Introduction:

Purpose of this code is to train a CNN on cifor10 dataset to predict each image label and then show future map of 2 layer after training.

In this code I first import same library then then dataset in the second step prepare dataset for fitting model on it then see the result of training and find out accuracy finally show future map to argue effect of each layer on training prosses

Dependencies:

```
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models
```

in this code I just import some relevant library and all are last version scene 01/12/2023

Load CIFAR-10 dataset and have normalized it by dividing it on 255

```
# Load CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) =
keras.datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Step define the CNN model this is the thing which will be fitted on training dataset and then Display the model summary the model summary provides a compact overview of the model architecture, including the types and shapes of each layer, the total number of parameters, # and the output shape of each layer

```
model = keras.Sequential([
    # First convolutional layer with 32 filters, each of size (3, 3), using ReLU
activation
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)), # Max pooling layer with a pool size of (2, 2)
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(), # Flatten layer to convert 2D feature maps to a vector
    # Fully connected layer with 128 neurons using ReLU activation
    layers.Dense(128, activation='relu'),
    # Dropout layer with a dropout rate of 0.5 to reduce overfitting
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    # Output layer with 10 neurons (assuming it's a classification task) using
softmax activation
    layers.Dense(10, activation='softmax'),
])
model.summary()
```

and here is the model summery

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 128)               131200

 dropout (Dropout)           (None, 128)               0
...
Total params: 196426 (767.29 KB)
Trainable params: 196426 (767.29 KB)
Non-trainable params: 0 (0.00 Byte)
```

Compiling a model in machine learning involves specifying several aspects of how the model should be trained. The compilation step is crucial because it sets up the training process by defining key parameters.

```
model.compile(
    # Optimizer: 'adam' is an adaptive learning rate optimization algorithm.
    optimizer='adam',

    # Loss function: 'sparse_categorical_crossentropy' is suitable for
classification tasks
    # with integer labels, where each input can belong to exactly one class.
    loss='sparse_categorical_crossentropy',

    # Metrics: 'accuracy' is a common metric for classification tasks to measure
    # the fraction of correctly classified samples.
    metrics=['accuracy']
)
```

Train the model

The fit method is used to train the model on the provided training data. It takes input data (train_images), corresponding labels (train_labels), and specifies the number of epochs (complete passes through the entire dataset) and the batch size (number of samples per gradient update).

```
model.fit(
    train_images,     # Input training images
    train_labels,     # Corresponding labels
    epochs=5,         # Number of training epochs
    batch_size=32      # Batch size for each training step
)
```

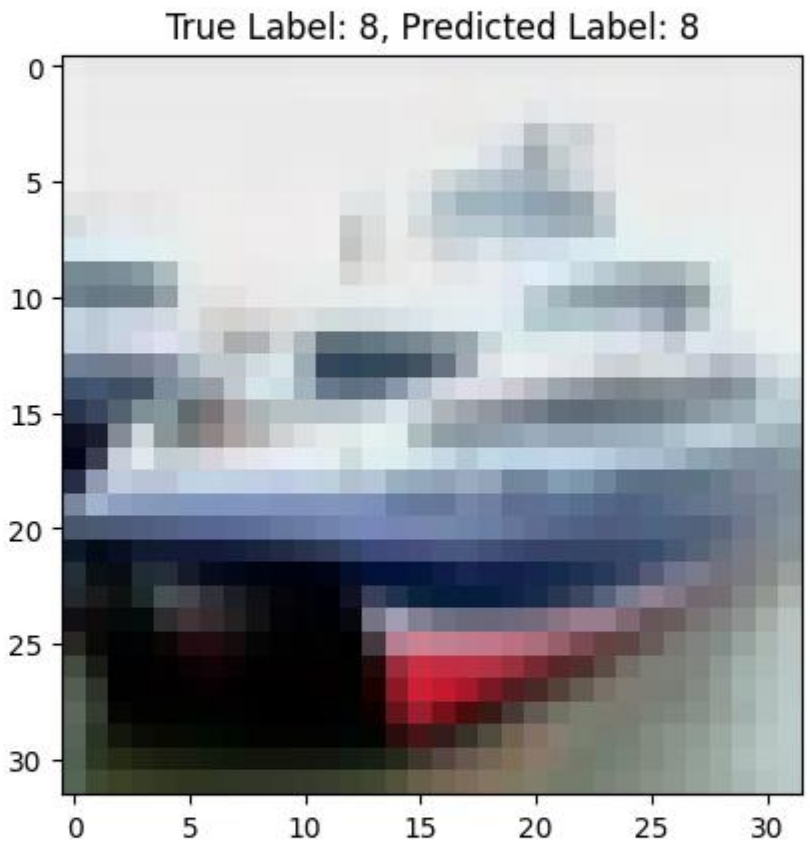
Evaluate the model on the test set the evaluate method computes the loss and metrics specified during model compilation on the provided test data (test_images and test_labels).

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"\nTest Accuracy: {test_acc}")
```

Display predictions for the first few examples from the test set using the trained model

```
for i in range(5):
    # Make predictions for the i-th test image
    prediction = model.predict(test_images[i:i+1])

    # Get the predicted label (index with the highest probability)
    predicted_label = prediction.argmax()
    # Display the i-th test image along with true and predicted labels
    plt.imshow(test_images[i])
    plt.title(f"True Label: {test_labels[i][0]}, Predicted Label:
{predicted_label}")
    plt.show()
```



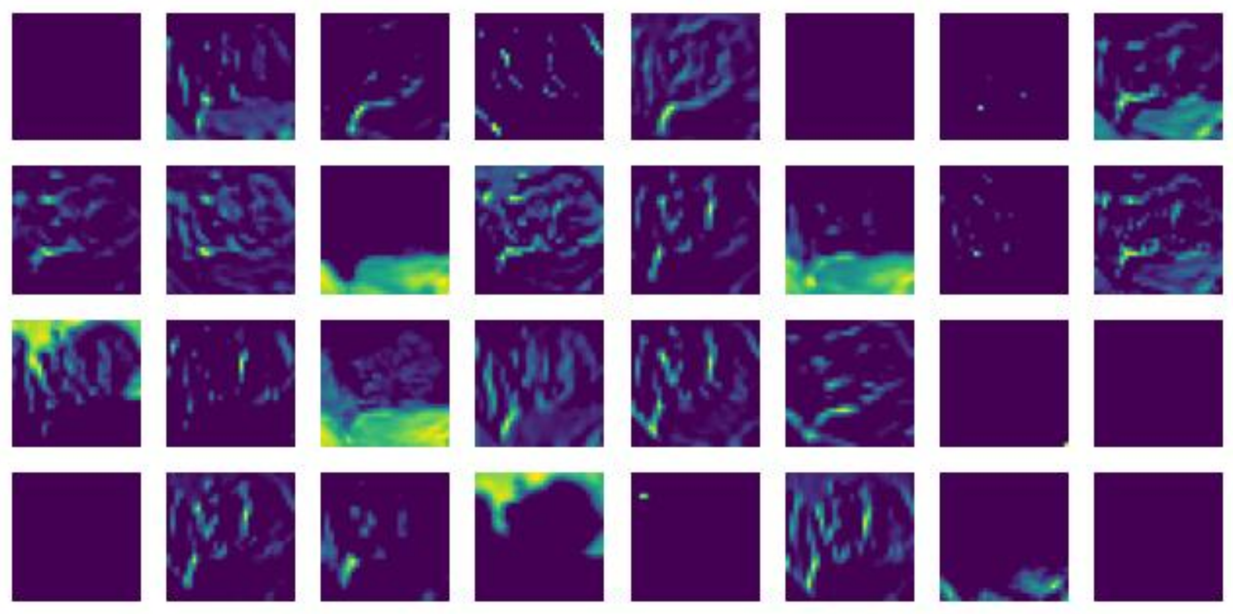
Create a model that outputs the activations of the second Conv2D layer The activation_model is a new model that takes the same input as the original model but outputs the activations of all layers, including intermediate layers.

```
# Create a model that outputs the activations of the second Conv2D layer
layer_outputs = [layer.output for layer in model.layers]
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```python
layer_index = 0
# Get the activations of the second Conv2D layer for a sample input image
sample_image = test_images[0:1]
activations = activation_model.predict(sample_image)

# Visualize the feature maps for the chosen layer
layer_activation = activations[layer_index]
plt.figure(figsize=(8, 8))
for i in range(layer_activation.shape[3]):
    plt.subplot(8, 8, i + 1)
    plt.imshow(layer_activation[0, :, :, i], cmap='viridis')
    plt.axis('off')

plt.show()
```

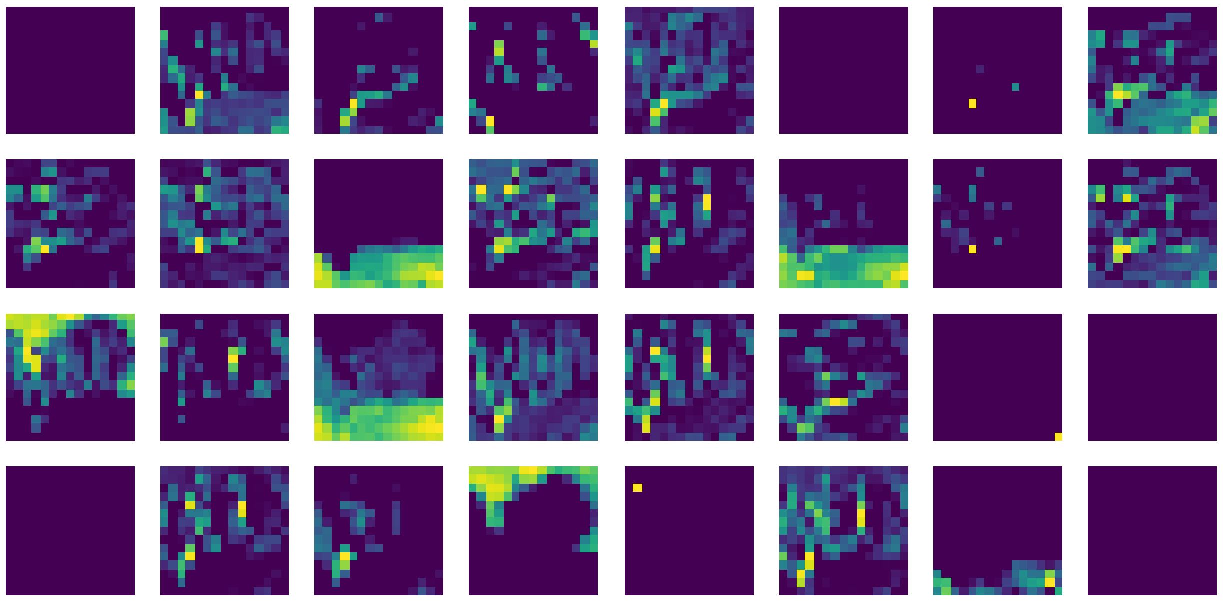


```python
layer_index = 1
```

```
sample_image = test_images[0:1]
activations = activation_model.predict(sample_image)

# Visualize the feature maps for the chosen layer
layer_activation = activations[layer_index]
plt.figure(figsize=(8, 8))
for i in range(layer_activation.shape[3]):
    plt.subplot(8, 8, i + 1)
    plt.imshow(layer_activation[0, :, :, i], cmap='viridis')
    plt.axis('off')

plt.show()
```



As we can see here The complexity of features in the visualizations may not necessarily correlate with the layer index. It's more accurate to say that the complexity of features tends to increase as we go deeper into the network. In a CNN, earlier layers learn simple features like edges and textures, while deeper layers learn more complex and abstract features, capturing high-level patterns and representations. When you observe more complex features in the visualizations for layer_index = 0, it could be due to several reasons:

Visualization Interpretation: The visualizations may highlight certain aspects of the features that might appear more complex to the human eye, even if the network is learning simpler features at that stage.

Model Architecture: The specific architecture of your model, the number of filters, and the receptive field of the convolutional layers can influence the nature of the features learned at each layer.

Dataset Characteristics: The content and complexity of the dataset itself can influence the features learned by the model. more complex features in the visualizations for layer_index = 0, it doesn't necessarily mean that the first layer is capturing more complex information than the second layer. It could be a visual interpretation, or it might be influenced by the factors mentioned above. It's generally expected that deeper layers capture more abstract and complex features. find that the visualizations are counterintuitive, it could be useful to investigate further or consider additional analysis techniques to better understand the features learned by each layer.