Transfer Learning Models for Convolutional Neural Network (CNNs):

1. VGG16
2. MobileNet
3. DenseNet
4. Inception
5. ResNet
6. EfficientNet
7. NASNet

Transfer learning is a machine learning technique where a model trained on one task is used as a starting point for a related task. In the context of CNNs, this involves leveraging the knowledge learned from a large dataset to solve a new problem with limited data.

```
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Class weight calculation
from sklearn.utils.class_weight import compute_class_weight

# Keras library
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dro
pout, GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image import ImageDataGener
ator
from keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from keras import regularizers
from keras.callbacks import ReduceLROnPlateau

# Different CNN Model
```

```
from tensorflow.keras.applications import VGG16, ResNet50, Incept
ionV3, MobileNetV2, DenseNet121

# To chain two different data augmented images for training
from itertools import chain

#  Distributed Computing
import tensorflow as tf

import warnings
warnings.filterwarnings("ignore")
```

Why Transfer Learning for CNNs?

- **Limited Data:** CNNs require large datasets for optimal performance. Transfer learning allows us to train effective models with smaller datasets.
- **Computational Efficiency:** Pre-trained models can be fine-tuned more efficiently than training a model from scratch.
- **Improved Accuracy:** Transfer learning often leads to better performance, especially for tasks with similar characteristics to the original training data.

Common Transfer Learning Approaches for CNNs

1. **Feature Extraction:**
    - **Frozen Layers:** The initial layers of a pre-trained CNN are frozen, and only the final layers are trained on the new task. This assumes that the early layers extract generic features that are useful for a variety of tasks.
    - **Fine-tuning:** Some layers are fine-tuned along with the new layers. This allows the network to adapt to the specific characteristics of the new task.
2. **Feature Fusion:**
    - **Multiple Models:** Features extracted from multiple pre-trained models are combined to create a more robust representation.
    - **Early Fusion:** Features are combined at an early stage, typically before the fully connected layers.
    - **Late Fusion:** Features are combined at a later stage, after the fully connected layers.

Popular Pre-trained CNN Architectures

- **VGGNet:** Known for its depth and simplicity.
- **ResNet:** Uses residual connections to overcome the vanishing gradient problem.
- **InceptionNet:** Combines different-sized convolutions to extract features at multiple scales.
- **MobileNet:** Designed for efficient inference on mobile devices.
- **EfficientNet:** A compound scaling method that balances depth, width, and resolution.

Best Practices for Transfer Learning

- **Choose a suitable pre-trained model:** Consider the similarity between the original task and your new task.
- **Experiment with different layers to freeze or fine-tune:** Start with freezing more layers and gradually unfreeze them as needed.
- **Adjust the learning rate:** Use a lower learning rate for fine-tuning to avoid overfitting.
- **Data augmentation:** Augment your training data to increase its diversity and prevent overfitting.
- **Regularization techniques:** Employ techniques like dropout or L1/L2 regularization to prevent overfitting.

By effectively applying transfer learning techniques, you can significantly accelerate the development of CNN models for various tasks, even with limited data.

```
BATCH_SIZE = 48

image_height = 299
image_width = 299

# Data agumentation and pre-processing using tensorflow
data_generator_1 = ImageDataGenerator(
                        rescale=1./255,
                        rotation_range=5,
                        width_shift_range=0.05,
                        height_shift_range=0.05,
                        shear_range=0.05,
                        zoom_range=0.05,
                        brightness_range = [0.95,1.05],
                        horizontal_flip=False,
                        vertical_flip=False,
                        fill_mode='nearest'
                    )
print('Data Augmentation 1 was created')

data_generator_2 = ImageDataGenerator(
                        rescale=1./255,
                        rotation_range=10,
                        width_shift_range=0.1,
                        height_shift_range=0.1,
                        shear_range=0.1,
                        zoom_range=0.1,
                        brightness_range = [0.9,1.1],
```

```
                    horizontal_flip=False,
                    vertical_flip=False,
                    fill_mode='nearest'
                )
print('Data Augmentation 2 was created')

data_generator_3 = ImageDataGenerator (rescale=1./255)
```

```
Data Augmentation 1 was created
Data Augmentation 2 was created
```

```
train_generator1 = data_generator_1.flow_from_directory(
    directory = "/kaggle/input/brain-tumor-classification-mri/Training",
    color_mode = "rgb",
    target_size = (image_height, image_width), # image height , image width
    class_mode = "categorical",
    batch_size = BATCH_SIZE,
    shuffle = True,
    seed = 42)
print('Data Augmentation 1 was used to generate train data set\n')
```

```
Found 2870 images belonging to 4 classes.
Data Augmentation 1 was used to generate train data set
```

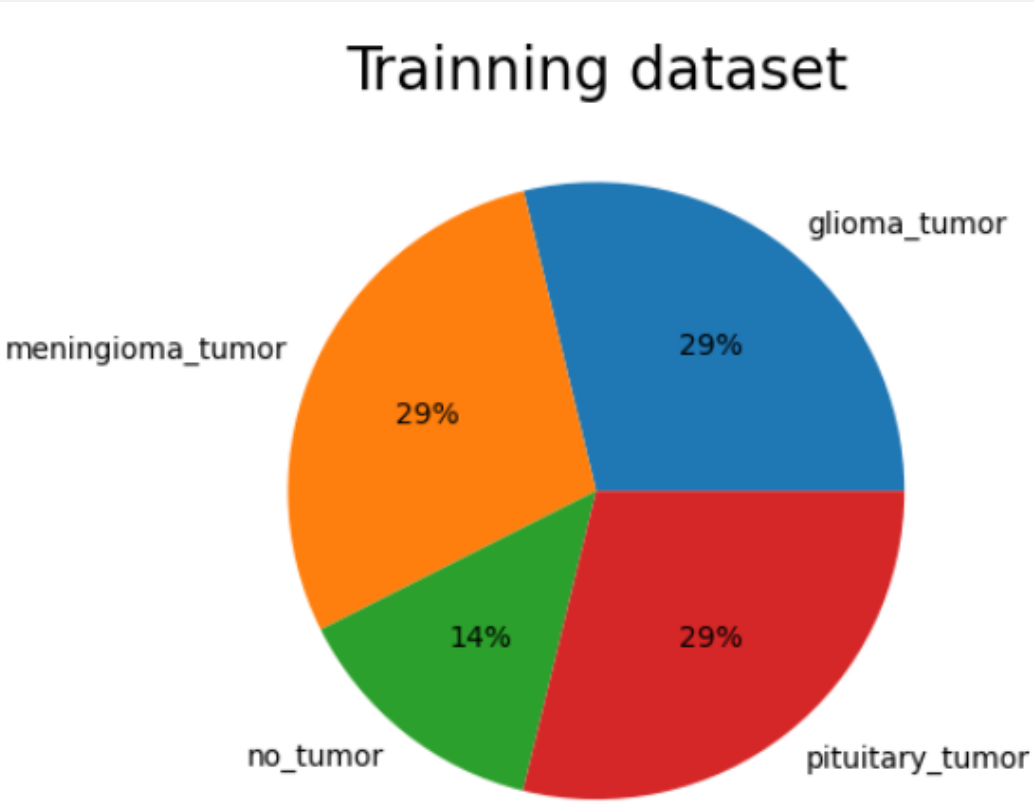
```
test_generator = data_generator_3.flow_from_directory(
    directory = "/kaggle/input/brain-tumor-classification-mri/Testing",
    color_mode = "rgb",
    target_size = (image_height, image_width), # image height , image width
    class_mode = "categorical",
    batch_size = BATCH_SIZE,
    shuffle = True,
    seed = 42)
```

```
Found 394 images belonging to 4 classes.
```

```
dict_class = train_generator1.class_indices
print('Dictionary: {}'.format(dict_class))
class_names = list(dict_class.keys())  # storing class/breed names in a list
print('Class labels: {}'.format(class_names))
```

```
Dictionary: {'glioma_tumor': 0, 'meningioma_tumor': 1, 'no_tumor': 2, 'pituitary_tumo
r': 3}
Class labels: ['glioma_tumor', 'meningioma_tumor', 'no_tumor', 'pituitary_tumor']
```

```
frequency = np.unique(train_generator1.classes, return_counts=True)
plt.title("Trainning dataset", fontsize='20')
plt.pie(frequency[1], labels = class_names, autopct='%1.0f%%');
```



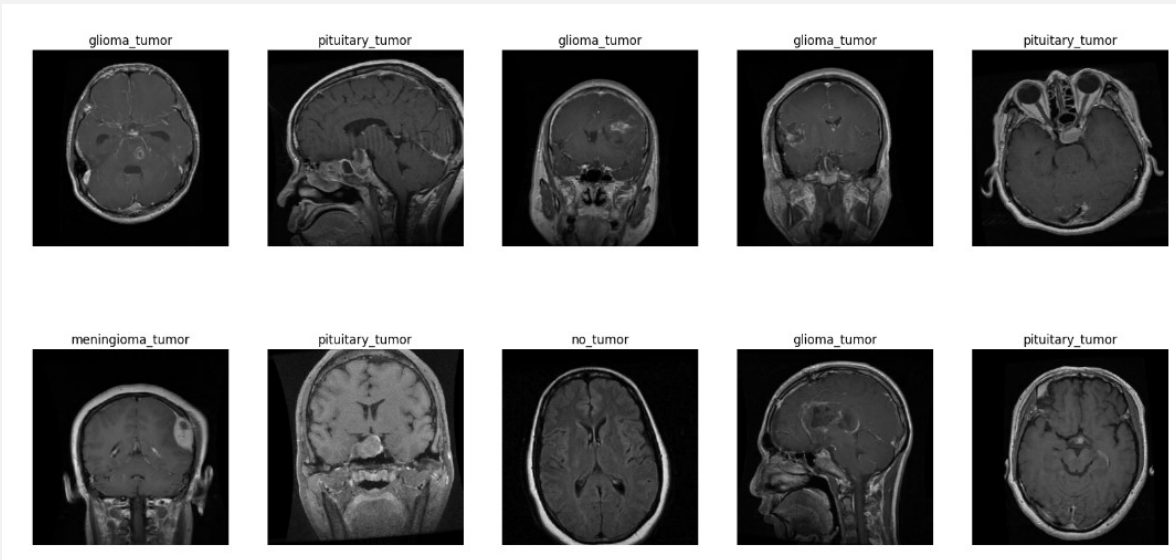
```
# Dataset characteristics
print("Dataset Characteristics of Train Data Set:\n")
print("Number of images:", len(train_generator1.classes))
print("Number of glioma_tumor images:", len([label for label in train_
generator1.classes if label == 0]))
print("Number of meningioma_tumor images:", len([label for label in
train_generator1.classes if label == 1]))
print("Number of no_tumor images:", len([label for label in train_gen
erator1.classes if label == 2]))
print("Number of pituitary_tumor images:", len([label for label in trai
n_generator1.classes if label == 3]))
print()

# Dataset characteristics
print("Dataset Characteristics of Test Data Set:\n")
print("Number of images:", len(test_generator.classes))
print("Number of glioma_tumor images:", len([label for label in test_g
enerator.classes if label == 0]))
print("Number of meningioma_tumor images:", len([label for label in
test_generator.classes if label == 1]))
print("Number of no_tumor images:", len([label for label in test_gene
rator.classes if label == 2]))
print("Number of pituitary_tumor images:", len([label for label in test
_generator.classes if label == 3]))
print()
```

```
Dataset Characteristics of Train Data Set:

Number of images: 2870
Number of glioma_tumor images: 826
Number of meningioma_tumor images: 822
Number of no_tumor images: 395
Number of pituitary_tumor images: 827

Dataset Characteristics of Test Data Set:

Number of images: 394
Number of glioma_tumor images: 100
Number of meningioma_tumor images: 115
Number of no_tumor images: 105
Number of pituitary_tumor images: 74
```

```
class_weights = compute_class_weight(class_weight = "balanced", classes= np.unique(train_generator1.classes), y= train_generator1.classes)
class_weights = dict(zip(np.unique(train_generator1.classes), class_weights))
class_weights
```

```
{0: 0.8686440677966102,
 1: 0.8728710462287105,
 2: 1.8164556962025316,
 3: 0.8675937122128174}
```

```
print('Train image data from Data Augmentation 1')
img, label = next(train_generator1)
# print(len(label))

plt.figure(figsize=[20, 15])
for i in range(15):
    plt.subplot(3, 5, i+1)
    plt.imshow(img[i])
    plt.axis('off')
    plt.title(class_names[np.argmax(label[i])])
plt.show()
```



Convolutional neural networks (CNNs)

```
# Define the epochs for training
EPOCHS = 2

# Define the number of GPUs to use
num_gpus = 2

# Merge augmented image data for training
# merged_train_generator = chain(train_generator1, train_generator2
, train_generator3)

# Define early stopping criteria
early_stopping = EarlyStopping(monitor='val_accuracy', patience=2,
verbose=1, restore_best_weights=True)

# Define the ReduceLROnPlateau callback
reduce_lr = ReduceLROnPlateau(monitor='val_accuracy', factor=0.00
1, patience=10, verbose=1)

# For development purpose, we first limit the train data set to the or
iginal image data set
# train_data = merged_train_generator
# train_data = train_generator1
train_data = train_generator1
# train_data = test_generator
```

VGG16 is a convolutional neural network (CNN) architecture that was introduced in 2014. It was developed by researchers from the University of Oxford and is known for its simplicity and effectiveness in image classification tasks.

Key features of VGG16:

- **Simple Architecture:** VGG16 uses a stack of 16 convolutional layers, followed by three fully connected layers and a final softmax layer for classification.
- **Small Filters:** The convolutional layers in VGG16 use small 3x3 filters, which are repeated multiple times to increase the depth of the network.

- **Uniform Stride:** The convolutional layers use a uniform stride of 1, which means that the filters are applied to every pixel in the input image.
- **Max Pooling:** After each block of convolutional layers, a max pooling layer is used to reduce the spatial dimensions of the feature maps.

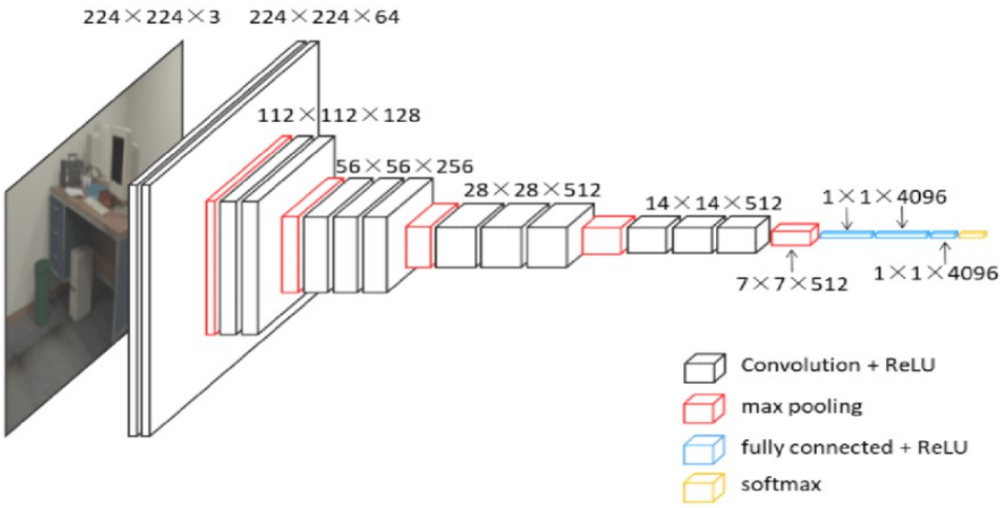
Benefits of VGG16:

- **Simplicity:** VGG16's architecture is relatively simple and easy to understand, making it a popular choice for researchers and practitioners.
- **Effectiveness:** VGG16 achieved state-of-the-art performance on the ImageNet classification dataset when it was introduced, demonstrating its effectiveness in image classification tasks.
- **Pre-trained Models:** Pre-trained VGG16 models are widely available, which can be used as a starting point for transfer learning tasks in other domains.

Applications of VGG16:

- **Image Classification:** VGG16 is commonly used for image classification tasks, such as object recognition, scene classification, and facial recognition.
- **Transfer Learning:** VGG16 can be used as a feature extractor for transfer learning tasks, where the pre-trained model is fine-tuned on a smaller dataset to solve a related task.
- **Object Detection:** VGG16 has been used as a component of object detection architectures, such as Faster R-CNN and SSD.

In summary, VGG16 is a powerful and versatile CNN architecture that has made significant contributions to the field of computer vision. Its simplicity, effectiveness, and availability of pre-trained models make it a popular choice for various image classification and transfer learning tasks.



```
# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():

    # Load the pre-trained VGG16 model without the top classificatio
n layer
    base_model_VGG16 = VGG16(weights='imagenet', include_top=Fal
se, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_VGG16.layers:
        layer.trainable = False

    # Create a new model and add the VGG16 base model
    model_VGG16 = Sequential()
    model_VGG16.add(base_model_VGG16)

    # Add a fully connected layer and output layer for classification
    model_VGG16.add(GlobalAveragePooling2D())
    model_VGG16.add(Dense(128, activation='relu',kernel_regularizer
=regularizers.l2(0.001)))
    model_VGG16.add(Dropout(0.4))
    model_VGG16.add(Dense(64, activation='relu',kernel_regularizer=r
egularizers.l2(0.001)))
    model_VGG16.add(Dropout(0.2))
    model_VGG16.add(Dense(4, activation='softmax'))

    # Model summary
    print("Model Summary (VGG16):")
    model_VGG16.summary()
    print()

    # Compile the model
    model_VGG16.compile(optimizer='adam', loss='categorical_crosse
ntropy', metrics=['accuracy'])

    # Train the model
```

```
history_VGG16 = model_VGG16.fit(train_data, epochs=EPOCHS, validation_data=test_generator, callbacks=[early_stopping], class_weight=class_weights)

# Validate the model
val_loss_VGG16, val_accuracy_VGG16 = model_VGG16.evaluate(test_generator, steps=len(test_generator))
print(f'Validation Loss: {val_loss_VGG16:.4f}')
print(f'Validation Accuracy: {val_accuracy_VGG16:.4f}')
```

Layer (type)	Output Shape	Param #
vgg16 (Functional)	?	14,714,688
global_average_pooling2d (GlobalAveragePooling2D)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)
dropout (Dropout)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)
dropout_1 (Dropout)	?	0 (unbuilt)
dense_2 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 88s 1s/step - accuracy: 0.2499 - loss: 1.6866 - val_accuracy: 0.3604 - val_loss: 1.4790
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 71s 1s/step - accuracy: 0.4193 - loss: 1.4111 - val_accuracy: 0.4873 - val_loss: 1.3680
Restoring model weights from the end of the best epoch: 2.
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 304ms/step - accuracy: 0.4723 - loss: 1.3854
Validation Loss: 1.3712
Validation Accuracy: 0.5025
```

MobileNetV2 is a convolutional neural network (CNN) architecture designed specifically for mobile and embedded vision applications. It was introduced in 2018 and is known for its high efficiency and accuracy.

Key features of MobileNetV2:

- **Inverted Residual Blocks:** MobileNetV2 uses inverted residual blocks as the building blocks of its architecture. These blocks consist of a 1x1 expansion layer, a depthwise separable convolution, a 1x1 projection layer, and a residual connection.
- **Depthwise Separable Convolutions:** Depthwise separable convolutions are a key component of MobileNetV2. They decompose the standard convolution operation into two separate operations: depthwise convolutions and pointwise convolutions. This decomposition significantly reduces the number of parameters and computations.

- **Pointwise Convolutions:** Pointwise convolutions are used to combine the features produced by the depthwise convolutions. They are equivalent to 1x1 convolutions.
- **ReLU6 Activation:** MobileNetV2 uses the ReLU6 activation function, which is a variant of the ReLU function with a maximum value of 6. This helps to prevent the vanishing gradient problem.

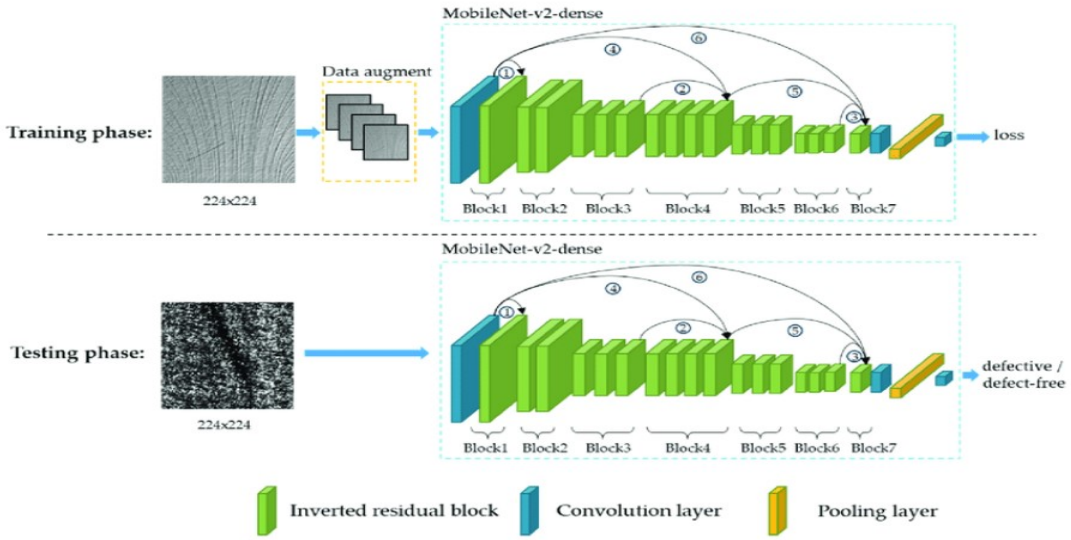
Benefits of MobileNetV2:

- **High Efficiency:** MobileNetV2 is designed to be highly efficient, making it suitable for mobile and embedded devices with limited computational resources.
- **High Accuracy:** Despite its efficiency, MobileNetV2 achieves state-of-the-art accuracy on a variety of image classification benchmarks.
- **Transfer Learning:** MobileNetV2 can be used for transfer learning, where the pre-trained model is fine-tuned on a smaller dataset to solve a related task.

Applications of MobileNetV2:

- **Mobile Vision:** MobileNetV2 is widely used in mobile vision applications, such as object detection, image classification, and facial recognition.
- **Embedded Vision:** MobileNetV2 can be deployed on embedded devices, such as drones and robots, for tasks like real-time object tracking and scene understanding.
- **Edge Computing:** MobileNetV2 is well-suited for edge computing applications, where models are deployed on devices at the edge of the network to reduce latency and bandwidth requirements.

In summary, MobileNetV2 is a highly efficient and accurate CNN architecture that is specifically designed for mobile and embedded vision applications. Its inverted residual blocks, depthwise separable convolutions, and pointwise convolutions make it a popular choice for developers working on resource-constrained devices.



```
%%time

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():

    # Load the pre-trained MobileNetV2 model without the top classifi
cation layer
    base_model_MobileNet = MobileNetV2(weights='imagenet', includ
e_top=False, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_MobileNet.layers:
        layer.trainable = False

    # Create a new model and add the MobileNetV2 base model
    model_MobileNet = Sequential()
    model_MobileNet.add(base_model_MobileNet)

    # Add a global average pooling layer and output layer for classific
ation
    model_MobileNet.add(GlobalAveragePooling2D())
    model_MobileNet.add(Dense(128, activation='relu', kernel_regulari
zer=regularizers.l2(0.001)))
    model_MobileNet.add(Dropout(0.4))
    model_MobileNet.add(Dense(64, activation='relu', kernel_regulariz
er=regularizers.l2(0.001)))
    model_MobileNet.add(Dropout(0.2))
    model_MobileNet.add(Dense(4, activation='softmax'))

    # Model summary
    print("Model Summary (MobileNetV2):")
    model_MobileNet.summary()
    print()

    # Compile the model
    model_MobileNet.compile(optimizer='adam', loss='categorical_cro
ssentropy', metrics=['accuracy'])
```

```
    # Train the model
    history_MobileNet = model_MobileNet.fit(train_data, epochs=EPOCHS, validation_data=test_generator, callbacks=[early_stopping], class_weight=class_weights)

    # Validate the model
    val_loss_MobileNet, val_accuracy_MobileNet = model_MobileNet.evaluate(test_generator, steps=len(test_generator))
    print(f'Validation Loss: {val_loss_MobileNet:.4f}')
    print(f'Validation Accuracy: {val_accuracy_MobileNet:.4f}')
```

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	?	2,257,984
global_average_pooling2d_1 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_3 (Dense)	?	0 (unbuilt)
dropout_2 (Dropout)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)
dropout_3 (Dropout)	?	0 (unbuilt)
dense_5 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 84s 1s/step - accuracy: 0.4051 - loss: 1.6491 - val_accuracy: 0.4112 - val_loss: 1.7908
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 65s 957ms/step - accuracy: 0.6881 - loss: 0.9968 - val_accuracy: 0.3959 - val_loss: 2.0328
Epoch 2: early stopping
Restoring model weights from the end of the best epoch: 1.
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 176ms/step - accuracy: 0.4386 - loss: 1.8038
Validation Loss: 1.8498
Validation Accuracy: 0.4467
CPU times: user 2min 36s, sys: 10.9 s, total: 2min 46s
Wall time: 2min 35s
```

DenseNet, introduced in 2017, is a convolutional neural network (CNN) architecture known for its efficient use of parameters and its ability to achieve high accuracy with relatively fewer layers. Unlike traditional CNNs, where each layer's output is passed to the next layer, DenseNet connects every layer to every other layer after it. This dense connectivity pattern enhances information flow and gradient propagation, leading to improved performance.

Key Features of DenseNet:

1. Dense Connectivity:

- Every layer is connected to every other layer after it.

- This creates a dense highway network, where features from earlier layers are reused by later layers.
- This helps to alleviate the vanishing gradient problem and enhances information flow.

2. Growth Rate (k):
- Each layer adds k new feature maps to the network.
- The growth rate controls the network's depth and width.
- A small growth rate can lead to a more compact network, while a larger growth rate can res ult in a deeper network.

3. Transition Layers:
- Transition layers are used to reduce the number of feature maps and the spatial dimensions of the network.
- They typically consist of a batch normalization layer, a 1x1 convolution, and a 2x2 average pooling layer.

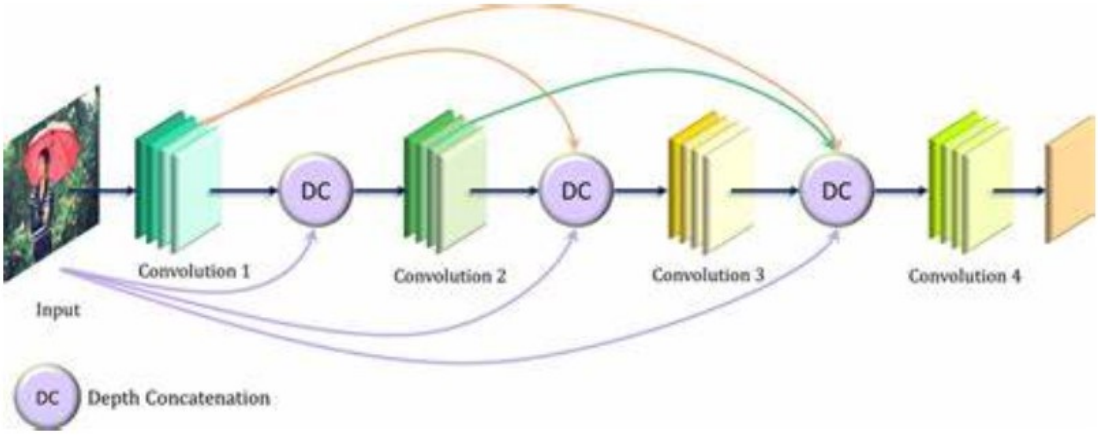
Benefits of DenseNet:

- **Efficient Parameter Usage:** DenseNet can achieve high accuracy with fewer parameters compared to traditional CNNs.
- **Improved Information Flow:** The dense connectivity pattern helps to propagate information more effectively through the network, leading to better gradient flow and reduced vanishing gradient problems.
- **Feature Reuse:** Features from earlier layers are reused by later layers, which can help to improve the network's ability to learn complex patterns.
- **Reduced Overfitting:** The dense connectivity pattern can help to reduce overfitting by encouraging feature reuse and preventing the network from learning redundant features.

Applications of DenseNet:

- **Image Classification:** DenseNet has been successfully used for various image classification tasks, such as ImageNet classification and fine-grained object recognition.
- **Object Detection:** DenseNet has been incorporated into object detection architectures like Faster R-CNN and SSD.
- **Semantic Segmentation:** DenseNet has been applied to semantic segmentation tasks, where the goal is to assign a semantic label to each pixel in an image.

In conclusion, DenseNet is a powerful and efficient CNN architecture that has made significant contributions to the field of computer vision. Its dense connectivity pattern and efficient parameter usage make it a popular choice for various image analysis tasks.




```
%%time

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():

    # Load the pre-trained DenseNet121 model without the top classif
ication layer
    base_model_DenseNet = DenseNet121(weights='imagenet', includ
e_top=False, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_DenseNet.layers:
        layer.trainable = False

    # Create a new model and add the DenseNet121 base model
    model_DenseNet = Sequential()
    model_DenseNet.add(base_model_DenseNet)

    # Add a global average pooling layer and output layer for classific
ation
    model_DenseNet.add(GlobalAveragePooling2D())
    model_DenseNet.add(Dense(128, activation='relu', kernel_regulari
zer=regularizers.l2(0.001)))
    model_DenseNet.add(Dropout(0.4))
    model_DenseNet.add(Dense(64, activation='relu', kernel_regulariz
er=regularizers.l2(0.001)))
```

```
    model_DenseNet.add(Dropout(0.2))
    model_DenseNet.add(Dense(4, activation='softmax'))

    # Model summary
    print("Model Summary (DenseNet121):")
    model_DenseNet.summary()
    print()

    # Compile the model
    model_DenseNet.compile(optimizer='adam', loss='categorical_cro
ssentropy', metrics=['accuracy'])

    # Train the model
    history_DenseNet = model_DenseNet.fit(train_data, epochs=EPOC
HS, validation_data=test_generator, callbacks=[early_stopping], clas
s_weight=class_weights)

    # Validate the model
    val_loss_DenseNet, val_accuracy_DenseNet = model_DenseNet.ev
aluate(test_generator, steps=len(test_generator))
    print(f'Validation Loss: {val_loss_DenseNet:.4f}')
    print(f'Validation Accuracy: {val_accuracy_DenseNet:.4f}')
```

Layer (type)	Output Shape	Param #
densenet121 (Functional)	?	7,037,504
global_average_pooling2d_2 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_6 (Dense)	?	0 (unbuilt)
dropout_4 (Dropout)	?	0 (unbuilt)
dense_7 (Dense)	?	0 (unbuilt)
dropout_5 (Dropout)	?	0 (unbuilt)
dense_8 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 117s 1s/step - accuracy: 0.3769 - loss: 1.6322 - val_accuracy: 0.4010 - val_loss: 1.7286
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 68s 1s/step - accuracy: 0.6416 - loss: 0.9998 - val_accuracy: 0.4112 - val_loss: 1.8933
Epoch 2: early stopping
Restoring model weights from the end of the best epoch: 1.
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 200ms/step - accuracy: 0.3999 - loss: 1.7517
Validation Loss: 1.6654
Validation Accuracy: 0.4315
CPU times: user 3min 31s, sys: 12.5 s, total: 3min 43s
Wall time: 3min 15s
```

InceptionV3 is a convolutional neural network (CNN) architecture introduced in 2015 that is known for its depth, width, and computational efficiency. It builds upon the ideas of the Inception modules introduced in earlier Inception versions, incorporating several enhancements to improve performance.

Key Features of InceptionV3:

- **Inception Modules:** The core building block of InceptionV3 is the Inception module. It consists of a parallel combination of different convolutional filters with different sizes (1x1, 3x3, 5x5) and a pooling layer. This allows the network to capture features at different scales.
- **Factorization:** InceptionV3 uses a factorization technique to decompose 5x5 convolutions into two 3x3 convolutions. This reduces the computational cost while maintaining performance.
- **Label Smoothing:** To regularize the network and prevent over fitting, InceptionV3 uses label smoothing. This technique assigns a small probability to incorrect classes, forcing the network to be less confident in its predictions.
- **Auxiliary Classifiers:** To improve training stability, InceptionV3 includes auxiliary classifiers at intermediate layers. These classifiers help to guide the training process, especially in the early stages.

Benefits of InceptionV3:

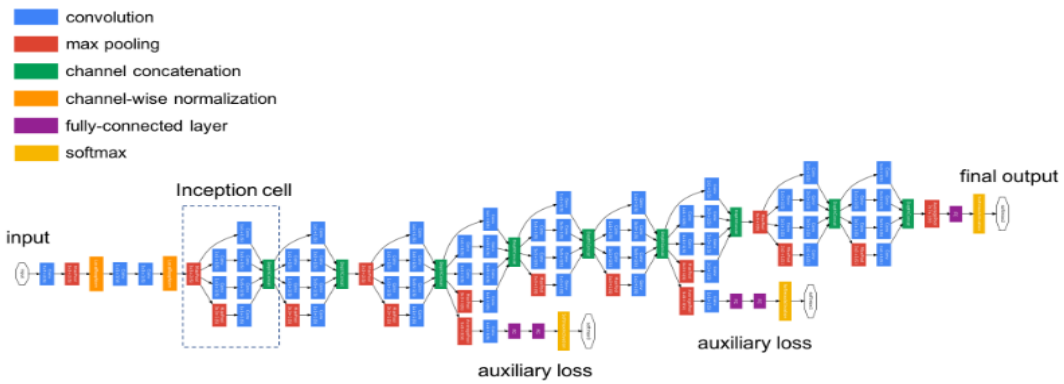
- **High Accuracy:** InceptionV3 has achieved state-of-the-art performance on various image classification benchmarks, including Image Net.
- **Computational Efficiency:** The factorization technique and auxiliary classifiers help to improve the computational efficiency of the network.
- **Flexibility:** The Inception modules allow the network to capture features at different scales, making it more flexible and adaptable to various image tasks.

Applications of InceptionV3:

- **Image Classification:** InceptionV3 is widely used for image classification tasks, such as object recognition, scene classification, and fine-grained categorization.
- **Object Detection:** InceptionV3 has been incorporated into object detection architectures like Faster R-CNN and SSD.

- **Semantic Segmentation:** InceptionV3 has been applied to semantic segmentation tasks, where the goal is to assign a semantic label to each pixel in an image.

In conclusion, InceptionV3 is a powerful and efficient CNN architecture that has made significant contributions to the field of computer vision. Its Inception modules, factorization techniques, and auxiliary classifiers make it a popular choice for various image analysis tasks.




```
%%time

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():

    # Load the pre-trained InceptionV3 model without the top classifi
cation layer
    base_model_Inception = InceptionV3(weights='imagenet', include_
top=False, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_Inception.layers:
        layer.trainable = False

    # Create a new model and add the InceptionV3 base model
    model_Inception = Sequential()
    model_Inception.add(base_model_Inception)

    # Add a global average pooling layer and output layer for classific
ation
    model_Inception.add(GlobalAveragePooling2D())
```

```
    model_Inception.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
    model_Inception.add(Dropout(0.4))
    model_Inception.add(Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
    model_Inception.add(Dropout(0.2))

    model_Inception.add(Dense(4, activation='softmax'))

    # Model summary
    print("Model Summary (InceptionV3):")
    model_Inception.summary()
    print()

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

    # Train the model with EarlyStopping
    history_Inception = model_Inception.fit(train_data, epochs=EPOCHS, validation_data=test_generator, callbacks=[early_stopping], class_weight=class_weights)

    # Validate the model
    val_loss_Inception, val_accuracy_Inception = model_Inception.evaluate(test_generator, steps=len(test_generator))
    print(f'Validation Loss: {val_loss_Inception:.4f}')
    print(f'Validation Accuracy: {val_accuracy_Inception:.4f}')
```

Layer (type)	Output Shape	Param #
inception_v3 (Functional)	?	21,802,784
global_average_pooling2d_3 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_9 (Dense)	?	0 (unbuilt)
dropout_6 (Dropout)	?	0 (unbuilt)
dense_10 (Dense)	?	0 (unbuilt)
dropout_7 (Dropout)	?	0 (unbuilt)
dense_11 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 98s 1s/step - accuracy: 0.4116 - loss: 1.6481 - val_accuracy: 0.4924 - val_loss: 1.4737
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 66s 974ms/step - accuracy: 0.6280 - loss: 1.1214 - val_accuracy: 0.5178 - val_loss: 1.4841
Restoring model weights from the end of the best epoch: 2.
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 165ms/step - accuracy: 0.5253 - loss: 1.4499
Validation Loss: 1.3977
Validation Accuracy: 0.5127
CPU times: user 2min 59s, sys: 10.7 s, total: 3min 10s
Wall time: 2min 52s
```

ResNet (Residual Network), introduced in 2015, is a type of convolutional neural network (CNN) architecture that has significantly impacted the field of deep learning. The key innovation in ResNet is the introduction of residual blocks, which allow the network to learn residual functions instead of the entire underlying mapping. This enables the training of extremely deep networks without suffering from the vanishing gradient problem.

Residual Blocks:

- **Identity Mapping:** A residual block consists of a stack of layers followed by an identity connection. This identity connection allows the network to bypass the stack of layers and directly pass the input to the output.
- **Residual Function:** The residual function is the difference between the output of the stack of layers and the input. This residual function is learned by the network.

Why Residual Blocks Help:

- **Vanishing Gradient Problem:** Residual blocks help to alleviate the vanishing gradient problem, which occurs when gradients become very small during backpropagation, making it difficult for the network to learn. The identity connection provides a direct path for gradients to flow, ensuring that they don't vanish completely.
- **Easier Optimization:** Residual blocks make it easier for the network to learn deep representations. By learning the residual function, the network can focus on learning the differences between the input and output, rather than the entire mapping.

Benefits of ResNet:

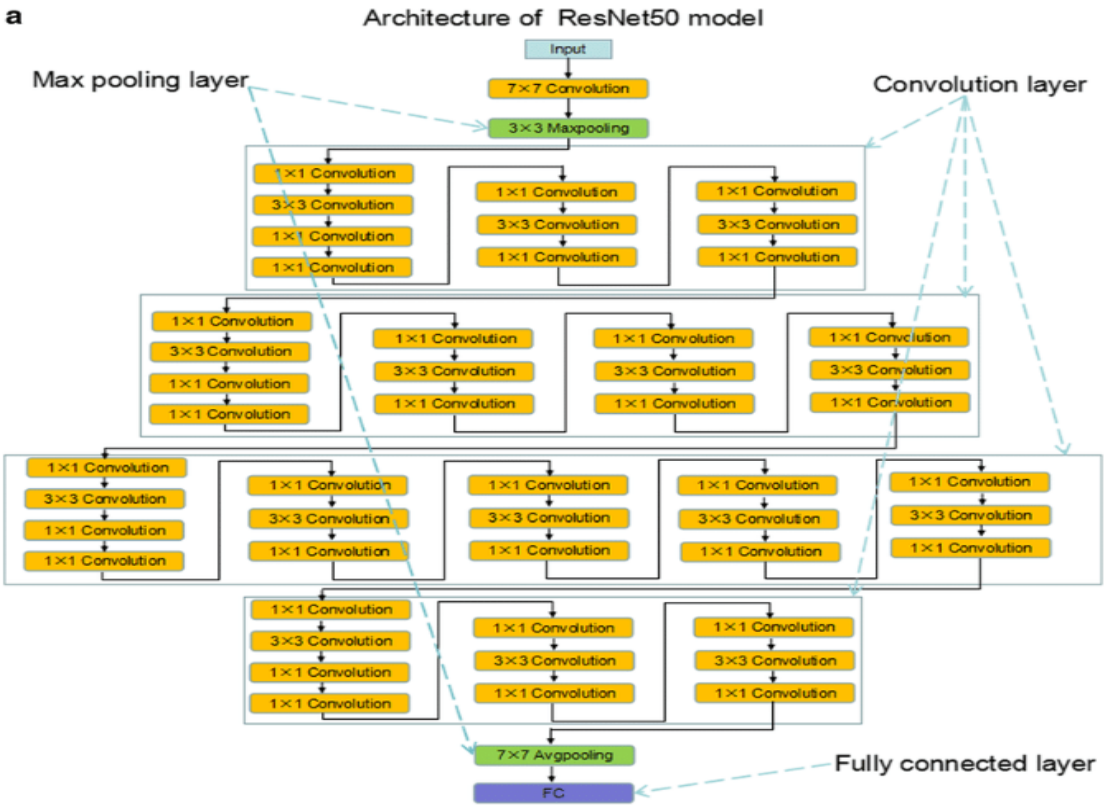
- **Deep Networks:** ResNet has enabled the training of extremely deep networks, which have been shown to improve performance on various tasks.
- **Improved Accuracy:** ResNet has achieved state-of-the-art performance on many image classification benchmarks, such as ImageNet.
- **Faster Training:** Residual blocks can help to speed up training by making it easier for the network to learn deep representations.

Applications of ResNet:

- **Image Classification:** ResNet is widely used for image classification tasks, such as object recognition and scene classification.

- **Object Detection:** ResNet has been incorporated into object detection architectures like Faster R-CNN and SSD.
- **Semantic Segmentation:** ResNet has been applied to semantic segmentation tasks, where the goal is to assign a semantic label to each pixel in an image.

In conclusion, ResNet is a powerful and influential CNN architecture that has enabled the training of extremely deep networks and has achieved state-of-the-art performance on various tasks. The introduction of residual blocks has been a major breakthrough in deep learning, allowing for the development of more complex and accurate models.




```
%%time

import tensorflow as tf
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
, Dropout
from tensorflow.keras import regularizers

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])
```

```
# Open a strategy scope
with strategy.scope():

    # Load the pre-trained ResNet50 model without the top classificat
ion layer
    base_model_ResNet = ResNet50(weights='imagenet', include_top=
False, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_ResNet.layers:
        layer.trainable = False

    # Create a new model and add the ResNet50 base model
    model_ResNet = Sequential()
    model_ResNet.add(base_model_ResNet)

    # Add a global average pooling layer and output layer for classific
ation
    model_ResNet.add(GlobalAveragePooling2D())
    model_ResNet.add(Dense(128, activation='relu', kernel_regularize
r=regularizers.l2(0.001)))
    model_ResNet.add(Dropout(0.4))
    model_ResNet.add(Dense(64, activation='relu', kernel_regularizer
=regularizers.l2(0.001)))
    model_ResNet.add(Dropout(0.2))

    model_ResNet.add(Dense(4, activation='softmax'))

    # Model summary
    print("Model Summary (ResNet50):")
    model_ResNet.summary()
    print()

    # Compile the model
    model_ResNet.compile(optimizer='adam', loss='categorical_cross
entropy', metrics=['accuracy'])

    # Train the model with EarlyStopping
    history_ResNet = model_ResNet.fit(train_data, epochs=EPOCHS, v
alidation_data=test_generator, callbacks=[early_stopping], class_we
ight=class_weights)
```

```
    # Validate the model
    val_loss_ResNet, val_accuracy_ResNet = model_ResNet.evaluate(test_generator, steps=len(test_generator))
    print(f'Validation Loss: {val_loss_ResNet:.4f}')
    print(f'Validation Accuracy: {val_accuracy_ResNet:.4f}')
```

Layer (type)	Output Shape	Param #
resnet50 (Functional)	?	23,587,712
global_average_pooling2d_4 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_12 (Dense)	?	0 (unbuilt)
dropout_8 (Dropout)	?	0 (unbuilt)
dense_13 (Dense)	?	0 (unbuilt)
dropout_9 (Dropout)	?	0 (unbuilt)
dense_14 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 94s 1s/step - accuracy: 0.2606 - loss: 1.7344 - val_accuracy: 0.2893 - val_loss: 1.5052
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 67s 992ms/step - accuracy: 0.2525 - loss: 1.5070 - val_accuracy: 0.2335 - val_loss: 1.4806
Epoch 2: early stopping
Restoring model weights from the end of the best epoch: 1.
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 246ms/step - accuracy: 0.2515 - loss: 1.5305
Validation Loss: 1.5281
Validation Accuracy: 0.2538
CPU times: user 2min 52s, sys: 10.9 s, total: 3min 3s
Wall time: 2min 49s
```

EfficientNet is a family of convolutional neural network (CNN) architectures designed to achieve state-of-the-art accuracy with significantly fewer computational resources compared to previous models. It introduces a novel scaling method that uniformly scales the network's depth, width, and resolution.

Key Features of EfficientNet:

- **Compound Scaling:** EfficientNet uses a compound scaling method that scales the network's depth, width, and resolution in a balanced manner. This ensures that the network's performance improves while maintaining computational efficiency.

- **EfficientNet-B0 to EfficientNet-B7:** The EfficientNet family consists of a series of models, ranging from EfficientNet-B0 to EfficientNet-B7. These models differ in their size and complexity, allowing for a trade-off between accuracy and computational cost.
- **MobileNetV2 Building Blocks:** EfficientNet is based on MobileNetV2 building blocks, which are highly efficient and effective for mobile and embedded vision applications.

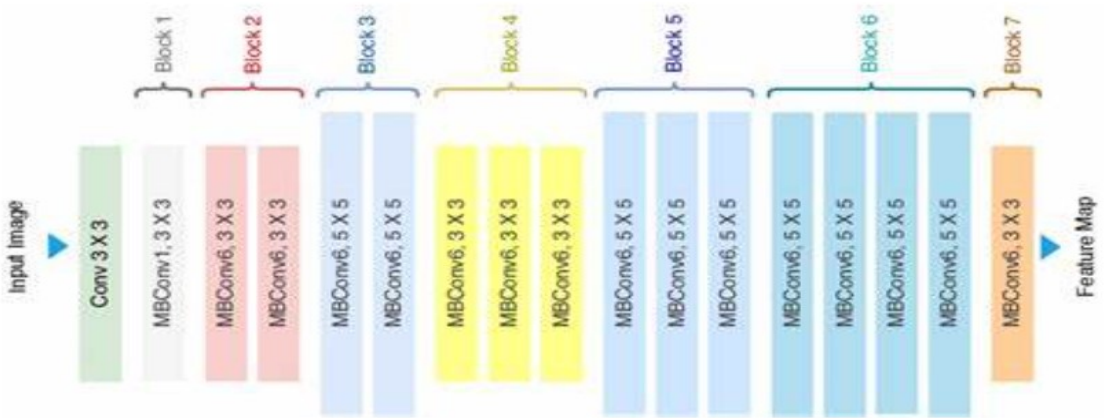
Benefits of EfficientNet:

- **High Accuracy:** EfficientNet achieves state-of-the-art accuracy on various image classification benchmarks, often surpassing previous models with significantly fewer parameters.
- **Computational Efficiency:** EfficientNet is designed to be computationally efficient, making it suitable for deployment on resource-constrained devices.
- **Scalability:** The compound scaling method allows EfficientNet to be easily scaled to different sizes and complexities, providing flexibility for various applications.

Applications of EfficientNet:

- **Image Classification:** EfficientNet is widely used for image classification tasks, such as object recognition, scene classification, and fine-grained categorization.
- **Object Detection:** EfficientNet has been incorporated into object detection architectures like Faster R-CNN and SSD.
- **Semantic Segmentation:** EfficientNet has been applied to semantic segmentation tasks, where the goal is to assign a semantic label to each pixel in an image.

In conclusion, EfficientNet is a powerful and efficient CNN architecture that has made significant contributions to the field of computer vision. Its compound scaling method and efficient building blocks make a popular choice for various image analysis tasks, especially in scenarios where computational resources are limited.



```
%%time

import tensorflow as tf
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
, Dropout
from tensorflow.keras import regularizers

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():

    # Load the pre-trained EfficientNetB0 model without the top class
ification layer
    base_model_EfficientNet = EfficientNetB0(weights='imagenet', inc
lude_top=False, input_shape=(image_height, image_width, 3))

    # Set the layers of the base model as non-trainable (freeze them)
    for layer in base_model_EfficientNet.layers:
        layer.trainable = False

    # Create a new model and add the EfficientNetB0 base model
    model_EfficientNet = Sequential()
    model_EfficientNet.add(base_model_EfficientNet)

    # Add a global average pooling layer and output layer for classific
ation
    model_EfficientNet.add(GlobalAveragePooling2D())
    model_EfficientNet.add(Dense(128, activation='relu', kernel_regul
arizer=regularizers.l2(0.001)))
    model_EfficientNet.add(Dropout(0.4))
    model_EfficientNet.add(Dense(64, activation='relu', kernel_regular
izer=regularizers.l2(0.001)))
    model_EfficientNet.add(Dropout(0.2))

    model_EfficientNet.add(Dense(4, activation='softmax'))

    # Model summary
```

```
    print("Model Summary (EfficientNetB0):")
    model_EfficientNet.summary()
    print()

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

    # Train the model with EarlyStopping
    history_EfficientNet = model_EfficientNet.fit(train_data, epochs=EPOCHS, validation_data=test_generator, callbacks=[early_stopping], class_weight=class_weights)

    # Validate the model
    val_loss_EfficientNet, val_accuracy_EfficientNet = model_EfficientNet.evaluate(test_generator, steps=len(test_generator))
    print(f'Validation Loss: {val_loss_EfficientNet:.4f}')
    print(f'Validation Accuracy: {val_accuracy_EfficientNet:.4f}')
```

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	?	4,049,571
global_average_pooling2d_5 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_15 (Dense)	?	0 (unbuilt)
dropout_10 (Dropout)	?	0 (unbuilt)
dense_16 (Dense)	?	0 (unbuilt)
dropout_11 (Dropout)	?	0 (unbuilt)
dense_17 (Dense)	?	0 (unbuilt)

```
60/60 ━━━━━━━━━━━━━━━━━━━━ 98s 1s/step - accuracy: 0.2314 - loss: 1.6641 - val_accuracy: 0.2183 - val_loss: 1.5422
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 65s 959ms/step - accuracy: 0.2513 - loss: 1.5095 - val_accuracy: 0.2284 - val_loss: 1.4732
Epoch 2: early stopping
Restoring model weights from the end of the best epoch: 1.
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 193ms/step - accuracy: 0.1881 - loss: 1.5437
Validation Loss: 1.5347
Validation Accuracy: 0.1878
CPU times: user 2min 52s, sys: 10.5 s, total: 3min 3s
Wall time: 2min 51s
```

NASNet is a convolutional neural network (CNN) architecture that was designed using neural architecture search (NAS) techniques. This means that the network's architecture was not manually designed by humans but rather was automatically generated by a machine learning algorithm.

Key Features of NASNet:

- **Neural Architecture Search:** NASNet was generated using a reinforcement learning algorithm that searched through a vast space of possible architectures to find the best one.
- **Inception-like Modules:** NASNet is based on Inception-like modules, which are a combination of different convolutional filters with different sizes.
- **Transfer Learning:** NASNet can be used for transfer learning, where a pre-trained model is fine-tuned on a smaller dataset to solve a related task.

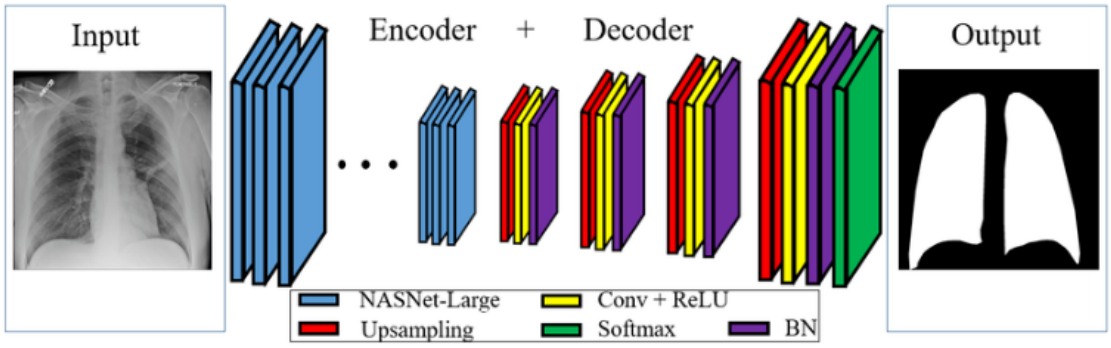
Benefits of NASNet:

- **High Accuracy:** NASNet has achieved state-of-the-art performance on various image classification benchmarks.
- **Efficient Architecture:** The architecture of NASNet is designed to be computationally efficient.
- **Automation:** The use of neural architecture search eliminates the need for manual design, making it easier to explore a wider range of architectures.

Applications of NASNet:

- **Image Classification:** NASNet is widely used for image classification tasks, such as object recognition, scene classification, and fine-grained categorization.
- **Object Detection:** NASNet has been incorporated into object detection architectures like Faster R-CNN and SSD.
- **Semantic Segmentation:** NASNet has been applied to semantic segmentation tasks, where the goal is to assign a semantic label to each pixel in an image.

In conclusion, NASNet is a powerful and efficient CNN architecture that has made significant contributions to the field of computer vision. Its use of neural architecture search allows for the automatic generation of high-performing models, making it a valuable tool for researchers and practitioners.




```
%%time

import tensorflow as tf
from tensorflow.keras.applications import NASNetMobile
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
, Dropout
from tensorflow.keras import regularizers

# Create a MirroredStrategy (adjust device names if needed)
strategy = tf.distribute.MirroredStrategy(devices=['/gpu:0', '/gpu:1'])

# Open a strategy scope
with strategy.scope():
  # Load the pre-trained NASNetMobile model without top classificat
ion layer
  base_model_NASNet = NASNetMobile(weights='imagenet', include_
top=False, input_shape=(image_height, image_width, 3))

  # Set the layers of the base model as non-trainable (freeze them)
  for layer in base_model_NASNet.layers:
    layer.trainable = False

  # Create a new model and add the NASNetMobile base model
  model_NASNet = Sequential()
  model_NASNet.add(base_model_NASNet)

  # Add a global average pooling layer and output layer for classifica
tion
  model_NASNet.add(GlobalAveragePooling2D())
```

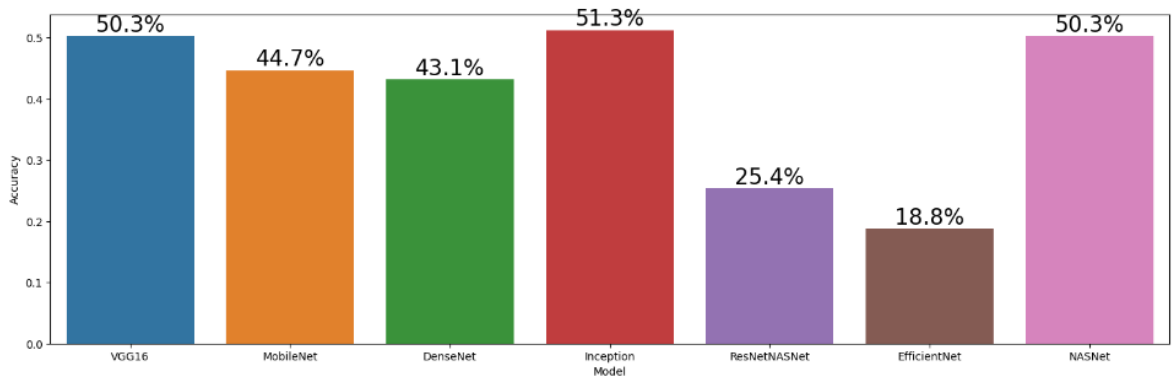
```
  model_NASNet.add(Dense(128, activation='relu', kernel_regularizer
=regularizers.l2(0.001)))
  model_NASNet.add(Dropout(0.4))
  model_NASNet.add(Dense(64, activation='relu', kernel_regularizer=
regularizers.l2(0.001)))
  model_NASNet.add(Dropout(0.2))
  model_NASNet.add(Dense(4, activation='softmax'))

  # Model summary
  print("Model Summary (NASNetMobile):")
  model_NASNet.summary()
  print()

  # Compile the model
  model_NASNet.compile(optimizer='adam', loss='categorical_crosse
ntropy', metrics=['accuracy'])

history_NASNet = model_NASNet.fit(train_data, epochs=EPOCHS, val
idation_data=test_generator, callbacks=[early_stopping], class_weig
ht=class_weights)

  #Validate the model
val_loss_NASNet, val_accuracy_NASNet = model_NASNet.evaluate(t
est_generator, steps=len(test_generator))
print(f'Validation Loss: {val_loss_NASNet:.4f}')
print(f'Validation Accuracy: {val_accuracy_NASNet:.4f}')
```

Layer (type)	Output Shape	Param #
NASNet (Functional)	?	4,269,716
global_average_pooling2d_6 (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_18 (Dense)	?	0 (unbuilt)
dropout_12 (Dropout)	?	0 (unbuilt)
dense_19 (Dense)	?	0 (unbuilt)
dropout_13 (Dropout)	?	0 (unbuilt)
dense_20 (Dense)	?	0 (unbuilt)

```
Epoch 1/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 140s 1s/step - accuracy: 0.4293 - loss: 1.5059 - val_accuracy: 0.5635 - val_loss: 1.4621
Epoch 2/2
60/60 ━━━━━━━━━━━━━━━━━━━━ 64s 948ms/step - accuracy: 0.6694 - loss: 1.0520 - val_accuracy: 0.5584 - val_loss: 1.4447
Restoring model weights from the end of the best epoch: 1.
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 186ms/step - accuracy: 0.5102 - loss: 1.5139
Validation Loss: 1.5138
Validation Accuracy: 0.5025
CPU times: user 3min 48s, sys: 12.3 s, total: 4min
Wall time: 3min 37s
```

Model Performance Comparison

```
data = {
    'VGG16': val_accuracy_VGG16,
    'MobileNet': val_accuracy_MobileNet,
    'DenseNet': val_accuracy_DenseNet,
    'Inception': val_accuracy_Inception,
    'ResNetNASNet' : val_accuracy_ResNet,
    'EfficientNet' : val_accuracy_EfficientNet,
    'NASNet' : val_accuracy_NASNet
}


df = pd.DataFrame.from_dict(data, orient='index', columns=['Accuracy'])
df = df.reset_index().rename(columns={'index': 'Model'})


plt.figure(figsize=[15, 5])

# Create bar chart
sns.barplot(x='Model', y='Accuracy', data=df)

# Add labels to bars
ax = plt.gca()
for bar in ax.containers:
    ax.bar_label(bar, label_type='edge', labels=[f"{x:.1%}" for x in bar.datavalues], fontsize=20)

# Adjust the layout
plt.tight_layout()

plt.show()
```



Prediction Result Samples

```
test_generator.reset()
img, label = next(test_generator)

prediction = model_Inception.predict(img)
test_pred_classes = np.argmax(prediction, axis=1)

plt.figure(figsize=[20, 20])
for i in range(20):
    plt.subplot(5, 4, i+1)
    plt.imshow(img[i])
    plt.axis('off')
    plt.title("Label : {}\n Prediction : {} {:.1f}%".format(class_names[np.
argmax(label[i])], class_names[test_pred_classes[i]], 100 * np.max(p
rediction[i])))
plt.show()
```