Ctrl + S

Importing Needed Packages

```
md"""
# Importing Needed Packages
"""
```

```
begin
    using Markdown
    using StatsFuns
    using Plots
    using Random
    using LinearAlgebra
end
```

Building The Neral Network

```
md"""
# Building The Neral Network
"""
```

Defining a `Weights` Type to hold all Weight Vectors for Convenience

```
md"""
## Defining a `Weights` Type to hold all Weight Vectors for Convenience
"""
```

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

Defining The Forward Pass Function

σ (the Sigmoid function) will be used as the activation function for all hidden layers. A linear activation function is used for the output layer

```
md"""
## Defining The Forward Pass Function
σ (the Sigmoid function) will be used as the activation function for all hidden
layers. A linear activation function is used for the output layer
"""
```

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

```
σ = StatsFuns.logistic
```

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

```
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
end
```

Defining The Back Propagation Function

We have chosen to perform Back Propagation using matrices and linear algebra in this assignment as defined in section `5.2` of the first lecture notes. This choice was made for Three main reasons:

- Lack of comfort with Julia (for loops having a scope of their own is something that I deeply struggled with)
- For consistency (since the forward pass function from the lecture also used linear algebra
- The code was much more intuitive for me once I started using the matrices approach. fewer variables were vague because of notation and I could easily track matrix sizes which lead to me catching errors early on

This approach, I believe, gave me a much deeper understanding of back propagation.

```
md"""
## Defining The Back Propagation Function

We have chosen to perform Back Propagation using matrices and linear algebra in this assignment as defined in section `5.2` of the first lecture notes. This choice was made for Three main reasons:
- Lack of comfort with Julia (for loops having a scope of their own is something that I deeply struggled with)
- For consistency (since the forward pass function from the lecture also used linear algebra
- The code was much more intuitive for me once I started using the matrices approach. fewer variables were vague because of notation and I could easily track matrix sizes which lead to me catching errors early on

This approach, I believe, gave me a much deeper understanding of back propagation.
"""
```

σ̇ is a function that we defined to caculate the differential of Sigmoid.

```
md"""
σ̇ is a function that we defined to caculate the differential of Sigmoid.
"""
```

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

```
function σ̇(x)
    σ.(x) .* (1 .- σ.(x))
end
```

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

```
function backProp(z::Float64, o::Array{Float64}, t::Float64, y::Array{Float64}, x::Array{Float64}, weights::Weights)
    δₒ = o - [t]
    ∂E_∂U = δₒ.*transpose(y)

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

    δₕ₁ = (Diagonal(σ̇(x))*transpose(weights.V)*δₕ₂)[2:end]
    ∂E_∂W = δₕ₁.*transpose([1,z...])
    return ∂E_∂W, ∂E_∂V, ∂E_∂U
end
```

Writing Functions to Perform Batch Training for Each Epoch

We started wirh writing a function to train over a batch of a given size. the function will calculate the average gradients over this patch, update the weights and then return them.

The function will also calculate the cost (Quadratic cost) for each time a forward pass is made. The sum of the batch cost is returned

```
md"""
## Writing Functions to Perform Batch Training for Each Epoch

We started wirh writing a function to train over a batch of a given size. the functior
```

```
will calculate the average gradients over this patch, update the weights and then
return them.

The function will also calculate the cost (Quadratic cost) for each time a forward
pass is made. The sum of the batch cost is returned
"""
```

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

```
function batchLearn(z::Array{Float64}, t::Array{Float64}, weights::Weights,
η::Float64)
    cost = 0
    ∂E_∂W = zeros(6, 2)
    ∂E_∂V = zeros(3, 7)
    ∂E_∂U = zeros(1, 4)
    lenBatch = length(z)
    for i in 1:lenBatch
        oₜₑₘₚ, yₜₑₘₚ, xₜₑₘₚ = forwardProp(z[i], weights)
        ∂E_∂Wₜₑₘₚ, ∂E_∂Vₜₑₘₚ, ∂E_∂Uₜₑₘₚ = backProp(z[i], oₜₑₘₚ, t[i], yₜₑₘₚ, xₜₑₘₚ,
weights)
        ∂E_∂W .+= ∂E_∂Wₜₑₘₚ
        ∂E_∂V .+= ∂E_∂Vₜₑₘₚ
        ∂E_∂U .+= ∂E_∂Uₜₑₘₚ
        cost += .5 * (oₜₑₘₚ[1] - t[i])^2
    end

    ∂E_∂W ./= lenBatch
    ∂E_∂V ./= lenBatch
    ∂E_∂U ./= lenBatch

    weights.W += -η .* ∂E_∂W
    weights.V += -η .* ∂E_∂V
    weights.U += -η .* ∂E_∂U
    return weights, cost
end
```

Next, We defined a function that completes the training over one epoch by calling `batchLearn` iteratively until all the training dataset is exhausted. Each time `batchLearn` is called the `weights` are updated and are passed to the next batch.

`trainOneEpoch` takes a `batchSize` parameter which controls how many datapoints are considered a batch.

Additionally, the sum of cost is divided by the number of datapoints in the training set and returned as the average cost. This is done just to give indication that the network is learning each epoch.

```
md"""
Next, We defined a function that completes the training over one epoch by calling `batchLearn` iteratively until all the training dataset is exhausted. Each time `batchLearn` is called the `weights` are updated and are passed to the next batch.

`trainOneEpoch` takes a `batchSize` parameter which controls how many datapoints are considered a batch.

Additionally, the sum of cost is divided by the number of datapoints in the training set and returned as the average cost. This is done just to give indication that the network is learning each epoch.
"""
```

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

```
function trainOneEpoch(z::Array{Float64}, t::Array{Float64}, weights::Weights, batchSize::Int64, η::Float64)
    cost = 0
    lenData = length(z)
    curBatchStart = 1
    while curBatchStart < lenData
        batchInputs = z[curBatchStart:curBatchStart + batchSize]
        batchTargets = t[curBatchStart:curBatchStart + batchSize]
        weights, cost = batchLearn(batchInputs, batchTargets, weights, η)
        cost += cost
        curBatchStart += batchSize
        if (curBatchStart + batchSize) > lenData
            batchSize = lenData - curBatchStart
        end
    end
    cost /= lenData
    return weights, cost
end

```

Finally, `TrainNN` is a function where we can pass the entire training dataset, the targets and define the number of epochs and the batch size desired. The function will then carry out the training by calling `trainOneEpoch` for the number of epochs defined. After each epoch, the average cose will be printed.

```
md"""
Finally, `TrainNN` is a function where we can pass the entire training dataset, the targets and define the number of epochs and the batch size desired. The function will then carry out the training by calling `trainOneEpoch` for the number of epochs defined. After each epoch, the average cose will be printed.
"""
```

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

```
function TrainNN(z::Array{Float64}, t::Array{Float64}, noEpochs::Int64, batchSize::Int64, η::Float64)
    weights = Weights(randn(6, 2), randn(3, 7), randn(1, 4))
    for epochNo in 1:noEpochs
        weights, cost = trainOneEpoch(z, t, weights, batchSize, η)
        println("Epoch Number: $epochNo, Total Error: $cost")
    end
    return weights
end
```

Training the Network and Plotting the results

This network takes one input and performs regression to predict the value of one output. the function `f(x) = x^2 + 2x + 1` is used to generate an Ad-hoc training dataset.

```
md"""
# Training the Network and Plotting the results

This network takes one input and performs regression to predict the value of one output. the function `f(x) = x^2 + 2x + 1` is used to generate an Ad-hoc training dataset.
"""
```

```
Float64[0.4624, 3.2761, 0.2916, 3.4596, 0.1296, 1.5376, 1.2769, 0.0025, 0.8836, 2
```

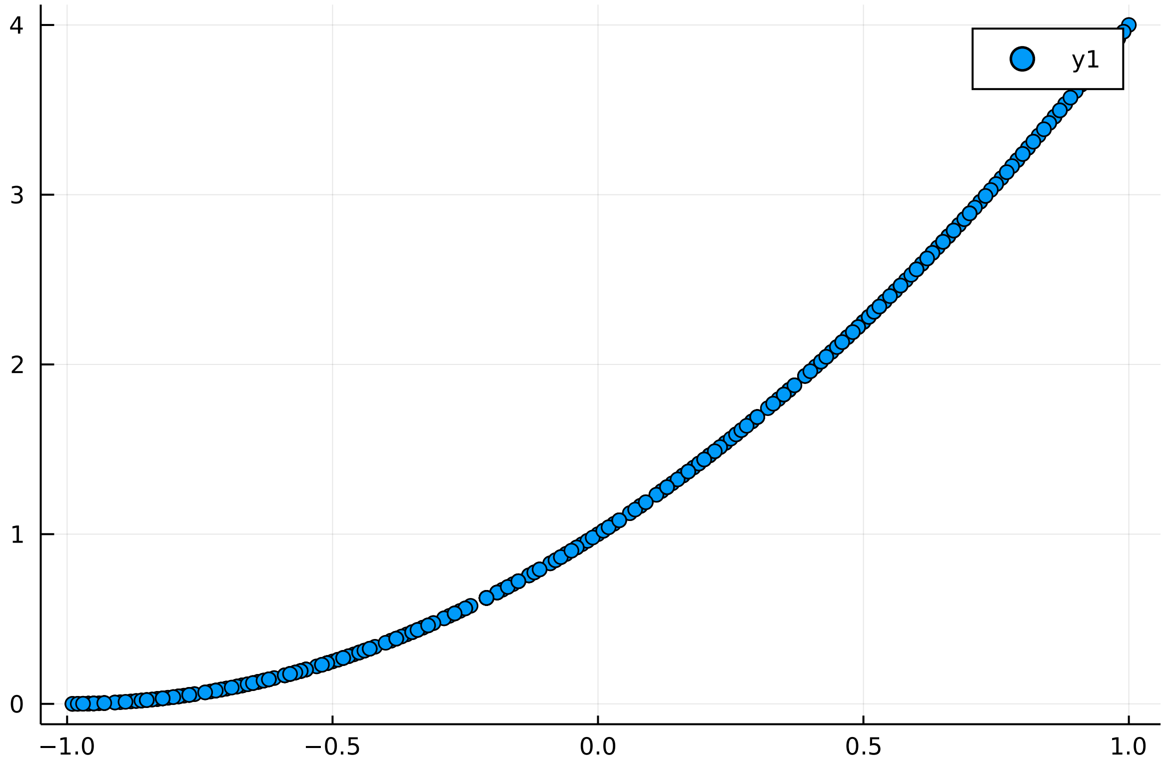
```
begin
    Random.seed!(132)
    z = rand(-1:0.01:1, 500)
    f(x) = x^2 + 2x + 1
    t = f.(z)
end
```

```
weights =
 Weights(6×2 Matrix{Float64}:    , 3×7 Matrix{Float64}:
           0.514008      3.43387     -0.778382    1.54044   -0.297441  -2.00314   0.548547
          -0.00425086   -1.36168      1.20354    -0.801922   0.680602   1.58882  -0.239358
           1.34197      -2.17224      0.0953074   2.71851   -1.84885   -1.61553  -1.30436
          -1.29378      -0.942875
          -1.34565      -2.2294
           0.708612     -2.41011
```

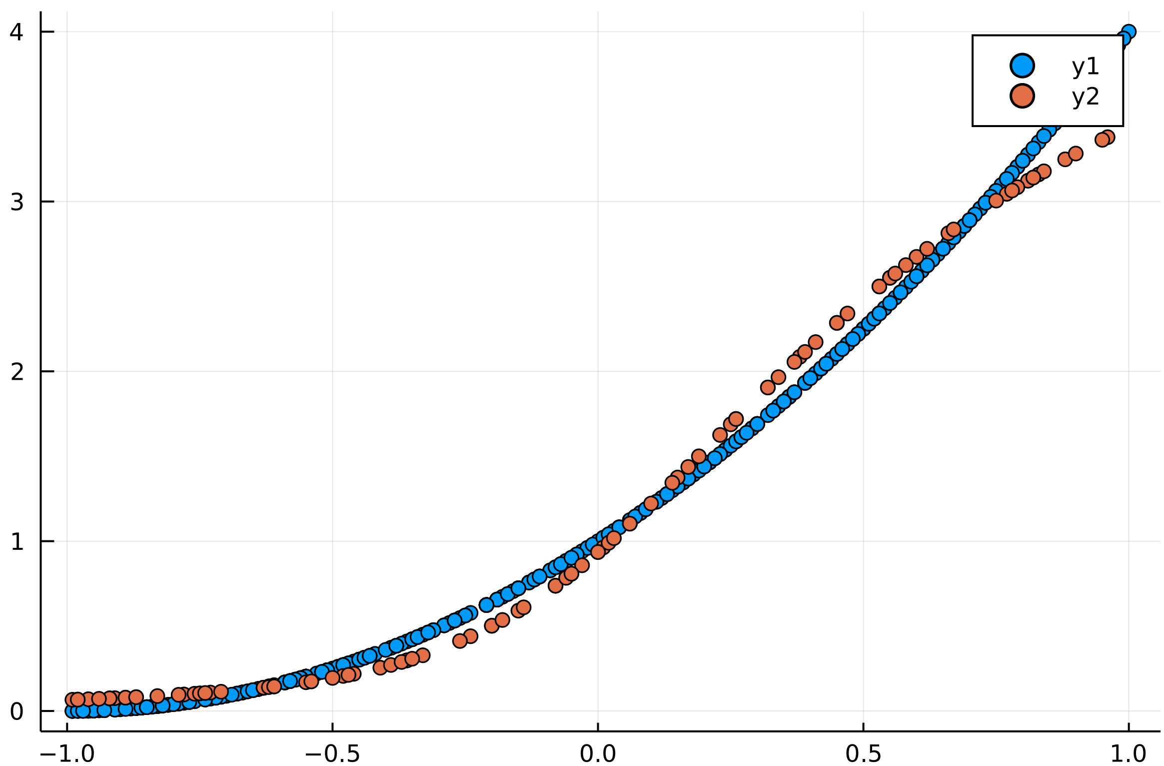
```
weights = TrainNN(z, t, 1000, 5, 0.0001)
```

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

```
function predict(z::Vector{Float64}, weights::Weights)
    o = []
    for z_ in z
        x̄ = weights.W * [1,z_...]
        x = σ.(x̄)
        pushfirst!(x, 1.)
        ȳ = weights.V * x
        y = σ.(ȳ)
        pushfirst!(y, 1.)
        push!(o, (weights.U * y)[1])
    end
    return o
end
```




```julia
scatter(z, f.(z))
```

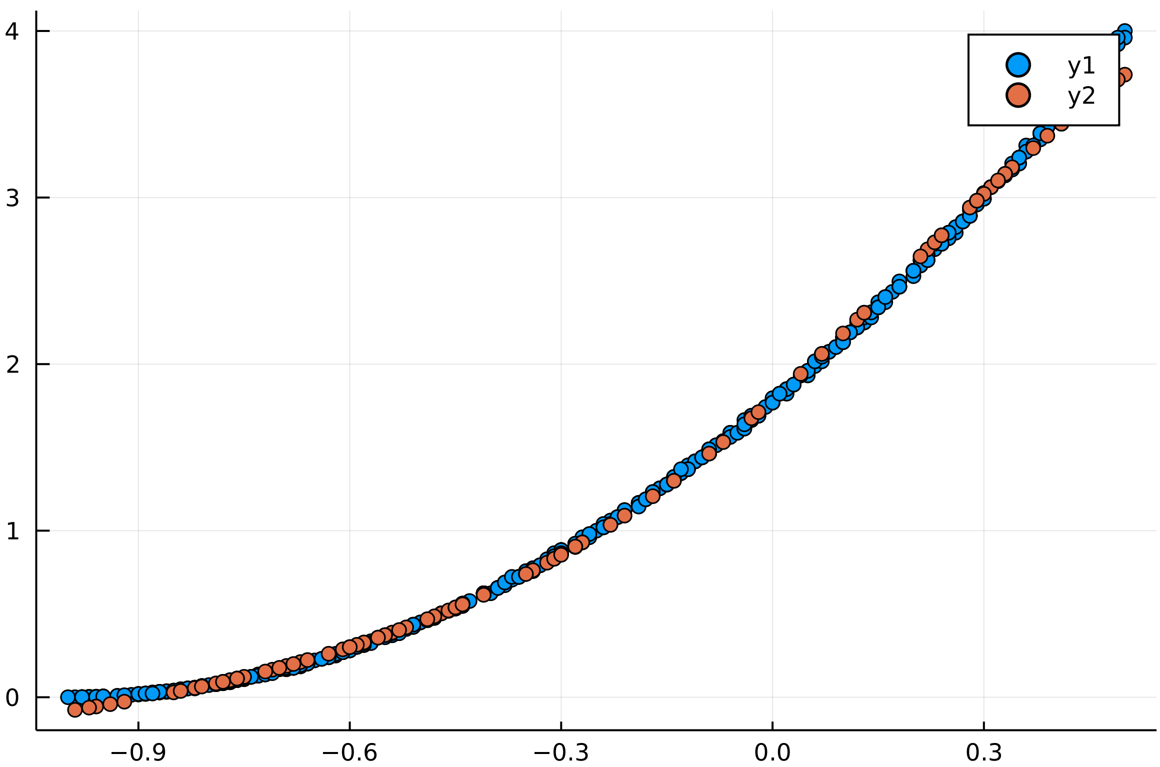



```julia
begin
    zₜₑₛₜ = rand(-1:0.01:1, 100)
    scatter!(zₜₑₛₜ, predict(zₜₑₛₜ, weights))
end
```

Observations

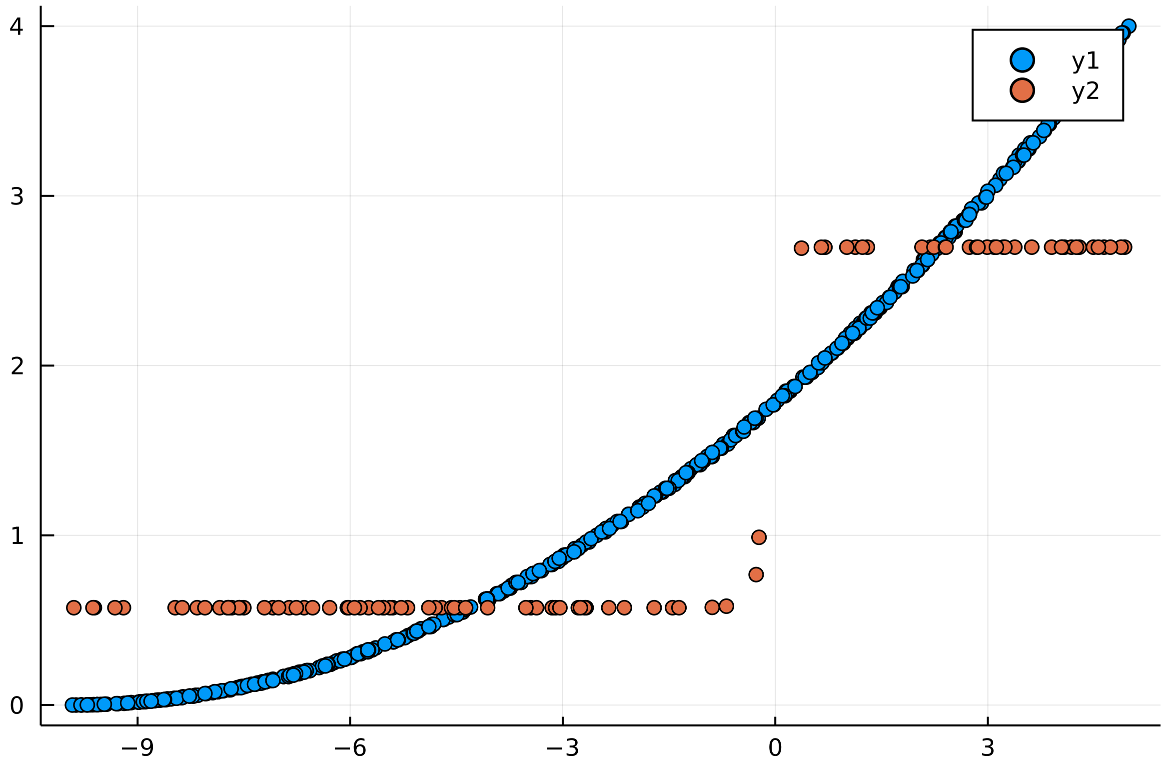
It could be observed from the plot above that the model is fitting the range from (-1 to 1) reasonably well. However this model suffers from overfitting when ranges more broad than this are used. and the model tends to just predict the average y-value for every point on the x-axis. Below we will try two more ranges one more narrow and one more broad. We will observe that with increased "broadness" the model performs much worse.

```
md"""
# Observations
It could be observed from the plot above that the model is fitting the range from (-1 to 1) reasonably well. However this model suffers from overfitting when ranges more broad than this are used. and the model tends to just predict the average y-value for every point on the x-axis.
Below we will try two more ranges one more narrow and one more broad. We will observe that with increased "broadness" the model performs much worse.
"""
```




```
begin
    begin
        Random.seed!(132)
        z_2 = rand(-1:0.01:.5, 500)
        t_2 = f.(z_2)
    end
    weights2 = TrainNN(z_2, t, 1000, 100, 0.001)
    scatter(z_2, f.(z))
    begin
        zₜₑₛₜ2  = rand(-1:0.01:.5, 100)
        scatter!(zₜₑₛₜ2, predict(zₜₑₛₜ2, weights2))
    end
```

```
end
```




```
begin
    begin
        Random.seed!(132)
        z_3 = rand(-10:0.01:5, 500)
        t_3 = f.(z_3)
    end
    weights3 = TrainNN(z_3, t, 1000, 100, 0.001)
    scatter(z_3, f.(z))
    begin
        zₜₑₛₜ3  = rand(-10:0.01:5, 100)
        scatter!(zₜₑₛₜ3, predict(zₜₑₛₜ2, weights3))
    end
end
```