Documentation

(ResNet)

Introduction:

ResNet50, introduced in the paper "Deep Residual Learning for Image Recognition" by He et al. in 2015, is a powerful convolutional neural network (CNN) architecture that won the ILSVRC 2015 competition. It addresses the degradation problem in deep networks by introducing residual connections, allowing the training of much deeper networks without suffering from vanishing gradients.

Architecture Overview:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ResNet50 is a 50-layer deep CNN composed of:

- An initial convolutional layer and max pooling.

- Four stages of convolutional blocks, each containing multiple identity and convolutional blocks.

- A global average pooling layer.

- A final fully connected (dense) layer with softmax activation for classification.

Key Components:

1. Convolutional Block : Introduces a residual connection with a convolutional layer to match dimensions when there's a change in input size (e.g., due to stride).

```python
def convolutional_block(X, filters, kernel_size, strides):
    F1, F2, F3 = filters
    # Where downsampling happens
    X_shortcut = X

    X = Conv2D(F1, (1, 1), strides=strides, padding='valid')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    X = Conv2D(F2, kernel_size, strides=(1, 1), padding='same')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    X = Conv2D(F3, (1, 1), strides=(1, 1), padding='valid')(X)
    X = BatchNormalization(axis=3)(X)

    X_shortcut = Conv2D(F3, (1, 1), strides=strides, padding='valid')(X_shortcut)
    X_shortcut = BatchNormalization(axis=3)(X_shortcut)

    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

2. Identity Block : A block where the input and output dimensions are the same, allowing for direct addition.

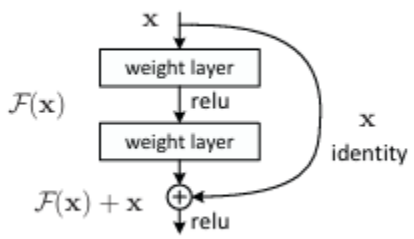


Figure 2. Residual learning: a building block.

```python
def identity_block(X, filters, kernel_size):
    F1, F2, F3 = filters


    X_shortcut = X

    # Reduction
    X = Conv2D(F1, (1, 1), strides=(1, 1), padding='valid')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    # Filter
    X = Conv2D(F2, kernel_size, strides=(1, 1), padding='same')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    # Expansion
    X = Conv2D(F3, (1, 1), strides=(1, 1), padding='valid')(X)
    X = BatchNormalization(axis=3)(X)

    # shortcut
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

3. Residual Connection : Enables the network to learn residual functions with reference to the layer inputs, improving training by mitigating the vanishing gradient problem.

Step-by-Step Explanation :

1. Initial Layers :

- Conv1 : 7x7 convolution with 64 filters, stride 2.
- Batch Normalization and ReLU Activation .
- Max Pooling : 3x3 kernel, stride 2.

```
    # Stage 1
    X = Conv2D(64, (7, 7), strides=(2, 2), padding='same')(X_input)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2), padding='same')(X)
```

2. Stage 1 :

- Convolutional Block : Adjusts dimensions using a 1x1 convolution in the shortcut path.

- Conv layers with filter sizes: 64, 64, 256.

- Residual connection matches output dimensions.

- Two Identity Blocks : Same dimensions, allowing direct addition.

- Conv layers with filter sizes: 64, 64, 256.

```
# Stage 2
X = convolutional_block(X, [64, 64, 256], kernel_size=(3, 3), strides=(1, 1))
X = identity_block(X, [64, 64, 256], kernel_size=(3, 3))
X = identity_block(X, [64, 64, 256], kernel_size=(3, 3))
```

3. Stage 2 :

- Similar structure with increased filters to 128, followed by identity blocks.

```
# Stage 3
X = convolutional_block(X, [128, 128, 512], kernel_size=(3, 3), strides=(2, 2))
X = identity_block(X, [128, 128, 512], kernel_size=(3, 3))
X = identity_block(X, [128, 128, 512], kernel_size=(3, 3))
X = identity_block(X, [128, 128, 512], kernel_size=(3, 3))
```

4. Stage 3 and Stage 4 :

- Filters increase to 256 and 512 respectively.

- Incorporate convolutional and identity blocks, deepening the network.

```
# Stage 4
X = convolutional_block(X, [256, 256, 1024], kernel_size=(3, 3), strides=(2, 2))
for _ in range(5):
    X = identity_block(X, [256, 256, 1024], kernel_size=(3, 3))

# Stage 5
X = convolutional_block(X, [512, 512, 2048], kernel_size=(3, 3), strides=(2, 2))
X = identity_block(X, [512, 512, 2048], kernel_size=(3, 3))
X = identity_block(X, [512, 512, 2048], kernel_size=(3, 3))
```

5. Final Layers :

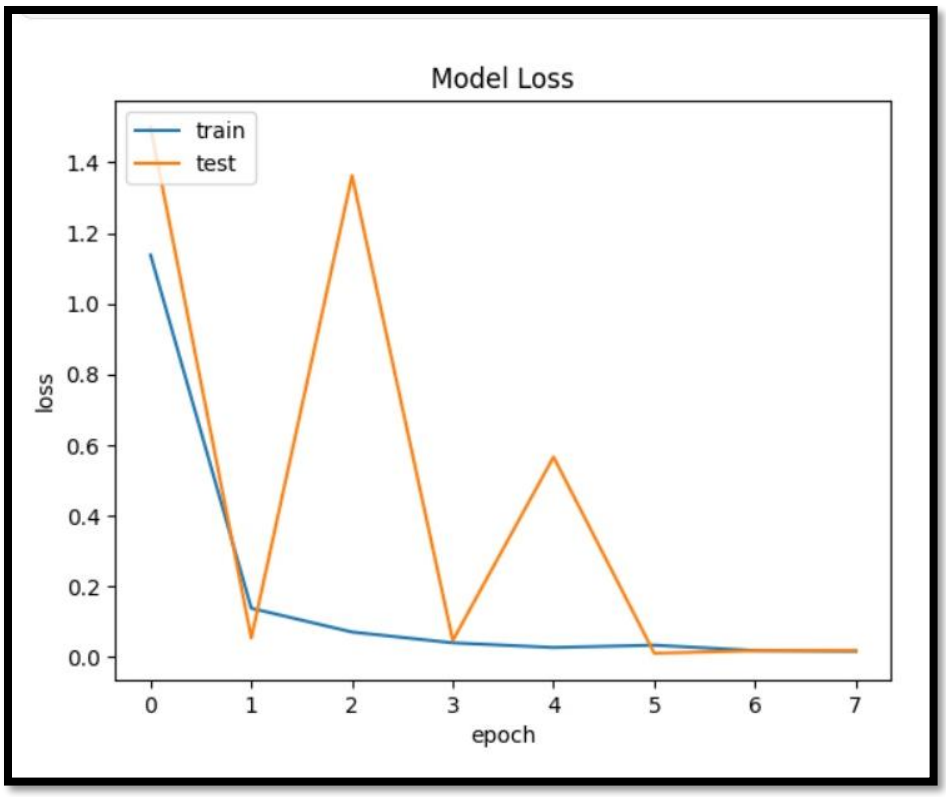
   - Global Average Pooling : Reduces each feature map to a single value.
   - Dense Layer : Outputs class probabilities using softmax activation.

```
# Average Pooling
X = GlobalAveragePooling2D()(X)
X = Dense(classes, activation='softmax')(X)
```

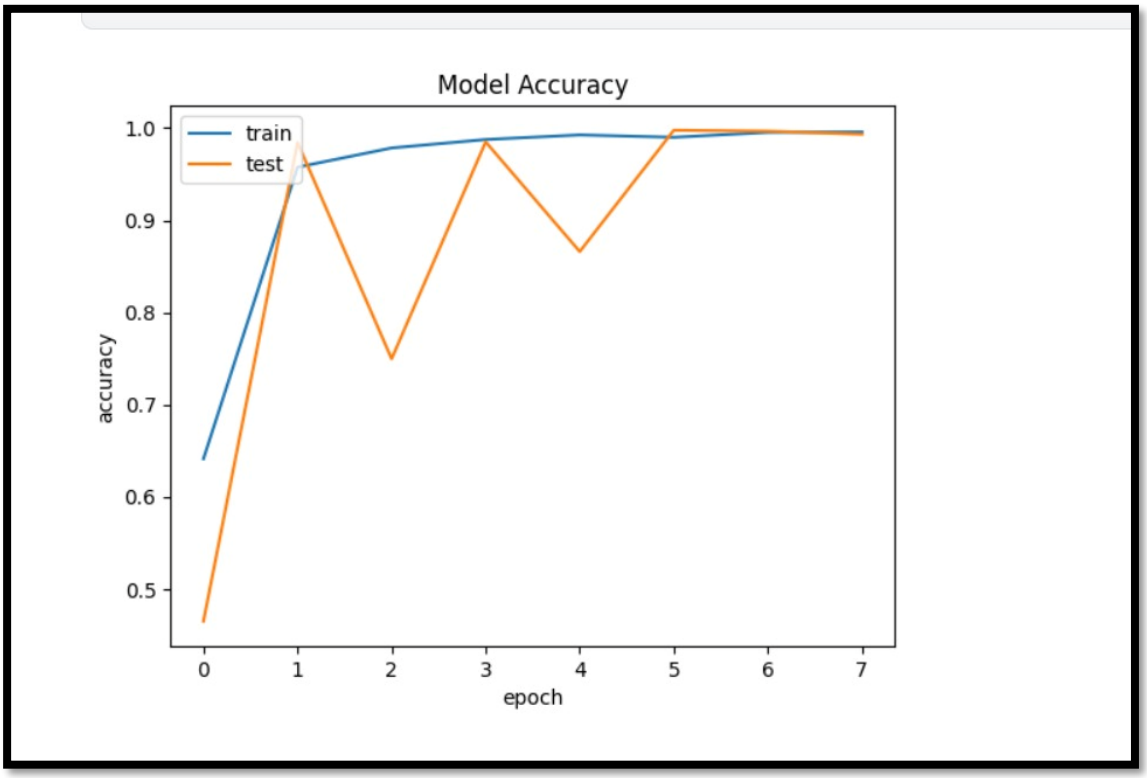
Results:

Loss:

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

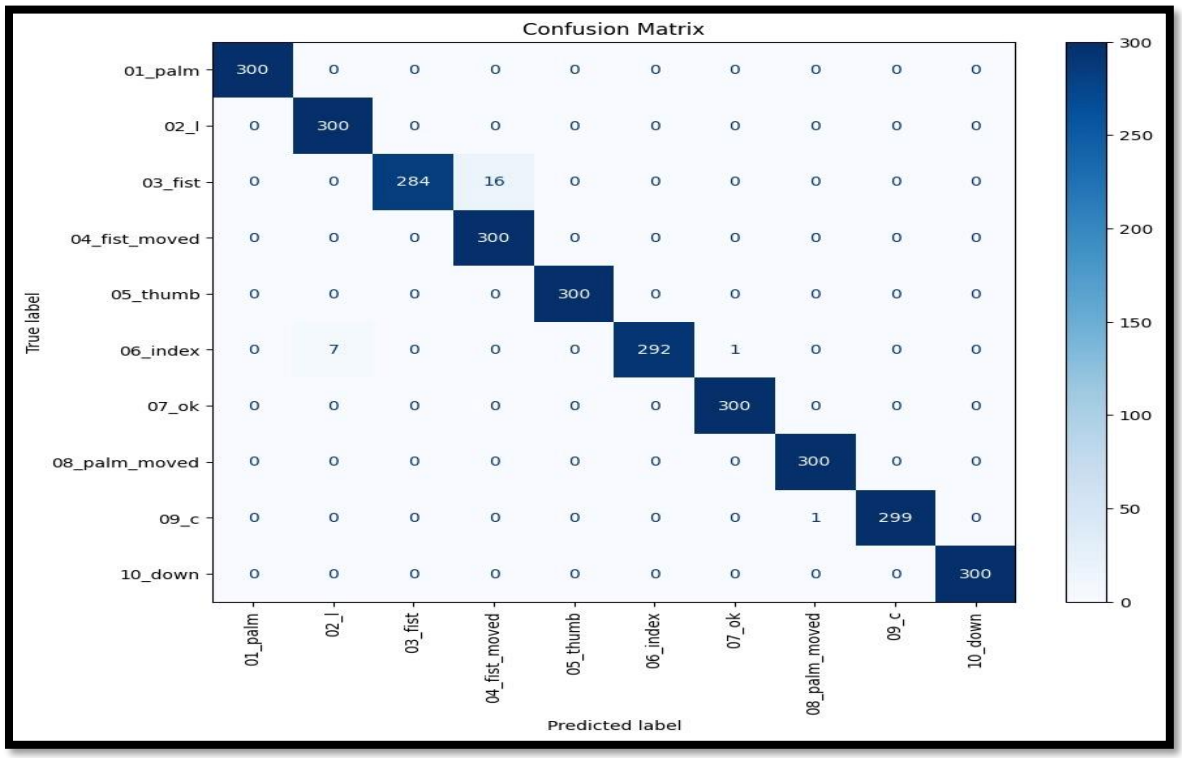

Model Loss
train
test
loss
epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1
2
3
4
5
6
7

Accuracy:


Model Accuracy
train
test
accuracy
epoch
0.5
0.6
0.7
0.8
0.9
1.0
0
1
2
3
4
5
6
7

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Get predictions and true labels
y_pred = model.predict(X_test)
y_pred_classes = y_pred.argmax(axis=1)
y_true = Y_test_int  # True labels (integer form)

# Compute
cm = confusion_matrix(y_true, y_pred_classes)


disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=label_encoder.classes_)
fig, ax = plt.subplots(figsize=(10, 8))
disp.plot(cmap=plt.cm.Blues, ax=ax)
ax.set_title("Confusion Matrix")


plt.xticks(rotation=90)

plt.show()
```

Confusion matrix visualization :



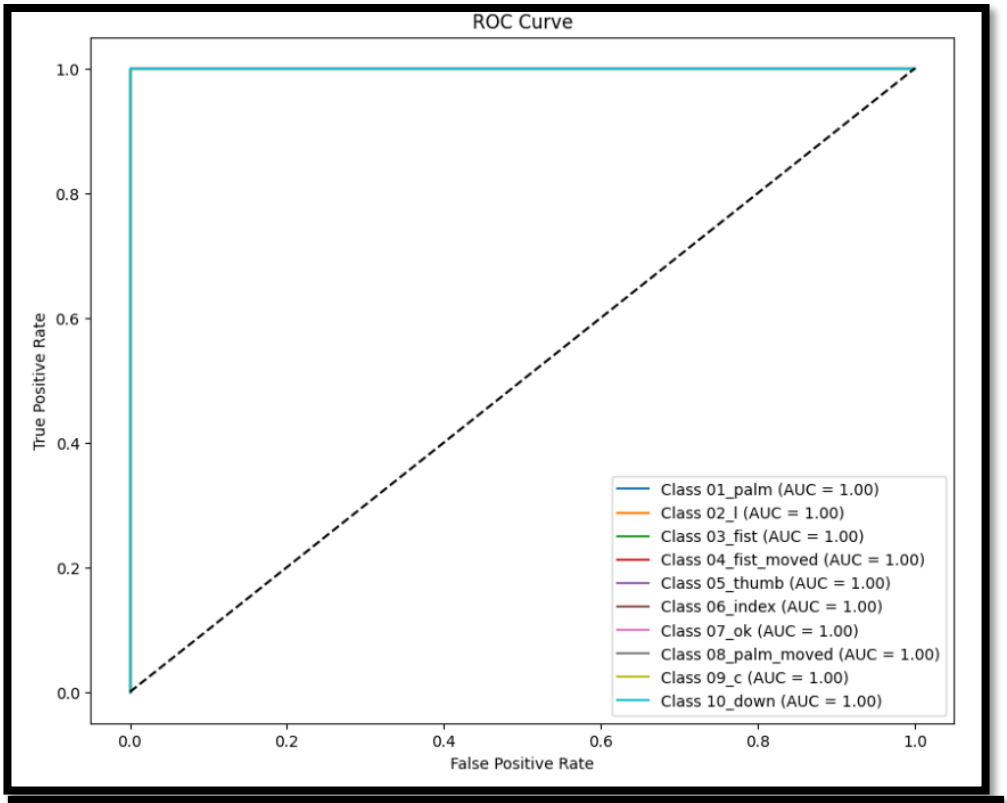
```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Get predictions and true labels
y_pred = model.predict(X_test)
y_pred_classes = y_pred.argmax(axis=1)
y_true = Y_test_int  # True labels (integer form)

# Compute
cm = confusion_matrix(y_true, y_pred_classes)


disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=label_encoder.classes_)
fig, ax = plt.subplots(figsize=(10, 8))
disp.plot(cmap=plt.cm.Blues, ax=ax)
ax.set_title("Confusion Matrix")


plt.xticks(rotation=90)

plt.show()
```

Recall,Precision,F-score:

```python
from sklearn.metrics import classification_report

report = classification_report(y_true, y_pred_classes, target_names=label_encoder.classes_)
print(report)
```

```
              precision    recall  f1-score   support

     01_palm       1.00      0.97      0.99       300
        02_l       1.00      1.00      1.00       300
     03_fist       1.00      0.70      0.82       300
04_fist_moved       0.75      1.00      0.86       300
     05_thumb       1.00      1.00      1.00       300
     06_index       1.00      1.00      1.00       300
        07_ok       1.00      1.00      1.00       300
08_palm_moved       0.99      1.00      1.00       300
        09_c       1.00      1.00      1.00       300
     10_down       1.00      1.00      1.00       300

    accuracy                           0.97      3000
   macro avg       0.97      0.97      0.97      3000
weighted avg       0.97      0.97      0.97      3000
```

ROC, AUC visualization:




```
from sklearn.metrics import roc_auc_score

micro_auc = roc_auc_score(y_test_binarized, y_pred, average='micro')
macro_auc = roc_auc_score(y_test_binarized, y_pred, average='macro')

print(f"Micro-average AUC: {micro_auc}")
print(f"Macro-average AUC: {macro_auc}")
```

```
Micro-average AUC: 0.9999856666666667
Macro-average AUC: 1.0
```

Implementation from the Original Paper:

The model was implemented by closely following the architecture described in the original ResNet paper:

- Residual Blocks : Implemented both identity and convolutional blocks as per the paper's specifications.

- Filter Sizes and Strides : Maintained the filter sizes and strides to ensure consistency.

- Batch Normalization : Applied after each convolutional layer to stabilize and accelerate training.

- Activation Functions : Used ReLU activations throughout, as recommended.

Pros and Cons of ResNet50:

Pros:

- Mitigates Vanishing Gradients : Residual connections allow gradients to flow directly through the network, facilitating the training of very deep architectures.

- Improved Accuracy : Deeper networks can capture complex patterns, leading to higher performance.

- Versatility : Effective in various applications beyond image classification, like object detection and segmentation.

Cons:

- Computationally Intensive : Depth increases computational requirements, leading to longer training times and the need for more powerful hardware.

- Memory Consumption : Larger models consume more memory, which can be a limitation for some systems.

(DenseNet Model)

Introduction

DenseNet is a state-of-the-art convolutional neural network (CNN) architecture designed to enhance the flow of information and gradients throughout the network. This architecture effectively addresses the vanishing gradient problem, enabling the training of very deep networks.

Architecture Overview

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

DenseNet features a unique dense connectivity pattern, where each layer receives inputs from all preceding layers. This design results in a highly efficient model that requires fewer parameters while achieving high accuracy.

Key Components:

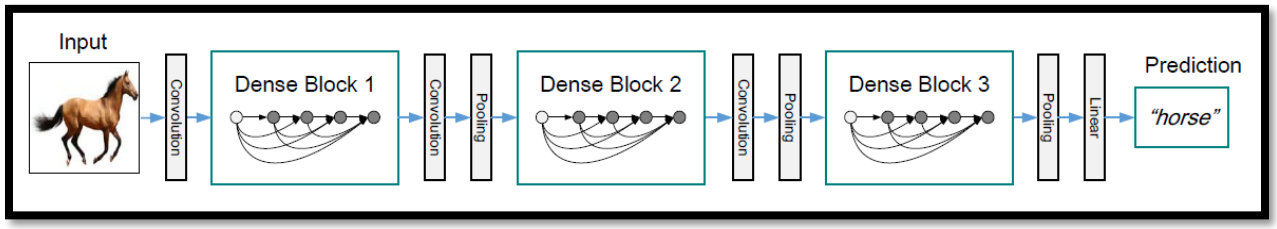
Dense Block: Each layer within a dense block receives inputs from all previous layers, promoting feature reuse.

Transition Layer: Reduces the number of feature maps and performs down-sampling.

Global Average Pooling Layer: Condenses each feature map into a single value.

Fully Connected Layer: Outputs class probabilities using softmax activation

Step-by-Step Explanation



Initial Layers:

Convolution: 7x7 convolution with 64 filters, stride 2.

Batch Normalization and ReLU Activation.

Max Pooling: 3x3 kernel, stride 2.

Dense Blocks:

Dense Block 1:

Convolutional layers with filter sizes: 64, 64, 256.

Each layer's output is concatenated with the inputs of subsequent layers.

Transition Layer 1:

1x1 convolution followed by 2x2 average pooling.

Dense Block 2:

Increased filters to 128.

Transition Layer 2:

Similar structure to Transition Layer 1.

Dense Block 3:

Filters increase to 256.

Transition Layer 3:

Similar structure to previous transition layers.

Dense Block 4:

Filters increase to 512.

Final Layers:

Global Average Pooling: Reduces each feature map to a single value.

Dense Layer: Outputs class probabilities using softmax activation.

Pros and Cons of DenseNet

Pros:

Mitigates Vanishing Gradients: Dense connections enable gradients to flow directly through the network.

Improved Accuracy: Deeper networks can capture complex patterns, resulting in higher performance.

Versatility: Effective in various applications beyond image classification, including object detection and segmentation.

Cons:

Computationally Intensive: Increased depth leads to higher computational demands.

Memory Consumption: Larger models require more memory, which can be a limitation for certain systems.

1- Define the DenseNet Model

```
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 22
4, 3))

# Add custom layers
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Global average pooling
x = Dropout(0.5)(x)  # Dropout for regularization
x = Dense(128, activation='relu')(x)  # Fully connected layer
x = Dropout(0.5)(x)  # Additional dropout
predictions = Dense(len(label_encoder.classes_), activation='softmax')(x)  # Output l
ayer
```

2- Create the model

```
model = Model(inputs=base_model.input, outputs=predictions)

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Display the model summary
model.summary()
```

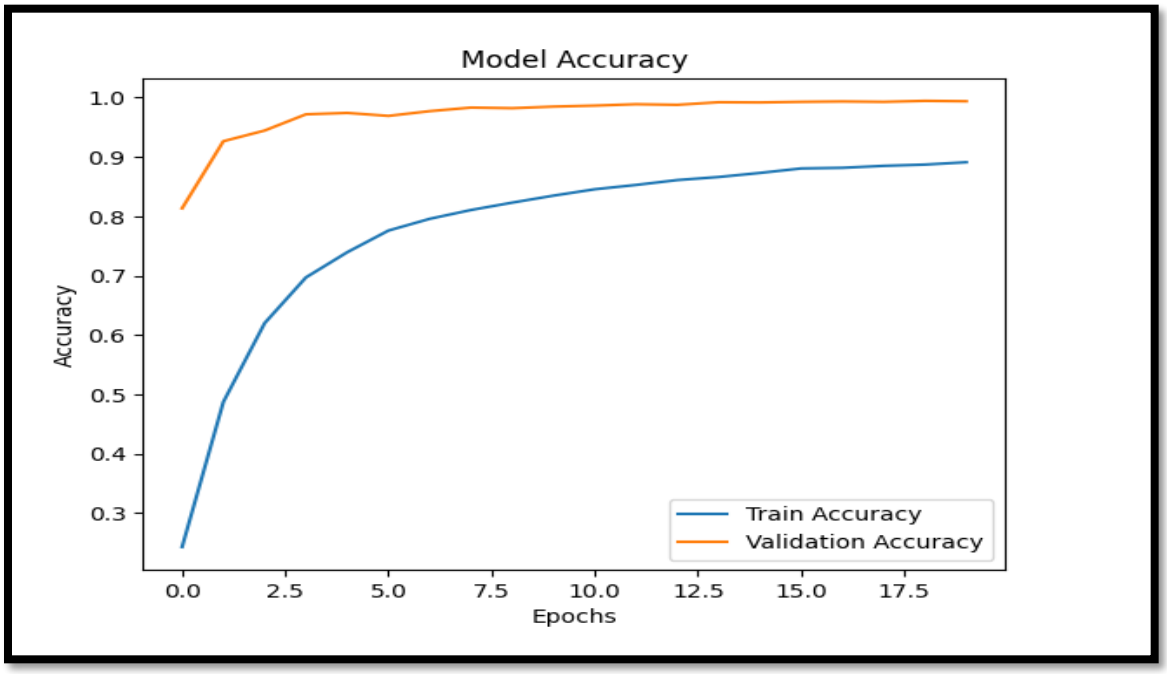
3- Train the model

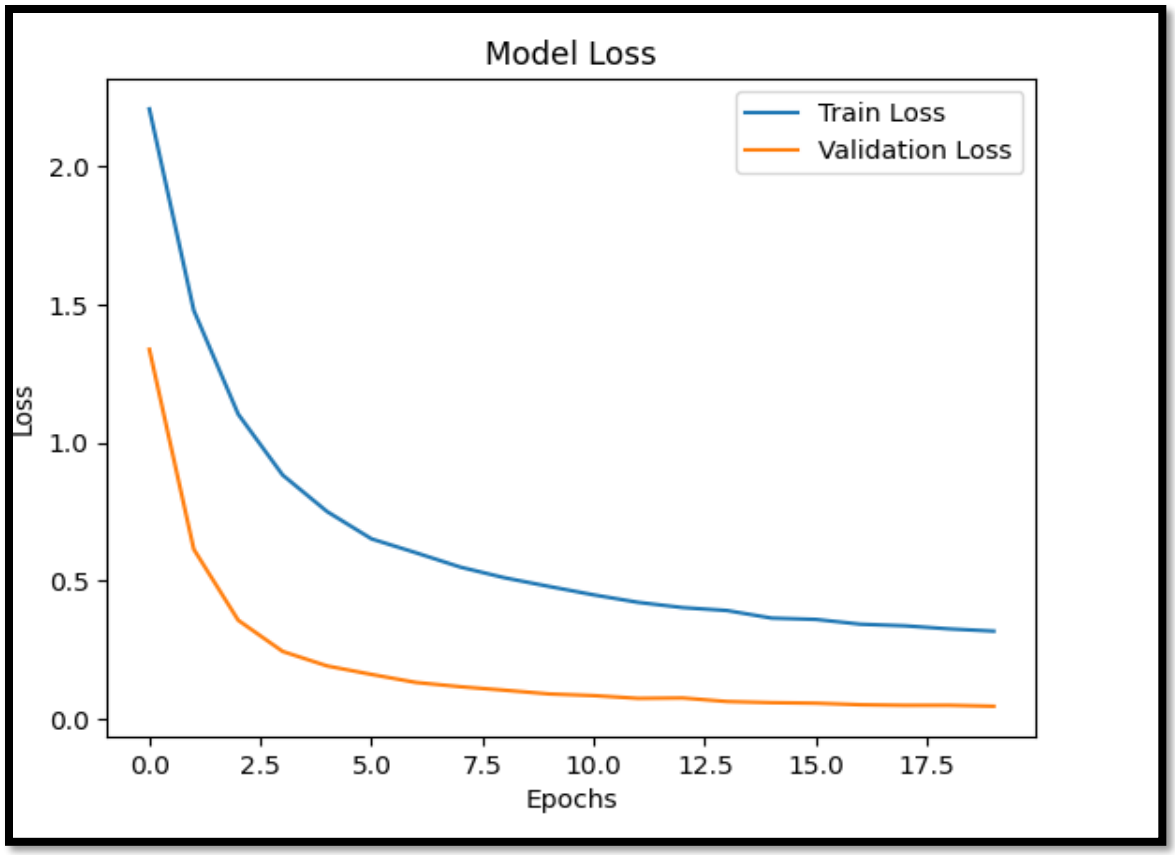
```
# --- Train the Model ---
history = model.fit(
    datagen.flow(X_train, Y_train, batch_size=32),
    validation_data=(X_val, Y_val),
    epochs=20,
    verbose=1
)
```

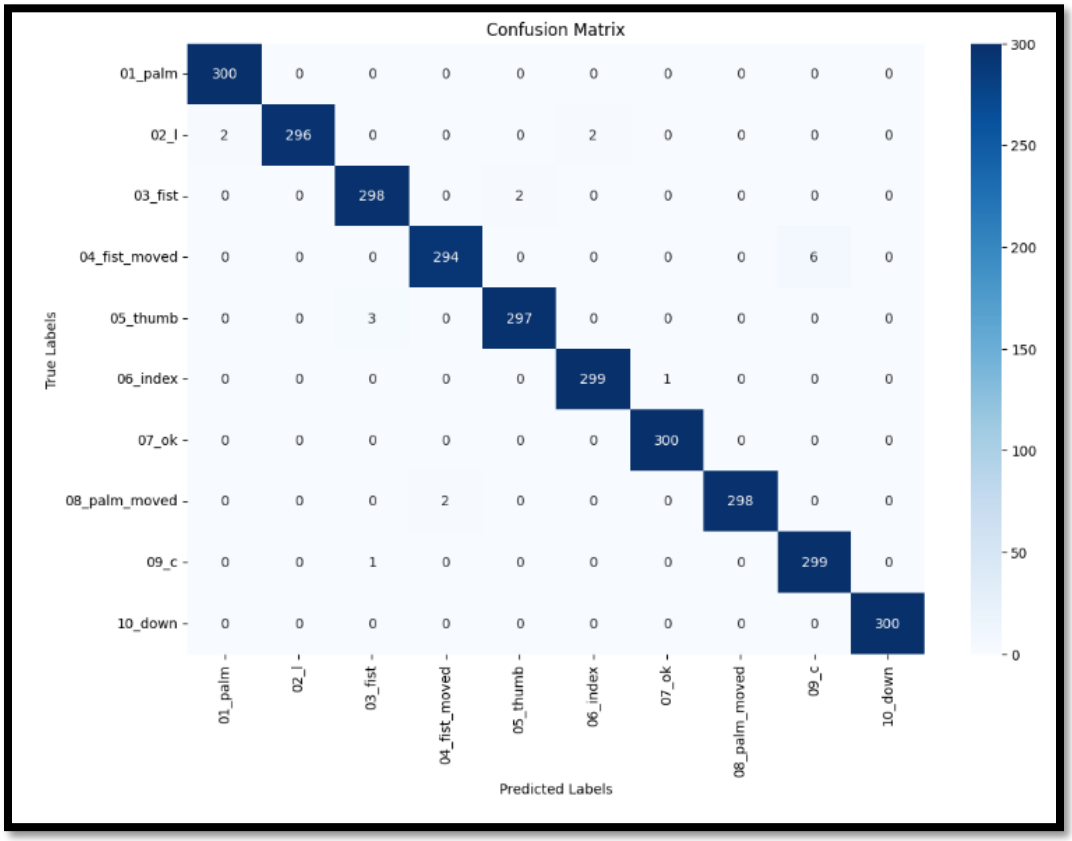
4- Evaluate the Model

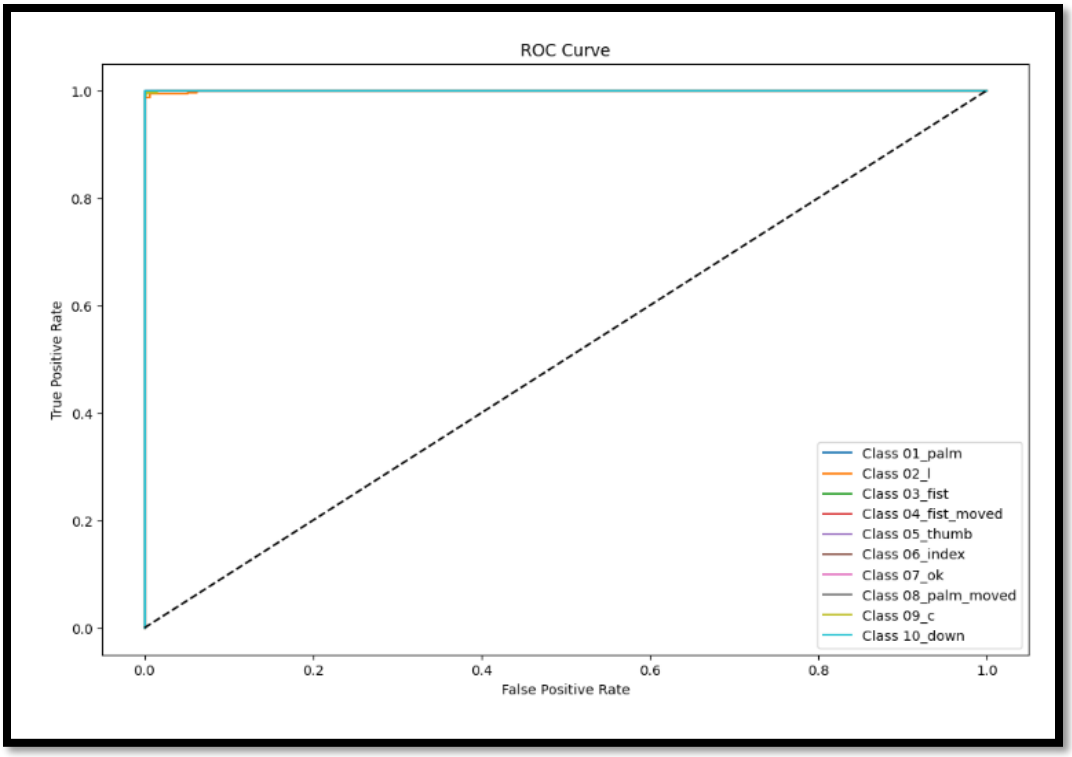
```
test_loss, test_acc = model.evaluate(X_test, Y_test)
print(f"Test Loss: {test_loss}, Test Accuracy: {test_acc}")

# --- Visualize Results ---
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# --- Generate Classification Report ---
from sklearn.metrics import classification_report

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(Y_test, axis=1)

print(classification_report(y_true_classes, y_pred_classes, target_names=label_encode
```

Result


Model Accuracy
Accuracy
Epochs
Train Accuracy
Validation Accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Model Loss
Loss
Epochs
Train Loss
Validation Loss
2.0
1.5
1.0
0.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Confusion Matrix
01_palm 300 0 0 0 0 0 0 0 0 0
02_l 2 296 0 0 0 2 0 0 0 0
03_fist 0 0 298 0 2 0 0 0 0 0
04_fist_moved 0 0 0 294 0 0 0 0 6 0
05_thumb 0 0 3 0 297 0 0 0 0 0
06_index 0 0 0 0 0 299 1 0 0 0
07_ok 0 0 0 0 0 0 300 0 0 0
08_palm_moved 0 0 0 2 0 0 0 298 0 0
09_c 0 0 1 0 0 0 0 0 299 0
10_down 0 0 0 0 0 0 0 0 0 300
True Labels
01_palm
02_l
03_fist
04_fist_moved
05_thumb
06_index
07_ok
08_palm_moved
09_c
10_down
Predicted Labels
300
250
200
150
100
50
0


ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Class 01_palm
Class 02_l
Class 03_fist
Class 04_fist_moved
Class 05_thumb
Class 06_index
Class 07_ok
Class 08_palm_moved
Class 09_c
Class 10_down

(Xception)

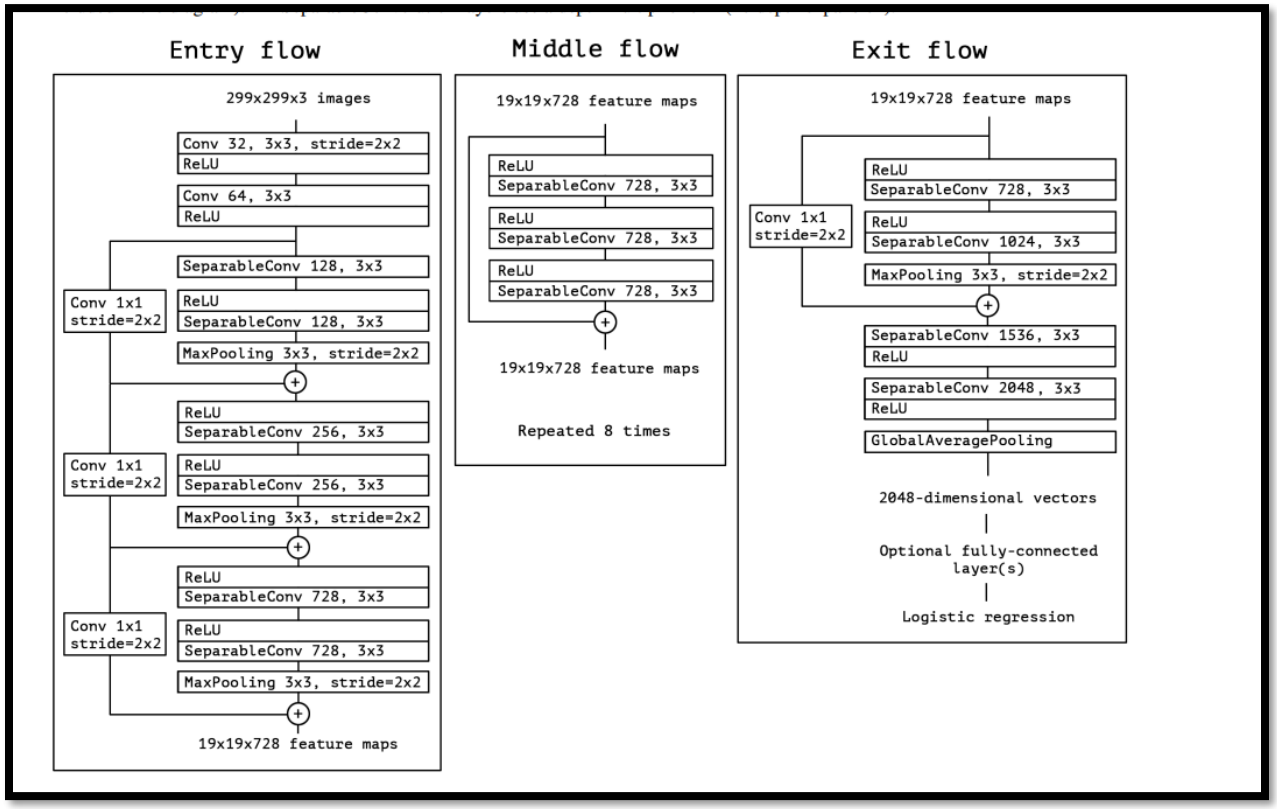
Introduction:

Xception, introduced in the paper "Xception: Deep Learning with Depthwise Separable Convolutions" by Chollet in 2016, is an advanced convolutional neural network (CNN) architecture that builds upon the Inception model. It utilizes depthwise separable convolutions to improve model efficiency and performance, making it particularly effective for various computer vision tasks.

Architecture Overview

Xception consists of:

- An initial entry block followed by multiple depthwise separable convolutional blocks.
- A final fully connected (dense) layer with softmax activation for classification.



Key Components:

1. **Depthwise Separable Convolution**:
    - Composed of two layers: depthwise convolution and pointwise convolution.
    - Reduces the number of parameters significantly while maintaining performance.
2. **Entry Block**:
    - Initial convolution followed by batch normalization and ReLU activation.
    - Max pooling to downsample the feature maps.
3. **Separable Convolution Blocks**:
    - Each block consists of depthwise separable convolutions followed by residual connections to facilitate gradient flow.
4. **Final Layers**:
    - Global Average Pooling to reduce each feature map to a single value.
    - Dense Layer to output class probabilities using softmax activation.

Step-by-Step Explanation

3-Initial Layers:

```
base_model = Xception(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

- The model begins by loading the **Xception** architecture without the top classification layer (include_top=False) and using pre-trained weights from ImageNet.
- Input shape is defined as (224, 224, 3), which corresponds to the standard input size for Xception.

2- Global Average Pooling and Dropout

```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.5)(x)  # Add dropout for regularization
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)  # Add another dropout layer
predictions = Dense(len(label_encoder.classes_), activation='softmax')(x)
```

Global Average Pooling:

- This layer reduces each feature map to a single value, effectively summarizing the feature maps and reducing dimensionality.

Dropout Layer:

- A dropout layer is added to reduce overfitting by randomly setting a fraction (0.5) of the input units to 0 during training.

- **Dense Layer**:
  - A fully connected layer with 128 units and ReLU activation is added to learn complex patterns.
- **Another Dropout Layer**:
  - Another dropout layer is added for further regularization.
- **Output Layer**:
  - The final output layer uses softmax activation to produce class probabilities based on the number of classes in the label encoder.

3. Model Compilation

```
model = Model(inputs=base_model.input, outputs=predictions)

# Freeze base model layers for transfer learning
for layer in base_model.layers:
    layer.trainable = False

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- The model is compiled with:
  - **Optimizer**: Adam with a learning rate of 0.0001.
  - **Loss Function**: Categorical crossentropy, suitable for multi-class classification.
  - **Metrics**: Accuracy to evaluate the model's performance.

4-Train the Model

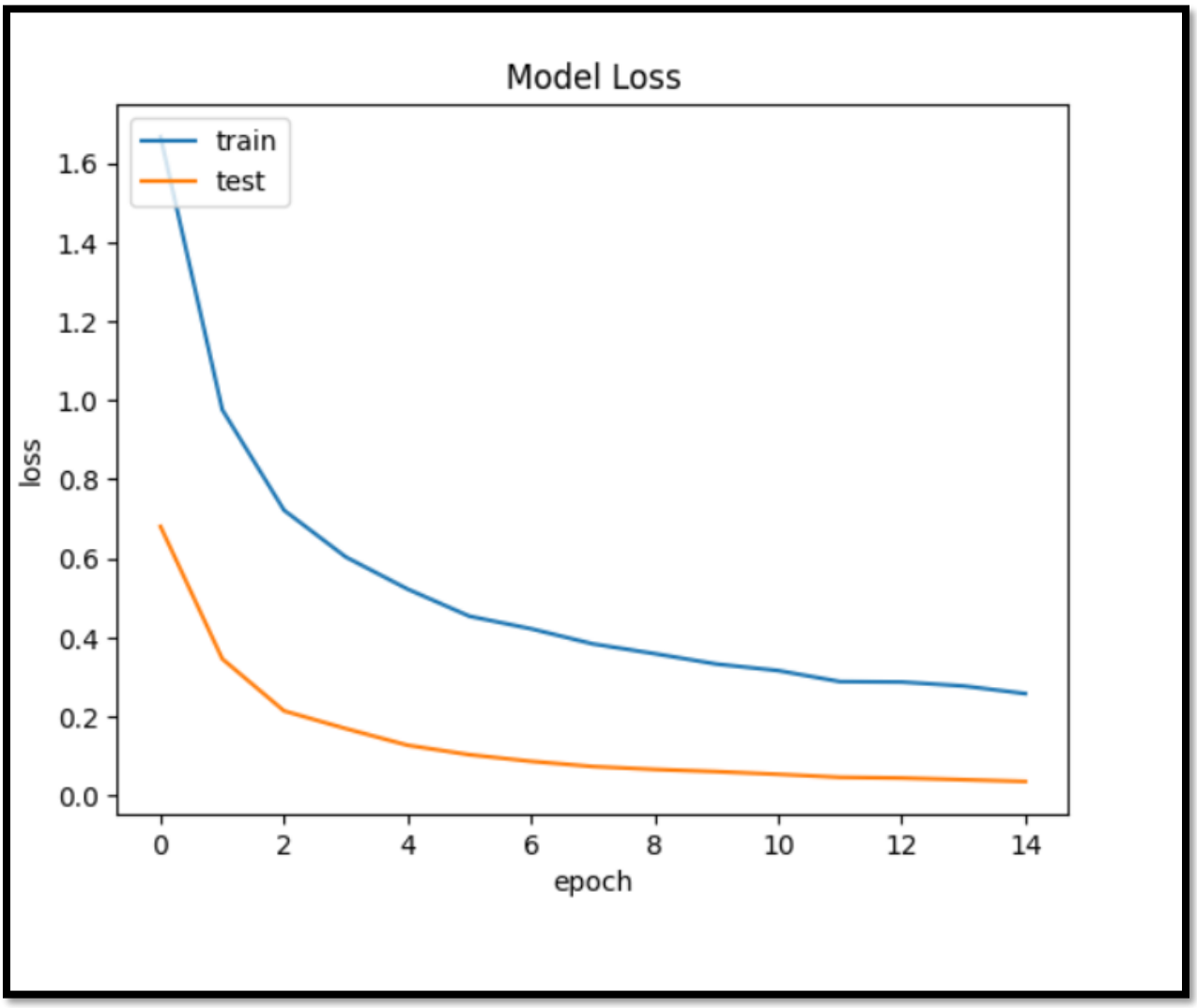
```
history = model.fit(
    datagen.flow(X_train, Y_train, batch_size=32),
    validation_data=(X_val, Y_val),
    epochs=15
)
```

```
Epoch 1/15
```

- The model is trained using the fit method with data augmentation provided by datagen for the training set and validation data.

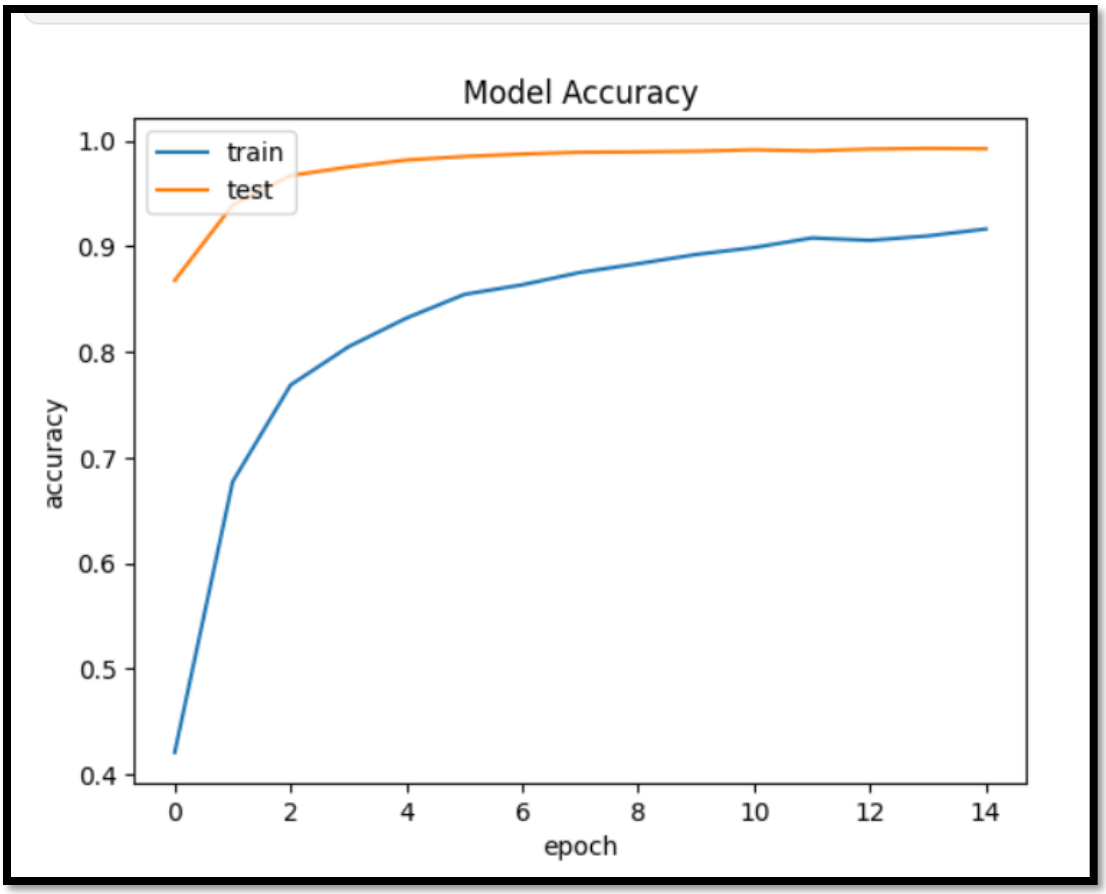
Results

Loss and Accuracy Plots



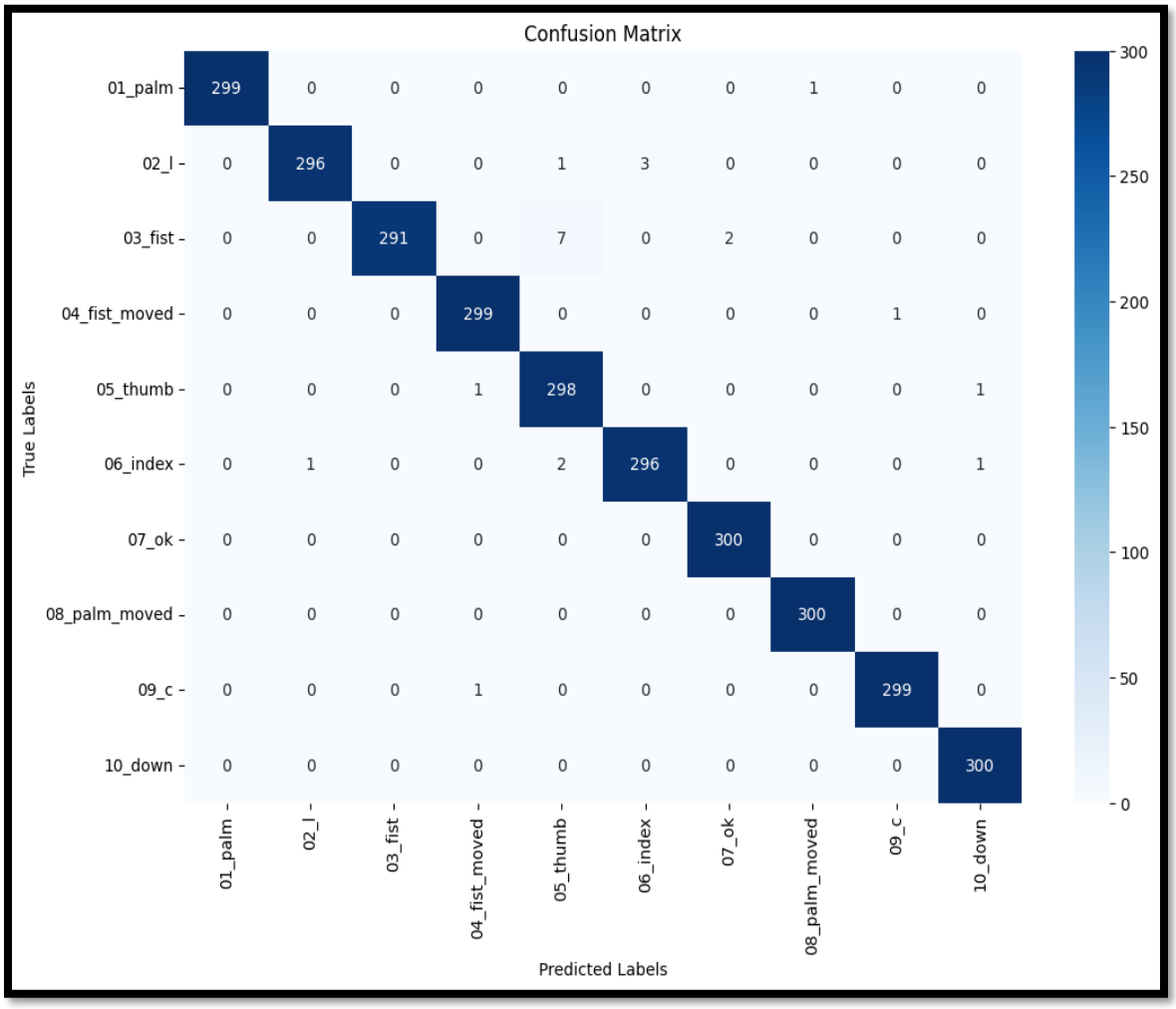
- **Accuracy**:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

- 

- **Confusion Matrix Visualization**:

```
# Confusion Matrix
conf_matrix = confusion_matrix(Y_true_classes, Y_pred_classes)
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes
_, yticklabels=label_encoder.classes_)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()

# Classification Report
print(classification_report(Y_true_classes, Y_pred_classes, target_names=label_encoder.classes
_))
```

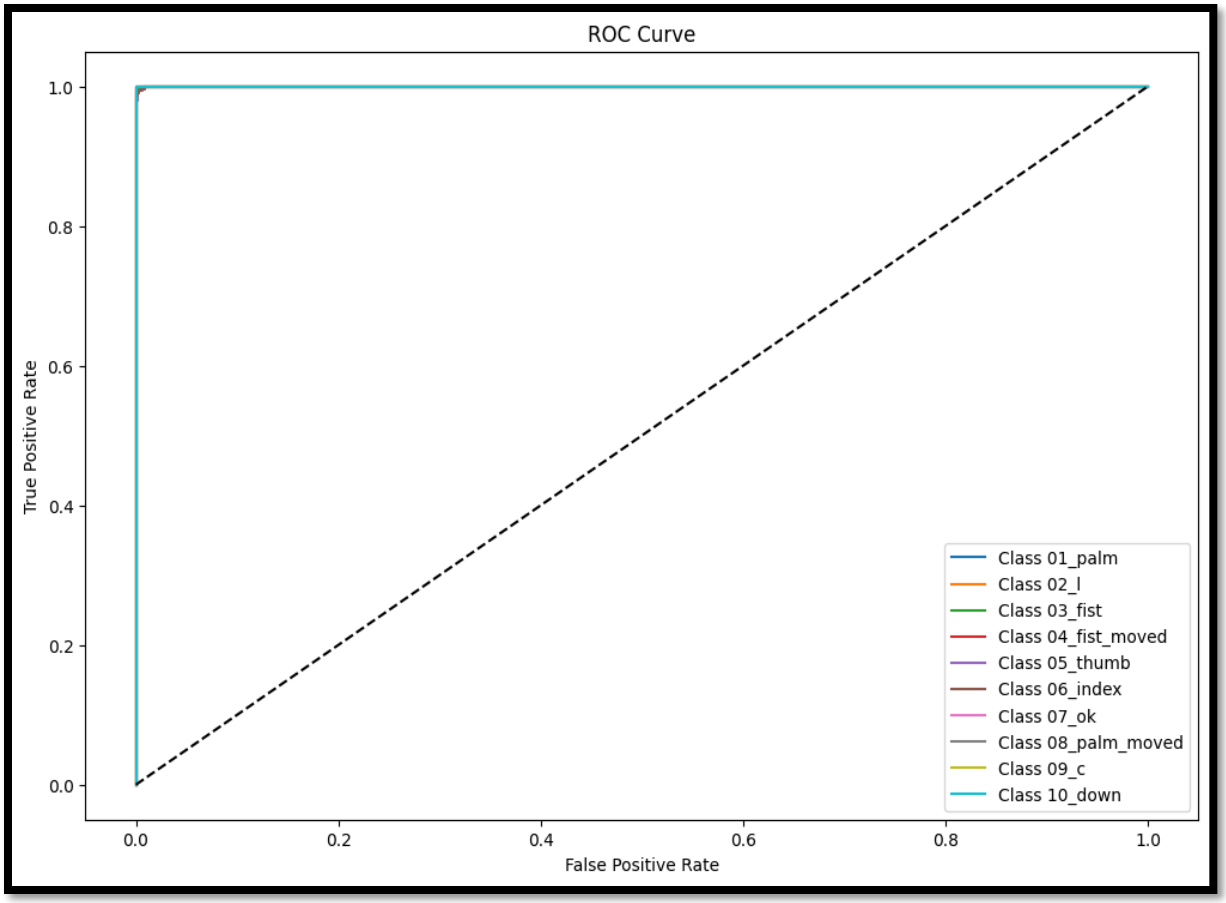


- **Recall, Precision, F-score**:

```
# Classification Report
print(classification_report(Y_true_classes, Y_pred_classes, target_names=label_encoder.classes
_))
```

- **ROC, AUC Visualization**:

```
# ROC-AUC Curve
roc_auc = roc_auc_score(Y_test, Y_pred, multi_class='ovr')
print(f"ROC-AUC Score: {roc_auc:.2f}")

# Plot ROC Curve
fpr = {}
tpr = {}
thresholds = {}
for i in range(len(label_encoder.classes_)):
    fpr[i], tpr[i], thresholds[i] = roc_curve(Y_test[:, i], Y_pred[:, i])

plt.figure(figsize=(12, 8))
for i in range(len(label_encoder.classes_)):
    plt.plot(fpr[i], tpr[i], label=f"Class {label_encoder.classes_[i]}")
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```



Pros and Cons of Xception

Pros:

- **Efficiency**: Depthwise separable convolutions significantly reduce the number of parameters, making the model lighter and faster.
- **Improved Performance**: Achieves state-of-the-art results on various benchmarks, outperforming traditional models.
- **Flexibility**: Effective in a range of applications, including image classification, object detection, and segmentation.

Cons:

- **Complexity**: The architecture can be more complex to implement compared to simpler models.
- **Computational Requirements**: While more efficient than some architectures, it still requires significant computational resources for training.

Model	Results	Pros	Cons	Best For
ResNet50	Loss: 0.0184 Accuracy: 99.17%	- Mitigates vanishing gradients - Captures complex patterns - Versatile for various tasks	- Computationally intensive - High memory consumption	- Very deep networks requiring robust training
DenseNet	Loss: 0.0522 Accuracy: 99.37%	- Mitigates vanishing gradients - Improved accuracy - Effective for various applications	- High computational demands - Requires more memory	- Tasks needing deep networks with high feature reuse
Xception	Loss: 0.073 Test Accuracy: 99.27%	- Efficient with fewer parameters - State-of-the-art performance - Versatile across applications	- More complex to implement - High computational needs	- Large datasets with diverse classes

References

- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition . *arXiv preprint arXiv:1512.03385*.

- Chollet, F. (2016). Xception: Deep Learning with Depthwise Separable Convolutions . *arXiv preprint arXiv:1610.02357*.

- Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2016). Densely Connected Convolutional Networks . *arXiv preprint arXiv:1608.06993*.