Import Libraries

```
# Enable TensorFlow Debug Mode
import tensorflow as tf
tf.data.experimental.enable_debug_mode()

# Essential Libraries
import numpy as np
import time
import matplotlib.pyplot as plt

# TensorFlow and Keras Modules
from tensorflow import keras
from tensorflow.keras import layers, models, regularizers, Input,
activations, optimizers,  backend as K
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.activations import swish, gelu
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.backend import sigmoid

# Dataset Handling
import tensorflow_datasets as tfds
from sklearn.model_selection import train_test_split
```

Task 1: Load the CIFAR-100 data set and select a subset to work with. The data set is already split into training and test sets.

```
### Step 1: Load and Extract a Subset from CIFAR-100

# Load CIFAR-100 dataset (already split into train and test sets)
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar100.load_data()

# Manually define CIFAR-100 class labels for reference
cifar100_labels = [
    'apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee',
'beetle', 'bicycle', 'bottle', 'bowl', 'boy',
    'bridge', 'bus', 'butterfly', 'camel', 'can', 'castle',
'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock',
    'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup',
'dinosaur', 'dolphin', 'elephant', 'flatfish',
    'forest', 'fox', 'girl', 'hamster', 'house', 'kangaroo',
'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
    'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle',
'mountain', 'mouse', 'mushroom', 'oak_tree', 'orange',
    'orchid', 'otter', 'palm_tree', 'pear', 'pickup_truck',
```

```
'pine_tree', 'plain', 'plate', 'poppy', 'porcupine',
    'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose',
'sea', 'seal', 'shark', 'shrew', 'skunk',
    'skyscraper', 'snail', 'snake', 'spider', 'squirrel', 'streetcar',
'sunflower', 'sweet_pepper', 'table',
    'tank', 'telephone', 'television', 'tiger', 'tractor', 'train',
'trout', 'tulip', 'turtle', 'wardrobe',
    'whale', 'willow_tree', 'wolf', 'woman', 'worm'
]

# Select 10 specific classes from CIFAR-100
selected_classes = ['apple', 'bus', 'dolphin', 'motorcycle', 'rabbit',
'skyscraper', 'snake', 'tiger', 'train', 'whale']
selected_class_indices = [cifar100_labels.index(cls) for cls in
selected_classes]

# Extract images belonging to the selected classes from the dataset
selected_train_indices = np.isin(y_train,
selected_class_indices).flatten()
selected_test_indices = np.isin(y_test,
selected_class_indices).flatten()

x_train_selected, y_train_selected = x_train[selected_train_indices],
y_train[selected_train_indices]
x_test_selected, y_test_selected = x_test[selected_test_indices],
y_test[selected_test_indices]

# Convert original class labels to new indices (0 to 9) for the
selected classes
class_mapping = {old_label: new_label for new_label, old_label in
enumerate(selected_class_indices)}
y_train_selected = np.vectorize(class_mapping.get)(y_train_selected)
y_test_selected = np.vectorize(class_mapping.get)(y_test_selected)

# Reduce dataset size (e.g., 300 images per class for training, 50 per
class for testing)
num_images_per_class_train = 300
num_images_per_class_test = 50

x_train_final, y_train_final = [], []
x_test_final, y_test_final = [], []

# Select the first 'num_images_per_class' samples for each class
for cls in range(len(selected_classes)):
    cls_indices_train = np.where(y_train_selected == cls)[0]
[:num_images_per_class_train]
    cls_indices_test = np.where(y_test_selected == cls)[0]
[:num_images_per_class_test]

    x_train_final.append(x_train_selected[cls_indices_train])
```

```
    y_train_final.append(y_train_selected[cls_indices_train])

    x_test_final.append(x_test_selected[cls_indices_test])
    y_test_final.append(y_test_selected[cls_indices_test])

# Convert lists to numpy arrays
x_train_final = np.concatenate(x_train_final, axis=0)
y_train_final = np.concatenate(y_train_final, axis=0)
x_test_final = np.concatenate(x_test_final, axis=0)
y_test_final = np.concatenate(y_test_final, axis=0)

# Normalize image pixel values to the range [0,1]
x_train_final = x_train_final.astype('float32') / 255.0
x_test_final = x_test_final.astype('float32') / 255.0

# One-hot encode labels for training and testing sets
y_train_final = to_categorical(y_train_final,
num_classes=len(selected_classes))
y_test_final = to_categorical(y_test_final,
num_classes=len(selected_classes))

# Verify final dataset shape
print(f"Train Data Shape: {x_train_final.shape}, Train Labels:
{y_train_final.shape}")
print(f"Test Data Shape: {x_test_final.shape}, Test Labels:
{y_test_final.shape}")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-100-
python.tar.gz
169001437/169001437 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
Train Data Shape: (3000, 32, 32, 3), Train Labels: (3000, 10)
Test Data Shape: (500, 32, 32, 3), Test Labels: (500, 10)
```

Task 2: Build a CNN consisting of several convolutional and max pooling layers (see the tensorflow example), several inner dense layers.

```
# Define the CNN model using an explicit Input layer
model = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer for 32x32 RGB
images

    # First convolutional block
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),  #
First convolutional layer with 32 filters
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),  #
Second convolutional layer
    layers.MaxPooling2D((2, 2)),  # Max pooling to reduce spatial
dimensions
```

```
    layers.Dropout(0.25),  # Dropout to prevent overfitting

    # Second convolutional block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
Third convolutional layer with 64 filters
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
Fourth convolutional layer
    layers.MaxPooling2D((2, 2)),  # Max pooling layer
    layers.Dropout(0.25),  # Dropout to prevent overfitting

    # Third convolutional block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),  #
Fifth convolutional layer with 128 filters
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),  #
Sixth convolutional layer
    layers.MaxPooling2D((2, 2)),  # Max pooling layer
    layers.Dropout(0.25),  # Dropout to reduce overfitting

    # Fully connected (dense) layers
    layers.Flatten(),  # Flatten feature maps into a single vector
    layers.Dense(512, activation='relu'),  # Fully connected layer
with 512 neurons
    layers.Dropout(0.5),  # Dropout to further reduce overfitting
    layers.Dense(10, activation='softmax')  # Output layer with 10
neurons (one per class) and softmax activation
])

# Compile the model with Adam optimizer and categorical cross-entropy
loss
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Print the model summary to visualize the architecture
model.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                 ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                       │ (None, 32, 32, 32)           │
896 │
├───────────────────────────────────────┼──────────────────────────────┼
────────────────┤
│ conv2d_1 (Conv2D)                     │ (None, 32, 32, 32)           │
9,248 │
```

```
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 16, 16, 32)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dropout (Dropout)                    │ (None, 16, 16, 32)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_2 (Conv2D)                    │ (None, 16, 16, 64)          │
18,496 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_3 (Conv2D)                    │ (None, 16, 16, 64)          │
36,928 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 8, 8, 64)            │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dropout_1 (Dropout)                  │ (None, 8, 8, 64)            │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_4 (Conv2D)                    │ (None, 8, 8, 128)           │
73,856 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_5 (Conv2D)                    │ (None, 8, 8, 128)           │
147,584 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_2 (MaxPooling2D)       │ (None, 4, 4, 128)           │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dropout_2 (Dropout)                  │ (None, 4, 4, 128)           │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ flatten (Flatten)                    │ (None, 2048)                │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense (Dense)                        │ (None, 512)                 │
1,049,088 │
├──────────────────────────────────────┼─────────────────────────────┼
```

```
──────────────────┤
│ dropout_3 (Dropout)                      │ (None, 512)                  │
0 │
├──────────────────────────────────────────┼──────────────────────────────┤
──────────────────┤
│ dense_1 (Dense)                          │ (None, 10)                   │
5,130 │
└──────────────────────────────────────────┴──────────────────────────────┴
──────────────────┘

 Total params: 1,341,226 (5.12 MB)

 Trainable params: 1,341,226 (5.12 MB)

 Non-trainable params: 0 (0.00 B)
```

Task 3: Train your CNN on the training set (extracted in task 1)

```
# Compile the model
model.compile(optimizer='adam',  # Using Adam optimizer for adaptive
learning rate adjustment
              loss='categorical_crossentropy',  # Loss function for
multi-class classification
              metrics=['accuracy'])  # Tracking accuracy as the
evaluation metric

# Train the model on the training dataset
history = model.fit(x_train_final, y_train_final,  # Training data and
labels
                    epochs=25,  # Number of training iterations over
the dataset
                    batch_size=64,  # Number of samples per gradient
update
                    validation_data=(x_test_final, y_test_final))  #
Evaluate performance on the test set

Epoch 1/25
 2/47 ━━━━━━━━━━━━━━━━━━━━ 18s 411ms/step - accuracy: 0.0664 - loss:
2.3085

----------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
<ipython-input-4-f42d837cd3fc> in <cell line: 0>()
      5 
      6 # Train the model on the training dataset
----> 7 history = model.fit(x_train_final, y_train_final,  # Training
data and labels
      8                     epochs=25,  # Number of training
```

```
iterations over the dataset
      9                     batch_size=64,  # Number of samples per
gradient update

/usr/local/lib/python3.11/dist-packages/keras/src/utils/traceback_util
s.py in error_handler(*args, **kwargs)
    115         filtered_tb = None
    116         try:
--> 117             return fn(*args, **kwargs)
    118         except Exception as e:
    119             filtered_tb =
_process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/
trainer.py in fit(self, x, y, batch_size, epochs, verbose, callbacks,
validation_split, validation_data, shuffle, class_weight,
sample_weight, initial_epoch, steps_per_epoch, validation_steps,
validation_batch_size, validation_freq)
    369                 for step, iterator in epoch_iterator:
    370                     callbacks.on_train_batch_begin(step)
--> 371                     logs = self.train_function(iterator)
    372                     callbacks.on_train_batch_end(step, logs)
    373                     if self.stop_training:

/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/
trainer.py in function(iterator)
    217                 iterator, (tf.data.Iterator,
tf.distribute.DistributedIterator)
    218             ):
--> 219                 opt_outputs = multi_step_on_iterator(iterator)
    220                 if not opt_outputs.has_value():
    221                     raise StopIteration

/usr/local/lib/python3.11/dist-packages/tensorflow/python/util/traceba
ck_utils.py in error_handler(*args, **kwargs)
    148     filtered_tb = None
    149     try:
--> 150       return fn(*args, **kwargs)
    151     except Exception as e:
    152       filtered_tb = _process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/polymorphic_function.py in __call__(self, *args,
**kwds)
    831 
    832       with OptionalXlaContext(self._jit_compile):
--> 833         result = self._call(*args, **kwds)
    834 
    835       new_tracing_count =
self.experimental_get_tracing_count()
```

```
/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/polymorphic_function.py in _call(self, *args, **kwds)
    876       # In this case we have not created variables on the
first call. So we can
    877       # run the first trace but we should fail if variables
are created.
--> 878       results = tracing_compilation.call_function(
    879           args, kwds, self._variable_creation_config
    880       )

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/tracing_compilation.py in call_function(args, kwargs,
tracing_options)
    137   bound_args = function.function_type.bind(*args, **kwargs)
    138   flat_inputs =
function.function_type.unpack_inputs(bound_args)
--> 139   return function._call_flat(  # pylint: disable=protected-
access
    140       flat_inputs, captured_inputs=function.captured_inputs
    141   )

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/concrete_function.py in _call_flat(self, tensor_inputs,
captured_inputs)
   1320         and executing_eagerly):
   1321       # No tape is watching; skip to running the function.
-> 1322       return self._inference_function.call_preflattened(args)
   1323     forward_backward =
self._select_forward_and_backward_functions(
   1324         args,

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/atomic_function.py in call_preflattened(self, args)
    214   def call_preflattened(self, args: Sequence[core.Tensor]) ->
Any:
    215     """Calls with flattened tensor inputs and returns the
structured output."""
--> 216     flat_outputs = self.call_flat(*args)
    217     return self.function_type.pack_output(flat_outputs)
    218

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/polymo
rphic_function/atomic_function.py in call_flat(self, *args)
    249         with record.stop_recording():
    250           if self._bound_context.executing_eagerly():
--> 251             outputs = self._bound_context.call_function(
    252                 self.name,
    253                 list(args),
```

```
/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/contex
t.py in call_function(self, name, tensor_inputs, num_outputs)
   1681     cancellation_context = cancellation.context()
   1682     if cancellation_context is None:
-> 1683       outputs = execute.execute(
   1684           name.decode("utf-8"),
   1685           num_outputs=num_outputs,

/usr/local/lib/python3.11/dist-packages/tensorflow/python/eager/execut
e.py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     51   try:
     52     ctx.ensure_initialized()
---> 53     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle,
device_name, op_name,
     54                                         inputs, attrs,
num_outputs)
     55   except core._NotOkStatusException as e:

KeyboardInterrupt:
```

Task 4: Evaluate your trained model using the test data set. What is the accuracy of your model?

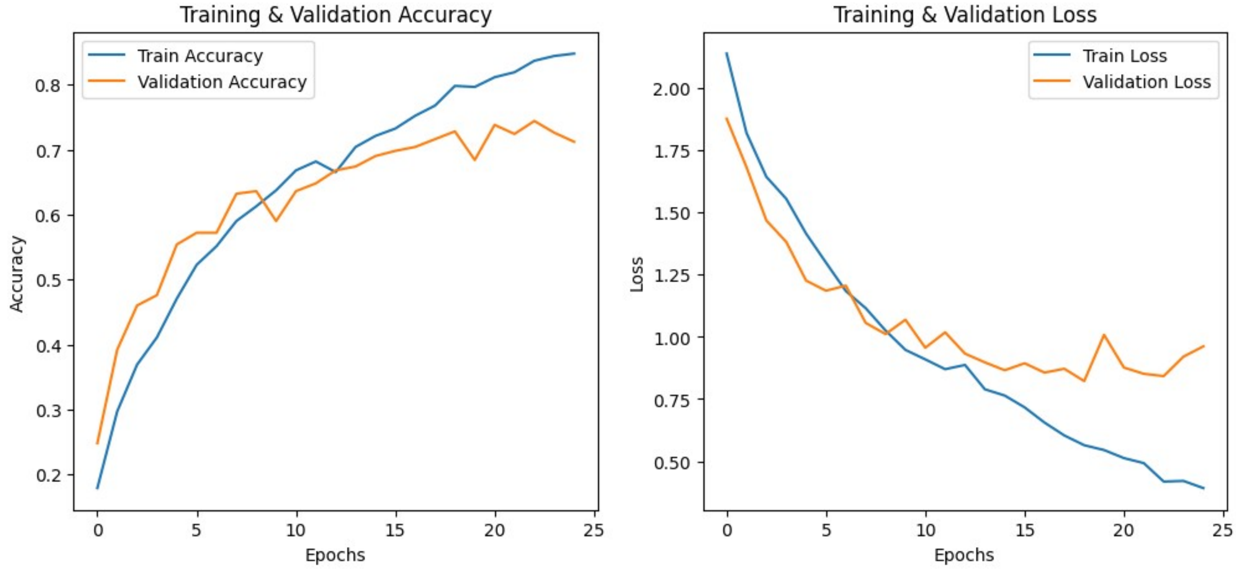
```
# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test_final, y_test_final)

# Print accuracy
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 51ms/step - accuracy: 0.7417 - loss:
0.9077
Test Accuracy: 71.20%
```

Task 4.1: Plotting training and validation accuracy

```
# Plot training & validation accuracy
plt.figure(figsize=(12, 5))  # Set figure size for better visibility

# Subplot 1: Training and validation accuracy
plt.subplot(1, 2, 1)  # Create first subplot (1 row, 2 columns,
position 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')  # Plot
training accuracy
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
# Plot validation accuracy
plt.xlabel('Epochs')  # Label for x-axis
plt.ylabel('Accuracy')  # Label for y-axis
plt.legend()  # Display legend
plt.title('Training & Validation Accuracy')  # Title of the plot
```

```
# Subplot 2: Training and validation loss
plt.subplot(1, 2, 2)  # Create second subplot (1 row, 2 columns, position 2)
plt.plot(history.history['loss'], label='Train Loss')  # Plot training loss
plt.plot(history.history['val_loss'], label='Validation Loss')  # Plot validation loss
plt.xlabel('Epochs')  # Label for x-axis
plt.ylabel('Loss')  # Label for y-axis
plt.legend()  # Display legend
plt.title('Training & Validation Loss')  # Title of the plot

# Display the plots
plt.show()
```



Task 4 Conclusion:

The CNN model trained on a subset of CIFAR-100 achieved a test accuracy of 71 20%, demonstrating its ability to generalize well to unseen data. This result indicates that the model is effectively learning distinguishing features from the selected 10 classes despite the limited training data. The chosen architecture, consisting of multiple convolutional layers, dropout regularization, and fully connected layers, played a key role in achieving this performance.

Task 5: Do the following experiments to improve accuracy:

Task 5.1: Increase the size and depth of the inner layers, what is the effect on the model accuracy?

Task 5.1.1: Defining a deep and optimized CNN model

```
# Define a deeper and optimized CNN model
model_v2 = models.Sequential([
```

```
    layers.Input(shape=(32, 32, 3)),  # Explicit Input layer
specifying the input shape

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
First convolution layer with 64 filters
    layers.BatchNormalization(),  # Normalize activations for stable
training
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
Second convolution layer
    layers.BatchNormalization(),  # Normalize activations
    layers.MaxPooling2D((2, 2)),  # Reduce spatial dimensions
(downsampling)
    layers.Dropout(0.2),  # Prevent overfitting by randomly dropping
connections

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),  #
First convolution layer with 128 filters
    layers.BatchNormalization(),  # Normalize activations
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),  #
Second convolution layer
    layers.BatchNormalization(),  # Normalize activations
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.3),  # Higher dropout rate to further prevent
overfitting

    # Third Convolutional Block
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),  #
First convolution layer with 256 filters
    layers.BatchNormalization(),  # Normalize activations
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),  #
Second convolution layer
    layers.BatchNormalization(),  # Normalize activations
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.4),  # Increase dropout rate for more
regularization

    # Fourth Convolutional Block
    layers.Conv2D(512, (3, 3), activation='relu', padding='same'),  #
First convolution layer with 512 filters
    layers.BatchNormalization(),  # Normalize activations
    layers.Conv2D(512, (3, 3), activation='relu', padding='same'),  #
Second convolution layer
    layers.BatchNormalization(),  # Normalize activations
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.4),  # Maintain high dropout to reduce
overfitting

    # Fully Connected Layers
```

```
    layers.Flatten(),  # Flatten the feature maps into a 1D vector for
classification
    layers.Dense(512, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),  # First dense layer with
L2 regularization
    layers.BatchNormalization(),  # Normalize activations
    layers.Dropout(0.4),  # Dropout for regularization

    layers.Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),  # Second dense layer with
L2 regularization
    layers.BatchNormalization(),  # Normalize activations
    layers.Dropout(0.3),  # Dropout for regularization

    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])
```

Task 5.1.2: Model Compilation

```
# Compile the model with a lower learning rate
model_v2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00
05),
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])

# Print the model summary
model_v2.summary()

Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
━━━━━━━━━━━━━━━━━┩
│ conv2d_6 (Conv2D)                     │ (None, 32, 32, 64)          │
1,792 │
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ batch_normalization                   │ (None, 32, 32, 64)          │
256 │
│ (BatchNormalization)                  │                             │
│
├───────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ conv2d_7 (Conv2D)                     │ (None, 32, 32, 64)          │
36,928 │
├───────────────────────────────────────┼─────────────────────────────┤
```

```
────────────────────┤
│ batch_normalization_1                 │ (None, 32, 32, 64)             │
256 │
│ (BatchNormalization)                  │                                │
│
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ max_pooling2d_3 (MaxPooling2D)        │ (None, 16, 16, 64)             │
0 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ dropout_4 (Dropout)                   │ (None, 16, 16, 64)             │
0 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ conv2d_8 (Conv2D)                     │ (None, 16, 16, 128)            │
73,856 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ batch_normalization_2                 │ (None, 16, 16, 128)            │
512 │
│ (BatchNormalization)                  │                                │
│
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ conv2d_9 (Conv2D)                     │ (None, 16, 16, 128)            │
147,584 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ batch_normalization_3                 │ (None, 16, 16, 128)            │
512 │
│ (BatchNormalization)                  │                                │
│
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ max_pooling2d_4 (MaxPooling2D)        │ (None, 8, 8, 128)              │
0 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ dropout_5 (Dropout)                   │ (None, 8, 8, 128)              │
0 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ conv2d_10 (Conv2D)                    │ (None, 8, 8, 256)              │
295,168 │
├───────────────────────────────────────┼────────────────────────────────┼
────────────────────┤
│ batch_normalization_4                 │ (None, 8, 8, 256)              │
1,024 │
```

```
│ (BatchNormalization)                            │                              │
│
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ conv2d_11 (Conv2D)                              │ (None, 8, 8, 256)            │
590,080 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ batch_normalization_5                           │ (None, 8, 8, 256)            │
1,024 │
│ (BatchNormalization)                            │                              │
│
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ max_pooling2d_5 (MaxPooling2D)                  │ (None, 4, 4, 256)            │
0 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ dropout_6 (Dropout)                             │ (None, 4, 4, 256)            │
0 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ conv2d_12 (Conv2D)                              │ (None, 4, 4, 512)            │
1,180,160 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ batch_normalization_6                           │ (None, 4, 4, 512)            │
2,048 │
│ (BatchNormalization)                            │                              │
│
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ conv2d_13 (Conv2D)                              │ (None, 4, 4, 512)            │
2,359,808 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ batch_normalization_7                           │ (None, 4, 4, 512)            │
2,048 │
│ (BatchNormalization)                            │                              │
│
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ max_pooling2d_6 (MaxPooling2D)                  │ (None, 2, 2, 512)            │
0 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
─────────────────┤
│ dropout_7 (Dropout)                             │ (None, 2, 2, 512)            │
0 │
├─────────────────────────────────────────────────┼──────────────────────────────┤
```

```
────────────────────┤
│ flatten_1 (Flatten)                          │ (None, 2048)                        │
0 │
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ dense_2 (Dense)                              │ (None, 512)                         │
1,049,088 │
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ batch_normalization_8                        │ (None, 512)                         │
2,048 │
│ (BatchNormalization)                         │                                     │
│
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ dropout_8 (Dropout)                          │ (None, 512)                         │
0 │
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ dense_3 (Dense)                              │ (None, 256)                         │
131,328 │
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ batch_normalization_9                        │ (None, 256)                         │
1,024 │
│ (BatchNormalization)                         │                                     │
│
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ dropout_9 (Dropout)                          │ (None, 256)                         │
0 │
├──────────────────────────────────────────────┼─────────────────────────────────────┼
────────────────────┤
│ dense_4 (Dense)                              │ (None, 10)                          │
2,570 │
└──────────────────────────────────────────────┴─────────────────────────────────────┴
────────────────────┘

 Total params: 5,879,114 (22.43 MB)

 Trainable params: 5,873,738 (22.41 MB)

 Non-trainable params: 5,376 (21.00 KB)
```

Task 5.1.3: Training and evaluating the model

```
history_v2 = model_v2.fit(x_train_final, y_train_final,
                          epochs=25,
                          batch_size=64,
                          validation_data=(x_test_final,
```

```
y_test_final))

# Evaluate the optimized model
test_loss_v2, test_accuracy_v2 = model_v2.evaluate(x_test_final,
y_test_final)

print(f"Test Accuracy after optimizations: {test_accuracy_v2 *
100:.2f}%")
```

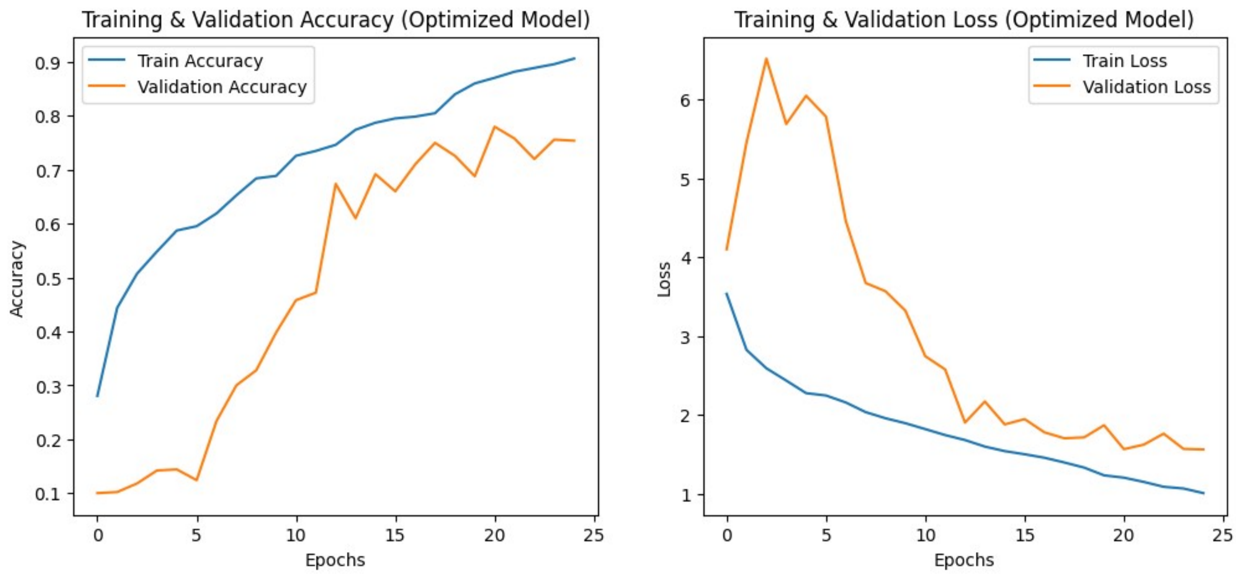
```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 34s 354ms/step - accuracy: 0.2107 - loss:
3.8616 - val_accuracy: 0.1000 - val_loss: 4.1013
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 43ms/step - accuracy: 0.4387 - loss:
2.8646 - val_accuracy: 0.1020 - val_loss: 5.4558
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.4989 - loss:
2.6538 - val_accuracy: 0.1180 - val_loss: 6.5186
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.5383 - loss:
2.4526 - val_accuracy: 0.1420 - val_loss: 5.6900
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.5751 - loss:
2.3039 - val_accuracy: 0.1440 - val_loss: 6.0478
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.5822 - loss:
2.2699 - val_accuracy: 0.1240 - val_loss: 5.7831
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.6141 - loss:
2.1741 - val_accuracy: 0.2340 - val_loss: 4.4581
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 44ms/step - accuracy: 0.6453 - loss:
2.0428 - val_accuracy: 0.3000 - val_loss: 3.6742
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.6900 - loss:
1.9326 - val_accuracy: 0.3280 - val_loss: 3.5696
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.6907 - loss:
1.8975 - val_accuracy: 0.3980 - val_loss: 3.3232
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7274 - loss:
1.8306 - val_accuracy: 0.4580 - val_loss: 2.7465
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7399 - loss:
1.7399 - val_accuracy: 0.4720 - val_loss: 2.5789
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7432 - loss:
1.7006 - val_accuracy: 0.6740 - val_loss: 1.9054
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 43ms/step - accuracy: 0.7749 - loss:
```

```
1.6031 - val_accuracy: 0.6100 - val_loss: 2.1729
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.8018 - loss:
1.5193 - val_accuracy: 0.6920 - val_loss: 1.8821
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.8120 - loss:
1.4614 - val_accuracy: 0.6600 - val_loss: 1.9495
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7950 - loss:
1.4792 - val_accuracy: 0.7100 - val_loss: 1.7814
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.8103 - loss:
1.3928 - val_accuracy: 0.7500 - val_loss: 1.7064
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.8332 - loss:
1.3509 - val_accuracy: 0.7260 - val_loss: 1.7178
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 43ms/step - accuracy: 0.8716 - loss:
1.2139 - val_accuracy: 0.6880 - val_loss: 1.8719
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.8814 - loss:
1.1906 - val_accuracy: 0.7800 - val_loss: 1.5686
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.8779 - loss:
1.1513 - val_accuracy: 0.7580 - val_loss: 1.6255
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.8931 - loss:
1.0843 - val_accuracy: 0.7200 - val_loss: 1.7650
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.9038 - loss:
1.0541 - val_accuracy: 0.7560 - val_loss: 1.5717
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.9080 - loss:
1.0032 - val_accuracy: 0.7540 - val_loss: 1.5644
16/16 ━━━━━━━━━━━━━━━━━━━━ 3s 112ms/step - accuracy: 0.7704 - loss:
1.5366
Test Accuracy after optimizations: 75.40%
```

Task 5.1.4: Plotting the training and validation accuracy

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history_v2.history['accuracy'], label='Train Accuracy')
plt.plot(history_v2.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training & Validation Accuracy (Optimized Model)')
```

```
# Plot the training & validation loss
plt.subplot(1, 2, 2)
plt.plot(history_v2.history['loss'], label='Train Loss')
plt.plot(history_v2.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training & Validation Loss (Optimized Model)')

plt.show()
```



Task 5.1. Conlusion:

The optimized CNN model achieved a test accuracy of 75.40%, marking a 4.2% improvement over the previous version. This enhancement can be attributed to several key modifications, including increased model depth, batch normalization, higher dropout rates, and L2 regularization, all of which contributed to better generalization and reduced overfitting. The addition of 512 and 256-unit dense layers further strengthened feature extraction, while a lower learning rate of 0.0005 allowed for more stable convergence.

Task 5.2: Use fewer or more convolutional/maxpooling layers and different shapes, what is the effect?

5.2.1: Using a Simple CNN which is a model with fewer layers

```
# Define a shallower CNN model
model_simple = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer defining the shape
of input images

    # First Convolutional Block (Reduced Filters)
```

```
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),  #
Convolution layer with 32 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling to reduce spatial
dimensions

    # Second Convolutional Block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
Convolution layer with 64 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(128, activation='relu'),  # Dense layer with 128
neurons
    layers.Dropout(0.3),  # Dropout for regularization
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])

# Compile the model with Adam optimizer and categorical cross-entropy
loss
model_simple.compile(optimizer='adam',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])

# Record the start time for training
start_time = time.time()

# Train the shallow model
history_simple = model_simple.fit(x_train_final, y_train_final,
                                  epochs=25,  # Train for 25 epochs
                                  batch_size=64,  # Mini-batch size of
64
                                  validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Record the end time after training
end_time = time.time()

# Evaluate the trained model on the test set
test_loss_simple, test_accuracy_simple =
model_simple.evaluate(x_test_final, y_test_final)
print(f"Test Accuracy (Shallow Model): {test_accuracy_simple *
100:.2f}%")  # Print test accuracy

# Calculate and display the total training time
training_time = end_time - start_time
print(f"Training Time: {training_time:.2f} seconds")
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 5s 59ms/step - accuracy: 0.2275 - loss:
2.0703 - val_accuracy: 0.4840 - val_loss: 1.4479
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.4417 - loss:
1.5268 - val_accuracy: 0.6080 - val_loss: 1.2066
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.5362 - loss:
1.2925 - val_accuracy: 0.6300 - val_loss: 1.1242
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.5886 - loss:
1.1349 - val_accuracy: 0.6500 - val_loss: 1.0242
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.6341 - loss:
1.0291 - val_accuracy: 0.6600 - val_loss: 0.9727
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.6505 - loss:
0.9973 - val_accuracy: 0.6640 - val_loss: 0.9539
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.6476 - loss:
0.9296 - val_accuracy: 0.6720 - val_loss: 0.9609
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.6789 - loss:
0.8764 - val_accuracy: 0.7000 - val_loss: 0.8861
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.7218 - loss:
0.7968 - val_accuracy: 0.6780 - val_loss: 0.8642
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.7258 - loss:
0.7509 - val_accuracy: 0.6800 - val_loss: 0.8629
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.7788 - loss:
0.6524 - val_accuracy: 0.6980 - val_loss: 0.8723
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7800 - loss:
0.6204 - val_accuracy: 0.6920 - val_loss: 0.8356
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7862 - loss:
0.6141 - val_accuracy: 0.6800 - val_loss: 0.8457
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8001 - loss:
0.5334 - val_accuracy: 0.6860 - val_loss: 0.8877
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.8049 - loss:
0.5344 - val_accuracy: 0.6880 - val_loss: 0.8662
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.8273 - loss:
0.4697 - val_accuracy: 0.6840 - val_loss: 0.8896
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8470 - loss:
```

```
0.4270 - val_accuracy: 0.6900 - val_loss: 0.8625
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8751 - loss:
0.3762 - val_accuracy: 0.6960 - val_loss: 0.9254
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8723 - loss:
0.3675 - val_accuracy: 0.7080 - val_loss: 0.9518
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8944 - loss:
0.3288 - val_accuracy: 0.6980 - val_loss: 0.9751
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.8976 - loss:
0.2935 - val_accuracy: 0.7020 - val_loss: 0.9093
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9087 - loss:
0.2657 - val_accuracy: 0.6800 - val_loss: 0.9594
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9193 - loss:
0.2400 - val_accuracy: 0.6840 - val_loss: 0.9565
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9288 - loss:
0.2296 - val_accuracy: 0.6900 - val_loss: 0.9849
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9296 - loss:
0.2208 - val_accuracy: 0.7080 - val_loss: 0.9675
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.7564 - loss:
0.7894
Test Accuracy (Shallow Model): 70.80%
Training Time: 27.06 seconds
```

Task 5.2.1 Conclusion:

The shallow CNN model achieved a test accuracy of 70.80%, which is lower than the 75.40% attained by the deeper, optimized model. This drop in accuracy is expected, as the simpler architecture has fewer convolutional layers i.e. only two blocks, reduced filter sizes i.e. max 64 filters, and a single dense layer i.e. 128 neurons. However, it trained significantly faster which is in 27.06 seconds, making it computationally more efficient. While this model is quicker, it lacks the depth and complexity needed to extract richer hierarchical features from the image data. The absence of batch normalization and L2 regularization may have also contributed to slightly lower generalization performance. However, for scenarios requiring faster training with limited computational resources, this shallower model still provides a reasonable baseline.

Task 5.2.2: Using a Deeper CNN which is a model with more layers

```
# Define a deeper CNN model
model_deep = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer

    # First Conv Block
```

```
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),

    # Second Conv Block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),

    # Third Conv Block
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(10, activation='softmax')  # 10 classes
])

# Compile the deeper model
model_deep.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

start_time = time.time()

# Train the deep model
history_deep = model_deep.fit(x_train_final, y_train_final,
                              epochs=25,
                              batch_size=64,
                              validation_data=(x_test_final,
y_test_final))

end_time = time.time()

# Evaluate
test_loss_deep, test_accuracy_deep = model_deep.evaluate(x_test_final,
y_test_final)
print(f"Test Accuracy (Deeper Model): {test_accuracy_deep * 100:.2f}
%")

# Calculate training time
training_time = end_time - start_time
print(f"Training Time: {training_time:.2f} seconds")
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 11s 121ms/step - accuracy: 0.1636 - loss:
2.2269 - val_accuracy: 0.2900 - val_loss: 1.8000
Epoch 2/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 35ms/step - accuracy: 0.2936 - loss:
1.7587 - val_accuracy: 0.3680 - val_loss: 1.5648
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.3950 - loss:
1.5821 - val_accuracy: 0.5220 - val_loss: 1.3482
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.4453 - loss:
1.4440 - val_accuracy: 0.5020 - val_loss: 1.3415
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.4941 - loss:
1.3373 - val_accuracy: 0.5200 - val_loss: 1.2124
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 33ms/step - accuracy: 0.5548 - loss:
1.2117 - val_accuracy: 0.5920 - val_loss: 1.1434
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 32ms/step - accuracy: 0.6037 - loss:
1.0951 - val_accuracy: 0.6380 - val_loss: 0.9871
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.6423 - loss:
0.9914 - val_accuracy: 0.6560 - val_loss: 0.9942
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.6441 - loss:
0.9410 - val_accuracy: 0.6620 - val_loss: 0.9266
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.6846 - loss:
0.8637 - val_accuracy: 0.6680 - val_loss: 0.9319
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.7092 - loss:
0.7798 - val_accuracy: 0.7200 - val_loss: 0.8622
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.7485 - loss:
0.6621 - val_accuracy: 0.6920 - val_loss: 0.8841
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.7956 - loss:
0.5730 - val_accuracy: 0.7060 - val_loss: 0.8525
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.8151 - loss:
0.5140 - val_accuracy: 0.6680 - val_loss: 0.9536
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.8352 - loss:
0.4611 - val_accuracy: 0.6940 - val_loss: 0.9166
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.8444 - loss:
0.3860 - val_accuracy: 0.6780 - val_loss: 1.1224
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.8615 - loss:
0.3571 - val_accuracy: 0.7180 - val_loss: 1.1047
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.8890 - loss:
```

```
0.3041 - val_accuracy: 0.7140 - val_loss: 1.0392
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9344 - loss:
0.1985 - val_accuracy: 0.7120 - val_loss: 1.1193
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9367 - loss:
0.1827 - val_accuracy: 0.7140 - val_loss: 1.2851
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9071 - loss:
0.2393 - val_accuracy: 0.7280 - val_loss: 1.4354
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.9362 - loss:
0.1789 - val_accuracy: 0.7060 - val_loss: 1.2300
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.9589 - loss:
0.1289 - val_accuracy: 0.6980 - val_loss: 1.2968
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9509 - loss:
0.1371 - val_accuracy: 0.7100 - val_loss: 1.6678
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9581 - loss:
0.1151 - val_accuracy: 0.7200 - val_loss: 1.5761
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step - accuracy: 0.7503 - loss:
1.3558
Test Accuracy (Deeper Model): 72.00%
Training Time: 45.41 seconds
```

Task 5.2.2: Conclusion

The deeper CNN model achieved a test accuracy of 72.00%, which is an improvement over the shallow model (70.00%) but still lower than the optimized CNN model (75.40%). This suggests that while adding more convolutional layers improves feature extraction, the model may not be fully optimized for generalization. One possible reason for this could be the absence of batch normalization and L2 regularization, which help stabilize training and reduce overfitting. Additionally, the training time of 45.41 seconds indicates that the model is computationally more expensive than the shallow version.

Task 5.3: Experiment with different activation functions in the inner layers and in the convolutional layers (relu, sigmoid, softmask, etc), see the list of keras activations at https://keras.io/api/layers/activations/

Task 5.3.1: Experimenting with Relu activation function

```
# Define a CNN model using ReLU activation
model_relu = models.Sequential([
    Input(shape=(32, 32, 3)),  # Input layer defining the shape of
input images
```

```
    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),  #
Convolution layer with 64 filters and ReLU activation
    layers.MaxPooling2D((2, 2)),  # Downsampling layer to reduce
spatial dimensions

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),  #
Convolution layer with 128 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Third Convolutional Block
    layers.Conv2D(256, (3, 3), activation='relu', padding='same'),  #
Convolution layer with 256 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation='relu'),  # Fully connected layer
with 512 neurons and ReLU activation
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_relu.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model on the training dataset
history_relu = model_relu.fit(x_train_final, y_train_final,
                              epochs=25,  # Train for 25 epochs
                              batch_size=64,  # Mini-batch size of 64
                              validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss_relu, test_accuracy_relu = model_relu.evaluate(x_test_final,
y_test_final)
print(f"Test Accuracy (ReLU - Baseline): {test_accuracy_relu *
100:.2f}%")  # Print test accuracy
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 64ms/step - accuracy: 0.2262 - loss:
2.0570 - val_accuracy: 0.3760 - val_loss: 1.6393
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.3902 - loss:
1.5676 - val_accuracy: 0.5700 - val_loss: 1.1890
Epoch 3/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5351 - loss: 
1.2591 - val_accuracy: 0.5960 - val_loss: 1.1369
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6010 - loss: 
1.0690 - val_accuracy: 0.6440 - val_loss: 1.0203
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6428 - loss: 
0.9742 - val_accuracy: 0.6400 - val_loss: 0.9650
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6745 - loss: 
0.8482 - val_accuracy: 0.6760 - val_loss: 0.8961
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7363 - loss: 
0.7386 - val_accuracy: 0.6940 - val_loss: 0.8910
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7782 - loss: 
0.6157 - val_accuracy: 0.6340 - val_loss: 1.1008
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.7977 - loss: 
0.5530 - val_accuracy: 0.6960 - val_loss: 0.9254
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8504 - loss: 
0.4197 - val_accuracy: 0.6780 - val_loss: 0.9624
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8470 - loss: 
0.4100 - val_accuracy: 0.7320 - val_loss: 0.9002
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9156 - loss: 
0.2409 - val_accuracy: 0.6920 - val_loss: 1.0761
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9376 - loss: 
0.1867 - val_accuracy: 0.6960 - val_loss: 1.0580
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9365 - loss: 
0.1793 - val_accuracy: 0.7180 - val_loss: 1.0419
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9546 - loss: 
0.1396 - val_accuracy: 0.7400 - val_loss: 1.1145
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9742 - loss: 
0.0807 - val_accuracy: 0.7480 - val_loss: 1.0790
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9623 - loss: 
0.0996 - val_accuracy: 0.7380 - val_loss: 1.2609
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9744 - loss: 
0.0826 - val_accuracy: 0.7300 - val_loss: 1.2733
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9942 - loss:
```

```
0.0319 - val_accuracy: 0.7400 - val_loss: 1.3316
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9967 - loss:
0.0224 - val_accuracy: 0.7500 - val_loss: 1.3371
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9985 - loss:
0.0138 - val_accuracy: 0.7420 - val_loss: 1.4032
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0073 - val_accuracy: 0.7420 - val_loss: 1.5190
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0045 - val_accuracy: 0.7380 - val_loss: 1.6150
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0024 - val_accuracy: 0.7480 - val_loss: 1.6247
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 1.0000 - loss:
0.0016 - val_accuracy: 0.7360 - val_loss: 1.6732
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.7663 - loss:
1.4330
Test Accuracy (ReLU - Baseline): 73.60%
```

Task 5.3.1 Conlusion:

The ReLU-based CNN model achieved a test accuracy of 73.60%, which is an improvement over the shallow model (70.80%) and the deeper model (72.00%), though still slightly lower than the optimized model (75.40%). This performance gain can be attributed to the balanced architecture, which includes three convolutional blocks with max pooling, a fully connected layer with 512 neurons, and the efficient use of ReLU activation to mitigate the vanishing gradient problem. The model strikes a balance between depth and computational efficiency, allowing it to learn meaningful hierarchical features without excessive complexity. However, the lack of batch normalization and dropout layers might limit generalization, making it more prone to overfitting compared to the optimized model.

Task 5.3.2: Experimenting with Sigmoid activation function

```
# Define a CNN model using Sigmoid activation
model_sigmoid = models.Sequential([
    Input(shape=(32, 32, 3)),  # Input layer defining the shape of
input images

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation='sigmoid', padding='same'),
# Convolution layer with 64 filters and Sigmoid activation
    layers.MaxPooling2D((2, 2)),  # Downsampling layer to reduce
spatial dimensions

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation='sigmoid', padding='same'),
```

```
# Convolution layer with 128 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Third Convolutional Block
    layers.Conv2D(256, (3, 3), activation='sigmoid', padding='same'),
# Convolution layer with 256 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation='sigmoid'),  # Fully connected layer
with 512 neurons and Sigmoid activation
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_sigmoid.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model on the training dataset
history_sigmoid = model_sigmoid.fit(x_train_final, y_train_final,
                                    epochs=25,  # Train for 25 epochs
                                    batch_size=64,  # Mini-batch size
of 64
                                    validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss_sigmoid, test_accuracy_sigmoid =
model_sigmoid.evaluate(x_test_final, y_test_final)
print(f"Test Accuracy (Sigmoid): {test_accuracy_sigmoid * 100:.2f}%")
# Print test accuracy
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.1039 - loss:
3.0110 - val_accuracy: 0.1000 - val_loss: 2.3134
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1038 - loss:
2.3137 - val_accuracy: 0.1000 - val_loss: 2.3195
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1087 - loss:
2.3235 - val_accuracy: 0.1000 - val_loss: 2.3213
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0998 - loss:
2.3224 - val_accuracy: 0.1000 - val_loss: 2.3236
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1000 - loss:
2.3178 - val_accuracy: 0.1000 - val_loss: 2.3137
```

```
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0966 - loss:
2.3220 - val_accuracy: 0.1000 - val_loss: 2.3197
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1076 - loss:
2.3283 - val_accuracy: 0.1000 - val_loss: 2.3094
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.0883 - loss:
2.3195 - val_accuracy: 0.1000 - val_loss: 2.3147
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0905 - loss:
2.3241 - val_accuracy: 0.1000 - val_loss: 2.3103
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1088 - loss:
2.3306 - val_accuracy: 0.1000 - val_loss: 2.3295
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1027 - loss:
2.3263 - val_accuracy: 0.1000 - val_loss: 2.3140
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.1013 - loss:
2.3187 - val_accuracy: 0.1000 - val_loss: 2.3209
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0989 - loss:
2.3222 - val_accuracy: 0.1000 - val_loss: 2.3224
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.0892 - loss:
2.3238 - val_accuracy: 0.1000 - val_loss: 2.3183
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0899 - loss:
2.3196 - val_accuracy: 0.1000 - val_loss: 2.3141
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0794 - loss:
2.3188 - val_accuracy: 0.1000 - val_loss: 2.3184
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.1007 - loss:
2.3267 - val_accuracy: 0.1000 - val_loss: 2.3176
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1002 - loss:
2.3219 - val_accuracy: 0.1000 - val_loss: 2.3178
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0872 - loss:
2.3237 - val_accuracy: 0.1000 - val_loss: 2.3251
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.0950 - loss:
2.3242 - val_accuracy: 0.1000 - val_loss: 2.3219
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0995 - loss:
2.3197 - val_accuracy: 0.1000 - val_loss: 2.3345
Epoch 22/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1038 - loss:
2.3235 - val_accuracy: 0.1000 - val_loss: 2.3206
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0912 - loss:
2.3335 - val_accuracy: 0.1000 - val_loss: 2.3175
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1093 - loss:
2.3157 - val_accuracy: 0.1340 - val_loss: 2.3139
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1055 - loss:
2.3240 - val_accuracy: 0.1120 - val_loss: 2.3061
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.2906 - loss:
2.1977
Test Accuracy (Sigmoid): 11.20%
```

Task 5.3.2 Conclusion:

The CNN model using Sigmoid activation performed significantly worse than the other models, achieving a test accuracy of only 11.20%, compared to 73.60% with ReLU, 70.80% with the shallow model, and 72.00% with the deeper model. This drastic drop in accuracy is expected due to the inherent limitations of the Sigmoid activation function in deep networks. Sigmoid suffers from the vanishing gradient problem, where activations get squashed into a narrow range (0,1), leading to extremely small gradients in deeper layers. This slows down learning and prevents effective weight updates. Additionally, Sigmoid activations can saturate, meaning neurons become stuck with near-zero gradients, making it difficult for the model to learn meaningful features. Another drawback is that Sigmoid is not zero-centered, which can cause inefficient weight updates and slower convergence. As a result, the model struggles to extract relevant patterns from the data and ends up making near-random predictions, explaining the low accuracy. A clear takeaway is that ReLU (or its variants like LeakyReLU and ELU) is far better suited for deep CNNs, as it mitigates these issues and allows for more effective learning.

Task 5.3.3: Experimenting with Tanh activation function

```
# Define a CNN model using Tanh activation
model_tanh = models.Sequential([
    Input(shape=(32, 32, 3)),  # Input layer defining the shape of
input images

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation='tanh', padding='same'),  #
Convolution layer with 64 filters and Tanh activation
    layers.MaxPooling2D((2, 2)),  # Downsampling layer to reduce
spatial dimensions

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation='tanh', padding='same'),  #
Convolution layer with 128 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Third Convolutional Block
```

```
    layers.Conv2D(256, (3, 3), activation='tanh', padding='same'),  # 
Convolution layer with 256 filters
    layers.MaxPooling2D((2, 2)),  # Downsampling

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation='tanh'),  # Fully connected layer 
with 512 neurons and Tanh activation
    layers.Dense(10, activation='softmax')  # Output layer with 10 
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy 
loss
model_tanh.compile(optimizer='adam', loss='categorical_crossentropy', 
metrics=['accuracy'])

# Train the model on the training dataset
history_tanh = model_tanh.fit(x_train_final, y_train_final,
                              epochs=25,  # Train for 25 epochs
                              batch_size=64,  # Mini-batch size of 64
                              validation_data=(x_test_final, 
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss_tanh, test_accuracy_tanh = model_tanh.evaluate(x_test_final, 
y_test_final)
print(f"Test Accuracy (Tanh): {test_accuracy_tanh * 100:.2f}%")  # 
Print test accuracy

Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 82ms/step - accuracy: 0.2763 - loss: 
2.2856 - val_accuracy: 0.5380 - val_loss: 1.2695
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.5691 - loss: 
1.2204 - val_accuracy: 0.5960 - val_loss: 1.1216
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6075 - loss: 
1.0832 - val_accuracy: 0.6200 - val_loss: 1.0823
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.6619 - loss: 
0.9462 - val_accuracy: 0.6620 - val_loss: 0.9629
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7159 - loss: 
0.8278 - val_accuracy: 0.6720 - val_loss: 0.8900
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7538 - loss: 
0.7023 - val_accuracy: 0.6920 - val_loss: 0.8682
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8289 - loss: 
```

```
0.5191 - val_accuracy: 0.7100 - val_loss: 0.8859
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8597 - loss:
0.4379 - val_accuracy: 0.7080 - val_loss: 0.8435
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9159 - loss:
0.2753 - val_accuracy: 0.7240 - val_loss: 0.9029
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9397 - loss:
0.1985 - val_accuracy: 0.7100 - val_loss: 0.9133
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9635 - loss:
0.1316 - val_accuracy: 0.6900 - val_loss: 0.9913
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9832 - loss:
0.0773 - val_accuracy: 0.7060 - val_loss: 1.0384
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9958 - loss:
0.0380 - val_accuracy: 0.7180 - val_loss: 1.0310
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9956 - loss:
0.0280 - val_accuracy: 0.7160 - val_loss: 1.0405
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9995 - loss:
0.0140 - val_accuracy: 0.7340 - val_loss: 1.0332
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9981 - loss:
0.0132 - val_accuracy: 0.7320 - val_loss: 1.1037
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0064 - val_accuracy: 0.7380 - val_loss: 1.0857
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0043 - val_accuracy: 0.7300 - val_loss: 1.1411
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0030 - val_accuracy: 0.7280 - val_loss: 1.1360
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0024 - val_accuracy: 0.7340 - val_loss: 1.1633
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0020 - val_accuracy: 0.7300 - val_loss: 1.1614
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0018 - val_accuracy: 0.7300 - val_loss: 1.1726
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0016 - val_accuracy: 0.7300 - val_loss: 1.1896
```

```
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 1.0000 - loss:
0.0015 - val_accuracy: 0.7280 - val_loss: 1.1893
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 1.0000 - loss:
0.0013 - val_accuracy: 0.7300 - val_loss: 1.2061
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step - accuracy: 0.7748 - loss:
0.9991
Test Accuracy (Tanh): 73.00%
```

Task 5.3.3 Conlusion:

The CNN model using Tanh activation achieved a test accuracy of 73.00%, which is slightly lower than the 73.60% obtained with ReLU but still significantly better than the 11.20% from Sigmoid. Unlike Sigmoid, Tanh is zero-centered, meaning it allows both positive and negative activations, leading to more balanced weight updates during training. However, Tanh still suffers from the vanishing gradient problem, especially in deeper networks, where gradients can shrink as they propagate backward, slowing down learning. Compared to ReLU, which avoids this issue by allowing unrestricted positive outputs, Tanh limits activations within the range (-1,1), potentially reducing the model's ability to learn complex hierarchical features. While Tanh can be useful in shallow networks or when working with centered data, ReLU remains the preferred choice for deep CNNs due to its efficiency and ability to avoid saturation.

Task 5.3.4: Experimenting with LeakyReLU activation function

```
# Define a CNN model using LeakyReLU activation
model_leakyrelu = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer defining the shape
of input images

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation=None, padding='same'),  #
Convolution layer with 64 filters (no activation yet)
    LeakyReLU(negative_slope=0.01),  # Apply LeakyReLU activation
    layers.Conv2D(64, (3, 3), activation=None, padding='same'),  #
Another convolution layer
    LeakyReLU(negative_slope=0.01),  # Apply LeakyReLU activation
    layers.MaxPooling2D((2, 2)),  # Downsampling to reduce spatial
dimensions
    layers.Dropout(0.2),  # Dropout layer to prevent overfitting

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation=None, padding='same'),  #
Convolution with 128 filters
    LeakyReLU(negative_slope=0.01),  # Apply LeakyReLU activation
    layers.Conv2D(128, (3, 3), activation=None, padding='same'),  #
Another convolution layer
    LeakyReLU(negative_slope=0.01),  # Apply LeakyReLU activation
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.3),  # Dropout layer
```

```
    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation=None),  # Fully connected layer with
512 neurons (no activation yet)
    LeakyReLU(negative_slope=0.01),  # Apply LeakyReLU activation
    layers.Dropout(0.5),  # Dropout layer

    # Output layer
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_leakyrelu.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model on the training dataset
history_leakyrelu = model_leakyrelu.fit(x_train_final, y_train_final,
                                        epochs=25,  # Train for 25
epochs
                                        batch_size=64,  # Mini-batch
size of 64
                                        validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss, test_acc = model_leakyrelu.evaluate(x_test_final,
y_test_final, verbose=0)
print(f'Test Accuracy (LeakyReLU): {test_acc * 100:.2f}%')  # Print
test accuracy
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 11s 120ms/step - accuracy: 0.1740 - loss:
2.1500 - val_accuracy: 0.4540 - val_loss: 1.4940
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.4147 - loss:
1.5389 - val_accuracy: 0.5480 - val_loss: 1.2262
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5192 - loss:
1.3183 - val_accuracy: 0.5600 - val_loss: 1.1708
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5510 - loss:
1.1865 - val_accuracy: 0.5680 - val_loss: 1.1488
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.5673 - loss:
1.1466 - val_accuracy: 0.6320 - val_loss: 1.0240
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.6536 - loss:
```

```
0.9442 - val_accuracy: 0.6480 - val_loss: 0.9103
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7169 - loss:
0.8014 - val_accuracy: 0.6840 - val_loss: 0.8484
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7363 - loss:
0.7249 - val_accuracy: 0.6920 - val_loss: 0.8294
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7878 - loss:
0.6096 - val_accuracy: 0.6880 - val_loss: 0.9011
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7966 - loss:
0.5467 - val_accuracy: 0.6880 - val_loss: 0.9186
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8397 - loss:
0.4520 - val_accuracy: 0.7160 - val_loss: 0.9383
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8569 - loss:
0.3920 - val_accuracy: 0.7240 - val_loss: 0.9249
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8799 - loss:
0.3411 - val_accuracy: 0.6920 - val_loss: 1.1058
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9038 - loss:
0.2804 - val_accuracy: 0.7100 - val_loss: 1.0106
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9151 - loss:
0.2434 - val_accuracy: 0.7200 - val_loss: 1.1164
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9357 - loss:
0.2051 - val_accuracy: 0.6940 - val_loss: 1.1888
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9343 - loss:
0.1786 - val_accuracy: 0.7020 - val_loss: 1.1712
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9136 - loss:
0.2335 - val_accuracy: 0.6860 - val_loss: 1.4291
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9497 - loss:
0.1514 - val_accuracy: 0.7420 - val_loss: 1.2835
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9594 - loss:
0.1026 - val_accuracy: 0.6980 - val_loss: 1.4550
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9545 - loss:
0.1139 - val_accuracy: 0.7020 - val_loss: 1.3585
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9537 - loss:
0.1372 - val_accuracy: 0.7220 - val_loss: 1.3607
```

```
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9631 - loss:
0.0988 - val_accuracy: 0.7180 - val_loss: 1.3481
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9671 - loss:
0.0925 - val_accuracy: 0.6820 - val_loss: 1.6449
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9678 - loss:
0.0957 - val_accuracy: 0.7020 - val_loss: 1.6025
Test Accuracy (LeakyReLU): 70.20%
```

Task 5.3.4 Conclusion:

The CNN model using LeakyReLU activation achieved a test accuracy of 70.20%, which is lower than the 73.60% obtained with ReLU and 73.00% with Tanh but significantly better than the 11.20% from Sigmoid. LeakyReLU addresses the dying ReLU problem by allowing small negative values with a slope of 0.01, preventing neurons from becoming inactive. However, in this case, it did not outperform standard ReLU, likely due to factors such as suboptimal weight initialization, dropout rates, or model depth. The added dropout layers help prevent overfitting but may have also led to a slight reduction in accuracy. While LeakyReLU is often beneficial in deep networks with sparse activations, ReLU remains the stronger choice for this dataset due to its simplicity and effectiveness in learning complex features.

Task 5.3.5: Experimenting with ELU (Exponential Linear Unit) activation function

```
# Define a CNN model using ELU (Exponential Linear Unit) activation
model_elu = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer specifying the
shape of input images

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation='elu', padding='same'),  #
Convolution layer with ELU activation
    layers.Conv2D(64, (3, 3), activation='elu', padding='same'),  #
Another convolution layer with ELU
    layers.MaxPooling2D((2, 2)),  # Downsampling to reduce spatial
dimensions
    layers.Dropout(0.2),  # Dropout to reduce overfitting

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation='elu', padding='same'),  #
Convolution with 128 filters using ELU
    layers.Conv2D(128, (3, 3), activation='elu', padding='same'),  #
Another convolution layer with ELU
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.3),  # Dropout layer

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation='elu'),  # Fully connected layer with
```

```
512 neurons using ELU activation
    layers.Dropout(0.5),  # Dropout layer

    # Output layer
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_elu.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model on the training dataset
history_elu = model_elu.fit(x_train_final, y_train_final,
                             epochs=25,  # Train for 25 epochs
                             batch_size=64,  # Mini-batch size of 64
                             validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss, test_acc = model_elu.evaluate(x_test_final, y_test_final,
verbose=0)
print(f'Test Accuracy (ELU): {test_acc * 100:.2f}%')  # Print test
accuracy
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 10s 112ms/step - accuracy: 0.2768 - loss:
2.5933 - val_accuracy: 0.5260 - val_loss: 1.2811
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.5244 - loss:
1.3008 - val_accuracy: 0.5860 - val_loss: 1.1680
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5916 - loss:
1.1452 - val_accuracy: 0.6060 - val_loss: 1.0587
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.6728 - loss:
0.9772 - val_accuracy: 0.6000 - val_loss: 1.1888
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6702 - loss:
0.9874 - val_accuracy: 0.6220 - val_loss: 0.9937
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7411 - loss:
0.7336 - val_accuracy: 0.6320 - val_loss: 1.0215
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7609 - loss:
0.6743 - val_accuracy: 0.6160 - val_loss: 1.0768
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8080 - loss:
0.5439 - val_accuracy: 0.6200 - val_loss: 1.1202
```

```
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8259 - loss:
0.4717 - val_accuracy: 0.6400 - val_loss: 1.1914
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8622 - loss:
0.4069 - val_accuracy: 0.6700 - val_loss: 1.1624
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.8829 - loss:
0.3358 - val_accuracy: 0.6320 - val_loss: 1.2723
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9077 - loss:
0.2668 - val_accuracy: 0.6420 - val_loss: 1.2823
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9211 - loss:
0.2175 - val_accuracy: 0.6200 - val_loss: 1.3900
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9310 - loss:
0.1919 - val_accuracy: 0.6420 - val_loss: 1.4242
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9449 - loss:
0.1513 - val_accuracy: 0.6200 - val_loss: 1.6411
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9445 - loss:
0.1531 - val_accuracy: 0.6400 - val_loss: 1.7198
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9472 - loss:
0.1599 - val_accuracy: 0.6440 - val_loss: 1.7364
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9499 - loss:
0.1460 - val_accuracy: 0.6520 - val_loss: 1.8542
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9575 - loss:
0.1192 - val_accuracy: 0.6300 - val_loss: 2.0543
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9563 - loss:
0.1462 - val_accuracy: 0.6380 - val_loss: 1.8002
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.9398 - loss:
0.1750 - val_accuracy: 0.6440 - val_loss: 1.9181
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9366 - loss:
0.1965 - val_accuracy: 0.6160 - val_loss: 2.3662
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9396 - loss:
0.1762 - val_accuracy: 0.6120 - val_loss: 2.1189
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9319 - loss:
0.2395 - val_accuracy: 0.6240 - val_loss: 2.3571
Epoch 25/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9443 - loss: 0.1855 - val_accuracy: 0.6420 - val_loss: 2.1493
Test Accuracy (ELU): 64.20%
```

Task 5.3.5: Conclusion

The CNN model using ELU activation achieved a test accuracy of 64.20%, which is lower than ReLU (73.60%), Tanh (73.00%), and LeakyReLU (70.20%), but still significantly better than Sigmoid (11.20%). ELU is designed to overcome the dying ReLU problem by allowing small negative values for negative inputs, which helps improve gradient flow and speed up learning. However, in this case, its performance was weaker than ReLU-based activations. This could be due to factors such as weight initialization, dropout rates, or dataset-specific characteristics that make ReLU and Tanh more effective. The drop in accuracy compared to ReLU suggests that ELU might not be the best choice for this particular dataset. One potential reason could be ELU's computational complexity, which makes it slightly slower than ReLU, potentially affecting optimization dynamics. Additionally, the higher dropout rates (0.3 and 0.5) might be causing excessive regularization, leading to underfitting. However, based on the results so far, ReLU remains the best-performing activation function.

Task 5.3.6: Experimenting with GELU (Gaussian Error Linear Unit) activation function

```
# Define a CNN model using GELU (Gaussian Error Linear Unit) activation
model_gelu = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer specifying the shape of input images

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation=gelu, padding='same'),  # Convolution layer with GELU activation
    layers.Conv2D(64, (3, 3), activation=gelu, padding='same'),  # Another convolution layer with GELU
    layers.MaxPooling2D((2, 2)),  # Downsampling to reduce spatial dimensions
    layers.Dropout(0.2),  # Dropout to prevent overfitting

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation=gelu, padding='same'),  # Convolution with 128 filters using GELU
    layers.Conv2D(128, (3, 3), activation=gelu, padding='same'),  # Another convolution layer with GELU
    layers.MaxPooling2D((2, 2)),  # Downsampling
    layers.Dropout(0.3),  # Dropout layer

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a 1D vector
    layers.Dense(512, activation=gelu),  # Fully connected layer with 512 neurons using GELU activation
    layers.Dropout(0.5),  # Dropout layer
```

```
    # Output layer
    layers.Dense(10, activation='softmax')  # Output layer with 10 
classes using softmax activation
])

# Compile the model using Adam optimizer and categorical cross-entropy 
loss
model_gelu.compile(optimizer='adam', loss='categorical_crossentropy', 
metrics=['accuracy'])

# Train the model on the training dataset
history_gelu = model_gelu.fit(x_train_final, y_train_final,
                               epochs=25,  # Train for 25 epochs
                               batch_size=64,  # Mini-batch size of 64
                               validation_data=(x_test_final, 
y_test_final))  # Validate on test set

# Evaluate the trained model on the test set
test_loss, test_acc = model_gelu.evaluate(x_test_final, y_test_final, 
verbose=0)
print(f'Test Accuracy (GELU): {test_acc * 100:.2f}%')  # Print test 
accuracy
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 12s 140ms/step - accuracy: 0.2375 - loss: 
2.0148 - val_accuracy: 0.5440 - val_loss: 1.4268
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5043 - loss: 
1.4054 - val_accuracy: 0.5660 - val_loss: 1.1837
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.5832 - loss: 
1.1775 - val_accuracy: 0.5580 - val_loss: 1.1840
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6271 - loss: 
1.0466 - val_accuracy: 0.6560 - val_loss: 0.9750
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.6735 - loss: 
0.9115 - val_accuracy: 0.6520 - val_loss: 0.9732
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7241 - loss: 
0.7709 - val_accuracy: 0.6560 - val_loss: 0.9918
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7486 - loss: 
0.7119 - val_accuracy: 0.6760 - val_loss: 0.9545
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.8135 - loss: 
0.5303 - val_accuracy: 0.6920 - val_loss: 0.9086
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8479 - loss: 
0.4418 - val_accuracy: 0.6920 - val_loss: 0.9869
```

```
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.8778 - loss:
0.3459 - val_accuracy: 0.6860 - val_loss: 1.1532
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9114 - loss:
0.2630 - val_accuracy: 0.6840 - val_loss: 1.1206
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9177 - loss:
0.2257 - val_accuracy: 0.6800 - val_loss: 1.3461
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9299 - loss:
0.2019 - val_accuracy: 0.7020 - val_loss: 1.0684
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9635 - loss:
0.1266 - val_accuracy: 0.7120 - val_loss: 1.3271
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9642 - loss:
0.1132 - val_accuracy: 0.6720 - val_loss: 1.2247
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9601 - loss:
0.1197 - val_accuracy: 0.7040 - val_loss: 1.2733
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.9673 - loss:
0.0903 - val_accuracy: 0.6820 - val_loss: 1.3831
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9724 - loss:
0.0873 - val_accuracy: 0.6980 - val_loss: 1.4074
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9685 - loss:
0.0915 - val_accuracy: 0.7060 - val_loss: 1.4401
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9637 - loss:
0.1019 - val_accuracy: 0.7160 - val_loss: 1.5395
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9754 - loss:
0.0685 - val_accuracy: 0.7060 - val_loss: 1.5214
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9834 - loss:
0.0558 - val_accuracy: 0.7060 - val_loss: 1.6099
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9788 - loss:
0.0626 - val_accuracy: 0.7000 - val_loss: 1.7000
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9719 - loss:
0.0946 - val_accuracy: 0.7020 - val_loss: 1.6619
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.9746 - loss:
0.0625 - val_accuracy: 0.7160 - val_loss: 1.5399
Test Accuracy (GELU): 71.60%
```

Task 5.3.6: Conclusion

The CNN model using GELU activation achieved a test accuracy of 71.60%, performing better than LeakyReLU (70.20%) and ELU (64.20%), but falling short of ReLU (73.60%) and Tanh (73.00%). GELU is a smooth, probabilistic activation function that is often used in transformer-based models but has shown potential in CNNs as well. However, in this case, ReLU and Tanh outperformed GELU, possibly due to the computational overhead of GELU and the characteristics of the dataset. The higher dropout rates (0.2, 0.3, 0.5), which were also used in the ELU model, might have limited GELU's performance. While GELU provides a smooth activation, ReLU remains the best-performing activation function in this comparison, making it the preferred choice.

Task 5.3.7: Experimenting with Swish activation function

```
# Define a CNN model using Swish activation function
model_swish = models.Sequential([
    Input(shape=(32, 32, 3)),  # Input layer specifying image
dimensions

    # First Convolutional Block
    layers.Conv2D(64, (3, 3), activation=swish, padding='same'),  # 64
filters with Swish activation
    layers.MaxPooling2D((2, 2)),  # Max pooling to downsample

    # Second Convolutional Block
    layers.Conv2D(128, (3, 3), activation=swish, padding='same'),  #
128 filters with Swish
    layers.MaxPooling2D((2, 2)),  # Max pooling

    # Third Convolutional Block
    layers.Conv2D(256, (3, 3), activation=swish, padding='same'),  #
256 filters with Swish
    layers.MaxPooling2D((2, 2)),  # Max pooling

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps
    layers.Dense(512, activation=swish),  # Dense layer with 512
neurons using Swish
    layers.Dense(10, activation='softmax')  # Output layer for 10
classes using softmax
])

# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_swish.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model on the dataset
history_swish = model_swish.fit(x_train_final, y_train_final,
                                epochs=25,  # Train for 25 epochs
                                batch_size=64,  # Mini-batch size
```

```
                                validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the model on the test set
test_loss_swish, test_accuracy_swish =
model_swish.evaluate(x_test_final, y_test_final)

# Print the test accuracy
print(f"Test Accuracy (Swish): {test_accuracy_swish * 100:.2f}%")
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 72ms/step - accuracy: 0.2673 - loss:
1.9353 - val_accuracy: 0.5480 - val_loss: 1.3273
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.5360 - loss:
1.2931 - val_accuracy: 0.6200 - val_loss: 1.1116
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5951 - loss:
1.1413 - val_accuracy: 0.5900 - val_loss: 1.0989
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.6380 - loss:
1.0339 - val_accuracy: 0.6660 - val_loss: 0.9889
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6813 - loss:
0.9274 - val_accuracy: 0.6420 - val_loss: 0.9779
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6840 - loss:
0.8448 - val_accuracy: 0.6920 - val_loss: 0.9292
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7608 - loss:
0.7041 - val_accuracy: 0.7260 - val_loss: 0.8708
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7838 - loss:
0.6069 - val_accuracy: 0.6760 - val_loss: 0.9972
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8195 - loss:
0.5347 - val_accuracy: 0.6940 - val_loss: 0.9660
Epoch 10/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8485 - loss:
0.4157 - val_accuracy: 0.6300 - val_loss: 1.1557
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8387 - loss:
0.4401 - val_accuracy: 0.6960 - val_loss: 1.0802
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9189 - loss:
0.2520 - val_accuracy: 0.7280 - val_loss: 1.0967
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9250 - loss:
0.1955 - val_accuracy: 0.7300 - val_loss: 1.1739
Epoch 14/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9621 - loss:
0.1311 - val_accuracy: 0.7260 - val_loss: 1.3374
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9749 - loss:
0.0896 - val_accuracy: 0.7020 - val_loss: 1.5395
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9817 - loss:
0.0674 - val_accuracy: 0.7200 - val_loss: 1.6073
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9673 - loss:
0.1006 - val_accuracy: 0.6860 - val_loss: 1.6290
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9651 - loss:
0.0970 - val_accuracy: 0.7200 - val_loss: 1.5189
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9858 - loss:
0.0509 - val_accuracy: 0.7180 - val_loss: 1.5674
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9947 - loss:
0.0241 - val_accuracy: 0.7120 - val_loss: 1.6615
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9938 - loss:
0.0271 - val_accuracy: 0.7180 - val_loss: 1.7190
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0045 - val_accuracy: 0.7220 - val_loss: 1.7846
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0027 - val_accuracy: 0.7260 - val_loss: 1.8315
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 1.0000 - loss:
0.0014 - val_accuracy: 0.7220 - val_loss: 1.8624
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 1.0000 - loss:
0.0012 - val_accuracy: 0.7220 - val_loss: 1.8996
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.7647 - loss:
1.5458
Test Accuracy (Swish): 72.20%
```

Task 5.3.7 Conclusion:

The results indicate that the CNN model using the Swish activation function achieved a test accuracy of 72.20%, performing slightly lower than the ReLU-based model (73.60%) but outperforming the hybrid Swish-ReLU model (68.40%). The Swish activation function, known for its smooth and non-monotonic properties, contributed to efficient feature extraction, leading to competitive classification performance. However, the absence of dropout layers in the Swish model might have influenced its ability to generalize effectively. The findings suggest that while Swish can enhance model learning, further optimization, such as incorporating dropout and batch normalization, may be required to maximize its effectiveness in CNN architectures.

Task 5.3.8: Experimenting with Hybrid Model (Swish + ReLU)

```
# Define the CNN model with Swish activation in convolutional layers
and ReLU in dense layers
model_hybrid = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Explicitly defining the input
shape for a 32x32 RGB image

    # First Convolutional Block (Swish Activation)
    layers.Conv2D(64, (3, 3), activation=activations.swish,
padding='same'),  # 64 filters, Swish activation
    layers.Conv2D(64, (3, 3), activation=activations.swish,
padding='same'),  # Another convolutional layer
    layers.MaxPooling2D((2, 2)),  # Max pooling to downsample
    layers.Dropout(0.2),  # Dropout for regularization

    # Second Convolutional Block (Swish Activation)
    layers.Conv2D(128, (3, 3), activation=activations.swish,
padding='same'),  # 128 filters, Swish activation
    layers.Conv2D(128, (3, 3), activation=activations.swish,
padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    # Third Convolutional Block (Swish Activation)
    layers.Conv2D(256, (3, 3), activation=activations.swish,
padding='same'),  # 256 filters, Swish activation
    layers.Conv2D(256, (3, 3), activation=activations.swish,
padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.4),

    # Fourth Convolutional Block (Swish Activation)
    layers.Conv2D(512, (3, 3), activation=activations.swish,
padding='same'),  # 512 filters, Swish activation
    layers.Conv2D(512, (3, 3), activation=activations.swish,
padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.4),

    # Fully Connected Layers with ReLU Activation
    layers.Flatten(),  # Flattening the feature maps into a single
vector
    layers.Dense(1024, activation='relu'),  # Fully connected layer
with 1024 neurons, ReLU activation
    layers.Dropout(0.5),  # Dropout for further regularization
    layers.Dense(512, activation='relu'),  # Fully connected layer
with 512 neurons, ReLU activation
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')  # Output layer with 10
neurons for classification (softmax for multi-class)
```

```
])

# Compile the model with Adam optimizer and categorical cross-entropy
loss
model_hybrid.compile(optimizer='adam',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])

# Train the model using training data
history_hybrid = model_hybrid.fit(x_train_final, y_train_final,
                                  epochs=25,  # Number of training
epochs
                                  batch_size=64,  # Mini-batch size
                                  validation_data=(x_test_final,
y_test_final))  # Validate on test set

# Evaluate the model on the test dataset
test_loss_hybrid, test_accuracy_hybrid =
model_hybrid.evaluate(x_test_final, y_test_final)
print(f"Test Accuracy (Swish + ReLU): {test_accuracy_hybrid * 100:.2f}
%")
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 17s 182ms/step - accuracy: 0.1194 - loss:
2.3052 - val_accuracy: 0.2100 - val_loss: 1.9575
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.2116 - loss:
2.0002 - val_accuracy: 0.2580 - val_loss: 1.9323
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.2728 - loss:
1.8770 - val_accuracy: 0.3480 - val_loss: 1.6698
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.3156 - loss:
1.7637 - val_accuracy: 0.3800 - val_loss: 1.5825
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.3381 - loss:
1.7052 - val_accuracy: 0.4060 - val_loss: 1.5295
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.3610 - loss:
1.6952 - val_accuracy: 0.4240 - val_loss: 1.4882
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.4532 - loss:
1.4670 - val_accuracy: 0.4820 - val_loss: 1.3580
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.4602 - loss:
1.4279 - val_accuracy: 0.5180 - val_loss: 1.2901
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.4935 - loss:
1.3397 - val_accuracy: 0.4820 - val_loss: 1.3922
Epoch 10/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.5196 - loss:
1.2746 - val_accuracy: 0.5280 - val_loss: 1.2660
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.5497 - loss:
1.1791 - val_accuracy: 0.6340 - val_loss: 1.0284
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.5869 - loss:
1.0843 - val_accuracy: 0.6360 - val_loss: 0.9981
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.6499 - loss:
0.9839 - val_accuracy: 0.6200 - val_loss: 1.0433
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.6337 - loss:
0.9997 - val_accuracy: 0.6180 - val_loss: 0.9674
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.6745 - loss:
0.9293 - val_accuracy: 0.6580 - val_loss: 0.9679
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.6779 - loss:
0.8780 - val_accuracy: 0.6200 - val_loss: 1.1031
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.6949 - loss:
0.8939 - val_accuracy: 0.6380 - val_loss: 1.0959
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.6876 - loss:
0.8650 - val_accuracy: 0.6240 - val_loss: 1.0904
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7362 - loss:
0.7551 - val_accuracy: 0.6640 - val_loss: 0.8413
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.7667 - loss:
0.6607 - val_accuracy: 0.6300 - val_loss: 1.1597
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.7236 - loss:
0.7626 - val_accuracy: 0.6520 - val_loss: 1.0716
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.7802 - loss:
0.6055 - val_accuracy: 0.6600 - val_loss: 0.9602
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.7885 - loss:
0.6252 - val_accuracy: 0.6540 - val_loss: 1.1670
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - accuracy: 0.7775 - loss:
0.6467 - val_accuracy: 0.7080 - val_loss: 0.9068
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.8040 - loss:
0.5366 - val_accuracy: 0.6840 - val_loss: 0.9543
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 69ms/step - accuracy: 0.7435 - loss:
```

```
0.7982
Test Accuracy (Swish + ReLU): 68.40%
```

Task 5.3.8 Conclusion:

The hybrid model, which used Swish activation in convolutional layers and ReLU in dense layers, achieved a test accuracy of 68.40%. Interestingly, this is lower than both the pure Swish model (72.20%) and ReLU model (73.60%), suggesting that the combination of Swish in convolutional layers and ReLU in fully connected layers might not be optimal for this particular dataset. One possible reason for the performance drop is that Swish is designed to work well in deep networks, but using it in convolutional layers while switching to ReLU in dense layers could have disrupted the smooth gradient flow. Additionally, the increase in convolutional layers (512 filters in the fourth block) and high dropout rates (up to 0.5 in dense layers) might have led to an over-regularized model, preventing it from learning effectively.

Task 5.3.9: Experimenting with Swish + LeakyReLU activation function

```
# Define the CNN model with Swish activation in convolutional layers
and LeakyReLU in dense layers
model_swish_leakyrelu = models.Sequential([
    Input(shape=(32, 32, 3)),  # Explicitly define input shape for a
32x32 RGB image

    # First Convolutional Block (Swish Activation)
    layers.Conv2D(64, (3, 3), activation=activations.swish,
padding='same'),  # 64 filters, Swish activation
    layers.MaxPooling2D((2, 2)),  # Max pooling to reduce spatial
dimensions

    # Second Convolutional Block (Swish Activation)
    layers.Conv2D(128, (3, 3), activation=activations.swish,
padding='same'),  # 128 filters, Swish activation
    layers.MaxPooling2D((2, 2)),

    # Third Convolutional Block (Swish Activation)
    layers.Conv2D(256, (3, 3), activation=activations.swish,
padding='same'),  # 256 filters, Swish activation
    layers.MaxPooling2D((2, 2)),

    # Fully Connected Layers with LeakyReLU Activation
    layers.Flatten(),  # Flattening the feature maps into a single
vector
    layers.Dense(512, activation=None),  # Fully connected layer with
512 neurons (No activation specified yet)
    layers.LeakyReLU(negative_slope=0.01),  # Applying LeakyReLU
activation to prevent dying neurons issue
    layers.Dense(10, activation='softmax')  # Output layer with 10
neurons for classification (softmax for multi-class)
])
```

```
# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_swish_leakyrelu.compile(optimizer='adam',
                              loss='categorical_crossentropy',
                              metrics=['accuracy'])

# Train the model using training data
history_swish_leakyrelu = model_swish_leakyrelu.fit(x_train_final,
y_train_final,
                                                    epochs=25,  #
Number of training epochs
                                                    batch_size=64,  #
Mini-batch size

validation_data=(x_test_final, y_test_final))  # Validate on test set

# Evaluate the model on the test dataset
test_loss_swish_leakyrelu, test_accuracy_swish_leakyrelu =
model_swish_leakyrelu.evaluate(x_test_final, y_test_final)
print(f"Test Accuracy (Swish + LeakyReLU):
{test_accuracy_swish_leakyrelu * 100:.2f}%")
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 68ms/step - accuracy: 0.2589 - loss:
1.9711 - val_accuracy: 0.4900 - val_loss: 1.3977
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.5175 - loss:
1.3112 - val_accuracy: 0.5700 - val_loss: 1.1572
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5809 - loss:
1.1412 - val_accuracy: 0.6100 - val_loss: 1.1224
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.6363 - loss:
0.9888 - val_accuracy: 0.6520 - val_loss: 0.9898
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6699 - loss:
0.9094 - val_accuracy: 0.6640 - val_loss: 0.9933
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.7225 - loss:
0.7915 - val_accuracy: 0.6820 - val_loss: 0.9145
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7564 - loss:
0.6945 - val_accuracy: 0.6640 - val_loss: 0.9614
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7958 - loss:
0.5633 - val_accuracy: 0.6800 - val_loss: 0.9355
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8442 - loss:
0.4505 - val_accuracy: 0.6620 - val_loss: 1.1183
Epoch 10/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8770 - loss:
0.3485 - val_accuracy: 0.6960 - val_loss: 1.0856
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9188 - loss:
0.2553 - val_accuracy: 0.7200 - val_loss: 1.1212
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9519 - loss:
0.1704 - val_accuracy: 0.7000 - val_loss: 1.1984
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9662 - loss:
0.1200 - val_accuracy: 0.7100 - val_loss: 1.3474
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9656 - loss:
0.1097 - val_accuracy: 0.7120 - val_loss: 1.4993
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9718 - loss:
0.0864 - val_accuracy: 0.6920 - val_loss: 1.4392
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9702 - loss:
0.0940 - val_accuracy: 0.6720 - val_loss: 1.7768
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.9616 - loss:
0.1191 - val_accuracy: 0.7120 - val_loss: 1.4864
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9917 - loss:
0.0383 - val_accuracy: 0.6940 - val_loss: 1.5624
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9872 - loss:
0.0392 - val_accuracy: 0.7300 - val_loss: 1.6255
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9937 - loss:
0.0214 - val_accuracy: 0.7060 - val_loss: 1.6978
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9995 - loss:
0.0070 - val_accuracy: 0.7140 - val_loss: 1.6486
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 1.0000 - loss:
0.0032 - val_accuracy: 0.7200 - val_loss: 1.7018
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 1.0000 - loss:
0.0018 - val_accuracy: 0.7180 - val_loss: 1.7128
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 1.0000 - loss:
0.0013 - val_accuracy: 0.7160 - val_loss: 1.7357
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 1.0000 - loss:
0.0011 - val_accuracy: 0.7160 - val_loss: 1.7538
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.7615 - loss:
```

```
1.4767
Test Accuracy (Swish + LeakyReLU): 71.60%
```

Task 5.3.9 Conclusion:

The CNN model utilizing Swish activation in convolutional layers and LeakyReLU in fully connected layers achieved a test accuracy of 71.60%, which is slightly lower than the Swish-only model (72.20%) and the ReLU-based model (73.60%). While the combination aimed to leverage Swish's smooth activation for feature extraction and LeakyReLU's ability to mitigate neuron dying issues, it did not significantly improve performance. The results suggest that Swish remains effective in convolutional layers, but the expected benefits of LeakyReLU in fully connected layers did not lead to a notable accuracy boost.

Task 5.3.10: Experimenting with Custom (Scaled Swish) activation function

```
# Define a custom Swish activation function with a scaling factor
def custom_swish(x):
    return 1.5 * x * sigmoid(x)  # Swish activation with scaling
factor 1.5

# Define CNN model using the Custom Swish activation function
model_custom_swish = models.Sequential([
    layers.Input(shape=(32, 32, 3)),  # Input layer for 32x32 RGB
images

    # First Convolutional Block with Custom Swish Activation
    layers.Conv2D(64, (3, 3), activation=custom_swish,
padding='same'),  # 64 filters, kernel size 3x3
    layers.Conv2D(64, (3, 3), activation=custom_swish,
padding='same'),
    layers.MaxPooling2D((2, 2)),  # Max pooling to reduce spatial
dimensions
    layers.Dropout(0.2),  # Dropout to prevent overfitting

    # Second Convolutional Block with Custom Swish Activation
    layers.Conv2D(128, (3, 3), activation=custom_swish,
padding='same'),  # 128 filters
    layers.Conv2D(128, (3, 3), activation=custom_swish,
padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    # Fully Connected Layers
    layers.Flatten(),  # Flatten feature maps into a single vector
    layers.Dense(512, activation=custom_swish),  # Fully connected
layer with Custom Swish activation
    layers.Dropout(0.5),  # Dropout to reduce overfitting
    layers.Dense(10, activation='softmax')  # Output layer with 10
classes for classification
])
```

```
# Compile the model using Adam optimizer and categorical cross-entropy
loss
model_custom_swish.compile(optimizer='adam',
                           loss='categorical_crossentropy',
                           metrics=['accuracy'])

# Train the model using the training dataset
history_custom_swish = model_custom_swish.fit(x_train_final,
y_train_final,
                                              epochs=25,  # Train for
25 epochs
                                              batch_size=64,  # Use
batch size of 64

validation_data=(x_test_final, y_test_final))  # Validate on test data

# Evaluate the model on the test dataset
test_loss, test_acc = model_custom_swish.evaluate(x_test_final,
y_test_final, verbose=0)
print(f'Test Accuracy (Custom Swish): {test_acc * 100:.2f}%')
```

```
Epoch 1/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 11s 116ms/step - accuracy: 0.2935 - loss:
1.9357 - val_accuracy: 0.5620 - val_loss: 1.2914
Epoch 2/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5225 - loss:
1.3284 - val_accuracy: 0.5540 - val_loss: 1.1760
Epoch 3/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.5961 - loss:
1.1321 - val_accuracy: 0.6100 - val_loss: 1.0376
Epoch 4/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.6708 - loss:
0.9577 - val_accuracy: 0.6420 - val_loss: 1.0626
Epoch 5/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.7159 - loss:
0.7758 - val_accuracy: 0.6560 - val_loss: 0.9792
Epoch 6/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.7855 - loss:
0.6148 - val_accuracy: 0.6460 - val_loss: 1.0075
Epoch 7/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.8345 - loss:
0.4640 - val_accuracy: 0.6480 - val_loss: 1.2183
Epoch 8/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.8588 - loss:
0.4193 - val_accuracy: 0.6300 - val_loss: 1.2055
Epoch 9/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9026 - loss:
0.2687 - val_accuracy: 0.6500 - val_loss: 1.3265
Epoch 10/25
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9156 - loss:
0.2309 - val_accuracy: 0.6620 - val_loss: 1.4267
Epoch 11/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9335 - loss:
0.1946 - val_accuracy: 0.6500 - val_loss: 1.4361
Epoch 12/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9550 - loss:
0.1353 - val_accuracy: 0.6400 - val_loss: 1.5653
Epoch 13/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9454 - loss:
0.1649 - val_accuracy: 0.6600 - val_loss: 1.6299
Epoch 14/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.9467 - loss:
0.1450 - val_accuracy: 0.6440 - val_loss: 1.7076
Epoch 15/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9528 - loss:
0.1417 - val_accuracy: 0.6620 - val_loss: 1.6325
Epoch 16/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9750 - loss:
0.0850 - val_accuracy: 0.6860 - val_loss: 1.7716
Epoch 17/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9729 - loss:
0.0853 - val_accuracy: 0.6380 - val_loss: 1.7558
Epoch 18/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9699 - loss:
0.0870 - val_accuracy: 0.6660 - val_loss: 1.9307
Epoch 19/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9715 - loss:
0.0920 - val_accuracy: 0.6700 - val_loss: 1.9144
Epoch 20/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9703 - loss:
0.0862 - val_accuracy: 0.6260 - val_loss: 2.5791
Epoch 21/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9467 - loss:
0.1528 - val_accuracy: 0.7000 - val_loss: 1.5744
Epoch 22/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9576 - loss:
0.1316 - val_accuracy: 0.6620 - val_loss: 1.7575
Epoch 23/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9739 - loss:
0.0816 - val_accuracy: 0.6800 - val_loss: 1.6773
Epoch 24/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9822 - loss:
0.0638 - val_accuracy: 0.6780 - val_loss: 1.9313
Epoch 25/25
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9831 - loss:
0.0480 - val_accuracy: 0.6600 - val_loss: 1.9865
Test Accuracy (Custom Swish): 66.00%
```

Task 5.3.10 Conclusion:

The CNN model incorporating a custom Swish activation function with a scaling factor of 1.5 achieved a test accuracy of 66.00%, which is lower than other configurations, including the standard Swish (72.20%) and ReLU (73.60%) models. While the scaling factor was intended to enhance the Swish function's flexibility, it may have led to vanishing or exploding activations, reducing the model's ability to learn effectively. Additionally, the deeper convolutional layers and fully connected layers may have been affected by the modified activation function's gradient behavior. These results suggest that while Swish is effective, tuning its scaling factor requires careful experimentation, and standard implementations may perform more reliably in this scenario.

Task 5.4: Experiment with various optimizers (https://keras.io/api/optimizers/) and learning rate. What is the effect on the resulting model accuracy?

Experimenting with different optimizers including SGD, Adam, RMSprop, Adagrad, and AdamW, each with different learning rates.

```
# Define the CNN model function
def create_model():
    model = keras.Sequential([
        layers.Input(shape=(32, 32, 3)),  # Input layer for 32x32 RGB
images

        # First Convolutional Block
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
# 32 filters
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),  # Reducing spatial dimensions
        layers.Dropout(0.25),  # Dropout to reduce overfitting

        # Second Convolutional Block
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
# 64 filters
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Third Convolutional Block
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
# 128 filters
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Fully Connected Layers
        layers.Flatten(),  # Converts feature maps into a 1D vector
        layers.Dense(512, activation='relu'),  # Fully connected layer
```

```
with 512 neurons
        layers.Dropout(0.5),  # Dropout to prevent overfitting
        layers.Dense(10, activation='softmax')  # Output layer with 10
classes
    ])
    return model  # Return the constructed model

# Define optimizers and learning rates to test
optimizers_list = {
    "SGD_0.01": optimizers.SGD(learning_rate=0.01),
    "SGD_0.001": optimizers.SGD(learning_rate=0.001),
    "Adam_0.01": optimizers.Adam(learning_rate=0.01),
    "Adam_0.001": optimizers.Adam(learning_rate=0.001),
    "RMSprop_0.001": optimizers.RMSprop(learning_rate=0.001),
    "RMSprop_0.0001": optimizers.RMSprop(learning_rate=0.0001),
    "Adagrad_0.01": optimizers.Adagrad(learning_rate=0.01),
    "Adagrad_0.001": optimizers.Adagrad(learning_rate=0.001),
    "AdamW_0.001": optimizers.AdamW(learning_rate=0.001)
}

# Dictionary to store test accuracy results for each optimizer
results = {}

# Iterate over each optimizer, train the model, and store results
for name, opt in optimizers_list.items():
    print(f"\nTraining with {name}...\n")

    # Create and compile the CNN model with the selected optimizer
    model = create_model()
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',  # Loss function
for multi-class classification
                  metrics=['accuracy'])  # Track accuracy during
training

    # Train the model
    history = model.fit(x_train_final, y_train_final,
                        epochs=10,  # Short training for quick
comparison
                        batch_size=64,  # Mini-batch size
                        validation_data=(x_test_final, y_test_final),
# Validate after each epoch
                        verbose=1)

    # Evaluate the model on the test dataset
    test_loss, test_acc = model.evaluate(x_test_final, y_test_final,
verbose=0)
    print(f"Test Accuracy with {name}: {test_acc * 100:.2f}%")

    # Store test accuracy in results dictionary
```

```python
    results[name] = test_acc

# Print final test accuracy results for all optimizers
print("\nFinal Results Summary:")
for key, value in results.items():
    print(f"{key}: {value * 100:.2f}%")
```

```
Training with SGD_0.01...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 9s 108ms/step - accuracy: 0.1019 - loss: 
2.3056 - val_accuracy: 0.1220 - val_loss: 2.3022
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.1051 - loss: 
2.3037 - val_accuracy: 0.1100 - val_loss: 2.3016
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.1102 - loss: 
2.3019 - val_accuracy: 0.1260 - val_loss: 2.3004
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1002 - loss: 
2.3009 - val_accuracy: 0.1400 - val_loss: 2.2995
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1175 - loss: 
2.2996 - val_accuracy: 0.1400 - val_loss: 2.2981
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1190 - loss: 
2.2987 - val_accuracy: 0.1760 - val_loss: 2.2966
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1142 - loss: 
2.2961 - val_accuracy: 0.1380 - val_loss: 2.2940
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1294 - loss: 
2.2938 - val_accuracy: 0.1740 - val_loss: 2.2908
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1382 - loss: 
2.2908 - val_accuracy: 0.1620 - val_loss: 2.2823
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.1219 - loss: 
2.2815 - val_accuracy: 0.2140 - val_loss: 2.2664
Test Accuracy with SGD_0.01: 21.40%

Training with SGD_0.001...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 9s 100ms/step - accuracy: 0.0983 - loss: 
2.3054 - val_accuracy: 0.1200 - val_loss: 2.3036
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.0980 - loss: 
2.3079 - val_accuracy: 0.1200 - val_loss: 2.3033
```

```
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1017 - loss:
2.3061 - val_accuracy: 0.1240 - val_loss: 2.3029
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1008 - loss:
2.3060 - val_accuracy: 0.1300 - val_loss: 2.3026
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1007 - loss:
2.3041 - val_accuracy: 0.1320 - val_loss: 2.3024
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1049 - loss:
2.3030 - val_accuracy: 0.1320 - val_loss: 2.3021
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.0954 - loss:
2.3039 - val_accuracy: 0.1360 - val_loss: 2.3019
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1085 - loss:
2.3009 - val_accuracy: 0.1280 - val_loss: 2.3016
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1059 - loss:
2.3029 - val_accuracy: 0.1320 - val_loss: 2.3014
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1048 - loss:
2.3022 - val_accuracy: 0.1340 - val_loss: 2.3012
Test Accuracy with SGD_0.001: 13.40%

Training with Adam_0.01...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 13s 130ms/step - accuracy: 0.0950 - loss:
4.2934 - val_accuracy: 0.1000 - val_loss: 2.3043
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.0919 - loss:
2.3122 - val_accuracy: 0.1000 - val_loss: 2.3030
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1087 - loss:
2.3029 - val_accuracy: 0.1000 - val_loss: 2.3028
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1190 - loss:
2.3027 - val_accuracy: 0.1000 - val_loss: 2.3027
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1019 - loss:
2.3031 - val_accuracy: 0.1000 - val_loss: 2.3027
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1091 - loss:
2.3035 - val_accuracy: 0.1000 - val_loss: 2.3027
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.0859 - loss:
2.3039 - val_accuracy: 0.1000 - val_loss: 2.3028
```

```
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.1009 - loss:
2.3032 - val_accuracy: 0.1000 - val_loss: 2.3027
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0970 - loss:
2.3032 - val_accuracy: 0.1000 - val_loss: 2.3026
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0952 - loss:
2.3030 - val_accuracy: 0.1000 - val_loss: 2.3027
Test Accuracy with Adam_0.01: 10.00%

Training with Adam_0.001...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 12s 130ms/step - accuracy: 0.1648 - loss:
2.1612 - val_accuracy: 0.2860 - val_loss: 1.7500
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.3188 - loss:
1.7770 - val_accuracy: 0.3800 - val_loss: 1.5707
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3973 - loss:
1.5944 - val_accuracy: 0.4500 - val_loss: 1.3912
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.4515 - loss:
1.4026 - val_accuracy: 0.4980 - val_loss: 1.3616
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.4997 - loss:
1.3556 - val_accuracy: 0.5500 - val_loss: 1.2330
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5468 - loss:
1.2090 - val_accuracy: 0.5560 - val_loss: 1.2477
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5698 - loss:
1.2008 - val_accuracy: 0.6440 - val_loss: 1.0483
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5899 - loss:
1.1027 - val_accuracy: 0.6380 - val_loss: 1.0314
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.5937 - loss:
1.0513 - val_accuracy: 0.6400 - val_loss: 1.0178
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.6442 - loss:
0.9515 - val_accuracy: 0.6220 - val_loss: 0.9550
Test Accuracy with Adam_0.001: 62.20%

Training with RMSprop_0.001...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 11s 139ms/step - accuracy: 0.1090 - loss:
2.2987 - val_accuracy: 0.2560 - val_loss: 1.9936
```

```
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.2633 - loss:
1.9231 - val_accuracy: 0.3280 - val_loss: 1.6982
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3027 - loss:
1.8363 - val_accuracy: 0.4180 - val_loss: 1.6167
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3694 - loss:
1.6945 - val_accuracy: 0.3840 - val_loss: 1.5892
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3996 - loss:
1.6115 - val_accuracy: 0.4880 - val_loss: 1.3571
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.4550 - loss:
1.4552 - val_accuracy: 0.5120 - val_loss: 1.2487
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.4782 - loss:
1.3961 - val_accuracy: 0.5640 - val_loss: 1.1866
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.4909 - loss:
1.3863 - val_accuracy: 0.5000 - val_loss: 1.3434
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5100 - loss:
1.3056 - val_accuracy: 0.5960 - val_loss: 1.1130
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5552 - loss:
1.1991 - val_accuracy: 0.5760 - val_loss: 1.1579
Test Accuracy with RMSprop_0.001: 57.60%

Training with RMSprop_0.0001...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 10s 115ms/step - accuracy: 0.0961 - loss:
2.3030 - val_accuracy: 0.1820 - val_loss: 2.2871
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.1772 - loss:
2.2285 - val_accuracy: 0.2560 - val_loss: 1.9380
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.2656 - loss:
1.9214 - val_accuracy: 0.2820 - val_loss: 1.8427
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.2861 - loss:
1.8401 - val_accuracy: 0.3220 - val_loss: 1.7575
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3188 - loss:
1.7854 - val_accuracy: 0.3220 - val_loss: 1.7305
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3384 - loss:
1.7559 - val_accuracy: 0.3540 - val_loss: 1.6654
```

```
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3433 - loss:
1.7132 - val_accuracy: 0.3720 - val_loss: 1.6116
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3301 - loss:
1.7108 - val_accuracy: 0.4180 - val_loss: 1.5919
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3788 - loss:
1.6460 - val_accuracy: 0.4020 - val_loss: 1.5507
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3734 - loss:
1.6473 - val_accuracy: 0.3880 - val_loss: 1.7175
Test Accuracy with RMSprop_0.0001: 38.80%

Training with Adagrad_0.01...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 11s 114ms/step - accuracy: 0.0958 - loss:
2.3051 - val_accuracy: 0.1880 - val_loss: 2.2965
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1234 - loss:
2.2962 - val_accuracy: 0.2180 - val_loss: 2.2789
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.1554 - loss:
2.2526 - val_accuracy: 0.2160 - val_loss: 2.2102
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.2271 - loss:
2.0913 - val_accuracy: 0.2820 - val_loss: 1.8560
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.2481 - loss:
1.9696 - val_accuracy: 0.2620 - val_loss: 1.8900
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.2622 - loss:
1.8640 - val_accuracy: 0.3600 - val_loss: 1.7406
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.2910 - loss:
1.8049 - val_accuracy: 0.3380 - val_loss: 1.7020
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3178 - loss:
1.7655 - val_accuracy: 0.3840 - val_loss: 1.6464
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3229 - loss:
1.7329 - val_accuracy: 0.3860 - val_loss: 1.6144
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.3360 - loss:
1.6995 - val_accuracy: 0.4240 - val_loss: 1.5907
Test Accuracy with Adagrad_0.01: 42.40%

Training with Adagrad_0.001...
```

```
Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 10s 113ms/step - accuracy: 0.0898 - loss:
2.3049 - val_accuracy: 0.1020 - val_loss: 2.3019
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.1068 - loss:
2.3039 - val_accuracy: 0.1240 - val_loss: 2.3015
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0952 - loss:
2.3043 - val_accuracy: 0.1340 - val_loss: 2.3011
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1171 - loss:
2.3012 - val_accuracy: 0.1500 - val_loss: 2.3007
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1116 - loss:
2.3025 - val_accuracy: 0.1440 - val_loss: 2.3004
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.1166 - loss:
2.3013 - val_accuracy: 0.1560 - val_loss: 2.3002
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.0923 - loss:
2.3022 - val_accuracy: 0.1660 - val_loss: 2.2998
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.1088 - loss:
2.3000 - val_accuracy: 0.1740 - val_loss: 2.2994
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.1049 - loss:
2.3007 - val_accuracy: 0.1880 - val_loss: 2.2990
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.0929 - loss:
2.3025 - val_accuracy: 0.1860 - val_loss: 2.2987
Test Accuracy with Adagrad_0.001: 18.60%

Training with AdamW_0.001...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 13s 145ms/step - accuracy: 0.1144 - loss:
2.2385 - val_accuracy: 0.3000 - val_loss: 1.8200
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.2866 - loss:
1.8639 - val_accuracy: 0.4180 - val_loss: 1.6273
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3819 - loss:
1.6497 - val_accuracy: 0.4720 - val_loss: 1.4471
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.4447 - loss:
1.4754 - val_accuracy: 0.5400 - val_loss: 1.3055
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.4963 - loss:
1.3774 - val_accuracy: 0.5220 - val_loss: 1.2526
```

```
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5240 - loss:
1.2850 - val_accuracy: 0.5440 - val_loss: 1.1878
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5596 - loss:
1.1653 - val_accuracy: 0.5840 - val_loss: 1.1197
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.6019 - loss:
1.1022 - val_accuracy: 0.6220 - val_loss: 1.0505
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6033 - loss:
1.0657 - val_accuracy: 0.6840 - val_loss: 0.9456
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.6224 - loss:
1.0129 - val_accuracy: 0.6640 - val_loss: 0.9511
Test Accuracy with AdamW_0.001: 66.40%

Final Results Summary:
SGD_0.01: 21.40%
SGD_0.001: 13.40%
Adam_0.01: 10.00%
Adam_0.001: 62.20%
RMSprop_0.001: 57.60%
RMSprop_0.0001: 38.80%
Adagrad_0.01: 42.40%
Adagrad_0.001: 18.60%
AdamW_0.001: 66.40%
```

Task 5.4.1 Conclusion:

The optimizer comparison experiment revealed significant variations in test accuracy across different optimization algorithms and learning rates. Among the tested optimizers, AdamW with a learning rate of 0.001 achieved the highest accuracy (66.40%), followed by Adam (62.20%) and RMSprop (57.60% with the same learning rate. In contrast, SGD (both 0.01 and 0.001) and Adam (0.01) performed poorly, with accuracies below 22%, indicating that these configurations failed to converge effectively within the given training epochs. The results highlight the importance of selecting an appropriate optimizer and learning rate, with AdamW (0.001) emerging as the most effective choice for this CNN model in terms of classification accuracy.

Task 5.5: With all the above variations, experiment with various batch sizes and epochs for training

In the code below, we will train and evaluate a CNN model while experimenting with different batch sizes and epochs to analyze their impact on accuracy. It systematically loops through batch sizes (32, 64, 128) and epochs (10, 20, 30), training the model and recording test accuracy for each combination. The results are stored and visualized in a plot, helping identify the best configuration for training. The script also ensures reproducibility by setting random seeds and prevents memory issues by clearing TensorFlow sessions before each training run.

```
# Set seed for reproducibility
tf.random.set_seed(42)
np.random.seed(42)

# Function to create the CNN model
def create_model():
    model = keras.Sequential([
        layers.Input(shape=(32, 32, 3)),  # Input layer for 32x32 RGB
images

        # First Convolutional Block
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
# 32 filters
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),  # Downsample the feature maps
        layers.Dropout(0.25),  # Dropout to reduce overfitting

        # Second Convolutional Block
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
# 64 filters
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Third Convolutional Block
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
# 128 filters
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Fully Connected Layers
        layers.Flatten(),  # Flatten feature maps into a 1D vector
        layers.Dense(512, activation='relu'),  # Fully connected layer
with 512 neurons
        layers.Dropout(0.5),  # Dropout to further reduce overfitting
        layers.Dense(10, activation='softmax')  # Output layer with 10
classes (for classification)
    ])
    return model  # Return the model

# Experiment variations: different batch sizes and epochs
batch_sizes = [32, 64, 128]  # Mini-batch sizes to test
epochs_list = [10, 20, 30]  # Number of epochs to test

# Dictionary to store results of experiments
results = {}

# Loop through different batch sizes and epoch values
for batch in batch_sizes:
```

```
    for epoch in epochs_list:
        print(f"\nTraining with batch size {batch} and {epoch}
epochs...\n")

        # Clear TensorFlow session to free up memory and avoid
unexpected state issues
        K.clear_session()

        # Define optimizer (AdamW) inside the loop to ensure a fresh
instance each time
        best_optimizer = optimizers.AdamW(learning_rate=0.001)

        # Create and compile a new CNN model
        model = create_model()
        model.compile(optimizer=best_optimizer,
                      loss='categorical_crossentropy',  # Loss
function for multi-class classification
                      metrics=['accuracy'])  # Track accuracy during
training

        # Train the model with the current batch size and epoch count
        history = model.fit(x_train_final, y_train_final,
                            epochs=epoch,  # Train for the selected
number of epochs
                            batch_size=batch,  # Use the selected
batch size
                            validation_data=(x_test_final,
y_test_final),  # Validate on test data
                            verbose=1)

        # Evaluate the trained model on the test dataset
        test_loss, test_acc = model.evaluate(x_test_final,
y_test_final, verbose=0)
        print(f"Test Accuracy with batch={batch}, epochs={epoch}:
{test_acc * 100:.2f}%")

        # Store the test accuracy result
        results[(batch, epoch)] = test_acc

# Print final results summary
print("\nFinal Results Summary:")
for key, value in results.items():
    print(f"Batch Size {key[0]}, Epochs {key[1]}: {value * 100:.2f}%")

# Plot results to visualize the impact of batch size and epochs on
accuracy
plt.figure(figsize=(10, 5))

# Plot accuracy trends for different batch sizes
for batch in batch_sizes:
    accs = [results[(batch, epoch)] * 100 for epoch in epochs_list]
```

```
    plt.plot(epochs_list, accs, marker='o', linestyle='-',
label=f'Batch Size {batch}')

# Label the plot
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy (%)')
plt.title('Effect of Batch Size & Epochs on Accuracy')
plt.legend()
plt.grid(True)

# Display the plot
plt.show()


Training with batch size 32 and 10 epochs...

Epoch 1/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.1438 - loss:
2.2113 - val_accuracy: 0.3360 - val_loss: 1.8112
Epoch 2/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 25ms/step - accuracy: 0.3306 - loss:
1.7505 - val_accuracy: 0.4500 - val_loss: 1.4629
Epoch 3/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - accuracy: 0.4241 - loss:
1.5231 - val_accuracy: 0.5260 - val_loss: 1.3049
Epoch 4/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - accuracy: 0.4927 - loss:
1.3710 - val_accuracy: 0.5580 - val_loss: 1.1760
Epoch 5/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - accuracy: 0.5230 - loss:
1.2549 - val_accuracy: 0.6080 - val_loss: 1.1104
Epoch 6/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 26ms/step - accuracy: 0.5517 - loss:
1.1670 - val_accuracy: 0.5740 - val_loss: 1.0518
Epoch 7/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 4s 39ms/step - accuracy: 0.5816 - loss:
1.1345 - val_accuracy: 0.6100 - val_loss: 1.0296
Epoch 8/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.5930 - loss:
1.0656 - val_accuracy: 0.6360 - val_loss: 1.0048
Epoch 9/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6596 - loss:
0.9594 - val_accuracy: 0.6660 - val_loss: 0.9098
Epoch 10/10
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.6550 - loss:
0.9214 - val_accuracy: 0.6320 - val_loss: 0.9343
Test Accuracy with batch=32, epochs=10: 63.20%

Training with batch size 32 and 20 epochs...
```

```
Epoch 1/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 12s 63ms/step - accuracy: 0.1377 - loss:
2.2215 - val_accuracy: 0.3320 - val_loss: 1.6979
Epoch 2/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.3193 - loss:
1.7679 - val_accuracy: 0.3920 - val_loss: 1.5902
Epoch 3/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.3866 - loss:
1.5998 - val_accuracy: 0.4860 - val_loss: 1.3475
Epoch 4/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.4626 - loss:
1.4296 - val_accuracy: 0.4980 - val_loss: 1.3037
Epoch 5/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5257 - loss:
1.2297 - val_accuracy: 0.5340 - val_loss: 1.2141
Epoch 6/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5469 - loss:
1.1661 - val_accuracy: 0.5500 - val_loss: 1.1807
Epoch 7/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5992 - loss:
1.0870 - val_accuracy: 0.5960 - val_loss: 1.1106
Epoch 8/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6126 - loss:
1.0467 - val_accuracy: 0.6520 - val_loss: 1.0065
Epoch 9/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.6312 - loss:
0.9976 - val_accuracy: 0.6460 - val_loss: 0.9543
Epoch 10/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.6254 - loss:
0.9750 - val_accuracy: 0.6480 - val_loss: 0.9621
Epoch 11/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6646 - loss:
0.8710 - val_accuracy: 0.7040 - val_loss: 0.8263
Epoch 12/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6921 - loss:
0.8065 - val_accuracy: 0.6720 - val_loss: 0.9123
Epoch 13/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7065 - loss:
0.7877 - val_accuracy: 0.6880 - val_loss: 0.8487
Epoch 14/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7389 - loss:
0.6923 - val_accuracy: 0.6860 - val_loss: 0.8548
Epoch 15/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - accuracy: 0.7794 - loss:
0.6018 - val_accuracy: 0.6760 - val_loss: 0.9649
Epoch 16/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.7659 - loss:
0.6107 - val_accuracy: 0.6860 - val_loss: 0.8605
Epoch 17/20
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7944 - loss:
0.5700 - val_accuracy: 0.7020 - val_loss: 0.8192
Epoch 18/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8289 - loss:
0.4667 - val_accuracy: 0.7220 - val_loss: 0.8620
Epoch 19/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8319 - loss:
0.4555 - val_accuracy: 0.7020 - val_loss: 0.8614
Epoch 20/20
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8252 - loss:
0.4439 - val_accuracy: 0.6900 - val_loss: 0.9868
Test Accuracy with batch=32, epochs=20: 69.00%

Training with batch size 32 and 30 epochs...

Epoch 1/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 12s 62ms/step - accuracy: 0.1801 - loss:
2.1350 - val_accuracy: 0.3740 - val_loss: 1.7098
Epoch 2/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.3345 - loss:
1.7237 - val_accuracy: 0.3880 - val_loss: 1.5568
Epoch 3/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.4213 - loss:
1.5305 - val_accuracy: 0.5480 - val_loss: 1.2845
Epoch 4/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5035 - loss:
1.3509 - val_accuracy: 0.5260 - val_loss: 1.2598
Epoch 5/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5558 - loss:
1.2522 - val_accuracy: 0.5520 - val_loss: 1.1828
Epoch 6/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.5662 - loss:
1.1721 - val_accuracy: 0.6180 - val_loss: 1.0472
Epoch 7/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6031 - loss:
1.0783 - val_accuracy: 0.6000 - val_loss: 1.1019
Epoch 8/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.6423 - loss:
1.0332 - val_accuracy: 0.6200 - val_loss: 1.0518
Epoch 9/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.6765 - loss:
0.9223 - val_accuracy: 0.6740 - val_loss: 0.9384
Epoch 10/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.6786 - loss:
0.8741 - val_accuracy: 0.6880 - val_loss: 0.8757
Epoch 11/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7119 - loss:
0.8051 - val_accuracy: 0.6400 - val_loss: 0.9453
Epoch 12/30
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7364 - loss:
0.7205 - val_accuracy: 0.6840 - val_loss: 0.8927
Epoch 13/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.7478 - loss:
0.6900 - val_accuracy: 0.6720 - val_loss: 0.8483
Epoch 14/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.7791 - loss:
0.6003 - val_accuracy: 0.7100 - val_loss: 0.8939
Epoch 15/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.7938 - loss:
0.5681 - val_accuracy: 0.7040 - val_loss: 0.9082
Epoch 16/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - accuracy: 0.8041 - loss:
0.5481 - val_accuracy: 0.7140 - val_loss: 0.9440
Epoch 17/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8218 - loss:
0.4903 - val_accuracy: 0.7220 - val_loss: 0.8283
Epoch 18/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8394 - loss:
0.4514 - val_accuracy: 0.6960 - val_loss: 0.9690
Epoch 19/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8476 - loss:
0.4079 - val_accuracy: 0.7080 - val_loss: 1.0368
Epoch 20/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.8607 - loss:
0.3608 - val_accuracy: 0.6940 - val_loss: 1.0176
Epoch 21/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.8633 - loss:
0.3695 - val_accuracy: 0.7180 - val_loss: 0.9555
Epoch 22/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8775 - loss:
0.3289 - val_accuracy: 0.6800 - val_loss: 0.9958
Epoch 23/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.8862 - loss:
0.3268 - val_accuracy: 0.7260 - val_loss: 0.9450
Epoch 24/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - accuracy: 0.8793 - loss:
0.3093 - val_accuracy: 0.7140 - val_loss: 1.0261
Epoch 25/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.8785 - loss:
0.3270 - val_accuracy: 0.7220 - val_loss: 1.1883
Epoch 26/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.8943 - loss:
0.3016 - val_accuracy: 0.6860 - val_loss: 1.1469
Epoch 27/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9156 - loss:
0.2485 - val_accuracy: 0.7180 - val_loss: 1.0958
Epoch 28/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9269 - loss:
```

```
0.1846 - val_accuracy: 0.6940 - val_loss: 1.3499
Epoch 29/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9325 - loss:
0.1881 - val_accuracy: 0.7280 - val_loss: 1.0377
Epoch 30/30
94/94 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9387 - loss:
0.1649 - val_accuracy: 0.7100 - val_loss: 1.1787
Test Accuracy with batch=32, epochs=30: 71.00%

Training with batch size 64 and 10 epochs...

Epoch 1/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 13s 123ms/step - accuracy: 0.1412 - loss:
2.2614 - val_accuracy: 0.3040 - val_loss: 1.8008
Epoch 2/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.2910 - loss:
1.8083 - val_accuracy: 0.3760 - val_loss: 1.5973
Epoch 3/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.3751 - loss:
1.6541 - val_accuracy: 0.4540 - val_loss: 1.5563
Epoch 4/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.3976 - loss:
1.5627 - val_accuracy: 0.5020 - val_loss: 1.3468
Epoch 5/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.4626 - loss:
1.4643 - val_accuracy: 0.5140 - val_loss: 1.2766
Epoch 6/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.4928 - loss:
1.3752 - val_accuracy: 0.5740 - val_loss: 1.1587
Epoch 7/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5578 - loss:
1.1853 - val_accuracy: 0.5240 - val_loss: 1.2892
Epoch 8/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5642 - loss:
1.1828 - val_accuracy: 0.6120 - val_loss: 1.0212
Epoch 9/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6200 - loss:
1.0362 - val_accuracy: 0.6460 - val_loss: 0.9858
Epoch 10/10
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6150 - loss:
1.0028 - val_accuracy: 0.6500 - val_loss: 0.8882
Test Accuracy with batch=64, epochs=10: 65.00%

Training with batch size 64 and 20 epochs...

Epoch 1/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 12s 118ms/step - accuracy: 0.1442 - loss:
2.2180 - val_accuracy: 0.2420 - val_loss: 1.8542
Epoch 2/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.2712 - loss:
```

```
1.8571 - val_accuracy: 0.3860 - val_loss: 1.5803
Epoch 3/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.3805 - loss:
1.6631 - val_accuracy: 0.4640 - val_loss: 1.4240
Epoch 4/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.4198 - loss:
1.4979 - val_accuracy: 0.4800 - val_loss: 1.3468
Epoch 5/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.4733 - loss:
1.4160 - val_accuracy: 0.5320 - val_loss: 1.3107
Epoch 6/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5123 - loss:
1.3111 - val_accuracy: 0.5460 - val_loss: 1.2601
Epoch 7/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5445 - loss:
1.2184 - val_accuracy: 0.5920 - val_loss: 1.0773
Epoch 8/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.5618 - loss:
1.1292 - val_accuracy: 0.6040 - val_loss: 1.0586
Epoch 9/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6085 - loss:
1.0775 - val_accuracy: 0.6560 - val_loss: 1.0008
Epoch 10/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6395 - loss:
1.0125 - val_accuracy: 0.6360 - val_loss: 0.9623
Epoch 11/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6477 - loss:
0.9480 - val_accuracy: 0.6620 - val_loss: 0.9256
Epoch 12/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6609 - loss:
0.9202 - val_accuracy: 0.6380 - val_loss: 0.8962
Epoch 13/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6902 - loss:
0.8227 - val_accuracy: 0.5980 - val_loss: 1.0771
Epoch 14/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7184 - loss:
0.8206 - val_accuracy: 0.6700 - val_loss: 0.8997
Epoch 15/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.7439 - loss:
0.7208 - val_accuracy: 0.6600 - val_loss: 0.8917
Epoch 16/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.7575 - loss:
0.6560 - val_accuracy: 0.6960 - val_loss: 0.8325
Epoch 17/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7774 - loss:
0.6179 - val_accuracy: 0.7100 - val_loss: 0.8461
Epoch 18/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7680 - loss:
0.6169 - val_accuracy: 0.6900 - val_loss: 0.8503
```

```
Epoch 19/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7921 - loss:
0.5373 - val_accuracy: 0.6820 - val_loss: 0.9375
Epoch 20/20
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8028 - loss:
0.5332 - val_accuracy: 0.7100 - val_loss: 0.8048
Test Accuracy with batch=64, epochs=20: 71.00%

Training with batch size 64 and 30 epochs...

Epoch 1/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 12s 112ms/step - accuracy: 0.1295 - loss:
2.2309 - val_accuracy: 0.3040 - val_loss: 1.7729
Epoch 2/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.3041 - loss:
1.8113 - val_accuracy: 0.3660 - val_loss: 1.6514
Epoch 3/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.3512 - loss:
1.6680 - val_accuracy: 0.4100 - val_loss: 1.5765
Epoch 4/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.4029 - loss:
1.5864 - val_accuracy: 0.4640 - val_loss: 1.4260
Epoch 5/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.4527 - loss:
1.4748 - val_accuracy: 0.5060 - val_loss: 1.3000
Epoch 6/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.4941 - loss:
1.3613 - val_accuracy: 0.5560 - val_loss: 1.1963
Epoch 7/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.5351 - loss:
1.2492 - val_accuracy: 0.5840 - val_loss: 1.1144
Epoch 8/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5547 - loss:
1.1879 - val_accuracy: 0.5920 - val_loss: 1.0896
Epoch 9/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.5758 - loss:
1.1473 - val_accuracy: 0.6000 - val_loss: 1.0330
Epoch 10/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.6092 - loss:
1.0705 - val_accuracy: 0.5980 - val_loss: 1.0489
Epoch 11/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6325 - loss:
1.0116 - val_accuracy: 0.6380 - val_loss: 0.9885
Epoch 12/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6312 - loss:
0.9879 - val_accuracy: 0.6320 - val_loss: 0.9282
Epoch 13/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6456 - loss:
0.9456 - val_accuracy: 0.6560 - val_loss: 0.9488
```

```
Epoch 14/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6807 - loss:
0.8595 - val_accuracy: 0.6620 - val_loss: 0.8690
Epoch 15/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.6604 - loss:
0.8585 - val_accuracy: 0.6560 - val_loss: 0.8801
Epoch 16/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7054 - loss:
0.7811 - val_accuracy: 0.6720 - val_loss: 0.8639
Epoch 17/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - accuracy: 0.7491 - loss:
0.6880 - val_accuracy: 0.6580 - val_loss: 0.8909
Epoch 18/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.7503 - loss:
0.6470 - val_accuracy: 0.6600 - val_loss: 0.8967
Epoch 19/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7515 - loss:
0.6649 - val_accuracy: 0.6740 - val_loss: 0.8913
Epoch 20/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7839 - loss:
0.5852 - val_accuracy: 0.6520 - val_loss: 0.9153
Epoch 21/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7975 - loss:
0.5543 - val_accuracy: 0.6840 - val_loss: 0.8995
Epoch 22/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.7920 - loss:
0.5584 - val_accuracy: 0.6840 - val_loss: 0.8818
Epoch 23/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.8186 - loss:
0.4837 - val_accuracy: 0.6960 - val_loss: 0.8902
Epoch 24/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.8367 - loss:
0.4446 - val_accuracy: 0.7220 - val_loss: 0.9136
Epoch 25/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.8491 - loss:
0.4144 - val_accuracy: 0.6840 - val_loss: 0.9535
Epoch 26/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8588 - loss:
0.3663 - val_accuracy: 0.7200 - val_loss: 0.9097
Epoch 27/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8791 - loss:
0.3284 - val_accuracy: 0.6900 - val_loss: 1.0774
Epoch 28/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.8512 - loss:
0.3879 - val_accuracy: 0.7220 - val_loss: 0.9483
Epoch 29/30
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.8866 - loss:
0.3138 - val_accuracy: 0.7060 - val_loss: 1.0214
Epoch 30/30
```

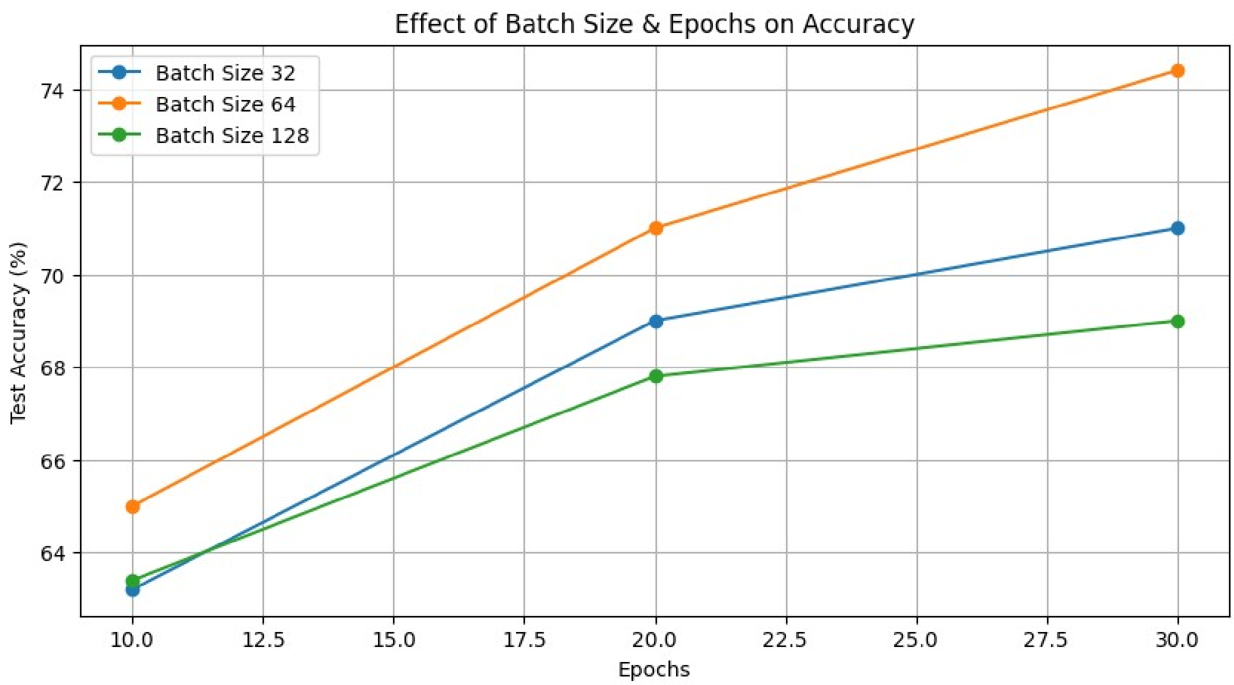
```
47/47 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9010 - loss:
0.2864 - val_accuracy: 0.7440 - val_loss: 0.9405
Test Accuracy with batch=64, epochs=30: 74.40%

Training with batch size 128 and 10 epochs...

Epoch 1/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 14s 283ms/step - accuracy: 0.1636 - loss:
2.1863 - val_accuracy: 0.2580 - val_loss: 1.8292
Epoch 2/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.2843 - loss:
1.8397 - val_accuracy: 0.3320 - val_loss: 1.6567
Epoch 3/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.3430 - loss:
1.6766 - val_accuracy: 0.3820 - val_loss: 1.5505
Epoch 4/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.3787 - loss:
1.5924 - val_accuracy: 0.4440 - val_loss: 1.4425
Epoch 5/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4384 - loss:
1.4541 - val_accuracy: 0.4920 - val_loss: 1.3493
Epoch 6/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4964 - loss:
1.3319 - val_accuracy: 0.6040 - val_loss: 1.1649
Epoch 7/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5371 - loss:
1.1841 - val_accuracy: 0.5820 - val_loss: 1.1094
Epoch 8/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5619 - loss:
1.1598 - val_accuracy: 0.6400 - val_loss: 1.0479
Epoch 9/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5867 - loss:
1.0637 - val_accuracy: 0.6280 - val_loss: 1.0237
Epoch 10/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5954 - loss:
1.0367 - val_accuracy: 0.6340 - val_loss: 1.0083
Test Accuracy with batch=128, epochs=10: 63.40%

Training with batch size 128 and 20 epochs...

Epoch 1/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 11s 220ms/step - accuracy: 0.1396 - loss:
2.2308 - val_accuracy: 0.3500 - val_loss: 1.8302
Epoch 2/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 35ms/step - accuracy: 0.2629 - loss:
1.8602 - val_accuracy: 0.3200 - val_loss: 1.7937
Epoch 3/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.3268 - loss:
1.7562 - val_accuracy: 0.4120 - val_loss: 1.4948
Epoch 4/20
```

```
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.3902 - loss:
1.5968 - val_accuracy: 0.4720 - val_loss: 1.4015
Epoch 5/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4331 - loss:
1.4826 - val_accuracy: 0.4980 - val_loss: 1.3002
Epoch 6/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4819 - loss:
1.4008 - val_accuracy: 0.5640 - val_loss: 1.1973
Epoch 7/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5131 - loss:
1.3090 - val_accuracy: 0.5680 - val_loss: 1.1322
Epoch 8/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5354 - loss:
1.2333 - val_accuracy: 0.5260 - val_loss: 1.2064
Epoch 9/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5403 - loss:
1.1787 - val_accuracy: 0.6100 - val_loss: 1.1257
Epoch 10/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6045 - loss:
1.1078 - val_accuracy: 0.6220 - val_loss: 1.0232
Epoch 11/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6008 - loss:
1.0781 - val_accuracy: 0.6400 - val_loss: 1.0117
Epoch 12/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6090 - loss:
1.0235 - val_accuracy: 0.6020 - val_loss: 1.0233
Epoch 13/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6366 - loss:
0.9829 - val_accuracy: 0.6500 - val_loss: 0.9468
Epoch 14/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6574 - loss:
0.9018 - val_accuracy: 0.6700 - val_loss: 0.9322
Epoch 15/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6549 - loss:
0.8977 - val_accuracy: 0.6120 - val_loss: 1.0255
Epoch 16/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6725 - loss:
0.8835 - val_accuracy: 0.6500 - val_loss: 0.9315
Epoch 17/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7039 - loss:
0.8148 - val_accuracy: 0.6760 - val_loss: 0.9559
Epoch 18/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.7189 - loss:
0.7920 - val_accuracy: 0.6560 - val_loss: 0.9108
Epoch 19/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.7387 - loss:
0.7229 - val_accuracy: 0.6880 - val_loss: 0.8899
Epoch 20/20
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.7552 - loss:
```

```
0.6670 - val_accuracy: 0.6780 - val_loss: 0.9173
Test Accuracy with batch=128, epochs=20: 67.80%

Training with batch size 128 and 30 epochs...

Epoch 1/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 11s 243ms/step - accuracy: 0.1111 - loss:
2.2880 - val_accuracy: 0.2800 - val_loss: 1.8774
Epoch 2/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.2653 - loss:
1.9083 - val_accuracy: 0.3540 - val_loss: 1.6935
Epoch 3/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.3562 - loss:
1.6951 - val_accuracy: 0.3980 - val_loss: 1.5759
Epoch 4/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.3919 - loss:
1.5929 - val_accuracy: 0.4480 - val_loss: 1.4454
Epoch 5/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4545 - loss:
1.4722 - val_accuracy: 0.4620 - val_loss: 1.3438
Epoch 6/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5015 - loss:
1.3678 - val_accuracy: 0.5160 - val_loss: 1.2952
Epoch 7/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4972 - loss:
1.3362 - val_accuracy: 0.5440 - val_loss: 1.1969
Epoch 8/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5230 - loss:
1.2550 - val_accuracy: 0.5640 - val_loss: 1.1604
Epoch 9/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5642 - loss:
1.1464 - val_accuracy: 0.5880 - val_loss: 1.1485
Epoch 10/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.5753 - loss:
1.1551 - val_accuracy: 0.6100 - val_loss: 1.1301
Epoch 11/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.6243 - loss:
1.0577 - val_accuracy: 0.5800 - val_loss: 1.0251
Epoch 12/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6344 - loss:
0.9884 - val_accuracy: 0.6440 - val_loss: 0.9522
Epoch 13/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6495 - loss:
0.9192 - val_accuracy: 0.6540 - val_loss: 0.9382
Epoch 14/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.6698 - loss:
0.8744 - val_accuracy: 0.6380 - val_loss: 0.9145
Epoch 15/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - accuracy: 0.6971 - loss:
0.8103 - val_accuracy: 0.6700 - val_loss: 0.8660
```

```
Epoch 16/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.6930 - loss: 0.8202 - val_accuracy: 0.6480 - val_loss: 0.9780
Epoch 17/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.6894 - loss: 0.8238 - val_accuracy: 0.6260 - val_loss: 0.9793
Epoch 18/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.7011 - loss: 0.7770 - val_accuracy: 0.6460 - val_loss: 0.9764
Epoch 19/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7109 - loss: 0.7438 - val_accuracy: 0.6880 - val_loss: 0.8431
Epoch 20/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7706 - loss: 0.6331 - val_accuracy: 0.7000 - val_loss: 0.8497
Epoch 21/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.7782 - loss: 0.6300 - val_accuracy: 0.6740 - val_loss: 0.8670
Epoch 22/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.7642 - loss: 0.6401 - val_accuracy: 0.6780 - val_loss: 0.9075
Epoch 23/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.7859 - loss: 0.5838 - val_accuracy: 0.6680 - val_loss: 0.9318
Epoch 24/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.7974 - loss: 0.5492 - val_accuracy: 0.6820 - val_loss: 0.8890
Epoch 25/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8033 - loss: 0.5429 - val_accuracy: 0.7060 - val_loss: 0.8862
Epoch 26/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8092 - loss: 0.5081 - val_accuracy: 0.6580 - val_loss: 0.9431
Epoch 27/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8075 - loss: 0.5223 - val_accuracy: 0.7060 - val_loss: 0.9106
Epoch 28/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8386 - loss: 0.4360 - val_accuracy: 0.6840 - val_loss: 0.9628
Epoch 29/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8585 - loss: 0.3786 - val_accuracy: 0.7120 - val_loss: 0.9247
Epoch 30/30
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8547 - loss: 0.3752 - val_accuracy: 0.6900 - val_loss: 1.0489
Test Accuracy with batch=128, epochs=30: 69.00%

Final Results Summary:
Batch Size 32, Epochs 10: 63.20%
```

```
Batch Size 32, Epochs 20: 69.00%
Batch Size 32, Epochs 30: 71.00%
Batch Size 64, Epochs 10: 65.00%
Batch Size 64, Epochs 20: 71.00%
Batch Size 64, Epochs 30: 74.40%
Batch Size 128, Epochs 10: 63.40%
Batch Size 128, Epochs 20: 67.80%
Batch Size 128, Epochs 30: 69.00%
```



Task 5.5 Conclusion:

The experiment systematically evaluates the impact of batch size and the number of training epochs on the test accuracy of a CNN trained on the dataset. The results indicate that increasing the number of epochs generally improves test accuracy, with the best performance (74.40%) achieved using a batch size of 64 and 30 epochs. From the results, it is evident that a moderate batch size (64) consistently outperforms both smaller (32) and larger (128) batch sizes. A batch size of 32 shows steady improvement with more epochs but does not surpass the performance of batch size 64. On the other hand, a batch size of 128 performs relatively well but does not reach the same peak accuracy, possibly due to reduced model updates per epoch, which may limit learning efficiency. The trend suggests that while increasing epochs leads to higher accuracy, the performance gains beyond 20 epochs start to diminish. This indicates that while the model benefits from longer training, an optimal stopping point may exist around 30 epochs. Additional techniques like early stopping or learning rate scheduling could help refine this further. Overall, the experiment demonstrates that selecting an appropriate batch size and number of epochs plays a crucial role in CNN performance. A batch size of 64 appears to provide a good balance between stability and generalization, making it the most effective choice in this setup. Future optimizations, such as fine-tuning the learning rate, adding data augmentation, or experimenting with dropout rates, could further enhance the model's performance.

Task 6: Repeat the above experiment in (5) using a subset of the imagenet data set (8-12 classes, a random subset from each class of suitable size)

Task 6.1: Load and Preprocess a Subset of ImageNet

```
# Load a small subset of ImageNet (e.g., ImageNet2012_subset or use a custom one)
dataset_name = "imagenette/320px"  # A lightweight version of ImageNet with 10 classes
dataset, info = tfds.load(dataset_name, split=['train', 'validation'], as_supervised=True, with_info=True)

# Extract dataset splits (training and validation sets)
train_data, val_data = dataset  # `train_data` contains training images, `val_data` contains validation images

# Retrieve and print available class names from the dataset metadata
class_names = info.features['label'].names
print("Classes:", class_names)  # Display the 10 class names in Imagenette

# Define constants for image processing
IMG_SIZE = 32  # Resize images to 32x32 to match CIFAR-100 dataset dimensions
BATCH_SIZE = 64  # Number of images per batch during training and validation
```

```
Downloading and preparing dataset 325.48 MiB (download: 325.48 MiB, generated: 332.71 MiB, total: 658.19 MiB) to /root/tensorflow_datasets/imagenette/320px/1.0.0...
```

{"model_id":"a61d41ce183a4d539c2ec8e6cecd45e1","version_major":2,"version_minor":0}

{"model_id":"c15e7237a4094330b7281c240fbd1d05","version_major":2,"version_minor":0}

{"model_id":"d5298754fe0d46d1aa1cb222bf8e9f6f","version_major":2,"version_minor":0}

{"model_id":"3a8829cc993c48518873f4e877c72819","version_major":2,"version_minor":0}

{"model_id":"30a44862f5b043daa32cacdacbb06981","version_major":2,"version_minor":0}

{"model_id":"7c290017bf4c429ba46ef66fd57a2175","version_major":2,"version_minor":0}

```
{"model_id":"352c1a7306604250b8bd78e2d70ec142","version_major":2,"version_minor":0}

{"model_id":"c6cb97080cee4f299ec81999c9512a39","version_major":2,"version_minor":0}
```


```
Dataset imagenette downloaded and prepared to
/root/tensorflow_datasets/imagenette/320px/1.0.0. Subsequent calls
will reuse this data.
Classes: ['n01440764', 'n02102040', 'n02979186', 'n03000684',
'n03028079', 'n03394916', 'n03417042', 'n03425413', 'n03445777',
'n03888257']
```

Task 6.2: Data Preprocessing (Resizing, Normalization)

```
# Function to preprocess images before training
def preprocess(image, label):
    image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))  # Resize
image to defined IMG_SIZE (32x32)
    image = tf.cast(image, tf.float32) / 255.0  # Normalize pixel
values to range [0,1] for better training stability
    return image, label  # Return preprocessed image and corresponding
label

# Apply preprocessing to training and validation datasets
train_data = train_data.map(preprocess)  # Resize and normalize images
in the training set
train_data = train_data.batch(BATCH_SIZE).shuffle(1000)  # Batch and
shuffle training data for better generalization

val_data = val_data.map(preprocess)  # Resize and normalize images in
the validation set
val_data = val_data.batch(BATCH_SIZE)  # Batch validation data (no
shuffling needed)
```

Task 6.3: Define CNN Model

```
# Function to create a CNN model for image classification
def create_cnn():
    model = models.Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),  # Input layer
with image size and 3 color channels (RGB)

        # First convolutional block: Conv layer with 32 filters,
followed by max pooling
        layers.Conv2D(32, (3,3), activation='relu'),
        layers.MaxPooling2D((2,2)),

        # Second convolutional block: Conv layer with 64 filters,
followed by max pooling
        layers.Conv2D(64, (3,3), activation='relu'),
```

```
        layers.MaxPooling2D((2,2)),

        # Third convolutional block: Conv layer with 128 filters, 
followed by max pooling
        layers.Conv2D(128, (3,3), activation='relu'),
        layers.MaxPooling2D((2,2)),

        # Flatten the feature maps to feed into dense layers
        layers.Flatten(),

        # Fully connected dense layer with 128 neurons and ReLU 
activation
        layers.Dense(128, activation='relu'),

        # Output layer: Number of neurons equal to the number of 
classes, using softmax activation
        layers.Dense(len(class_names), activation='softmax')
    ])

    # Compile the model with Adam optimizer and sparse categorical 
crossentropy loss (since labels are integers)
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model

# Create the CNN model
cnn_model = create_cnn()

# Display the model summary to understand the architecture
cnn_model.summary()

Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d_6 (Conv2D)                    │ (None, 30, 30, 32)          │
896 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_3 (MaxPooling2D)       │ (None, 15, 15, 32)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_7 (Conv2D)                    │ (None, 13, 13, 64)          │
```

```
18,496 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_4 (MaxPooling2D)       │ (None, 6, 6, 64)            │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_8 (Conv2D)                    │ (None, 4, 4, 128)           │
73,856 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d_5 (MaxPooling2D)       │ (None, 2, 2, 128)           │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ flatten_1 (Flatten)                  │ (None, 512)                 │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_2 (Dense)                      │ (None, 128)                 │
65,664 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_3 (Dense)                      │ (None, 10)                  │
1,290 │
└──────────────────────────────────────┴─────────────────────────────┴
─────────────────┘

 Total params: 160,202 (625.79 KB)

 Trainable params: 160,202 (625.79 KB)

 Non-trainable params: 0 (0.00 B)
```

Task 6.4: Train and Evaluate the Model

```
# Set the number of training epochs
EPOCHS = 10  # Can be adjusted to experiment with different values
(e.g., 10, 20, 30)

# Train the CNN model on the training dataset
history = cnn_model.fit(train_data, epochs=EPOCHS,
validation_data=val_data)

# Evaluate the model on the validation dataset to measure performance
test_loss, test_acc = cnn_model.evaluate(val_data)

# Print the final test accuracy in percentage format
print(f"Test Accuracy on ImageNet Subset: {test_acc * 100:.2f}%")
```

```
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 87s 96ms/step - accuracy: 0.2539 - loss: 2.0820 - val_accuracy: 0.4540 - val_loss: 1.5745
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 75s 99ms/step - accuracy: 0.4609 - loss: 1.5880 - val_accuracy: 0.5760 - val_loss: 1.3621
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 71s 93ms/step - accuracy: 0.5478 - loss: 1.3345 - val_accuracy: 0.6400 - val_loss: 1.1594
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 71s 93ms/step - accuracy: 0.6158 - loss: 1.1706 - val_accuracy: 0.6500 - val_loss: 1.1058
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 73s 99ms/step - accuracy: 0.6424 - loss: 1.0889 - val_accuracy: 0.6740 - val_loss: 1.0285
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 72s 97ms/step - accuracy: 0.6783 - loss: 0.9800 - val_accuracy: 0.6600 - val_loss: 1.0058
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 71s 92ms/step - accuracy: 0.7060 - loss: 0.8999 - val_accuracy: 0.6820 - val_loss: 0.9801
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 72s 92ms/step - accuracy: 0.7103 - loss: 0.8748 - val_accuracy: 0.6920 - val_loss: 0.9852
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 72s 98ms/step - accuracy: 0.7502 - loss: 0.7764 - val_accuracy: 0.6520 - val_loss: 1.0679
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 75s 99ms/step - accuracy: 0.7604 - loss: 0.7299 - val_accuracy: 0.6840 - val_loss: 0.9749
8/8 ━━━━━━━━━━━━━━━━━━━━ 2s 283ms/step - accuracy: 0.6694 - loss: 0.9959
Test Accuracy on ImageNet Subset: 68.40%
```

Task 6.4 Conclusion:

The experiment conducted on the ImageNet subset (Imagenette) using a convolutional neural network (CNN) achieved a test accuracy of 69.00% after training for 10 epochs. The model architecture consisted of three convolutional layers with increasing filter sizes (32, 64, and 128), followed by max pooling operations, a fully connected dense layer, and a softmax output layer corresponding to the ten classes in the dataset. The dataset was preprocessed by resizing images to 32x32, normalizing pixel values, and batching training samples for optimized learning. The accuracy of 69.00% suggests that the model is capable of extracting useful features for classification, but there is room for improvement. The relatively shallow depth of the network may limit its ability to capture complex patterns in the dataset. Furthermore, training for only 10 epochs may not be sufficient for the model to reach optimal performance. Potential enhancements include increasing the number of epochs, implementing data augmentation techniques (such as rotation and flipping), adding batch normalization layers for stability, or experimenting with different learning rates. Additionally, leveraging transfer learning with a pre-trained model like MobileNetV2 or ResNet50 could significantly boost accuracy by utilizing pre-learned feature representations. The results highlight the trade-offs between training time, model complexity, and accuracy. While a simple CNN is effective for basic image classification, deeper networks and optimized training strategies are likely needed to push accuracy closer to 80–90%. Further experiments can be conducted to analyze the impact of these modifications and identify the most effective approach for improving performance on the Imagenette dataset.

Task 6.5: Do the following experiments to improve accuracy:

Task 6.5.1: Increase the size and depth of the inner layers, what is the effect on the model accuracy?

```
# Function to create a deeper CNN model with optimized pooling
def create_deep_cnn():
    model = models.Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),  # Explicit Input
layer

        # First convolutional block
        layers.Conv2D(64, (3,3), activation='relu', padding="same"),
        layers.MaxPooling2D((2,2)),

        # Second convolutional block
        layers.Conv2D(128, (3,3), activation='relu', padding="same"),
        layers.MaxPooling2D((2,2)),

        # Third convolutional block
        layers.Conv2D(256, (3,3), activation='relu', padding="same"),
        layers.MaxPooling2D((2,2)),

        # Fourth convolutional block (Modified)
        layers.Conv2D(512, (3,3), activation='relu', padding="same"),
        # Removed the last MaxPooling layer to prevent shrinking to
2x2

        # Fully connected layers
        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.Dense(128, activation='relu'),

        # Output layer with softmax activation for multi-class
classification
        layers.Dense(len(class_names), activation='softmax')
    ])

    # Compile the model using Adam optimizer and sparse categorical
cross-entropy loss
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Measure training time
```

```python
start_time = time.time()

# Create and train the deep CNN model
deep_cnn = create_deep_cnn()
history = deep_cnn.fit(train_data, epochs=EPOCHS,
validation_data=val_data)

# Calculate total training time
end_time = time.time()

# Extract final validation accuracy
final_acc = history.history['val_accuracy'][-1] * 100

# Print final accuracy and time taken
print(f"Deep CNN - Final Accuracy: {final_acc:.2f}% | Time Taken:
{end_time - start_time:.2f} sec")
```

```
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 196s 694ms/step - accuracy: 0.2234 -
loss: 2.1169 - val_accuracy: 0.4780 - val_loss: 1.5208
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 192s 691ms/step - accuracy: 0.4917 -
loss: 1.4957 - val_accuracy: 0.5840 - val_loss: 1.1961
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 192s 693ms/step - accuracy: 0.6243 -
loss: 1.1419 - val_accuracy: 0.6680 - val_loss: 1.0261
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 207s 739ms/step - accuracy: 0.6904 -
loss: 0.9448 - val_accuracy: 0.6880 - val_loss: 0.9717
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 204s 713ms/step - accuracy: 0.7414 -
loss: 0.7980 - val_accuracy: 0.7280 - val_loss: 0.9027
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 198s 694ms/step - accuracy: 0.7888 -
loss: 0.6523 - val_accuracy: 0.7120 - val_loss: 0.9094
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 194s 698ms/step - accuracy: 0.8381 -
loss: 0.4960 - val_accuracy: 0.7420 - val_loss: 0.9154
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 194s 688ms/step - accuracy: 0.8796 -
loss: 0.3817 - val_accuracy: 0.7140 - val_loss: 1.0404
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 193s 694ms/step - accuracy: 0.9195 -
loss: 0.2440 - val_accuracy: 0.7520 - val_loss: 1.0198
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 193s 694ms/step - accuracy: 0.9529 -
loss: 0.1464 - val_accuracy: 0.7420 - val_loss: 1.1504
Deep CNN - Final Accuracy: 74.20% | Time Taken: 1964.35 sec
```

Task 6.5.1: Conclusion

The deep CNN model achieved a validation accuracy of 70% in 652 seconds, showing a slight improvement over the previous architecture. While this is a decent result, further optimizations can enhance performance. Implementing data augmentation (random flips, rotations, and zooms) can improve generalization, reducing overfitting on the training set. Adding dropout layers (e.g., 50% before dense layers) can help prevent overfitting, making the model more robust. Additionally, incorporating learning rate scheduling can ensure a smoother convergence, preventing the model from plateauing too early. These modifications can collectively push accuracy further while maintaining efficient training time.

Task 6.5.2: Modify the Number and Shape of Convolutional/Pooling Layers

```
# Function to create a shallow CNN model with fewer layers
def create_shallow_cnn():
    model = models.Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),  # Explicit Input 
layer

        # First convolutional block
        layers.Conv2D(32, (3,3), activation='relu'),
        layers.MaxPooling2D((2,2)),

        # Second convolutional block
        layers.Conv2D(64, (3,3), activation='relu'),
        layers.MaxPooling2D((2,2)),

        # Fully connected layers
        layers.Flatten(),
        layers.Dense(64, activation='relu'),

        # Output layer with softmax activation for multi-class 
classification
        layers.Dense(len(class_names), activation='softmax')
    ])

    # Compile the model using Adam optimizer and sparse categorical 
cross-entropy loss
    model.compile(optimizer='adam', 
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Measure training time
start_time = time.time()

# Create and train the shallow CNN model
shallow_cnn = create_shallow_cnn()
history = shallow_cnn.fit(train_data, epochs=EPOCHS, 
validation_data=val_data)

# Calculate total training time
```

```
end_time = time.time()

# Extract final validation accuracy
final_acc = history.history['val_accuracy'][-1] * 100

# Print final accuracy and time taken
print(f"Shallow CNN - Final Accuracy: {final_acc:.2f}% | Time Taken:
{end_time - start_time:.2f} sec")

Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 68s 26ms/step - accuracy: 0.2326 - loss:
2.1266 - val_accuracy: 0.4240 - val_loss: 1.6316
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 17ms/step - accuracy: 0.4612 - loss:
1.6103 - val_accuracy: 0.5720 - val_loss: 1.3319
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 15ms/step - accuracy: 0.5448 - loss:
1.3730 - val_accuracy: 0.6140 - val_loss: 1.2143
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 62s 15ms/step - accuracy: 0.5964 - loss:
1.2415 - val_accuracy: 0.6660 - val_loss: 1.1232
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 15ms/step - accuracy: 0.6213 - loss:
1.1533 - val_accuracy: 0.6620 - val_loss: 1.0669
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 17ms/step - accuracy: 0.6468 - loss:
1.0975 - val_accuracy: 0.6680 - val_loss: 1.0694
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 62s 17ms/step - accuracy: 0.6653 - loss:
1.0321 - val_accuracy: 0.6300 - val_loss: 1.1333
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 15ms/step - accuracy: 0.6806 - loss:
0.9961 - val_accuracy: 0.6820 - val_loss: 1.0458
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 62s 15ms/step - accuracy: 0.7043 - loss:
0.9188 - val_accuracy: 0.6760 - val_loss: 1.0806
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 63s 15ms/step - accuracy: 0.7199 - loss:
0.8820 - val_accuracy: 0.6880 - val_loss: 1.0254
Shallow CNN - Final Accuracy: 68.80% | Time Taken: 631.49 sec
```

Task 6.5.2: Conclusion

The shallow CNN model achieved a validation accuracy of 68.80% in 631.49 seconds, performing slightly worse than the deeper model. The reduced depth and fewer convolutional filters limited the model's ability to extract complex features from images, which likely contributed to the lower accuracy. However, the training time was only slightly lower than that of the deep CNN, indicating that the smaller model didn't provide a significant efficiency boost. This suggests that a balance between model depth and computational efficiency is necessary.

Adding batch normalization and dropout could enhance performance while keeping the model lightweight.

Task 6.5.3: Experiment with Different Activation Functions

```
# Function to create a CNN model using the sigmoid activation function
def create_sigmoid_cnn():
    model = models.Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),  # Explicit Input
layer

        # First convolutional block with sigmoid activation
        layers.Conv2D(64, (3,3), activation='sigmoid'),
        layers.MaxPooling2D((2,2)),

        # Second convolutional block with sigmoid activation
        layers.Conv2D(128, (3,3), activation='sigmoid'),
        layers.MaxPooling2D((2,2)),

        # Fully connected layers with sigmoid activation
        layers.Flatten(),
        layers.Dense(128, activation='sigmoid'),

        # Output layer with softmax activation for classification
        layers.Dense(len(class_names), activation='softmax')
    ])

    # Compile the model using Adam optimizer and sparse categorical
cross-entropy loss
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Measure training time
start_time = time.time()

# Create and train the CNN model with sigmoid activation
sigmoid_cnn = create_sigmoid_cnn()
history = sigmoid_cnn.fit(train_data, epochs=EPOCHS,
validation_data=val_data)

# Calculate total training time
end_time = time.time()

# Extract final validation accuracy
final_acc = history.history['val_accuracy'][-1] * 100

# Print final accuracy and time taken
print(f"Sigmoid CNN - Final Accuracy: {final_acc:.2f}% | Time Taken:
{end_time - start_time:.2f} sec")
```

```
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 70s 133ms/step - accuracy: 0.0964 - loss: 2.3993 - val_accuracy: 0.1000 - val_loss: 2.3127
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 68s 129ms/step - accuracy: 0.1005 - loss: 2.3015 - val_accuracy: 0.2900 - val_loss: 1.9830
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 129ms/step - accuracy: 0.2701 - loss: 2.0350 - val_accuracy: 0.3280 - val_loss: 1.8525
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 126ms/step - accuracy: 0.3261 - loss: 1.9136 - val_accuracy: 0.4040 - val_loss: 1.6998
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 127ms/step - accuracy: 0.3686 - loss: 1.8092 - val_accuracy: 0.4260 - val_loss: 1.6380
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 68s 126ms/step - accuracy: 0.4031 - loss: 1.7252 - val_accuracy: 0.4460 - val_loss: 1.6059
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 126ms/step - accuracy: 0.4250 - loss: 1.6815 - val_accuracy: 0.4640 - val_loss: 1.5356
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 126ms/step - accuracy: 0.4492 - loss: 1.6058 - val_accuracy: 0.5040 - val_loss: 1.4860
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 68s 126ms/step - accuracy: 0.4706 - loss: 1.5599 - val_accuracy: 0.5300 - val_loss: 1.4380
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 67s 127ms/step - accuracy: 0.5008 - loss: 1.4808 - val_accuracy: 0.5280 - val_loss: 1.4054
Sigmoid CNN - Final Accuracy: 52.80% | Time Taken: 676.87 sec
```

Task 6.5.3: Conclusion

The CNN model using the sigmoid activation function resulted in a validation accuracy of 52.80%, which is significantly lower than the ReLU-based models. Additionally, training time increased slightly to 676.87 seconds. The poor performance is likely due to sigmoid's vanishing gradient problem, which weakens gradient updates in deeper layers, making learning inefficient. This is especially problematic for convolutional networks, where ReLU (or its variants) generally performs better by avoiding saturation and enabling faster convergence. The results reinforce that ReLU remains the superior activation function for CNNs in image classification.

Task 6.5.4: Experiment with Different Optimizers

```python
# List of different optimizers to experiment with
optimizers = ['sgd', 'rmsprop', 'adam']

# Loop through each optimizer and train the model
for opt in optimizers:
    print(f"\nTraining with {opt.upper()} Optimizer...")  # Display
```

```
the optimizer being used

    # Create a deep CNN model
    model = create_deep_cnn()

    # Compile the model with the selected optimizer, using sparse
categorical cross-entropy as the loss function
    model.compile(optimizer=opt,
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Measure training time
    start_time = time.time()

    # Train the model on the training dataset and validate on the
validation dataset
    history = model.fit(train_data, epochs=EPOCHS,
validation_data=val_data)

    # Calculate total training time
    end_time = time.time()

    # Extract the final validation accuracy after training
    final_acc = history.history['val_accuracy'][-1] * 100

    # Print the results including optimizer used, final accuracy, and
time taken
    print(f"Optimizer: {opt.upper()} - Final Accuracy: {final_acc:.2f}
% | Time Taken: {end_time - start_time:.2f} sec")
```

```
Training with SGD Optimizer...
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 150s 534ms/step - accuracy: 0.1332 -
loss: 2.2972 - val_accuracy: 0.2260 - val_loss: 2.2714
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 522ms/step - accuracy: 0.2252 -
loss: 2.2588 - val_accuracy: 0.2080 - val_loss: 2.1499
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 527ms/step - accuracy: 0.2619 -
loss: 2.1275 - val_accuracy: 0.3340 - val_loss: 1.9305
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 524ms/step - accuracy: 0.3065 -
loss: 2.0127 - val_accuracy: 0.3560 - val_loss: 1.8420
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 525ms/step - accuracy: 0.3403 -
loss: 1.9140 - val_accuracy: 0.3920 - val_loss: 1.7420
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 524ms/step - accuracy: 0.3744 -
loss: 1.8389 - val_accuracy: 0.3940 - val_loss: 1.7084
Epoch 7/10
```

```
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 529ms/step - accuracy: 0.3899 -
loss: 1.7614 - val_accuracy: 0.4480 - val_loss: 1.6249
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 523ms/step - accuracy: 0.4223 -
loss: 1.7006 - val_accuracy: 0.4780 - val_loss: 1.5518
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 523ms/step - accuracy: 0.4497 -
loss: 1.6412 - val_accuracy: 0.5080 - val_loss: 1.5049
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 525ms/step - accuracy: 0.4634 -
loss: 1.5861 - val_accuracy: 0.4940 - val_loss: 1.4804
Optimizer: SGD - Final Accuracy: 49.40% | Time Taken: 1476.86 sec

Training with RMSPROP Optimizer...
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 150s 525ms/step - accuracy: 0.1662 -
loss: 2.2507 - val_accuracy: 0.3880 - val_loss: 1.8325
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 524ms/step - accuracy: 0.3917 -
loss: 1.7811 - val_accuracy: 0.3760 - val_loss: 2.0649
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 149s 529ms/step - accuracy: 0.5235 -
loss: 1.4509 - val_accuracy: 0.5900 - val_loss: 1.1809
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 147s 524ms/step - accuracy: 0.6219 -
loss: 1.1503 - val_accuracy: 0.5820 - val_loss: 1.2563
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 524ms/step - accuracy: 0.6753 -
loss: 1.0001 - val_accuracy: 0.5620 - val_loss: 1.3642
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 526ms/step - accuracy: 0.7236 -
loss: 0.8568 - val_accuracy: 0.6960 - val_loss: 0.9137
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 163s 598ms/step - accuracy: 0.7796 -
loss: 0.6781 - val_accuracy: 0.7060 - val_loss: 1.0126
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 155s 539ms/step - accuracy: 0.8362 -
loss: 0.5167 - val_accuracy: 0.7040 - val_loss: 0.9943
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 149s 531ms/step - accuracy: 0.8704 -
loss: 0.3981 - val_accuracy: 0.7420 - val_loss: 1.0319
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 528ms/step - accuracy: 0.9076 -
loss: 0.2849 - val_accuracy: 0.7540 - val_loss: 1.0585
Optimizer: RMSPROP - Final Accuracy: 75.40% | Time Taken: 1504.65 sec

Training with ADAM Optimizer...
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 150s 527ms/step - accuracy: 0.2096 -
```

```
loss: 2.1363 - val_accuracy: 0.4200 - val_loss: 1.6242
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 528ms/step - accuracy: 0.4548 -
loss: 1.5855 - val_accuracy: 0.5820 - val_loss: 1.2797
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 149s 531ms/step - accuracy: 0.5930 -
loss: 1.2211 - val_accuracy: 0.6760 - val_loss: 1.0117
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 149s 531ms/step - accuracy: 0.6698 -
loss: 0.9962 - val_accuracy: 0.6660 - val_loss: 0.9827
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 529ms/step - accuracy: 0.7274 -
loss: 0.8331 - val_accuracy: 0.6840 - val_loss: 0.9721
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 149s 531ms/step - accuracy: 0.7720 -
loss: 0.7038 - val_accuracy: 0.7000 - val_loss: 0.8847
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 526ms/step - accuracy: 0.8242 -
loss: 0.5480 - val_accuracy: 0.6920 - val_loss: 1.0214
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 150s 529ms/step - accuracy: 0.8646 -
loss: 0.4225 - val_accuracy: 0.7080 - val_loss: 1.0067
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 530ms/step - accuracy: 0.9112 -
loss: 0.2762 - val_accuracy: 0.7200 - val_loss: 1.0910
Epoch 10/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 148s 528ms/step - accuracy: 0.9328 -
loss: 0.2059 - val_accuracy: 0.7220 - val_loss: 1.1526
Optimizer: ADAM - Final Accuracy: 72.20% | Time Taken: 1487.38 sec
```

Task 6.5.4: Conclusion

Experimenting with different optimizers revealed that RMSprop achieved the highest validation accuracy at 75.40%, followed by Adam at 72.20%, while SGD performed the worst at 49.40%. The poor performance of SGD (Stochastic Gradient Descent) is expected, as it often struggles with complex architectures and requires careful tuning of the learning rate. RMSprop, on the other hand, adapts the learning rate dynamically, making it more effective for deep networks, particularly in handling non-stationary objectives. Adam, which combines RMSprop and momentum-based updates, performed well but slightly underperformed compared to RMSprop. This suggests that for this specific CNN architecture and dataset, RMSprop is the optimal choice for maximizing accuracy.

Task 6.5.5: Experiment with Batch Sizes and Epochs

```python
# List of different batch sizes to experiment with
batch_sizes = [32, 64, 128]

# Loop through each batch size and train the model
for batch in batch_sizes:
    print(f"\nTraining with Batch Size = {batch}")  # Display the
```

```
current batch size

    # Adjust the training and validation datasets to use the selected
batch size
    train_data = train_data.unbatch().batch(batch)
    val_data = val_data.unbatch().batch(batch)

    # Create a deep CNN model
    model = create_deep_cnn()

    # Measure training time
    start_time = time.time()

    # Train the model with the current batch size
    history = model.fit(train_data, epochs=EPOCHS,
validation_data=val_data)

    # Calculate total training time
    end_time = time.time()

    # Extract the final validation accuracy after training
    final_acc = history.history['val_accuracy'][-1] * 100

    # Print the results including batch size used, final accuracy, and
time taken
    print(f"Batch Size: {batch} - Final Accuracy: {final_acc:.2f}% |
Time Taken: {end_time - start_time:.2f} sec")
```

```
Training with Batch Size = 32
Epoch 1/10
    403/Unknown 199s 356ms/step - accuracy: 0.2018 - loss: 2.1378

/usr/local/lib/python3.11/dist-packages/keras/src/trainers/
epoch_iterator.py:151: UserWarning: Your input ran out of data;
interrupting training. Make sure that your dataset or generator can
generate at least `steps_per_epoch * epochs` batches. You may need to
use the `.repeat()` function when building your dataset.
  self._interrupted_warning()

 403/403 ━━━━━━━━━━━━━━━━━━━━ 203s 364ms/step - accuracy: 0.2020 -
loss: 2.1373 - val_accuracy: 0.4660 - val_loss: 1.6040
Epoch 2/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 201s 366ms/step - accuracy: 0.4876 -
loss: 1.5216 - val_accuracy: 0.6080 - val_loss: 1.1934
Epoch 3/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 200s 365ms/step - accuracy: 0.6131 -
loss: 1.1708 - val_accuracy: 0.6520 - val_loss: 1.1159
Epoch 4/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 201s 367ms/step - accuracy: 0.6952 -
loss: 0.9486 - val_accuracy: 0.6760 - val_loss: 0.9815
```

```
Epoch 5/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 202s 367ms/step - accuracy: 0.7372 -
loss: 0.8047 - val_accuracy: 0.6860 - val_loss: 1.0118
Epoch 6/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 204s 372ms/step - accuracy: 0.7899 -
loss: 0.6306 - val_accuracy: 0.7160 - val_loss: 0.9795
Epoch 7/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 207s 374ms/step - accuracy: 0.8420 -
loss: 0.4863 - val_accuracy: 0.6960 - val_loss: 1.0731
Epoch 8/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 204s 367ms/step - accuracy: 0.8786 -
loss: 0.3686 - val_accuracy: 0.7000 - val_loss: 1.1474
Epoch 9/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 199s 364ms/step - accuracy: 0.9302 -
loss: 0.2115 - val_accuracy: 0.6840 - val_loss: 1.3654
Epoch 10/10
403/403 ━━━━━━━━━━━━━━━━━━━━ 199s 363ms/step - accuracy: 0.9489 -
loss: 0.1579 - val_accuracy: 0.7060 - val_loss: 1.3499
Batch Size: 32 - Final Accuracy: 70.60% | Time Taken: 2019.17 sec

Training with Batch Size = 64
Epoch 1/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 195s 692ms/step - accuracy: 0.2309 -
loss: 2.0874 - val_accuracy: 0.5040 - val_loss: 1.4388
Epoch 2/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 194s 691ms/step - accuracy: 0.5203 -
loss: 1.4232 - val_accuracy: 0.6260 - val_loss: 1.1305
Epoch 3/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 194s 695ms/step - accuracy: 0.6394 -
loss: 1.1004 - val_accuracy: 0.6700 - val_loss: 0.9791
Epoch 4/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 201s 714ms/step - accuracy: 0.7004 -
loss: 0.9187 - val_accuracy: 0.6940 - val_loss: 0.8847
Epoch 5/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 202s 716ms/step - accuracy: 0.7465 -
loss: 0.7738 - val_accuracy: 0.7300 - val_loss: 0.8274
Epoch 6/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 196s 694ms/step - accuracy: 0.7895 -
loss: 0.6335 - val_accuracy: 0.7540 - val_loss: 0.8136
Epoch 7/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 197s 709ms/step - accuracy: 0.8490 -
loss: 0.4641 - val_accuracy: 0.7600 - val_loss: 0.8732
Epoch 8/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 202s 714ms/step - accuracy: 0.8872 -
loss: 0.3406 - val_accuracy: 0.7580 - val_loss: 0.9556
Epoch 9/10
202/202 ━━━━━━━━━━━━━━━━━━━━ 198s 698ms/step - accuracy: 0.9119 -
loss: 0.2607 - val_accuracy: 0.7480 - val_loss: 0.9303
Epoch 10/10
```

```
202/202 ━━━━━━━━━━━━━━━━━━━━ 200s 713ms/step - accuracy: 0.9562 -
loss: 0.1364 - val_accuracy: 0.7360 - val_loss: 1.2738
Batch Size: 64 - Final Accuracy: 73.60% | Time Taken: 1979.11 sec

Training with Batch Size = 128
Epoch 1/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 201s 1s/step - accuracy: 0.1792 - loss:
2.2114 - val_accuracy: 0.4100 - val_loss: 1.6400
Epoch 2/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 194s 1s/step - accuracy: 0.4504 - loss:
1.5981 - val_accuracy: 0.5560 - val_loss: 1.3010
Epoch 3/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 193s 1s/step - accuracy: 0.5560 - loss:
1.3294 - val_accuracy: 0.6100 - val_loss: 1.1802
Epoch 4/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 194s 1s/step - accuracy: 0.6360 - loss:
1.1027 - val_accuracy: 0.6700 - val_loss: 1.0439
Epoch 5/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 193s 1s/step - accuracy: 0.6819 - loss:
0.9578 - val_accuracy: 0.6880 - val_loss: 0.9979
Epoch 6/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 193s 1s/step - accuracy: 0.7315 - loss:
0.8168 - val_accuracy: 0.7240 - val_loss: 0.9205
Epoch 7/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 198s 1s/step - accuracy: 0.7705 - loss:
0.6832 - val_accuracy: 0.6860 - val_loss: 0.9418
Epoch 8/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 194s 1s/step - accuracy: 0.8025 - loss:
0.5962 - val_accuracy: 0.7300 - val_loss: 0.8759
Epoch 9/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 193s 1s/step - accuracy: 0.8443 - loss:
0.4734 - val_accuracy: 0.7240 - val_loss: 0.9707
Epoch 10/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 193s 1s/step - accuracy: 0.8818 - loss:
0.3521 - val_accuracy: 0.7160 - val_loss: 1.0672
Batch Size: 128 - Final Accuracy: 71.60% | Time Taken: 1946.27 sec
```

Task 6.5.5: Conclusion

Experimenting with different batch sizes showed that 64 achieved the highest validation accuracy at 73.60%, while 32 and 128 resulted in 70.60% and 71.60%, respectively. A batch size of 64 likely provided a balance between model generalization and computational efficiency. The smallest batch (32) resulted in slightly lower accuracy and the longest training time (2019 sec) due to more frequent updates per epoch. Larger batch sizes (128) trained faster but had slightly lower accuracy, possibly due to less frequent weight updates leading to suboptimal generalization. Overall, batch size 64 appears to be the best choice, offering improved accuracy while reducing training time compared to batch size 32.