DEEP LEARNING & APPLICATIONS

Convolutional Neural Networks

9th May 2022

CONTENTS

CONVOLUTIONAL NEURAL NETWORK (CNN)

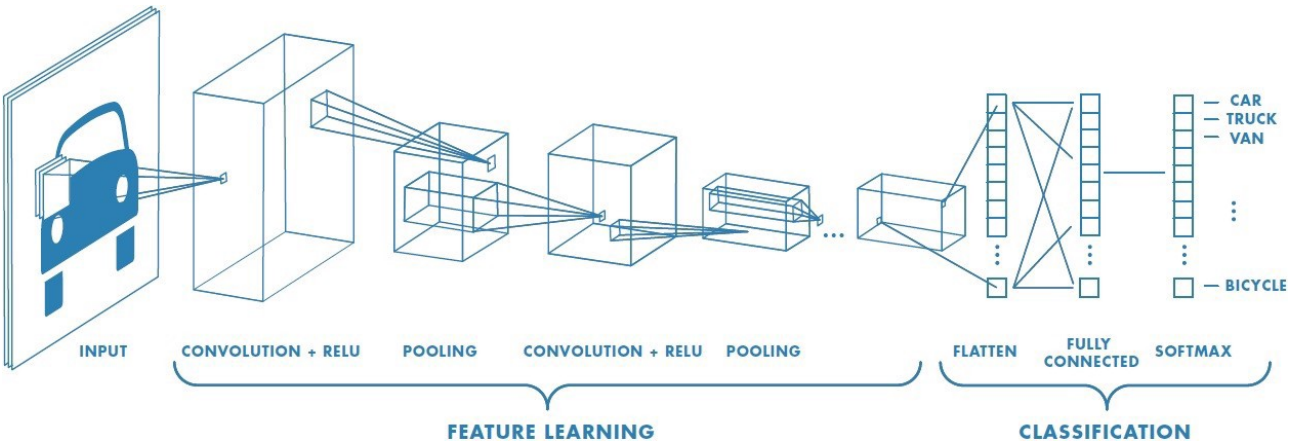


Figure describes the CNN architecture for vehicle classification. Reference: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

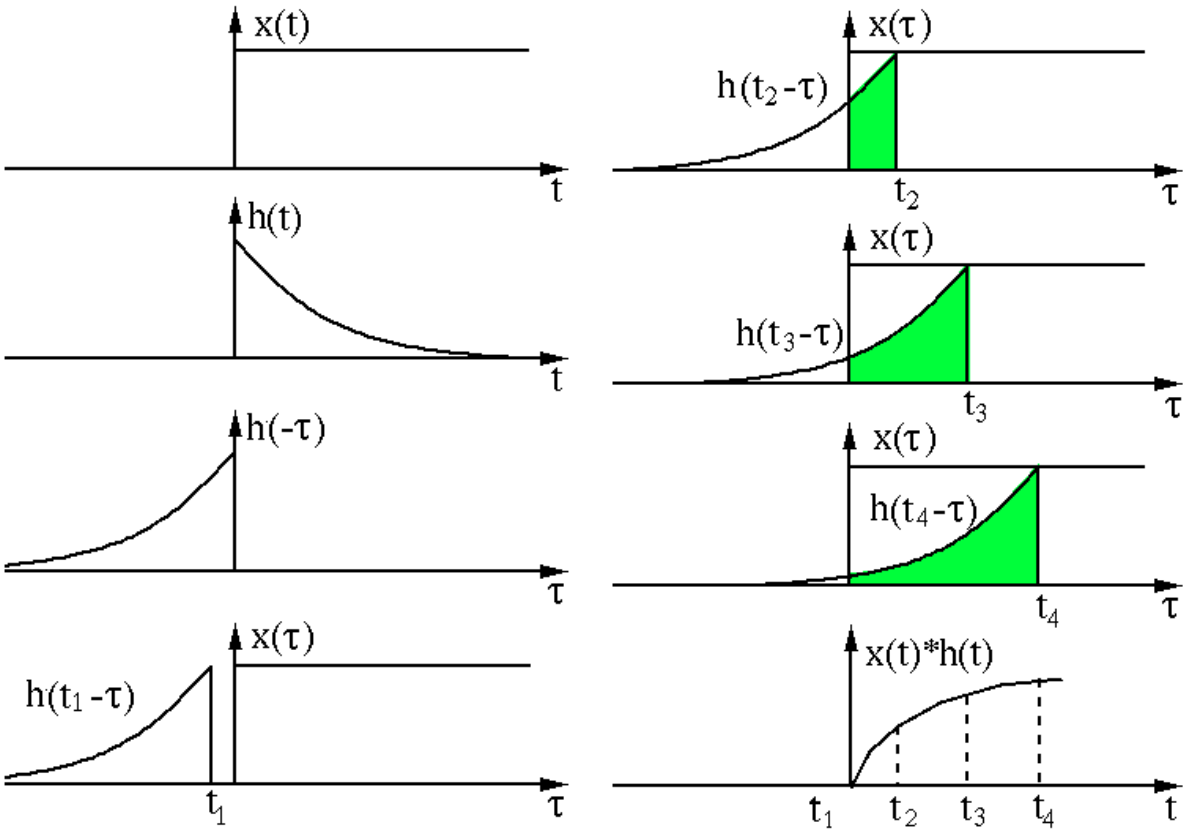
- Fully Connected Neural Network (FCNN) are the type of neural network where all the nodes in one layer are connected to all the nodes in other layer.
- While working with images, for classification or pattern recognition task, the neighbouring pixels are more important than the every pixel available. So instead working with whole pixels in the image for feature extractions, it is good to consider only the neighbouring pixels.
- In this way, we can achieve sparse connection which leads to faster computation, less memory etc.
- One such way to obtain this is by performing convolution and the network associated to this is called Convolutional Neural Network (CNN).
- In this report, CNN is break down into three parts such as Convolution Operation, Building Layers and Forming Network to understand the work flow of CNNs.
- Finally the network is trained using stochastic gradient descent algorithm and compared it with VGG19 pertained network.
- Also, the weights of the network trained using SGD, VGG19 is viewed using Grad-CAM.
- We have used Caltech-101 dataset of three classes each image of size 224 x 224 x 3.

THE CONVOLUTION THEORY

Convolution is a mathematical operation on two functions (x and h) that produces a third function $x(t) * h(t)$ that expresses how the shape of one is modified by the other. It is defined as the integral of the product of the two functions after one is reversed and shifted. The integral is evaluated for all values of shift, producing the convolution function.

1D Continuous case: $$x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau$$

1D Discrete case: $$x(n) * h(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k)$$

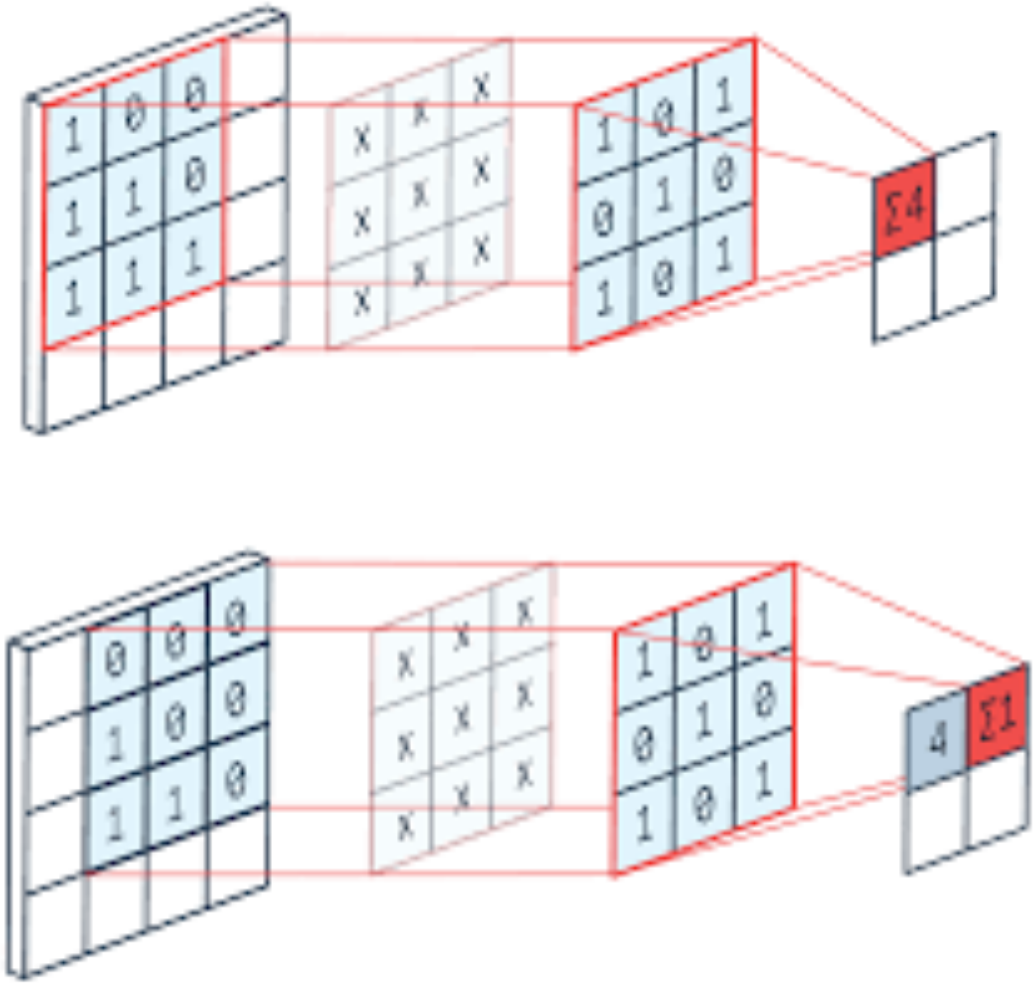


1D convolution example describes how the function1 changes wrt to function 2. Reference: https://abhipray.com/posts/sigproc/lti_lds/

Similarly the two dimensional convolution is given by,

2D Discrete case: $$x(m,n) * h(m,n) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} x(k,l)h(n-k,n-l)$$

This is without the stride and zero padding. Stride is the no of rows and columns to skip while performing convolution and zero padding is the intuition to include the zeros around corners of the pixels.



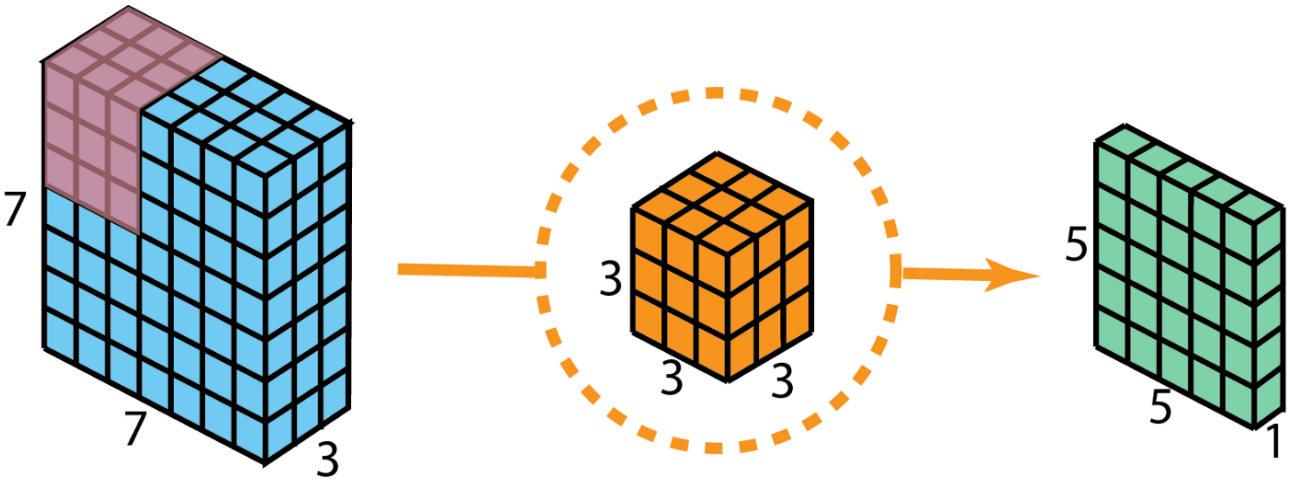
2D convolution with stride 1 and padding 0. Reference: https://medium.com/analytics-vidhya/2d-convolution-using-python-numpy-43442ff5f381

The equation is modified with stride and padding as follows,

$$s_{ij} = (\mathbf{I} * \mathbf{K})_{ij} = \sum_{a=-\left\lfloor \frac{m}{2} \right\rfloor}^{\left\lfloor \frac{m}{2} \right\rfloor} \sum_{b=-\left\lfloor \frac{n}{2} \right\rfloor}^{\left\lfloor \frac{n}{2} \right\rfloor} I_{i+a, j+b} \; K_{\left\lfloor \frac{m}{2} \right\rfloor + a, \left\lfloor \frac{n}{2} \right\rfloor + b}$$

3D Convolution: Here we are considered that the depth of the filter is same as the depth of the input. Mathematically it is given by,

$$s_{kij} = (\mathbf{I} * \mathbf{K})_{kij} = \sum_{k=1}^{D_1} \left[\sum_{a=-\left\lfloor \frac{m}{2} \right\rfloor}^{\left\lfloor \frac{m}{2} \right\rfloor} \sum_{b=-\left\lfloor \frac{n}{2} \right\rfloor}^{\left\lfloor \frac{n}{2} \right\rfloor} I_{k,\, i+a, j+b} \; K_{k,\, \left\lfloor \frac{m}{2} \right\rfloor + a, \left\lfloor \frac{n}{2} \right\rfloor + b} \right]$$



3D convolution with stride 1 and padding 0 and depth of the input is equal to the depth of the filter. Reference: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

It is important to note here is that the output is of dimension 2 even though the filter and input is of dimension 3.

Convolution is the fundamental building block for the convolutional neural network. Here we assume the depth of the filter is always equal to the depth of the input.

Relationship Between Input, Filter and Output Dimensions

Let $I(m,n)$ be the image of dimension mxn and the filter be $F(k,l)$ of dimension kxl. Let the stride be S and the padding be P then,

$$Om = \frac{m-k+2P}{S}+1$$

$$On = \frac{n-l+2P}{S}+1$$

Where the feature map is given by $F_m(Om, On)$.

KAIMING INITIALIZATION

Another important task to take care is that to initialise weights. The requirement is to Initialize the weights such that activation values will not blow up or shrink. One such method that takes care is Kaiming Initialisation. The is for ReLU activation.

Initialize the weights in a layer such that they are drawn from a distribution such that $\frac{1}{2}n_j var(w_j) = 1$ where the n_j is the number of incoming connections to a neuron in jth layer. ReLU need to account for factor of 2 as half the time it will not produce the output.

This leads to draw the weights from zero mean Gaussian distribution whose standard deviation is $\frac{2}{n_j}$. Also, initialising the bias term (w_0) to 0.

ALGORITHM

```
def kaiming(n, size):
    mean, sd = 0, np.sqrt(2/n)
    weights = np.random.normal(mean, sd, size=size)
    return weights
```

1. Input the image $I(m,n)$ and initialise the filter $F(k,l)$ using kaiming initialisation.
2. Compute $F(-k,-l)$
3. Compute the expected output dimensions using
   $Om = \frac{m-k+2P}{S} + 1$ and $On = \frac{n-l+2P}{S} + 1$
4. Define empty array with expected dimensions as feature map
5. *for i in range(0, m, stride):*
   for j in range(0, n, stride):
   if I[i:i+k, j:j+l].shape == F.shape:
   *feature_map.append((F*I[i:i+k, j:j+l]).sum())*

RESULTS

A image is chosen from each classification from the Caltech-101 dataset and performed the convolution. Image is loaded as grayscale and the result is attached.

FILTER VALUES		
-2.72900218959001E-01	6.72669997706184E-01	1.30577593308025E-01
3.7225265525887E-01	1.51827193173132E-01	3.3016813284712E-01
1.83242777787809E-01	-1.94519704685973E-02	1.39481206653026E-01

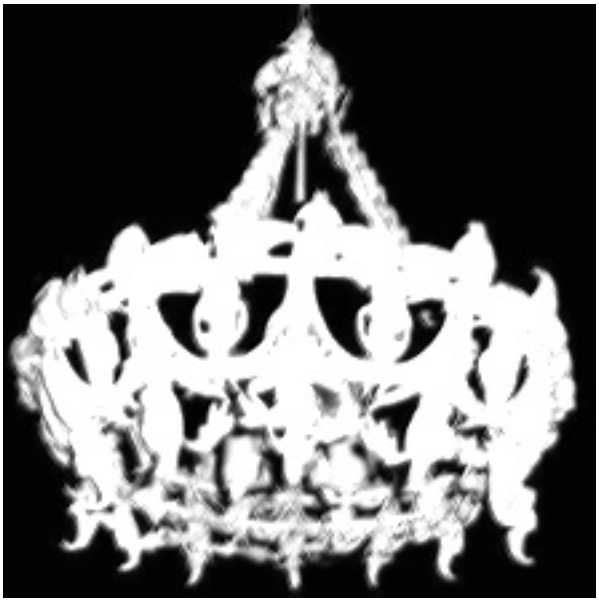
INPUT IMAGE

Input dimension: 224 x 224 x 3

RESULTANT GRAYSCALE IMAGE

Grayscale dimension: 224 x 224

RESULTANT FEATURE MAP



Feature Map dimension: 222 x 222

EXPECTED OUTPUT DIMENSION (THEORY)

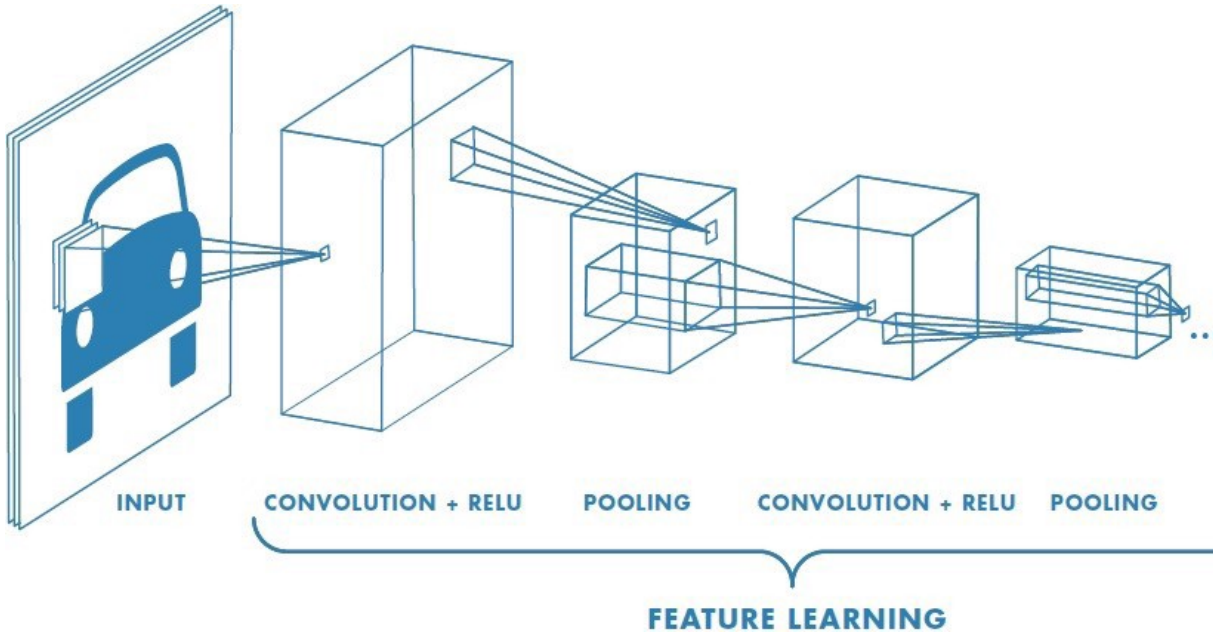
Input Grayscale image $I(m,n)$ size: $m = 224$ and $n = 224$
Filter size $F(k,l)$: $k = 3$ and $l = 3$
Stride: $S = 1$
Padding: $P = 0$

Output feature map dimension:

$$Om = \frac{m - k + 2P}{S} + 1 = \frac{224 - 3 + 2(0)}{1} + 1 = 222$$

$$On = \frac{n - l + 2P}{S} + 1 = \frac{224 - 3 + 2(0)}{1} + 1 = 222$$

BUILDING CONVOLUTIONAL LAYERS



Now the next step is to build the convolution layer with ReLU activation function. Initialising 3*3 kaiming filter for each of the layers. Building the input layer by flattening the input 224 x 224. Passed to convolution class written earlier.

Architecture

Layers	Details	Size
Input Layer	224 x 224	50176 x 1
Convolutional Layer I	S = 1, P = 0, F = 3 x 3, K = 32	222 x 222 x 32
Max Pooling	-	-
Convolutional Layer II	S = 1, P = 0, F = 3 x 3, K = 64	220 x 220 x 64
Max Pooling	-	-
Flattening	-	-

ALGORITHM

```
def ConVlayer(self, image, K, size, n):
    output = []
    ReLUout = []
    filters = []
    for i in range(K):
        filtr = self.kaiming(n, size)
        filters.append(filtr)
        res = cv.convolve(cv, inpt=image, filtr=filtr, stride = 1, padding = 0)
        output.append(res)
        ReLUout.append(self.ReLUActivation(res))

    self.output = np.array(output)
    self.ReLUout = np.array(ReLUout)
    self.filters = np.array(filters)
```

```
def ConVDepthlayer(self, images, K, size, n):
    output = []
    ReLUout = []
    filters = []
    for i in range(K):
        filtr = self.kaiming(n, size)
        filters.append(filtr)
        res = cv.convwithDepth(cv, inpt=images, filtr=filtr, stride = 1, padding = 0)
        output.append(res)
        ReLUout.append(self.ReLUActivation(res))

    self.output = np.array(output)
    self.ReLUout = np.array(ReLUout)
    self.filters = np.array(filters)
```

```
def poolingLayer(self, featureMaps, size, stride):
    m, n = featureMaps[0].shape
    fm, fn = size
    result = []

    poolout = []
    for featureMap in featureMaps:
        for i in range(0, m, stride):
            for j in range(0, n, stride):
                if featureMap[i:i+fm, j:j+fn].shape == size:
                    result.append(max(featureMap[i:i+fm, j:j+fn].flatten()))

        Outm, Outn = int(((m-fm)/stride)+1), int(((n-fn)/stride)+1)
        poolout.append(np.array(result).reshape((Outm, Outn)))
        result = []

    self.poolout = np.array(poolout)
```

RESULTS

The same image considered above is passed to the two layers of convolution with k = 32 and 64 in each layers. The resultant feature maps and their respective filters as image as shown.

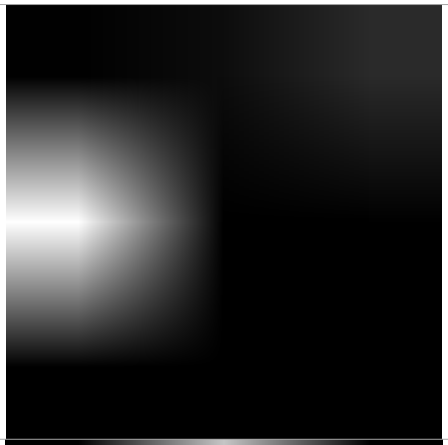
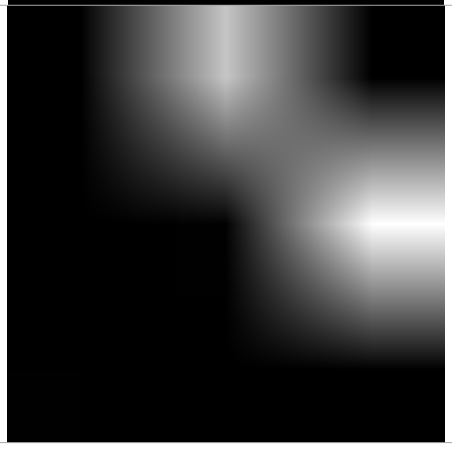
Input Image Dimension: 224 x 224

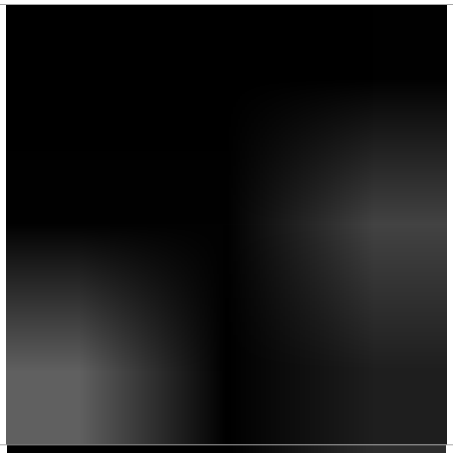
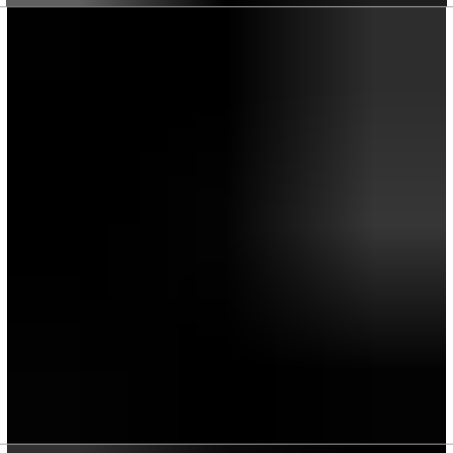
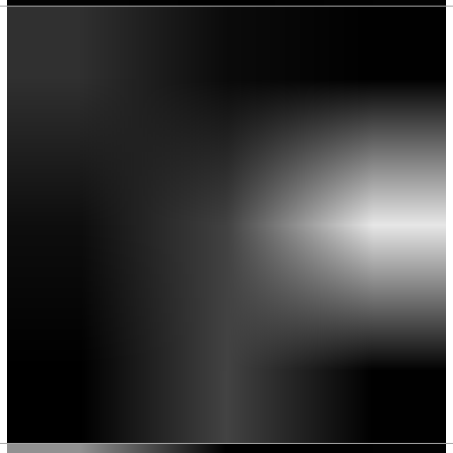
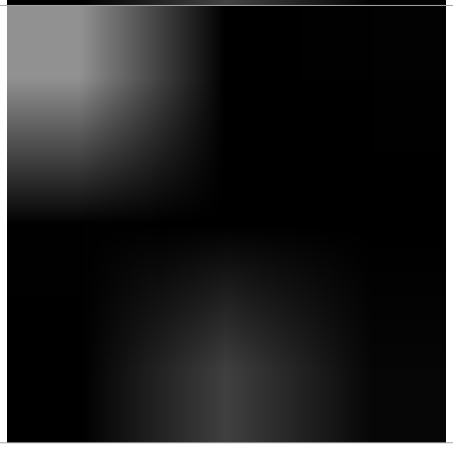
CONVOLUTION LAYER I

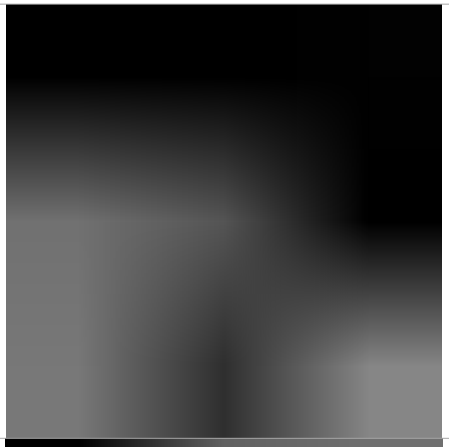
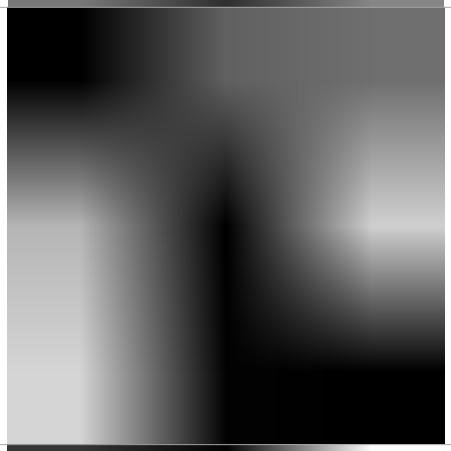
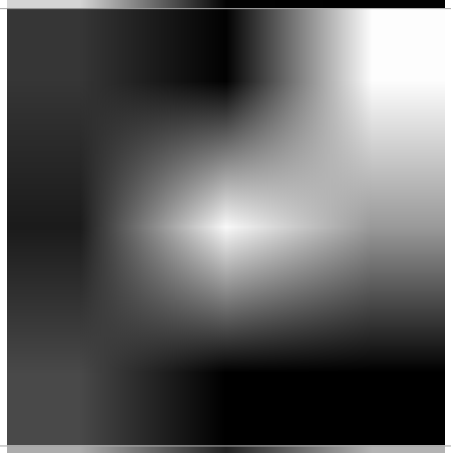
Dimension:
224 x 224

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

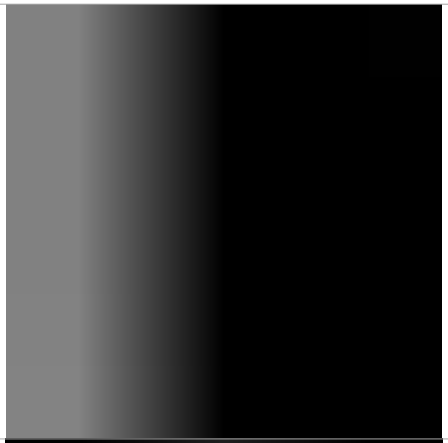
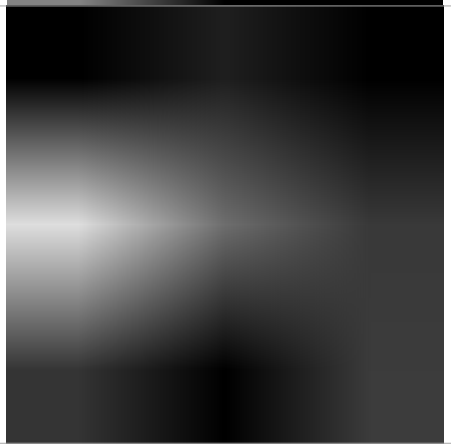
FILTERS	FEATURE MAPS
	
	
	
	

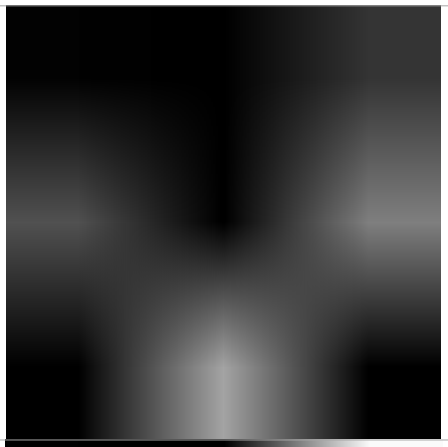
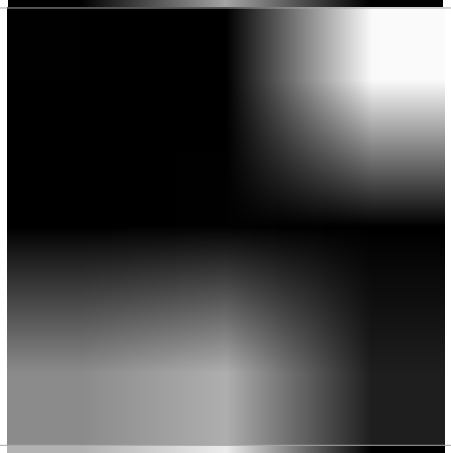
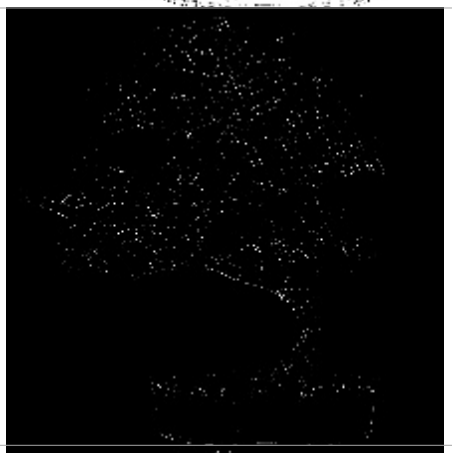
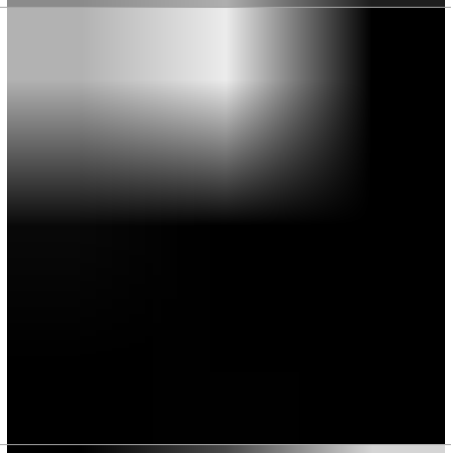
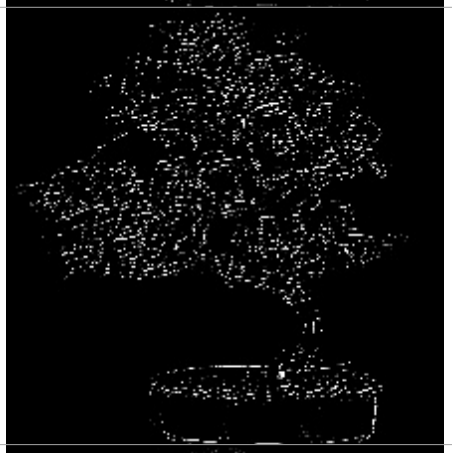
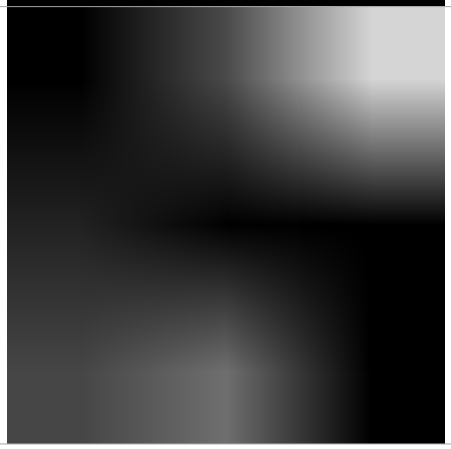
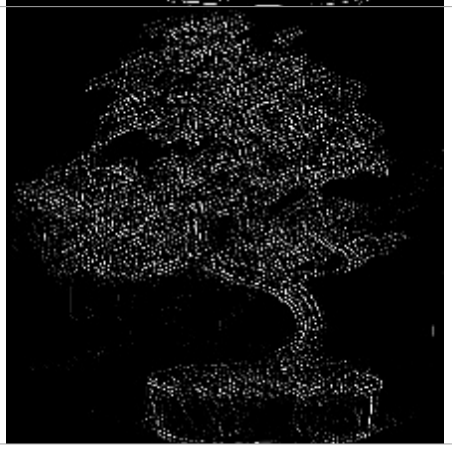
CONVOLUTION LAYER II

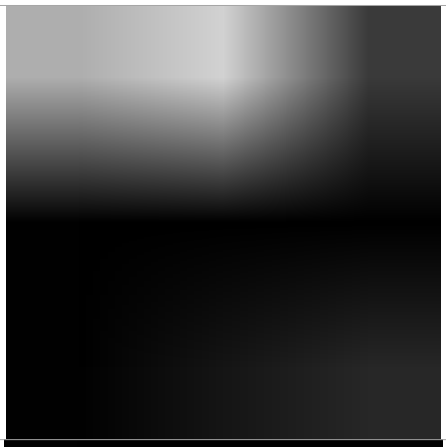
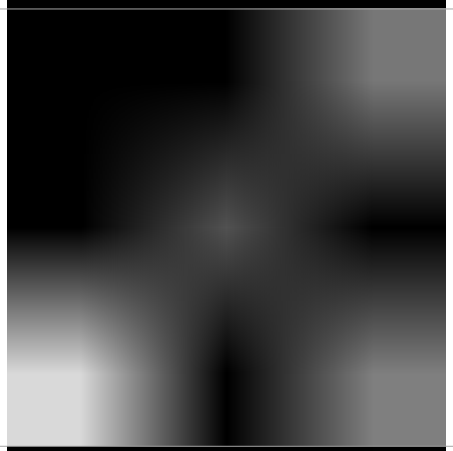
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

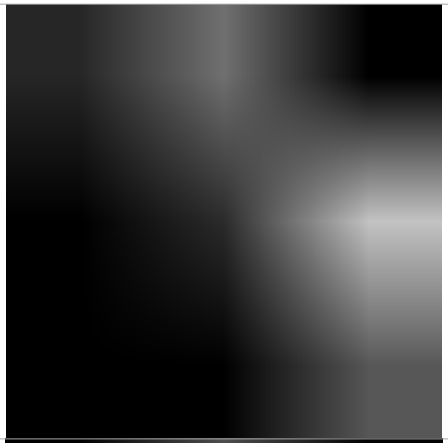
FILTERS	FEATURE MAPS
	
	
	
	

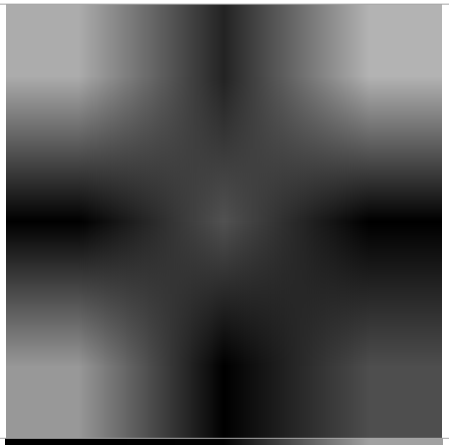
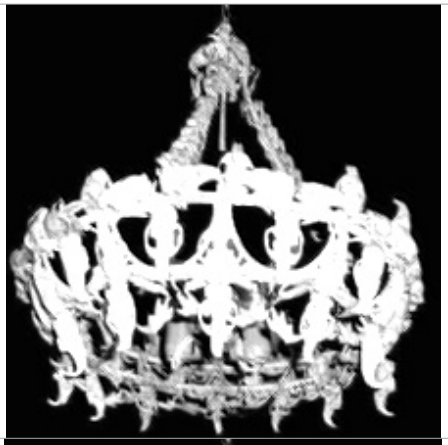
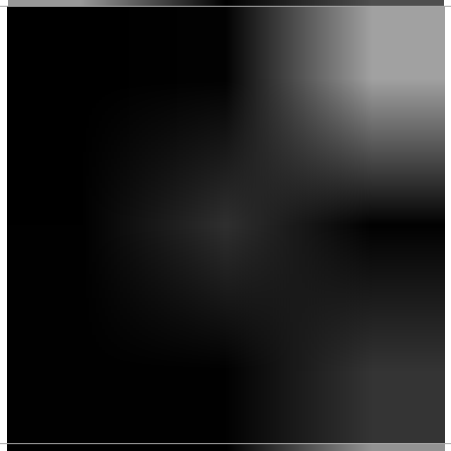
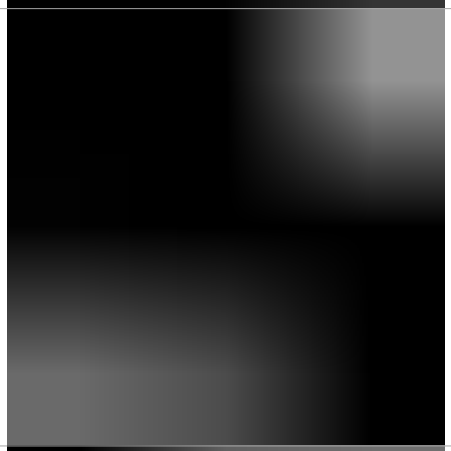
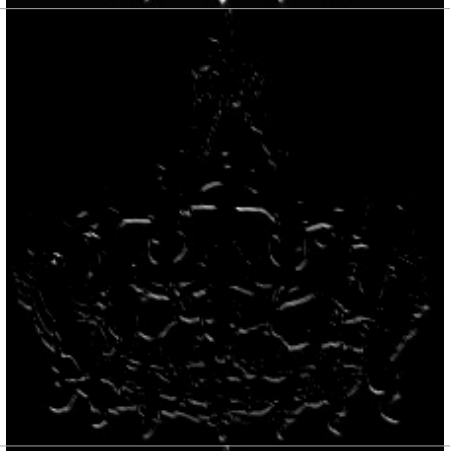
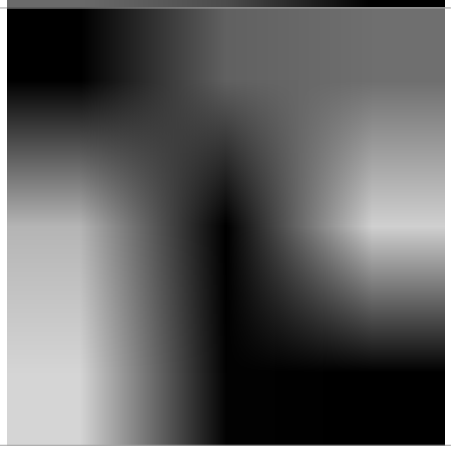
CONVOLUTION LAYER I

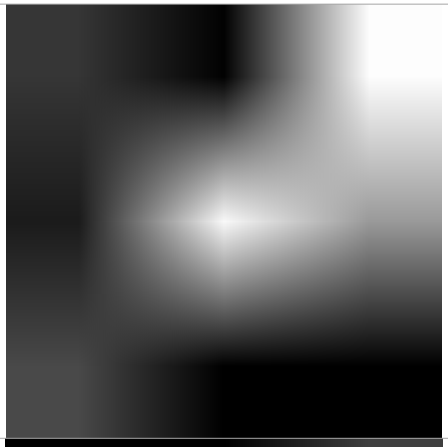
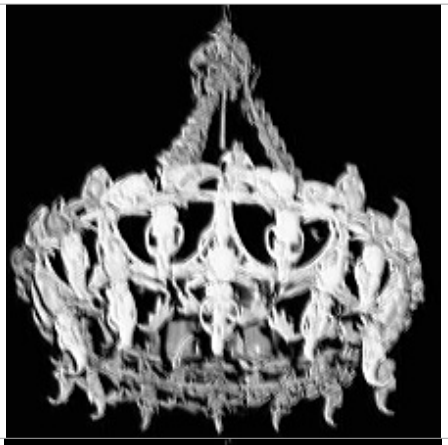
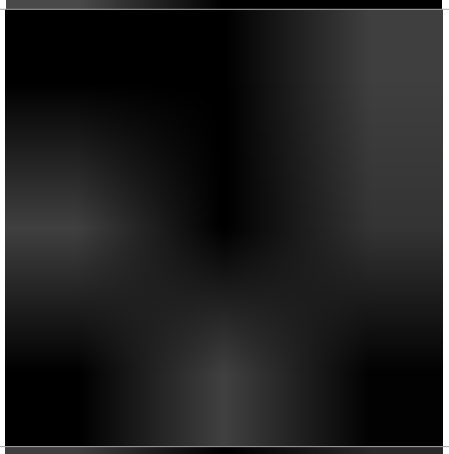
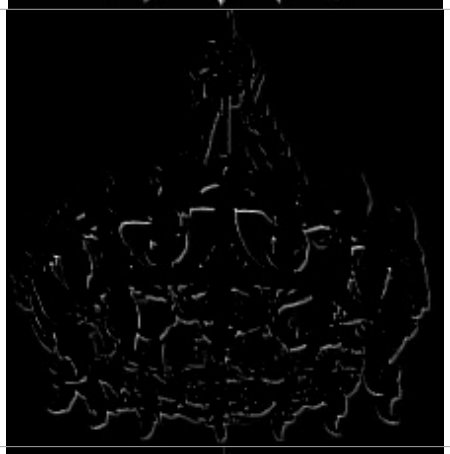
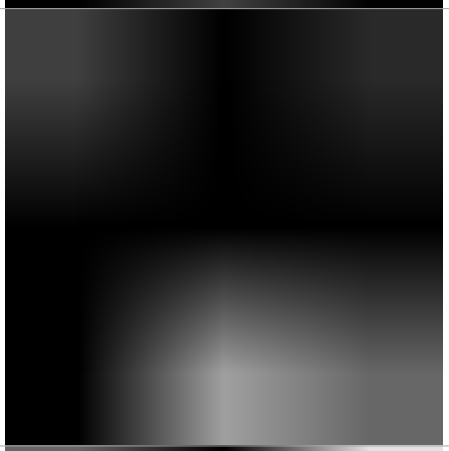
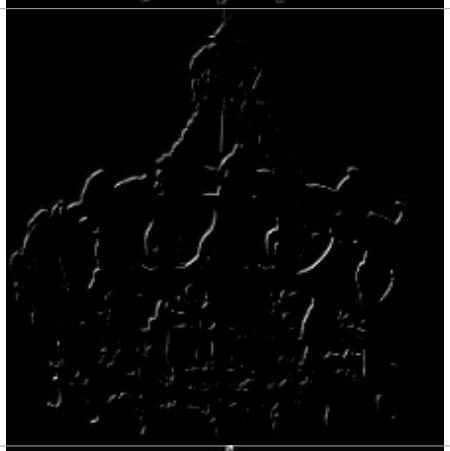
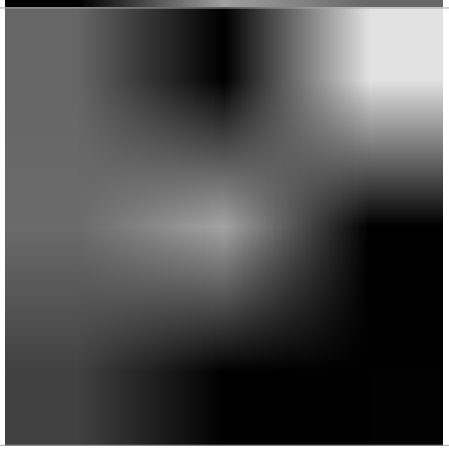
Dimension:
224 x 224

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

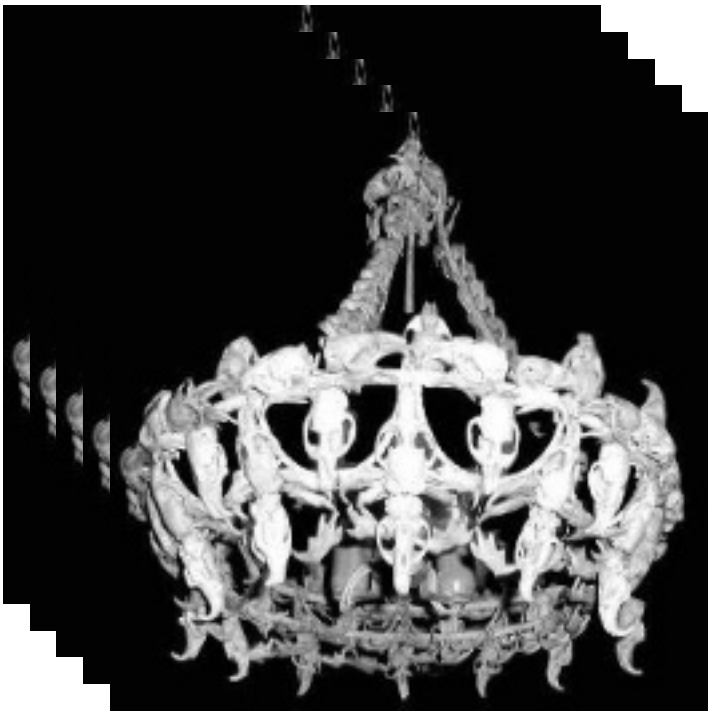
Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

FILTERS	FEATURE MAPS
	
	
	
	

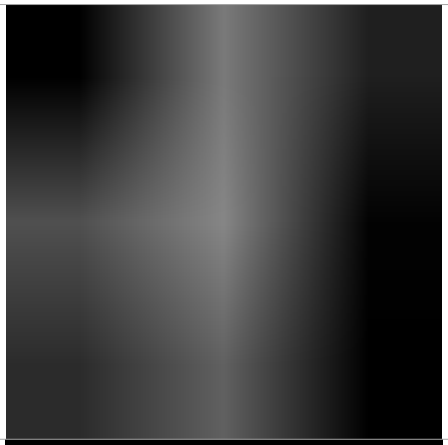
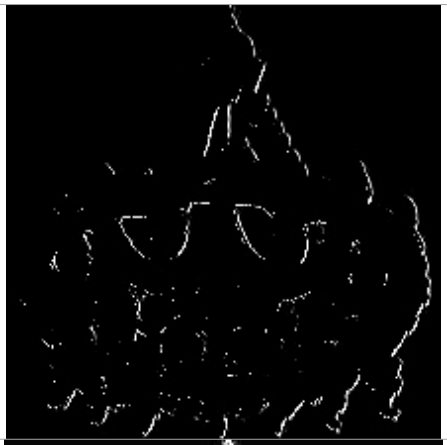
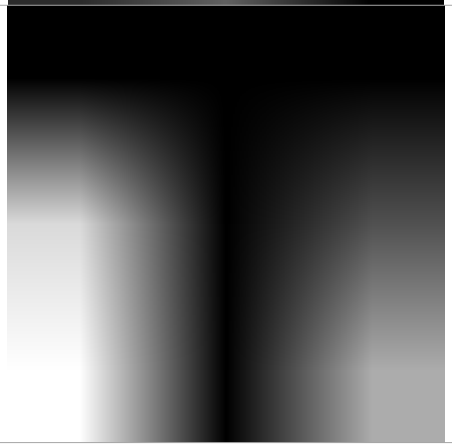
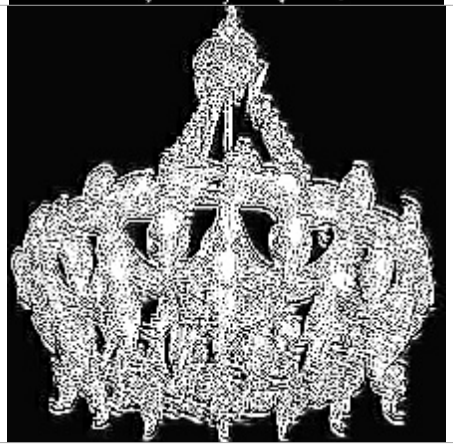
CONVOLUTION LAYER II

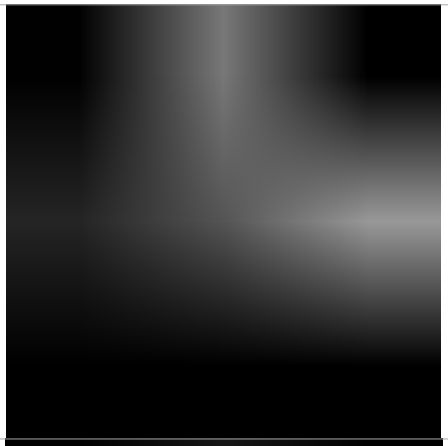
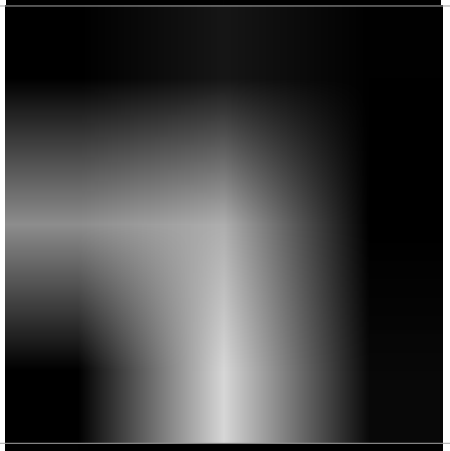
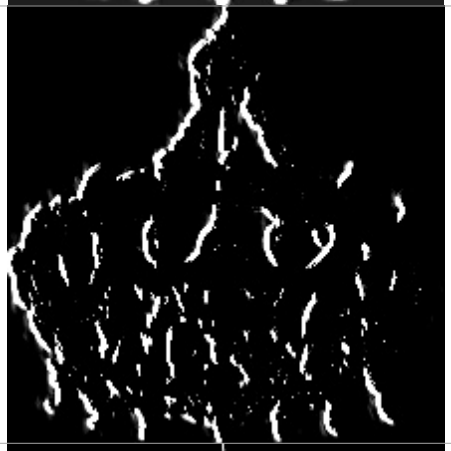
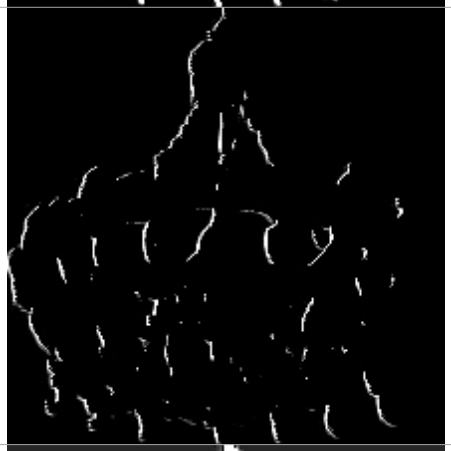
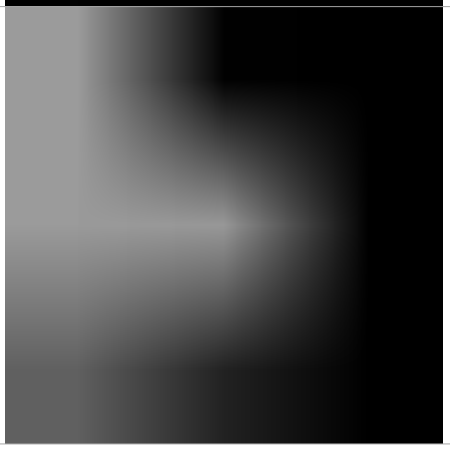
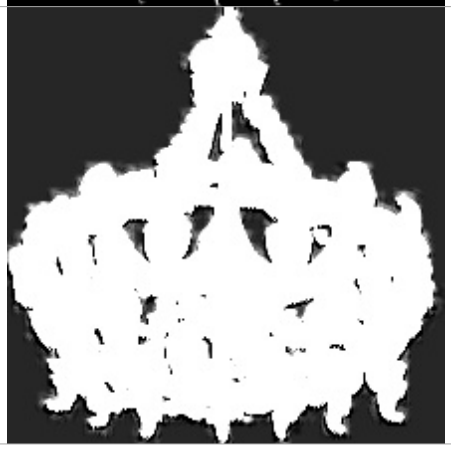


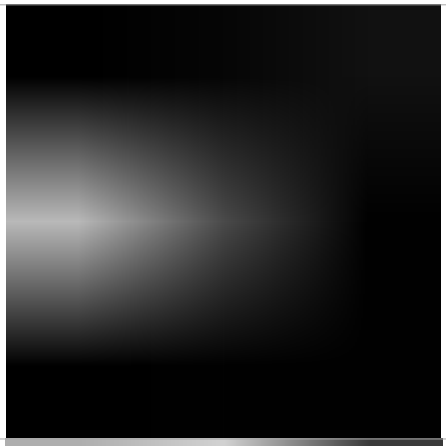
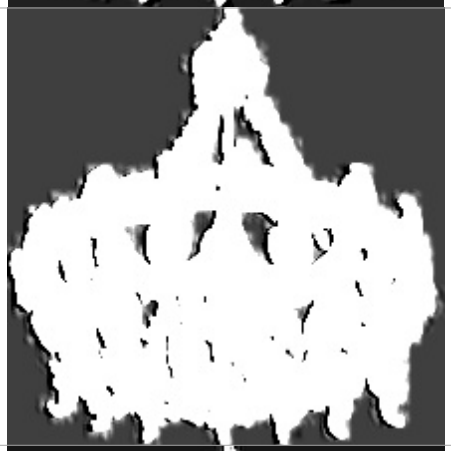
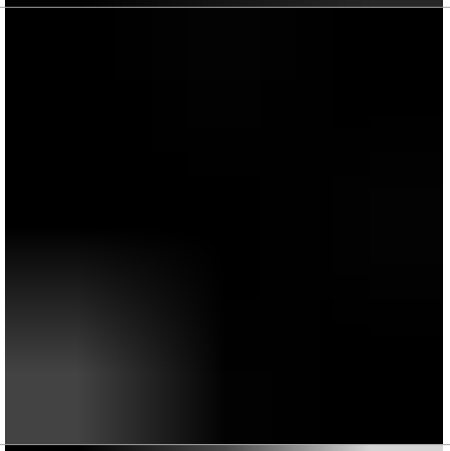
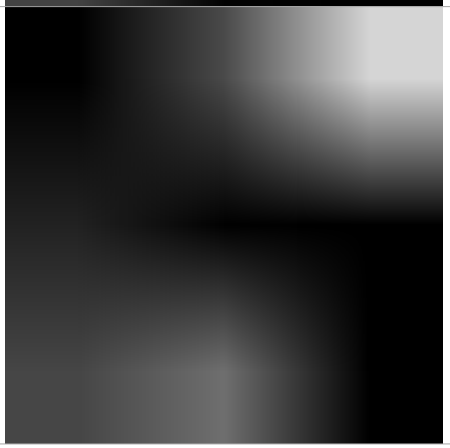
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

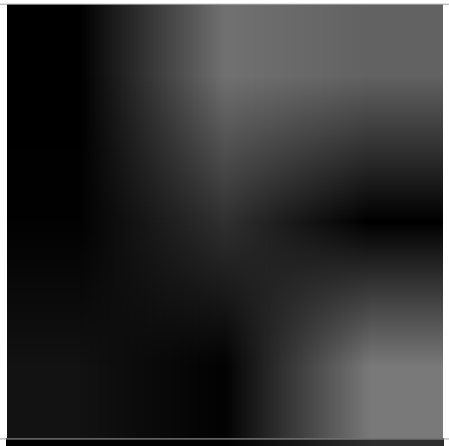
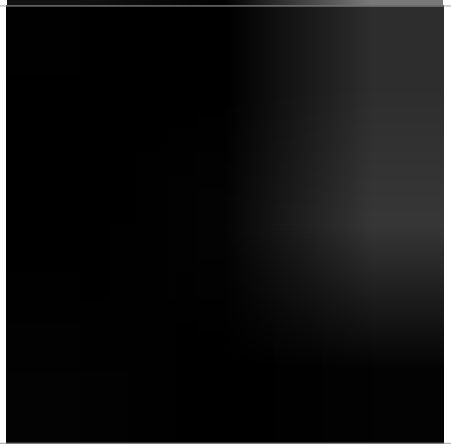
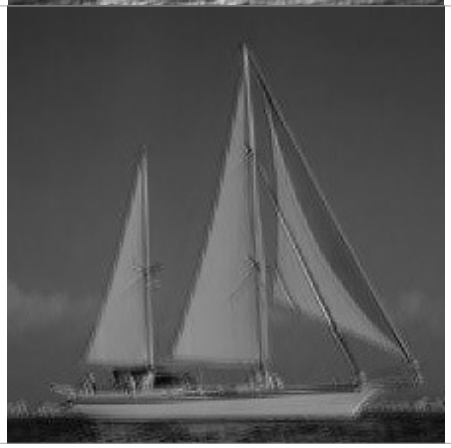
FILTERS	FEATURE MAPS
	
	
	
	

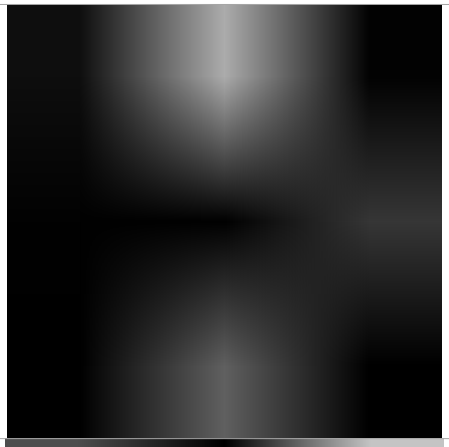
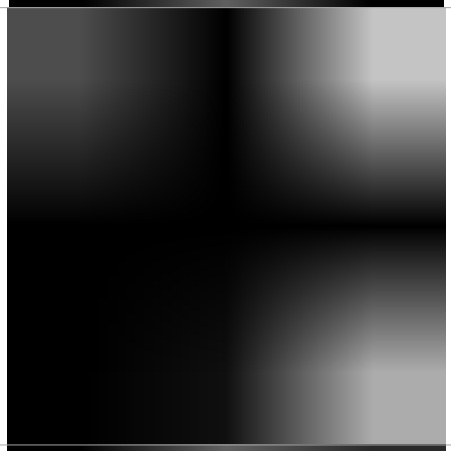
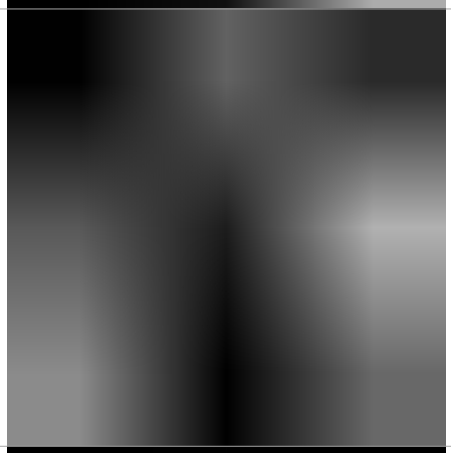
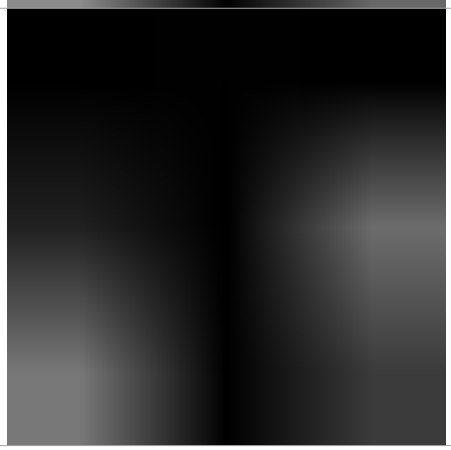
CONVOLUTION LAYER I

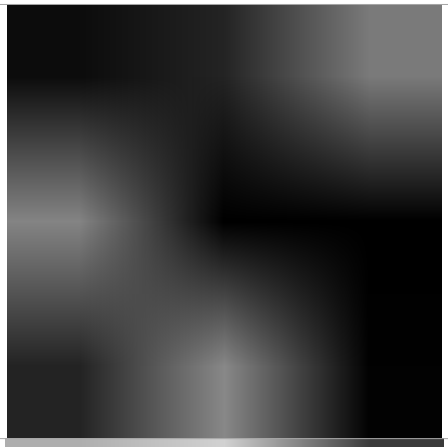
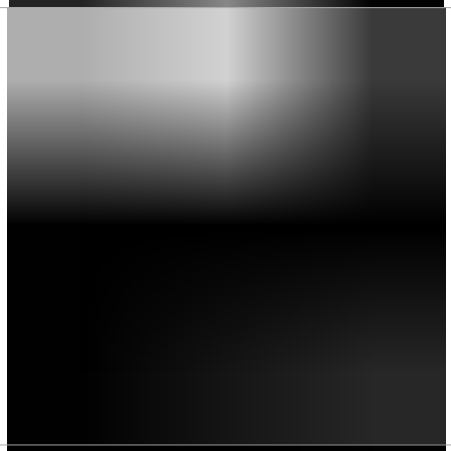
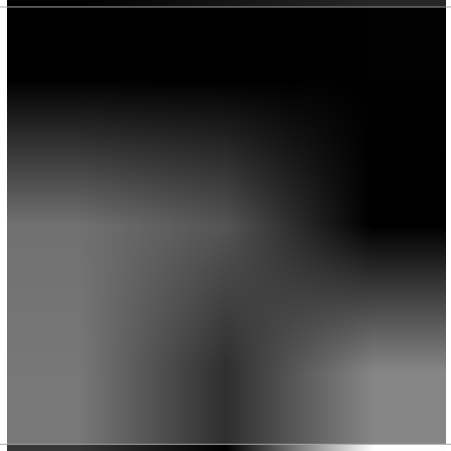
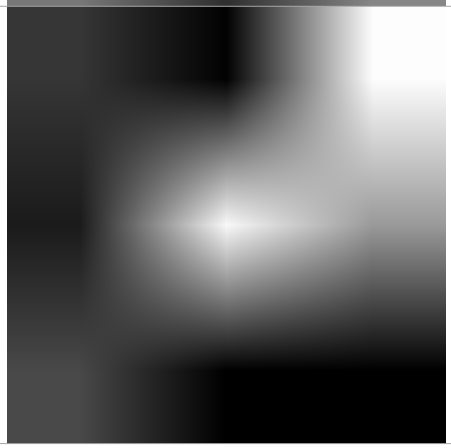
Dimension:
224 x 224

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

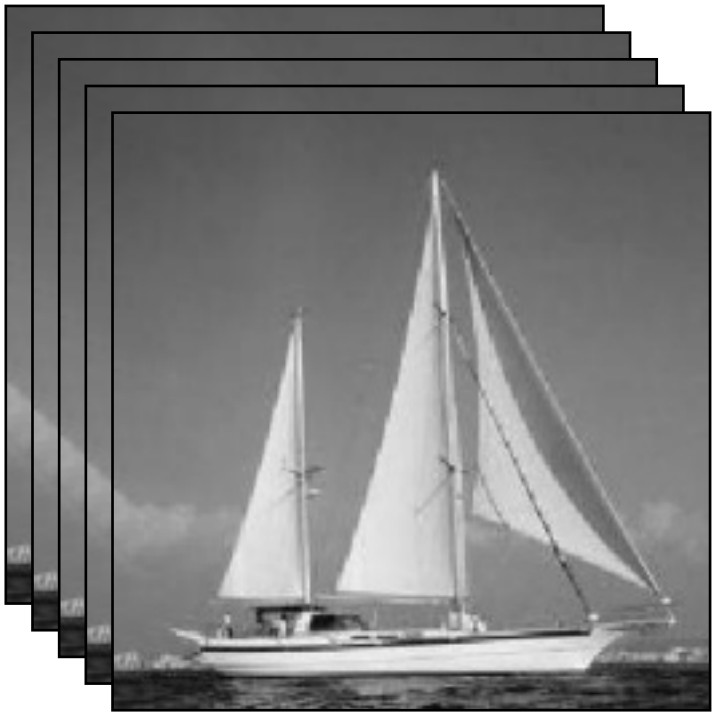
Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

FILTERS	FEATURE MAPS
	
	
	
	

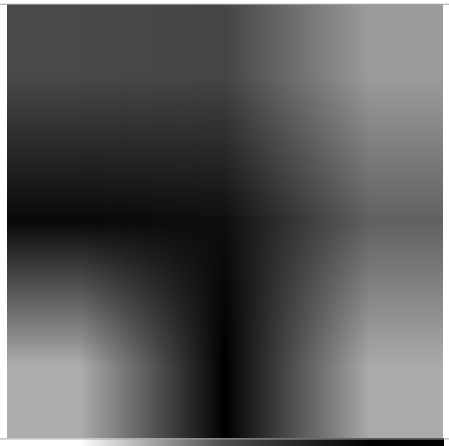
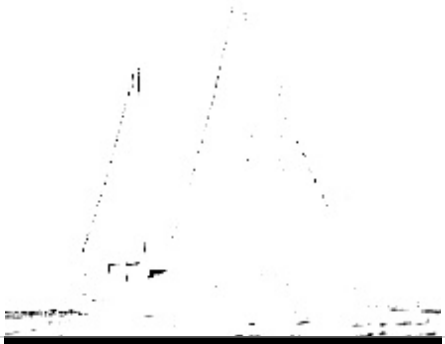
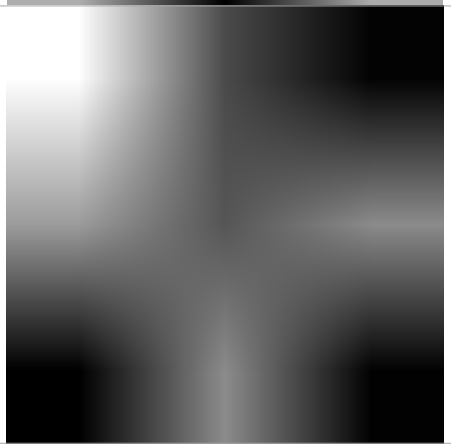
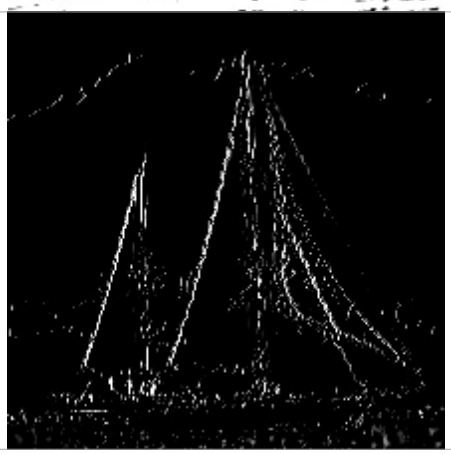
CONVOLUTION LAYER II

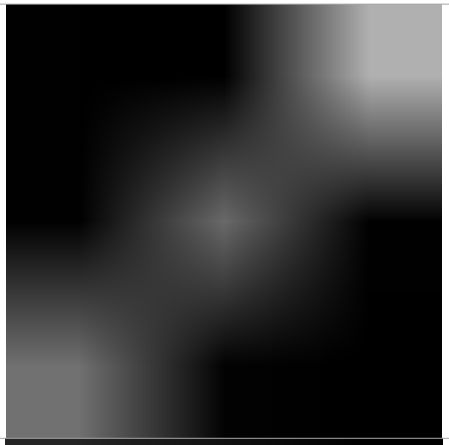
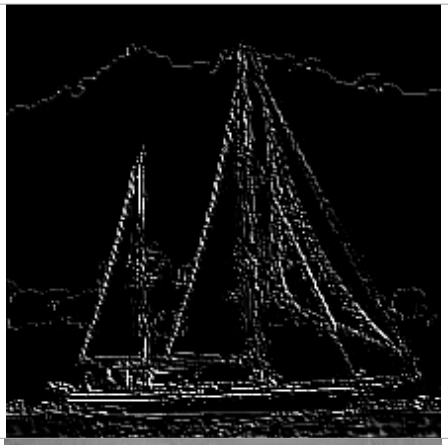
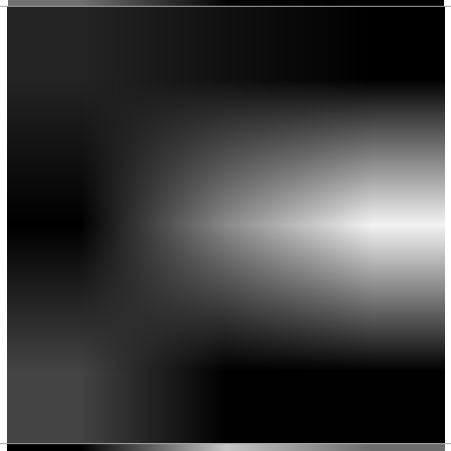
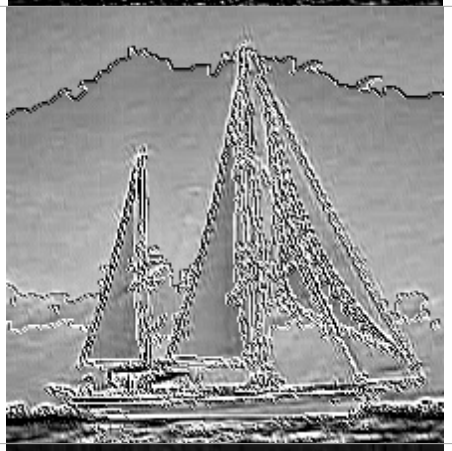
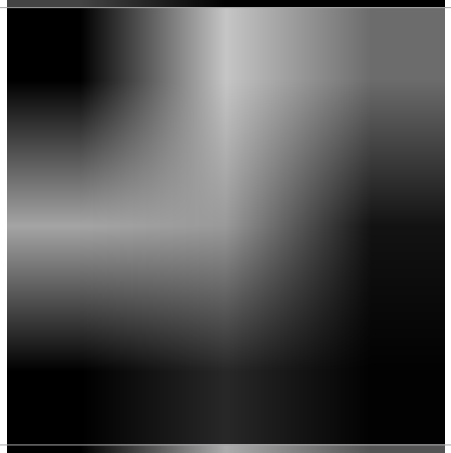
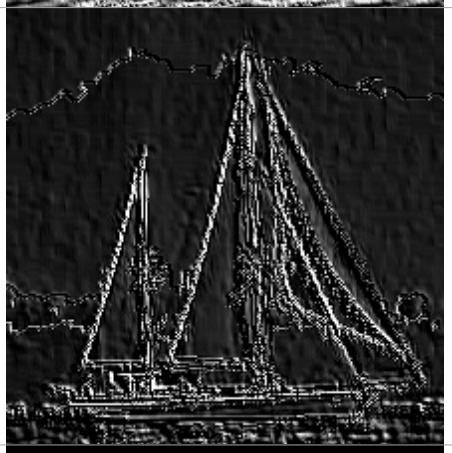
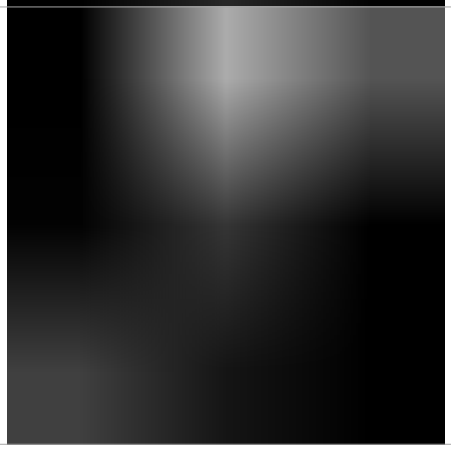
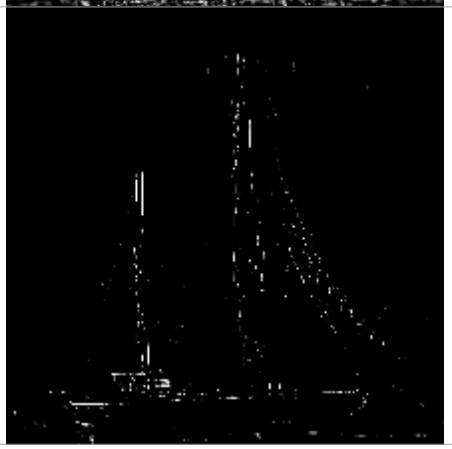


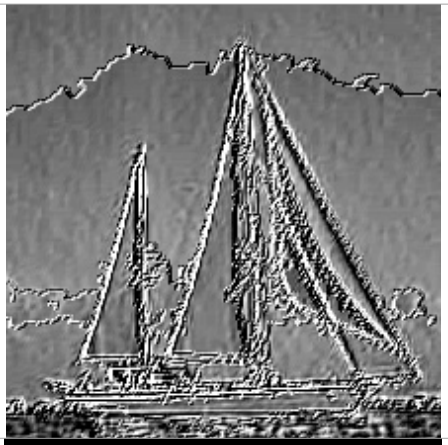
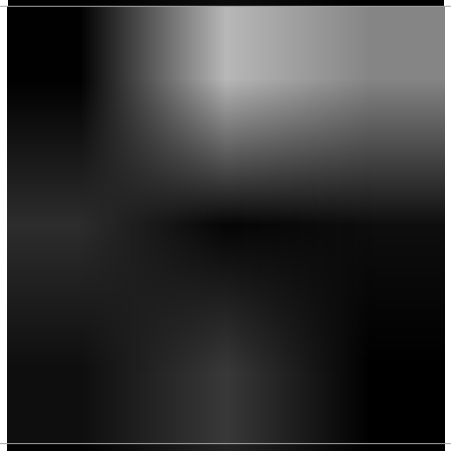
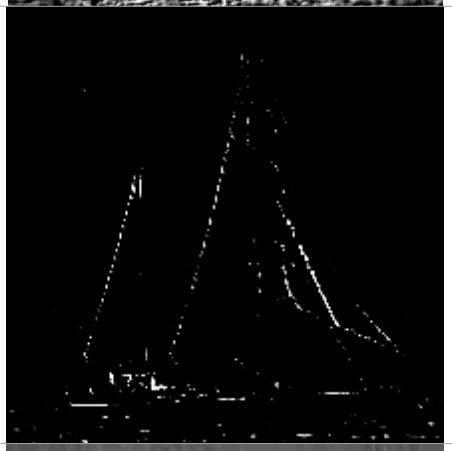
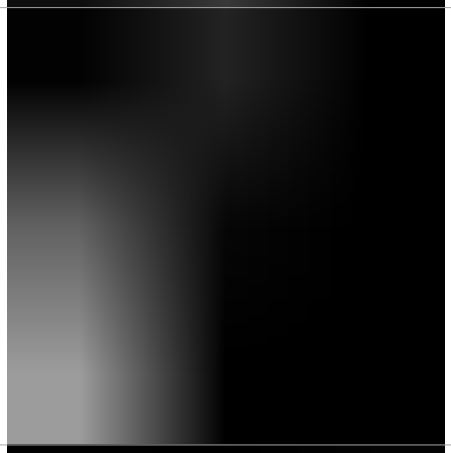
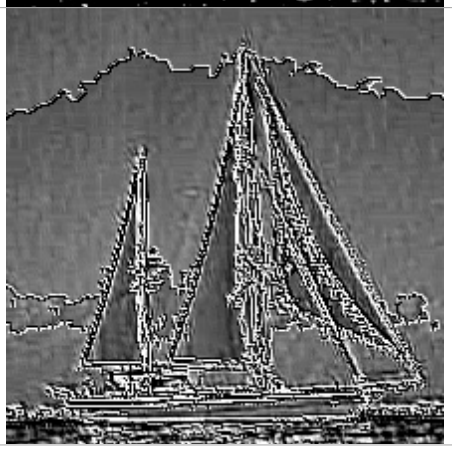
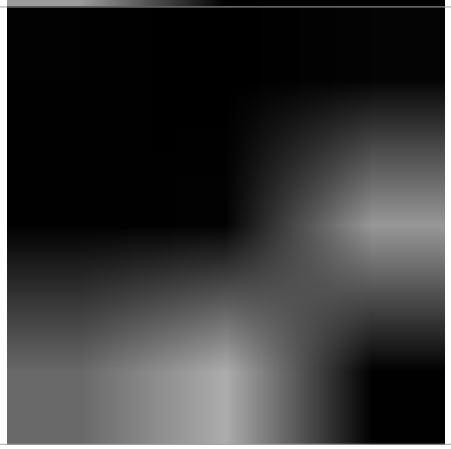
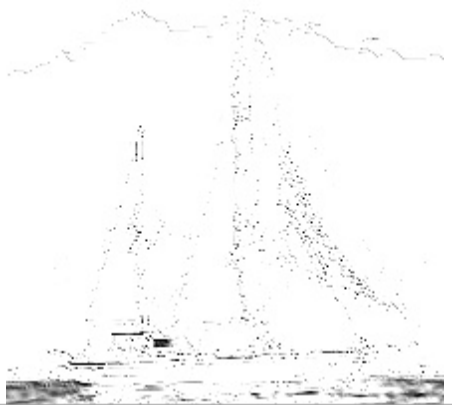
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

FILTERS	FEATURE MAPS
	
	
	
	

EXPECTED OUTPUT DIMENSION (THEORY)

Input Grayscale image $I(m,n)$ size: $m = 224$ and $n = 224$
Filter size $F(k,l)$: $k = 3$ and $l = 3$
Stride: $S = 1$
Padding: $P = 0$

LAYER 1 Output feature map dimension: ($K = 32$)

$$Om = \frac{m - k + 2P}{S} + 1 = \frac{224 - 3 + 2(0)}{1} + 1 = 222$$

$$On = \frac{n - l + 2P}{S} + 1 = \frac{224 - 3 + 2(0)}{1} + 1 = 222$$

Total dimension of layer 1 with 32 filters: 222 **x** 222 **x** 32

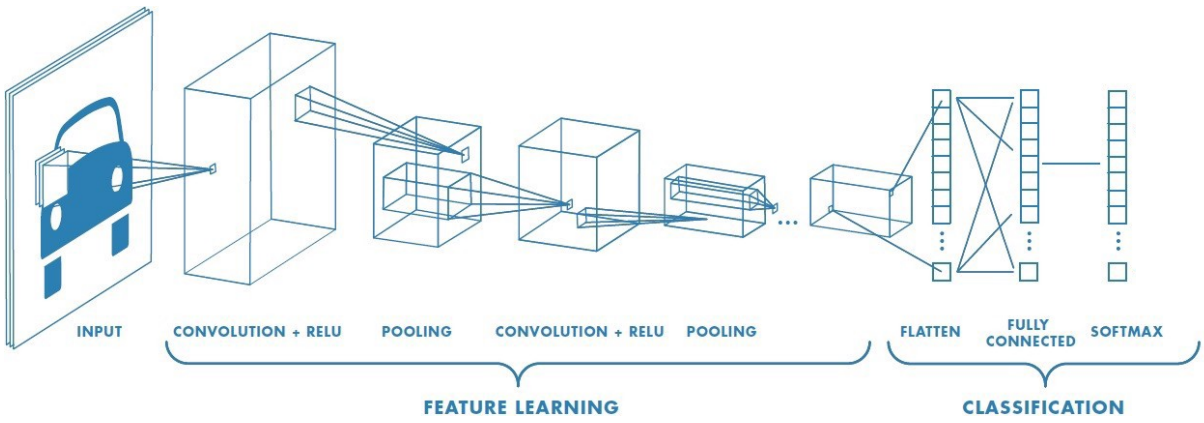
LAYER 2 Output feature map dimension: ($K = 64$)

$$Om = \frac{m - k + 2P}{S} + 1 = \frac{222 - 3 + 2(0)}{1} + 1 = 220$$

$$On = \frac{n - l + 2P}{S} + 1 = \frac{222 - 3 + 2(0)}{1} + 1 = 220$$

Total dimension of layer 2 with 64 filters: 220 **x** 220 **x** 64

FORMING CONVOLUTIONAL NEURAL NETWORK



Forming the complete convolutional neural network with two convolutional layers built above with max pooling and the result is fed to the fully connected deep neural network with one hidden layer with ReLU activation. Output uses softmax for classification task.

Architecture Details

Layers	Details	Dimension
Input Layer	224 x 224 x 3	150528
Convolutional Layer I	S = 1, P = 0, F = 3 x 3, K = 32	222 x 222 x 32
Max Pooling	S = 1, P = 0, F = 2 x 2	221 x 221 x 32
Convolutional Layer II	S = 1, P = 0, F = 3 x 3, K = 64	219 x 219 x 64
Max Pooling	S = 1, P = 0, F = 2 x 2	218 x 218 x 64
Flattening		3041536
Fully Connected Layer	ReLU Activation	128
Output Layer	Softmax Activation	3

FORWARD PROPAGATION

We have developed each class for each layers with necessary functions back propagation and forward propagation.

```
class ConvolutionLayer:

    def forward_prop(self, image, K, size, n):
        output = []
        ReLUout = []
        filters = []
        for i in range(K):
            filtr = kaiming_initialization.initialize(kaiming_initialization, n, size)
            filters.append(filtr)
            res = convolution.convolve(convolution, inpt=image, filtr=filtr, stride = 1, padding = 0)
            output.append(res)
            ReLUout.append(ReLUActivation.forward_prop(ReLUActivation, res))

        self.output = np.array(output)
        self.ReLUout = np.array(ReLUout)
        self.filters = np.array(filters)

    def inputlayerConV(self, image, K, size, n):
        output = []
        ReLUout = []
        filters = []
        for i in range(K):
            filtr = kaiming_initialization.initialize(kaiming_initialization, n, size)
            filters.append(filtr)
            res = convolution.inptconvolve(convolution, inpt=image, filtr=filtr, stride = 1, padding = 0)
            output.append(res)
            ReLUout.append(ReLUActivation.forward_prop(ReLUActivation, res))

        self.output = np.array(output)
        self.ReLUout = np.array(ReLUout)
        self.filters = np.array(filters)
```

```
class MaxPool:

    def forward_prop(self, featureMaps, size, stride):
        m, n = featureMaps[0].shape
        fm, fn = size
        result = []
        poolout = []
        for featureMap in featureMaps:
            for i in range(0, m, stride):
                for j in range(0, n, stride):
                    if featureMap[i:i+fm, j:j+fn].shape == size:
                        result.append(max(featureMap[i:i+fm, j:j+fn].flatten()))

            Outm, Outn = int(((m-fm)/stride)+1), int(((n-fn)/stride)+1)
            poolout.append(np.array(result).reshape((Outm, Outn)))
            result = []

        self.maxpoolout = np.array(poolout)
```

```
class SoftmaxActivation:

    def forward_prop(self, x):
        x = np.array(x, dtype=np.float64)
        exp = np.exp(x)
        return exp/np.sum(exp)
```

```
class ReLUActivation:

    def forward_prop(self, x):

        out = []
        if len(x.shape) == 1:
            for i in x:
                if i>=0:
                    out.append(i)
                else:
                    out.append(0)
        else:
            for i in x:
                for j in i:
                    if j>=0:
                        out.append(j)
                    else:
                        out.append(0)

        return np.array(out).reshape(x.shape)
```

```
class Deepneuralnetwork:

    def forward_prop(self, xn, wh):
        self.xn = xn
        self.wh = wh
        h1 = np.dot(wh.T, xn)
        self.output = ReLUActivation.forward_prop(ReLUActivation, h1)

    def Outforward_prop(self, xn, wo):
        self.Oxn = xn
        self.wo = wo
        out = np.dot(wo.T, xn)
        self.foutput = SoftmaxActivation.forward_prop(SoftmaxActivation, out)

    def flattenBackprop(self, dy):
        if dy.shape[0] == self.xn.shape[0]:
            dy = dy.T
        dw = dy.dot(self.xn)
        dx = np.dot(dy.T, self.wh.T)
        self.wh -= 0.001 * dw.T
        return dx

    def backprop(self, dy):
        if dy.shape[0] == self.output.shape[0]:
            dy = dy.T
        dw = dy.dot(self.output)
        dx = np.dot(dy.T, self.wh.T)
        self.wh -= 0.001 * dw.T
        return dx

    def Outbackprop(self, dy):
        if dy.shape[0] == self.foutput.shape[0]:
            dy = dy.T
        dw = dy.dot(self.foutput)
        dx = np.dot(dy.T, self.wo.T)
        self.wo -= 0.001 * dw.T
```

```
class Flattening:

    def flatten(self, x):
        self.flatten_out = x.flatten()
```

```
class loss:

    def cross_entropy(self, inputs, labels):

        out_num = labels.shape[0]
        p = np.sum(labels.reshape(1,out_num)*inputs)
        loss = -np.log(p)
        return loss
```

Single forward propagation is shown below.

```
layer1 = ConvolutionLayer()
layer2 = MaxPool()
layer3 = ConvolutionLayer()
layer4 = MaxPool()
layer5 = Flattening()
layer6 = Deepneuralnetwork()
layer7 = Deepneuralnetwork()
kaiming = kaiming_initialization()
losses = loss()
```

```
image = color_images[0]

layer1.inputlayerConV(image, 32, (3, 3, 3), 27)
layer2.forward_prop(layer1.ReLUout, (2, 2), 1)
layer3.forward_prop(layer2.maxpoolout, 64, (32, 3, 3), 27)
layer4.forward_prop(layer3.ReLUout, (2, 2), 1)
layer5.flatten(layer4.maxpoolout)

wh = kaiming.initialize(layer5.flatten_out.shape[0]*128, (layer5.flatten_out.shape[0], 128))
wo = kaiming.initialize(128, (128, 3))
layer6.forward_prop(layer5.flatten_out, wh)
layer7.Outforward_prop(layer6.output, wo)
```

```
acc = 0
total_acc = 0
labell = np.argmax(layer7.foutput)
entropyloss = losses.cross_entropy(layer7.foutput, labels)
if np.argmax(layer7.foutput) == np.argmax(labels):
    acc += 1
    total_acc += 1

dy = labels
```

Now we have to put this in loop for each training example and measure the loss and update the weights.

BACKWARD PROPAGATION

To implement back propagation, we need to derive the gradients of each layer. The gradients of each layer is given.

SOFTMAX LAYER

Let p_c be the predicted probability of the correct class and Out_s be the output of softmax layer.

$$L = -ln(p_c)$$

$$\frac{\partial L}{\partial out_s(i)} = \begin{cases} 0 & \text{if } i \neq c \\ -\frac{1}{p_i} & \text{if } i = c \end{cases}$$

Where the $Out_s(c)$ be written as,

$$out_s(c) = \frac{e^{t_c}}{\sum_i e^{t_i}} = \frac{e^{t_c}}{S}$$

t_i is the total class for class i.

$$out_s(c) = e^{t_c} S^{-1}$$

$$\begin{aligned} \frac{\partial out_s(c)}{\partial t_k} &= -e^{t_c} S^{-2} (\frac{\partial S}{\partial t_k}) \\ &= -e^{t_c} S^{-2} (e^{t_k}) \\ &= \boxed{\frac{-e^{t_c} e^{t_k}}{S^2}} \end{aligned}$$

Note the gradient is for t_k. Similarly with respect to t_c,

$$\begin{aligned}\frac{\partial out_s(c)}{\partial t_c} &= \frac{Se^{t_c} - e^{t_c}\frac{\partial S}{\partial t_c}}{S^2} \\ &= \frac{Se^{t_c} - e^{t_c}e^{t_c}}{S^2} \\ &= \boxed{\frac{e^{t_c}(S - e^{t_c})}{S^2}}\end{aligned}$$

$$\boxed{\frac{\partial out_s(k)}{\partial t} = \begin{cases} \frac{-e^{t_c}e^{t_k}}{S^2} & \text{if } k \neq c \\ \frac{e^{t_c}(S-e^{t_c})}{S^2} & \text{if } k = c \end{cases}}$$

LOSS GRADIENTS

It is very much straight forward. With the weight w and bias b, the loss is calculated by,

$$t = w * input + b$$

$$\frac{\partial t}{\partial w} = input$$

$$\frac{\partial t}{\partial b} = 1$$

$$\frac{\partial t}{\partial input} = w$$

$$\boxed{\begin{aligned}\frac{\partial L}{\partial w} &= \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial w} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial b} \\ \frac{\partial L}{\partial input} &= \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial input}\end{aligned}}$$

MAX POOLING

For Max pooling, $\frac{\partial L}{\partial inp} = 0 \forall$ non maximum pixel. Which means that,

$\frac{\partial o/p}{\partial i/p} = 1$ which is $\frac{\partial L}{\partial i/p} = \frac{\partial L}{\partial o/p}$

CONVOLUTIONAL LAYER

$$\begin{aligned} \text{out(i, j)} &= \text{convolve(image, filter)} \\ &= \sum_{x=0}^{3} \sum_{y=0}^{3} \text{image}(i+x, j+y) * \text{filter}(x, y) \end{aligned}$$

$$\frac{\partial \text{out}(i,j)}{\partial \text{filter}(x,y)} = \text{image}(i+x, j+y)$$

$$\boxed{\frac{\partial L}{\partial \text{filter}(x,y)} = \sum_i \sum_j \frac{\partial L}{\partial \text{out}(i,j)} * \frac{\partial \text{out}(i,j)}{\partial \text{filter}(x,y)}}$$

The weights are updated by stochastic gradient descent algorithm. The filters are initialised by kaiming and fully-connected layers are initialised with weights randomly sampled from a zero mean normal distribution with a standard deviation inversely proportional to the square root of the number of units.

RESULTS

The stopping criteria considered is by the number of epoch. No of epoch chosen here is 25. Each color image has 3*3*3 filters and only the feature[0] is plotted in result.

ACCURACY & LOSS

Data	Accuracy	Loss
Training	1.00	2.7E-06
Validation	0.75	53.7344
Test	0.74	53.2120

CONFUSION MATRIX

ACTUAL CLASS				
PREDICTED CLASS		Ketch	Chandelier	Bonsai
	Ketch	12	3	5
	Chandelier	3	14	3
	Bonsai	1	0	19

INPUTS FOR FEATURE MAPS

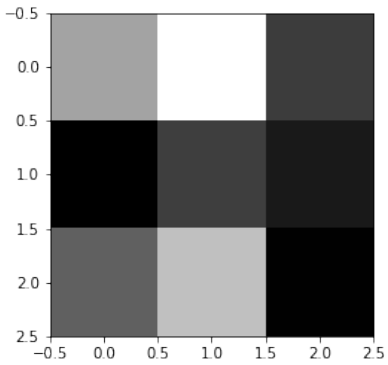
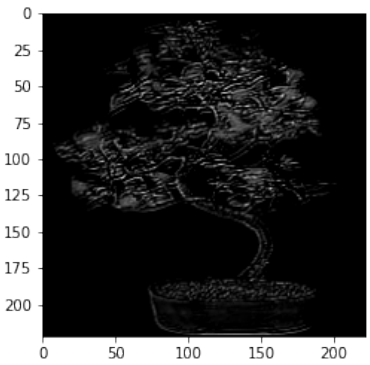
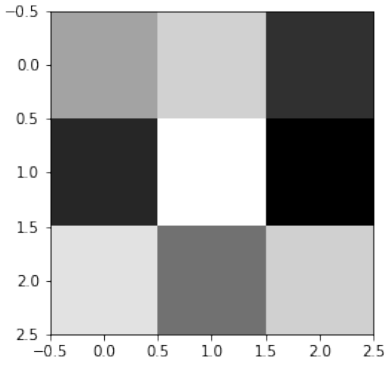
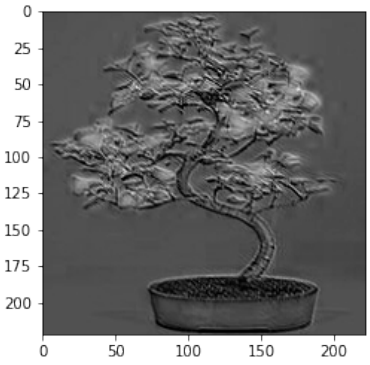
Bonsai	Chandelier	Ketch

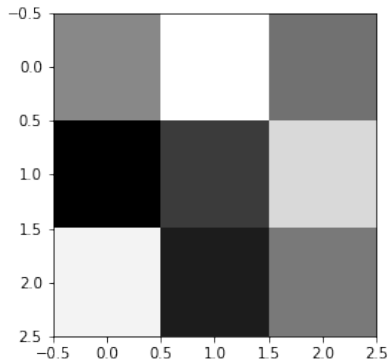
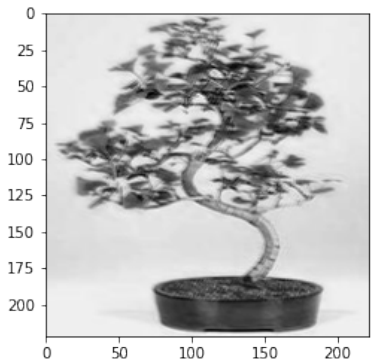
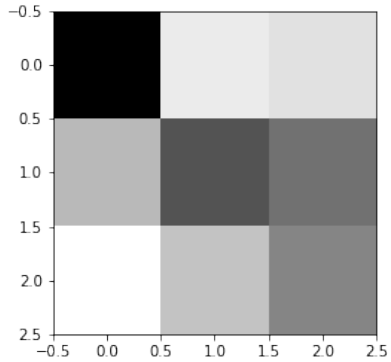
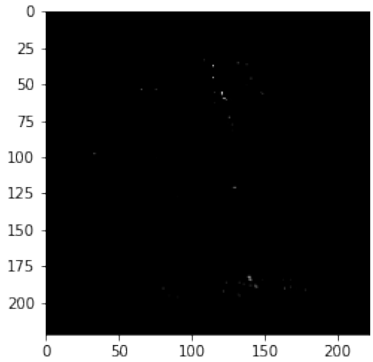
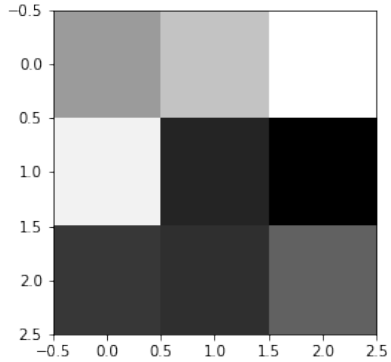
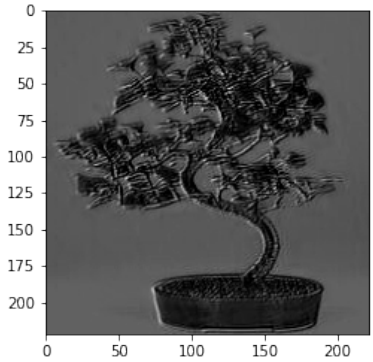
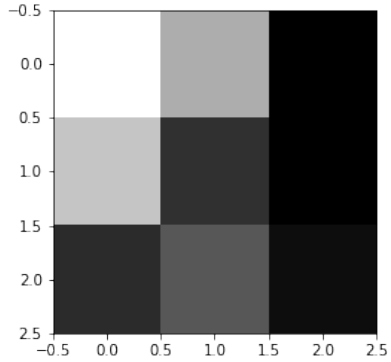
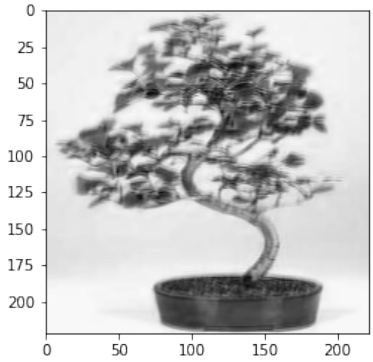
CONVOLUTION LAYER I

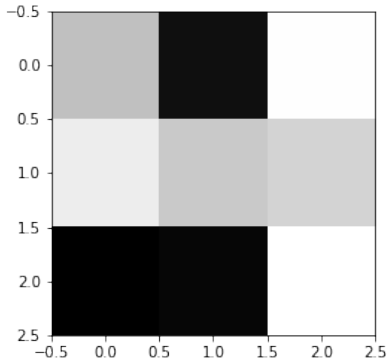
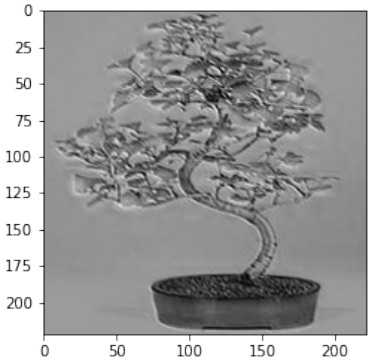
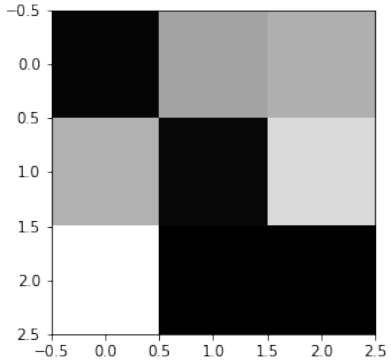
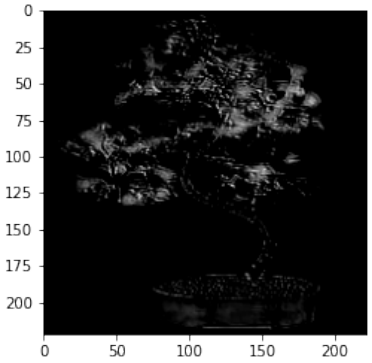
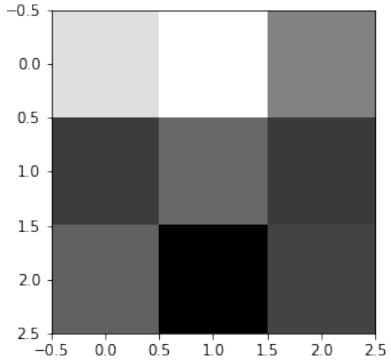
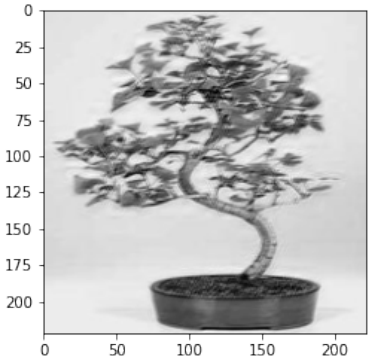
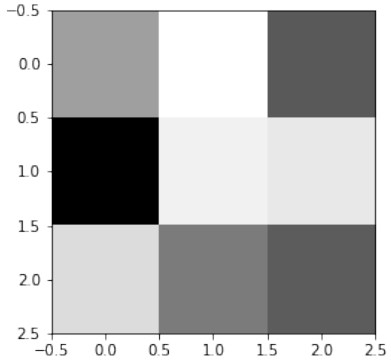
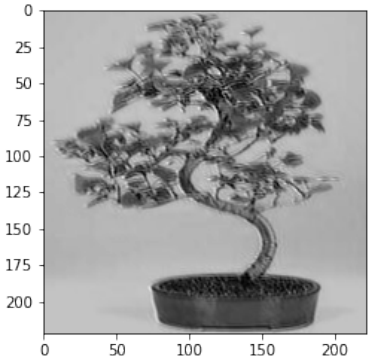
Dimension:
224 x 224 x 3

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

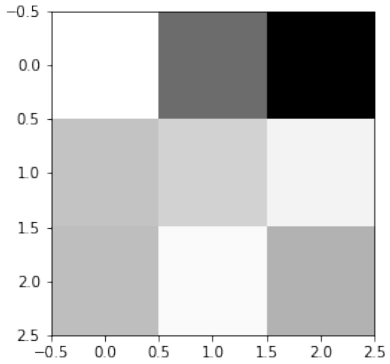
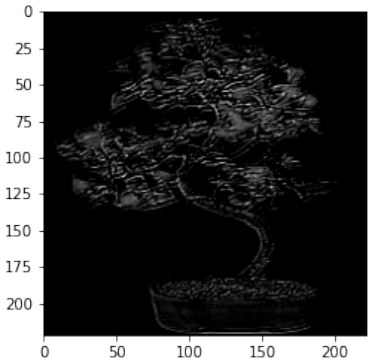
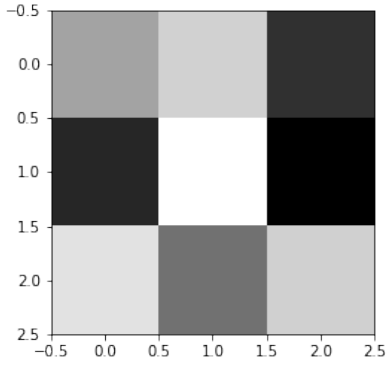
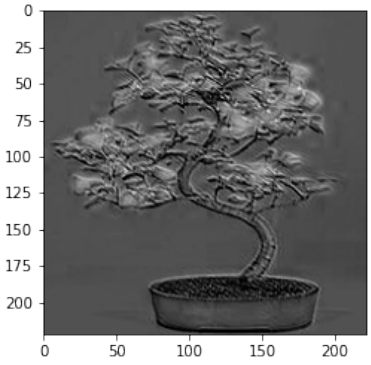
FILTERS	FEATURE MAPS
	
	
	
	

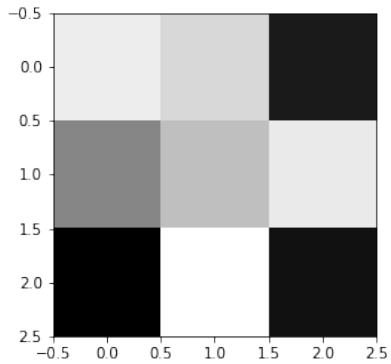
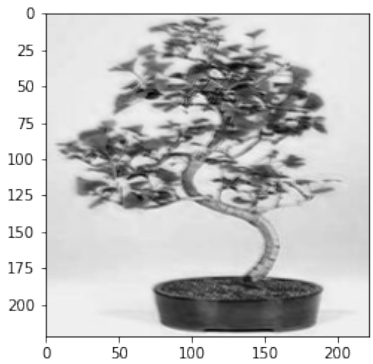
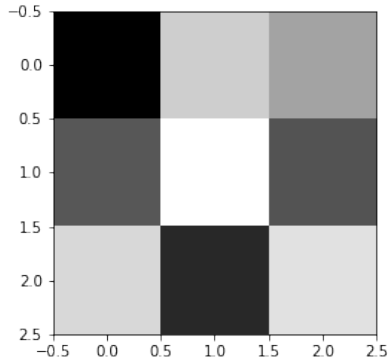
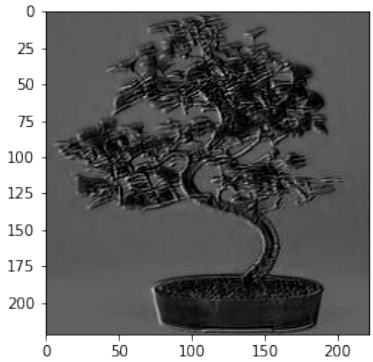
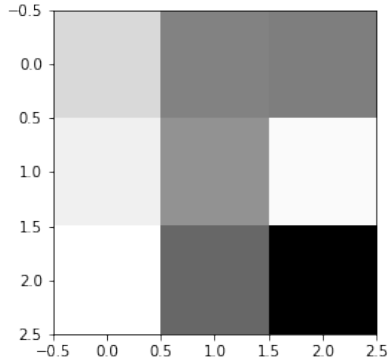
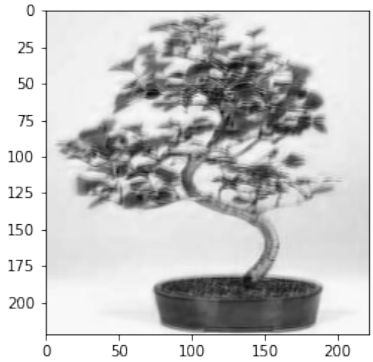
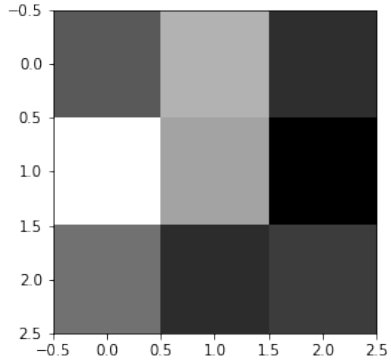
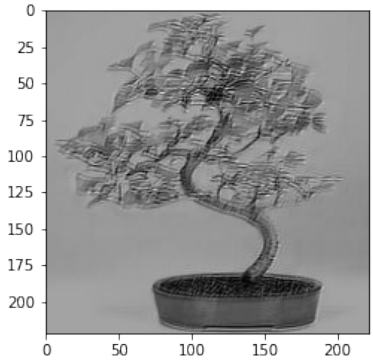
CONVOLUTION LAYER II

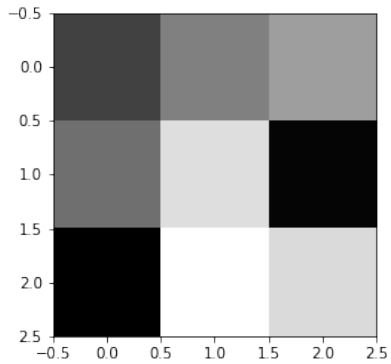
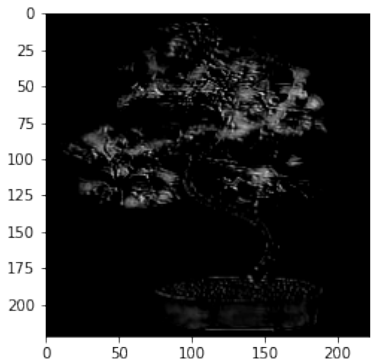
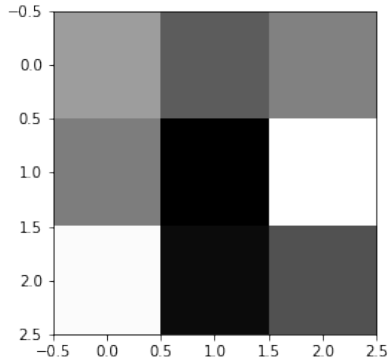
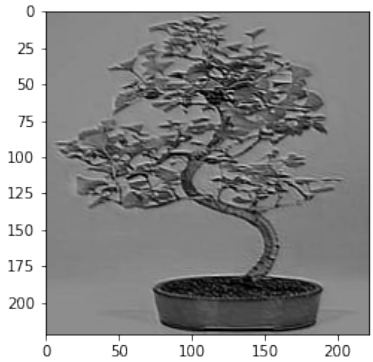
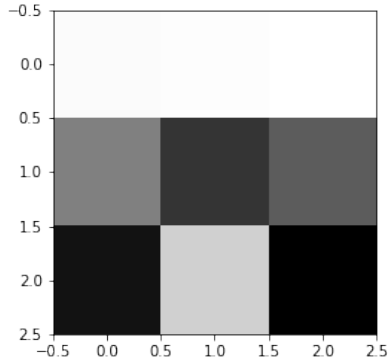
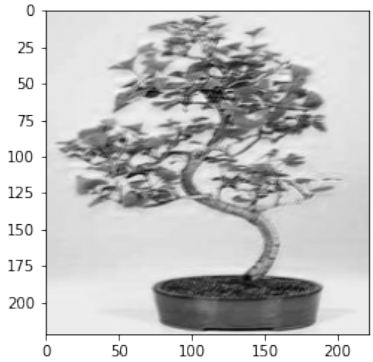
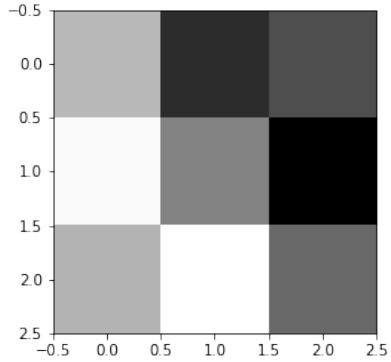
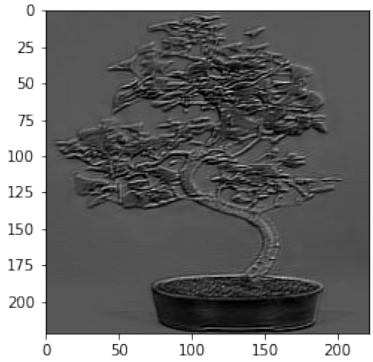
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

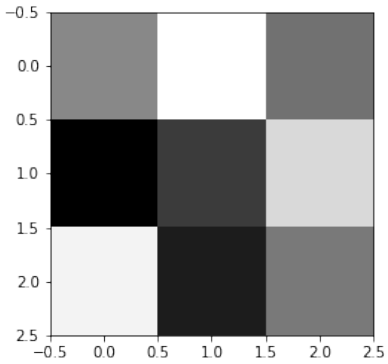
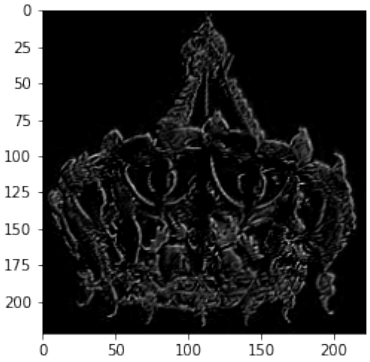
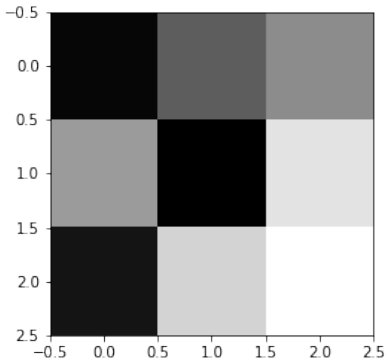
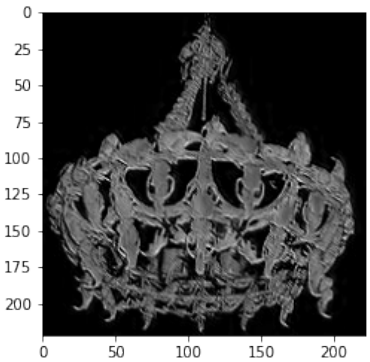
FILTERS	FEATURE MAPS
	
	
	
	

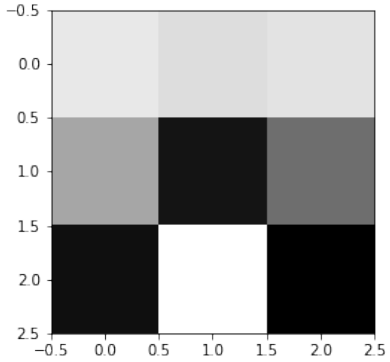
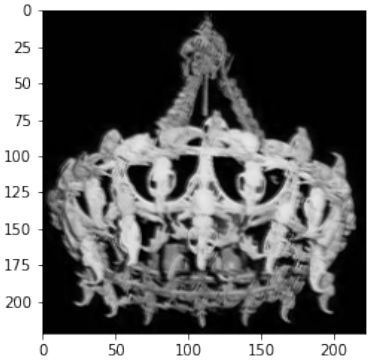
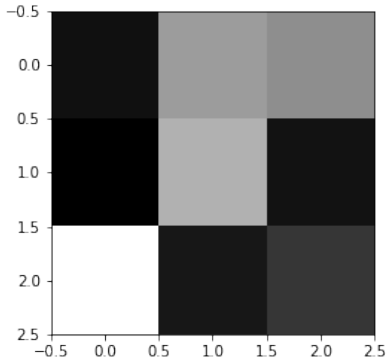
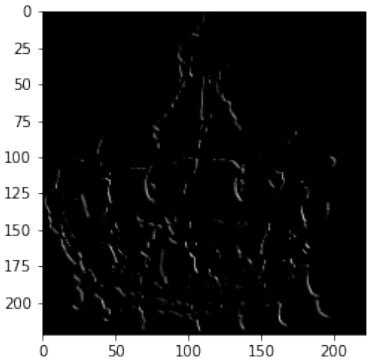
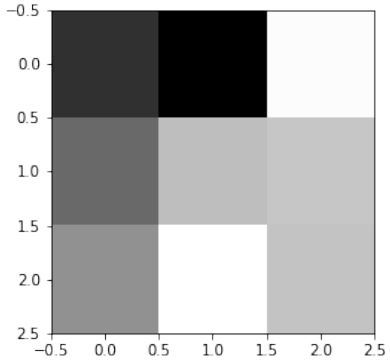
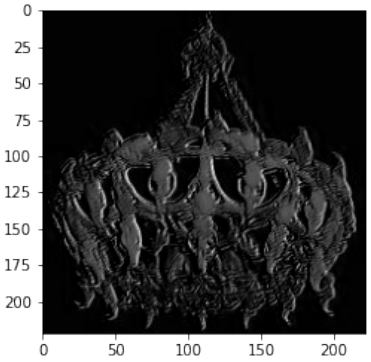
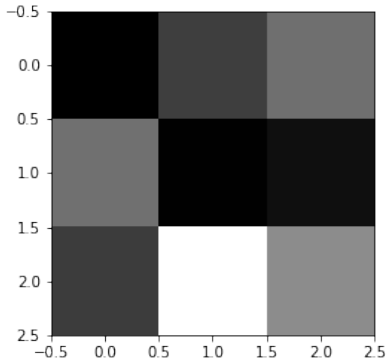
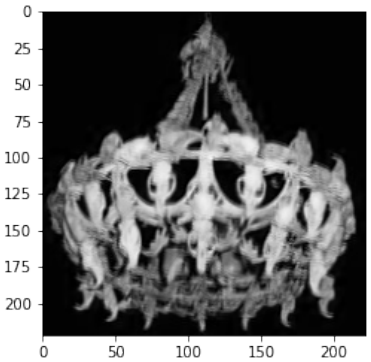
CONVOLUTION LAYER I

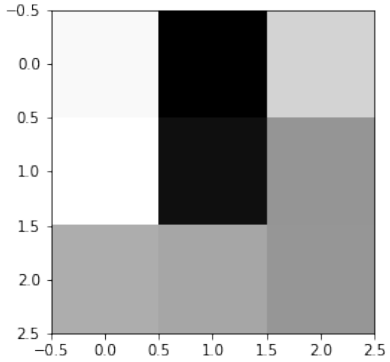
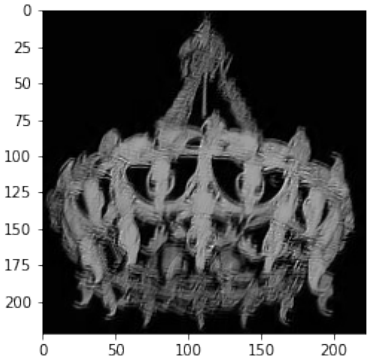
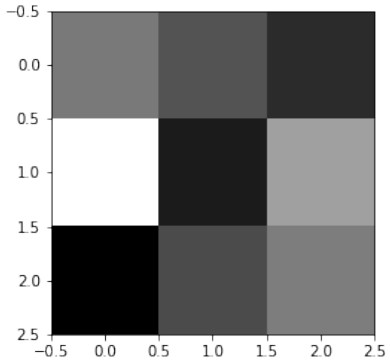
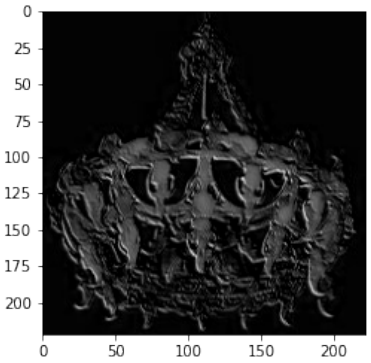
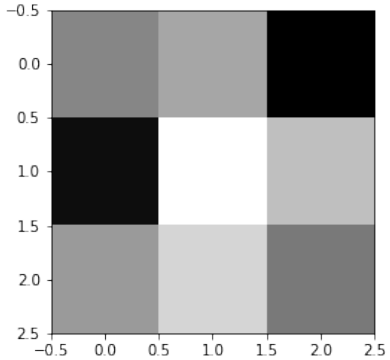
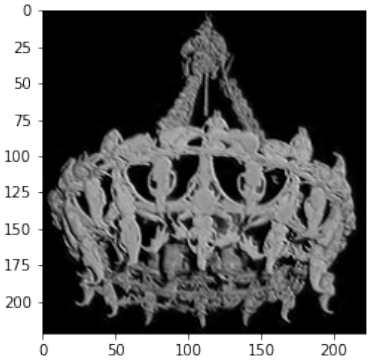
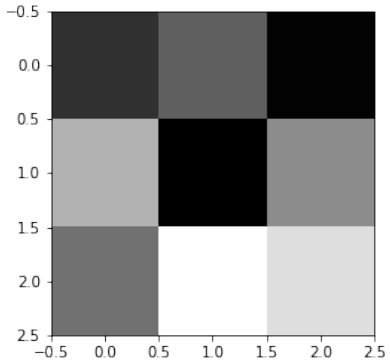
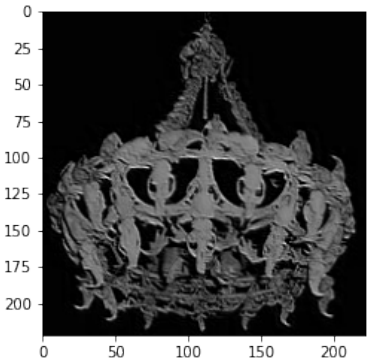
Dimension:
224 x 224 x 3

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

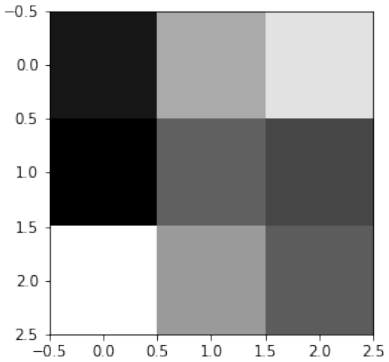
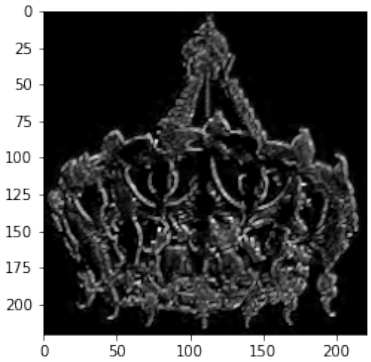
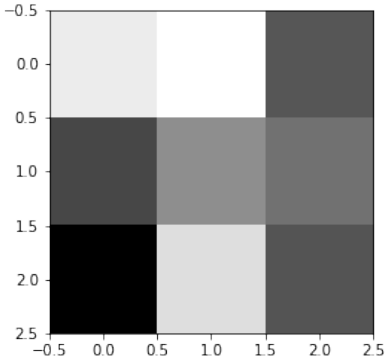
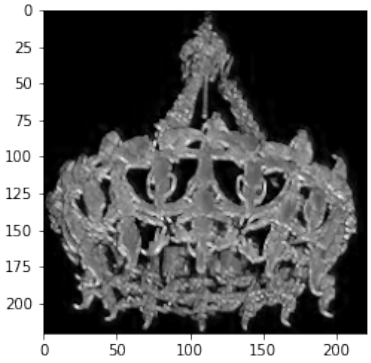
FILTERS	FEATURE MAPS
	
	
	
	

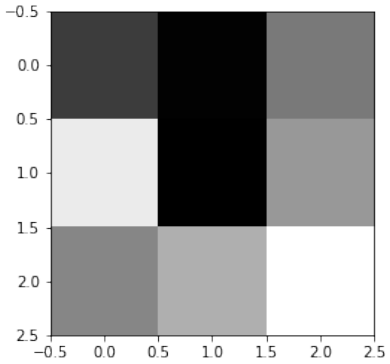
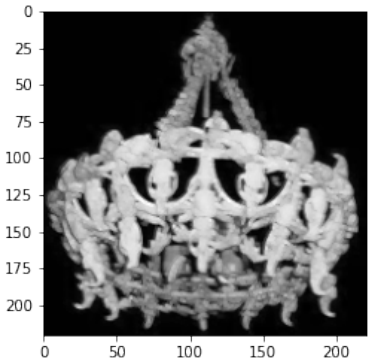
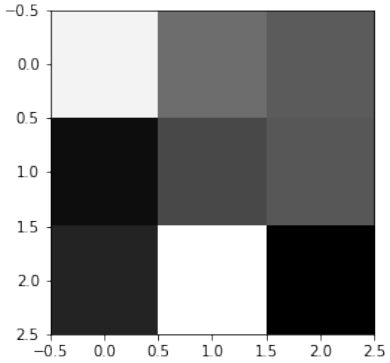
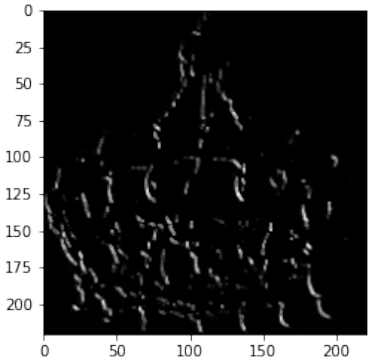
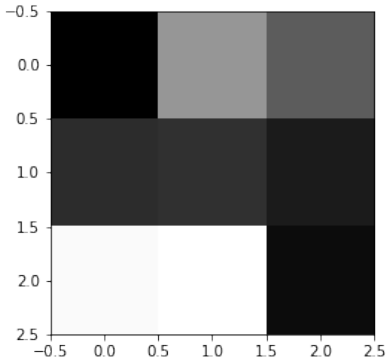
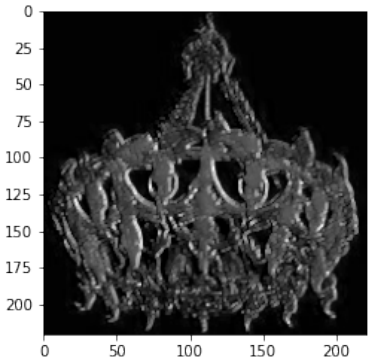
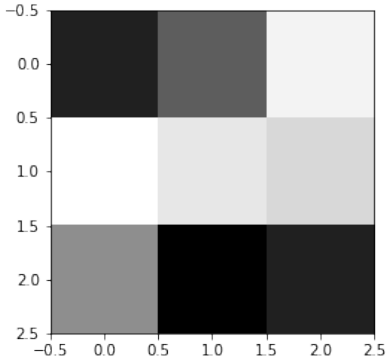
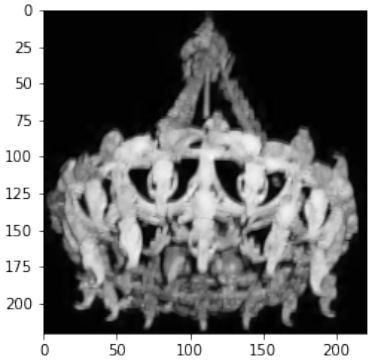
CONVOLUTION LAYER II

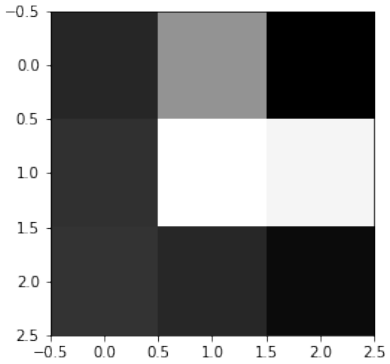
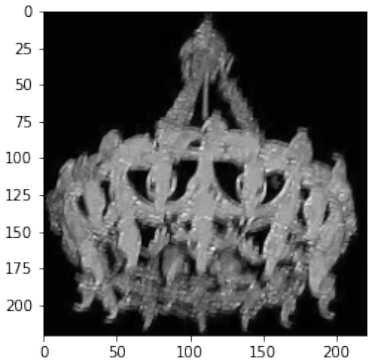
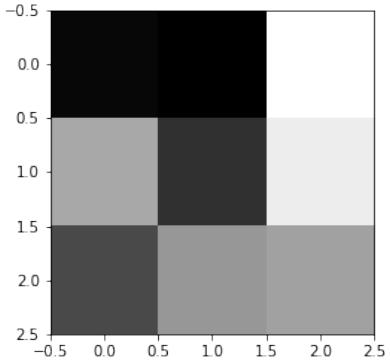
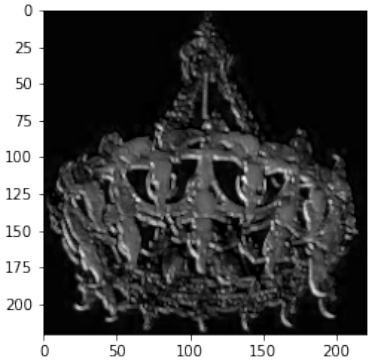
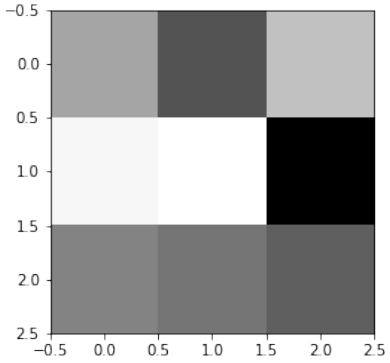
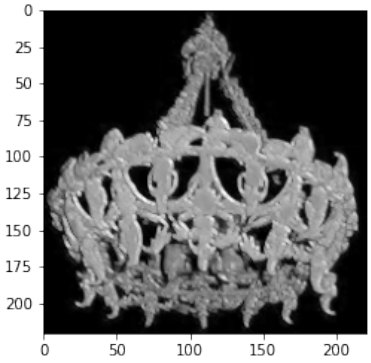
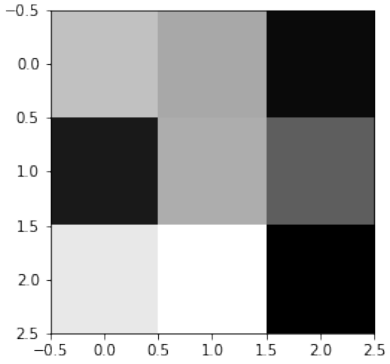
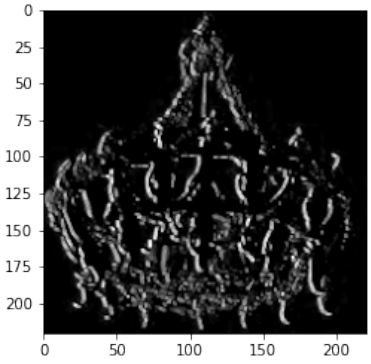
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

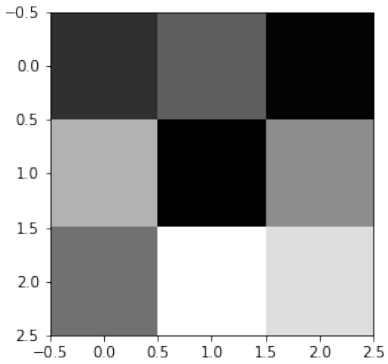
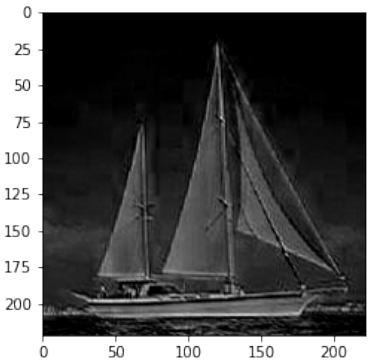
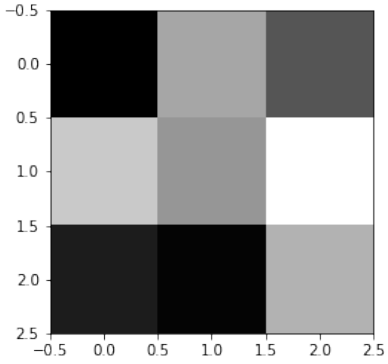
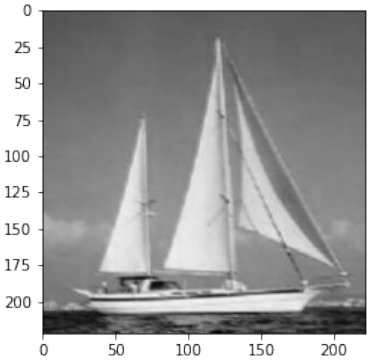
FILTERS	FEATURE MAPS
	
	
	
	

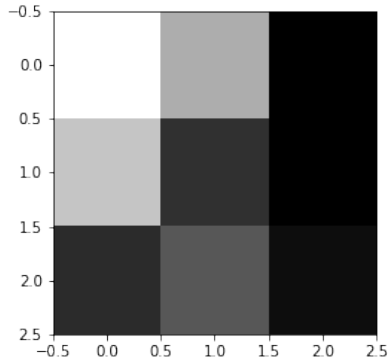
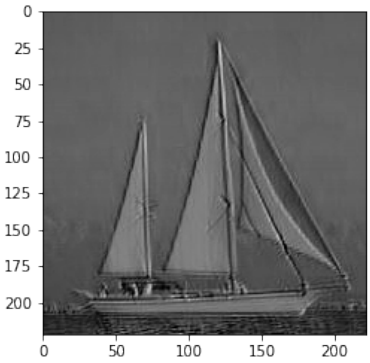
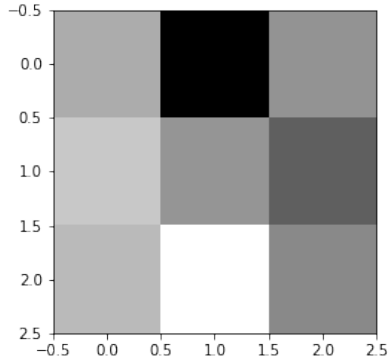
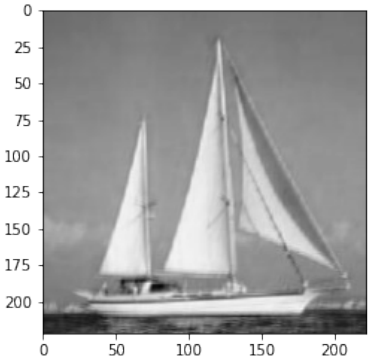
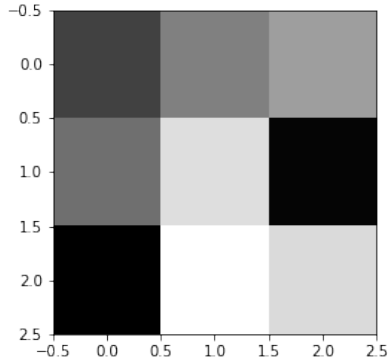
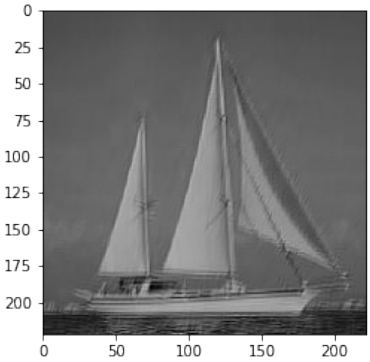
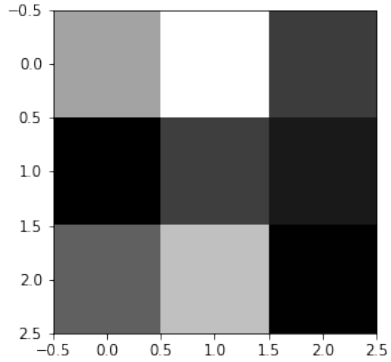
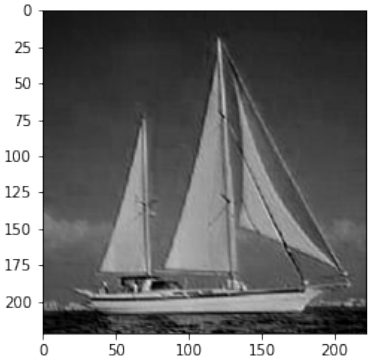
CONVOLUTION LAYER I

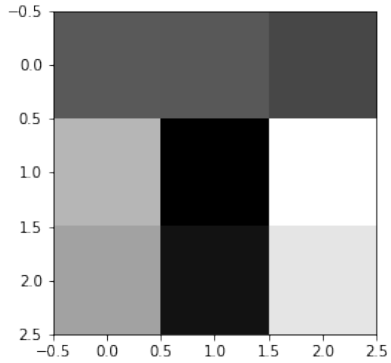
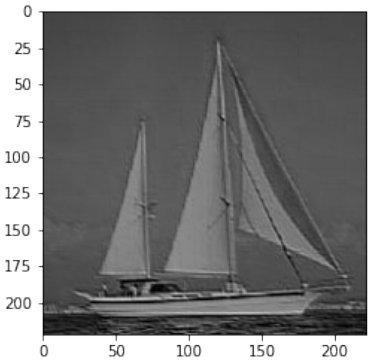
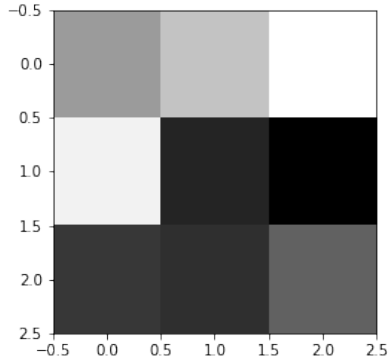
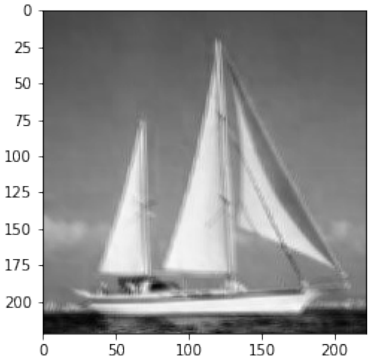
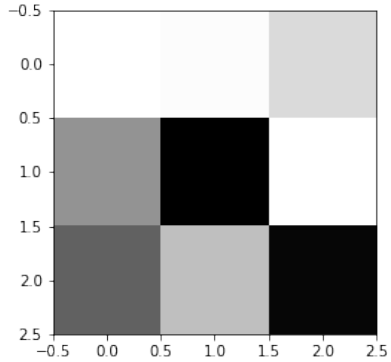
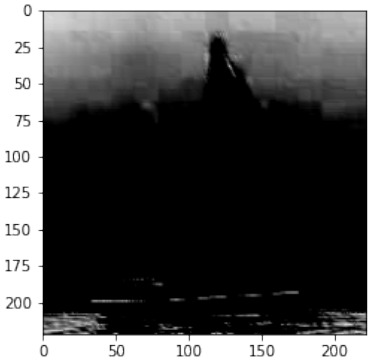
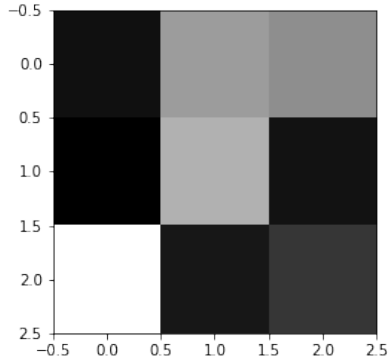
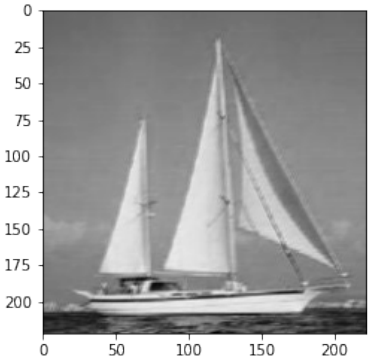
Dimension:
224 x 224 x 3

Stride: 1
Padding: 0, K = 32
Filter Size: 3 x 3

Feature Map:
222 x 222 x 32

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

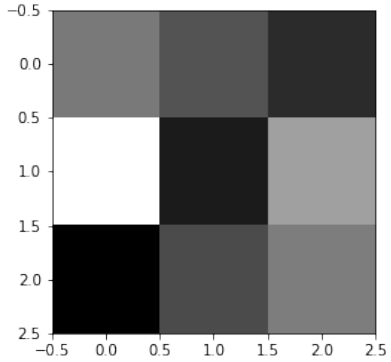
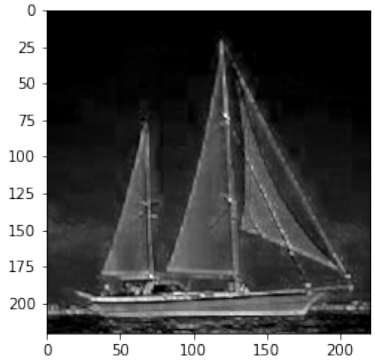
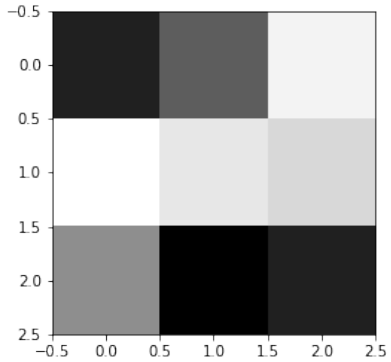
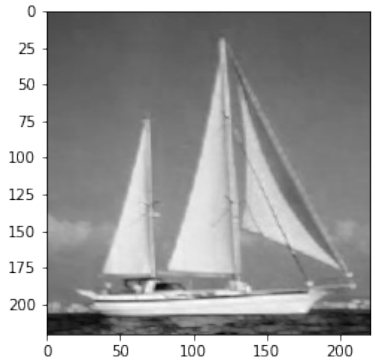
FILTERS	FEATURE MAPS
	
	
	
	

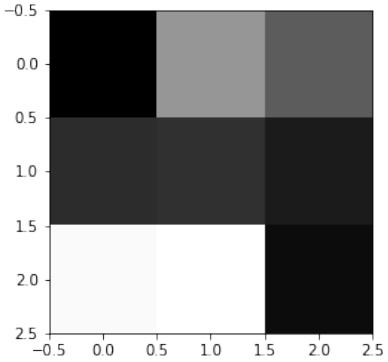
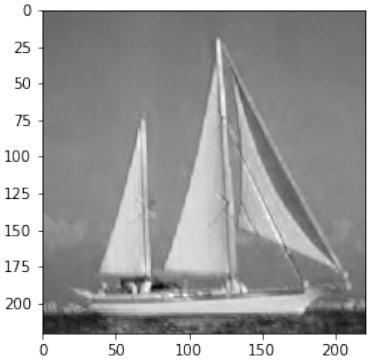
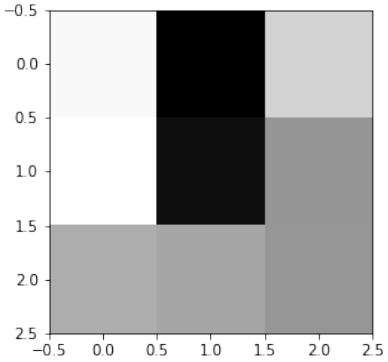
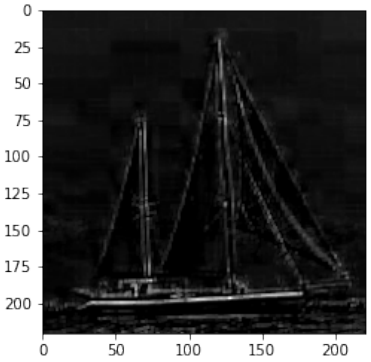
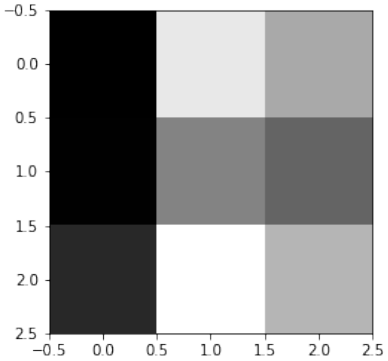
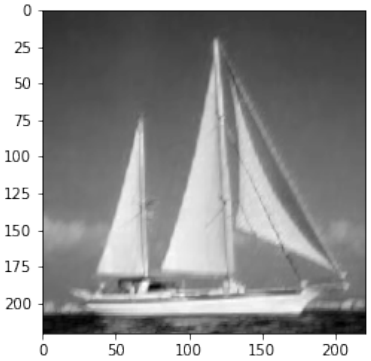
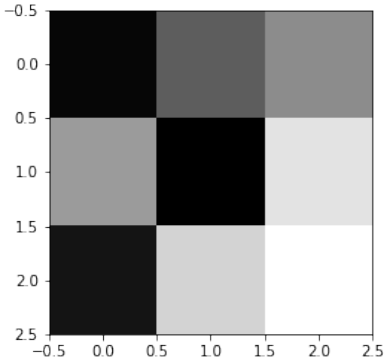
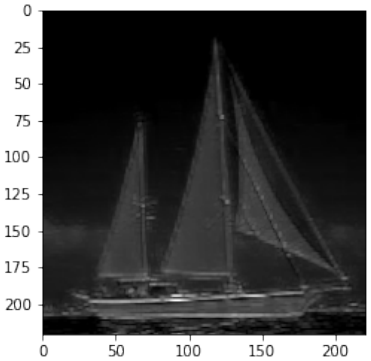
CONVOLUTION LAYER II

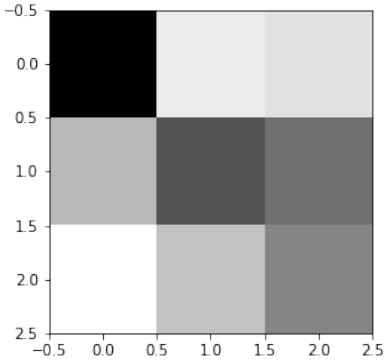
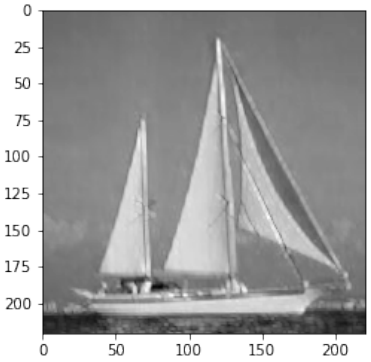
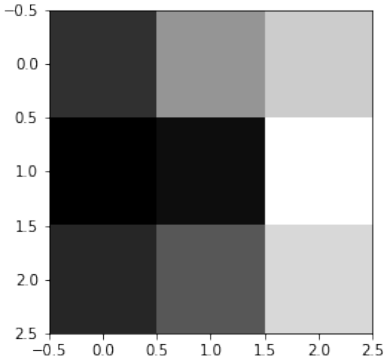
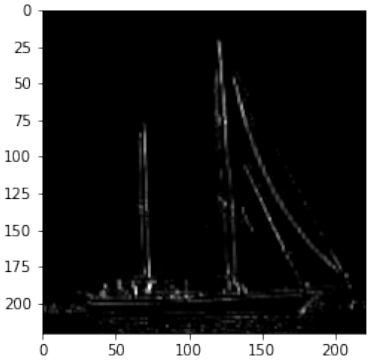
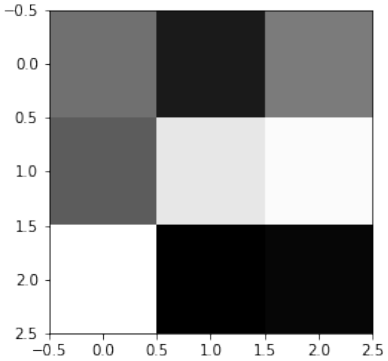
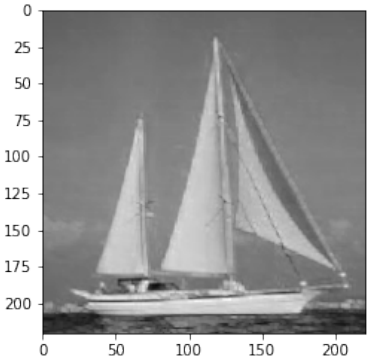
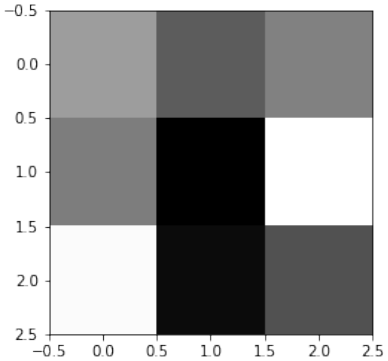
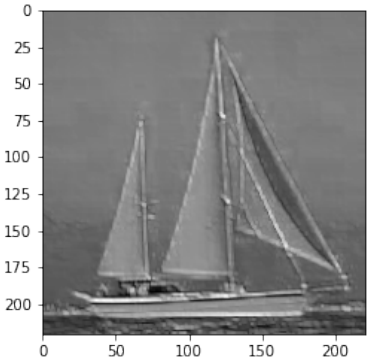
Dimension:
224 x 224 x 32

Stride: 1
Padding: 0, K = 64
Filter Size: 3 x 3

Feature Map:
220 x 220 x 64

FILTERS	FEATURE MAPS
	
	

FILTERS	FEATURE MAPS
	
	
	
	

FILTERS	FEATURE MAPS
	
	
	
	

VGG19 PRE-TRAINED MODEL

The VGG network is the a successor of the AlexNet and it was created by a group named as Visual Geometry Group.

VGG19 is a variant of VGG model which in short consists of 19 layers (16 convolution layers, 3 Fully connected layer, 5 MaxPool layers and 1 SoftMax layer). Here, making use of pertained VGG19, by modifying the classification layer to 3 nodes and by retraining the classification layer alone.

ARCHITECTURE

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv4 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv4 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv4 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0

Modifying the classification layer to 3 by below snippet and the code is trained and tested.

```
last_layer = pre_trained_model.get_layer('block5_pool')
last_output = last_layer.output
```

```
x = tf.keras.layers.GlobalMaxPooling2D()(last_output)
x = tf.keras.layers.Dense(512, activation='relu')(x)
x = tf.keras.layers.Dense(3, activation='softmax')(x)
```

RESULTS

The stopping criteria considered is by the validation loss. The accuracy, loss and the confusion matrix is presented.

ACCURACY AND LOSS

Data	Accuracy	Loss
Training	1.00	0.0040
Validation	1.00	0.0426
Test	1.00	0.0220

```
Epoch 20/25
4/4 [==============================] - 19s 5s/step - loss: 0.0076 - acc: 1.0000
Epoch 21/25
4/4 [==============================] - 19s 5s/step - loss: 0.0043 - acc: 1.0000
Epoch 22/25
4/4 [==============================] - 19s 5s/step - loss: 0.0061 - acc: 1.0000
Epoch 23/25
4/4 [==============================] - 19s 5s/step - loss: 0.0064 - acc: 1.0000
Epoch 24/25
4/4 [==============================] - 21s 5s/step - loss: 0.0049 - acc: 1.0000
Epoch 25/25
4/4 [==============================] - 20s 5s/step - loss: 0.0040 - acc: 1.0000
```

```
2/2 [==============================] - 8s 4s/step - loss: 0.0220 - acc: 1.0000
test_loss, test accuracy [0.02201954834163189, 1.0]
```

CONFUSION. MATRIX

ACTUAL CLASS				
PREDICTED CLASS		Ketch	Chandelier	Bonsai
	Ketch	20	0	0
	Chandelier	0	20	0
	Bonsai	0	0	20

COMPARISON WITH SELF CODED CNN

The accuracy and loss of training, validation and test data are improved a lot by using VGG19 model when compared with self coded CNN. Obtained an 100% accuracy for our test data.

COMPARISON CHART

Data	Accuracy		Loss	
	CNN	VGG19	CNN	VGG19
Training	1.00	1.00	2.7E-06	0.0040
Validation	0.75	1.00	53.7344	0.0426
Test	0.74	1.00	53.2120	0.0220

VISUALISATION OF INFLUENCE OF INPUT PIXELS

Visualizing the influence of input pixels on any of the 5 neurons in the last convolutional layer of a pretrained VGG19 network after passing the input image. Considered the guided-backpropagation algorithm to find the influence.

GUIDED BACKPROPAGATION

The gradient tells about the influence as shown. Compute these partial derivatives w.r.t all the inputs i.e. x0 , x1 ,…, xMN and then visualize this gradient matrix as an image itself. Back propagate the gradients till the first hidden layer.

$$\frac{\partial h_j}{\partial x_i} = 0 \Rightarrow \text{No influence}$$

$$\frac{\partial h_j}{\partial x_i} = \text{large} \Rightarrow \text{High influence}$$

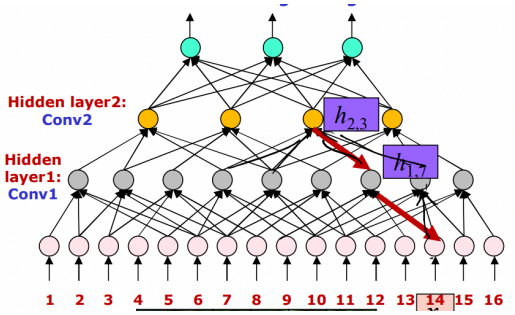
$$\frac{\partial h_j}{\partial x_i} = \text{small} \Rightarrow \text{Low influence}$$

We feed an input image to the CNN and do a forward pass. We considered one neuron in some feature map at some layer finding the influence of the input on this neuron by retaining this neuron and set all other neurons in the layer to zero. We now back propagate all the way to the inputs.

Negative gradients and the gradients associated with –ve activation values is also set to zero.

EXAMPLE

$$h_{2,3} = \sum_{i=1}^{9} w_{i,3}^{(h2)} \, h_{1,i} \qquad h_{1,i} = \sum_{j=1}^{16} w_{j,i}^{(h1)} \, x_j$$



$$\frac{\partial h_{2,3}}{\partial x_{14}} = \sum_{i=1}^{9} \frac{\partial h_{2,3}}{\partial h_{1,i}} \frac{\partial h_{1,i}}{\partial x_{14}}$$

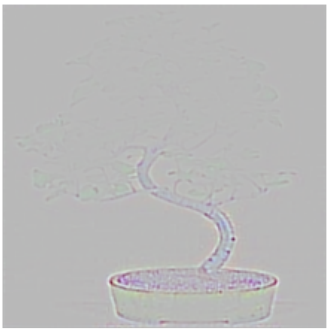
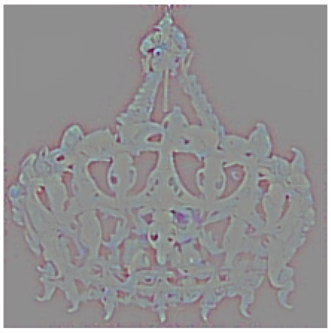
$$\frac{\partial h_{2,3}}{\partial x_{14}} = \sum_{i=1}^{9} \left[\frac{\partial \sum_{i=1}^{9} w_{i,3}^{(h2)} h_{1,i}}{\partial h_{1,i}} \right] \left[\frac{\partial \sum_{j=1}^{16} w_{j,i}^{(h1)} x_j}{\partial x_{14}} \right]$$

$$\frac{\partial h_{2,3}}{\partial x_{14}} = \sum_{i=1}^{9} \left[w_{i,3}^{(h2)} \right] \left[w_{14,i}^{(h1)} \right]$$

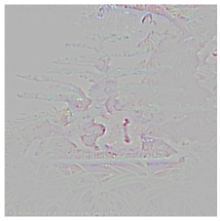
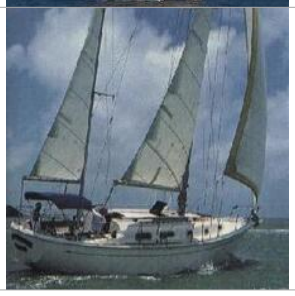
$$\frac{\partial h_{2,3}}{\partial x_{14}} = \left(w_{7,3}^{(h2)} * w_{14,7}^{(h1)} \right) + \left(w_{8,3}^{(h2)} * w_{14,8}^{(h1)} \right)$$

IMAGES CONSIDERED IN QUESTION 1

Visualized the influence of input pixels similar to the Grad-CAM based visualisation algorithm and result is presented.

INPUT IMAGE	RESULTANT GRADIENT
	
	
	

FEW MORE EXAMPLES

INPUT IMAGE	RESULTANT GRADIENT
	
	
	
	
	
	

Gradient Weighted Class Activation Mapping (Grad CAM)

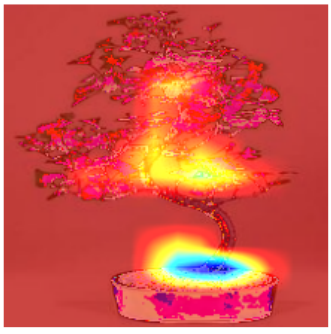
Grad CAM is the class activation mapping function which helps us to visualise the uses the gradients of any target concept, flowing into the final convolutional layer to produce a coarse localization map highlighting important regions in the image for predicting the concept.

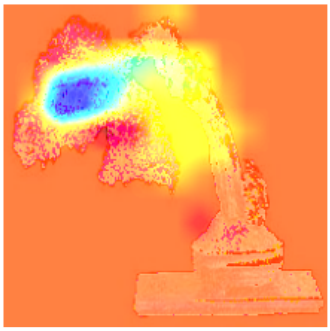
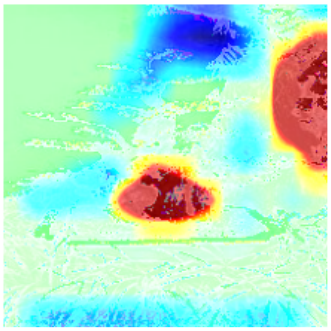
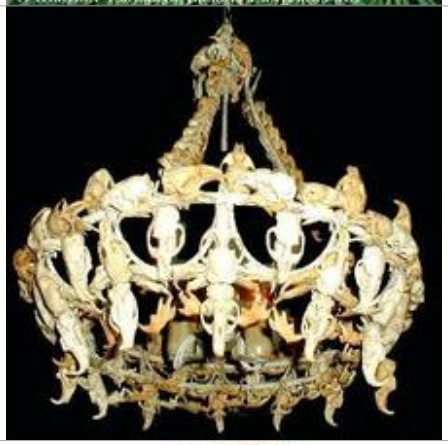
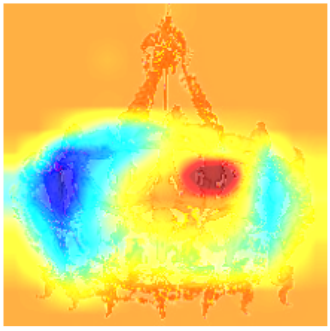
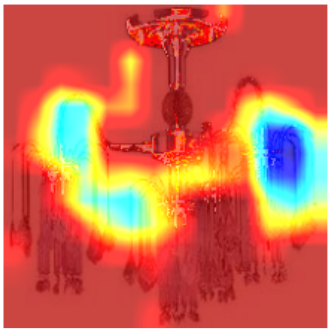
REFERENCE

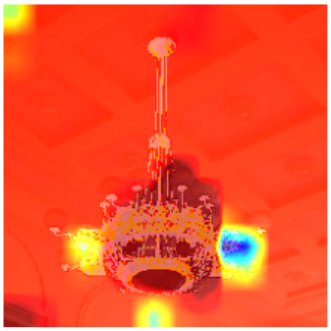
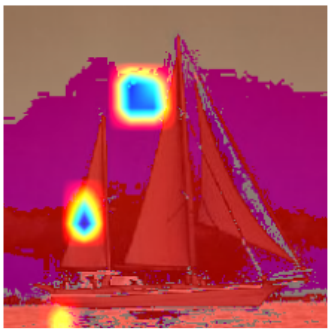
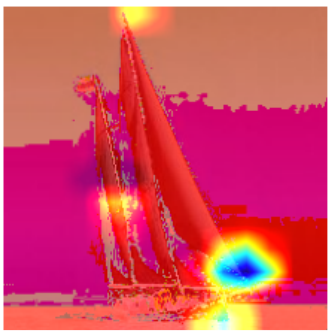
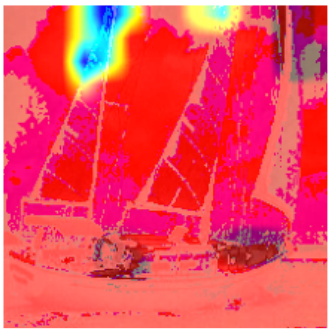
https://pyimagesearch.com/2020/03/09/grad-cam-visualize-class-activation-maps-with-keras-tensorflow-and-deep-learning/

Grad CAM uses heat maps to visualise the important information learnt by the network. We have downloaded code from the pyimagesearch.com referenced above and obtained the results.

The same input images are chosen and also they are compared with the guided back propagation maps.

INPUT IMAGE	RESULTANT HEAT MAPS
	

INPUT IMAGE	RESULTANT HEAT MAPS
	
	
	
	

INPUT IMAGE	RESULTANT HEAT MAPS
	
	
	
	

GITHUB LINK

https://github.com/Rajesh-Smartino/Deep-Learning

TEAM DETAILS
(Group 22)

Name	Rollno	Mailid
Rajesh R	S21005	s21005@students.iitmandi.ac.in
Naisarg Pandya	S21012	s21012@students.iitmandi.ac.in
Ashok Kumar	D19042	d19042@students.iitmandi.ac.in