

A2:CNN Report

Boxi Chen

Student ID:010671402

CSCI611 - Applied Machine Learning Spring 2025

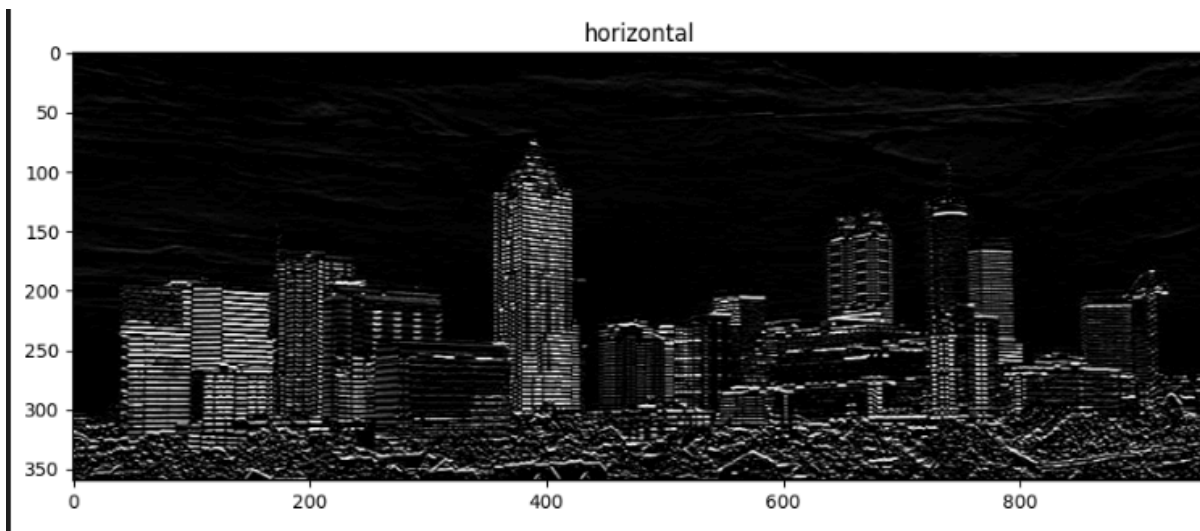
Professor:Prof. Bo Shen

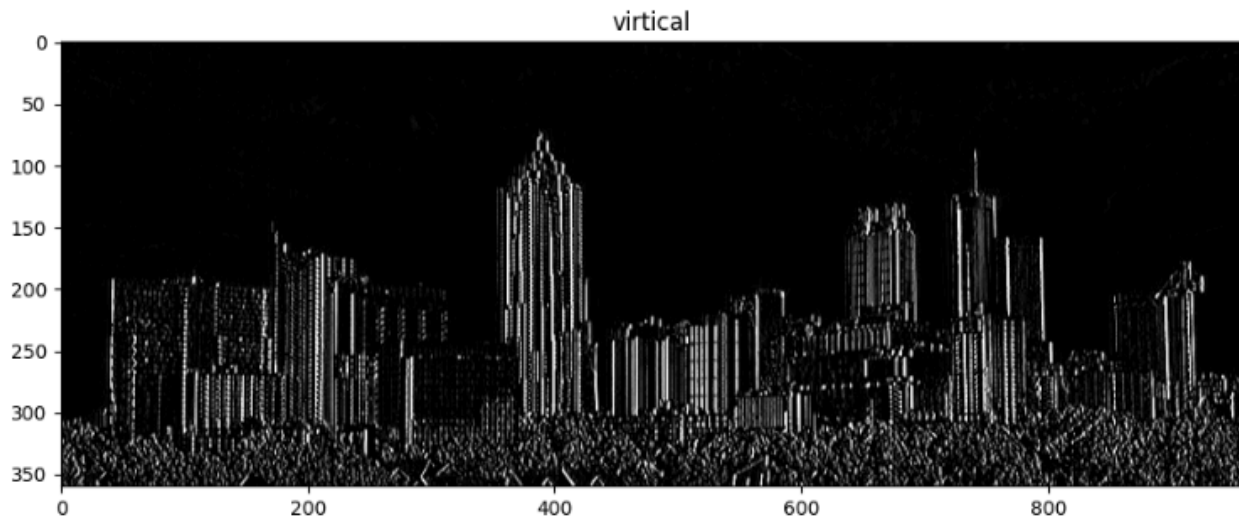
Department of Computer Science
California State University, Chico

Part 1 Image_Filter.zip:

```
## TODO: Create and apply a vertical edge detection operator
vertical = np.array([[ -1,  0,  1],
                    [ -1,  0,  1],
                    [ -1,  0,  1]])
```

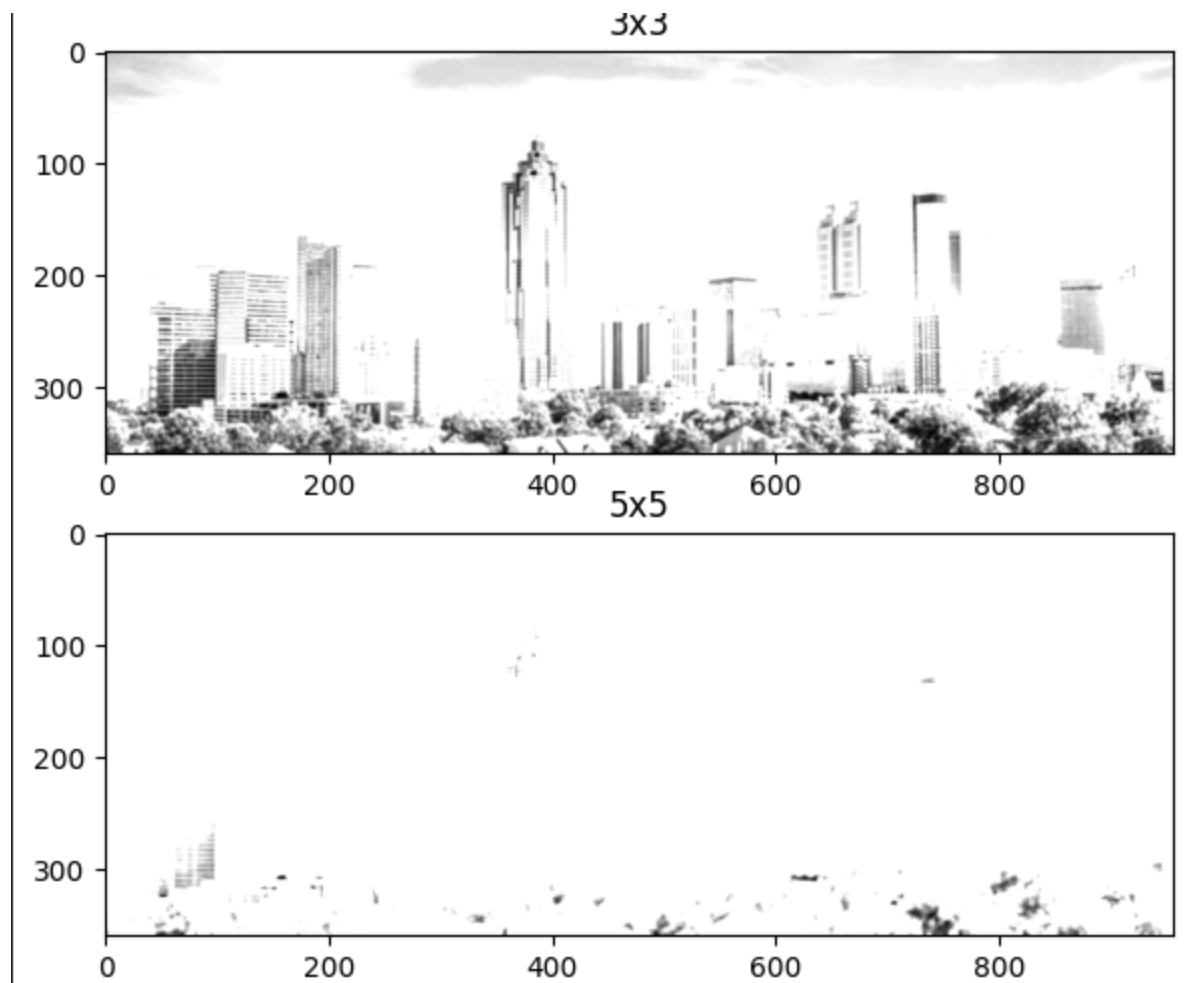
```
filtered_image2 = cv2.filter2D(gray, -1, vertical)
fig.add_subplot(2,2,2)
plt.imshow(filtered_image2, cmap='gray')
plt.title('vertical')
```





The answer of the code already given, we can compare the horizon filter, the picture is more focused on vertical lines/pixels.

```
# TODO: blur image using a 3x3 average
#fig.add_subplot(4,1,3)
#plt.title('3x3')
S3x3 = np.array([[ 1, 1, 1],
                  [ 1, 1, 1],
                  [ 1, 1, 1]])
blurred_image = cv2.filter2D(gray, -1, S3x3/4.0)
fig.add_subplot(4,1,3)
plt.imshow(blurred_image, cmap='gray')
plt.title('3x3')
# TODO: blur image using a 5x5 average
#fig.add_subplot(4,1,4)
#plt.title('5x5')
S5x5 = np.array([[ 1, 1, 1, 1, 1],
                  [ 1, 1, 1, 1, 1],
                  [ 1, 1, 1, 1, 1],
                  [ 1, 1, 1, 1, 1],
                  [ 1, 1, 1, 1, 1]])
blurred_image = cv2.filter2D(gray, -1, S5x5/4.0)
fig.add_subplot(4,1,4)
plt.imshow(blurred_image, cmap='gray')
plt.title('5x5')
plt.show()
```



Given the same value of 1 for different dimensions 3x3, 5x5 we can see different kernel filter impact to the picture, the bigger kernel filter the picture more blurred under all kernel filters divided by 4 for normalized weight.

TODO

Other image processing/filtering you can try:

- Other Edge Detector (e.g. Sobel Operator) A common 3×3 kernel for edge detection is the **Sobel operator**, which detects edges in a specific direction. Below are the Sobel kernels for detecting vertical and horizontal edges:

Vertical Edge Detection Kernel:

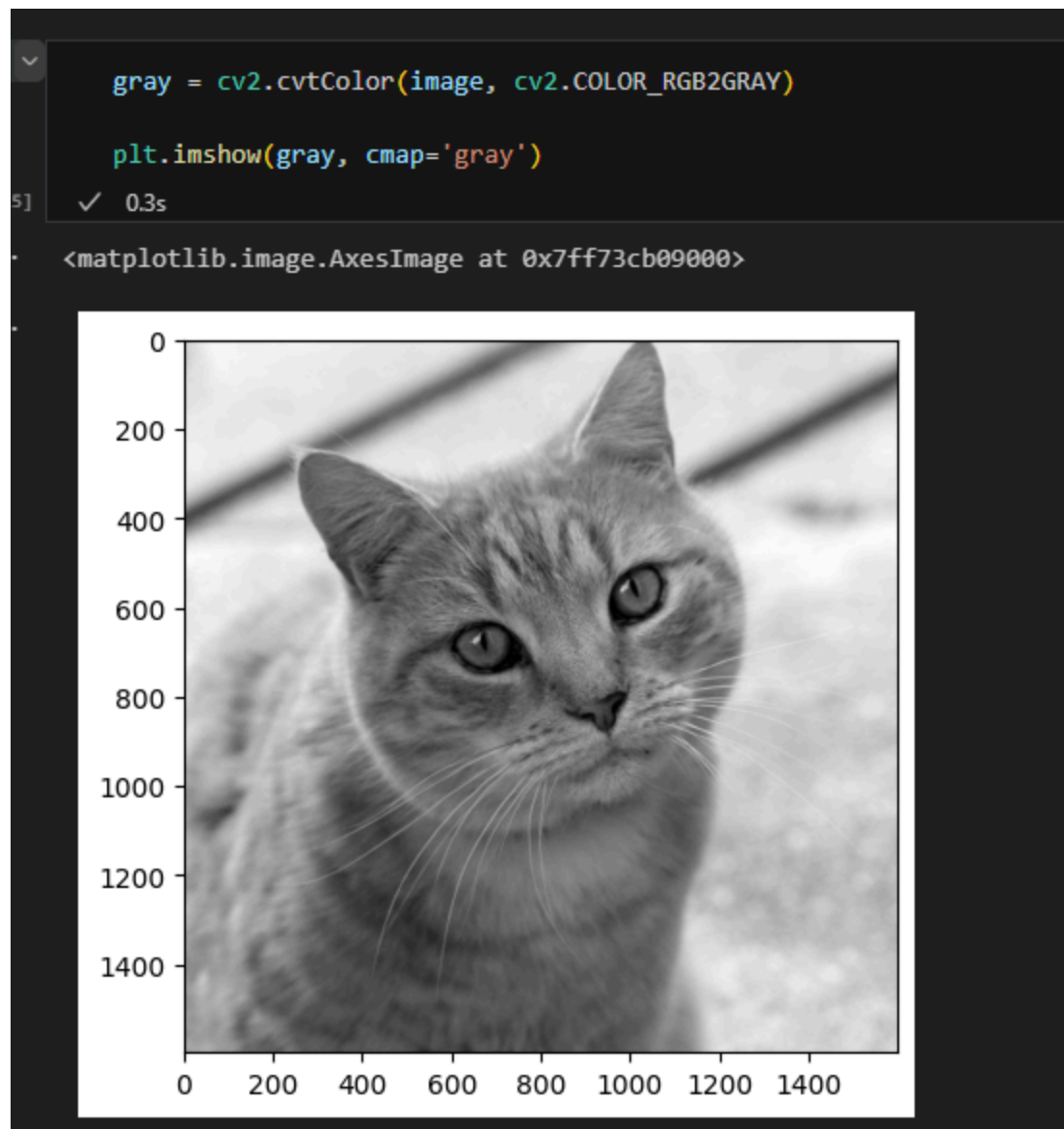
$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Horizontal Edge Detection Kernel:

$$K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

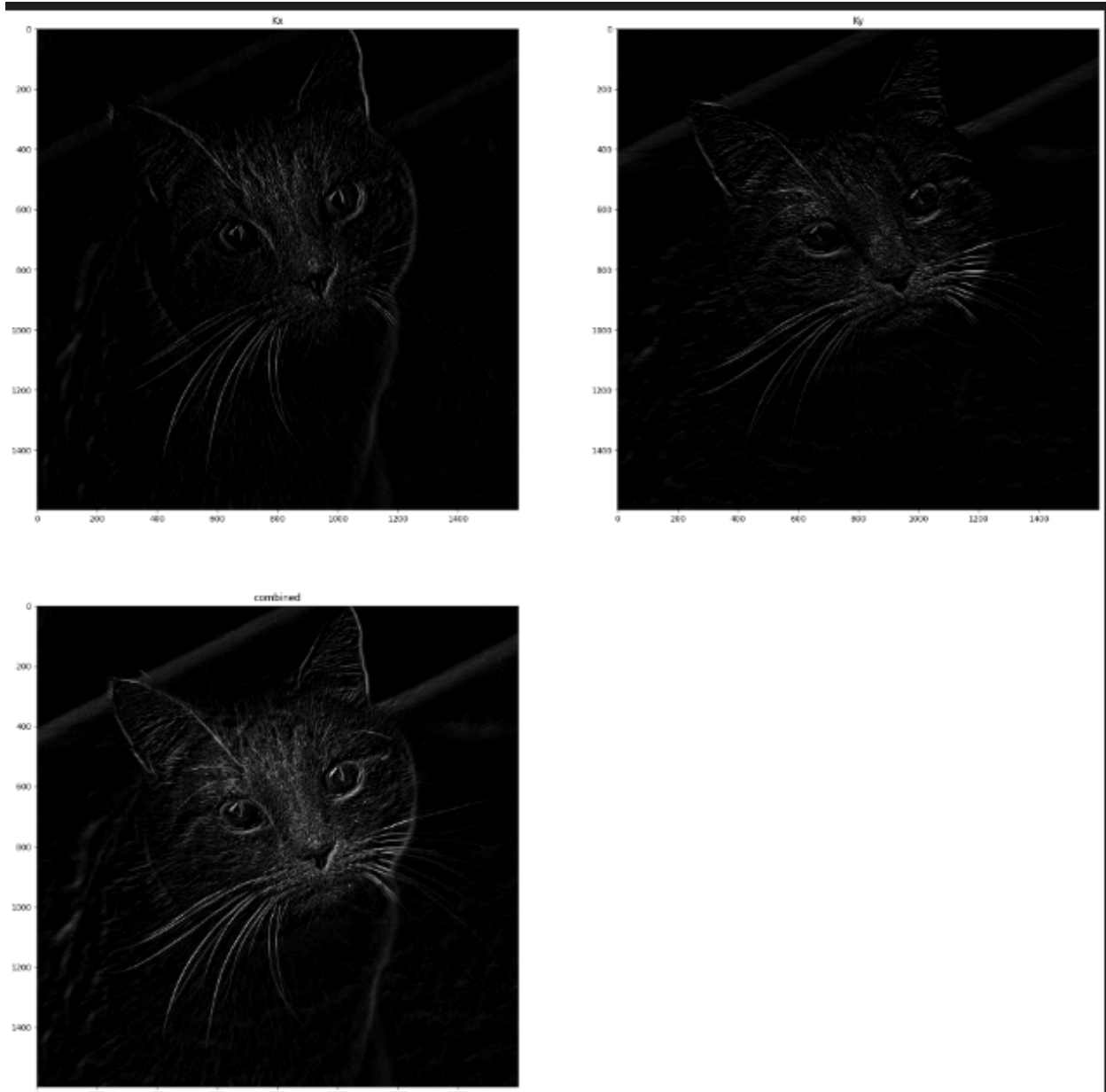
These kernels are convolved with an image to highlight vertical and horizontal edges, respectively.

- Corner Detection (use the kernels we discussed in slides)
- Scaling (after the blurring, can you pick one pixel out of the following?)
 - 2×2
 - 4×4
- Use other images of your choice
- For a challenge, see if you can put the image through a series of filters: first one that blurs the image (takes an average of pixels), and then one that detects the edges.



Loading the cat gray picture

```
✓ Kx = np.array([[ -1, 0, 1],  
                 [ -2, 0, 2],  
                 [ -1, 0, 1]])  
  
✓ Ky = np.array([[ -1, -2, -1],  
                 [ 0, 0, 0],  
                 [ 1, 2, 1]])  
  
fig = plt.figure(figsize=(24,24))  
  
filtered_image = cv2.filter2D(gray, -1, Kx)  
fig.add_subplot(2,2,1)  
plt.imshow(filtered_image, cmap='gray')  
plt.title('Kx')  
  
filtered_image2 = cv2.filter2D(gray, -1, Ky)  
fig.add_subplot(2,2,2)  
plt.imshow(filtered_image2, cmap='gray')  
plt.title('Ky')  
  
combined = cv2.addWeighted(filtered_image, 1, filtered_image2, 1, 0)  
fig.add_subplot(2,2,3)  
plt.imshow(combined, cmap='gray')  
plt.title('combined')
```



```
combined = cv2.addWeighted(filtered_image, 1, filtered_image2, 1, 0)
```

We can see differences between the Kx and Yx, Kx is more focus vertical and Yx is more focus horizontal, then when i set combined x and y weight to 1 and mix them together to get a clear picture.

```
fig = plt.figure(figsize=(24,24))
```

```
S2x2 = np.array([[ 1,  1],
                  [ 1,  1]])

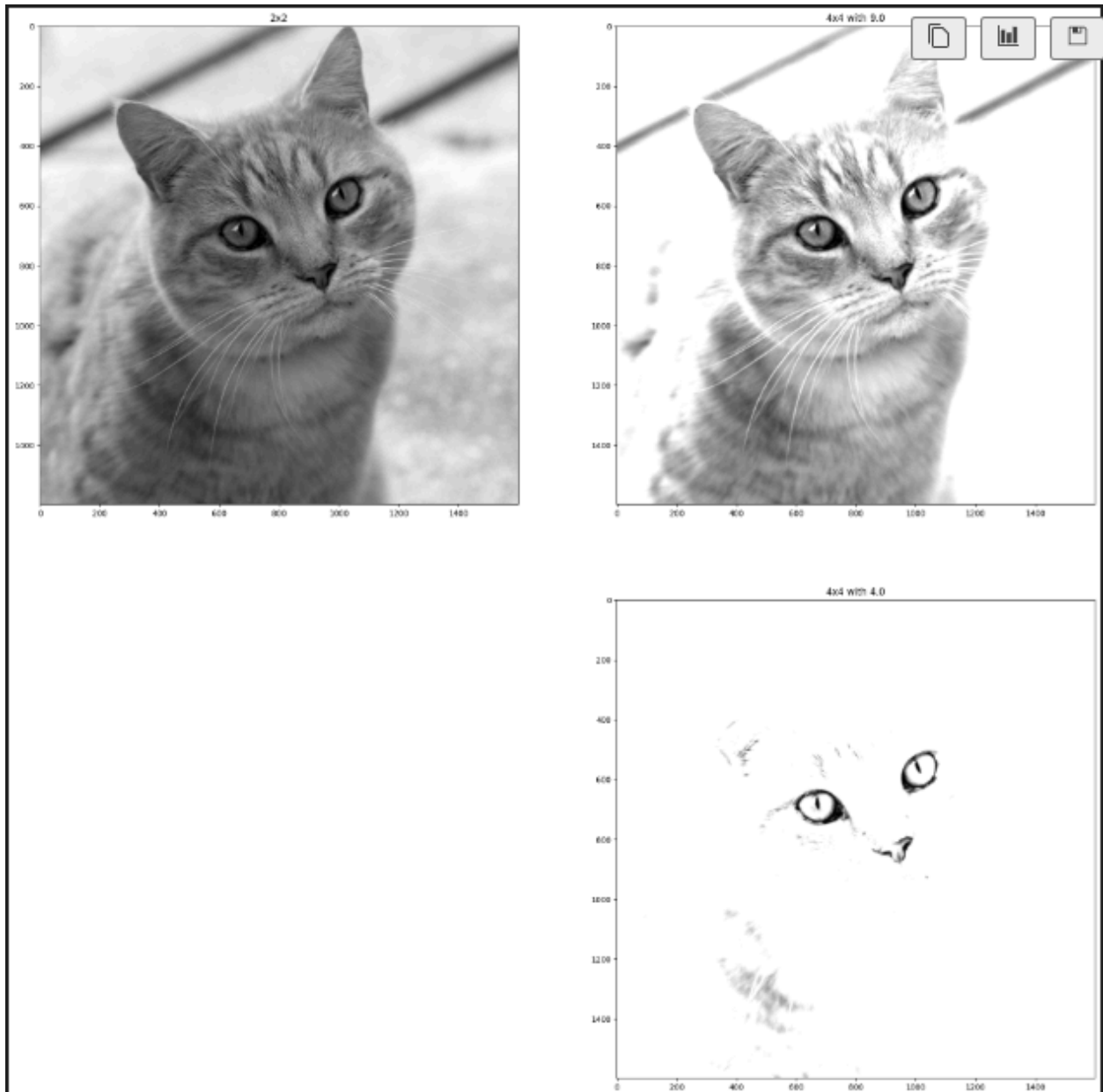
S4x4 = np.array([[ 1,  1,  1,  1],
                  [ 1,  1,  1,  1],
```

```
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]])

blurred_image = cv2.filter2D(gray, -1, S2x2/9.0)
fig.add_subplot(2,2,1)
plt.imshow(blurred_image, cmap='gray')
plt.title('2x2')

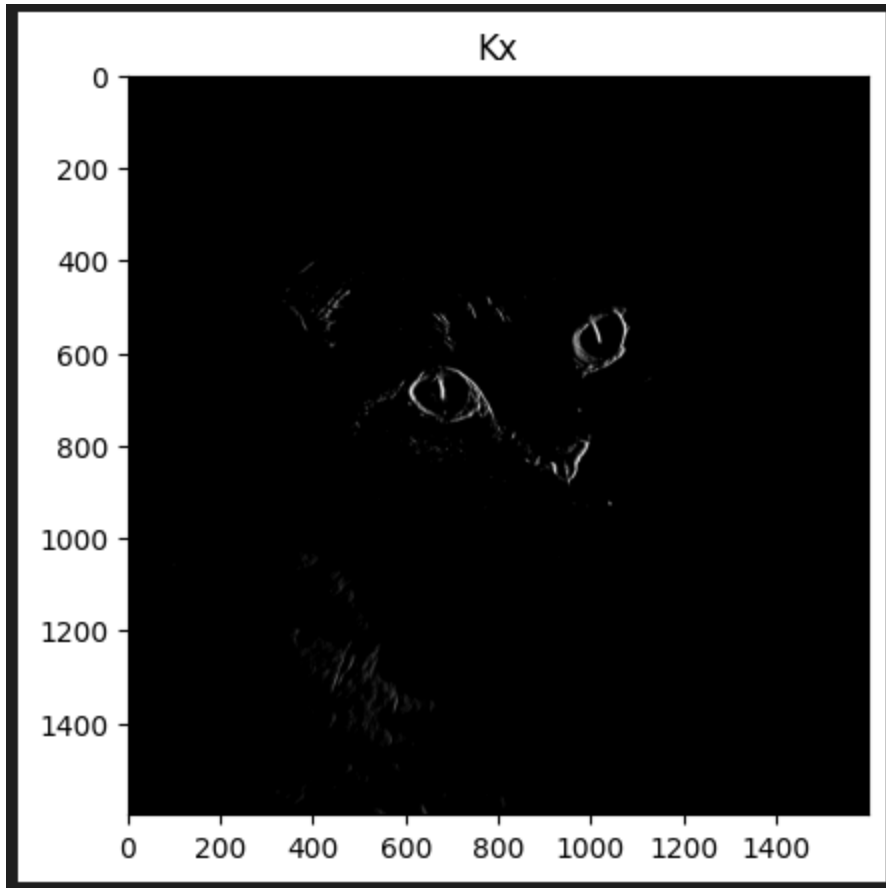
blurred_image = cv2.filter2D(gray, -1, S4x4/9.0)
fig.add_subplot(2,2,2)
plt.imshow(blurred_image, cmap='gray')
plt.title('4x4 with 9.0')

blurred_image = cv2.filter2D(gray, -1, S4x4/4.0)
fig.add_subplot(2,2,4)
plt.imshow(blurred_image, cmap='gray')
plt.title('4x4 with 4.0')
```

When we change the size of the kernel filter we can see the image is more blurred, more focus the detail, when size of filter weight/ 4, we can see the picture is much blurred than 9.0 which is more focus detail.

```
filtered_image = cv2.filter2D(blurred_image, -1, Kx)
plt.imshow(filtered_image, cmap='gray')
plt.title('Kx')
```



Using images that apply S4x4/4.0 to Kx edge detect, we can see there is not much difference due to the fact that all variables are detailed.

Part 2 Build_cnn.zip:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # TOTO: Build multiple convolutional layers (sees 32x32x3 image
        # tensor in the first hidden layer)
        # for example, conv1, conv2 and conv3
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32,
kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=16,
kernel_size=3, stride=2, padding=0)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, stride=1, padding=0)

        # max pooling layer
```

```

self.pool = nn.MaxPool2d(2, 2)

# TODO: Build some linear layers (fully connected)
# for example, fc1 and fc2
self.fc1 = nn.Linear(32 * 1 * 1, 100)
self.fc2 = nn.Linear(100, 10)

# TODO: dropout layer (p=0.25, you can adjust)
# example self.dropout = nn.Dropout(0.25)
self.dropout = nn.Dropout(0.25)

def forward(self, x):
    # add sequence of convolutional and max pooling layers
    # assume we have 2 convolutional layers defined above
    # and we do a maxpooling after each conv layer
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = F.relu(self.conv3(x))

    #print(f"Shape before flatten: {x.shape}")
    # TODO: flatten x at this point to get it ready to feed into the
fully connected layer(s)
    # Can use this but need to figure out the actual value for a, b
and c
    # x = x.view(-1, a * b * c)
    x = x.view(-1, 32 * 1 * 1)
    #print(f"batch_size: {x.size(0)}")

    #x = x.view(x.size(0), -1)
    # optional add dropout layer
    x = self.dropout(x)

    # add 1st hidden layer, with relu activation function
    x = F.relu(self.fc1(x))
    # optional add dropout layer
    x = self.dropout(x)
    # add 2nd hidden layer, with relu activation function
    x = self.fc2(x)
    return x

```

$$H_{\text{out}} = \frac{H_{\text{in}} + 2P - K}{S} + 1$$

We have convolutional layer conv1 that input 3 channel sense RGB picture then output is 32 channel with 32*32 pixel picture.

$$(32+2(1)-3) / 1 + 1 = 32$$

We have convolutional layer conv2 need 32 channel which from conv1 and since padding and stride is not perfect, so we lost some pixel so picture which 15*15

$$(32+2(0)-3) / 2 + 1 = 15$$

Third convolutional layer con3 need 16 which from con2 and we lost some pixel because padding and stride, so it become 13*13

$$(15+2(0)-3) / 1 + 1 = 13$$

Then we have maxpool 2*2, so we eventually have 6*6 pixel picture

$$13 / 2 = 6.5 = 6$$

```
self.fc1 = nn.Linear(32 * 1 * 1, 100)
```

But reason why we using Linear 32*1*1 because of our forward function first two time call conv1 then will apply maxpool, so it cut the picture to half every time so math will change to below

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = F.relu(self.conv3(x))
```

$$(32+2(1)-3) / 1 + 1 = 32$$

$$32 / 2 = 16$$

$$(16+2(0)-3) / 2 + 1 = 7$$

$$7 / 2 = 3.5 = 3$$

$$(3+2(0)-3) / 1 + 1 = 1$$

So eventually we had 32 channel of 1*1 picture then we combine to train the model

```
Epoch: 1      Training Loss: 2.225502      Validation Loss: 2.031201
Validation loss decreased (inf --> 2.031201). Saving model ...
Epoch: 2      Training Loss: 1.977334      Validation Loss: 1.814740
Validation loss decreased (2.031201 --> 1.814740). Saving model ...
Epoch: 3      Training Loss: 1.802740      Validation Loss: 1.668203
Validation loss decreased (1.814740 --> 1.668203). Saving model ...
Epoch: 4      Training Loss: 1.695071      Validation Loss: 1.568709
Validation loss decreased (1.668203 --> 1.568709). Saving model ...
Epoch: 5      Training Loss: 1.623666      Validation Loss: 1.504402
Validation loss decreased (1.568709 --> 1.504402). Saving model ...
```

```
Test Loss: 1.495284
```

```
Test Accuracy of airplane: 43% (438/1000)
Test Accuracy of automobile: 60% (601/1000)
Test Accuracy of bird: 10% (100/1000)
Test Accuracy of cat: 5% (56/1000)
Test Accuracy of deer: 42% (425/1000)
Test Accuracy of dog: 61% (614/1000)
Test Accuracy of frog: 64% (641/1000)
Test Accuracy of horse: 53% (537/1000)
Test Accuracy of ship: 58% (585/1000)
Test Accuracy of truck: 51% (516/1000)
```

```
Test Accuracy (Overall): 45% (4513/10000)
```

We can see test loss is 1.5 and test accuracy is 45% using SGD with lr 0.01



Which is okay, but not really good.

Switch to Adam with lr 0.01 using code :

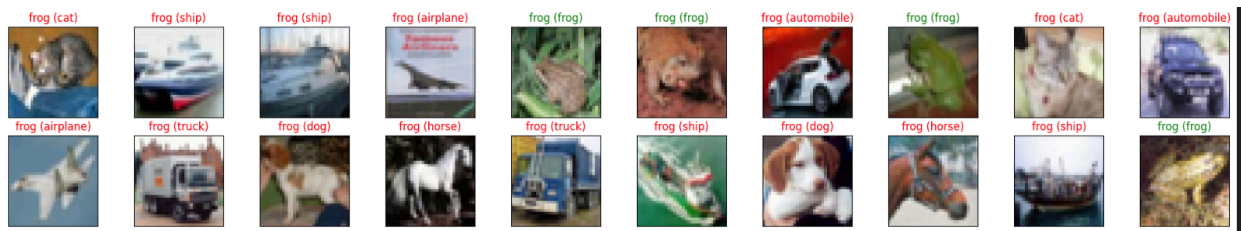
```
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

```
Epoch: 1      Training Loss: 2.304510      Validation Loss: 2.305003
Validation loss decreased (inf --> 2.305003). Saving model ...
Epoch: 2      Training Loss: 2.304250      Validation Loss: 2.304262
Validation loss decreased (2.305003 --> 2.304262). Saving model ...
Epoch: 3      Training Loss: 2.304451      Validation Loss: 2.303670
Validation loss decreased (2.304262 --> 2.303670). Saving model ...
Epoch: 4      Training Loss: 2.304320      Validation Loss: 2.304309
Epoch: 5      Training Loss: 2.304205      Validation Loss: 2.303130
Validation loss decreased (2.303670 --> 2.303130). Saving model ...
```

Test Loss: 2.303204

Test Accuracy of airplane: 0% (0/1000)
Test Accuracy of automobile: 0% (0/1000)
Test Accuracy of bird: 0% (0/1000)
Test Accuracy of cat: 0% (0/1000)
Test Accuracy of deer: 0% (0/1000)
Test Accuracy of dog: 0% (0/1000)
Test Accuracy of frog: 100% (1000/1000)
Test Accuracy of horse: 0% (0/1000)
Test Accuracy of ship: 0% (0/1000)
Test Accuracy of truck: 0% (0/1000)

Test Accuracy (Overall): 10% (1000/10000)



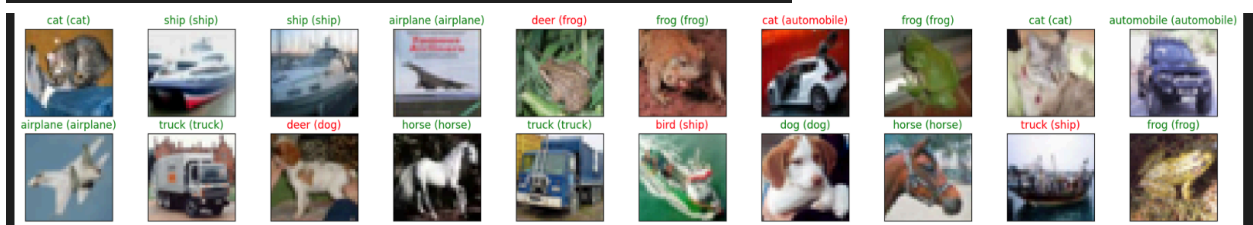
For some reason it guesses all pictures as fog, and loss is 2.3, accuracy is 10% which is really bad, maybe Adam optimizer asking smaller lr than SGD causes this problem. So I used ADAM with lr with 0.001

```
Epoch: 1      Training Loss: 1.794639      Validation Loss: 1.559116
Validation loss decreased (inf --> 1.559116). Saving model ...
Epoch: 2      Training Loss: 1.561987      Validation Loss: 1.435943
Validation loss decreased (1.559116 --> 1.435943). Saving model ...
Epoch: 3      Training Loss: 1.479734      Validation Loss: 1.388820
Validation loss decreased (1.435943 --> 1.388820). Saving model ...
Epoch: 4      Training Loss: 1.425180      Validation Loss: 1.327199
Validation loss decreased (1.388820 --> 1.327199). Saving model ...
Epoch: 5      Training Loss: 1.384403      Validation Loss: 1.302884
Validation loss decreased (1.327199 --> 1.302884). Saving model ...
```

Test Loss: 1.301875

Test Accuracy of airplane: 66% (669/1000)
Test Accuracy of automobile: 65% (655/1000)
Test Accuracy of bird: 21% (210/1000)
Test Accuracy of cat: 32% (325/1000)
Test Accuracy of deer: 42% (423/1000)
Test Accuracy of dog: 58% (585/1000)
Test Accuracy of frog: 62% (629/1000)
Test Accuracy of horse: 64% (641/1000)
Test Accuracy of ship: 52% (523/1000)
Test Accuracy of truck: 64% (645/1000)

Test Accuracy (Overall): 53% (5305/10000)



We can see the result is much better, loss is 1.3, accuracy is 53% much better.

For summary of CNN file section, I think modify of hidden layer will help improve of accuracy, also adjust of learning step also improve accuracy, the reason why we having such low accuracy i think we used too much max layer in 16*16 picture which make it really small in the end which is 1*1.