ASSIGNMENT 2: CNN REPORT

Author: Shambhavi Danayak
Professor: Bo Shen
Student ID: 012654513
Submission date: 02/21/2025

Image Processing using OpenCV

Jupyter Notebook:

Note: For this assignment a lot of research was conducted using OpenCV documentation, Pytorch documentation and reused code from coursebook (O'Reilly Machine learning with pytorch and Scikit-learn).

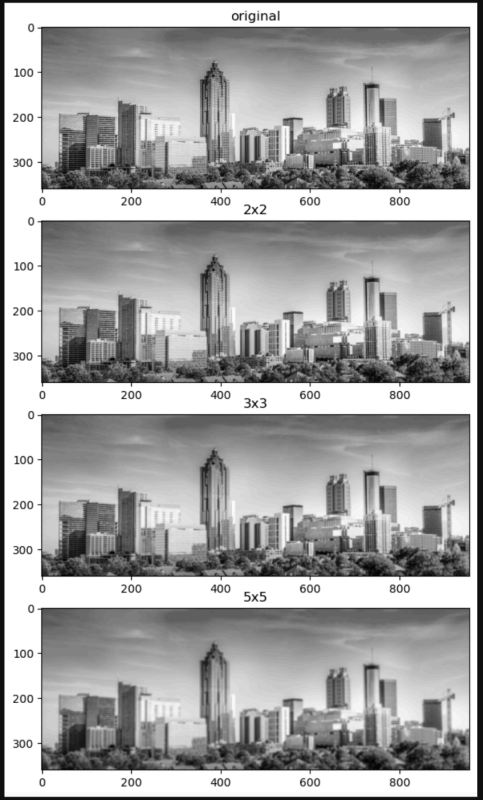
Image_filtering.ipynb starts with Installing OpenCv and importing necessary libraries such as matplotlib, cv2 etc. Then we read and display the building image ('building2.jpg'). The aim of this notebook is to utilize convolution and OpenCV for image filtering such as blurring, edge detection, corner detection etc.

Original Image:

TODO 1:
 # TODO: blur image using a 3x3 average
 # TODO: blur image using a 5x5 average

```
[6]: # Create a custom kernel for blurring

     # 2x2 kernel for averaging blurring
     S2x2 = np.array([[ 1, 1],
                      [ 1, 1]])

     # 3x3 kernel
     S3x3 = np.array([[1, 1, 1],
                      [1, 1, 1],
                      [1, 1, 1]])
     # 5x5 kernel
     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]])


     fig = plt.figure(figsize=(48, 12))
     fig.add_subplot(4,1,1)
     plt.imshow(gray, cmap='gray')
     plt.title('original')

     # Filter the image using filter2D, which has inputs: (grayscale image, bit-depth, kernel)
     #divide by 4 to normalize, ensures kernel elements equals thereby maintains the overall brightness
     blurred_image = cv2.filter2D(gray, -1, S2x2/4.0)
     fig.add_subplot(4,1,2)
     plt.imshow(blurred_image, cmap='gray')
     plt.title('2x2')

     # TODO: blur image using a 3x3 average
     blurred_image_3x3 = cv2.filter2D(gray, -1, S3x3/9.0)
     fig.add_subplot(4,1,3)
     plt.imshow(blurred_image_3x3, cmap='gray')
     plt.title('3x3')

     # TODO: blur image using a 5x5 average
     blurred_image_5x5 = cv2.filter2D(gray, -1, S5x5/25.0)
     fig.add_subplot(4,1,4)
     plt.imshow(blurred_image_5x5, cmap='gray')
     plt.title('5x5')

     plt.show()
```

```
plt.show()
```



Code explanation:

To implement blurring of images using a 3x3 kernel, I used the opencv function cv2.filter2d(). This function Convolves an image with the kernel. Below are the function parameters,

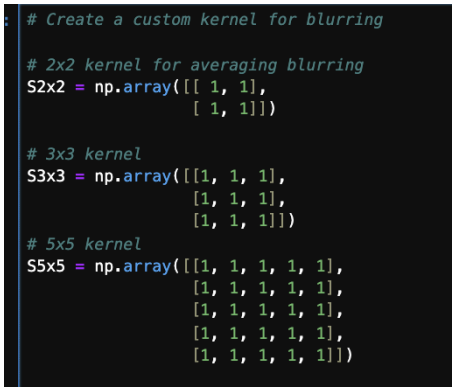
```
filter2D(src, dst, ddepth, kernel, anchor= Point(-1,-1), delta=0,
borderType= BORDER_DEFAULT)
```

Parameters

Src: input image
Dst: output image of the same size and the same number of channels as src
Depth: depth of destination image
Kernel: convolution kernel (like I defined a 3x3, 5x5, custom kernel)
Anchor: relative position of a filtered point within the kernel.
Delta: optional values for filtered pixels before saving
Bordertype: pixel extrapolation

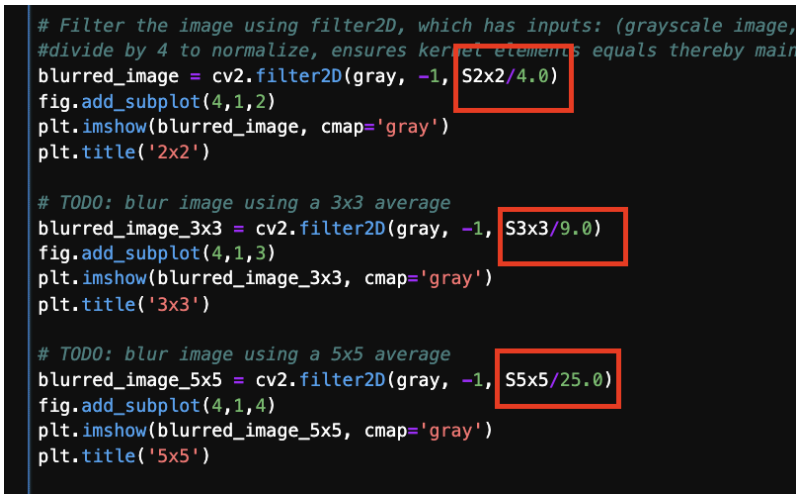
Overall steps to blurring of the image using a custom 2x2, 3x3 and 5x5 kernels:

1. Define custom kernels: simply define a 2x2, 3x3 and 5x5 matrix with equal weight to each pixel ensuring each pixel gets averaged equally.



```
# Create a custom kernel for blurring

# 2x2 kernel for averaging blurring
S2x2 = np.array([[ 1, 1],
                 [ 1, 1]])

# 3x3 kernel
S3x3 = np.array([[1, 1, 1],
                 [1, 1, 1],
                 [1, 1, 1]])
# 5x5 kernel
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]])
```

2. Apply each kernel using filter2D and normalize by dividing each kernel with the sum of all values in each kernel to keep the brightness unchanged.



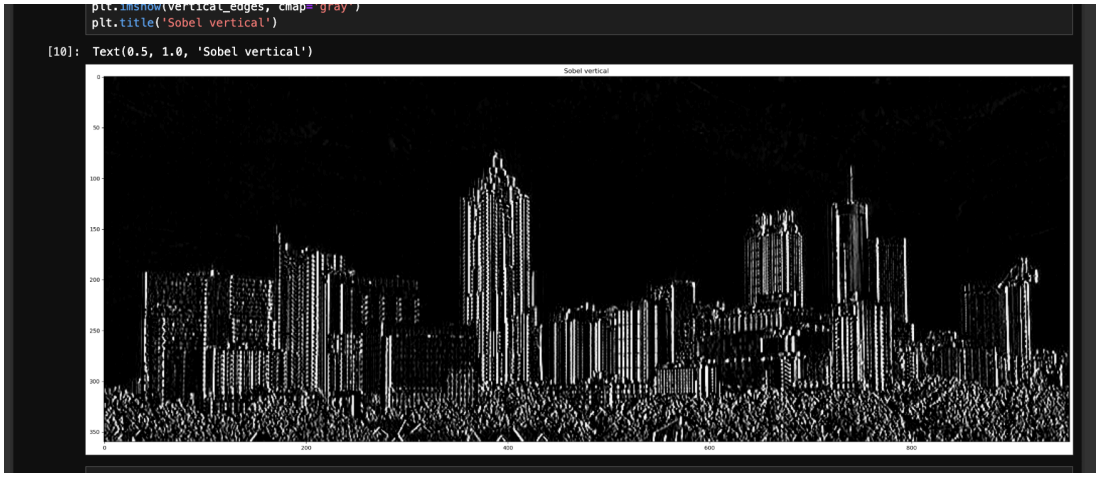
```
# Filter the image using filter2D, which has inputs: (grayscale image,
#divide by 4 to normalize, ensures kernel elements equals thereby main
blurred_image = cv2.filter2D(gray, -1, S2x2/4.0)
fig.add_subplot(4,1,2)
plt.imshow(blurred_image, cmap='gray')
plt.title('2x2')

# TODO: blur image using a 3x3 average
blurred_image_3x3 = cv2.filter2D(gray, -1, S3x3/9.0)
fig.add_subplot(4,1,3)
plt.imshow(blurred_image_3x3, cmap='gray')
plt.title('3x3')

# TODO: blur image using a 5x5 average
blurred_image_5x5 = cv2.filter2D(gray, -1, S5x5/25.0)
fig.add_subplot(4,1,4)
plt.imshow(blurred_image_5x5, cmap='gray')
plt.title('5x5')
```

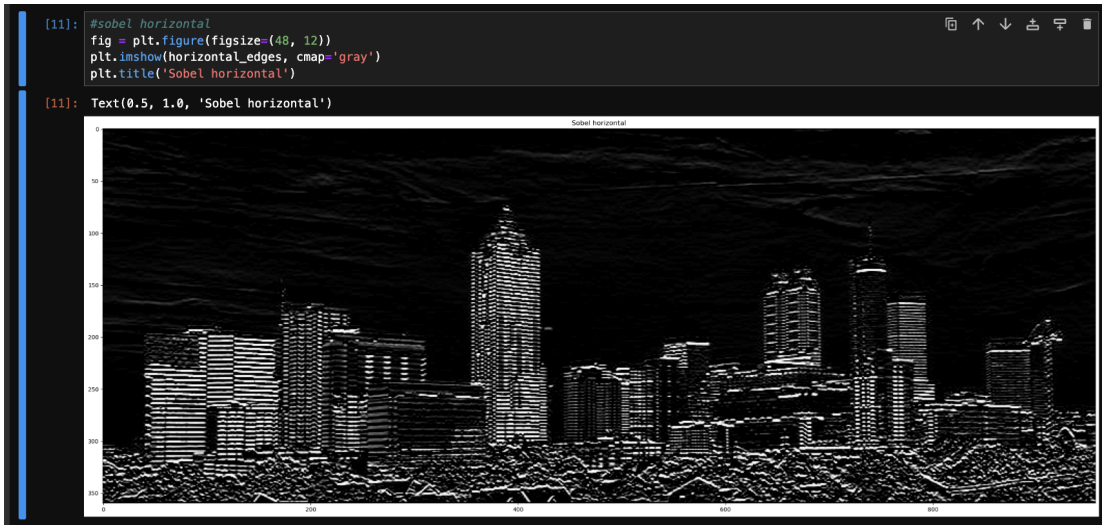
3. Finally display each blurred image using matplotlib

TODO 2:

- **Horizontal and vertical edge detection using Sobel Operator**

RESULTING IMAGES:

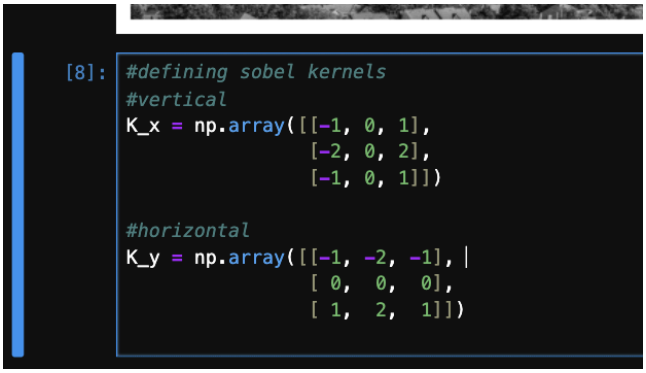




Sobel Operator is widely used for edge detection by calculating gradient of pixel intensity in an image to highlight the corner.

Steps:

1. Convert image to grayscale for easier image processing
2. Define Sobel Kernels

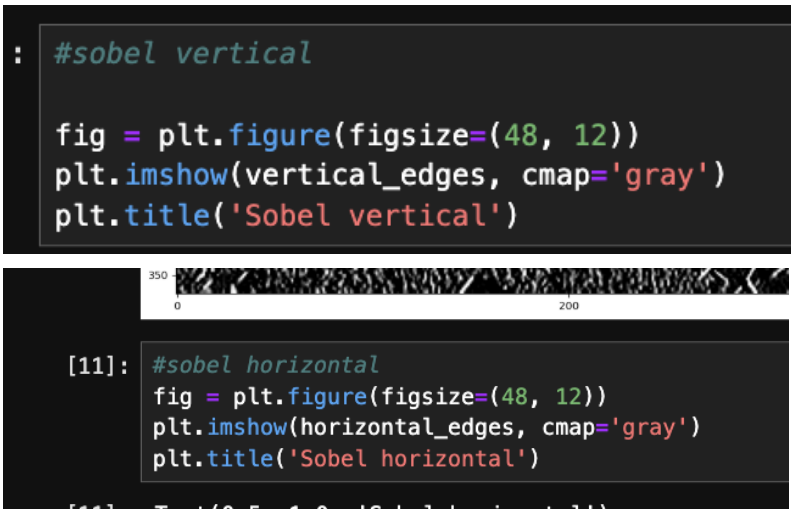


```
[8]: #defining sobel kernels
     #vertical
     K_x = np.array([[-1, 0, 1],
                     [-2, 0, 2],
                     [-1, 0, 1]])

     #horizontal
     K_y = np.array([[-1, -2, -1],
                     [ 0,  0,  0],
                     [ 1,  2,  1]])
```

3. Use cv2.filter2D() to apply the sobel vertical and horizontal kernels

```
[9]: #now that sobel is defined, i apply edge detection
     #vertical detection
     vertical_edges = cv2.filter2D(gray, -1, K_x)
     horizontal_edges = cv2.filter2D(gray, -1, K_y)
```

4. Finally plot the resulting filtered image highlighting vertical and horizontal edges using matplotlib



```
:  #sobel vertical

   fig = plt.figure(figsize=(48, 12))
   plt.imshow(vertical_edges, cmap='gray')
   plt.title('Sobel vertical')
```



```
[11]: #sobel horizontal
      fig = plt.figure(figsize=(48, 12))
      plt.imshow(horizontal_edges, cmap='gray')
      plt.title('Sobel horizontal')
```

- **Corner Detection (use the kernels discussed in the slides)**
  For this todo there are multiple ways we can perform edge using custom kernels (eg. ones from lecture slides), using inbuilt functions for corner detection provided by OpenCV such as Corner Harris, Shi-Tomasi, etc.

  Using CornerHarris:
  Documentation: https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html
  The function runs Harris Corner Detector on the image. Basically, for each pixel it calculates a 2x2 gradient covariance matrix (M) over a given block size (neighborhood

size) and uses it to determine the corner response. The final corner determination is calculated using Harris corner function.

```
[104]: # #using cv2.cornerHarris

dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)
dst = cv2.dilate(dst, None)

gray = cv2.cvtColor(np.uint8(gray), cv2.COLOR_GRAY2RGB)

# Mark detected corners in red
gray[dst > 0.01 * dst.max()] = [255, 0, 0]  # Red color in (B, G, R)

# Display the grayscale image with red corners
plt.imshow(gray)
plt.axis('off')
plt.show()
```

Steps:

1. Turn the image to grayscale
2. Use cv2.cornerHarris() for edge detection:
   Parameters: src, dst, block size, ksize, k, borderType
3. Then I dilated the destination(dst) using cv2.dilate() to expand the bright region in dst ensuring the visibility of detected corners.
4. Now since we want the image to remain grayscale and corner be highlighted as red dots, I converted the grayscale image to a 3-channel image so RGB value (detected red corners) can be added to the display. Without this line of code if you try to overlay red dots on your grayscale image the code will error out saying unable to convert a gray image which is a single-channel image to RGB a 3-channel image.

Using Kernels:

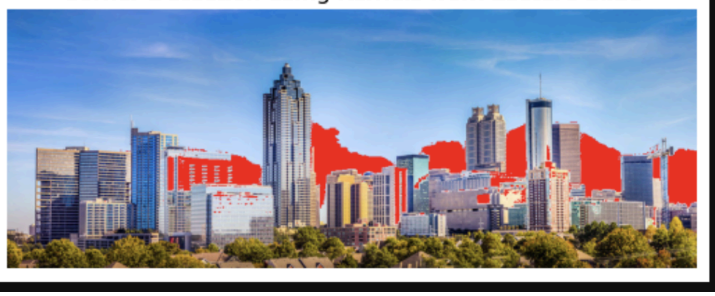
Not a standard way of implementing corner detection as can be very limited in case of complicated images.

Here I defined 4 custom kernels to detect Upper-left, Upper- right, Bottom-left and Bottom-right. Each kernel highlights a different type of corner in an image, then is independently applied and the maximum response is taken from all the four kernels.

Here I did not change (for learning purpose) the image to a 3-channel grayscale rather used colored image for final display of red detected corners.

```
[112]: #defining kernels
kernels = [
    np.array([[1,  1,  0],
              [0,  1,  1],
              [0,  0,  0]], dtype=np.float32),

    np.array([[1,  1,  0],
              [1,  1,  0],
              [0,  0,  0]], dtype=np.float32),

    np.array([[0,  0,  0],
              [1,  1,  0],
              [0,  1,  1]], dtype=np.float32),

    np.array([[0,  0,  0],
              [0,  1,  1],
              [1,  1,  1]], dtype=np.float32),
]

corner_response = np.zeros_like(gray)

# filter2d for each kernel
for kernel in kernels:
    response = cv2.filter2D(gray, -1, kernel)
    corner_response = np.maximum(corner_response, response)

#thershold for string corner detection
threshold = 0.9 * corner_response.max()
corners = (corner_response > threshold).astype(np.uint8) * 255

image[corners > 0] = [0, 0, 255]
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Corner Detection using Kernels from Lecture Slide")
plt.show()
```

```
plt.show()
```



- **Scaling (after the blurring, can you pick one pixel out of the following?)**
    - **2x2**
    - **4x4**

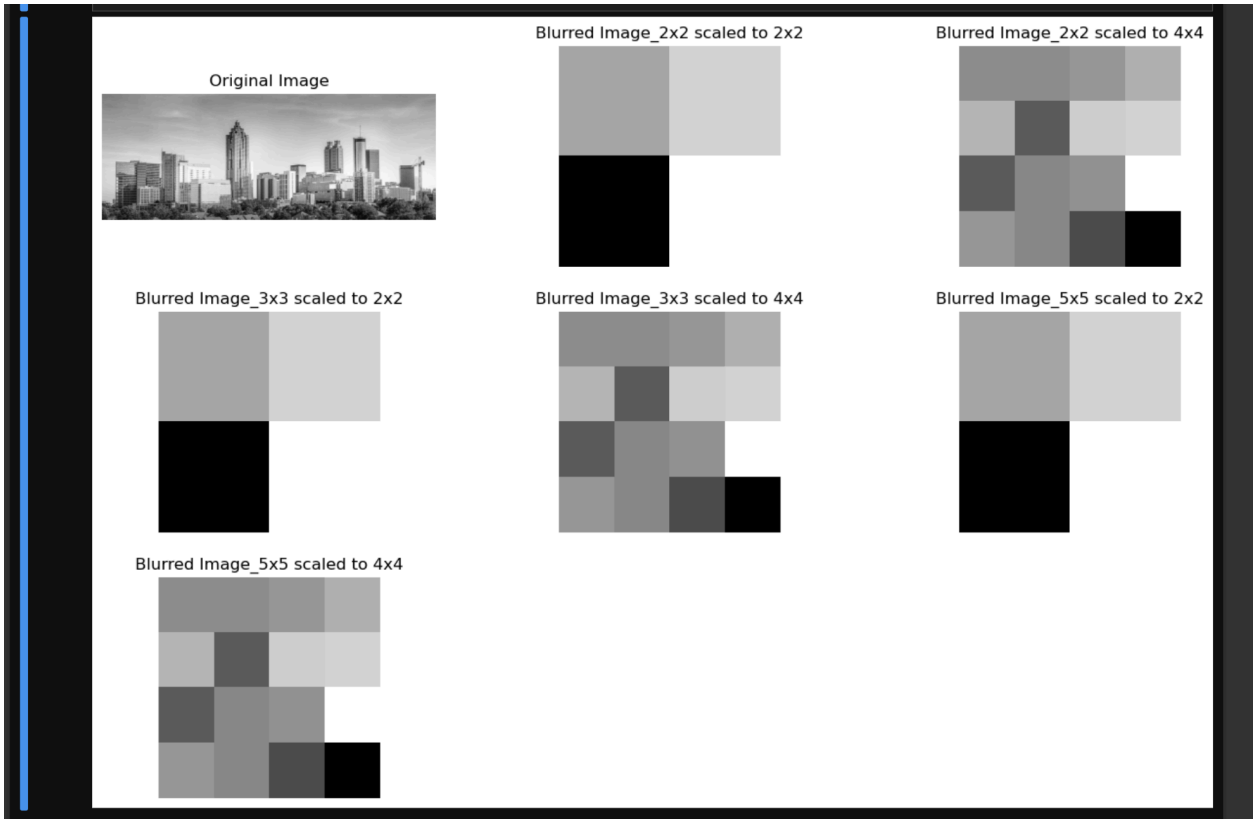
Documentation: https://www.geeksforgeeks.org/image-resizing-using-opencv-python/
For this Todo I started by blurring the image (reused the example code provided in the notebook) then used OpenCV function cv2.resize() to scale these blurred images to 2x2 and 4x4.

The function cv2.resize(blurred_2x2, (2, 2), interpolation=cv2.INTER_LINEAR) scales the image to a specified width and height.

Parameters:

Src: source image

Interpolation: the method used for resizing. Here it is specified as INTER_LINEAR. It works by taking the weighted average of surrounding pixels.



- **Use other images of your choice :** Downloaded a simple tennis court image (saved as OtherImage.jpg)
- **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.**

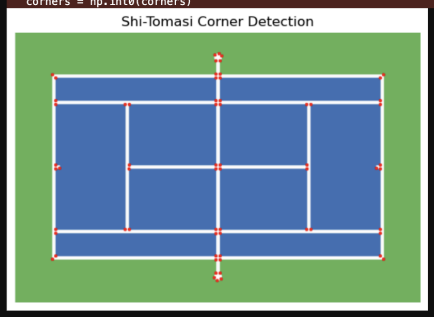
Documentation:

https://docs.opencv.org/4.x/d4/d8c/tutorial_py_shi_tomasi.html
https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga8c45db9afe636703801b0b2e440fce37

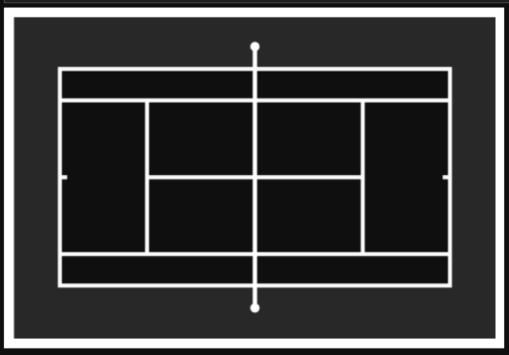
Then finally I implemented some image filtering on my choice of image (simple tennis court). Filtering includes performing corner detection using Shi-tomsai approach, blurring of the image using OpenCV in-built function blur() and calculated average of pixels using openCV inbuilt function cv2.mean(src).

[71]:
```
#https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html#ga1d6bb77486c8f92d79c8793ad995d541

corners = cv2.goodFeaturesToTrack(gray, maxCorners=100, qualityLevel=0.01, minDistance=10)
corners = np.int0(corners)

# Mark the detected corners on the original image
for i in corners:
    x, y = i.ravel()
    cv2.circle(image_rgb, (x, y), 5, (255, 0, 0), -1)

# Display the image with detected corners
plt.imshow(image_rgb)
plt.title("Shi-Tomasi Corner Detection")
plt.axis("off")
plt.show()
```

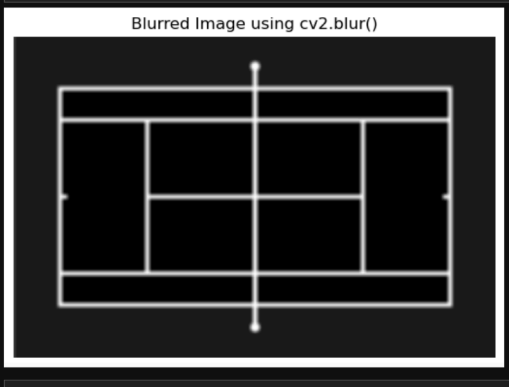
```
/var/folders/rc/7hcf34jn40g30rgtz1vjrrnw0000gn/T/ipykernel_24281/1412757133.py:4: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Deprecated NumPy 1.24)
  corners = np.int0(corners)
```



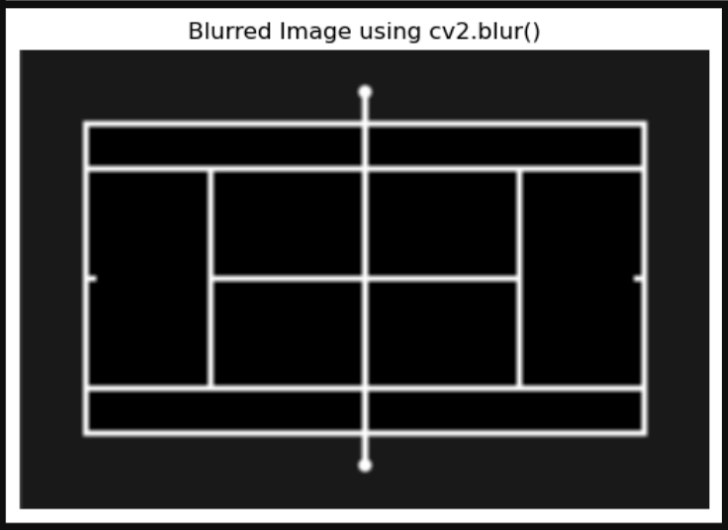
[73]:
```
#2. Blurring this Image but instead of using custom kernels i will try using OpenCV fucntion blur()
#https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga8c45db9afe636703801b0b2e440fce37
img = cv2.imread('OtherImage.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY).astype(np.float32)
plt.imshow(gray, cmap='gray')
plt.axis('off')
plt.show()
```



[80]:
```
blurred = cv2.blur(gray, (12, 12))
plt.imshow(blurred, cmap='gray')
plt.axis('off')
plt.title("Blurred Image using cv2.blur()")
plt.show()
```



[79]:
```
blurred = cv2.blur(gray, (9, 9))
plt.imshow(blurred, cmap='gray')
plt.axis('off')
plt.title("Blurred Image using cv2.blur()")
plt.show()
```



[81]:
```
#takes an average of pixels
#using OpneCV mean()
# https://docs.opencv.org/4.x/d2/de8/group__core__array.html#ga191389f8a0e58180bb13a727782cd461
mean_value = cv2.mean(gray)

print("Mean Pixel Value of the Image:", mean_value[0])
```

```
Mean Pixel Value of the Image: 143.39767489257306
```

[ ]:

CONVOLUTIONAL NEURAL NETWORKS- BUILD MODEL

Note: For this assignment a lot of research was conducted using OpenCV documentation, Pytorch documentation and reused code from coursebook (O'Reilly Machine learning with pytorch and Scikit-learn).
I Trained this model on CPU
The notebook starts by loading the CIFAR-10 dataset, Visualizes a batch of Training Data, and explores images more closely.
Then it asks to define model architecture, define Loss and Optimizer function and finally train and test the model.

TODO: Define the Network Architecture

```
Net(
  (conv_layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layers): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=2048, out_features=512, bias=True)
    (2): ReLU()
    (3): Linear(in_features=512, out_features=10, bias=True)
  )
  (dropout): Dropout(p=0.25, inplace=False)
)
```

1. Import Required Libraries:
   > torch.nn as nn: contains essential neural network components such as loss functions. Activation functions, layers etc.
   > torch.nn.functional : for functional implementation of activation functions
2. To define the layers of the model I have used nn.sequential(), this acts like a container for layer details, a convenient way to define model layers, eliminating the need to explicitly specify layer connections.
   Within this sequential,
   > nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),: Applies a 2D convolution on an image with 3 input channels (RGB), out_channel increases number of feature maps, kernel size is defined as 3x3, padding keeps spatial dimensions same after convolution.
   > nn.RELU: activation function for all hidden layers, RELU stands for Rectified Linear unit. It helps neural networks learn more complex relationships in the data, thereby helping the model to generalize better.

> nn.MaxPools2d(2,2): maxPolling reduces the spatial size of the feature map.Here it uses 2x2 filter and stride=2.

3. Fully Connected Layers: Again for this layering I used nn.sequential.
   > nn.Flatten(): changes multidimensional tensor to 1D vector. This needs to be done before feeding data into a linear (fully connected) layer.
   >nn.linear(128 * 4 * 4, 512): connects the flattened output to 512 neurons.

TODO: Specify and Compare SGC with Adam optimizer

```
TODO: try to compare with ADAM optimizer

[62]: import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer_sgd = optim.SGD(model.parameters(), lr=0.01)

# TODO, compare with optimizer ADAM https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
optimizer_adam= optim.Adam(model.parameters(), lr=0.001)
```

Training the network:

Training Results with SGD (saves as model_trained.pt):
Ran 10 epochs with a learning rate of 0.1
RESULTS:

```
Epoch: 1        Training Loss: 2.012252         Validation Loss: 1.671934
Validation loss decreased (inf --> 1.671934).  Saving model ...
Epoch: 2        Training Loss: 1.544421         Validation Loss: 1.468491
Validation loss decreased (1.671934 --> 1.468491).  Saving model ...
Epoch: 3        Training Loss: 1.356382         Validation Loss: 1.297869
Validation loss decreased (1.468491 --> 1.297869).  Saving model ...
Epoch: 4        Training Loss: 1.212309         Validation Loss: 1.138193
Validation loss decreased (1.297869 --> 1.138193).  Saving model ...
Epoch: 5        Training Loss: 1.087832         Validation Loss: 1.112456
Validation loss decreased (1.138193 --> 1.112456).  Saving model ...
Epoch: 6        Training Loss: 0.979882         Validation Loss: 0.979463
Validation loss decreased (1.112456 --> 0.979463).  Saving model ...
Epoch: 7        Training Loss: 0.888720         Validation Loss: 0.898009
Validation loss decreased (0.979463 --> 0.898009).  Saving model ...
Epoch: 8        Training Loss: 0.807652         Validation Loss: 0.900681
Epoch: 9        Training Loss: 0.733619         Validation Loss: 0.839800
Validation loss decreased (0.898009 --> 0.839800).  Saving model ...
Epoch: 10       Training Loss: 0.663293         Validation Loss: 0.832516
Validation loss decreased (0.839800 --> 0.832516).  Saving model ...
```

OBSERVATIONS MADE (using SGD optimizer):

1. Convergence is happening i.e. training loss is consistently decreasing (from 1.97 -> 0.62) suggesting SGD is performing good.
2. No Signs of overfitting: For the 10 epochs run we see trailing loss is lower than validation loss and validation loss is not increasing as well.

Results using Adam Optimizer(saved as model_trained_with_adam.pt):
Runs 10 epochs as well
Learning rate is 0.01 much lower than SGD because unlike SGD Adam dynamically updates learning rates so it works best with smaller values.

```
        valid_loss_min = valid_loss

Epoch: 1        Training Loss: 0.938161         Validation Loss: 0.876345
Validation loss decreased (inf --> 0.876345).  Saving model ...
Epoch: 2        Training Loss: 0.701895         Validation Loss: 0.780278
Validation loss decreased (0.876345 --> 0.780278).  Saving model ...
Epoch: 3        Training Loss: 0.537921         Validation Loss: 0.818282
Epoch: 4        Training Loss: 0.387152         Validation Loss: 0.845727
Epoch: 5        Training Loss: 0.269015         Validation Loss: 0.947663
Epoch: 6        Training Loss: 0.193520         Validation Loss: 1.201517
Epoch: 7        Training Loss: 0.161094         Validation Loss: 1.327034
Epoch: 8        Training Loss: 0.139349         Validation Loss: 1.340844
Epoch: 9        Training Loss: 0.134765         Validation Loss: 1.467664
Epoch: 10       Training Loss: 0.115975         Validation Loss: 1.565384
```

OBSERVATION MADE (using Adam):

1. After 3rd epoch the validation loss keeps on increasing suggesting that the model is not generalizing well instead is memorizing the training data leading to overfitting.

Test result on unseen data that we had kept aside as test_data, test_loader.
With SGD optimizer mode overall accuracy is ~70%
With Adam it is higher ~73% but that is expected as the model is overfitting.

Next Steps that I plan to implement in future for better model results:

1. Apply early stopping to prevent overfitting
2. Use Grid search to find optimal hyperparameters for the model.
3. Due to limited CPU capacity of personal devices only 10 epochs were run. I intend to use the School A100 machine or CUDA for a higher number of epochs.

REFERENCES:

1. [1] S. Raschka, Y. Liu, and V. Mirjalili, “Machine learning with pytorch and Scikit-Learn,” O’Reilly Online Learning,

https://learning.oreilly.com/library/view/machine-learning-with/9781801819312/ (accessed Feb. 21, 2025).
2. [1] “PYTORCH documentation¶,” PyTorch documentation - PyTorch 2.6 documentation, https://pytorch.org/docs/stable/index.html (accessed Feb. 21, 2025).
3. [1] “OpenCV modules,” OpenCV, https://docs.opencv.org/4.x/index.html (accessed Feb. 21, 2025).
4. [1] Comment et al., “Image resizing using opencv: Python,” GeeksforGeeks, https://www.geeksforgeeks.org/image-resizing-using-opencv-python/ (accessed Feb. 21, 2025).
5. [1] GeeksforGeeks, “ReLU activation function in deep learning,” GeeksforGeeks, https://www.geeksforgeeks.org/relu-activation-function-in-deep-learning/ (accessed Feb. 21, 2025).

Thank you!