## 1. Task Description

- **Objective**: Classify celebrity faces based on their identities.
- **Dataset**: Pins Face Recognition Dataset.
    - We only used the highest 50 classes contain images.

## 2. Problem Context

Facial recognition tasks require robust feature extraction to achieve accurate classification. The choice of neural network architecture significantly impacts performance due to variations in feature extraction capabilities and generalization.

## 3. Models Overview

### 1) ResNet

- **Architecture**: Deep residual learning framework using skip connections to mitigate vanishing gradients.
- **Train Accuracy**: 0.84
- **Test Accuracy**: 0.56

### 2) DenseNet

- **Architecture**: Dense connectivity between layers to enhance feature reuse and reduce redundancy.
- **Train Accuracy**: 0.71
- **Test Accuracy**: 0.55

### 3) Xception

- **Architecture:** Depth wise separable convolutions for efficient computation and better feature extraction.
- **Train Accuracy:** 0.75
- **Test Accuracy:** 0.64

---

# ResNet

## 1. Introduction

- What is ResNet?
  ResNet, short for Residual Network, is a deep learning architecture introduced by Kaiming He et al. in 2015. It won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2015 by achieving state-of-the-art accuracy.
- **Purpose**: ResNet solves the vanishing gradient problem, enabling the training of very deep neural networks by introducing residual connections.

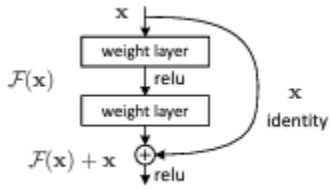


Figure 2. Residual learning: a building block.

## 2. Key Concepts

### 1. The Vanishing Gradient Problem

- In deep networks, gradients diminish during backpropagation, making it hard for weights in earlier layers to update.
- ResNet addresses this by allowing gradients to flow through shortcut connections.

### 2. Residual Learning

- Instead of learning the target mapping H(x), ResNet learns the residual mapping F(x)=H(x) − x.
- The original mapping becomes H(x) = F(x) + x.

### 3. Shortcut Connections

- **Definition**: Skip connections bypass one or more layers, adding the input xxx directly to the output of the next layer.
- **Advantages**:
  - Eases optimization.
  - Prevents performance degradation as the network depth increases.

## 3. Architecture

### 1. Basic Block

A residual block contains:

- Two or more convolutional layers.
- Batch normalization (BN) after each convolution.
- ReLU activation.
- A shortcut connection that adds the input x to the output F(x).

### 2. Building Blocks

1. Identity Block:
   - Input and output dimensions are the same.
   - Shortcut connection adds the input directly to the output.
2. Convolutional Block:
   - Used when input and output dimensions differ.
   - Employs a 1×1 convolution to match dimensions

### 3. ResNet Variants

- ResNet-18, ResNet-34: Use basic blocks with fewer layers
- ResNet-50, ResNet-101, ResNet-152: Use bottleneck blocks, reducing computation while increasing depth.

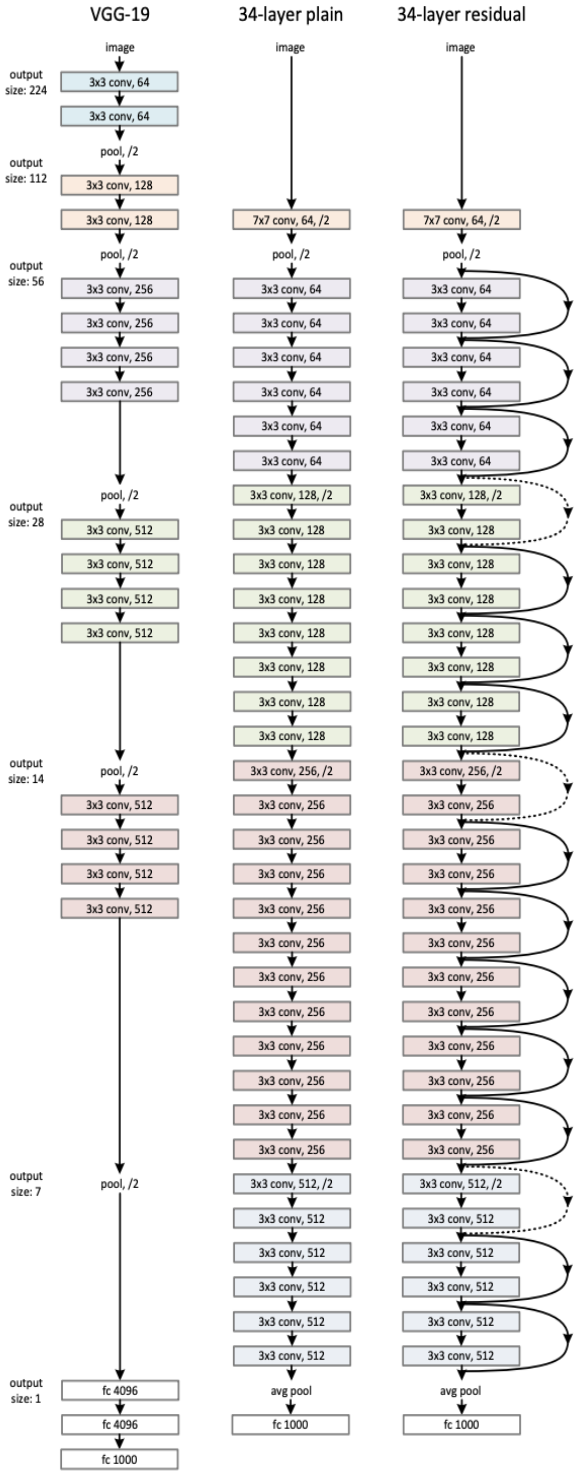



Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

## 4. Steps to Implement Architecture

### 1. Input Layer

Define the input tensor using tf.keras.Input with the specified input_shape

```
def identity_block(x, filter):
    # copy tensor to variable called x_skip
    x_skip = x

    # Layer 1
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)

    # Layer 2
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)

    # Adjust the number of channels in x_skip to match x using a 1x1 convolution
    # if necessary
    num_channels_x = x.shape[-1]  # Get the number of channels in x
    num_channels_x_skip = x_skip.shape[-1]  # Get the number of channels in x_skip

    if num_channels_x != num_channels_x_skip:  # If channel numbers don't match
        x_skip = tf.keras.layers.Conv2D(filter, (1, 1), padding='same')(x_skip)  # App

    # Add Residue
    x = tf.keras.layers.Add()([x, x_skip])
    x = tf.keras.layers.Activation('relu')(x)
    return x
```

### 2. Initial Convolutional Layer

- Apply a 3×3 convolution with stride 2×2 to reduce spatial dimensions.
- Add batch normalization and ReLU activation.
- Use a 3×3 max-pooling layer to further reduce dimensions.

```
def convolutional_block(x, filter):
    # copy tensor to variable called x_skip
    x_skip = x

    # Layer 1
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same', strides = (2,2))(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)

    # Layer 2
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)

    # Processing Residue with conv(1,1)
    x_skip = tf.keras.layers.Conv2D(filter, (1,1), strides = (2,2))(x_skip)

    # Add Residue
    x = tf.keras.layers.Add()([x, x_skip])
    x = tf.keras.layers.Activation('relu')(x)
    return x
```

3. **Residual Stages**

   The architecture is divided into four stages, each containing residual blocks:

   1. **Stage 1:**
      - Two identity blocks with 64 filters.
      - Identity blocks maintain the same spatial dimensions.
   2. **Stage 2:**
      - A convolutional block with 128 filters to downsample the feature map using stride 2×2.
      - One identity block with 128 filters.
   3. **Stage 3:**
      - A convolutional block with 256 filters to downsample the feature map.
      - One identity block with 256 filters.
   4. **Stage 4:**
      - A convolutional block with 512 filters to downsample the feature map.
      - One identity block with 512 filters

4. **Global Average Pooling**

   A global average pooling layer reduces the spatial dimensions to 1×1, providing a compact feature vector**.**

5. **Dropout**

   Apply a dropout layer with a dropout rate of 0.8 for regularization.

6. **Fully Connected Layers**

   1. A dense layer with 128 units, ReLU activation, and L2 regularization to reduce overfitting.
   2. A final dense layer with num_classes units and softmax activation for classification.

```python
from tensorflow.keras.regularizers import l2

def ResNet(input_shape, num_classes):
    inputs = tf.keras.Input(shape=input_shape)

    # Initial Conv Layer
    x = tf.keras.layers.Conv2D(32, (3, 3), strides=(2, 2), padding='same')(inputs)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)
    x = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    # Stage 1
    x = identity_block(x, 64)
    x = identity_block(x, 64)

    # Stage 2
    x = convolutional_block(x, 128)
    x = identity_block(x, 128)

    # Stage 3
    x = convolutional_block(x, 256)
    x = identity_block(x, 256)

    # Stage 4
    x = convolutional_block(x, 512)
    x = identity_block(x, 512)

    # Global Average Pooling
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dropout(0.8)(x)  # Dropout for regularization

    # Dense Layer with L2 Regularization
    x = tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=l2(0.01))(x)

    # Final Output Layer
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

    # Create Model
    model = tf.keras.Model(inputs, outputs)
    return model
```

## 5. Advantages

- Overcomes vanishing gradients.
- Enables deeper networks with improved performance.
- Easy optimization due to residual connections.
- Generalization capability makes it highly effective for transfer learning.

## 6. Disadvantages

- Computationally intensive for very deep architectures.
- Higher memory requirements compared to simpler models.
- Susceptible to diminishing returns with increasing depth.

## 7. Applications

- **Image Classification:** State-of-the-art accuracy on ImageNet and CIFAR datasets.
- **Object Detection:** Backbone for detection frameworks like Faster R-CNN and YOLO.

- **Segmentation:** Used in semantic and instance segmentation tasks (e.g., Mask R-CNN).
- **Transfer Learning:** Pretrained ResNet models are widely used in downstream tasks.

## 8. References

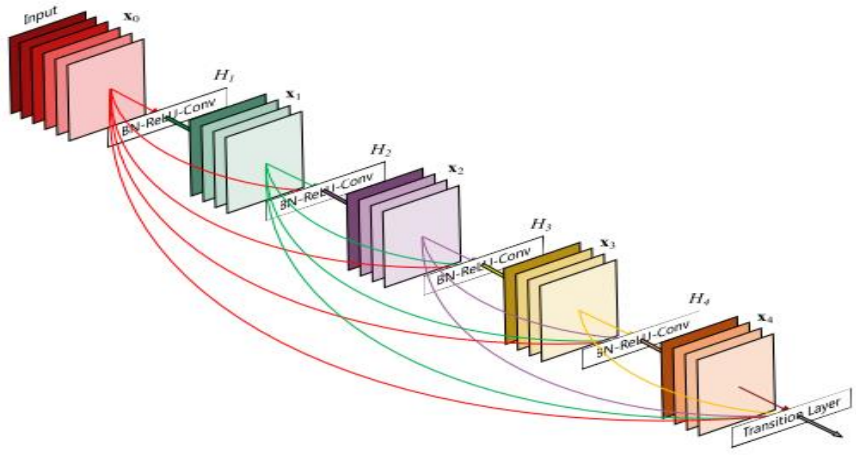

1. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition* (2015).
2. PyTorch ResNet Documentation.

# DenseNet

## 1. Introduction

DenseNet (Dense Convolutional Network) is a deep learning architecture proposed by **Gao Huang et al.** in 2017. It connects each layer in a feed-forward fashion, addressing **vanishing gradient problems** while improving feature reuse and computational efficiency.

- **Purpose**: DenseNet mitigates redundancy and enhances feature propagation by connecting all layers directly.
- **Key Benefit**: Reduces the number of parameters and avoids learning redundant features.
- **DenseNet Highlights**:
  - **Efficient Gradient Flow**: Resolves the vanishing gradient problem.
  - **Feature Reuse**: Enhances performance by utilizing prior layers' features.
  - **Pre-trained Models**: Trained on **ImageNet**, providing generalizable weights for new tasks




## 2. Key Concepts of DenseNet

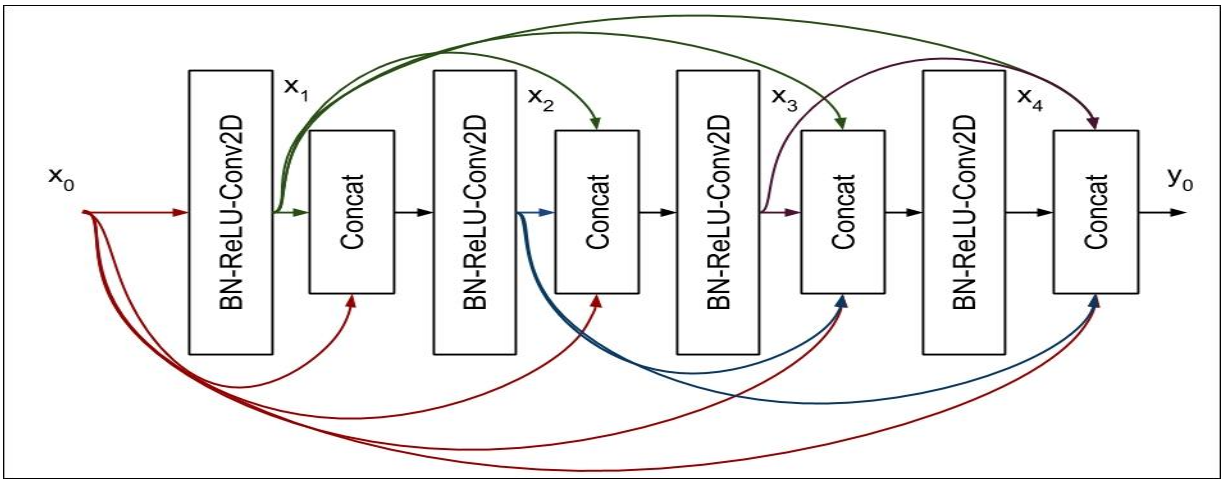
### 1. Dense Connectivity

- Each layer receives input from **all previous layers** and passes its output to subsequent layers.

- **Equation**:

$$x_l = H_l([x_0, x_1, \ldots, x_{l-1}])$$

where $H_l$ is a composite function of Batch Normalization (BN), ReLU activation, and Convolution.




## 2. Feature Reuse

- DenseNet improves efficiency by reusing features across multiple layers, reducing the need for redundant learning.

## 3. Growth Rate

- The growth rate k determines the number of new feature maps added per layer.
- The total feature maps in a layer grow linearly as: $c_l = c_0 + k \cdot (l-1)$

## 4. Transition Layers

- These are used between **Dense Blocks** for down-sampling.
- A Transition Layer consists of:
  - 1×1 Convolution
  - 2×2 Average Pooling

## 5.Pre-trained Weights

- DenseNet models are pre-trained on **ImageNet**, a dataset with over 1.2 million labeled images across 1,000 classes.
- These weights allow the model to **transfer knowledge** to other tasks without training from scratch.

## 3. Architecture

DenseNet-121 is a **121-layer variant** of the DenseNet model. It consists of:

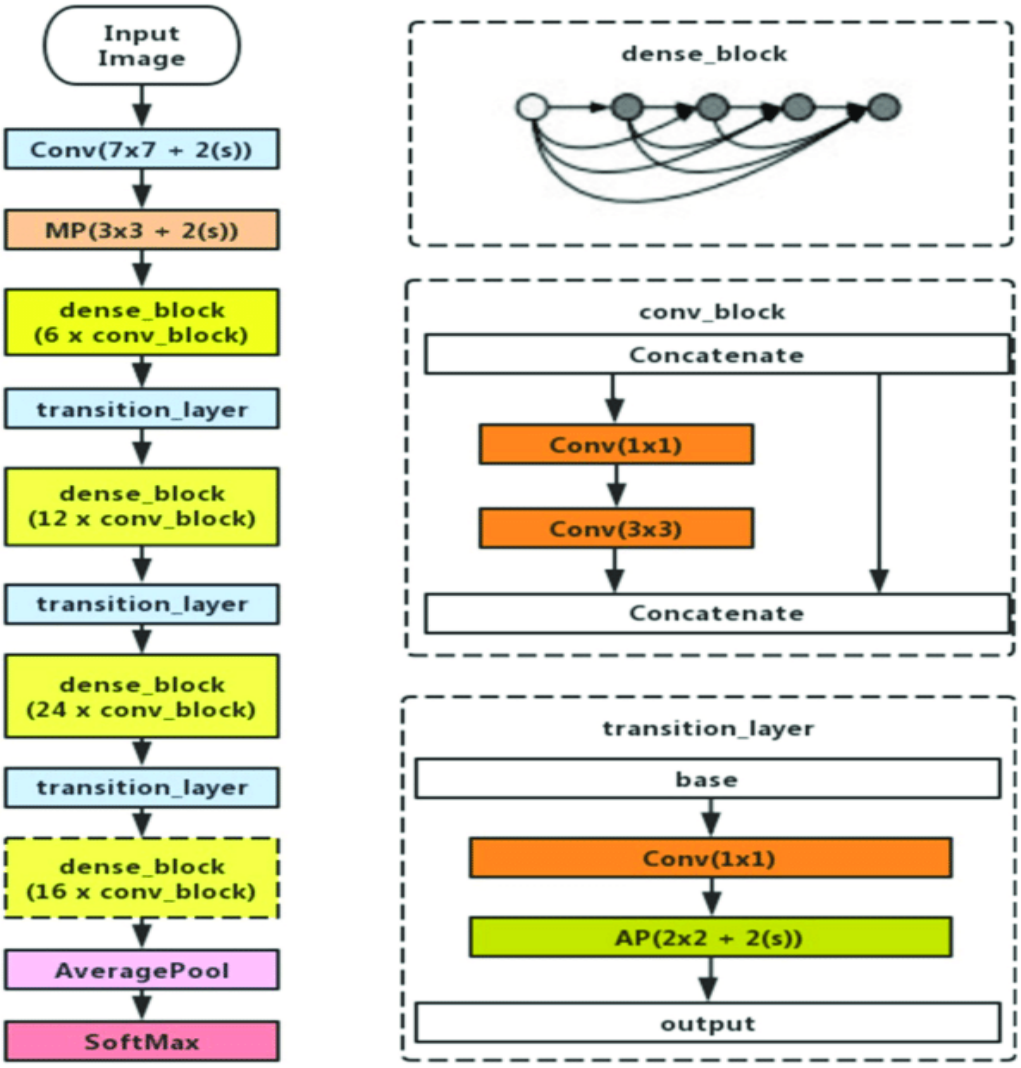
1. **Initial Convolutional Layer**:
   7×7 convolution, stride 2, followed by 3×3 max pooling.

2. **Dense Blocks**:
   - **Block 1**: 6 convolutional layers
   - **Block 2**: 12 convolutional layers
   - **Block 3**: 24 convolutional layers
   - **Block 4**: 16 convolutional layers

3. **Transition Layers**:
   Down-sampling using **1×1 convolution** and **2×2 average pooling** between Dense Blocks.

4. **Global Average Pooling**:
   Reduces the feature map dimensions to 1×1.

5. **Classification Layer**:
   Fully connected (FC) layer with **softmax activation** for classification




## 4. Pre-trained DenseNet for Transfer Learning

### Why Use Pre-trained DenseNet?

- **Faster Training**: Pre-trained weights provide a good initialization, reducing time and resources.
- **High Accuracy**: Generalizable features from ImageNet training improve task performance.
- **Robust Features**: The dense connectivity improves feature propagation.

## Steps for Transfer Learning

1. **Load Pre-trained DenseNet-121**:
   Use a library like **PyTorch** or **TensorFlow** to load pre-trained weights.

```
from tensorflow.keras.applications import DenseNet121
model = DenseNet121(weights='imagenet', include_top=True)
```

2. **Modify the Final Layer**:
   Replace the fully connected layer to match the number of target classes.

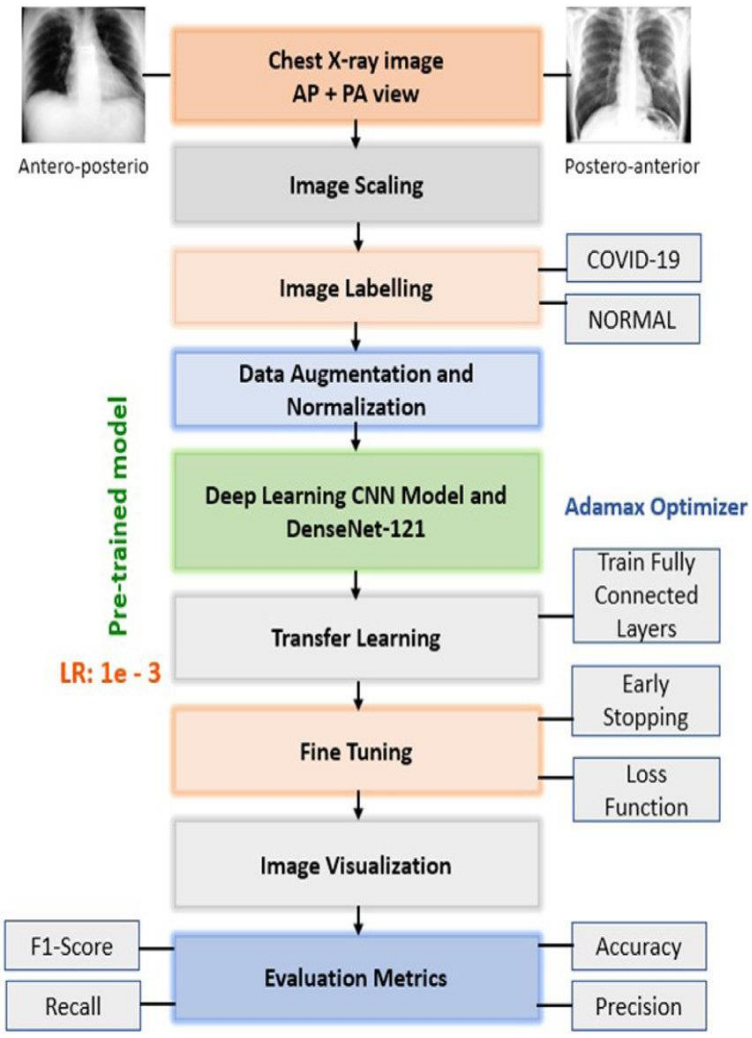
```
# Add custom classification layers
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Pool the feature maps
x = Dense(512, activation='relu')(x)
x = Dropout(0.25)(x)  # Regularization
x = Dense(256, activation='relu')(x)
x = Dropout(0.25)(x)  # Regularization
predictions = Dense(num_classes, activation='softmax')(x)  # Final layer for classification

# Create the full model
model = Model(inputs=base_model.input, outputs=predictions)
```

3. **Fine-tuning**:
   Freeze the early layers to retain ImageNet features.
   Fine-tune only the last Dense Block and classifier on the new dataset.

```
# Freeze all layers except the last 9 layers
for layer in model.layers[:-9]:
    layer.trainable = False
```

4. **Training**:
   Train the modified model on your dataset while leveraging the pre-trained weights.

```
history = model.fit(
    train_dataset,
    validation_data=test_dataset,
    epochs=30,
    shuffle=True,
    verbose=1
)
```




## 5. Advantages of Pre-trained DenseNet

1. **Efficiency**: Reduces computational load and training time.
2. **High Accuracy**: Leverages pre-trained knowledge from a large-scale dataset.
3. **Gradient Flow**: Dense connections improve training dynamics.
4. **Parameter Efficiency**: Fewer parameters compared to other deep architectures like ResNet.

## 6. Disadvantages of Pre-trained DenseNet

1. **Memory-Intensive**: Dense connections require more memory to store intermediate outputs.
2. **Slower Inference**: Dense connectivity can increase computation time during inference.

3. **Not Ideal for All Tasks**: Computational overhead may not justify use for simpler datasets.

## 7. Applications

DenseNet-121 is widely used for:

- **Image Classification**: Used for tasks like medical image analysis, weather recognition, and more.
- **Medical Imaging**: Pre-trained DenseNet is effective for tasks like COVID-19 detection and disease classification.
- **Acoustic Event Classification**: Extracts features from spectrogram images for audio event classification
- **Transfer Learning**: Fine-tuned on smaller datasets for domain-specific tasks.

## 9. References


1. Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). *Densely Connected Convolutional Networks*.
2. Deep Architecture based on DenseNet-121 Model for Weather Image Recognition
3. PyTorch DenseNet Documentation

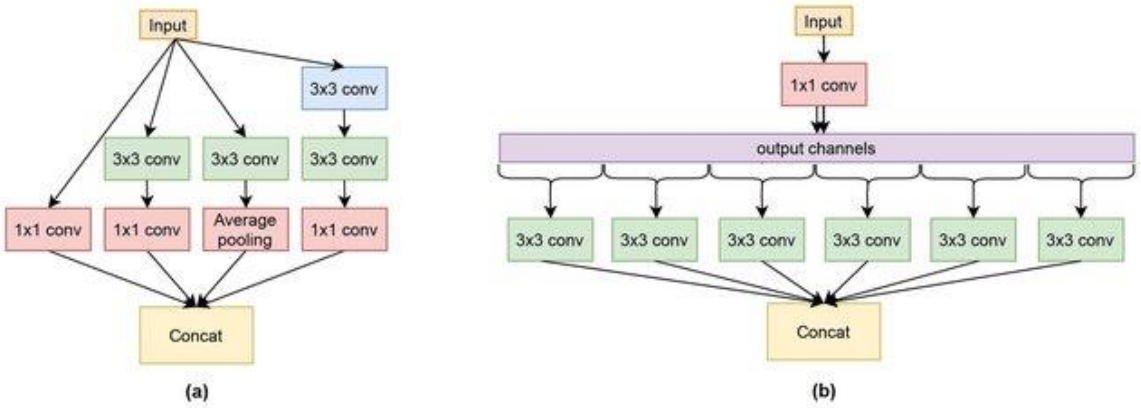
# Xception

## 1. Introduction

- What's depth wise separable convolutional neural networks ?
- It's a type of CNNs that have less number of parameters which can reduce over-fitting and are computationally cheaper than other architectures like DenseNet.
- It was introduced in 2017 by François Chollet.

## 2.Core Concept

The key idea is that spatial convolutions and channel-wise convolutions can be separated, making the model more efficient while maintaining its representational power.



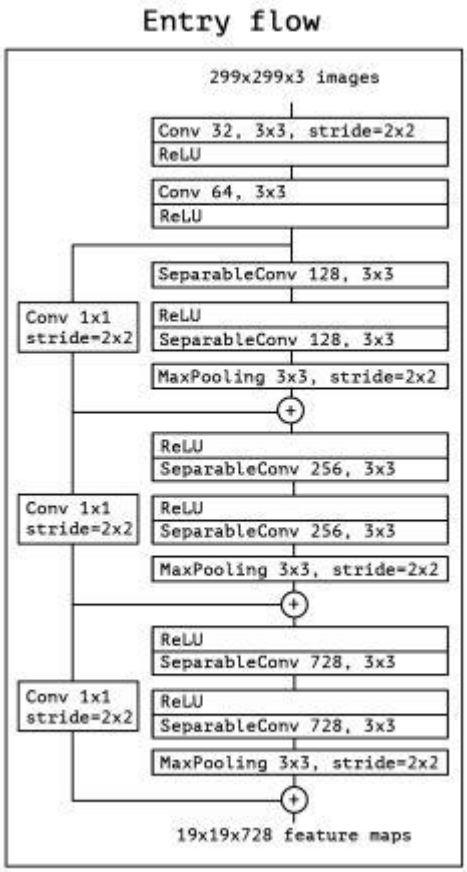

- Figure (a) on the left is the inception module basic architecture and (b) is the xception module which shows the difference
- In the inception module we can see multiple convolution filters of different sizes in parallel which focuses on capturing multi-scale spatial features by applying various convolution sizes.
- In the xception module it factorizes convolutions into depthwise separable convolutions for efficiency which leads to efficient computation without losing representation power.
- Paper

## 3. Architecture

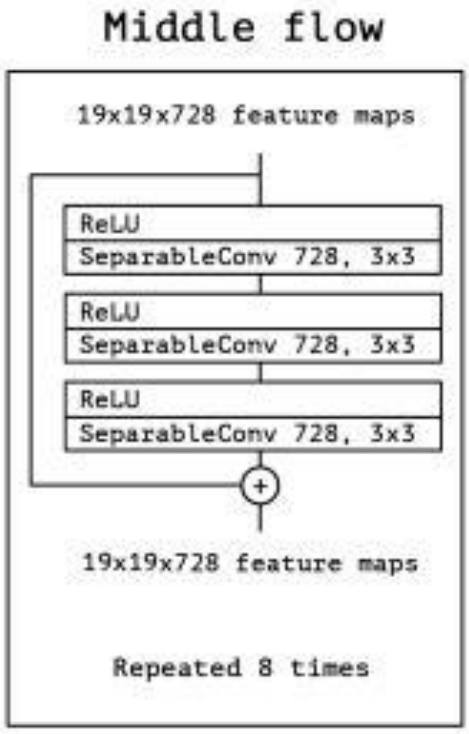
It is divided into three main sections which are :

1. Entry flow :

    - Begins with a standard convolution and max-pooling layer to reduce spatial dimensions.
    - Followed by three sets of depthwise separable convolutions, each with residual connections.
    - Each block progressively reduces the spatial dimensions while increasing the feature maps.

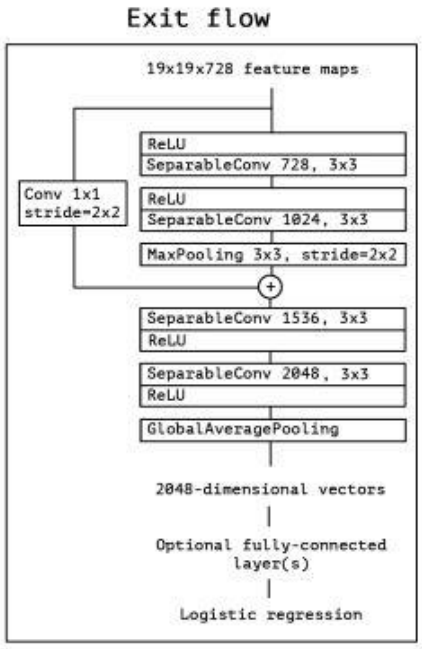


2. Middle flow :
    - Consists of eight identical blocks.
    - Each block contains three depthwise separable convolution layers without any spatial reduction.
    - Focuses on learning complex feature hierarchies.



3. Exit flow :
    - Comprises depthwise separable convolutions followed by global average pooling.

    - A final fully connected layer with a softmax activation outputs the class probabilities.



## 4. Implementing the Xception Model Using Transfer Learning

- Import Required Libraries

```
import tensorflow as tf
from tensorflow.keras.applications import
Xception
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense,
GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image
import ImageDataGenerator
from tensorflow.keras.optimizers import
Adam
```

- Load Pre-trained Xception Model

```
base_model = Xception(weights='imagenet',
include_top=False, input_shape=(299, 299,
3))
```

- Freeze the Base Model Layers

```
#to preserve pre-trained knowledge and
avoid overfitting
for layer in base_model.layers:
    layer.trainable = False
```

- Add Custom Classification Layers

```
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Convert
feature maps to a single vector per image
```

```
x = Dense(1024, activation='relu')(x)  #
Add a fully connected layer
predictions = Dense(10,
activation='softmax')(x)  # 10 classes for
classification

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

- Prepare the Data

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

test_datagen =
ImageDataGenerator(rescale=1.0/255)

train_generator =
train_datagen.flow_from_directory(
    'data/train',
    target_size=(299, 299),
    batch_size=32,
    class_mode='categorical'
)

validation_generator =
test_datagen.flow_from_directory(
    'data/validation',
```

```
    target_size=(299, 299),
    batch_size=32,
    class_mode='categorical'
)
```

- Compile the Model

```
model.compile(optimizer=Adam(learning_rate
=0.001),

loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- Train the Model

```
history = model.fit(
    train_generator,

steps_per_epoch=train_generator.samples //
train_generator.batch_size,
    validation_data=validation_generator,

validation_steps=validation_generator.samp
les // validation_generator.batch_size,
    epochs=10
)
```

- Fine-Tune the Model

```
for layer in base_model.layers[-30:]:  #
Unfreeze the last 30 layers
    layer.trainable = True

model.compile(optimizer=Adam(learning_rate
```

```
=0.0001),

loss='categorical_crossentropy',
              metrics=['accuracy'])

history_finetune = model.fit(
    train_generator,

steps_per_epoch=train_generator.samples //
train_generator.batch_size,
    validation_data=validation_generator,

validation_steps=validation_generator.samp
les // validation_generator.batch_size,
    epochs=5
)
```

- Evaluate the Model

```
loss, accuracy =
model.evaluate(validation_generator)
print(f"Validation Loss: {loss}")
print(f"Validation Accuracy: {accuracy}")
```

- Save the Model

```
model.save('xception_transfer_learning.h5'
)
```

- Load and Test the Model

```
from tensorflow.keras.models import
load_model
```

```
model =
load_model('xception_transfer_learning.h5'
)

# Testing on new data
test_image =
tf.keras.utils.load_img('test_image.jpg',
target_size=(299, 299))
test_image =
tf.keras.utils.img_to_array(test_image) /
255.0
test_image = tf.expand_dims(test_image,
axis=0)

prediction = model.predict(test_image)
print("Predicted class:",
tf.argmax(prediction[0])
```

## 5. Advantages

- **Fewer Parameters:** Reduces the risk of overfitting.
- **Computational Efficiency:** Requires less computational power compared to DenseNet and ResNet.
- **Strong Performance:** Balances efficiency and accuracy across tasks.

## 6. Disadvantages

- **Limited Flexibility:** May not perform optimally on tasks requiring highly diverse feature representations.
- **Specialized Use:** Works best with depthwise separable convolutions; less effective in certain configurations.

## 7. Applications

- Image Classification

- Object Detection

- Semantic Segmentation

- Transfer Learning for domain-specific tasks

## 8. References


Understanding Inception: Simplifying the Network Architecture | by Arjun Sarkar | The Startup | Medium

Inception Module Explained | Papers With Code

(PDF) A Deep Learning Method for Early Detection of Diabetic Foot Using Decision Fusion and Thermal Images

https://arxiv.org/pdf/1610.02357 (main paper)s

**Comparison of Models**

| | ResNet | DenseNet | Xception |
|---|---|---|---|
| **Train Accuracy** | 0.84 | 0.71 | 0.75 |
| **Test Accuracy** | 0.56 | 0.55 | 0.64 |
| **Pros** | • Handles deeper networks effectively with residual connections.<br>• High capacity for learning complex features. | • Efficient parameter usage with dense connections.<br>• Mitigates gradient vanishing better than traditional networks. | • Highly efficient feature extraction with depth wise separable convolutions.<br>• Balanced training and testing performance, showing strong generalization. |
| **Cons** | • Prone to overfitting with small datasets.<br>• Computationally intensive. | • Limited capacity to generalize in this specific task.<br>• Computationally heavier due to dense connectivity. | • Requires significant computational resources. |

Based on the task of celebrity face classification using the Pins Face Recognition dataset, Xception is the most suitable architecture for the following reasons:

- **Highest Test Accuracy**: Demonstrates superior generalization to unseen data.
- **Balanced Performance**: Avoids overfitting observed in ResNet and underfitting seen in DenseNet.
- **Feature Extraction Efficiency**: Depthwise separable convolutions effectively capture discriminative facial features.
- **Adaptability to Dataset Size:** We used only 50 classes, making it relatively small for deep learning. Xception's architecture is less prone to overfitting compared to ResNet, allowing it to perform better with limited data.
- **Task-Specific Strengths:** Facial recognition requires identifying subtle differences in facial features, such as shape, texture, and expressions. Xception's architecture excels at capturing these fine details due to its efficient convolution operations.