# Alphabet Classification

**1.Introduction :**

Alphabet Classification is a critical area of computer vision with applications ranging from automated document processing to accessibility tools. This project aims to develop an effective solution for recognizing handwritten characters from a dataset that combines Arabic and English alphabets.

Deep learning architectures have significantly advanced the field of character recognition, enabling high accuracy and efficiency. This documentation focuses on three key architectures—ResNet, DenseNet121, and Xception—each of which brings unique ideas and innovations to the task of image classification.

* **ResNet**: ResNet (Residual Network) introduces residual connections, allowing for the training of very deep networks by addressing the vanishing gradient problem. This innovation enables the model to learn identity mappings and stack deeper layers without degradation in performance.
* **DenseNet121**: DenseNet (Densely Connected Network) leverages dense connections where each layer is connected to every other layer in a feed-forward fashion. This architecture promotes feature reuse, reduces the number of parameters, and enhances efficiency, making it a robust choice for image classification tasks.
* **Xception**: Xception (Extreme Inception) is a depthwise separable convolution-based network that builds upon the principles of the Inception architecture. By decoupling spatial and cross-channel correlations, Xception achieves computational efficiency while maintaining high accuracy, especially on large-scale datasets.

**2.Steps For Implementation**

1)**Data Preprocessing Steps**

1. **Dataset Loading**:  
   The English and Arabic handwritten character datasets were loaded separately for preprocessing.
2. **Reshaping the English Dataset**:  
   The English dataset, originally in a 28x28 dimension, was reshaped to match the size of the Arabic dataset (32x32 pixels).
3. **Adjusting Dimensions for Model Compatibility**:  
   The dimensions of the Arabic dataset were adjusted to match the expected input format for the model. This involved reshaping the data to a 4D format, ensuring both datasets were compatible for concatenation.
4. **Label Adjustment**:  
   To avoid overlap between the Arabic and English labels, the Arabic labels were incremented by 27. This shift ensures that the two sets of labels remain distinct during training.
5. **Dataset Concatenation**:  
   After adjusting the labels and dimensions, the English and Arabic datasets were concatenated to form a unified dataset.
6. **Data Splitting**:  
   The concatenated dataset was split into training and validation sets, with 20% allocated for validation.
7. **TensorFlow Dataset Creation**:  
   TensorFlow datasets were created from the training and validation sets, enabling efficient data loading and manipulation.
8. **Dataset Preparation**:  
   The datasets were shuffled, batched, and prefetched to improve training performance by ensuring efficient data feeding during model training.
9. **Image Resizing and Label Encoding**:  
   The images were resized to 224x224 pixels and converted to RGB format, ensuring compatibility with the model's input requirements. Additionally, the labels were one-hot encoded to provide the required format for multi-class classification.
10. **Output Shape**:  
    After preprocessing, the dataset's final shape is as follows:
    * Batch Images Shape: (100, 224, 224, 3)
    * Batch Labels Shape: (100, 54)

**2) ResNet Architecture :**

The ResNet(Residual Network) model is a deep learning architecture designed to overcome the vanishing gradient problem in very deep neural networks. It introduces **residual connections**, also called **skip connections**, which allow the network to skip one or more layers during the forward pass.

**Key Concepts:**

1. **Residual Connections**:  
   Instead of each layer learning a direct transformation of the input, ResNet layers learn the residual (difference) between the input and the output. This helps gradients flow more easily during backpropagation, allowing for the training of much deeper networks (hundreds or even thousands of layers).
2. **Identity Mapping**:  
   These skip connections create an identity mapping, where the input is passed directly to later layers, bypassing the intermediate layers. This helps mitigate the issue of gradients becoming very small and helps the network learn more effectively.
3. **Key Features**:

-Residual blocks with skip connections

-Convolutional layers with Batch Normalization and ReLU activations

-Global Average Pooling followed by a Fully Connected layer

We made our ResNet Model from Scratch it has 34-layer with 4 stages of residual blocks.

**1. ResidualBlock Class**

This is the building block of the ResNet architecture, implementing a residual connection.

* **\_\_init\_\_()**:
  + Initializes two convolutional layers with batch normalization.
  + If downsample=True, it creates a downsampling path using a 1x1 convolution to adjust the dimensions of the input to match the residual block’s output.
  + Otherwise, the identity mapping (no operation) is applied.
* **call()**:
  + Defines the forward pass.
  + The input goes through the convolutional layers (conv1 and conv2) with ReLU activation and batch normalization.
  + The residual (shortcut connection) is added to the output of the second convolution.
  + A final ReLU activation is applied.
* **Serialization Methods**:
  + get\_config() and from\_config() allow the block to be serialized and deserialized, enabling model saving and loading.

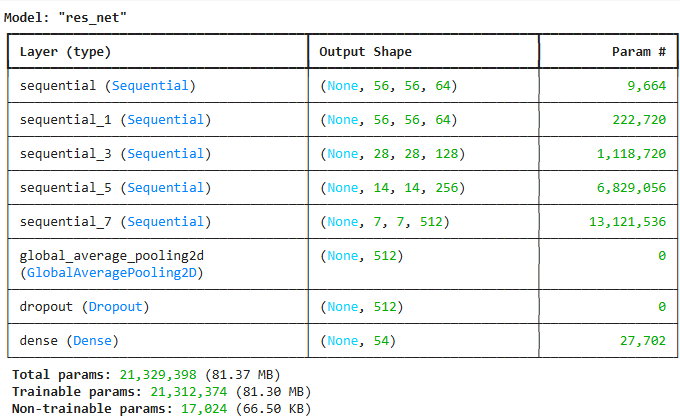
**2. ResNet Class**

The overall ResNet model that stacks multiple **ResidualBlock** layers.

* **\_\_init\_\_()**:
  + Initializes:
    - **Initial Convolution**: A sequence of layers (Conv2D, BatchNorm, ReLU, MaxPooling) that processes the input before feeding it into the residual blocks.
    - **Stages (Stage 1 to Stage 4)**:  
      These are sequences of residual blocks created using \_build\_stage().  
      Each stage increases the number of filters and performs downsampling (stride=2) for stages 2–4.
    - **Classification Head**:  
      Includes:
      * GlobalAveragePooling: Reduces feature maps to a single vector per channel.
      * Dropout: Adds regularization.
      * Fully Connected Layer: Outputs predictions for the specified number of classes.
* **\_build\_stage()**:
  + Constructs a stage by stacking residual blocks.
  + The first block of the stage performs downsampling if strides != 1, while subsequent blocks maintain the same dimensions.
* **call()**:
  + Defines the forward pass for the ResNet model:
    - The input is processed by the **initial convolutional layers**.
    - Passes through the four stages of residual blocks.
    - The output is globally pooled, regularized with dropout, and classified with the fully connected layer.
* **Serialization Methods**:
  + get\_config() and from\_config() allow the model to be saved and reloaded.
* **build()**:
  + Builds the model by defining the input shape. This ensures that all layers are initialized properly before training.

**3. Build and Compile the Model**

* **build\_resnet()**:
  + A utility function to create a ResNet model. By default, it builds a ResNet architecture with [3, 4, 6, 3] blocks per stage .
  + Input Parameters:
    - num\_classes: The number of output classes (54 in your case).
    - resnet\_type: The ResNet variant (currently, only "ResNet" is supported).
* **Instantiate the Model**:
  + A ResNet model is built using build\_resnet() with:
    - num\_classes=54 for 54 output classes (matching your problem's requirements).
    - Input shape of (None, 224, 224, 3) for RGB images resized to 224x224 pixels.



A diagram of a graph

Description automatically generated with medium confidence

**3) DenseNet Architecture : (**DenseNet-121)

**- Dense Connectivity:**  
 In DenseNet, each layer receives input from all preceding layers in a dense block. This means that instead of just passing the output of one layer to the next, all previous layer outputs are concatenated and passed to the next layer. This approach helps preserve more features and allows the model to learn more complex representations.

**- Improved Gradient Flow:**  
 DenseNet’s architecture reduces the problem of vanishing gradients by making sure that the gradients are more easily propagated back through the network. This leads to better training of deeper networks.

**- Parameter Efficiency:**  
 DenseNet-121 uses fewer parameters than traditional CNNs because it reuses features across layers. Instead of learning separate feature maps for each layer, DenseNet concatenates feature maps from previous layers, reducing redundancy and computational cost while improving accuracy.

**- Dense Blocks and Transition Layers**:  
The network is built from dense blocks, where each block contains several layers connected in a dense pattern. Between these blocks, transition layers are used to reduce the number of feature maps, typically using convolution and pooling operations. These transition layers help the network manage computational complexity.

DenseNet-121, used with transfer learning, is a densely connected convolutional network where each layer is connected to every other layer in a feed-forward fashion.

* Layers: 121 layers
* Key Features:
  + Dense blocks with feature reuse
  + Transition layers with pooling
  + Dropout for regularization

1. **Loading the Pretrained DenseNet121 Model:**

* **DenseNet121**: This is a pre-trained DenseNet121 model, which is a deep convolutional neural network that was originally trained on the ImageNet dataset.
* **weights='imagenet'**: The model is initialized with weights trained on ImageNet, which has been trained on millions of images across 1000 classes.
* **include\_top=False**: This excludes the final classification layer (top) of the DenseNet121 model, so we can replace it with our own custom layers for the specific task at hand.
* **input\_shape=(224, 224, 3)**: This specifies that the input image size should be 224x224 pixels with 3 color channels (RGB).

1. **Freezing the Base Model Layers:**

trainable=False: This freezes the weights of the DenseNet121 base model, meaning these weights will not be updated during training

1. **Adding Custom Layers on Top of the Pretrained Base:**

* **Dense\_base\_model**: The pre-trained DenseNet121 model (without its final classification layers) is added as the first layer.
* **GlobalAveragePooling2D()**: This layer performs global average pooling, which reduces the feature maps to a single value per feature map. This helps reduce the size of the model and avoid overfitting.
* **Fully Connected Layers**: Three dense (fully connected) layers are added, each followed by:
* **Dense Layer**: These layers are used for learning high-level features. Each layer has a different number of neurons (1024, 512, 256).
* **activation='relu'**: ReLU activation function is used for non-linearity.
* **kernel\_initializer='he\_normal'**: The weights are initialized using the He normal initializer, which is commonly used for ReLU activation functions to prevent vanishing gradients.
* **BatchNormalization()**: This normalizes the activations of the previous layer, which helps to stabilize training and accelerate convergence.
* **Dropout**: Dropout is used for regularization to prevent overfitting. The dropout rate decreases with each layer (50%, 40%, 30%).
* **Dense(num\_classes, activation='softmax')**: The final output layer has num\_classes units (corresponding to the number of classes for classification) with a softmax activation to output probabilities for each class.

**A screenshot of a computer program

Description automatically generated**

**A diagram of a network

Description automatically generated**

**4) Xception Architecture :**

The Xception model (Extreme Inception) is a deep learning architecture that builds on the Inception model but replaces its convolutional layers with depthwise separable convolutions, a more efficient type of convolution. This design allows Xception to achieve excellent performance with fewer parameters.

Key Concepts:

1. Depthwise Separable Convolutions:  
   Xception uses depthwise separable convolutions, which split a standard convolution into two steps:
   * Depthwise Convolution: Applies a single filter per input channel to capture spatial patterns.
   * Pointwise Convolution: Applies a 1x1 convolution across all channels to combine the spatial patterns learned from the depthwise convolution.  
     This results in fewer parameters and computations compared to traditional convolutions.
2. Extreme Version of Inception:  
   The model can be seen as an "extreme" version of the Inception architecture, as it simplifies Inception modules by fully factorizing the convolutions into depthwise separable ones.
3. Efficient Feature Learning:  
   By decoupling spatial and channel-wise feature learning, Xception improves efficiency and enables deeper architectures while maintaining strong performance on tasks like image classification.
4. Residual Connections:  
   Xception also incorporates residual connections to improve gradient flow during training, similar to ResNet.
5. **Load Pretrained Xception Model :**

* **Xception**: This is a deep convolutional neural network based on depthwise separable convolutions. The model is pre-trained on **ImageNet**, which is a large image dataset commonly used in computer vision tasks.
* **weights="imagenet"**: This initializes the model with pre-trained weights from ImageNet.
* **include\_top=False**: Excludes the top (final) classification layer of the Xception model, allowing you to add your custom layers.
* **input\_shape=(224, 224, 3)**: Specifies the shape of the input images as 224x224 pixels with 3 color channels (RGB).

1. **Freezing All Layers Except the Last 20:**

* This code freezes all layers in the Xception model except for the last 20 layers.
* **Freezing layers**: By setting trainable = False, you prevent these layers from being updated during training. This helps retain the learned features from ImageNet.

1. **Adding Custom Layers:**

* **GlobalAveragePooling2D**: This reduces the spatial dimensions of the output from the base model, averaging over all feature maps to produce a 1D vector per feature map. It helps to prevent overfitting and reduces the number of parameters.
* **Fully Connected (Dense) Layers**:
* The first **Dense layer** has 1024 units with ReLU activation. It acts as a dense layer that learns high-level features.
* **Dropout** with a rate of 0.5 is applied after the first and second Dense layers to prevent overfitting.
* The second **Dense layer** has 512 units, and **Dropout** is applied again (with a rate of 0.5).
* **Output Layer**:
* The output layer has num\_classes neurons, where num\_classes is 54 in your case. The activation function used here is **softmax**, which is typically used for multi-class classification problems as it outputs probabilities for each class.

1. **Creating the Final Model:**

Model(inputs=Xception\_base\_model.input, outputs=predictions): This combines the pre-trained Xception base model with the new custom layers you added. The final model has the input shape of (224, 224, 3) and the output layer that predicts one of the 54 classes.

1. **Compiling the Model:**

* **optimizer=Adam(learning\_rate=1e-4)**: Adam is the optimizer used here, and a low learning rate of 1e-4 is specified. Fine-tuning requires smaller learning rates to avoid drastic updates to the pre-trained weights.
* **loss="categorical\_crossentropy"**: This loss function is used for multi-class classification problems where the target labels are one-hot encoded.
* **metrics=["accuracy"]**: The model will evaluate and report accuracy during training and validation.

A screenshot of a computer program

Description automatically generated

**3.Comparison Between the 3 models** :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Aspect** | **ResNet-34** |  | **DenseNet-121** | **Xception** |
| **Training Approach** | Built from scratch |  | Transfer learning with pretrained weights | Transfer learning with pretrained weights |
| **Dataset Adaptation** | Requires more tuning on this dataset |  | Adapts quickly due to feature reuse | High adaptability with advanced feature extraction |
| **Feature Utilization** | Moderate, relies on deeper layers |  | Excellent due to dense connections | Excellent due to depthwise separable convolutions |
| **Computational Cost** | Higher due to training from scratch |  | Moderate, efficient parameter utilization | Low, computationally efficient |
| **Suitability** | Good baseline for custom implementations |  | Effective for large datasets with varied features | Highly suitable for datasets with complex patterns |

**Pros and Cons of Each Model**

**ResNet-71**

* **Pros**:
  + Customizability since it is built from scratch
  + Effectively handles vanishing gradient problem with residual blocks
* **Cons**:
  + Requires more computational resources for training
  + Slower convergence compared to pretrained models

**DenseNet-121**

* **Pros**:
  + Promotes feature reuse, leading to efficient parameter utilization
  + Transfer learning accelerates training
  + High accuracy due to dense connectivity
* **Cons**:
  + Higher memory consumption due to concatenated features
  + Increased complexity can make fine-tuning challenging

**Xception**

* **Pros**:
  + Efficient computational performance with depthwise separable convolutions
  + Pretrained weights enhance adaptability to diverse datasets
  + Achieves the highest accuracy among the models
* **Cons**:
  + Complex architecture can be harder to interpret
  + Requires careful preprocessing and tuning for optimal results

**References :**

**ResNet :**

-Paper Title: *Deep Residual Learning for Image Recognition*

Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Link: <https://arxiv.org/abs/1512.03385>

-Image classification based on RESNET (<https://www.researchgate.net/publication/346212393_Image_classification_based_on_RESNET>)

**DenseNet :**

-Paper Title: *Densely Connected Convolutional Networks*

Authors: Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger

Link: <https://arxiv.org/abs/1608.06993>

-DenseNet121 (<https://paperswithcode.com/lib/timm/densenet>)

**Xception :**

-Paper Title: *Xception: Deep Learning with Depthwise Separable Convolutions*

Authors: François Chollet

Link: <https://arxiv.org/abs/1610.02357>

-Xception (<https://paperswithcode.com/model/xception?variant=xception-1>)