|  |
| --- |
| **Chapter 4**  **Methodology** |
|  |

This chapter illustrates the overall technique of our proposed system. Here we describe our working procedure

**4.1 Structure of Model**

**Test Set**

**Load Dataset**

**Split Dataset into Training Set and Test Set**

**Load Dataset**

**Training Set**

**Validation Set**=**Test Set**

**Deep Learning Algorithms**

CNN

VGG

Inception

LSTM

Bi-LSTM

### [ResNet](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet)

### [ResNet-50](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)

### [ResNet-100](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)

### [ResNet-150](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)

### [ResNet-150](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)V2

### [ResNet-100](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)V2

### [ResNet-50](https://github.com/FahimHabib20/Bangla-Handwritten-Character-Recognition-BHCR-using-Deep-Learning-Models/blob/main/ResNet_version/ResNet50.ipynb)V2

**Analyze**

**Performance**

**Performance**

**Evaluation**

**Figure : Diagram of the proposed method**

The proposed approach includes the three important stages namely: Data Preprocessing step, Training step and Prediction step. Flow diagram is shown in Figure 4.1 and current section includes the brief discussions of the same

**4.1.1 Data Preprocessing Step**In our BHCR research, data preprocessing plays a crucial role in preparing the input data for training and evaluation of deep learning models. Bangla Handwritten Character DB CMATERdb 3.1. The dataset is partitioned into 12,000 images for training and 3,000 images for testing, with each image annotated and classified into one of 50 distinct character classes. The following steps outline the data preprocessing pipeline utilized in our study:

1. **ImageDataGenerator Configuration:**

We utilize the ImageDataGenerator class from the Keras preprocessing module to perform real-time data augmentation and normalization. This class allows for the configuration of various preprocessing techniques to be applied to the input images during training and testing phases.

1. **Normalization:**

We rescale the pixel values of the input images to a range between 0 and 1 by dividing each pixel value by 255. This normalization step ensures that the input data falls within a consistent range, facilitating stable and efficient training of the deep learning models.

1. **Data Augmentation (Optional):**

Additional preprocessing techniques such as rotation, zoom, and horizontal/vertical flips can be incorporated into the ImageDataGenerator configuration to augment the training dataset. Data augmentation helps to increase the diversity and robustness of the training data, thereby improving the generalization performance of the trained models.

1. **Image Loading and Resizing:**

We load the input images from the specified directory (train\_dir and test\_dir) using the flow\_from\_directory method of the ImageDataGenerator class. During loading, the images are resized to a uniform size of 50x50 pixels to ensure consistency in input dimensions across all samples.

1. **Grayscale Conversion:**

To simplify the input data and reduce computational complexity, we convert the color images to grayscale using the color\_mode='grayscale' parameter. This conversion retains essential information for character recognition while reducing the input dimensionality and computational overhead associated with processing color images.

1. **Batch Generation:**

The flow\_from\_directory method generates batches of data on-the-fly from the specified directory, allowing for efficient processing of large datasets that do not fit into memory. Each batch consists of a specified number of images (batch\_size), along with their corresponding labels encoded in categorical format (class\_mode='categorical').

1. **Train-Validation Split:**

We split the dataset into training and validation sets using separate directory paths (train\_dir and test\_dir) for training and testing images, respectively. The train\_generator and validation\_generator objects are then created to generate batches of training and validation data during model training.

Training Step:

1. Data Preprocessing:
   * Images from the training directory are preprocessed using ImageDataGenerator.
   * Rescaling is performed to normalize pixel values to the range [0, 1].
   * Images are resized to 50x50 dimensions and converted to grayscale.
2. Model Architecture:
   * The model architecture is based on ResNet-152V2.
   * Input images are fed into a series of convolutional layers with batch normalization and activation.
   * Residual blocks are used to enable training of very deep networks.
   * Global average pooling is applied to reduce spatial dimensions.
   * Two dense layers with ReLU activation are added for classification, followed by a softmax layer for output.
3. Model Compilation:
   * The model is compiled using the Adam optimizer and categorical cross-entropy loss function.
   * Accuracy is chosen as the metric for evaluation during training.
4. Model Training:
   * The model is trained using the fit method.
   * Training data is provided using the train\_generator.
   * Validation data is provided using the test\_generator.
   * Training is performed for 3 epochs.

Prediction Step:

1. Generate Predictions:
   * The trained model is used to generate predictions on the test data.
   * Predictions are obtained using the predict method applied to the test\_generator.
2. Evaluate Predictions:
   * Predicted labels are converted from probabilities to class labels using argmax.
   * True labels are obtained from the test generator.
   * Classification report is generated using classification\_report from sklearn.metrics.
   * Test accuracy is calculated using the evaluate method of the model.

Overview of the Model:

* Input: Grayscale images resized to 50x50 dimensions.
* Architecture: ResNet-152V2 variant with residual blocks.
* Output: Probability distribution over the classes (number of classes determined by the number of subfolders in the training directory).
* Training: Adam optimizer, categorical cross-entropy loss function, trained for 3 epochs.
* Evaluation Metrics: Accuracy, precision, recall, and F1-score are calculated for each class using the test data.
  + **The dataset is collected from Kaggle, [11].**

Creation of three classes where each class contains :

80% of the image for Training

20% of the image for Test

Validation = Test

* + - * 10% of the image for Validation

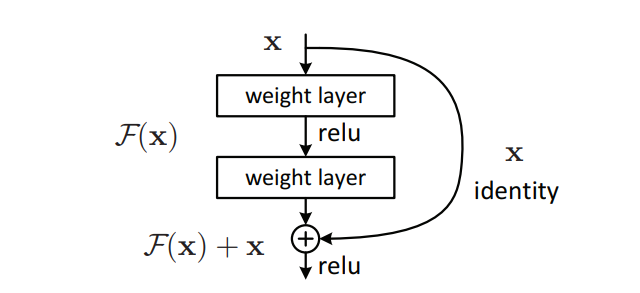
|  |
| --- |
| **Deep Learning Application** |
| Inputs: Dataset containing images of BCHR  Fine-tuning with ResNet  Training\_set (80%) Test\_set(20%) Validation\_set =Test\_set  **<<**  Images are in .bmp >>  Performance: Success rate / Error rate  Decision : % Acceptable accuracy |

**ResNet Model Architecture**

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),

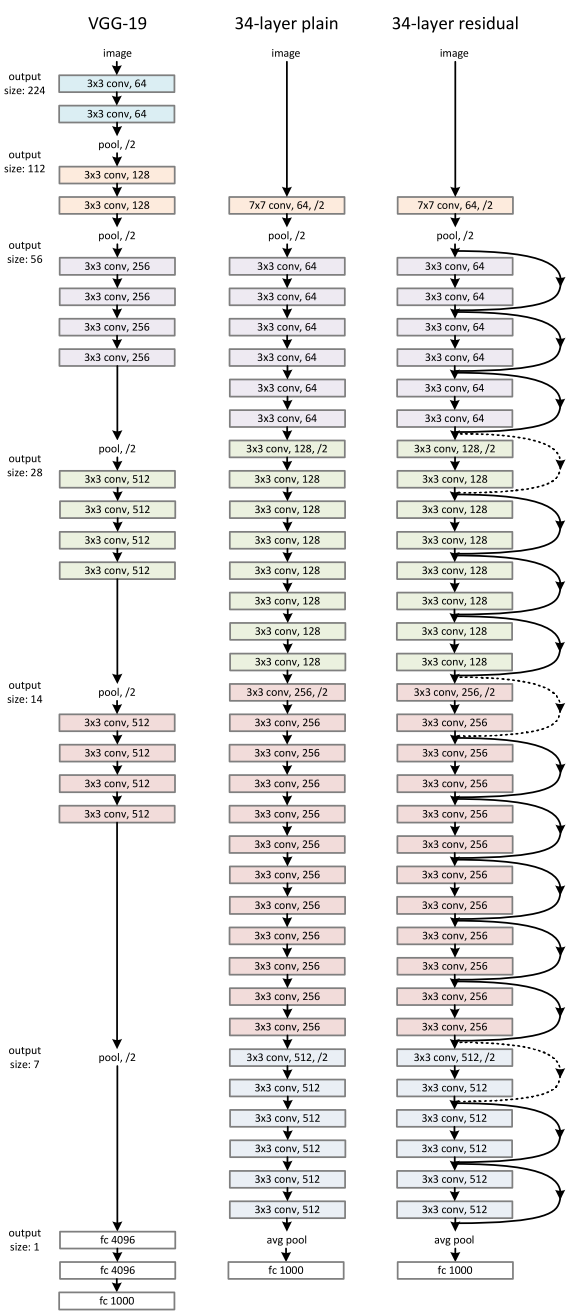
***F(x) := H(x) - x* which gives *H(x) := F(x) + x*.**



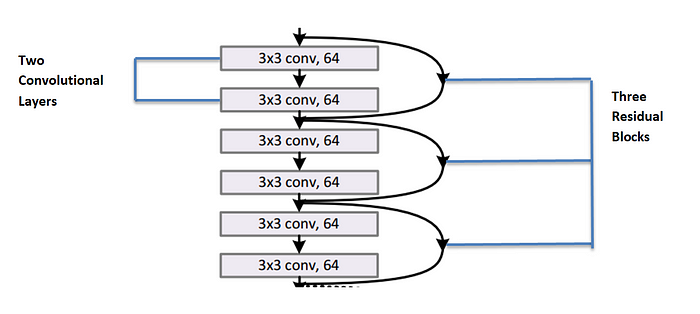
Fig

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.

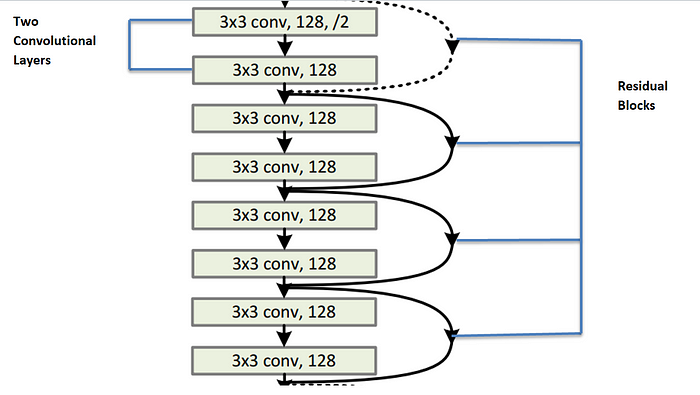
The VGG-19-inspired 34-layer plain network architecture used by ResNet is followed by the addition of the shortcut connection. The architecture is subsequently transformed into the residual network by these short-cut connections, as depicted in the following figure:



In this diagram we can see the VGC-19 ,34 layer plain network and 34 layered residual network.  
In this residual network there are total 16 residual blocks.

****

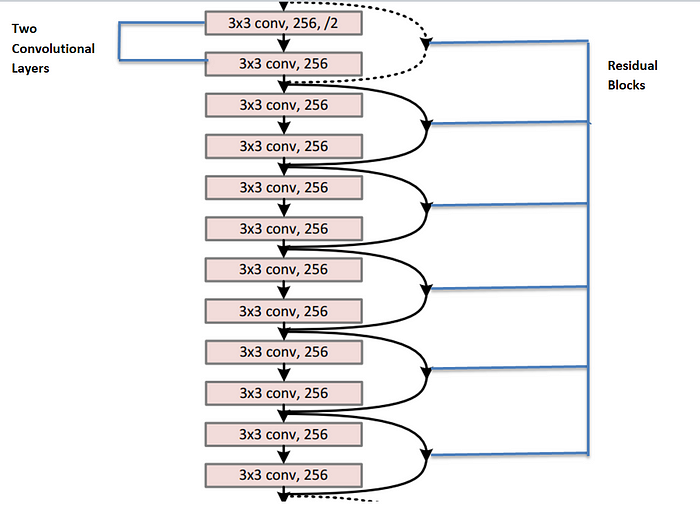
The first set consists of 3 residual blocks. Each residual block consists of 2 convolution layers where each convolution layer consists of 64 filters of size 3x3 and a skip connection which performs identity mapping.

****

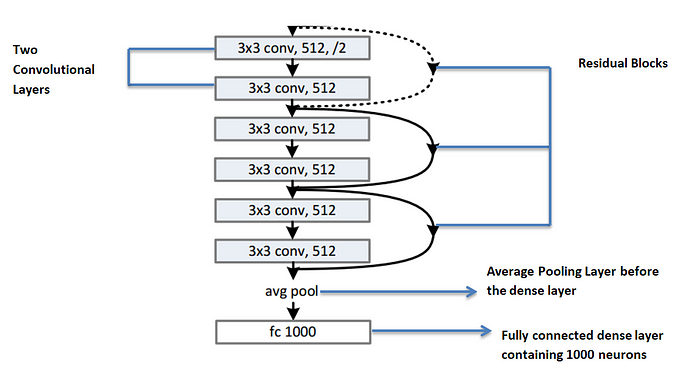
Second set consists of 4 residual blocks. Each residual block contains 2 convolutional layers where each layer consists of 128 kernels of of size 3x3 and a skip connection .

**·** Dotted line skip connections represents the connections when the dimension are increased then it has to match the dimension of the output of convolutional layers.

When the dimensions increase (dotted line shortcuts ), we consider two options:   
The first is that the shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. The second is that the projection shortcut is used to match dimensions (done by 1×1 convolutions). For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

****

Third set consists of 6 residual blocks where each residual blocks contains two convolutional layers . Each convolution layer has 256 filters of size 3x3.

****

The fourth set consists of 3 residual blocks where each residual blocks consists of 2 convolutional layers .Each convolutional layer contains 512 filters of 3x3 each.

· After that the feature map is passed through average pooling layer and then it is passes through dense layer containing 1000 neurons to classify 1000 classes.

**ResNet using Keras**

An open-source, Python-based neural network framework called Keras may be used with TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. It is made to make deep neural network experimentation quick. The following ResNet implementations are part of Keras Applications and offer ResNet V1 and ResNet V2 with 50, 101, or 152 layers,ResNetV2 and the original ResNet (V1) vary primarily in that V2 applies batch normalisation before each weight layer.