A Comparative Study on Efficiencies of Variants of Convolutional Neural Networks based on Image Classification Task.

Ayush Sharma

Abstract— Deep neural networks have shown their high performance on image classification tasks but meanwhile more training difficulties. Due to its complexity and vanishing gradient, it usually takes a long time and a lot of computational resources to train deeper neural networks. Deep Residual networks (ResNets), however, can make the training process easier and faster. And at the same time, it achieves better accuracy compared to their equivalent neural networks. Deep Residual Networks have been proven to be a very successful model on image classification. Deep neural networks demonstrate to have a high performance on image classification tasks while being more difficult to train. We built two very different networks from scratch based on the idea of Densely Connected Convolution Networks. The architecture of the networks is designed based on the image resolution of this specific dataset and by calculating the Receptive Field of the convolution layers. We also used some non-conventional techniques related to image augmentation and Early stopping to improve the accuracy of our models. The networks are trained under high constraints and low computation resources.

Index Terms— Data augmentation, Simple ConvNet, VGG16, Densenet-34, Densenet-50.

1 Introduction

Image classification is a fundamental problem in computer vision and machine learning. It has been attracting a lot of researches on it. In recent years, there are many successful-breakthroughs in the field of image classification.

Deep neural networks are the basis of state-of-the-art results for image recognition, object detection, face recognition,speech recognition, machine translation, image caption generation, and driverless car technology.

The Convolutional Neural Networks (CNNs), in domains like computer vision, mostly reduced the need for handcrafted features due to its ability to learn the problem specific features from the raw input data. However, the selection of dataset specific CNN architecture, which mostly performed by either experience or expertise is a time consuming and error prone process. The CNN architectures, and recently datasets are categorized as deep, shallow, wide, etc. A deep neural network is typically updated by stochastic gradient descent and the parameters θ (weights) are updated ∂l by $\theta^{t} = \theta^{t-1} - \epsilon * \partial\theta$. where L is a loss function and ϵ is the learning rate. It is well known that too small a learning rate will make a training algorithm converge slowly while too large a learning rate will make the training algorithm diverge. Despite the success of deep learning models, our theoretical understanding about neural networks remains limited. Careful selection of network width (number of neurons in FC layers, number of filters in convolution layers) and network depth (number of trainable layers) plays a vital role in designing deep neural networks in order to obtain better performance. Deep neural networks usually provide better results in the field of machine learning and computer vision compared to the handcrafted feature descriptors.

In this paper, we present our approach for building classification models on the given data that has 90,000 training samples belonging to 200 classes, each class having 450 training samples. Each training sample is a 64*64*3 (RGB) image. The validation set consists of 10,000 samples and the testing set consist of another 10,000 samples. It is to design the best model that categorize the 10,000 testing images. We did not use any pre-trained network available in standard libraries. Our biggest challenge was low computation power provided by Google Colab free services. Google Colab provides only 12 hours of continuous computation time, after which the session needs to be reconnected. Also, we restricted our models from using any dense or fully connected layers, any dropout layers. The rest of the report is organized as follows : Analysis of Previous Works in section II, The Convolution Operation in section III, Proposed Methodology in section IV, Results in section V, Error Analysis and Deduction in section VI and Conclusion, Outlook and Future Work with Scope of Improvements in section VII.

2 Analysis of previous work

Deep convolutional neural networks have enabled the field of image recognition to advance in an unprecedented pace over the past decade. A. Krizhevsky, I. Sutskever, and G. E. Hinton in their Imagenet classification with deep convolutional neural networks introduced AlexNet in 2012 which has 60 million parameters and 650,000 neurons. The model consists of five convolutional layers, and some of them are followed by max-pooling layers. To fight overfitting, their work proposes data augmentation methods and includes the technique of dropout in the model. The model achieves a top-5 error rate of 18.9% in the ILSVRC-2010 contest. A technique called local response normalization is employed to help generalization, which is similar to the idea of batch normalization given by S. Ioffe and C. Szegedy in their work of Batch normalization: Accelerating deep network training by reducing internal covariate shift. Around 2011, a good ILSVRC classification error

rate was 25%.
Inception module introduced by Szegedy use variable filtersizes to capture different visual patterns of different sizes, and approximate the optimal sparse structure by the inception module. Specifically, inception module consists of one poolingoperation and three types of convolution operations With thehelp of inception module, the network parameters can be dramatically reduced to 5 millions which are much less than those of AlexNet (60 millions) and ZFNet (75 millions).AlexNet achieved 16% error rate on the ImageNet challenge. In the next couple of years, with more and more layers in neural networks, VGG19 with 19 layers and GoogleNet with 22 layers reduced the error rates to a few percent. Although CNNs have made some breakthrough on the accuracy, they are hard to train for two reasons:

- First, the so called vanishing gradient problem the effect of multiplying n of those small numbers from activation function to compute gradients in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n, thus the front layers train very slowly.
- Second, CNNs usually have even more parameters in their models, introducing more complexity, so takes longer to train than just a dense network.

There are several Image Classification competitions such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where several convolution neural network architectures and models are presented. In 2015, the ImageNet challenge was won by the ResNet model with 152 layers which achieved a top-5 classification error of 3.57% And finally in 2015 comes the ResNet. The main difference in ResNets is that they have shortcut connections parallel to their normal convolutional layers. These shortcut are always alive and gradients can easily propagate through them, resulting in faster training. ResNet with 152 layers achieves the best results of 3% error rate, which is even better than human judges.

3 The Convolution Operation

Convolution is an operation on two functions of a real-valued argument. Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x(t), the position of the spaceship at time t. Both x and t are real valued, that is, we can get a direct reading from the laser sensor at any instant in time. Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceships position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w(a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da.$$

This operation is called convolution. The convolution operation is typically denoted with an asterisk : s(t) = (x ∗ w)(t). Usually, when we work with data on a computer,time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are depended only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. We often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I*K)(i,j) = \sum_m \sum_n I(m,n)K(i-m,j-n) \quad (1)$$

Convolution leverages three important ideas that can help improve a machine learning system:
1) Sparse interactions
2) Parameter sharing
3) Equivariant representations

3.1 CNN vs Other Networks

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions(also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires (mxn) parameters, and the algorithms used in practice have O(mn) runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only (kn) parameters an O(kn) runtime.

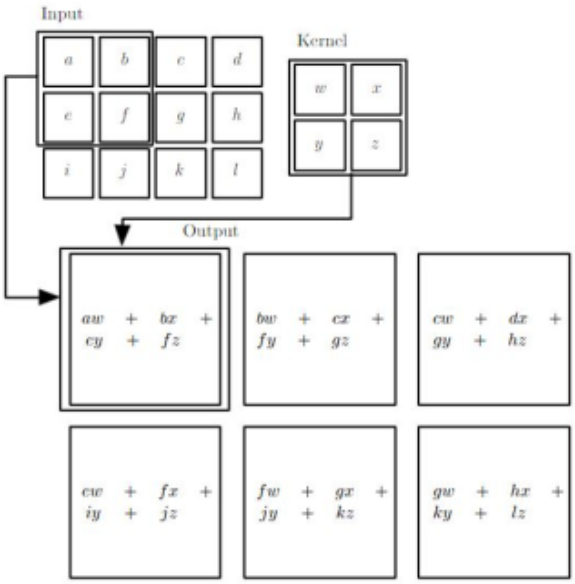


Fig. 1. An example of 2-D convolution

Parameter sharing refers to using same parameters for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation it is still O(k n) but it does further reduce the storage requirements of the model to k parameters.

3.2 Receptive Field(RF)

It is a local region (including its depth) on the output volume of the previous layer that a neuron is connected to. This term has been prevalent in Neurosciences since the study of Hubel and Wiesel in which they suggested local features are detected in early visual layers of the visual cortex and are then progressively combined to create more complex patterns in a hierarchical manner. As an example, assume that the input RGB image to a CNN has size [32 x 32 x 3]. For a filter size of 5 x 5, then each neuron in the first convolutional layer will be connected to a [5 x 5 x 3] region in the input volume. Thus, a total of 5 x 5 x 3 = 75 weights (+1 bias parameter) needs to be learned. Notice that RF is a 3D tensor with its depth being equal to the depth of the volume in the previous layer. Here, for simplicity, we discard the depth in our calculation.

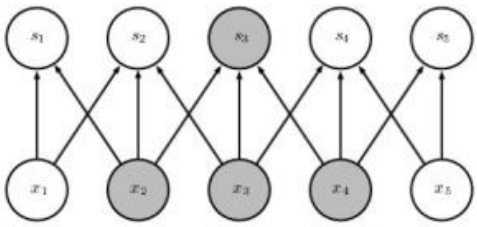


Fig. 2. Sparse connectivity, viewed from below. We highlight one input unit,x3, and highlight the output units in s that are aected by this unit. It shows when s is formed by convolution with a kernel of width 3, only three outputs are aected by x. The output unit s3, and highlighted input units x that aect s3 are known as the receptive field of s3.

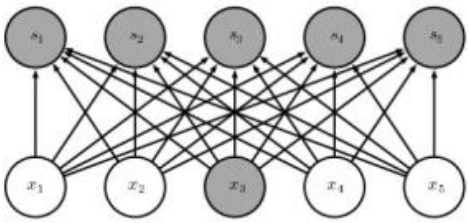


Fig. 3. When s is formed by matrix multiplication, connectivity is no longer sparse, so all the outputsare aected by x3

3.3 Effective Receptive Field(ERF)

It is the area of the original image that can possibly influence the activation of a neuron. One important point to notice here is that RF and ERF are the same for the first convolutional layer. However, they differ as we moves along the CNN hierarchy. The RF is simply equal to filter size over the previous layer but ERF traces the hierarchy back to the input image and indicates the extent of the input image which can modulate the activity of a neuron.

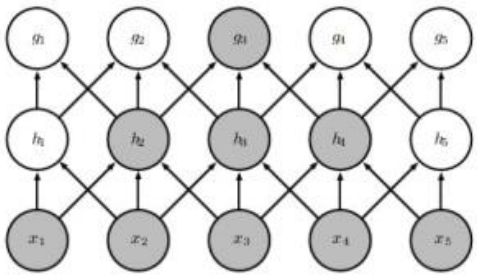


Fig. 4. The receptive eld of the units in the deeper layers of a convolutional networkis larger than the receptive eld of the units in the shallow layers.

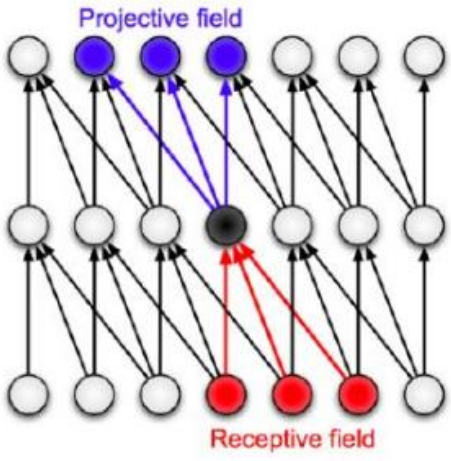


Fig. 5. Schematic plot demonstrating receptive and projective fields of a neuron

3.4 Projective Field(PF)

It is the set of neurons to which a neuron projects its output.

4. Proposed Methodology

4.1 Analysis of the dataset

The given dataset consist of 200 different classes with 90,000 training samples and 10,000 validation samples. The resolution of the images is just 64x64 pixels in RGB colorspace, which makes it more challenging to extract information from it. A glance at the images shows that it is difficult for the human eye to detect objects in some images. After each epoch the validation loss and accuracy is evaluated on the trained model.

4.2 Densenet vs Resnets

Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output. In the Dense Convolutional Network (DenseNet), it connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connectionsone between each layer and its subsequent layer it has L(L+1)/2 for each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters. We used the concept of DenseNet to design our model architecture. The main challenge with very deep neural networks is the problem of vanishing gradients. This problem was first overcome by introducing residual networks which uses a shortcut connection to pass input from one block to another. In contrast to ResNet, DenseNet does not aggregate features through summation; instead, they are combined by concatenation. Thus, the information is passed from one layer to all the subsequent layers ensuring better flow of information and gradients throughout he network. Deep residual networks (ResNets) consist of many stacked "Residual Units". Different from Resnet, a layer in densenet receives all the outs of previous layers and concatenate them in the depth dimension. In Resnet, a layer only receives outputs from the previous second or third layer, and the outputs are added together on the same depth, therefore it wont change the depth by adding shortcuts. In other words, in Resnet the output of layer of k is

$$x[k] = f(w * x[k-1] + x[k-2]) \quad (2)$$

, while in Densenet it is

$$x[k] = f(w * H(x[k-1], x[k-2], \cdots, x[1])) \quad (3)$$

where H means stacking over the depth dimension. Besides, Resnet makes learn identity function easy, while Densenet directly adds identity function. Densenet is more efficient on some image classification benchmarks. In Standard ConvNet, input image goes through multiple convolution and obtain high-level features.

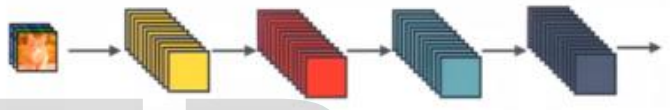


Fig. 6. A standard convolutional neural network

In ResNet, identity mapping is proposed to promote the

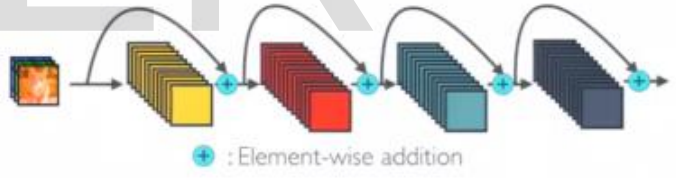


Fig. 7. ResNet concept

gradient propagation. Element-wise addition is used.
In DenseNet, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. Concatenation is used.

4.3 Analysis of Deep Residual Networks

The ResNets developed are modularized architectures thatstack building blocks of the same connecting shape. The original Residual Unit performs the following computations:

$$y_l = h(x_l) + f(x_l, W_l) \quad (4)$$

$$x_{l+1} = f(y_l) \quad (5)$$

Here x_l is the input feature to the l th Residual Unit.

$W_l = \{W_{l,k} \mid 1 \le k \le K\}$ is a set of weights (and biases)

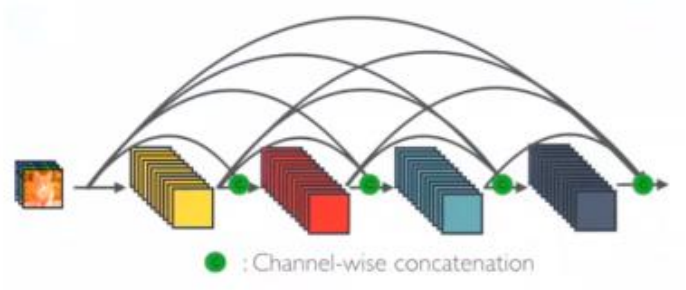


Fig. 8. DenseNet concept

associated with the l th Residual Unit, and K is the number of layers in a Residual Unit. 'f' denotes the residual function, e.g., a stack of two 3x3 convolutional layers. The function f is the operation after element-wise addition, 'f' being a ReLU function. The function h is set as an identity mapping: $h(x_l) = x_l$. If f is also an identity mapping: $x_l+1 \equiv y_l$, we can put Eqn.4 in Eqn.5 and obtain:

$$x_{l+1} = x_l + f(x_l, W_l) \quad (6)$$

Recursively,
$(x_{l+2} = x_{l+1} + f(x_{l+1}, W_{l+1}) = x_l + f(x_l, W_l) + f(x_{l+1}, W_{l+1}))$

we have ,

$$x_L = x_l + \sum_{i=l}^{L-1} f(x_i, W_i)$$

$$\sum_{i=1}^{L-1} f$$

$$x_L = x_0 + \sum_{i=0}^{L-1} f(x_i, W_i),$$

$$\prod_{i=0}^{L-1} W_i x_0$$

for any deeper unit L and any shallower unit l. Eqn.8 exhibits some properties like :

- The feature x_L of any deeper unit L can be represented as the feature x_l of any shallower unit l plus a residual

function in a form of
indicating that the model is in a residual fashion between any units L and l.

- The feature ,

of any deep unit L, is the summation of the outputs of all preceding residual functions (plus x_0). This is in contrast to "a plain network" where feature x_L is a series of matrix

VECTOR PRODUCTS,SAY,

4.4 Approach

The details of a simple ConvNet are as shown in Fig.9 and Fig.10. The input output dimensions are shown in Fig.11. VGG16 was publised in 2014 and is one of the simplest (among the other cnn architectures used in Imagenet competition). It's key characteristics are:

1) This network contains total 16 layers in which weights and bias parameters are learnt

2) A total of 13 convolutional layers are stacked one after the other and 3 dense layers for classification

3) The number of filters in the convolution layers follow an increasing pattern

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 128)       3584
_________________________________________________________________
batch_normalization (BatchNo (None, 64, 64, 128)       512
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 128)       147584
_________________________________________________________________
batch_normalization_1 (Batch (None, 64, 64, 128)       512
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 32, 128)       0
_________________________________________________________________
activation (Activation)      (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 128)       147584
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 128)       147584
_________________________________________________________________
batch_normalization_3 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 128)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 128)       147584
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 128)       147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
activation_2 (Activation)    (None, 16, 16, 128)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
batch_normalization_6 (Batch (None, 8, 8, 256)         1024
```

Fig. 9. Number of input output parameters in each layer of simple convnet

```
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 512)         1180160
_________________________________________________________________
batch_normalization_7 (Batch (None, 8, 8, 512)         2048
_________________________________________________________________
activation_3 (Activation)    (None, 8, 8, 512)         0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 4096)              33558528
_________________________________________________________________
batch_normalization_8 (Batch (None, 4096)              16384
_________________________________________________________________
activation_4 (Activation)    (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              4195328
_________________________________________________________________
batch_normalization_9 (Batch (None, 1024)              4096
_________________________________________________________________
activation_5 (Activation)    (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 200)               205000
_________________________________________________________________
activation_6 (Activation)    (None, 200)               0
=================================================================
Total params: 40,202,312
Trainable params: 40,189,000
Non-trainable params: 13,312
```

Fig. 10. Number of input output parameters in each layer(contd.)

4) The informative features are obtained by max pooling layers applied at different steps in the architecture.

5) The dense layers comprises of 4096, 4096, and number of classes nodes(200).

6) The cons of this architecture are that it is slow to train and produces the model with very large size. VGG16 model is shown in Fig.12. The model details of VGG16 is shown in Fig.13 with layer connection shown in Fig.14.

For both of our residual networks, we implemented the number of layers in our model by first calculating the receptive field shown in Fig.3. Throughout our model, we used (3x3) kernels with strides (1,1). So, for the first layer, the receptive field is (3x3) as each kernel convolutes over (3x3) pixels or 9 pixels at a time. Every such convolution operation decreases the spatial dimensions of the matrix by 2, thus increasing the receptive field of the network by 2. While at the MaxPooling layer, the spatial dimensions of the matrix are reduced by half, hence doubling the entire receptive field. The receptive field of the networks is shown in Fig.16: As we reach the receptive field of original image size 64x64, we expect our model to detect the object in each image distinctly and be able to classify them. However, we have gone further, to a receptive field of more than double the original image size, so that network learns the background details too. Many images in our dataset are confusing due to background domination, so for those images, we want our models to understand the context in which our objects are found. For example, in the class bullfrog we can observe that most of the images have a green background in addition to our object, i.e. bullfrog shown in Fig.17. We want our network to learn that in addition to our object.

4.5 Model Design

In the beginning, without much effort, we implemented the vanilla DenseNet-34 and DenseNet-50 models as our Network 1 and 2 respectively. For DenseNet-34, we achieved an accuracy of on the training set and 0.5576 with a batch size of 128 images running for 75 epochs. For DenseNet-50 we achieved an accuracy of 0.6146 on the training set with a batch size of 60 images running for 75 epochs. Although this shows the learning power of DenseNet models, we notice an early saturation in validation accuracy, due to the use of deep networks on shallow datasets. Network 1 DenseNet-50:

1) We built a custom architecture of 3 bottleneck blocks having 5 convolution layers of increasing channels and 1 MaxPooling layer at the end of each block.

2) Just before applying concatenation, we feed the output of each block to space to depth function. This function ensures that the spatial dimensions of both the layers are equal before concatenation.

3) We concatenate the skip connections from the output of each block with the output of the next block, preserving the information from both the blocks before being fed to the subsequent block.

4) The final layers in the model have 1x1 convolution layer followed by a GlobalAveragePooling layer which averages the spatial dimensions of a matrix of any size. This layer gives us the ability to design a model which can take input image of any size.

For Network 2(DenseNet-34) we modified the DenseNet-18 architecture as follows:

1) We used the first convolution layer that initially

consisted of 64 (7x7) filters with stride (2,2), by 32 (3x3) filters with stride (1,1) and removed the max pooling layer.

2) We removed the 1st block consisting 4 convolution layers of 64 (3x3) filters.

3) We removed the skip connections after every 2 convolution layers instead maintained it between every 4 convolution layers. We replaced the original add function in shortcuts with concatenation so that it preserves the channels from the previous block and not merge them. We added a Batch Normalization and ReLU activation layer after each shortcut.

4) As per requirement of the project, we replaced final FC layers with 1x1 convolution layer for decreasing the number of channels to the required number of classes followed by a GlobalAveragePooling layer.

4.6 Image Augmentation

Deep CNNs are particularly dependent on the availability of large quantities of training data. An elegant solution to alleviate the relative scarcity of the data compared to the number of parameters involved in CNNs is data augmentation. Data augmentation consists in transforming the available data into new data without altering their natures. The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations. We will employ four distinct forms of data augmentation at training time, all of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In order to artificially increase the amount of training data and avoid overfitting, we rely on image augmentation. For Network 1, we fed images with 32x32 resolutions for the beginning few epochs followed by 64x64 resolution images. Then we fed images with 16x16 resolutions for the next few numbers of epochs and finally fed the model with 64x64 resolutions images. We used the following augmentations:

- scale
- coarse dropout
- rotate
- additive gaussian noise
- crop and pad

For VGG16, ConvNet and DenseNet-50 we kept default 64x64 image resolution as input size throughout our complete run. Instead of applying image augmentations after a certain number of iterations as in Network 1, we applied a random sequence of 11 transformations. The intensity of each transformation was randomly determined within a specified range, but these range parameters were manually provided so that these transformed training images could closely represent the images in the validation dataset. The augmentations applied were :

- Horizontal Flip
- Vertical Flip
- Gaussian Blur
- Crop and Pad
- Scale
- Translate
- Rotate
- Shear
- Coarse Dropout
- Multiply

* Contrast Normalisation

A group of eight randomly selected images with augmentation is shown in Fig.28 and Fig.29.

4.7 Regularizers, Optimizers, Hyperparameters and Callbacks

Batch Normalization helps to normalise the inputs of the previous layer at each batch keeping the values in a comparable range with the mean equal to 0 and standard deviation equal to 1. This ensures the activations of our models do not get skewed at any one particular point and also increases the speed of computation. For all the networks we applied Batch Normalization after every convolution layer and then passed these values to the ReLU activation function. We used accuracy as our metric and categorical cross-entropy as our loss function.

1) Optimizers used: **Gradient Descent** is the most basic but most used optimization algorithm. Its used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm. Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model parameters also known as weights are modified depending on the losses so that the loss can be minimized. Update rule is:

$$\theta^{t+1} = \theta^{t} - \eta * \nabla J(\theta^{t}) \quad (9)$$

However, it may trap at local minima. Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take long time to converge to the minima and this require large memory. Using GD causes resource exhaust error. The error due to large batch size(>64) is shown in Fig.30.

Stochastic Gradient Descent goes over the entire data once before updating the parameters.

$$\theta^{t+1} = \theta^{t} - \eta * \nabla J(\theta^{t}; x(i), y(i)) \quad (10)$$

where x(i) ,y(i) are the training examples.

Mini-Batch Gradient Descent where the dataset is divided into various batches and after every batch, the parameters are updated.

$$\theta^{t+1} = \theta^{t} - \eta * \nabla J(\theta^{t}; B(i)) \quad (11)$$

B(i) are the batches of training examples.

Momentum based Gradient Descent accelerates the convergence towards the relevant direction and reduces unnecessary oscillations to the irrelevant direction. In the regions of gentle slope it is

able to take large steps. However, it may run past the global minima. It is an exponential weighted average method.

$$\text{update}^{t} = \gamma * \text{update}^{t-1} + \eta * \nabla\theta(t) \quad (12)$$
$$\theta^{t+1} = \theta^{t} - \text{update}^{t} \quad (13)$$

Nesterov Accelerated Gradient Momentum may be a good-method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is a look ahead method.

$$w_{look\_ahead} = w_{t} - \gamma * \text{update}_{t-1} \quad (14)$$
$$\text{update}_{t} = \gamma * \text{update}_{t-1} + \eta * \nabla w_{look\_ahead} \quad (15)$$
$$w_{t+1} = w_{t} - \text{update}_{t} \quad (16)$$

Algorithm	Number of steps in 1 epoch
GD(Batch)	1
SGD	N, =number of data points
Mini-batch GD	N/B, B= mini batch size

Gradient Descent with Adaptive Learning Rate is generally used when the input feature vectors are largely sparse in nature i.e. its value is 0 for most inputs. Gradients will not be updated and therefore learning will fail. We decay the learning rate for parameters inversely proportional to their update history.

$$v_t = v_{t-1} + (\nabla w_t)^2 \quad (17)$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t \quad (18)$$

For dense features with time η will decrease and for sparse features η will increase.

RMSprop: In Adagrad the decay of the learning rate is more aggressive and as a result after a while the parameters will start receiving very small updates of the decayed learning rate. To avoid this decay the denominator in η and prevent its rapid growth.

$$v_t = \beta * v_{t-1} + (1 - \beta) * (\nabla w_t)^2 \quad (19)$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t \quad (20)$$

Adam Adaptive Moment Estimation works with momentum of first and second order. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also
keeps an exponentially decaying average of past gradients m(t). m(t) and v(t) are values of the first moment which is the Mean and the second moment which is the uncentered variance of the gradients respectively.

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t \quad (21)$$
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * \nabla (w_t)^2 \quad (22)$$
$$p_t = \frac{m_t}{1 - \beta_1(t)} \quad (23)$$
$$q_t = \frac{v_t}{1 - \beta_2(t)} \quad (24)$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{p_t + \epsilon}} * q_t \quad (25)$$

Optimizer	Time/epoch(s)	Memory used
SGD	>10,000	least
Mini batch GD	2800	15.6 GB
MBGD with NAG	2600	16 GB
Adagrad	3200	14.2 GB
RMSprop	2000	15 GB
Adam	1700-1800	20 GB

We used ReduceLROnPlateau as a callback function for reducing the learning rate if validation loss stagnates for 5 epochs. We used L2 kernel regularizer with a lambda value of 2e-4. As Google Colab notebook is restricted to 12 hours of continuous use, for both the model runs we used a Model Checkpointer to save the model with best validation accuracy as we had to run over 100 epochs for each network.

As shown in Fig.31 by monitoring the validation loss we stopped our training near to the minima. By changing min delta i.e. a threshold to whether quantify a loss at some epoch as improvement or not. If the difference of loss is below min delta , it is quantified as no improvement. Another parameter is patience that represents the number of epochs before stopping once the validation loss starts to increase (stops improving). This depends on the implementation, if used very small batches or a large learning rate the loss will be zig-zag (accuracy will be more noisy) so better set a large patience argument. If you use large batches and a small learning rate your loss will be smoother so you can use a smaller patience argument. In our case the patience parameter was set to 20.

Batch Normalization is usually the first step of data preprocessing. Global data normalization transforms all the data to have zero-mean and unit variance. However, as the data flows through a deep network, the distribution of input to internal layers will be changed, which will lose the learning capacity and accuracy of the network. Batch Normalization (BN) introduces a a normalization step that fixes the means and variances of layer inputs where the estimations of mean and variance are computed after each mini-batch rather than the entire training set. Suppose the layer to normalize has a d dimensional input, i.e. x = [x 1 , x 2 , · · · , x d] T . We first normalize the k-th dimension as follows:

$$\hat{x}_k = \frac{(x_k - \mu_B)}{\sqrt{\delta_B^2 + \epsilon}} \quad (26)$$

where μ_B and δ^2_B are the mean and variance of mini-batch respectively, and ϵ is a constant value.

L2-norm regularization modifies the objective function by adding additional terms that penalize the model complexity. Formally, if the loss function is L(θ, x, y), then the regularized loss will be:

$$E(\theta, x, y) = L(\theta, x, y) + \lambda R(\theta) \quad (27)$$

where R(θ) is the regularization term, and λ is the regularization strength.

After every epoch or if the validation loss decreases, the weights of the model are stored as Hierarchical Data Format version 5 as the model is complex and has millions of parameters.

5 Results

For the network simple covnet we trained our model with 40 Million parameters for 108 epochs with a batch size of 60 images shown in Fig.34. The predicted and true classes are shown in Fig.35 and Fig.36. Predicted class labels are shown in Fig.37. Metrics like precision, recall and f1-score is given in Fig.38.

For the network VGG16, we trained our model with 40 Million parameters for 75 epochs with a batch size of 128 images shown in Fig.39, Fig.40 and Fig.41. Note that validation loss did not improve from 2.52.

For the network DenseNet-34, we trained our model with 11.8 Million parameters for 75 epochs with a batch size of 128 images. Training accuracy, loss, validation accuracy and loss is shown in Fig.45 Here, in Fig.45 the validation loss did not improve from 2.10. For the network DenseNet-50, we trained our model on 17.8 Million parameters for 75 epochs with a batch size of 64 images. Training accuracy, loss, validation accuracy and loss is shown in Fig.49 Here, in Fig.45 the validation loss did not improve from 2.94.

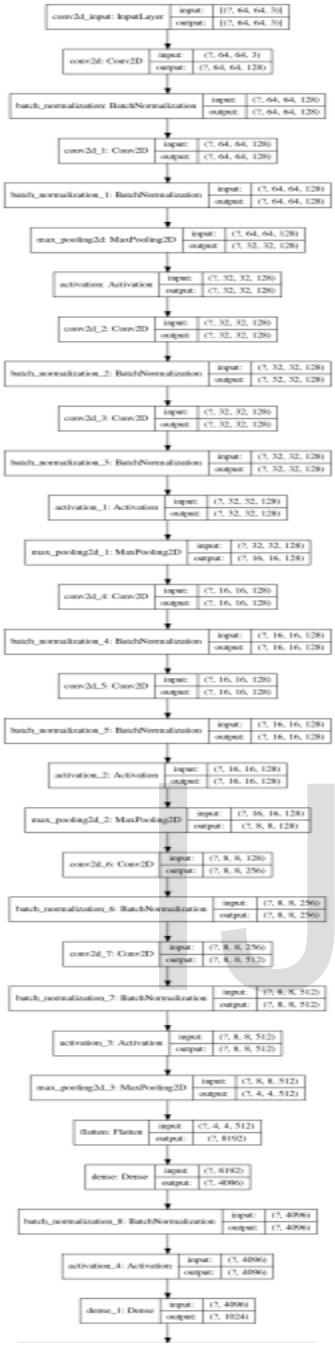
Model	Train loss	Train acc.	Val. loss	Val. acc.
ConvNet	2.8422	0.3514	2.4221	0.4344
VGG16	1.8608	0.6866	4.3329	0.4196
DenseNet-34	2.3584	0.5576	2.4755	0.4729
DenseNet-50	2.0096	0.6146	2.9440	0.5648

6 Error Analysis and Deduction

Some interesting deductions can be made by observing the error pattern. Models were not able to identify between an 'Egyptian cat' and 'Tabby cat'. Same for 'Labrador retriever' with 'Golden retriever', 'Sports car' with 'convertible cars' i.e. it was not able to distinguish between subclasses. These errors were due to low resolution of the images. Another observation that can be made was in the correctly classifed class, there was a clear distinction between the object and the background. The model performs very well on detecting objects that cover the entire 64x64 resolution and has a minimal background. Though for some classes the model was able to detect zoomed in and zoomed out object. This could be because we fed low-resolution images and used the scaling augmentation. In the incorrectly classified images, since there are not many details in the images, given the low resolution of 64x64, the model can detect the type of the object (such as a dog), but unable to further sub-classify it into the correct class (such as Labrador retriever and Golden retriever).

7 Conclusion,Outlook and Future Work with Scope of Improvements

Deep CNNs have made breakthroughs in processing image, video, speech and text. In this paper, we have given an extensive survey on recent advances of CNNs. We have discussed the improvements of CNN on different aspects, namely, layer design, activation function, loss function, regularization, optimization and fast computation. Beyond surveying the advances of each aspect of CNN, we have also introduced the application of CNN on many tasks, including image classification, object detection, object tracking, pose estimation, text detection, visual saliency detection, action recognition, scene labeling, speech and natural language processing. Although CNNs have achieved great success in experimental evaluations, there are still lots of issues that deserve further investigation. Firstly, since the recent CNNs are becoming deeper and deeper, they require large-scale dataset and massive computing power for training. Manually collecting labeled dataset requires huge amounts of human efforts. Thus, it is desired to explore unsupervised learning of CNNs. Meanwhile, to speed up training procedure, although there are already some asynchronous SGD algorithms which have shown promising result by using CPU and GPU clusters, it is still worth to develop effective and scalable parallel training algorithms. At testing time, these deep models are highly memory demanding and time consuming, which makes them not suitable to be deployed on mobile platforms that have limited resources. It is important to investigate how to reduce the complexity and obtain fast-to-execute models without loss of accuracy. Furthermore, one major barrier for applying CNN on a new task is that it requires considerable skill and experience to select suitable hyperparameters such as the learning rate, kernel sizes of convolutional filters, the number of layers etc. These hyper-parameters have internal dependencies which make them particularly expensive fortuning. Recent works have shown that there exists a big room to improve current optimization techniques for learning deep CNN architectures. We wish to confirm our sincere and heartfelt gratitude to all our esteemed mentors without whose guidance this course and project would not have been successful.



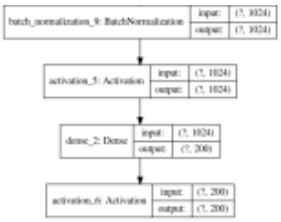


Fig. 11. Layer connection in simple covnet.

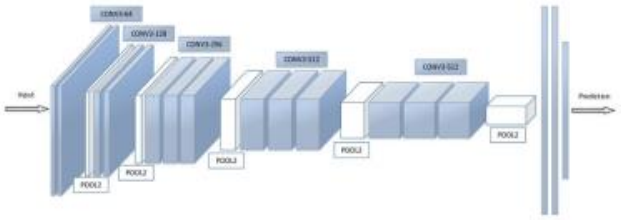


Fig. 12. VGG16 architecture

Model: "model_1"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 3)	0
conv2d_1 (Conv2D)	(None, 64, 64, 64)	1792
conv2d_2 (Conv2D)	(None, 64, 64, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 64)	0
conv2d_3 (Conv2D)	(None, 32, 32, 128)	73856
conv2d_4 (Conv2D)	(None, 32, 32, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 128)	0
conv2d_5 (Conv2D)	(None, 16, 16, 256)	295168
conv2d_6 (Conv2D)	(None, 16, 16, 256)	590080
conv2d_7 (Conv2D)	(None, 16, 16, 256)	590080
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 256)	0
conv2d_8 (Conv2D)	(None, 8, 8, 512)	1180160
conv2d_9 (Conv2D)	(None, 8, 8, 512)	2359808
conv2d_10 (Conv2D)	(None, 8, 8, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 512)	0
conv2d_11 (Conv2D)	(None, 4, 4, 512)	2359808
conv2d_12 (Conv2D)	(None, 4, 4, 512)	2359808
conv2d_13 (Conv2D)	(None, 4, 4, 512)	2359808
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 512)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 4096)	8392704
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 200)	819400

Total params: 40,708,104
Trainable params: 40,708,104
Non-trainable params: 0

Fig. 13. Layer details and parameter count of a VGG16 model

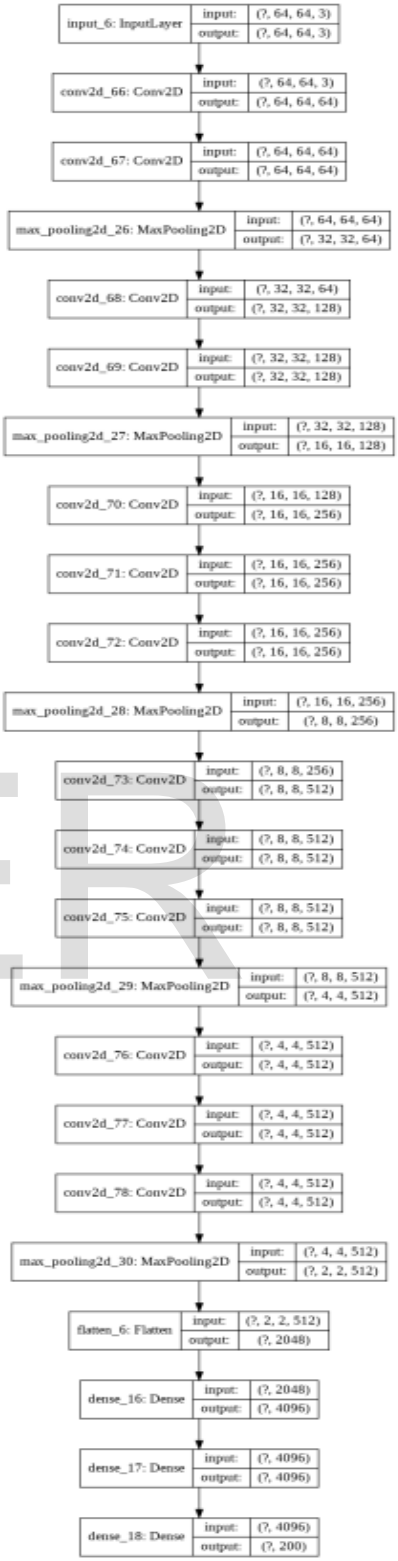


Fig. 14. Inter layer connection and their input-output dimensions count of a VGG16 network

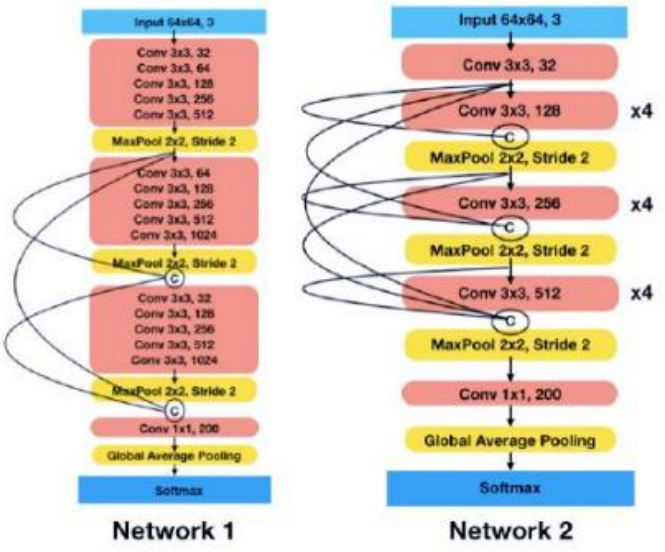


Fig. 15. Pictorial representation of the proposed Dense networks

Network 1

- Convolution Layers 1 - 5 = 11x11 (11 = 3+2+2+2+2)
- MaxPooling Layer 6 = 22x22
- Convolution Layers 7 - 11 = 32x32 (32 = 22+2+2+2+2+2)
- MaxPooling Layer 12 = 64x64
- Concatenation Layer 13 = No Change
- Convolution Layers 14 -18 = 74x74 (74 = 64+2+2+2+2+2)
- MaxPooling Layer 19 = 148x148
- Concatenation Layer 20 = No Change

Network 2

- Convolution Layer 0 = 3x3
- Convolution Layers 1 - 4 = 11x11 (11 = 3+2+2+2+2)
- MaxPooling after Concatenation Layer 5 = 22x22
- Convolution Layers 6 - 9 = 30x30 (30 = 22+2+2+2+2)
- MaxPooling after Concatenation Layer 10 = 60x60
- Convolution Layers 11 - 14 = 68x68 (68 = 60+2+2+2+2)
- MaxPooling after Concatenation Layer 15 = 136x136

Fig. 16. Details of network 1 and 2

Fig. 17. Images of Bullfrog

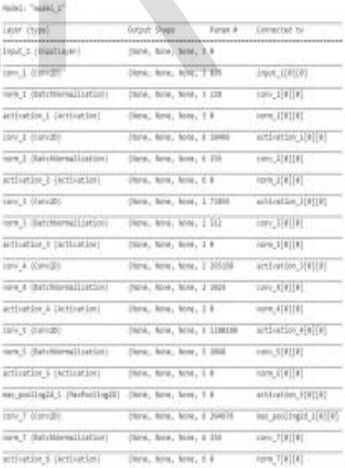


Fig. 18. Layers of Resnet-50

conv_8 (Conv2D)	(None, None, None, 1	73856	activation_6[0][0]
norm_8 (BatchNormalization)	(None, None, None, 1	512	conv_8[0][0]
activation_7 (Activation)	(None, None, None, 1	0	norm_8[0][0]
conv_9 (Conv2D)	(None, None, None, 2	295168	activation_7[0][0]
norm_9 (BatchNormalization)	(None, None, None, 2	1024	conv_9[0][0]
activation_8 (Activation)	(None, None, None, 2	0	norm_9[0][0]
conv_10 (Conv2D)	(None, None, None, 5	1180160	activation_8[0][0]
norm_10 (BatchNormalization)	(None, None, None, 5	2048	conv_10[0][0]
activation_9 (Activation)	(None, None, None, 5	0	norm_10[0][0]
conv_11 (Conv2D)	(None, None, None, 1	4719616	activation_9[0][0]
norm_11 (BatchNormalization)	(None, None, None, 1	4096	conv_11[0][0]
activation_10 (Activation)	(None, None, None, 1	0	norm_11[0][0]
lambda_1 (Lambda)	(None, None, None, 2	0	max_pooling2d_1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, None, None, 1	0	activation_10[0][0]
concatenate_1 (Concatenate)	(None, None, None, 3	0	lambda_1[0][0] max_pooling2d_2[0][0]
conv_14 (Conv2D)	(None, None, None, 3	884768	concatenate_1[0][0]
norm_14 (BatchNormalization)	(None, None, None, 3	128	conv_14[0][0]
activation_11 (Activation)	(None, None, None, 3	0	norm_14[0][0]
conv_15 (Conv2D)	(None, None, None, 1	36992	activation_11[0][0]
norm_15 (BatchNormalization)	(None, None, None, 1	512	conv_15[0][0]
activation_12 (Activation)	(None, None, None, 1	0	norm_15[0][0]
conv_16 (Conv2D)	(None, None, None, 2	295168	activation_12[0][0]
norm_16 (BatchNormalization)	(None, None, None, 2	1024	conv_16[0][0]

Fig. 19. Layers of Resnet-50(contd.)

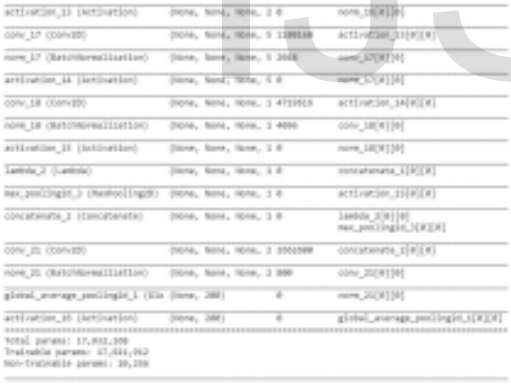


Fig. 20. Layers of Resnet-50(contd.)

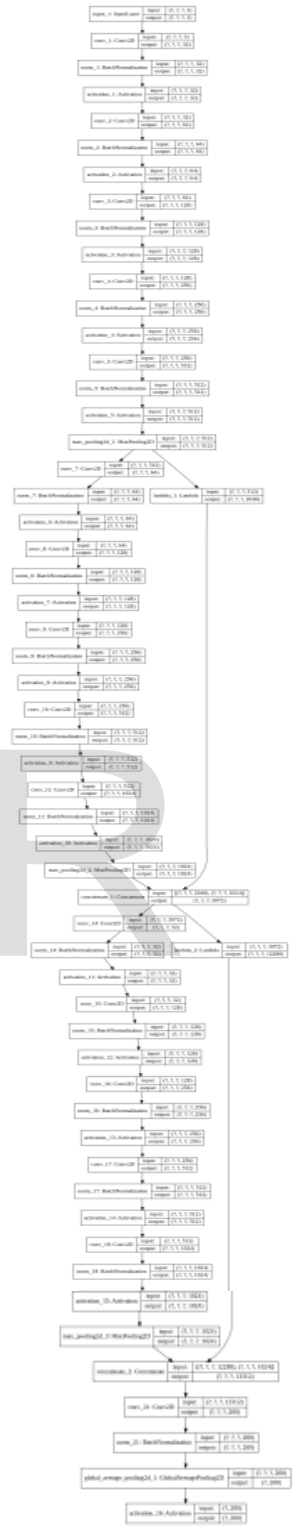


Fig. 21. Inter layer connection with i/p, o/p dimensions for DenseNet-50 model

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 64, 64, 3)	0	
conv2d_1 (Conv2D)	(None, 64, 64, 32)	896	input_1[0][0]
batch_normalization_1 (BatchNor	(None, 64, 64, 32)	128	conv2d_1[0][0]
activation_1 (Activation)	(None, 64, 64, 32)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 64, 64, 128)	36992	activation_1[0][0]
batch_normalization_2 (BatchNor	(None, 64, 64, 128)	512	conv2d_2[0][0]
activation_2 (Activation)	(None, 64, 64, 128)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 64, 64, 128)	147584	activation_2[0][0]
batch_normalization_3 (BatchNor	(None, 64, 64, 128)	512	conv2d_3[0][0]
activation_3 (Activation)	(None, 64, 64, 128)	0	batch_normalization_3[0][0]
conv2d_4 (Conv2D)	(None, 64, 64, 128)	147584	activation_3[0][0]
batch_normalization_4 (BatchNor	(None, 64, 64, 128)	512	conv2d_4[0][0]
activation_4 (Activation)	(None, 64, 64, 128)	0	batch_normalization_4[0][0]
conv2d_5 (Conv2D)	(None, 64, 64, 128)	147584	activation_4[0][0]
batch_normalization_5 (BatchNor	(None, 64, 64, 128)	512	conv2d_5[0][0]
activation_5 (Activation)	(None, 64, 64, 128)	0	batch_normalization_5[0][0]
concatenate_1 (Concatenate)	(None, 64, 64, 160)	0	activation_1[0][0] activation_3[0][0]
batch_normalization_6 (BatchNor	(None, 64, 64, 160)	640	concatenate_1[0][0]
activation_6 (Activation)	(None, 64, 64, 160)	0	batch_normalization_6[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 160)	0	activation_6[0][0]
conv2d_6 (Conv2D)	(None, 32, 32, 256)	368896	max_pooling2d_1[0][0]
batch_normalization_7 (BatchNor	(None, 32, 32, 256)	1024	conv2d_6[0][0]

Fig. 22. Layers of Densenet-34

activation_7 (Activation)	(None, 32, 32, 256)	0	batch_normalization_7[0][0]
conv2d_7 (Conv2D)	(None, 32, 32, 256)	590080	activation_7[0][0]
batch_normalization_8 (BatchNor	(None, 32, 32, 256)	1024	conv2d_7[0][0]
activation_8 (Activation)	(None, 32, 32, 256)	0	batch_normalization_8[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 256)	590080	activation_8[0][0]

Fig. 23. Layers of Densenet-34(contd.)

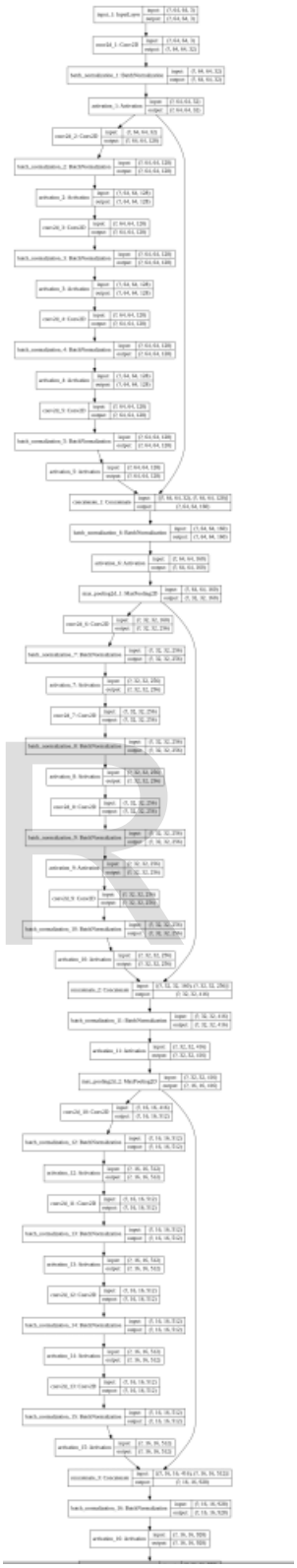


Fig. 24. Layers of Densenet-34(contd.)

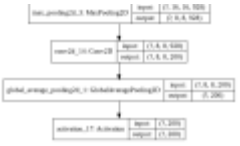
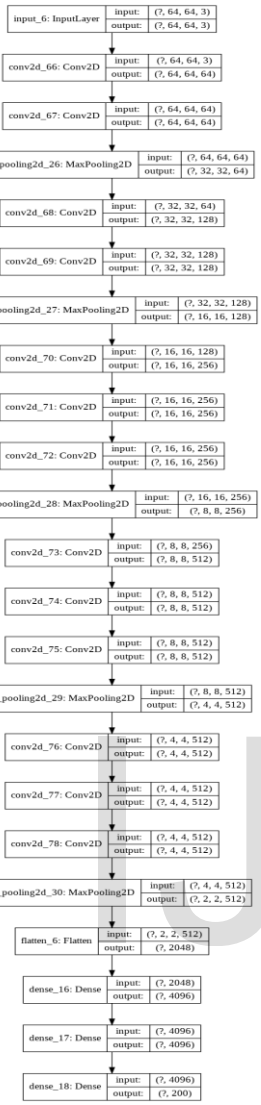


Fig. 25. Inter layer connection with i/p, o/p dimensions for DenseNet-34 model



Model: "model_1"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 3)	0
conv2d_1 (Conv2D)	(None, 64, 64, 64)	1792
conv2d_2 (Conv2D)	(None, 64, 64, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 64)	0
conv2d_3 (Conv2D)	(None, 32, 32, 128)	73856
conv2d_4 (Conv2D)	(None, 32, 32, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 128)	0
conv2d_5 (Conv2D)	(None, 16, 16, 256)	295168
conv2d_6 (Conv2D)	(None, 16, 16, 256)	590080
conv2d_7 (Conv2D)	(None, 16, 16, 256)	590080
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 256)	0
conv2d_8 (Conv2D)	(None, 8, 8, 512)	1180160
conv2d_9 (Conv2D)	(None, 8, 8, 512)	2359808
conv2d_10 (Conv2D)	(None, 8, 8, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 512)	0
conv2d_11 (Conv2D)	(None, 4, 4, 512)	2359808
conv2d_12 (Conv2D)	(None, 4, 4, 512)	2359808
conv2d_13 (Conv2D)	(None, 4, 4, 512)	2359808
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 512)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 4096)	8392704
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 200)	819400

Total params: 40,708,104
Trainable params: 40,708,104
Non-trainable params: 0

Fig.27.Input ,output dimensions for VGG16 model

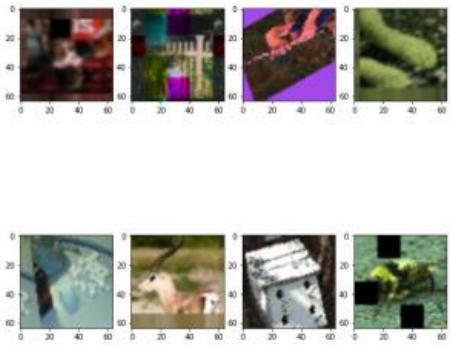


Fig. 28. Group of eight randomly selected training images from DenseNet-34 showing the various augmentations used.

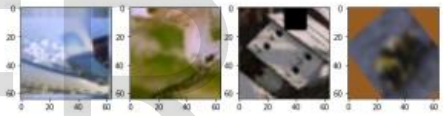


Fig. 29. Group of eight randomly selected training images from DenseNet-50 showing the various augmentations used.

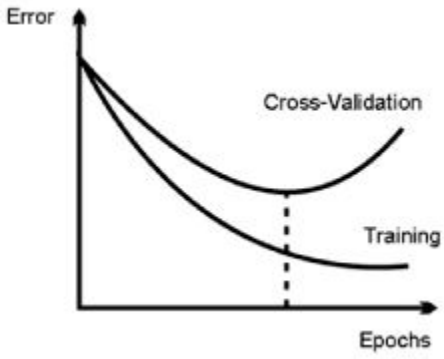


Fig. 30. The dotted line shows the sweet spot where the training must stop

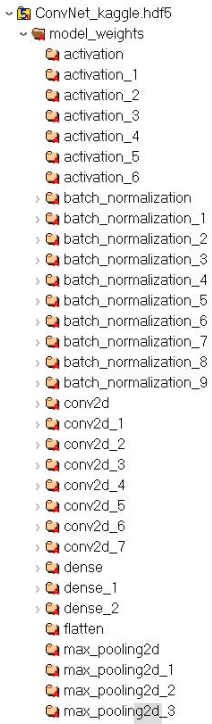


Fig. 31. Figure shows how data is managed in hierarchical format.

	0	1	2	3	4	5	6	7	8	9	10
0	-0.03075...	0.08001...	-0.04149...	0.03017...	-0.07478...	-0.02443...	-0.07721...	0.00764...	0.01609...	-0.04735...	0.04081...
1	0.04162...	-0.05274...	0.01754...	-0.04157...	-0.07135...	-0.03158...	0.00387...	-0.07453...	0.03261...	-0.02545...	-0.01592...
2	0.04621...	0.05625...	-0.04597...	-0.02784...	-0.05887...	-0.01288...	-0.06209...	-0.04264...	-0.07829...	-0.02893...	0.05404...
3	-0.01789...	-0.00982...	-0.02602...	-0.07729...	-0.06257...	0.05498...	-0.06976...	-0.07587...	0.01187...	0.05274...	0.01349...
4	0.05272...	-0.01543...	4.61822...	0.04646...	-0.04965...	-0.01710...	0.05958...	0.0414256	-0.03973...	0.02034...	-0.02741...
5	-0.03073...	-0.07632...	-0.06182...	-0.06992...	-0.01973...	0.02849...	0.07338...	-0.03534...	0.00750...	-0.04627...	-0.04870...
6	0.03762...	-0.04996...	-0.01451...	0.02011...	-0.06958...	-0.03959...	0.02110...	-0.01565...	-0.02622...	0.05254...	-0.00953...
7	-9.23847...	-0.07885...	0.09243...	-0.05210...	0.06987...	0.06573...	0.07178...	0.05464...	-0.05901...	0.03868...	-0.06662...
8	0.05108...	0.05091...	-0.06187...	-0.06850...	0.04034...	-0.01236...	-0.07135...	0.04967...	-0.01929...	5.74264...	0.05410...
9	-0.01546...	-0.06355...	0.05229...	-0.03290...	-0.04532...	-0.06529...	-0.04532...	-0.00190...	-0.05031...	0.03821...	-0.03651...
10	0.01350...	0.02171...	0.04226...	-0.02741...	-0.05422...	0.02437...	0.00505...	-0.01285...	0.07042...	0.02911...	0.05393...

Fig. 32 Figure shows a weight matrix(10X10)

	precision	recall	f1-score	support
goldfish	0.7843	0.8000	0.7921	50
European fire salamander	0.6290	0.7800	0.6964	50
bullfrog	0.3684	0.4200	0.3925	50
tailed frog	0.3902	0.3200	0.3516	50
American alligator	0.4203	0.5800	0.4874	50
boa constrictor	0.4146	0.3400	0.3736	50
trilobite	0.3908	0.6800	0.4964	50
scorpion	0.3023	0.2600	0.2796	50
black widow	0.4912	0.5600	0.5234	50
tarantula	0.4655	0.5400	0.5000	50
centipede	0.3721	0.3200	0.3441	50
goose	0.4528	0.4800	0.4660	50
koala	0.4615	0.7200	0.5625	50
jellyfish	0.7209	0.6200	0.6667	50
brain coral	0.4576	0.5400	0.4954	50

Fig. 33. precision, recall and f1-score using simple covnet

	precision	recall	f1-score	support
goldfish	0.6964	0.7800	0.7358	50
European fire salamander	0.8409	0.7400	0.7872	50
bullfrog	0.5490	0.5600	0.5545	50
tailed frog	0.3448	0.2000	0.2532	50
American alligator	0.4872	0.3800	0.4270	50
boa constrictor	0.3333	0.2800	0.3043	50
trilobite	0.4928	0.6800	0.5714	50
scorpion	0.3488	0.3000	0.3226	50
black widow	0.3571	0.7000	0.4730	50
tarantula	0.4219	0.5400	0.4737	50
centipede	0.4186	0.3600	0.3871	50
goose	0.4000	0.4400	0.4190	50
koala	0.7674	0.6600	0.7097	50
jellyfish	0.7021	0.6600	0.6804	50
brain coral	0.5472	0.5800	0.5631	50
snail	0.4500	0.1800	0.2571	50
slug	0.3333	0.4200	0.3717	50
sea slug	0.5556	0.6000	0.5769	50
American lobster	0.5143	0.3600	0.4235	50
spiny lobster	0.5476	0.4600	0.5000	50
black stork	0.4694	0.4600	0.4646	50
king penguin	0.7255	0.7400	0.7327	50
albatross	0.6977	0.6000	0.6452	50
dugong	0.8205	0.6400	0.7191	50
Chihuahua	0.3226	0.2000	0.2469	50
Yorkshire terrier	0.5152	0.3400	0.4096	50
golden retriever	0.4545	0.5000	0.4762	50
Labrador retriever	0.1724	0.1000	0.1266	50
German shepherd	0.4259	0.4600	0.4423	50
standard poodle	0.2632	0.2000	0.2273	50
tabby	0.3590	0.2800	0.3146	50
Persian cat	0.4828	0.2800	0.3544	50
Egyptian cat	0.4000	0.2000	0.2667	50
cougar	0.4516	0.2800	0.3457	50
lion	0.4630	0.5000	0.4808	50
brown bear	0.6216	0.4600	0.5287	50
ladybug	0.7174	0.6600	0.6875	50
fly	0.4630	0.5000	0.4808	50
bee	0.5000	0.6000	0.5455	50
grasshopper	0.3667	0.2200	0.2750	50
walking stick	0.3659	0.3000	0.3297	50
cockroach	0.3134	0.4200	0.3590	50
mantis	0.4483	0.5200	0.4815	50
dragonfly	0.4138	0.4800	0.4444	50
monarch	0.8070	0.9200	0.8598	50
sulphur butterfly	0.6897	0.8000	0.7407	50

Fig. 34. Precision, recall and f1-score of VGG16 model

	precision	recall	f1-score	support
goldfish	0.8980	0.8800	0.8889	50
European fire salamander	0.8542	0.8200	0.8367	50
bullfrog	0.4706	0.6400	0.5424	50
tailed frog	0.4615	0.3600	0.4045	50
American alligator	0.4630	0.5000	0.4808	50
boa constrictor	0.6571	0.4600	0.5412	50
trilobite	0.5323	0.6600	0.5893	50
scorpion	0.9231	0.2400	0.3810	50
black widow	0.9091	0.4000	0.5556	50
tarantula	0.8333	0.4000	0.5405	50
centipede	0.4407	0.5200	0.4771	50
goose	0.3061	0.6000	0.4054	50
koala	0.9189	0.6800	0.7816	50
jellyfish	0.7917	0.7600	0.7755	50
brain coral	0.4902	0.5000	0.4950	50
snail	0.6047	0.5200	0.5591	50
slug	0.3846	0.4000	0.3922	50
sea slug	0.3645	0.7800	0.4968	50
American lobster	0.7059	0.4800	0.5714	50
spiny lobster	0.5250	0.4200	0.4667	50
black stork	0.6071	0.6800	0.6415	50
king penguin	0.7273	0.8000	0.7619	50
albatross	0.6316	0.7200	0.6729	50
dugong	0.7660	0.7200	0.7423	50
Chihuahua	0.7500	0.1800	0.2903	50
Yorkshire terrier	0.6087	0.5600	0.5833	50
golden retriever	0.5854	0.4800	0.5275	50
Labrador retriever	0.5000	0.1200	0.1935	50
German shepherd	0.6571	0.4600	0.5412	50
standard poodle	0.4286	0.1800	0.2535	50
tabby	0.4884	0.4200	0.4516	50
Persian cat	0.7179	0.5600	0.6292	50
Egyptian cat	0.4722	0.3400	0.3953	50
cougar	0.7500	0.3000	0.4286	50
lion	0.6500	0.5200	0.5778	50
brown bear	0.7838	0.5800	0.6667	50
ladybug	0.6154	0.6400	0.6275	50
fly	0.6750	0.5400	0.6000	50
bee	0.8529	0.5800	0.6905	50
grasshopper	0.3462	0.3600	0.3529	50
walking stick	0.3421	0.5200	0.4127	50
cockroach	0.3429	0.4800	0.4000	50
mantis	0.5000	0.5000	0.5000	50
dragonfly	0.3855	0.6400	0.4812	50
monarch	0.8136	0.9600	0.8807	50
sulphur butterfly	0.7759	0.9000	0.8333	50

Fig. 35. Precision, recall and f1-score for model resnet-34

	precision	recall	f1-score	support
goldfish	0.9200	0.9200	0.9200	50
European fire salamander	0.8302	0.8800	0.8544	50
bullfrog	0.6744	0.5800	0.6237	50
tailed frog	0.5588	0.3800	0.4524	50
American alligator	0.8333	0.5000	0.6250	50
boa constrictor	0.5091	0.5600	0.5333	50
trilobite	0.5443	0.8600	0.6667	50
scorpion	0.3803	0.5400	0.4463	50
black widow	0.5132	0.7800	0.6190	50
tarantula	0.6667	0.4800	0.5581	50
centipede	0.5333	0.6400	0.5818	50
goose	0.5439	0.6200	0.5794	50
koala	0.9730	0.7200	0.8276	50
jellyfish	0.7708	0.7400	0.7551	50
brain coral	0.7292	0.7000	0.7143	50
snail	0.5833	0.4200	0.4884	50
slug	0.5952	0.5000	0.5435	50
sea slug	0.6957	0.6400	0.6667	50
American lobster	0.5577	0.5800	0.5686	50
spiny lobster	0.5536	0.6200	0.5849	50
black stork	0.5942	0.8200	0.6891	50
king penguin	0.9302	0.8000	0.8602	50
albatross	0.8043	0.7400	0.7708	50
dugong	0.7458	0.8800	0.8073	50
Chihuahua	0.6957	0.3200	0.4384	50
Yorkshire terrier	0.8378	0.6200	0.7126	50
golden retriever	0.5854	0.4800	0.5275	50
Labrador retriever	0.4186	0.3600	0.3871	50
German shepherd	0.7949	0.6200	0.6966	50
standard poodle	0.8182	0.3600	0.5000	50
tabby	0.4286	0.3600	0.3913	50
Persian cat	0.6731	0.7000	0.6863	50
Egyptian cat	0.5143	0.3600	0.4235	50
cougar	0.6667	0.4400	0.5301	50
lion	0.6327	0.6200	0.6263	50
brown bear	0.6923	0.7200	0.7059	50
ladybug	0.7917	0.7600	0.7755	50
fly	0.5345	0.6200	0.5741	50
bee	0.7111	0.6400	0.6737	50
grasshopper	0.4651	0.4000	0.4301	50
walking stick	0.4103	0.6400	0.5000	50
cockroach	0.4545	0.6000	0.5172	50
mantis	0.5254	0.6200	0.5688	50
dragonfly	0.4561	0.5200	0.4860	50
monarch	0.7812	1.0000	0.8772	50
sulphur butterfly	0.7188	0.9200	0.8070	50

Fig. 36. Precision, recall and f1-score for model densenet-34

• *Ayush Sharma is currently pursuing bachelors degree program in Computer Science and Engineering in Vellore Institute of Technology,Vellore,Tamil Nadu, India, PH-7061870355. E-mail: ayushsharma.2k1@gmail.com*