Implementation of a Convolutional Neural Network (CNN) for image classification

We will be using Keras (a high-level API built on top of TensorFlow) to build the CNN. We will train the model on the CIFAR-10 dataset, a popular image dataset, which contains 60,000 32 x 32 colour images in 10 classes, with 6,000 images per class.

1. Importing necessary libraries

We begin by importing the libraries required for our analysis: `tensorflow` for building and training the neural network, and `matplotlib.pyplot` for visualising the dataset and training results.

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
```

2. Loading and preprocessing the image dataset

The dataset is loaded using `cifar10.load_data()`, which returns training and testing sets. We normalise the pixel values by dividing by 255 so that the values are between 0 and 1 (which helps the model train better). Moreover, the target labels are one-hot encoded using `to_categorical()`, which converts the labels to a binary matrix format for multi-class classification.

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

3. Visualising the dataset

Now, we display the first 9 images of the training dataset, with each image being shown with its label. The `argmax()` function gives the index of the class with the highest probability in the one-hot encoded label.

```
fig, axes = plt.subplots(3, 3, figsize=(6, 6))

for i, ax in enumerate(axes.flat):
  ax.imshow(x_train[i])
  ax.set_title(f'Label: {y_train[i].argmax()}')
  ax.axis('off')

plt.show()
```

4. Building the CNN

We build the CNN model as a sequential network where each layer is added one after another.

- `Conv2D` Layer: It applies 32 filters of size 3 x 3 to the input image, using ReLU (Rectified Linear Unit) as the activation function.
- `MaxPooling2D` Layer: It reduces the spatial dimensions of the input (downsampling) by taking the maximum value over 2 x 2 patches.
- `Flatten` Layer: It converts the 2D matrix from the previous layer into a 1D vector to feed into the dense layer.
- `Dense` Layer: This is a fully connected layer with 64 neurons. It helps the model learn higher-level representations.
- `Output` Layer: It uses a softmax activation function to predict a probability distribution over the 10 classes.

```
model = models.Sequential()

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.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))
```

Show hidden output

5. Compiling the model

The model is compiled with the Adam optimiser, which is commonly used for training deep learning models. Categorical cross-entropy is used as the loss function since we are doing multi-class classification. Moreover, accuracy is used as the evaluation metric.

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

6. Training the model

Next, the model is trained for 10 epochs (iterations through the full dataset). We pass both the training data and validation data so that the model's performance is evaluated after each epoch on unseen data.

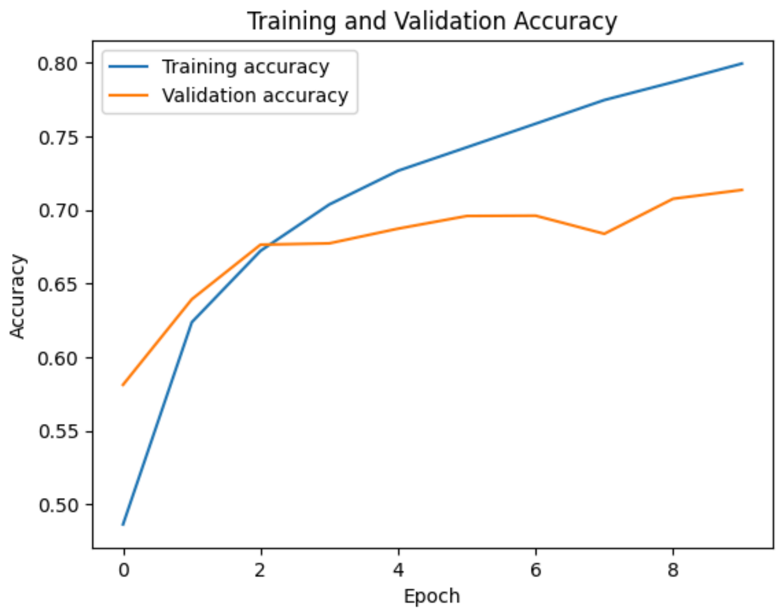
```
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 68s 43ms/step - accuracy: 0.3910 - loss: 1.6740 - val_accuracy: 0.5812 - val_loss: 1.2002
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 42ms/step - accuracy: 0.6102 - loss: 1.1075 - val_accuracy: 0.6393 - val_loss: 1.0317
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 66s 42ms/step - accuracy: 0.6669 - loss: 0.9573 - val_accuracy: 0.6764 - val_loss: 0.9427
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 79s 41ms/step - accuracy: 0.7014 - loss: 0.8617 - val_accuracy: 0.6773 - val_loss: 0.9345
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 40ms/step - accuracy: 0.7263 - loss: 0.7904 - val_accuracy: 0.6873 - val_loss: 0.9175
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 42ms/step - accuracy: 0.7435 - loss: 0.7358 - val_accuracy: 0.6959 - val_loss: 0.9057
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 42ms/step - accuracy: 0.7656 - loss: 0.6773 - val_accuracy: 0.6961 - val_loss: 0.9237
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 41ms/step - accuracy: 0.7804 - loss: 0.6330 - val_accuracy: 0.6838 - val_loss: 0.9653
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 40ms/step - accuracy: 0.7939 - loss: 0.5902 - val_accuracy: 0.7076 - val_loss: 0.9063
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 65s 41ms/step - accuracy: 0.8063 - loss: 0.5583 - val_accuracy: 0.7136 - val_loss: 0.9027
```

7. Visualising training history

After training, we plot the training and validation accuracy over the epochs to visualise the model's performance. The plot shows how the model improves in accuracy over time, and whether there is any overfitting (if validation accuracy starts to decrease while training accuracy increases).

```
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



8. Evaluating the model

The model is evaluated on the test set using the `evaluate()` function, which returns the loss and accuracy. This gives an indication of how well the model generalises to unseen data.

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.7158 - loss: 0.8883
Test accuracy: 0.7135999798774719
```

9. Making predictions

Finally, the model makes predictions on the test data. Each prediction is a probability distribution over the 10 classes, and `argmax()` returns the index of the class with the highest predicted probability.

```
predictions = model.predict(x_test)
print(f'Predicted class for first image: {predictions[0].argmax()}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step
Predicted class for first image: 3
```