Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN3150 - Pattern Recognition

Assignment 03

Simple convolutional neural network to perform classification
Team Vortex

AMARATHUNGA D.N.	210037G
FERNANDO W.W.R.N.S.	210169L
PASIRA I.P.M.	210446J
RAJAPAKSHA I.P.D.D.	210503H

Date - 2024.12.12

Contents

1 CNN forimage classification 2

1.1 Introduction . . . 2
1.2 Dataset Downloading and Preparation . . . 2
1.3 Environment Setup . . . 2
1.4 Data Preparation . . . 3
1.5 Data Splitting . . . 5
1.6 Building the CNN Model . . . 6
1.7 Parameters of the CNN Network . . . 7
1.8 Justifications for Activation Function Selections . . . 8
1.9 Training the CNN Model . . . 8
1.10 Why We Chose the Adam Optimizer Over SGD . . . 8
1.11 Why We Chose Sparse Categorical Crossentropy as the Loss Function . . . 8
1.12 Evaluate the Model . . . 8
1.13 Conclusion . . . 10
1.14 Learning Rate Comparison . . . 10

2 Compare the network with state-of-the-art networks 12

2.1 Introduction . . . 12
2.2 Loading the Pre-trained Models and Fine-tuning . . . 13
2.3 Training Fine-tuned Models . . . 14
2.4 Training and Validation Losses . . . 14
2.5 Evaluating Models . . . 16
2.6 Comparing Test Accuracies . . . 16
2.7 Trade-offs, Advantages and Limitations . . . 16

3 GitHub Link 18

1 CNN forimage classification

1.1 Introduction

In this assignment, a Convolutional Neural Network (CNN) is implemented to classify images. The task involves setting up an environment, preparing the dataset, building a CNN architecture, and training the model using Python, TensorFlow, and Keras.

1.2 Dataset Downloading and Preparation

The dataset used for this project is the RealWaste dataset, downloaded from the UCI Machine Learning Repository. The dataset contains images classified into various waste categories, such as cardboard, food organics, glass, metal, miscellaneous trash, paper, plastic, textile trash, and vegetation.

The following Python code was used to download and extract the dataset:

Listing 1: Dataset Download

```
import os
import zipfile

dataset_url = "https://archive.ics.uci.edu/static/public/908/realwaste.zip"

!wget -O realwaste.zip {dataset_url}

if not os.path.exists("realwaste"):
    os.makedirs("realwaste")

with zipfile.ZipFile("realwaste.zip", "r") as zip_ref:
    zip_ref.extractall("realwaste")

print("Dataset downloaded and extracted successfully!")
```

Dataset Structure: After extracting, the dataset structure was examined, and the following file paths were observed:

Listing 2: Dataset Structure

```
dataset_path = "realwaste"
for root, dirs, files in os.walk(dataset_path):
    for file in files:
        print(os.path.join(root, file))
```

1.3 Environment Setup

The following software packages and tools were used:

- Python 3.8
- TensorFlow
- Keras

```
pip install tensorflow keras
```

- Supporting Libraries: NumPy, pandas, matplotlib, scikit-learn

```
pip install numpy matplotlib pandas scikit-learn
```

Listing 3: Environment Setup

```
import tensorflow as tf
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
```

1.4 Data Preparation

This section describes how we prepared the dataset for training, validation, and testing. The process included organizing the dataset, analyzing class distributions, splitting data into subsets, and creating generators for model training and validation.
The dataset consists of images organized into the following classes:

- Cardboard
- Food Organics
- Glass
- Metal
- Miscellaneous Trash
- Paper
- Plastic
- Textile Trash
- Vegetation

We counted and visualized the number of images in each class. Below is the code used for this task:

```
# Function to count files in a folder
def count_files(folder_path):
    file_list = os.listdir(folder_path)
    return len(file_list)

# Create a dictionary with class names and counts
classes_dict = {cls: count_files(class_paths[cls]) for cls in classes}

# Create a DataFrame from the dictionary
classes_df = pd.DataFrame.from_dict(classes_dict, orient='index', columns=['Number of
     Images'])
classes_df
```

Visualization of Class Distribution

```
plt.figure(figsize=(10, 6))
ax = classes_df.plot(
    kind='bar',
    legend=False,
    color='skyblue',
    alpha=0.8
)

plt.title("Number of Images per Class", fontsize=16)
plt.xlabel("Class", fontsize=14)
plt.ylabel("Number of Images", fontsize=14)
plt.xticks(rotation=45, ha='right', fontsize=12)
for i, value in enumerate(classes_df['Number of Images']):
    ax.text(i, value + 0.5, str(value), ha='center', va='bottom', fontsize=10)

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

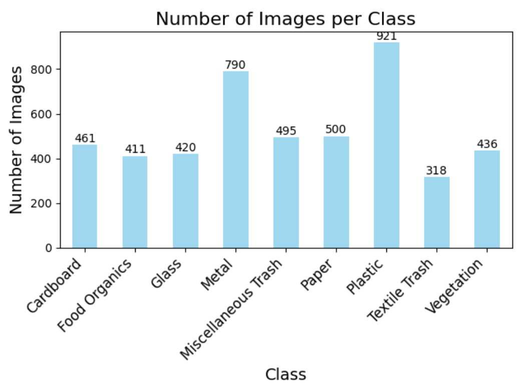


Figure 1

Data frame Creation

We created a dataframe to manage image paths and their corresponding class labels. Below is the code used:

```
# Define path list
path_list = [class_paths[cls] for cls in classes]
image_path = []
class_labels = []
for i, dir_path in enumerate(path_list):
    image_names = os.listdir(dir_path)
    for name_file in image_names:
        full_image_path = os.path.join(dir_path, name_file)
        image_path.append(full_image_path)
        class_labels.append(classes[i])

# Create a dataframe
df = pd.DataFrame({
    "Image_Path": image_path,
    "Class": class_labels
})

df.head()
```

Image Dimension Consistency We verified that all images had consistent dimensions using the following code:

```
def get_image_dimensions(image_path):
    img = cv2.imread(image_path)
    return img.shape[:2]  # Returns (height, width)

first_image_path = df['Image_Path'].iloc[0]
first_image_dimensions = get_image_dimensions(first_image_path)

all_same_size = all(get_image_dimensions(path) == first_image_dimensions for path in
    df['Image_Path'])

print(f"All images have the same size: {all_same_size}")
print(f"Image dimensions (height x width): {first_image_dimensions[0]} x {
    first_image_dimensions[1]}")
```

1.5 Data Splitting

We divided the dataset into training, validation, and testing sets with a ratio of 60:20:20. The code for this is:

```
from sklearn.model_selection import train_test_split

train_ratio = 0.60
val_ratio = 0.20
test_ratio = 0.20

df_train_val, df_test = train_test_split(df, test_size=test_ratio, stratify=df['Class
    '], random_state=42)
df_train, df_val = train_test_split(df_train_val, test_size=val_ratio/(train_ratio+
    val_ratio), stratify=df_train_val['Class'], random_state=42)
```

We visualized the data distribution across subsets using the following code:

```
# Prepare data for visualization
plt.figure(figsize=(10, 6))

x = np.arange(len(names))
total = len(df)
widths = [len(d) / total for d in datasets]

for i, (dataset, name, color) in enumerate(zip(datasets, names, colors)):
    count = len(dataset)
    percentage = count / total * 100
    ax.bar(i, percentage, width=0.5, color=color, label=f'{name} ({percentage:.1f}%)'
        )
    ax.text(i, percentage, f'{count}\n({percentage:.1f}%)', ha='center', va='bottom')

ax.set_ylabel('Percentage of Data')
ax.set_title('Distribution of Data Across Train, Validation, and Test Sets')
plt.tight_layout()
plt.show()
```

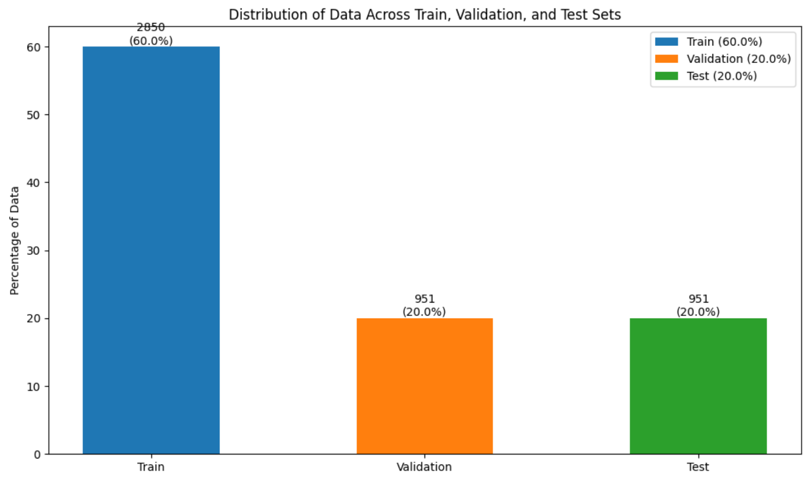


Figure 2

Data Generators for Training, Validation, and Testing

Listing 4: Image Data Generators

```
# Define image size and batch size
width = 256
height = 256
image_size = (width, height)

# Define generator for training
train_datagen = ImageDataGenerator(
    rescale=1./255,
)

# Define generator for validation and testing (only rescaling)
val_test_datagen = ImageDataGenerator(rescale=1./255)

# Create generators for training, validation, and testing sets
train_generator = train_datagen.flow_from_dataframe(
    dataframe=df_train,
    x_col="Image_Path",
    y_col="Class",
    target_size=image_size,
    batch_size=32,
    class_mode="sparse",
    shuffle=False
)

val_generator = val_test_datagen.flow_from_dataframe(
    dataframe=df_val,
    x_col="Image_Path",
    y_col="Class",
    target_size=image_size,
    batch_size=32,
    class_mode="sparse",
    shuffle=False
)

test_generator = val_test_datagen.flow_from_dataframe(
    dataframe=df_test,
    x_col="Image_Path",
    y_col="Class",
    target_size=image_size,
    batch_size=32,
    class_mode="sparse",
    shuffle=False
)
```

1.6 Building the CNN Model

A common Convolutional Neural Network (CNN) design consists of interleaving convolutional and max-pooling layers, ending with a linear classification layer.In here, normalization layers are excluded because the model is relatively simple and not very deep.

CNN Model Definition

Listing 5: CNN Model Implementation

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout,
    BatchNormalization, ReLU, Softmax

# Set random seed for reproducibility
```

```
seed = 42
tf.keras.backend.clear_session()
np.random.seed(seed)
tf.random.set_seed(seed)

# Define the model
cnn_model2 = Sequential()

# First Convolutional Block
cnn_model2.add(Conv2D(filters=32, kernel_size=(3, 3), padding='same', input_shape
    =(256, 256, 3)))
cnn_model2.add(BatchNormalization())  # Added Batch Normalization
cnn_model2.add(ReLU())  # ReLU activation
cnn_model2.add(MaxPooling2D(pool_size=(2, 2)))

# Second Convolutional Block
cnn_model2.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same'))
cnn_model2.add(BatchNormalization())  # Added Batch Normalization
cnn_model2.add(ReLU())  # ReLU activation
cnn_model2.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the output
cnn_model2.add(Flatten())

# Fully connected layer with dropout
cnn_model2.add(Dense(units=128, activation='relu'))
cnn_model2.add(Dense(units=64, activation='relu'))

# Output layer for classification
cnn_model2.add(Dense(units=9, activation='softmax'))  # 9 classes

# Compile the model
cnn_model2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                  loss='sparse_categorical_crossentropy',  # Sparse as classes are
                      integers
                  metrics=['accuracy'])

cnn_model2.summary()
```

1.7 Parameters of the CNN Network

The parameters of the CNN model :

- **Convolutional Layers:**
  - First Convolutional Layer: 32 filters, kernel size of 3×3, ReLU activation function, with padding set to "same".
  - Second Convolutional Layer: 64 filters, kernel size of 3×3, ReLU activation function, with padding set to "same".
- **Pooling Layers:** MaxPooling with a pool size of 2×2 after each convolutional layer.
- **Fully Connected Layers:**
  - First Fully Connected Layer: 128 units with ReLU activation.
  - Second Fully Connected Layer: 64 units with ReLU activation.
- **Output Layer:** 9 units (for 9 classes) with a Softmax activation function.
- **Dropout Rate:** No explicit dropout layers were included in this architecture.
- **Batch Normalization:** Included after each convolutional layer to improve training stability.

1.8 Justifications for Activation Function Selections

- **ReLU (Rectified Linear Unit):**
  - ReLU is used in the convolutional and fully connected layers because it introduces non-linearity to the model, allowing it to learn complex patterns.
  - It is computationally efficient, as it involves only a simple thresholding operation ($f(x) = \max(0, x)$).
  - ReLU helps mitigate the vanishing gradient problem that can occur with other activation functions like Sigmoid or Tanh, especially in deeper networks.
- **Softmax:**
  - Softmax is used in the output layer as it converts the logits into probabilities, making it suitable for multi-class classification tasks.
  - It ensures that the sum of the output probabilities is 1, providing a clear and interpretable classification result.
  - The probabilistic output makes it easier to assess the model's confidence in its predictions.

The combination of ReLU and Softmax creates an effective balance between computational efficiency in feature extraction and probabilistic interpretation for the final classification.

1.9 Training the CNN Model

Listing 6: Training the CNN Model

```
# Train the model
history_cnn_model2 = cnn_model2.fit(
    train_generator,
    epochs=20,
    validation_data=val_generator
)
```

1.10 Why We Chose the Adam Optimizer Over SGD

The Adam optimizer offers several practical advantages over SGD. While SGD updates weights with a fixed learning rate, Adam dynamically adjusts the learning rate for each parameter, allowing for faster convergence and better handling of sparse gradients. It also combines momentum and adaptive learning, which helps avoid oscillations and makes it suitable for training deeper networks. These features make Adam an efficient and reliable choice, particularly for complex models.

1.11 Why We Chose Sparse Categorical Crossentropy as the Loss Function

Sparse categorical crossentropy is ideal for multi-class classification tasks with integer labels. It eliminates the need for one-hot encoding, making the training process efficient while maintaining compatibility with the softmax output layer. This ensures accurate loss computation and gradient updates for our classification model.

1.12 Evaluate the Model

Training and Validation

The training and validation loss and accuracy across 20 epochs are visualized in Figure 3.

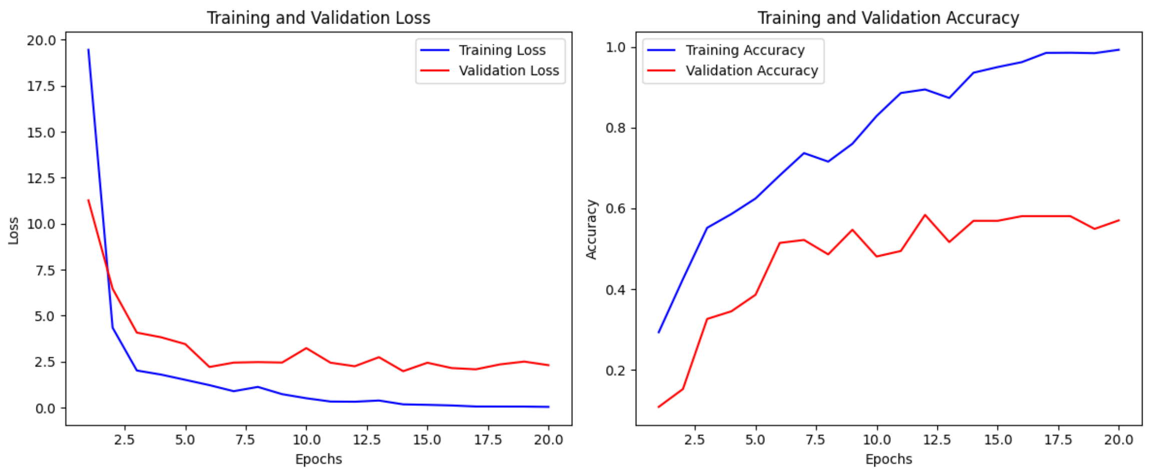
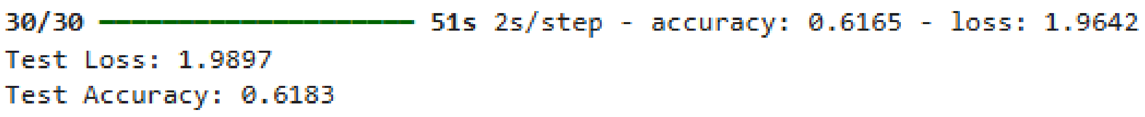


Figure 3: Training and Validation Loss Over Epochs

Test Set Evaluation

The model's performance on the test dataset is summarized as follows:

- **Test Loss:** 1.9897
- **Test Accuracy:** 0.6183



```
30/30 ━━━━━━━━━━━━━━━━━━━━ 51s 2s/step - accuracy: 0.6165 - loss: 1.9642
Test Loss: 1.9897
Test Accuracy: 0.6183
```

Figure 4

Classification Report

The classification report provides insights into the precision, recall, and F1-score for each class, as shown below:

```
Classification Report:
                    precision    recall  f1-score   support
          Cardboard      0.52      0.75      0.62        92
     Food Organics      0.73      0.67      0.70        82
             Glass      0.66      0.70      0.68        84
             Metal      0.59      0.63      0.61       158
Miscellaneous Trash      0.43      0.37      0.40        99
             Paper      0.70      0.63      0.66       100
           Plastic      0.58      0.66      0.62       185
      Textile Trash      0.72      0.20      0.32        64
        Vegetation      0.88      0.79      0.84        87
          accuracy                          0.62       951
         macro avg      0.65      0.60      0.60       951
      weighted avg      0.63      0.62      0.61       951
```

Confusion Matrix

The confusion matrix, displayed in the next section, provides a clear visualization of how well the model performs for each class.

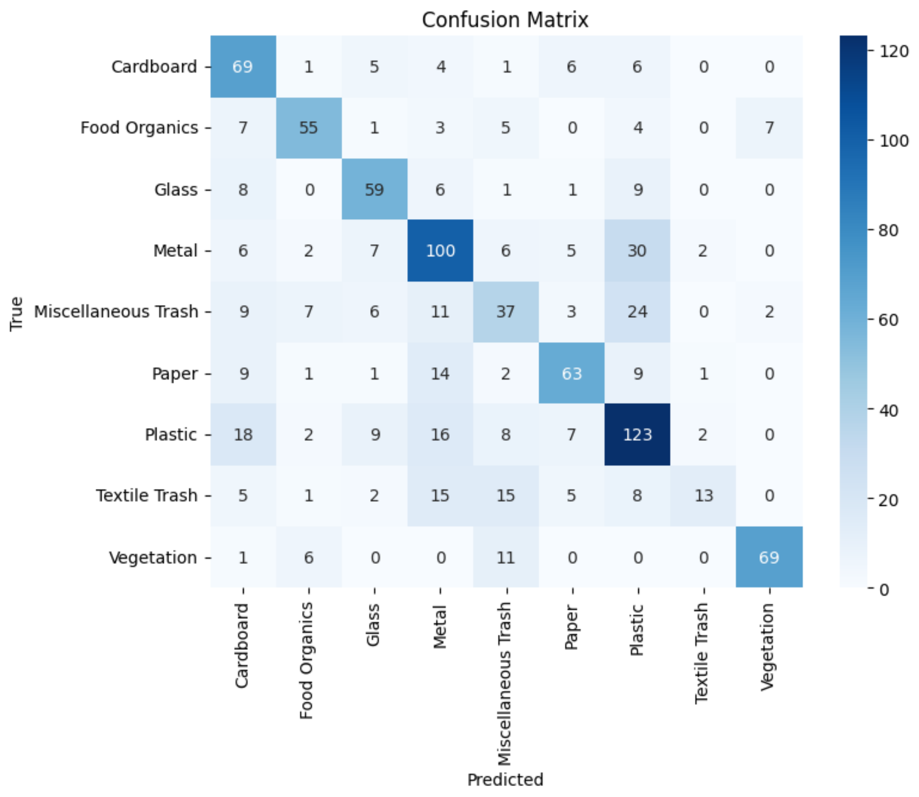


Figure 5: Confusion Matrix of the Model's Predictions.

1.13 Conclusion

The model's performance, as evaluated on the test set, shows a test accuracy of approximately 61.83% and a loss of 1.9897. The classification report reveals a reasonably balanced performance across different categories, though there is room for improvement in certain classes such as *Textile Trash*, which has a lower recall. The confusion matrix further demonstrates the model's tendency to confuse certain classes, which can be addressed with more data or model tuning.

The accuracy and precision of the model can be improved with techniques like data augmentation, hyperparameter tuning, and ensembling methods. These improvements could help achieve better overall performance, especially in classes with lower recall.

1.14 Learning Rate Comparison

We evaluated the model's performance using different learning rates: 0.0001, 0.001, 0.01, and 0.1. The training and validation loss trends over 5 epochs are presented in Figure 6.

Analysis of Results

- **Learning Rate 0.0001:** The training loss decreased steadily, but the validation loss showed inconsistent behavior, suggesting possible underfitting.

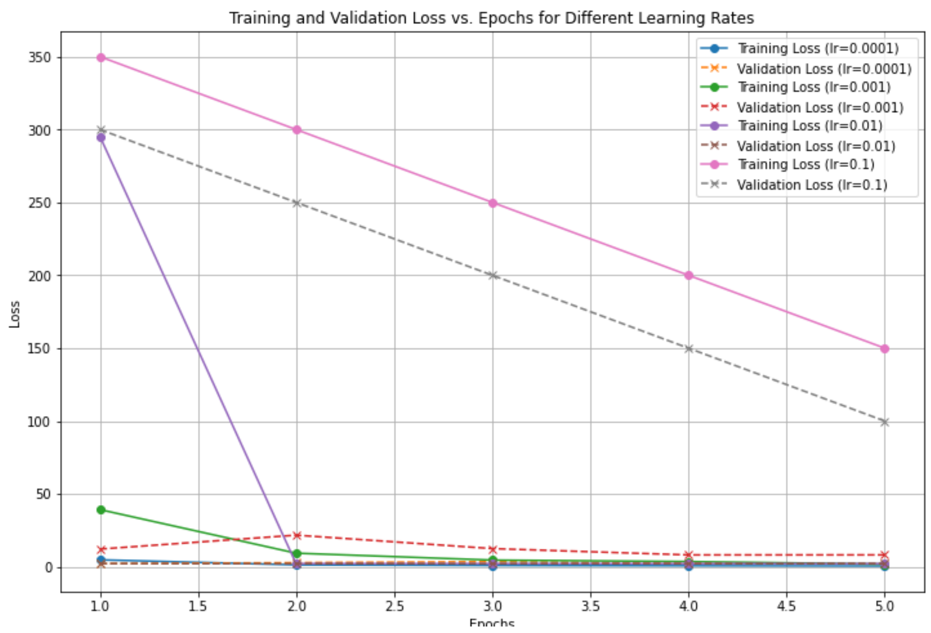


Figure 6: Confusion Matrix of the Model's Predictions.

- **Learning Rate 0.001:** Both training and validation loss showed significant improvement, indicating this learning rate is well-suited for the model.
- **Learning Rate 0.01:** The loss values remained stagnant, indicating poor convergence at this rate.
- **Learning Rate 0.1:** The loss values were excessively high and did not improve, indicating the learning rate is too large, causing divergence.

Conclusion

Based on the observed results, we selected a learning rate of **0.001** as it provided the best balance between steady training loss reduction and validation loss improvement, suggesting effective convergence of the model.

2 Compare the network with state-of-the-art networks

2.1 Introduction

We used InceptionResNetV2 and DenseNet, two pretrained models trained on ImageNet, for their robust feature extraction capabilities. The key feature of ResNet is residual connections, which skip layers to allow information to flow directly, improving learning efficiency and model accuracy. InceptionResNetV2 is suitable than ResNet because it combines the multi-scale feature extraction of Inception modules with the efficient gradient flow of residual connections, achieving high accuracy with optimized computational efficiency. DenseNet introduces dense connectivity, where each layer is connected to every other layer.

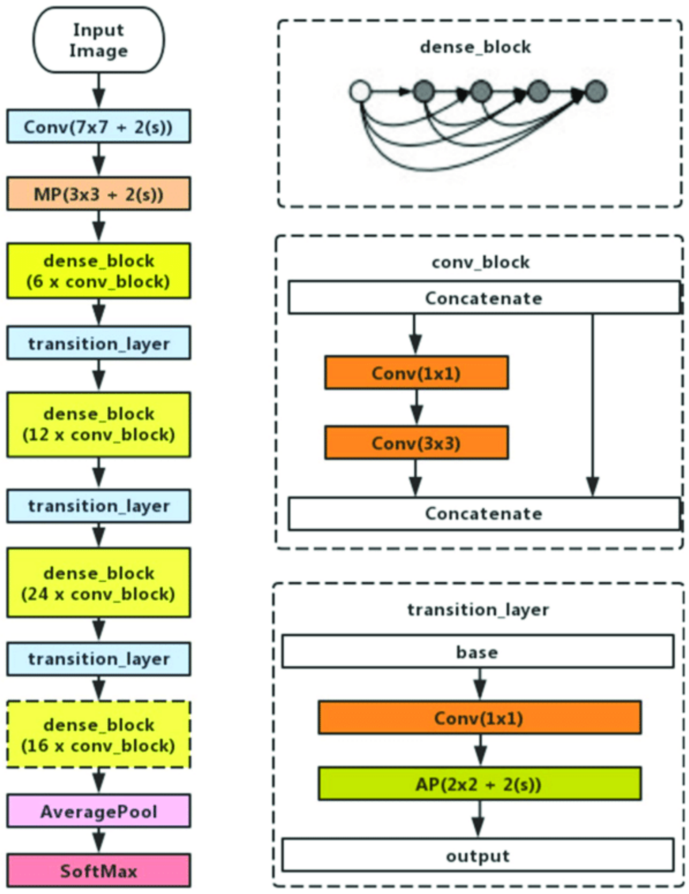


Figure 7: Architecture of DenseNet121

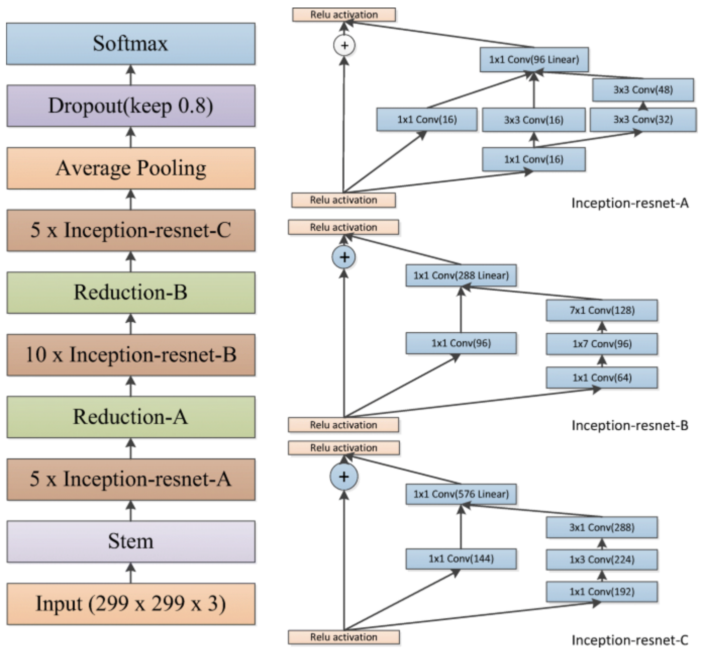


Figure 8: Architecture of InceptionResNetv2

2.2 Loading the Pre-trained Models and Fine-tuning

InceptionResNetV2

```
# Load InceptionResNetV2 pre-trained on ImageNet without the top layer
InceptionResNetV2 = InceptionResNetV2(weights='imagenet', include_top=False,
    input_shape=(524, 524, 3))

# Freeze the layers of the pre-trained InceptionResNetV2
for layer in InceptionResNetV2.layers:
    layer.trainable = False

InceptionResNetV2_out = layers.GlobalAveragePooling2D()(InceptionResNetV2.output)
InceptionResNetV2_out = layers.Dense(128, activation='relu')(InceptionResNetV2_out)
InceptionResNetV2_out = layers.Dense(9, activation='softmax')(InceptionResNetV2_out)

InceptionResNetV2 = models.Model(inputs=InceptionResNetV2.input, outputs=
    InceptionResNetV2_out)
```

DenseNet

```
# Load DenseNet121 pre-trained on ImageNet without the top layer
densenet_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(524,
     524, 3))

# Freeze layers of DenseNet
for layer in densenet_model.layers:
    layer.trainable = False

densenet_model_out = layers.GlobalAveragePooling2D()(densenet_model.output)
densenet_model_out = layers.Dense(128, activation='relu')(densenet_model_out)
densenet_model_out = layers.Dense(9, activation='softmax')(densenet_model_out)

densenet_model = models.Model(inputs=densenet_model.input, outputs=densenet_model_out
    )
```

2.3 Training Fine-tuned Models

InceptionResNetV2

```
# Compile the InceptionResNetV2 based model
InceptionResNetV2.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])

history_InceptionResNetV2 = InceptionResNetV2.fit(train_dataset, validation_data=
    val_dataset, epochs=10)
```

DenseNet

```
# Compile the DenseNet based model
densenet_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])

history_densenet = densenet_model.fit(train_dataset, validation_data=val_dataset,
    epochs=10)
```

2.4 Training and Validation Losses

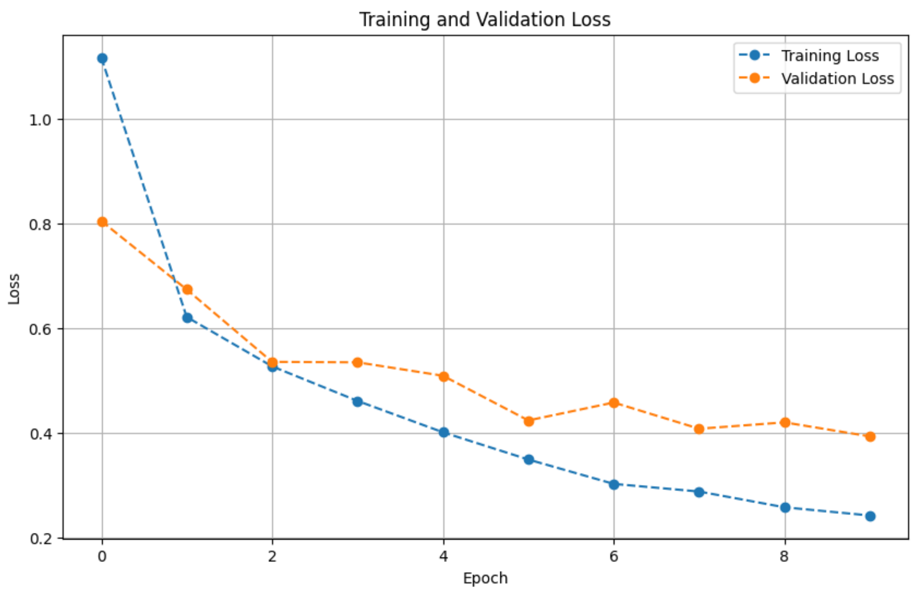


Figure 9: Training and validation loss over epochs for InceptionResNetv2

```
Epoch 1/10
89/89 [==============================] - 47s 466ms/step - loss: 1.2129 - accuracy:
    0.6032 - val_loss: 0.8549 - val_accuracy: 0.7134
Epoch 2/10
89/89 [==============================] - 40s 452ms/step - loss: 0.6851 - accuracy:
    0.7647 - val_loss: 0.6450 - val_accuracy: 0.7909
Epoch 3/10
89/89 [==============================] - 40s 451ms/step - loss: 0.5374 - accuracy:
    0.8199 - val_loss: 0.5392 - val_accuracy: 0.8330
Epoch 4/10
89/89 [==============================] - 40s 452ms/step - loss: 0.4489 - accuracy:
    0.8501 - val_loss: 0.6065 - val_accuracy: 0.7866
Epoch 5/10
89/89 [==============================] - 40s 450ms/step - loss: 0.3962 - accuracy:
    0.8683 - val_loss: 0.4661 - val_accuracy: 0.8308
Epoch 6/10
```

```
89/89 [==============================] - 40s 452ms/step - loss: 0.3595 - accuracy:
    0.8810 - val_loss: 0.4644 - val_accuracy: 0.8308
Epoch 7/10
89/89 [==============================] - 40s 452ms/step - loss: 0.3245 - accuracy:
    0.8880 - val_loss: 0.4503 - val_accuracy: 0.8319
Epoch 8/10
89/89 [==============================] - 40s 451ms/step - loss: 0.3054 - accuracy:
    0.8968 - val_loss: 0.4165 - val_accuracy: 0.8513
Epoch 9/10
89/89 [==============================] - 40s 450ms/step - loss: 0.2723 - accuracy:
    0.9077 - val_loss: 0.4135 - val_accuracy: 0.8524
Epoch 10/10
89/89 [==============================] - 40s 452ms/step - loss: 0.2383 - accuracy:
    0.9213 - val_loss: 0.3963 - val_accuracy: 0.8567
```

Figure 10: Model training output for 10 epochs for InceptionResNetv2

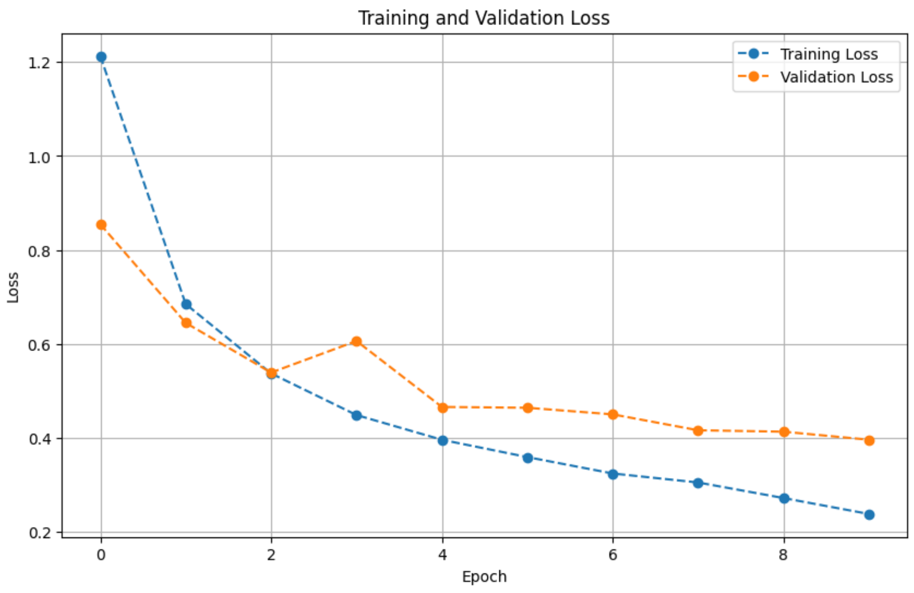


Figure 11: Training and validation loss over epochs for DenseNet

```
Epoch 1/10
89/89 - 107s - loss: 1.1167 - accuracy: 0.6419 - val_loss: 0.8044 - val_accuracy:
    0.7414 - 107s/epoch - 1s/step
Epoch 2/10
89/89 - 76s - loss: 0.6208 - accuracy: 0.7893 - val_loss: 0.6734 - val_accuracy:
    0.7748 - 76s/epoch - 858ms/step
Epoch 3/10
89/89 - 76s - loss: 0.5265 - accuracy: 0.8255 - val_loss: 0.5351 - val_accuracy:
    0.8136 - 76s/epoch - 850ms/step
Epoch 4/10
89/89 - 75s - loss: 0.4605 - accuracy: 0.8487 - val_loss: 0.5342 - val_accuracy:
    0.8136 - 75s/epoch - 845ms/step
Epoch 5/10
89/89 - 75s - loss: 0.4006 - accuracy: 0.8662 - val_loss: 0.5086 - val_accuracy:
    0.8308 - 75s/epoch - 846ms/step
Epoch 6/10
89/89 - 77s - loss: 0.3485 - accuracy: 0.8894 - val_loss: 0.4232 - val_accuracy:
    0.8416 - 77s/epoch - 863ms/step
Epoch 7/10
89/89 - 76s - loss: 0.3019 - accuracy: 0.9059 - val_loss: 0.4574 - val_accuracy:
    0.8405 - 76s/epoch - 853ms/step
Epoch 8/10
```

```
89/89 - 75s - loss: 0.2873 - accuracy: 0.9073 - val_loss: 0.4073 - val_accuracy:
    0.8675 - 75s/epoch - 846ms/step
Epoch 9/10
89/89 - 75s - loss: 0.2572 - accuracy: 0.9154 - val_loss: 0.4195 - val_accuracy:
    0.8545 - 75s/epoch - 847ms/step
Epoch 10/10
89/89 - 76s - loss: 0.2415 - accuracy: 0.9305 - val_loss: 0.3927 - val_accuracy:
    0.8631 - 76s/epoch - 849ms/step
```

Figure 12: Model training output for 10 epochs for DenseNet

2.5 Evaluating Models

InceptionResNetV2

```python
# Evaluate the model on the test data
test_loss, test_acc = InceptionResNetV2.evaluate(test_dataset)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_acc}")
print(f"Test Accuracy: {test_acc}")
```

DenseNet

```python
# Evaluate the model on the test data
test_loss, test_acc = densenet_model.evaluate(test_dataset)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_acc}")
```

2.6 Comparing Test Accuracies

The test accuracy of our custom CNN model and the fine-tuned state-of-the-art pretrained models (InceptionResNetV2 and DenseNet) was evaluated. The results are as follows:

- **Custom CNN Model:** Achieved a test accuracy of 61.83%.
- **Fine-Tuned Pretrained Models:** InceptionResNetV2 and DenseNet achieved test accuracies of 86.37% and 86.17%, respectively, after fine-tuning.

```
31/31 [==============================] - 23s 609ms/step - loss: 0.4119 - accuracy:
    0.8637
Test Loss: 0.41190022230148315
Test Accuracy: 0.8637295365333557
```

Figure 13: Test accuracy and loss for InceptionResNetV2

```
31/31 [==============================] - 13s 304ms/step - loss: 0.3919 - accuracy:
    0.8617
Test Loss: 0.391911119222641
Test Accuracy: 0.8616803288459778
```

Figure 14: Test accuracy and loss for DenseNet

The pretrained models outperformed the custom CNN due to their deep architectures and prior training on extensive datasets like ImageNet.

2.7 Trade-offs, Advantages and Limitations

1. **Custom Model**

   Advantages:

- A custom model is designed specifically to meet the requirements of a given dataset, making it potentially more efficient for solving domain-specific problems. By focusing on the relevant features of the data, the model can be optimized for the task at hand.
- Custom models can reduce computational costs, especially during training and inference, making them suitable for environments with limited computational resources.
- Custom architectures provide the flexibility to design the model as needed, including adjusting the number of layers, types of layers, and their configurations. This flexibility allows for experimenting with novel structures or specific optimizations to better fit the problem.

Limitations:

- Training a custom model from scratch requires significant time and computational resources. It involves learning all features from the data, which can be time-consuming, particularly for large and complex datasets.
- Due to the potential lack of sufficient training data or suboptimal design, custom models are more prone to underfitting or overfitting. This leads to poor generalization on unseen data.
- Custom models often achieve lower accuracy compared to pretrained models, especially when dealing with complex tasks. Without access to the massive datasets that pretrained models are trained on, the custom model may struggle to generalize well.

2. **Pretrained Model**

Advantages:

- By using pretrained weights and fine-tuning them on a new dataset, pretrained models can achieve high performance with significantly reduced training time.
- Pretrained models have already learned rich, hierarchical features from massive datasets. These features capture general patterns that are transferable across tasks, making pretrained models excellent at handling diverse, complex tasks without requiring extensive retraining.
- Pretrained models often provide superior performance out-of-the-box and even more so after fine-tuning on a specific task. This is particularly true when the target dataset is similar to the dataset the model was originally trained on, ensuring efficient adaptation.

Limitations:

- Fine-tuning a pretrained model can be computationally expensive, particularly when dealing with large models. Fine-tuning may involve adjusting all or most of the layers, requiring significant memory and processing power.
- While pretrained models perform well on general tasks, they may not fully adapt to highly specialized datasets. In cases where the target dataset is very different from the one the model was originally trained on, fine-tuning may not provide optimal results, and the pretrained model might not generalize well.
- Pretrained models are often much larger than custom models due to their deep architecture and large number of parameters. This can pose challenges for deployment in resource-constrained environments where memory and storage capacity are limited.

3 GitHub Link

Here is a link to: Simple-convolutional-neural-network-to-perform-classification.

Table 1: Trade-offs of Custom and Pretrained Models

Factor	**Custom Model**	**Pretrained Model**
Performance	Can be optimized for specific tasks and datasets, but may not perform as well as pre-trained models for complex tasks.	Can achieve superior results with fine-tuning, especially when datasets are similar.
Training Time	Requires significant time and computational resources to train from scratch. Longer development cycle.	Significantly reduced training time since the model already contains learned features. Fine-tuning may still take time.
Computational Resources	Custom models can be designed to be more resource-efficient, making them suitable for constrained environments. Requires more resources for training.	Pretrained models are generally larger and more computationally expensive during fine-tuning.
Flexibility	Highly flexible in terms of design and architecture.	Limited flexibility; the architecture is already defined. Modifications usually involve fine-tuning or minor adjustments.
Overfitting/Underfitting	More prone to overfitting or underfitting, especially if there is insufficient data or poor model design.	Less prone to overfitting/underfitting after fine-tuning, especially when using large datasets.
Generalization	Can struggle to generalize well if not trained properly or if the dataset is too complex.	Pretrained models generalize better across tasks.
Accuracy	Custom models may achieve lower accuracy initially, particularly on complex tasks.	Pretrained models tend to have higher accuracy on general tasks and perform better with less training data.
Deployment	Custom models can be optimized for deployment in resource-constrained environments.	Pretrained models are usually larger and more resource-hungry, which can make deployment challenging in constrained environments.
Data Dependency	Requires sufficient high-quality data to train effectively.	Can work well with smaller datasets. Might not perform as well on highly specialized data.