MNIST digits classification using a dnn with Flux.jl

Classifying MNIST digits dataset with a simple multi-layer-perceptron (MLP)

📖 Table of Contents


```
using PlutoUI
```

```
PlutoUI.TableOfContents(indent=true, depth=4, aside=true)
```

```
# using Pkg; Pkg.activate("."); Pkg.instantiate()
```

```
begin
    using Flux    , Images    , MLDatasets    , Plots
    using CUDA
    using Statistics
    using Random    , Statistics, LinearAlgebra
end
```

```
const USE_CUDA = true
```

```
const USE_CUDA = true
```

Load train and test data

```
((28, 28, 60000), (28, 28, 10000))
```

```
begin
    train_x, train_y = MNIST.traindata(Float32)
    test_x,  test_y  = MNIST.testdata(Float32)

    size(train_x), size(test_x)
end
```

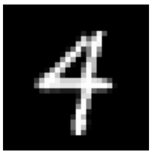
```
TaskLocalRNG()
```

```
Random.seed!(42)
```

Let's check an image, randomly picked in the train set.

```
ix = 59070
```

```
ix = Random.rand(1:size(train_x)[3])
```



```
begin
    img = train_x[:, :, ix]
    colorview(Gray, img')
end
```

Let's check the label for this image.

```
4
```

```
train_y[ix]
```

Prepare the data

As we are going to use a simple ANN for classifying the digits, we need to turn this gray images (of shape 28 x 28 x 1 - 1 being for the unique channel) into a vector of length 28 × 28 = 784.

To achieve this we need to flatten the images using Flux. The same processing needs to be applied for both the train and the test sets.

```
(784×60000 Matrix{Float32}:                                                            , 784×10000 Matrix{
 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    0.0  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  0.0  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  0.0  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  0.0     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  0.0  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  0.0  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     0.0  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  0.0  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  0.0  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.
 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  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  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.
```

```julia
xtrain, xtest = Flux.flatten(train_x), Flux.flatten(test_x)
```

📖 Table of Contents

For the labels we need to onehotencode them. This will be useful during the training stage to evaluate the loss function.

This needs to be apllied on both the train and test labels.

```
(10×60000 OneHotMatrix(::Vector{UInt32}) with eltype Bool:                 , 10×10000 OneHotM
 ⋅  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
ytrain, ytest = Flux.onehotbatch(train_y, 0:9), Flux.onehotbatch(test_y, 0:9)
```

```
const BATCH_SIZE = 32
```

```julia
const BATCH_SIZE = 32
```

```
DataLoader((784×10000 Matrix{Float32}:                                                          , 10×100
             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    ⋅  ⋅
             0.0  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  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    ⋅  1
             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    ⋅  ⋅
             0.0  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  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  0.0  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    ⋅  ⋅
             ⋮                        ⋮              ⋱                 ⋮                          1  ⋅
             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    ⋅  ⋅
             0.0  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  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  0.0  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  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  0.0  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
```

```julia
begin
    train_loader = Flux.Data.DataLoader((xtrain, ytrain), batchsize=BATCH_SIZE, shuffle=true)
    test_loader = Flux.Data.DataLoader((xtest, ytest), batchsize=BATCH_SIZE)
end
```

Defining the model

```
const N_UNITS = [48, 10]
```

```julia
const N_UNITS = [48, 10]  # 48 units in the 1st hidden layer and 10 in the output layer
```

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

```julia
if CUDA.functional() && USE_CUDA
    @info "Training using CUDA GPU"
    CUDA.allowscalar(false)
    device = gpu
else
    @info "Training using CPU"
    device = cpu
end
```

```
Training using CUDA GPU
```

```
(28, 28)
```

```julia
w, h = size(train_x)[1:2]
```

```
model = Chain(
          Dense(784, 48, relu),                 # 37_680 parameters
          Dense(48, 10),                        # 490 parameters
        )                   # Total: 4 arrays, 38_170 parameters, 576 bytes.
```

```
model = Flux.Chain(
    Flux.Dense(w * h, N_UNITS[1], Flux.relu),
    Flux.Dense(N_UNITS...),   # no activation, ouptut the logit...
    # Flux.softmax            # and finally  apply the softnax activation
) |> device
```

Table of Contents

Define the loss

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

```
function loss_acc(model, data_loader, device)
    loss, acc = 0.0, 0.0
    n = 0
    for (x, y) ∈ data_loader
        x, y = device(x), device(y)
        ŷ = model(x)
        loss += Flux.Losses.logitcrossentropy(ŷ, y, agg=sum)
        acc += sum(Flux.onecold(ŷ) .== Flux.onecold(y))
        n += size(x)[end]
    end
    loss / n, acc / n
end
```

Get the parameters of the model

```
prms = Flux.params(model);  # The parameters of the model ≡ 2 weights matrices and 2 bias vectors
```

```
# prms[1] |> size, extrema(prms[1])  # weight matrix for 1st layer
```

```
# prms[2] |> size  # the bias vector for the 1st layer
```

```
# prms[3] |> size, extrema(prms[3]) # the weigth matrix for the 2nd (output) layer
```

```
# prms[4] |> size  # the bias vector for the 2nd (output and last) layer
```

Define the optimizer

```
ADAM(0.01, (0.9, 0.999), 1.0e-8, IdDict())
```

```
begin
    η = 0.01
    opt = Flux.ADAM(η)
end
```

Training the model

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,   more ,0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
```

```
begin
    const EPOCHS = 100
    loss_train_hist, loss_test_hist = fill(zero(Float32), EPOCHS), fill(zero(Float32), EPOCHS)
    acc_train_hist, acc_test_hist = fill(zero(Float32), EPOCHS), fill(zero(Float32), EPOCHS)
end
```

```
 for epoch ∈ 1:EPOCHS
    for (x, y) ∈ train_loader
        x, y = device(x), device(y)
        grads = Flux.gradient(() -> Flux.Losses.logitcrossentropy(model(x), y), prms)
        Flux.Optimise.update!(opt, prms, grads)
    end

    loss_train_hist[epoch], acc_train_hist[epoch] = loss_acc(model, train_loader, device)
    loss_test_hist[epoch], acc_test_hist[epoch] = loss_acc(model, test_loader, device)

    if epoch % 50 == 0
        print("Epoch: $(epoch)")
        println("- train loss: $(loss_train_hist[epoch]) train acc: $(acc_train_hist[epoch]) -
 test loss: $(loss_test_hist[epoch]) test acc: $(acc_test_hist[epoch])")
    end
end
```

```
Epoch: 50- train loss: 0.034383327 train acc: 0.99448335 - test loss: 1.
8785203 test acc: 0.9608
Epoch: 100- train loss: 0.06771462 train acc: 0.99443334 - test loss: 2.9964
797 test acc: 0.9586
```

Let's determine the accuracy of this model

```julia
# begin
#   ŷ_raw = model(xtest)
#   ŷ = Flux.onecold(ŷ_raw) .- 1
# end
```

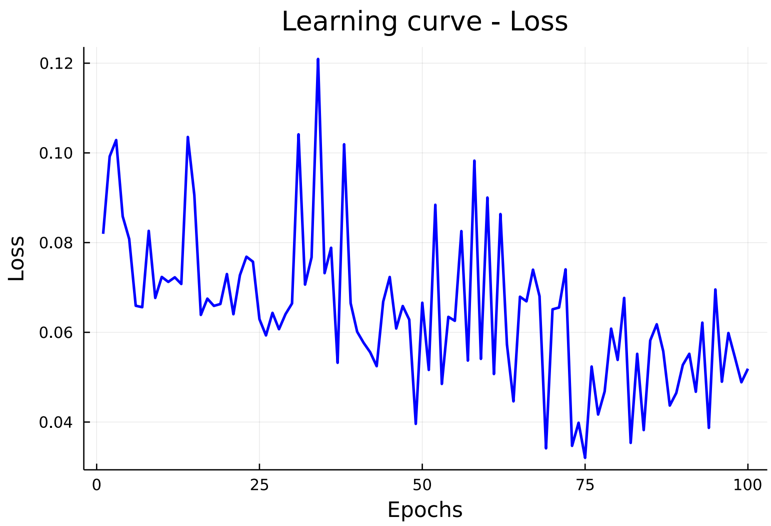
```julia
# begin
#   y = Flux.onecold(ytest) .- 1
#   Statistics.mean(ŷ .== y)
# end
# 0.9608 with 32 units in hidden layer
# 0.9681 with 48 units  "   "     "
```

```julia
# display some results
# check = ŷ .== y
```

```julia
# TODO: check errors
```

📖 Table of Contents

Let's plot the loss (during the training)




```julia
begin
    gr(size = (600, 400))
    loss_curve = plot(
        1:EPOCHS,
        loss_train_hist,
        xlabel = "Epochs",
        ylabel = "Loss",
        title = "Learning curve - Loss",
        legend = false,
        color = :blue,
        linewidth = 2
    )
end
```

```julia
html"""
<style>
  main {
    max-width: calc(800px + 25px + 6px);
  }

  .plutoui-toc.aside {
    background-color: linen;
    color: black;
  }

  h4, h5 {
    background: wheat;
    text-decoration: underline overline dotted darkred;
  }
</style>
"""
```