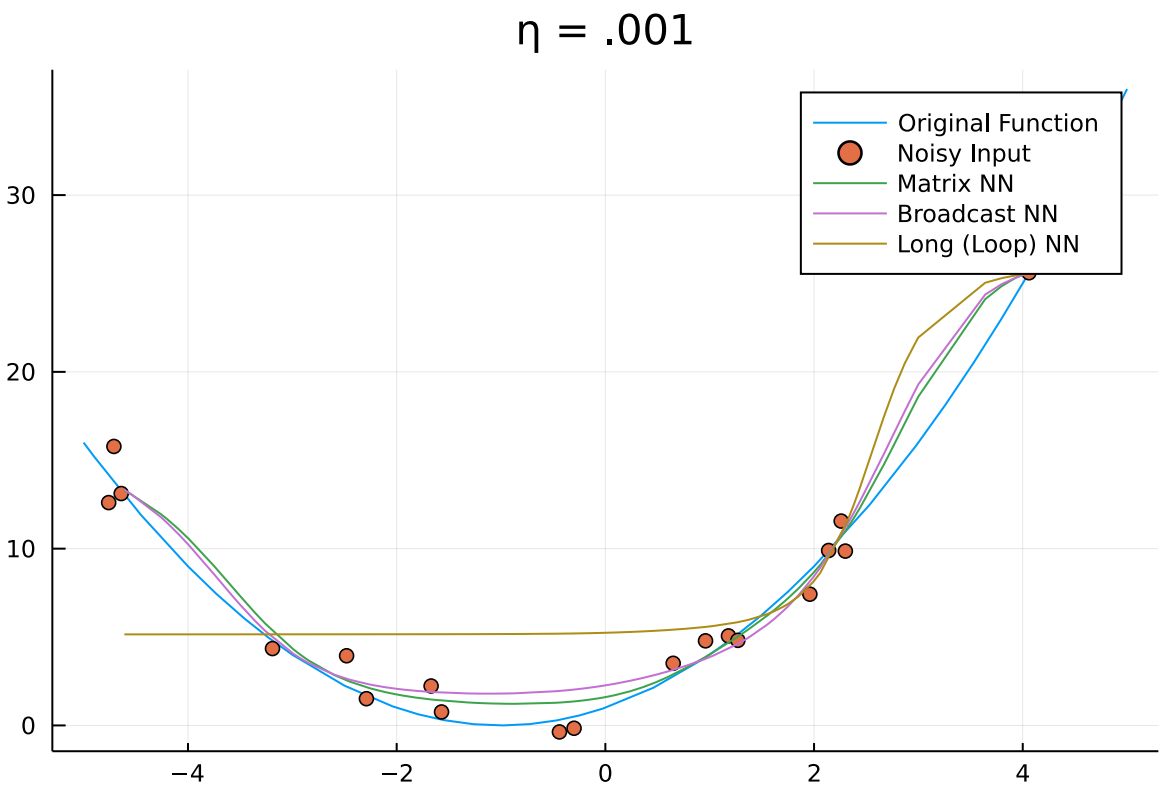
However if the learning rate is reasonable, that will give the NN the best chance of convergence. These three cases, we believe, are nicely displayed in the three examples below.



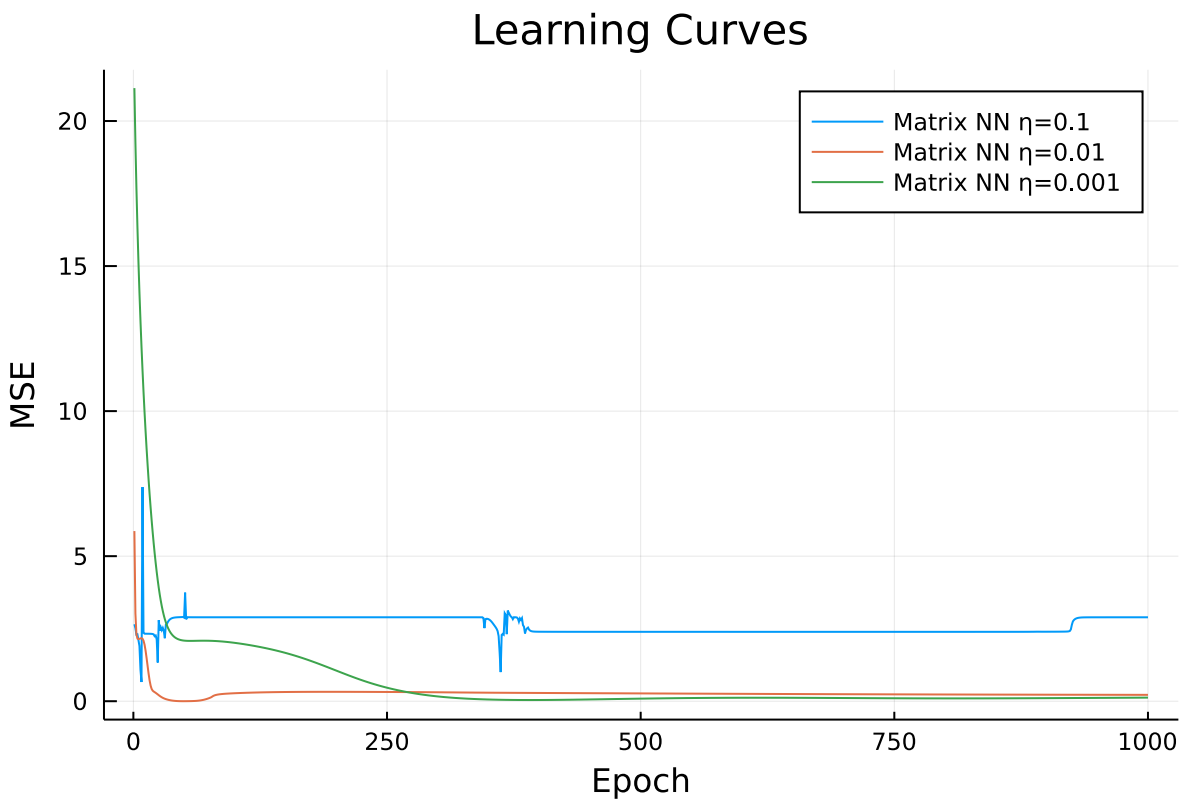

```
begin
    weights_matrix_eta_1, cost_matrix_eta_1 = train(x, t, 1000, 5,
            backProp, .1, 6, 3)
    weights_broadcast_eta_1, _ = train(x, t, 1000, 5,
            backPropBroadcast, .1, 6, 3)
    weights_long_eta_1, _ = train(x, t, 1000, 5,
            backPropLong, .1, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_eta_1, weights_broadcast_eta_1, weights_long_eta_1],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "η = .1")
end
```




```
begin
    weights_matrix_eta_2, cost_matrix_eta_2 = train(x, t, 1000, 5,
            backProp, .01, 6, 3)
    weights_broadcast_eta_2, _ = train(x, t, 1000, 5,
            backPropBroadcast, .01, 6, 3)
    weights_long_eta_2, _ = train(x, t, 1000, 5,
            backPropLong, .01, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_eta_2, weights_broadcast_eta_2, weights_long_eta_2],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "η = .01")
end
```




```
begin
    weights_matrix_eta_3, cost_matrix_eta_3 = train(x, t, 1000, 5,
            backProp, .001, 6, 3)
    weights_broadcast_eta_3, _ = train(x, t, 1000, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_eta_3, _ = train(x, t, 1000, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_eta_3, weights_broadcast_eta_3, weights_long_eta_3],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "η = .001")
end
```



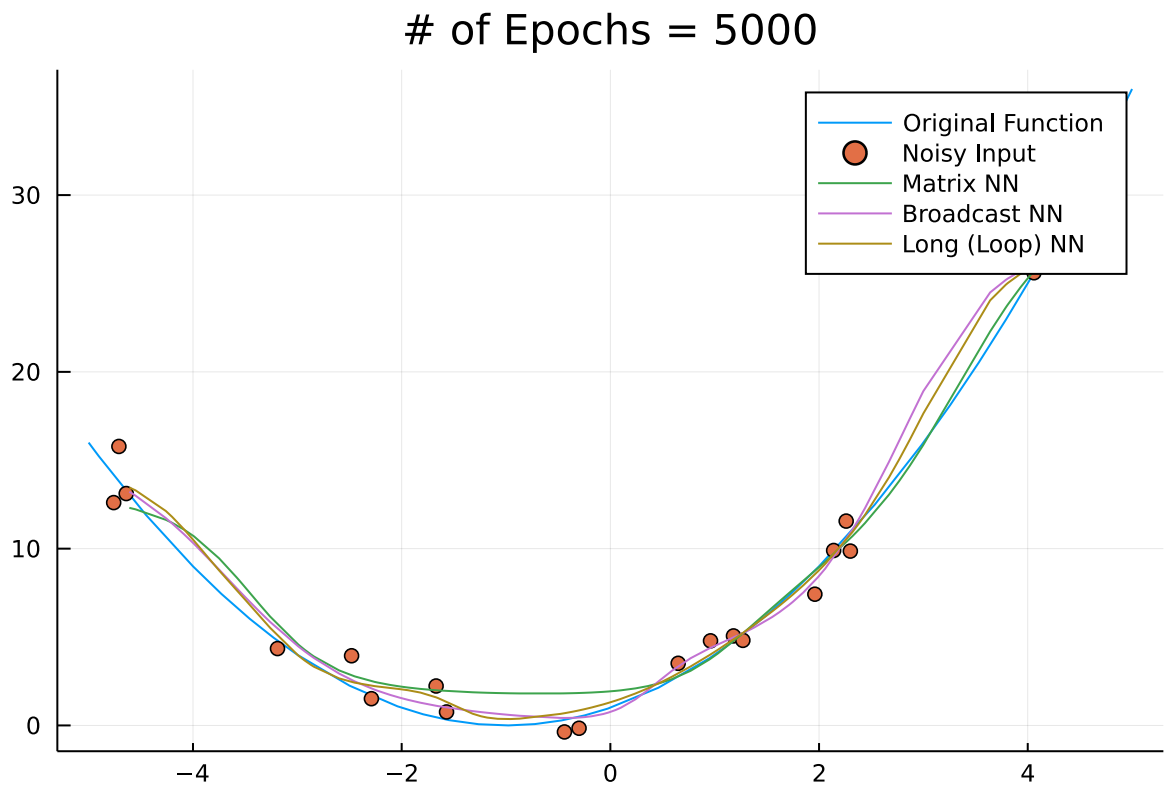

```
learningPlots([cost_matrix_eta_1, cost_matrix_eta_2, cost_matrix_eta_3],
              ["Matrix NN η=0.1", "Matrix NN η=0.01", "Matrix NN η=0.001"], "Learning Curves")
```

Effect of the Number of Epochs Changes on the Fit

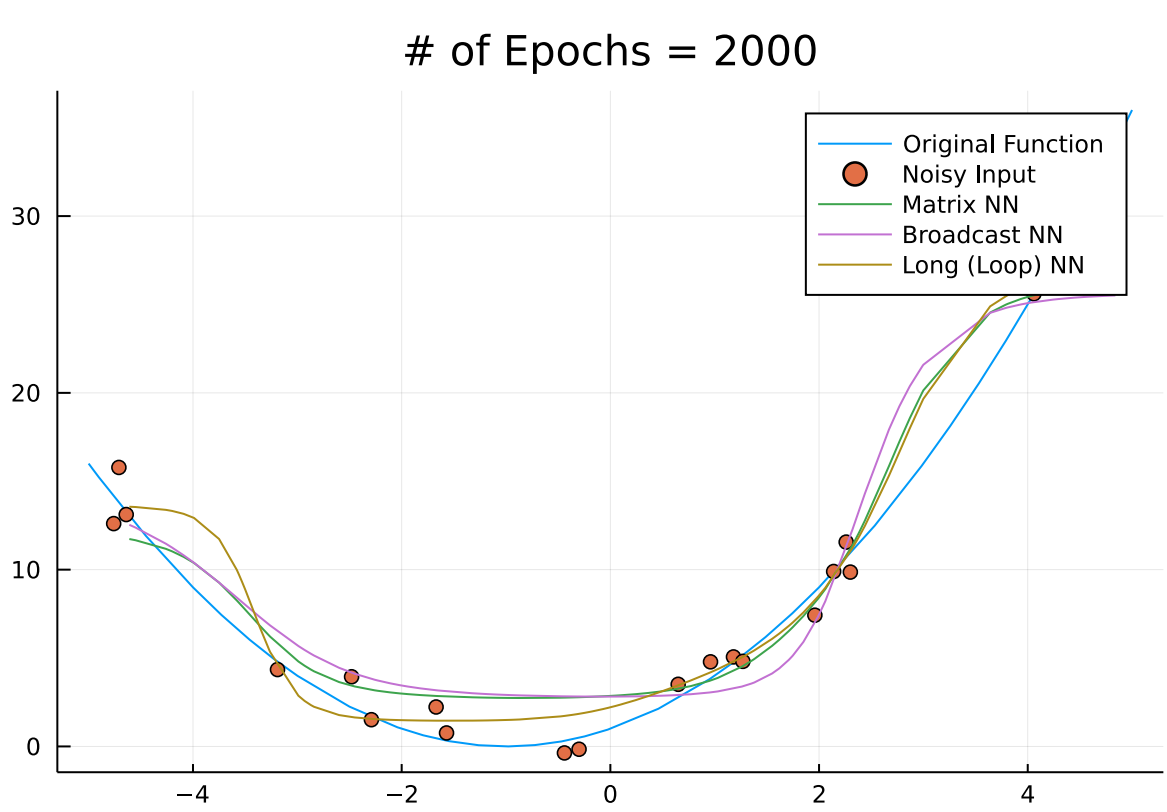
The number of epochs is basically the number of times our NN is going to complete a pass over all datapoints in the training dataset. More passes will result in the NN fitting the training dataset more and more closely, provided all other hyper parameters are adequate.

In this experiment, we know that the learning rate is too low. but that's only going to accentuate the effect of the change in the number of epochs.

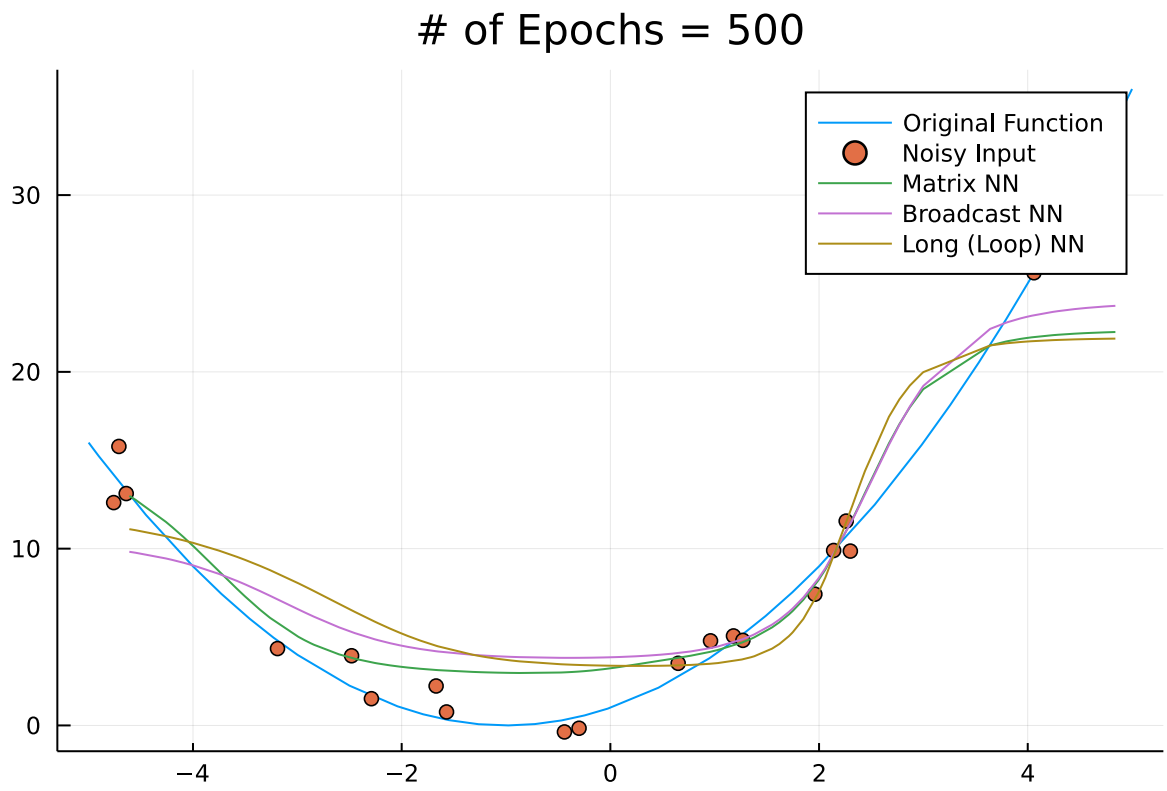
If the NN does not get exposed to the training set enough the fit is going to be very poor. and if the NN is overtrained on the same dataset it is going to suffer from overfitting and failure to generalize. We can see examples of these phenomena below.



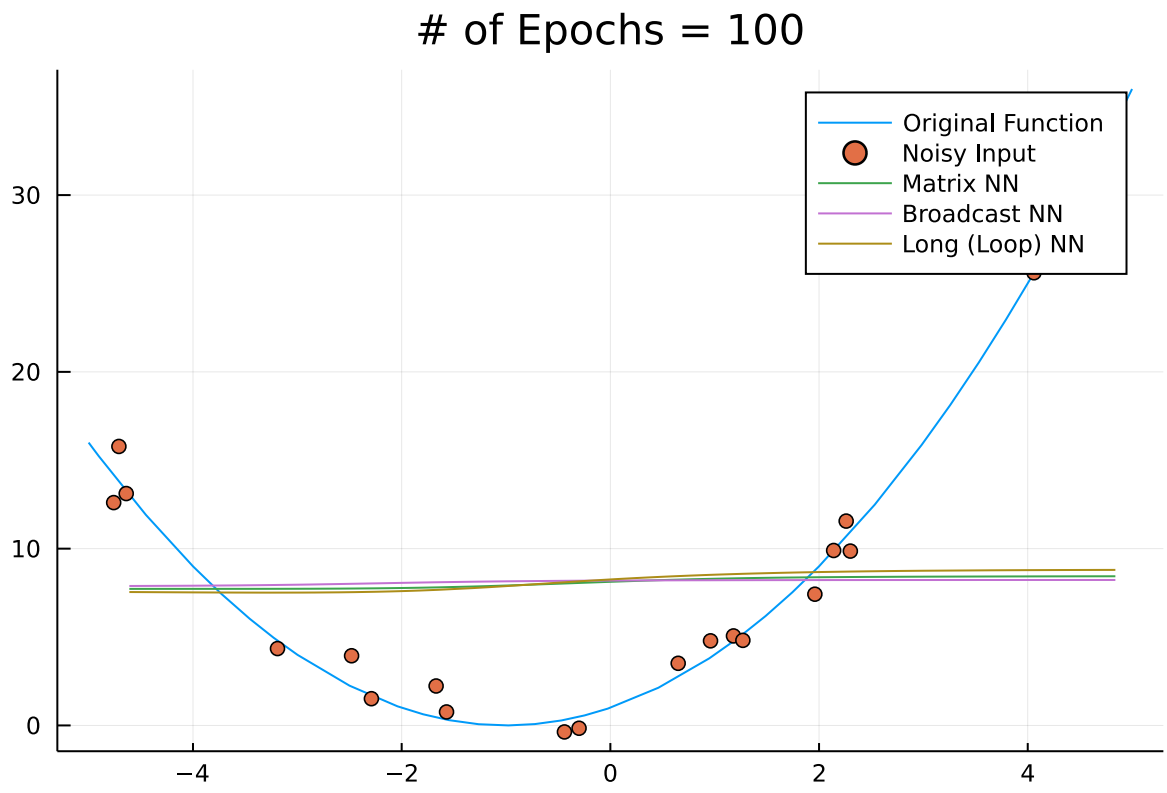

```
begin
    weights_matrix_epoch_1, cost_matrix_epoch_1 = train(x, t, 5000, 5,
            backProp, .001, 6, 3)
    weights_broadcast_epoch_1, _ = train(x, t, 5000, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_epoch_1, _ = train(x, t, 5000, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_epoch_1, weights_broadcast_epoch_1,
weights_long_epoch_1],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of Epochs = 5000")
end
```



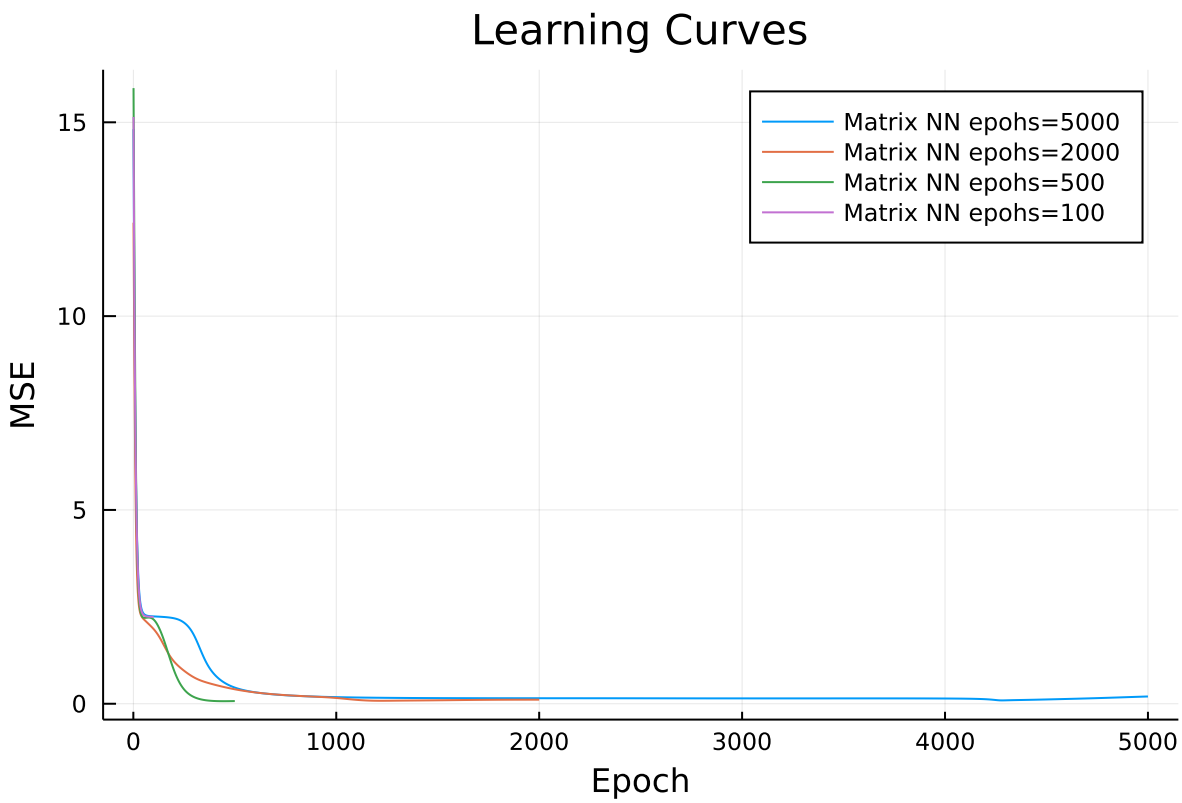

```
begin
    weights_matrix_epoch_2, cost_matrix_epoch_2 = train(x, t, 2000, 5,
            backProp, .001, 6, 3)
    weights_broadcast_epoch_2, _ = train(x, t, 2000, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_epoch_2, _ = train(x, t, 2000, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_epoch_2, weights_broadcast_epoch_2,
weights_long_epoch_2],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of Epochs = 2000")
end
```




```
begin
    weights_matrix_epoch_3, cost_matrix_epoch_3 = train(x, t, 500, 5,
            backProp, .001, 6, 3)
    weights_broadcast_epoch_3, _ = train(x, t, 500, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_epoch_3, _ = train(x, t, 500, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_epoch_3, weights_broadcast_epoch_3,
weights_long_epoch_3],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of Epochs = 500")
end
```




```
begin
    weights_matrix_epoch_4, cost_matrix_epoch_4 = train(x, t, 100, 5,
            backProp, .001, 6, 3)
    weights_broadcast_epoch_4, _ = train(x, t, 100, 5,
            backPropBroadcast, .001, 6, 3)
    weights_long_epoch_4, _ = train(x, t, 100, 5,
            backPropLong, .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_epoch_4, weights_broadcast_epoch_4,
weights_long_epoch_4],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN"], "# of Epochs = 100")
end
```




```
learningPlots([cost_matrix_epoch_1, cost_matrix_epoch_2, cost_matrix_epoch_3, cost_matrix_epoch_4],
              ["Matrix NN epohs=5000", "Matrix NN epohs=2000", "Matrix NN epohs=500", "Matrix NN epohs=100"], "Learning Curves")
```

Bonus question

Working with the (a maybe modified) loss function, use Flux to **take gradiants automatically** to do the training. Are you getting the same/similar results compared to the hand crafted version?

We have to refactor our NN generating code to utilize Flux.gradient. We will start by modifying $forwarProp$, because Flux.gradient does not support mutable arrays in the differentiable function.

All of our modified functions for Flus.gradient will have the FG suffix in the function's name.

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

```
function forwardPropFG(z::Float64, weights::Weights)

    x̄ = weights.W * [1.,z...]
    x = σ.(x̄)
    ȳ = weights.V * [1.,x...]
    y = σ.(ȳ)
    o = weights.U * [1.,y...]

    return o[1]
end
```

We will define a loss function that calls the modified $forwardPropFG$. The loss function uses the same MSE concept that we used in the previous NNs. Ha"ving the same loss function makes it easy to compare the learning curves across all of our implementations.

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

```
function costFuncFG(z::Float64, t::Float64, weights::Weights)
    return (forwardPropFG(z, weights) - t)^2
end
```

We made a function called $backPropFG$ which uses Flux.gradient to calclate $\frac{\partial E}{\partial U}$, $\frac{\partial E}{\partial V}$, and $\frac{\partial E}{\partial W}$ by performing autodiff on the loss function with respect to each of the weight vectors.

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

```
function backPropFG(z::Float64, t::Float64, weights::Weights)
    gs = gradient(() -> costFuncFG(z, t, weights), Flux.params(weights.W, weights.V,
weights.U))
    ∂E_∂U = gs[weights.U]
    ∂E_∂V = gs[weights.V]
    ∂E_∂W = gs[weights.W]
    return ∂E_∂U, ∂E_∂V, ∂E_∂W
end
```

Finally, we have refactored our training code to call our FG functions. We have chosen not to modify the previous implementation of the training code and instead make new FG ones for the sake of simplicity

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

```
function batchLearnFG(z::Array{Float64}, t::Array{Float64},
        η::Float64, weights::Weights)

    cost = 0
    ∂E_∂W = zeros(size(weights.W,1), size(weights.W,2))
    ∂E_∂V = zeros(size(weights.V,1), size(weights.V,2))
    ∂E_∂U = zeros(size(weights.U,1), size(weights.U,2))
    lenBatch = length(z)

    for i in 1:lenBatch
        cost = costFuncFG(z[i], t[i], weights)
        ∂E_∂U, ∂E_∂V, ∂E_∂W = backPropFG(z[i], t[i], weights)
    end

    weights.W -= η .* ∂E_∂W
    weights.V -= η .* ∂E_∂V
```

```
    weights.U -= η .* ∂E_∂U

    return weights, cost
end
```

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

```
function trainOneEpochFG(z::Array{Float64}, t::Array{Float64},
        batchSize::Int64, η::Float64, weights::Weights)

    cost = 0
    lenData = length(z)
    curBatchStart = 1

    curBatchSize = batchSize
    while curBatchStart < lenData
        batchInputs = z[curBatchStart:curBatchStart + curBatchSize]
        batchTargets = t[curBatchStart:curBatchStart + curBatchSize]
        weights, cost = batchLearnFG(batchInputs, batchTargets, η, weights)
        cost += cost
        curBatchStart += curBatchSize + 1

        if (curBatchStart + batchSize) > lenData
            curBatchSize = lenData - curBatchStart
        end

    end

    cost /= lenData

    return weights, cost
end
```

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

```
function trainFluxGradient(z::Array{Float64}, t::Array{Float64},
        noEpochs::Int64, batchSize::Int64,η::Float64,
        d_h1::Int64, d_h2::Int64)

    weights::Weights=init_weights(d_h1, d_h2)
    cost_hist = zeros(noEpochs)

    for epochNo in 1:noEpochs
        weights, cost = trainOneEpochFG(z, t, batchSize, η, weights)
        cost_hist[epochNo] = cost
        # println("Epoch Number: $epochNo, Total Error: $cost")
    end

    return weights, cost_hist
end
```

To compare the Flux.gradient implementation with our vanilla implementations, we are going to train a NN using trainFluxGradient giving it the same hyper parameters we have used in the test cases before:

No. of Epochs = 1500

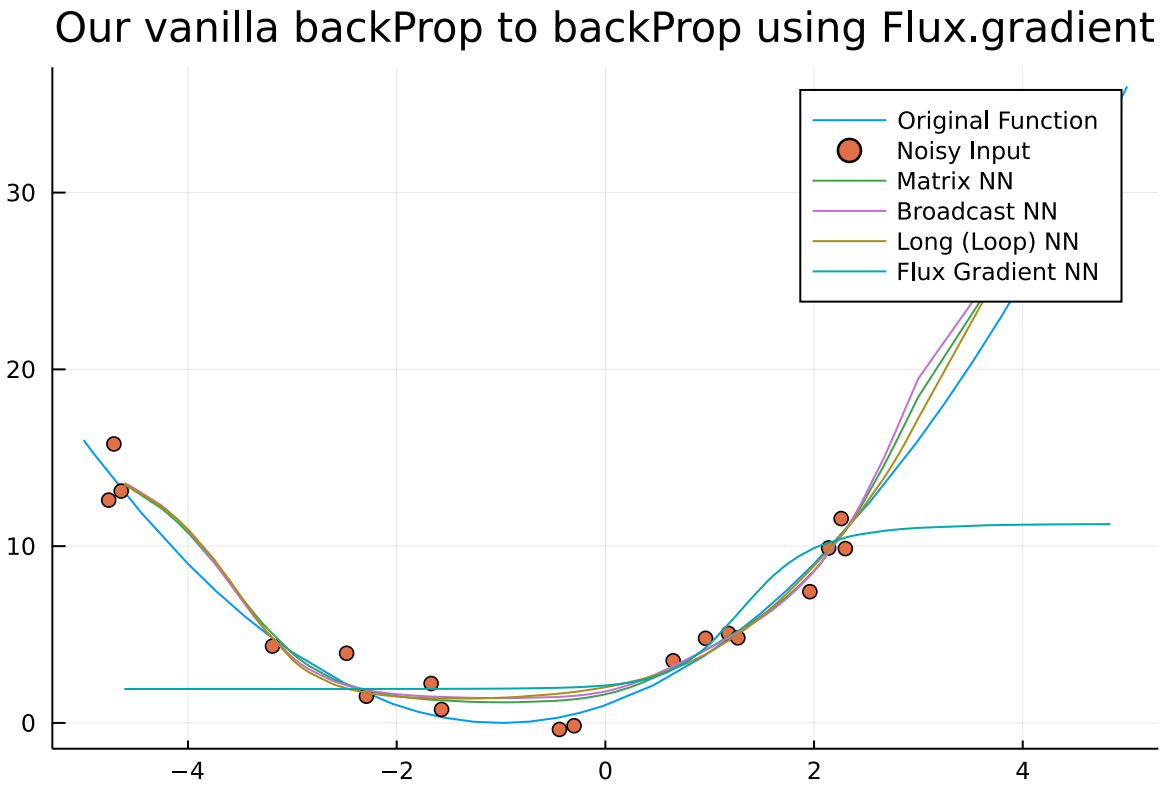
Batch Size = 3

Learning Rate = 0.001

No. of 1st hidden layer nodes = 6

No. of 2nd hidden layer nodes = 3

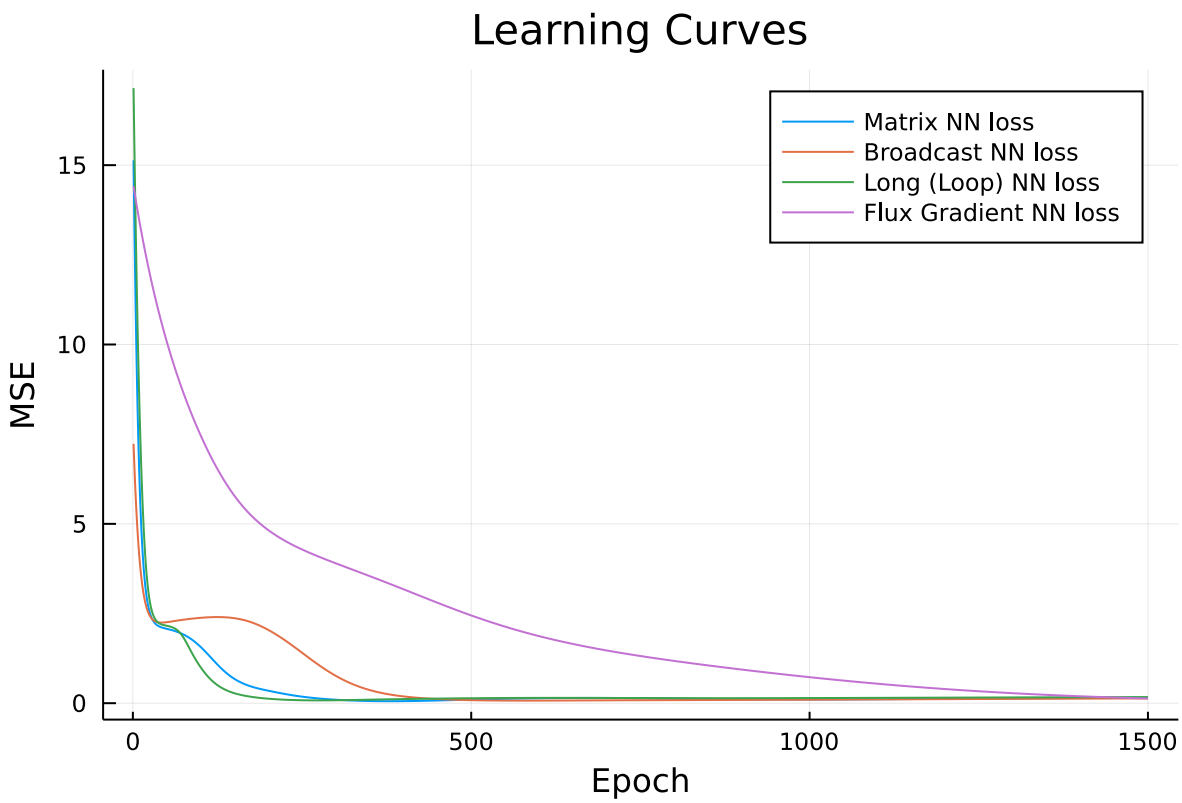
Afterwards, we will plot the predictions of each implementation to judge the fit and will also plot the learning curves of each implementation




```
begin
    weights_FG_test, cost_history_FG_test = trainFluxGradient(x, t, 1500, 5,
            .001, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_matrix_test, weights_broadcast_test, weights_long_test,
weights_FG_test],
            ["Matrix NN", "Broadcast NN", "Long (Loop) NN", "Flux Gradient NN"], "Our
vanilla backProp to backProp using Flux.gradient")
end
```

As could be seen from the plot above, it seems that the Flux.gradient implementation failed to converge given the same test data and the same hyper parameters as our vanilla implementations

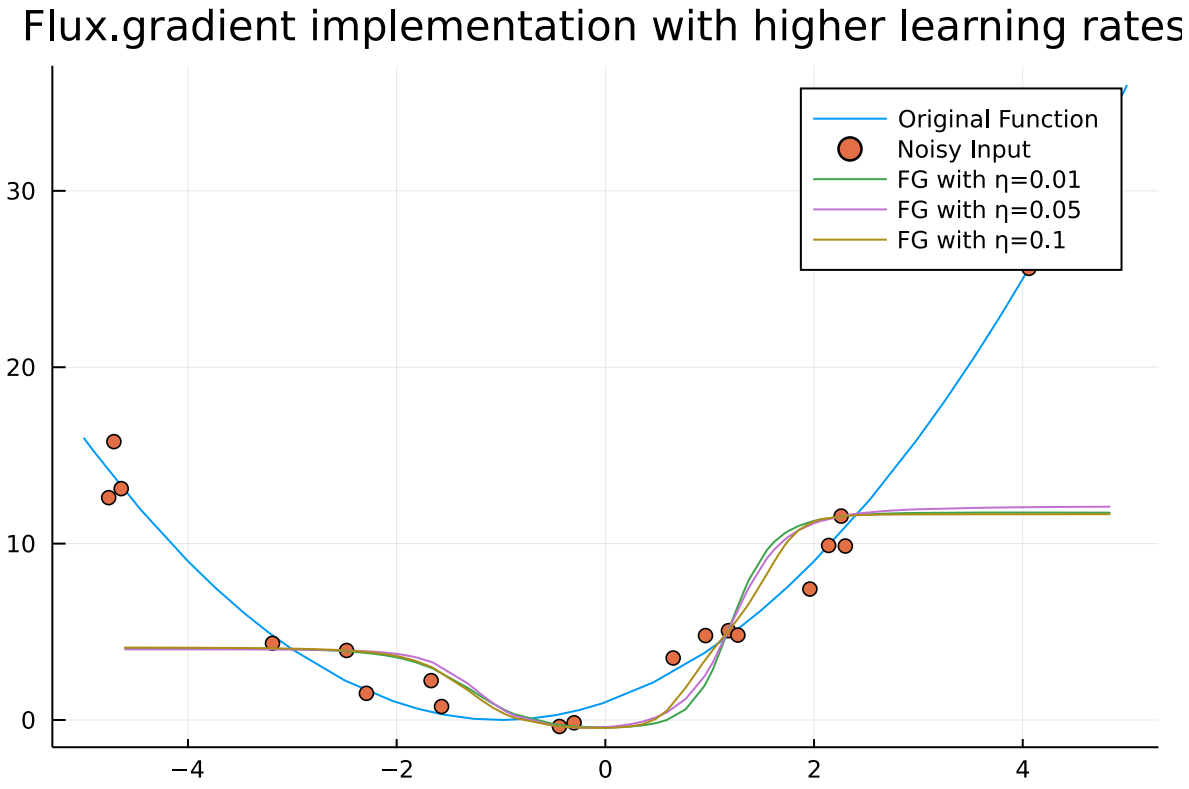
It should also be noted that looking at the learning curve below, perhaps the Flux.gradient model would benefit from a higher learning rate.



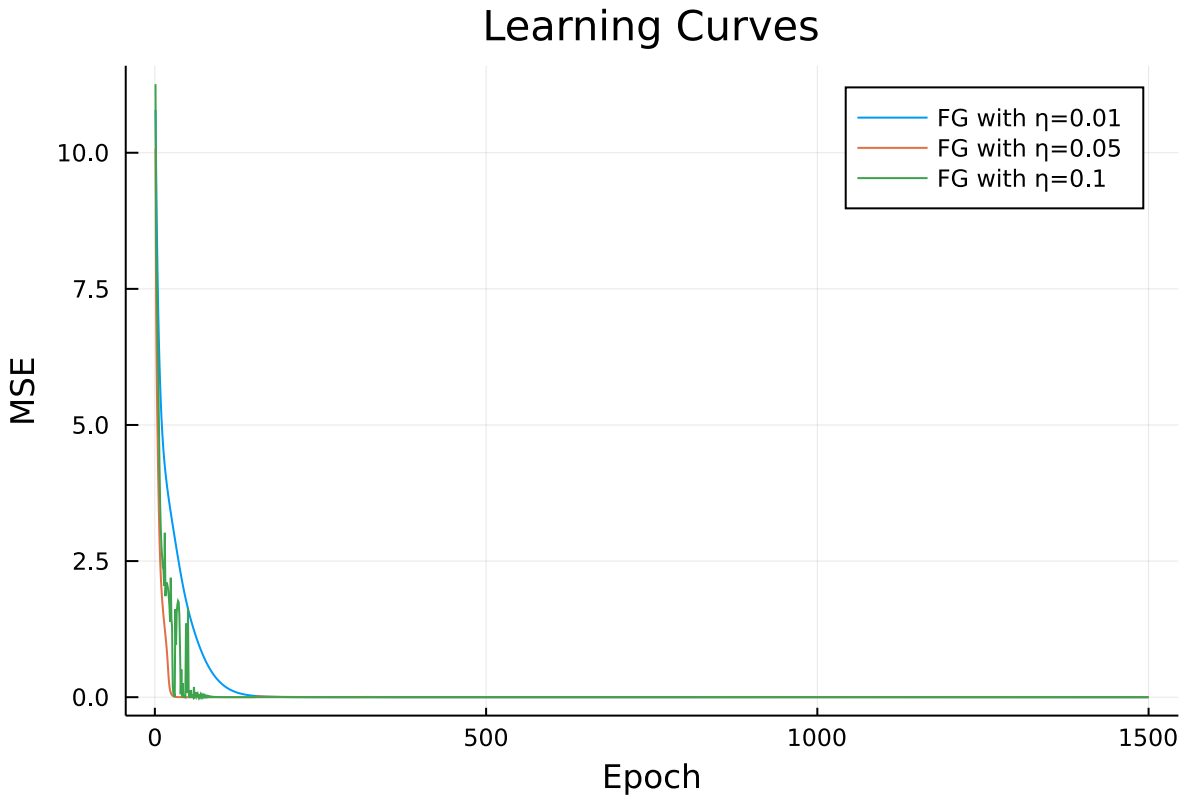

```
learningPlots([cost_history_matrix_test, cost_history_broadcast_test,
cost_history_long_test, cost_history_FG_test],
            ["Matrix NN loss", "Broadcast NN loss", "Long (Loop) NN loss", "Flux
Gradient NN loss"], "Learning Curves")
```

In the following experiment we have tried increasing the FG implementation's learning rate. By looking at the learning curves, this helped the model converge in much fewer iterations. However, the goodness of fit is not much improved. we can see the model still underfitting our data and struggling to fit the datapoints close to the boundaries of the dataset's range (they could be considered outliers since our data contains a big cluster of points in the middle with much fewer points close the boundaries).

This dataset exhibits heteroscedasticity, and with such a small sample, it is hard to perform regression effectively.




```
begin
    weights_FG_test_2, cost_history_FG_test_2 = trainFluxGradient(x, t, 1500, 5,
            .01, 6, 3)
    weights_FG_test_3, cost_history_FG_test_3 = trainFluxGradient(x, t, 1500, 5,
        .05, 6, 3)
    weights_FG_test_4, cost_history_FG_test_4 = trainFluxGradient(x, t, 1500, 5,
        .1, 6, 3)
    comparisonPlots!(f, x, t, x_test_,
            [weights_FG_test_2, weights_FG_test_3, weights_FG_test_4],
            ["FG with η=0.01", "FG with η=0.05", "FG with η=0.1"], "Flux.gradient
implementation with higher learning rates")
end
```




```
learningPlots([cost_history_FG_test_2, cost_history_FG_test_3,
cost_history_FG_test_4],
            ["FG with η=0.01", "FG with η=0.05", "FG with η=0.1"], "Learning Curves")
```