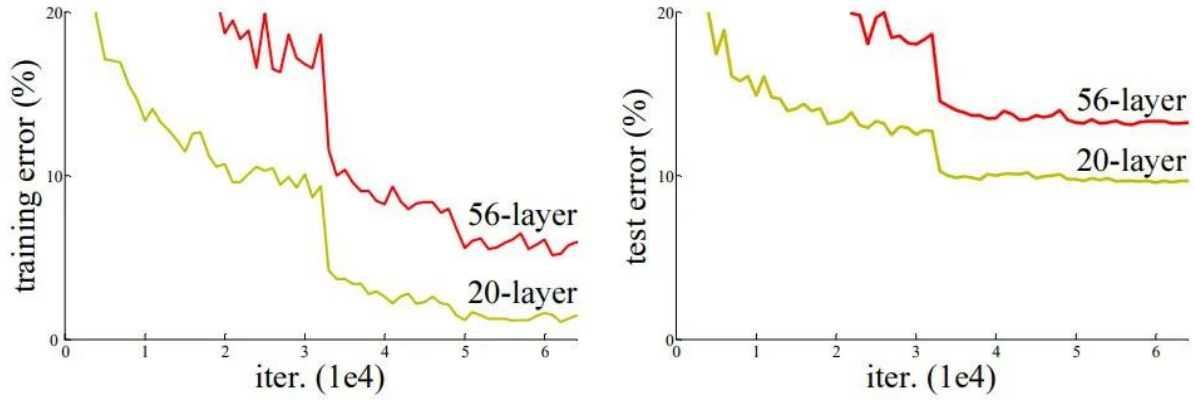
Deep learning and neural networks documentation (alphabet classification)

Resnet :

Every consecutive winning architecture uses more layers in a deep neural network to lower the error rate after the first CNN-based architecture (AlexNet) that won the ImageNet 2012 competition. This is effective for smaller numbers of layers, but when we add more layers, a typical deep learning issue known as the Vanishing/Exploding gradient arises. This results in the gradient becoming zero or being overly large. Therefore, the training and test error rate similarly increases as the number of layers is increased.

We can see from the following figure that a 20-layer CNN architecture performs better on training and testing datasets than a 56-layer CNN architecture. The authors came to the conclusion that the error rate is caused by a vanishing/exploding gradient after further analysis of the error rate.

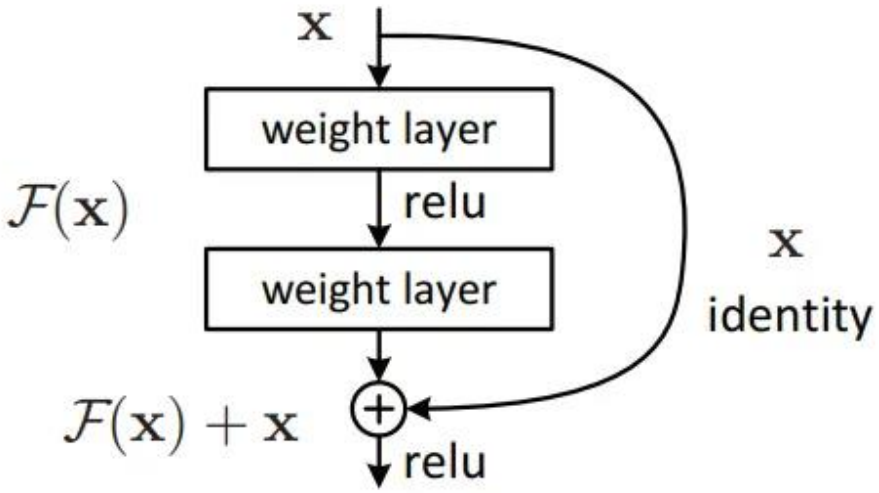


A novel architecture called **Residual Network** was launched by Microsoft Research experts in 2015 with the proposal of **ResNet.**

The Residual Blocks idea was created by this design to address the issue of the vanishing/exploding gradient. We apply a method known as skip connections in this network. The skip connection bypasses some levels in between to link-layer activations to subsequent layers. This creates a leftover block. These leftover blocks are stacked to create resnets.

The strategy behind this network is to let the network fit the residual mapping rather than have layers learn the underlying mapping. Thus, let the network fit instead of using, say, the initial mapping of H(x),

The benefit of including this kind of skip link is that regularisation will skip any layer that degrades architecture performance. As a result, training an extremely deep neural network is possible without encountering issues with vanishing or expanding gradients.



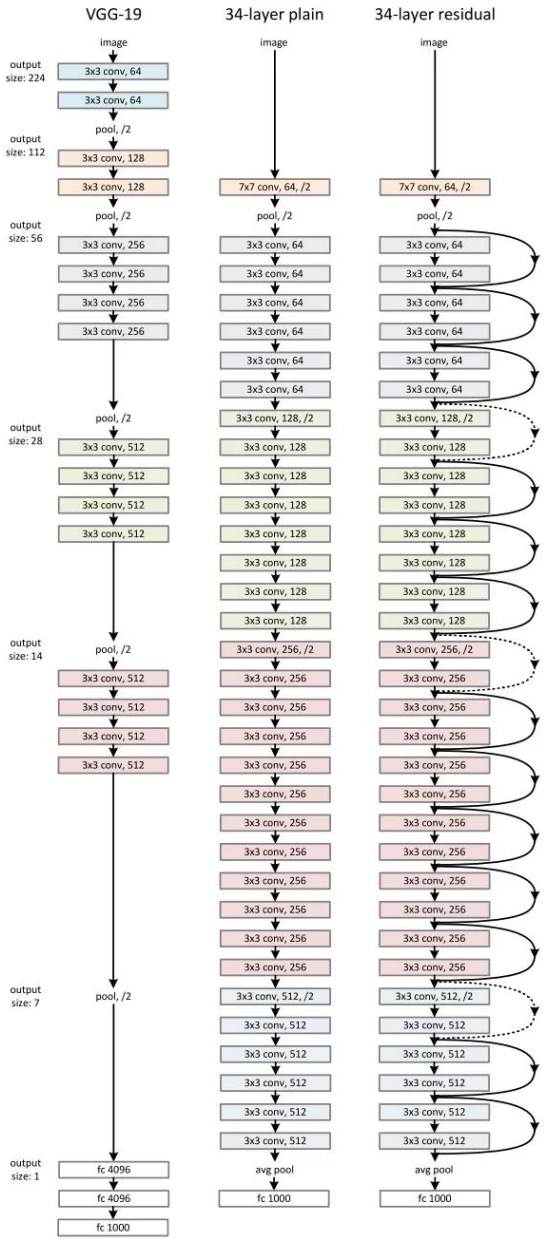
Why do we need ResNet?

We stack extra layers in the Deep Neural Networks, which improves accuracy and performance, often in order to handle a challenging issue. The idea behind layering is that when additional layers are added, they will eventually learn features that are more complicated. For instance, when recognising photographs, the first layer may pick up on edges, the second would pick up on textures, the third might pick up on

objects, and so on. However, it has been discovered that the conventional Convolutional neural network model has a maximum depth threshold.

ResNet Architecture

The VGG-19-inspired 34-layer plain network architecture used by ResNet is followed by the addition of the shortcut connection. The architecture is subsequently transformed into the residual network by these short-cut connections, as depicted in the following figure:



ResNet Architecture for Alphabetic Classification

1. Introduction

This document describes the implementation of a **ResNet (Residual Network)** architecture used for alphabetic character classification. The goal is to classify images containing alphabetic characters into one of 54 classes.

2. Understanding ResNet

ResNet (introduced by He et al.) solves the **vanishing gradient problem** in deep neural networks by using **residual learning**. In ResNet, shortcuts (identity connections) allow the gradients to flow directly through the network, enabling deeper models to train efficiently.

- **Residual Learning**: Instead of learning the target mapping H(x), the network learns the residual F(x) = H(x) - x.
- **Shortcut Connections**: These connections skip layers, directly connecting input to output using addition.

The ResNet architecture implemented here includes:

- **3 types of convolutional layers** in the residual module:
  - 1x1 (bottleneck layers)
  - 3x3 (feature extraction layers)
  - 1x1 (projection layers)
- **Batch Normalization** to stabilize training.
- **ReLU activation** for non-linearity.

3. Code Documentation

3.1 Residual Module

The residual_module function builds a **single residual block**.

Inputs:

- data: Input tensor to the module.
- K: Number of filters for the final convolutional layer.
- stride: Stride for the 3x3 convolution.
- chanDim: Channel dimension for batch normalization.
- red: Whether to perform dimensionality reduction using a shortcut convolution.
- reg: Regularization parameter for kernel weights.
- bnEps: Small epsilon value for numerical stability in batch normalization.
- bnMom: Momentum for batch normalization updates.

Structure of the Residual Block:

1. **First Layer**:
    a. Batch normalization → ReLU activation → 1x1 convolution (bottleneck).
2. **Second Layer**:
    a. Batch normalization → ReLU activation → 3x3 convolution (main feature extraction).
    b. Stride determines spatial size reduction (used when red=True).
3. **Third Layer**:
    a. Batch normalization → ReLU activation → 1x1 convolution (restores filter size K).
4. **Shortcut Connection**:
    a. If red=True, a 1x1 convolution with stride is applied to the input to match dimensions.
    b. Otherwise, input is passed unchanged.
5. **Final Step**:

The output from the 1x1 convolution is added to the shortcut branch.

3.2 ResNet Build Method

The build method constructs the full ResNet architecture.

Inputs:

- width, height, depth: Dimensions of the input images.
- classes: Number of output classes.
- stages: List indicating the number of residual blocks per stage.
- filters: List specifying the number of filters for each stage.
- reg: Regularization strength.
- bnEps, bnMom: Batch normalization parameters.
- dataset: Dataset type ("cifar" or "tiny_imagenet") for dataset-specific preprocessing.

Steps:

1. **Input Configuration**:
    a. Accepts channels_first or channels_last formats.
    b. Performs initial **batch normalization**.
2. **Initial Convolution**:
    a. For CIFAR datasets: A **3x3 convolution** without bias and with padding.
    b. For Tiny ImageNet: A **5x5 convolution** followed by batch normalization, ReLU, and max pooling.
3. **Building Stages**:
    a. Each stage begins with a **residual module** where the input spatial dimensions are reduced using stride=(2, 2) except for the first stage (stride=(1, 1)).
    b. Additional residual modules are stacked based on the stages list.
4. **Final Layers**:
    a. Batch normalization → ReLU → Average pooling with kernel size (8,8).
    b. Flatten the feature maps and add a **fully connected dense layer** with softmax activation for classification.

4. Hyperparameters and Model Configuration

- **Epochs**: 20
- **Learning Rate**: 0.1 (decays during training)

- **Batch Size**: 128
- **Optimizer**: SGD (Stochastic Gradient Descent)
- **Loss Function**: Categorical Cross-Entropy
- **Regularization**: L2 with a strength of 0.0001

5. Training Process

Steps:

1. Data is passed through an augmentation pipeline using aug.flow.
2. The training runs for 20 epochs, updating weights using SGD.
3. After each epoch, validation accuracy is computed on the validation set.

6. Conclusion

This ResNet architecture implements **residual learning** using stacked residual blocks. The model is designed to handle alphabetic character classification tasks efficiently by leveraging:

- Shortcut connections for gradient flow.
- Batch normalization for stable training.
- Bottleneck layers (1x1 convolutions) for parameter efficiency.

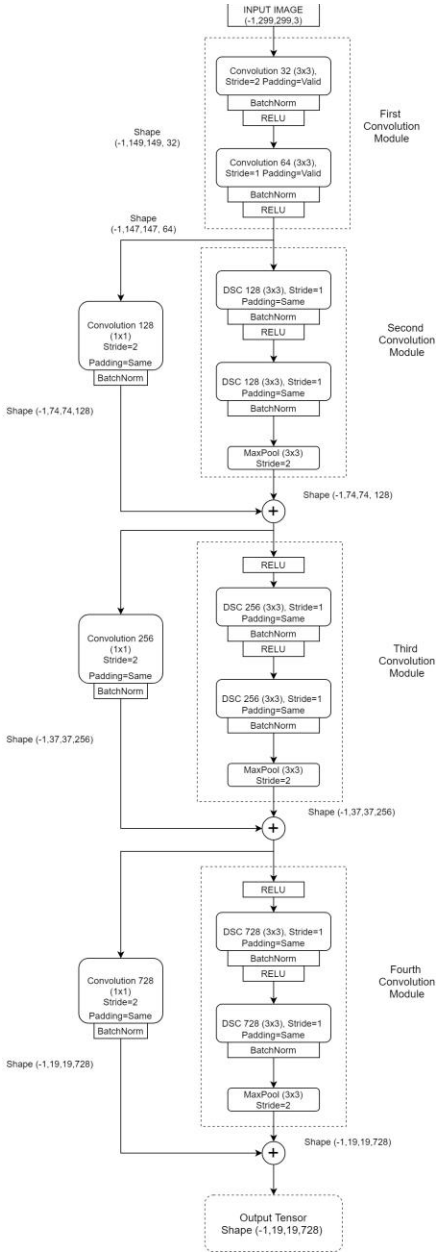
References : Resnet Architecture Explained. In their 2015 publication “Deep... | by Siddhesh Bangar | Medium
Residual Networks (ResNet) - Deep Learning - GeeksforGeeks
Papers : (PDF) Image classification based on RESNET , [1512.03385] Deep Residual Learning for Image Recognition , Deep Residual Learning for Image Recognition | IEEE Conference Publication | IEEE Xplore

Xception :

The author has split the entire Xception Architecture into 14 modules where each module is just a bunch of ***DSC*** and pooling layers. The 14 modules are grouped into three groups viz. the entry flow, the middle flow, and the exit flow. And each of the groups has four, eight, and two modules respectively. The final group, i.e the exit flow, can optionally have fully connected layers at the end.

Note: All the DSC layers in the architecture use a filter size of 3x3, stride 1, and "same" padding. And all the MaxPooling layers use a 3x3 kernel and a stride of 2.

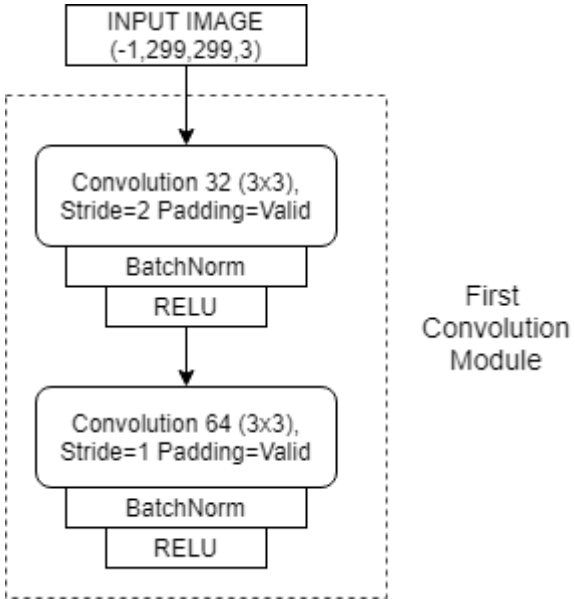
Entry Flow of Xception



The very first module contains conventional convolution layers and they don't have any ***DSC*** ones. They take input tensors of size (-1, 299, 299, 3). The **-1** in the first

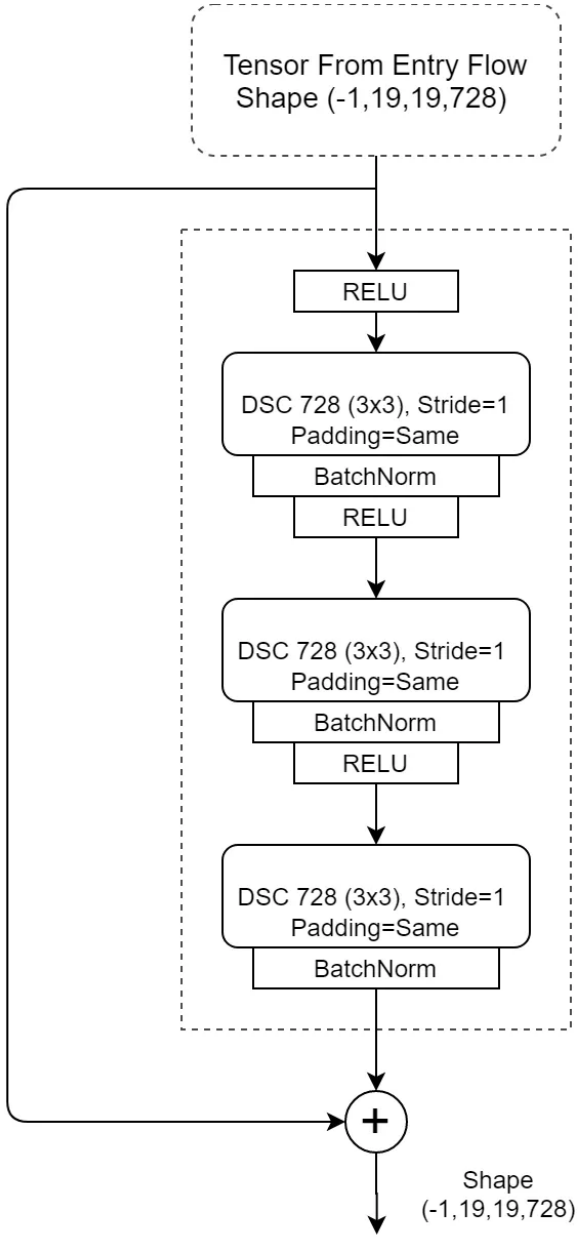
dimension represents the batch size. A negative **-1** just denotes that the batch size can be anything.

And every convolution layer, both conventional and DSC, is followed by a Batch Normalization layer. The convolutions that have a stride of 2 reduces it by almost half. And the output's shape is shown by the side which is calculated using the convolution formula that we saw before.



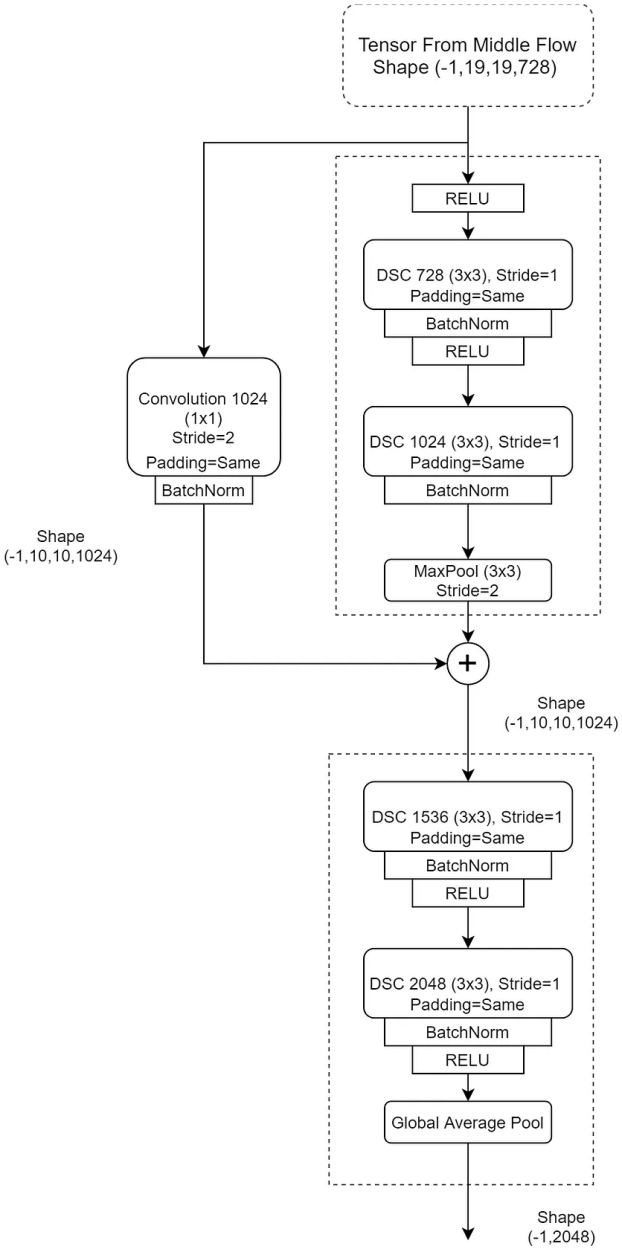
Excluding the first module, all the others in the entry flow have residual skip connections. The parallel skip connections have a pointwise convolution layer that gets added to the output from the main path.

Middle Flow of Xception



In the middle flow, there are ***eight*** such modules, one after the other. The above module is repeated eight times to form the middle flow. All the 8 modules in the middle flow use a stride of 1 and don't have any pooling layers. Therefore, the spatial size of the tensor that's passed from the entry flow remains the same. The channel depth remains the same too as all the middle flow modules have 728 filters. And that's the same as the input's depth.

Exit Flow of Xception



The exit flow has just two convolution modules and the second one doesn't have any skip connection. The second module uses Global Average Pooling, unlike the earlier modules which used Maxpooling. The output vector of the average pooling

layer can be fed to a logistic regression layer directly. But we can optionally use intermediate Fully Connected layers too.

To Sum Up

The Xception model contains almost the same number of parameters as the Inception V3 but outperforms Inception V3 by a small margin on the ImageNet dataset. But it beats Inception V3 with a better margin on the JFT image classification dataset (Google's internal dataset). Performing better with almost the same number of parameters can be attributed to its architecture engineering.

Reference : Xception: Meet The Xtreme Inception | by Anirudh S | Towards Data Science

Paper :[1610.02357] Xception: Deep Learning with Depthwise Separable Convolutions , Xception: Deep Learning with Depthwise Separable Convolutions

Xception Model for Alphabetic Classification

1. Introduction

This documentation explains the implementation of an **Xception**-based deep learning model for **alphabetic character classification**. The Xception model is a powerful CNN architecture that improves upon Inception by fully utilizing **depthwise separable convolutions**.

The custom model leverages a pre-trained **Xception model** as a feature extractor, which has been fine-tuned to classify images into **54 alphabetic classes**.

2. Xception Architecture Overview

The **Xception (Extreme Inception)** architecture is a variant of the Inception model that uses **depthwise separable convolutions** to improve computational efficiency while maintaining high accuracy.

- **Depthwise Separable Convolutions**:
- Instead of a standard convolution, the operation is broken into:
    - **Depthwise Convolution**: Applies a single convolutional filter per input channel.
    - **Pointwise Convolution**: Combines the outputs using 1x1 convolutions.
- **Pre-trained Xception Model**:
    - Trained on **ImageNet**, this model can extract rich features from input images.
    - We use it as a **base model** without its fully connected layers (`include_top=False`).
- **Custom Layers**:

Added on top of Xception for our alphabetic classification task:

- **GlobalAveragePooling2D**: Reduces feature maps to a single vector.

- **Dense Layers**: Fully connected layers to map features to the desired number of classes.

3. Code Documentation

3.1 Xception Base Model

1. **Xception as Feature Extractor**:
    a. weights='imagenet': Loads pre-trained weights from ImageNet.
    b. include_top=False: Excludes the top fully connected layers.
    c. input_shape=(71, 71, 3): Specifies input dimensions.
2. **Freezing the Base Model**:
    a. base_model.trainable = False: Freezes the weights to prevent updates during training.

This allows the model to use pre-trained features effectively.

3.2 Model Customization

The model is customized by adding new layers for classification:

- **Input Layer**: Accepts images of shape (71, 71, 3).
- **GlobalAveragePooling2D**: Replaces flattening, reducing dimensions of the feature maps to a single vector for each channel.
- **Dense Layer**:
    - **128 neurons** with ReLU activation for non-linearity.
- **Output Layer**:
    - **54 neurons** for 54 alphabetic classes.
    - activation='softmax': Ensures output values represent class probabilities.

3.3 Model Compilation

- **Optimizer**:
- adam optimizer ensures fast convergence and adaptive learning rates.

- **Loss Function**:

categorical_crossentropy for multi-class classification.

- **Metrics**:

accuracy to evaluate model performance.

3.4 Early Stopping Callback

The EarlyStopping callback monitors validation loss and stops training if performance no longer improves.

- **monitor='val_loss'**: Tracks validation loss.
- **patience=3**: Stops training if the validation loss does not improve for 3 consecutive epochs.
- **restore_best_weights=True**: Ensures the model reverts to the best weights during training.

4. Training and Evaluation

4.1 Model Training

- **Data Augmentation**: aug2.flow applies real-time data augmentation to training images.
- **Validation Data**: Used to evaluate the model after each epoch.
- **Epochs**: The model trains for up to **10 epochs** but stops early if validation loss stagnates.

4.2 Model Evaluation

The model is evaluated on both the test and validation datasets:

- **evaluate**: Calculates test loss and accuracy.
- The same evaluation is repeated for validation data.

4.3 Saving the Model

The trained model is saved for future inference:

- **save**: Saves the model in .keras format.

5. Conclusion

This implementation of Xception for alphabetic classification leverages:

1. **Pre-trained Xception** as a feature extractor to reduce training time and computational cost.
2. **Global Average Pooling** to reduce dimensionality and overfitting.
3. **Custom Dense Layers** for classification into 54 classes.
4. **Early Stopping** to ensure efficient training without overfitting.

References : Xception: Meet The Xtreme Inception | by Anirudh S | Towards Data Science

Papers : Xception: Deep Learning with Depthwise Separable Convolutions

Densenet :

What is DenseNet?

DenseNet, short for Dense Convolutional Network, is a deep learning architecture for convolutional neural networks (CNNs) introduced by Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger in their paper titled "Densely Connected Convolutional Networks" published in 2017. DenseNet revolutionized the field of computer vision by proposing a novel connectivity pattern within CNNs, addressing challenges such as feature reuse, vanishing gradients, and parameter efficiency. Unlike traditional CNN architectures where each layer is connected only to subsequent layers, DenseNet establishes direct connections between all layers within a block. This dense connectivity enables each layer to receive feature maps

from all preceding layers as inputs, fostering extensive information flow throughout the network.

Key Characteristics of DenseNet

Alleviated Vanishing Gradient Problem: Dense connections ensure that gradients can flow directly to earlier layers, mitigating the vanishing gradient issue common in deep networks.

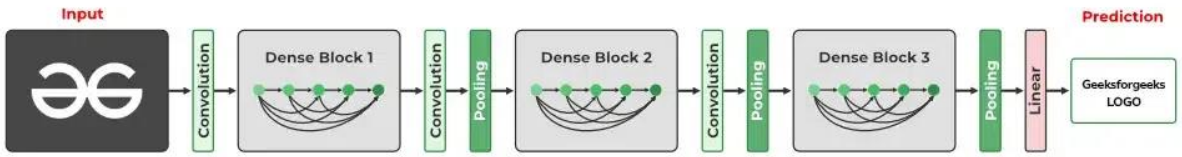
Improved Feature Propagation: Each layer has direct access to the gradients from the loss function and the original input signal, promoting better feature propagation.

Feature Reuse: By concatenating features from all preceding layers, DenseNet encourages feature reuse, reducing redundancy and improving efficiency.

Reduced Parameters: Despite its dense connections, DenseNet is parameter-efficient. It eliminates the need to relearn redundant features, resulting in fewer parameters compared to traditional networks.

Architecture of DenseNet

DenseNet introduces a paradigm shift by connecting each layer to every other layer in a feed-forward manner. Unlike traditional CNNs, which have a single connection between consecutive layers, DenseNet ensures that each layer receives inputs from all preceding layers and passes its output to all subsequent layers. This results in a network with L(L+1)/2 direct connections for L layers, significantly enhancing information flow.



Dense Block :

Dense blocks are the building blocks of DenseNet architectures. Each dense block consists of multiple convolutional layers, typically followed by batch normalization and a non-linear activation function (e.g., ReLU). Importantly, each layer within a dense block receives feature maps from all preceding layers as inputs, facilitating feature reuse and propagation.

Within a dense block, each layer receives the concatenated output of all preceding layers as its input. If a dense block has m layers, and each layer produces k feature maps (where k is known as the growth rate), the l-th layer will have k×(l+l0)k×(l+l0) input feature maps (where l0l0 is the number of input channels to the dense block).

Transition Layer

Transition layers are used to connect dense blocks. They serve two main purposes: reducing the number of feature maps and downsampling the spatial dimensions of the feature maps. This helps in maintaining the computational efficiency and compactness of the network. A typical transition layer consists of:

Batch Normalization: Normalizes the feature maps.

1x1 Convolution: Reduces the number of feature maps.

Average Pooling: Downsamples the spatial dimensions.

Growth Rate (k)

The growth rate (k) is a critical hyperparameter in DenseNet. It defines the number of feature maps each layer in a dense block produces. A larger growth rate means more information is added at each layer, but it also increases the computational cost. The choice of k affects the network's capacity and performance.

Advantages of DenseNet

Reduced Vanishing Gradient Problem: Dense connections improve gradient flow and facilitate the training of very deep networks.

Feature Reuse: Each layer has access to all preceding layers' feature maps, promoting the reuse of learned features and enhancing learning efficiency.

Fewer Parameters: DenseNets often have fewer parameters compared to traditional CNNs with similar depth due to efficient feature reuse.

Improved Accuracy: DenseNets have shown high accuracy on various benchmarks, such as ImageNet and CIFAR.

Limitations of DenseNet

High Memory Consumption: Dense connections increase memory usage due to the storage requirements for feature maps, making DenseNet less practical for devices with limited memory.

Computational Complexity: The extensive connectivity leads to increased computational demands, resulting in longer training times and higher computational costs, which may not be ideal for real-time applications.
Implementation Complexity: Managing and concatenating a large number of feature maps adds complexity to the implementation, requiring careful tuning of hyperparameters and regularization techniques to maintain performance and stability.
Risk of Overfitting: Although DenseNet reduces overfitting through better feature reuse, there is still a risk, particularly if the network is not properly regularized or if the training data is insufficient.

DenseNet Model for Alphabetic Classification

1. Introduction

This documentation explains the implementation of a **DenseNet**-based deep learning model for **alphabetic character classification**. DenseNet is a convolutional neural network (CNN) architecture that emphasizes feature reuse, making it computationally efficient and highly accurate for visual recognition tasks.

2. DenseNet Architecture Overview

DenseNet stands for **Densely Connected Convolutional Networks**. Unlike traditional CNNs, where each layer passes its output to only the next layer, DenseNet connects **each layer** to every other layer. This ensures feature reuse, reduces redundancy, and improves gradient flow during backpropagation.

Key Characteristics:

1. **Dense Blocks**:

2. Each layer passes its feature maps to all subsequent layers within the same block. The outputs are concatenated to form the input for the next layer.

3. **Growth Rate**:

Determines how many new feature maps each layer contributes to the output.

4. **Pre-trained DenseNet**:

Using **DenseNet121** with pre-trained weights on ImageNet allows faster convergence and better generalization.

3. Code Documentation

3.1 DenseNet Base Model

The **DenseNet121** model is loaded as a feature extractor:

- **weights='imagenet'**: Loads pre-trained weights on the ImageNet dataset.
- **include_top=False**: Removes the fully connected layers at the top of the network, allowing us to add custom classification layers.
- **input_shape=(71, 71, 3)**: Specifies the shape of input images (height, width, channels).

3.2 Freezing Pre-trained Layers

The pre-trained DenseNet layers are frozen to prevent them from being updated during training:

- This ensures that the model leverages the pre-trained features without modifying them.

3.3 Custom Model Construction

The model is customized by adding new layers to classify 54 alphabetic classes:

python
Copy code

- **GlobalAveragePooling2D**: Reduces the feature maps into a single vector per channel. This minimizes overfitting and reduces the number of parameters.
- **Dense Layer**:
    - 54 neurons correspond to the 54 alphabetic classes.
    - activation='softmax': Outputs probabilities for each class.

3.4 Model Compilation

The model is compiled using the following configuration:

- **Loss Function**: categorical_crossentropy is used for multi-class classification.
- **Optimizer**: adam is an adaptive learning rate optimizer that combines momentum and RMSprop.
- **Metrics**: Accuracy is used to measure the performance of the model.

4. Training and Evaluation

4.1 Model Training

The model is trained using augmented data:

- **Data Augmentation**: The aug2.flow generates augmented versions of the training data in real time. This helps improve generalization.
- **Validation Data**: Used to evaluate model performance at the end of each epoch.
- **Epochs**: The model trains for a maximum of 10 epochs.

4.2 Model Evaluation

The model is evaluated on both the test and validation datasets:

- **evaluate**: Computes the loss and accuracy on unseen test and validation data.

5. Conclusion

The DenseNet-based model for alphabetic classification leverages pre-trained **DenseNet121** as a feature extractor. Key highlights of this approach include:

1. **DenseNet Advantages**:
    a. Efficient use of features through dense connections.
    b. Improved gradient flow, resulting in faster convergence.
2. **Custom Layers**:
    a. A **Global Average Pooling** layer reduces feature maps efficiently.
    b. A **Dense Softmax Layer** performs classification into 54 alphabetic classes.
3. **Efficient Training**:
    a. Freezing the DenseNet base reduces computation time.
    b. Data augmentation improves generalization.

This model achieves high accuracy on test and validation datasets while remaining computationally efficient.

References : DenseNet Explained - GeeksforGeeks

Paper : Densely Connected Convolutional Networks | IEEE Conference Publication | IEEE Xplore , [1608.06993] Densely Connected Convolutional Networks

Comparison

Which Architecture is Best?

For a **merged dataset of EMNIST and AHCD**, I recommend **DenseNet** as the top choice due to the following reasons:

1. **Feature Reuse**: DenseNet connects each layer to all subsequent layers, maximizing feature propagation. For alphabetic classification, where small, subtle features are critical, DenseNet excels.

2. **Gradient Flow**: The dense connections ensure better gradient flow, especially when training on small or medium datasets, reducing the risk of vanishing gradients.
3. **Small Dataset Suitability** : DenseNet performs exceptionally well on smaller datasets, thanks to its compact parameter usage and efficient architecture.
4. **Handling Both English and Arabic Characters**:
    a. EMNIST has subtle strokes for English alphabets.
    b. AHCD characters have more complex curves and structure.

DenseNet effectively learns features across varying levels of complexity.

Runner-Up: Xception

Xception is also a strong contender due to its lightweight and efficient use of **depthwise separable convolutions**. If computational efficiency is a primary concern, Xception can be a great alternative. However, DenseNet may still outperform it in accuracy on this dataset.

Why Not ResNet?

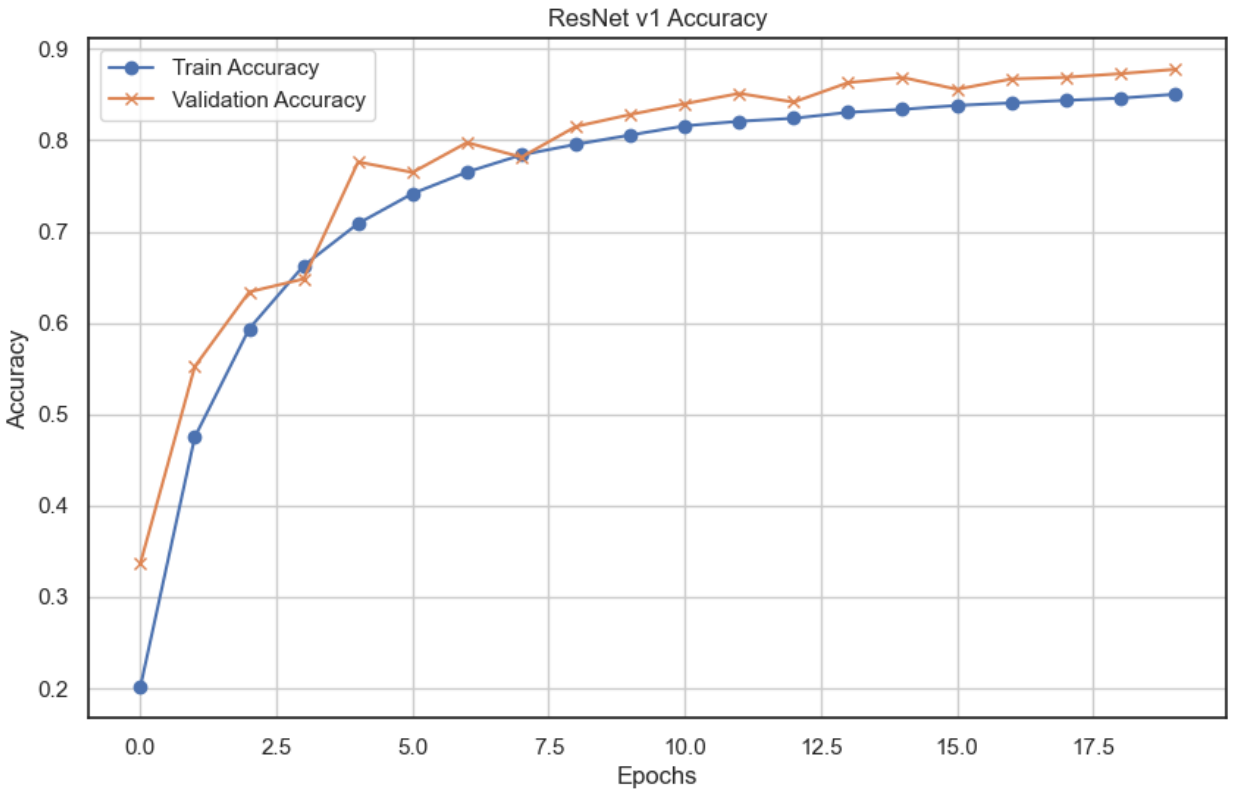
While ResNet is powerful, it may not fully exploit the smaller datasets' feature richness compared to DenseNet. ResNet's deeper layers may introduce unnecessary complexity without the benefit of DenseNet's dense connections for feature reuse.

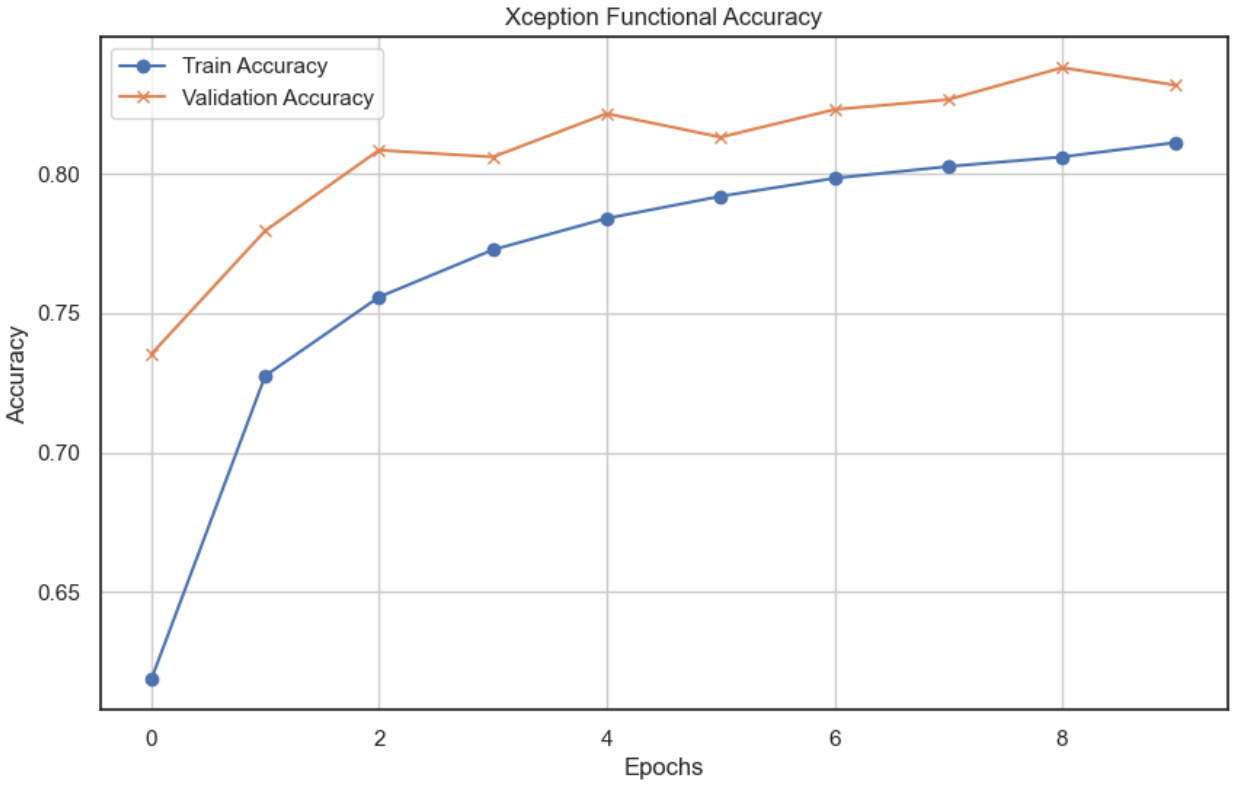
Key architecture comparison

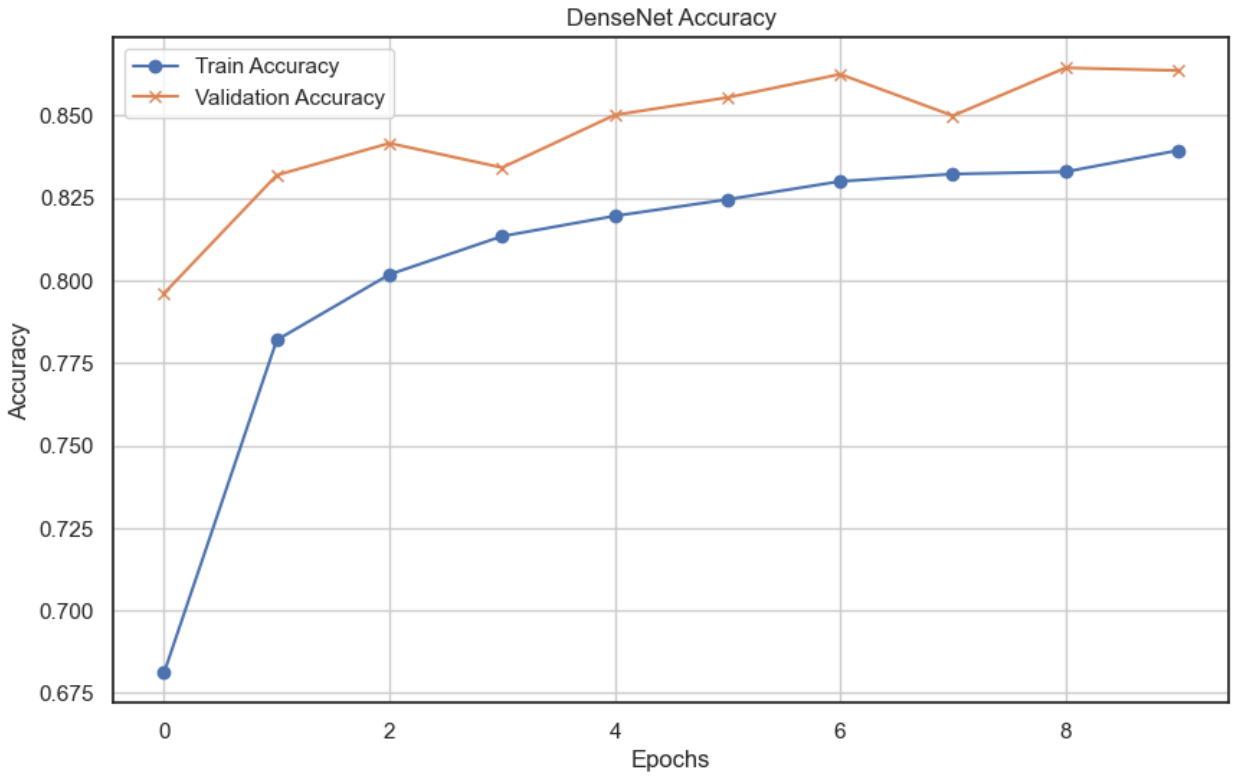
Feature	ResNet	Xception	DenseNet
Main concept	Residual learning with skip connections	Depthwise separable convolutions	Dense connections (feature reuse)
Depth and parameters	Moderate (depends on stages)	Lightweight and computationally efficient	Deeper but very efficient with parameters
Gradient flow	Excellent due to residual connections	Good due to reduced parameters per layer	Excellent due to dense connections
Computational cost	Moderate to high	Lower computational cost	Moderate , but optimized for deep networks
Overfitting risk	Low due to skip connections	Low due to efficient feature extraction	Very low due to feature reuse and fewer parameters
Performance on small data	Strong , especially if fine-tuned	Very strong for small to medium-sized data	Exceptional , handles small datasets well

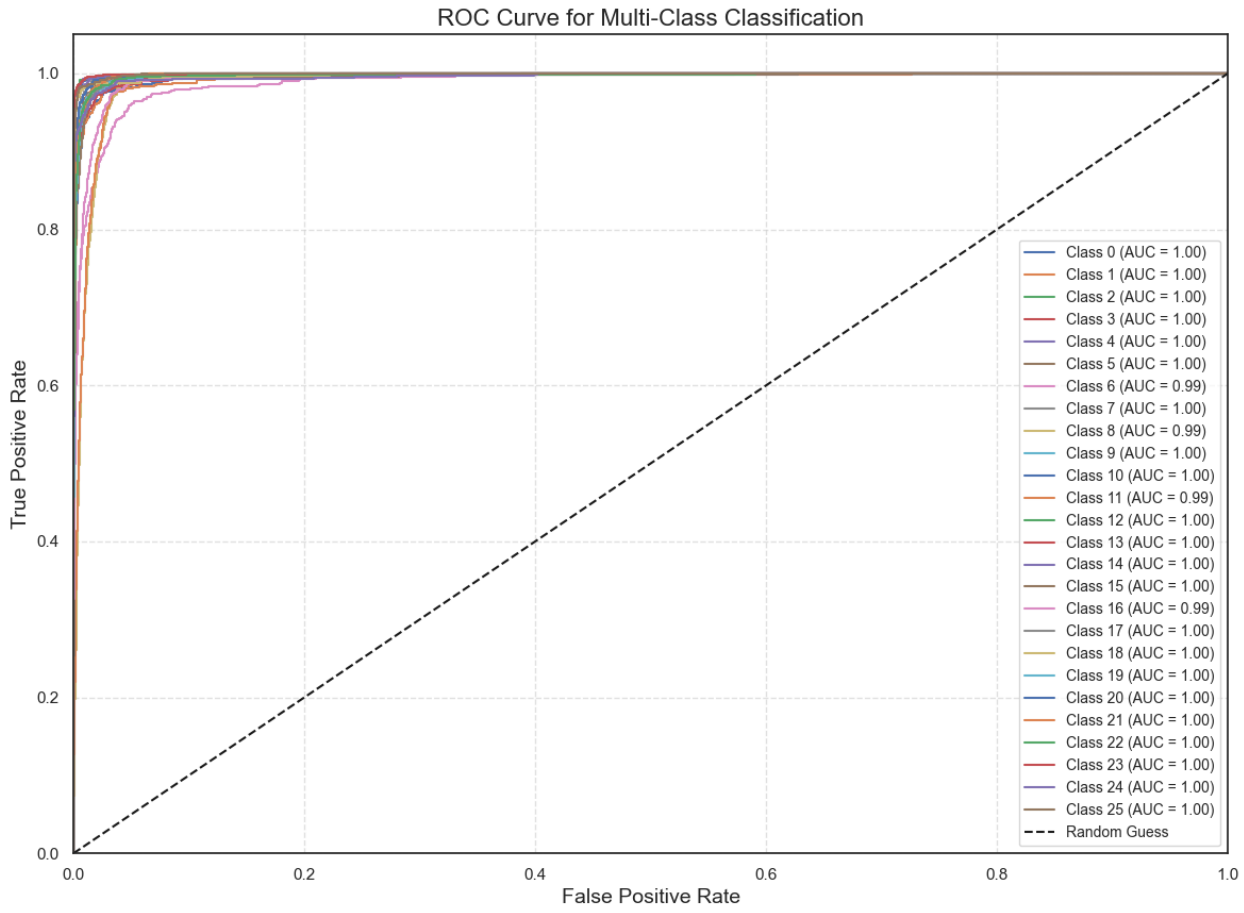
Model	Pros	Cons
ResNet	-Handles deep networks well -Strong gradient flow -Modular and scalable	-Parameter heavy -Computationally expensive -Less efficient on small data
Xception	-Efficient computation -Lightweight -Great for small to medium datasets	-Not ideal for very small datasets -Lack of feature reuse -Limited depth
DenseNet	-Feature reuse improves efficiency	-Higher memory usage -Slower training time

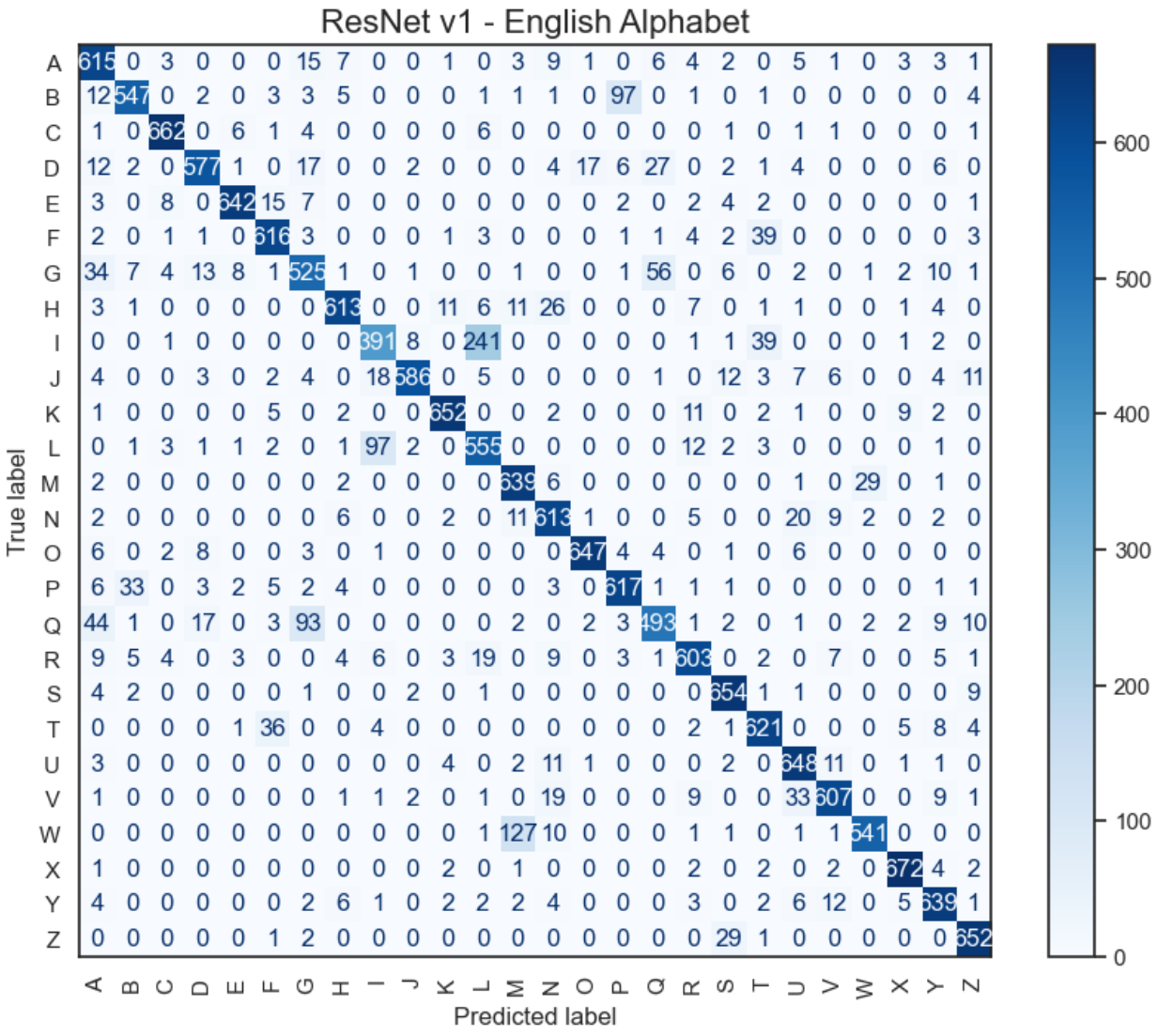
	-Compact model -Excels on small datasets	-More complex structure

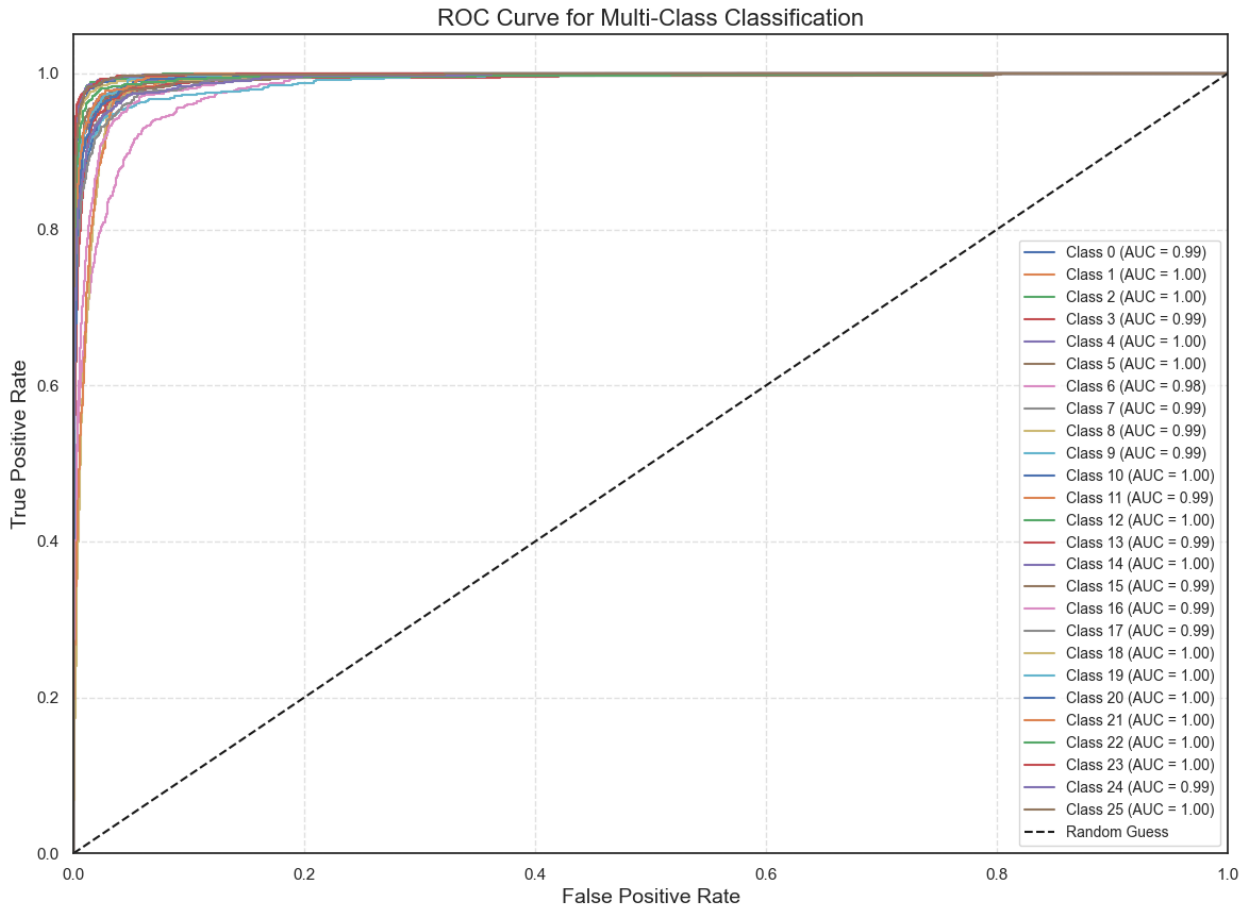


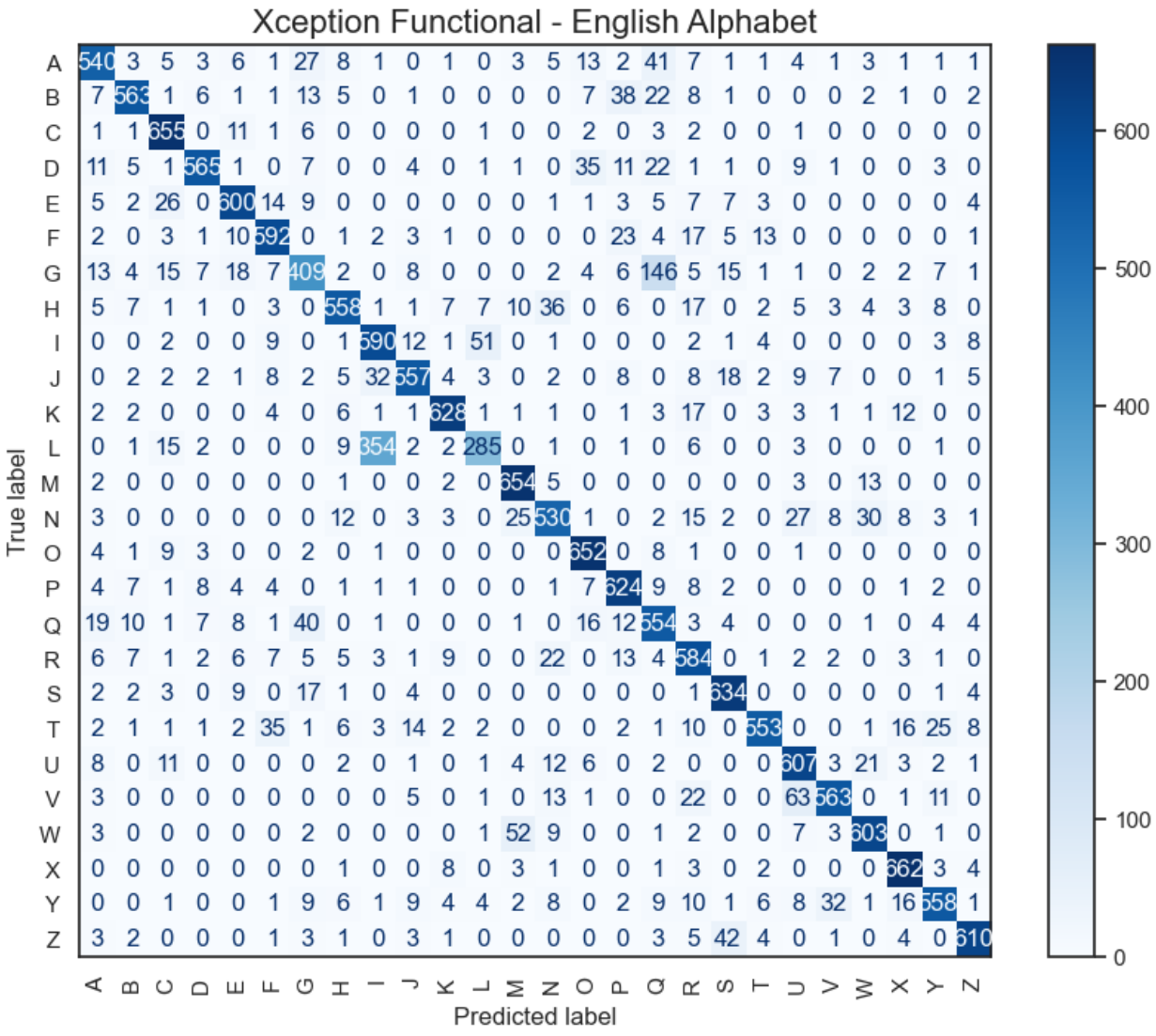


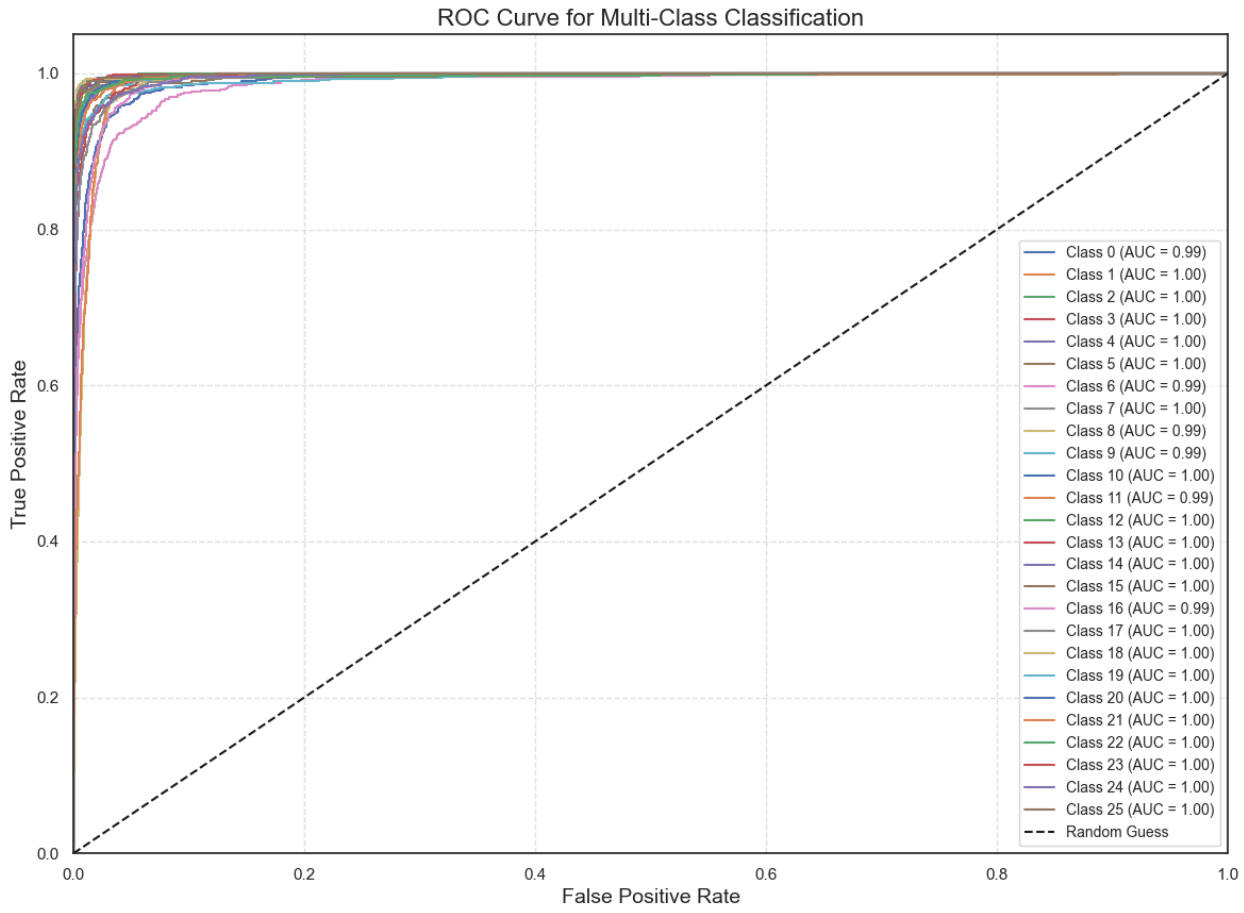

DenseNet Accuracy
Train Accuracy
Validation Accuracy
Accuracy
0.850
0.825
0.800
0.775
0.750
0.725
0.700
0.675
0
2
4
6
8
Epochs

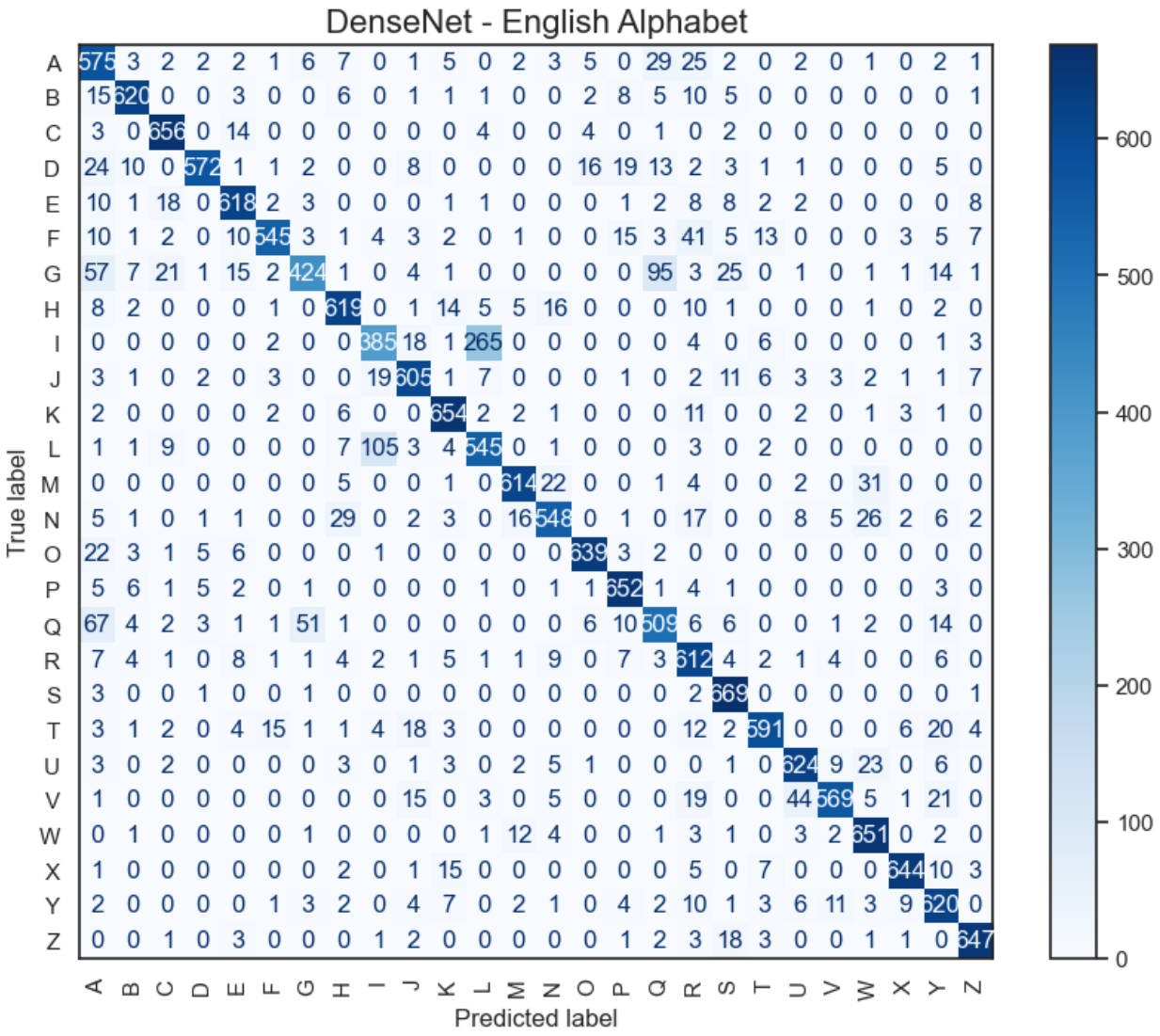

ROC Curve for Multi-Class Classification
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Class 0 (AUC = 1.00)
Class 1 (AUC = 1.00)
Class 2 (AUC = 1.00)
Class 3 (AUC = 1.00)
Class 4 (AUC = 1.00)
Class 5 (AUC = 1.00)
Class 6 (AUC = 0.99)
Class 7 (AUC = 1.00)
Class 8 (AUC = 0.99)
Class 9 (AUC = 1.00)
Class 10 (AUC = 1.00)
Class 11 (AUC = 0.99)
Class 12 (AUC = 1.00)
Class 13 (AUC = 1.00)
Class 14 (AUC = 1.00)
Class 15 (AUC = 1.00)
Class 16 (AUC = 0.99)
Class 17 (AUC = 1.00)
Class 18 (AUC = 1.00)
Class 19 (AUC = 1.00)
Class 20 (AUC = 1.00)
Class 21 (AUC = 1.00)
Class 22 (AUC = 1.00)
Class 23 (AUC = 1.00)
Class 24 (AUC = 1.00)
Class 25 (AUC = 1.00)
Random Guess


ResNet v1 - English Alphabet
True label
Predicted label
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
600
500
400
300
200
100
0


ROC Curve for Multi-Class Classification
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Class 0 (AUC = 0.99)
Class 1 (AUC = 1.00)
Class 2 (AUC = 1.00)
Class 3 (AUC = 0.99)
Class 4 (AUC = 1.00)
Class 5 (AUC = 1.00)
Class 6 (AUC = 0.98)
Class 7 (AUC = 0.99)
Class 8 (AUC = 0.99)
Class 9 (AUC = 0.99)
Class 10 (AUC = 1.00)
Class 11 (AUC = 0.99)
Class 12 (AUC = 1.00)
Class 13 (AUC = 0.99)
Class 14 (AUC = 1.00)
Class 15 (AUC = 0.99)
Class 16 (AUC = 0.99)
Class 17 (AUC = 0.99)
Class 18 (AUC = 1.00)
Class 19 (AUC = 1.00)
Class 20 (AUC = 1.00)
Class 21 (AUC = 1.00)
Class 22 (AUC = 1.00)
Class 23 (AUC = 1.00)
Class 24 (AUC = 0.99)
Class 25 (AUC = 1.00)
Random Guess


Xception Functional - English Alphabet
True label
Predicted label
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
600
500
400
300
200
100
0


ROC Curve for Multi-Class Classification
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Class 0 (AUC = 0.99)
Class 1 (AUC = 1.00)
Class 2 (AUC = 1.00)
Class 3 (AUC = 1.00)
Class 4 (AUC = 1.00)
Class 5 (AUC = 1.00)
Class 6 (AUC = 0.99)
Class 7 (AUC = 1.00)
Class 8 (AUC = 0.99)
Class 9 (AUC = 0.99)
Class 10 (AUC = 1.00)
Class 11 (AUC = 0.99)
Class 12 (AUC = 1.00)
Class 13 (AUC = 1.00)
Class 14 (AUC = 1.00)
Class 15 (AUC = 1.00)
Class 16 (AUC = 0.99)
Class 17 (AUC = 1.00)
Class 18 (AUC = 1.00)
Class 19 (AUC = 1.00)
Class 20 (AUC = 1.00)
Class 21 (AUC = 1.00)
Class 22 (AUC = 1.00)
Class 23 (AUC = 1.00)
Class 24 (AUC = 1.00)
Class 25 (AUC = 1.00)
Random Guess



- **DenseNet**: Best choice for **merged EMNIST and AHCD datasets** due to its feature reuse and excellent performance on small datasets.
- **Xception**: Good alternative if **computational efficiency** and speed are priorities.
- **ResNet**: While powerful, it may be overkill for a small-to-medium dataset like ours and may not perform as efficiently as DenseNet.