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();
```

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 $a = i*w = 1.2$

\[
MSE = C = (a - y)^2
\]

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

\[
\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a} \frac{\partial a}{\partial w} = 2(a-y)i = 2(1.5w-0.5)1.5 = 4.5w-1.5
\]
set learning rate \( r = 0.1 \)

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

In [33]:

```python
function onedim(w0,DO)
    ini = 1.5;
r = 0.1;
AO = ini * w0;
i = 0;
    while (AO-DO)^2 > 0.000000000000000001 
        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]:

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

Out[29]:

(0.5, 35, 0.5000000000572522)

Now, let's see a complicated 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 ignore the bias for now.

\[
H_1 = n_1 \times w_1 + n_2 \times w_3 + n_3 \times w_5 \text{ and } out \, H_1 = \frac{1}{1+e^{-H_1}}
\]

\[
H_2 = n_1 \times w_2 + n_2 \times w_4 + n_3 \times w_6 \text{ and } out \, H_2 = \frac{1}{1+e^{-H_2}}
\]

\[
H_3 = out \, H_1 \times w_7 + out \, H_2 \times w_9 \text{ and } out \, H_3 = \frac{1}{1+e^{-H_3}}
\]

\[
H_4 = out \, H_1 \times w_8 + out \, H_2 \times w_10 \text{ and } out \, H_4 = \frac{1}{1+e^{-H_4}}
\]

\[
y = out \, H_3 \times w_11 + out \, H_4 \times w_12 \text{ and } \frac{1}{1+e^y}
\]

\[
E = (out \, y - T)^2
\]

For example, we want to find \( \frac{\partial E}{\partial w_{11}} \) and \( \frac{\partial E}{\partial w_{07}} \)

\[
\frac{\partial E}{\partial w_{11}} = \frac{\partial E}{\partial out} \frac{\partial out}{\partial y} \frac{\partial y}{\partial w_{11}} = 2 \times (out \, y - T) \times \frac{e^{-y}}{(1+e^{-y})^2} \times out \, H_3
\]

\[
\frac{\partial E}{\partial w_{07}} = \frac{\partial E}{\partial out} \frac{\partial out}{\partial y} \frac{\partial y}{\partial out \, H_3} \frac{\partial out \, H_3}{\partial H_3} \frac{\partial H_3}{\partial w_{07}} = 2 \times (out \, y - T) \times \frac{e^{-y}}{(1+e^{-y})^2} \times w_{11} \times \frac{e^{-H_3}}{(1+e^{-H_3})^2} \times out \, H_1
\]

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]:

```matlab
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;
    out_h[1] = sigmoid(relu(h[1]));
    out_h[2] = sigmoid(relu(h[2]));
    out_h[3] = sigmoid(relu(h[3]));
    out_h[4] = sigmoid(relu(h[4]));
    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[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[1];
        dedw[2] = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[2];
        dedw[3] = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[3];
        dedw[4] = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[4];
        dedw[5] = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[5];
        dedw[6] = 2*E*sigmoidPrime(y)*(w[11]*sigmoidPrime(h[3])*w[7]*sigmoidPrime(h[4])*w[8]*sigmoidPrime(h[1])]
        +w[12]*sigmoidPrime(h[4])*w[9]*sigmoidPrime(h[4])*w[10]*sigmoidPrime(h[2])*input[6];
        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]));
        dedw[10] = 2*E*sigmoidPrime(y)*(w[12]*sigmoidPrime(h[4]));
        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
    out_h[1] = sigmoid(relu(h[1]));
    out_h[2] = sigmoid(relu(h[2]));
    out_h[3] = sigmoid(relu(h[3]));
    out_h[4] = sigmoid(relu(h[4]));
    out_y = sigmoid(relu(y));
    E = out_y-T;
    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
\[ V_L = b_L + A_L v_{L-1} \] or simply \( w = b + Av \)

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

\[ \delta_{ij} = 1, \text{ for } i = j, \text{ otherwise } = 0 \]

\[ \frac{\partial w_i}{\partial b_i} = \delta_{ij} \frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} \]

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

derivatives of \( \frac{\partial w_1}{\partial b_1} = 1, \frac{\partial w_1}{\partial b_2} = 0, \frac{\partial w_1}{\partial a_{11}} = v_1, \frac{\partial w_1}{\partial a_{12}} = v_2, \frac{\partial w_1}{\partial a_{21}} = 0, \frac{\partial w_1}{\partial a_{22}} = 0 \)

\[ M = \begin{bmatrix} 1 & 0^T \\ b & A \end{bmatrix} \]

\[ \begin{bmatrix} \frac{\partial w_i}{\partial b_i} \end{bmatrix} = \begin{bmatrix} 1 \\ b + Av \end{bmatrix} \]

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

\[ v_1 = R(b_1 + A_1 v_0) \text{ and } w = b_2 + A_2 v_1 = b_2 + A_2 R(b_1 + A_1 v_0) \]

By chain rule, equation 5

\[ \frac{\partial w}{\partial A_1} = \frac{\partial [A_2 R(b_1 + A_1 v_0)]}{\partial A_1} = A_2 R'(b_1 + A_1 v_0) \frac{\partial (b_1 + A_1 v_0)}{\partial A_1} \]

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

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

\[ F = x^2 (x + y) \text{ nodes } c = x^2 \text{ and } s = x + y \text{ and } F = cs \]

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 \( F(v) = A_L(RA_{L-1}(\ldots(RA_2(RA_1v)))) \) is forward through the network.

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 \( A_Lv_{L-1} \). The derivative of \( Av \) with respect to \( A \) contains \( v \)'s: \( \frac{\partial F}{\partial A_{jk}} = \delta_{ij}v_k \). Next is the derivative of \( A_LReLU(A_{L-1}v_{L-1}) \) 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 example 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 \( y = \beta_1 + \beta_2x \). To fit the data, we hope minimize the loss function

\[
L(\beta) = \frac{1}{N} \sum_{j=1}^{N} (y_j - \hat{y}_j)^2 = \sum_{j=1}^{N} \frac{1}{N} (\beta_0 + \beta_1x_j - y_j)^2.
\]

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

The gradient is

\[
\nabla L = \left( \frac{\partial L}{\partial \beta_0}, \frac{\partial L}{\partial \beta_1} \right) = \left( \frac{2}{N} \sum_{j=1}^{N} (\beta_0 + \beta_1x_j - y_j), \frac{2}{N} \sum_{j=1}^{N} x_j (\beta_0 + \beta_1x_j - y_j) \right)
\]

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

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

In [2]:

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

Set the original \( \hat{\beta} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \). \( \beta_1 \) is the intercept and \( \beta_2 \) is the coefficient.

The learning rate is 0.001 and the error is 0.01.
In [7]:

\[
\begin{align*}
\beta &= [0.0; 0.0] \\
\alpha &= 0.001 \\
\text{tol}_L &= 0.001
\end{align*}
\]

Out[7]:

0.001

In [3]:

# Calculate gradient

function Bgrad(beta, x, y)
    grad = [0.0; 0.0]
    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)
    loss = sqrt(mean(error))
    return loss
end

Out[5]:

loss_function (generic function with 1 method)
In [8]:

```jupyter
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", "," loss":loss)
    i = i+1
```

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 $\beta$, which is $\begin{bmatrix} 3 \\ 1 \end{bmatrix}$.

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

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]:

```matlab
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]:

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

Out[13]:

10
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 adaptative, 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 \( V_t = \sum_{t'=1}^{t} g_t^2 \). 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

\[ V_t = \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot g_t^2, \beta_2 = 0.99 \]

Now, our expression is

\[ \beta_{t+1} = \beta_n - \frac{\alpha}{\sqrt{V_t + \epsilon}} \cdot m_t \]

\( \epsilon \) is a very small number to ensure that no division by 0 and is always set to be \( 10\epsilon - 8 \).
At the beginning, $m_t$, $v_t$ will be close to 0, so we will transfer them to $\tilde{m}_t = \frac{m_t}{(1-\beta_1)}$, $\tilde{v}_t = \frac{v_t}{(1-\beta_2)}$. 

\[ m_t = b_1 \cdot m_t + (1-b_1) \cdot \text{grad} \\
\text{return } m \\
\]

\[ v_t = b_2 \cdot v_t + (1-b_2) \cdot \text{grad}^2 \\
\text{return } v \\
\]

\[ \beta = \beta - \alpha \cdot m / (\text{norm}(v)+10e-8) \\
\text{return } \beta \\
\]
In [19]:

```python
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)
    i = i + 1
end
```

It runs 168600/50 = 3372 times.

CNN

Edge detection using convolution.

In [2]:

```python
imgs = MNIST.images()
labels = MNIST.labels();## Edge detection using convolution.
```
In [3]:

Sample_image = imgs[1]

Out[3]:

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 [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]:

```python
function edge_detector(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_detector` (generic function with 1 method)

In [7]:

```python
using Plots
gr()
```

Out[7]:

Plots.GRBackend()

In [8]:

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

Out[8]:

![Heatmap Image](image-url here)
In [9]:

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

Out[9]:

3x3 Array{Float64,2}:
-0.5  -1.0  -0.5
 0.0   0.0   0.0
 0.5   1.0   0.5

In [10]:

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

Out[10]:

![Heatmap Image](image-url)
In [11]:

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

Out[11]:

![Image](image_url)

**Smoothing images using convolution**

In [12]:

```python
function smoothing(img, smoother)
    N = Int(sqrt(length(img)))
    n = Int(sqrt(length(smoothen)))
    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]:

3x3 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]:

![Heatmap of smoothed image](image_url)

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 $\begin{bmatrix} A \end{bmatrix}$, could be extended to $\begin{bmatrix} 0 & 0 & 0 \\ 0 & A_{n \times n} & 0 \\ 0 & 0 & 0 \end{bmatrix}$

In [16]:

```python
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]:

```python
extended_image = extend_edge(image, 0, 1)
filtered_extended_image = edge_detector(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]:

```julia
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
        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]:

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

Out[19]: