UNIVERSITY OF LEICESTER

Department of Informatics

CO7201 Individual Project

Final Report

Data Augmentation for Small Datasets

Achyuth Reddy Bommineni

Word Count: 10520

[09/09/2022]

Abstract

In the past few years, the usage of deep learning models in business applications has seen a significant rise. The main reason behind this was growth in computational power and fruitful results when utilizing these algorithms. The introduction of convolutional neural networks has increased the demand for computer vision to solve problems like medical image analysis, finding text from images, object detection and many more. As these models are data hungry, we will run out of data in certain instances. Data augmentation comes in handy to help with data scarcity.

This report explores all possible classical augmentation techniques and implements them for classification problems on datasets such as fashion Mnist, weather prediction and wheat rust prediction (CGIAR). We extend our work by implementing a general adversarial network for the fashion Mnist dataset and understanding its use case for data augmentation. The final result of this project is implementing traditional augmentation techniques using the ImageDataGenerator library from Keras, analysing the performance difference between model trained using augmented data and original data, and analysing how this augmented data perform using a custom-built and pre-trained model. We used VGG-16 as our pretrained model for this project.

Overall, Models with dataset consisting augmented data performed well compared to original dataset. Training Accuracies for10%, 25% Fashion Mnist dataset with augmented data reached close to 90% compared to training with only original set scored 10%. For CGIAR dataset, which is extremely small dataset, testing accuracies with augmented data is almost 10% higher than without.

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amount to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Achyuth Reddy

Date:[09/09/2022]

Contents

1.Introduction

1.1. Background

Deep learning is a subfield of machine learning which got its due importance in the early 2010s. The main idea behind deep learning is to mimic the architecture similar to that of the human brain to understand the various level of complexities in data. With the advancement in the development of computational power technologies have been heaped for embedding deep learning in many user applications. Modern research in building new frameworks, updating hardware resources and fruitful results when employed pushing this field way ahead.

Deep learning has its own advantages along with some cons. The maximum benefit of these models can be achieved by having huge amounts of data [1]. This served as the greatest point for organisations or institutions that generate a lot of data such as the banking sector but fields like the medical industry and research community are on another side. There are various reasons why these suffer from the scarcity of data. To list a few, class imbalance in the medical sector as only a few people are expected to suffer from medical conditions compared to others, and organizations decline to share their private data due to security reasons associated with it.

With fewer samples, models try to overfit as they may fail to capture the information of data. Does having few samples in the dataset effects deep learning? To some extent, the performance of these models increases logarithmically with an increase in the size of the dataset [2]. To overcome the existing problem of having small datasets, data augmentation comes in handy to help. It is a process of expanding existing sample space by generating new samples by either performing transformations over the data or using deep learning models. The focus of this report, data augmentation not only increases quantity but also reduces the overfitting of models as the variety of samples produced by augmenting effectively leads models to generalize.

In recent years, computer vision tasks such as object detection, medical image analysis and image segmentation got much attention due to the benefits of complex convolutional neural networks, so we constrained our research only to extent of working with image datasets.

1.2. Aim

The main aim of this project is to investigate different techniques employed in augmenting data, develop those techniques according to need of the business problem and analyse the performance of models with augmented data.

1.3. Objectives

The objectives for this project “Data Augmentation for small datasets” are as follows:

- Exploring data augmentation techniques; Geometric and colour transformations, Generative adversarial networks are well-known augmentation approaches used by the research community and for real-world applications. Geometric and colour transformations are also known to be as traditional ways of augmenting data. They include flipping, cropping, injecting noises, brightening and many others. Generative adversarial networks (GAN) involve building deep learning models that are trained to

understand the feature space of real-world data and generate images close approximate to actual data.

- The primary objective is to implement traditional techniques on chosen datasets and then extending it by using GAN.
- The second objective of this project addresses the following questions related to performance. They are:
    - To evaluate whether the model trained with the dataset containing augmentation samples outperforms the model trained with only the original dataset.
    - While using augmented data, does the custom model perform better than pre-trained models?
    - Does the pretrained model have any advantage by using dataset with combination of augmented data and real data compare to using only real data?
    - Analyse the impact on the model performance while considering the dataset by applying traditional augmentation techniques, generative adversarial network augmentation method and both techniques.

1.4. Challenges & Risks

There are certain challenges and risks that we may come across while developing this project.

- Training of deep learning needs heavy computation power. Due to restricted access to computation power, choosing of datasets in a way that can address our top three performance goals will be quite challenging.
- As discussed in objective section, there are many transformations that we can apply on data samples but not every transformation is meaningful, and some might lead to underperformance. So, understanding of datasets is necessity and apply right transformations.

1.5. Restrictions

Due to hardware limitations, GAN implementation was only reserved to one dataset with low dimensional features So as to demonstrate its use case in data augmentation and to address final part of performance objective.

1.5. Report Structure

The next part of this report is continued by brief research on the background study related to data augmentation and other concepts involved. Then in chapter 3, there will be a discussion on the background of datasets used for this research along with all technical pre-requisites that are needed for developing models and limitations of this report. Chapter 4 introduces most of the traditional techniques and general architecture of GAN that we used for data augmentation, and model architecture for classifying images. Chapter 5 presents results answering all questions related to performance analysis which is part of the minor objective of my project. The final chapter of my research concludes with a complete overview of the research with suggestions for possible extensions for this work.

2.Literature Review

The study on number of images needed per class for training deep learning model for classification task is conducted by Saleh Shahinfar, Paul meek and Greg Falzon in their paper [7]. They considered dataset consist of 8 classes in total with 6 classes of animal species, one class of vegetation and other class representing other rare species. The dataset is prepared by considering sub-sampling data from database of wildlife species from Australia, Serengeti, and Wisconsin. Three variants of pretrained models ResNet and DenseNet were used for this analysis. Overall, as number of training images increase contributed positively on the performance that showed logarithmic trend over three different metrics such as per class accuracy, per class precision and per class true positive rate. The extent of improvement slowed drastically after set of 150 images per class. So, they suggested ideally 150 to 500 per each class is sufficient for training the model.

Zeshan Hussain and their team investigated the effects of different data augmentation tasks on image classification tasks in the medical industry [8]. They used classical augmentation techniques such as powers, shearing, rotating images with angle range (10,175), jitter, flipping the images vertically and horizontally, adding gaussian noise, gaussian filters with a variance range between 0.1 and 0.9, and scaling. The tests were carried out on a dataset from DDSM that consists of 1650 mass cases and 1651 non-mass cases. Each augmentation technique was tested separately using the pre-trained model VGG-Net 16. Each set was trained over 2500 iterations with the same hyperparameters such as learning rate of 1e-3, dropout parameter set to 0.5 and L2 regularization 1e-7. Overall, the validation and training scores of all techniques were above 50%. Rotation is a highly significant type as its validation and training accuracy is over 88% and noise augmentation was least effective with an accuracy of 66% on validation and 62% on training. All other methods' performance on the validation set was close in the range of 73% and 88%. My report provides evidence for considering classical augmentation techniques very important.

In the paper generative adversarial network [9], Ian Goodfellow and their team proposed a new framework for generating samples. The framework consists of two models: generator and discriminator. The role of the generator is to produce images by capturing the distribution of data. The role of the discriminator is to predict whether the image that it received is from the original dataset or the one generated by the generator. Both models are multilayer perceptron’s which are trained together. To learn the distribution of the original dataset, the generator is fed with a vector (normally known as noise) to generate an image that possesses the same shape as the original image. Then, it is passed into the discriminator model to test how probable it resembles a real one. The parameters of the generator never get updated directly but instead updated via gradients of the discriminator. The experiments were conducted on Mnist handwritten dataset, CIFAR-10, and Toronto face database. Results of those experiments does not claim that their model performed better when considering other alternatives but provided good scope for improvement.

The paper conditional generative adversarial nets [10] published in 2014 introduced a new method for training GANs by conditioning them explicitly. Normally, all supervised learning networks are supplied with annotated data. While training general GANs we are not using additional information along with data, this implies they miss useful features. To get

maximum out of available data, both generator and discriminator are fed with extra information as an additional layer for training. The extra information can be either class labels of the data or other primary modalities. Suppose we have label dataset of dog vs cat with each shape 224* 224 *3, where the last one"3" signifies channel. As we are aware of class label, we can convert this label as channel with same input dimension of generator and discriminator. The objective function can be modified as below equation [10]:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

Our present report is highly influenced by the survey paper "A survey on Image Data Augmentation for deep learning" by Connor shorten and Taghi M. Khoshgoftarr published in 2015 [10]. Authors conducted deep research into existing techniques that are used in image data augmentation for reducing overfitting due to limited data. The techniques include all geometric transformations, photometric transformations, GANs, neural style transfer and meta learning. Apart from augmentation techniques, authors discussed designing principles that are need for successfully building image data augmentation. They are Size of final dataset, class imbalance alleviation, impact of image resolution, curriculum, and test-time augmentation. According to investigation by Wu et al [12], the model with mix of high and low resolutions images in data outperformed models that trained separately on high and low resolution images. There is no strategy that determines the size of final data set for training a model after augmentation, but the final dataset should not possess images which are obtained highly from colour-space transformations as this will leads to overfit and model performs worse than being trained original data. For example, if there are 100 images of dog and cat each, we perform colour transformation over them and obtain 500 images each. Then training model over this dataset reduces the performance as model never generalizes. The best thing augmentation techniques is that they can be stacked one on another type. Try different methods increase diversity in data compare to perform only single method.

3. Deep Learning

In this chapter, we discuss all technical background information that is used for successfully completing this project. This includes detailed explanation of how deep learning works, key parameters for our deep learning models and short brief on transfer learning.

3.1. General Working process of Deep learning

Deep learning model mimic the structure of human brain to identifying complex information within data. In brain neurons are basic parts, nodes act as pillar for models. So, the terms node and neurons are used interchangeably. Every layer consists of many such nodes which take input, processes them and outputs. There are many such layers that make up a model. At first, the input is converted into n sized vector/tensor, then it is passed for first layer. As every layer consists of many nodes/neurons, at every node, the vector is multiplied by weights. These weights are initialized either randomly or from specific distribution. Then activation function is applied over this calculated weighted sum. The result from node is passed as input to next layer and this chain continues till end of the model. This forward moment of inputs from one layer to another layer is known as forward propagation. Then loss is computed over the results that our model predicts. This function is known as loss function or cost function. The main objective of our model is to minimize this loss function. If our loss is high, then we calculate gradients at each layer with respective to cost function and update all the weights. The process of traversing back to weights and updating them with calculated gradient descents is known as backpropagation. The process of training is continued until we reach convergence point where loss is minimal.

3.2. Parameters of Deep Learning

3.2.1. Convolutional layer

Convolutional layer is primary source of block for building any convolutional neural network. In terms of mathematical definition, convolution is an operation performed on one function by using other function that results in third function. Let us consider, f be our first function and g be our second function. We want to perform multiplicative operation on f by g then their output is given by {f * g}. In view of CNN, Convolution layer feature maps input with set of kernel filters for detecting useful information. Convolution layer reduces computational need by identifying key parts in image such as edges. Throughout the process of training, we try to learn values of these filters. The parameters of convolution layer are:

- Kernel:
  Let our image be represented as matrix with shape m * m and kernel size of k * k, then result after performing convolution operation will be in shape of (m – k + 1, m – k +1). The kernel can be of any shape (not exactly a square matrix). In practice, if our input is square matrix then kernel of square type is preferred. The value of k will be odd so as to reserve central tendency.
- Padding
  If our business needs the shape of image to be unaltered even after applying kernel, then we should add few rows and columns with zero value to image before passing it to convolution layer. Let, P denotes number of padding rows and columns, the value

of P depends on size of the kernel k. Generally, P is equal to (k-1) / 2. Padding is mainly used when there is crucial information present at corners of image.

- Strides
  Normally, when we apply convolution on image, the kernel moves from left to right by one position and then in same fashion top to bottom. We can specific how many steps our kernel should shift. Suppose if stride is (2,3), it means shift 2 units horizontally and 3 unit vertically.
  Finally, if (m, m) denotes shape of image matrix, (k, k) represents size of kernel, stride of (s, s), padding of 'p' and then output of convolution layer is given as follows:

  Output = (m + 2*p – k) / (s + 1)

Table 1 5x5 Image

2	3	6	4	3
1	7	5	2	0
0	5	1	3	8
6	0	9	1	2
4	1	0	1	5

Table 2 : 3x3 kernel

1	0	0
0	1	0
0	0	1

Table 3: Final Output

10	11	16
15	9	10
0	15	7

From above table 1,2 & 3 depicts process of convolution with (5,5) image and (3, 3) kernel with stride of (1,1), no padding and we are not using any activation function here. We perform dot product between coloured green matrix and our kernel to get result of coloured green element in final output. The calculation can be seen as in figure below:

2	3	6	*	1	0	0	=	10
1	7	5		0	1	0		
0	5	1		0	0	1		
First box from above image			*Dot operation*	*Kernel*				*End Result*

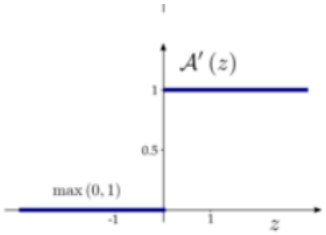
Similarly, we calculated all other values from left to right and top to bottom moving one step at once. We exclude boxes, when our kernel column or row falls outside actual considered image.

Similarly, to convolution layer there is convolution transpose layer which does operation quite opposite to first one. The main idea behind convolution layer is to reduce dimensions of existing image. This is generally termed as down sampling. So, it is mainly used in object identification and classification tasks. In certain tasks, there is a use case where we need to generate images or segmentation. The main idea is to build complete image from some arbitrarily vector that closely identify or represent actual image. This is generally known as Up sampling. Normally, in Up sampling the values are filled either by nearest neighbours, maximum value, bi-linear interpolation or by filling all other positions with zero. But in transposed convolution, we use kernel with desired shape and multiply each element of input with kernel forming new matrix / vector of desired output shape. In the processing of training, we try to find desired values for elements in kernel.

3.2.2. Activation Functions

Activation function plays key role in deciding whether a node of model layer to be activated or not. When you consider node to be as exact replica neuron in terms of brain architecture. The neurons decide to fire or not by basing on intensity of signal it reaches as input. In similar fashion, activation function helps node whether the input should be sent to another layer or not. There are three types of activation function exits. They are

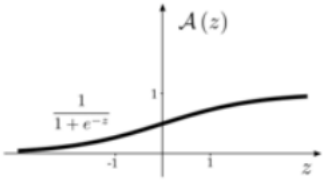
- Step Function
  It is very simple function that fires neuron if the calculated weighted sum value is greater than or equal to threshold value. The neuron does not activate if weighted sum is less than threshold value. The gradient of step function is equal to zero, hence we cannot use this for deep learning models as it does not back propagate. Another main disadvantage is that we cannot use it for multi class classification problem.



Step Function (source: [17])

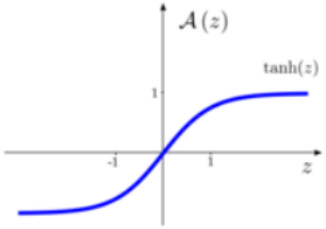
- Linear Function
  In linear function, the activation of neuron depends on weighted sum value of input. The main disadvantage of this type of activation function is the model never learning complexity in data due to its linear structure. The results from first layer of model to last layer remains almost same. The back propagation yields constant value which make model more difficult to learn.

- Non-Linear Function
  To overcome problems associated with above step-function and linear function such as failing to back propagate and learn complexities in the data, non-linear versions of activations are introduced. Non-linear activations are derivable So during the process of training nodes can update their weights according to loss function. There are many types of functions exist. We will discuss few most popular ones. They are listed below:
  - Sigmoid
    The sigmoid function outputs by taking any real value in the range between 0 and 1. The shape of this function is “S” curve. The maximum value of this function is 1, it means the more positive our input will incline in resulting 1 and the more negative our input will incline towards 0 as it is minimum that function can output. The above restriction was great disadvantage for this function as it leads to vanishing gradient problem. So, it is mainly used when output layer and belongs to binary type classification problem.



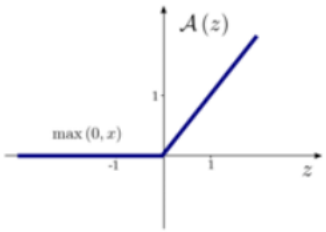
Sigmoid Function (source: [17])

- Tanh
  The range of output for tanh function is in between -1 and +1. The shape of this function is like that of above logistic function/sigmoid function but main difference between both is tanh is centred at zero while sigmoid is not. Similarly, Tanh to suffer with vanishing gradient problem but it is most preferred non-linear function in hidden layers compare to sigmoid.



Tanh Function (source: [17])

- Relu
  Relu activation function is quite similar to that of linear activation function, but it only activates when calculated weighted sum is more than zero. It overcomes problems associated with sigmoid function. This function to has one disadvantage as weighted sum becomes less than zero, then overall neuron will not activate forever. This problem is also known as dying Relu.



Relu Function (source: [17])

- Leaky Relu
  To address the issue of dying Relu, the function is adjusted by multiply with very small value.
- SoftMax
  The SoftMax takes any real valued vector as input and outputs probability related to them. It is mostly used for multiclass classification problems and in only last layer of neural network.

There are other activations apart from those mentioned above, some were extensions to leaky Relu but most widely used was relu and leaky relu for hidden layers.

3.2.3. Dropout layer

Complexity of architectures of deep learning models increasing with increase in critical problems to be addressed. These complex structures try to grasp minute information with data that leads to drop in the performance on unseen data. This is generally termed as overfitting. Regularization techniques like L2 norm and L1 norm are introduced to penalize weights as one possible solution to overfitting. The second possible solution to this is stopping training of model once the score of validation becomes worse comparing to training set. This is also unknown as early stopping. Dropouts are another form of regularization technique that not only addresses over fitting but can also help in combing different complex neural architectures together [16]. Dropout refers to dropping of connections to neurons randomly. While training we specify a real number between 0 and 1that signifies percent of neurons to be dropped.

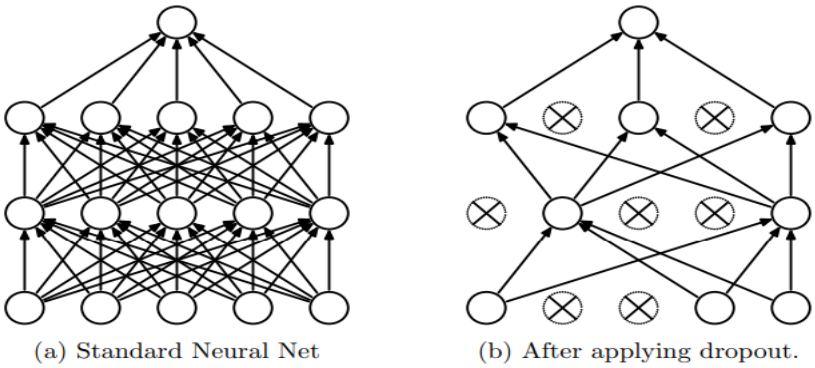


Fig: 3.2.1 Neural networks with and without neural network (*source: [16]*)

From figure 3.2.1, we can see how dropout has changed architecture of neural network model.

3.2.4. Pooling Layer

Output of every convolution layer tries to map its features with location derived from input. This location dependency reduces the performance of our model. Pooling layer is introduced to remove dependency on input location. Pooling layer is placed after applying non-linearity to convolution layer's output i.e., after applying activation function. It also consists of matrix or filter that is similar to kernel in convolution layer. This filter is moved along the feature map from left to right and top to bottom. Main advantages of pooling are they used cost of computation and reduce model invariant due to distortions. There are three types of pooling layers exists. They are listed below:

- Max Pooling
  In max pooling layer, a matrix (filter) of specified size and stride is considered to down sample the feature maps generated by before layers specifically convolution layer basing on maximum value. Below tables give details about how max pooling layer works.

Table 4 5x5 Image

2	3	6	4	3
1	7	5	2	0
0	5	1	3	8
6	0	9	1	2
4	1	0	1	5

Table 5 Max pooling layer

Table 6 Output of max pooling layer

7	7	8
9	9	9
9	9	9

From above table 4 represents our image of shape (5,5) passing through max pooling layer of size (3,3). Let us consider coloured box, in max pooling layer we will find maximum value out of all 9 elements and replace first output element with it. Similarly, we pass from left to right and top to bottom until our shaded region remains with image boundaries.

- Average Pooling

  In Average pooling, down sampling the feature maps by filter with desired shape and stride is done basing on mean value of elements. Below tables give details about how average pooling layer works.

Table 7 5x5 input image

2	3	6	4	3
1	7	5	2	0
0	5	1	3	8
6	0	9	1	2
4	1	0	1	5

Table 8 Average layer of 3x3

Table 9 Output of average layer

3.33	4	3.55
3.77	3.66	3.44
2.88	2.33	3.33

The procedure of average pool layer working is almost similar to max pooling layer. The only difference is that we find average of shaded region and replace it in output.

- Min Pooling

  In min pooling layer, a matrix (filter) of specified size and stride is considered to down sample the feature maps generated by before layers specifically convolution layer basing on minimum value. Below tables give details about how min pooling layer works.

Table 10 5x5 input image

2	3	6	4	3
1	7	5	2	4
0	5	13	3	8
6	10	9	6	7
4	1	8	1	5

Table 11 Minimum pooling layer 3x3

Table 12 Output of min pooling layer

0	2	2
0	1	3
0	1	1

The working procedure is similar to max pooling layer, but we replace with minimum value here.

Pooling layer is avoided if the model is too deep with many layers instead it is replaced my convolution layer with different strides. Max pooling layer is most used one compared to others as it preserves generalization.

3.2.5. Batch Normalization layer

Training of deep learning models is very complicated as the distribution of inputs changes across various layers present internally. The changes in distribution of inputs are commonly termed as co-variate shift. This problem can be seen in deep learning models due to effect of internal layers. The problem can be erased by adapting to domain specific changes. Another approach for reducing this internal co-variant shift is by making changes to the distribution of input before passing it to another layer. The performance of deep learning models can be improved by having whitening inputs that is inputs with mean zero and variance of one. As cost of normalizing both inputs and outputs together are high, so each input is normalized. Whitening every layer`s input induce linearity that eventually hurt model in learning distribution of data. Two other learnable parameters have been added to whiten matrix that usually get updated batch wise that kept non-linearity introduced by other layers.

3.2.6. Optimizers

Optimizers are method or algorithms that is used to update weights of all neurons basing on loss function that we choose to minimize. They play key role in convergence of training model. Choosing right optimizer is important as it may affect how you reach global minima, and it depends on type of business problem we are addressing. There three major types of optimizer variants exist. They are briefly discussed below:

- Gradient descent
  These optimizers minimize the loss of model by calculating gradients of loss function and updating weights continuously with respect to calculated gradients.
  - ❖ Batch Gradient descent
    We first pass all our samples to model, then calculate error with respect to loss function. Then we find gradients and update our parameters accordingly. It is computationally intensive as it depends on complete training dataset and very slow if our dataset is large.

  - ❖ Stochastic Gradient descent
    Instead of passing all samples for training model and then update parameters, we pass single sample at a time to model and then update weights based on gradients. Even though it addresses computational bottleneck, the major disadvantage is computed gradient does not

reflect entire training dataset. This leads to fluctuations in the loss function due to gradients variance.

- ❖ Mini- Batch Stochastic Gradient descent
  The main benefit of batch GD is that represents whole dataset while the benefit with stochastic GD is that it creates noise which helps in escaping from saddle points. To take advantage from both above methods such as batch gradient descent and stochastic gradient descent, mini-batch stochastic was introduced. In this changes to parameters are done by calculating gradients after passing only sub-set of training data to the model. Mini-Batch GD is very effective in the case of huge datasets.

- Momentum Based
  In Mini-batch GD, the gradients oscillate to and forth in an attempt to converge to global minima. The back and forth moment increase time for training our model. Instead of directly making changes to parameters based on gradient descent, keeping record of previous gradients, and assigning some weightage to it will help to be along axis. This exponential weighted moving average can smoothen the path of traversal. This is known as momentum. While updating parameters, we first calculate momentum term and then adjust accordingly so as to keep track of the direction of path.

- Adaptive Gradient method
  All above mentioned methods, address by considering changes to gradients and fixing learning rate hyper-parameter constant. Even with changes in parameters are based on gradient descent or momentum based algorithms, sometimes learning rate plays key role as it decides how large or how small the next step should be.
  - ❖ AdaGrad
    As per this algorithm, the learning rate is equal to the tuning parameter by the square root of the sum of the summation of squares of all previous gradients plus epsilon. The addition of epsilon is to alleviate dividing by zero error. The main disadvantage with AdaGrad is the monotonically decreasing learning rate. During the process of training, the denominator part of the learning rate increases with an increase in the number of previous gradients. This growth will gradually shrink the learning rate, and over time it vanishes.

  - ❖ Adadelta
    Adadelta is developed to overcome the vanishing learning rate due to increase in a summation of past gradients in Adagrad. Instead of using summation of accumulated previous gradients, they proposed to use exponentially decaying average of gradients.

  - ❖ RMSProp

This algorithm was developed at same time period of Adadelta, even the intuition behind RMSProp is same. They proposed usage of exponential decaying of gradients.

- Adam
  Adagrad works well when gradients are sparse and in case of non-stationary settings RMSProp works well. Adam was introduced to take advantage of benefits from RMSProp and Adagrad methods. Running average of both first and second moments of gradients are utilised for updating any parameters serves as main advantage of this algorithm [15].

Overall, from all above types of optimizers, Adam is mostly widely used algorithm in deep learning models. It does not mean Adam is best among all other optimizer, while trailing deep learning we have to change them to observe in which case our model is performing better.

3.3. Transfer Learning

The utilization of existing trained model for solving present business problem by either extending it or using as it defined is known as transfer learning. Transfer learning was introduced to overcome few problems associated with traditional machine learning models such as training cost. Training of traditional ML models are computationally intensive compared to models trained by transfer learning as they require huge amounts of data. Time for ml model training is longer as the model needs to find best optimal solution, but transfer learning models extend training from already existing weights which are trained over huge dataset. Similarly, like other models, before applying transfer learning, we need to have good understanding of present dataset and know the details of dataset used for training pre-trained models. Pre-trained models are nothing but models that have already been trained. They are key part of Transfer learning. Suppose if our present dataset belongs to images from medical domain and we are using pre-trained model that is trained using dataset consisting fashion products, then results of our present model using transfer learning might end with poor performance due to learning variability. In such cases, we should use existing models and add additional layers to help model understand variability in domain. There are many pre-trained models such as VGG-16, Inception V3 and many more that are available as in-built packages. In our work, we are going to use VGG -16 from keras library to train on our datasets with augmented and without augmented data.

VGG-16 is one of the famous models which was published in 2013 but won ImageNet competition by using it in 2014. The model is shown in figure below.

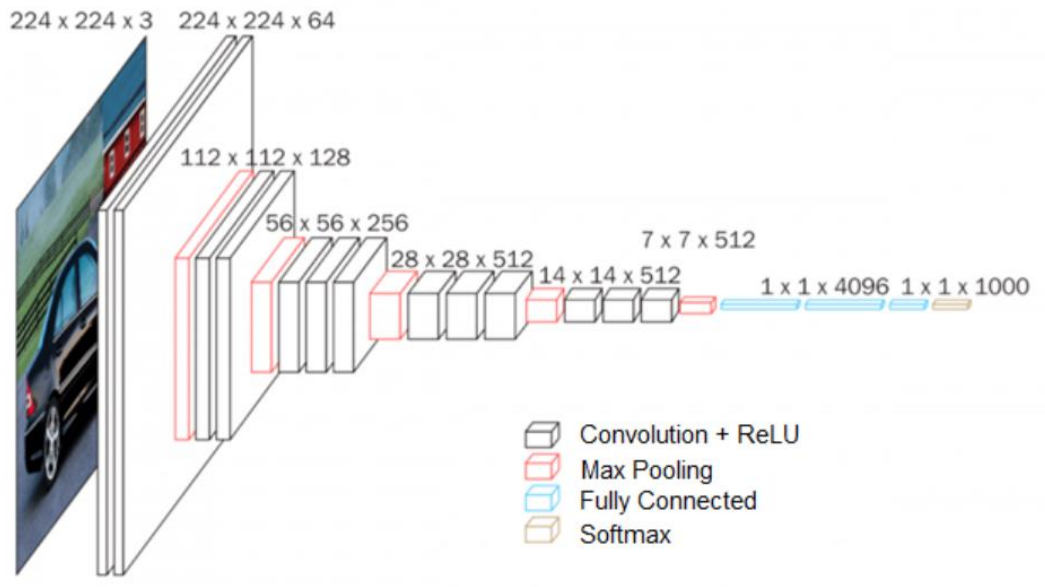


Fig. 3.3.1 VGG network (source: [18])

From fig 3.3.1, we could see that initial input size of VGG network is (224, 224, 3) dimensional vector. First two layers are convolutional with 64 kernels of (3,3) size, zero padding and (1,1) stride followed by one max pooling layer of stride (2,2). The parameters of convolution layer like kernel size, padding and stride are same throughout the architecture. This made VGG stand out exceptional compared to previous architectures like Alex Net which got different sizes of kernel and strides. Then, two layers of convolution with 128 kernels each. Next three sets of one max pooling layer followed three convolution layers are repeated with 256kernels in first set, 512kernels in second set, and 512 kernels in third set respectively. Then, (7, 7) dimensional max pooling is applied for 512 inputs from previous layer. Two dense layers with 4096 neurons each followed by another layer of 1000 fully connected neurons. Finally, last dense layer with 1000 neurons and SoftMax as activation function. Relu activation function was used in all hidden layers.

4. Details of Augmentation

This chapter provides clear details about datasets that we used in our project, explanation about all traditional methods of augmentation.

4.1. Datasets

4.1.1. Fashion Mnist:

Zalando is German online retailer that sells fashion products such as shoes, shirts, trousers and many more. Fashion Mnist dataset is prepared from article of Zalando that is intended to replace original Mnist. Original Mnist dataset consists of handwritten images of numbers which is widely used by research community for bench marking their results. This present dataset consists of around 60,000 images as training set and 10,000 images set for testing. In our research, we are only going to focus on training set which consists of images with quality of 28 * 28pixels each and 10 different labels. The labels include trousers, sandals, shirts, sneakers, coat, pullovers, dress, T-shirt, ankle boot and bag. About 20,000 images are reserved for purpose of testing the accuracy of model. This dataset is present in Keras data library.

4.1.2. Weather

In 2018, the Mendeley data site published a weather dataset for image classification [14]. It consists of images taken at various instants of four different weather conditions cloud, shine, sunshine, and rain. There are 1125 images with ununiformly distributed class labels.

4.1.3. CGIAR computer vision for crop

The dataset is from consultative group for international agriculture group (CIGAR) hosted on Zindi platform for classifying wheat rust disease [13]. Throughout the continent of Africa, wheat rust is very common disease among plants that leads to low yielding per year which in turn affects livelihood and increase food scarcity. The dataset set consists of 876 images and belongs to one of three class labels. The three class labels denote either healthy plant or stem rust or leaf rust. Proportion of each class among dataset is unequal making it unbalanced data with multi class classification problem.

4.2. Data Augmentation

4.2.1. Traditional data augmentation

Data augmentation which involves no physical learning process for generating data from existing data is considered as Traditional type of augmentation. They are mostly classified into geometric transformations, noise injection and colour space arrangements.

We chose to apply below nine techniques to our datasets by specified batch size respectively. In implementation, batch size presents number of augmented images we needed from method. Images are chosen at random from data set we pass.

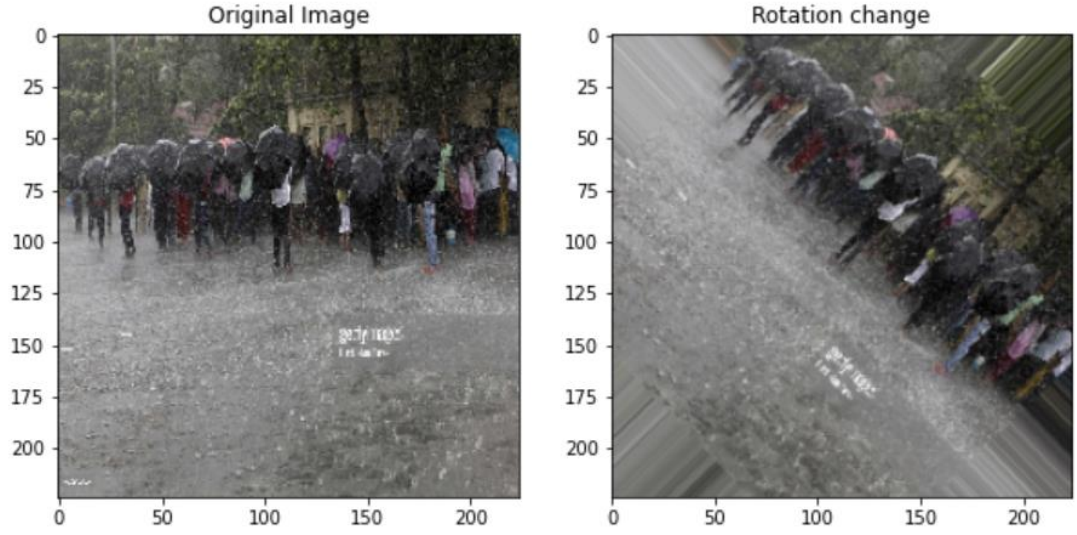
Geometric Transformations

Rotation

Rotation is one of the most classical forms of technique for augmenting data. The image is rotated along x-axis either towards left side or right side with desired angle range between 0 and 360. Every angle of rotation produces unique type of image.

Let us consider x and y be the co-ordinates of the image that we intended to rotate by angle "θ", then new image co-ordinates are given by equation as follows below [3]

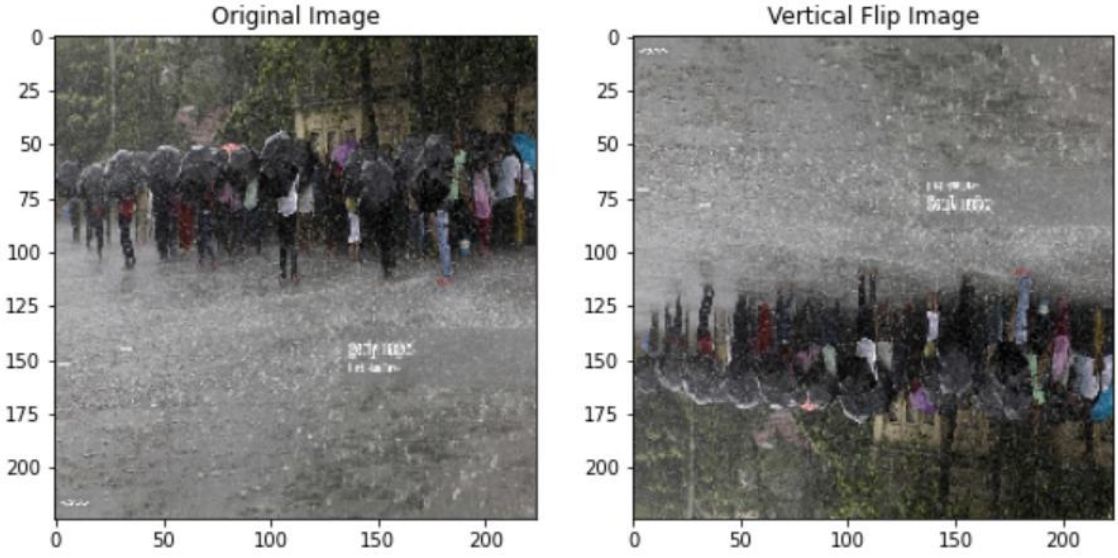
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix}$$



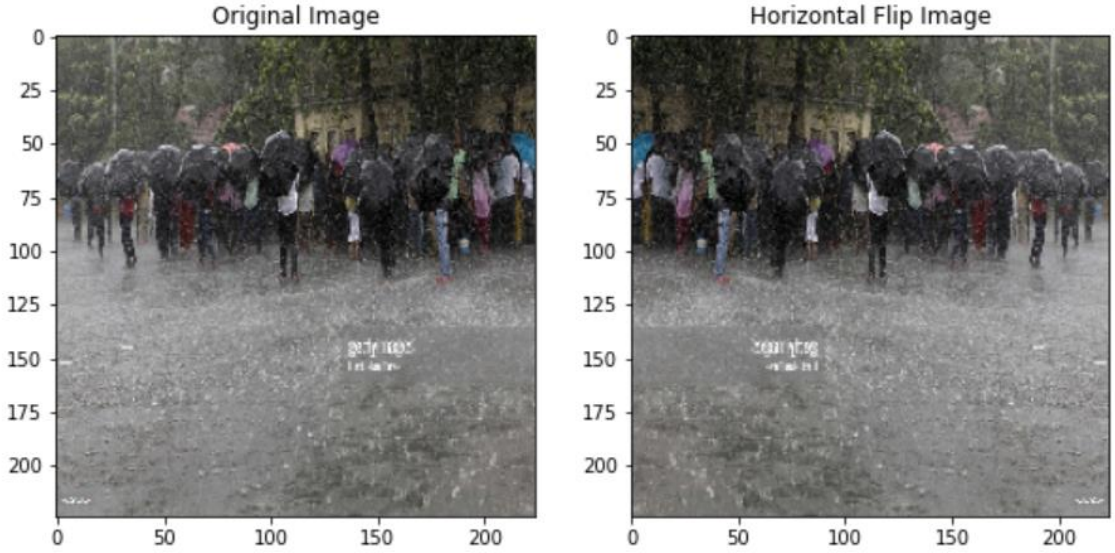
Flipping

Flipping protects actual image structure while rearranging the pixels along either horizontally or vertically. Not in every case flipping makes sense. Before applying flipping technique, we must understand context of data carefully.

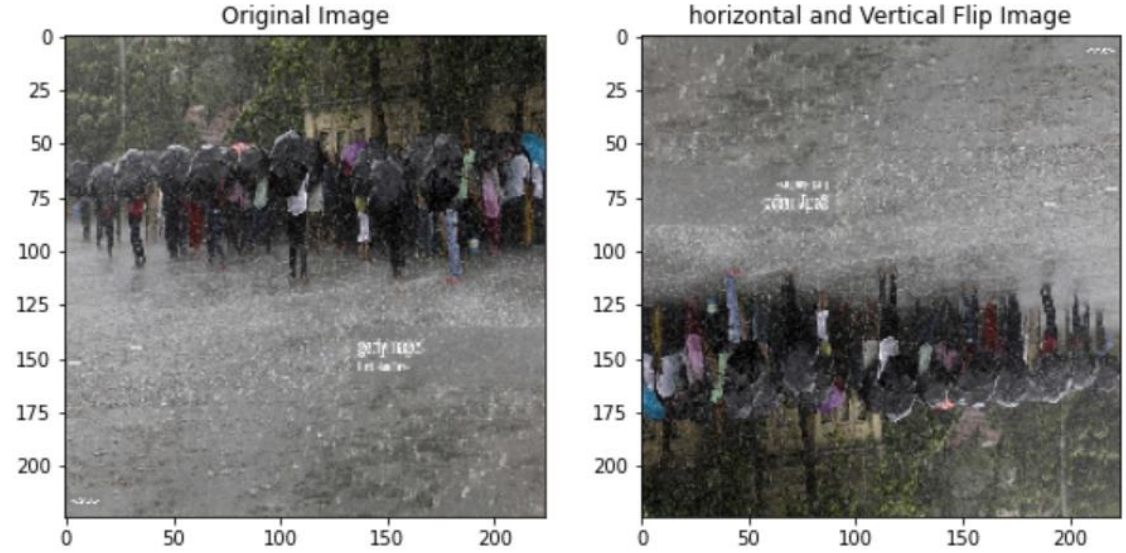
Vertical Flipping: Reversing of image along x-axis in such a way that lower part will be towards upper part and upper part on the lower end.



Horizontal Flipping: Flipping the image along horizontal axis is known to be as Horizontal Flipping. It is also known as left-right reversal or mirror image. Physical dimensions such as height and width remain unaltered.

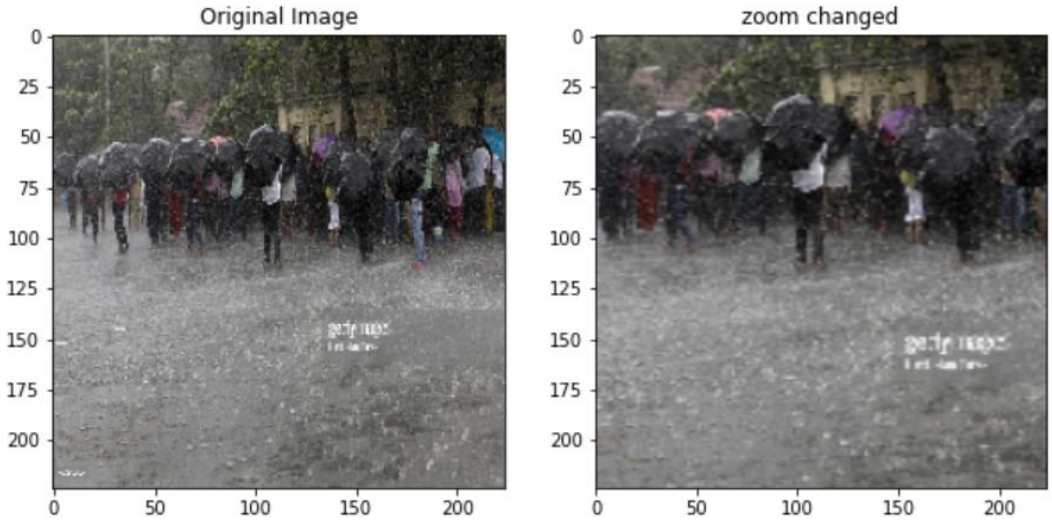


Vertical and Horizontal Flipping: This form of augmentation is achieved by performing reversal of active layers of image along with horizontal axis followed by vertical axis. The resulted image will possess completely different view from human perspective.



Zooming

Zooming is a method of scaling the image outward. Zooming focus on centre part of the image. The range lies in interval of [-1, +1], where upper bound denotes deep focus.



Noise Injection

Noise injection is all about manipulating random pixels of image that is either by increasing the brightness or the darkness and at the end resulting in plausible images. Adding noises must be fundamental image processing method because real world is far from perfect. In this report, we discuss three most popular types of noises. They are:

Gaussian Noise

Gaussian noise mainly tries to alter grey values within our real image, so its distribution is represented in terms of grey values(g). The gaussian distribution is equation is represented as below:

$$P(g) = \sqrt{\frac{e^{\frac{-(g-\mu)^2}{2\sigma}}}{2\pi\sigma^2}} \qquad [4]$$

Where μ signifies mean and σ denotes standard deviation

Poisson Noise

This noise is also known as shot noise or quantum noise or photon noise as this is generally present in medical images. Below equation of distribution denotes Poisson distribution:

$$P(x) = e^{-\lambda}\frac{\lambda}{x!} \qquad [4]$$

Where the x denotes 0,1,2,3 and so on, constant value of e is 2.7182 and λ denotes number of events per interval.

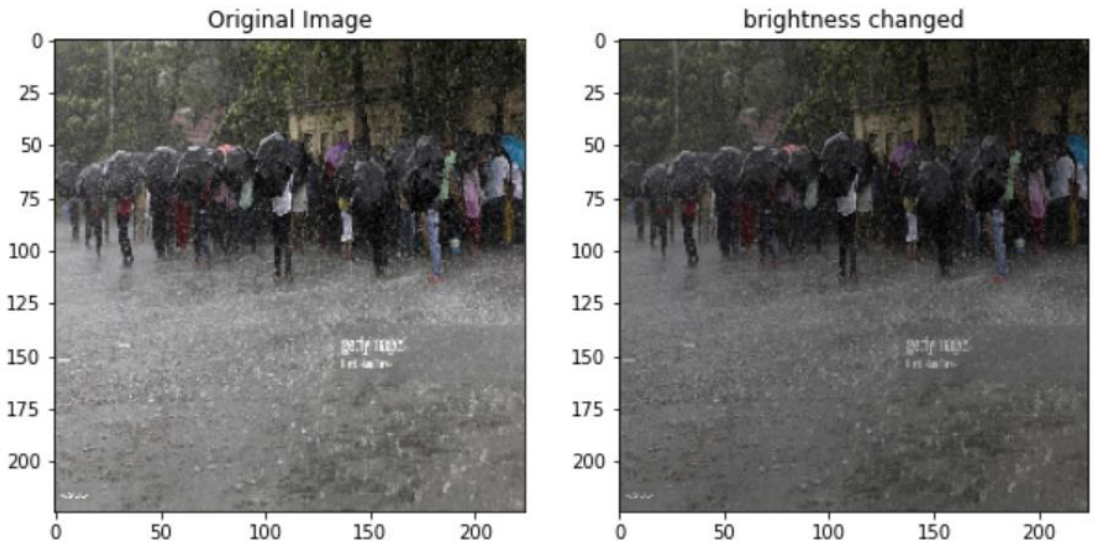
Speckle Noise

Speckle noise is quite similar to gaussian noise but is not as common as gaussian, and also known as multiplicative noise. Images with this noise will appear in medical imaging which are based on laser and ultrasound. The probability density function is represented by gamma distribution.

Colour space arrangements

Normally every image is stored in the form of height * width * channels matrix which represents red, green, blue values. Humans can easily detect any changes or similarities between different colour transformations, but machine fail to infer. By applying variety of

shades on images improves diversity that in turn leverages model training. Below sample represents changes in brightness.



ZCA_Whitening

Zero Component Analysis is commonly termed as ZCA whitening. ZCA transformations project edges of image which can help image processing as convolutional neural network layers try to detect edges.

Let X be data or image that we are using. Steps followed are as follows:

1) We perform normalization of data

$$X' = X / 255 \quad [5]$$

2) Then we perform mean(μ) normalization to above result

$$X'' = X' - \mu \quad [5]$$

3) Later we calculate singular value decomposition for the mean normalized co-variance matrix. Finally, we apply Whitening formula as given below:

$$X_{ZCA} = U.diag\left(\frac{1}{\sqrt{diag(S)+\varepsilon}}\right).U^T.X' \quad [5]$$

Where U signifies eigen vector matrix

diag(A) signifies diagonal matrix of given matrix A

ε signifies whitening co-efficient

S signifies Eigen matrix of singular value decomposition of co-variance matrix

U^T represents transpose of U

4.2.2. GAN

GANs mainly consists of two models: Generator and Discriminator. The role of generator is to generate images that is as realistic as it possible. The role of discriminator is to trace the real images.

In our present experiment, as our dataset is images with labels. To use of extra information of our dataset, we built Conditional generative adversarial networks. In both discriminator and generator, we used convolutional neural layers and dense layers. The model structure and training process of GANs are cleared explained in next section.

5.Methodology

In this chapter, we will discuss about ImageDataGenerator and how we utilized it for achieving traditional augmentation, implementation of GAN model that we used for generating fashion Mnist images, and custom models that is implemented for classification.

ImageDataGenerator

ImageDataGenerator is one of the popular classes of keras library that helps in dealing with image datasets. It got many built in methods that can deal with traditional data augmentation. This includes rotation, cropping, zoom, shear transformation, colour transformations and ability to include other defined methods or functions using preprocessing function argument. Actually, ImageDataGenerator library is on fly generator that is augmentation is done at real time during training of model. The end results do not include original data but only consists of transformed images. In our project, we defined DataGenerator class which utilized existing keras class in the defined methods.

```
107
108     def transform(self,data,labels,batch_size = 10):
109         """
110             Generates augmented images with specific transformations that all fitted
111             data: the data to be transformed
112             labels: class labels
113             batch_size: number of samples per transformation
114             returns: Two numpy arrays representing transformed data and corresponding data
115         """
116         shape = data[0].shape
117         generateddataset_x = [] # for storing generated image
118         generateddataset_y = [] # for storing corresponding label of image
119         for transformation in self.transformations:
120             transformation.fit(data)
121             it = transformation.flow(data,labels,batch_size = batch_size)
122             for x_batch,y_batch in it:
123                 for i in range(batch_size):
124                     img = (x_batch[i].reshape(shape)).astype("uint8")
125                     generateddataset_x.append(img)
126                     generateddataset_y.append(y_batch[i])
127                 break # As transformation. flow() returns gnerator, the loop continues indefinitely
128         return np.array(generateddataset_x, dtype = np.float),np.array(generateddataset_y,dtype = np.float)
129
```

Fig.5.0. Transform method from DataGenerator class

Transform method is one of the important methods in our DataGenerator class for performing all classical augmentation techniques which is shown in snippet of code (figure 5.0). It takes three arguments. First two arguments are data and labels which represents data to be transformed with corresponding labels. The third argument batch_size represents number of samples we need per each transformation and default value is 10. Variables generateddataset_x and generateddataset_y is declared to store images and corresponding labels generated after performing necessary transformations. The class variable self.transformations consists of list of transformations to be performed on our dataset. We defined 12 techniques such as horizontal flip, vertical flip, shifting, poison noise, gaussian noise, speckle noise, shear, colour brightness, ZCA whitening, rotation and zoom. We can exclude any technique by setting it False at time initializing our class object. When we call fit, it calculates all necessary internal data statistics that need for transformation. This is

mainly needed for ZCA whitening. Then, we call method flow on transformation which returns generator. In next for loop, we are iterating over this generator and storing transformed data. As generator runs indefinitely, we apply break to explicitly stop the iteration.

Generally, in our project, let our original dataset consists of 1000 samples. If we want to generate 50 percent extra samples by applying 10 transformations. Then we calculate batch_size parameter of our transform method by dividing number of original samples by product of percentage of samples needed and transformations. The above formulation helps in generating equal number of samples per technique.

GAN model for Fashion Mnist Dataset:

```
1  def fashionMnistGenerator(noise_dim = 100):
2      """
3          Generator model for fashion mnist dataset
4          returns Model
5
6          noise_dim: size of our initial noise
7      """
8      #Label embedding
9      label_input = Input(shape = (1,))
10     embedding_layer = Embedding(input_dim = 10 , output_dim = 50) (label_input)
11     dense_layer1 = Dense(49)(embedding_layer) # dense layer for generating vector that is equal to dimension of image input
12     label_shape = Reshape((7,7,1))(dense_layer1) # Converting above generated vector into desired for concatenating with ima
13     #reading the image
14     noise_input = Input( shape = (100,))
15     dense_layer2 = Dense(7*7*63, activation = "relu")(noise_input)
16     batch_norm = BatchNormalization()(dense_layer2)
17     noise_reshape = Reshape((7,7,63))(batch_norm) # Reshaping our noise into (3,3,128)
18     #Adding label as extra channel to our image
19     merge_image = Concatenate()([noise_reshape, label_shape])
20     #second convolution transpose layer
21     convtrans1 = Conv2DTranspose(32, kernel_size = (3,3), strides = (2,2),padding = "same",activation = "relu")(merge_image)
22     #third convolution transpose layer
23     convtrans2 = Conv2DTranspose(32, kernel_size = (3,3), strides = (2,2),padding = "same", activation = "relu")(convtrans1)
24     #Convolution layer
25     conv1 = Conv2D(1,kernel_size =(7,7),padding = "same",activation = 'tanh') (convtrans2)
26     #Model
27     mnist_gen_model = Model([noise_input,label_input],conv1)
28
29     return mnist_gen_model
30
```

Fig 5.1. generator for fashion Mnist

The main purpose of generator in GAN is to rebuild the image from some arbitrary noise that should resemble exactly like original image. Lines 9 to 11 from figure 5.1 depict conversion of our label to a vector that can be added as extra channel to image. In the first step, we are modifying our label with dimension (1,) into (50,) by passing it to embedding layer. Generally, we use one hot encoded for class representations. In one hot encoding, every label has fixed representation which does not give meaningful representation for model learning. Embedding layer is one such layer which learns representation for each in process of training. This vector representation is fed into dense layer of 49 nodes, reshaped into (7,7,1) vector which is similar to shape of initial noise vector that will be used for reconstructing image. Parallelly, the noise vector of size 100 is passed to dense layer of 7*7*63 nodes. Batch normalization is performed for the above results in order to have vector values in constraint limit (max, min). Reshaped noise is concatenated with embedded label so the concatenated vector will be (7,7,64) dimensional. Then, we applied two layers of convolution transpose operation with each having same parameters. Conv2DTranspose is similar to convolution operation but performs opposite way. We have convolution as last layer which outputs image

having same dimensions as original dataset (28,28,1). This is overall summary about our generator model architecture as seen in above figure.

In [307]:

```
def fashionMnistDiscriminator(image_shape = (28,28,1)):
    """
        discriminator model for fashion mnist dataset
        returns model
        image_shape: represents shape of image
    """
    #label embedding
    label_input = Input(shape = (1,))
    embedding_layer = Embedding(input_dim = 10 , output_dim = 50) (label_input)
    dense_layer1 = Dense(28*28)(embedding_layer) # dense layer for generating vector that is equal to dimension of image inp
    label_shape = Reshape((28,28,1))(dense_layer1) # Converting above generated vector into desired for concatenating with i
    #reading actual image
    input_image = Input(shape = image_shape)
    #Merging label embedding as extra channel to image
    merge_image = Concatenate()([input_image, label_shape])
    #first convolution layer
    conv_1 = Conv2D(64,kernel_size = (5,5), strides = (2,2), padding = "same") (merge_image)
    leakyrelu_layer1 = LeakyReLU(alpha = 0.2) (conv_1)
    #second convolution layer
    conv_2 = Conv2D(128, kernel_size = (5,5),strides = (2,2), padding = "same" ) (leakyrelu_layer1)
    leakyrelu_layer2 = LeakyReLU(alpha = 0.2)(conv_2)
    #Flattening output from above inorder to pass to dense layer
    flatten = Flatten()(leakyrelu_layer2)
    #dense layer
    dense_layer1 = Dense(128)(flatten)
    leakyrelu_layer3 = LeakyReLU(alpha = 0.2)(dense_layer1)
    #Dropping out few neurons inorder to reduce overfitting
    drop_1 = Dropout(0.5)(leakyrelu_layer3)
    #dense layer
    dense_layer2 = Dense(1, activation = "sigmoid")(drop_1)
    #Model initialization
    mnist_dis_model = Model([input_image,label_input],dense_layer2)
    adam_opt = adam_v2.Adam(learning_rate=0.0002,beta_1 = 0.5) #setting optimizer
    #Compiling
    mnist_dis_model.compile(loss = "binary_crossentropy", optimizer= adam_opt, metrics = ['accuracy'])
    return mnist_dis_model
```

Fig.5.2. Discriminator model for fashion Mnist

The main purpose of discriminator is to classify whether the images belong to one generated by our generator or from original dataset. The label embedding is same as we discussed in generator. This label embedding is added as extra channel to image of size (28,28,1) and resulting in shape (28,28,2). The output from above layer is passed to convolution layer that consists of 64 kernels with size of (5,5) and leaky relu activation is applied to it. Then, this output is fed into another convolution layer with same parameters as above layer expect now128 kernels. Again, leaky relu activation is applied to have non-linearity and output is flattened before passing it to dense layer of 128 neurons. Fifty percent of those neurons are dropped from network in order to stabilize and then fed to dense layer. Last year got sigmoid activation function as we are dealing binary type of classification. The model is then compiled for minimizing binary cross entropy loss using Adam optimizer. This is overall summary about our discriminator model architecture as seen in above figure.

In CGAN model, we first make our discriminator untrainable so that generator can be trained along with it. We then built model with generator as first set of layers and discriminator being second set of layers. Then, final model is compiled using similar parameters as discriminator.

```
for epoch in range(epochs):
    for batch in range(batches):
        # generating fake images
        noise_shape = np.random.randn(batch_size  * shapeofnoise).reshape(batch_size,shapeofnoise)
        fake_label = np.random.randint(0,num_labels,batch_size)
        fake_images = generator.predict([noise_shape,fake_label])
        fake_y = np.zeros((batch_size,1))

        # getting sample from real dataset
        indices = np.random.randint(0,xtrain.shape[0],batch_size)
        real_images = xtrain[indices]
        true_label =  ytrain[indices]
        real_y = np.ones((batch_size,1))

        #Training discriminator to distinguish fake from real
        fake_loss, facc = discriminator.train_on_batch([fake_images,fake_label],fake_y)
        real_loss, racc = discriminator.train_on_batch([real_images,true_label],real_y)

        # Training generator to update
        gan_images = np.random.randn(n_samples * shapeofnoise).reshape(n_samples,shapeofnoise)
        gan_labels = np.random.randint(0,num_labels,n_samples)
        gan_y = np.ones((n_samples,1))
        cgan_loss,gan_acc = cgan.train_on_batch([gan_images,gan_labels],gan_y)
```

Fig.5.3. Training CGAN model

Training of conditional generative adversarial networks are completely different from that of traditional deep learning models. To train our GAN model, we used only 50 percent of original fashion Mnist dataset so as to mimic the case of small datasets. We ran our experiment for 50 epochs with 64 images per each epoch. So, the training has been around 23,000 iterations. In first set of steps from line 4 to 7 of figure5.3, we generate 64 random noise vectors of size (100) and generate random fake labels. Then, we pass them to our generator to generate fake images. We assign these fake images value of zero denoting it is generated by generator. In second set of steps from line 8 to 12, we randomly select 64 samples from our original dataset and assign value of 1 to variable real_y denoting it from real set. In third set of steps from line 13 to 15, first we train our discriminator by passing fake images and classify whether they are fake or not. Then we pass real images to discriminator for training and classifying whether are real or not. We can pass both by stacking and train our discriminator, but training separately performs well. In our final step, we will train our CGAN model by generate some random noise, labels and extra variable gan_y for discriminator to judge whether it is real or fake by assigning value of 1. The main reason behind this assignment of 1 is to fool the model by saying images generated by generator are real images. The accuracy and loss of either discriminator or cgan does not play huge role in analysing performance of models, but plotting samples generated at frequent intervals will help to analyse quality of images.

Custom Models for three datasets

We have used different models for every dataset as the business problem is different. The description of each model was listed clearly below with snippets of code that we used in this project.

Fashion Mnist

```python
def classificationModelMNIST():
    """
    Model for classifying fashion mnist dataset
    """

    mnist_model = Sequential()

    # first convolutional layer
    mnist_model.add(Conv2D(32,(3,3),activation = "relu",input_shape = (28,28,1)))
    mnist_model.add(MaxPooling2D(2,2))
    mnist_model.add(Flatten())

    #Dense layer
    mnist_model.add(Dense(100,activation = "relu"))
    mnist_model.add(Dropout(0.5)) #dropping few neurons so as to keep it away from overfitting

    #final layer of our model, there are 10 class labels in our fashionmnist dataset
    mnist_model.add(Dense(10, activation = "softmax"))

    #Compiling our model
    mnist_model.compile(optimizer = adam_v2.Adam(0.0002),loss = "categorical_crossentropy", metrics = ['accuracy'])

    return mnist_model
```

Fig.5.4. Custom model for fashion Mnist dataset

The model for classifying things of fashion dataset is very simple with combination of both convolution and dense layers. As every step of model follows one after another so we define it as sequential model. First layer of our model is convolution with 32 kernels of (3,3) dimension and relu activation is performed over the weighted sum to understand complexity structures. Then, we apply max pooling with kernel size of (2,2) and then result is flattened. The output is passed to first dense layer with 100 nodes and relu activation followed by dropping fifty percent of neurons for normalizing. As our samples belongs to 10 different types of classes. Our final dense layer consists of 10 neurons with SoftMax activation.

Weather Prediction

```python
def weatherClassificationModel():

    weathermodel = Sequential()

    #set of First layer
    weathermodel.add(Conv2D(32,(3,3),activation = "relu", input_shape = (224,224,3)))
    weathermodel.add(MaxPooling2D(3,3)) # performing max pooling operation on the output from above convlution
    weathermodel.add(BatchNormalization()) # performing batch normalization

    #set of Second layer
    weathermodel.add(Conv2D(16,(3,3),activation = "relu"))
    weathermodel.add(MaxPooling2D(3,3))
    weathermodel.add(BatchNormalization())

    weathermodel.add(Flatten()) # flattening output from above into single vector
    weathermodel.add(Dropout(0.5)) #dropping 50% neurons

    #first dense layer
    weathermodel.add(Dense(64,activation = "relu"))

    # Final layer, as our dataset consists of four class labels
    weathermodel.add(Dense(4,activation = "softmax"))

    #optimizer
    adam = adam_v2.Adam(0.002)
    #Compiling model
    weathermodel.compile(optimizer = adam,loss = "categorical_crossentropy", metrics = ['accuracy'])

    return weathermodel
```

Fig.5.5. Custom model for weather prediction dataset

The dimensions of weather dataset vary from image to image, so we did pre-process and converted them into (224,224,3) dimensional tensor before passing to model for training. The following set of layers: convolution layer with (3,3) kernel and relu activation, max pooling and batch normalization is performed twice with 32 convolutional kernels in first layer and 16 convolutional kernels in second layer. Fifty percent of neurons are dropped after the output from above layer are flattened. Then, it passed to dense layer with 64 neurons and relu activation. Final dense layer has neurons similar to number of class labels i.e., 4.

CGIAR

```
def cgiarModel():

    cgiar_model =  Sequential()

    #First set of layers
    cgiar_model.add(Conv2D(128,(3,3),activation = "relu",input_shape=(224,224,3)))
    cgiar_model.add(MaxPooling2D((3,3)))

    #Second set of layers
    cgiar_model.add(Conv2D(128,(3,3),activation = "relu"))
    cgiar_model.add(Conv2D(64,(3,3),activation = "relu"))
    cgiar_model.add(MaxPooling2D((3,3)))

    #Third set layers
    cgiar_model.add(Conv2D(64,(3,3), activation = "relu"))
    cgiar_model.add((MaxPooling2D((3,3))))
    cgiar_model.add(Flatten())

    #Final set layers
    cgiar_model.add(Dropout(0.5))
    cgiar_model.add(Dense(32, activation = "relu"))
    cgiar_model.add(Dense(3, activation = "softmax"))

    #compiling model
    optimizer = adam_v2.Adam(0.0002)
    cgiar_model.compile(loss = "categorical_crossentropy",optimizer = optimizer,metrics=['accuracy'])

    return cgiar_model

```

Unlike other complex models, we didn`t use batch normalization layers for predicting wheat rust dataset because using of it degrading performance of model. There is one max pooling layer with kernel size of (3,3) in between two convolution 2d layers with 128 kernels of (3,3) size and same relu activation units. Similarly, next set of layers are same as above but only difference is 64 kernels instead of 128. Then, a pooling layer followed by flattening layer. The output from above layer is dropped by fifty percent to reduce any overfitting. Then, output from previous layer is passed to first dense layer with 32 neurons followed by final dense layer of 3 neurons equal to number of class labels.

Overall, in all these models we used only relu function as activation in hidden layers as it most preferred one and SoftMax for final layer as activation function. Every model is compiled with same parameters. As it is multi-class classification with more than 2 class labels for each dataset, our objective is to reduce categorical cross entropy loss. Adam optimizer with learning rate of 0.0002 is chosen.

Pretrained Model

We extended vgg-16 model with weights of ImageNet to solve both weather prediction and CGIAR datasets. Our extended model consists of vgg-16 layers as first part without directly adapting dense layers. Then we flattened results from above layers and passed to dense layer with desired neurons equal to respective class label count.

6. Results

All our experiments are trained for 20 epochs to benchmark our results. The number of epochs depends on various factors. Mainly in training process, we must concentrate on both training and errors to determine exact number of iterations needed. But to address our objectives, we restricted for only 20 epochs assuming that our datasets are small and might overfitting easily. Model checkpoints are created to observe best accuracy and store their weights into local system. In all this experiments we just added 50percent of augmentation data additionally to actual amount used for train.

How does the size of dataset impacts model performance?

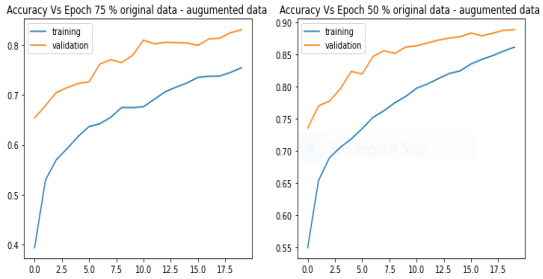


Fig.6.1. With Augmentation

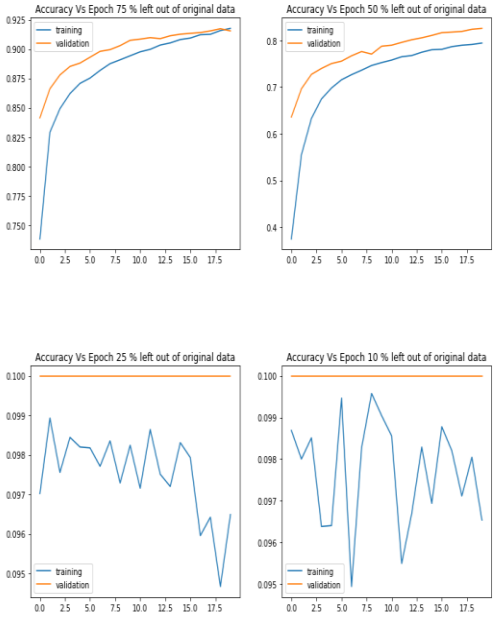


Fig.6.2. Without Augmentation

To know the role of how the size of data would impact performance of our model, we considered large dataset that is fashion Mnist with 60,000 images and took samples with 75,50,25,10 percent of original data. Then, we trained both with augmented dataset and without augmentation using model (Figure 5.4) which we discussed in the above section. The training and validation scores are plotted for both as shown in figures 6.1 and 6.2. Except for 75percent of original data, all other sampled datasets with augmented data resulted in good scores comparing to models directly training on original sampled dataset. Both 25%, 10% without any augmentation datasets accuracy on validation was 10 percent and does not show improvement on training but with augmentation the scores reached up to 90percent. This signifies that increasing size of dataset by augmentation does help in training our model when number of samples are less. We will further investigate about in our next part about

before statement. The performance of model is almost similar when we used 75percent samples, the reasons of this behaviour has not been investigated in our project.

Does model have any advantage when training with augmented data compare to original data?

Dataset	**Training**		**Testing**	
	Augmented	**Un-Augmented**	**Augmented**	**Un-Augmented**
Fashion Mnist	88.81	82.57	88.16	64.48
Weather	88.67	85.71	92.92	92.03
CGIAR	73.09	71.06	63.63	57.92

Table 6.1. Accuracies of three datasets with and without augmentation

We used 50percent of original dataset of fashion Mnist for this analysis and around 20000 images are reserved for testing as per keras data library. For weather prediction and CGIAR dataset, about 10 percent of data is reserved for model accuracy and rest of the data is used for training and validation splits. From table 6.1, we can observe that scores during training have no huge variation between augmented and un-augmented data but there is huge marginal difference during testing. Training and testing accuracies of fashion Mnist dataset was around 88percent for dataset combination of both real and augmented data but there is about 20percent variation for original dataset. Weather dataset performed well on both with much variation. CGIAR being extremely small dataset with around 800 images in total was able to classify with score around 73% during training but showed huge difference on test set with augmented data scored 63% and trained on original set scored 57%. This clearly signifies that augmentation really helps if our dataset is small. Type of techniques to be applied varies depending on type of business problem we addressing as not every technique is suitable and sometimes it may lead to drastic changes in score.

Do custom models outperform pretrained models?

Dataset	**Pretrained**		**Custom**	
	Augmented	**Un-Augmented**	**Augmented**	**Un-Augmented**
Weather	93.80	93.80	92.92	92.03
CGIAR	56.81	47.72	63.63	57.92

Table 6.2. Scores of pretrained and custom model on weather and CGIAR datasets

We have performed experiments on custom model and pretrained model as discussed in previous chapter 5. Number of iterations per training is same and discussed at starting of this chapter. The end results on testing set were listed in table 6.2. We can observe that weather dataset performed well regardless of whether it is by using transfer learning or by custom built model. Both scores are above 90percent with pretrained model performing better than custom but it is negligible as variation is not too high to differentiate. For CGIAR, the variation in scores is quite notable with custom model performing better than VGG. Accuracy of augmented data with pretrained is around 56% and with custom model it is 7% more. The

score by using just original data with custom model reached 57 percent which is better than performances of pretrained model both with augmentation and without augmentation. To sum up, we cannot completely say that either custom model or pretrained model outperforms other. Custom models have advantage of chance to change the complexity according to needs that if our model is overfitting then we can remove few layers to reduce it or if our model is underfitting then we can add additional layers. When using transfer learning, we will have advantage of using weights that was trained on larger datasets, but we need to have clear understanding of business problem and check whether data resembles the pre-trained model.

Do images from GAN augmentation have any impact?

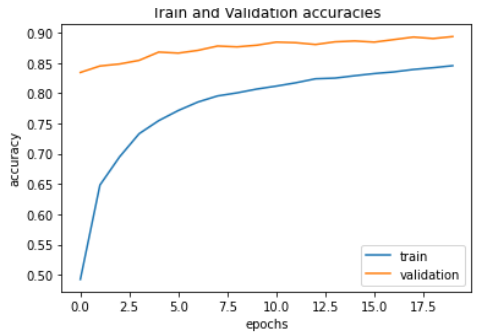


Fig. 6.3. With GAN augmentation

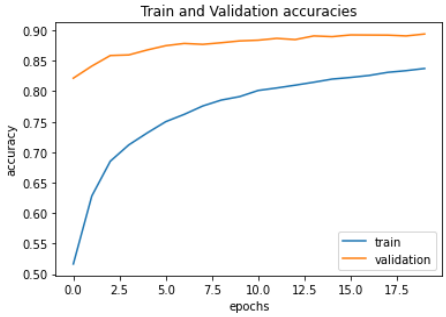


Fig. 6.4. Combination of GAN and traditional augmentation

We utilized only 50 percent of samples from original fashion Mnist data to mimic the case of small dataset. Even same amount was used for training GANs as discussed in chapter 5. When using both techniques, we used 25percent of data from traditional type of augmentation and other 25 percent from GAN so as to maintain equal amounts. From figures 6.3 and 6.4, we can observe that in both training process scores reached as high as 90 percent. While using images generated by generative adversarial networks resulted in rise of above 1% percent by traditional augmentation type during training phase. Testing score was around 88 percent which is similar to traditional augmentation. Finally, we can say that for Fashion Mnist dataset there is no huge difference between type of augmentation that is either general transformations or GANs have same model performance. But when using high dimensional data is high possibility of positive effect due to its adaption of feature space which is not further investigated in our project due to computational constraints we have at our end.

7.Conclusion and future work

Conclusion

The main goal of this project is to address data scarcity in small datasets. In this project, we explored possible ways to handle this problem with traditional augmentation and general adversarial networks. We implemented classical techniques by using ImageDataGenerator class from python's Keras library. These techniques include rotation, shear transformation, colour transformations and noise additions such as gaussian noise, speckle, and poison noise. With the Fashion Mnist dataset, we successfully demonstrated the use case of GAN for data augmentation. All models for three datasets (like fashion Mnist, weather prediction and CGIAR) performed better using augmented data than with original data. From the results, we discussed concluded that data augmentation will help datasets with small amounts of data to improve their model performance. We also noticed the drastic change in model accuracy when we considered sampled data from a large dataset. With small samples, the model trained on transformed data has comparatively better performance than the model trained with only sampled data, but this difference got reduced with an increase in sample size to a level that there is no difference by using either data augmentation or original data. As for this project, images generated by GAN do not have differences from the traditional augmentation type. Performance was almost similar either by combining both techniques or using them separately. We cannot conclude that the custom model performs better than the pre-trained model but can say custom models have an advantage over them for modifying the architecture according to the needs of business problems. Overall, data augmentation is necessary when we have small datasets for better model performance at training time and on unseen data.

Future Work

This project explored a few interesting ideas, such as investigating and implementing techniques in data augmentation and answering performance analysis research questions. There are other research questions that we have to address as we cannot explore them due to time constraints. Given the time, I will address a few of them. The following is the list of ideas.

- While working with GANs, I realised they are a type of deep learning network. So, there is one possible question does the amount of data affects the quality of generated images? If it affects the quality of generated images, can we use them to address the data scarcity problem?
- Does the amount of augmented data affect the performance of model training and testing? Suppose our dataset consists of 1500 samples with 10+ class labels. Does generating images by augmentation of 50 per cent, double or more amount to the original dataset, help the model? or will the model perform poorly on production data?
- Outliers play a primary role in determining the quality of data. Figuring out ways to deal with them if our dataset is small will improve performance.
- This project only addressed data scarcity in computer vision, is there any methods to treat problem in datasets with textual data?

8.References

1. Yann LeCun, Yoshua Bengio & Geoffrey Hinton, 2015. “Deep Learning” Available at: https://www.researchgate.net/publication/277411157_Deep_Learning
2. Chen sun, Abhinav Shrivastava, Saurabh Singh & Abhinav Gupta “Revisiting unreasonable effectiveness of data in deep learning Era”. Available at: https://openaccess.thecvf.com/content_ICCV_2017/papers/Sun_Revisiting_Unreasonable_Effectiveness_ICCV_2017_pape
3. Roger L. Easton Jr, 2020. “Fundamentals of Digital Image processing “, (ch:04, page: 43) Image Processing Operations.
4. Ajay Kumar boyat, Brijendra Kumar joshi,2015. “Review Paper: Noise Models in digital image processing” Available at: https://www.aircconline.com/sipij/V6N2/6215sipij06.pdf
5. Kuntal Kumar Pal & Sudeep K.S, 2019. “Pre-processing for image classification by convolutional neural networks” Available at: https://ieeexplore-ieee-org.ezproxy4.lib.le.ac.uk/stamp/stamp.jsp?tp=&arnumber=7808140
6. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5977656/
7. Saleh Shahinfar, Paul Meek & Greg Falzon, 2020. “How many images do I need?” Understanding how sample size per class effects deep learning model performance metrics for balanced designs in autonomous wildlife monitoring Available at: https://www.sciencedirect.com/science/article/abs/pii/S1574954120300352
8. Hussain Z, Gimenez F, Yi D, Rubin D. Differential Data Augmentation Techniques for Medical Imaging Classification Tasks. AMIA Annu Symp Proc. 2018 Apr 16; 2017:979-984. PMID: 29854165; PMCID: PMC5977656.
9. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, 2014. “Generative Adversarial Networks”. https://arxiv.org/abs/1406.2661
10. Mehdi Mirza, Simon Osindero, 2014. “Conditional Generative Adversarial Networks”. https://arxiv.org/abs/1411.1784
11. Shorten, C., Khoshgoftaar, T.M. A survey on Image Data Augmentation for Deep Learning. J Big Data 6, 60 (2019). https://doi.org/10.1186/s40537-019-0197-0
12. Ren W, Shengen Y, Yi S, Qingqing D, Gang S. Deep image: scaling up image recognition. CoRR, abs/1501.02876, 2015.
13. https://zindi.africa/competitions/iclr-workshop-challenge-1-cgiar-computer-vision-for-crop-disease
14. https://data.mendeley.com/datasets/4drtyfjtfy/1
15. Diederik P. kingma, Jimmy Ba, 2014. Adam: A method for stochastic Optimization. Available at: https://arxiv.org/pdf/1412.6980.pdf
16. Nitish Srivastava, Geoffrey Hinton, Alex krizhevsty, Ilya Sutskever, Ruslan Salakhutdinov,2014. Dropout: A simple way to prevent neural networks from overfitting
17. Thermodynamics-based Artificial Neural Networks for constitutive modeling - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Some-of-the-most-common-activation-

functions-and-their-first-order-gradient-From-left-to_fig4_346898697 [accessed 31 Aug 2022]
18. Great Learning Team, https://www.mygreatlearning.com/blog/introduction-to-vgg16/