Experiment No 8

Title: A simple CNN Make a train and validation dataset of images with vertical and horizontal images Defining the CNN to predict the knowledge from image classification Visualizing the learned CNN Model

Aim: To define the CNN to predict the knowledge from image classification Visualizing the learned CNN Model.

Theory: Convolutional Neural Networks (CNNs) are a class of deep learning models primarily designed for processing and analyzing grid-like data, with a strong emphasis on their application in computer vision tasks.

Key components include:

- **Convolutional Layers**: CNNs use convolutional layers to process input data, where a filter slides over the data to detect local patterns. The output is called a feature map.
- **Pooling Layers**: Pooling reduces the size of feature maps while retaining essential information, commonly using max or average pooling. This helps reduce computation and prevents overfitting.
- **Activation Functions**: Non-linear functions like ReLU introduce non-linearity, making the model more powerful and effective.
- **CNN Architecture:** CNNs consist of multiple convolutional, activation, and pooling layers, followed by fully connected layers. Early layers capture simple features, while deeper layers capture complex patterns.
- **Weight Sharing:** Filters are shared across the input space, allowing the network to detect the same features in different parts of the input, enhancing efficiency.
- **Striding**: Filters can move with a stride, reducing the spatial dimensions of the output feature maps.
- **Padding:** Adding pixels to the input before convolution preserves feature map dimensions, important for deeper layers.
- **Transfer Learning:** Pre-trained CNNs on large datasets (e.g., ImageNet) are fine-tuned for specific tasks, improving performance on smaller datasets.
- **Applications:** CNNs excel in image classification, object detection, face recognition, medical image analysis, and even extend to NLP and speech recognition.
- **Training:** CNNs are trained using stochastic gradient descent (SGD) with backpropagation to adjust weights based on the loss function.

Conclusion: A simple CNN was built to classify images as either horizontal or vertical. The trained model learned to distinguish between the two orientations, achieving accuracy on both the training and validation datasets. The model's architecture and learned features can be visualized to better understand its performance.

CNN for Image Classification

In [1]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
from tensorflow.keras.utils import to_categorical
```

In [2]:
```
# Step 1: Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

In [3]:
```
# Step 2: Filter out only the airplane (class 0) and ship (class 8) image
# These two classes will be used for binary classification based on image
classes_to_keep = [0, 8]  # We are focusing on airplanes (0) and ships (8
```

In [4]:
```

# Create a filter to select only these two classes from the training and
train_filter = np.isin(y_train, classes_to_keep).flatten()
test_filter = np.isin(y_test, classes_to_keep).flatten()
```

In [5]:
```
# Apply the filter to the dataset
x_train = x_train[train_filter]
y_train = y_train[train_filter]
x_test = x_test[test_filter]
y_test = y_test[test_filter]
```

In [6]:
```
# Step 3: Convert labels from class-based (0 = airplane, 8 = ship) to ori
# We label airplanes (class 0) as "horizontal" (0) and ships (class 8) as
y_train = np.where(y_train == 0, 0, 1)
y_test = np.where(y_test == 0, 0, 1)
```

In [7]:
```
# Step 4: Normalize pixel values to a range between 0 and 1
# This is done by dividing all pixel values by 255 since pixel values ran
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

In [8]:
```
# Step 5: Convert labels to one-hot encoding format for binary classifica
# One-hot encoding changes the labels from single integers (0 or 1) to ve
y_train = to_categorical(y_train, 2)
y_test = to_categorical(y_test, 2)
```

In [9]:
```
# Step 6: Define the CNN model for binary classification (horizontal vs v

# Initialize the model as a Sequential model (a stack of layers)
model = Sequential()
```

In [11]:

```
# Add the first convolutional layer
# 32 filters, each of size 3x3, with ReLU activation, applied to 32x32x3
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))

# Add a max-pooling layer to reduce the spatial dimensions (downsampling)
model.add(MaxPooling2D(pool_size=(2, 2)))

# Add the second convolutional layer
# 64 filters, each of size 3x3, with ReLU activation
model.add(Conv2D(64, (3, 3), activation='relu'))

# Add another max-pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# Add the third convolutional layer
# 128 filters, each of size 3x3, with ReLU activation
model.add(Conv2D(128, (3, 3), activation='relu'))

# Add another max-pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the output of the final pooling layer into a 1D vector
# This is required before feeding the data into fully connected layers
model.add(Flatten())

# Add a fully connected (dense) layer with 128 units and ReLU activation
model.add(Dense(128, activation='relu'))

# Add a dropout layer to prevent overfitting by randomly dropping 50% of
model.add(Dropout(0.5))

# Add the final output layer with 2 units (one for each class: horizontal
# The softmax activation function converts the output into probabilities
model.add(Dense(2, activation='softmax'))
```

```
C:\Users\Shantanu\anaconda3\lib\site-packages\keras\src\layers\convoluti
onal\base_conv.py:99: UserWarning: Do not pass an `input_shape`/`input_d
im` argument to a layer. When using Sequential models, prefer using an `
Input(shape)` object as the first layer in the model instead.
  super().__init__(
```

In [12]:

```
# Step 7: Compile the model
# Use the Adam optimizer, categorical crossentropy for loss (since it's a
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
```

In [13]:

```
# Step 8: Train the model
# Train the model for 20 epochs with a batch size of 64
# Use 20% of the training data as the validation set during training
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validatio
```

```
Epoch 1/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 6s 31ms/step - accuracy: 0.5846 - loss: 0.6
616 - val_accuracy: 0.7015 - val_loss: 0.5690
Epoch 2/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.7542 - loss: 0.5
044 - val_accuracy: 0.8415 - val_loss: 0.3670
Epoch 3/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.8344 - loss: 0.3
788 - val_accuracy: 0.8715 - val_loss: 0.3266
Epoch 4/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.8678 - loss: 0.3
198 - val_accuracy: 0.8895 - val_loss: 0.2853
Epoch 5/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.8915 - loss: 0.2
644 - val_accuracy: 0.8990 - val_loss: 0.2491
Epoch 6/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 27ms/step - accuracy: 0.9056 - loss: 0.2
371 - val_accuracy: 0.9030 - val_loss: 0.2473
Epoch 7/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.9280 - loss: 0.1
947 - val_accuracy: 0.8930 - val_loss: 0.2731
Epoch 8/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.9276 - loss: 0.1
864 - val_accuracy: 0.9160 - val_loss: 0.2236
Epoch 9/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - accuracy: 0.9439 - loss: 0.1
420 - val_accuracy: 0.9165 - val_loss: 0.2401
Epoch 10/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 25ms/step - accuracy: 0.9587 - loss: 0.1
225 - val_accuracy: 0.8955 - val_loss: 0.2742
```

In [14]:

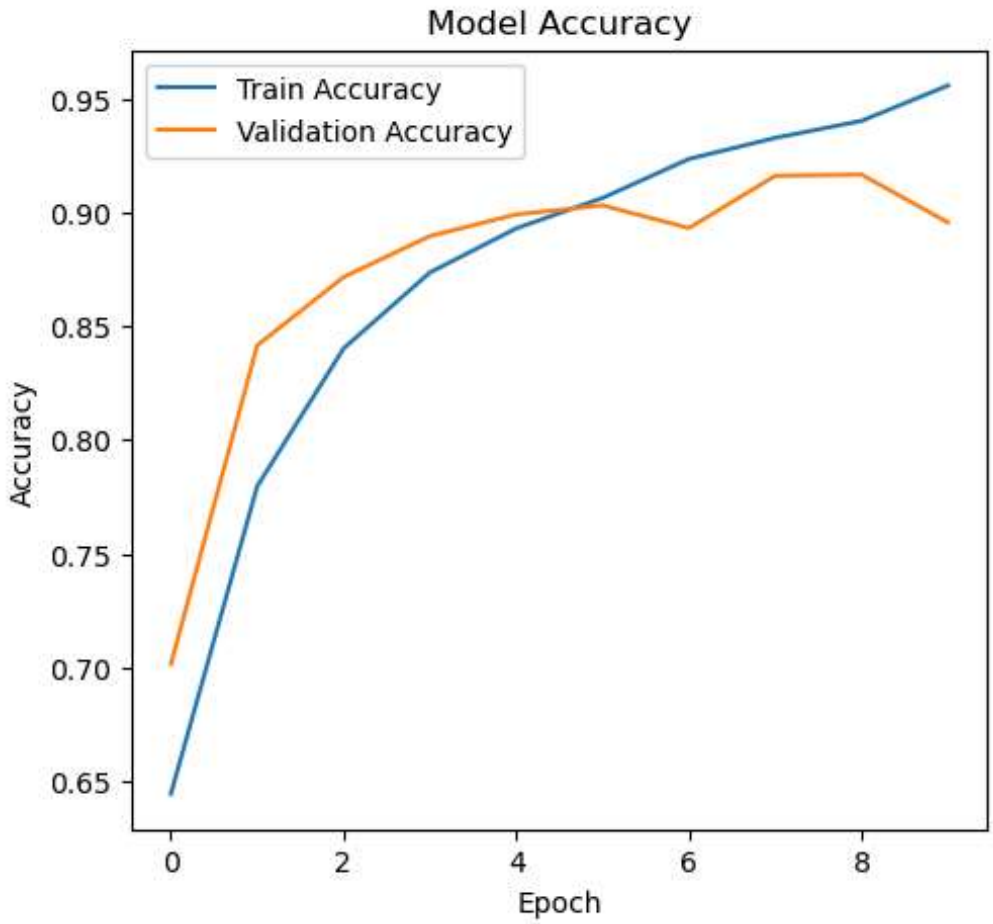
```
# Step 9: Evaluate the model on the test set
# Evaluate the trained model on the test dataset and print the loss and a
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}')
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.8765 - loss: 0.3407
Test Loss: 0.3123, Test Accuracy: 0.8855
```

In [15]:

```
# Plot training & validation accuracy values
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
```

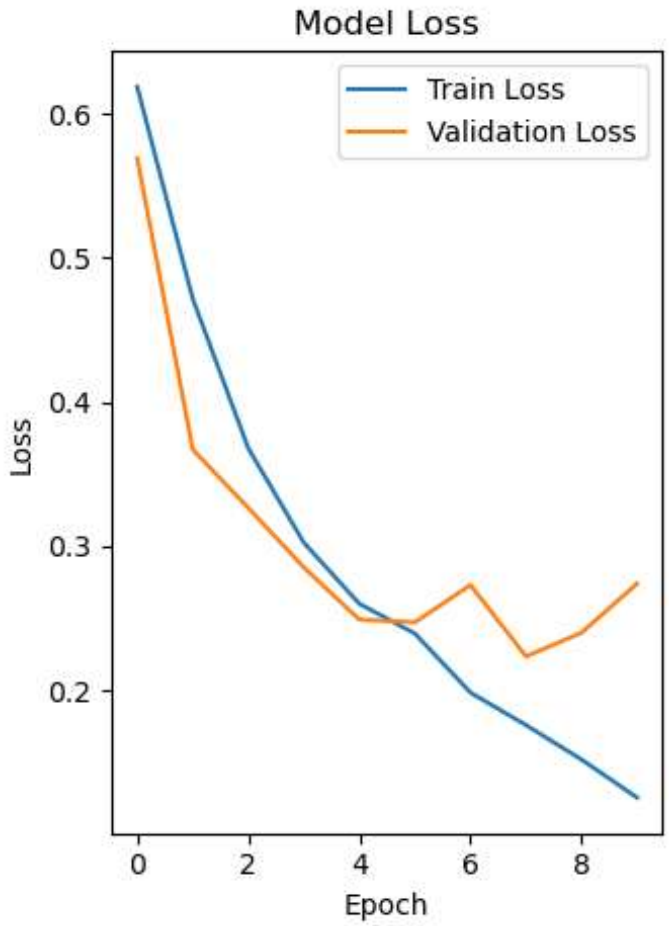
Out[15]: <matplotlib.legend.Legend at 0x208525e4160>



In [16]:

```
# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()

plt.tight_layout()
plt.show()
```



In [23]:

```python
# Import necessary libraries for visualization
import matplotlib.pyplot as plt
import numpy as np

# Define a function to display the image and prediction result
def display_sample_prediction(index):
    # Get the test image and corresponding label
    image = x_test[index]
    true_label = y_test[index]

    # Predict the class using the trained model
    prediction = model.predict(np.expand_dims(image, axis=0))

    # Convert the prediction to a class label (0 or 1)
    predicted_label = np.argmax(prediction, axis=1)[0]

    # Map labels to the actual categories (horizontal/vertical)
    class_names = ['Horizontal (Airplane)', 'Vertical (Ship)']

    # Plot the image
    plt.imshow(image)
    plt.title(f"True: {class_names[np.argmax(true_label)]}\nPredicted: {c
    plt.axis('off')
    plt.show()

# Pick any index from the test set (you can modify this index)
sample_index = 10
display_sample_prediction(sample_index)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
```