BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

A Report

On

Natural Disaster Drone Images Inpainting and Classification

Submitted To:

Prof. Pratik Narang
Associate Professor,
Department of Computer Science and Information Systems, BITS Pilani

Submitted By:

Group F4
Arjoo Kumari (2020B3A70770P)
Adarsh Goel (2020B3A70821P)
Devashish Siwatch (2020A7PS0113P)

Task 1: Image Inpainting using GANs

1. Architecture

a. Generator: The architecture utilized by us for the generator is taken from the research paper, *"Coarse-to-Fine Image Inpainting via Region-wise Convolutions and Non-Local Correlation"* by Ma Y. & et. al. The generator utilizes the basic encoder and decoder architecture. There are two stages involved in the image generation namely: Coarse stage and Fine stage. At the coarse stage, the framework first infers the semantic contents from the existing regions using region-wise convolution filters, rather than the identical ones. Then, it further enhances the quality of the composited image using the non-local operation, which takes the correlation between different regions into consideration. At the fine stage, the two different regions are considered together using a style loss over the whole image, which perceptually enhances the image quality. With the two-stage progressive generation, the framework will make the restored images more realistic and perceptually consistent.

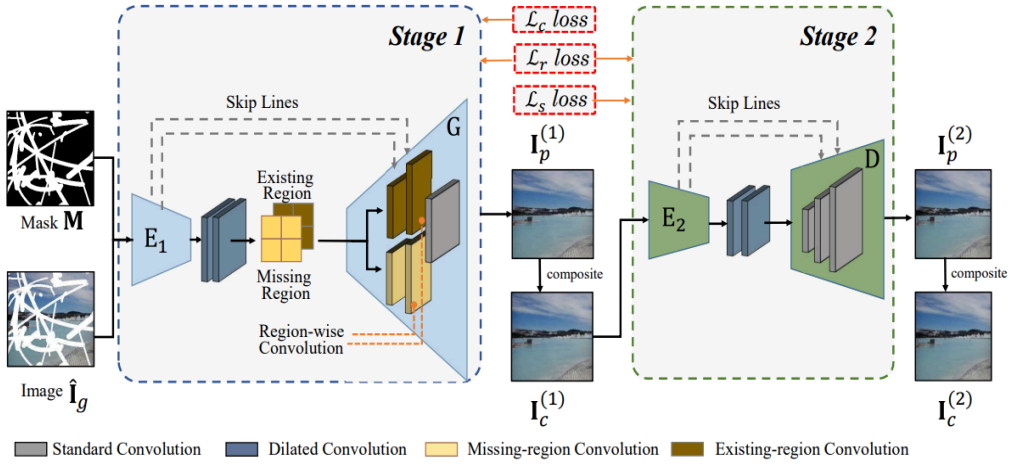


Fig 1: *Architecture of the generator.*
(Source: "Coarse-to-Fine Image Inpainting via Region-wise Convolutions and Non-Local Correlation" by Ma. Y & et.al.)

As shown in Figure 1, the framework takes the incomplete image $\hat{I}_g$ and a binary mask M as input, and attempts to restore the complete image close to ground truth image I_g, where M indicates the missing regions (the mask value is 0 for missing pixels and 1 for elsewhere), $\hat{I}_g = I_g \odot M$ and $\odot$ denotes dot product. To accomplish this goal, network E_1, E_2 serve as encoders in two stages respectively to extract semantic features from corresponding input images. A decoder G composing of the proposed region-wise convolutional layers is employed after encoder E1 to restore the semantic contents for different regions, and generates the predicted image $I_p^{(1)} = G(E_1(\hat{I}_g, M))$ at the coarse stage. After feeding the composited image $I_c^{(1)} = \hat{I}_g + I_p^{(1)} \odot (1-M)$ from the coarse stage to encoder E_2, another decoder D at the second stage further synthesizes the refined image $I_p^{(2)} = D(E_2(I_c^{(1)}, M))$. Based on the

encoder-decoder architectures, we finally have the visually and semantically realistic inpainting result $I_c^{(1)} = \hat{I}_g + I_p^{(2)} \odot (1 - M)$ close to the ground truth image I_g.

In the coarse stage, input image and mask are taken as input. Then, standard convolution layers are used on the image images to get output X_c to extract features from the input image. After this, diluted convolution layers are used on X_c to expand the area covered without pooling to get X_d. The objective is to cover more information from the output obtained with every convolution operation. This method offers a wider field of view at the same computational cost. After this, using the mask we extract the features of existing and missing regions using X_c and X_d. Then, region-wise convolution and upscaling is done on the resulting feature map. For image inpainting tasks, the input images are composed of both existing regions with valid pixels and missing regions (masked regions) with invalid pixels in the mask to be synthesized. Only relying on the same convolution filters, we can hardly restore the semantic features over different regions, which in practice usually leads to the visual artifacts such as color discrepancy, blur and obvious edge responses surrounding the missing regions. This is the reason for using region-wise convolutions in the decoder network G at the coarse stage. Now, the decoder can separately generate the corresponding contents for different regions using different convolution filters.

For the fine stage, all the architecture is the same except there are no regional convolutional layers.

Discriminator: The discriminator takes in the portion of the generated image ($I_c^{(1)}$ and $I_c^{(2)}$) which was initially missing and the missing portion of the true image. The discriminator utilizes standard convolutional layers to generate final features. The feature map of ground truth image and generated images are compared through loss function to discriminate. Figure 2 shows the full architecture used by the inpainting model.

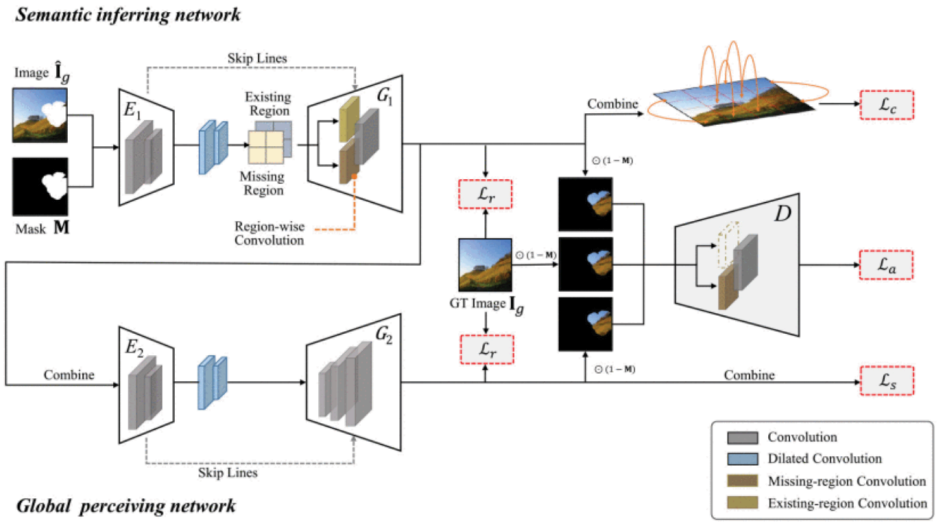


Fig 2: *Architecture of our Inpainting GAN model.*
(Source: "Coarse-to-Fine Image Inpainting via Region-wise Convolutions and Non-Local Correlation" by Ma. Y & et.al.)

2. Loss Function

Overall loss function for the model is as follows:

$$L = L_r + \lambda_1 L_c + \lambda_2 L_s - \lambda_3 L_a$$

a. **Reconstruction Loss (L_r)**: We employ L_1 loss to promise the predicted images at the two stages, including both the existing regions and the missing ones, consistent with the ground truth at the pixel level: $Lr = \| I_p^{(1)} - I_g \|_1 + \| I_p^{(2)} - I_g \|_1$. (3) The reconstruction loss is useful for region-wise convolution filters to learn to generate meaningful contents for different regions, especially at the first stage of the generator.

b. **Correlation Loss (L_c):** The reconstruction loss treats all pixels independently without consideration of their correlation, while the relationship among distant local patches plays a critical role in keeping the semantic and visual consistency between the generated missing regions and the existing ones is not taken care of. Therefore, a correlation loss that can help to determine the expected non-local operation is considered here.

$$L_c = \sigma \sum_{i,j} \| f_{ij} (I_c^{(1)}) - f_{ij} (I_g) \|_1$$

The non-local operation uses the simple outer product between different positions, rather than the non-local block. Formally, given an image $I_c^{(1)}$, $\Psi(I_c^{(1)})$ denotes the $c \times h \times w$ feature map computed by feature extraction method Ψ. In practice, in order to index an output position in space dimension easily, we reshape the feature map to the size of $c \times n$, where $n = h \times w$. Correspondingly, $\Psi^i (I_g)$ is the i-th column in the reshaped feature map $\Psi(I_g)$, where $i = 1, \ldots, n$, of length c. Then, a pairwise function f_{ij} can be defined as a non-local operation, which generates a $n \times n$ gram matrix evaluating the correlation between position i and j:

$$f_{ij} (I_c^{(1)}) = (\Psi^i (I_c^{(1)}))^T (\Psi^j (I_c^{(1)})) .$$

c. **Style Loss(L_s):** Although non-local correlation loss is capable of capturing long distance dependencies, enhancing the restoration of details, it still fails to avoid visual artifacts in unstable generative models. Therefore, we append a style loss to produce clean results and further refine the images perceptually as a whole at the second stage. The style loss is widely used in image inpainting and style transfer tasks meanwhile poses as an effective tool to combat "checkerboard" artifacts. After projecting image $I_c^{(1)}$ into a higher level feature space using a pre-trained VGG, we could obtain the feature map $\Phi_p(I_p^{(2)})$ of the p-th layer with size $c_p \times h_p \times w_p$, and thus the style loss is formulated as follows:

$$L_s = \sum_p \delta p \, || \, (\Phi_p(I_c^{(2)}))^T \, (\Phi_p(I_c^{(2)})) \; - (\Phi_p(I_g))^T \, ((\Phi_p(I_g))||_1$$

d. Adversarial Loss (L_a): Thus, we deploy the region-wise generative adversarial mechanism to the framework, penalizing input images at the scale of patches, which could further preserve local details. While training the region-wise generators, the generated patches will be considered as real and thus labelled as 1. As the discriminator improves, the generator enhances its ability to generate realistic images. After several iterations, the generators and discriminator gradually reach a balance, eliminating the unpleasant artifacts and generating visually realistic inpainting results. Formally, given $I_p^{(1)}$, $I_p^{(2)}$, and I_g , we minimize the following loss to train the discriminator:

$L_a = \alpha \, E(M' - D([I_g \odot (1-M), M])) + E(-D([I_p^{(1)} \odot (1-M), M])) +$

$E(-D \, (I_p^{(1)} \odot (1-M), M]))$

where α is a hyperparameter to define the significance of each part of adversarial loss. M′ is the label matrix indicating the validity of corresponding patches over the entire image, obtained by the nearest interpolation method from the mask M.

The code for this task was taken from this GitHub repository and later modified as per our needs and requirements. The model training was done completely by us as this was not a pretrained model.

3. Training & Implementation Details

a) Training process: Firstly, we pre-trained and stored the weights for the generator using L_r , L_c and L_s for 10 epochs. We then, alternatively trained the generator and discriminator alternatively for 16 epochs storing weights after every 5 epochs. We stopped the training after this. During these 26 epochs we trained the model for the continuous and discontinuous alternatively. After there was no significant loss on the discriminator, we decided to train the model on a continuous mask for the generator only as it was having higher loss values.

b) Data Augmentation: To better train our model, we created a function to extract the mask from the corrupted image. We increased our data by creating our own continuous and discontinuous masks and creating augmented corrupted images to better train our model.

c) Hyperparameters: We took the batch size of 8 images as our training data was less i.e., 5000 images and to achieve better accuracy and generalization. The coefficient for correlational loss was taken to be 0.00001, for style loss it is taken as 0.001 and for adversarial loss it was taken as 1. These values were taken after experimentation done by the authors of *"Coarse-to-Fine Image Inpainting via*

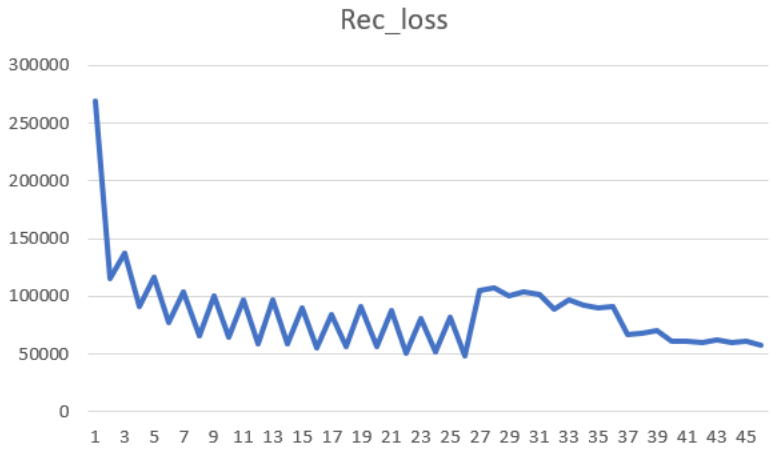
Region-wise Convolutions and Non-Local Correlation." Moreover, Adam optimizer to train the model was used with $b_1 = 0.5$ and $b_2 = 0.9$.

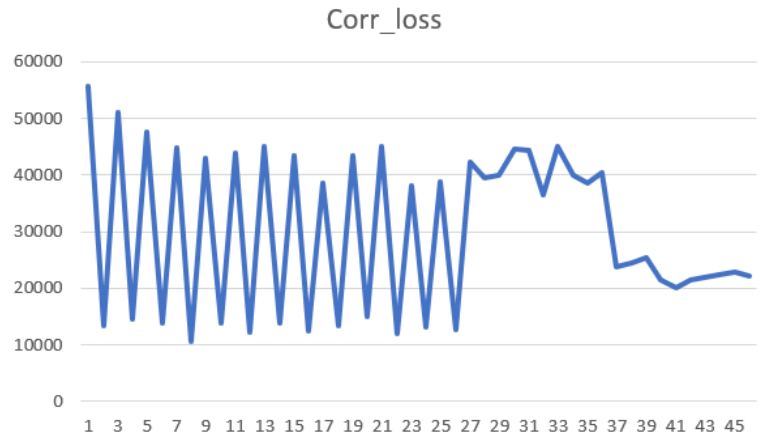
d) Training and Testing Time:

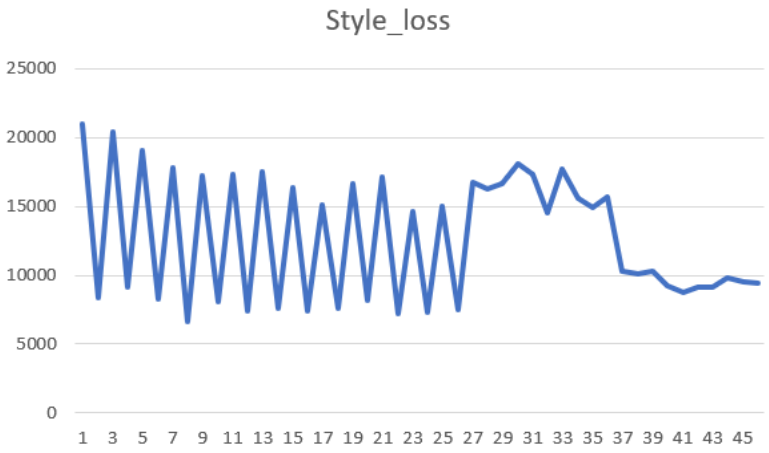
Total training time: ~9 hrs 30 minutes.

Total experimentation: >18 hrs

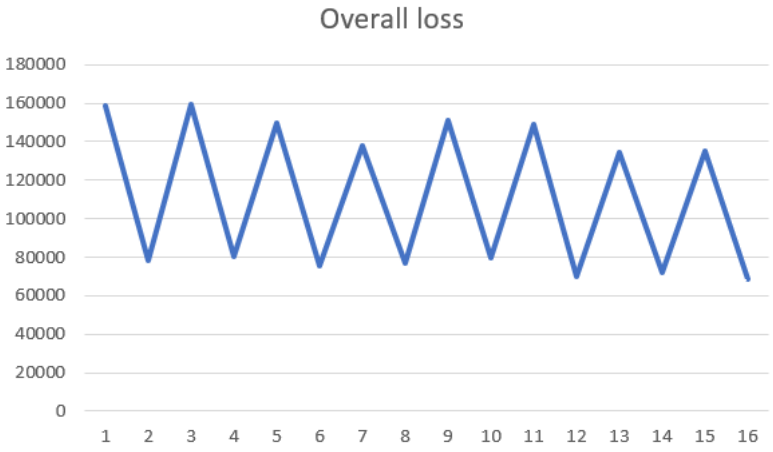
4. Loss – Accuracy Curves







Remember from epoch 1 to 26, we are alternatively training our model on discontinuous and continuous masks. That is why we see a zig-zag pattern. Considering alternative epochs, loss is decreasing. From epoch 27 to the end, we only trained on continuous masks, that is why we see continuously decreasing loss at the end. From epoch 37 to 46 we have decreased the brush size of continuous masks by half which is the reason for the sharp drop. This was done to ensure that the model learns to in-paint smaller regions as well.



5. Results

a) **PSNR:** Average PSNR value for validation set is 41.39 dB.

b) **SSIM:** Average SSIM value for validation set is 0.9483.

Task 2: Multiclass Image Classification using CNNs

1. Architecture

For the multiclass classification problem, we have chosen DenseNet121 as the base model. The rationale behind this choice is DenseNet's unique architecture which

allows it to receive direct input from all previous layers. This promotes feature reuse and helps in combating the vanishing gradient problem. DenseNet121 specifically denotes a DenseNet with 121 layers. The base DenseNet121 model is pre-trained on ImageNet for 100 epochs but we have made its layers trainable to allow the model to adapt to the specific features of the target dataset.

Global average pooling and global max pooling layers are added to the output of the DenseNet121 base model. These layers reduce the spatial dimensions of the output while retaining important information. Global average pooling and global max pooling capture different aspects of spatial information in the feature maps. Combining them is expected to enhance the model's ability to recognize patterns at different scales. After that, we have added a minimum layer that computes the element-wise minimum of the two inputs, which are the global average pooling and global max pooling outputs. The rationale behind it is that taking the element-wise minimum allows the model to focus on the most relevant information from both pooling methods which can help capture diverse patterns from both pooling methods. In other words, this allows us to aggregate the information for both the pooling layers.

Throughout the model, we have introduced three dropout layers with varying dropout rates to introduce regularization and prevent overfitting. Additionally, three dense/fully-connected layers with different activation functions (sigmoid and relu) and varying numbers of neurons. These layers learn additional non-linearity and enable the model to learn complex mappings from the input features to higher-level features. The final dense layer with soft-max activation produces the output of the model which is a probability distribution over the five classes. In these dense layers, we have first introduced the layer with sigmoid activation function so that this layer can help emphasize or suppress certain features, providing attention to specific patterns in the data which is helpful in image classification. The next two layers use relu which is the standard choice for deep layers as it allows the network to learn complex patterns and accelerates convergence during training. Additionally, the number of neurons are increased in each subsequent layer to allow the model to learn more intricate representations before the final classification.

a) DenseNet121 Architecture

DenseNets were introduced to address some of the limitations of traditional CNN architectures, such as the vanishing gradient problem and the degradation problem. DenseNet121 is one of the many pre-trained DenseNets available in open source.

The DenseNet architecture details are as follows:

Dense Block: As mentioned earlier, in DenseNets each layer receives direct input from all previous layers. However, this type of modelling is not feasible when the size of feature maps changes. To resolve this issue, DenseNets are divided into DenseBlocks. In a DenseBlock the dimensions of the feature maps remain the same, however the number of filters between them is changed.

In a dense block, each layer receives input from all preceding layers and passes its own feature maps to all subsequent layers. This dense connectivity pattern helps in feature reuse and encourages the flow of information throughout the network. In DenseNet121, there are four dense blocks.

Transition Block: Between consecutive dense blocks, there are transition blocks that serve two purposes: they reduce the spatial dimensions (width and height) of the feature maps and also reduce the number of channels. This reduction helps in controlling the number of parameters and computation in the network.

Bottleneck Layers: Within each dense block, there are bottleneck layers. A bottleneck layer consists of a batch normalization (BN) layer, followed by a 1x1 convolutional layer with a smaller number of filters (referred to as the growth rate), and finally a 3x3 convolutional layer with the same growth rate. The 1x1 convolutional layer is responsible for reducing the number of channels, and the subsequent 3x3 convolutional layer preserves spatial information. The growth rate in DenseNet121 is 32.

Global Average Pooling (GAP): After the last dense block, a global average pooling layer is applied. This layer reduces each feature map to a single value by taking the average of all values in the feature map. This helps in converting the spatial information into a vector, which is then fed into a fully connected layer for the final classification.

The specific details of DenseNet121 are shown in the diagram below.

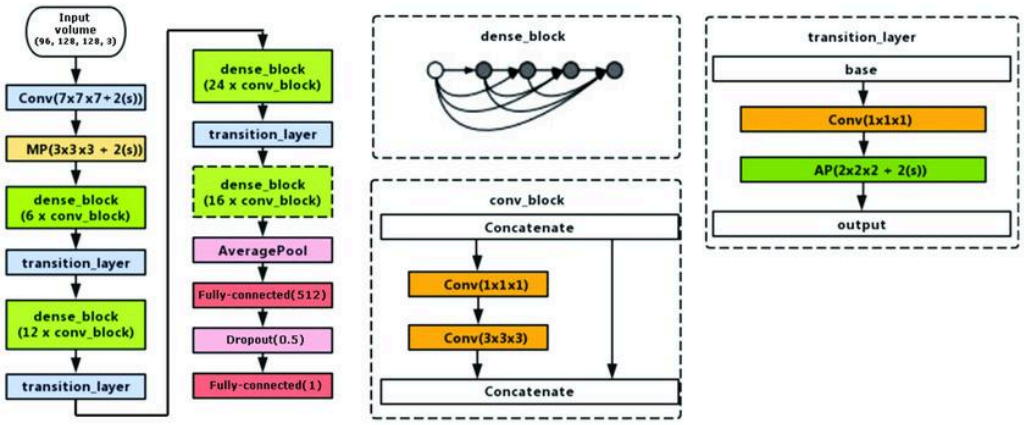


Fig 3: *Architecture of Dense net.*

***(Source**: 3D Convolutional Neural Networks for Stalled Brain Capillary Detection by Solovyev, Roman & Kalinin, Alexandr & Gabruseva, Tatiana. (2021))*

The inspiration to use this model is derived from the classification done in this Github repository.

2. Other Attempts: Failures and Successes

In the initial phase of building the model, we built a basic CNN multiclass classifier as used by us during the classification of MNIST-Fashion Dataset. However, its performance was very poor, even after multiple attempts at hyper-parameter tuning. Thus, we scoured the web for a good model to train an image classification and came across the GitHub link mentioned above. Drawing inspiration from it, we came up with a model that used VGG16 as the baseline and added similar layers to the current model on top. The probabilities obtained in the final layer were then fitted to a Random Classifier, which learned to classify these images based on the combination of probabilities of it belonging to that particular class. The intuition behind this was that since there is high overlap between the images belonging to different classes, combining the probability of it belonging to different classes will reduce the misclassification. The model was successfully trained on smaller training data (nearly half the total training data) with nearly 33% accuracy on validation data (which was also not the complete data). However, due to the limitations on RAM space usage by Colab prevented us from training and testing this model on the complete dataset. Thus, we decided to train the model on DenseNet. First, the attempt was to build our own DenseNet similar to DenseNet121 but with only two DataBlocks. However, the training on DenseNet was extremely slow. (Note: It might also have been that we miss-implemented either the model or the training process.) Thus, eventually we ended up with the current model.

3. Loss Function

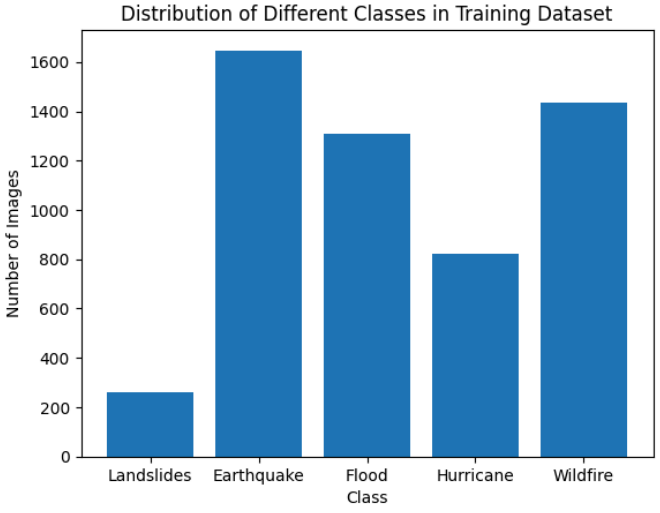
The loss function used is categorical cross-entropy loss. Given y_i as the one-hot encoded ground truth labels and y_i hat as the predicted probabilities, the categorical cross-entropy loss is calculated as follows:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output} \\ \text{size}}} y_i \cdot \log \hat{y}_i$$

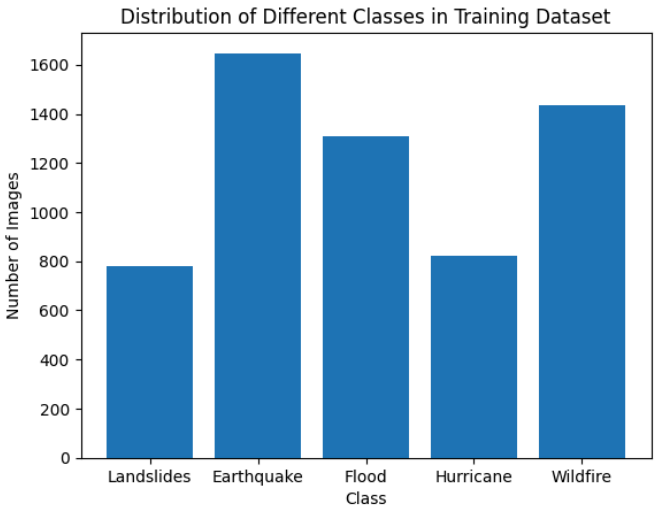
It is used because it measures the dissimilarity between the true distribution (ground truth) and the predicted distribution (model output). Thus, the training process adjusts parameters (weights and biases) to minimize the categorical cross-entropy loss and thus improving the alignment between the predicted class probabilities and the actual class labels.

4. Training and Implementation Details

a) Data Augmentation: On the exploratory data analysis of our training data, we found that the data was highly biased against Landslides class, as shown in the graph below.



Thus, we performed data augmentation to remove this bias by mirroring and blurring landslides images and adding them to our training data. This led to a much more balanced data as the landslides data was tripled. This can be seen in the graph below.



b) Training: For training, we have implemented batch training with batch size = 32. Batch size 32 was giving the beat performance among batch sizes 8, 16, 32 and 64. We have additionally added the feature of early stopping which stops the training if the model does not improve the validation loss for 3 consecutive epochs. This is done to prevent overfitting. The training was initially started for 30 epochs. However, early stopping was encountered and thus the stable minima in validation loss appeared to have been reached at 6 epochs. Thus, the best model stored that value (here the validation accuracy was also maximum) and that is the value used for the testing purposes.

c) Optimizer: Adam optimizer was used because it utilizes adaptive learning rates for each parameter. It calculates individual learning rates for each parameter based on their historical gradients. This adaptability helps handle different scales of gradients, allowing for faster convergence.

d) Hyperparameter Tuning: We focused on two hyperparameters for this stage. One was batch size during training and the other was the learning rate. Accordingly the best combination came out to be (batch_size=16, learning_rate=0.0001). The values were ranging as follows: batch size = [8,16,32,64] and learning_rate = [1, 0.1, 0.01, 0.001, 0.0001].

e) Training Time:

Total training time is 1 hr 40.

Total training time including hyperparameter tuning runs is > 6hr.

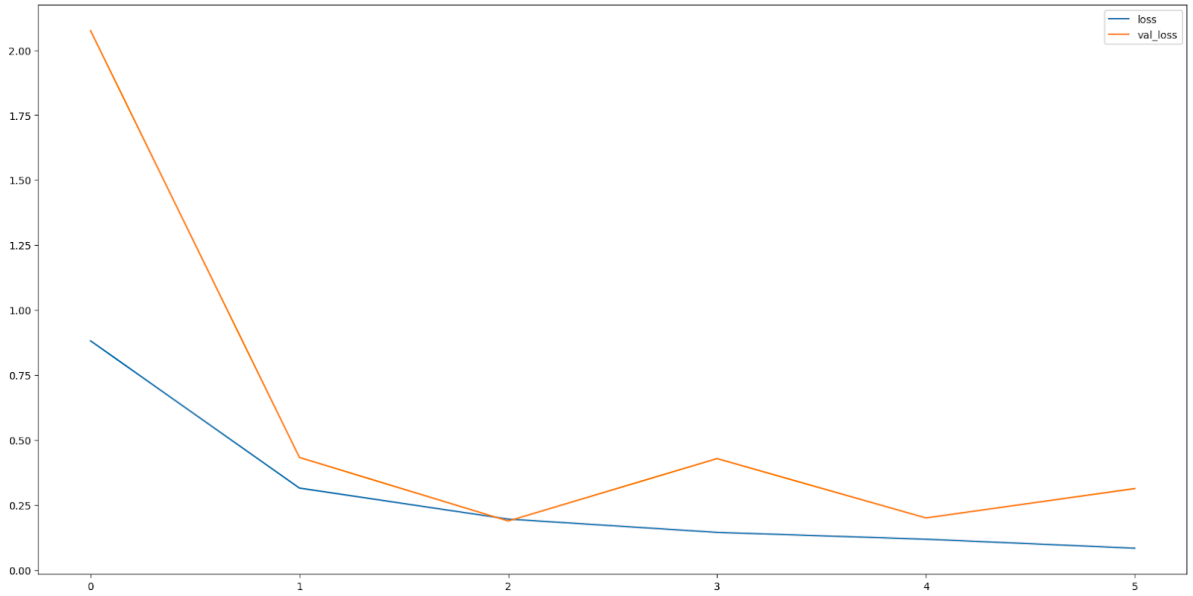
f) Parameters

Total number of parameters: 7335877

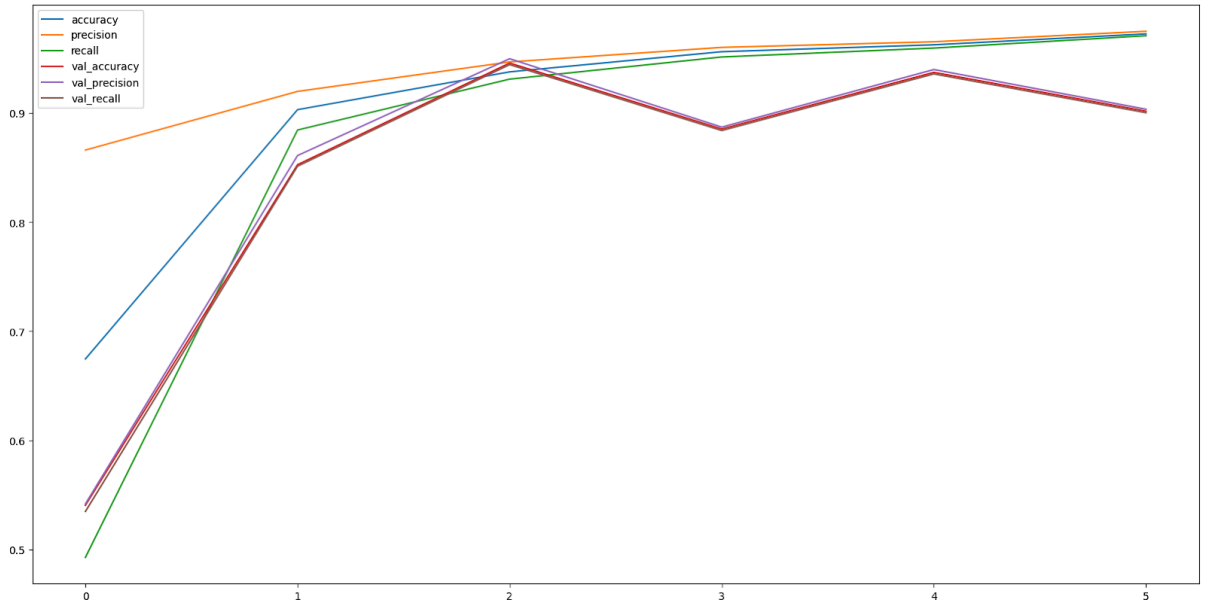
Total number of trainable parameters:725229

5. Loss – Accuracy Curves

a. Training and Validation Loss Curves



b. Accuracy, Precision and Recall Curves for Training and Validation



6. Results

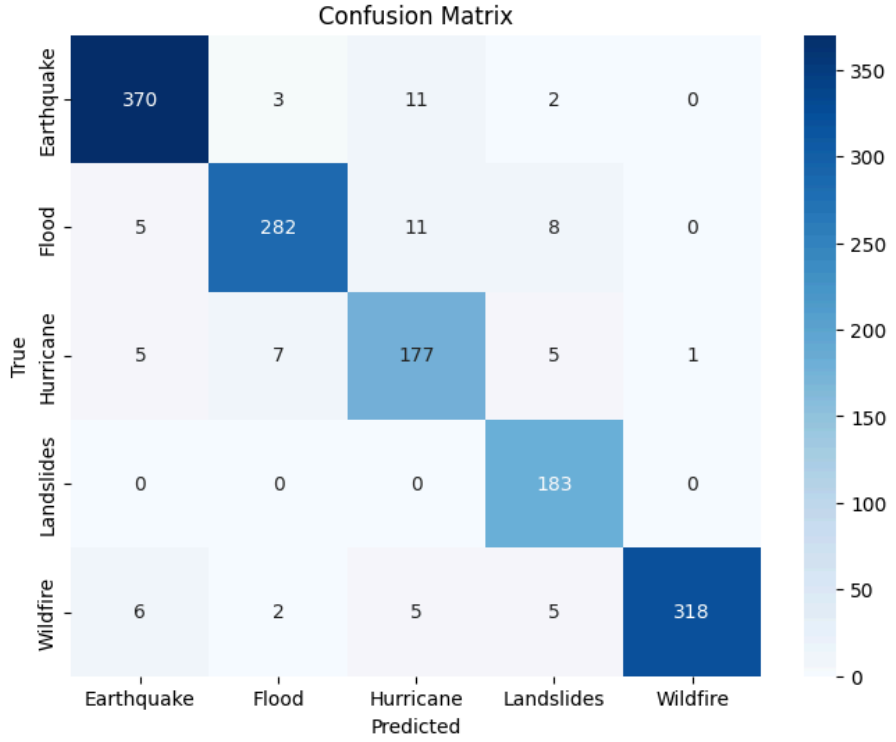
Test Loss: 0.1879

Test Accuracy: 0.9459

Test Precision: 0.9499

Test Recall: 0.9445

Confusion matrix is as follows:



Classification Report is as shown below:

	precision	recall	f1-score	support
Earthquake	0.96	0.96	0.96	386
Flood	0.96	0.92	0.94	306
Hurricane	0.87	0.91	0.89	195
Landslides	0.90	1.00	0.95	183
Wildfire	1.00	0.95	0.97	336
accuracy			0.95	1406
macro avg	0.94	0.95	0.94	1406
weighted avg	0.95	0.95	0.95	1406