This is a sample notebook for TP2 in pytorch, corresponding to `tgabor.py`.
This is a "Markdown" (comment) cell. Double click on it to see the source code. Click on run while selected to see it formatted.

Below is a "Code cell". Click on "Run" to execute it.

```
In [1]: import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torchvision

        import math
        import time

        import matplotlib.pyplot as plt
        import numpy as np
```

Here we check whether a GPU is available.
You may comment or uncomment the second line to run with or withou a GPU.
You will need to re-run all the cells after this one to see the difference.

```
In [2]: device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        #device = "cpu"
        print(device)
```

```
cuda:0
```

The next two cells contain two versions of the same kernel initialization function.
The first one uses sequential loops while the second one uses parallel tensor operations.
This function corresponds to eight calls to the TP2 `gabor_kernel()` function, each corresponding to a different orientation but to a signle wavelength. It only has to be called four times, one for each wavelength.
Unlike in `TP2 gabor_kernel()`, the result is not a 2D array of complex numbers, it is a 3D array of real numbers. The third dimension is $2 \times 8 = 16$. Real parts are in feature maps 0 to 7 and imaginary parts are in feature maps 8 to 15.
Both functions include execution time measurement.

```
In [3]: def gabor_kernel_init_seq(weight,lambd = 16.0, nt = 8, n = 0, sl = 0.7, st = 1.
        4, nl = 4.0):
            if torch.cuda.is_available(): torch.cuda.synchronize()
            t0 = time.time()
            if n <= 0: n = 1+2*int(nl*lambd)
            gl = -0.5/(sl*sl)  # lambda direction scale factor
            gt = -0.5/(st*st)  # theta direction scale factor
            for t in range (0, nt):
                theta = t*math.pi/nt  # orientation
                c = math.cos(theta)/lambd  # rotation and scaling
                s = math.sin(theta)/lambd  # parameters
                x0 = 0.5*(n-1)*(c+s)  # translation
                y0 = 0.5*(n-1)*(c-s)  # parameters
                sc = 1.0/(2*math.pi*sl*st*lambd*lambd)  # Gaussian normalization factor
                for y in range (0,n):
                    for x in range (0,n):
                        xr = c*x+s*y-x0  # centering, rotation and scaling
                        yr = c*y-s*x-y0  # centering, rotation and scaling
                        a = 2.0*math.pi*xr  # wave phase
                        g = sc*math.exp(gl*xr*xr+gt*yr*yr)  # Gaussian amplitude
                        weight[t+0*nt, 0, y, x] = g*math.cos(a)  # real component
                        weight[t+1*nt, 0, y, x] = g*math.sin(a)  # imaginary component
            if torch.cuda.is_available(): torch.cuda.synchronize()
            print("kernel init sequential %dx%d: %.2f ms"% (n, n, 1000*(time.time()-t
        0)))
```

```
In [4]: def gabor_kernel_init(weight,lambd = 16.0, nt = 8, n = 0, sl = 0.7, st = 1.4, nl
        = 4.0):
            if torch.cuda.is_available(): torch.cuda.synchronize()
            t0 = time.time()
            if n <= 0: n = 1+2*int(nl*lambd)
            gl = -0.5/(sl*sl)  # lambda direction scale factor
            gt = -0.5/(st*st)  # theta direction scale factor
            x = torch.tensor(range(n)).unsqueeze(0).expand(n,n)  # x coordinate
            y = torch.tensor(range(n)).unsqueeze(1).expand(n,n)  # y coordinate
            for t in range (0, nt):
                theta = t*math.pi/nt  # orientation
                c = math.cos(theta)/lambd  # rotation and scaling
                s = math.sin(theta)/lambd  # parameters
                x0 = 0.5*(n-1)*(c+s)  # translation
                y0 = 0.5*(n-1)*(c-s)  # parameters
                sc = 1.0/(2*math.pi*sl*st*lambd*lambd)  # Gaussian normalization factor
                xr = c*x+s*y-x0  # centering, rotation and scaling
                yr = c*y-s*x-y0  # centering, rotation and scaling
                a = 2.0*math.pi*xr  # wave phase
                g = sc*torch.exp(gl*xr*xr+gt*yr*yr)  # Gaussian amplitude
                weight[t+0*nt, 0] = g*torch.cos(a)  # real component
                weight[t+1*nt, 0] = g*torch.sin(a)  # imaginary component
            if torch.cuda.is_available(): torch.cuda.synchronize()
            print("kernel init %dx%d: %.2f ms"% (n, n, 1000*(time.time()-t0)))
```

Here, we define the whole Gabor transform computation as a single network.
At this points, the kernels contain random initial values.
There is a set of filters (`conv2d()`) per wavelength since the size of the fliters depend upon it.
The size of the filters are large compared to typical CNN filters.
`F.adaptive_avg_pool2d()` does a global average pooling resulting in a 1×1 image, hence the need for a squeeze, after which the results can be concatenated.
Unlike in classical convolutional neural networks where convolutions are combined squentially, the convolutions here are computed in parallel and concatenated (after the global pooling and squaring).
The last two lines computes the sum of the "energies" for the real and imaginary parts.

In [5]:
```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv0 = nn.Conv2d(1, 16, 25, bias=False)
        self.conv1 = nn.Conv2d(1, 16, 49, bias=False)
        self.conv2 = nn.Conv2d(1, 16, 97, bias=False)
        self.conv3 = nn.Conv2d(1, 16, 193, bias=False)

    def forward(self, x):
        x0 = torch.squeeze(F.adaptive_avg_pool2d(self.conv0(x)**2, (1, 1)),3)/1
        x1 = torch.squeeze(F.adaptive_avg_pool2d(self.conv1(x)**2, (1, 1)),3)/2
        x2 = torch.squeeze(F.adaptive_avg_pool2d(self.conv2(x)**2, (1, 1)),3)/4
        x3 = torch.squeeze(F.adaptive_avg_pool2d(self.conv3(x)**2, (1, 1)),3)/8
        y = torch.cat((x0, x1, x2, x3), dim = 2)
        y1, y2 = torch.split(y, 8, dim = 1)
        return((y1+y2).transpose(1,2))
```

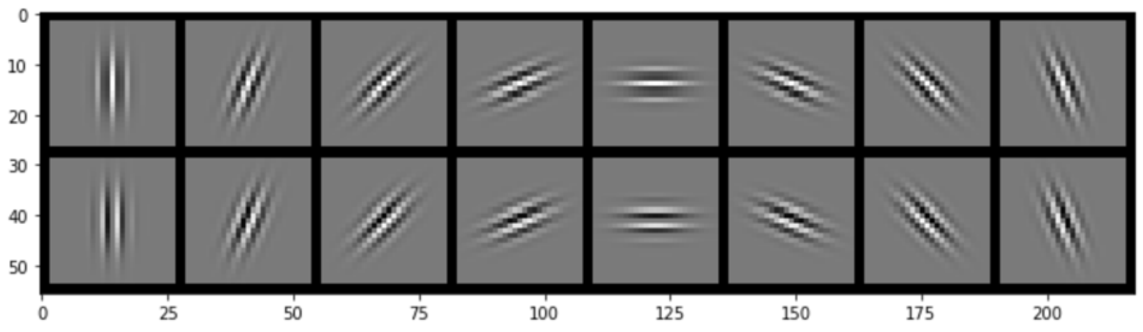
Here we do the net instanciation and initialization.
If a GPU is available and selected, the net structure is moved onto it.
Depending upon whether `net.to(device)` is called at the beginning or at the end, the filter initialization is done on the GPU or on the CPU. You just need to move the # sign and re-run the cell to see the difference in execution time (comment the `device = "cpu"` line in the beginning). Not much difference for this.
You may also try to replace `gabor_kernel_init()` with `gabor_kernel_init_seq()`. Parallelism significantly improves both on CPU and GPU. Sequential execution on GPU is worse thant on CPU.

In [6]:
```python
net = Net()
net.to(device)
if torch.cuda.is_available(): torch.cuda.synchronize()
t0 = time.time()
gabor_kernel_init(net.conv0.weight,lambd = 3.0) # slower due to initializations
gabor_kernel_init(net.conv0.weight,lambd = 3.0)
gabor_kernel_init(net.conv1.weight,lambd = 6.0)
gabor_kernel_init(net.conv2.weight,lambd = 12.0)
gabor_kernel_init(net.conv3.weight,lambd = 24.0)
if torch.cuda.is_available(): torch.cuda.synchronize()
print("total init time: %.2f ms"% (1000*(time.time()-t0)))
#net.to(device)
print(net)
```

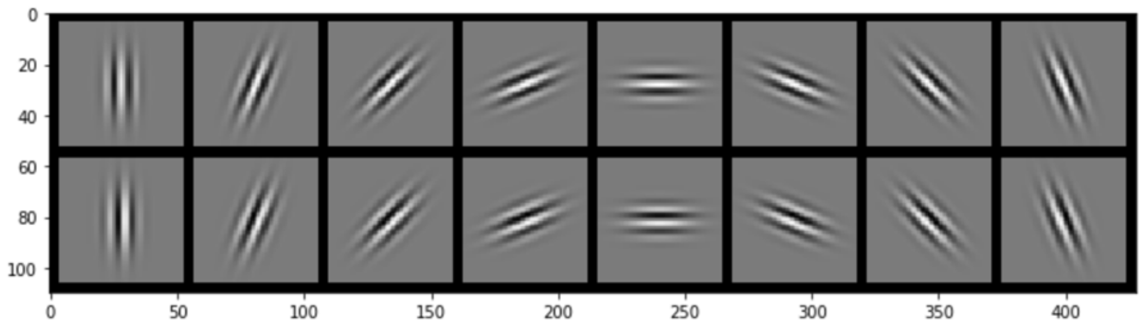
```
kernel init 25x25: 25.03 ms
kernel init 25x25: 1.86 ms
kernel init 49x49: 1.97 ms
kernel init 97x97: 3.19 ms
kernel init 193x193: 7.35 ms
total init time: 40.35 ms
Net(
  (conv0): Conv2d(1, 16, kernel_size=(25, 25), stride=(1, 1), bias=False)
  (conv1): Conv2d(1, 16, kernel_size=(49, 49), stride=(1, 1), bias=False)
  (conv2): Conv2d(1, 16, kernel_size=(97, 97), stride=(1, 1), bias=False)
  (conv3): Conv2d(1, 16, kernel_size=(193, 193), stride=(1, 1), bias=False)
)
```

Visualize the kernels in the following four cells, each corresponding to a scale (wavelength). Though the kernels look similar at all scales, each is twice as big as the previous one, as can bee seen from the scale in pixels of the displayed kernels. The top row correspond to the real part and the bottopn row corresponds to ihe imaginary part. Columns correspond to orientations.

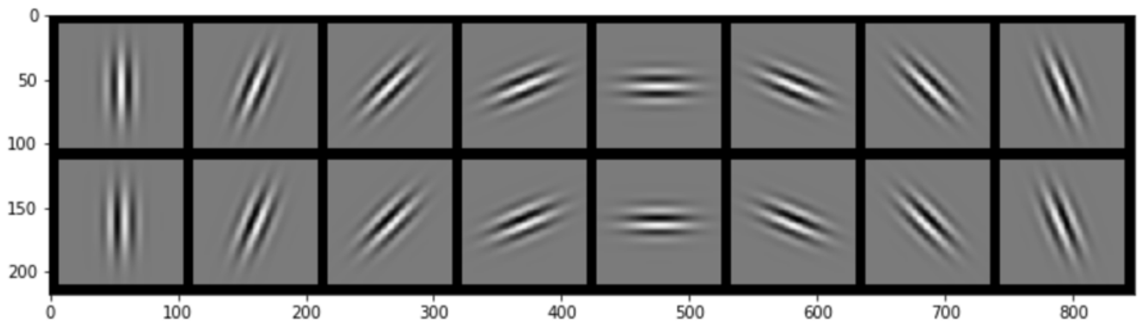
```
In [7]: plt.rcParams["figure.figsize"] = (12,6)
        k0 = list(net.parameters())[0]
        print(k0.shape)
        p = torchvision.utils.make_grid(k0, padding = 2, normalize = True)
        plt.imshow(np.transpose(p.to("cpu").detach().numpy(), (1, 2, 0)))

        torch.Size([16, 1, 25, 25])

Out[7]: <matplotlib.image.AxesImage at 0x7fa317e1fb50>
```



```
In [8]: k1 = list(net.parameters())[1]
        print(k1.shape)
        p = torchvision.utils.make_grid(k1, padding = 4, normalize = True)
        plt.imshow(np.transpose(p.to("cpu").detach().numpy(), (1, 2, 0)))

        torch.Size([16, 1, 49, 49])

Out[8]: <matplotlib.image.AxesImage at 0x7fa2fc0ce750>
```



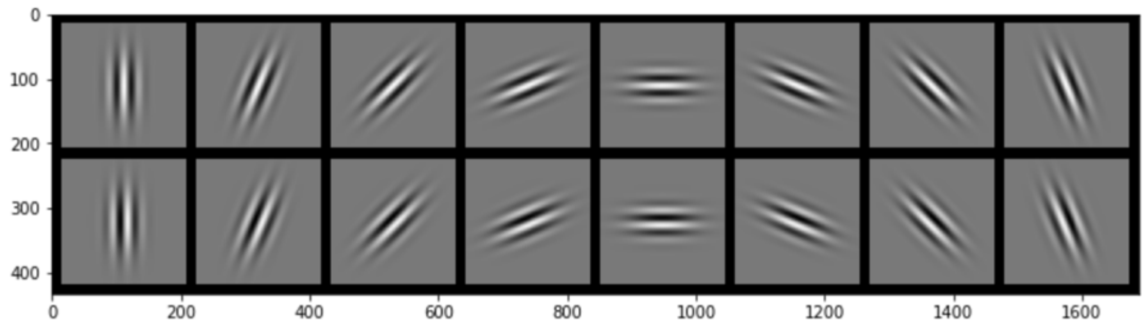
```
In [9]: k2 = list(net.parameters())[2]
        print(k2.shape)
        p = torchvision.utils.make_grid(k2, padding = 8, normalize = True)
        plt.imshow(np.transpose(p.to("cpu").detach().numpy(), (1, 2, 0)))

        torch.Size([16, 1, 97, 97])

Out[9]: <matplotlib.image.AxesImage at 0x7fa2fc0cea50>
```



```
In [10]: k3 = list(net.parameters())[3]
         print(k3.shape)
         p = torchvision.utils.make_grid(k3, padding = 16, normalize = True)
         plt.imshow(np.transpose(p.to("cpu").detach().numpy(), (1, 2, 0)))
```

```
torch.Size([16, 1, 193, 193])
```

```
Out[10]: <matplotlib.image.AxesImage at 0x7fa2f47f3250>
```



Read 4 images, make a batch from them and send them to device.
Or create a random batch of arbitrary size for computation performance evaluation.

```
In [11]: from PIL import Image
         import torchvision.transforms.functional as TF
         x = torch.cat((255*TF.to_tensor(Image.open('cat.jpg')).unsqueeze(0),
                        255*TF.to_tensor(Image.open('houses.jpg')).unsqueeze(0),
                        255*TF.to_tensor(Image.open('bur0.jpg')).unsqueeze(0),
                        255*TF.to_tensor(Image.open('bur1.jpg')).unsqueeze(0))).to(devic
         e)

         b = 4
         #x = torch.randn(b,1,256,256).to(device)
         print(x.shape)
```

```
torch.Size([4, 1, 256, 256])
```

Compute the number of floating point operations per image: one addition and one multiplication per output pixel, per kernel element for each filter and at all resolutions (cropping taken into account).

```
In [12]: nops = ((64.0*64*193*193)+(160.0*160*97*97)+(208.0*208*49*49)+(232*232*25*25))*1
         6.0*2.0
         print("Gflops/image: %.3f"% (nops/1000000000.0))
```

```
Gflops/image: 16.991
```

Run several times to get statistics on execution times. Can be tried on CPU and on GPU. First iterations are slower.

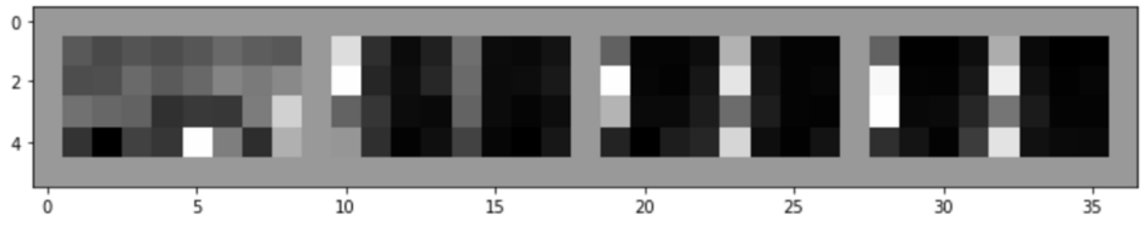
```
In [13]: tt = 0
         for i in range(20):
             if torch.cuda.is_available(): torch.cuda.synchronize()
             t0 = time.time()
             y = net(x)
             if torch.cuda.is_available(): torch.cuda.synchronize()
             dt = time.time()-t0
             print("iter %2d: %.2f ms, %.3f Tflops"% (i, dt*1000, b*nops/dt/100000000000
         0))
             if (i > 3): tt += dt

         dt = tt/16
         print("Average: %.2f ms, %.3f Tflops"% (dt*1000, b*nops/dt/1000000000000))
```

```
iter  0: 308.90 ms, 0.220 Tflops
iter  1: 40.45 ms, 1.680 Tflops
iter  2: 39.84 ms, 1.706 Tflops
iter  3: 38.81 ms, 1.751 Tflops
iter  4: 36.15 ms, 1.880 Tflops
iter  5: 35.87 ms, 1.895 Tflops
iter  6: 36.55 ms, 1.859 Tflops
iter  7: 36.93 ms, 1.840 Tflops
iter  8: 35.55 ms, 1.912 Tflops
iter  9: 34.01 ms, 1.998 Tflops
iter 10: 34.27 ms, 1.983 Tflops
iter 11: 34.13 ms, 1.991 Tflops
iter 12: 34.20 ms, 1.987 Tflops
iter 13: 31.57 ms, 2.152 Tflops
iter 14: 31.30 ms, 2.171 Tflops
iter 15: 32.47 ms, 2.093 Tflops
iter 16: 33.13 ms, 2.051 Tflops
iter 17: 33.23 ms, 2.045 Tflops
iter 18: 32.08 ms, 2.118 Tflops
iter 19: 31.05 ms, 2.189 Tflops
Average: 33.91 ms, 2.004 Tflops
```

Print and plot results. Should be consistent with the numpy / scipy version of TP2.

```
In [14]: print(y.shape)
         print(y)
         gy = torchvision.utils.make_grid(y.unsqueeze(1), padding = 1, normalize = True,
         scale_each = True, pad_value = 0.60)
         npy = np.transpose(gy.to("cpu").detach().numpy(), (1, 2, 0))
         plt.rcParams["figure.figsize"] = (12,6)
         plt.imshow(npy)
         print("\n\n cat, houses, bur0, bur1:")
```

```
torch.Size([4, 4, 8])
tensor([[[ 1.0065,  0.8843,  0.9673,  0.9153,  0.9879,  1.1373,  1.0379,
            0.9988],
          [ 0.9149,  0.9327,  1.1381,  1.0138,  1.1274,  1.3426,  1.2645,
            1.3810],
          [ 1.1976,  1.1202,  1.0789,  0.6856,  0.7596,  0.7403,  1.2825,
            1.9464],
          [ 0.7145,  0.3097,  0.8191,  0.7331,  2.3015,  1.2941,  0.6627,
            1.6894]],

         [[19.9091,  4.4179,  1.1953,  3.0932, 10.1122,  1.2978,  0.9888,
            1.8669],
          [22.8977,  3.5579,  1.4379,  3.6866,  9.6574,  1.2347,  1.3890,
            2.3957],
          [ 9.0294,  5.0205,  1.2536,  1.0404,  9.0134,  1.1716,  0.6645,
            1.3802],
          [13.6040,  4.4125,  0.5186,  1.5334,  6.1166,  0.7428,  0.1657,
            2.2692]],

         [[ 6.1645,  0.6255,  0.6519,  1.1414, 10.9845,  1.4242,  0.6019,
            0.5905],
          [15.5254,  0.6862,  0.5434,  1.6851, 14.0147,  1.6827,  0.5797,
            0.7974],
          [11.0596,  0.9496,  0.9229,  2.0302,  6.7708,  2.0791,  0.5636,
            0.5496],
          [ 2.4682,  0.3234,  2.1093,  2.6130, 13.1027,  1.1669,  0.4861,
            1.4716]],

         [[ 6.6406,  0.4883,  0.5232,  1.3305, 11.4115,  1.1014,  0.4217,
            0.5299],
          [16.1493,  0.6945,  0.6392,  1.9503, 15.4724,  1.5547,  0.5859,
            0.8016],
          [16.5331,  0.9758,  1.0721,  2.8559,  7.8222,  2.1446,  0.7047,
            0.7356],
          [ 3.4310,  1.7028,  0.6558,  4.2130, 14.7780,  1.5179,  1.0552,
            0.9988]]], device='cuda:0', grad_fn=<TransposeBackward0>)


 cat, houses, bur0, bur1:
```



Use another network to display the images tranformed by the Gabor filters (no global averaging, zero padding).

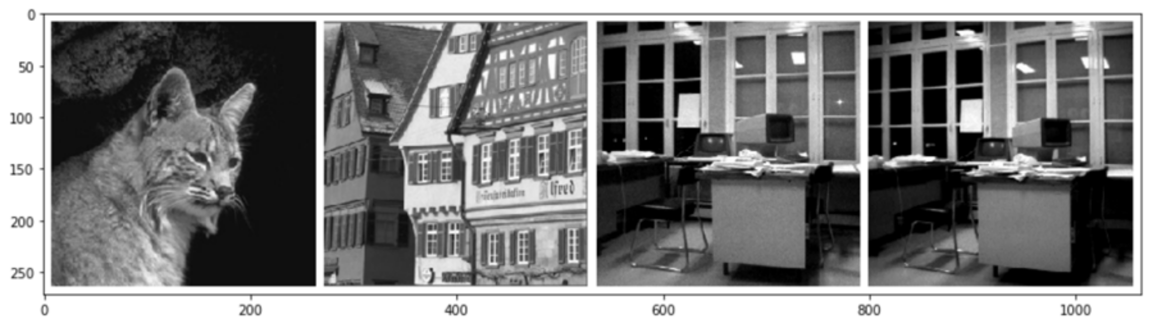
In [15]:
```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv0 = nn.Conv2d(1, 16, 25, bias=False, padding = 12)
        self.conv1 = nn.Conv2d(1, 16, 49, bias=False, padding = 24)
        self.conv2 = nn.Conv2d(1, 16, 97, bias=False, padding = 48)
        self.conv3 = nn.Conv2d(1, 16, 193, bias=False, padding = 96)

    def forward(self, x):
        x0 = (self.conv0(x)**2).unsqueeze(2)/1
        x1 = (self.conv1(x)**2).unsqueeze(2)/2
        x2 = (self.conv2(x)**2).unsqueeze(2)/4
        x3 = (self.conv3(x)**2).unsqueeze(2)/8
        y = torch.cat((x0, x1, x2, x3), dim = 2)
        y1, y2 = torch.split(y, 8, dim = 1)
        return((y1+y2).transpose(1,2))
```

In [16]:
```
net2 = Net2()
net2.to(device)
gabor_kernel_init(net2.conv0.weight,lambd = 3.0)
gabor_kernel_init(net2.conv1.weight,lambd = 6.0)
gabor_kernel_init(net2.conv2.weight,lambd = 12.0)
gabor_kernel_init(net2.conv3.weight,lambd = 24.0)
print(net2)
```

```
kernel init 25x25: 2.04 ms
kernel init 49x49: 2.29 ms
kernel init 97x97: 3.03 ms
kernel init 193x193: 6.63 ms
Net2(
  (conv0): Conv2d(1, 16, kernel_size=(25, 25), stride=(1, 1), padding=(12, 1
2), bias=False)
  (conv1): Conv2d(1, 16, kernel_size=(49, 49), stride=(1, 1), padding=(24, 2
4), bias=False)
  (conv2): Conv2d(1, 16, kernel_size=(97, 97), stride=(1, 1), padding=(48, 4
8), bias=False)
  (conv3): Conv2d(1, 16, kernel_size=(193, 193), stride=(1, 1), padding=(96, 9
6), bias=False)
)
```

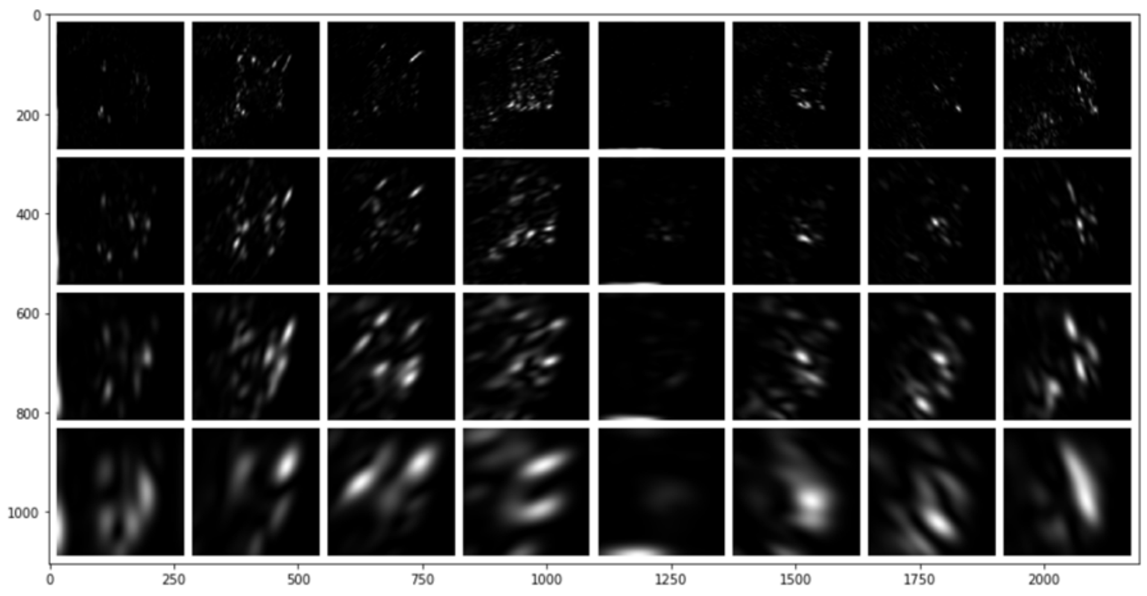
In [17]:
```
plt.rcParams["figure.figsize"] = (15,8)
gx = torchvision.utils.make_grid(x, padding = 8, normalize = True, pad_value =
1)
plt.imshow(np.transpose(gx.to("cpu").detach().numpy(), (1, 2, 0)))
y = net2(x)
print(y.shape)
z = torch.reshape(y, (4, 32, 1, 256, 256))
print(z.shape)
```

```
torch.Size([4, 4, 8, 256, 256])
torch.Size([4, 32, 1, 256, 256])
```



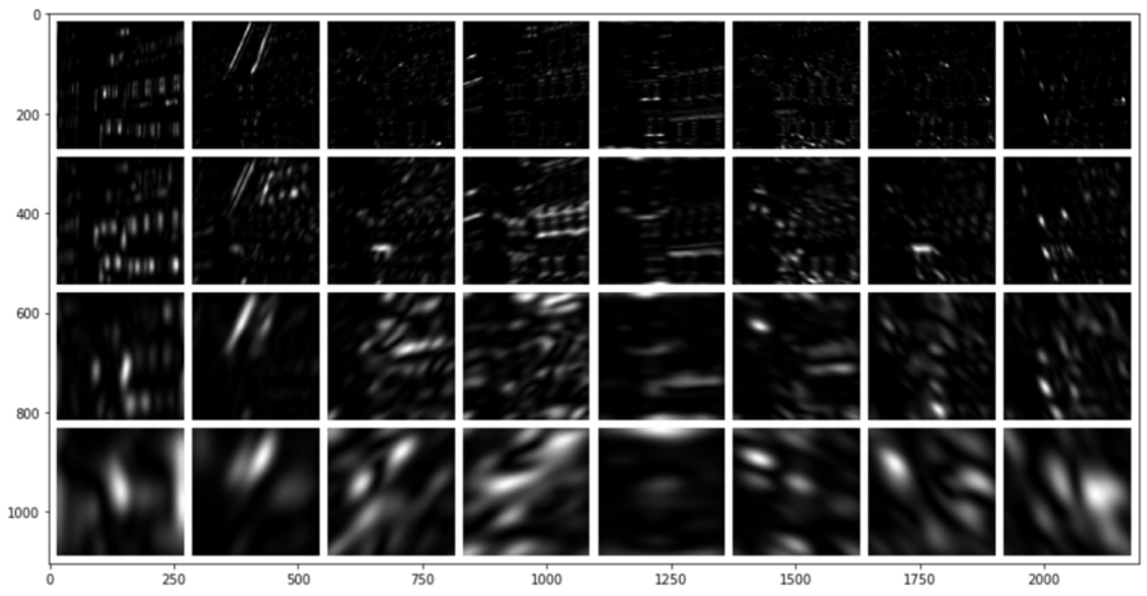
In [18]:
```python
# cat.jpg
gz = torchvision.utils.make_grid(z[0], padding = 16, normalize = True, scale_eac
h = True, pad_value = 1)
plt.imshow(np.transpose(gz.to("cpu").detach().numpy(), (1, 2, 0)))
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x7fa2f464cf90>
```



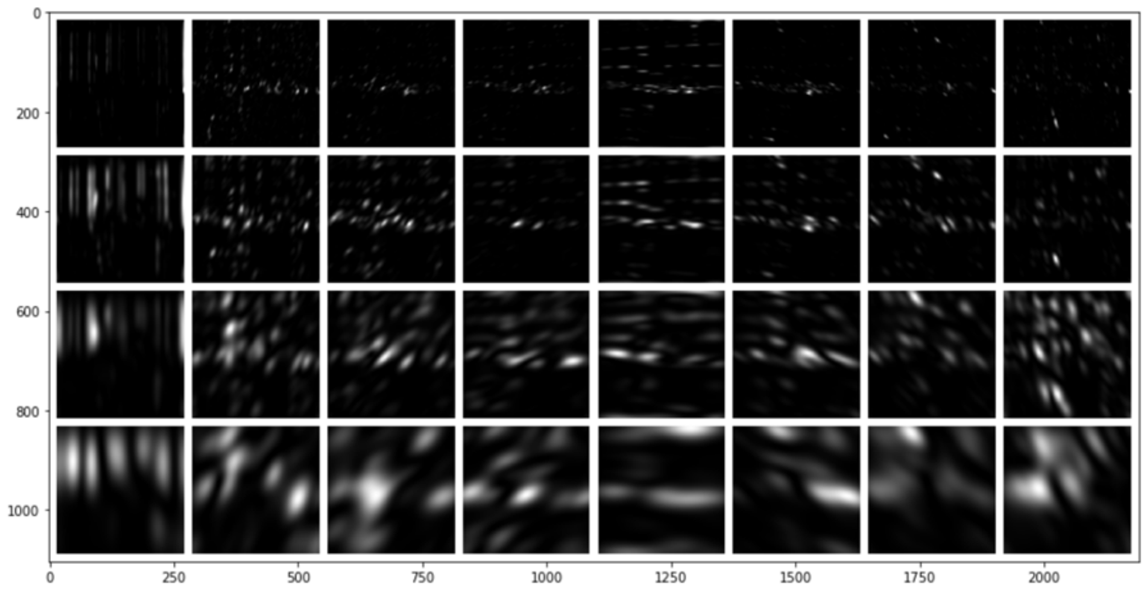
In [19]:
```python
# houses.jpg
gz = torchvision.utils.make_grid(z[1], padding = 16, normalize = True, scale_eac
h = True, pad_value = 1)
plt.imshow(np.transpose(gz.to("cpu").detach().numpy(), (1, 2, 0)))
```

Out[19]:
```
<matplotlib.image.AxesImage at 0x7fa2f45c2550>
```



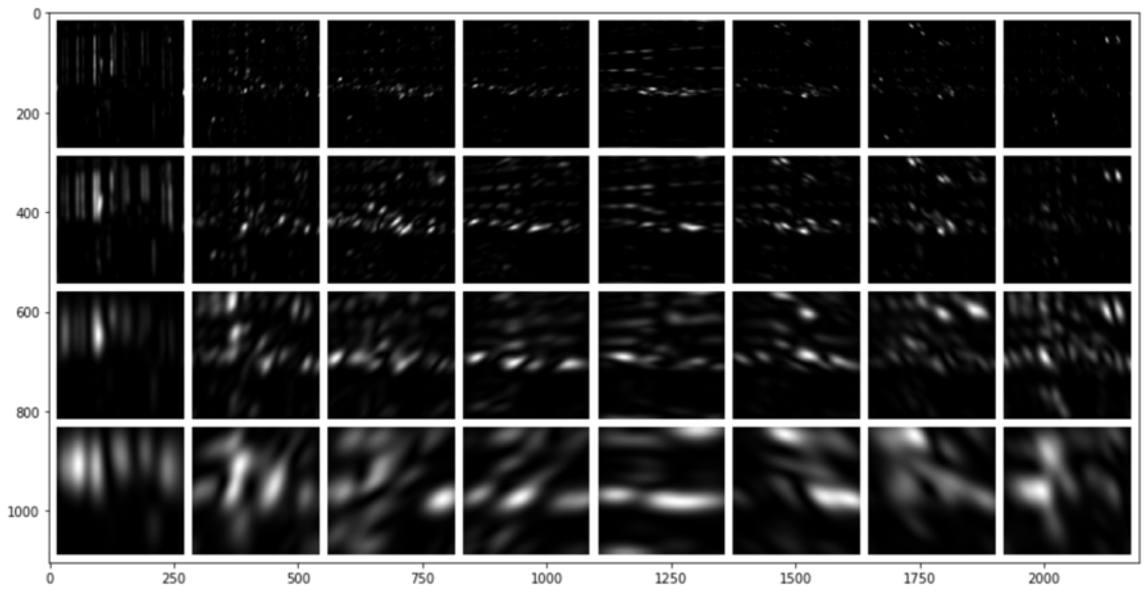
In [20]:
```
# bur0.jpg
gz = torchvision.utils.make_grid(z[2], padding = 16, normalize = True, scale_eac
h = True, pad_value = 1)
plt.imshow(np.transpose(gz.to("cpu").detach().numpy(), (1, 2, 0)))
```

Out[20]:
```
<matplotlib.image.AxesImage at 0x7fa2f45b0c50>
```



In [21]:
```
# bur1.jpg
gz = torchvision.utils.make_grid(z[3], padding = 16, normalize = True, scale_eac
h = True, pad_value = 1)
plt.imshow(np.transpose(gz.to("cpu").detach().numpy(), (1, 2, 0)))
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x7fa2f4529390>
```


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000