DEEP LEARNING

Xiaoqi Zhuang 45521225

Ziqing Yan 45551781

Zhipei Tao 45495184

Dabang Sheng 45687514

In [1]:

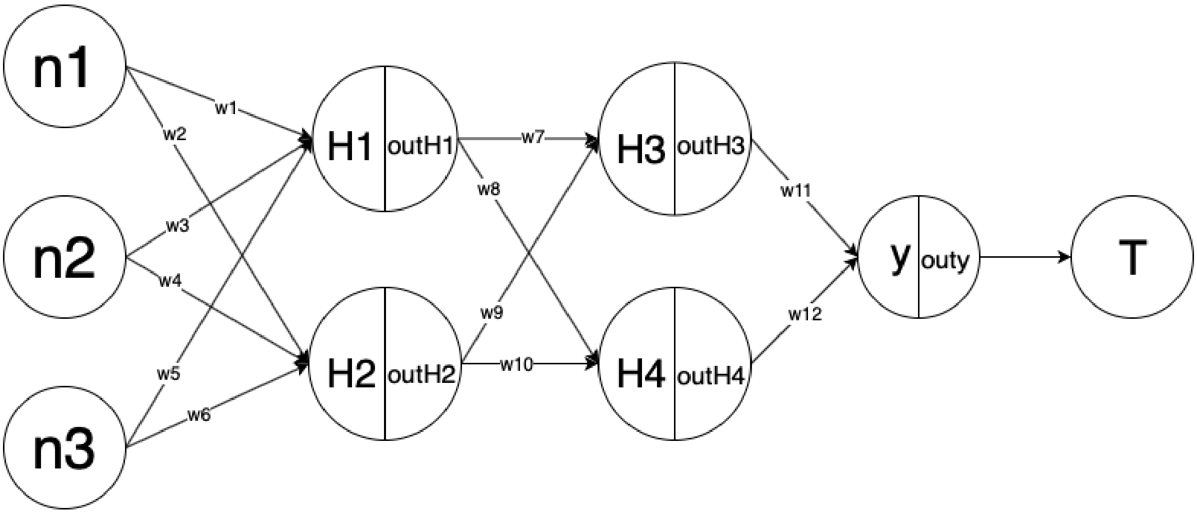
```
using Statistics
using LinearAlgebra
using Flux.Data.MNIST, PyPlot
```

The Construction of Deep Neural Networks and Backpropagation

In [24]:

```
imgs   = MNIST.images()
labels = MNIST.labels();## The Construction of Deep Neural Networks and Backpropaga
```

Out[24]:



Backpropagation algorithm is one of two ways to compute the derivatives $\frac{\partial F}{\partial x}$ backpropagation is the process taking the error and feeding backward to the error though the net work. mathematics of gradient descent tells us how to take an error to nudge weight then we calculated the error coming out of the hidden layer and keep going back and that is back propagation and how the hidden errors are calculated.

One dimentional example: Input i = 1.5, intial weight y = 0.8, desired output = 0.5, actural output a = i*w = 1.2
MSE = C = $(a-y)^2$

$\frac{\partial C}{\partial a}$ = 2(a-y) = 2*(1.2-0.5), $\frac{\partial a}{\partial w}$ = i = 1.5

$\frac{\partial C}{\partial w}$ = $\frac{\partial C}{\partial a}\frac{\partial a}{\partial w}$ = 2(a-y)*i = 2(1.5w-0.5)*1.5 = 4.5w-1.5

set learning rate r = 0.1

then w_1 = w_0 -r* $\frac{\partial C}{\partial w}$ = 0.8-0.1(4.5w_0-1.5) = 0.55$w_0 + 0.15$

In [33]:

```
function onedim(w0,DO)
    ini = 1.5;
    r = 0.1;
    AO = ini *w0;
    i = 0;
    while (AO-DO)^2 > 0.0000000000000000001
        i += 1
        w0 = 0.55*w0 + 0.15
        AO = ini *w0
    end
    return DO,i,AO
end
```

Out[33]:

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

In [29]:

```
onedim(0.8,0.5)
```

Out[29]:

```
(0.5, 35, 0.500000000572522)
```

Now, let's see a complicte example here.

As figure shows that, there are three inputs, two hidden layers with two neurons of each, and one output y.

In this case, we have 12 weights in total and let's just ignor the biase for now.

$H1 = n1 * w1 + n2 * w3 + n3 * w5$ and $outH1 = \frac{1}{1+e^{-H1}}$

$H2 = n1 * w2 + n2 * w4 + n3 * w6$ and $outH2 = \frac{1}{1+e^{-H2}}$

$H3 = outH1 * w7 + outH2 * w9$ and $outH3 = \frac{1}{1+e^{-H3}}$

$H4 = outH1 * w8 + outH2 * w10$ and $outH4 = \frac{1}{1+e^{-H4}}$

$y = outH3 * w11 + outH4 * w12$ and $outy = \frac{1}{1+e^{y}}$

$E = (outy - T)^2$

For example, we want to find $\frac{\partial E}{\partial w11}$ and $\frac{\partial E}{\partial w7}$

$\frac{\partial E}{\partial w11}$ = $\frac{\partial E}{\partial outy}\frac{\partial outy}{\partial y}\frac{\partial y}{\partial w11}$ = $2 * (outy - T) * \frac{e^{-y}}{(1+e^{-y})^2} * outH3$

$\frac{\partial E}{\partial w7}$ = $\frac{\partial E}{\partial outy}\frac{\partial outy}{\partial y}\frac{\partial y}{\partial outH3}\frac{\partial outH3}{\partial H3}\frac{\partial H3}{\partial w7}$ = $2 * (outy - T) * \frac{e^{-y}}{(1+e^{-y})^2} * w11 * \frac{e^{-H3}}{(1+e^{-H3})^2} * outH1$

and more...

Let let's put this on code. we will use function called relu and sigmoid which we will define them first.

In [34]:

```
function relu(X)
    rel = max.(0,X)
    return rel #, X
end
function Leaky_relu(X)
    if X >= 0
        return X
    else
        return 0.01*X
    end
end
function sigmoid(X)
    sigma = 1 ./(1 .+ exp.(.-X))
    return sigma
end
function sigmoidPrime(X)
    sigmaP = (exp.(.-X)) ./((1 .+ exp.(.-X)).^2)
    return sigmaP
end
```

Out[34]:

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

Here is the main code part.

In [35]:

```
#input = [1,2,3], T = must be less than 1!!
function dim3221(input,T)
    w = rand(12,1);
    dedw = zeros(12,1);
    h = zeros(4,1);
    out_h = zeros(4,1);
    r = 0.01; #set learning rate r = 0.01
    k = 0;
    h[1] = input[1]*w[1]+input[2]*w[3]+input[3]*w[5];
    h[2] = input[1]*w[2]+input[2]*w[4]+input[3]*w[6];
    out_h[1] = sigmoid(relu(h[1]));
    out_h[2] = sigmoid(relu(h[2]));
    h[3] = out_h[1]*w[7]+out_h[2]*w[9];
    h[4] = out_h[1]*w[8]+out_h[2]*w[10];
    out_h[3] = sigmoid(relu(h[3]));
    out_h[4] = sigmoid(relu(h[4]));
    y = out_h[3]*w[11]+out_h[4]*w[12];
    out_y = sigmoid(relu(y));
    E = out_y-T;
    while E^2 > 0.000000001 # we can change the
        k = k + 1
        dedw[1]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])*input[
        dedw[2]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[9]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input
        dedw[3]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])*input[
        dedw[4]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[9]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input
        dedw[5]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])*input[
        dedw[6]  = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[9]*sigmoidPrime(h
                            +w[12]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input
        dedw[7]  = 2*E*sigmoidPrime(y)*w[11]*sigmoidPrime(h[3])*out_h[1];
        dedw[8]  = 2*E*sigmoidPrime(y)*w[12]*sigmoidPrime(h[4])*out_h[1];
        dedw[9]  = 2*E*sigmoidPrime(y)*w[11]*sigmoidPrime(h[3])*out_h[2];
        dedw[10] = 2*E*sigmoidPrime(y)*w[12]*sigmoidPrime(h[4])*out_h[2];
        dedw[11] = 2*E*sigmoidPrime(y)*out_h[3];
        dedw[12] = 2*E*sigmoidPrime(y)*out_h[4];
        for i = 1: length(w)
            w[i] = w[i] - r*dedw[i]
        end
        h[1] = input[1]*w[1]+input[2]*w[3]+input[3]*w[5];
        h[2] = input[1]*w[2]+input[2]*w[4]+input[3]*w[6];
        out_h[1] = sigmoid(relu(h[1]));
        out_h[2] = sigmoid(relu(h[2]));
        h[3] = out_h[1]*w[7]+out_h[2]*w[9];
        h[4] = out_h[1]*w[8]+out_h[2]*w[10];
        out_h[3] = sigmoid(relu(h[3]));
        out_h[4] = sigmoid(relu(h[4]));
        y = out_h[3]*w[11]+out_h[4]*w[12];
        out_y = sigmoid(relu(y));
        E = out_y-T;
    end
    return k,out_y,T,E
    # k means how many time the while loop runs
    # out_y means our output
    # T is the value we want
    # E means the error
```

```
end
```

Out[35]:

```
dim3221 (generic function with 1 method)
```

In [36]:

```
dim3221([0.4,0.1,0.9],0.9)
```

Out[36]:

```
(38130, 0.8999683820338049, 0.9, -3.1617966195107705e-5)
```

V_L = b_L + $A_L v_{L-1}$ or simply w = b +Av

Our goal is to find the derivatives $\frac{\partial w_i}{\partial b_i}$ and $\frac{\partial w_i}{\partial A_{jk}}$

$\delta_{ij} = 1, for$ i = j, $otherwise = 0$

$\frac{\partial w_i}{\partial b_i} = \delta_{ij}$ $\frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} v_k$

$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ = $\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ + $\begin{bmatrix} a_{11} v_1 + a_{12} v_2 \\ a_{21} v_1 + a_{22} v_2 \end{bmatrix}$

derivatives of $\frac{\partial w1}{\partial b1} = 1$, $\frac{\partial w1}{\partial b2} = 0$, $\frac{\partial w1}{\partial a11} = v_1$, $\frac{\partial w1}{\partial a12} = v_2$, $\frac{\partial w1}{\partial a21} = 0$, $\frac{\partial w1}{\partial a22} = 0$

$M = \begin{bmatrix} 1 & O^T \\ b & A \end{bmatrix}$ has M $\begin{bmatrix} 1 \\ v \end{bmatrix}$ = $\begin{bmatrix} 1 \\ b + Av \end{bmatrix}$

$M = \begin{bmatrix} 1 & O^T \\ b & A \end{bmatrix}$, $\frac{\partial w_i}{\partial M_{jk}} = \delta_{ij} v_k$ for i > 0

$v_1 = R(b_1 + A_1 v_0)$ and $w = b_2 + A_2 v_1 = b_2 + A_2 R(b_1 + A_1 v_0)$

By chain rule, equation 5

$\frac{\partial w}{\partial A1} = \frac{\partial[A2R(b1+A1v0)]}{\partial A1}$ = $A_2 R'(b_1 + A_1 v_0)\frac{\partial(b1+A1v0)}{\partial A1}$

notice that how these formulas go BACKWARDS from w to v. write A and b for the matrix A_{L-1} and the vector b_{L-1}

$w = A_L(R(Av + b))$ and $= A_L R'(b + Av)\frac{\partial(b+Av)}{\partial A}$

$F = x^2(x + y)$ nodes $c = x^2$and $s = x + y$ and F=cs

for example $x = 2$ and $y = 3$, the edges lead to $c = 2$ and $s = 5$ and $F = 20$ this agrees with the alfebra that we normally crowd into one line: $F = x^2(x + y) = 2^2(2 + 3) = 4(5) = 20$ $c = x^2 = 4$ and $s = x + y = 5$ and $F = cs = 20$

now er compute the derivative of each step $\frac{\partial c}{\partial x} = 2x$, $\frac{\partial s}{\partial x} = 1$, $\frac{\partial F}{\partial c} = s$, $\frac{\partial F}{\partial s} = c$

$\frac{\partial F}{\partial x} = \frac{\partial F}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial F}{\partial s}\frac{\partial s}{\partial x}$ = (s)(2x)+(c)(1) = (5)(4)+(4)(1) = 24

Training the network = optimizing the weights $F(v) = A_L(RA_{L-1}(\ldots(RA_2(RA_1 v))))$ is forward through the net.

The derivatives of F with respect to the matrices A (and the bias vectors b) are the easiest for the last matrix A_L in $A_L v_{L-1}$. the dericative of Av with respect to A contains v's: $\frac{\partial F_i}{\partial A_{jk}} = \delta_{ij} v_k$. Next is the derivative of $A_L ReLU(A_{L-1} v_{L-1})$ with respect to A_{L-1}

SGD and ADAM

1. full-batch gradient descent

The above neural network has one output. In general, we will have multiple outputs and take the highest value as the result. If there are several outputs, the loss function will be more complicated and will take a long time by using the traditional gradient descent method.

The following exalmple will use gradient descent to fill the data. We design the data is $y = 3 + x$, but the computer does not know and it will use gradient descent to minimize the loss function $(Ax - y)^2$ to find the coefficient and intercept.

First, we assume the function is $y = \beta_1 + \beta_2 x$. To fit the data, we hope minimize the loss function $L(\beta) = \frac{1}{N} \sum_{j=1}^{N} \left(y_j - \hat{y}_j\right)^2 = \sum_{j=1}^{N} \frac{1}{N} \left(\beta_0 + \beta_1 x_j - y_j\right)^2$.

y_j is the real data and $\hat{y}_j$ is the data according to the predict function by real x.

The gradient is $\nabla L = \left(\frac{\partial L}{\partial \beta_0}, \frac{\partial L}{\partial \beta_1}\right) = \left(\frac{2}{N} \sum_{j=1}^{N} \left(\beta_0 + \beta_1 x_j - y_j\right), \frac{2}{N} \sum_{j=1}^{N} x_j \left(\beta_0 + \beta_1 x_j - y_j\right)\right)$

The process is $\beta_{n+1} = \beta_n - \alpha \nabla L$. α is the learing rate which should be tested many times.

We want that the algorithm will find that $\beta_1 = 3$ and $\beta_2 = 1$.

In [2]:

```
#Create the dataset.
x = collect(1:1:50)
y = 3 .+ x
```

```
...
```

Set the original $\beta = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. β_1 is the intercept and β_2 is the coefficient.

The learning rate is 0.001 and the error is 0.01.

In [7]:

```
beta = [0.0;0.0]
alpha = 0.001
tol_L = 0.001
```

Out[7]:

```
0.001
```

In [3]:

```
#calculate gradient
function Bgrad(beta,x,y)
    grad = [0.0;0.0]
    grad[1] = 2 * mean(beta[1].+beta[2]*x-y)
    grad[2] = 2 * mean(x.*(beta[1].+beta[2]*x-y))
    return grad
end
```

Out[3]:

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

In [4]:

```
#update new beta
function newBeta(beta,alpha,grad)
    beta = beta - alpha * grad
    return beta
end
```

Out[4]:

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

In [5]:

```
# the loss_function
function loss_function(beta,x,y)
    error =  (beta[1] .+ beta[2] * x - y) .^ 2
    loss = sqrt(mean(error))
    return loss
end
```

Out[5]:

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

In [8]:

```
i = 1
loss = loss_function(beta,x,y)
while loss >tol_L
    grad = Bgrad(beta,x,y)
    beta = newBeta(beta,alpha,grad)
    loss = loss_function(beta,x,y)
    println("times:",i," ","beta",beta," ","loss:",loss)
    i = i+1
end
```

```
641468174
times:108 beta[0.18372743016687956, 1.0836752825896852] loss:1.3870658
98481354
times:109 beta[0.18509253589447186, 1.0836347234446848] loss:1.3863935
588708876
times:110 beta[0.186456979927004, 1.083594183959543] loss:1.3857215451
573768
times:111 beta[0.1878207625852133, 1.0835536641247305] loss:1.38504985
71828513
times:112 beta[0.1891838841896816, 1.0835131639307223] loss:1.38437849
47894198
times:113 beta[0.1905463450608354, 1.0834726833679984] loss:1.38370745
78192647
times:114 beta[0.1919081455189458, 1.0834322224270425] loss:1.38303674
61146464
times:115 beta[0.19326928588412876, 1.0833917810983444] loss:1.3823663
595179032
times:116 beta[0.19462976647634495, 1.0833513593723965] loss:1.3816962
97871448
times:117 beta[0.19598958761540003, 1.0833109572396982] loss:1.3810265
```

It runs 6587 times and the loss is still less than 0.01. But it works, we can see that it can find the real β, which is $\begin{bmatrix} 3 \\ 1 \end{bmatrix}$.

The problem is that I try many times to find the leanring rate $\alpha = 0.001$ and everytime it will run for very lone time.

However, now our dataset is just one-dimensional and have 50 elements, which is much smaller than the real dataset.

2.Stochastic Gradient Decent, SGD

Hence, we introduce an new method: Stochastic Gradient Decent(also SGD) to help us reduce time.

The difference is that everytime it will just use one random number in x and y and run many times.

In [9]:

```
#calculate the stochastic gradient
function Sgrad(beta,x,y)
    grad = [0.0;0.0]
    r = rand(1:length(x))
    grad[1] = 2 * mean(beta[1].+beta[2]*x[r]-y[r])
    grad[2] = 2 * mean(x[r].*(beta[1].+beta[2]*x[r]-y[r]))
    return grad
end
```

Out[9]:

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

In [10]:

```
beta = [1;1]
alpha = 0.0001
tol_L = 0.01
```

Out[10]:

```
0.01
```

In [11]:

```
i = 1
loss = loss_function(beta,x,y)
while  loss >tol_L
    grad = Sgrad(beta,x,y)
    beta = newBeta(beta,alpha,grad)
    if i % 100 == 0
        loss = loss_function(beta,x,y)
        println("times:",i," ","beta",beta," ","loss:",loss)
    end
    i = i+1
end
```

```
253224866
times:4300 beta[1.3893828612775228, 1.0479154649895916] loss:0.7932614
574572691
times:4400 beta[1.3992309423553722, 1.0592271425411584] loss:0.8594746
972411568
times:4500 beta[1.4055357610820676, 1.051163457742707] loss:0.79316937
78141998
times:4600 beta[1.4120053948842979, 1.0414341336638513] loss:0.7999579
111663546
times:4700 beta[1.4217033979086147, 1.044968361644588] loss:0.77935547
367015
times:4800 beta[1.429495849216307, 1.0451860080966302] loss:0.77468827
89474012
times:4900 beta[1.4375557844576499, 1.046784570264398] loss:0.76961091
78734952
times:5000 beta[1.4448084354388686, 1.052150103250409] loss:0.78559056
12820694
times:5100 beta[1.4525098331785429, 1.0457458918407316] loss:0.7621942
484324443
times:5200 beta[1.460054851026043, 1.0487161101627058] loss:0.76344432
```

It runs 95400 times (95400/50 = 1908 times) , which is much quicker than previous gradient descent method.

4. Mini-batch Stochastic Gradient Decent

SGD just uses one random sample and the Full-batch gradient decent uses all samples. Now we want to have a trade-off between speed and stability. So in the Mini-batch Stochastic Gradient Decent, we use a number designed by us to choose samples.

In [12]:

```
function minibatch_grad(beta,batch_size,x,y)
    grad = [0.0;0.0]
    r_1 = rand(1:(length(x)-10))
    r_2 = r_1 + batch_size
    grad[1] = 2 * mean(beta[1].+beta[2]*x[r_1:r_2]-y[r_1:r_2])
    grad[2] = 2 * mean(x[r_1:r_2].*(beta[1].+beta[2]*x[r_1:r_2]-y[r_1:r_2]))
    return grad
end
```

Out[12]:

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

In [13]:

```
beta = [1;1]
alpha = 0.0001
tol_L = 0.01
batch_size = 10
```

Out[13]:

```
10
```

In [14]:

```
i = 1
loss = loss_function(beta,x,y)
while  loss >tol_L
    grad = minibatch_grad(beta,batch_size,x,y)
    beta = newBeta(beta,alpha,grad)
    if i % 100 == 0
        loss = loss_function(beta,x,y)
        println("times:",i," ","beta",beta," ","loss:",loss)
    end
    i = i+1
end
```

```
times:117200 beta[2.970442731764448, 1.000880737113736] loss:0.0145577
25868854603
times:117300 beta[2.970550276154386, 1.0009082477851456] loss:0.014537
708485265454
times:117400 beta[2.97067407727816, 1.0010771573638926] loss:0.0156550
15034219653
times:117500 beta[2.970760704193464, 1.000874315742149] loss:0.0144018
97950173497
times:117600 beta[2.9708725061278627, 1.0010472907405723] loss:0.01530
6089517897309
times:117700 beta[2.970983185754576, 1.0009751377766416] loss:0.014671
494744141051
times:117800 beta[2.9710681578109934, 1.0007960483601828] loss:0.01436
9707852067886
times:117900 beta[2.971179417674486, 1.0010362208686656] loss:0.015144
456933223877
times:118000 beta[2.971275308803145, 1.000918215751462] loss:0.0142750
79247103977
times:118100 beta[2.9714125015241493, 1.0011727287360597] loss:0.01697
4669757959915
```

In general, we almost use mini-batch stochastic gradient decent in the deep learning.

However, mini-batch stochastic gradient decent still does not solve the typical problems like finding the local minimum.

3. ADAM

In order to suppress the oscillation of SGD, ADAM believes that the gradient descent process can add inertia.

Thus we change the gradient to be $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t, \beta_1 = 0.9$

We want it to be adapitive, so we want our algorithm can metric history update frequency. The idea is to use gradients from early steps.

Thus we introduce a new variable $V_t = \sum_{\tau=1}^{t} g_\tau^2$. However, it is a typical decreasing stepsize, which may make the process end early. Therefore, we use a period instead of the whole process.Then $V_t = \beta_2 * V_{t-1} + (1-\beta_2)\, g_t^2\ , \beta_2 = 0.99$

Now, our expression is $\beta_{n+1} = \beta_n - \frac{\alpha}{\sqrt{v_t+\epsilon}} * m_t$

ϵ is a very samll number to ensure that no division by 0 and is always set to be $10e-8$.

In [15]:

```
function momentum(grad, b_1,m)
    m = b_1*m .+ (1-b_1)*grad
    return m
end
```

Out[15]:

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

In [16]:

```
function v_t(grad,b_2,v)
    v = b_2*v .+ (1-b_2)*grad.^2
    return v
end
```

Out[16]:

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

In [17]:

```
function adam_beta(beta,alpha,m,v)
    beta = beta - alpha .* m / (norm(v)+10e-8)
    return beta
end
```

Out[17]:

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

In [18]:

```
beta = [1;1]
alpha = 0.01
tol_L = 0.01
m = 0
v = 0
b_1 = 0.9
b_2 = 0.99
```

Out[18]:

```
0.99
```

At the beginning, m_t, v_t will be close to 0, so we will transfer them to $\tilde{m}_t = \frac{m_t}{(1-\beta_1^t)}$,$\tilde{v}_t = \frac{v_t}{(1-\beta_2^t)}$

In [19]:

```
i = 1
loss = loss_function(beta,x,y)
while loss > tol_L
    grad = Sgrad(beta,x,y)
    m = momentum(grad, b_1,m)
    m = m /(1-b_1^i)
    v = v_t(grad, b_2,v)
    v = v /(1-b_2^i)
    beta = adam_beta(beta,alpha,m,v)
    if i % 100 == 0
        loss = loss_function(beta,x,y)
        println("times:",i," ","beta",beta," ","loss:",loss)
    end
    i = i+1
end
```

```
13406631346
times:147100 beta[2.35200738816057, 1.0185719898299614] loss:0.3197610
0155201903
times:147200 beta[2.3541114546807442, 1.019617183351939] loss:0.318364
0782196124
times:147300 beta[2.3560573583712014, 1.0187701304654138] loss:0.31732
58543629600 7
times:147400 beta[2.3581068113645998, 1.0211314523494774] loss:0.32188

36371068729
times:147500 beta[2.3596503820862664, 1.0178302993618489] loss:0.31730
53639946139
times:147600 beta[2.361952495337308, 1.0171145441127418] loss:0.318828
0253420584
times:147700 beta[2.36373894429926, 1.0176511520862155] loss:0.3154953
8621836426
times:147800 beta[2.3656104693095212, 1.0157353140428125] loss:0.32544
819304261857
times:147900 beta[2.367544200874093, 1.0202018319323225] loss:0.314246
9803893593
```

It runs $168600/50 = 3372$ times.

CNN

Edge detection using convolution.

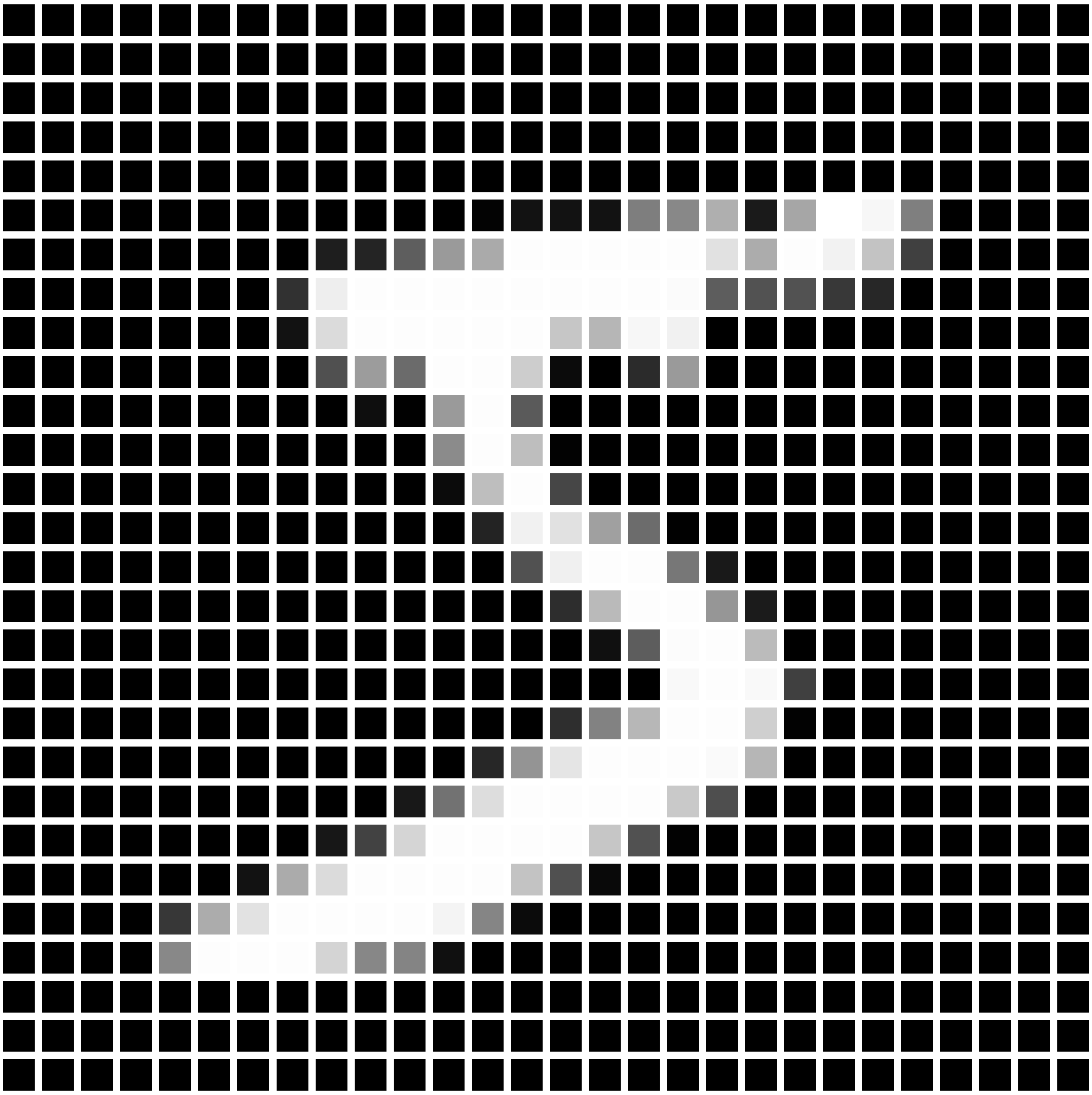
In [2]:

```
imgs   = MNIST.images()
labels = MNIST.labels();## Edge detection using convolution.
```

In [3]:

```
Sample_image = imgs[1]
```

Out[3]:



In [4]:

```
image = float.(Sample_image);
```

In [5]:

```
x_filter = 1/2*[-1 0 1;
                -2 0 2;
                -1 0 1]
```

Out[5]:

```
3×3 Array{Float64,2}:
 -0.5  0.0  0.5
 -1.0  0.0  1.0
 -0.5  0.0  0.5
```

In [6]:

```
function edge_detecter(img, filter)
    N = Int(sqrt(length(img)))
    n = Int(sqrt(length(filter)))
    result_image = []
    for j in 1:N-n+1
        for i in 1:N-n+1
            window_matrix = img[i:i+n-1,j:j+n-1]
            filtered_value = ones(n)'*(window_matrix .* filter)*ones(n)
            push!(result_image, filtered_value)
        end
    end
    return reshape(result_image,(N-2,N-2))
end
```

Out[6]:

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

In [7]:

```
using Plots
gr()
```

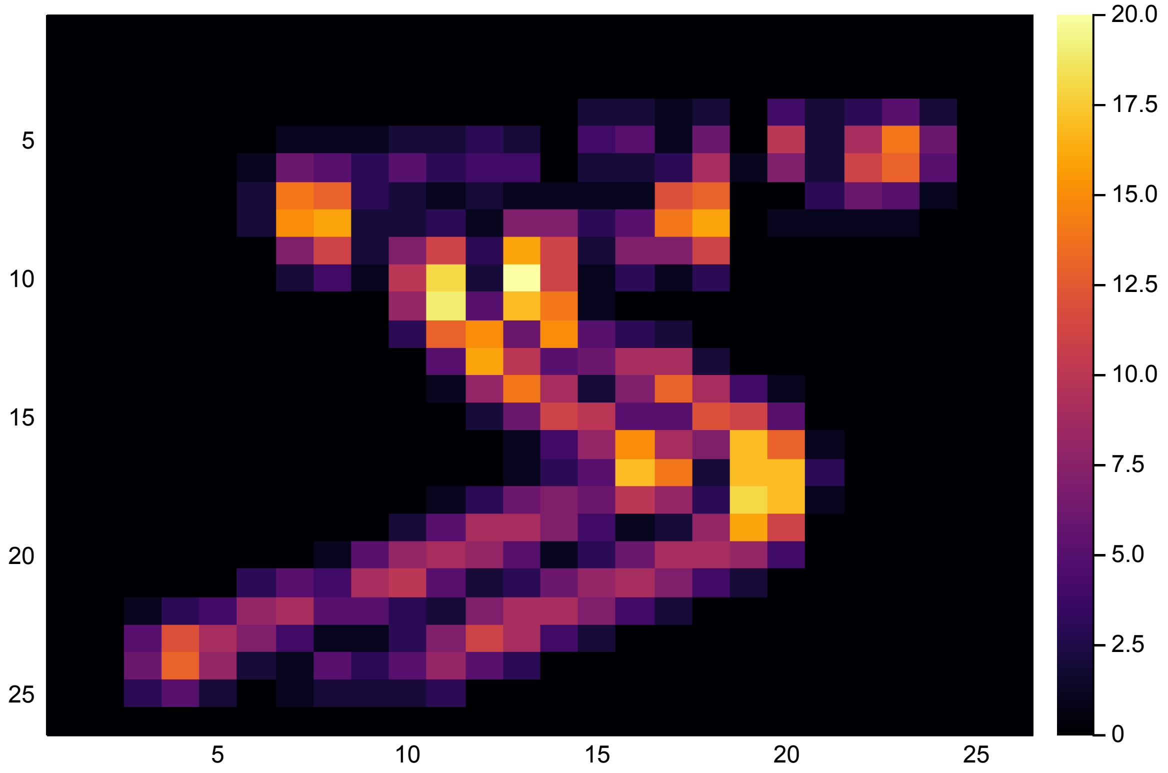
Out[7]:

```
Plots.GRBackend()
```

In [8]:

```
filtered_image_1 = edge_detecter(image, x_filter);
filtered_image_x = abs.(Int.(round.(10 .* filtered_image_1 ));)
heatmap(filtered_image_x, yflip = true)
```

Out[8]:



In [9]:

```
y_filter = 1/2*[-1 -2 -1;
                0 0 0;
                1 2 1]
```

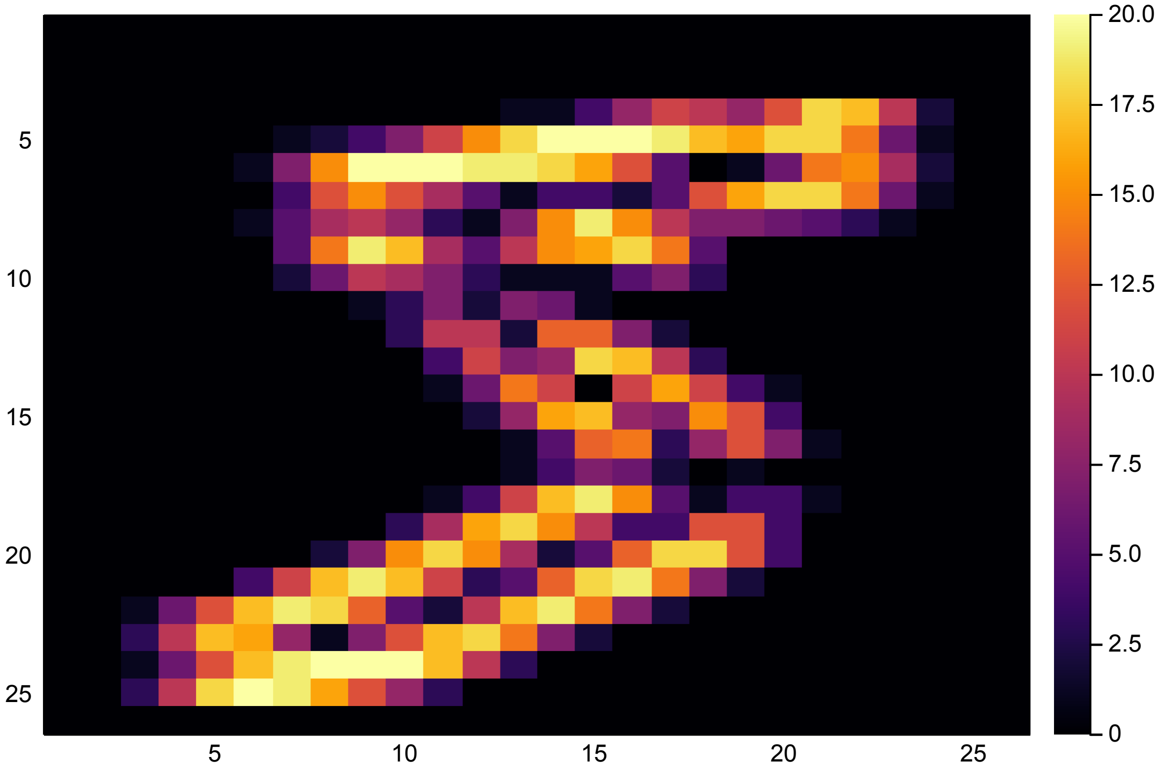
Out[9]:

```
3×3 Array{Float64,2}:
 -0.5  -1.0  -0.5
  0.0   0.0   0.0
  0.5   1.0   0.5
```

In [10]:

```
filtered_image_2 = edge_detecter(image, y_filter);
filtered_image_y = abs.(Int.(round.(10 .* filtered_image_2));)
heatmap(filtered_image_y, yflip = true)
```

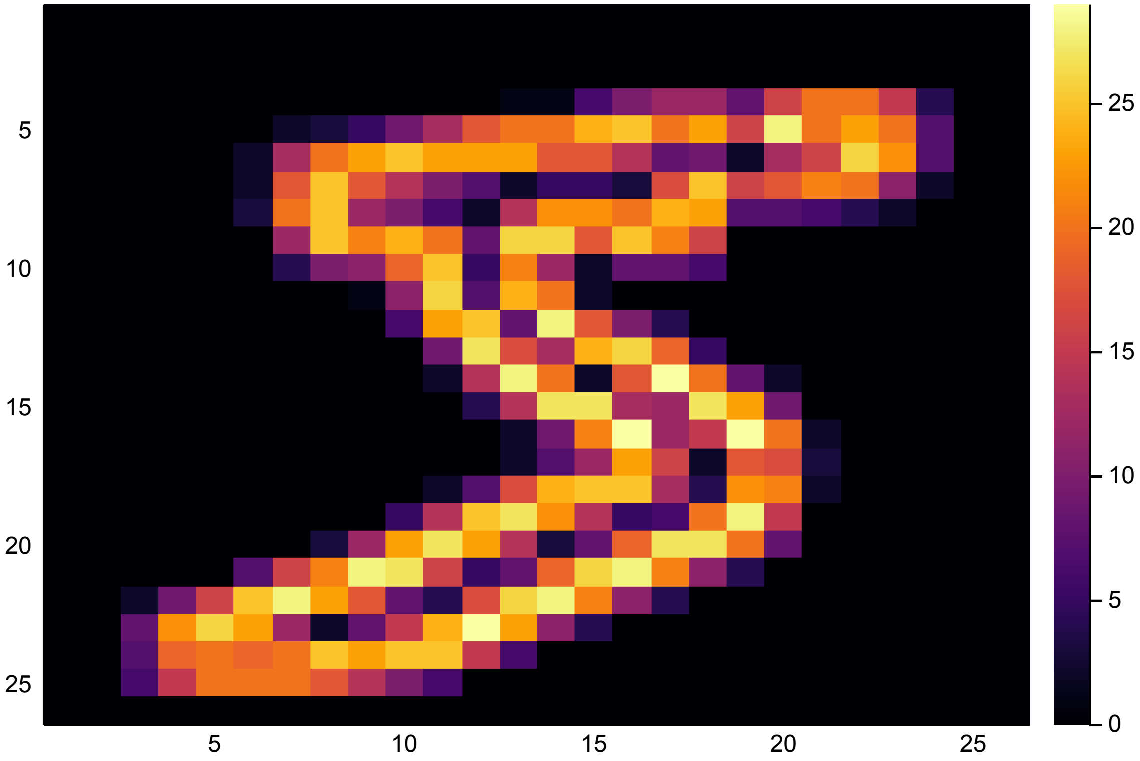
Out[10]:



In [11]:

```
edge_image = filtered_image_x + filtered_image_y
heatmap(edge_image, yflip = true)
```

Out[11]:



Smoothing images using convolution

In [12]:

```
function smoothing(img, smoother)
    N = Int(sqrt(length(img)))
    n = Int(sqrt(length(smoother)))
    weight = ones(n)'*smoother*ones(n)
    result_image = []
    for j in 1:N-n+1
        for i in 1:N-n+1
            window_matrix = img[i:i+n-1,j:j+n-1]
            smoothed_value = ones(n)'*(window_matrix .* smoother)*ones(n)
            push!(result_image, smoothed_value)
        end
    end
    return reshape(result_image,(N-2,N-2))
end
```

Out[12]:

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

In [13]:

```
smooth_matrix = [1 2 1;
                 2 8 2;
                 1 2 1]
```

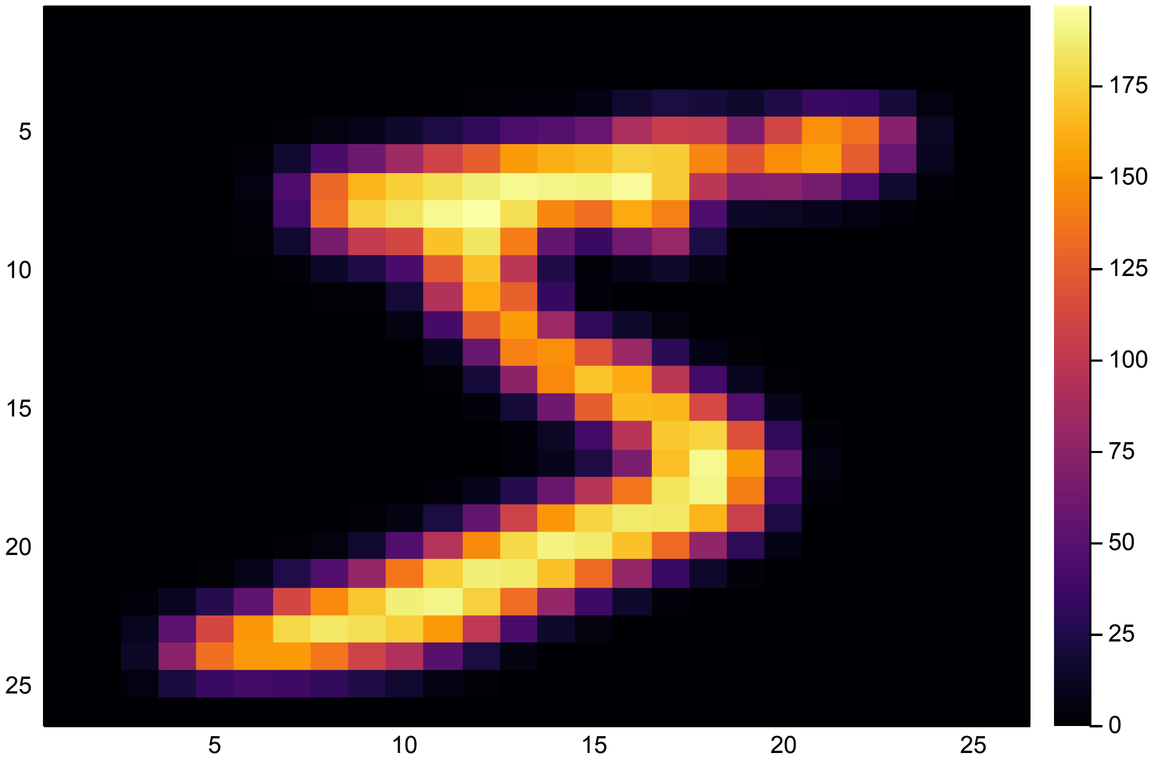
Out[13]:

```
3×3 Array{Int64,2}:
 1  2  1
 2  8  2
 1  2  1
```

In [14]:

```
smoothed_image_1 = smoothing(image, smooth_matrix);
#smoothed_image = abs.(Int.(round.(10 .* smoothed_image_1)))
smoothed_image = Int.(round.(10 .* smoothed_image_1))
heatmap(smoothed_image, yflip = true)
```

Out[14]:



In [15]:

```
[size(Sample_image),size(filtered_image_x),size(filtered_image_y),size(smoothed_imag
```

Out[15]:

```
4-element Array{Tuple{Int64,Int64},1}:
 (28, 28)
 (26, 26)
 (26, 26)
 (26, 26)
```

Edge issue

$n \times n$ matrix $\left[A \right]$, could be extended to $\begin{bmatrix} 0 & 0 & 0 \\ 0 & A_{n\times n} & 0 \\ 0 & 0 & 0 \end{bmatrix}$

In [16]:

```
function extend_edge(img, value, edge_num)
    N = Int(sqrt(length(img)))
    head_edge = fill(value, N+2*edge_num, edge_num)
    side_edge = fill(value,N, edge_num)
    return [head_edge';
            side_edge img side_edge;
            head_edge']
end
```

Out[16]:

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

In [17]:

```
extended_image = extend_edge(image, 0, 1)
filtered_extended_image = edge_detecter(extended_image, x_filter);
filtered_extended_image_x = Int.(round.(10 .* filtered_extended_image));
size(filtered_extended_image_x)
```

Out[17]:

```
(28, 28)
```

Max Pooling

In [18]:

```
function max_pooling(img, pooling_size, step)
    N = Int(sqrt(length(img)))
    output_array = []
    if N%pooling_size == 0
        for j in 1:step:N-pooling_size
            for i in 1:step:N-pooling_size
                window = img[i:i+pooling_size,j:j+pooling_size]
                result = maximum(window)
                push!(output_array, result)

            end
        end
    end
    if N%pooling_size != 0
        for j in 1:step:N-pooling_size
            for i in 1:step:N-pooling_size
                window = img[i:i+pooling_size,j:j+pooling_size]
                result = maximum(window)
                push!(output_array, result)
            end
        end
    end
    size = Int(sqrt(length(output_array)))
    output_image = reshape(output_array,size,size)
end
```

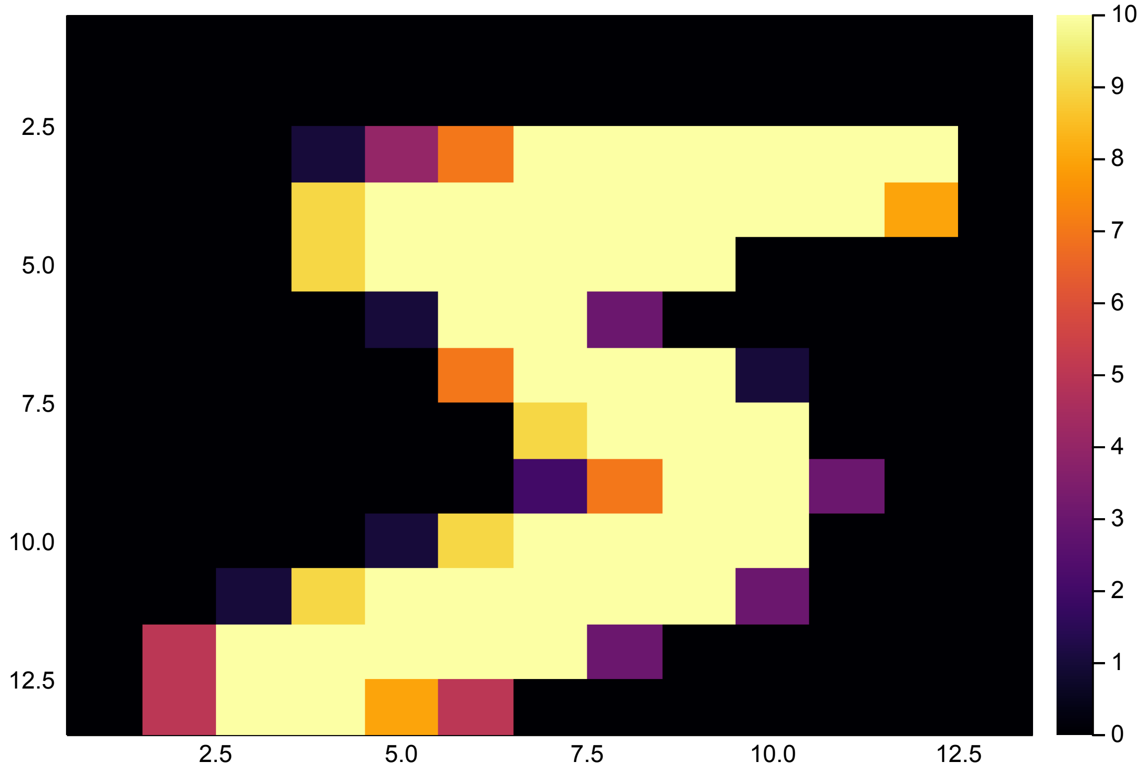
Out[18]:

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

In [19]:

```
pooled_image_1 = max_pooling(image, 2, 2);
pooled_image = Int.(round.(10 .* pooled_image_1))
heatmap(pooled_image, yflip=true)
```

Out[19]:



In [ ]:

```

```