VIT-Vellore, SCOPE

CSE6037 - Deep Learning and its Applications

SUTHAR MANAN BHARATKUMAR 20MAI0016

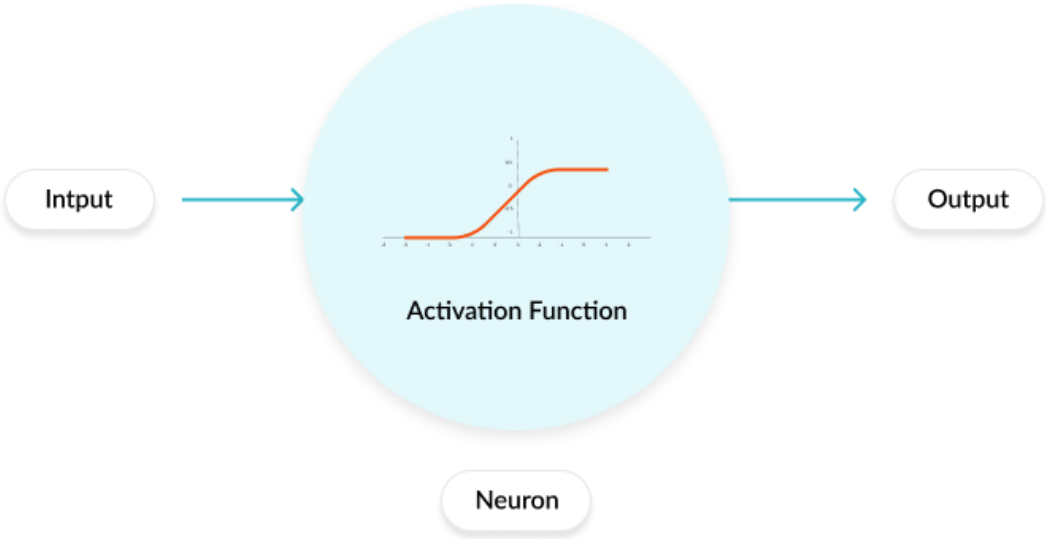
Assessment 2

GitHub Link: https://github.com/manansuthar55/CSE6037_20MAI0016/tree/main/Assessment_2

Problem 1: Demonstrate Activation Functions used in Neural Networks

Activation Functions for Neural Networks

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction.



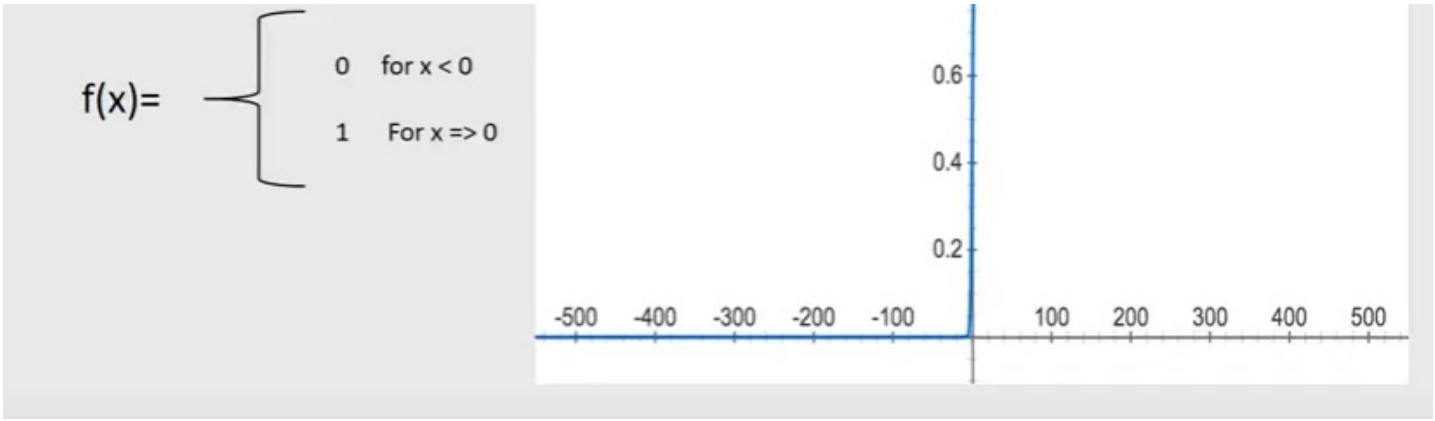
The Activation Functions can be basically divided into 2 types-

1. **Linear Activation Function**
2. **Non-linear Activation Functions**

There are several activations which are generally preffered, which we will see here...

1. Step function

The first thing that comes to our mind when we have an activation function would be a threshold based classifier i.e. whether or not the neuron should be activated based on the value from the linear transformation. In other words, if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e. its output is not considered for the next hidden layer.



In [19]:

```
import numpy as np
demonp = np.random.uniform(-5, 5, (5))
demonp
```

Out[19]:

```
array([ 4.71378107, -2.47828926,  3.85750137, -4.81662257, -2.92057655])
```

In [20]:

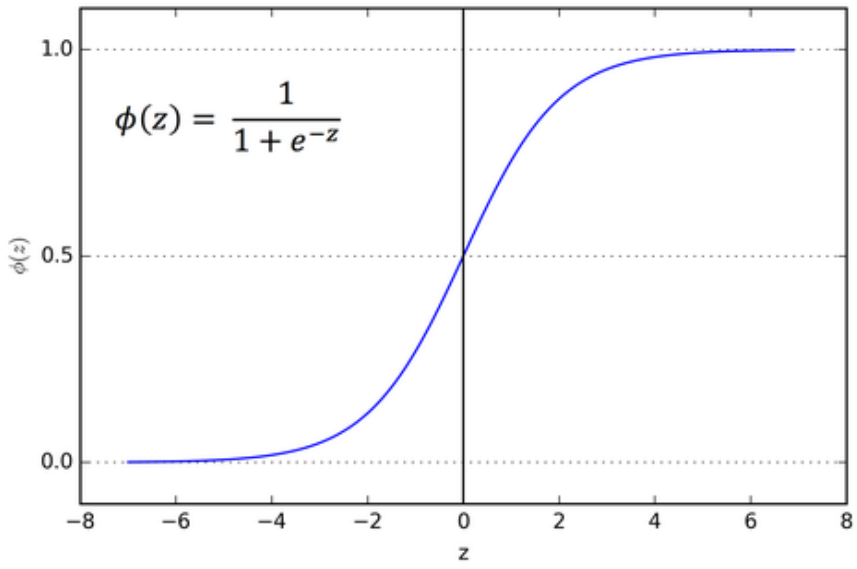
```
def binary_step(x):
    if x<0:
        return 0
    else:
        return 1
step=[]
for i in demonp:
    step.append(binary_step(i))
step
```

Out[20]:

```
[1, 0, 1, 0, 0]
```

2. Sigmoid

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output.Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice. The function is differentiable.That means, we can find the slope of the sigmoid curve at any two points. The function is monotonic but function's derivative is not. The logistic sigmoid function can cause a neural network to get stuck at the training time.



In [21]:

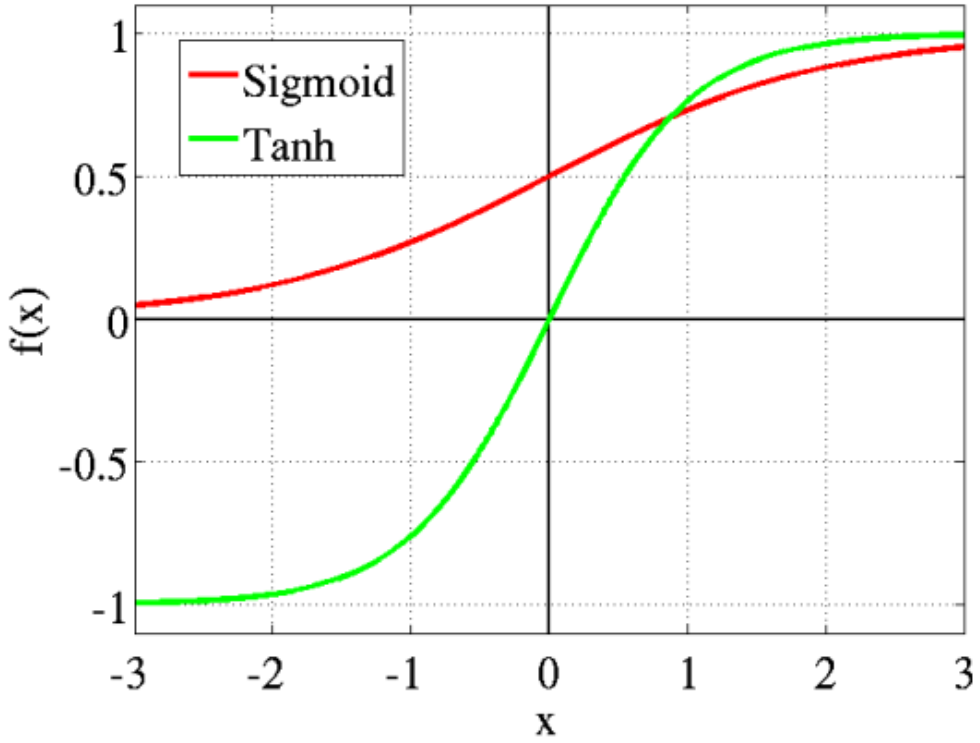
```
def sigmoid(x):
    return 1/(1+np.exp(-x))

sgmd = sigmoid(demonp)
sgmd
```

Out[21]:

```
array([0.99110897, 0.07739427, 0.97931615, 0.00802909, 0.05114571])
```

3. Tanh or hyperbolic tangent

tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped). The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph. The function is differentiable. The function is monotonic while its derivative is not monotonic. The tanh function is mainly used classification between two classes. Both tanh and logistic sigmoid activation functions are used in feed-forward nets.



In [22]:

```
import math
def tanhactfun(x):
    return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))
tan = tanhactfun(demonp)
tan
```

Out[22]:

```
array([ 0.99983906, -0.98602442,  0.99910823, -0.99986898, -0.99420585])
```

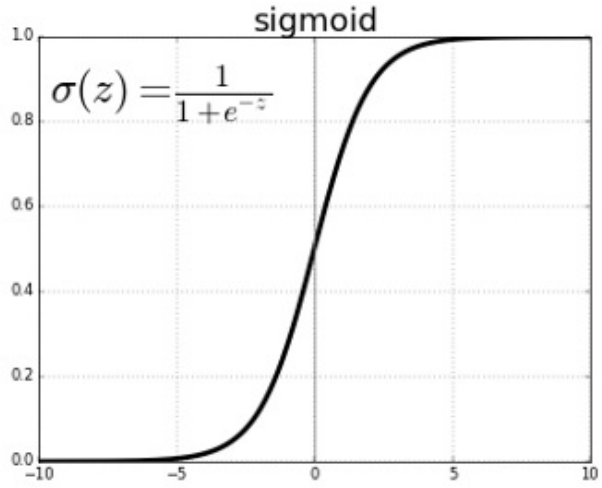
4. ReLU (Rectified Linear Unit)

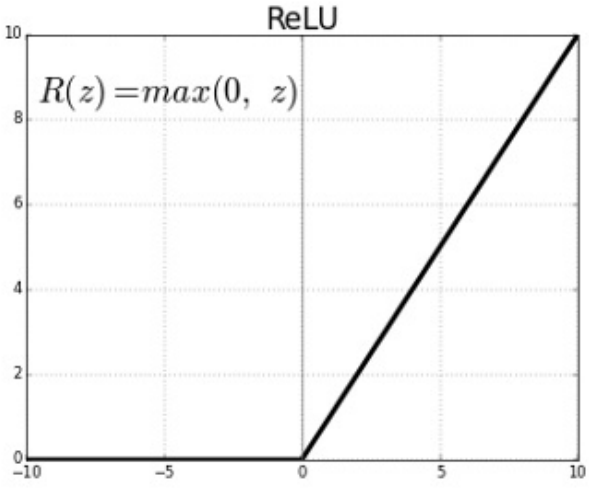
The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning. As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

Range: 0 to infinity

The function and its derivative both are monotonic. But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in

turns affects the resulting graph by not mapping the negative values appropriately.





In [23]:

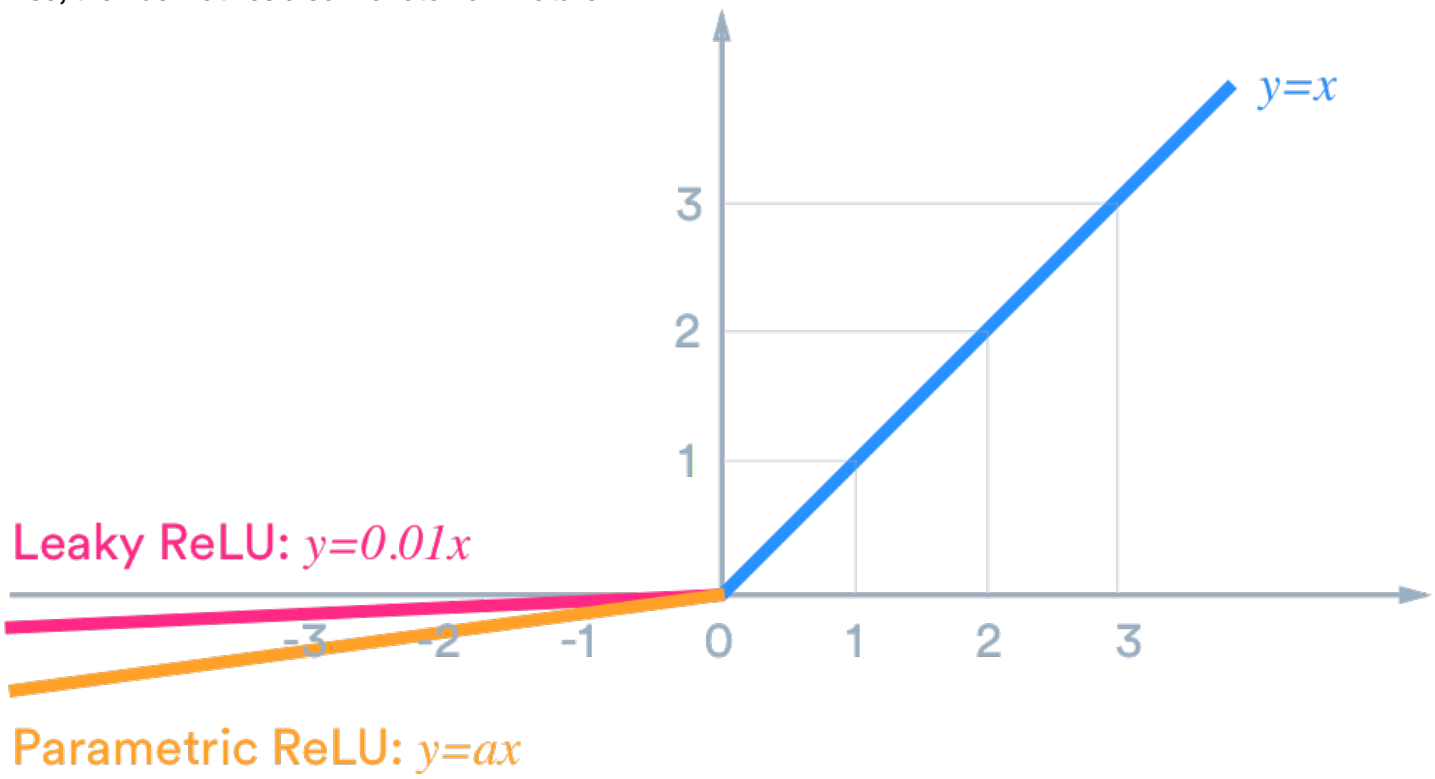
```
def reluactfun(x):
    return (max(0,x))
reluact=[]
for i in demonp:
    reluact.append(reluactfun(i))
reluact
```

Out[23]:

```
[4.713781074034406, 0, 3.8575013747604565, 0, 0]
```

5. Leaky ReLU

It is an attempt to solve the dying ReLU problem. The leak helps to increase the range of the ReLU function. Usually, the value of a is 0.01 or so. When a is not 0.01 then it is called Randomized ReLU. Therefore the range of the Leaky ReLU is (-infinity to infinity). Both Leaky and Randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.



In [24]:

```
def reluactfun(x):
    return (max(0.1*x, x))
reluact=[]
```

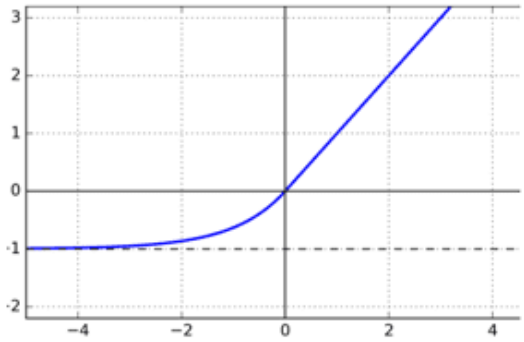
```
for i in demonp:
    reluact.append(reluactfun(i))
reluact
```

Out[24]:

```
[4.713781074034406,
 -0.24782892640304022,
 3.8575013747604565,
 -0.4816622574567537,
 -0.2920576546944821]
```

6. ELU (Exponential Linear Unit)

Exponential Linear Unit or ELU for short is also a variant of Rectiufied Linear Unit (ReLU) that modifies the slope of the negative part of the function. Unlike the leaky relu and parametric ReLU functions, instead of a straight line, ELU uses a log curve for defning the negatice values.



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

In [25]:

```
def elu_function(x, a):
    return max(a*(np.exp(x)-1),x)
elu=[]
a = 0.55
for i in demonp:
    elu.append(elu_function(i,a))
elu
```

Out[25]:

```
[60.760068146359714,
 -0.503862364086107,
 25.49079570769976,
 -0.5455482570054477,
 -0.5203535694720284]
```

7. Swish

Swish is a lesser known activation function which was discovered by researchers at Google. Swish is as computationally efficient as ReLU and shows better performance than ReLU on deeper models. The values for swish ranges from negative infinity to infinity.

In [26]:

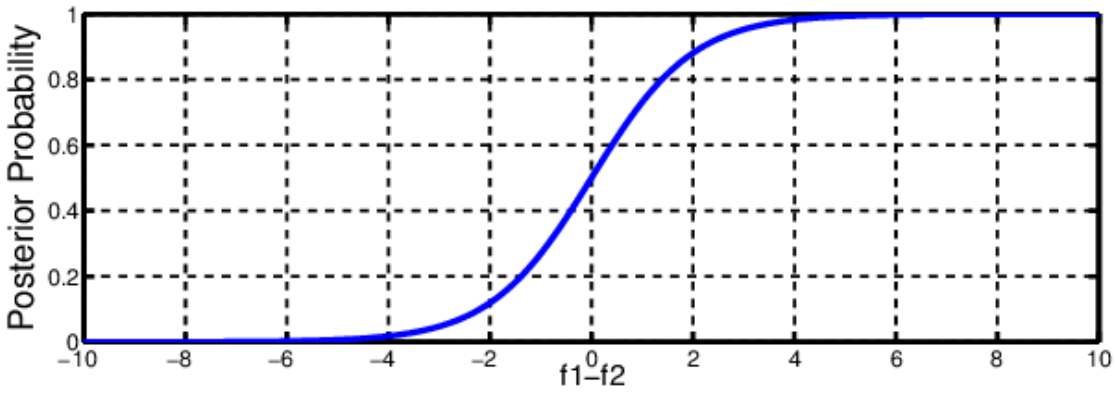
```
def swish_function(x):
    return x/(1-np.exp(-x))
swish=[]
for i in demonp:
    swish.append(swish_function(i))
swish
```

Out[26]:

```
[4.756450209698867,
 0.22693182881506965,
 3.9407324272723794,
 0.03930425132587174,
 0.166395840790475]
```

8. Softmax

Softmax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems. The softmax function can be used for multiclass classification problems.



In [27]:

```
def softmax_function(x):
    z = np.exp(x)
    z_ = z/z.sum()
    return z_
softm = softmax_function(demonp)
softm
```

Out[27]:

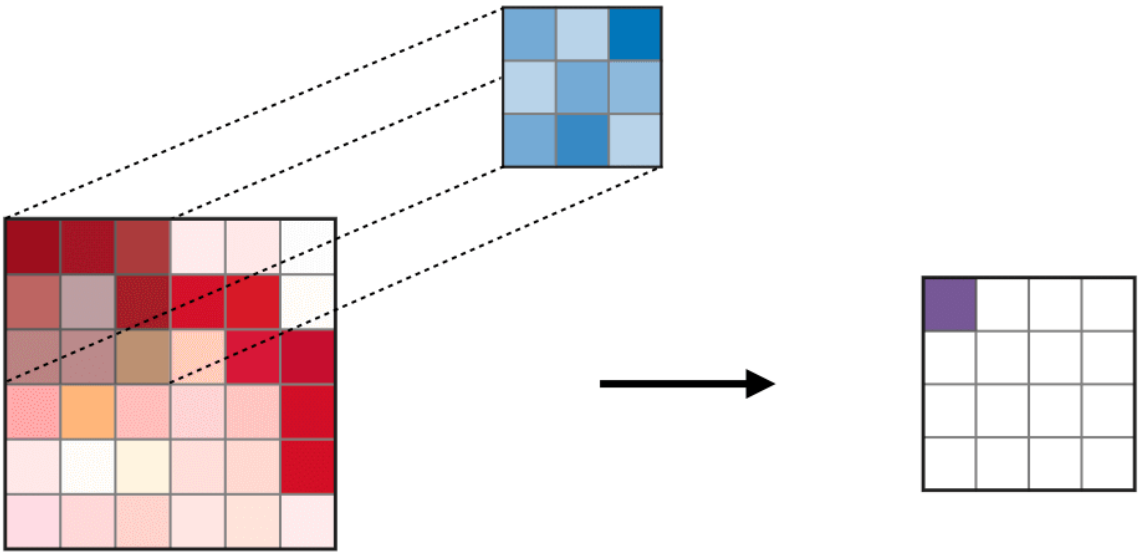
```
array([7.01238671e-01, 5.27702798e-04, 2.97843626e-01, 5.09171566e-05,
       3.39083354e-04])
```

Assessment 2

Problem 2: Try different hyperparameters and perform convolution transform on an image

Convolution operation on an image

The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input *I* with respect to its dimensions. Its hyperparameters include the filter size *F* and stride *S*. The resulting output *O* is called feature map or activation map.



In [2]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import skimage.measure
```

Loading the image form local directory and checking the entropy of the image

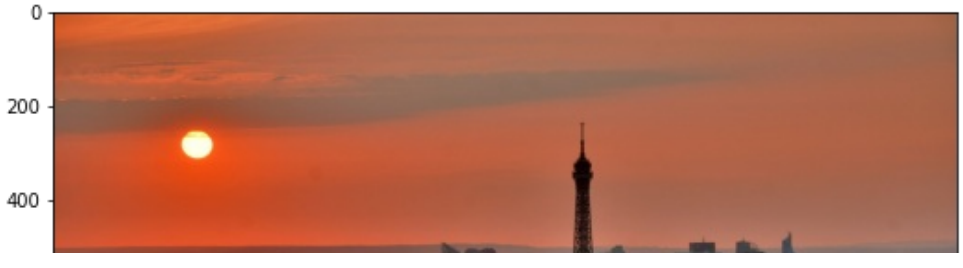
In [3]:

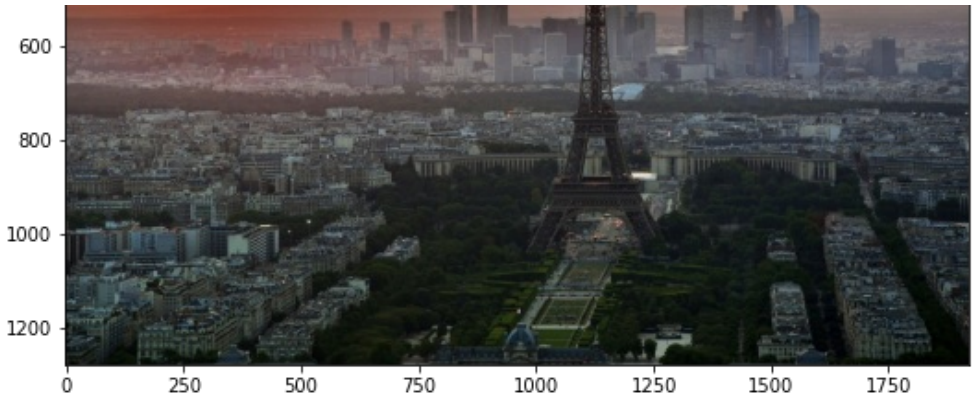
```
input_image = cv2.imread('/content/drive/MyDrive/DumpYard/paris-843229_1920.jpg')
input_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
fig, ax = plt.subplots(1, figsize=(9,6))
entropy1 = skimage.measure.shannon_entropy(input_image)
print("Entropy for the original image is:",entropy1)
plt.imshow(input_image)
```

```
Entropy for the original image is: 7.4473113904233825
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x7f257706e150>
```

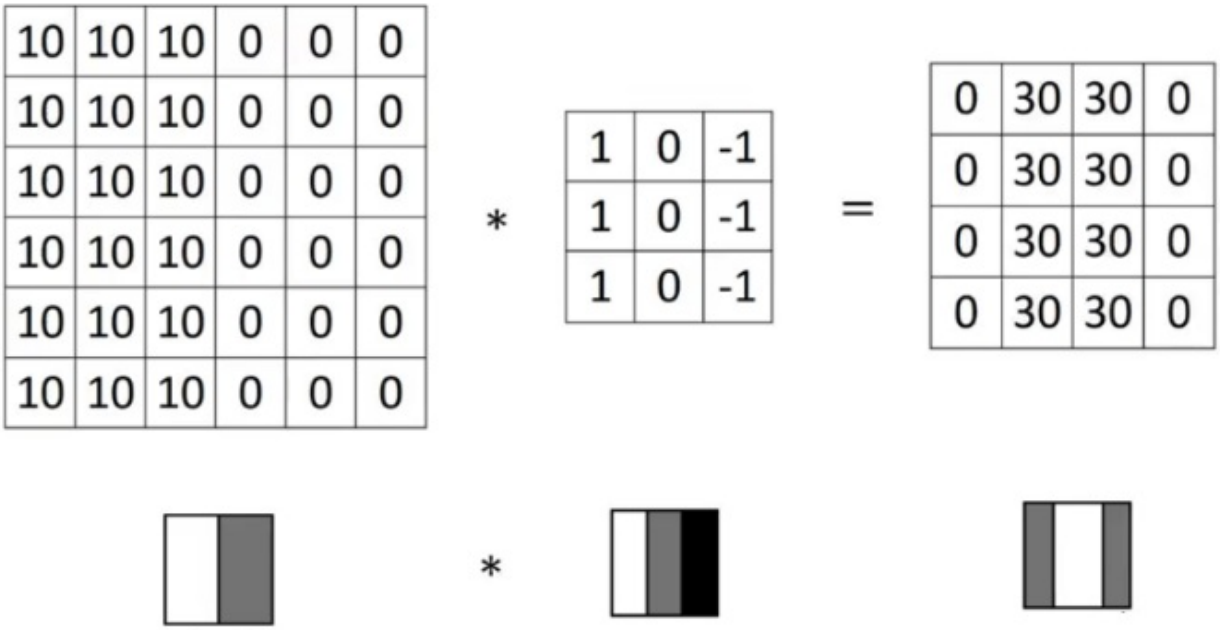




Use a simple 3x3 filter on the image and demonstrate the transformed image with its entropy

Filter

A filter of size *F × F* applied to an input containing *C* channels is a *F × F × C* volume that performs convolutions on an input of size *I × I × C* and produces an output feature map (also called activation map) of size *O × O × 1*.



Kernel

In [4]:

```
sharpenfilter = np.array(([[0, -1, 0], [-1, 9, -1], [0, -1, 0]]), np.float32)/9
sharpenfilter
```

Out[4]:

```
array([[ 0.        , -0.11111111,  0.        ],
       [-0.11111111,  1.        , -0.11111111],
       [ 0.        , -0.11111111,  0.        ]], dtype=float32)
```

Transformation and output image with its Entropy

In [5]:

```
sharpen = cv2.filter2D(src=input_image, kernel=sharpenfilter, ddepth=-1)
fig, ax = plt.subplots(1, figsize=(9,6))
entropy2 = skimage.measure.shannon_entropy(sharpen)
print("Entropy for the sharpened image is:",entropy2)
plt.imshow(sharpen)
```

```
Entropy for the sharpened image is: 6.647542774487621
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x7f25762e1190>
```

Use a simple 5x5 filter on the image and demonstrate the transformed image with its entropy

Filter or Kernel

In [6]:

```
gaussian_blur_filter = np.array(([[1,4,6,4,1], [4,16,24,16,4], [6,24,36,24,6], [4,16,24
,16,4], [1,4,6,4,1]]), np.float32)/256
gaussian_blur_filter
```

Out[6]:

```
array([[0.00390625, 0.015625  , 0.0234375 , 0.015625  , 0.00390625],
       [0.015625  , 0.0625    , 0.09375   , 0.0625    , 0.015625  ],
       [0.0234375 , 0.09375   , 0.140625  , 0.09375   , 0.0234375 ],
       [0.015625  , 0.0625    , 0.09375   , 0.0625    , 0.015625  ],
       [0.00390625, 0.015625  , 0.0234375 , 0.015625  , 0.00390625]],
      dtype=float32)
```

Transformation and Output image with Entropy

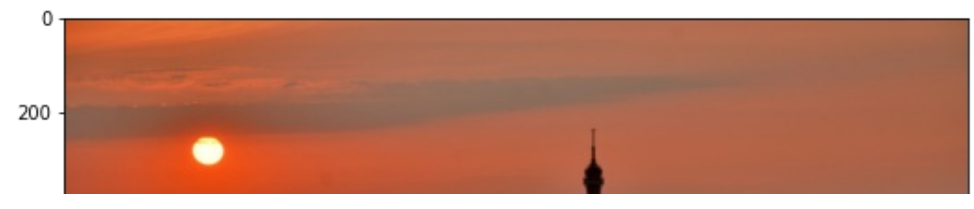
In [7]:

```
gaussianBlur = cv2.filter2D(src=input_image, kernel=gaussian_blur_filter, ddepth=-1)
entropy3 = skimage.measure.shannon_entropy(gaussianBlur)
print("Entropy for the blurred image is:",entropy3)
fig, ax = plt.subplots(1, figsize=(9,6))
plt.imshow(gaussianBlur)
```

```
Entropy for the blurred image is: 7.395531620068165
```

Out[7]:

```
<matplotlib.image.AxesImage at 0x7f25746579d0>
```



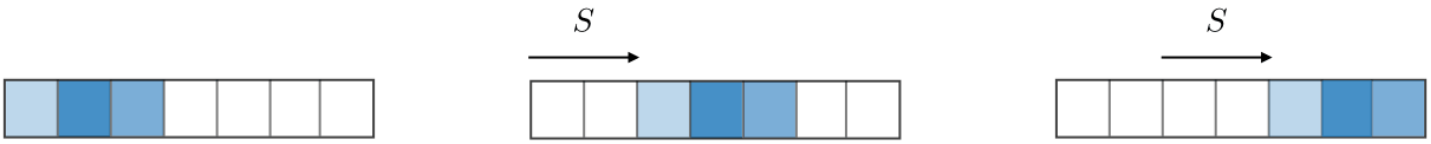
In [8]:

```
grey_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)
```

Filtering an image with 3x3 filter with a stride = 2 and then calculate its Entropy

Stride

For a convolutional or a pooling operation, the stride *S* denotes the number of pixels by which the window moves after each operation.



In [9]:

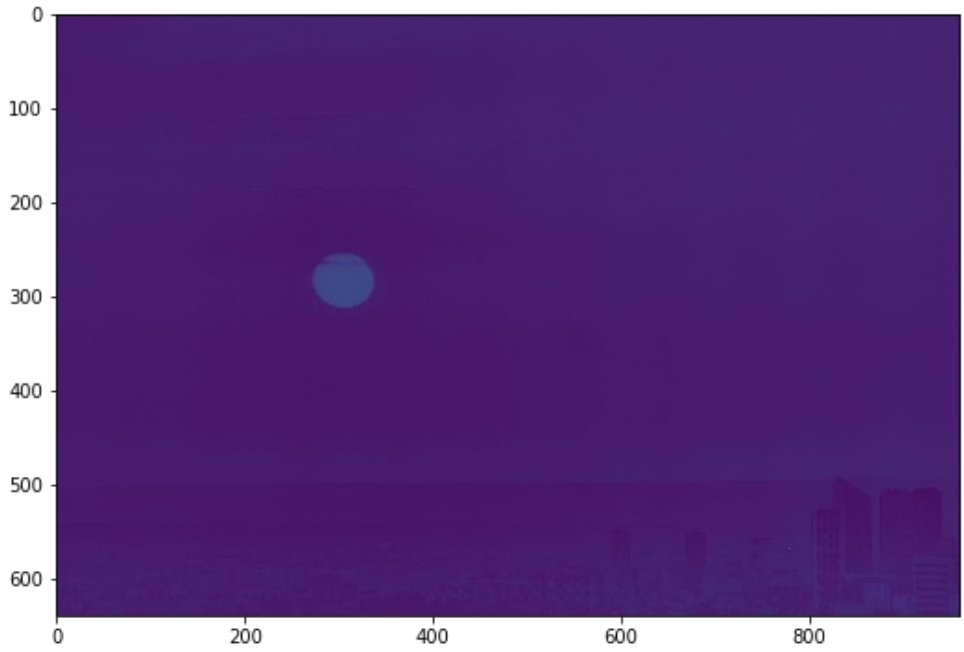
```
def convolve2D(image, kernel, strides):
    kernel = np.flipud(np.fliplr(kernel))
    xKernShape = kernel.shape[0]
    yKernShape = kernel.shape[1]
    xImgShape = image.shape[0]
    yImgShape = image.shape[1]
    xOutput = int(((xImgShape - xKernShape + 2) / strides) + 1)
    yOutput = int(((yImgShape - yKernShape + 2) / strides) + 1)
    output = np.zeros((xOutput, yOutput))
    for y in range(image.shape[1]):
        if y > image.shape[1] - yKernShape:
            break
        if y % strides == 0:
            for x in range(image.shape[0]):
                if x > image.shape[0] - xKernShape:
                    break
                try:
                    if x % strides == 0:
                        output[x, y] = (kernel * image[x: x + xKernShape, y: y + yKernS
hape]).sum()
                except:
                    break
    return output

kernel = np.array(([[0, -1, 0], [-1, 9, -1], [0, -1, 0]]), np.float32)/9
optimg = convolve2D(grey_image, kernel, strides = 2)
entropy6 = skimage.measure.shannon_entropy(optimg)
print("Entropy for an image with kernel = 3x3 and stride = 2 is:",entropy6)
fig, ax = plt.subplots(1, figsize=(9,6))
plt.imshow(optimg)
```

```
Entropy for an image with kernel = 3x3 and stride = 2 is: 3.222862544247595
```

Out[9]:

```
<matplotlib.image.AxesImage at 0x7f25745ac690>
```

<matplotlib.image.AxesImage at 0x7f25745ac690>



Filtering an image with 5x5 filter with a stride = 2 and then calculate its Entropy

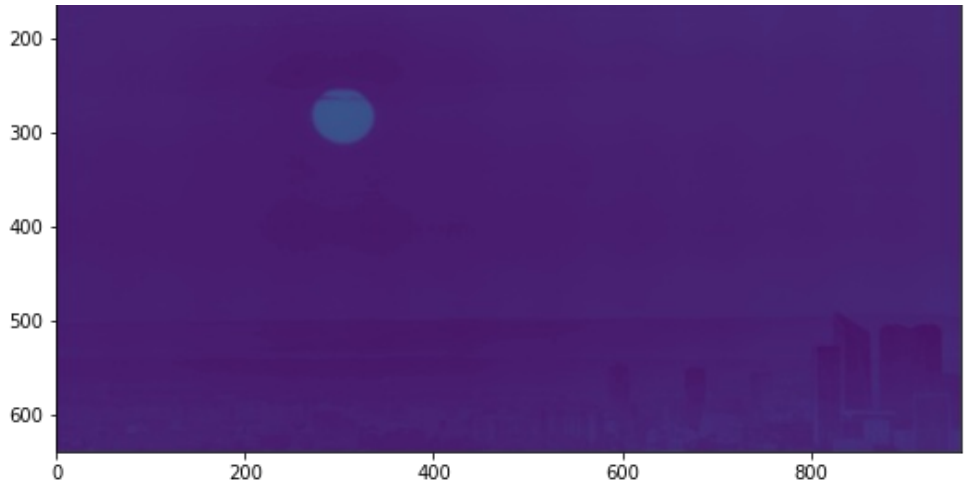
In [10]:

```
def convolve2D(image, kernel, strides):
    kernel = np.flipud(np.fliplr(kernel))
    xKernShape = kernel.shape[0]
    yKernShape = kernel.shape[1]
    xImgShape = image.shape[0]
    yImgShape = image.shape[1]
    xOutput = int(((xImgShape - xKernShape + 2 ) / strides) + 1)
    yOutput = int(((yImgShape - yKernShape + 2 ) / strides) + 1)
    output = np.zeros((xOutput, yOutput))
    for y in range(image.shape[1]):
        if y > image.shape[1] - yKernShape:
            break
        if y % strides == 0:
            for x in range(image.shape[0]):
                if x > image.shape[0] - xKernShape:
                    break
                try:
                    if x % strides == 0:
                        output[x, y] = (kernel * image[x: x + xKernShape, y: y + yKernS
hape]).sum()
                except:
                    break
    return output

kernel = np.array(([[1,4,6,4,1], [4,16,24,16,4], [6,24,36,24,6], [4,16,24,16,4], [1,4,6
,4,1]]), np.float32)/256
optimg = convolve2D(grey_image, kernel, strides = 2)
entropy7 = skimage.measure.shannon_entropy(optimg)
print("Entropy for an image with kernel = 5x5 and stride = 2 is:",entropy7)
fig, ax = plt.subplots(1, figsize=(9,6))
plt.imshow(optimg)
```

```
Entropy for an image with kernel = 5x5 and stride = 2 is: 4.189303058746414
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x7f2574581b90>
```



Filtering an image with 3x3 filter with stride = 1 and zero padding, then calculate its Entropy

Zero Padding

Zero-padding denotes the process of adding *P* zeroes to each side of the boundaries of the input.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

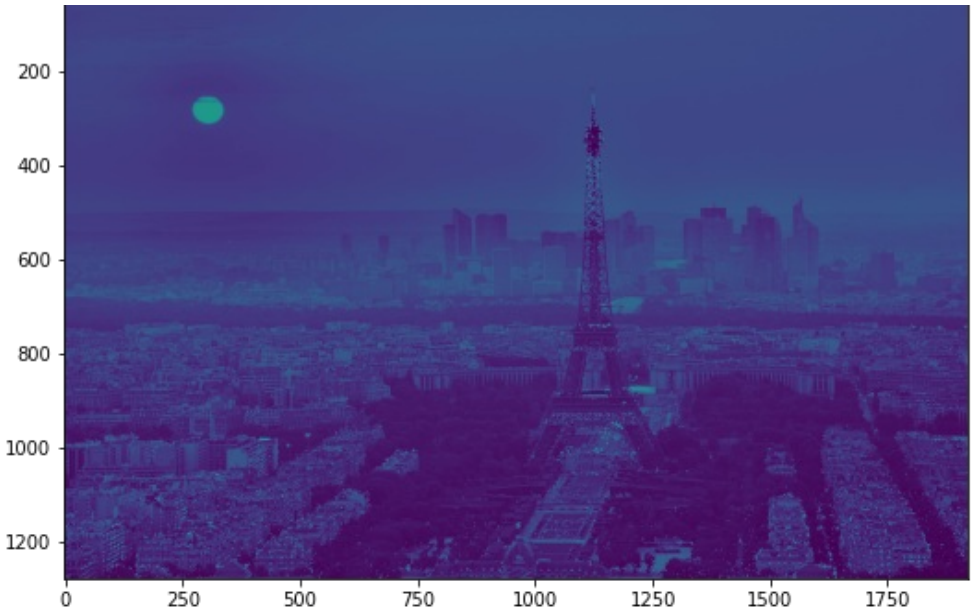
In [11]:

```
def convolve2d(image, kernel):
    output = np.zeros_like(image)
    image_padded = np.zeros((image.shape[0] + 2, image.shape[1] + 2))
    image_padded[1:-1, 1:-1] = image
    for x in range(image.shape[1]):
        for y in range(image.shape[0]):
            output[y, x]=(kernel * image_padded[y: y+3, x: x+3]).sum()
    return output
# For a 3x3 Kernel and stride = 1 and Zero_padding
KERNEL = np.array(([[0, -1, 0], [-1, 9, -1], [0, -1, 0]]), np.float32)/9
stride_1_3x3 = convolve2d(grey_image, kernel=KERNEL)
entropy4 = skimage.measure.shannon_entropy(stride_1_3x3)
print("Entropy for the 3x3 image with zero padding and stride = 1 is:",entropy4)
fig, ax = plt.subplots(1, figsize=(9,6))
plt.imshow(stride_1_3x3)
```

```
Entropy for the 3x3 image with zero padding and stride = 1 is: 6.055754583579611
```

Out[11]:

```
<matplotlib.image.AxesImage at 0x7f25745972d0>
```



Filtering an image with 5x5 filter with stride = 1 and zero padding, then calculate its Entropy

In [12]:

```
def convolve2d(image, kernel):
    output = np.zeros_like(image)
    image_padded = np.zeros((image.shape[0] + 4, image.shape[1] + 4))
    image_padded[2:-2, 2:-2] = image
    for x in range(image.shape[1]):
        for y in range(image.shape[0]):
            output[y, x]=(kernel * image_padded[y: y+5, x: x+5]).sum()
    return output
# For a 5x5 Kernel and stride = 1 and Zero_padding
KERNEL = np.array(([[1,4,6,4,1], [4,16,24,16,4], [6,24,36,24,6], [4,16,24,16,4], [1,4,6
,4,1]]), np.float32)/256
opimg = convolve2d(grey_image, kernel=KERNEL)
entropy5 = skimage.measure.shannon_entropy(opimg)
print("Entropy for the 5x5 image with zero padding and stride = 1 is:",entropy5)
fig, ax = plt.subplots(1, figsize=(9,6))
plt.imshow(opimg)
```

```
Entropy for the 5x5 image with zero padding and stride = 1 is: 6.715918610760809
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x7f257459bc50>
```