Machine Learning Assignment

Student name: Dominic George

Student ID: 23031633

GitHub Link:

Introduction

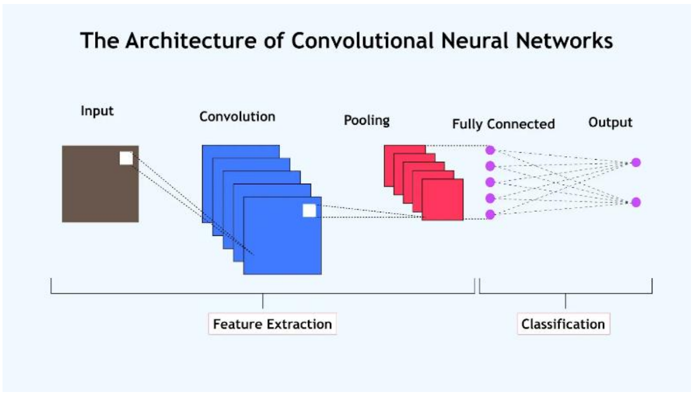
A Convolutional Neural Network (CNN) is a specialized type of neural network primarily used for image processing and tasks that involve visual data. CNNs are particularly effective in handling images, videos, and spatial data because they are designed to automatically and adaptively learn spatial hierarchies of features from input data. Unlike traditional fully connected neural networks, CNNs utilize convolutional layers that are capable of detecting patterns such as edges, shapes, textures, and more complex structures in images. Image processing is a field that deals with the manipulation and analysis of digital images. It involves applying various techniques to enhance image quality, extract useful information, and perform tasks like object detection and image segmentation.

For practical use, I've worked with datasets like CIFAR-10 (60,000 images for classification) and MNIST (70,000 grayscale images of digits) to benchmark CNN models.

Why CNNs?
Traditional neural networks struggle with image data because images are high-dimensional. CNNs are designed specifically to handle image data efficiently.

- **Automatic Feature Extraction**: CNNs can automatically learn relevant features from images without the need for manual feature engineering.
- **Translation Invariance**: CNNs are robust to translations in the input image, meaning they can recognize objects regardless of their position in the image.

- **Parameter Sharing**: By using convolutional layers, CNNs reduce the number of parameters compared to fully connected networks, making them more efficient.



Different types of CNN models

1. LeNet
2. AlexNet
3. Resnet
4. Googlenet
5. VGG

Key components of CNNs:

What is a Convolutional Layer?

The convolutional layer in CNNs extracts features from input data, like images, by applying convolution. It uses filters (kernels) that slide over the input, creating a feature map that highlights patterns such as edges, textures, or shapes.

Key Components of a Convolutional Layer

1. **Filters (Kernels)**:
    - Small matrices used to identify specific features in input data.

- Each filter is trained to recognize distinct patterns, such as vertical or horizontal edges.
- The total number of filters in a convolutional layer defines the depth of the resulting feature map.

2. **Stride**:
    - Stride refers to the number of pixels the filter moves across the input data during the convolution operation.
    - A stride of 1 means the filter moves one pixel at a time, while a stride of 2 means it moves two pixels at a time.
    - The choice of stride affects the spatial dimensions of the output feature map.
3. **Padding**:
    - Padding involves adding extra pixels around the border of the input data to control the output size.
    - There are different types of padding:
        - **Valid Padding**: No padding is added, which reduces the size of the output feature map.
        - **Same Padding**: Padding is added to ensure that the output feature map has the same spatial dimensions as the input.
4. **Activation Function**:
    - After the convolution operation, an activation function (commonly ReLU) is applied to introduce non-linearity into the model.
    - This non-linearity allows the network to learn complex patterns and relationships in the data.

How Does a Convolutional Layer Work?

1. **Input**:
    - The input to a convolutional layer is typically a multi-dimensional array, such as an image represented as a height x width x channels.

2. **Convolution Operation**:
    - The filter is applied to the input data by performing an element-wise multiplication followed by a summation.
    - For each position of the filter on the input, the output value is calculated as follows:

$$Z(i,j) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} X(i+m, j+n) \cdot K(m,n)$$

3. **Output Feature Map**:
    - The convolution operation produces a feature map that emphasizes the features detected by the filters.
    - The dimensions of the output feature map are influenced by the input size, filter size, stride, and padding.

Benefits of Convolutional Layers

- **Parameter Sharing:** Convolutional layers apply the same filter across the entire input, significantly reducing the number of parameters compared to fully connected layers.
- **Local Connectivity:** Each neuron in a convolutional layer connects to a small region of the input, enabling the network to focus on local patterns.
- **Hierarchical Feature Learning:** Stacking multiple convolutional layers allows the network to learn progressively complex features, starting from simple edges to more abstract shapes in deeper layers.

What is an Activation Function?

Activation functions are essential in CNNs and neural networks as they introduce non-linearity, enabling the model to learn complex patterns. Applied after convolution, functions like ReLU prevent the network from acting as a linear model and greatly impact performance and convergence.

Key Roles of Activation Functions

1. **Introducing Non-Linearity**: Activation functions control a neuron's output range, which is important for specific tasks
2. **Controlling Output**: Activation functions can control the range of output values from a neuron, which can be important for certain tasks.
3. **Enabling Gradient Descent**: Activation functions must be differentiable to enable backpropagation, allowing the network's weights to be updated during training.

Common Activation Functions

Here are some of the most commonly used activation functions in CNNs:

1. **ReLU (Rectified Linear Unit)**:
    - **Formula**

      $$f(x) = \max(0, x)$$

    - **Characteristics**:
        - Outputs zero for negative inputs and the input value for positive inputs.
        - Computationally efficient and helps mitigate the vanishing gradient problem.
        - However, it can suffer from the "dying ReLU" problem, where neurons can become inactive and stop learning.
2. **Leaky ReLU**:
    - **Formula**

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Where:

- $f(x)$ is the output of the Leaky ReLU function.
- α is a small constant (typically a value like 0.01).
- x is the input to the function.

- **Characteristics**:
  - Similar to ReLU but allows a small, non-zero gradient when the input is negative.
  - Helps prevent the dying ReLU problem.

3. **Sigmoid**:

- **Formula**: $$f(x) = \frac{1}{1 + e^{-x}}$$
- **Characteristics**:
  - Outputs values between 0 and 1, making it ideal for binary classification tasks.
  - May experience the vanishing gradient problem in deep networks, as gradients can become very small for extreme input values.

Tanh (Hyperbolic Tangent):

- **Formula**:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **Characteristics**:
  - Outputs values between -1 and 1, centering the data around zero.

        - Generally performs better than the sigmoid function but can still suffer from the vanishing gradient problem.

4. **Softmax**:
    - **Formula**

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

    - **Characteristics**:
        - Used in the output layer of multi-class classification problems.
        - Converts raw scores into probabilities that sum to 1, allowing for interpretation as class probabilities.

What is a Pooling Layer?

The pooling layer in CNNs reduces the spatial dimensions of feature maps, improving efficiency and preventing overfitting by summarizing local features. Max pooling is often used to retain key features while ensuring translation invariance.

Key Functions of Pooling Layers

1. **Dimensionality Reduction:** Pooling layers reduce feature map size, decreasing parameters and computations, which speeds up training and helps prevent overfitting.
2. **Translation Invariance:** Pooling layers provide translation invariance by summarizing local features, making the network less sensitive to small shifts or distortions in the input image.
3. **Feature Extraction:** Pooling layers preserve key features and discard less important information, enabling the network to focus on essential patterns for classification or detection.

Types of Pooling Layers

There are several types of pooling operations commonly used in CNNs:

1. **Max Pooling**:

- **Operation**: Max pooling selects the maximum value from a defined window as it slides over the feature map.
- **Example**: For a 2x2 max pooling operation, the output will be the maximum value from each 2x2 region of the input feature map.
- **Advantages**: Max pooling is effective at retaining the most prominent features and is widely used in CNN architectures.

2. Average Pooling:

- **Operation**: Average pooling calculates the average value from a defined window as it slides over the feature map.
- **Example**: For a 2x2 average pooling operation, the output will be the average of each 2x2 region of the input feature map.
- **Advantages**: Average pooling can be useful in certain contexts, but it may not retain the most salient features as effectively as max pooling.

3. **Global Average Pooling**:

- **Operation**: Global average pooling computes the average of all values in the feature map, resulting in a single value for each feature map.
- **Advantages**: This method is often used before the final classification layer in CNNs, as it reduces the feature map to a single value per channel, making it suitable for fully connected layers.

How Does a Pooling Layer Work?

1. **Input**: The input to a pooling layer is typically a multi-dimensional array (tensor) representing the output of a convolutional layer.
2. **Pooling Operation**: The pooling layer applies the chosen pooling operation over the input feature map using a defined window size and stride.

3. **Output Feature Map**: The result of the pooling operation is a down-sampled feature map that retains the most important features while reducing the spatial dimensions.

Fully Connected Layer

After several convolutional and pooling layers, the output is flattened and passed through fully connected layers to make the final classification.

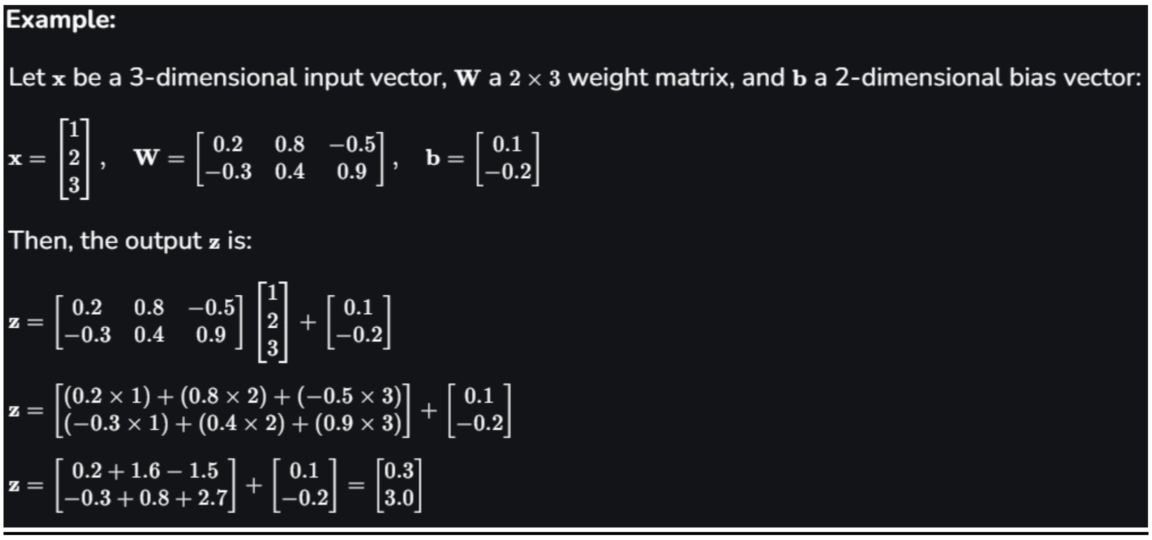
Some of the important concepts and techniques in the image processing are as follows:

1. **Understanding Digital Images:** Digital images consist of pixels, which represent color and brightness in color spaces like RGB or CMYK. Understanding pixels and color spaces is essential for analyzing and enhancing images in image processing.
2. **Image Filtering:** Image filtering techniques enhance quality by reducing noise, sharpening edges, and applying transformations. Common filters include Gaussian for smoothing, median for noise reduction, and edge enhancement for highlighting boundaries.
3. **Object Detection:** Object detection methods locate objects in images by analyzing features like shape, texture, and color, using techniques like Haar cascades and deep learning for accurate detection.
4. **Image Segmentation:** Image segmentation splits an image into regions, distinguishing the foreground from the background, which aids in object recognition and editing. Techniques like thresholding, region-based segmentation, and clustering enhance image analysis.
5. **Deep Learning for Image Processing:** Deep learning, especially CNNs, has revolutionized image processing by automatically learning features, improving accuracy and efficiency in tasks like classification, object detection, and segmentation.

Example for edge detection and image filtering

Math behind the technique

$$S(i,j) = (I * K)(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n) \cdot K(m,n)$$



Example:

Let $\mathbf{x}$ be a 3-dimensional input vector, $\mathbf{W}$ a 2×3 weight matrix, and $\mathbf{b}$ a 2-dimensional bias vector:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} 0.2 & 0.8 & -0.5 \\ -0.3 & 0.4 & 0.9 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

Then, the output $\mathbf{z}$ is:

$$\mathbf{z} = \begin{bmatrix} 0.2 & 0.8 & -0.5 \\ -0.3 & 0.4 & 0.9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

$$\mathbf{z} = \begin{bmatrix} (0.2 \times 1) + (0.8 \times 2) + (-0.5 \times 3) \\ (-0.3 \times 1) + (0.4 \times 2) + (0.9 \times 3) \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

$$\mathbf{z} = \begin{bmatrix} 0.2 + 1.6 - 1.5 \\ -0.3 + 0.8 + 2.7 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 3.0 \end{bmatrix}$$

Filters (Kernels) in CNNs

- **What are Filters (Kernels)?**
  - A filter is a small matrix of weights that is used to perform a convolution operation on an input image to extract local features.
  - Filters are typically small in size but slide across the entire image to learn different patterns in different parts of the image.
- **How do Filters Work?**
  - The filter moves over the input image or feature map, one step at a time. This process is called convolution.
  - At each position, the filter performs a dot product between the filter and the section of the image it is covering, producing a single. The output of these dot products across the entire image forms a value.

Why Use Filters?

- Filters allow the network to learn various patterns in the image, such as:
  - Edges: Vertical, horizontal, and diagonal edges.
  - Textures: Specific patterns like stripes or circles.
  - More complex shapes: As the network deepens, filters in deeper layers can learn to detect higher-level features like parts of objects.

Applications of CNN

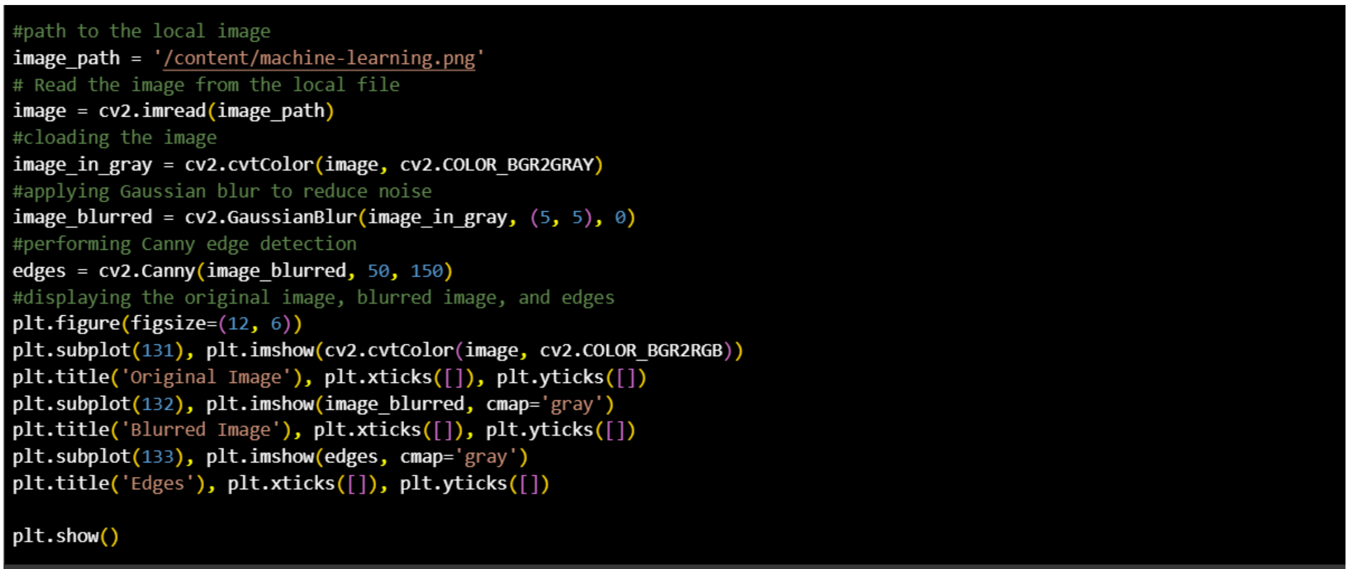
1. **Object Detection**
2. **Facial Recognition**
3. **Medical Image Analysis**

Disadvantages of CNN

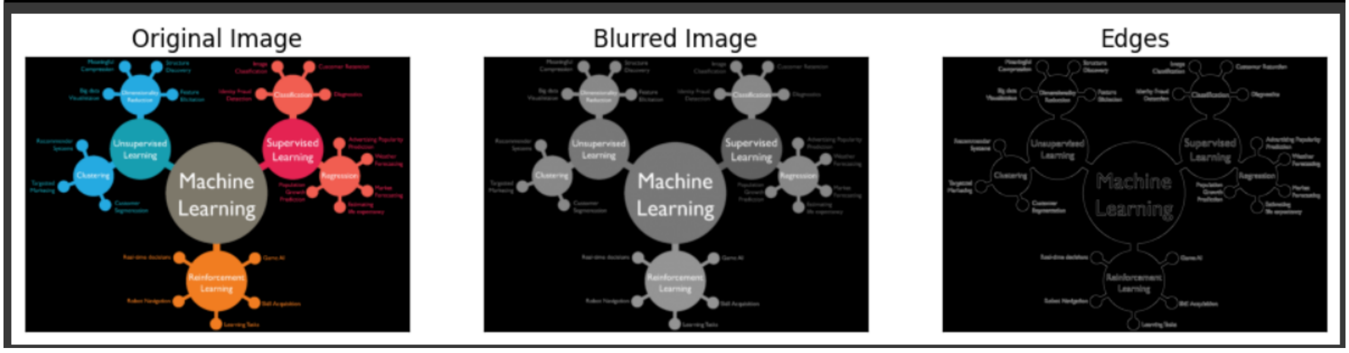
- **Complexity**: CNNs can be complex and difficult to train, especially for large datasets.
- **Resource-Intensive**: CNNs require significant computational resources for training and deployment.
- **Data Requirements**: CNNs need large amounts of labelled data for training.
- **Interpretability**: CNNs can be difficult to interpret, making it challenging to understand their predictions.

Implementation:

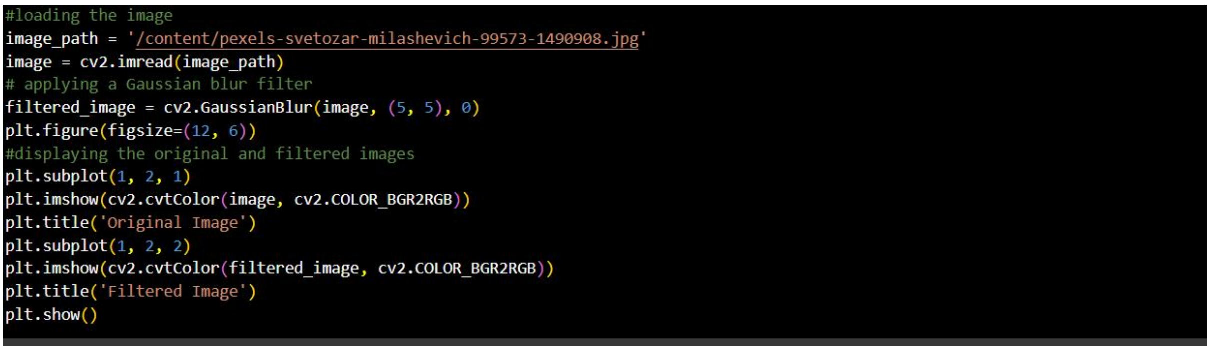
Reducing noise and edge reduction:



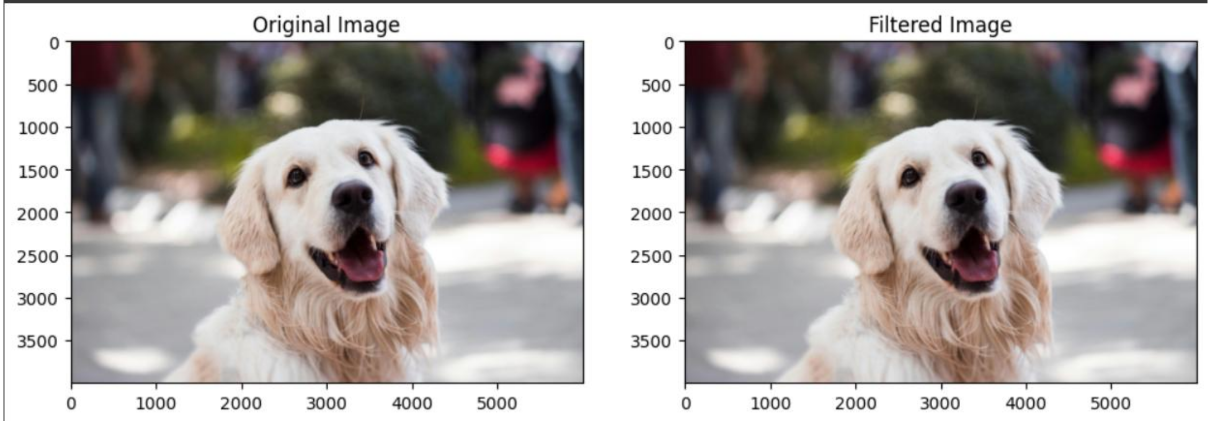
```
#path to the local image
image_path = '/content/machine-learning.png'
# Read the image from the local file
image = cv2.imread(image_path)
#cloading the image
image_in_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#applying Gaussian blur to reduce noise
image_blurred = cv2.GaussianBlur(image_in_gray, (5, 5), 0)
#performing Canny edge detection
edges = cv2.Canny(image_blurred, 50, 150)
#displaying the original image, blurred image, and edges
plt.figure(figsize=(12, 6))
plt.subplot(131), plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132), plt.imshow(image_blurred, cmap='gray')
plt.title('Blurred Image'), plt.xticks([]), plt.yticks([])
plt.subplot(133), plt.imshow(edges, cmap='gray')
plt.title('Edges'), plt.xticks([]), plt.yticks([])

plt.show()
```



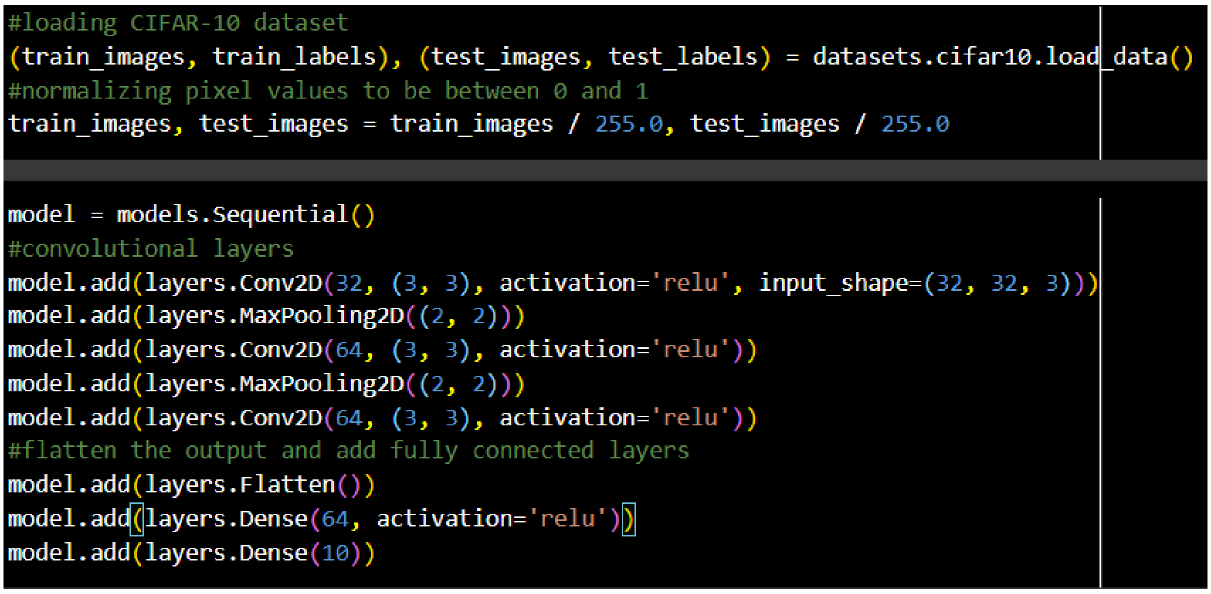
Applying Gaussian Blur:



```
#loading the image
image_path = '/content/pexels-svetozar-milashevich-99573-1490908.jpg'
image = cv2.imread(image_path)
# applying a Gaussian blur filter
filtered_image = cv2.GaussianBlur(image, (5, 5), 0)
plt.figure(figsize=(12, 6))
#displaying the original and filtered images
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(filtered_image, cv2.COLOR_BGR2RGB))
plt.title('Filtered Image')
plt.show()
```

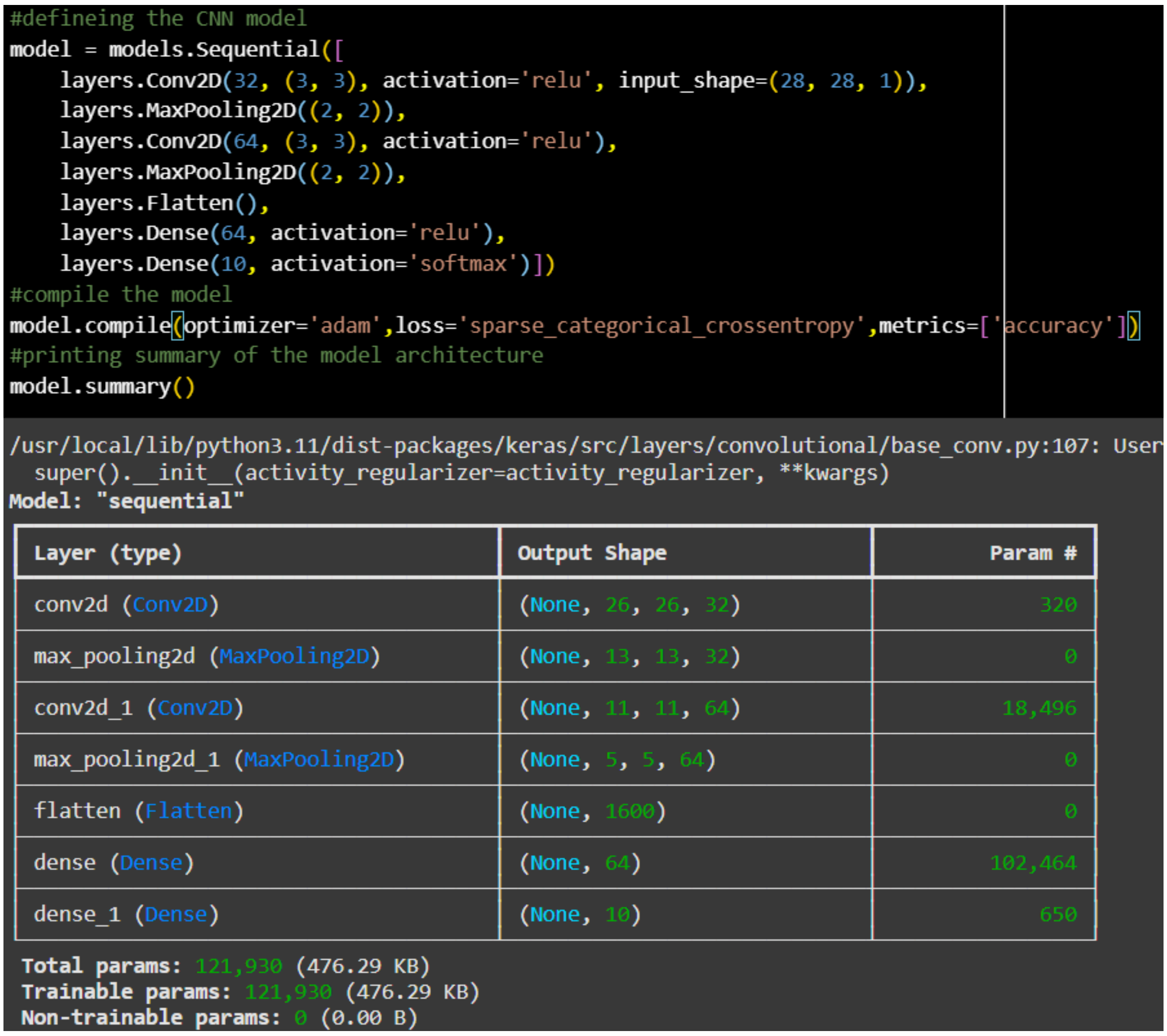


Building a CNN model:



```
#loading CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
#normalizing pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

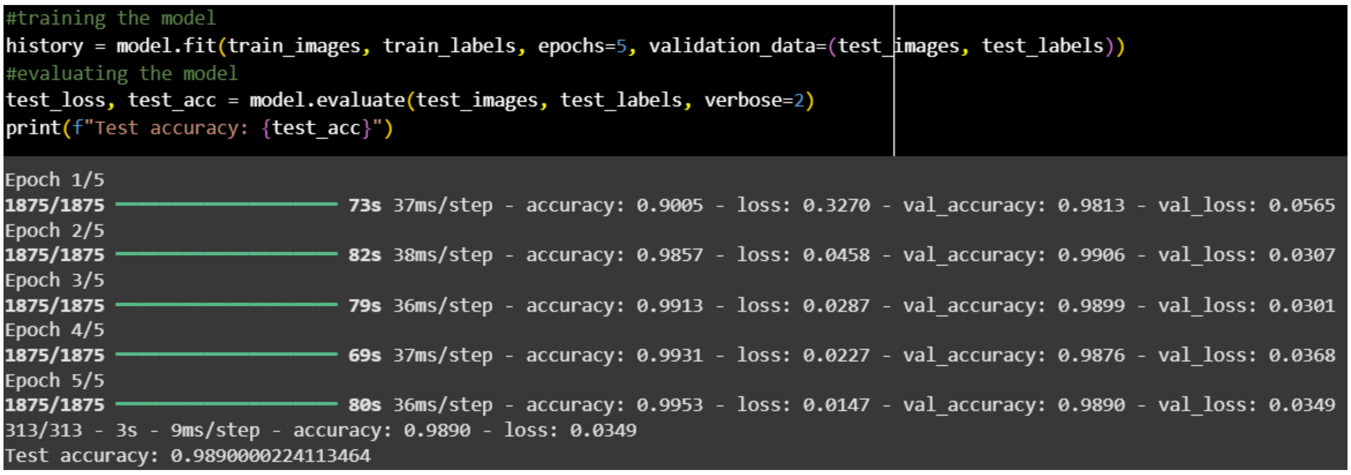
```
model = models.Sequential()
#convolutional layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
#flatten the output and add fully connected layers
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```



```
#defineing the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')])
#compile the model
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
#printing summary of the model architecture
model.summary()
```

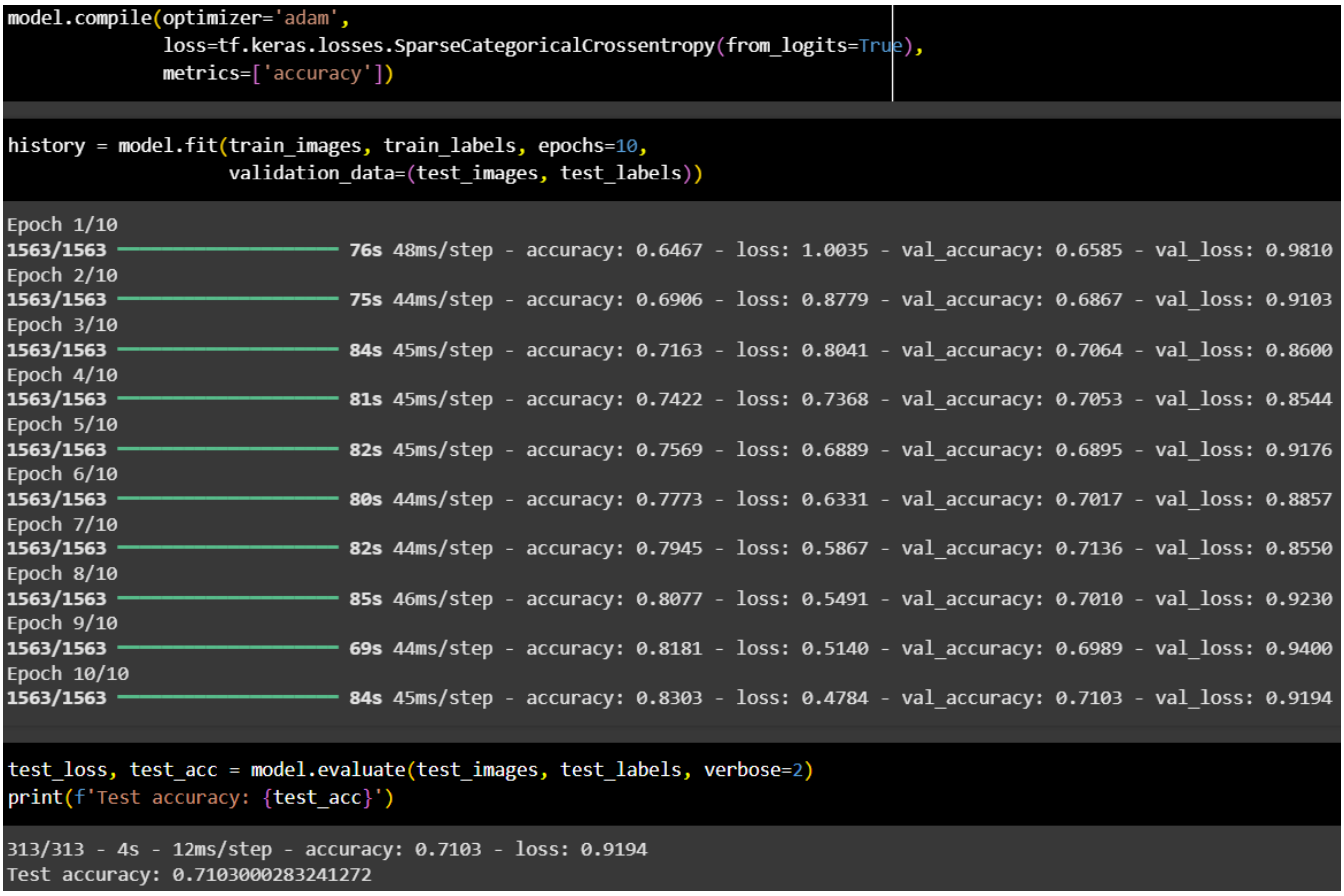
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: User
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
┌──────────────────────────────┬──────────────────────────┬───────────────┐
│ Layer (type)                 │ Output Shape             │       Param # │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ conv2d (Conv2D)              │ (None, 26, 26, 32)       │           320 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D) │ (None, 13, 13, 32)       │             0 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)            │ (None, 11, 11, 64)       │        18,496 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)│ (None, 5, 5, 64)        │             0 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ flatten (Flatten)            │ (None, 1600)             │             0 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ dense (Dense)                │ (None, 64)               │       102,464 │
├──────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_1 (Dense)              │ (None, 10)               │           650 │
└──────────────────────────────┴──────────────────────────┴───────────────┘
 Total params: 121,930 (476.29 KB)
 Trainable params: 121,930 (476.29 KB)
 Non-trainable params: 0 (0.00 B)
```

Training and evaluating the model:



```
#training the model
history = model.fit(train_images, train_labels, epochs=5, validation_data=(test_images, test_labels))
#evaluating the model
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"Test accuracy: {test_acc}")
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 73s 37ms/step - accuracy: 0.9005 - loss: 0.3270 - val_accuracy: 0.9813 - val_loss: 0.0565
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 82s 38ms/step - accuracy: 0.9857 - loss: 0.0458 - val_accuracy: 0.9906 - val_loss: 0.0307
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 79s 36ms/step - accuracy: 0.9913 - loss: 0.0287 - val_accuracy: 0.9899 - val_loss: 0.0301
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 69s 37ms/step - accuracy: 0.9931 - loss: 0.0227 - val_accuracy: 0.9876 - val_loss: 0.0368
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 80s 36ms/step - accuracy: 0.9953 - loss: 0.0147 - val_accuracy: 0.9890 - val_loss: 0.0349
313/313 - 3s - 9ms/step - accuracy: 0.9890 - loss: 0.0349
Test accuracy: 0.9890000224113464
```



```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

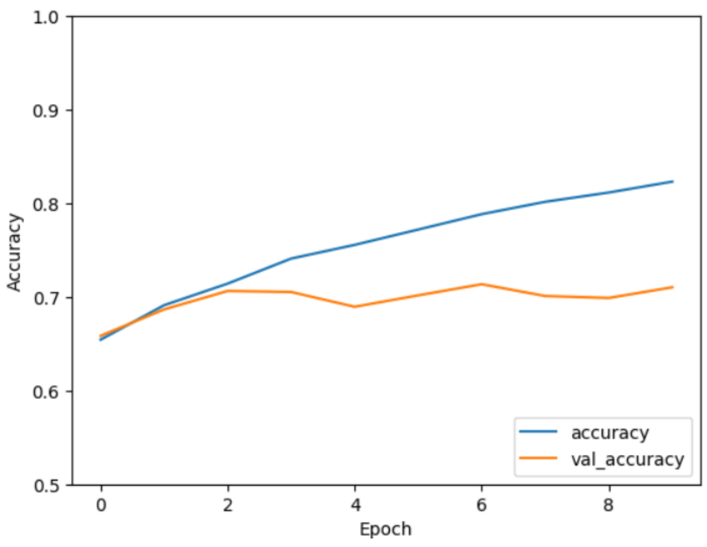
```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 48ms/step - accuracy: 0.6467 - loss: 1.0035 - val_accuracy: 0.6585 - val_loss: 0.9810
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 44ms/step - accuracy: 0.6906 - loss: 0.8779 - val_accuracy: 0.6867 - val_loss: 0.9103
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 45ms/step - accuracy: 0.7163 - loss: 0.8041 - val_accuracy: 0.7064 - val_loss: 0.8600
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 45ms/step - accuracy: 0.7422 - loss: 0.7368 - val_accuracy: 0.7053 - val_loss: 0.8544
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 45ms/step - accuracy: 0.7569 - loss: 0.6889 - val_accuracy: 0.6895 - val_loss: 0.9176
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 44ms/step - accuracy: 0.7773 - loss: 0.6331 - val_accuracy: 0.7017 - val_loss: 0.8857
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 44ms/step - accuracy: 0.7945 - loss: 0.5867 - val_accuracy: 0.7136 - val_loss: 0.8550
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 85s 46ms/step - accuracy: 0.8077 - loss: 0.5491 - val_accuracy: 0.7010 - val_loss: 0.9230
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 69s 44ms/step - accuracy: 0.8181 - loss: 0.5140 - val_accuracy: 0.6989 - val_loss: 0.9400
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 45ms/step - accuracy: 0.8303 - loss: 0.4784 - val_accuracy: 0.7103 - val_loss: 0.9194
```

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')
```

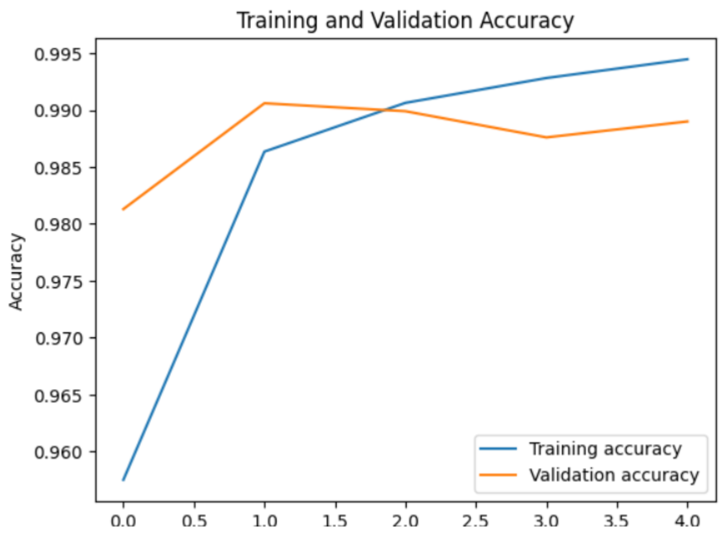
```
313/313 - 4s - 12ms/step - accuracy: 0.7103 - loss: 0.9194
Test accuracy: 0.7103000283241272
```

Visualization of training and validation accuracy:

1.



2.



Conclusion:

In conclusion, Convolutional Neural Networks (CNNs) have revolutionized image processing by efficiently analyzing visual data and extracting features, excelling in tasks like classification, detection, and segmentation. Despite challenges such as high computational needs, CNNs are crucial in deep learning and applications like medical imaging and autonomous driving. Understanding CNNs is key for advancing image processing and real-world uses.

Reference:

1. LeCun, Y., Bengio, Y, & Hinton, G (2015). Deep learning. *Nature*, 521(7553), 436-444.

https://doi.org/10.1038/nature14539

2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 1097-1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

3. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770-778. https://doi.org/10.1109/CVPR.2016.90

4. https://www.geeksforgeeks.org/math-behind-convolutional-neural-networks/

5.Almabetter, my previous institution.

https://gist.github.com/almabetter-school-ds/d25c14f2eb52f1c6243246e3dc86342c#file-image-processing-student_-material-ipynb