National University of Technology

Computer Science Department

Semester Spring– 2025

Program: Artificial intelligence

Course: Programming for AI

Course Code: CS282

Assignment - 03

Submitted To:

Umar Aftab

Submitted By:

Muhammad Ahad Imran

F23607034

```
julia> iris = dataset("datasets", "iris")
150×5 DataFrame
 Row │ SepalLength  SepalWidth  PetalLength  PetalWidth  Species
     │ Float64      Float64     Float64      Float64     Cat…
─────┼──────────────────────────────────────────────────────────────
   1 │         5.1         3.5          1.4         0.2  setosa
   2 │         4.9         3.0          1.4         0.2  setosa
   3 │         4.7         3.2          1.3         0.2  setosa
   4 │         4.6         3.1          1.5         0.2  setosa
   5 │         5.0         3.6          1.4         0.2  setosa
   6 │         5.4         3.9          1.7         0.4  setosa
   7 │         4.6         3.4          1.4         0.3  setosa
   8 │         5.0         3.4          1.5         0.2  setosa
   9 │         4.4         2.9          1.4         0.2  setosa
  10 │         4.9         3.1          1.5         0.1  setosa
  11 │         5.4         3.7          1.5         0.2  setosa
  ⋮  │      ⋮           ⋮            ⋮           ⋮          ⋮
 141 │         6.7         3.1          5.6         2.4  virginica
 142 │         6.9         3.1          5.1         2.3  virginica
 143 │         5.8         2.7          5.1         1.9  virginica
 144 │         6.8         3.2          5.9         2.3  virginica
 145 │         6.7         3.3          5.7         2.5  virginica
 146 │         6.7         3.0          5.2         2.3  virginica
 147 │         6.3         2.5          5.0         1.9  virginica
 148 │         6.5         3.0          5.2         2.0  virginica
 149 │         6.2         3.4          5.4         2.3  virginica
 150 │         5.9         3.0          5.1         1.8  virginica
                                            129 rows omitted
```

```
julia> X = Matrix(iris[:, 1:4])'  # size: (4, 150)
4×150 adjoint(::Matrix{Float64}) with eltype Float64:
 5.1  4.9  4.7  4.6  5.0  5.4  4.6  5.0  4.4  4.9  5.4  4.8  …  6.9  6.7  6.9  5.8  6.8  6.7  6.7  6.3  6.5  6.2  5.9
 3.5  3.0  3.2  3.1  3.6  3.9  3.4  3.4  2.9  3.1  3.7  3.4     3.1  3.1  3.1  2.7  3.2  3.3  3.0  2.5  3.0  3.4  3.0
 1.4  1.4  1.3  1.5  1.4  1.7  1.4  1.5  1.4  1.5  1.5  1.6     5.4  5.6  5.1  5.1  5.9  5.7  5.2  5.0  5.2  5.4  5.1
 0.2  0.2  0.2  0.2  0.2  0.4  0.3  0.2  0.2  0.1  0.2  0.2     2.1  2.4  2.3  1.9  2.3  2.5  2.3  1.9  2.0  2.3  1.8
```

```
julia> labels = iris[:, :Species]
150-element CategoricalArrays.CategoricalArray{String,1,UInt8}:
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 ⋮
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
```

```
julia> label_map = Dict("setosa" => 1, "versicolor" => 2, "virginica" => 3)
Dict{String, Int64} with 3 entries:
  "virginica"  => 3
  "setosa"     => 1
  "versicolor" => 2
```

```
julia> y_int = [label_map[string(lbl)] for lbl in labels]
150-element Vector{Int64}:
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 ⋮
 3
 3
```

```
julia> Y = Flux.onehotbatch(y_int, 1:3)
3×150 OneHotMatrix(::Vector{UInt32}) with eltype Bool:
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  …  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

```
julia> X = (X .- mean(X, dims=2)) ./ std(X, dims=2)
4×150 Matrix{Float64}:
 -0.897674  -1.1392     -1.38073   -1.50149    -1.01844  …   1.03454    0.551486   0.793012  0.430722   0.0684325
  1.0156    -0.131539    0.327318   0.0978893   1.24503     -0.131539  -1.27868   -0.131539  0.786174  -0.131539
 -1.33575   -1.33575    -1.3924    -1.2791     -1.33575      0.816859   0.703564   0.816859  0.930154   0.760211
 -1.31105   -1.31105    -1.31105   -1.31105    -1.31105      1.44399    0.919223   1.05042   1.44399    0.788031
```

```
julia> Random.seed!(69)
TaskLocalRNG()
```

```
julia> datasett = [(X[:, i], Y[:, i]) for i in 1:size(X, 2)]
150-element Vector{Tuple{Vector{Float64}, OneHotArrays.OneHotVector{UInt32}}}:
 ([-0.8976738791967663, 1.015601990713633, -1.3357516342415194, -1.311052148205131], [1, 0, 0])
 ([-1.1392004834649534, -0.1315388120502606, -1.3357516342415194, -1.311052148205131], [1, 0, 0])
 ([-1.3807270877331417, 0.32731750905529733, -1.3923992862449763, -1.311052148205131], [1, 0, 0])
 ([-1.5014903898672363, 0.09788934850251835, -1.2791039822380623, -1.311052148205131], [1, 0, 0])
 ([-1.01843718133086, 1.2450301512664121, -1.3357516342415194, -1.311052148205131], [1, 0, 0])
 ([-0.5353839727944835, 1.9333146329247481, -1.165808678231148, -1.0486667949952981], [1, 0, 0])
 ([-1.5014903898672363, 0.7861738301608542, -1.3357516342415194, -1.1798594716002144], [1, 0, 0])
 ([-1.01843718133086, 0.7861738301608542, -1.2791039822380623, -1.311052148205131], [1, 0, 0])
 ([-1.7430169941354234, -0.36096697260303956, -1.3357516342415194, -1.311052148205131], [1, 0, 0])
 ([-1.1392004834649534, 0.09788934850251835, -1.2791039822380623, -1.4422482481008472], [1, 0, 0])
 ([-0.5353839727944835, 1.4744583118191912, -1.2791039822380623, -1.311052148205131], [1, 0, 0])
 ([-1.2599637855990482, 0.7861738301608542, -1.2224563302346052, -1.311052148205131], [1, 0, 0])
 ([-1.2599637855990482, -0.1315388120502606, -1.3357516342415194, -1.4422482481008472], [1, 0, 0])
 ⋮
 ([0.18919584001008014, -0.1315388120502606, 0.590268533876023, 0.7880306774735305], [0, 0, 1])
 ([1.2760655592169265, 0.09788934850251835, 0.9301544458967661, 1.1816087072882795], [0, 0, 1])
 ([1.0345389549487385, 0.09788934850251835, 1.04344974990368, 1.5751867371030284], [0, 0, 1])
 ([1.2760655592169265, 0.09788934850251835, 0.7602114898863943, 1.4439940604981119], [0, 0, 1])
 ([-0.05233076425810807, -0.8198232937085965, 0.7602114898863943, 0.9192233540784467], [0, 0, 1])
```

```
julia> train_data, test_data = splitobs(shuffleobs(datasett), at=0.8)
(Tuple{Vector{Float64}, OneHotArrays.OneHotVector{UInt32}}[([0.18919584001008014, 0.7861738301608542, 0.4203255778656516
7, 0.5256453242636979], [0, 1, 0]), ([2.2421719762896783, -0.1315388120502606, 1.3266880099209655, 1.4439940604981119],
[0, 0, 1]), ([0.18919584001008014, -1.9669640964724904, 0.137087317848366, -0.2615107353658002], [0, 1, 0]), ([1.0345389
549487385, 0.5567456696080753, 1.1000974019071375, 1.1816087072882795], [0, 0, 1]), ([-0.8976738791967663, 1.01560199071
3633, -1.3357516342415194, -1.311052148205131], [1, 0, 0]), ([0.7930123506805502, -0.1315388120502606, 0.986802097900223
, 0.7880306774735305], [0, 0, 1]), ([-0.05233076425810807, -0.8198232937085965, 0.19373496985182292, -0.2615107353658002
], [0, 1, 0]), ([0.06843253787598658, 0.32731750905529733, 0.590268533876023, 0.7880306774735305], [0, 1, 0]), ([0.18919
584001008014, -0.36096697260303956, 0.42032557786565167, 0.39445264765878146], [0, 1, 0]), ([0.5514857464123619, -0.8198
232937085965, 0.6469161858794804, 0.7880306774735305], [0, 0, 1])  …  ([-0.5353839727944835, 1.9333146329247481, -1.3923
992862449763, -1.0486667949952981], [1, 0, 0]), ([-1.7430169941354234, -0.36096697260303956, -1.3357516342415194, -1.311
052148205131], [1, 0, 0]), ([-1.01843718133086, -2.4258204175780484, -0.14615094216891966, -0.2615107353658002], [0, 1,
0]), ([-0.05233076425810807, -0.8198232937085965, 0.7602114898863943, 0.9192233540784467], [0, 0, 1]), ([2.4836985805578
666, 1.703886472371969, 1.4966309659313373, 1.0504160306833632], [0, 0, 1]), ([0.9137756528146437, -0.1315388120502606,
0.3636779258621947, 0.2632599710538650], [0, 1, 0]), ([0.18919584001008014, -0.1315388120502606, 0.590268533876023, 0.788
0306774735305], [0, 0, 1]), ([1.0345389549487385, -0.1315388120502606, 0.7035638378829373, 0.6568380008686141], [0, 1, 0
```

```
julia> model = Chain(
           Dense(4, 16, relu),
               Dense(16, 8, relu),
                   Dense(8, 3),
                       softmax
                       )
Chain(
  Dense(4 => 16, relu),                 # 80 parameters
  Dense(16 => 8, relu),                 # 136 parameters
  Dense(8 => 3),                        # 27 parameters
  softmax,
)                   # Total: 6 arrays, 243 parameters, 1.254 KiB.
```

```
julia> loss(x, y) = Flux.crossentropy(model(x), y)
loss (generic function with 1 method)
```

```
julia> accuracy(x, y) = mean(Flux.onecold(model(x)) .== Flux.onecold(y))
accuracy (generic function with 1 method)
```

```
julia> opt_state = Flux.setup(Adam(), model)
(layers = ((weight = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32[0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0;
 … ; 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0], Float32[0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0; … ; 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0],
 (0.9, 0.999))), bias = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32[0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], Float32[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0], (0.9, 0.999))), σ = ()), (weight = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32
[0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0; … ; 0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0], Float32[0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0
.0 0.0; … ; 0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0], (0.9, 0.999))), bias = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilo
n=1.0e-8), (Float32[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], Float32[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], (0.9, 0.99
9))), σ = ()), (weight = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32[0.0 0.0 … 0.0 0.0; 0.0 0.0 …
0.0 0.0; 0.0 0.0 … 0.0 0.0], Float32[0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0], (0.9, 0.999))), bias = Le
af(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32[0.0, 0.0, 0.0], Float32[0.0, 0.0, 0.0], (0.9, 0.999))),
σ = ()), ()),)
```

```
julia> epochs = 50
50

julia> train_loss = Float64[]
Float64[]

julia> train_acc = Float64[]
Float64[]
```

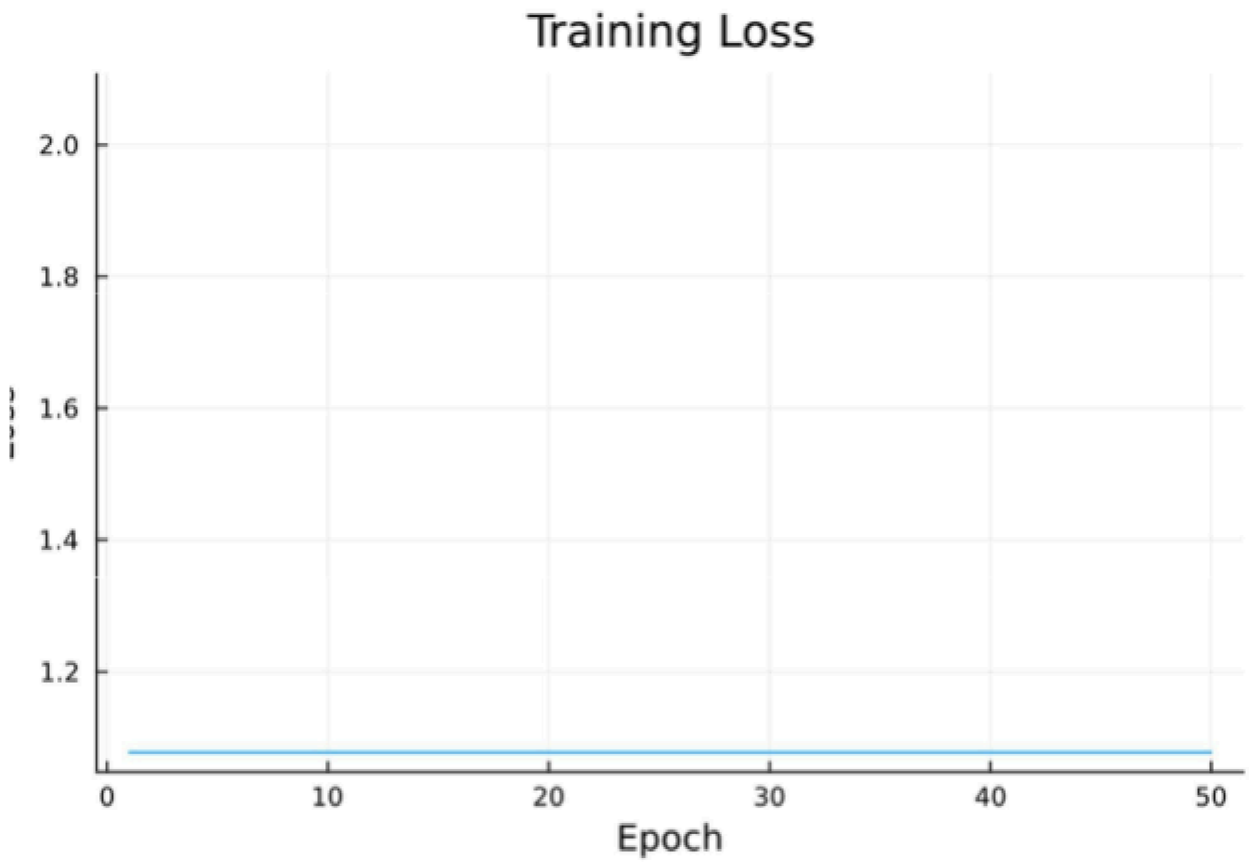
```
julia> for epoch in 1:epochs
           for (x, y) in train_data
                   x = Float32.(x)  # Ensure correct type
                           y = Float32.(y)
                                   gs = Flux.gradient(model) do m
                   loss(x, y)
               end
               Flux.update!(opt_state, model, gs[1])
                   end
           l = loss(hcat(first.(train_data)...), hcat(last.(train_data)...))
               a = accuracy(hcat(first.(train_data)...), hcat(last.(train_data)...))
                   push!(train_loss, l)
           push!(train_acc, a)
           println("Epoch $epoch - Loss: $(round(l, digits=4)) | Accuracy: $(round(a * 100, digits=2))%")
           end
```

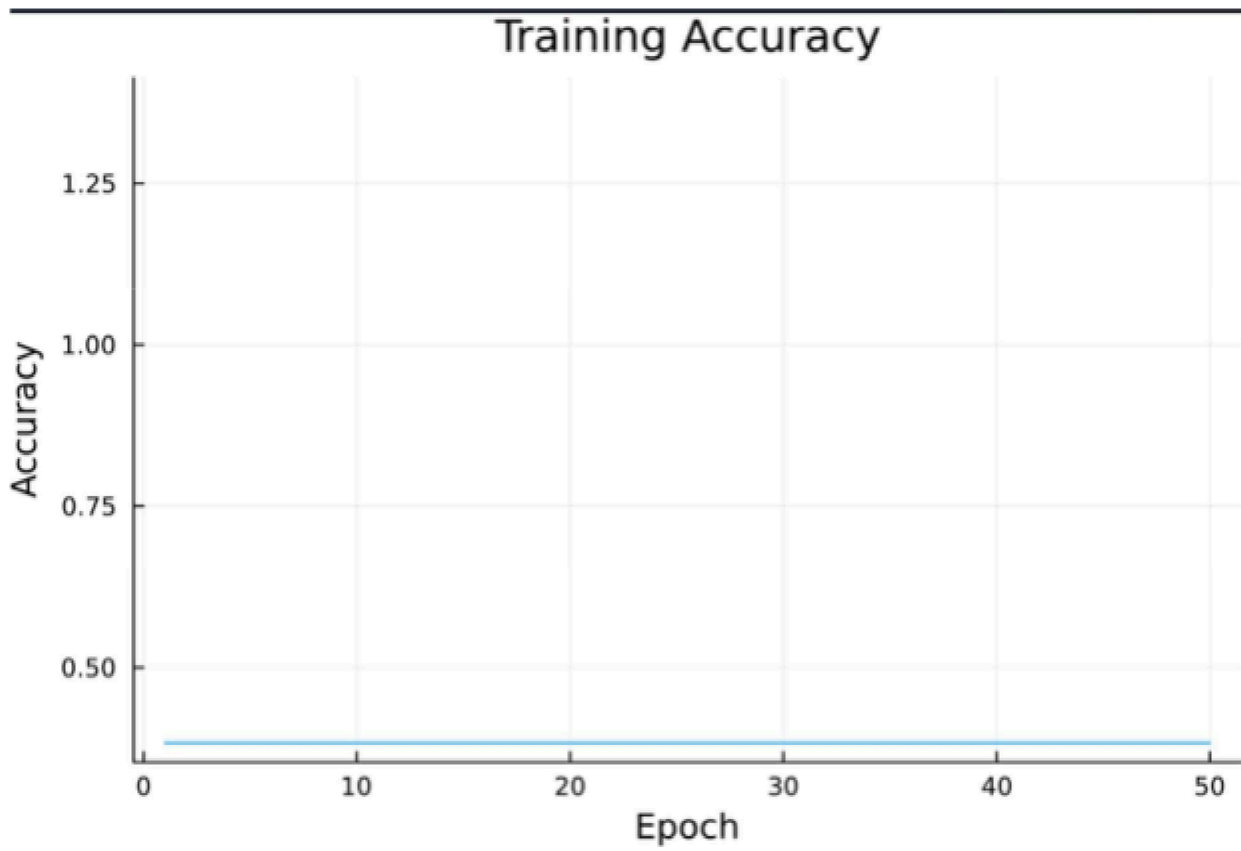
```
Epoch 1 - Loss: 1.078 | Accuracy: 38.33%
Epoch 2 - Loss: 1.078 | Accuracy: 38.33%
Epoch 3 - Loss: 1.078 | Accuracy: 38.33%
Epoch 4 - Loss: 1.078 | Accuracy: 38.33%
Epoch 5 - Loss: 1.078 | Accuracy: 38.33%
Epoch 6 - Loss: 1.078 | Accuracy: 38.33%
Epoch 7 - Loss: 1.078 | Accuracy: 38.33%
Epoch 8 - Loss: 1.078 | Accuracy: 38.33%
Epoch 9 - Loss: 1.078 | Accuracy: 38.33%
Epoch 10 - Loss: 1.078 | Accuracy: 38.33%
Epoch 11 - Loss: 1.078 | Accuracy: 38.33%
Epoch 12 - Loss: 1.078 | Accuracy: 38.33%
Epoch 13 - Loss: 1.078 | Accuracy: 38.33%
Epoch 14 - Loss: 1.078 | Accuracy: 38.33%
Epoch 15 - Loss: 1.078 | Accuracy: 38.33%
Epoch 16 - Loss: 1.078 | Accuracy: 38.33%
Epoch 17 - Loss: 1.078 | Accuracy: 38.33%
Epoch 18 - Loss: 1.078 | Accuracy: 38.33%
Epoch 19 - Loss: 1.078 | Accuracy: 38.33%
Epoch 20 - Loss: 1.078 | Accuracy: 38.33%
Epoch 21 - Loss: 1.078 | Accuracy: 38.33%
Epoch 22 - Loss: 1.078 | Accuracy: 38.33%
Epoch 23 - Loss: 1.078 | Accuracy: 38.33%
```

```
julia> test_X = hcat(first.(test_data)...)
4×30 Matrix{Float64}:
 -1.1392     0.0684325  -0.897674  0.551486   0.430722  …  0.551486  -1.74302   -0.173094   0.551486  -0.776911
 -1.27868   -0.131539   -1.27868   0.556746  -1.96696      0.786174  -0.131539  -0.590395  -0.590395   0.786174
  0.420326   0.760211   -0.429389  0.533621   0.420326     1.04345   -1.3924     0.193735   0.760211  -1.33575
  0.656838   0.788031   -0.130318  0.525645   0.394453     1.57519   -1.31105    0.132067   0.394453  -1.31105
```

```
julia> test_Y = hcat(last.(test_data)...)
3×30 OneHotMatrix(::Vector{UInt32}) with eltype Bool:
 ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅  1
 ⋅  ⋅  1  1  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1  1  1  1  1  ⋅  ⋅  1  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅
 1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  1  1  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  1  1  ⋅  ⋅  ⋅  1  1  ⋅  ⋅  1  ⋅
```

```
julia> println("Test Accuracy: ", round(accuracy(test_X, test_Y) * 100, digits=2), "%"
Test Accuracy: 16.67%
```


Training Loss
2.0
1.8
1.6
1.4
1.2
0
10
20
30
40
50
Epoch


Training Accuracy
Accuracy
1.25
1.00
0.75
0.50
0
10
20
30
40
50
Epoch

How Flux.jl Simplifies Deep Learning

Flux.jl provides a clean and intuitive way to define models using `Chain` and `Dense` layers. It supports automatic differentiation via Zygote.jl, ensuring seamless backpropagation. Additionally, GPU acceleration is built-in—just move the model and data to the GPU without modifying code.

Key Features

Automatic Differentiation (AD)

Definition: Automatic differentiation is a computational technique for efficiently and accurately evaluating derivatives (gradients) of functions. It works by decomposing functions into elementary operations and applying the chain rule.

Key Points in Flux.jl:

- Utilizes Zygote.jl for AD, performing source-to-source transformation to compute gradients.
- Supports dynamic AD—works seamlessly with native Julia control flow (e.g., loops, conditionals).
- Eliminates the need to manually construct a computational graph.
- Essential for training models using gradient descent, backpropagation, and optimization algorithms.

Benefits:

- Reduces human error in derivative calculations.
- Enables easy experimentation with complex model architectures.
- Supports differentiation through any differentiable Julia code.

GPU Acceleration

Definition: GPU acceleration offloads compute-intensive operations (such as matrix multiplications and convolutions) to the Graphics Processing Unit (GPU), which is optimized for parallel computation.

Key Points in Flux.jl:

- Leverages CUDA.jl, an interface for NVIDIA's CUDA API.
- Allows running the entire training loop (model, data, loss, gradients) on the GPU.
- Integrates seamlessly with Flux models—no extensive code modifications required.

Benefits:

- Significantly speeds up deep neural network training, especially for large datasets or complex models.
- Efficiently utilizes memory and computation resources on modern GPUs.
- Fully compatible with Julia, enabling high-performance computing in a streamlined environment.