Aim

To develop image classification models using various different supervised learning algorithms and to compare the results generated by them.

ABSTRACT

There are many methods available which can classify the images, but there also remains a confusion when there are a lot of choices as it remains unclear that which model can best analyze and intelligently classify the images.

In this project we try to build multiple generalized image classification models using the concept of neural network, decision tree, convolution neural network with multiple feature extractor like AlexNet, VGGNet.

For this study we have used commonly available Pokemon dataset as it’s resemblance to common things will make our model more generalized and hence our study. The dataset is available on Kaggle with the name Pokemon Generation one.

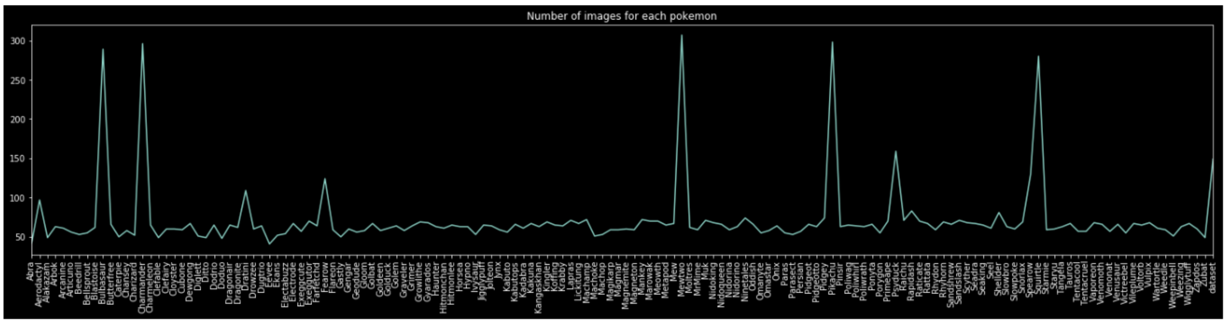
There are many factors which can influence the image classification and thereby making it quite complex. During our study we have tried to minimize the effect of such external factors by providing the same environment in terms of the dimensions of the input(image) which has been converted into a 96X96X3 RGB image, the number of iterations for training so that our results remains unbiased.

The motivation for undertaking this project was that image classification is a widely use concept and has its roots in almost every machine learning or artificial intelligence applications, whether it be defense sector, medical science, utility etc.

Dataset Description

The dataset contains image samples of 150 Pokémons with 10842 images. All the images of the Pokemons are separated on the basis of folders with names of Pokemons. Hence making it easy to understand.

```
Total number of pokemons: 150
Total number of images: 10842
```



Tools Used

1. **Pandas**

   Pandas is an open-source Python library which gives you a set of tools to do data analysis, data manipulation and cleaning.

2. **Numpy**

   Numpy is a library in Python Programming language for scientific computing. It provides multidimensional high performance array and matrices objects along with tools to manipulate or work with these arrays.

3. **Graphviz**

   It is a software used for visualization. Graphviz takes details/description of a graph in text format and converts it into diagram in a suitable format.

4. **Matplotlib**

   This library is used for visualization in Python for 2 dimensional plots of lists. Matplotlib has different types of plots like line, bar, scatter, histogram etc. Plots help in understanding patterns and trends in data.

5. **Keras**

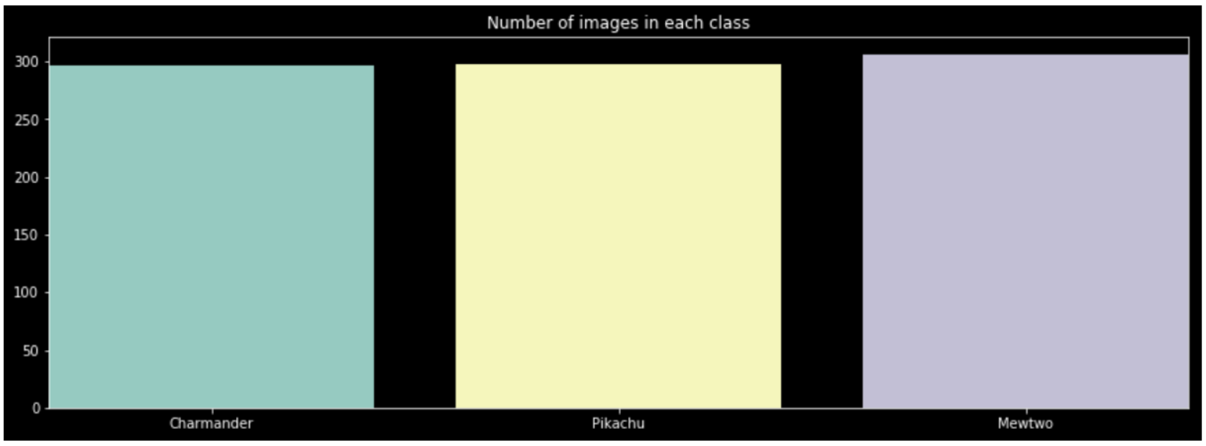
   This is an open-source python library used for developing and evaluating deep learning models. It runs on top of TensorFlow, Theano or CNTK. High performance API used to specify and train differentiable models.

6. **Scikit-Learn**

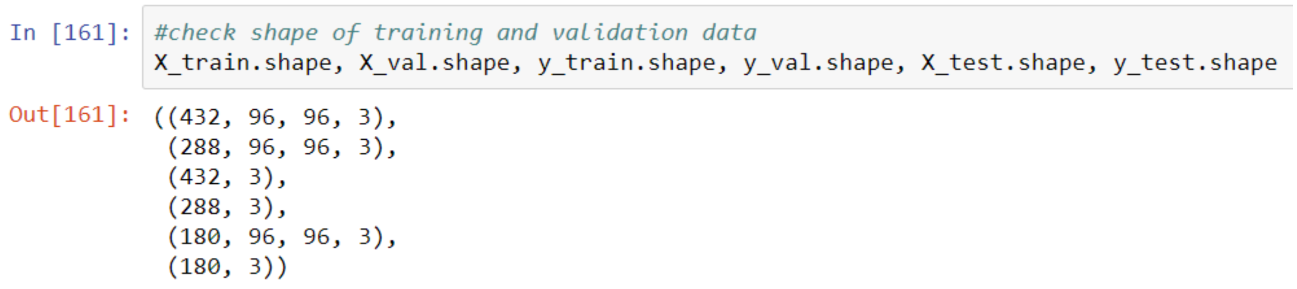
   Scikit-learn is an open-source, commercially usable Python library used for Machine learning, data mining and data analysis. It is built upon popular libraries which are NumPy, pandas and Matplotlib.

Data Description

Number of images in each class taken into consideration.



Shape of Training, Validation and Test sets.



```
In [161]: #check shape of training and validation data
          X_train.shape, X_val.shape, y_train.shape, y_val.shape, X_test.shape, y_test.shape

Out[161]: ((432, 96, 96, 3),
           (288, 96, 96, 3),
           (432, 3),
           (288, 3),
           (180, 96, 96, 3),
           (180, 3))
```

Methodology

Out of the 149 Pokemons in our dataset with chose 3 Pokemons with the highest number of images as our in focus dataset. Since the number of images varied considerably over the entire dataset.

Decision Tree Model:

Decision Trees (DTs) are a supervised, non-parametric learning method used to classify or predict data. The main target of a Decision Tree model is to predict the value of a variable or classify data in different classes.

A decision tree represents all the possible solutions available to us in a graphical manner. It has a root node and it further develops branches for each solution at a particular node.

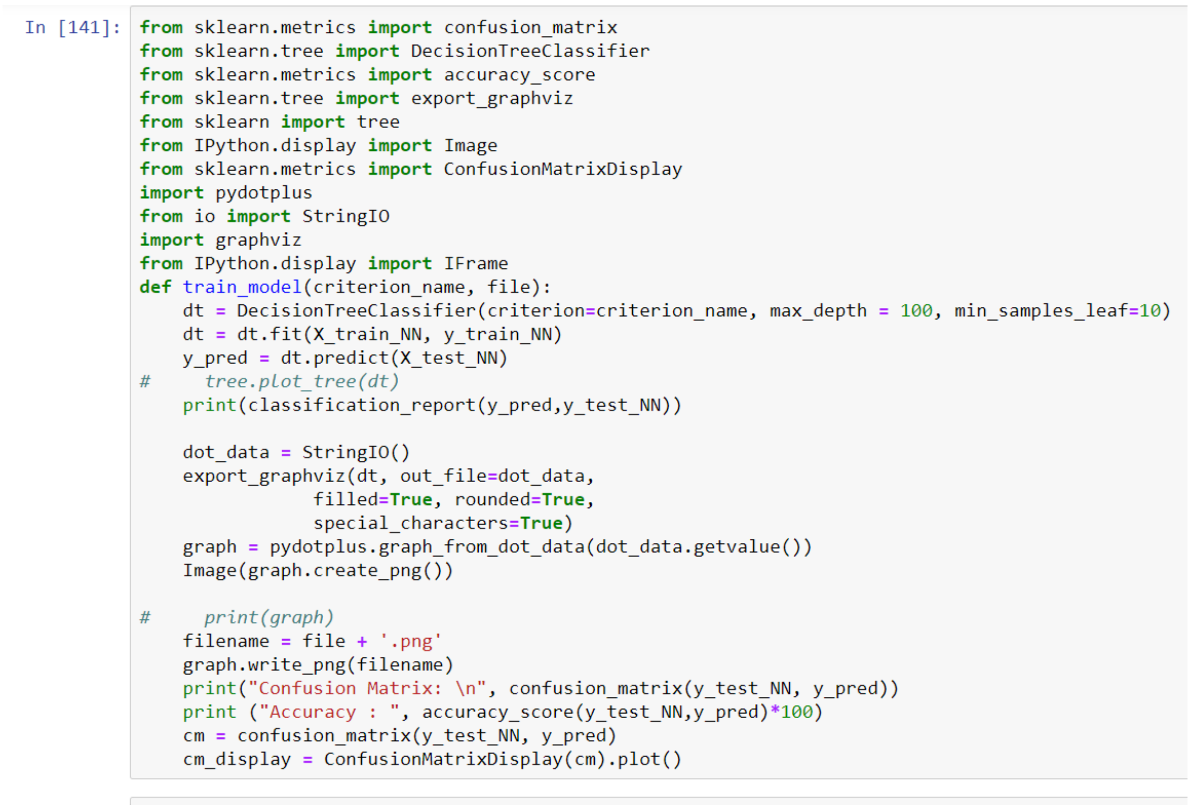
Some Advantages of Decision Trees are:

- Easy to visualize and interpret
- Pre-processing is less expensive than other algorithms in terms of code and time.
- Handles multi-classification problems easily.
- The output doesn't get affected because of some empty values.
- Data normalization is not required

Some disadvantages of Decision Trees are:

- A small change in the input data can lead to an entirely different solution.
- Can overfit data easily and create overly complex trees.
- If there is a dominant class in the data, it might lead to a biased Decision Tree.

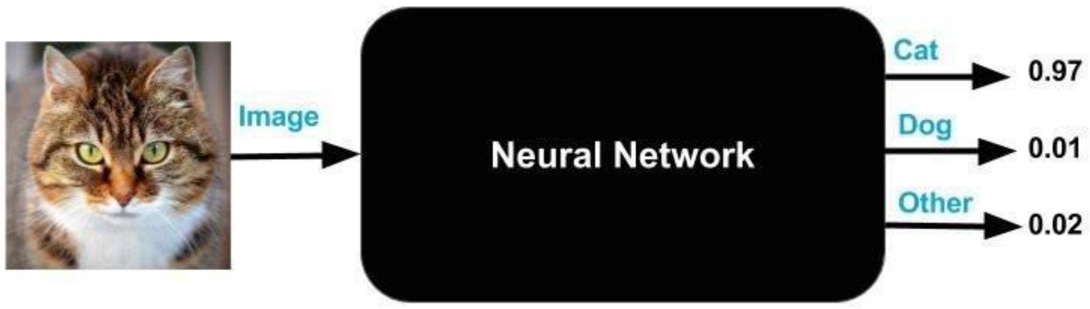
Code Snippet:



```
In [141]: from sklearn.metrics import confusion_matrix
          from sklearn.tree import DecisionTreeClassifier
          from sklearn.metrics import accuracy_score
          from sklearn.tree import export_graphviz
          from sklearn import tree
          from IPython.display import Image
          from sklearn.metrics import ConfusionMatrixDisplay
          import pydotplus
          from io import StringIO
          import graphviz
          from IPython.display import IFrame
          def train_model(criterion_name, file):
              dt = DecisionTreeClassifier(criterion=criterion_name, max_depth = 100, min_samples_leaf=10)
              dt = dt.fit(X_train_NN, y_train_NN)
              y_pred = dt.predict(X_test_NN)
          #     tree.plot_tree(dt)
              print(classification_report(y_pred,y_test_NN))

              dot_data = StringIO()
              export_graphviz(dt, out_file=dot_data,
                          filled=True, rounded=True,
                          special_characters=True)
              graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
              Image(graph.create_png())

          #     print(graph)
              filename = file + '.png'
              graph.write_png(filename)
              print("Confusion Matrix: \n", confusion_matrix(y_test_NN, y_pred))
              print ("Accuracy : ", accuracy_score(y_test_NN,y_pred)*100)
              cm = confusion_matrix(y_test_NN, y_pred)
              cm_display = ConfusionMatrixDisplay(cm).plot()
```

<u>Neural Networks Model:</u>

Neural network layers consist of a large number of interconnected 'nodes', each of which comprises of an 'activation function' inside them. The network get the pattern through the 'input layer'. These input layers communicate to a number of 'hidden layers'. Most of the processing is done in hidden layers using an arrangement of weighted 'connections'.
An 'output layer' receives the output from the hidden layers.

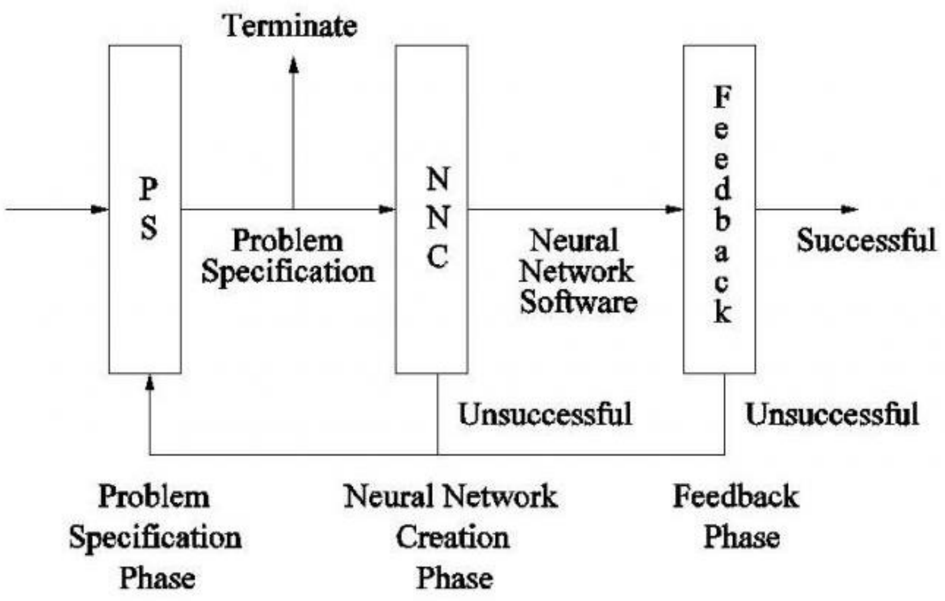


Some advantages of Neural Networks are:

- A neural network is not limited to the input provided to them.
- It stores input in its own layers rather than a database
- Even if a neuron is non-responsive, a neural network still works
- Parallel working of multiple neurons without affecting the efficiency or accuracy

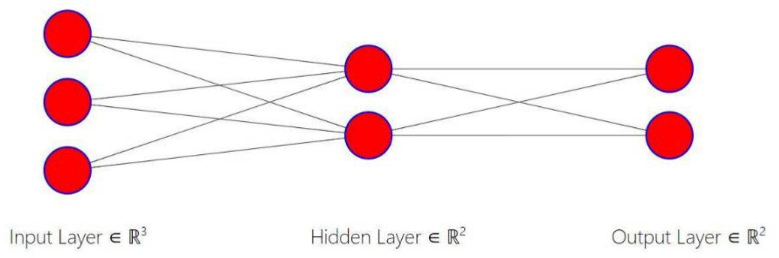
Some disadvantages of Neural Networks are:

- Blackbox nature- you are not aware of the reasons behind the particular output of your neural network.
- Time of development takes more time.
- Computations are more expensive for a neural network than any standard algorithm.

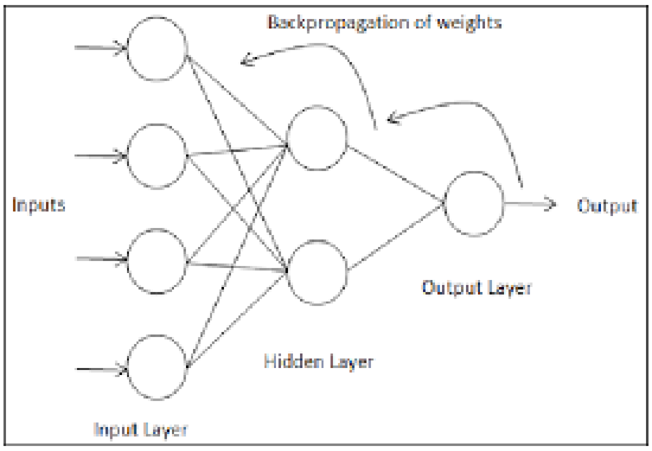


In our Neural Networks class we have implemented forward propagation and backward propagation. Here depth is number of hidden layers, width is the number of hidden nodes on each layer.

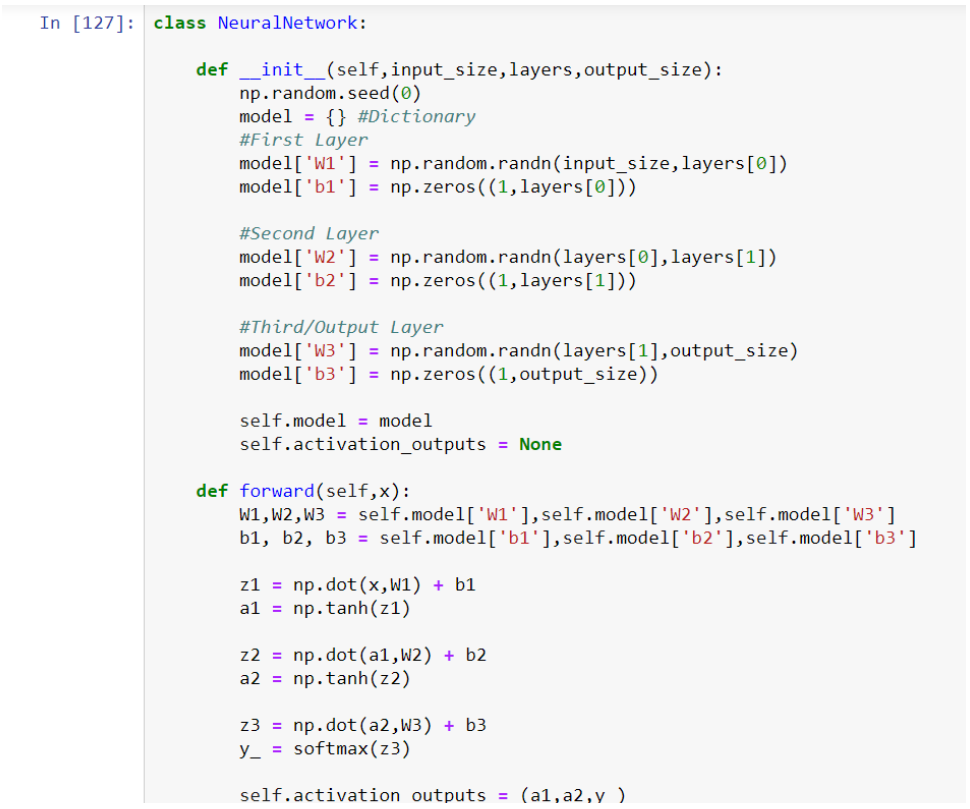
- Forward Propagation
  The input provides the starting information that then propagates to the hidden nodes at each layer and produces the output y. In forward propagation, depth is more essential than width of the hidden layers.



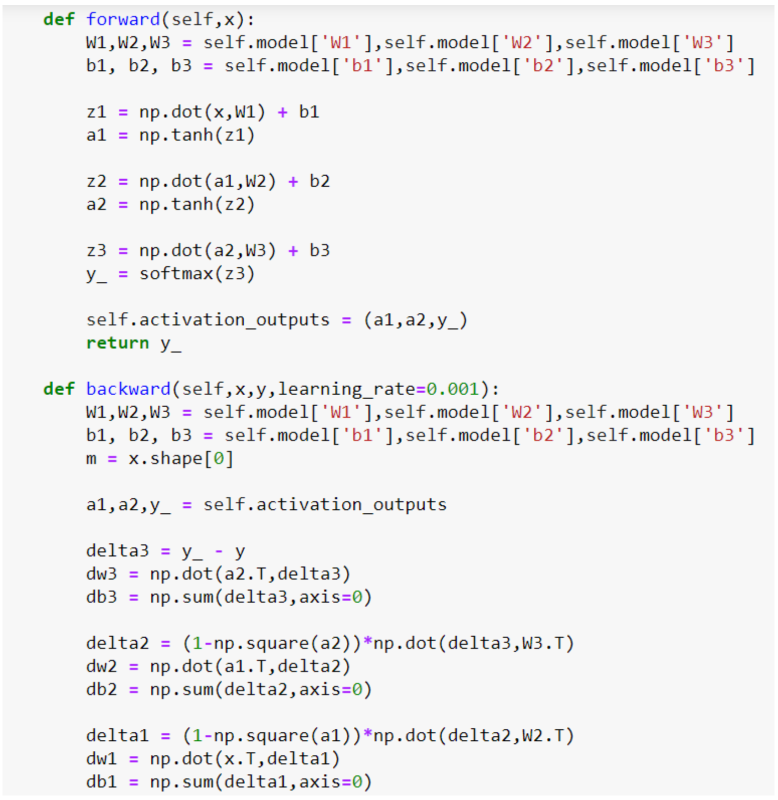
- Backward Propagation
  The Backpropagation algorithm finds the minimum error function in weight space using gradient descent. The value of weights that helps to achieve minimum error is a solution to the learning problem. Backpropagation allows the cost to go backward to compute the gradient.



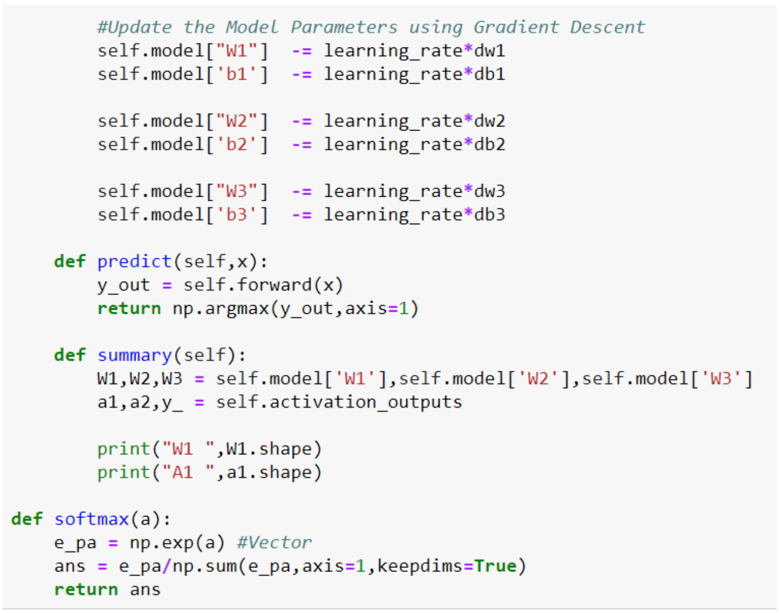
<u>Code Snippet:</u>



```
In [127]: class NeuralNetwork:

              def __init__(self,input_size,layers,output_size):
                  np.random.seed(0)
                  model = {} #Dictionary
                  #First Layer
                  model['W1'] = np.random.randn(input_size,layers[0])
                  model['b1'] = np.zeros((1,layers[0]))

                  #Second Layer
                  model['W2'] = np.random.randn(layers[0],layers[1])
                  model['b2'] = np.zeros((1,layers[1]))

                  #Third/Output Layer
                  model['W3'] = np.random.randn(layers[1],output_size)
                  model['b3'] = np.zeros((1,output_size))

                  self.model = model
                  self.activation_outputs = None

              def forward(self,x):
                  W1,W2,W3 = self.model['W1'],self.model['W2'],self.model['W3']
                  b1, b2, b3 = self.model['b1'],self.model['b2'],self.model['b3']

                  z1 = np.dot(x,W1) + b1
                  a1 = np.tanh(z1)

                  z2 = np.dot(a1,W2) + b2
                  a2 = np.tanh(z2)

                  z3 = np.dot(a2,W3) + b3
                  y_ = softmax(z3)

                  self.activation_outputs = (a1,a2,y_)
```



```
    def forward(self,x):
        W1,W2,W3 = self.model['W1'],self.model['W2'],self.model['W3']
        b1, b2, b3 = self.model['b1'],self.model['b2'],self.model['b3']

        z1 = np.dot(x,W1) + b1
        a1 = np.tanh(z1)

        z2 = np.dot(a1,W2) + b2
        a2 = np.tanh(z2)

        z3 = np.dot(a2,W3) + b3
        y_ = softmax(z3)

        self.activation_outputs = (a1,a2,y_)
        return y_

    def backward(self,x,y,learning_rate=0.001):
        W1,W2,W3 = self.model['W1'],self.model['W2'],self.model['W3']
        b1, b2, b3 = self.model['b1'],self.model['b2'],self.model['b3']
        m = x.shape[0]

        a1,a2,y_ = self.activation_outputs

        delta3 = y_ - y
        dw3 = np.dot(a2.T,delta3)
        db3 = np.sum(delta3,axis=0)

        delta2 = (1-np.square(a2))*np.dot(delta3,W3.T)
        dw2 = np.dot(a1.T,delta2)
        db2 = np.sum(delta2,axis=0)

        delta1 = (1-np.square(a1))*np.dot(delta2,W2.T)
        dw1 = np.dot(x.T,delta1)
        db1 = np.sum(delta1,axis=0)
```



```
        #Update the Model Parameters using Gradient Descent
        self.model["W1"]  -= learning_rate*dw1
        self.model['b1']  -= learning_rate*db1

        self.model["W2"]  -= learning_rate*dw2
        self.model['b2']  -= learning_rate*db2

        self.model["W3"]  -= learning_rate*dw3
        self.model['b3']  -= learning_rate*db3

    def predict(self,x):
        y_out = self.forward(x)
        return np.argmax(y_out,axis=1)

    def summary(self):
        W1,W2,W3 = self.model['W1'],self.model['W2'],self.model['W3']
        a1,a2,y_ = self.activation_outputs

        print("W1 ",W1.shape)
        print("A1 ",a1.shape)

def softmax(a):
    e_pa = np.exp(a) #Vector
    ans = e_pa/np.sum(e_pa,axis=1,keepdims=True)
    return ans
```

Convolutional Neural Networks:

CNN is a deep learning algorithm that has shown higher accuracies as compared to a deep neural network in classifying images. The design of CNN is similar to the connectivity pattern of Neurons in a Human Brain. CNN is a series of layers. Each of these layers converts one volume of activations to another using a differentiable function.

Some advantages of Convolutional Neural Networks are:

- Without any human direction, characteristic features are detected
- CNN’s are much better than Neural networks in the task of image recognition
- CNN’s can be used to extract important attributes for a new problem

Some disadvantages of Convolutional Neural Networks are:

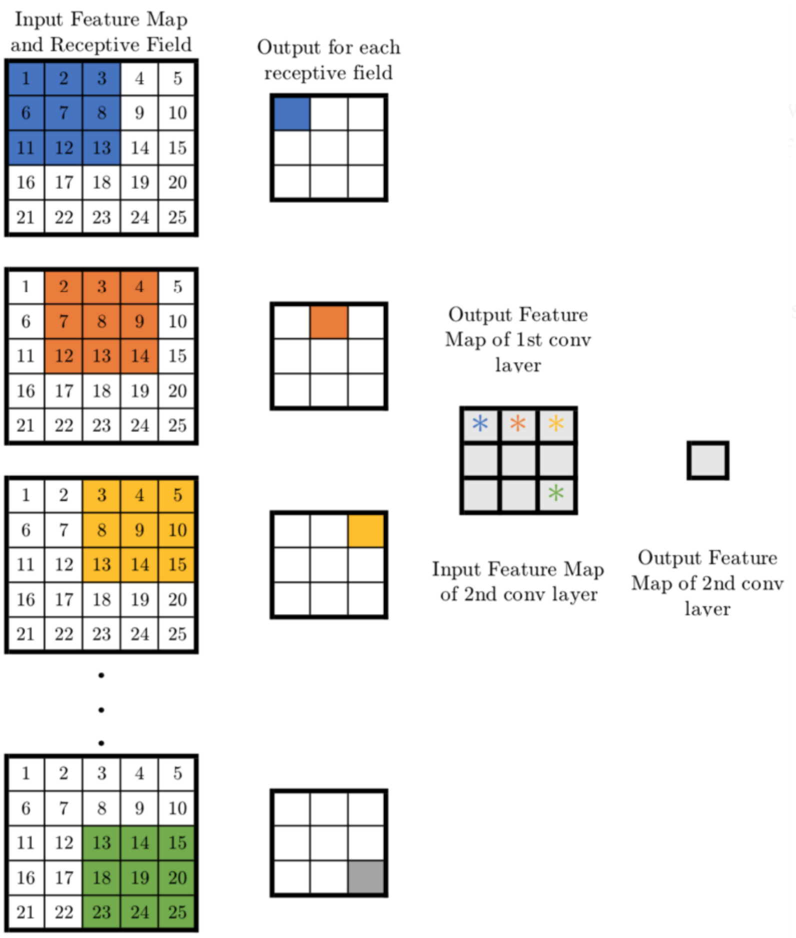
- CNN’s do not take into consideration the spatial relationships
- Computationally expensive like all NN.
- Overfitting is a common problem with CNN’s.

CNN Model using VGGNet Architecture:

VGG 16 was created by Andrew Zisserman and Karen Simonyan of a group of Oxford University in 2014 in the paper “VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION”.

VGGNet aims to reduce the number of parameters in the Convolutional Layers and thereby improving the training duration of the dataset.

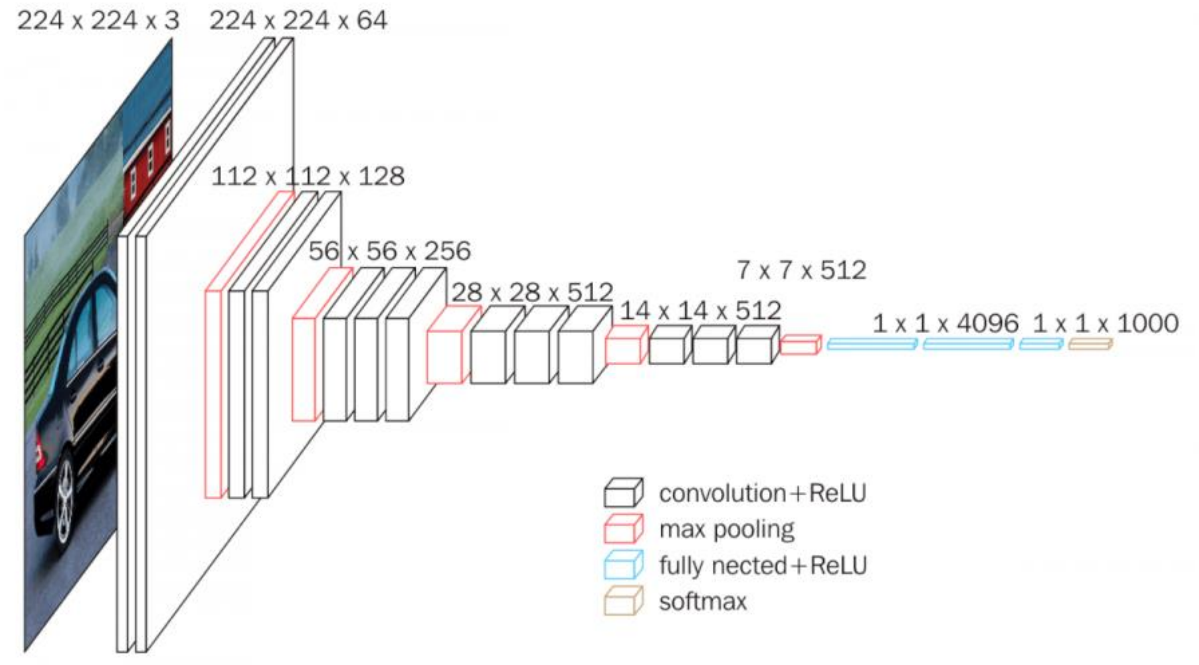
Example of a VGGNet with 5x5 size, implemented on 3x3 conv layers.


Input Feature Map and Receptive Field
Output for each receptive field
Output Feature Map of 1st conv layer
Input Feature Map of 2nd conv layer
Output Feature Map of 2nd conv layer

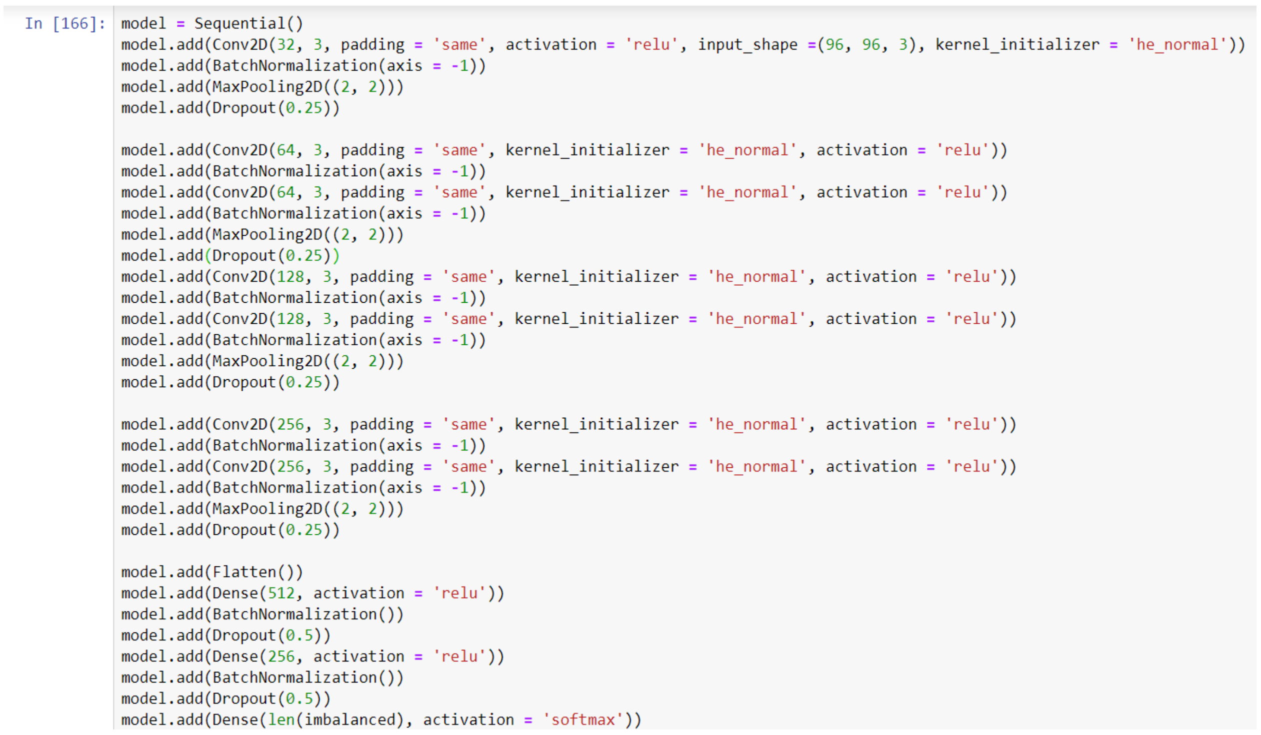
VGGNet consists of a CONV (convolutional layers), FC (fully connected layers) and max pooling layers.

Convolutional layers and fully connected layers use filters of size 3X3. Max pooling layers use filers of size 2 X 2.

Sample Architecture of a VGGNet architecture.



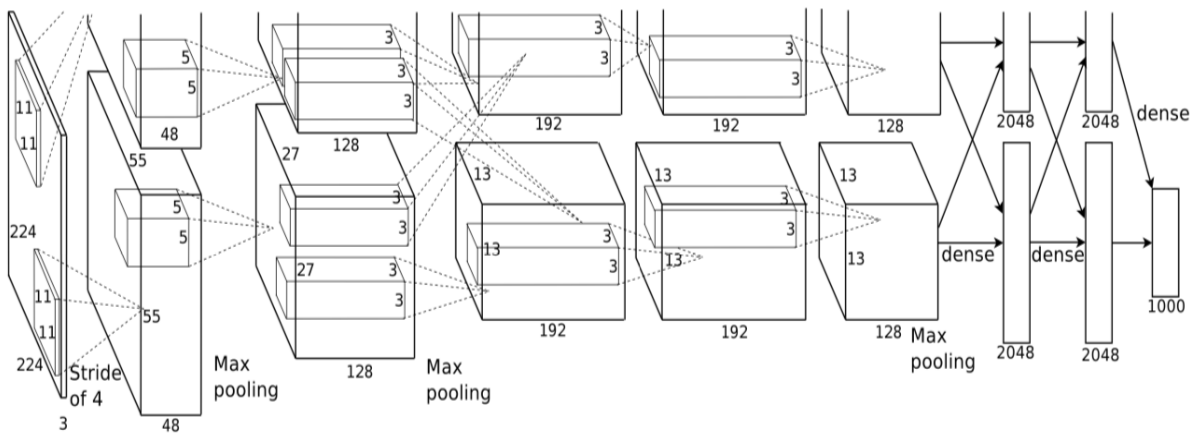
Code Snippet:



```
In [166]: model = Sequential()
          model.add(Conv2D(32, 3, padding = 'same', activation = 'relu', input_shape =(96, 96, 3), kernel_initializer = 'he_normal'))
          model.add(BatchNormalization(axis = -1))
          model.add(MaxPooling2D((2, 2)))
          model.add(Dropout(0.25))

          model.add(Conv2D(64, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(Conv2D(64, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(MaxPooling2D((2, 2)))
          model.add(Dropout(0.25))
          model.add(Conv2D(128, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(Conv2D(128, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(MaxPooling2D((2, 2)))
          model.add(Dropout(0.25))

          model.add(Conv2D(256, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(Conv2D(256, 3, padding = 'same', kernel_initializer = 'he_normal', activation = 'relu'))
          model.add(BatchNormalization(axis = -1))
          model.add(MaxPooling2D((2, 2)))
          model.add(Dropout(0.25))

          model.add(Flatten())
          model.add(Dense(512, activation = 'relu'))
          model.add(BatchNormalization())
          model.add(Dropout(0.5))
          model.add(Dense(256, activation = 'relu'))
          model.add(BatchNormalization())
          model.add(Dropout(0.5))
          model.add(Dense(len(imbalanced), activation = 'softmax'))
```

CNN Model using AlexNet Architecture:

There are 3 fully-connected layers and 5 convolutional layers,i.e., a total of 8 layers.



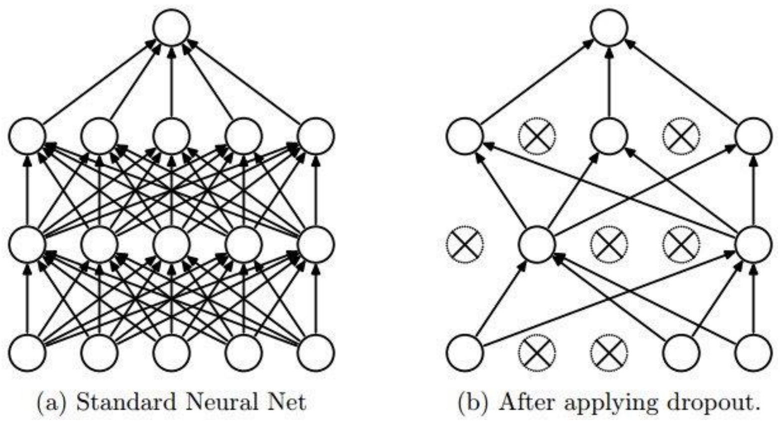
The features that put AlexNet at an advantage are:

- **ReLU Nonlinearity**. In place of the standard tanh function, AlexNet uses Rectified Linear Units (ReLU). ReLU provides it an edge over others in terms of training time.

- **Multiple GPUs.** Multi-GPUs can be trained using AlexNet. You can put 50% of neurons on a GPU and the rest of the 50% on another one. You can train a bigger model now, at the same time you can cut the costs incurred due to training time.

- **Overlapping Pooling.** Usually**,** CNN’s “pool” outputs of adjacent groups of neurons with no overlapping. But on using the overlap, a reduction by about 0.5% was observed in error and a study also found that those models are hard to overfit.

- **Overfitting in AlexNet -** In AlexNet major concern in terms of overfitting was it used 60 million+ parameters. Data Augmentation and Dropout could be implemented to undertake this problem.



<u>Relu (Rectified Linear Activation Unit)</u>:

Relu, used in deep neural networks, is a piecewise linear function. It is represented as:

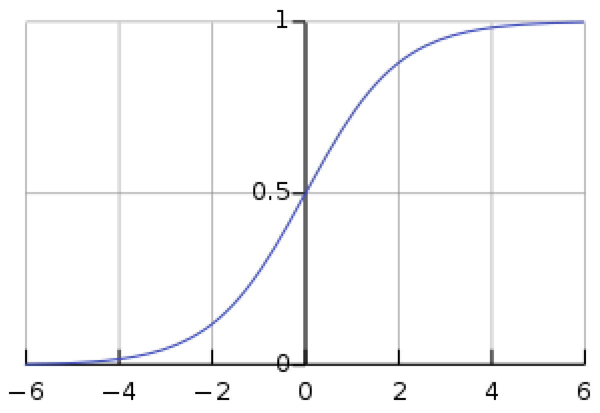
$$f(x) = \max(0, x) \qquad (1)$$

The maximum value between zero and the input value is our output. For negative values output is zero and the input value itself,

$$f(x) = \begin{cases} 0, if\ x < 0 \\ x, if\ x \geq 0 \end{cases} \qquad (2)$$

otherwise.

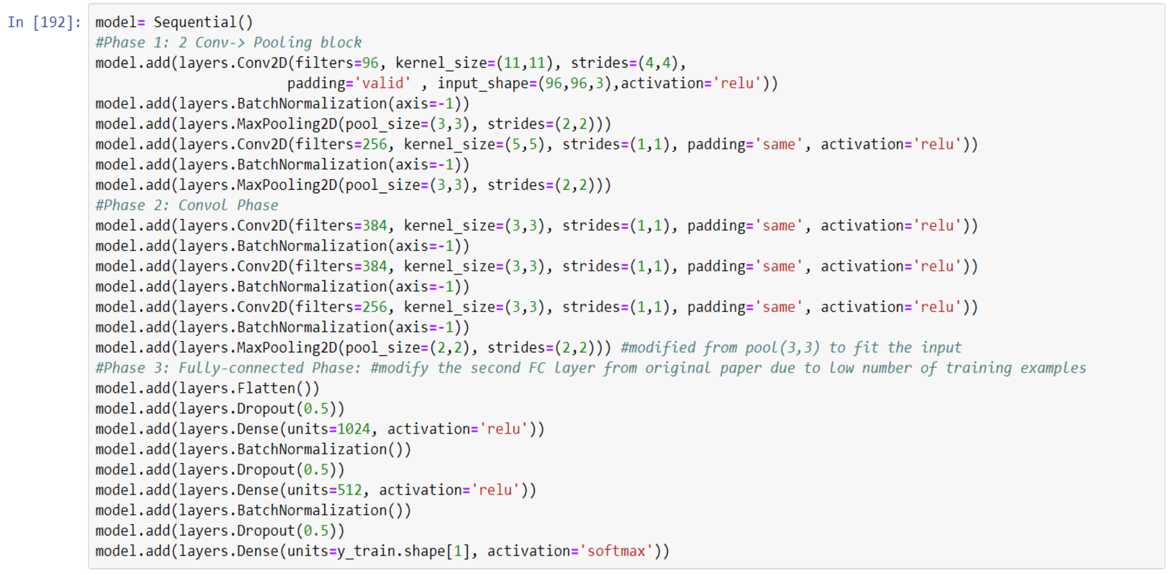
here x = an input value



Advantages of ReLu:

- **Computation saving** - the ReLu function is able to stimulate the training speed of deep neural networks as compared to conventional activation functions. Because of positive input, the derivative of ReLu is 1. Deep neural networks do not take extra time for calculating error terms throughout the training phase.

Code Snippet:



```
In [192]: model= Sequential()
          #Phase 1: 2 Conv-> Pooling block
          model.add(layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4),
                                  padding='valid' , input_shape=(96,96,3),activation='relu'))
          model.add(layers.BatchNormalization(axis=-1))
          model.add(layers.MaxPooling2D(pool_size=(3,3), strides=(2,2)))
          model.add(layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same', activation='relu'))
          model.add(layers.BatchNormalization(axis=-1))
          model.add(layers.MaxPooling2D(pool_size=(3,3), strides=(2,2)))
          #Phase 2: Convol Phase
          model.add(layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
          model.add(layers.BatchNormalization(axis=-1))
          model.add(layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
          model.add(layers.BatchNormalization(axis=-1))
          model.add(layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
          model.add(layers.BatchNormalization(axis=-1))
          model.add(layers.MaxPooling2D(pool_size=(2,2), strides=(2,2))) #modified from pool(3,3) to fit the input
          #Phase 3: Fully-connected Phase: #modify the second FC layer from original paper due to low number of training examples
          model.add(layers.Flatten())
          model.add(layers.Dropout(0.5))
          model.add(layers.Dense(units=1024, activation='relu'))
          model.add(layers.BatchNormalization())
          model.add(layers.Dropout(0.5))
          model.add(layers.Dense(units=512, activation='relu'))
          model.add(layers.BatchNormalization())
          model.add(layers.Dropout(0.5))
          model.add(layers.Dense(units=y_train.shape[1], activation='softmax'))
```

Adam Optimizer:

Adam is an optimization algorithm. It replaced the traditional stochastic gradient descent method to re-calculate weights based on the training set.

Advantages of adam optimizer:

- Easy implementation.
- Computation costs are minimized.
- Lesser storage requirements.
- Resistant to rescale of the gradients.
- Better to use in models with a large number of parameters or those with noisy gradients.
- Hyper-parameters requires very little tuning.
- Adam is basically a combination of the benefits of two extensions of stochastic gradient descent- RMS Propagation and Adaptive Gradient.

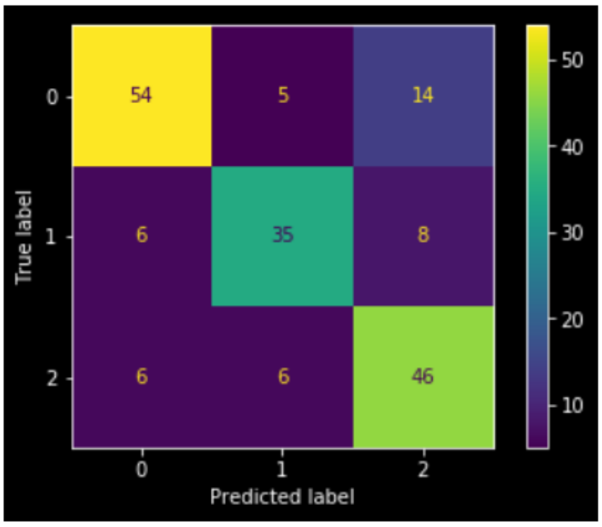
Result

Below are the results obtained from various models implemented in the project. The result for each model is as follows:

1. Decision Tree

```
              precision    recall  f1-score   support

           0       0.74      0.82      0.78        66
           1       0.71      0.76      0.74        46
           2       0.79      0.68      0.73        68

    accuracy                           0.75       180
   macro avg       0.75      0.75      0.75       180
weighted avg       0.75      0.75      0.75       180

Confusion Matrix:
 [[54  5 14]
 [ 6 35  8]
 [ 6  6 46]]
Accuracy :  75.0
```

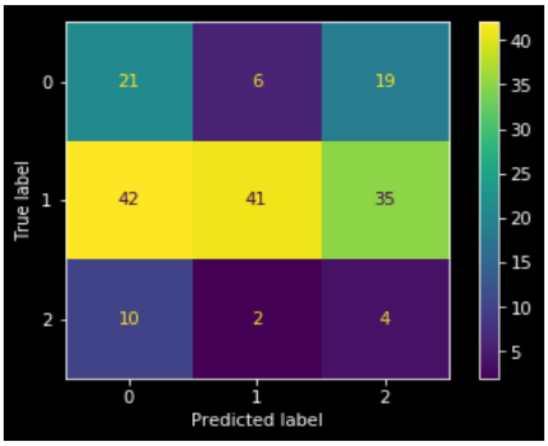
The decision tree gives any overall accuracy of 75%. Out of 180 test images, 135 images were correctly classified.



Entropy (used for Information Gain) was used as criteria in the decision tree with max_depth (The maximum depth of a tree possible) set to 100 and minimum samples required to be at leaf node set to 10.

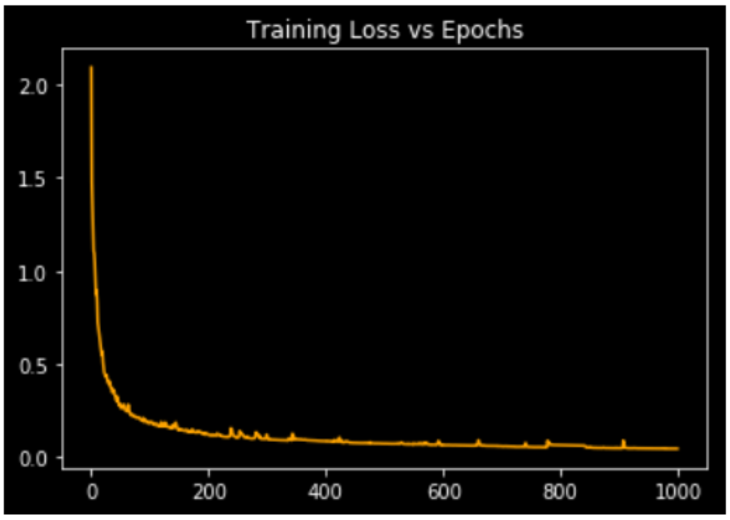
2. Neural Network

```
              precision    recall  f1-score   support

           0       0.29      0.46      0.35        46
           1       0.84      0.35      0.49       118
           2       0.07      0.25      0.11        16

    accuracy                           0.37       180
   macro avg       0.40      0.35      0.32       180
weighted avg       0.63      0.37      0.42       180
```



This model gave an overall accuracy of 37% on the test dataset with 66 images being correctly classified in the test dataset of 180 images. 3 Layers were defined in the dataset (2 hidden layers and 1 output layer).

Loss was computed for 1000 epochs with a learning rate = 0.0002.



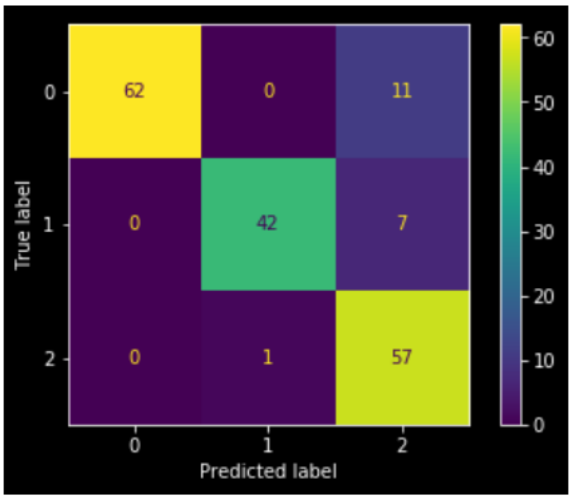
3. Convolutional Neural Network (VGGNet16 Architecture)

VGGNet aims to reduce the number of parameters in the Convolutional Layers and thereby improving the training duration of the dataset.

Model Summary:

```
Total params: 6,020,995
Trainable params: 6,017,603
Non-trainable params: 3,392
```

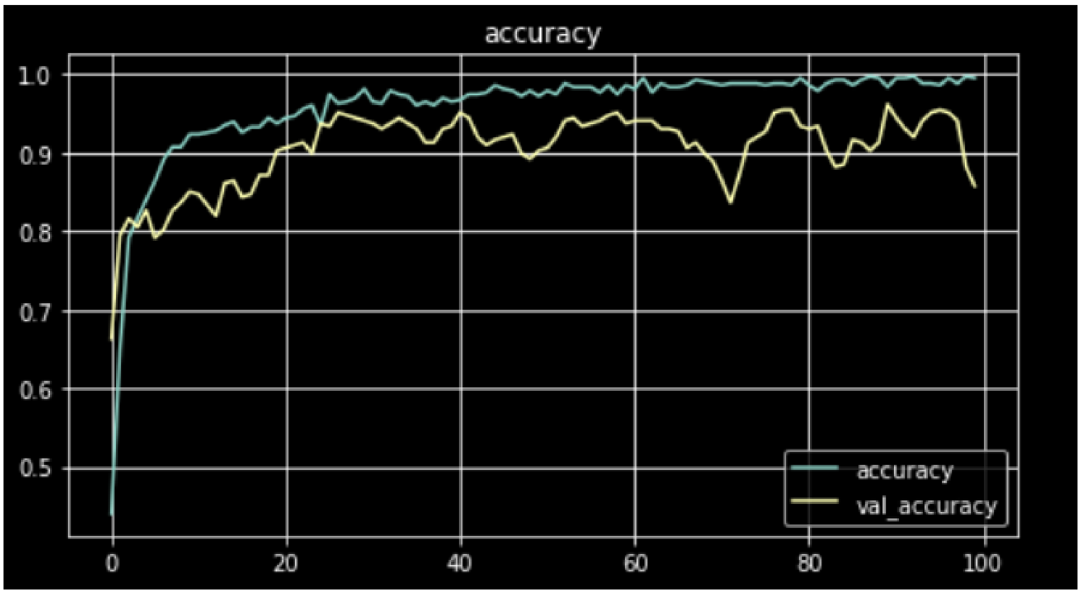
```
              precision    recall  f1-score   support

           0       1.00      0.85      0.92        73
           1       0.98      0.86      0.91        49
           2       0.76      0.98      0.86        58

    accuracy                           0.89       180
   macro avg       0.91      0.90      0.90       180
weighted avg       0.92      0.89      0.90       180
```

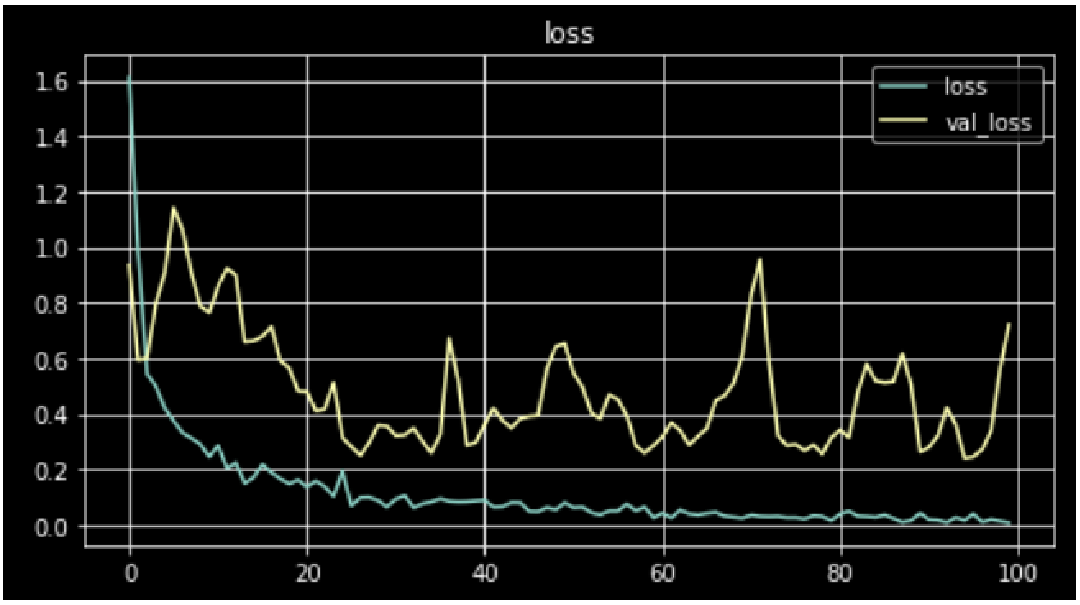


This model gave an overall accuracy of 89% on the test dataset and 161 images were correctly classified out of 180 images in the test dataset.

The training data was further divided into training and validation datasets.

Training accuracy and validation accuracy and Training loss and Validation loss was computed for 100 epochs with steps per epoch set to default.





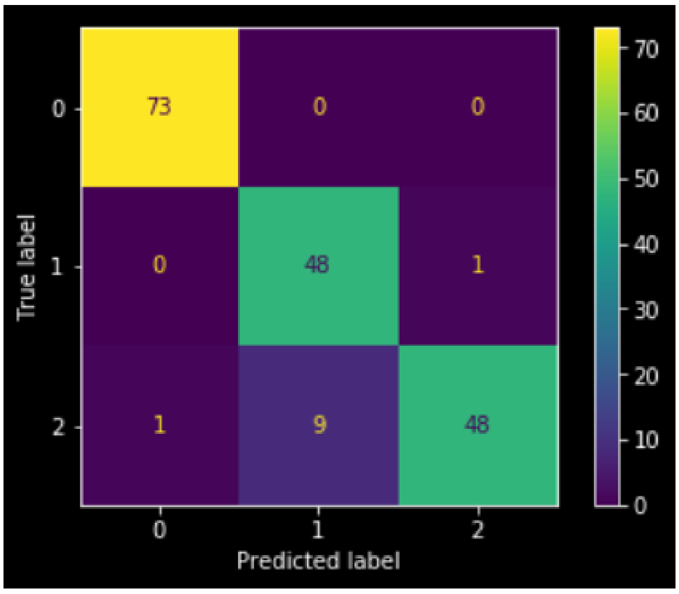
4. Convolutional Neural Network (AlexNet Architecture)

AlexNet consists of 5 CONV (Convolutional) layers and 3 FC (Fully Connected) layers. The activation used for this model is Rectified Linear Unit (ReLu).

Model Summary:

```
Total params: 5,334,787
Trainable params: 5,328,963
Non-trainable params: 5,824
```

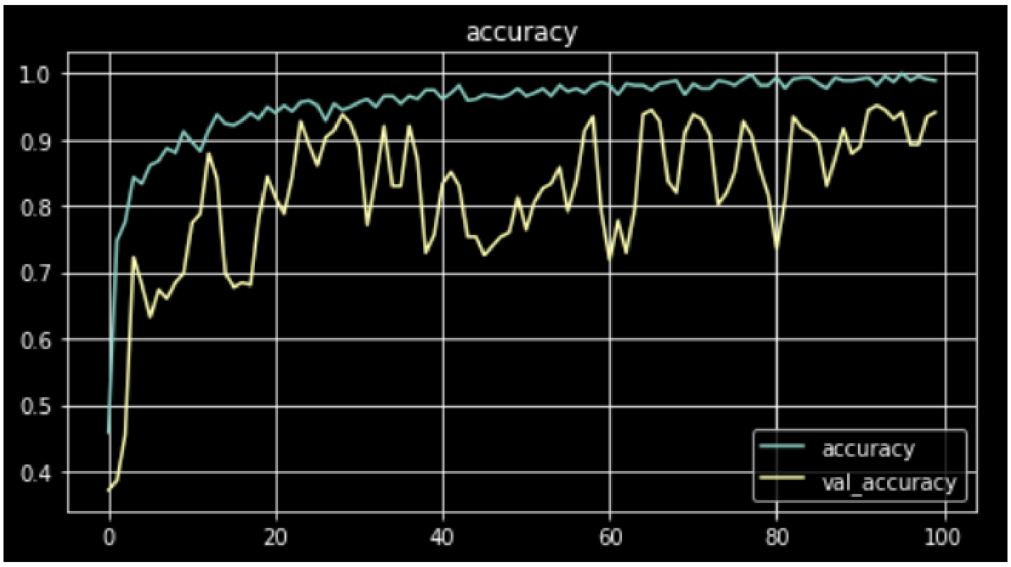
```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99        73
           1       0.84      0.98      0.91        49
           2       0.98      0.83      0.90        58

    accuracy                           0.94       180
   macro avg       0.94      0.94      0.93       180
weighted avg       0.94      0.94      0.94       180
```

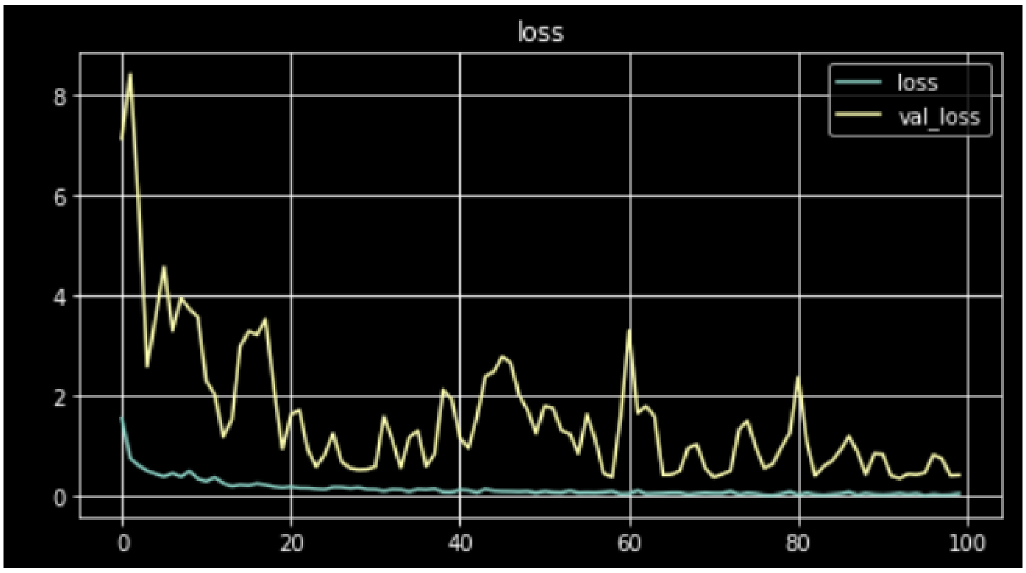


This model gave an overall accuracy of 94% on the test dataset and 169 images were correctly classified out of 180 images in the test dataset.

The training data was further divided into training and validation datasets.

Training accuracy and validation accuracy and Training loss and Validation loss was computed for 100 epochs with steps per epoch set to default.





Conclusion

As we can see from the results CNN using AlexNet architecture gives us the best accuracy on the Test set as compared to other models. We are using a 96x96x3 image feature vector for all the models.

Rank	Model Implemented	Accuracy
1	CNN using AlexNet Architecture	94%
2	CNN using VGGNet Architecture	89%
3	Decision Tree	75%
4	Simple Neural Network	37%

Future Scope

There are a number of games that have started to use image recognition and image processing to provide its users with more real experience. There is a trend where people have started to collect their memories in the visual form.

There are apps that help users to get some background information on the image that they have provided using computer vision.
Apart from advances in technology image recognition always faces an issue with deformation. Deformation does not change the objects.

When the model is trained it assumes a particular object or person to have a specific shape or face which is not ideal in the real-world. There are also issues of occlusion which is a barrier to our system in retrieving the amount the information it actually should. Better algorithms and datasets can tackle these problems to an extent.

Gantt Chart

	1 March to 15 March	15 March to 1 April	1 April to 10 April	11 April to 1 May	1 May to 18 May	19 May to 31 May
Topic Selection	🟦					
Gathering required knowledge		🟫	🟫	🟫		
Gathering data/Data preprocessing			🟥			
Implementation of various models				🟩	🟩	
Training and Testing					▨	▨
Evaluation						🟦
Report Formulation						🟩

BIBLIOGRAPHY

[1] Ned Horning, ‘Random Forests: An algorithm for image classification and generation of continuous field data sets’

[2] Elith, J., Leathwick, J.R. and Hastie, T., 2008, ‘A working guide to boosted regression trees. Journal of Animal Ecology’

[3] Oren Boiman, Eli Shechtman, Michal Irani, ‘In defense of Nearest Neighbor based image classification’, IEEE Conference on Computer Vision and Pattern Recognition

[4] M. Jain, P.S. Tomar, 2013, ‘Review of Image Classification Methods and Techniques’, International Journal of Engineering Research & Technology (IJERT)

[5] S.S. Nath, G. Mishra, J. Kar, S. Chakraborty, N. Dey, 2013, ‘A survey of image classification methods and techniques’, International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)