Weather Classification

Info About Dataset

I wanted to collect real, fresh outdoor images with fire classes as part of the Misk Foundation Data Science Immersive project. With MS Bing API, I collected and cleaned up to **18320** images for all classes. Further, I collected data from four Kaggle datasets; their credits are below.

Class	No. of Images
Sunny	6262
Rainy	1931
Foggy	1531
Cloudy	6698
Snowy	1898
Total	**18320**

Preprocessing

1. First, merge the original data with another dataset to increase the number of images in some classes `(rainy, foggy, snowy)` to handle the data imbalance.
2. second remove invalid files (not images), and Make all input images in these formats

`('.jpg', '.jpeg', '.png', '.gif', '.bmp')` , and define the number of color channels

3. split data to train, test, and validate

11720 validated image Train belonging to 5 classes.
3667 validated image Tests belonging to 5 classes.
2932 validated image Validation belonging to 5 classes.

4. one hot Encoding
5. Data augmentation through training to increase the number of images and Normalize pixel values to the range [0, 1], Resize images to match the input shape

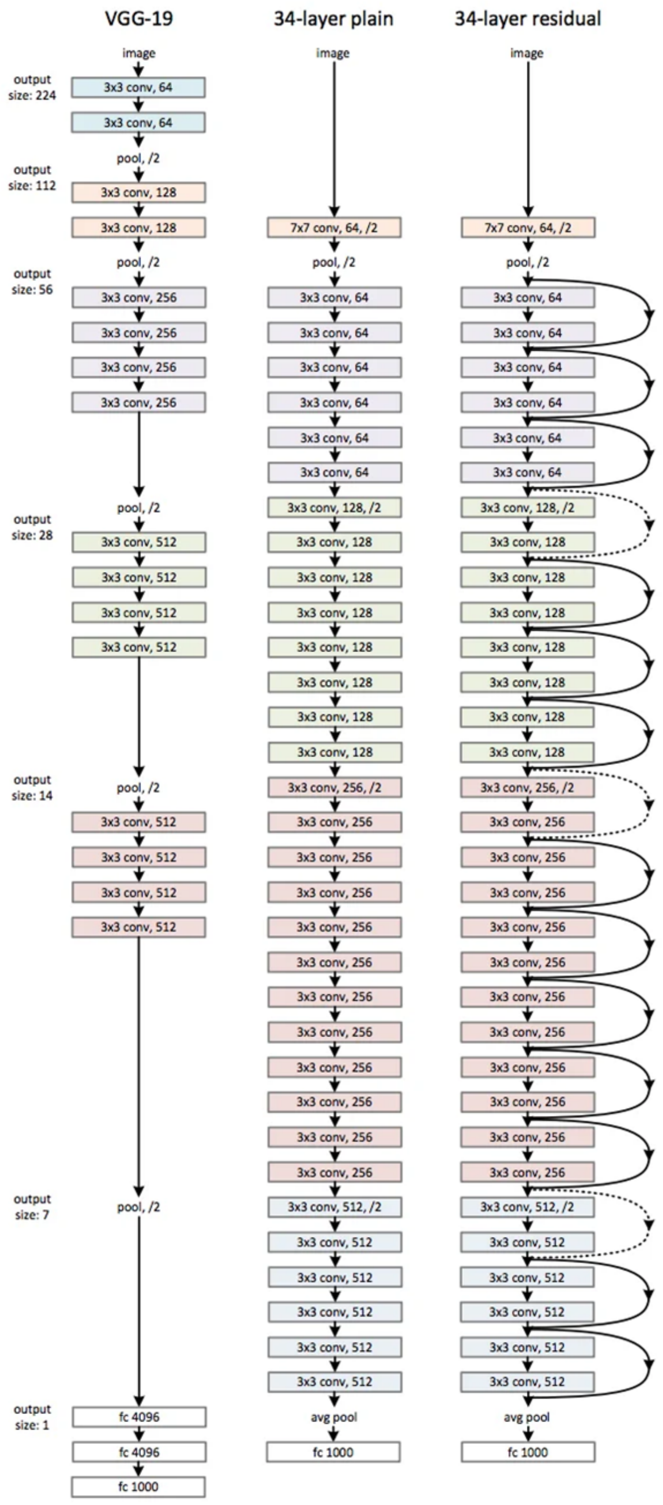
ResNets

What are ResNets and their Types?

ResNets are called Residual Networks. They are a special type of convolutional neural network (CNN) used for tasks such as image recognition. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun first introduced ResNet in 2015 in their paper "Deep Residual Learning for Image Recognition."

Different types of ResNets can be developed based on the depth of the network, such as ResNet-50 or ResNet-152. The number at the end of ResNet specifies the number of layers in the network or how deep the networks are

A ResNet can be called an upgraded version of the VGG architecture. The difference between them is the **skip connections used in ResNets**. In the figure below, we can see the architecture of the VGG as well as the 34-layer ResNet.



The steps of the Architecture :

Step 1: Implementing the Identity Block

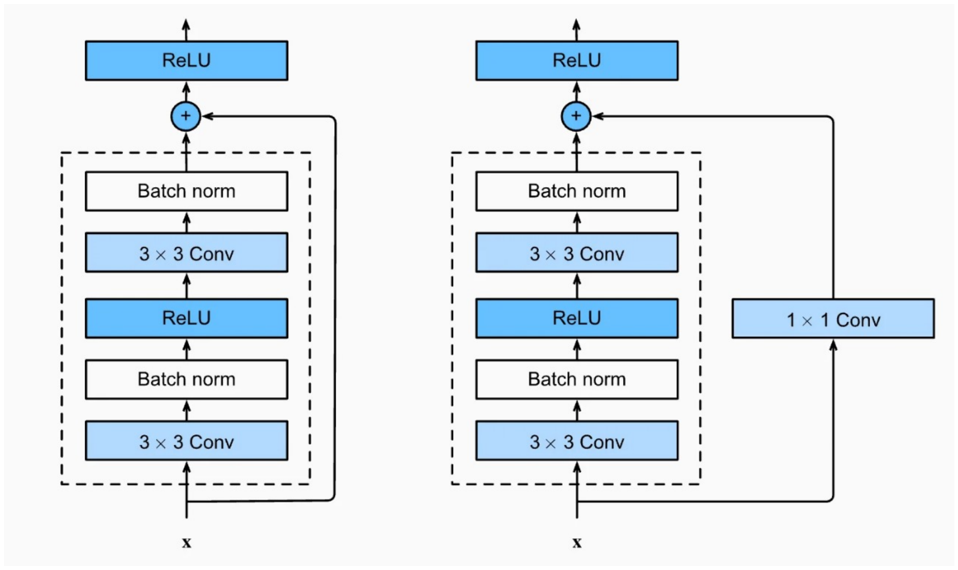
The **identity block** is a key component of ResNet that preserves the input shape while applying transformations. It adds the input directly to the output of the convolutional layers (shortcut connection). This helps the model learn residual mappings and avoids the vanishing gradient problem.

- **Key Layers Used:**
  - **Conv2D:** Applies two 3×3 convolutional layers with filter filters and "same" padding to preserve spatial dimensions.
  - **BatchNormalization:** Normalizes intermediate outputs to speed up training and improve stability.
  - **ReLU Activation:** Adds non-linearity after each convolution.
  - **Add:** Merges the input and output of the convolutional layers (shortcut connection).
- **Input and Output Shape:**
  - Input: Tensor of shape (height, width, channels).
  - Output: Tensor of the same shape as the input.
- **Implementation Details:**
  - The first convolution applies filters with a 3×3 kernel and uses L2 regularization to prevent overfitting.
  - After the second convolution, the input (shortcut) is added to the output of the convolutional layers.
  - Activation is applied after merging.

Step 2: Implementing the Convolutional Block

The **convolutional block** extends the identity block by introducing a convolutional layer in the shortcut path. This layer adjusts the dimensions of the input to match the output when downsampling is needed.

- **Key Layers Used:**
  - **Conv2D (Shortcut Path):** A 1×1 convolution is applied to the shortcut to align its dimensions with the output.
  - **Conv2D (Main Path):** Two 3×3 convolutional layers are applied to the main path for feature extraction.
  - **BatchNormalization:** Used after each convolution to stabilize learning.
  - **ReLU Activation:** Introduced after each batch normalization layer.
  - **Add:** Merges the shortcut path with the output of the main path.
- **Difference from Identity Block:**
  - Includes a **1×1 convolution** in the shortcut to handle dimension mismatches.
  - Uses strides greater than 1 for downsampling (e.g., stride=2).
- **Implementation Details:**
  - The shortcut path uses a 1×1 convolution with the same number of filters as the main path.
  - The main path applies two 3×3 convolutions with filter filters.
  - After merging the paths, the ReLU activation is applied.



Step 3: Building Residual Blocks

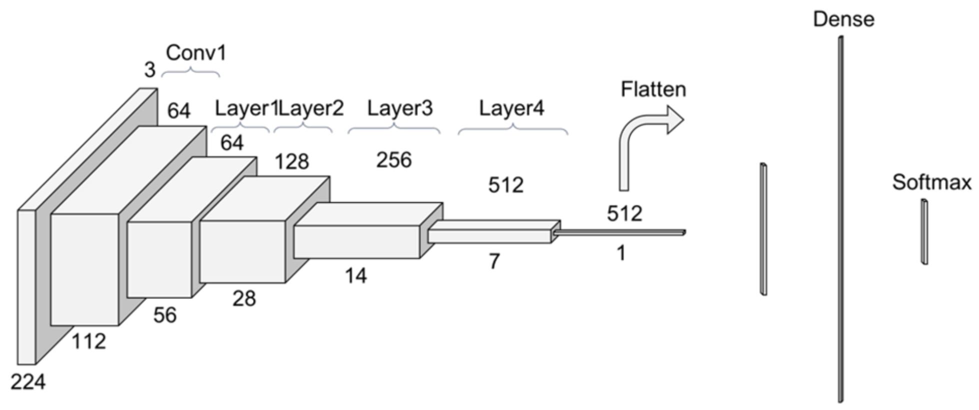
The **residual stack** combines convolutional and identity blocks to form a series of layers, creating depth in the ResNet architecture.

- **Process:**
  - A convolutional block is applied first to adjust dimensions and/or downsample the input.
  - Multiple identity blocks follow, preserving the dimensions while adding residual connections.
- **Configuration:**
  - Each stack corresponds to a stage in ResNet.
  - Example: ResNet34 has four stages with [3, 4, 6, 3] residual blocks per stage.
- **Implementation Details:**
  - For each stage, the number of filters doubles (e.g., 64, 128, 256, 512).
  - The stride for the first convolutional block in each stage is 2, except for the first stage, which has a stride of 1.

Step 4: Building the ResNet Model

The final ResNet model is constructed by stacking residual blocks, followed by global pooling and a fully connected output layer.

- **Process:**
  1. Apply an initial 7×7 convolutional layer with stride 2, followed by batch normalization and ReLU activation.
  2. Downsample the input using max pooling.
  3. Pass the output through four residual stacks, each with a specific number of blocks.
  4. Apply global average pooling to reduce the spatial dimensions.
  5. Add a dense output layer with softmax activation for classification.
- **Additional Details:**
  - Dropout is included after the global average pooling layer to reduce overfitting.
  - L2 regularization is applied to all convolutional and dense layers.



Pros and cons of ResNet :

Pros

1. Residual blocks allow for very deep network architectures without suffering from the vanishing gradient problem, allowing for better performance on complex tasks.
2. The skip connections in ResNet make it easier for the network to learn to represent the identity, which can improve the accuracy of the model.

Cons

1. The increased depth of the network can lead to longer training times and higher computational costs.
2. The skip connections can also make the network more prone to overfitting, requiring careful regularization and tuning.
3. **Memory Usage:** Storing activations for the skip connections and intermediate layers can consume significant GPU memory, especially for larger variants (e.g., ResNet50, ResNet101).
4. **Not Ideal for Small Datasets:** ResNet's deep architecture might lead to overfitting on small datasets if proper regularization techniques are not applied.

Why ResNet is Appropriate for the Project?

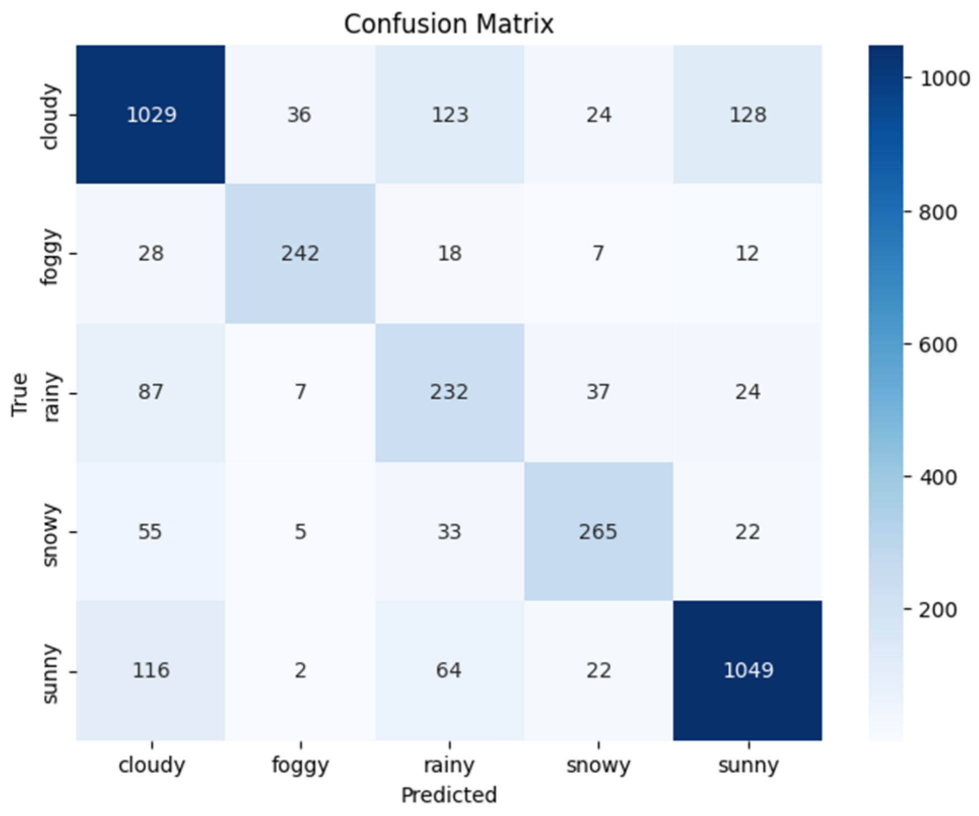
For the task of **weather classification based on images**, ResNet is an appropriate choice for several reasons:

1. **Ability to Learn Complex Visual Features:** Weather conditions (e.g., snow, fog, rain) have distinct visual patterns, such as textures and lighting variations. ResNet's hierarchical feature extraction allows it to capture these fine-grained patterns effectively.
2. **Robustness to Image Variability:** Weather images often have noise and variability in terms of lighting, angles, and background objects. ResNet's skip connections and deep layers help the model generalize better to such variability.
3. **Depth for Better Representations:** Weather classification benefits from deeper networks that can model intricate details. ResNet’s ability to add depth without degradation makes it suitable for this problem.
4. **Generalization Across Classes:** ResNet's architecture is well-suited for multiclass classification problems like this, where each weather type has unique yet overlapping features.
5. **Dataset Suitability:** If the dataset has enough samples per class, ResNet can leverage its depth to learn distinct features for each weather type. Techniques like data augmentation can be used to prevent overfitting in case of

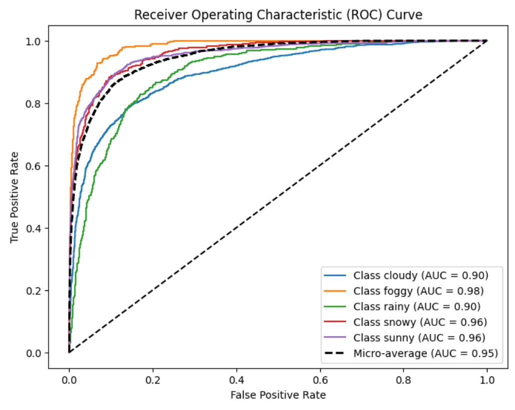
limited data.

6. **Transfer Learning Potential:** If the dataset is small, ResNet pre-trained on large datasets (e.g., ImageNet) can be fine-tuned for weather classification, reducing training time and improving performance.

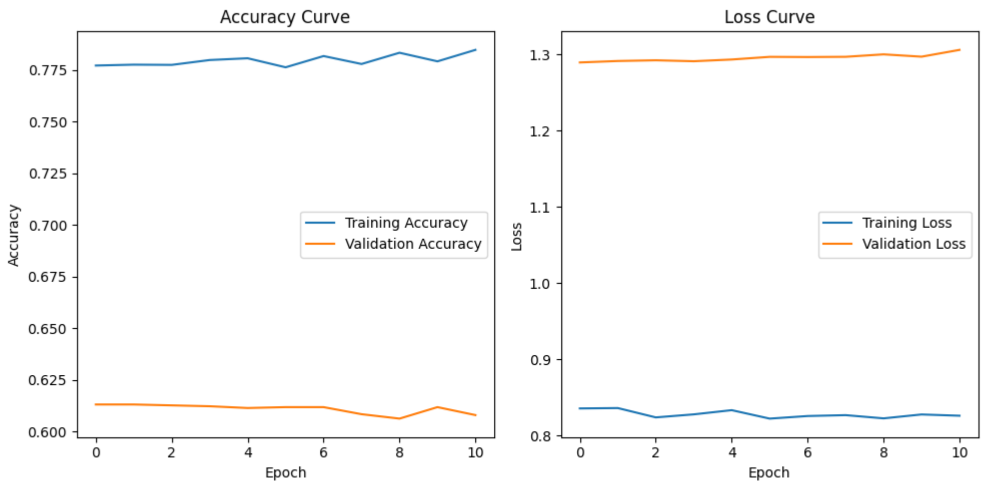
Confusion matrix for ResNet



ROC and AUC for Resnet



Accuracy & loss curve



References:

Paper of ResNet: https://arxiv.org/pdf/1512.03385

https://www.analyticsvidhya.com/blog/2021/08/how-to-code-your-resnet-from-scratch-in-tensorflow/

The website that offers the paper on ResNet model: https://arxiv.org/abs/1512.03385

Explain more of the residual model: https://www.iterate.ai/ai-glossary/resnet-deep-network-architecture#:~:text=While%20ResNet%20is%20effective%20for,overfitting%20if%20not%20carefully%20managed.

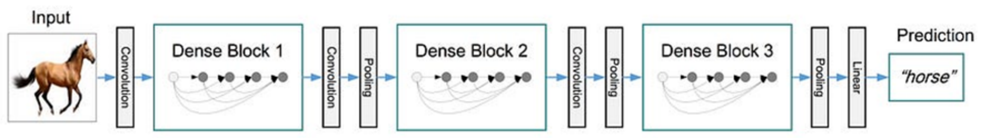
DenseNet

Architectures

Variant	**Layers**	**Parameters**	Dense Block	**Typical Use Cases**
DenseNet-121	121	7.98M	Four dense blocks with layers: [6, 12, 24, 16]	General-purpose image classification, object detection
DenseNet-169	169	14.15M	[6, 12, 32, 32]	Advanced image recognition, medical image analysis
DenseNet-201	201	20.01M	[6, 12, 48, 32]	High-accuracy tasks, detailed feature extraction
DenseNet-264	264	33.34M	[6, 12, 64, 48]	Complex visual tasks, extensive datasets

Why DenseNet is Appropriate for the Project?

DenseNet-121: is a convolutional neural network architecture designed for image classification tasks. It is a popular deep-learning model due to its efficient parameter usage and strong performance across various computer vision benchmarks. Here's an overview of DenseNet-121 and how to use it as a pre-trained model:



The steps of the Architecture DenseNet-121 :

1. Input Layer

- **Input size:** `(224 x 224 x 3)` for RGB images.
- The input images are resized to 224×224 pixels and normalized.

---

2. Initial Convolution and Pooling

1. **Convolution Layer:**
    - Kernel size: `7x7`
    - Stride: `2`
    - Output: `112 x 112 x 64`
    - Performs feature extraction with a larger receptive field.
2. **Batch Normalization:** Normalizes activations.
3. **ReLU Activation:** Applies non-linearity.
4. **Max Pooling:**
    - Kernel size: `3x3`
    - Stride: `2`
    - Output: `56 x 56 x 64`

---

3. Dense Blocks and Transition Layers

DenseNet-121 contains **4 Dense Blocks**, interspersed with **3 Transition Layers**.

Dense Block

- Each block consists of **several bottleneck layers**.
- Every bottleneck layer performs the following sequence:
    1. **Batch Normalization**
    2. **ReLU Activation**
    3. **1×1 Convolution** (reduces feature map size, called "bottleneck")
    4. **Batch Normalization**
    5. **ReLU Activation**
    6. **3×3 Convolution** (extracts features)
    7. **Concatenation:** The output of the current layer is concatenated with all previous layers' outputs within the block.

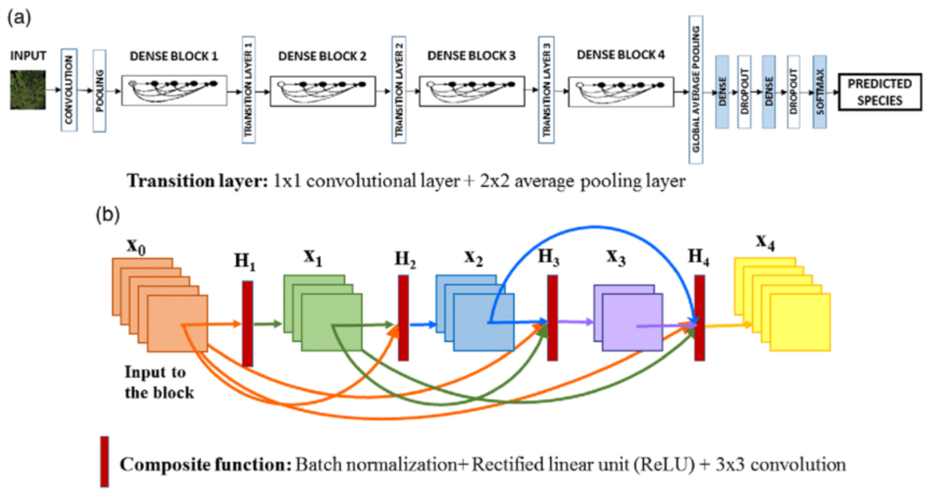
This dense connectivity reduces redundancy and enhances feature reuse.

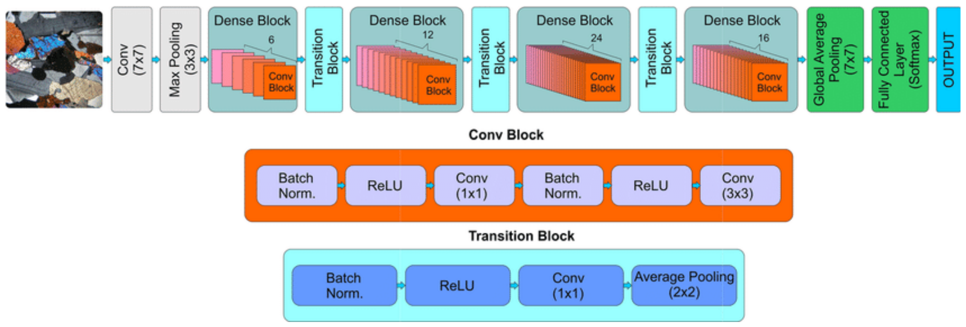
The number of output channels grows linearly with each layer due to the concatenation (controlled by the **growth rate**, typically 32 in DenseNet-121).

Transition Layer

- Each transition layer reduces the feature map size and dimensions.
- Operations:
    1. **Batch Normalization**

2. **1×1 Convolution** (reduces channel count)
3. **Average Pooling**
   - Kernel size: `2x2`
   - Stride: `2`





4. DenseNet-121 Configuration

DenseNet-121 uses the following setup:

Block	Number of Layers	Input Feature Maps	Output Feature Maps
Initial Convolution	1	3	64
Dense Block 1	6	64	256
Transition Layer 1	1	256	128
Dense Block 2	12	128	512
Transition Layer 2	1	512	256
Dense Block 3	24	256	1024
Transition Layer 3	1	1024	512
Dense Block 4	16	512	1024
Fully Connected Layer	1	1024	1000 (ImageNet classes)

5. Classification Layer

After the final dense block:

1. **Global Average Pooling:**
    - Converts the feature maps (1024 channels) into a single vector of size `1x1024`.
2. **Fully Connected Layer:**
    - Maps the 1024-dimensional vector to the output classes (default: 1000 for ImageNet).

---

6. Growth Rate

- Each Dense Block increases the number of channels by the **growth rate** (`k=32` by default).For example:
    - Dense Block 1 starts with 64 channels and grows to 256 after 6 layers (64 + 6 × 32 = 256).

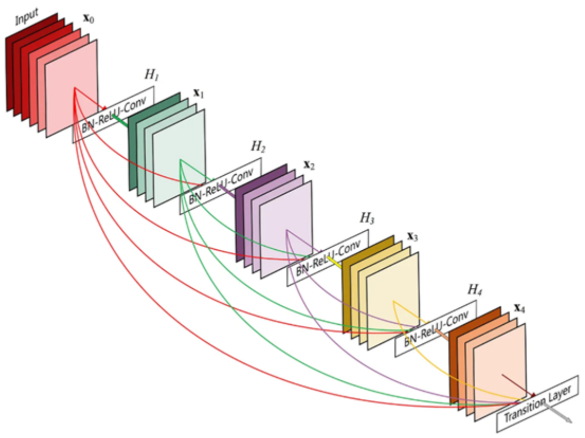


Figure 1: A 5-layer dense block with a growth rate of k = 4

---

7. Efficient Memory Use

- DenseNet uses **feature concatenation** instead of summation (as in ResNet).
- This approach minimizes the number of redundant features and reduces memory usage.

Pros and Cons of DenseNet

Pros

1. Efficient Feature Reuse

- **Dense Connectivity:** Each layer directly accesses the output of all previous layers within a block, promoting **feature reuse**.
    - Reduces the risk of vanishing gradients.
    - Encourages efficient learning of diverse features.

2. Fewer Parameters

- Compared to architectures like ResNet with similar performance, DenseNet uses **fewer parameters** due to the smaller size of individual layers and efficient connectivity.
    - For example, DenseNet-121 has ~8 million parameters, significantly fewer than ResNet-50 (~25 million).

3. Improved Gradient Flow

- DenseNet’s connectivity ensures **strong gradient flow** during backpropagation, making it easier to train deeper networks.

4. Compact Representations

- DenseNet reduces redundancy in features by concatenating outputs rather than summing them (as in ResNet).
    - This makes the network more parameter-efficient.

5. Better Generalization

- The dense connections promote the learning of diverse features, improving the **generalization** of unseen data.

6. Scalability

- DenseNet-121 is lightweight and suitable for applications with limited computational resources, such as mobile and embedded systems.

7. Versatility

- DenseNet can be used for tasks beyond classification, such as **semantic segmentation** and **object detection**, by modifying its architecture.

Cons

1. Computational Complexity

- **High Memory Usage:**
    - Feature concatenation causes a **growth in memory consumption**, as every layer retains the outputs of all preceding layers.
    - Training on high-resolution images or very deep DenseNets can be computationally expensive.

2. Slower Inference

- Dense connectivity increases the number of **feature maps passed to each layer**, which can lead to slower inference compared to models like ResNet.

3. Not Always Optimal for Very Large Models

- The dense connections become less effective as the network gets extremely deep (e.g., DenseNet-264), where the memory and computational overhead can outweigh the benefits.

4. Limited to Small Growth Rate

- A small **growth rate (k)** (e.g., 32 in DenseNet-121) limits the model's capacity. Increasing it for better performance can lead to excessive memory usage.

5. Challenging Customization

- Customizing DenseNet for very specific tasks can be harder due to its unique connectivity pattern. For example:
    - Removing or adding layers within a dense block can break the network's structure.

6. Diminishing Returns with Larger Models

- The performance gain from increasing depth is less significant compared to ResNet, making DenseNet less appealing for very deep architectures.

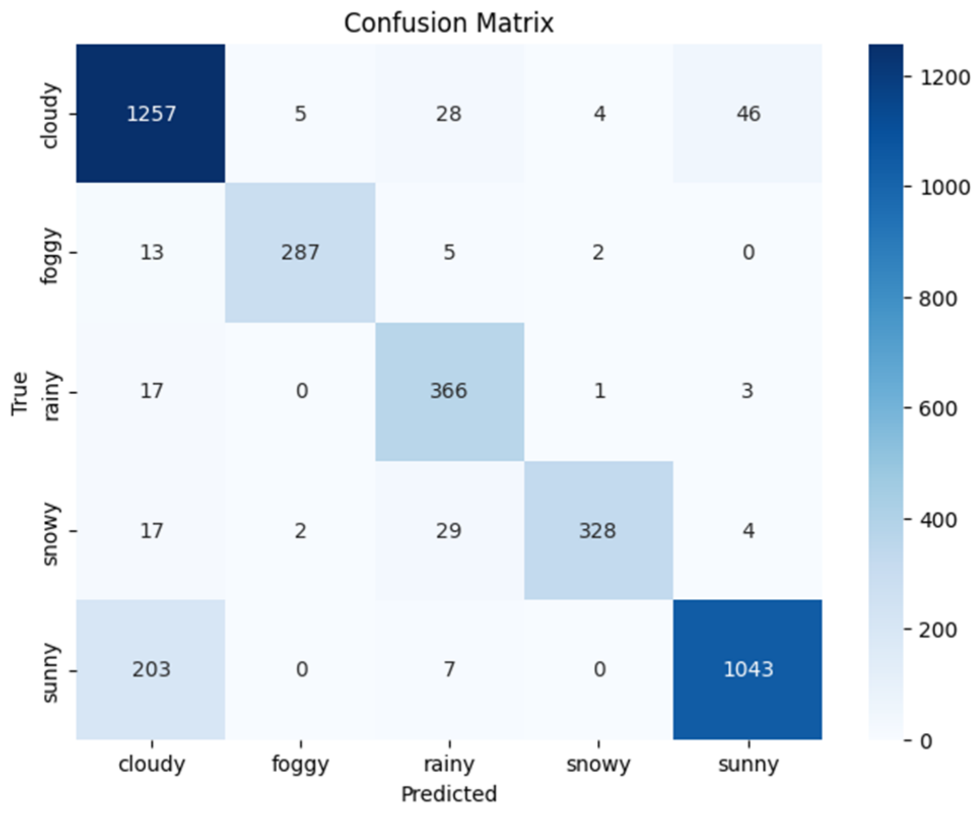
Load Pretrained model:

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

Arguments

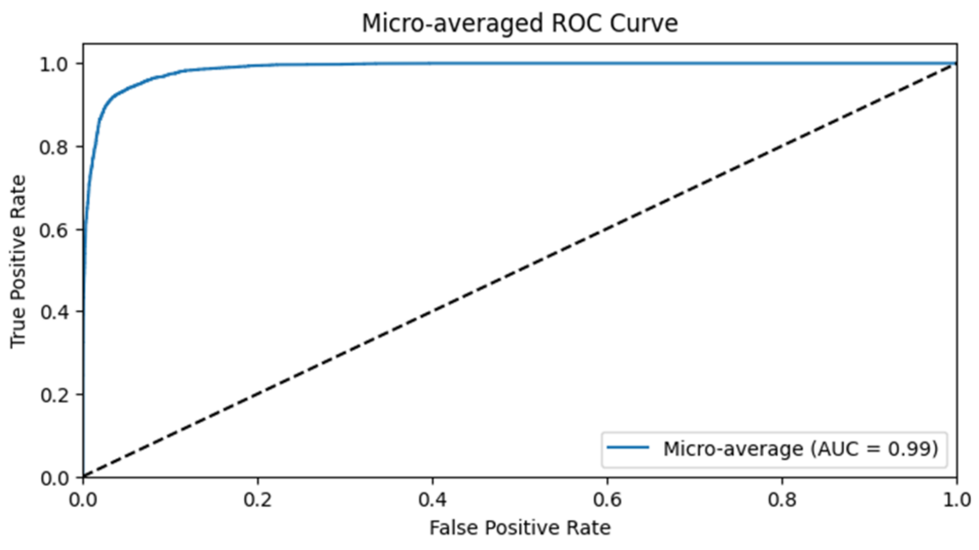
- Weights: one of None (random initialization), "imagenet" (pre-training on ImageNet), or the path to the weights file to be loaded. input_tensor: optional Keras tensor (i.e. output of layers.Input()) to use as image input for the model.
- **include_top**: whether to include the fully connected layer at the top of the network.
- **input_shape**: optional shape tuple, only to be specified if include_top is False (otherwise the input shape has to be (224, 224, 3) (with 'channels_last' data format)

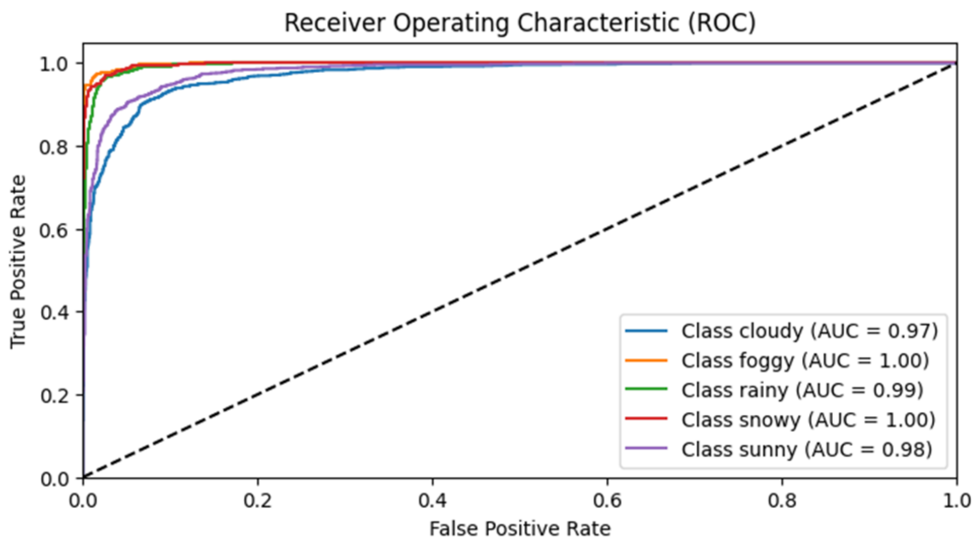
Confusion matrix for DenseNet



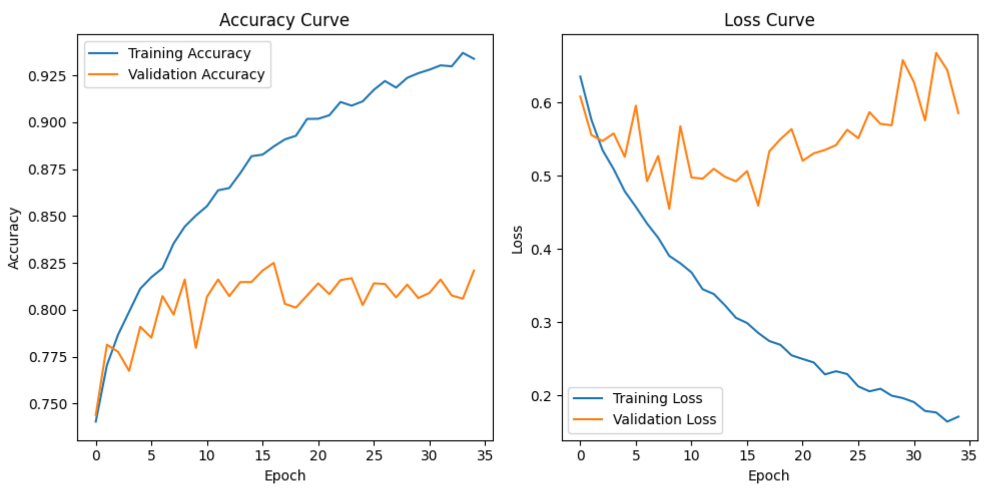
	cloudy	foggy	rainy	snowy	sunny
cloudy	1257	5	28	4	46
foggy	13	287	5	2	0
rainy	17	0	366	1	3
snowy	17	2	29	328	4
sunny	203	0	7	0	1043

ROC and AUC for DenseNet





Accuracy & loss curve



Reference:

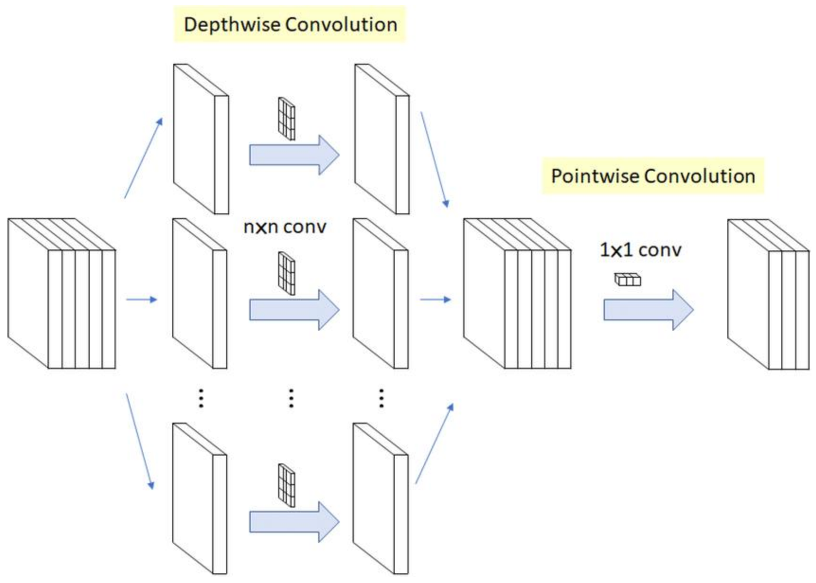
Paper of DenseNet: https://paperswithcode.com/lib/torchvision/densenet

https://keras.io/api/applications/densenet/#densenet121-function

---

Xception

Xception (**Extreme Inception**) is a convolutional neural network (CNN) architecture introduced by François Chollet in 2017. It builds on the Inception model, using **depthwise separable convolutions** to make the network more efficient and powerful. The key idea is to explicitly separate the learning of **spatial features** from **cross-channel features**, leading to a lightweight and high-performing architecture.



The steps of the architecture :

The Xception model consists of **36 convolutional layers** organized into three main flows:

1. Entry Flow

- Initial feature extraction and downsampling.
- **Steps**:
  - Standard Convolution: A `3x3` convolution layer for initial feature extraction.
  - Depthwise Separable Convolutions: A sequence of separable convolutions with ReLU activations.
  - Max Pooling: Reduces the spatial dimensions of feature maps.
  - Residual Connections: Ensures stable learning.

2. Middle Flow

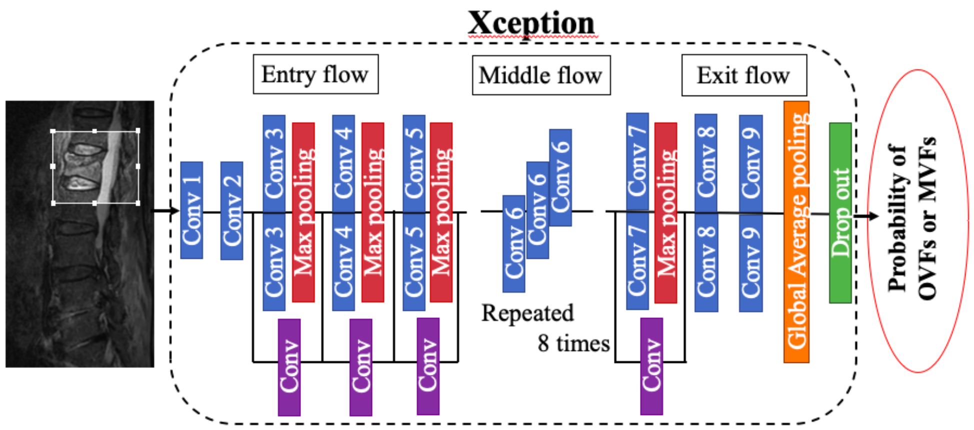
- The core feature extractor, repeats the same module 8 times.
- **Steps**:
  - Depthwise Separable Convolutions: Each module uses separable convolutions to refine features.
  - No downsampling: Retains spatial dimensions for deeper feature learning.

3. Exit Flow

- Aggregates high-level features and performs downsampling.
- **Steps**:
  - Depthwise Separable Convolutions: Further refinement of features.
  - Global Average Pooling: Reduces the feature map to a single vector per feature channel.
  - Dense Layer: Maps the features to the number of output classes.

Global Architecture

Flow	Number of Layers	Key Operations
Entry Flow	12	Convolution, Separable Convolutions, Pooling
Middle Flow	8 (repeated 8x)	Separable Convolutions
Exit Flow	16	Separable Convolutions, Pooling, Fully Connected



Parameters

1. **Input Shape:**
    - Default: `(299, 299, 3)` or (256,256,3) for RGB images.
    - Can be modified for smaller/larger images.
2. **Output Classes:**
    - Default: 1000 classes (ImageNet).
    - Customizable for specific tasks.
3. **Number of Parameters:**
    - Xception has approximately **22.9M parameters**, fewer than Inception v3 (~24M).

Pros and Cons of Xception

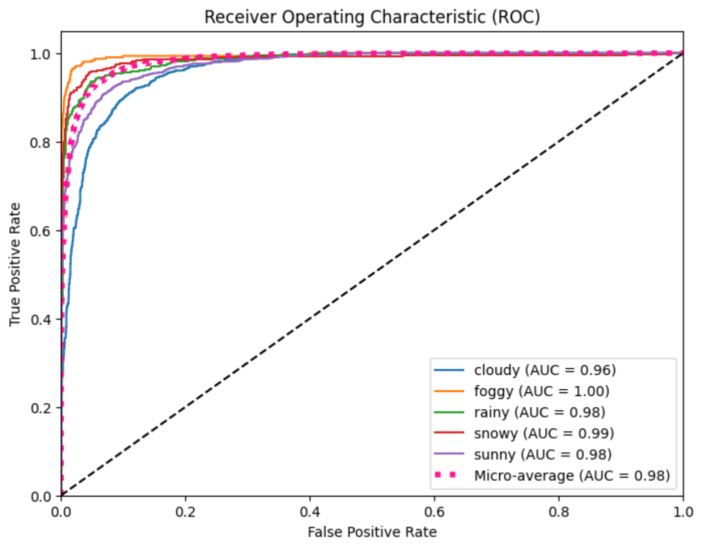
pros

1. Efficiency:
    - Uses fewer parameters and computations compared to traditional CNNs (e.g., Inception).
    - Suitable for tasks with limited computational resources.
2. Performance:
    - Achieves competitive or superior accuracy on datasets like ImageNet.
3. Feature Extraction:
    - Depthwise separable convolutions improve feature extraction efficiency.
4. Flexibility:
    - Can be adapted for classification, segmentation (e.g., DeepLabv3), and detection tasks.

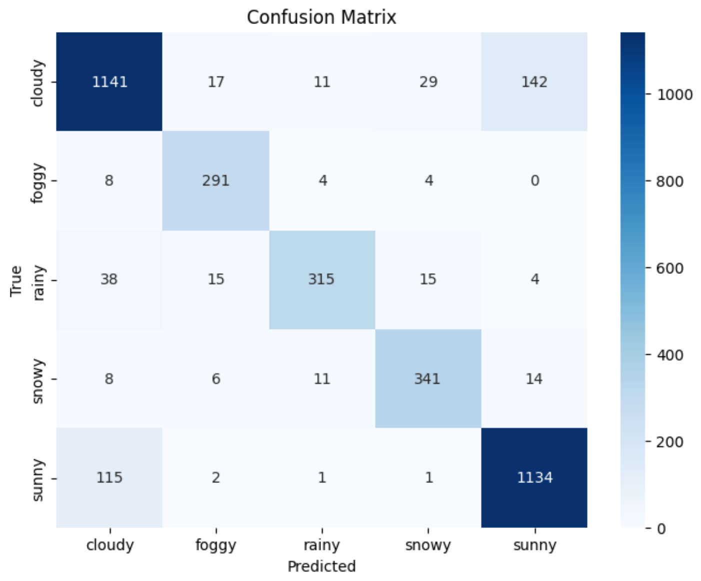
cons

1. Memory Usage:
    - Despite its efficiency, Xception can still consume significant memory due to its depth.
2. Overfitting:
    - May overfit on small datasets without proper regularization.
3. Input Size:
    - Optimized for larger input sizes (e.g., `299x299`). Using smaller inputs may degrade performance.

ROC and AUC for xception

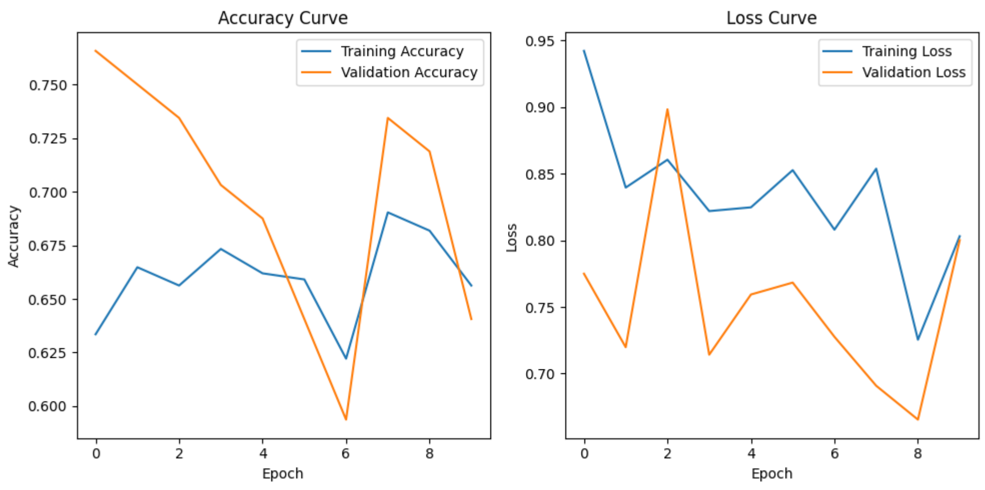


Confusion matrix for xception



True \ Predicted	cloudy	foggy	rainy	snowy	sunny
cloudy	1141	17	11	29	142
foggy	8	291	4	4	0
rainy	38	15	315	15	4
snowy	8	6	11	341	14
sunny	115	2	1	1	1134

Accuracy& loss curve



Reference:

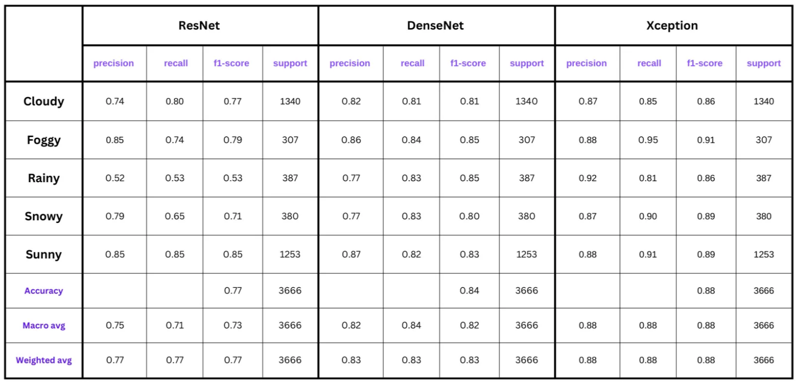
Paper of Xception: https://arxiv.org/abs/1610.02357

https://keras.io/api/applications/xception/

https://www.researchgate.net/figure/Architecture-of-the-Xception-deep-CNN-model_fig2_351371226

https://medium.com/@saba99/xception-cd1adc84290f

The Results for the 3 Models



	ResNet				DenseNet				Xception			
	precision	recall	f1-score	support	precision	recall	f1-score	support	precision	recall	f1-score	support
Cloudy	0.74	0.80	0.77	1340	0.82	0.81	0.81	1340	0.87	0.85	0.86	1340
Foggy	0.85	0.74	0.79	307	0.86	0.84	0.85	307	0.88	0.95	0.91	307
Rainy	0.52	0.53	0.53	387	0.77	0.83	0.85	387	0.92	0.81	0.86	387
Snowy	0.79	0.65	0.71	380	0.77	0.83	0.80	380	0.87	0.90	0.89	380
Sunny	0.85	0.85	0.85	1253	0.87	0.82	0.83	1253	0.88	0.91	0.89	1253
Accuracy			0.77	3666			0.84	3666			0.88	3666
Macro avg	0.75	0.71	0.73	3666	0.82	0.84	0.82	3666	0.88	0.88	0.88	3666
Weighted avg	0.77	0.77	0.77	3666	0.83	0.83	0.83	3666	0.88	0.88	0.88	3666