Face Classification

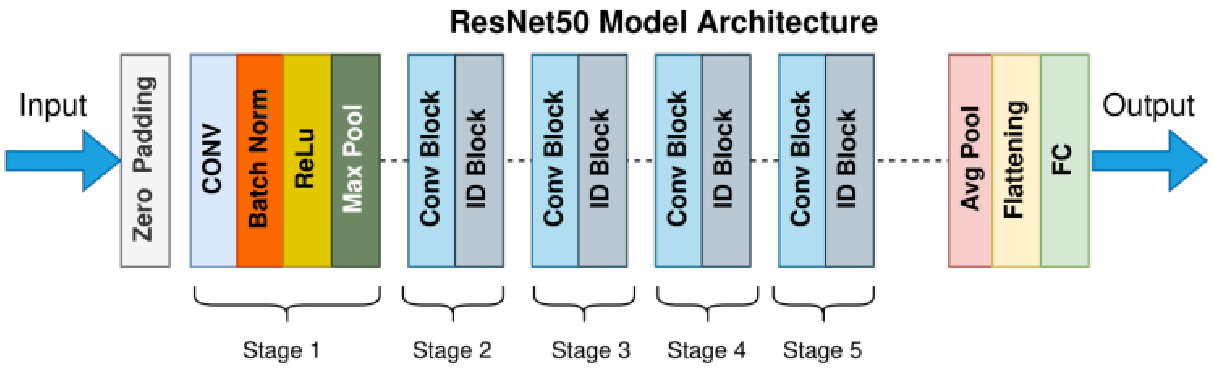
Architectures for Models

Overview:

This project compares three state-of-the-art architectures for face classification: ResNet50, Xception, and DenseNet121. Each architecture has unique structural innovations that contribute to its performance. This document explains each model in detail, highlighting their components, step-by-step implementation, and visual representations. Finally, we discuss their training and evaluation results.

Architectures:

1. ResNet50



Introduction:

ResNet (Residual Network), introduced by He et al. in 2015, tackles the vanishing gradient problem in deep networks by introducing residual connections. These allow the network to learn identity mappings, ensuring better gradient flow and faster convergence.

Key Components:

1. **Residual Connections: Shortcut connections bypass certain layers, creating residual blocks. The block computes:**
2. **where represents the learned residual mapping.**
3. **Bottleneck Design: Each block uses a 1x1, 3x3, and another 1x1 convolution to reduce computation while maintaining performance.**
4. **Downsampling: Performed via strided convolution in residual blocks.**

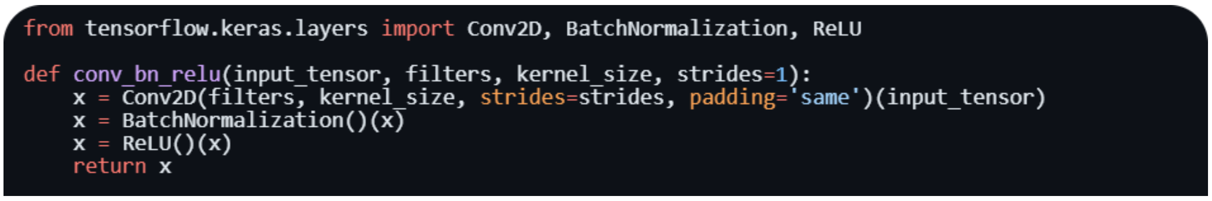
Step-by-Step Implementation with Code:

Build the CONV-BATCH_NORMALIZATION-RELU Block:

Now, we will build the Conv-BatchNorm-ReLU block as a function that will:

- **take as inputs:**
  - **a tensor (x)**
  - **the number of filters (filters)**
  - **the kernel size (kernel_size)**
  - **the strides (strides)**

- **run:**
  - **apply a Convolution layer followed by a Batch Normalization and a ReLU activation**
    - **return the tensor**



```python
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU

def conv_bn_relu(input_tensor, filters, kernel_size, strides=1):
    x = Conv2D(filters, kernel_size, strides=strides, padding='same')(input_tensor)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    return x
```

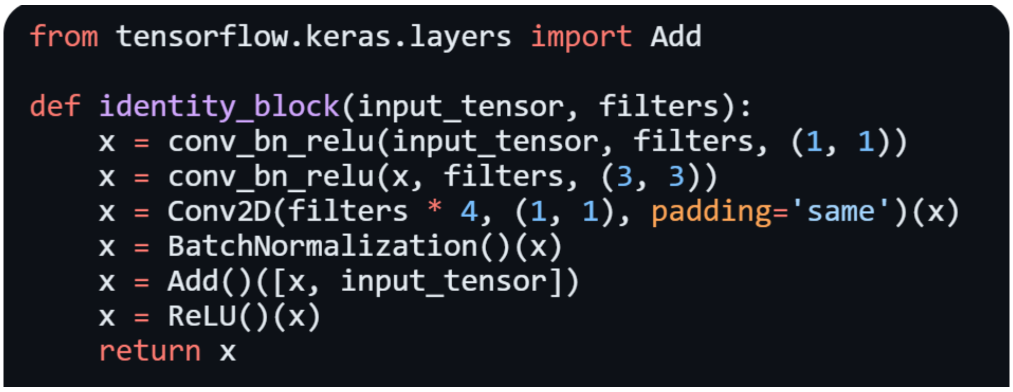
Build the Identity Block:

Next, we will build the Identity block as a function that will:

- **take as inputs:**
  - **a tensor (tensor)**
  - **the number of filters (filters)**
  - **the kernel size (kernel_size)**

- **run:**
  - **apply a 1x1 Conv-BatchNorm-ReLU block to tensor**
  - **apply a 3x3 Conv-BatchNorm-ReLU block**
  - **apply a 1x1 Convolution layer with 4 times the filters filters**
  - **apply a Batch normalization**
  - **add this tensor with tensor**
  - **apply a ReLU activation**

- **return the tensor**



```python
from tensorflow.keras.layers import Add

def identity_block(input_tensor, filters):
    x = conv_bn_relu(input_tensor, filters, (1, 1))
    x = conv_bn_relu(x, filters, (3, 3))
    x = Conv2D(filters * 4, (1, 1), padding='same')(x)
    x = BatchNormalization()(x)
    x = Add()([x, input_tensor])
    x = ReLU()(x)
    return x
```

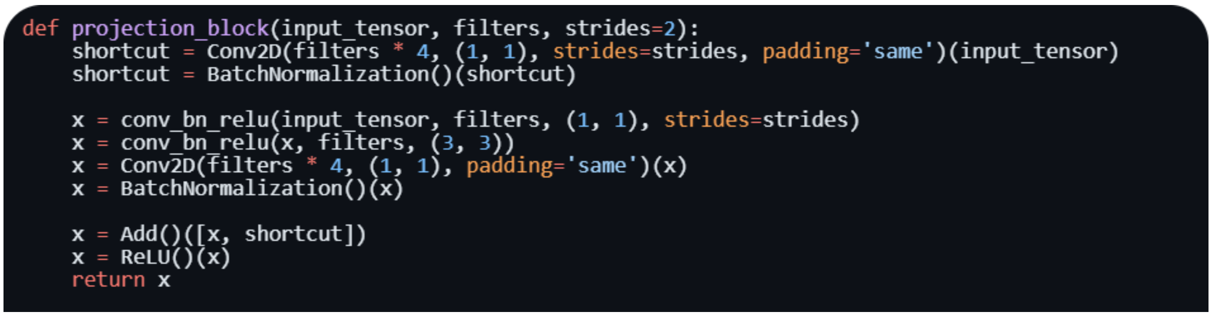
Build the Projection Block:

NOW, we will build the Projection block which is similar to the Identity one.

Remember, this time we need the strides because we want to downsample the tensors at specific blocks

"the 1x1 layers are responsible for reducing and then increasing (restoring) dimensions".

The downsampling at the main stream will take place at the first 1x1 Convolution layer*. The downsampling at the right stream will take place at its Convolution layer.



```python
def projection_block(input_tensor, filters, strides=2):
    shortcut = Conv2D(filters * 4, (1, 1), strides=strides, padding='same')(input_tensor)
    shortcut = BatchNormalization()(shortcut)

    x = conv_bn_relu(input_tensor, filters, (1, 1), strides=strides)
    x = conv_bn_relu(x, filters, (3, 3))
    x = Conv2D(filters * 4, (1, 1), padding='same')(x)
    x = BatchNormalization()(x)

    x = Add()([x, shortcut])
    x = ReLU()(x)
    return x
```

Build the ResNet Block:

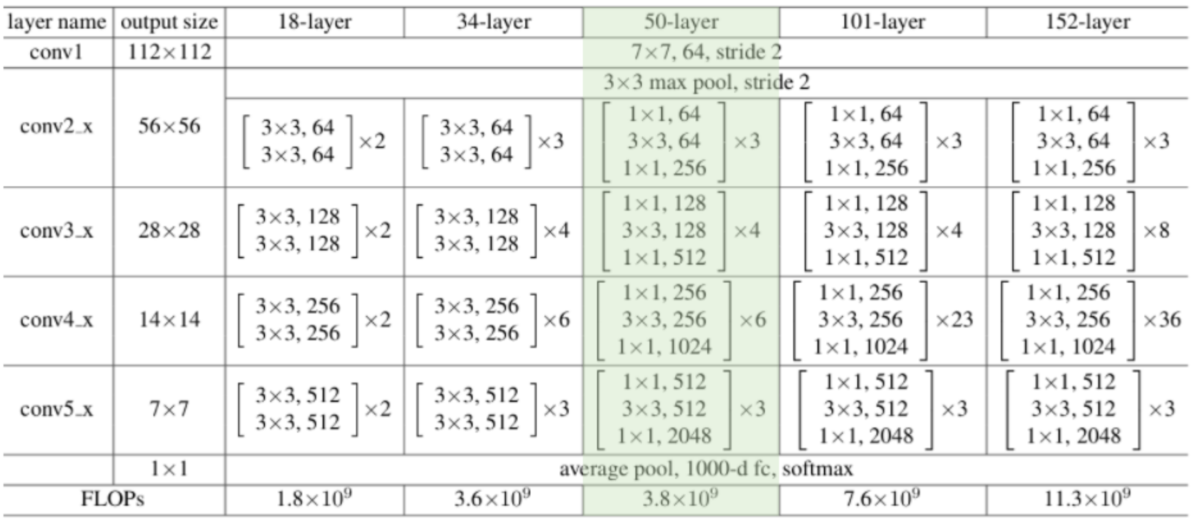
Let's build the ResNet block as a function that will:

- **take as inputs:**
  - **a tensor (x)**
  - **the number of filters (filters)**
  - **the total number of repetitions of internal blocks (reps)**
  - **the strides (strides)**
- **run:**
  - **apply a projection block with strides: strides**
  - **for apply an Identity block for reps - 1 times (the -1 is because the first block was a Convolution one)**

- **return the tensor**

```
def resnet_block(input_tensor, filters, num_blocks, strides=2):
    x = projection_block(input_tensor, filters, strides)
    for _ in range(num_blocks - 1):
        x = identity_block(x, filters)
    return x
```

Build the Final Model:



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64]×2	[3×3, 64; 3×3, 64]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3
conv3_x	28×28	[3×3, 128; 3×3, 128]×2	[3×3, 128; 3×3, 128]×4	[1×1, 128; 3×3, 128; 1×1, 512]×4	[1×1, 128; 3×3, 128; 1×1, 512]×4	[1×1, 128; 3×3, 128; 1×1, 512]×8
conv4_x	14×14	[3×3, 256; 3×3, 256]×2	[3×3, 256; 3×3, 256]×6	[1×1, 256; 3×3, 256; 1×1, 1024]×6	[1×1, 256; 3×3, 256; 1×1, 1024]×23	[1×1, 256; 3×3, 256; 1×1, 1024]×36
conv5_x	7×7	[3×3, 512; 3×3, 512]×2	[3×3, 512; 3×3, 512]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3
	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

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, GlobalAveragePooling2D, Dense

input_tensor = Input(shape=(224, 224, 3))
x = conv_bn_relu(input_tensor, 64, (7, 7), strides=2)
x = resnet_block(x, 64, 3)
x = resnet_block(x, 128, 4, strides=2)
x = resnet_block(x, 256, 6, strides=2)
x = resnet_block(x, 512, 3, strides=2)
x = GlobalAveragePooling2D()(x)
output_tensor = Dense(num_classes, activation='softmax')(x)

model = Model(input_tensor, output_tensor)
```

Print the Model Summary:

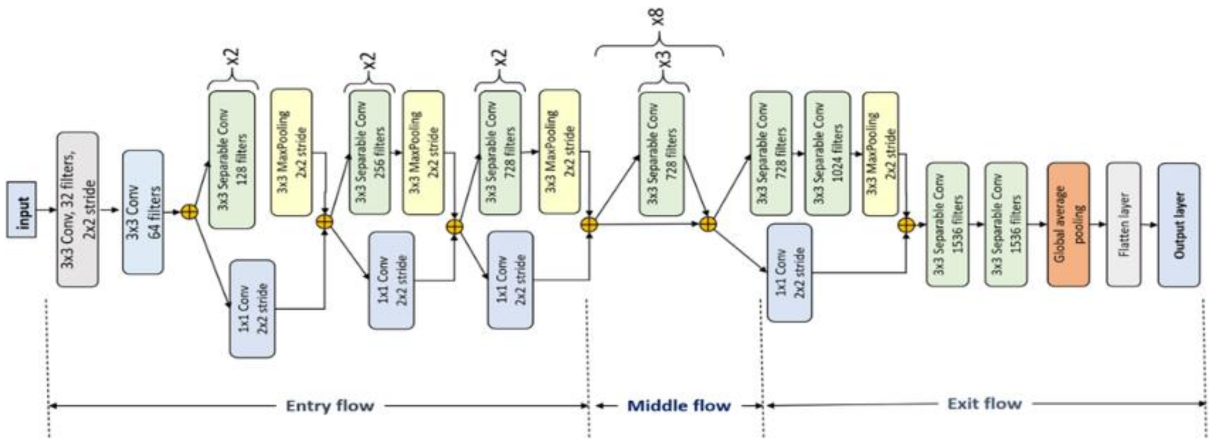
```
model.summary()
```

Plot the ResNet-50 Architecture:

```
from tensorflow.keras.utils import plot_model

plot_model(model, to_file='resnet50.png', show_shapes=True)
```

2. Xception


input
3x3 Conv, 32 filters, 2x2 stride
3x3 Conv 64 filters
x2
3x3 Separable Conv 128 filters
3x3 MaxPooling 2x2 stride
1x1 Conv 2x2 stride
x2
3x3 Separable Conv 256 filters
3x3 MaxPooling 2x2 stride
1x1 Conv 2x2 stride
x2
3x3 Separable Conv 728 filters
3x3 MaxPooling 2x2 stride
1x1 Conv 2x2 stride
x8
x3
3x3 Separable Conv 728 filters
3x3 Separable Conv 728 filters
3x3 Separable Conv 1024 filters
3x3 MaxPooling 2x2 stride
1x1 Conv 2x2 stride
3x3 Separable Conv 1536 filters
3x3 Separable Conv 1536 filters
Global average pooling
Flatten layer
Output layer
Entry flow
Middle flow
Exit flow

Introduction:

Xception (Extreme Inception), developed by Francois Chollet in 2016, extends the Inception architecture. It replaces inception modules with depthwise separable convolutions, improving efficiency and accuracy.

Key Components:

1. **Depthwise Separable Convolutions: Consist of two steps:**
   - **Depthwise Convolution: Applies a single filter to each input channel.**
   - **Pointwise Convolution: Uses a 1x1 convolution to mix information across channels.**
1. **Linear Bottleneck: Reduces dimensionality while maintaining representational power.**

Step-by-Step Implementation with Code:

Input Preprocessing:

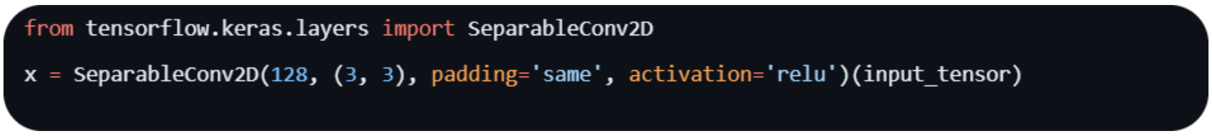
- **Xception model requires input data to be preprocessed using preprocess_input from tensorflow.keras.applications.xception to ensure consistency with the original model.**

```
from tensorflow.keras.applications.xception import preprocess_input

datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
train_generator = datagen.flow_from_directory(
    'data/train',
    target_size=(299, 299),
    batch_size=32,
    class_mode='categorical'
```

Depthwise Separable Convolutions:

- **Use the SeparableConv2D layer to perform depthwise separable convolutions.**

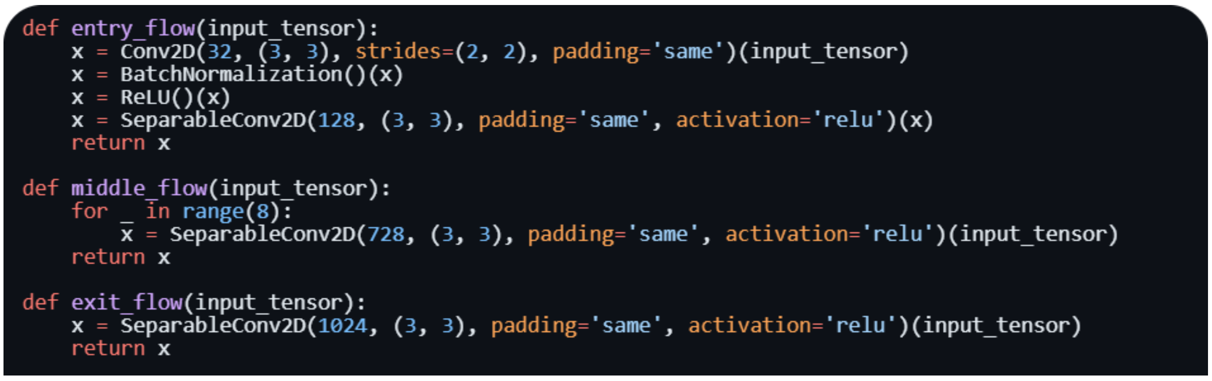


```
from tensorflow.keras.layers import SeparableConv2D

x = SeparableConv2D(128, (3, 3), padding='same', activation='relu')(input_tensor)
```

Entry, Middle, and Exit Flows:

- **Entry Flow: Applies initial convolution layers with batch normalization and ReLU activation, followed by depthwise separable convolutions.**
- **Middle Flow: Repeats a set of 8 depthwise separable convolutions to capture complex features.**
- **Exit Flow: Final layer of convolutions to capture high-level features before classification.**

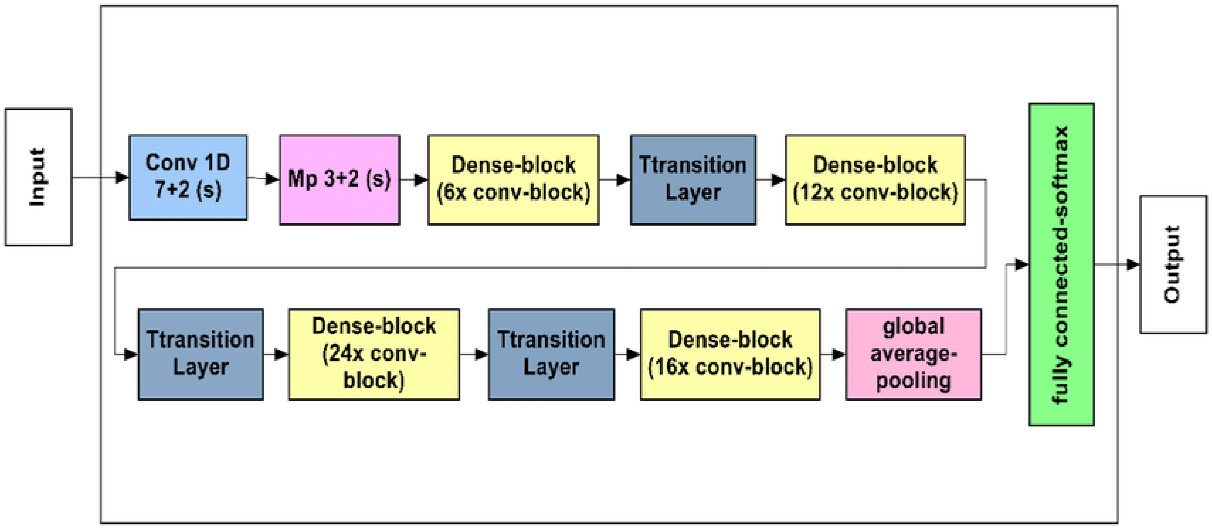


```
def entry_flow(input_tensor):
    x = Conv2D(32, (3, 3), strides=(2, 2), padding='same')(input_tensor)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = SeparableConv2D(128, (3, 3), padding='same', activation='relu')(x)
    return x

def middle_flow(input_tensor):
    for _ in range(8):
        x = SeparableConv2D(728, (3, 3), padding='same', activation='relu')(input_tensor)
    return x

def exit_flow(input_tensor):
    x = SeparableConv2D(1024, (3, 3), padding='same', activation='relu')(input_tensor)
    return x
```

3. DenseNet121



Introduction:

DenseNet, introduced by Huang et al. in 2017, connects each layer to every other layer, ensuring maximum information flow and feature reuse. This reduces the number of parameters while maintaining accuracy.

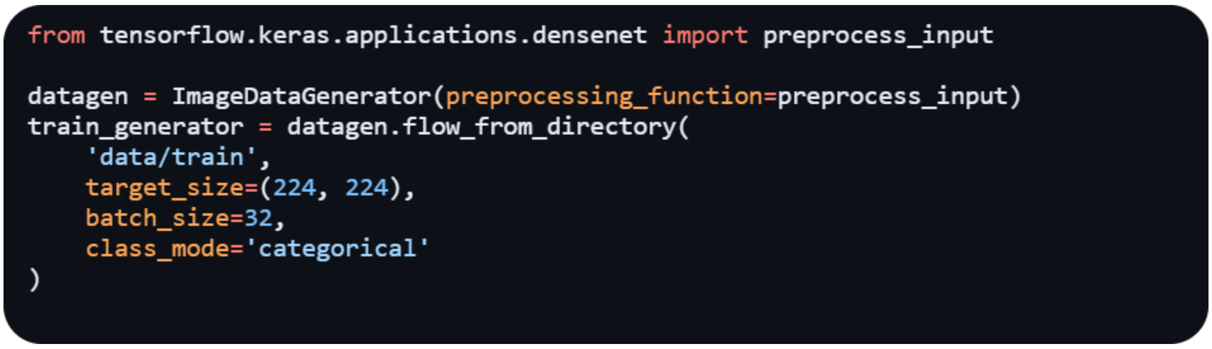
Key Components:

1. **Dense Blocks: Layers are densely connected. The output of each layer is concatenated with all previous layer outputs.**
2. **Transition Layers: Reduce feature map dimensions using 1x1 convolutions and pooling layers.**
3. **Growth Rate (k): Determines the number of filters added per layer.**

Step-by-Step Implementation with Code:

Input Preprocessing:

- **Similar to Xception, DenseNet uses a specific preprocessing function.**

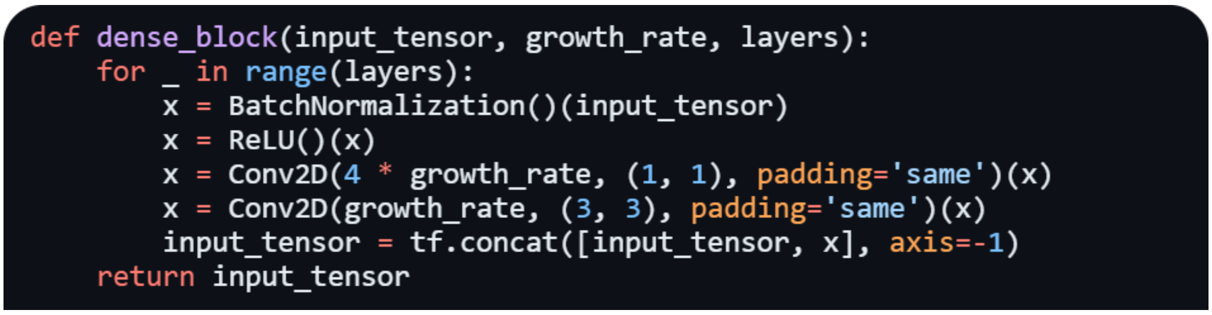


```
from tensorflow.keras.applications.densenet import preprocess_input

datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
train_generator = datagen.flow_from_directory(
    'data/train',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'
)
```

Dense Block:

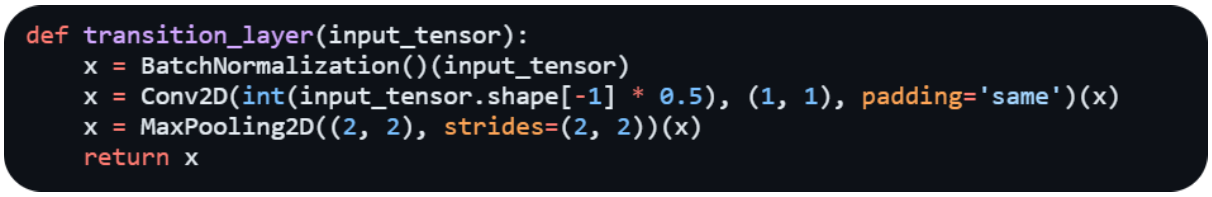
- **A dense block includes multiple layers where each layer receives input from all previous layers.**



```
def dense_block(input_tensor, growth_rate, layers):
    for _ in range(layers):
        x = BatchNormalization()(input_tensor)
        x = ReLU()(x)
        x = Conv2D(4 * growth_rate, (1, 1), padding='same')(x)
        x = Conv2D(growth_rate, (3, 3), padding='same')(x)
        input_tensor = tf.concat([input_tensor, x], axis=-1)
    return input_tensor
```

Transition Layer:

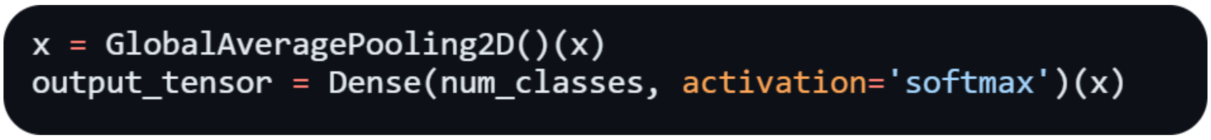
- **Transition layers help reduce the dimensionality of the feature maps.**



```
def transition_layer(input_tensor):
    x = BatchNormalization()(input_tensor)
    x = Conv2D(int(input_tensor.shape[-1] * 0.5), (1, 1), padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)
    return x
```

Final Layers:

- **After the dense blocks, apply global average pooling and a final softmax layer for classification.**



```
x = GlobalAveragePooling2D()(x)
output_tensor = Dense(num_classes, activation='softmax')(x)
```

References

1. **He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv:1512.03385*.**
2. **Chollet, F. (2016). Xception: Deep Learning with Depthwise Separable Convolutions. *arXiv:1610.02357*.**
3. **Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. *arXiv:1608.06993*.**

Face Classification

Dataset Preparation

1. **Preprocessing:**
   - **Resize images to match model input dimensions.**
   - **Normalize pixel values to [0, 1].**

1. **Data Augmentation:**
   - **Random flips, rotations, zooms, and translations.**
1. **Split:**
   - **Train/validation/test split ensuring class balance.**

Training (ResNet50)

Loss Function: Use Categorical Crossentropy for multi-class classification.

Optimizer: Adam optimizer with an initial learning rate of 0.001.

Callbacks:

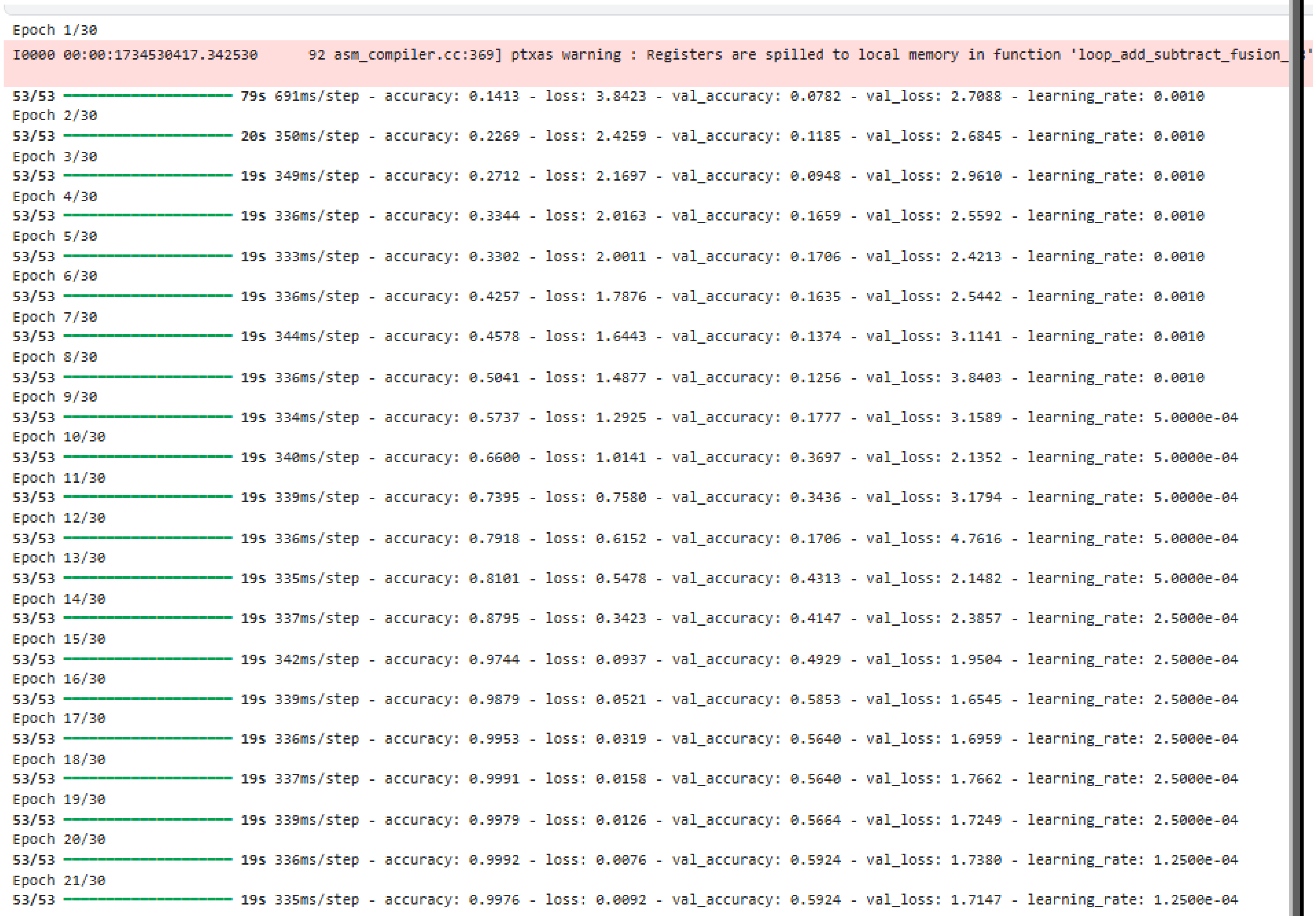
- **Early stopping: Monitors validation loss and stops training if it does not improve.**
- **ReduceLROnPlateau: Reduces the learning rate when validation performance plateaus.**

Epochs and Batch Size:

- **Train the model for 50 epochs with a batch size of 32.**

Validation:

- **Use a validation split of 20% from the training data.**

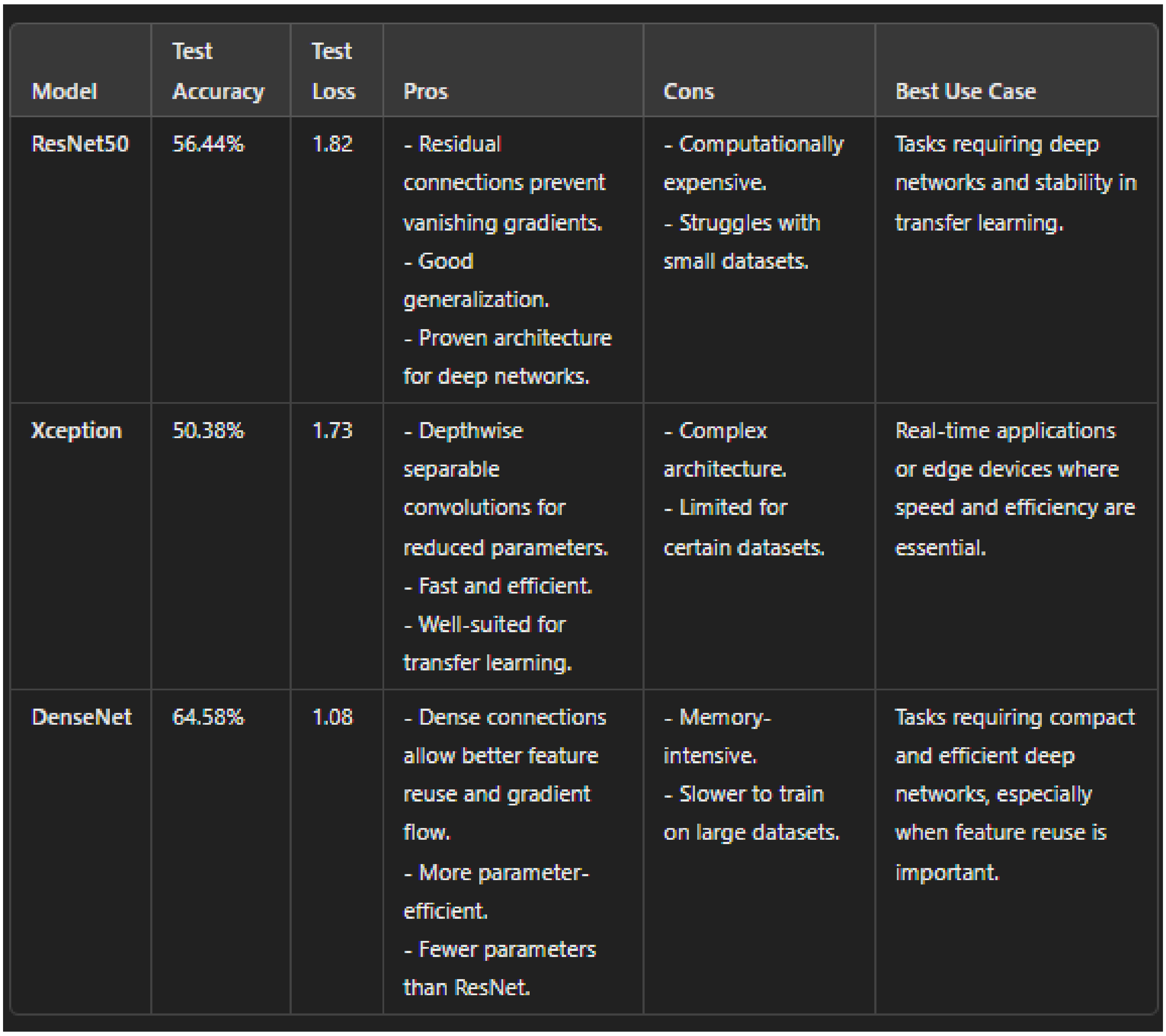


```
Epoch 1/30
I0000 00:00:1734530417.342530     92 asm_compiler.cc:369] ptxas warning : Registers are spilled to local memory in function 'loop_add_subtract_fusion_
53/53 ━━━━━━━━━━━━━━━━━━━━ 79s 691ms/step - accuracy: 0.1413 - loss: 3.8423 - val_accuracy: 0.0782 - val_loss: 2.7088 - learning_rate: 0.0010
Epoch 2/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 20s 350ms/step - accuracy: 0.2269 - loss: 2.4259 - val_accuracy: 0.1185 - val_loss: 2.6845 - learning_rate: 0.0010
Epoch 3/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 349ms/step - accuracy: 0.2712 - loss: 2.1697 - val_accuracy: 0.0948 - val_loss: 2.9610 - learning_rate: 0.0010
Epoch 4/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.3344 - loss: 2.0163 - val_accuracy: 0.1659 - val_loss: 2.5592 - learning_rate: 0.0010
Epoch 5/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 333ms/step - accuracy: 0.3302 - loss: 2.0011 - val_accuracy: 0.1706 - val_loss: 2.4213 - learning_rate: 0.0010
Epoch 6/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.4257 - loss: 1.7876 - val_accuracy: 0.1635 - val_loss: 2.5442 - learning_rate: 0.0010
Epoch 7/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 344ms/step - accuracy: 0.4578 - loss: 1.6443 - val_accuracy: 0.1374 - val_loss: 3.1141 - learning_rate: 0.0010
Epoch 8/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.5041 - loss: 1.4877 - val_accuracy: 0.1256 - val_loss: 3.8403 - learning_rate: 0.0010
Epoch 9/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 334ms/step - accuracy: 0.5737 - loss: 1.2925 - val_accuracy: 0.1777 - val_loss: 3.1589 - learning_rate: 5.0000e-04
Epoch 10/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 340ms/step - accuracy: 0.6600 - loss: 1.0141 - val_accuracy: 0.3697 - val_loss: 2.1352 - learning_rate: 5.0000e-04
Epoch 11/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 339ms/step - accuracy: 0.7395 - loss: 0.7580 - val_accuracy: 0.3436 - val_loss: 3.1794 - learning_rate: 5.0000e-04
Epoch 12/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.7918 - loss: 0.6152 - val_accuracy: 0.1706 - val_loss: 4.7616 - learning_rate: 5.0000e-04
Epoch 13/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 335ms/step - accuracy: 0.8101 - loss: 0.5478 - val_accuracy: 0.4313 - val_loss: 2.1482 - learning_rate: 5.0000e-04
Epoch 14/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 337ms/step - accuracy: 0.8795 - loss: 0.3423 - val_accuracy: 0.4147 - val_loss: 2.3857 - learning_rate: 2.5000e-04
Epoch 15/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 342ms/step - accuracy: 0.9744 - loss: 0.0937 - val_accuracy: 0.4929 - val_loss: 1.9504 - learning_rate: 2.5000e-04
Epoch 16/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 339ms/step - accuracy: 0.9879 - loss: 0.0521 - val_accuracy: 0.5853 - val_loss: 1.6545 - learning_rate: 2.5000e-04
Epoch 17/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.9953 - loss: 0.0319 - val_accuracy: 0.5640 - val_loss: 1.6959 - learning_rate: 2.5000e-04
Epoch 18/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 337ms/step - accuracy: 0.9991 - loss: 0.0158 - val_accuracy: 0.5640 - val_loss: 1.7662 - learning_rate: 2.5000e-04
Epoch 19/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 339ms/step - accuracy: 0.9979 - loss: 0.0126 - val_accuracy: 0.5664 - val_loss: 1.7249 - learning_rate: 2.5000e-04
Epoch 20/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 336ms/step - accuracy: 0.9992 - loss: 0.0076 - val_accuracy: 0.5924 - val_loss: 1.7380 - learning_rate: 1.2500e-04
Epoch 21/30
53/53 ━━━━━━━━━━━━━━━━━━━━ 19s 335ms/step - accuracy: 0.9976 - loss: 0.0092 - val_accuracy: 0.5924 - val_loss: 1.7147 - learning_rate: 1.2500e-04
```

Evaluation

1. **Metrics:**
   - **Calculate Accuracy, Precision, Recall, and F1-Score.**
   - **Generate ROC-AUC and Precision-Recall curves.**
1. **Confusion Matrix:**
   - **Visualize model predictions to identify class-wise performance.**
1. **Model Comparison:**
   - **Compare validation accuracy and loss across ResNet50, Xception, and DenseNet121.**

Comparison



Model	Test Accuracy	Test Loss	Pros	Cons	Best Use Case
ResNet50	56.44%	1.82	- Residual connections prevent vanishing gradients. - Good generalization. - Proven architecture for deep networks.	- Computationally expensive. - Struggles with small datasets.	Tasks requiring deep networks and stability in transfer learning.
Xception	50.38%	1.73	- Depthwise separable convolutions for reduced parameters. - Fast and efficient. - Well-suited for transfer learning.	- Complex architecture. - Limited for certain datasets.	Real-time applications or edge devices where speed and efficiency are essential.
DenseNet	64.58%	1.08	- Dense connections allow better feature reuse and gradient flow. - More parameter-efficient. - Fewer parameters than ResNet.	- Memory-intensive. - Slower to train on large datasets.	Tasks requiring compact and efficient deep networks, especially when feature reuse is important.

1. ResNet50 (Residual Networks)

Accuracy: 56.44%
Loss: 1.82

Pros:

- **Residual Connections: ResNet introduces skip (or residual) connections that allow the network to skip certain layers. This helps prevent the vanishing gradient problem, allowing the network to be much deeper without losing information during training.**
- **Good Generalization: Due to the residual connections, ResNet has improved training stability and generalization, especially for very deep networks.**
- **Proven Architecture: ResNet50 is a well-established architecture that works well in a wide variety of tasks, especially in vision-related tasks like object classification.**

Cons:

- **Heavy Computation: While ResNet is very deep, it can be computationally expensive due to the large number of layers and parameters.**
- **Not the Best for Smaller Datasets: When dealing with smaller datasets, ResNet may struggle to achieve optimal performance without additional techniques like fine-tuning.**

Best Use Case: ResNet50 is ideal for tasks that require deep networks with a large number of layers and is known for its stability in transfer learning tasks.

---

2. Xception

(Extreme Inception)

Accuracy: 50.38%
Loss: 1.73

Pros:

- **Depthwise Separable Convolutions: Xception uses depthwise separable convolutions, which break down the traditional convolutions into two parts (depthwise convolution followed by pointwise convolution). This significantly reduces the number of parameters and computational load.**
- **Efficient and Fast: Due to its efficient use of parameters, Xception tends to be faster and more computationally efficient compared to traditional models like VGG and ResNet.**
- **Better for Transfer Learning: Xception performs well when fine-tuned with pre-trained weights, especially when using fewer parameters.**

Cons:

- **Relatively Complex: Xception's architecture is more complex compared to simpler models like VGG or ResNet, which may require more expertise to fine-tune effectively.**
- **Limited to Certain Datasets: While it works well for general image recognition tasks, Xception might not be the best architecture for all use cases.**

Best Use Case: Xception works well for tasks where computational efficiency and speed are essential, such as real-time applications or edge devices.

3. DenseNet

(Densely Connected Convolutional Networks)

Accuracy: 64.58%
Loss: 1.08

Pros:

- **Dense Connections: In DenseNet, each layer is connected to every other layer in a feed-forward manner, which allows for better feature reuse and improved gradient flow. This makes DenseNet more parameter-efficient compared to other architectures.**
- **Better Feature Propagation: The dense connections improve feature propagation through the network, resulting in more compact and expressive representations of the input data.**
- **Fewer Parameters: Despite its depth, DenseNet typically has fewer parameters compared to traditional models like ResNet or VGG, making it more memory efficient.**

Cons:

- **Computationally Intensive: DenseNet requires more memory and computation to store intermediate features and manage connections between layers.**
- **Training Time: DenseNet can be slower to train, especially when the dataset is large, because of the complex connections between layers.**

Best Use Case: DenseNet is particularly effective when you want to work with datasets that contain complex patterns or when you're interested in maximizing feature reuse. It’s also ideal for tasks requiring better generalization and efficiency in deep networks.

Summary and Conclusion:

- **ResNet50 is known for its ability to train very deep networks, making it a good choice when depth is essential. It is a well-balanced choice for many tasks but can be computationally expensive.**
- **Xception is a more efficient model compared to ResNet in terms of parameters and computation. It performs well in transfer learning scenarios and is ideal when speed and efficiency are key.**
- **DenseNet performs the best in this case, achieving the highest accuracy (64.58%). Its dense connections allow it to effectively utilize features from earlier layers, improving the quality of learned representations and reducing the number of parameters. This architecture is best suited for applications requiring compact and effective deep networks.**

Thus, DenseNet stands out as the best choice in terms of performance in this comparison, especially for tasks that benefit from dense feature reuse and efficiency. However, the ideal model depends on your specific use case—ResNet50 for deep models, Xception for efficiency, or DenseNet for maximizing feature utilization.