2347215 Arunoth Symen A

Lab Program - 3

1. Data Preprocessing

In [ ]:
```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load the dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the pixel values to [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert class labels to one-hot encoding
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 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
```

Data Augmentation: To improve generalization, apply data augmentation techniques.

In [ ]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    horizontal_flip=True,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1
)
datagen.fit(x_train)
```

2. Network Architecture Design

Architecture Justification:

Input Layer: The CIFAR-10 images have a shape of 32x32x3 (RGB color).

Hidden Layers: ● Convolutional layers are used to capture spatial hierarchies and local patterns from the images.

● MaxPooling layers are used for down-sampling, reducing spatial dimensions.

● Dense layers at the end for classification.

Output Layer: Use a softmax activation function to predict the 10 classes.

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
```

```python
# Input layer with Conv and MaxPooling
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())

# Fully connected layers
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:
107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape
conv2d (Conv2D)	(None, 30, 30, 32)
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)
conv2d_1 (Conv2D)	(None, 13, 13, 64)
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)
flatten (Flatten)	(None, 2304)
dense (Dense)	(None, 128)
dropout (Dropout)	(None, 128)
dense_1 (Dense)	(None, 10)

Total params: 315,722 (1.20 MB)

Trainable params: 315,722 (1.20 MB)

Non-trainable params: 0 (0.00 B)

3. Activation Functions

● ReLU (Rectified Linear Unit): Helps to solve the vanishing gradient problem and allows faster training. It's popular in convolutional layers.

● Softmax: Used in the final layer to output class probabilities for multi-class classification.

4. Loss Function and Optimizer

For this classification problem:

- Use categorical crossentropy since it is suited for multi-class classification problems.
- Compare mean squared error and categorical hinge with categorical crossentropy.

In [ ]:
```
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.optimizers import Adam

# Compile model with categorical crossentropy and Adam optimizer
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

5. Training the Model

In [ ]:
```
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=50,
                    validation_data=(x_test, y_test))
```

```
Epoch 1/50
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_
adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**
kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing
`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignore
d.
  self._warn_if_super_not_called()
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 107s 129ms/step - accuracy: 0.2891 - loss: 1.9154 - val
_accuracy: 0.5128 - val_loss: 1.3687
Epoch 2/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 129s 113ms/step - accuracy: 0.4462 - loss: 1.5237 - val
_accuracy: 0.5690 - val_loss: 1.2181
Epoch 3/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 114ms/step - accuracy: 0.5029 - loss: 1.3841 - val
_accuracy: 0.5837 - val_loss: 1.1997
Epoch 4/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 144s 116ms/step - accuracy: 0.5285 - loss: 1.3178 - val
_accuracy: 0.6245 - val_loss: 1.0743
Epoch 5/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 116ms/step - accuracy: 0.5587 - loss: 1.2513 - val
_accuracy: 0.6237 - val_loss: 1.0612
Epoch 6/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.5700 - loss: 1.2211 - val_
accuracy: 0.6498 - val_loss: 1.0108
Epoch 7/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 114ms/step - accuracy: 0.5787 - loss: 1.1920 - val
_accuracy: 0.6394 - val_loss: 1.0251
Epoch 8/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 143s 116ms/step - accuracy: 0.5914 - loss: 1.1588 - val
_accuracy: 0.6683 - val_loss: 0.9551
Epoch 9/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 114ms/step - accuracy: 0.6007 - loss: 1.1381 - val_
accuracy: 0.6697 - val_loss: 0.9527
Epoch 10/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6071 - loss: 1.1152 - val_
accuracy: 0.6678 - val_loss: 0.9596
Epoch 11/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 113ms/step - accuracy: 0.6200 - loss: 1.0963 - val_
accuracy: 0.6938 - val_loss: 0.8904
Epoch 12/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.6246 - loss: 1.0712 - val_
accuracy: 0.6746 - val_loss: 0.9408
Epoch 13/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 113ms/step - accuracy: 0.6269 - loss: 1.0660 - val_
accuracy: 0.7007 - val_loss: 0.8703
Epoch 14/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 114ms/step - accuracy: 0.6351 - loss: 1.0430 - val_
accuracy: 0.6892 - val_loss: 0.9041
Epoch 15/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.6349 - loss: 1.0311 - val_
accuracy: 0.6968 - val_loss: 0.8652
Epoch 16/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 115ms/step - accuracy: 0.6445 - loss: 1.0155 - val
_accuracy: 0.6886 - val_loss: 0.8933
Epoch 17/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 88s 113ms/step - accuracy: 0.6512 - loss: 1.0078 - val_
accuracy: 0.6971 - val_loss: 0.8716
Epoch 18/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.6507 - loss: 0.9973 - val_
accuracy: 0.7067 - val_loss: 0.8509
Epoch 19/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6516 - loss: 1.0012 - val_
accuracy: 0.7019 - val_loss: 0.8609
```

```
Epoch 20/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 116ms/step - accuracy: 0.6580 - loss: 0.9848 - val
_accuracy: 0.6988 - val_loss: 0.8683
Epoch 21/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 116ms/step - accuracy: 0.6610 - loss: 0.9816 - val
_accuracy: 0.7112 - val_loss: 0.8487
Epoch 22/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 115ms/step - accuracy: 0.6614 - loss: 0.9735 - val
_accuracy: 0.7166 - val_loss: 0.8232
Epoch 23/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 115ms/step - accuracy: 0.6681 - loss: 0.9534 - val
_accuracy: 0.7098 - val_loss: 0.8229
Epoch 24/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 114ms/step - accuracy: 0.6672 - loss: 0.9468 - val_
accuracy: 0.7141 - val_loss: 0.8212
Epoch 25/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 144s 116ms/step - accuracy: 0.6739 - loss: 0.9450 - val
_accuracy: 0.7376 - val_loss: 0.7612
Epoch 26/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6668 - loss: 0.9566 - val_
accuracy: 0.7281 - val_loss: 0.8048
Epoch 27/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 140s 113ms/step - accuracy: 0.6740 - loss: 0.9375 - val
_accuracy: 0.7173 - val_loss: 0.8179
Epoch 28/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 114ms/step - accuracy: 0.6767 - loss: 0.9394 - val_
accuracy: 0.7297 - val_loss: 0.7824
Epoch 29/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 114ms/step - accuracy: 0.6743 - loss: 0.9294 - val
_accuracy: 0.7275 - val_loss: 0.7874
Epoch 30/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6806 - loss: 0.9176 - val_
accuracy: 0.7171 - val_loss: 0.8286
Epoch 31/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 114ms/step - accuracy: 0.6818 - loss: 0.9181 - val_
accuracy: 0.7223 - val_loss: 0.7997
Epoch 32/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6854 - loss: 0.9064 - val_
accuracy: 0.7381 - val_loss: 0.7556
Epoch 33/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 115ms/step - accuracy: 0.6869 - loss: 0.9025 - val
_accuracy: 0.7375 - val_loss: 0.7722
Epoch 34/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 144s 117ms/step - accuracy: 0.6894 - loss: 0.8964 - val
_accuracy: 0.7367 - val_loss: 0.7483
Epoch 35/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 116ms/step - accuracy: 0.6931 - loss: 0.8825 - val
_accuracy: 0.7152 - val_loss: 0.8165
Epoch 36/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 113ms/step - accuracy: 0.6948 - loss: 0.8957 - val_
accuracy: 0.7264 - val_loss: 0.7956
Epoch 37/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 145s 117ms/step - accuracy: 0.6955 - loss: 0.8871 - val
_accuracy: 0.7392 - val_loss: 0.7570
Epoch 38/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.6970 - loss: 0.8824 - val_
```

```
accuracy: 0.7477 - val_loss: 0.7344
Epoch 39/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6954 - loss: 0.8767 - val_
accuracy: 0.7443 - val_loss: 0.7535
Epoch 40/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 92s 117ms/step - accuracy: 0.6940 - loss: 0.8828 - val_
accuracy: 0.7490 - val_loss: 0.7302
Epoch 41/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 116ms/step - accuracy: 0.6985 - loss: 0.8776 - val
_accuracy: 0.7342 - val_loss: 0.7706
Epoch 42/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 89s 113ms/step - accuracy: 0.6923 - loss: 0.8788 - val_
accuracy: 0.7443 - val_loss: 0.7591
Epoch 43/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 116ms/step - accuracy: 0.6932 - loss: 0.8743 - val_
accuracy: 0.7467 - val_loss: 0.7177
Epoch 44/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 91s 117ms/step - accuracy: 0.6972 - loss: 0.8711 - val_
accuracy: 0.7476 - val_loss: 0.7366
Epoch 45/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 90s 115ms/step - accuracy: 0.6966 - loss: 0.8671 - val_
accuracy: 0.7374 - val_loss: 0.7619
Epoch 46/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 114ms/step - accuracy: 0.7016 - loss: 0.8559 - val
_accuracy: 0.7523 - val_loss: 0.7264
Epoch 47/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 114ms/step - accuracy: 0.7007 - loss: 0.8574 - val
_accuracy: 0.7520 - val_loss: 0.7237
Epoch 48/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 92s 117ms/step - accuracy: 0.7042 - loss: 0.8579 - val_
accuracy: 0.7397 - val_loss: 0.7525
Epoch 49/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 141s 116ms/step - accuracy: 0.7047 - loss: 0.8431 - val
_accuracy: 0.7317 - val_loss: 0.7810
Epoch 50/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 144s 118ms/step - accuracy: 0.7080 - loss: 0.8409 - val
_accuracy: 0.7594 - val_loss: 0.7135
```

6. Model Evaluation

In [ ]:

```python
from sklearn.metrics import classification_report, confusion_matrix

# Evaluate model
y_pred = model.predict(x_test)
y_pred_classes = y_pred.argmax(axis=1)
y_true = y_test.argmax(axis=1)

# Classification report
print(classification_report(y_true, y_pred_classes))

# Confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred_classes)
print(conf_matrix)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step
              precision    recall  f1-score   support

           0       0.76      0.84      0.80      1000
           1       0.82      0.92      0.87      1000
           2       0.79      0.57      0.66      1000
           3       0.65      0.47      0.55      1000
           4       0.75      0.70      0.72      1000
           5       0.68      0.64      0.66      1000
           6       0.79      0.83      0.81      1000
           7       0.69      0.89      0.78      1000
           8       0.85      0.85      0.85      1000
           9       0.79      0.88      0.83      1000

    accuracy                           0.76     10000
   macro avg       0.76      0.76      0.75     10000
weighted avg       0.76      0.76      0.75     10000

[[841  27  13   6   5   1   6  14  40  47]
 [ 10 919   1   2   0   1   5   1  12  49]
 [ 86   9 566  28  88  58  65  66  14  20]
 [ 23  15  48 474  60 164  71  69  34  42]
 [ 26   4  31  25 701  22  42 130  12   7]
 [ 14   6  21 139  30 644  28  88  13  17]
 [ 14  12  28  31  33  19 835  12   7   9]
 [ 16   5  11  12  20  30   3 885   2  16]
 [ 51  52   0   7   0   1   1   5 850  33]
 [ 20  66   1   2   1   1   6   6  18 879]]
```

7. Optimization Strategies

● Early Stopping: Stop training when validation loss stops decreasing to avoid overfitting.

● Learning Rate Scheduling: Gradually reduce the learning rate for smoother convergence.

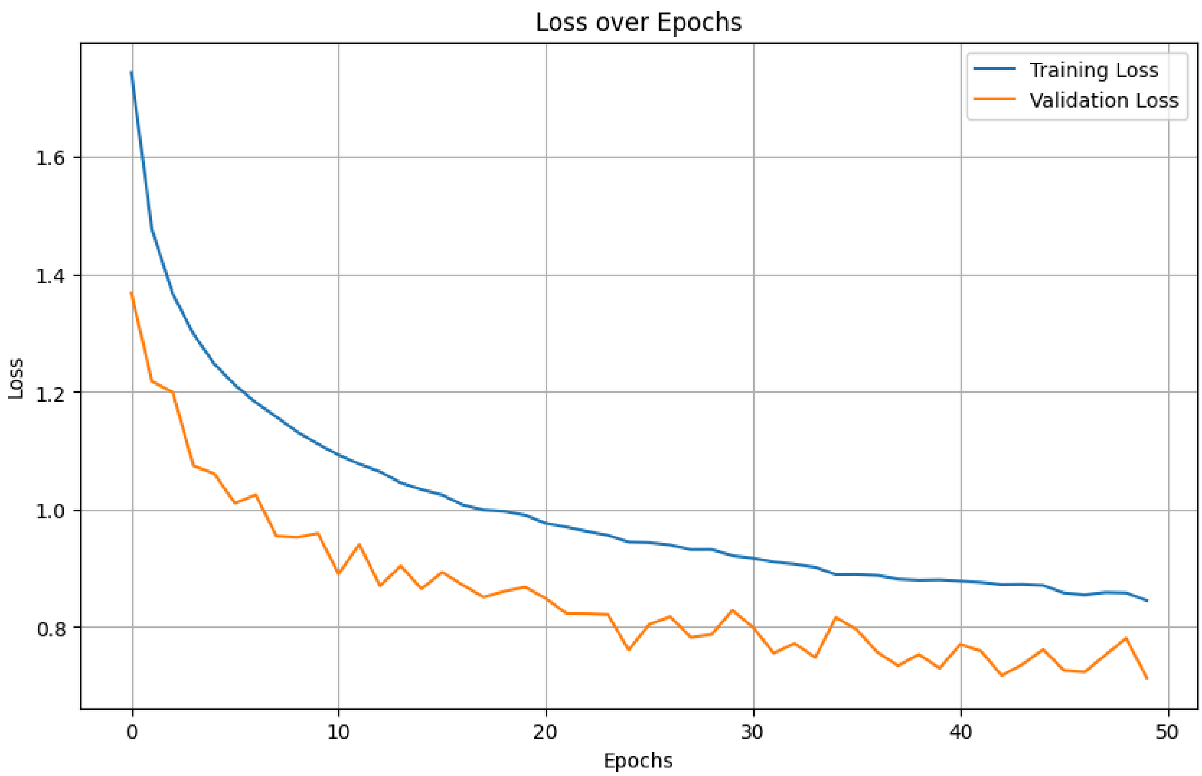
● Weight Initialization: Proper initialization (e.g., Xavier or He initialization) avoids vanishing/exploding gradients and ensures efficient training.

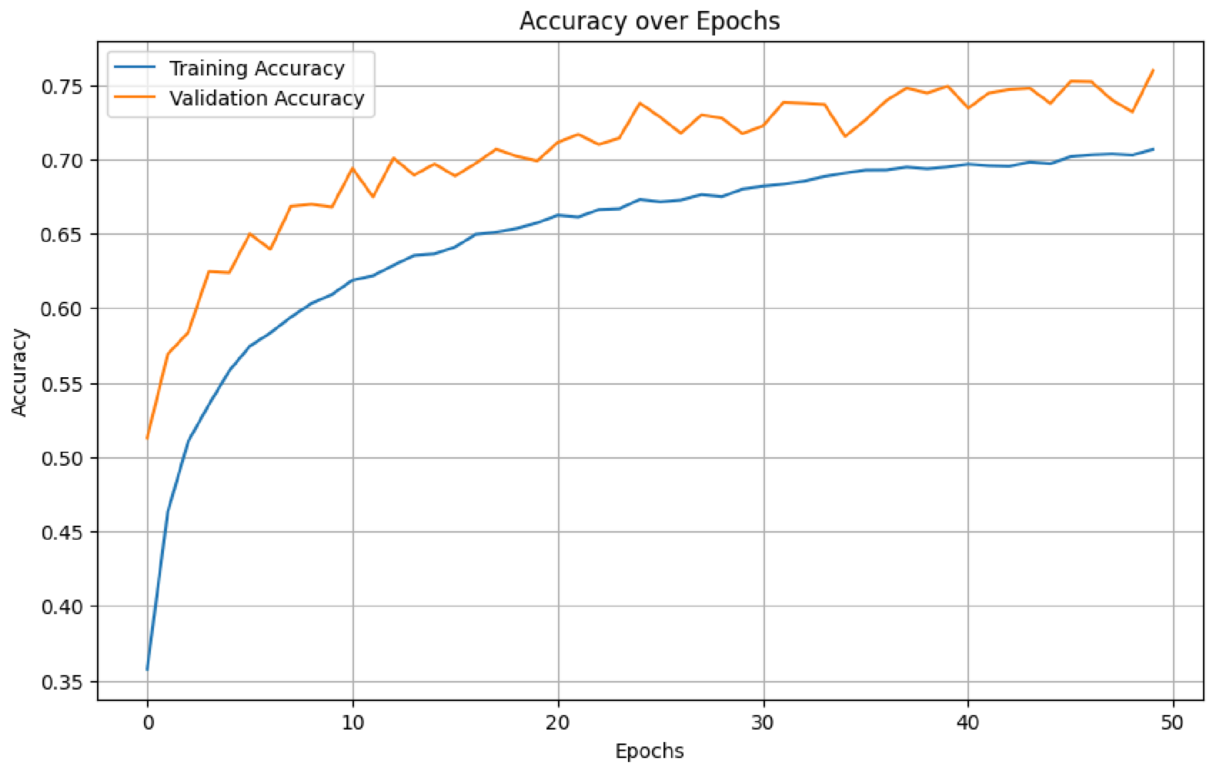
Weight Initialization Importance Good weight initialization speeds up convergence by ensuring that activations are properly distributed across layers. Without proper initialization, the network can get stuck in a local minimum.

8. Report

In [ ]:
```
import matplotlib.pyplot as plt

# Plot the training loss and validation loss
def plot_loss(history):
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
```

```
    plt.show()

# Plot the training accuracy and validation accuracy
def plot_accuracy(history):
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Accuracy over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

# Visualize
plot_loss(history)
plot_accuracy(history)
```





Here are some of the key elements you would include in a report:

● Architecture: Describe and justify the layers, number of filters, and activation functions. ● Training Results: Plot loss and accuracy over epochs for training and validation.

● Hyperparameters: List the values for learning rate, batch size, and number of epochs.

● Challenges and Solutions: Mention challenges like overfitting, slow convergence, or vanishing gradients, and how you addressed them (e.g., by using regularization, adjusting learning rate, or fine-tuning the architecture).