Table of Contents

In [1]:
```
using Plots, Interact
gr(
    markerstrokewidth=0.5,
    markerstrokecolor=:white,
    alpha=0.7
)
using Flux, Flux.Data.MNIST ## this is the Julia package for deep lea
using Flux: onehotbatch, argmax, crossentropy, throttle, mse
using Base.Iterators: repeated, partition
```

▶ Run

We now consider an example where the two classes are not linearly separable. The function in the next code cell generates two training datasets comprised of N points belonging to Class 1 and N points belonging to Class 2. Points in Class 1 (roughly) lie on a circle with radius `r1`, while points in Class 2 (roughly) lie on a circle with radius `r2`.

In [2]:
```
function generatedata_circle(r1, r2, N, σ=0.1)
    ϕ1 = LinRange(0, 2 * π, N)
    ϕ2 = LinRange(0, 2 * π, N)
    rx1 = r1 .+ σ * randn(N)
    rx2 = r2 .+ σ * randn(N)
    X1 = [rx1 .* cos.(ϕ1) rx1 .* sin.(ϕ1)]
    X2 = [rx2 .* cos.(ϕ2) rx2 .* sin.(ϕ2)]
    return X1', X2'
end
```

▶ Run

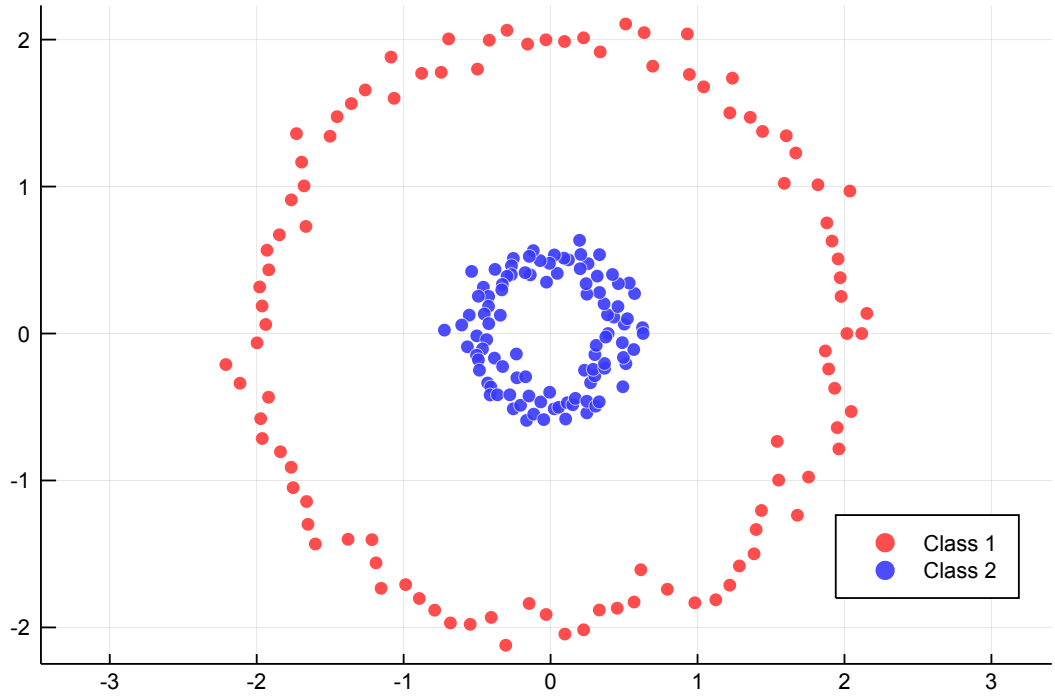
Out[2]:
```
generatedata_circle (generic function with 2 methods)
```

The code in the next cell generates samples for Class 1 and Class 2 and plots them on the 2-D plane.

In [3]:
```
X1c, X2c = generatedata_circle(2, 0.5, 100)
p1 = scatter(
    X1c[1, :], X1c[2, :];
    color="red",
    label="Class 1",
    aspectratio=:equal
)
scatter!(
    X2c[1, :], X2c[2, :];
    color="blue",
    label="Class 2",
    legend=:bottomright
)
```

▶ Run

Out[3]:



2.1 Create the training dataset

Let's create a training dataset by concatenating the datasets together.

In [4]:
```
X = [X1c X2c]
```

▶ Run

Out[4]:
```
2×200 Array{Float64,2}:
 2.11733  2.15228   1.97696   1.97093   1.95683   …   0.375723    0.62958
3
 0.0      0.136781  0.252298  0.379865  0.507728      -0.0238779  -1.54203
e-16
```

Then we associate the classes with labels: Class 1 is "+1", and Class 2 is "-1".

In [5]:
```
Y = [ones(1, size(X1c, 2)) -ones(1, size(X2c, 2))]
```
▶ Run

Out[5]:
```
1×200 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  -1.0  -1.0  -1.0  -1.0  -1.0  -1.0
```

What are the dimensionalities of the input and output of the network we would like to predict `Y` from `X` ? (select **all** that apply)

- [ ] the input has dimensions equal to the number of dimensions of `X` -- or equivalently to `size(X,2)`
- [x] the input has dimensions equal to the number of dimensions of `X[:,i]` -- or equivalently to `size(X,1)`
- [ ] the output has dimensions equal to the number of dimensions of `Y` -- or equivalently to `size(Y,2)`
- [x] the output has dimensions equal to the number of dimensions of `Y[i]` -- or equivalently `size(Y,1)

Submit

2.2 Training a *minimal* single hidden layer, single output neural network with linear activation

Our goal is to design a neural network that takes 2-dimensional vectors from the training dataset for Class 1 and maps them "as close as possible" to "+1", and maps elements of Class 2 "as close as possible" to "-1".

We will begin by using the mean squared error `mse` loss function and a `Dense` network with linear activation function.

The command

```
Dense(10, 2, relu)
```

produces a dense (i.e. fully-connected) layer with 10 input neurons and 2 outputs, and a `relu` activation function for each output neuron. Replacing `relu` with `σ` changes the activation function to the sigmoidal activation function.

The command

```
Dense(10, 2)
```

(without the last argument) produces a fully-connected layer with 10 input neurons and a *linear* activation function.

A single layer neural network with `n` inputs and `d` outputs and a linear activation function can be created using the command?

- [x] `m = Dense(n, d)`
- [ ] `m = Dense(d, n)`

Submit

Other activation functions can be used -- see the documentation (https://github.com/FluxML/Flux.jl/blob/master/docs/src/models/layers.md)

Examples include:

- `σ`
- `relu`
- `leakyrelu`
- `elu`
- `swish`

A single layer neural network with `n` inputs and `d` outputs, and a `sigmoid` activation function, can be created using the command?

- [ ] `m = Dense(n, d)`
- [x] `m = Dense(n, d, σ)`
- [ ] `m = Dense(d, n)`
- [ ] `m = Dense(d, n, σ)`

Submit

Before we train a neural network in `Flux`, we must:

1. create a input-output model
2. create a loss function
3. package the data tuples of input-output against which the loss function can be evaluated

Exercise:

Initialize a single output, linear neural network by filling in the `??` in the cell below.

In [6]:
```
1 m = Chain(Dense(size(X,1), 1)) ## TODO: Replace ?? -- Hint: What shou
```

Run

Out[6]:
```
Chain(Dense(2, 1))
```

The model is initialized with random weights. Let us pass as an input the matrix `X`. If you have set up the model correctly, the output should match the dimension.

In [7]:
```
1 println("Output has size $(size(m(X), 2))")
```

Validate

```
Output has size 200
```

Next we define the loss and loss function.

In [8]:
```
1 loss(x, y) = mse(m(x), y)
```

Run

Out[8]:
```
loss (generic function with 1 method)
```

We now package the dataset into `(X,Y)` tuples.

In [9]:
```
1 iters = 500
2 dataset = repeated((X, Y), iters)
```

Run

Out[9]:
```
Base.Iterators.Take{Base.Iterators.Repeated{Tuple{Array{Float64,2},Array
{Float64,2}}}}(Base.Iterators.Repeated{Tuple{Array{Float64,2},Array{Float
64,2}}}(([2.11733 2.15228 … 0.375723 0.629583; 0.0 0.136781 … -0.0238779
-1.54203e-16], [1.0 1.0 … -1.0 -1.0])), 500)
```

To train the network, we need to specify an optimizer. Choices are (https://github.com/FluxML/Flux.jl/blob/master/docs/src/training/optimisers.md)

- `Descent`
- `Momentum`
- `Nesterov`
- `ADAM`

(Most of these should sound familiar from what we've learned previously!) The learning rates for these optimizers can be set explicitly or implicitly. To use ADAM, for example:

```
opt = ADAM()
```

The syntax to train a model is

```
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.01
))
```

where the `cb=throttle(evalcb, 0.01)` sets a "callback" which shows the output of the `evalcb` function every 0.01 seconds.

We are now ready to train the network with the defined loss function and the `ADAM` optimizer as below.

In [10]:
```
opt = ADAM()
evalcb = () -> @show([loss(X, Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.01))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[2.0676]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.9934]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.91748]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.84603]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.77176]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.70488]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.64468]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.59247]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.54004]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.49303]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.45719]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.41756]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.37263]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.33292]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.32806]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.2923]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.25882]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.22856]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.21457]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.20456]
```

We now display the decision boundaries associated with the learned neural network. To that end, we will create the function in the next cell.

In [11]:
```
function display_decision_boundaries(X1c, X2c, m, x1range, x2range, τ
    D = [(m([x1; x2]).data)[1] for x2 in x2range, x1 in x1range]
    heatmap(x1range, x2range, sign.(D .- τ); color=:grays, xlim = [mi
    scatter!(X1c[1, :], X1c[2, :], color="red", label="Class 1", aspe
    scatter!(X2c[1, :], X2c[2, :], color="blue", label="Class 2")
end
```

▶ Run

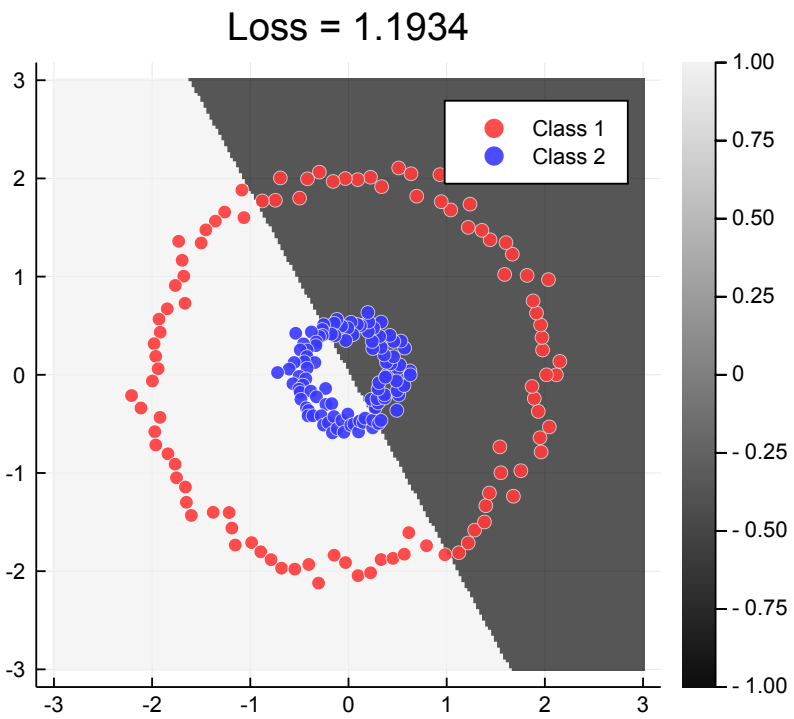
Out[11]:
```
display_decision_boundaries (generic function with 2 methods)
```

We now display the decision boundary. The elements in white correspond to the portion of the 2-D plane that we assign to the label "+1", *i.e.*, Class 1. The elements in black correspond correspond to the portion of the 2-D plane that we assign to the label "-1", *i.e.*, Class 2.

In [12]:
```
x1range = range(-3; stop=3, length=200)
x2range = range(-3; stop=3, length=200)
display_decision_boundaries(X1c, X2c, m, x1range, x2range)
plot!(; title="Loss = $(round(Tracker.data(loss(X,Y)), digits=4))")
```

▶ Run

Out[12]:



Does the network learn to classify the two categories?

- [ ] Yes! Perfectly!
- [x] No!

Submit

Let us try to understand why the network did not properly classify the boundaries, even with more iterations.

Do we expect a linear network to learn good decision boundaries? (select **all** that apply)

- [x] Yes, when the data is linearly separable.
- [x] No, when the data is not linearly separable: a linear network learns only linear separating boundaries.
- [ ] Yes, when the data is non-linearly separable.

Submit

Indeed. We can retreive the parameters learned using the commands in the next cell.

In [13]:
```
@show m.layers[1].W
@show m.layers[1].b
@show m.layers[1].σ;
```

▶ Run

```
(m.layers[1]).W = Float32[-0.361308 -0.196813] (tracked)
(m.layers[1]).b = Float32[0.00507161] (tracked)
(m.layers[1]).σ = identity
```

That the network we designed is a linear network is confirmed by the `(m.layers[1]).σ = identity` output.

2.3 Does changing the activation function from linear to non-linear make a difference?

We now make the network non-linear by utilizing the `relu` activation function. We retain the same structure though.

In [14]:

```
m = Chain(Dense(size(X,1), 1, relu)) ## TODO: Replace ??

loss_fn = mse
loss(x, y) = loss_fn(m(x), y)

iters = 5000
dataset = Base.Iterators.repeated((X, Y), iters)

opt = ADAM()
evalcb = () -> @show([loss(X, Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.1))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.899945]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.855158]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.840551]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.835785]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.834217]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832982]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832385]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832315]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832315]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832315]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.832314]
```

In [15]:

```
println("Output has size $(size(m(X), 2))")
```

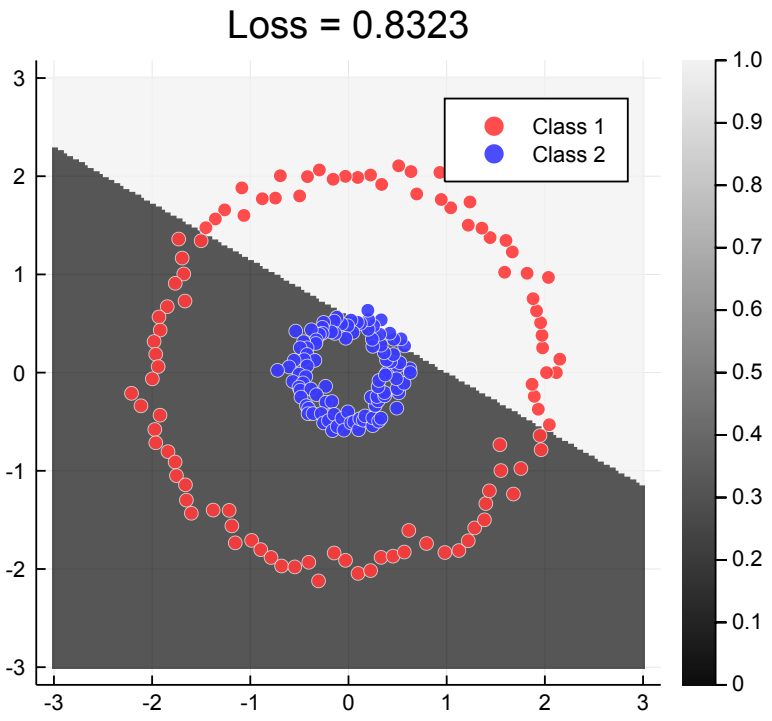
Validate

```
Output has size 200
```

In [16]:
```
display_decision_boundaries(X1c, X2c, m, x1range, x2range)
plot!(; title="Loss = $(round(Tracker.data(loss(X,Y)), digits=4))")
```

▶ Run

Out[16]:



Does the network learn the boundaries?

- [ ] Yes!
- [x] No.

Submit

Given this architecture (= number of layers and neurons per layer), what else can we change?

- [x] the activation function
- [ ] nothing else
- [ ] the number of layers

Submit

The cell below changes the activation function to the sigmoid activation function.

In [17]:

```
m = Chain(Dense(size(X,1), 1, sigmoid)) ## TODO: Replace ??

loss_fn = mse
loss(x, y) = loss_fn(m(x), y)

iters = 5000
dataset = Base.Iterators.repeated((X, Y), iters)

opt = ADAM()
evalcb = () -> @show([loss(X,Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.1))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.32428]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.24488]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.19209]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.13955]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.10928]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.09213]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.07439]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.06008]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.04657]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.04188]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.02183]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.00024]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.98548]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.972736]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.96353]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.957351]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.950015]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.944304]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.938508]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.934236]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.928977]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.923929]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.91972]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.915511]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.911525]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.906785]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.902243]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.897892]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.893976]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.891039]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.887534]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.884392]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.881268]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.878389]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.876136]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.873097]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.86779]
```

In [18]:
```
1  println("Output has size $(size(m(X), 2))")
```

Validate

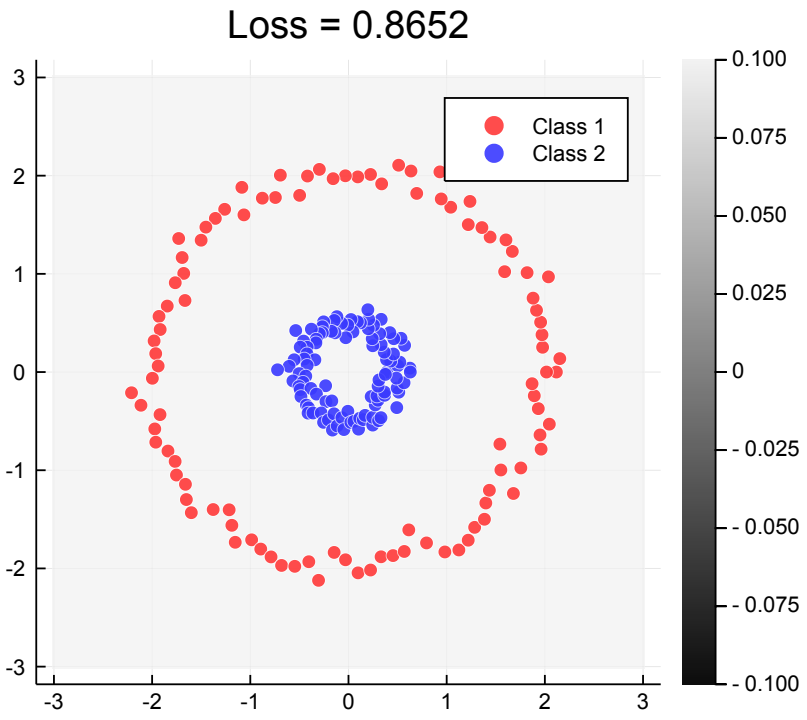
```
Output has size 200
```

We now display the decision boundary.

In [19]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range)
2  plot!(; title="Loss = $(round(Tracker.data(loss(X,Y)), digits=4))")
```

Run

Out[19]:



Does the network learn?

- [ ] Yes!
- [x] No (sigh)

Submit

What else could we change? (select **all** the apply)

- [ ] Nothing, we have tried everything...
- [x] The architecture -- we can add additional hidden layers
- [x] The activation function in every new hidden layer
- [x] The loss fuction, by going beyond `mse`

- [x] The optimizer and the implicit or explicit learning rate
- [ ] We could try clicking "Run" with our lucky left pinky finger...

Submit

2.4 Training a neural network with multiple hidden layer neurons

We now train a deeper network. We shall do so by utilizing the `Chain` command. The command

```
Chain(Dense(10, 2, relu), Dense(2, 2))
```

chains a Dense 10x2 layer with a relu activation function with a Dense 2x2 layer with a linear activation function.

Important: The number of input neurons of a layer has to match the number of output networks of the preceding layer!

For example,

```
Chain(Dense(2, 10, relu), Dense(2, 2))
```

does not constitute a valid network.

The following commands produce valid deep neural networks? (select **all** that apply)

- [ ] `Chain(Dense(2, 10, relu), Dense(2, 2))`
- [x] `Chain(Dense(10, 2, relu), Dense(2, 2))`
- [x] `Chain(Dense(10, 100, relu), Dense(100, 2, relu))`
- [x] `Chain(Dense(10, 100, relu), Dense(100, 10, relu), Dense(10, 2, relu))`

Submit

Exercise:

Design a network with the first hidden layer containing n neurons and a sigmoidal activation function, that connects to an output layer containing one neuron and a linear activation function. Change the number of neurons till the loss function is small enough. Be careful not to set the number too high at first -- you could be waiting a long time!

In [20]:
```
n = 4  ## TODO: number of neurons in hidden layer
active_fun = σ

m = Chain(Dense(size(X,1), n, active_fun),Dense(n, size(Y,1))) ##TODO

loss_fn = mse
loss(x, y) = loss_fn(m(x), y)

iters = 8000
dataset = Base.Iterators.repeated((X, Y), iters)

opt = ADAM()

evalcb = () -> @show([loss(X, Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.5))

lossXY = loss(X,Y).data
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[2.09534]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.02126]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.00283]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.00047]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.999243]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.972181]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.831861]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.62208]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.489656]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.406925]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.299537]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.191706]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0962953]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0397113]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0221321]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0110262]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00823905]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00753718]
```

Out[20]:
```
0.007520979867181552
```

In [21]:
```
println("Training loss is $(lossXY[end])")
```

Validate

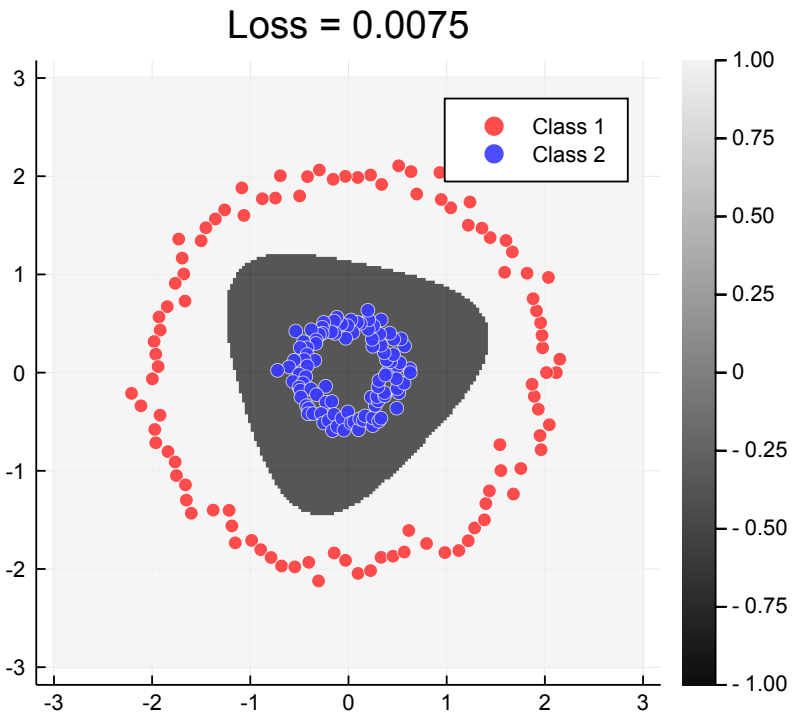
```
Training loss is 0.007520979867181552
```

We now display the decision boundaries.

In [22]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range)
2  plot!(; title="Loss = $(round(lossXY[end], digits=4))")
```

▶ Run

Out[22]:



Does the network learn decision boundaries that separate the red points from the blue points?

- [x] Yes :-)
- [ ] Stop asking!

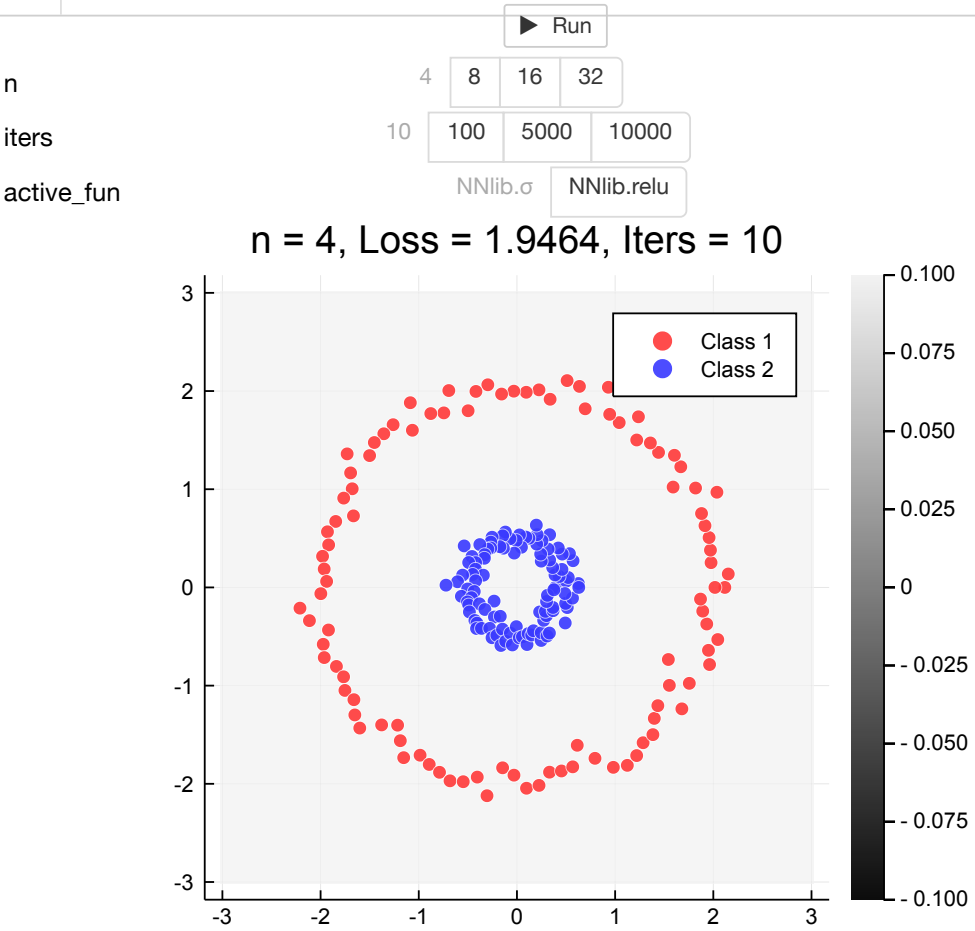
Submit

In the next cell we examine how the learning depends on the parameters such as the number of neurons, the number of iterations for which we run the algorithm and the choice of activation function.

In [23]:

```
loss_fn = mse
@manipulate for n in (4, 8, 16, 32), iters in (10, 100, 5000, 10000),
    m = Chain(Dense(size(X,1), n, active_fun), Dense(n, 1)) ##TODO: R

    loss(x, y) = loss_fn(m(x), y)
    dataset = Base.Iterators.repeated((X, Y), iters)

    opt = ADAM()
    Flux.train!(loss, params(m), dataset, opt)

    display_decision_boundaries(X1c, X2c, m, x1range, x2range)

    lossXY = loss(X,Y).data
    plot!(; title="n = $n, Loss = $(round(lossXY, sigdigits = 5)), It
end
```

Out[23]:



Does the decision boundary always nicely partition the red and blue points? (select **all** that apply)

- [x] Usually not when `n = 4` and `iters = 10`, regardless of `active_fun`.
- [ ] Always, for every value of `n`, `iters` and `active_fun`.
- [x] `n = 8, iters = 5000` always generates a good boundary.
- [x] `n = 4, iters = 100` does **not** produce a nice boundary, but `n = 4` and `iters = 10000` does.
- [x] As the number of neurons increases, and we train it for large enough iterations, the network eventually always learns.

Submit

2.5 Does increasing the number of layers always help?

In the cell below we chain another linear dense layer after the first hidden layer of the previous network.

Exercise

Replace the ?? and pick the smallest value of `n` that makes the network learn.

In [24]:

```
n = 7 ##TODO: Replace ??. Start with 2
iters = 10000

active_fun = σ
m = Chain(Dense(2,n,active_fun),Dense(n, n),Dense(n, 1))

loss_fn = mse
loss(x, y) = loss_fn(m(x), y) ## replace with mse

dataset = Base.Iterators.repeated((X, Y), iters)
opt = ADAM() ## replace with SGD, Nesterov

evalcb = () -> @show([loss(X,Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.5))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.14518]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.943793]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.471007]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0153013]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0113697]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0110124]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0107366]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0105098]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0102472]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00999956]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00978033]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00959448]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00935131]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00917795]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00903012]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0089334]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00889694]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00887212]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00884676]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00882519]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0088044]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00879079]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00877119]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00874006]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00869845]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00857556]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00843042]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00830318]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00817092]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00809054]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00801866]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00796113]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0079319]
```

In [25]:
```
1  lossXY = loss(X,Y).data
2  println("Training loss is $(lossXY)")
```

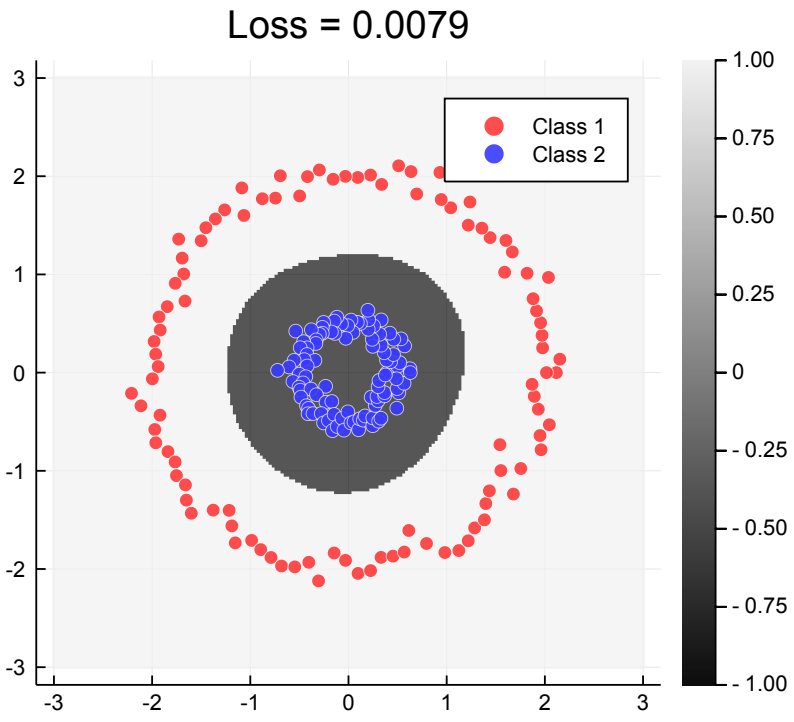
Validate

```
Training loss is 0.007911623492409614
```

In [26]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range)
2  plot!(; title="Loss = $(round(lossXY, digits=4))")
```

Run

Out[26]:



In [27]:
```
1  println("Number of neurons in hidden layer equals $(n)")
```

Validate

```
Number of neurons in hidden layer equals 7
```

What is the smallest `n` for which the network learns?

- [ ] 2
- [x] 3
- [ ] 4

Submit

Now we train the same network with a `relu` activation function in each layer. Increase n till the

network learns -- but do not change the architecture.

In [28]:

```
n = 3 ## TODO: Change till network learns

active_fun = relu

m = Chain(Dense(2, n, active_fun), Dense(n, n, active_fun), Dense(n,

loss_fn = mse
loss(x, y) = loss_fn(m(x), y)

iters = 10000

dataset = Base.Iterators.repeated((X, Y), iters)
opt = ADAM()

evalcb = () -> @show([loss(X, Y)])
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.5))

lossXY = loss(X, Y).data
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[1.0]
```

Out[28]: 1.0

In [29]:
```
1  lossXY = loss(X,Y).data
2  println("Training loss is $(lossXY)")
```

Validate

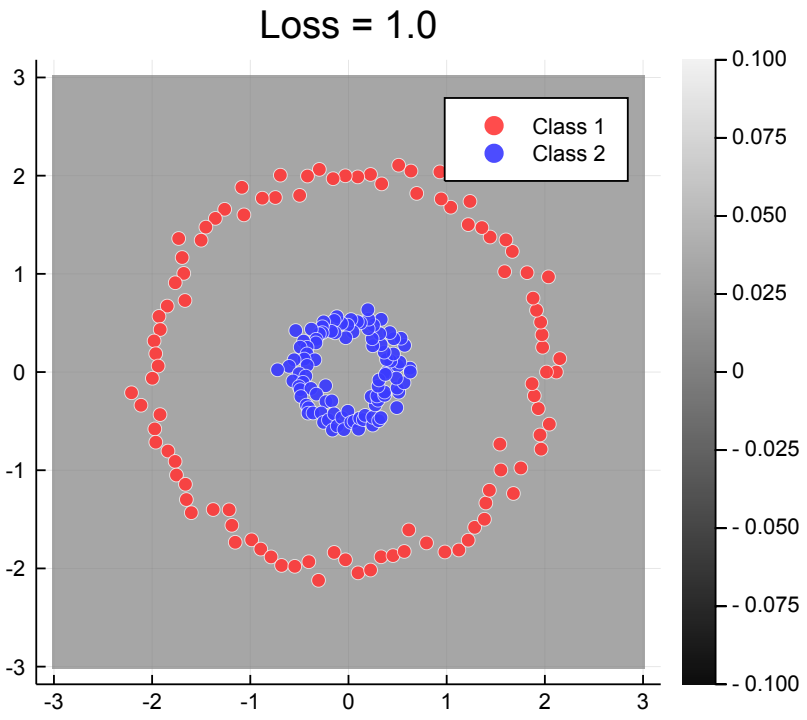
```
Training loss is 1.0
```

We now display the decision boundaries.

In [30]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range)
2  plot!(; title="Loss = $(round(lossXY, digits=4))")
```

Run

Out[30]:



Train the network at least 10 times. The network ? (select **all** that apply)

- ☑ sometimes produces a decision boundary that correctly separates the red and blue points, but the loss may still be high at around 0.5.
- ☑ sometimes does not produce a decision boundary that correctly separates the red and blue points; in these cases the loss is around 1.0.
- ☑ stopped working as consistently as before -- what happened??

Submit

Exercise:

Why does introducing the `relu` in the final layer break this network? This can be answered analytically!

Hint: How is the output encoding of the network related to the encoding of the classes we have chosen?

Since relu function is $max(0, x)$, the output will always be positive, hence it can not recognize class 2 properly. So the loss error is always greater than 0.5.

Edit / Render

Let us change the encoding of the class vector and see what happens.

In [31]:
```
1 Yalt = [zeros(1, size(X1c, 2)) ones(1, size(X2c, 2))]
```

Run

Out[31]:
```
1×200 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  1.0  1.0  1.0  1.0  1.0  1.0
1.0
```

And now let us train the network.

In [32]:

```
n = 3
iters = 10000
active_fun = relu
m = Chain(Dense(size(X,1), n, active_fun),Dense(n, 3, active_fun), De

loss_fn = mse
loss(x, y) = loss_fn(m(x), y)

dataset = Base.Iterators.repeated((X, Yalt), iters)

opt = ADAM()
evalcb = () -> @show([loss(X, Yalt)])

Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.5))
```

Run

```
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.495581]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.0391717]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00957226]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00424886]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00384241]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00382]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
[loss(X, Yalt)] = Tracker.TrackedReal{Float64}[0.00381986]
```

In [33]:

```
lossXY = loss(X,Yalt).data
println("Training loss is $(lossXY)")
```

Validate

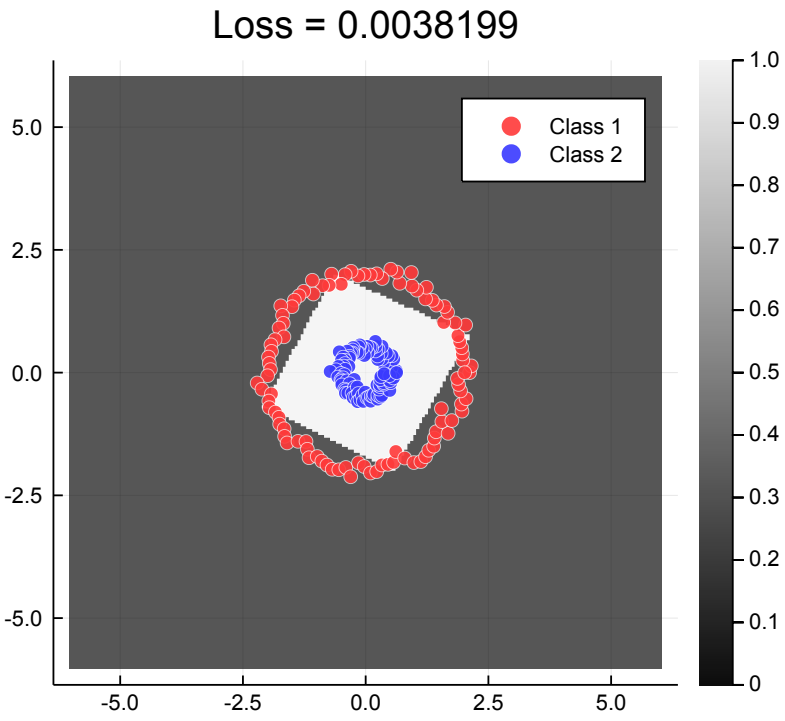
```
Training loss is 0.003819856825579477
```

Now we display the decision boundary (again).

In [34]:
```
display_decision_boundaries(X1c, X2c, m, 2 * x1range, 2 * x2range)
plot!(title = "Loss = $(round(lossXY, sigdigits = 5))")
```

▶ Run

Out[34]:



Hallelujah?

- [x] Amen!

Submit

Why?

- [x] The `relu` produces an non-negative output; when we encoded the class vector as ± 1 the `relu` in the final layer could never approximate the output well. Changing the encoding to $0, 1$ fixed this and the network learned again. **Encodings matter** !
- [ ] `relu` does not like to disappoint -- it is nice like that

Submit

3 Exercise: Design a deep neural network to classify the spiral dataset

We now consider a more complicated two-class classification example.

In [35]:
```
function generatedata_spiral(a, r, n)
    theta = LinRange(0, 4 * pi, n)
    x = zeros(2, n)
    for i in 1:n
        x[1, i] = a * (theta[i]^(1 / r)) * sin(theta[i])
        x[2, i] = a * (theta[i]^(1 / r)) * cos(theta[i])
    end
    return x
end
```

▶ Run

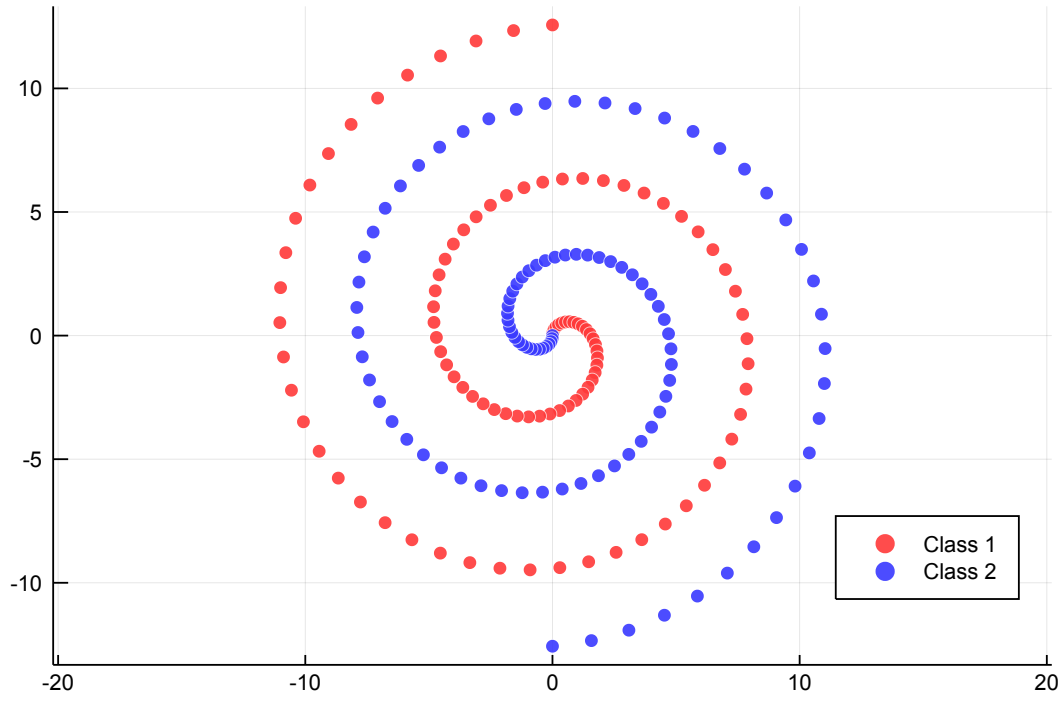
Out[35]:
```
generatedata_spiral (generic function with 1 method)
```

Each class is one arm of an Archimedean spiral (https://en.wikipedia.org/wiki/Archimedean_spiral). We visualize this dataset next.

In [36]:
```
X1c = generatedata_spiral(1, 1, 100)
X2c = generatedata_spiral(-1, 1, 100)
p1 = scatter(X1c[1, :], X1c[2, :]; color="red", label="Class 1", aspe
scatter!(X2c[1, :], X2c[2, :]; color="blue", label="Class 2", legend=
```

▶ Run

Out[36]:



Motivated by our insights from the previous exercise, we will code the class labels differently than "+1" and "-1". This gives us more versatility when chaining together blocks of a neural network. Choose the class labels and the decision boundary threshold accordingly in the next cell.

In [37]:
```
##TODO: Create training dataset
X = [X1c X2c]
Y = [0 * ones(1, size(X1c, 2))  1* ones(1, size(X2c, 2))] ## Fill in
τ =  0.5# decision boundary -- TODO: Fill in ?? based on how you popu
x1range = range(-20; stop=20, length=100)
x2range = range(-20; stop=20, length=100);
```

▶ Run

Exercise

Design a network to classify the spiral dataset.

In [38]:

```
active_fun = relu

n = 15 ## Keep smaller than 16
iters = 10000

m = Chain(Dense(size(X,1), n, active_fun),Dense(n, n, active_fun),Den

loss_fn = mse
loss(x, y) = loss_fn(m(x), y) ## replace with mse

evalcb = () -> @show([loss(X, Y)])
dataset = Base.Iterators.repeated((X, Y), iters)

opt = ADAM()
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 0.5))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.353477]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.206239]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.195398]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.179166]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.169555]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.155091]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.128292]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0365895]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0140936]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00872806]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00602819]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0043592]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00343008]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00303806]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00283268]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00273311]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00275831]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00258829]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00258091]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00258044]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00256586]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00256663]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00260238]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0025594]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00255736]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00255608]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0026367]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00255301]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00255172]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0025508]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254919]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254841]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254714]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254642]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254554]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254543]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254439]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254559]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254336]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254508]
```

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254279]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0025542]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254103]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254151]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00254037]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00255]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00340918]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00253962]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00253862]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00253816]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00253833]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.00253761]
```

In [39]:

```
lossXY = loss(X,Y).data
println("Training loss is $(lossXY)")
```

Validate

```
Training loss is 0.0025375174764556054
```

We need the number of numbers in your design to be fewer than 16.

In [40]:

```
println("Number of neurons equals $(n)")
```

Validate

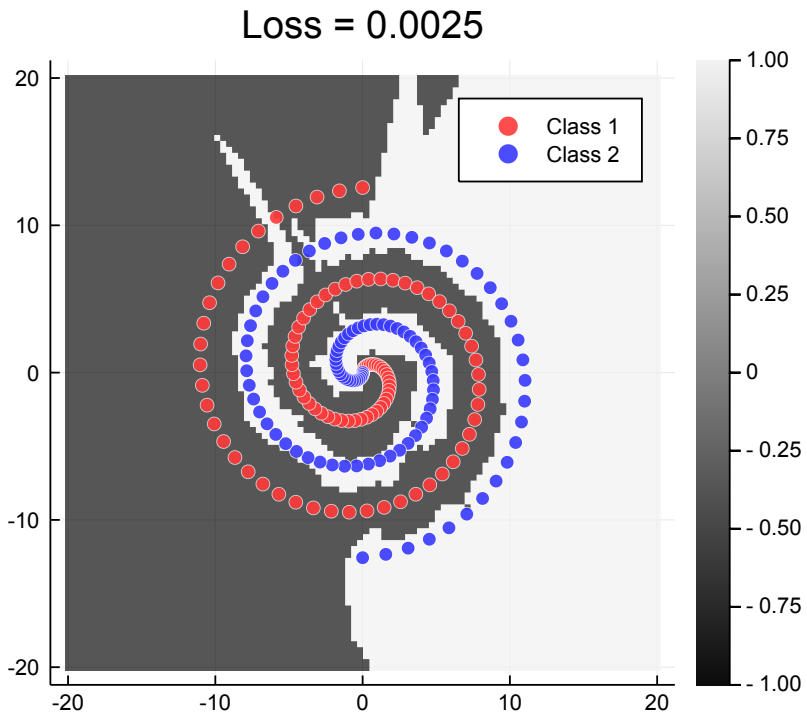
```
Number of neurons equals 15
```

Let us now display the decision boundary.

In [41]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range, τ)
2  plot!(; title="Loss = $(round(lossXY, digits=4))")
```

▶ Run

Out[41]:



Did your network eventually learn a decision boundary that separated the red and blue points?

- [x] Yes -- and it is quite a beautiful boundary indeed!
- [ ] No

Submit

Now we experiment with designing a network that works as above except we just change the activation function. Our goal is to create a network that works for both the `relu` and `sigmoid` activation functions.

In [42]:

```
active_fun = sigmoid

n = 16
m = Chain(Dense(size(X,1), n, active_fun),Dense(n, 20, active_fun),Der
 ##TODO: Start with same network as above before changing it

loss(x, y) = loss_fn(m(x), y) ## replace with mse
dataset = Base.Iterators.repeated((X, Y), iters)

opt = ADAM()
evalcb = () -> @show([loss(X,Y)])

Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 1))
```

▶ Run

```
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.330904]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.243337]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.235133]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.22891]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.221663]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.213304]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.207122]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.199025]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.180116]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.148151]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.111196]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0916494]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0828984]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0761753]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0676846]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0608323]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0564248]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0539466]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0509357]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0487051]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0467884]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.045669]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0446576]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0438985]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0434622]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0431006]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0429069]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0428029]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.042727]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0426706]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0426238]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425991]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.042586]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425662]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425544]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425451]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425378]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.042532]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425275]
[loss(X, Y)] = Tracker.TrackedReal{Float64}[0.0425235]
```

In [43]:
```
1  lossXY = loss(X,Y).data
2  println("Training loss is $(lossXY)")
```

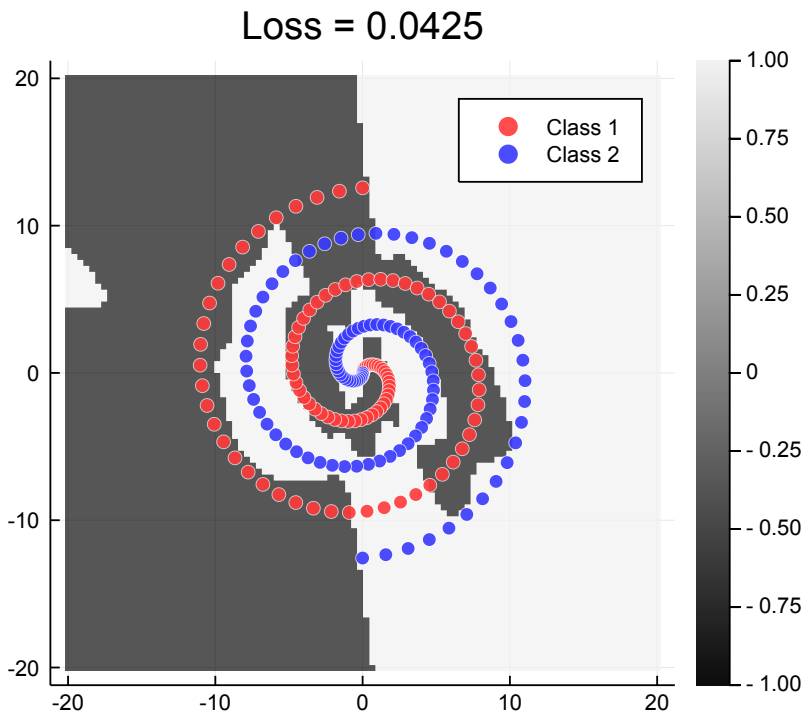
Validate

```
Training loss is 0.04252073973239017
```

In [44]:
```
1  display_decision_boundaries(X1c, X2c, m, x1range, x2range, τ)
2  plot!(; title="Loss = $(round(lossXY, digits=4))")
```

Run

Out[44]:



Does the decision boundary nicely separate the blue and red points?

- [x] Yes! It is spirally in a way that separates the spirals...
- [ ] No!

Submit

3.1 How are deep neural networks designed in practice?

Congratulations! You just designed two deep neural networks corresponding to two different architectures to classify the spiral dataset.

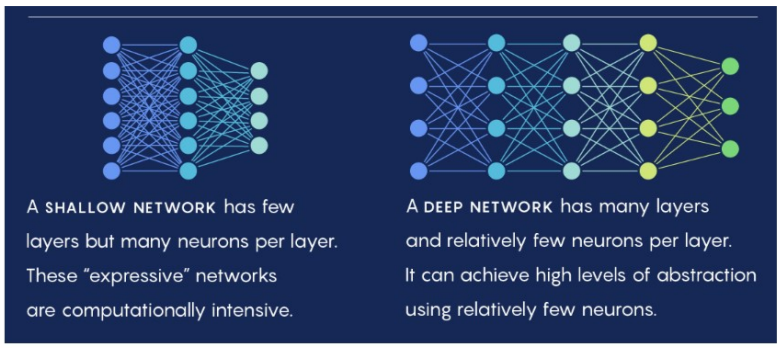
The design process allows you to make contact with how engineers design such networks in practice.

How did you know how to set the parameters of the neural network for classifying the spiral dataset?

- [ ] I just knew, I always do ...
- [x] I tried something, it didn't work, so I tried something else

Submit

Engineers in practice do this as well as excerpted below from [this article (https://www.quantamagazine.org/foundations-built-for-a-general-theory-of-neural-networks-20190131/)](https://www.quantamagazine.org/foundations-built-for-a-general-theory-of-neural-networks-20190131/).



Lucy Reading-Ikkanda/Quanta Magazine

Beyond those general guidelines, however, engineers largely have to rely on experimental evidence: They run 1,000 different neural networks and simply observe which one gets the job done.

"These choices are often made by trial and error in practice," Hanin said. "That's sort of a tough [way to do it] because there are infinitely many choices and one really doesn't know what's the best."

A better approach would involve a little less trial and error and a little more upfront understanding of what a given neural network

A deeper theory of deep learning is starting to take place, but as of now trial and error is still the way to go. The important point is that **many network configurations** can accomplish a good result.

4 Learning to recognize handwriting

We now illustrate how deep networks can learn to recognize handwriting. We first load the MNIST dataset. We load the training images of the digits 0-9, the corresponding labels. These are stored in the variables `X` and `labels` respectively in the next cell. We will test the accuracy of the

classifier by determining a label for a set of (independent from the training data) test images using the trained neural network andt then comparing this with the known label for each test image. The test images and the test labels are stored in the `test_X` and `test_Y` variables in the next cell.

In [45]:
```
imgs = MNIST.images()
labels = MNIST.labels()
X = hcat(float.(reshape.(imgs, :))...)

test_X = hcat(float.(reshape.(MNIST.images(:test), :))...)
test_Y = onehotbatch(MNIST.labels(:test), 0:9);
```

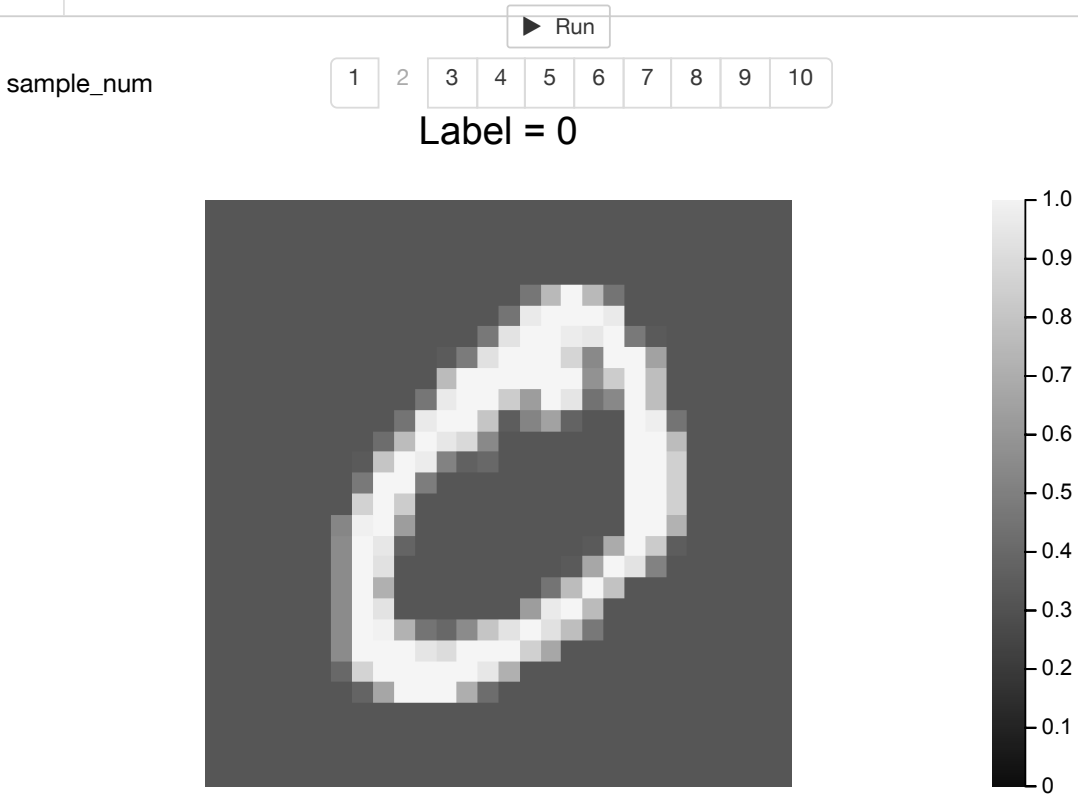
▶ Run

```
┌ Info: Downloading MNIST dataset
└ @ Flux.Data.MNIST /home/nbuser/.julia/packages/Flux/dkJUV/src/data/mnis
t.jl:24
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
urrent
                                 Dload  Upload   Total   Spent    Left  S
peed
100   469  100   469    0     0   2521      0 --:--:-- --:--:-- --:--:--
2521
100 9680k  100 9680k    0     0  12.9M      0 --:--:-- --:--:-- --:--:--
33.5M
┌ Info: Downloading MNIST dataset
└ @ Flux.Data.MNIST /home/nbuser/.julia/packages/Flux/dkJUV/src/data/mnis
t.jl:24
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
urrent
                                 Dload  Upload   Total   Spent    Left  S
peed
100   469  100   469    0     0   3065      0 --:--:-- --:--:-- --:--:--
3065
100 28881  100 28881    0     0  80002      0 --:--:-- --:--:-- --:--:--
80002
┌ Info: Downloading MNIST dataset
└ @ Flux.Data.MNIST /home/nbuser/.julia/packages/Flux/dkJUV/src/data/mnis
t.jl:24
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
urrent
                                 Dload  Upload   Total   Spent    Left  S
peed
100   467  100   467    0     0   2796      0 --:--:-- --:--:-- --:--:--
2779
100 1610k  100 1610k    0     0  2970k      0 --:--:-- --:--:-- --:--:--
2970k
┌ Info: Downloading MNIST dataset
└ @ Flux.Data.MNIST /home/nbuser/.julia/packages/Flux/dkJUV/src/data/mnis
t.jl:24
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
urrent
                                 Dload  Upload   Total   Spent    Left  S
peed
100   467  100   467    0     0   3032      0 --:--:-- --:--:-- --:--:--
3032
100  4542  100  4542    0     0  10070      0 --:--:-- --:--:-- --:--:--
4435k
```

Here are the first 10 samples of the training dataset.

In [46]:
```
@manipulate for sample_num in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    heatmap(
        reshape(X[:, sample_num], 28, 28);
        yflip=:true,
        color=:grays,
        ticks=:none,
        aspect_ratio=:equal,
        showaxis=:false
    )
    plot!(; title="Label = $(labels[sample_num])")
end
```

Out[46]:



And the associated "one hot encoded" labels. This maps digit `0` to e_1, digit `1` to e_2 and so on. It is numerically more efficient to store e_i as a `Boolean` vector of with `true` in the `i` -th element and `false` everywhere. This is the encoding produced by one-hot encoding the `labels` vector using the `onehotbatch` command as in the next cell.

In [47]:
```
Y = onehotbatch(labels, 0:9)
Y[:, 1:5]
```

▶ Run

Out[47]:
```
10×5 Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}:
 false   true  false  false  false
 false  false  false   true  false
 false  false  false  false  false
 false  false  false  false  false
 false  false   true  false  false
  true  false  false  false  false
 false  false  false  false  false
 false  false  false  false  false
 false  false  false  false  false
 false  false  false  false   true
```

Inspection of the first 10 columns of the `Y` array reveal that the first 10 samples belong to ? (select **all** that apply)

- [ ] digits 0, 1, 2, 3, 4
- [x] class labels 6, 1, 5, 2, 10
- [x] digits 5, 0, 4, 1, 9; because class label 1 corresponds to digit 0 and so on, and we index class labels from 1 to 10.
- [ ] digits 6, 1, 5, 2, 10; because we index class labels from 1 to 10, and class label 1 corresponds to digit 1, and class label 10 corresponds to digit 10.

Submit

4.1 Probability vectors and the softmax function

A vectort $x \in \mathbb{R}^n$ is a **probability vector** if its elements are non-negative and

$$\sum_i x_i = 1.$$

In other words, the sum of the elements of x adds up to one.

Which of the vectors are **not** probability vectors? (select **all** that apply)

- [x] $-e_1$ -- because the elements are non-negative and do not sum to one
- [ ] $2e_1 - e_2$ -- because the elements do not sum to one
- [x] $2e_1 - e_2$ -- because some elements of the vector are negative
- [x] any real-valued vector having whose entries are negative
- [x] any real-valued vector with non-negative entries any of whose individual entries are larger than 1

Submit

Clearly probability vectors are a subset of the set of real-valued vectors.

Which of the vectors are probability vectors ? (select **all** that apply)

- [x] e_1
- [x] e_2
- [ ] $e_1 + e_2$
- [ ] $\frac{1}{\sqrt{2}}e_1 + \frac{1}{\sqrt{2}}e_2$
- [x] $\frac{1}{2}e_1 + \frac{1}{2}e_2$
- [ ] some but not all one-hot encoded vectors
- [x] all one-hot encoded vectors because they have non-negative elements that sum to one

Submit

The `softmax` (https://en.wikipedia.org/wiki/Softmax_function) function takes as an input **any real-valued vector** $x \in \mathbb{R}^n$ and returns as its output probability vectors.

Since one-hot encoded vectors of the class labels are themselves probability vectors, we can thus design the neural network such that its output (post the soft-max layer) is as close as possible to the corresponding one-hot encoded vectors.

When we use such a softmax function, it is common to use the cross-entropy function to measure the closeness between the neural network output and the one-hot encodex vectors. The cross-entropy function is designed to measure the distance between probability vectors in a way that is more natural than the mean-squared error between the vectors in a way does not consider the fact that vectors are probability vectors.

To evaluate the performance of our network we create the function `accuracy` . The `oncold` function is the inverse of the `onehot` function and returns the index of the largest entry.

In [48]:
```
using Statistics: mean
using Flux: onecold
accuracy(x, y) = mean(onecold(m(x)) .== onecold(y))
```

Run

Out[48]:
```
accuracy (generic function with 1 method)
```

To create a callback function, as in the next cell, that displays the loss function and the accuracy attained on the training dataset.

In [49]:
```
evalcb = () -> @show([Tracker.data(loss(X,Y)), accuracy(test_X, test_
```

Run

Out[49]:
```
#28 (generic function with 1 method)
```

When we have one-hot-encoded the class labels, and we use a `softmax` output layer, then ? (select **all** that apply)

- [ ] for input corresponding to label i, we expect to get a *probability vector* that is close to $\pm e_i$
- [x] for input corresponding to label, i we expect to get a *probability vector* that is close to e_i
- [x] for input corresponding to label i, we expect that the largest element of the output of the `softmax` layer to be close to 1 but never greater than 1.
- [x] The index of the largest entry of the output of `softmax` layer is the class label -- this is equivalent to `onecold(softmax.(x))` if `x` is the input to the `softmax` layer
- [x] The closer the largest entry of the output of the `softmax` layer is to one, the more confident we can be of the accurarcy of the class label prediction
- [x] The further the largest entry of the output of the `softmax` layer is from one, the less confident we might be of the accurarcy of the class label prediction

Submit

4.2 Neural network with a single hidden layer and softmax output

We now train a dense linear network with the `crossentropy` (https://en.wikipedia.org/wiki/Cross_entropy) loss function.

In [50]:
```
loss_fn = crossentropy

m = Chain(Dense(28^2, size(Y,1)), softmax) ##TODO: Replace the ?? Hin
loss(x, y) = loss_fn(m(x), y)
opt = ADAM()
```

Run

Out[50]:
```
ADAM(0.001, (0.9, 0.999), IdDict{Any,Any}())
```

Exercise:

Train the single layer network until it attains an accuracy greater 85%.

Tip: If you do not meet the accuracy spec, then try training with more batches.

In what follows, the command

```
dataset = [(X[:, i], Y[:, i]) for i in partition(shuffle(1:size(X, 2)), batch_size)]
```

first randomly shuffles the columns of `X` and then creates a dataset composed of `batch_size` sized partitions.

In [51]:
```
using Flux: shuffle
epochs = 4 # each epoch is one pass through data
batch_size = 32
for epoch_idx in 1:epochs
    dataset = [(X[:, i], Y[:, i]) for i in partition(shuffle(1:size(X
    Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 1.
end
## it takes a while for output to appear -- be patient :)
```

▶ Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [2.49693, 0.1191]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.506148, 0.8838]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.348472, 0.9094]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.318651, 0.9151]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.310799, 0.9175]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.296195, 0.919]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.283155, 0.9211]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.28212, 0.9204]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.27704, 0.9206]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.270095, 0.9238]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.268485, 0.9243]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.267458, 0.9242]
```

In [52]:
```
test_accuracy = accuracy(test_X, test_Y)
println("Test accuracy is $(test_accuracy)")
```

Validate

```
Test accuracy is 0.9247
```

The classification accuracy ? (select **all** that apply)

- [x] starts off relatively low (much than 50%) and starts increasing after that
- [ ] starts off high (>80%) and stays high
- [x] goes beyond 85% eventually -- this beats what we achieved using `mse` classification in the `Learn2ClassifyMany`

Submit

Let's look at the digits alongside the corresponding output of our trained neural network. The cell below scrolls through all samples in `test_X.

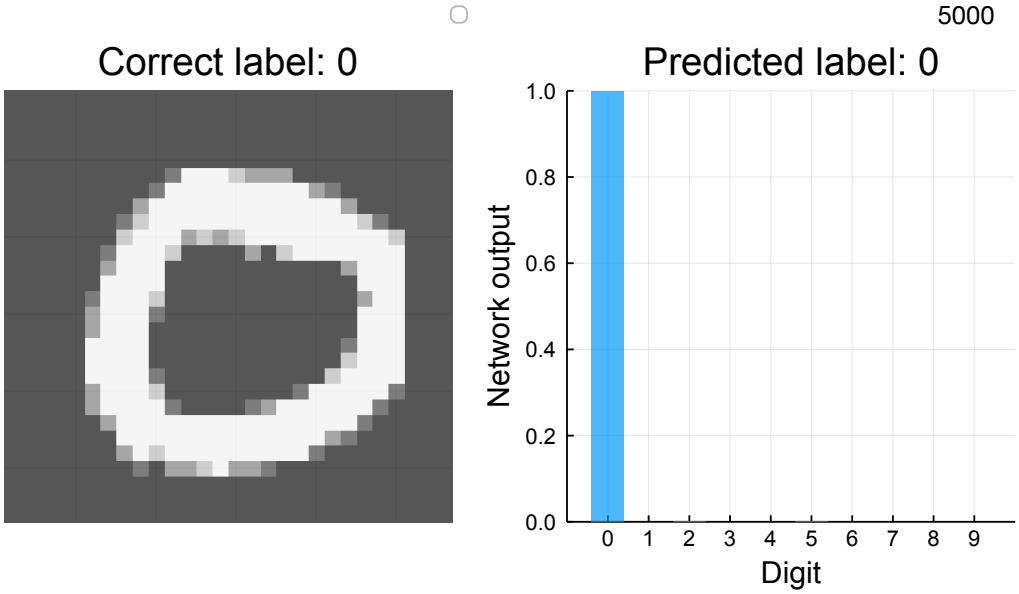
In [53]:

```
@manipulate for idx in 1:size(test_X, 2)
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    m_output = m(m_input)

    plot(
        heatmap(
            reshape(m_input, 28, 28);
            yflip=true,
            color=:grays,
            colorbar=false,
            axis=false,
            title="Correct label: $(correct_label)"
            ),
        bar(
            m_output.data;
            xticks=(1:10, 0:9),
            ylim=(0, 1.0),
            linewidth=0,
            xlabel="Digit",
            ylabel="Network output",
            label="",
            title="Predicted label: $(onecold(m_output) - 1)"
            );
        size=(600, 300)
    )
end
```

▶ Run

Out[53]:

idx 5000



Examining the output of the network and the (correct) predictions made as we vary the input images reveals that? (select **all** that apply)

- [x] when the network correctly predicts an image then the `correct label`, obtained from the labeled test set, matches the `predicted label`
- [x] the output of the neural network varies as we vary the input test digit image
- [ ] the output of the neural network remains fixed as we vary the input test digit image
- [ ] the output of the neural network with the softmax output layer are real-valued positive and negative numbers
- [x] the output of the neural network with the softmax output layer are real-valued positive numbers between 0 and 1
- [x] when the network correctly predicts a digit then the index of the neural network output that is the largest corresponds to the digit label

Submit

Now let's look specifically at the samples which the network classifies incorrectly. There aren't too many of these, and if you look at how poorly some of these digits were written, it's easy to see why the network got it wrong. In most of these cases, you can determine the actual label by looking at the *second-highest* bar, since the network is effectively "torn between" two choices.

In [54]:

```
incorrect_idx = Int64[]

for idx in 1:size(test_X, 2)
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    applied_label =  onecold(m(m_input)) - 1

    if correct_label != applied_label
        push!(incorrect_idx, idx)
    end
end

println("Number of incorrect examples = $(length(incorrect_idx))")
@manipulate for inc_idx in 1 : length(incorrect_idx)
    idx = incorrect_idx[inc_idx]
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    m_output = m(m_input)

    plot(
        heatmap(
            reshape(m_input, 28, 28);
            yflip=true,
            color=:grays,
            colorbar=false,
            axis=false,
            title="Correct label: $(correct_label)"
            ),
        bar(
            m_output.data;
            xticks=(1:10, 0:9),
            ylim=(0, 1.0),
            linewidth=0,
            xlabel="Digit",
            ylabel="Network Output",
            label = "",
            title="Predicted label: $(onecold(m_output) - 1)",
            color=:red
            );
        size=(600, 300)
    )
end
```

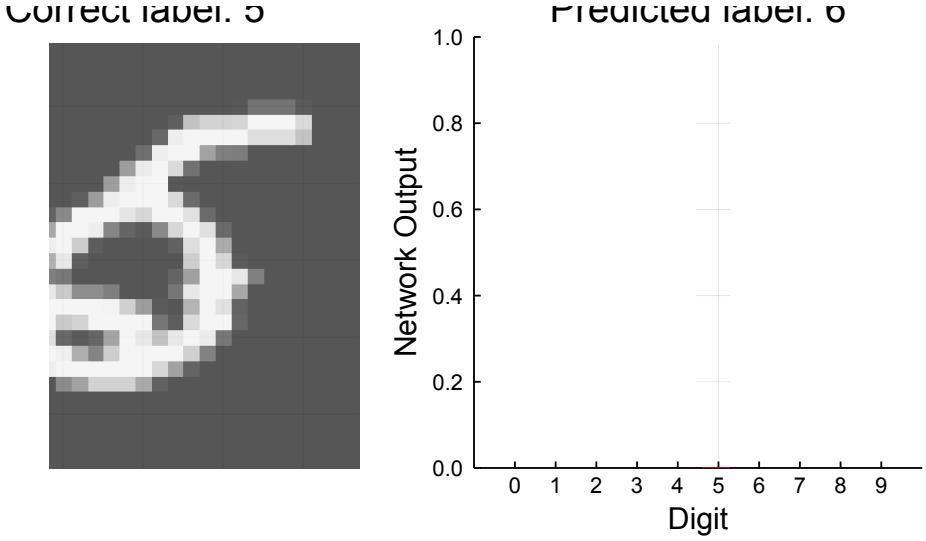
▶ Run

```
Number of incorrect examples = 753
```

Out[54]:

inc_idx ○ 706

Correct label: 5 Predicted label: 6



Examining the output of the network and the (wrong) predictions made as we vary the input images reveals that? (select **all** that apply)

- [ ] when the network incorrectly predicts an image then the `correct label` , obtained from the labeled test set, **does not** match the `predicted label`
- [x] the output of the neural network with the softmax output layer are real-valued positive numbers between 0 and 1
- [ ] when the network incorrectly predicts a digit then the index of the neural network output that is the largest corresponds to the digit label
- [x] when the network incorrectly predicts a digit then the index of the neural network output that is the largest **does not** correspond to the digit label

Submit

Exercise: Remark on the characterstics of some of the digits that were classified incorrectly.

The digits that were incorrectly predicted are written at the boundaries and they are also tilted which caused the algorithm to misclassify thesee images. Also some of the digits were very similarly written like 5,6 which are difficult to recognize even for a human.

Edit / Render

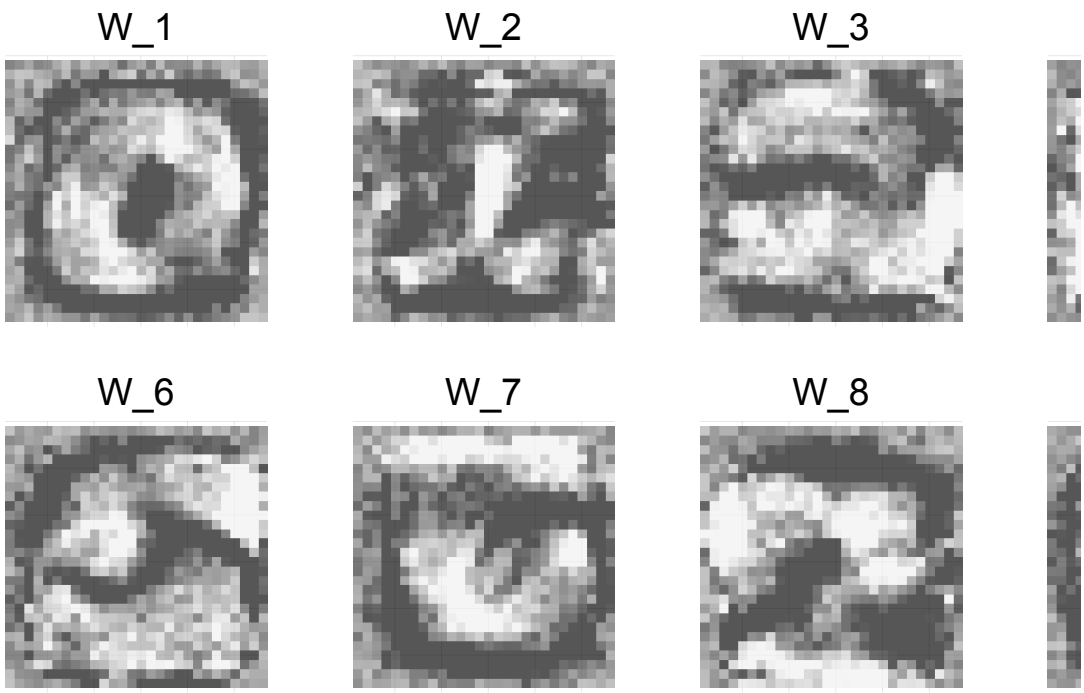
We now examine the weights learned.

In [55]:

```
using LaTeXStrings
plots = []

kwargs = (
    :clim => (-0.2, 0.2),
    :yflip => true,
    :color => :grays,
    :aspect_ratio => :equal,
    :ticks => :off,
    :colorbar => :false,
    :showaxis => :false
)

for idx in 1:10
    push!(
        plots,
        heatmap(
            reshape(Tracker.data(m.layers[1].W[idx, :]), 28, 28);
            title="W_$(idx)",
            kwargs...
        )
    )
end
plot(plots...; layout=(2, 5), size=(950, 400))
```

▶ Run

Out[55]:



Exercise:

Remark on the characteristics of the weight matrices learned.

The weights are concentrated at the middle and the value at the boundaries are similar for all the digits. We can see that the weights are concentrated along the shape of the digit.

Edit / Render

4.3 A deeper two-hidden-layer model with softmax output

Next we train a two layer network with a `relu` activation function and `crosentropy` loss.

In [56]:
```
loss_fn = crossentropy

m2 = Chain(Dense(size(X,1), n,relu),Dense(n,size(Y,1)),softmax) ##TOD
loss(x, y) = loss_fn(m2(x), y)
opt = ADAM()
```

Run

Out[56]:
```
ADAM(0.001, (0.9, 0.999), IdDict{Any,Any}())
```

Exercise:

Train the two layer network till it attains an accuracy greater 92%

Tip: If you do not meet the accuracy spec, then keep training by adding more epochs and/or increasing the batch size.

In [57]:

```
using Flux:shuffle
epochs = 100 # each epoch represents one pass through data
batch_size = 100
for epoch_idx in 1:epochs
    dataset = [(X[:, i], Y[:, i]) for i in partition(shuffle(1:size(X
    Flux.train!(loss, params(m2), dataset, opt; cb=throttle(evalcb, 2
end
## it takes a while for output to appear -- be patient and don't inte
```

▶ Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [2.24697, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.437909, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.311117, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.302625, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.280784, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.258979, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.247431, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.230069, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.224902, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.21809, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.21656, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.205646, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.201915, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.197048, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.193695, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.190168, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.18474, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.183016, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.176465, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.17326, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.170068, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.170199, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.166514, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.162527, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.160988, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.15617, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.15411, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.151876, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.152463, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.153332, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.146553, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.145955, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.150043, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.145136, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.145692, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.148455, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.138433, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.137762, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.137057, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.139045, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.135662, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.13381, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.13156, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.130791, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.13171, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.131125, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.131381, 0.9247]
```

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.127858, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.130559, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.128973, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.126089, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.123676, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.126048, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.124871, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.124478, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.122507, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.120997, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.119374, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.125297, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.120373, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.117858, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.118401, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.118665, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.116254, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.117045, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.116527, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.113892, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.114317, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.114448, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.111525, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.111199, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.110337, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.110731, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.111259, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.114884, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.110056, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.107698, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.107126, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.108017, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.109892, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.106856, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104512, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104752, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104153, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.103834, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104908, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104948, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.103038, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.104756, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.105149, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.10612, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.101238, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.103549, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.100586, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0990224, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0995781, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.100329, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0997178, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0963943, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.102042, 0.9247]
```

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0955087, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0968548, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0951145, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0969372, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0933023, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0953705, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0937339, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0950152, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0989633, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0921408, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0941888, 0.924
7]

[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0916474, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0903331, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0904337, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0926772, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0922369, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0912649, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0903862, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0915436, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0903596, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0887613, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0897099, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.087318, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0897583, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0897275, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0895232, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0876936, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0918858, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0899202, 0.924
```

```
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0877258, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0868791, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0873584, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.085024, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0872857, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0889195, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0864254, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0844636, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0839382, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0877655, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0831214, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0843294, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0855607, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0850729, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0830588, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0830194, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0820362, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0840152, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0828468, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0813287, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0813474, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0822339, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.081365, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0815122, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0790368, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.07986, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0807556, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0828821, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0819082, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.079173, 0.9247]
```

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0828956, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0827773, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0794406, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0802009, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0808766, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0791157, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0824003, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0773691, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.076555, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0771656, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0773923, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0798857, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0745264, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0742922, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0731996, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0761986, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0726386, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.073603, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0715646, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0751939, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0746099, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0758413, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0715973, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0734569, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0717031, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.072801, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0735454, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0704052, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0711867, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0710271, 0.924
7]
```

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0706951, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0692057, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0708115, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.073188, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0696067, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0676505, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0703103, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0717613, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0683602, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.067812, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0696254, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0690474, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0659756, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0674868, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0675761, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0657675, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0677671, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0682506, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0671407, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0644422, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0668731, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0654753, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0671696, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0696593, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0645604, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0657171, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0644085, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0642476, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0633791, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0653538, 0.924
```

```
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0652147, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0650864, 0.924
7]

[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0622936, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0647014, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0616965, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0632784, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0645685, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0663336, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0649736, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0631479, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0630457, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0624767, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0615115, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.064607, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0633384, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0613435, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0625607, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.062861, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0596387, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0607467, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0612998, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0605324, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0605185, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0628487, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.059079, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0587402, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0608348, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0633613, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.059114, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0595275, 0.924
```

```
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0613256, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0580598, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0603238, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.058153, 0.9247]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0583999, 0.924
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.057896, 0.9247]
```

In [58]:

```
test_accuracy = accuracy(test_X, test_Y)
println("Test accuracy is $(test_accuracy)")
```

Validate

```
Test accuracy is 0.9247
```

Let's look again at some of the digits and corresponding outputs for this network.

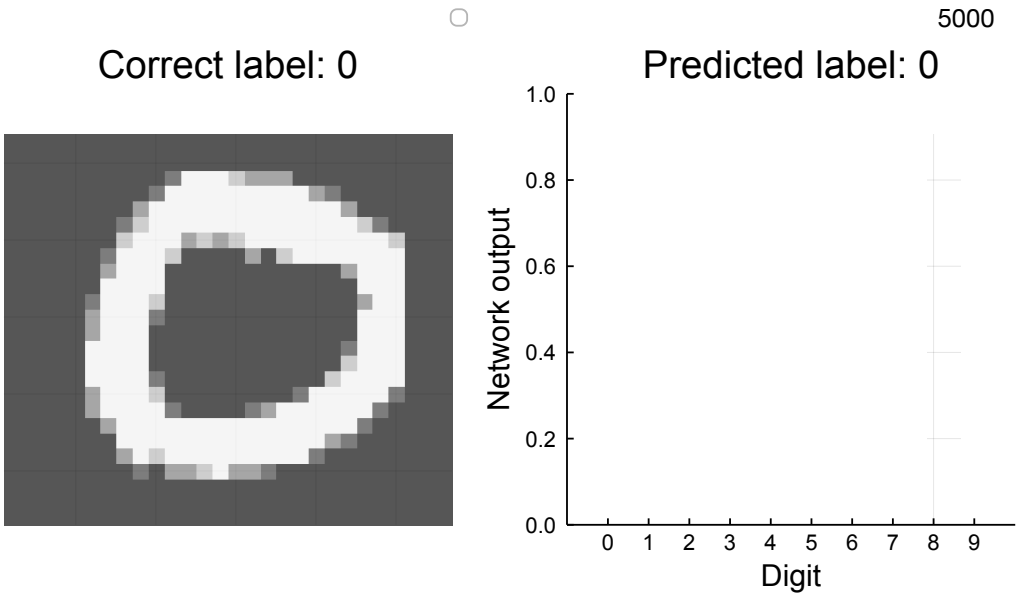
In [59]:

```
@manipulate for idx in 1:size(test_X, 2)
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    m_output = m2(m_input)

    plot(
        heatmap(
            reshape(m_input, 28, 28);
            yflip=true,
            color=:grays,
            colorbar=false,
            axis=false,
            title="Correct label: $(correct_label)"
            ),
        bar(
            m_output.data;
            xticks=(1:10, 0:9),
            ylim=(0, 1.0),
            linewidth=0,
            xlabel="Digit",
            ylabel = "Network output",
            label = "",
            title="Predicted label: $(onecold(m_output) - 1)"
            );
        size=(600, 300)
    )
end
```

▶ Run

Out[59]:

idx 5000



And here again are the handwritten digits that were classified incorrectly:

In [85]:

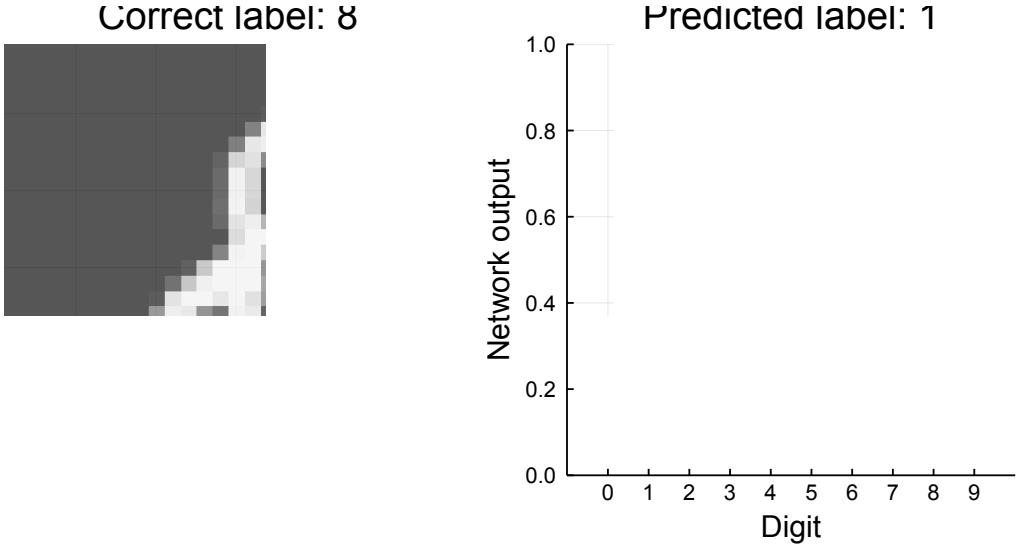
```
incorrect_idx = Int64[]

for idx in 1:size(test_X, 2)
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    applied_label =  onecold(m(m_input)) - 1

    if correct_label != applied_label
        push!(incorrect_idx, idx)
    end
end

println("Number of incorrect examples = $(length(incorrect_idx))")
@manipulate for inc_idx in 1 : length(incorrect_idx)
    idx = incorrect_idx[inc_idx]
    m_input = test_X[:, idx]
    correct_label = findfirst(test_Y[:, idx]) - 1
    m_output = m2(m_input)

    plot(
        heatmap(
            reshape(m_input, 28, 28);
            yflip=true,
            color=:grays,
            colorbar=false,
            axis=false,
            title="Correct label: $(correct_label)"
            ),
        bar(
            m_output.data;
            xticks=(1:10, 0:9),
            ylim=(0, 1.0),
            linewidth=0,
            xlabel="Digit",
            ylabel = "Network output",
            label = "",
            title="Predicted label: $(onecold(m_output) - 1)",
            color=:red
            );
        size=(600, 300)
    )
end
```

▶ Run

```
Number of incorrect examples = 540
```

Out[85]:

inc_idx ○ 512




```
BoundsError: attempt to access 540-element Array{Int64,1} at index [649]
Stacktrace:
 [1] getindex(::Array{Int64,1}, ::Int64) at ./array.jl:729
 [2] (::getfield(Main, Symbol("##34#35")))(::Int64) at ./In[54]:15
 [3] (::getfield(Observables, Symbol("##16#17")){getfield(Main, Symbol("#
##34#35")),Observable{Any}})(::Int64) at /home/nbuser/.julia/packages/Obse
rvables/qCJWB/src/Observables.jl:152
 [4] (::getfield(Observables, Symbol("#g#15")){getfield(Observables, Symb
ol("##16#17")){getfield(Main, Symbol("##34#35")),Observable{Any}},Tuple{W
idget{:slider,Int64}}})(::Int64) at /home/nbuser/.julia/packages/Observab
les/qCJWB/src/Observables.jl:136
 [5] #setindex!#1(::getfield(WebIO, Symbol("##44#45")), ::Function, ::Obs
ervable{Int64}, ::Int64) at /home/nbuser/.julia/packages/Observables/qCJW
B/src/Observables.jl:87
 [6] #setindex! at ./none:0 [inlined]
 [7] setexcludinghandlers at /home/nbuser/.julia/packages/Observables/qCJ
WB/src/Observables.jl:103 [inlined]
 [8] set_nosync(::Observable{Int64}, ::Int64) at /home/nbuser/.julia/pack
ages/WebIO/2mZPb/src/scope.jl:339
 [9] dispatch(::Scope, ::String, ::Int64) at /home/nbuser/.julia/package
s/WebIO/2mZPb/src/scope.jl:348
 [10] dispatch_command(::WebIO.IJuliaConnection, ::Dict{String,Any}) at /
home/nbuser/.julia/packages/WebIO/2mZPb/src/messaging.jl:104
 [11] dispatch(::WebIO.IJuliaConnection, ::Dict{String,Any}) at /home/nbu
ser/.julia/packages/WebIO/2mZPb/src/messaging.jl:81
 [12] (::getfield(WebIO, Symbol("##92#93")){WebIO.IJuliaConnection})(::IJ
ulia.Msg) at /home/nbuser/.julia/packages/WebIO/2mZPb/src/providers/ijuli
a.jl:21
 [13] comm_msg(::ZMQ.Socket, ::IJulia.Msg) at /home/nbuser/.julia/package
s/IJulia/fRegO/src/comm_manager.jl:134
 [14] #invokelatest#1 at ./essentials.jl:742 [inlined]
 [15] invokelatest at ./essentials.jl:741 [inlined]
 [16] eventloop(::ZMQ.Socket) at /home/nbuser/.julia/packages/IJulia/fReg
O/src/eventloop.jl:8
 [17] (::getfield(IJulia, Symbol("##15#18")))() at ./task.jl:259

KERNEL EXCEPTION
BoundsError: attempt to access 540-element Array{Int64,1} at index [649]
```

```
Stacktrace:
 [1] getindex(::Array{Int64,1}, ::Int64) at ./array.jl:729
 [2] (::getfield(Main, Symbol("##34#35")))(::Int64) at ./In[54]:15
 [3] (::getfield(Observables, Symbol("##16#17")){getfield(Main, Symbol("#
#34#35")),Observable{Any}})(::Int64) at /home/nbuser/.julia/packages/Obse
rvables/qCJWB/src/Observables.jl:152
 [4] (::getfield(Observables, Symbol("#g#15")){getfield(Observables, Symb
ol("##16#17")){getfield(Main, Symbol("##34#35")),Observable{Any}},Tuple{W
idget{:slider,Int64}}})(::Int64) at /home/nbuser/.julia/packages/Observab
les/qCJWB/src/Observables.jl:136
 [5] #setindex!#1(::getfield(WebIO, Symbol("##44#45")), ::Function, ::Obs
ervable{Int64}, ::Int64) at /home/nbuser/.julia/packages/Observables/qCJW
B/src/Observables.jl:87
 [6] #setindex! at ./none:0 [inlined]
 [7] setexcludinghandlers at /home/nbuser/.julia/packages/Observables/qCJ
WB/src/Observables.jl:103 [inlined]
 [8] set_nosync(::Observable{Int64}, ::Int64) at /home/nbuser/.julia/pack
ages/WebIO/2mZPb/src/scope.jl:339
 [9] dispatch(::Scope, ::String, ::Int64) at /home/nbuser/.julia/package
s/WebIO/2mZPb/src/scope.jl:348
 [10] dispatch_command(::WebIO.IJuliaConnection, ::Dict{String,Any}) at /
home/nbuser/.julia/packages/WebIO/2mZPb/src/messaging.jl:104
 [11] dispatch(::WebIO.IJuliaConnection, ::Dict{String,Any}) at /home/nbu
ser/.julia/packages/WebIO/2mZPb/src/messaging.jl:81
 [12] (::getfield(WebIO, Symbol("##92#93")){WebIO.IJuliaConnection})(::IJ
ulia.Msg) at /home/nbuser/.julia/packages/WebIO/2mZPb/src/providers/ijuli
a.jl:21
 [13] comm_msg(::ZMQ.Socket, ::IJulia.Msg) at /home/nbuser/.julia/package
s/IJulia/fRegO/src/comm_manager.jl:134
 [14] #invokelatest#1 at ./essentials.jl:742 [inlined]
 [15] invokelatest at ./essentials.jl:741 [inlined]
 [16] eventloop(::ZMQ.Socket) at /home/nbuser/.julia/packages/IJulia/fReg
O/src/eventloop.jl:8
 [17] (::getfield(IJulia, Symbol("##15#18")))() at ./task.jl:259
```

Exercise:

Remark on the characteristics of any digits that were incorrectly classified using a single layer and softmax output layer that are now correctly classified using the two layer + softmax network.

Earlier the no. of incorrectly classified digits were 753 and now it is 540, hence using 2 layer definitely improved the performance, also the weight matrix learned earlier were very similar for digit 6 and 0, hence the algorithm couldn't classify them correctly, which doesn't happen now.

Edit / Render

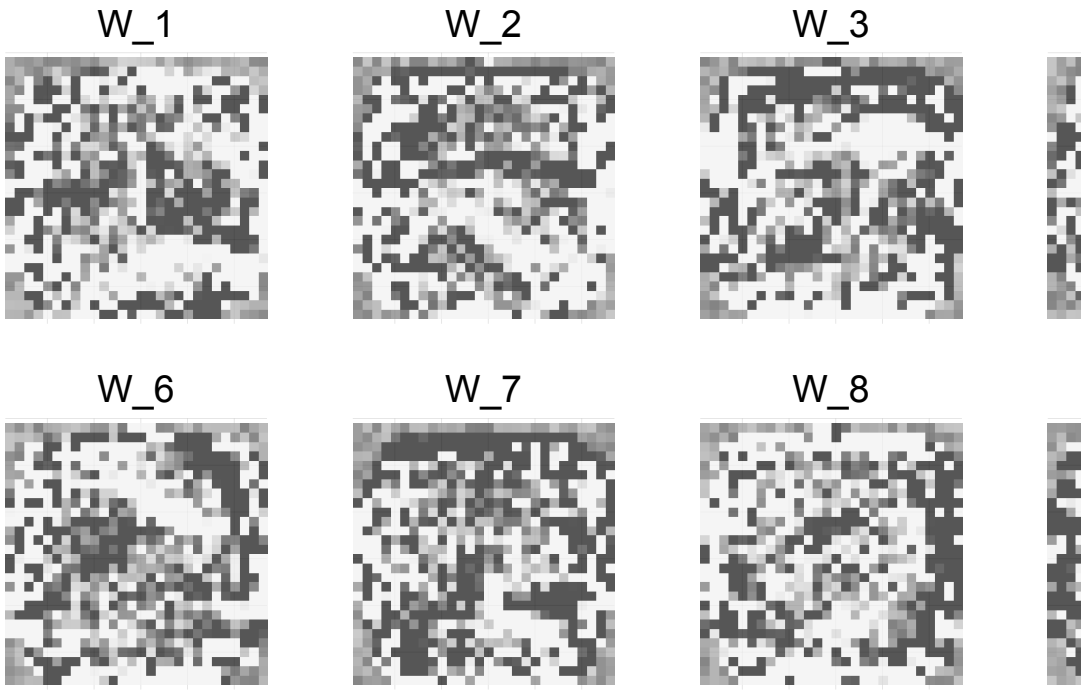
Finally, let's examine the parameters that have been learned.

In [61]:

```
using LaTeXStrings
plots = []

kwargs = (
    :clim => (-0.2, 0.2),
    :yflip => true,
    :color => :grays,
    :aspect_ratio => :equal,
    :ticks => :off,
    :colorbar => :false,
    :showaxis => :false
)

for idx in 1:10
    push!(
        plots,
        heatmap(
            reshape(Tracker.data(m2.layers[1].W[idx, :]), 28, 28);
            title="W_$idx",
            kwargs...
        )
    )
end
plot(plots...; layout=(2, 5), size=(950, 400))
```

▶ Run

Out[61]:



What information do the weight matrices convey?

- [ ] Nothing -- they are random
- [x] Like in the `Learn2ClassifyMany` codex, they convey information about the digits

Submit

Exercise: Remark on the characteristics of the weight matrices learned, as you see from their heatmap display.

Unlike earlier it is very difficult to recognize the digits from their weight matrix, they appear very noisy.

Edit / Render

4.4 Training with MSE loss instead of cross-entropy loss

Now let us change the loss function to `mse` instead of `crossentropy` in the next cell and train the network.

In [62]:
```
1  loss_fn = mse
2  m = Chain(Dense(28^2, 10,relu),Dense(10,size(Y,1),relu),softmax) ##TO
3  loss(x, y) = loss_fn(m(x), y) ## replace with mse
4  opt = ADAM()
```

Run

Out[62]: ADAM(0.001, (0.9, 0.999), IdDict{Any,Any}())

In [63]:
```
batches = 100
dataset = Base.Iterators.repeated((X, Y), batches)
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 5))
```

▶ Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0903597, 0.087]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0879424, 0.265]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0858977, 0.325
5]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0823362, 0.366]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0775828, 0.428
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0719916, 0.512
1]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0668563, 0.535
2]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0609458, 0.577
9]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0561877, 0.609
5]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0524267, 0.639
1]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0493988, 0.671]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0469132, 0.689
2]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0449118, 0.698]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0433075, 0.703
7]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.0420066, 0.708
8]
```

How well does it do with MSE?

- [ ] Loss function values are lower with MSE by a lot, and so is accuarcy.
- [ ] Loss function values are higher than with MSE, but so is accuarcy.
- [x] Loss function values are lower with MSE, but accuracy is about the same.

Submit

The point here is that we actually care about the test accuracy more than the exact numbers attained in the loss. Knowing the exact value of the training loss is does not give any insight on test accuracy attained. This is why we need to separately test the model on a portion of the data that is not being used to train the model. This is referred to as **cross-validation**.

Try different sizes for the intermediate layer, re-run your simulations and record the accuracy obtained. What do simulations reveal ? (select **all** that apply)

- [ ] there is a unique configuration for which the NN gives optimal performance
- [x] many configurations and architectures give good performance
- [x] it is unclear how to "optimally" pick the size

- [x] it is unclear what optimal size means based on the simulations we just ran -- how do the answers depend on the optimizer and the learning rate?
- [x] Despite all we have left to understand, it is **remarkably easy** to get a very good (> 90%) classification performance using a nicely labeled dataset with just a few lines of code!

Submit

Indeed -- this is what makes neural networks remarkably versatile and ready-to-use for many applications! In practice, neural networks can be continually trained by seeding them with the weights learned earlier. To that end, the Open Neural Network Exchange Format (ONXX) (https://onnx.ai/) is being developed to facilitate the sharing of neural network models across language platforms.

The state-of-the-art for handwritten digit recognition uses convolutional neural networks. These have a different configuration and we will discuss them next.

5 Additional exercises

5.1 Sensitivity to optimizer and learning rate

Training a neural network depends on many factors -- the optimizer plays a big role.

In the cell below, the optimizer is set in the command

```
opt = ADAM()
```

Other choices are `SGD` (Stochastic Gradient Descent) and `Nesterov` (Nesterov accelerated gradient descesnt).

We can set *learning rate* of the algorithm explicltly using the command

```
opt = ADAM(0.01)
```

where we have set the rate to be 0.01. We can do the same for the other optimizers.

In the following cell we use a learning rate of 0.01.

In [89]:
```
loss_fn = crossentropy
m = Chain(
        Dense(28^2, 32, relu),
        Dense(32, 160, relu),
        Dense(160, 10, relu),
        softmax)
loss(x, y) = loss_fn(m(x), y) ## replace with mse
opt = ADAM(0.01)
```

▶ Run

Out[89]:
```
ADAM(0.01, (0.9, 0.999), IdDict{Any,Any}())
```

Exercise:

Train the network with a learning rate of 0.01. What accuracy do we get? Write down your answer.

The accuracy is 87%.

✎ Edit / Render

In [90]:
```
batches = 100
dataset = Base.Iterators.repeated((X, Y), batches)
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 10))
```

▶ Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [2.2113, 0.1882]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.850022, 0.766]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.589854, 0.8158]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.515618, 0.8326]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.463514, 0.8436]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.422897, 0.8531]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.395963, 0.8579]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.375341, 0.862]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.359264, 0.8643]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.346218, 0.8666]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.334792, 0.8698]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.324466, 0.8717]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.315873, 0.8708]
```

Exercise:

Train the network with a learning rate of 1. What accuracy do we get? Note your answer.

Using a learning rate of 1, the loss goes to infinite.

✎ Edit / Render

In [66]:
```
loss_fn = crossentropy
m = Chain(Dense(28^2, 10), softmax)
batches = 50
loss(x, y) = loss_fn(m(x), y) ## replace with mse
dataset = Base.Iterators.repeated((X, Y), batches)
opt = ADAM(1)
Flux.train!(loss, params(m), dataset, opt, cb = throttle(evalcb, 10))
```

▶ Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [Inf, 0.4206]

Loss is infinite

Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] losscheck(::Tracker.TrackedReal{Float32}) at /home/nbuser/.julia/pac
kages/Tracker/SAr25/src/back.jl:154
 [3] gradient_(::getfield(Flux.Optimise, Symbol("##15#21")){typeof(loss),
Tuple{Array{Float64,2},Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}}},
::Tracker.Params) at /home/nbuser/.julia/packages/Tracker/SAr25/src/back.
jl:98
 [4] #gradient#24(::Bool, ::Function, ::Function, ::Tracker.Params) at /h
ome/nbuser/.julia/packages/Tracker/SAr25/src/back.jl:164
 [5] gradient at /home/nbuser/.julia/packages/Tracker/SAr25/src/back.jl:1
64 [inlined]
 [6] macro expansion at /home/nbuser/.julia/packages/Flux/dkJUV/src/optim
ise/train.jl:71 [inlined]
 [7] macro expansion at /home/nbuser/.julia/packages/Juno/oLB1d/src/progr
ess.jl:134 [inlined]
 [8] #train!#12(::getfield(Flux, Symbol("#throttled#18")){getfield(Flux,
Symbol("##throttled#10#14")){Bool,Bool,getfield(Main, Symbol("##28#29")),
Int64}}, ::Function, ::Function, ::Tracker.Params, ::Base.Iterators.Take
{Base.Iterators.Repeated{Tuple{Array{Float64,2},Flux.OneHotMatrix{Array{F
lux.OneHotVector,1}}}}}, ::ADAM) at /home/nbuser/.julia/packages/Flux/dkJ
UV/src/optimise/train.jl:69
 [9] (::getfield(Flux.Optimise, Symbol("#kw##train!")))(::NamedTuple{(:c
b,),Tuple{getfield(Flux, Symbol("#throttled#18")){getfield(Flux, Symbol
("##throttled#10#14")){Bool,Bool,getfield(Main, Symbol("##28#29")),Int6
4}}}}, ::typeof(Flux.Optimise.train!), ::Function, ::Tracker.Params, ::Ba
se.Iterators.Take{Base.Iterators.Repeated{Tuple{Array{Float64,2},Flux.One
HotMatrix{Array{Flux.OneHotVector,1}}}}}, ::ADAM) at ./none:0
 [10] top-level scope at In[66]:7
```

In [67]:
```
dataset = Base.Iterators.repeated((X, Y), batches)
Flux.train!(loss, dataset, opt; cb = throttle(evalcb, 10))
```

▶ Run

```
┌ Warning: train!(loss, data, opt) is deprecated; use train!(loss, param
s, data, opt) instead
|   caller = ip:0x0
└ @ Core :-1

Loss is infinite

Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] losscheck(::Tracker.TrackedReal{Float32}) at /home/nbuser/.julia/pac
kages/Tracker/SAr25/src/back.jl:154
 [3] gradient_(::getfield(Flux.Optimise, Symbol("##15#21")){typeof(loss),
Tuple{Array{Float64,2},Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}}},
::Tracker.Params) at /home/nbuser/.julia/packages/Tracker/SAr25/src/back.
jl:98
 [4] #gradient#24(::Bool, ::Function, ::Function, ::Tracker.Params) at /h
ome/nbuser/.julia/packages/Tracker/SAr25/src/back.jl:164
 [5] gradient at /home/nbuser/.julia/packages/Tracker/SAr25/src/back.jl:1
64 [inlined]
 [6] macro expansion at /home/nbuser/.julia/packages/Flux/dkJUV/src/optim
ise/train.jl:71 [inlined]
 [7] macro expansion at /home/nbuser/.julia/packages/Juno/oLB1d/src/progr
ess.jl:134 [inlined]
 [8] #train!#12(::getfield(Flux, Symbol("#throttled#18")){getfield(Flux,
Symbol("##throttled#10#14")){Bool,Bool,getfield(Main, Symbol("##28#29")),
Int64}}, ::Function, ::Function, ::Tuple{}, ::Base.Iterators.Take{Base.It
erators.Repeated{Tuple{Array{Float64,2},Flux.OneHotMatrix{Array{Flux.OneH
otVector,1}}}}}, ::Flux.Optimise.OldOptimiser) at /home/nbuser/.julia/pac
kages/Flux/dkJUV/src/optimise/train.jl:69
 [9] #train! at ./none:0 [inlined]
 [10] #train!#37 at /home/nbuser/.julia/packages/Flux/dkJUV/src/optimise/
deprecations.jl:125 [inlined]
 [11] (::getfield(Flux.Optimise, Symbol("#kw##train!")))(::NamedTuple{(:c
b,),Tuple{getfield(Flux, Symbol("#throttled#18")){getfield(Flux, Symbol
("##throttled#10#14")){Bool,Bool,getfield(Main, Symbol("##28#29")),Int6
4}}}}, ::typeof(Flux.Optimise.train!), ::Function, ::Base.Iterators.Take
{Base.Iterators.Repeated{Tuple{Array{Float64,2},Flux.OneHotMatrix{Array{F
lux.OneHotVector,1}}}}}, ::ADAM) at ./none:0
 [12] top-level scope at In[67]:2
```

Exercise:

Compare the accuracy attained with the previous example and collect your observations and findings.

Using a learning rate of 1 we get the training loss as infinite. Hence the algorithm doesn't learn with such high value of learning rate.

Edit / Render

5.2 Design challenge

Exercise:

Design a network to beat the 95% accuracy attained by the nearest neighbor based classification.

Experiment with adding more layers and changing the batch size.

In [114]:
```
loss_fn = crossentropy
m = Chain(Dense(28^2, 30,relu),Dense(30,20),Dense(20,size(Y,1),relu),
loss(x, y) = loss_fn(m(x), y)
opt = ADAM(0.01)
```

Run

Out[114]:
```
ADAM(0.01, (0.9, 0.999), IdDict{Any,Any}())
```

In [115]:
```
batches = 80
dataset = Base.Iterators.repeated((X, Y), batches)
Flux.train!(loss, params(m), dataset, opt; cb=throttle(evalcb, 10))
```

Run

```
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [2.07603, 0.2747]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.542322, 0.8396]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.366499, 0.9005]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.294489, 0.9179]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.252485, 0.9258]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.22224, 0.9353]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.198768, 0.9412]
[Tracker.data(loss(X, Y)), accuracy(test_X, test_Y)] = [0.175948, 0.9462]
```

In [82]:
```
test_accuracy = accuracy(test_X, test_Y)
println("Test accuracy is $(test_accuracy)")
```

Validate

```
Success! You needed to be within 0.5% of 0.95 (you got 0.946).
```

```
Test accuracy is 0.946
```

5.3 Using your trained network to recognize your handwriting

Now use the network you trained to recognize your own handwriting.

In [83]:
```
include("drawnumber.jl")

function extract_digit_data(app::Canvas)
    my_img_from_app = Gray.(image(app, (28, 28)))
    my_img_from_app = float(1 .- my_img_from_app)
    my_img = my_img_from_app
    my_img_vector = my_img[:]
    return my_img_vector
end
```

▶ Run

Out[83]: extract_digit_data (generic function with 1 method)

In [122]:
```
app = Canvas()
hbox(
    app,
    map(app) do img_matrix
        if img_matrix === nothing
            return html"<b>Draw something!</b>"
        end
        digit = extract_digit_data(app)
        output = Flux.Tracker.data(m(digit))
    end
)
```

▶ Run

Out[122]:

```
10-element Array{Float32,1}:
 2.2023397e-15
 2.2023397e-15
 1.0718362e-13
 1.0
 2.2023397e-15
 1.7075214e-13
 2.2023397e-15
 2.2023397e-15
 6.4329857e-15
 6.7138176e-14
```

Clear

5.4 Example of a digit in your writing that is correctly recognized

In [123]:
```
##TODO: Your code here
input = extract_digit_data(app)
correct_label =  onecold(m(input)) - 1
```

▶ Run

Out[123]: 3

Exercise:

The neural network you trained achieved 95% accuracy on the test dataset. This would suggest that you get the correct prediction 95% of the time whenever you write something.

Comment on whether that matches your empirical observatios. Which digits were particularly difficult to get the network to correctly identify even though it clearly looked like the digit you intended.

NO, I don't seem to get correct predictions 95% of the time, I get them only like 50% of the time! Some of the digits which are more curved like 0, 6, 8 were especially difficult to identify corrrectly.

Edit / Render

5.5 Example of a digit in your writing that is *incorrectly* recognized

In [108]:

```
app = Canvas()
hbox(
    app,
    map(app) do img_matrix
        if img_matrix === nothing
            return html"<b>Draw something!</b>"
        end
        digit = extract_digit_data(app)
        output = Flux.Tracker.data(m(digit))
    end
)
```

Run

Out[108]:

Clear

```
10-element Array{Float32,1}:
 2.2023397e-15
 2.2023397e-15
 1.0718362e-13
 1.0
 2.2023397e-15
 1.7075214e-13
 2.2023397e-15
 2.2023397e-15
 6.4329857e-15
 6.7138176e-14
```

In [124]:
```
##TODO: Your code here
input = extract_digit_data(app)
applied_label =  onecold(m(input)) - 1
```

▶ Run

Out[124]: 3

```
KERNEL EXCEPTION
BoundsError: attempt to access 540-element Array{Int64,1} at index [606]

Stacktrace:
 [1] getindex(::Array{Int64,1}, ::Int64) at ./array.jl:729
 [2] (::getfield(Main, Symbol("##34#35")))(::Int64) at ./In[54]:15
 [3] (::getfield(Observables, Symbol("##16#17")){getfield(Main, Symbol("#
#34#35")),Observable{Any}})(::Int64) at /home/nbuser/.julia/packages/Obse
rvables/qCJWB/src/Observables.jl:152
 [4] (::getfield(Observables, Symbol("#g#15")){getfield(Observables, Symb
ol("##16#17")){getfield(Main, Symbol("##34#35")),Observable{Any}},Tuple{W
idget{:slider,Int64}}})(::Int64) at /home/nbuser/.julia/packages/Observab
les/qCJWB/src/Observables.jl:136
 [5] #setindex!#1(::getfield(WebIO, Symbol("##44#45")), ::Function, ::Obs
ervable{Int64}, ::Int64) at /home/nbuser/.julia/packages/Observables/qCJW
B/src/Observables.jl:87
 [6] #setindex! at ./none:0 [inlined]
 [7] setexcludinghandlers at /home/nbuser/.julia/packages/Observables/qCJ
```

Exercise:

Remark on how you can systematically produce wrong predictions of a handwritten using the deep network you have just trained.

Hiot: What happens if you write the digit so that the center of the digit is closer to the boundaries of the box. Or if it occupies a smaller portion of the box. Remark on your experiments with this.

```
Writing the digit at the boundaries and writing tiltled digits produces
wrong predictions
```

✎ Edit / Render

5.6 (optional) Convolution neural network

Convolutional neural networks overcome the obstacles above. Train and show that such a network does this.

Use code from here (https://github.com/FluxML/model-zoo/blob/master/vision/mnist/conv.jl)

In [0]:
```
##TODO: Train model here
```

▶ Run

In [0]:
```
1 ##TODO: illustrate it working on a handwritten exmaple where deep net
```

▶ Run