Exercise 1:

Course name : .Build a Convolution Neural Network for Image Recognition.

Go through the modules of the course mentioned and answer the self-assessment questions given in the link below at the end of the course.

Representation of an image

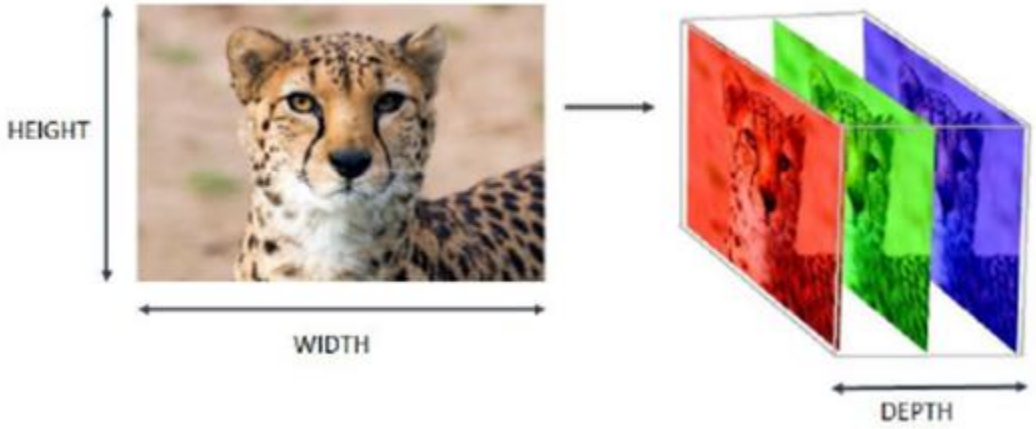
An image is an unstructured data represented by pixels.

To store an image on a computer, the image is broken down into tiny elements called pixels (short for picture element). A pixel is a number and it represents one colour. Usually, an 8 bit number is used to represent a pixel. Hence, each pixel of a black and white image can have 28 = 256 different shades of gray. It can also be represented as a decimal number from 0 to 255. Zero stands for black and 255 for white.

An image with a resolution of 1024 by 798 pixels has 1024 x 798 total number of pixels (817,152 pixels)

A pixel in a coloured image is an array of 3 numbers. They represent the red, green and blue (RGB) values. For an 8 bit representation, each colour can take 256 shades. The final colour of the pixel is given by the combination of the 3 values. Hence each pixel can have 256 x 256 x 256 values of nearly 17 million different colours.

So, the computer sees an image as an array of numbers. It represents an image with a colour depth that represents the number of bits that represent a pixel (in the examples mentioned above, we considered it to be 8) and the image resolution: the height and width of an image in pixels. As a colour image has 3 matrices, one for each colour, hence it is represented by its height, width and depth. It is sometimes called a 3D volume.



This figure shows the RGB decomposition of a colour image.

Feature

Can we teach a machine to identify objects the way humans do? Assume that we have a large number of images of cats. Is there a way to train a machine to identify cats with these given images? Can the machine identify a new cat that was not in the training dataset?

To answer these questions, we have to understand how we identify objects. We identify objects by their

features. Lets understand what is a feature.

Consider the two images a caracal and a cheetah:

- Caracal has pointy ears. Cheetah has round ears.
- Cheetah is spotted. Caracal is not spotted.
- Cheetah has dark black lines on its face.

Such differentiating data is called feature.

The abstraction of such features (a straight line, a curve , etc.) is shown in the diagram. A feature is a small image, usually of size 2 x 2 or 3 x 3 pixel dimension. These features are also called filters, kernels or weights in Deep learning.

0.5	0.2	0.1
0.2	0.5	0.2
0.1	0.2	0.5

But, how does a machine find out what features an image has? It learns features using the mathematical operation called convolution. Convolution means the similarity of a feature with a given image. We select a random feature and convolve (roll) it on the surface of the image. The learning process progressively modifies the random feature such that it has a good match with the given image. Multiple features are deployed, each starting at a different random state. Each of them specializes to learn different features of the image. The process of learning features from an image is called feature extraction.

Once we have extracted the features from an object, we can use them to identify the object.

Learnable filters

Convolutional neural network has two parts. The first part identifies the features in the image. The features learnt in the first phase is fed to the second part, a fully connected neural network. This part does the classification.

The question is how do we select a filter such that its overlap with the image is maximum? Convolutional neural networks uses learnable filters. We randomly select such filters and then learn the required filters by training, just as in ANN, where we begin with random weights and then learn the best weights through training. This is why filters are also called weights.

Sample filter -

0.5	0.2	0.1
0.2	0.5	0.2
0.1	0.2	0.5

Padding

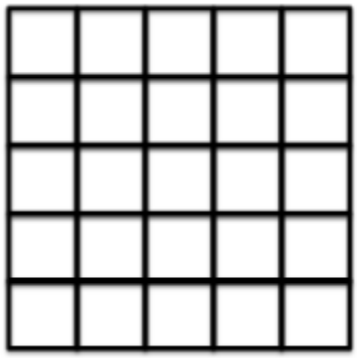
We observed that everytime we use the filter and scan the image for similarity, the size of the resulting image becomes smaller and smaller. We do not want that, because we want to preserve the original size of the image to extract some low level of features. The second issue is that the pixels in the periphery get covered only once but the ones in the middle get covered multiple times.

Thus, there are two issues with convolution -

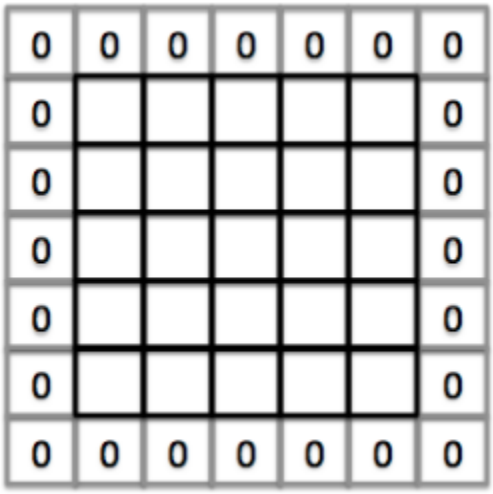
- Shrinking feature map and
- Losing information on the periphery of the image.

These two issues are solved by padding. In padding, the image is added with extra pixels at the boundary before the convolution. Zero padding means the value of these added pixels is zero. Sometimes the value of the edge pixel is added in the padded cell.

Add image



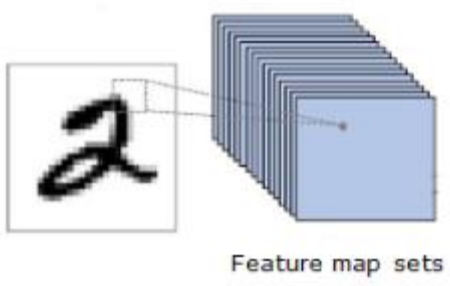
If zero padding = 1 , one pixel will be added around the original image with value = 0.



Feature map sets

As discussed, the feature map is a measure of similarity of the filter with the image. An image can have multiple distinguising features. To correctly identify the image we need to learn all of them. Hence we deploy a number of learnable filters. Each specializes to recognize one feature. And each filter results in a feature map. Hence we get a set for feature maps.

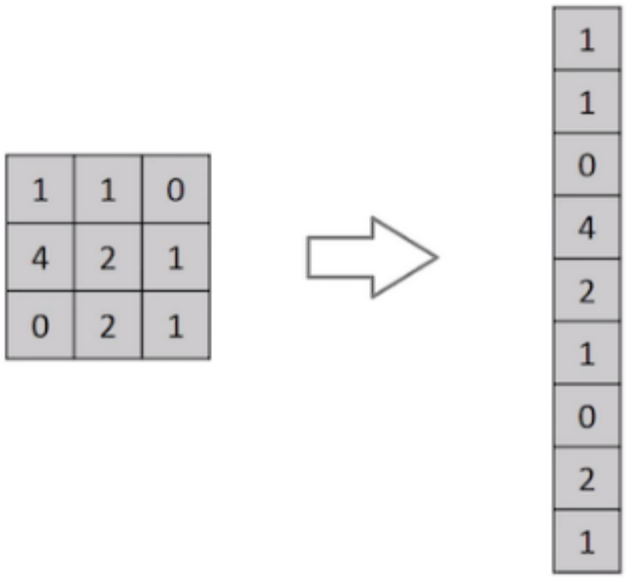
Part drawing showing feature map sets



Flattening

How are the pooled layer sets fed to a fully connected layer? Each element from the pooled set is still a matrix. However, a fully connected network does not accept a matrix but only scalars.

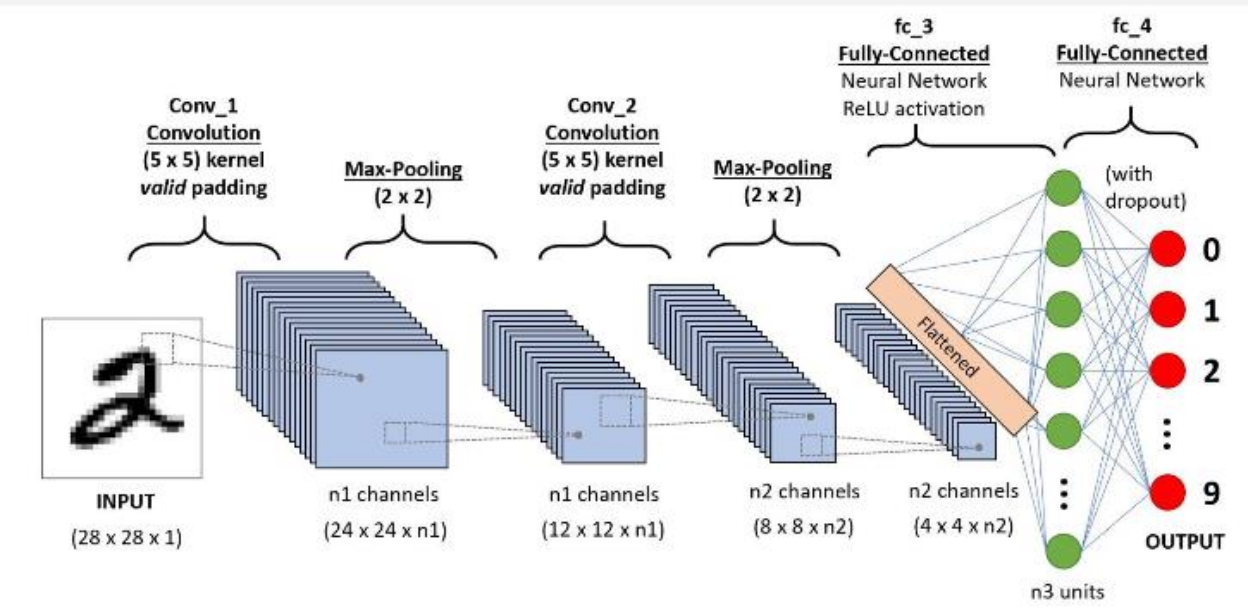
So we flatten the matrix; See the image below.\



Flattening of a 3x3 image matrix into a 9x1 vector

Neural networks can very easily overfit -- they learn very well on training set but perform poorly on validation set.

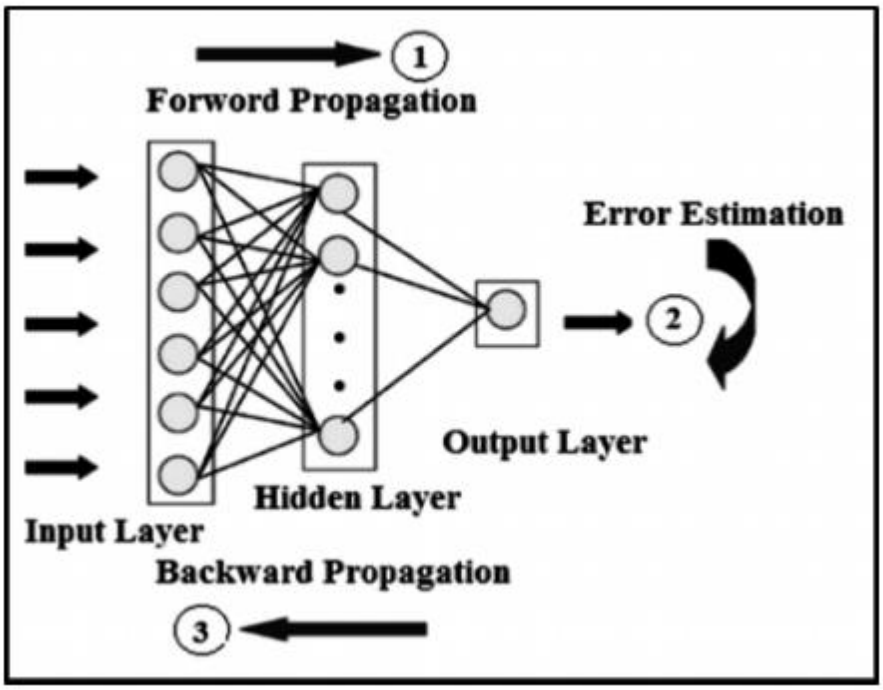
To get around this problem, the network is trained by randomly dropping a few weights (or filters) during each pass. This ensures that the filters do not “over specialize” in recognizing features and can work when a few features are marked off.



A CNN sequence to classify handwritten digits

Training

The fully connected network at the end is trained using Back-propagation and gradient descent discussed earlier. It is during this process that the distinguishing features of the images are learnt by the filters.


1
Forword Propagation
Error Estimation
2
Output Layer
Hidden Layer
Input Layer
Backward Propagation
3

Exercise 2

Module name: Understanding and Using ANN : Identifying age group of an actor Exercise : Design Artificial Neural Networks for Identifying and Classifying an actor using Kaggle Dataset.

We are going to take a scenario of identifying the age group of various movie characters just by considering their facial attributes and in turn will try to understand the implementation of deep neural networks in python.

We will use the Indian Movie Face Database (IMFDB)* created by Shankar Setty et.al. as a benchmark for facial recognition with wide variation. The database consists of thousands of images of 50+ actors taken from more than 100 videos. Since the database has been created manually by cropping the images from the video, there's high variability in terms of pose, expression, illumination, resolution, etc. The original database provides many attributes including:

- Expressions: Anger, Happiness, Sadness, Surprise, Fear, Disgust
- Illumination: Bad, Medium, High
- Pose: Frontal, Left, Right, Up, Down
- Occlusion: Glasses, Beard, Ornaments, Hair, Hand, None, Others
- Age: Child, Young, Middle and Old
- Makeup: Partial makeup, Over-makeup
- Gender: Male, Female

In this scenario, we will use a cleaned and formatted data set with 26742 images split as 19906 train images and 6636 test images respectively. The target here is to use the images and predict the age of the actor/actress within the available classes i.e. young, middle and old making it a multi-class classification problem.

You can download the train and test data sets. In each directory, you will find a folder consisting of images along with an excel file which has two columns, ID and Class. The ID column consists of image names like **352.jpg** and Class column holds the respective image character's age like **Old**.

**Data set reference:*

Shankar Setty, Moula Husain, Parisa Beham, Jyothi Gudavalli, Menaka Kandasamy, Radhesyam Vaddi, Vidyagouri Hemadri, J C Karure, Raja Raju, Rajan, Vijay Kumar and C V Jawahar. "Indian Movie Face Database: A Benchmark for Face Recognition Under Wide Variations"

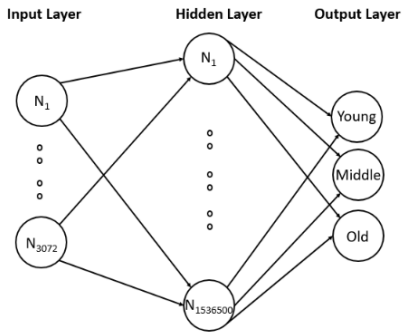
National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG), 2013.

In the data set, images belong to different sizes and hence cannot be fed directly to the input layer. An input layer has a defined number of nodes which are not changed during the process of implementation of the network. Therefore, before proceeding with building the network, we need to ensure that all the images have equal width and height.

For our problem statement, we will resize all the images to 32 x 32 shape. All the images have red, blue and green color components, therefore, the final shape becomes 32 x 32 x 3 giving us a total of 3072 nodes for the input layer.

Next, we will choose one hidden layer to start with along with 500 nodes making a total of 1536500 (3072 x 500) connections between the input and the hidden layer. We will use the ReLU activation function in this layer.

Next, we have the output layer having only three classes and hence three nodes making a total of 1503 (500 x 3) connections between hidden and output layer. In this layer, we will use the Softmax activation function.



While building the model, we will split the training data into training and validation data set and will find the loss and accuracy for both the data sets.

Now that we have defined the structure of our neural network, let us start implementing the code in Keras.

Before we proceed, let us take a look at the current challenges of the given data set:

- Variations in shape: For example, one image has a shape of (66, 46) whereas another has a shape of (102, 87), there is no consistency
- Multiple viewpoints/ profiles: faces with different viewpoints/profiles may exist
- Brightness and contrast: It varies across images and can introduce discrepancy in few cases
- Quality: Some images are found to be too pixelated

In this resource, we are going to handle the above challenges by performing image preprocessing, as well as implement a basic neural network.

Let us first import all the necessary libraries and modules which will be used throughout the code:

```
# Importing necessary libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.preprocessing import LabelEncoder
from tensorflow.python.keras import utils
from keras.models import Sequential
from keras.layers import Dense, Flatten, InputLayer
import keras
import imageio # To read images
from PIL import Image # For image resizing
```

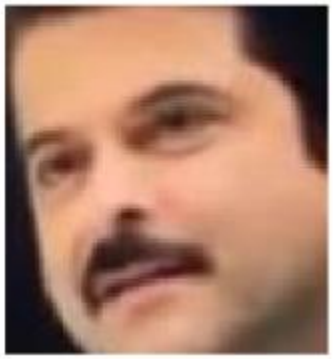
Next, let us read the train and test data sets into separate pandas DataFrames as shown below:

```
# Reading the data
train = pd.read_csv('age_detection_train/train.csv')
test = pd.read_csv('age_detection_test/test.csv')
```

Once, both the data sets are read successfully, we can display any random movie character along with their age group to verify the ID against the Class value, as shown below:

```
np.random.seed(10)
idx = np.random.choice(train.index)
img_name = train.ID[idx]
img = imageio.imread(os.path.join('age_detection_train/Train', img_name))

print('Age group:', train.Class[idx])
plt.imshow(img)
plt.axis('off')
plt.show()
```

```
Age group: MIDDLE
```



Next, we can start transforming the data sets to a one-dimensional array after reshaping all the images to a size of 32 x 32 x 3.

Let us reshape and transform the training data first, as shown below:

```
temp = []
for img_name in train.ID:
    img_path = os.path.join('age_detection_train/Train', img_name)
    img = imageio.imread(img_path)
    img = np.array(Image.fromarray(img).resize((32, 32))).astype('float32')
    temp.append(img)

train_x = np.stack(temp)
```

Next, let us reshape and transform the testing data, as shown below:

```
temp = []
for img_name in test.ID:
    img_path = os.path.join('age_detection_test/Test', img_name)
    img = imageio.imread(img_path)
    img = np.array(Image.fromarray(img).resize((32, 32))).astype('float32')
    temp.append(img)

test_x = np.stack(temp)
```

Next, let us normalize the values in both the data sets to feed it to the network. To normalize, we can divide each value by 255 as the image values lie in the range of 0-255.

```
# Normalizing the images
train_x = train_x / 255.
test_x = test_x / 255.
```

and label encodes the output classes to numerics:

```
# Encoding the categorical variable to numeric
lb = LabelEncoder()
train_y = lb.fit_transform(train.Class)
train_y = utils.np_utils.to_categorical(train_y)
```

Next, let us specify the network parameters to be used, as shown below:

```
# Specifying all the parameters we will be using in our network
input_num_units = (32, 32, 3)
hidden_num_units = 500
output_num_units = 3

epochs = 5
batch_size = 128
```

Next, let us define a network with one input layer, one hidden layer, and one output layer, as shown below:

```
model = Sequential([
  InputLayer(input_shape=input_num_units),
  Flatten(),
  Dense(units=hidden_num_units, activation='relu'),
  Dense(units=output_num_units, activation='softmax'),
])
```

We can also use summary() method to visualize the connections between each layer, as shown below:

```
# Printing model summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               1536500
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 1503
=================================================================
Total params: 1,538,003
Trainable params: 1,538,003
Non-trainable params: 0
_________________________________________________________________
```

Next, let us compile our network with SGD optimizer and use accuracy as a metric:

```
# Compiling and Training Network
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
```

Now, let us build the model, using the fit() method:

```
model.fit(train_x, train_y, batch_size=batch_size, epochs=epochs, verbose=1)
```

This results in the following log:

```
Epoch 1/5
19906/19906 [==============================] - 14s 717us/step - loss: 0.8961 - acc: 0.5817
Epoch 2/5
19906/19906 [==============================] - 13s 635us/step - loss: 0.8492 - acc: 0.6008
Epoch 3/5
19906/19906 [==============================] - 13s 630us/step - loss: 0.8316 - acc: 0.6125
Epoch 4/5
19906/19906 [==============================] - 13s 674us/step - loss: 0.8170 - acc: 0.6206
Epoch 5/5
19906/19906 [==============================] - 16s 820us/step - loss: 0.8086 - acc: 0.6278
```

We can observe in the above results, that the final accuracy is 62.78%. However, it is recommended that we use 20% to 30% of our training data as a validation data set to observe how the model works on unseen data.

The following code considers 20 percent of the training data as validation data set:

This results in the following log:

```
Train on 15924 samples, validate on 3982 samples
Epoch 1/5
15924/15924 [==============================] - 11s 669us/step - loss: 0.8014 - acc: 0.6301 - val_loss: 0.7952 - val_acc: 0.6369
Epoch 2/5
15924/15924 [==============================] - 11s 670us/step - loss: 0.7949 - acc: 0.6355 - val_loss: 0.7872 - val_acc: 0.6477
Epoch 3/5
15924/15924 [==============================] - 11s 681us/step - loss: 0.7920 - acc: 0.6343 - val_loss: 0.7879 - val_acc: 0.6464
Epoch 4/5
15924/15924 [==============================] - 11s 709us/step - loss: 0.7867 - acc: 0.6411 - val_loss: 0.7874 - val_acc: 0.6484
Epoch 5/5
15924/15924 [==============================] - 12s 765us/step - loss: 0.7824 - acc: 0.6451 - val_loss: 0.7965 - val_acc: 0.6364
```

We can observe, that the training accuracy is 64.51% and validation accuracy is 63.64%. Since both the results are quite close we can conclude that there's no overfitting in the model. However, the accuracy itself is too low. The accuracy can be increased by overcoming the previously stated challenges and some difference can even be observed by tuning the hyper-parameters which we are going to observe in the next resource.

With our baseline neural network, we can now predict the age group of test data and save the results in an output file, as shown below:

```
# Predicting and importing the result in a csv file
pred = model.predict_classes(test_x)
pred = lb.inverse_transform(pred)

test['Class'] = pred
test.to_csv('out.csv', index=False)
```

We can also perform the visual inspection on any random image, as shown below:

```
# Visual Inspection of predictions
idx = 2481
img_name = test.ID[idx]

img = imageio.imread(os.path.join('age_detection_test/Test', img_name))
plt.imshow(np.array(Image.fromarray(img).resize((128, 128))))
pred = model.predict_classes(test_x)
print('Original:', train.Class[idx], 'Predicted:', lb.inverse_transform(pred[idx]))
```

```
Original: MIDDLE Predicted: YOUNG
```

The network misidentified the current image from the middle age group as young. This could be due to the 64% accuracy of the model. Therefore, let us learn about hyper-parameter tuning and try to improve the results.

11-8-23

Exercise 3: Module name : Understanding and Using CNN : Image recognition Exercise: Design a CNN for Image Recognition which includes hyperparameter tuning.

https://infyspringboard.onwingspan.com/web/en/viewer/web-module/lex_auth_012785694443167744910_shared?collectionId=lex_auth_0127481425493114885 9_shared&collectionType=Course

Sol:

In this module, we will a small dataset, CIFAR-10 which can be downloaded from here(https://infyspringboard.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012782825259556864334_shared/web-hosted/assets/cifar10.zip) along with its labels((https://infyspringboard.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012782825259556864334_shared/web-hosted/assets/cifar10Labels.csv). This dataset consists of 60,000 images of shape 32x32x3 each categorized into one of the 10 categories (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). Further, each class has 6000 images.

Few of the images from the dataset are shown below:

Problem Statement

Before we proceed with the learning of a new architecture of the neural network, try to solve the multi-class classification problem using the neural network architecture we used in the previous module on the given CIFAR-10 dataset.

Convolution Neural Network is similar to multi-layer perceptron having made up of neurons with learnable parameters computing the loss function in the last layer. However, CNN architecture primarily made an explicit assumption that inputs are images but in recent times their applications are seen in the areas of text, speech and time series forecasting.

Let's understand the performance difference between the two using a small example. Consider an image with a dimension of 16x16x3.

Implementation using Multi-Layer Perceptron

For a classical neural network, all joined connections in its first hidden layer will result in 768 weights (16*16*3). However, with an increased number of neurons and image size (say 300x300x3 with 270000 weights) the structure with a vast number of parameters quickly leads to overfitting. Also, MLP are not known to preserve the spatial features within the images.

Implementation using Convolutional Neural Network

ConvNet pre-assumes the input to be images and hence arranges its neurons in a 3D volume structure: width, height, and depth.

So, an image of shape 16x16x3 will form a volume of similar shape i.e. 16x16x3 with neurons connecting only to a slice of preceding layer neurons. The resulting output has a dimension of 1x1xh where h represents the target labels.

Let us learn what are the components of a CNN and how does it work.

A basic CNN architecture consists of the following layers:

```
1. Input -> Convolutional -> Pooling -> Output
```

Among the Convolutional and Pooling layers, both can be repeated as many times as you like.

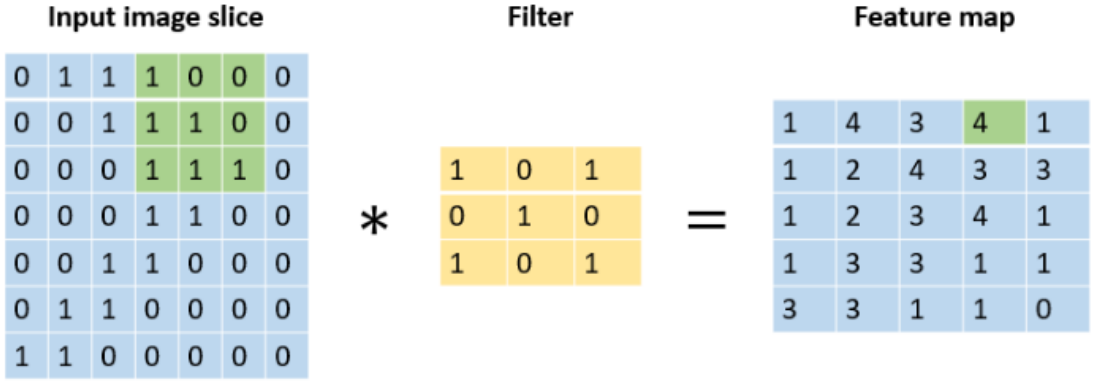
Input layer

Input layer having two dimensions works as a storage unit for holding raw image data with the preferable size in a multiple of 16, 32, 64, 224, 256, etc. for both height and width for the efficient use of memory fields.

Convolutional Layer

Once the image is loaded in the input layer, the succeeding hidden layers connect back to their preceding layers only on a local region known as the receptive field. This follows a convolution operation which is a combined integration between two functions. It depicts how one function modifies the shape of others.

Since images are represented as a form of a multi-dimensional matrix in the system, therefore, consider the below picture to learn how convolution takes place on a channel (RGB) of an image:



Here, at a time a certain image slice is chosen. The filter slides over the input volume convolving with the local region at a time. The number of pixels to jump for next convolution is governed by the stride. Stride with value 1 makes the filter slide over the input volume with 1 pixel, a value of 2 makes the filter slide with 2 pixels and so on. Larger the stride, lesser the spatial extent of output volume. In this illustration, the stride is taken as 1.

Sometimes, the filter size along with stride value doesn't fit the shape of the image, therefore, in such cases, extra padding of zero is preferred. Zero-padding across the input volume border provides us two benefits: first, it helps retain the border information of the image. Since with each convolution, the size of the image keeps reducing and hence without padding the border information may simply be removed. Second, it helps keep the shape of input and output volume equal. Since filter convolution may change the output volume spatial extent, hence padding helps to avoid such cases.

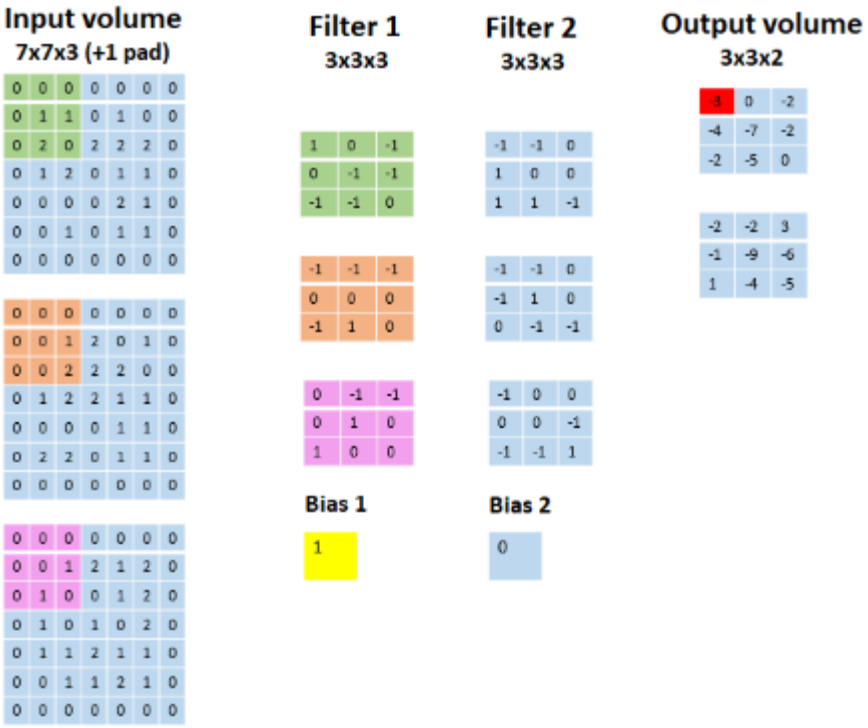
During the process, you can choose the number of filters where each one of them locates distinct features likes edges, blobs, etc. Distinct filters are indeed necessary to gain distinct features in the process. For instance, with distinct filters, we can attain features including a sharp image, blurred image, image edges, etc.

To wrap up this idea in one single example, consider the given image where we choose a stride value of two, zero-padding value as one along with two filters. So, to arrive at the output -3 (colored in red), you need to get the sum of the pointwise multiplication of the similar colored matrixes.

For instance,

$$\text{Output volume}_{\text{Red}} = \text{Input volume}_{\text{Green}} * \text{Filter } 1_{\text{Green}} + \text{Input volume}_{\text{Orange}} * \text{Filter } 1_{\text{Orange}} + \text{Input volume}_{\text{Pink}} * \text{Filter } 1_{\text{Pink}} + \text{Bias } 1$$

Similarly, you can proceed to find the values of other cells of the output matrixes. Note, the first output volume matrix is formed using Filter 1 and Bias 1 whereas the second output volume matrix is formed by Filter 2 and Bias 2.



This layer performs downsampling operation along the two dimensions (width and height), hence reducing the number of required parameters and thus reduced computation and a lesser chance of overfitting. It uses the MAX function and requires two hyperparameters the receptive field, and the stride rate. Padding is generally not used with pooling layer. Also, it doesn't introduce any new parameter as it works on a fixed function.

An alternative to pooling layer: Jost Tobias Springenberg et.al. in their paper "Striving for Simplicity: The All Convolutional Net" suggests that using a higher stride once and using only CONV layers can completely remove the need of having a pooling layer in the architecture.

Fully-Connected layer

Neurons in the Fully-Connected layer are connected to all the activations in previous layers as in ordinary neural networks. It uses the softmax activation function for classifying input images into various classes.

This page lists the conventions required in the process:

If you input a volume of size W1 * H1 * D1, it requires four hyperparameters:

1. Number of filters, K
2. Receptive field, F
3. The stride, S
4. The amount of zero-padding, P

Which produces a volume of size W2 * H2 * D2 where

- W2=(W1-F+2P)/(S+1)
- H2=(H1-F+2P)/(S+1)
- D2=K

Let us start by importing basic modules:

```
from matplotlib import pyplot as plt
%matplotlib inline
from sklearn.preprocessing import LabelEncoder
import keras
import pandas as pd
import numpy as np
from PIL import Image
import os
import warnings
warnings.filterwarnings('ignore')
```

Next, let us import the label file and view any random image along with its label:

```
labels = pd.read_csv('cifar10_Labels.csv', index_col=0)

# View an image
img_idx = 5
print(labels.label[img_idx])
Image.open('cifar10/'+str(img_idx)+'.png')
```

automobile

As we can observe the label is correct as per the image. Now, let us split the data into training and test, follow up with its transformation and normalization:

```
# Splitting data into Train and Test data
from sklearn.model_selection import train_test_split
y_train, y_test = train_test_split(labels.label, test_size=0.3, random_state=42)
train_idx, test_idx = y_train.index, y_test.index # Storing indexes for later use

# Reading images for training
temp = []
for img_idx in y_train.index:
    img_path = os.path.join('cifar10/', str(img_idx) + '.png')
    img = np.array(Image.open(img_path)).astype('float32')
    temp.append(img)
X_train = np.stack(temp)

# Reading images for testing
temp = []
for img_idx in y_test.index:
    img_path = os.path.join('cifar10/', str(img_idx) + '.png')
    img = np.array(Image.open(img_path)).astype('float32')
    temp.append(img)
X_test = np.stack(temp)

# Normalizing image data
X_train = X_train/255.
X_test = X_test/255.

```

The next preprocessing step it to label encode the image respective labels:

```
# One-hot encoding 10 output classes
encode_X = LabelEncoder()
encode_X_fit = encode_X.fit_transform(y_train)
y_train = keras.utils.np_utils.to_categorical(encode_X_fit)
```

Now, let us define the CNN network:

```
# Defining CNN network

num_classes = 10
model = keras.models.Sequential([
    # Adding first convolutional layer
    keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=1, padding='same', activation='relu',
                        kernel_regularizer=keras.regularizers.l2(0.001), input_shape=(32, 32, 3), name='Conv_1'),
    # Normalizing the parameters from last layer to speed up the performance (optional)
    keras.layers.BatchNormalization(name='BN_1'),
    # Adding first pooling layer
    keras.layers.MaxPool2D(pool_size=(2, 2), name='MaxPool_1'),
    # Adding second convolutional layer
    keras.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same', activation='relu',
                        kernel_regularizer=keras.regularizers.l2(0.001), name='Conv_2'),
    keras.layers.BatchNormalization(name='BN_2'),
    # Adding second pooling layer
    keras.layers.MaxPool2D(pool_size=(2, 2), name='MaxPool_2'),
    # Flattens the input
    keras.layers.Flatten(name='Flat'),
    # Fully-Connected layer
    keras.layers.Dense(num_classes, activation='softmax', name='pred_layer')
])
```

In the above model, we have used two convolution layers paired with max pool layers finally connecting with the Fully-Connected layer. We kept the "same" padding i.e., the output volume will have the same length as the original input. For no padding, you can choose the 'valid' argument. The stride is chosen as 1, a total number of 32 and 64 filters for each respective convolution layer and lastly keeping the kernel size as 3x3.

L2 regularization has been added to cost function via the convolution layers. Also, there's an addition of a new concept termed Batch Normalization. It is added due to the following reasons:

- Since we normalize the data before passing it to the input layer to increase the performance, therefore, we add this extra layer of normalization to normalize the values at the intermediate steps.
- It doesn't let the activation go higher or lower, therefore, you can use a higher learning rate to check for new feature possibilities.
- It works in complement to the dropout regularization. It has a slight regularization property as it adds some noise to each hidden layer activations and thus helps avoid overfitting.

Given below is the summary of the above network:

```
model.summary()
```

Layer (type)	Output Shape	Param #
Conv_1 (Conv2D)	(None, 32, 32, 32)	896
BN_1 (BatchNormalization)	(None, 32, 32, 32)	128
MaxPool_1 (MaxPooling2D)	(None, 16, 16, 32)	0
Conv_2 (Conv2D)	(None, 16, 16, 64)	18496
BN_2 (BatchNormalization)	(None, 16, 16, 64)	256
MaxPool_2 (MaxPooling2D)	(None, 8, 8, 64)	0
Flat (Flatten)	(None, 4096)	0
pred_layer (Dense)	(None, 10)	40970

Total params: 60,746
Trainable params: 60,554
Non-trainable params: 192

Let us now compile and train the model for just five epochs:

```
# Compiling the model
model.compile(loss='categorical_crossentropy',
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

cpfile = r'CIFAR10_checkpoint.hdf5' # Weights to be stored in HDF5 format
cb_checkpoint = keras.callbacks.ModelCheckpoint(cpfile, monitor='val_acc', verbose=1, save_best_only=True, mode='max')
epochs = 5
model.fit(X_train, y_train, epochs=epochs, validation_split=0.2, callbacks=[cb_checkpoint])
```

```
Train on 28000 samples, validate on 7000 samples
Epoch 1/5
28000/28000 [==============================] - 379s 14ms/step - loss: 1.7326 - acc: 0.4736 - val_loss: 1.7605 - val_acc: 0.4581

Epoch 00001: val_acc improved from -inf to 0.45814, saving model to CIFAR10_checkpoint.hdf5
Epoch 2/5
28000/28000 [==============================] - 351s 13ms/step - loss: 1.2379 - acc: 0.6019 - val_loss: 1.7944 - val_acc: 0.4793

Epoch 00002: val_acc improved from 0.45814 to 0.47929, saving model to CIFAR10_checkpoint.hdf5
Epoch 3/5
28000/28000 [==============================] - 353s 13ms/step - loss: 1.0563 - acc: 0.6564 - val_loss: 1.2411 - val_acc: 0.6099

Epoch 00003: val_acc improved from 0.47929 to 0.60986, saving model to CIFAR10_checkpoint.hdf5
Epoch 4/5
28000/28000 [==============================] - 379s 14ms/step - loss: 0.9514 - acc: 0.6893 - val_loss: 1.3454 - val_acc: 0.5811

Epoch 00004: val_acc did not improve
Epoch 5/5
28000/28000 [==============================] - 392s 14ms/step - loss: 0.8666 - acc: 0.7163 - val_loss: 1.2015 - val_acc: 0.6327

Epoch 00005: val_acc improved from 0.60986 to 0.63271, saving model to CIFAR10_checkpoint.hdf5
```

In every batch, the validation accuracy and training accuracy differs much showing a sign of overfitting. However, this model is just to provide you a basic instinct of developing a CNN model from scratch. You can further tune its hyperparameters to increase the performance.

Now, with the given model, let us now perform prediction:

```
# << DeprecationWarning: The truth value of an empty array is ambiguous >> can arise due to a NumPy version higher than 1.13.3.
# The issue will be updated in upcoming version.
pred = encode_X.inverse_transform(model.predict_classes(X_test[:10]))
act = y_test[:10]

res = pd.DataFrame([pred, act]).T
res.columns = ['predicted', 'actual']
res
```

	predicted	actual
0	truck	horse
1	ship	ship
2	ship	airplane
3	frog	frog
4	automobile	automobile
5	frog	frog
6	ship	ship
7	airplane	airplane
8	frog	frog
9	ship	dog

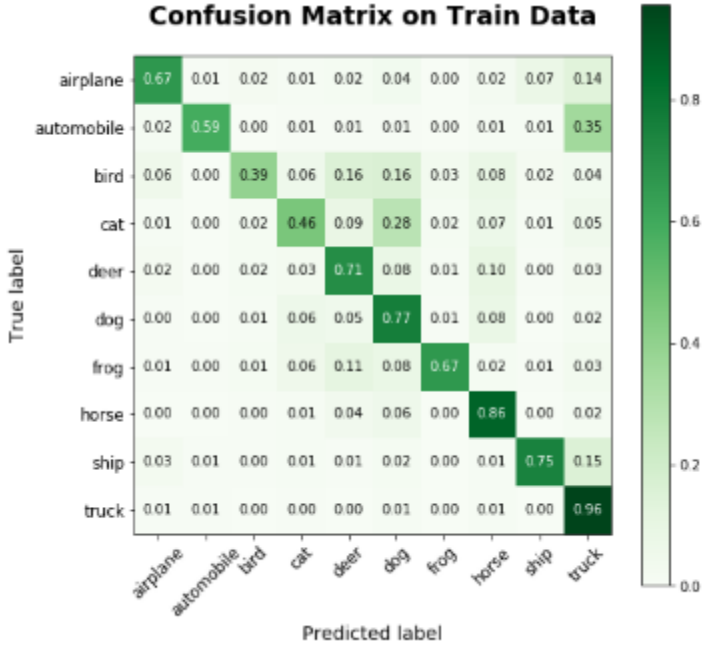
We can further proceed with train and test accuracy along with the confusion matrix to judge which class the model is predicting better:

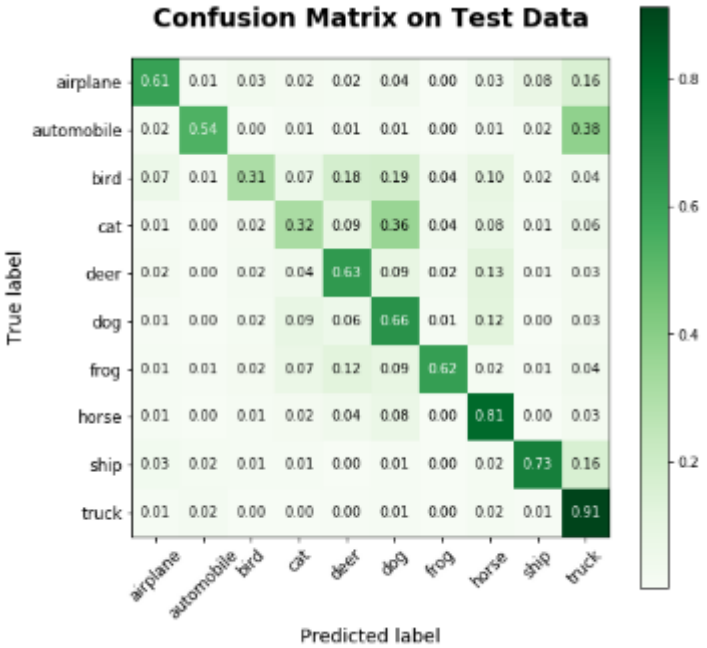
```
from mlxtend.evaluate import scoring

train_acc = scoring(encode_X.inverse_transform(model.predict_classes(X_train)),
                    encode_X.inverse_transform([np.argmax(x) for x in y_train]))
test_acc = scoring(encode_X.inverse_transform(model.predict_classes(X_test)), y_test)

print('Train accuracy: ', np.round(train_acc, 5))
print('Test accuracy: ', np.round(test_acc, 5))
```

```
Train accuracy:  0.3176
Test accuracy:  0.3874
```

```
from mlxtend.evaluate import confusion_matrix
from mlxtend.plotting import plot_confusion_matrix

def plot_cm(cm, text):
    class_names=['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
    plot_confusion_matrix(conf_mat=cm,
                          colorbar=True, figsize=(8, 8), cmap='Greens',
                          show_absolute=False, show_normed=True)
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45, fontsize=12)
    plt.yticks(tick_marks, class_names, fontsize=12)
    plt.xlabel('Predicted label', fontsize=14)
    plt.ylabel('True label', fontsize=14)
    plt.title(text, fontsize=19, weight='bold')
    plt.show()

# Train Accuracy
train_cm = confusion_matrix(y_target=encode_X.inverse_transform([np.argmax(x) for x in y_train]),
                          y_predicted=encode_X.inverse_transform(model.predict_classes(X_train)),
                          binary=False)
plot_cm(train_cm, 'Confusion Matrix on Train Data')
```

```
# Test Accuracy
test_cm = confusion_matrix(y_target=y_test,
                           y_predicted=encode_X.inverse_transform(model.predict_classes(X_test)),
                           binary=False)
plot_cm(test_cm, 'Confusion Matrix on Test Data')
```





The given model has quite a low accuracy on both train and test data, yet its prediction on categories like a truck, horse, and ship is quite remarkable

Exercise 4:

Predicting Sequential Data

Implement a Recurrence Neural Network for Predicting Sequential Data.

Forecasting is the process of predicting the future using current and previous data. The major challenge is understanding the patterns in the sequence of data and then using this pattern to analyse the future. If we were to hand-code the patterns, it would be tedious and changes for the next data. Deep Learning has proven to be better in understanding the patterns in both structured and unstructured data.
To understand the patterns in a long sequence of data, we need networks to analyse patterns across time. Recurrent Networks is the one usually used for learning such data. They are capable of understanding long and short term dependencies or temporal differences.

This post will show you how to implement a forecasting model using LSTM networks in **Keras** and with some cool visualizations. We'll be using the stock price of Google from yahoo finance but feel free to use any stock data that you like.

mplementation

Use colab to implement this code to make the visualizations easier, you can use your preferred method. We'll start off with importing necessary libraries:

```
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
from keras.preprocessing.sequence import TimeseriesGenerator
```

After you've downloaded your .csv file from yahoo finance or your source, load the data using pandas.

```
filename = "GOOG.csv"
df = pd.read_csv(filename)
print(df.info())
```

The info of data frame shows somewhat like this:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3797 entries, 0 to 3796
Data columns (total 7 columns):
Date         3797 non-null object
Open         3797 non-null float64
High         3797 non-null float64
Low          3797 non-null float64
Close        3797 non-null float64
Adj Close    3797 non-null float64
Volume       3797 non-null int64
dtypes: float64(5), int64(1), object(1)
memory usage: 207.7+ KB
```

For this tutorial, we require only Date and Close columns, everything else can be dropped.

```
1  df['Date'] = pd.to_datetime(df['Date'])
2  df.set_axis(df['Date'], inplace=True)
3  df.drop(columns=['Open', 'High', 'Low', 'Volume'], inplace=True)
```

Before we do the training and predictions, let's see how the data looks like. For all the visualizations, I'm using the Plotly python library.

Plotly:... cause its simply the best graphing library and it can produce some good looking graphs.

With plotly, we can define a trace and the layout and it does everything else.

The graph is oscillating from 2018 and the sequence is not smooth... Moving on.

Data Preprocessing

For our analysis, let train the model on the first 80% of data and test it on the remaining 20%.

```
close_data = df['Close'].values
close_data = close_data.reshape((-1,1))

split_percent = 0.80
split = int(split_percent*len(close_data))

close_train = close_data[:split]
close_test = close_data[split:]

date_train = df['Date'][:split]
date_test = df['Date'][split:]

print(len(close_train))
print(len(close_test))
```

Before we do the training, we need to do some major modification to our data. Remember, our data is still a sequence.. a list of numbers. The neural network is trained as a supervised model. Thus we need to convert the data from sequence to supervised data

Training a neural network of any machine learning model requires the data to be in {<features>,<target>} format. Similarly, we need to convert the given data into this format. Here, we introduce a concept of a look back.

Look back is nothing but the number of previous days' data to use, to predict the value for the next day. For example, let us say look back is 2; so in order to predict the stock price for tomorrow, we need the stock price of today and yesterday.

Coming back to the format, at a given day *x(t)*, the *features* are the values of *x(t-1), x(t-2), …., x(t-n)* where n is look back.

So if our data is like this,

```
[2, 3, 4, 5, 4, 6, 7, 6, 8, 9]
```

The required data format (n=3) would be this:

```
[2, 3, 4] -> [5]
[3, 4, 5] -> [4]
[4, 5, 4] -> [6]
[5, 4, 6] -> [7]
[4, 6, 7] -> [6]
[6, 7, 6] -> [8]
[7, 6, 8] -> [9]
```

There is a module in Keras that does exactly this: TimeseriesGenerator.

```
look_back = 15

train_generator = TimeseriesGenerator(close_train, close_train, length=look_back, batch_size=20)
test_generator = TimeseriesGenerator(close_test, close_test, length=look_back, batch_size=1)
```

Set look_back as 15, but you can play around with that value.

Neural Network

Now that our data is ready, we can move on to creating and training our network.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense

model = Sequential()
model.add(
    LSTM(10,
        activation='relu',
        input_shape=(look_back,1))
)
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

num_epochs = 25
model.fit_generator(train_generator, epochs=num_epochs, verbose=1)
```

A simple architecture of LSTM units trained using Adam optimizer and Mean Squared Loss function for 25 epochs. Note that instead of using model.fit(), we use model.fit_generator() because we have created a data generator.

To To know more about LSTM network, see this

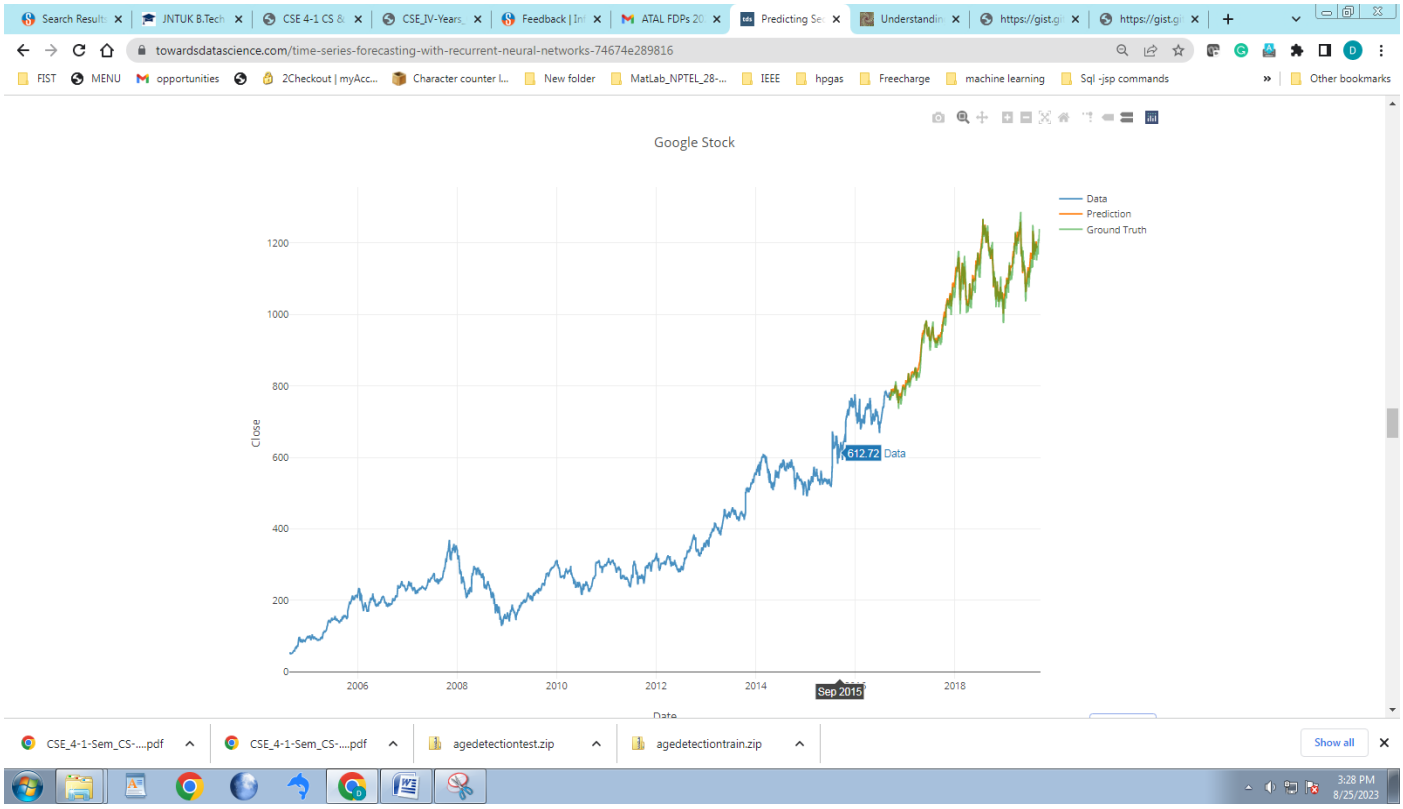
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

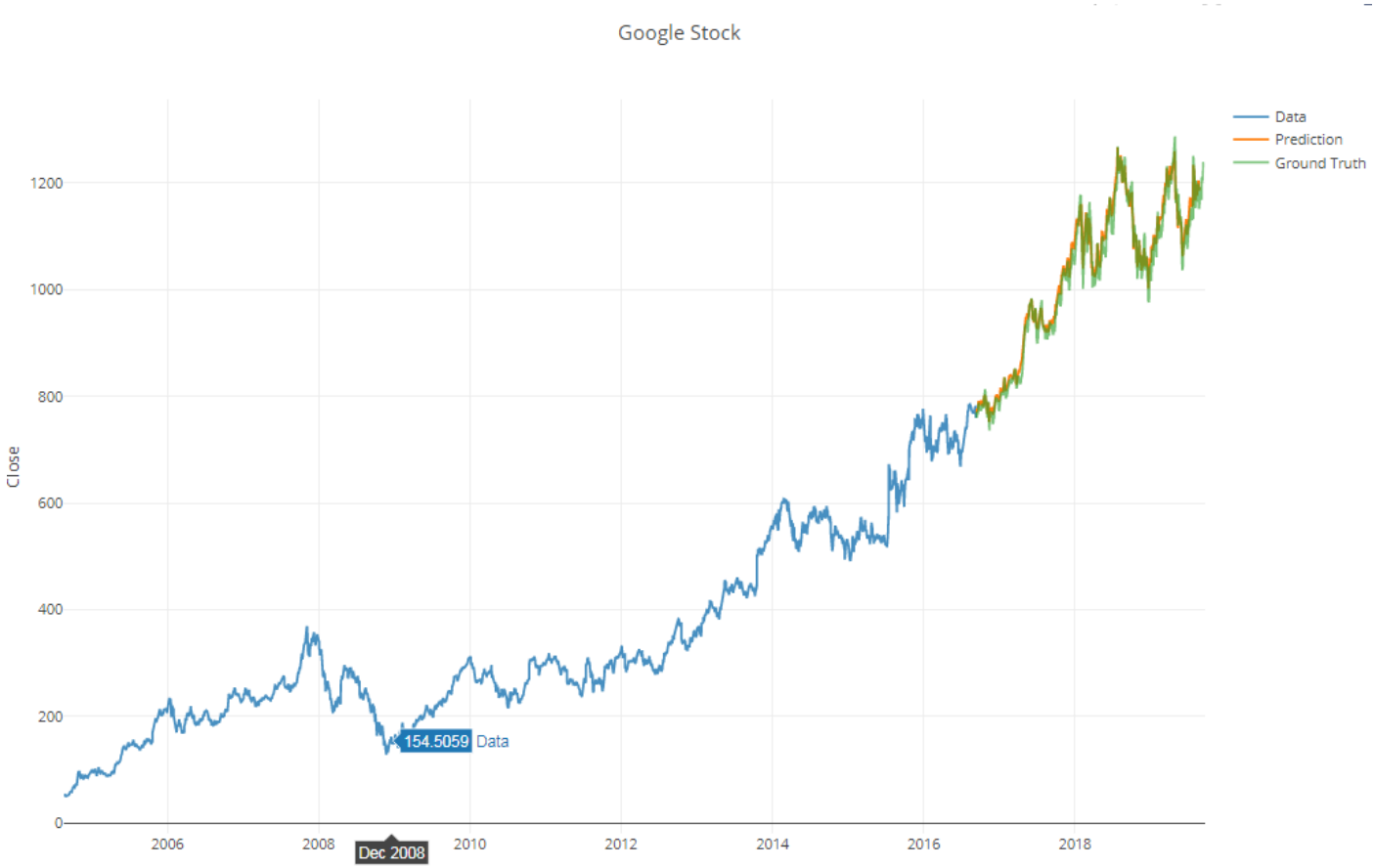
Prediction:

Now that we have completed training, let us see if the network performed well. We can test the model on testing data and see if the prediction and the actual values overlap.

```
prediction = model.predict_generator(test_generator)

close_train = close_train.reshape((-1))
close_test = close_test.reshape((-1))
prediction = prediction.reshape((-1))

trace1 = go.Scatter(
    x = date_train,
    y = close_train,
    mode = 'lines',
    name = 'Data'
)
trace2 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Prediction'
)
trace3 = go.Scatter(
    x = date_test,
    y = close_test,
    mode='lines',
    name = 'Ground Truth'
)
layout = go.Layout(
    title = "Google Stock",
    xaxis = {'title' : "Date"},
    yaxis = {'title' : "Close"}
)
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
fig.show()
```

Rather than computing loss between predicted and actual values, we can plot it.





From the graph, we can see that prediction and the actual value(ground truth) somewhat overlap. But if you zoom in, the fit is not perfect. We should expect this because it is inevitable as we are performing prediction.

Forecasting

Our testing shows the model is somewhat good. So we can move on to predicting the future or forecasting.

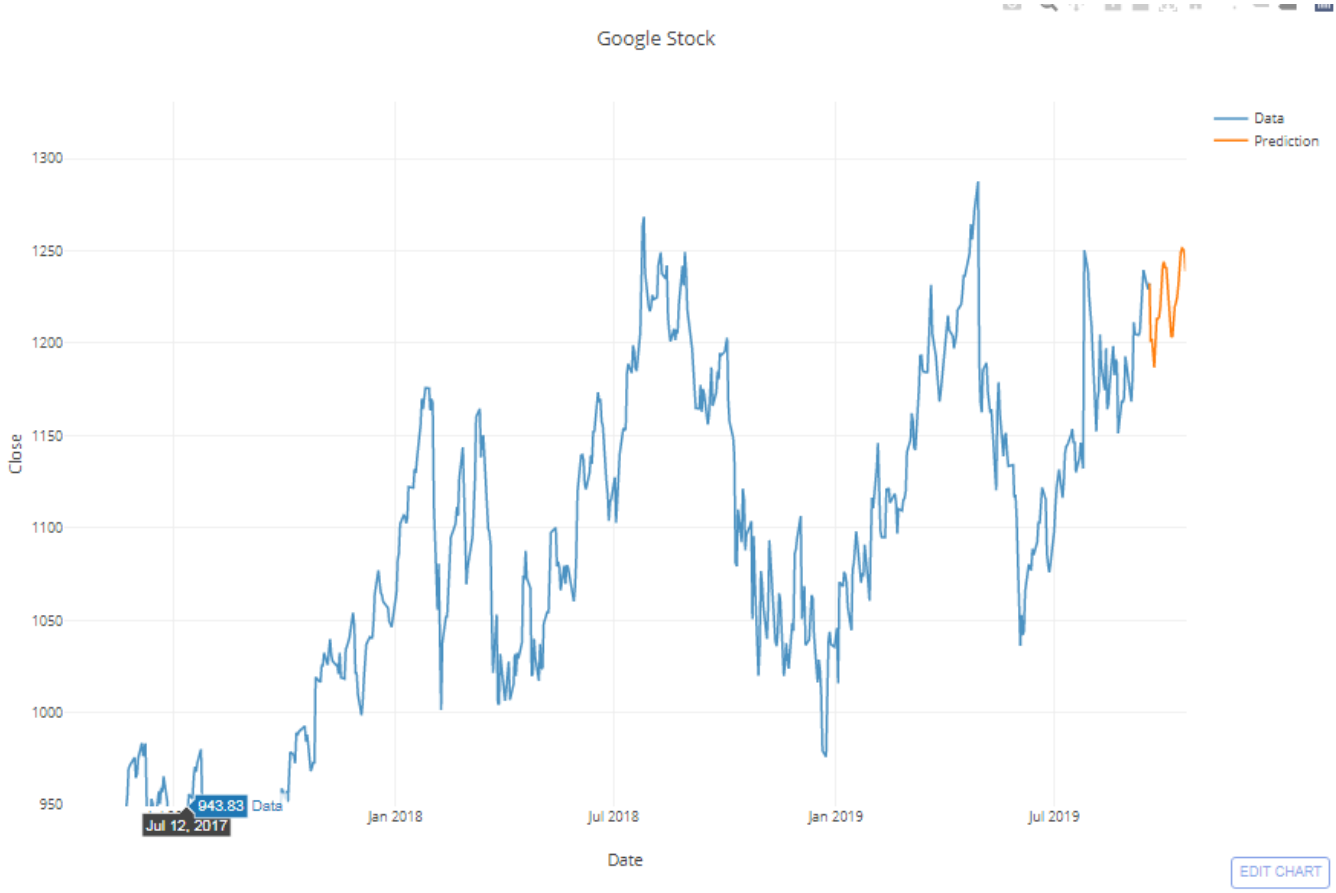
Foreshadowing: Since we are attempting to predict the future, there will be a great amount of uncertainty in the prediction.

Predicting the future is easy... To predict tomorrow's value, feed into the model the past *n*(look_back) days' values and we get tomorrow's value as output. To get the day after tomorrow's value, feed-in past n-1 days' values along with tomorrow's value and the model output day after tomorrow's value.

Forecasting for longer duration is not feasible. So, let's forecast a months stock price.

```
close_data = close_data.reshape((-1))

def predict(num_prediction, model):
    prediction_list = close_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

def predict_dates(num_prediction):
    last_date = df['Date'].values[-1]
    prediction_dates = pd.date_range(last_date, periods=num_prediction+1).tolist()
    return prediction_dates

num_prediction = 30
forecast = predict(num_prediction, model)
forecast_dates = predict_dates(num_prediction)
```

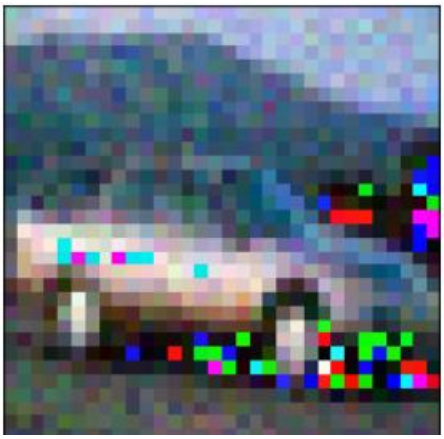
Now plotting the future values,

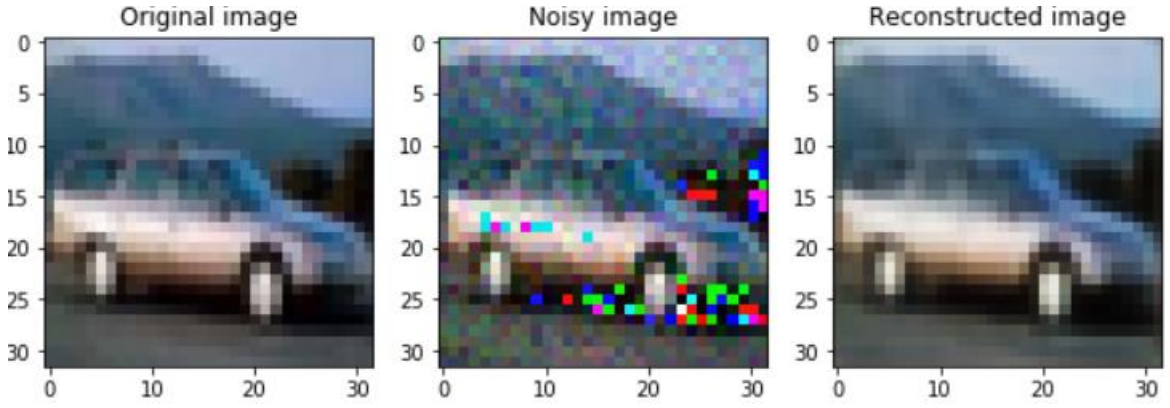


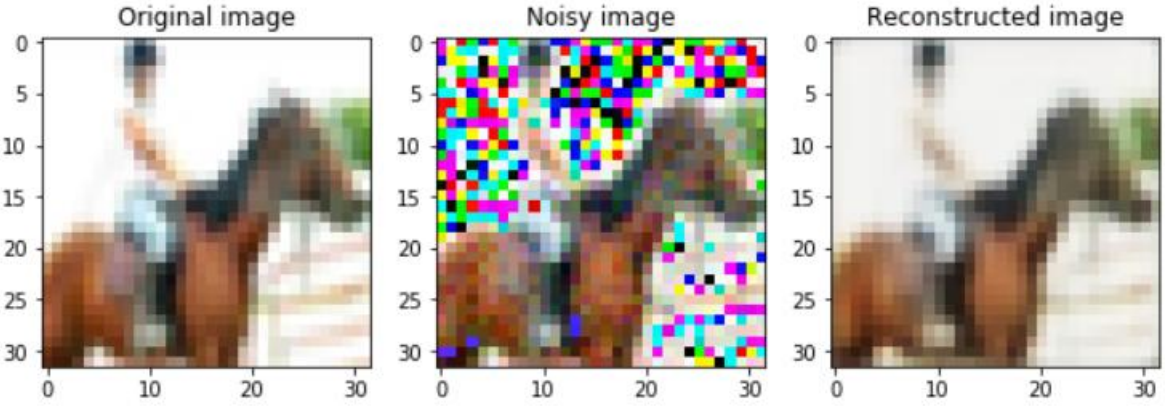
When predicting the future, there is a good possibility that model output is uncertain to a great extent. The model's output is fed back into it as input. This causes the model's noise and uncertainty to be repeated and amplified.

Conclusion

With this model, we have created a rudimentary model that is able to forecast to a certain extent.




Original image
Noisy image
Reconstructed image


Original image
Noisy image
Reconstructed image

EXERCISE-6

Advanced Deep Learning Architectures

Module Name: Advanced Deep Learning Architectures

Exercise: Implement Object Detection Using YOLO.

OBJECT DETECTION USING YOLO:

- YOLO – "You Only Look Once" are a series of end-to-end deep learning models designed for fast object detection, developed by Joseph Redmon, et al. in the 2015.
- Once the complexity of the image increases, it is not possible to have computational resources to build a Deep Learning model from scratch. So, predefined frameworks and pertained models come in handy.one such framework for object detection is YOLO.
- It's a supremely fast, state-of the art and accurate framework.
- It comes in different versions as shown in fig 5.
- YOLO is implemented on Darknet

How YOLO algorithm works

YOLO architecture is based on CNN and it can be customized according to user's requirement.

Step1: Read the input image

Let, C= number of classes. In the above example, C= 3 and the class label are C1=Chair, C2=laptop, C3 = Car

Step2: Divide the image into M×M grid of cells

For each grid cell Xij→Y, a label Y is calculated. The label Y is a N-dimensional vector, where, N depends on the number of classes. The description of each field is as shown in fig 7. For each grid cell Xij→Y, a label Y is calculated. The label Y is a N-dimensional vector, where, N depends on the number of classes. The description of each field is as shown in fig 7.

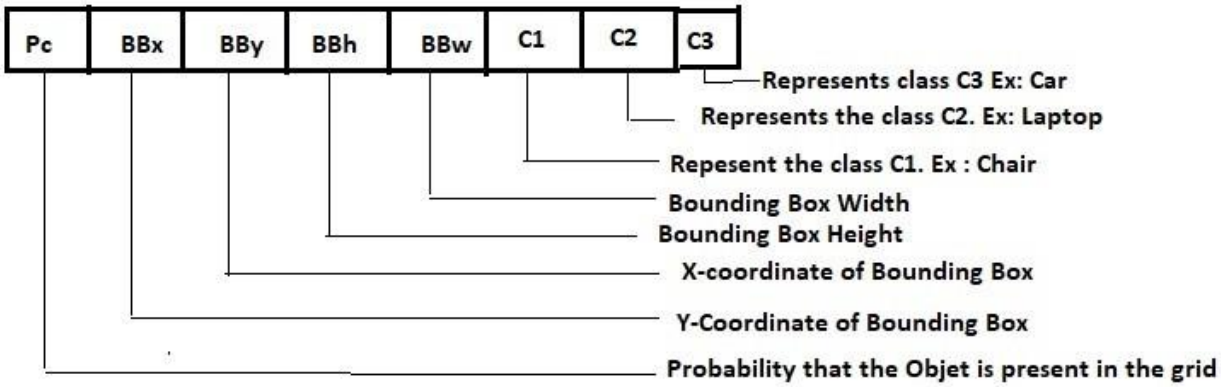


Fig 7. Vector representation of label Y

Step3: Apply Image classification and localization for each grid and predict the bounding box

- The (x, y) coordinates represent the center of the Bounding box relative to the grid cell location and (w, h) – dimension of Bounding box. Both are normalized between [0-1].
- IoU is applied to object detection. Intersection Over Union-IoU is an evaluation metric used to measure the accuracy of an object detector on a dataset.

Step4: Predict the class probabilities of the object

Class probabilities are predicted as P Class Object. This probability is conditioned on the grid cell containing one object.

The vector Y for first grid looks like this

0	BBx	BBy	BBh	BBw	0	0	0
Similarly, the vector Y for grid number 6 look like this:							
1	BBx	BBy	BBh	BBw	1	0	0

The output of this step results in 3x3x8 values i.e., for each grid 8-dimensional vector will be computed.

- In real time scenario the number of grids can be large number like 13x13 and accordingly Y vector varies.

Step5: Train the CNN

The last step is training the Convolutional Neural Network. The normal architecture of CNN is employed with convolutional layer and maxpooling.

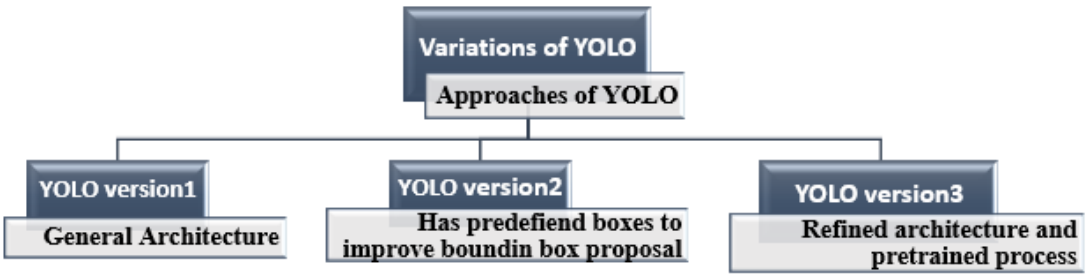


Fig 8. Versions of YOLO

What is Darknet?

- Darknet is an open-source framework that supports Object Detection and Image Classification tasks in the form of Convolutional Neural Networks.
- It is open source and written in C/CUDA
- It is used as the framework for training YOLO, i.e., it sets the architecture of the network
- Darknet is mainly used to implement YOLO algorithm
- The darknet is the executable code.
- This executable code can directly perform object detection in an image, video, camera, and network video stream.

Installation of darknet:

Rule to follow for the successful installation of Darknet:

- Applications should be installed in the correct order for the successful creation of the darknet framework.
- Darknet can be installed with any of the following two optional dependencies namely:

1. In CPU environment using OpenCV (original Darknet Framework, set the GPU flag in Make file when installing darknet to GPU=0.)
2. GPU environment for faster training

1. Steps to install darknet YOLO in CPU execution using OpenCV:

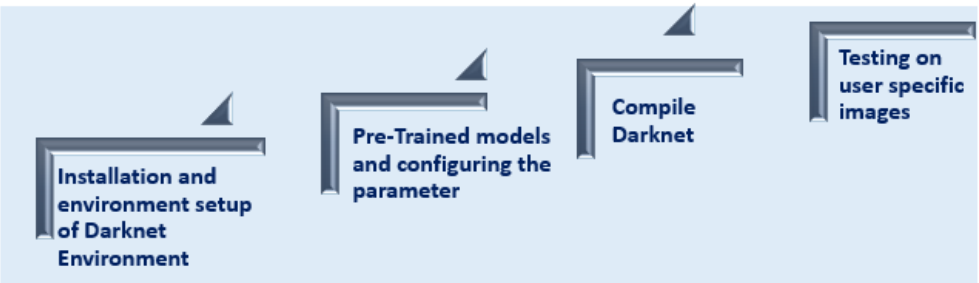
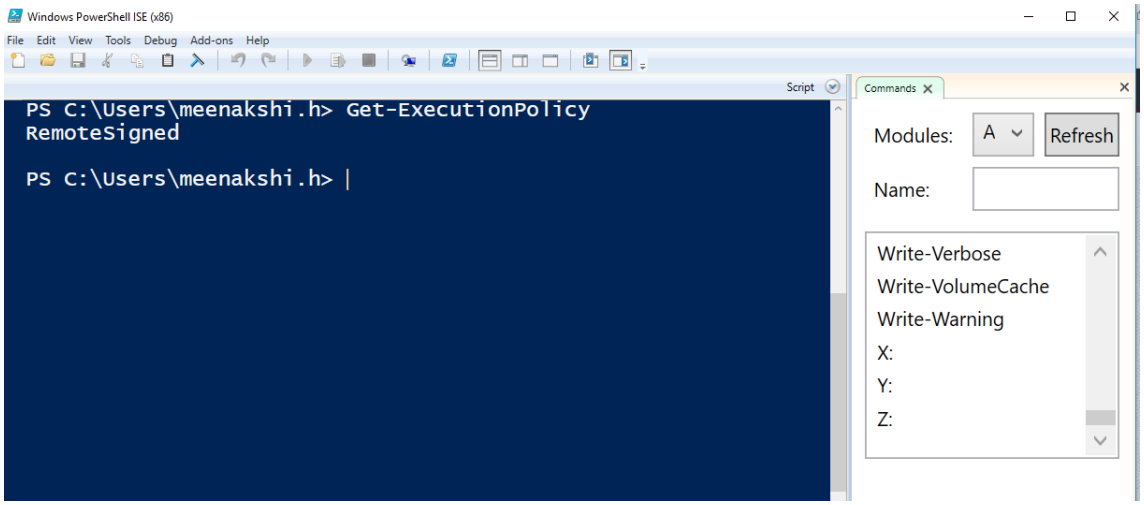


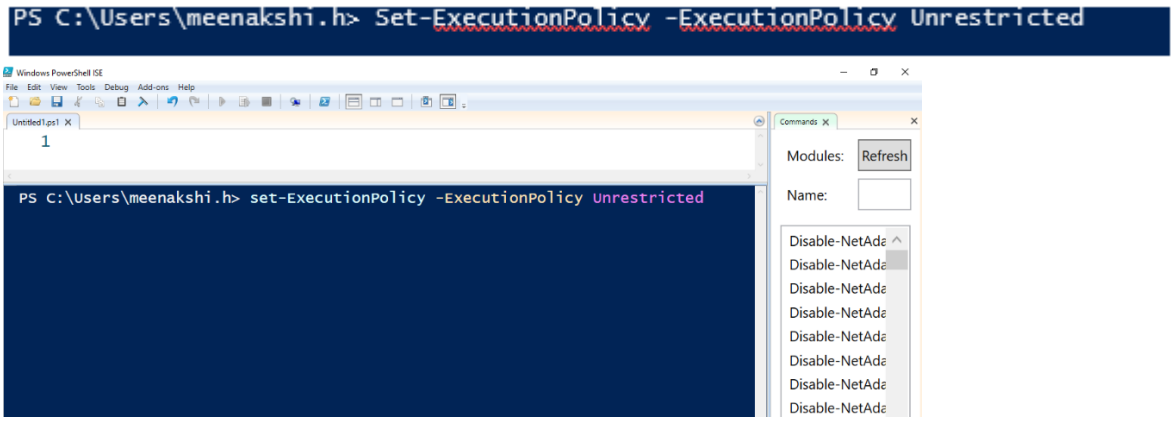
Fig 9. Installation and configuration of environment

1. A clone for the darknet can be created and downloaded from here: https://github.com/AlexeyAB/darknet
2. Extract it to a location of your choice. Darknet take 26.9 MB disk space.
3. Open a MS-PowerShell window in Administrator mode. By executing the command:

<Get-ExecutionPolicy>



4. If it returns restricted, then run the command below.



5. If this command executes correctly, the darknet is installed successfully.

Setting up Pre -Trained models: How to Train YOLO to detect your conventional objects

YOLO v4 Darknet is trained with COCO data set using Convolution Neural Network.

COCO Data set - Common Objects in Context			
No. of classes	Training images	validation images	Download link
80	80,000	40,000	https://cocodataset.org/.

Object detection using YOLO is dependent on preparing weights and few configuration files. The weights are pretrained for COCO data set.

Following steps illustrates how to train using YOLO v4:

Download configuration files-yolov4.cfg from here https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/yolov4.cfg and follow the make few changes in the pretrained parameters.

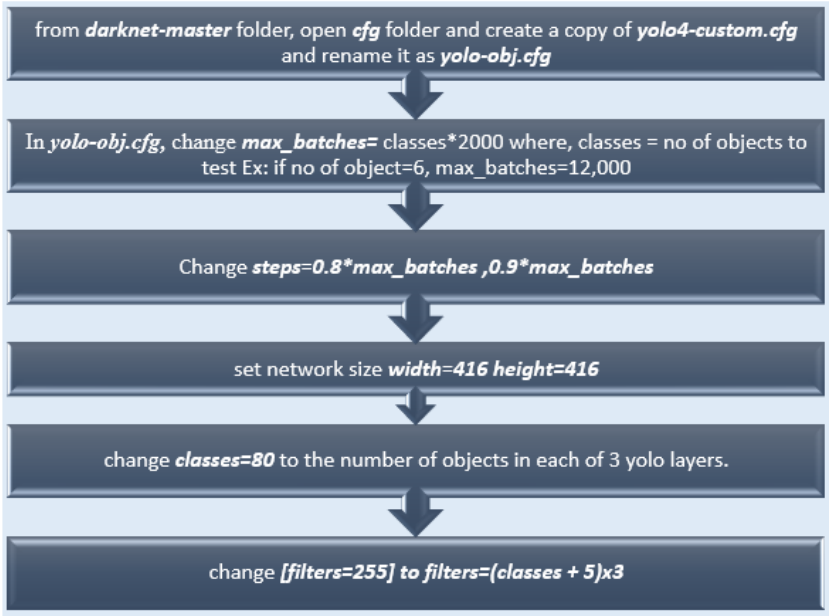


Fig 10. Steps for configuring the darknet files

2. Download the pre trained weights from the link yolov4. Conv .137 and save it in the darknet-master folder.

3. In a WordPad type the name of each object in separate lines and save the file as obj. names in darknet-master->data folder.

4. Create file obj.data in the folder darknet-master->data, and edit the following

```
classes=  3
train = data/train.txt
names = data/obj.names
```

5. Create a folder in darknet-master->data -> obj. Store all the images in obj

6. Create a train.txt file in a path: darknet-master->data folder-> train.txt. This file includes all training images.

data/obj/img1.jpg
data/obj/img2.jpg
data/obj/img3.jpg
data/obj/img4.jpg

7. In the darknet-master folder open Make file in wordpad and change GPU=0, CUDNN=1, OPENCV=1 as shown in the following picture. This is done to make the training on CPU.

Compile darknet:

To compile the darknet execute the following commands:

< make >

< . /darknet >

Train the network:

- The training process could take several hours even days.
- But colab only allow a maximum of 12 hours of running time in ordinary accounts. Those who are interested to train YOLO using darknet in google colab can find the detailshere: https://colab.research.google.com/drive/1lTGZsfMaGUpBG4inDIQwIJVW476ibX
- Training can be done parts by parts. After each 1000 epoch weights are saved in the backup folder so we could just retrain from there. For starting the training run the code.

TESTING: For testing run the following code

!. /darknet detector test data/obj.data cfg/yolo-obj.cfg backup/yolo-obj_12000.weights

What are the Advantages of YOLO over other decoders?

- Rather than using two step method for classification and localization of object, YOLO applies single CNN for both classification and localization of the object.
- YOLO can process images at about 40-90 FPS, so it is quite fast. This means streaming video can be processed in real-time, with negligible latency in a few milliseconds. The

architecture of YOLO makes it extremely fast. When compared with R-CNN, it is 1000 times faster and 100 times faster than fast R-CNN.

Limitations and drawbacks of the YOLO object detector:

1. It especially does not handle objects grouped close together: Since each grid cell predicts only two boxes and can only have one class, this limits the number of nearby objects that YOLO can predict, especially for small objects that appear in groups, such as flocks of birds which may.

2. It does not always handle small objects well: YOLO can detect only 49 objects. The reason for this limitation is due to the YOLO algorithm itself. The YOLO object detector divides an input image into an MxM grid where each cell in the grid predicts only a single object. If there exist multiple, small objects in a single cell then YOLO will be unable to detect them, ultimately leading to missed object detections.

EXERCISE–7

Optimization of Training in Deep Learning

Module Name: Optimization of Training in Deep Learning

Exercise Name: Design a Deep learning Network for Robust Bi-Tempered Logistic Loss.

Advanced loss functions and parameter tuning:

A **loss function** is used in **neural network** model to optimize the parameter values.Loss function can be classified into two broad categories as shown in fig 2.

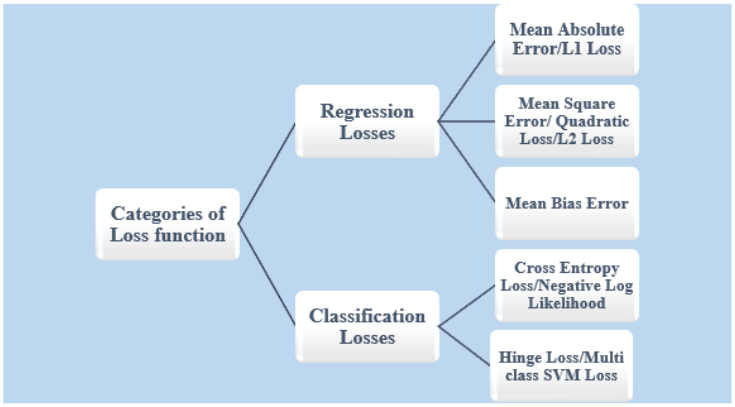


Fig 3. Loss functions

Following table compares the frequently used loss function in deep learning for regression and classification task respectively.

Mean Square Loss (regression)	**Cross entropy Loss(classification)**
Due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions.	An important aspect of this is that cross entropy loss penalizes heavily the predictions that are *confident but wrong*.
$MSE = \frac{\sum_{i=1}^{n}(actal - predicted)^2}{n}$	$cross\ entropy = -(actual_i\ log(predicted) + (1 - actual_i) log(1 - actual_i))$
In binary **classification** model is trained with **MSE** Cost function, it is not guaranteed to minimize the Cost function.	Distributions with long tails can be modeled poorly with too much weight given to the unlikely events

Demonstration of most used Loss function as optimization algorithm using CNN for MNIST.

Step1: The CNN model is compiled using Adagrad optimizer

```
1. # -*- coding: utf-8 -*-
2. """
```

```
Created on Mon Sep 14 01:29:13 2020

@author: meenakshi.h
"""

from keras.datasets import mnist
import tensorflow
 from tensorflow.keras.models import Sequential
 from tensorflow.keras.layers import Dense, Flatten
 from tensorflow.keras.layers import Conv2D, MaxPooling2D
 from tensorflow.keras.layers import BatchNormalization
```

```
# Model configuration
batch_size = 250
no_epochs = 5
no_classes = 10
validation_split = 0.2
verbosity = 1

# Load KMNIST dataset
(input_train, target_train), (input_test, target_test)
=mnist.load_data()

 # Shape of the input sets
 input_train_shape = input_train.shape
 input_test_shape = input_test.shape
```

```
# Keras layer input shape
input_shape = (input_train_shape[1], input_train_shape[2], 1)

# Reshape the training data to include channels
input_train = input_train.reshape(input_train_shape[0],
input_train_shape[1], input_train_shape[2], 1)
input_test = input_test.reshape(input_test_shape[0],

input_test_shape[1], input_test_shape[2], 1)
```

```
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')

# Normalize input data
input_train = input_train / 255
input_test = input_test / 255


```

```
# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
```

```
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(BatchNormalization())

 model.add(MaxPooling2D(pool_size=(2, 2)))
 model.add(BatchNormalization())
 model.add(Flatten())
 model.add(Dense(256, activation='relu'))
 model.add(BatchNormalization())
 model.add(Dense(no_classes, activation='softmax'))
```

```
     # Compile the model
model.compile(loss=tensorflow.keras.losses.sparse_categorical_cro
ssentropy,optimizer=tensorflow.keras.optimizers.Adagrad(
    learning_rate=0.001,
    initial_accumulator_value=0.1,

    epsilon=1e-07))

# Fit data to model
history = model.fit(input_train, target_train,
            batch_size=batch_size,
            epochs=no_epochs,
            verbose=verbosity,
            validation_split=validation_split)

# Generate generalization metric  s
score = model.evaluate(input_test, target_test, verbose=0)
 print(f'Test loss using Adagrad: {score[0]} / Test accuracy:

{score[1]}')
```

Step2: Compile the CNN model with Adadelta Optimizer given below replacing Adagrad in the above CNN model

```
model.compile(loss=tensorflow.keras.losses.sparse_categorical_cro
ssentropy,
optimizer =
tensorflow.keras.optimizers.Adadelta(learning_rate=0.001,

rho=0.95, epsilon=1e-07, name="Adadelta"))
```

Step3: Compile the CNN model with Adam-Adaptive momentum estimation given below replacing Adagrad in the above CNN model

```
model.compile(loss=tensorflow.keras.losses.sparse_categorical_crossentropy,
optimizer=tensorflow.keras.optimizers.Adam(learning_rate=0.01),

metrics=['accuracy'])
```

Step4: Compile the CNN model with Adabound momentum estimation given below replacing Adagrad in the above CNN model

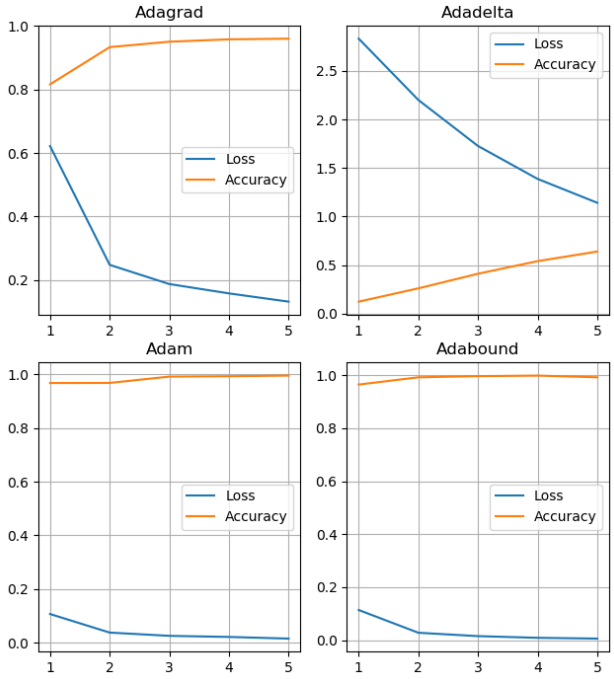
```
from keras_adabound import AdaBound
model.compile(loss=tensorflow.keras.losses.sparse_categorical_crossentropy,

optimizer=AdaBound(lr=1e-3, final_lr=0.1))
```

In each case the model is executed and for first 5 epoch, loss and the accuracy are recorded. In order to study the performance of the optimizer with respect to cross entropy loss function, a graph was plotted. The code for plotting the graph is given below:

```
import matplotlib.pyplot as plt
x=[1,2,3,4,5]

Loss1=[0.6218,0.2478,0.1874,0.158,0.132]
accuracy1=[0.8162,0.9332,0.9503,0.9578,0.9599]
Loss2=[2.8335,2.2018,1.7276,1.3882,1.1422]
accuracy2=[0.1234,0.2605,0.4108,0.5408,0.6399]
Loss3=[0.1073,0.0382,0.0261,0.022,0.0158]
accuracy3=[0.9673,0.9677,0.9912,0.9926,0.9949]
 Loss4=[0.1139,0.0281,0.0153,0.009,0.006]
 accuracy4=[0.9646,0.992,0.9963,0.9984,0.992]

 fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(6.4, 8.5))
 #adgrad
 axs[0, 0].plot(x, Loss1,  label="Loss")
 axs[0, 0].plot(x, accuracy1, label="Accuracy")
 axs[0, 0].legend()
 axs[0, 0].grid()
 axs[0, 0].set(title="Adagrad")
 #Adadelta
 axs[0, 1].plot(x, Loss2,  label="Loss")
 axs[0, 1].plot(x, accuracy2, label="Accuracy")
 axs[0, 1].legend()
 axs[0, 1].grid()
 axs[0, 1].set(title="Adadelta")
 #Adam
 axs[1, 0].plot(x, Loss3,  label="Loss")
 axs[1, 0].plot(x, accuracy3, label="Accuracy")
 axs[1, 0].legend()
 axs[1, 0].grid()
 axs[1, 0].set(title="Adam")
```

```
#Adabound
axs[1, 1].plot(x, Loss4,  label="Loss")
axs[1, 1].plot(x, accuracy4, label="Accuracy")
axs[1, 1].legend()
axs[1, 1].grid()
axs[1, 1].set(title="Adabound")

fig.tight_layout()
plt.show()
```

The below given graph depicts the comparison of Adagrad, Adadelta, Adam and adabound.



When we observe the above graph, we can understand that, Adabound optimization gives the maximum accuracy when as compared to other optimizers and, there is a control in the convergence and generalization.

Loss function can also define as algorithm which converts concepts to practical. So, it is a transformation function on neural networks which converts multiplication of matrix into deep learning. So, far we have seen comprehensive list of loss functions work and let us look at the very recent or advanced loss functions. The advanced loos function in deep learning models are used for specific purposes.

Robust Bi-Tempered Logistic Loss:

We know that, the deep learning model performance is dependent on the quality of training data. The real-world training data sets can be noisy. For example, corrupted images,

mislabeled data are few noisy data sets. The Loss function can fail in handling the noisy training data due to the following two reasons:

1. Highly deviated outliers: Loss function like logistic Loss function is sensitive to outliers

2. Mislabeled data samples: The neural network outputs the class label for each test sample by increasing the distance between the classes. During the process of increasing the decision boundary, the value of the loss function become reduced very fast, so that the training process tend to get close to the boundary of the outliers or mislabeled data samples. Consequently, prediction error occurs. So, a robust loss function is required. "Bi-tempered logistic loss function can be used to generalize he problem of noisy training data.

- As the name says, there are two modifiable parameters that can handle outliers and mislabeled data. They are:
- "temperatures"—t_1 symbolizes the boundedness, and t_2: indicates the rate of decay in the termination or end of the transfer function

- initialize t_1 and t_2 to 1.0 so that, the logistic loss function is recovered.
- If $t_1 < 1.0$ the boundedness gets increased and if $t_2 > 1.0$ makes transfer function heavy tailed.

How to use Bi-Tempred Logistic Loss:

```
#!/bin/bash
  set -e
  set -x

  virtualenv -p python3.
  source ./bin/activate

  pip install tensorflow
  pip install -r bitempered_loss/requirements.txt
        python -m bitempered_loss.loss_test
```

Clickhere: https://ai.googleblog.com/2019/08/bi-tempered-logistic-loss-for-training.html to know more

The usage of logistic loss using bi-tempered is proved by google for a binary or for two-class classification problem with two-layer on feed-forward neural network.

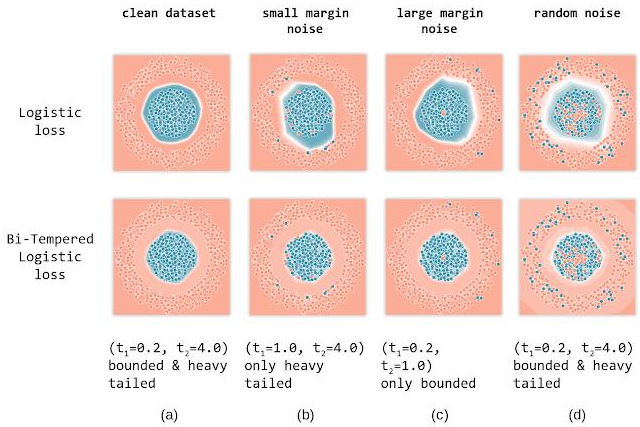


Fig.4.BiTemperedLossfunction[*Courtesy:GoogleATblog: https://ai.googleblog.com/2019/08/bi-tempered-logistic-loss-for-training.html*]

GANs Loss Functions:

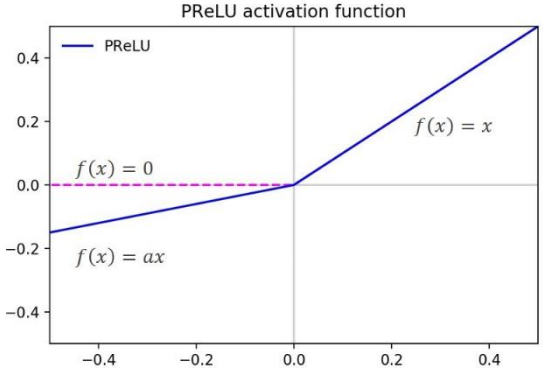
- Generative adversarial networks (GAN) are a powerful subclass of generative models.
- Two main components of GAN are Generator and Discriminator
- This involves the minimization of the generator's loss and maximization of the discriminator's loss.
- Discriminator loss aims at maximizing the probability given to real and fake images. This is a strategy aimed at reducing the worst-case-scenario possible loss. It's simply minimizing the maximum loss. This loss is also used in two-player games to reduce the maximum loss for a layer.

Advanced Activation Functions:

Parametric ReLU: It is a generalization of Leaky ReLU, where, the slope for negative inputs is not predetermined, rather it is considered as a learnable parameter. Formally, it is defined as,

$$y = f(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases}$$

where the coefficient 'a' decides the slope of the negative part.



The neural network should learn the best value of itself along with the rest of the model.

Thus, when a=0, the function becomes ordinary ReLU. When is a small and constant value, usually, 0.01, it becomes LReLU.

Even though we are adding one extra parameter to be learnt in this case, it is a negligible overhead when compared to the number of weights that are usually being learnt in deep neural networks.

The gradient of the PReLU function is given as –

$$f'(x) = \frac{\delta f(x)}{\delta a} = \begin{cases} 0, & x > 0 \\ x, & x \leq 0 \end{cases}$$

EXERCISE-8

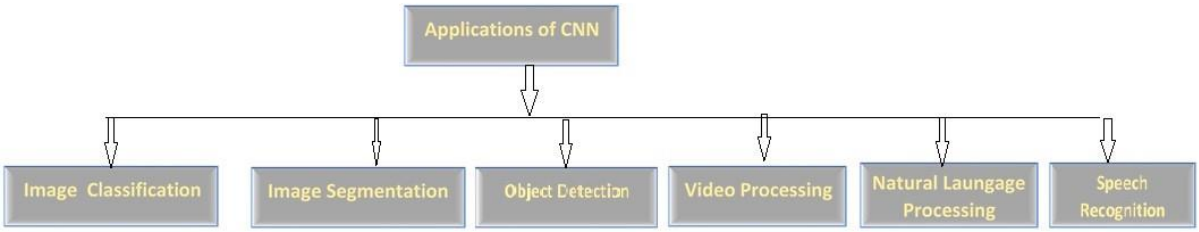
ADVANCED CNN

Module name: Advanced CNN

Exercise: Build AlexNet using Advanced CNN.

INTRODUCTION:

Convolutional Neural Network- CNN is a distinct kind of Neural Networks, which is proved to be the best technique to handle several interesting problems related to Computer Vision and Image Processing.



The learning capability of the deep CNN is mainly because of employing the multiple feature extraction techniques that can automatically extract the patterns from the data. This ability of CNN makes it more powerful and effective CNN.

The voluminous data that is being gathered and the developments in the hardware technology has influenced the new advanced architecture for CNN. Top industries such as Google, AT&T, Amazon, Tesla, Facebook and Microsoft are exploring new architectures of CNN through dynamic research group.

1. ConvNet:

- First multilayered CNNA supervised training
- A backpropagation algorithm
- An effective outcome was observed when implemented for handwritten digit recognition and recognition of zip code problems

2. LeNet:

- Improved form of ConvNet
- Used in a document processing for character recognition problems
- Proved to perform optimal in optical character recognition (OCR)and Biometric-fingerprint recognition
- Commercial use in Bank application and ATMs during 1993 and 1996
- Limitation: didn’t accomplish better results for other type image recognition problem

3. AlexNet:

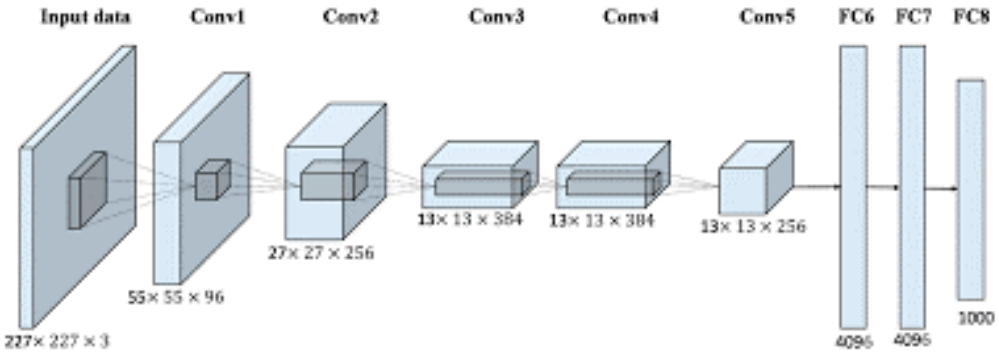
- The main innovation in 2012
- The Idea of basic hyperparameter like stride, filter dimensions, padding, and other was presented for each layer to leverage the image recognition problem

Other like GoogleNet, MobileNet, R-CNN, Fast R-CNN, Faster R-CNN

CASE STUDY OF CNN:

Alex Net:

AlexNet is an incredibly powerful model capable of achieving high accuracies on very challenging datasets. The architecture is as given below figure.



The hyperparameter of AlexNet as listed in below table:

AlexNet architecture Hyperparameter and other details		
Neural Network Layers	**Activation Functions**	**Overfitting Problem**
• consists of 8 layers • 5 convolutional layers • 3 fully connected layers	Uses **ReLU Nonlinear function instead of** tanh function	AlexNet had 60 million parameters, a major issue in terms of overfitting. Two methods were employed to reduce overfitting as given below: 1) **Data Augmentation: methods like** image translations and horizontal reflections, Principle Component Analysis (PCA) on the RGB pixel were used to reduce the error rate 2) **Dropout.** This technique consists of "turning off" neurons with a predetermined probability. dropout also increases the training time needed for the model's convergence.

DEMONSTRATION OF ALEXNET :

```
#------------------------------
AlexNet Demonstartion
#------------------------------
#Import keras
import numpy as np
from keras.datasets import mnist
import matplotlib.pyplot as plt
#-----------------------------------------------
#Load data set
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(x_train.shape)

print(x_test.shape)

element = 200
plt.imshow(x_train[element])
plt.show()
print("Label for the element", element,":", y_train[element])
x_train = x_train.reshape((-1, 28*28))
x_test = x_test.reshape((-1, 784))
print(x_train.shape)
print(x_test.shape)
x_train = x_train / 255
x_test = x_test / 255
#-----------------------------------------------------------------------


from keras.models import Sequential
from keras.utils import to_categorical
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization

#-----------------------------------------------------------------------
```

```
# creating model
model = Sequential()
# 1st Convolutional Layer
model.add(Conv2D(filters = 96, input_shape = (60000,784, 3),kernel_size = (11, 11), strides = (4,
4), padding = 'valid'))
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2),strides = (2, 2), padding = 'valid'))
# Batch Normalisation
model.add(BatchNormalization())


# 2nd Convolutional Layer
```

```
model.add(Conv2D(filters = 256, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2),padding = 'valid'))
# Batch Normalisation
model.add(BatchNormalization())


# 3rd Convolutional Layer
model.add(Conv2D(filters = 384, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Batch Normalisation

model.add(BatchNormalization())
```

```
# 4th Convolutional Layer
model.add(Conv2D(filters = 384, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Batch Normalisation

model.add(BatchNormalization())

# 5th Convolutional Layer
model.add(Conv2D(filters = 256, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
# Batch Normalisation

model.add(BatchNormalization())

# Flattening

model.add(Flatten())
```

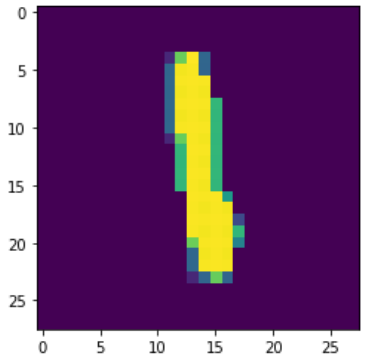
```
# 1st Dense Layer
model.add(Dense(4096, input_shape = (224*224*3, )))
model.add(Activation('relu'))
# Add Dropout to prevent overfitting
model.add(Dropout(0.4))
# Batch Normalisation

model.add(BatchNormalization())

# 2nd Dense Layer
```

```
model.add(Dense(4096))
model.add(Activation('relu'))
# Add Dropout
model.add(Dropout(0.4))
# Batch Normalisation

model.add(BatchNormalization())

# Output Softmax Layer
model.add(Dense(10))
model.add(Activation('softmax'))

#-----------------------------------------------------------------------
```

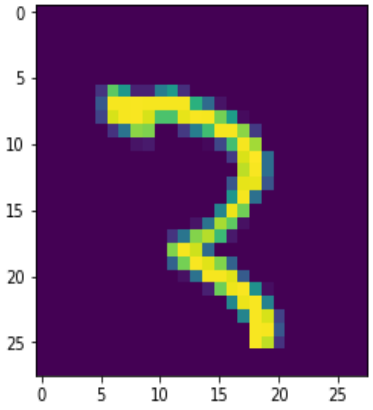
```
# compile the model
model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=['accuracy'])
y=to_categorical(y_train)

#-----------------------------------------------------------------------

# Fit the model
model.fit(x=x_train,y=to_categorical(y_train),epochs=10,batch_size=64,shuffle=True)

#-----------------------------------------------------------------------

# Evaluate the model
eval = model.evaluate(x_test, to_categorical(y_test))
print('eval')

#-----------------------------------------------------------------------

# Predictions
predictions = model.predict(x_test[0:100])
predictions[0]
np.argmax(predictions[0])

plt.imshow(x_test[0].reshape(28,28))
```

Output:

```
(60000, 28, 28)
(10000, 28, 28)
```



```
Label for the element 200 : 1
(60000, 784)
(10000, 784)
```



```
>>> Instructions for updating:
>>> Use tf.where in 2.0, which has the same broadcast rule as np.where
>>> Epoch 1/10
>>> 60000/60000 [==============================] - 21s 344us/step - loss: 0.1972 - acc: 0.9398
>>> Epoch 2/10
>>> 60000/60000 [==============================] - 20s 332us/step - loss: 0.0802 - acc: 0.9750
>>> Epoch 3/10
>>> 60000/60000 [==============================] - 19s 312us/step - loss: 0.0530 - acc: 0.9834
>>> Epoch 4/10
>>> 60000/60000 [==============================] - 19s 311us/step - loss: 0.0390 - acc: 0.9874
>>> Epoch 5/10
>>> 60000/60000 [==============================] - 19s 314us/step - loss: 0.0300 - acc: 0.9906
>>> Epoch 6/10
>>> 60000/60000 [==============================] - 19s 319us/step - loss: 0.0254 - acc: 0.9920
>>> Epoch 7/10
>>> 60000/60000 [==============================] - 19s 309us/step - loss: 0.0218 - acc: 0.9927
```

EXERCISE-9

AUTOENCODERS ADVANCED

Module name: Autoencoders Advanced

Exercise: Demonstration of Application of Autoencoders.

INTRODUCTION:

What are Autoencoders?

Autoencoders are the data encoding techniques based on Unsupervised Artificial Neural Networks. This special type of ANN is trained to encode the data so that in such a way that data is represented in compressed form. The Autoencoders are also trained to decode the data so that, the original data can be reconstructed as far as possible.

Architecture of Autoencoders:

The architecture for autoencoders is varied. In this section LSTM autoencoders is discussed. LSTM based autoencoders are used to encode and decode the sequence data.

Why sequence data is challenging to process?

- Sequence data are challenging for prediction task because the size of the is not fixed but it varies.
- Also, the temporal series of the data representation make it challenging to extract the features.

So, the building a predictive model to predict the sequence data involve sequence of operation and hence such problems are called as Sequence-to Sequence. Autoencoders comes as the best choice to handle sequence-to-sequence problems.

DEMONSTRATION OF APPLICATION OF AUTOENCODERS:

LSTM based autoencoders can be created to for various applications. Some of them are demonstrated below.

1. Reconstruction of sequence using Autoencoders:

Setp1: Building a simple autoencoders to create simple sequence

```
1. from numpy import array
2. from keras. models import Sequential
3. from keras. layers import LSTM
4. from keras. layers import Dense
5. from keras. layers import RepeatVector
```

```
from keras. layers import TimeDistributed
from keras. utils import plot_model
# lstm autoencoder recreate sequence

# define input sequence
sequence = array ([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
# Reshape input into [samples, timesteps, features]
n_in = len(sequence)
sequence = sequence. Reshape ((1, n_in, 1))
# define model
model = Sequential ()
model.add (LSTM (100, activation='relu', input_shape=(n_in,1)))
model.add (RepeatVector(n_in))
model.add (LSTM (100, activation='relu', return_sequences=True))
model.add (TimeDistributed (Dense(1)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(sequence, sequence, epochs=300, verbose=0)
plot_model(model, show_shapes=True, to_file='reconstruct_lstm_autoencoder.png')
# demonstrate recreation
yhat = model.predict(sequence, verbose=0)

print(yhat[0,:,0])
```

2. Prediction of the sequence of number using Autoencoders:

Like reconstruction, autoencoders can be used to predict the sequence, the code is as given below:

```
# lstm autoencoder predict sequence
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.utils import plot_model
# define input sequence
seq_in = array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
# reshape input into [samples, timesteps, features]
n_in = len(seq_in)
seq_in = seq_in.reshape((1, n_in, 1))
# prepare output sequence
seq_out = seq_in[:, 1:, :]
n_out = n_in - 1
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_in,1)))
model.add(RepeatVector(n_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
```

```
model.compile(optimizer='adam', loss='mse')
plot_model(model, show_shapes=True, to_file='predict_lstm_autoencoder.png')
# fit model
model.fit(seq_in, seq_out, epochs=300, verbose=0)
# demonstrate prediction
yhat = model.predict(seq_in, verbose=0)

print(yhat[0,:,0])
```

3. Outlier/Anomaly detection using Autoencoders:

Suppose the input data is highly correlated and requires a technique to detect the anomaly or an outlier then, Autoencoders is the best choice. Since, autoencoders can encode the data in the compressed for, they can handle the correlated data.

Let's train the autoencoders using MNIST data set using simple Feed Forward neural network.

Code: Simple 6 layered feed forward Autoencoders

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
Epoch 1/10
59/59 [==============================] - 7s 114ms/step - loss: 0.0758 - val_loss: 0.0516
Epoch 2/10
59/59 [==============================] - 6s 110ms/step - loss: 0.0440 - val_loss: 0.0369
Epoch 3/10
59/59 [==============================] - 7s 111ms/step - loss: 0.0342 - val_loss: 0.0311
Epoch 4/10
59/59 [==============================] - 7s 111ms/step - loss: 0.0297 - val_loss: 0.0274
Epoch 5/10
59/59 [==============================] - 7s 110ms/step - loss: 0.0267 - val_loss: 0.0251
Epoch 6/10
59/59 [==============================] - 6s 110ms/step - loss: 0.0246 - val_loss: 0.0233
Epoch 7/10
59/59 [==============================] - 6s 110ms/step - loss: 0.0231 - val_loss: 0.0221
Epoch 8/10
59/59 [==============================] - 6s 110ms/step - loss: 0.0220 - val_loss: 0.0211
Epoch 9/10
59/59 [==============================] - 7s 110ms/step - loss: 0.0211 - val_loss: 0.0204
Epoch 10/10
59/59 [==============================] - 7s 113ms/step - loss: 0.0203 - val_loss: 0.0195
```

Once the autoencoders is trained on MNIST data set, an anomaly detection can be done using 2 different images. First one of the images from the MNIST data set is chosen and feed to the trained autoencoders. Since, this image is not an anomaly, the error or loss function is expected to be very low. Next, when some random image is given as test image, the loss rate is expected to be very high as it is an anomaly.

Simple 6 layered Autoencoders build to train on MNIST data

```
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Input
```

```
from keras import optimizers
from keras.optimizers import Adam

(x_train, y_train), (x_test, y_test) = mnist.load_data()
train_x = x_train.reshape(60000, 784) / 255
val_x = x_test.reshape(10000, 784) / 255

autoencoder = Sequential ()
autoencoder.add (Dense (512, activation='elu', input_shape= (784,)))
autoencoder.add (Dense (128, activation='elu'))
autoencoder.add (Dense (10, activation='linear', name="bottleneck"))
autoencoder.add (Dense (128, activation='elu'))
autoencoder.add (Dense (512, activation='elu'))
autoencoder.add (Dense (784, activation='sigmoid'))
autoencoder. compile (loss='mean_squared_error', optimizer = Adam ())
trained_model = autoencoder. Fit (train_x, train_x, batch_size=1024, epochs=10, verbose=1,
validation_data= (val_x, val_x))
encoder = Model (autoencoder. Input, autoencoder.get_layer('bottleneck'). Output)
encoded_data = encoder. Predict (train_x) # bottleneck representation
decoded_output = autoencoder. Predict (train_x)        # reconstruction
encoding_dim = 10

# Return the decoder
encoded_input = Input(shape=(encoding_dim,))
decoder = autoencoder. layers[-3](encoded_input)
decoder = autoencoder. layers[-2](decoder)
decoder = autoencoder. layers[-1](decoder)

decoder = Model (encoded_input, decoder)
```

Anamoly Detection

```
# %matplotlib inline
from keras.preprocessing import image
# if the img.png is not one of the MNIST dataset that the model was trained on, the error will be
very high.
img = image.load_img("C:\Users\meenakshi.h\Desktop\Images\fig12.png", target_size=(28, 28),
color_mode = "grayscale")
input_img = image.img_to_array(img)
inputs = input_img.reshape(1,784)
target_data = autoencoder.predict(inputs)
dist = np.linalg.norm(inputs - target_data, axis=-1)

print(dist)
```

EXERCISE-10

Advanced GANs

Module name: Advanced GANs

Exercise: Demonstration of GAN

INTRODUCTION:

Can we train the machine to generate the following?

1. Given the image of a criminal whose face is masked with glass. Is it possible for the machine to automatically generate the photo without glass?

2. Suppose few simple sentences which describe the bird as: ***"A small bird with read head. The features fade from red to gray from head to tail."*** are given to machine. Is it possible for the machine to generate the image of the bird?

3. Given a photo of a person which is captured 10 years ago, is it possible for the machine to generate a photo of the same person for the present time?

Yes, advanced deep learning has made an amazing progress, where machines can be trained to generate or create new images. Deep learning models which can generate new images from the given training data set are called as **Generative Adversarial Networks** or in short **GAN**s. Before understanding GANs it is required to have the knowledge of Generative models.

Difference between Discriminative and Generative: Machine learning models can broadly fall under two categories namely, discriminative and generative. Following table lists the difference between discriminative and generative models.

Discriminative	Generative
Discriminative models are developed to predict a class label using the given labelled training set.	Generative models are the one that are used to create or generate new examples based on the input data distribution
Since, the samples are discriminated based on the labelled data set, they are Supervised Machine learning in nature.	Since, the input data is not labelled, generative models are called as Unsupervised machine learning
Example: Classification, logistic regression	Example: Naive Bayes can be Discriminative and generative, Latent Dirichlet Allocation-LDA, Gaussian Mixture Model-GMM and deep learning based model namely GANs- **Generative Adversarial Network**

The advanced Deep learning unsupervised Generative models that are the state-of the art include:

1) Variational Autoencoder, or VAE, and

2) The Generative Adversarial Network, or GAN

Let's look at the architectural aspects of GANs

ARCHITECTURE AND TYPES OF GAN:

The architecture of the deep learning GAN models consists of two modules namely **generator** and **Discriminator.**

1. A generator model:

- It is the learning component of a GAN models.
- It learns to generate the new data by incorporating the feedback received from the discriminator.
- It learns to allow the discriminator to classify its newly generated data as real.
- Hence, training the Generator also requires the discriminator to be considered.

2. A Discriminator models:

- It is a classifier in GANs
- Its job is to classify the output of the generator (newly generated data) from the real one.

The below given figure 1.Shows the architecture of GANs

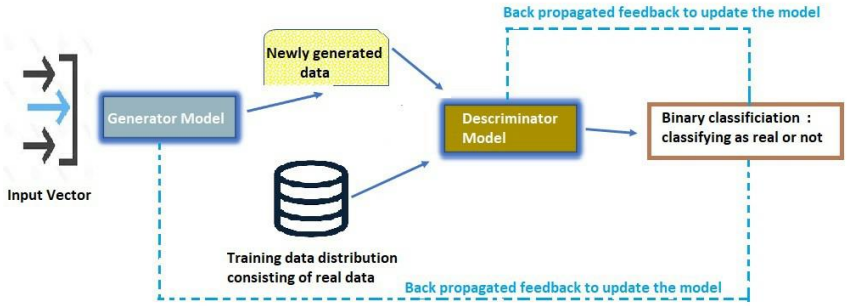


Fig 1. Architecture of GANs

DEMONSTRATION OF GAN:

The application of GAN models is varied. Here two case studies are considered to demonstrate GAN for Data set generation. They are as follows:

1. Image Augmentation using MNIST data set
2. New image generation for CIFAR data Set

1. Image Augmentation: Case study of GAN

Whenever the data set don't have enough samples to train the machine due to various constraints in data collection process then, it becomes necessary to use Augmentation. Particularly, when more complex object needs to be recognized then, Image data augmentation technique is used. It is a method of artificially escalating the size of a training dataset by creating artificially new set of images.

Using Keras Image augmentation is demonstrated by building deep learning GANs.

ImageDataGenerator class available in Keras is used in demonstration. It defines the configuration for image data preparation and augmentation.

In this demonstration following properties are demonstrated:

- Feature standardization.
- ZCA whitening.
- Random flips.

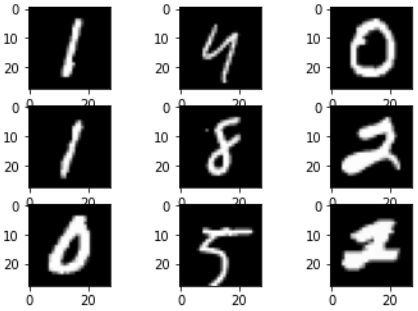
a. Feature Standardization

Using GANs model the pixel values across the entire dataset can be standardized. Feature standardization is the process of standardizing the pixel which is performed for each column in a tabular dataset. This can be done by setting the feature wise_center and feature wise_std_normalization arguments on the ImageDataGenerator class.

```
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True,
featurewise_std_normalization=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Output

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```



b. ZCA-Zero Component Analysis Whitening

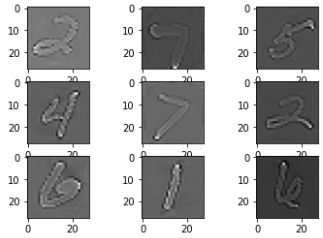
Suppose the pixel has many redundant pixels then, training process can't be effective. So, to reduce the redundant pixels whitening of an image is used. The process of transforming the original image using a linear algebra operation that reduces the redundancy in the matrix of pixel is called as Whitening transformation.

Advantage of whitening: Less redundant pixels in the image is expected to improve the structures and features in the image so that, machine can to the learn image effectively.
In this demonstration, ZCA is used to show GANs application in generating new image after eliminating the redundant pixels.

```
1.  # ZCA whitening
2.  from keras.datasets import mnist
3.  from keras.preprocessing.image import ImageDataGenerator
4.  from matplotlib import pyplot
5.  # load data
6.  (X_train, y_train), (X_test, y_test) = mnist.load_data()
7.  # reshape to be [samples][width][height][channels]
8.  X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
9.  X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
10. # convert from int to float
11. X_train = X_train.astype('float32')
12. X_test = X_test.astype('float32')
13. # define data preparation
14. datagen = ImageDataGenerator(zca_whitening=True)
15. # fit parameters from data
16. datagen.fit(X_train)
17. # configure batch size and retrieve one batch of images
18. for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
19.   # create a grid of 3x3 images
20.   for i in range(0, 9):
21.     pyplot.subplot(330 + 1 + i)
22.     pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
23.   # show the plot
24.   pyplot.show()
25.   break
```

Output :

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/image_data_generator.py:337: UserWarning: This ImageDataGenerator specifies `zca_whitening`, whi
  warnings.warn('This ImageDataGenerator specifies '
```

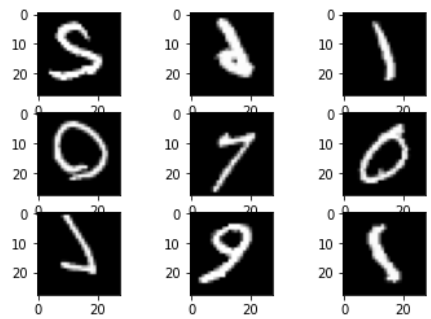


c. Random Flips

Random Flip can be used as augmentation technique on an image data to improve the performance on large and complex problems.

```
1.  # Random Flips
2.  from keras.datasets import mnist
3.  from keras.preprocessing.image import ImageDataGenerator
4.  from matplotlib import pyplot
5.  # load data
6.  (X_train, y_train), (X_test, y_test) = mnist.load_data()
7.  # reshape to be [samples][width][height][channels]
8.  X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
9.  X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
10. # convert from int to float
11. X_train = X_train.astype('float32')
12. X_test = X_test.astype('float32')
13. # define data preparation
14. datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
15. # fit parameters from data
16. datagen.fit(X_train)
17. # configure batch size and retrieve one batch of images
18. for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
19.   # create a grid of 3x3 images
20.   for i in range(0, 9):
21.     pyplot.subplot(330 + 1 + i)
22.     pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
23.   # show the plot
24.   pyplot.show()
25.   break
```

Output:



New image generation using CIFAR data Set

Step 1: Importing the required libraries

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model

from keras.optimizers import Adam,SGD
```

Step 2: Loading the data

```
#Loading the CIFAR10 data
(X, y), (_, _) = keras.datasets.cifar10.load_data()
  #Selecting a single class images
#The number was randomly chosen and any number
#between 1 to 10 can be chosen

X = X[y.flatten() == 8]
```

Step 3: Defining parameters to be used in later processes

```
#Defining the Input shape
image_shape = (32, 32, 3)

latent_dimensions = 100
```

Step 4: Defining a utility function to build the Generator

```
def build_generator():
    model = Sequential()
   #Building the input layer
      model.add(Dense(128 * 8 * 8, activation="relu",
                  input_dim=latent_dimensions))
      model.add(Reshape((8, 8, 128)))
        model.add(UpSampling2D())
         model.add(Conv2D(128, kernel_size=3, padding="same"))
      model.add(BatchNormalization(momentum=0.78))
      model.add(Activation("relu"))
        model.add(UpSampling2D())
        model.add(Conv2D(64, kernel_size=3, padding="same"))
      model.add(BatchNormalization(momentum=0.78))
```

```
        model.add(Activation("relu"))
          model.add(Conv2D(3, kernel_size=3, padding="same"))
        model.add(Activation("tanh"))
  #Generating the output image
        noise = Input(shape=(latent_dimensions,))
        image = model(noise)

  return Model(noise, image)
```

Step 5: Defining a utility function to build the Discriminator

```
def build_discriminator():
    #Building the convolutional layers
        #to classify whether an image is real or fake
        model = Sequential()
   model.add(Conv2D(32, kernel_size=3, strides=2,
                    input_shape=image_shape, padding="same"))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
          model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
        model.add(ZeroPadding2D(padding=((0,1),(0,1))))
        model.add(BatchNormalization(momentum=0.82))
        model.add(LeakyReLU(alpha=0.25))
        model.add(Dropout(0.25))
          model.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
        model.add(BatchNormalization(momentum=0.82))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
          model.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(LeakyReLU(alpha=0.25))
        model.add(Dropout(0.25))
         #Building the output layer
        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))
   image = Input(shape=image_shape)
        validity = model(image)

   return Model(image, validity)
```

Step 6: Defining a utility function to display the generated images

```
def display_images():
        r, c = 4,4
        noise = np.random.normal(0, 1, (r * c,latent_dimensions))
        generated_images = generator.predict(noise)
    #Scaling the generated images
        generated_images = 0.5 * generated_images + 0.5
     fig, axs = plt.subplots(r, c)
```

```
        count = 0
        for i in range(r):
            for j in range(c):
                axs[i,j].imshow(generated_images[count, :,:,])
                axs[i,j].axis('off')
                count += 1
        plt.show()

        plt.close()
```

Step 7: Building the Generative Adversarial Network

```
# Building and compiling the discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(0.0002,0.5),
                     metrics=['accuracy'])

#Making the Discriminator untrainable
#so that the generator can learn from fixed gradient
discriminator.trainable = False
 # Building the generator
generator = build_generator()
  #Defining the input for the generator
#and generating the images
z = Input(shape=(latent_dimensions,))
image = generator(z)
#Checking the validity of the generated image
valid = discriminator(image)
  #Defining the combined model of the Generator and the Discriminator
combined_network = Model(z, valid)
combined_network.compile(loss='binary_crossentropy',

              optimizer=Adam(0.0002,0.5))
```

Step 8: Training the network

```
num_epochs=15000
batch_size=32
display_interval=2500
losses=[]
  #Normalizing the input
X = (X / 127.5) - 1.
#Defining the Adversarial ground truths
valid = np.ones((batch_size, 1))
  #Adding some noise
valid += 0.05 * np.random.random(valid.shape)
fake = np.zeros((batch_size, 1))
fake += 0.05 * np.random.random(fake.shape)
```

```
for epoch in range(num_epochs):
  #Training the Discriminator
    #Sampling a random half of images
        index = np.random.randint(0, X.shape[0], batch_size)
        images = X[index]
      #Sampling noise and generating a batch of new images
        noise = np.random.normal(0, 1, (batch_size, latent_dimensions))
        generated_images = generator.predict(noise)
       #Training the discriminator to detect more accurately
        #whether a generated image is real or fake
        discm_loss_real = discriminator.train_on_batch(images, valid)
        discm_loss_fake = discriminator.train_on_batch(generated_images, fake)
        discm_loss = 0.5 * np.add(discm_loss_real, discm_loss_fake)
          #Training the Generator
         #Training the generator to generate images
        #which pass the authenticity test
        genr_loss = combined_network.train_on_batch(noise, valid)
        #Tracking the progress
        if epoch % display_interval == 0:

            display_images()
```

Output:

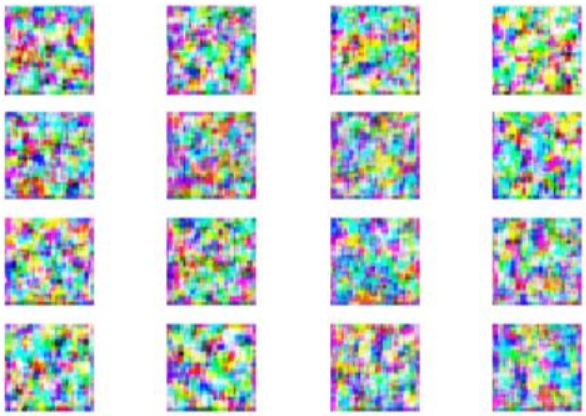
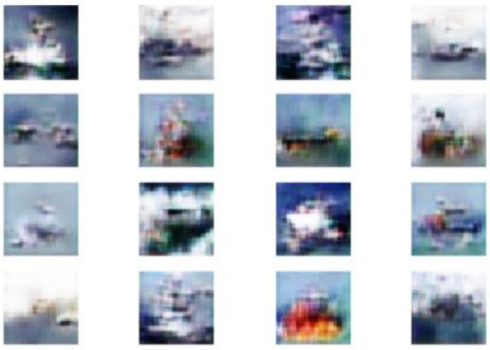


Fig1a.image generated at epoch size 20,



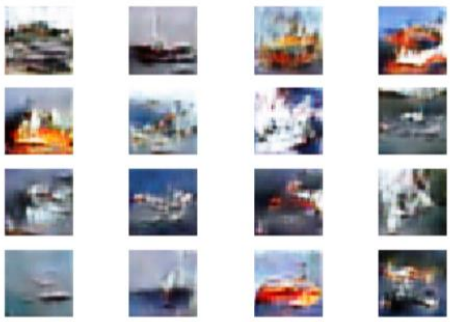


Fig1b image generated at epoch size 2000, ***Fig1c.image generated at epoch size 5000***