

IV B.Tech I SEM SOC LAB (PYTHON: DEEP LEARNING)

Concepts you must know before doing this course

Overview of Artificial Intelligence(AI) and its aspects

Using Python libraries such as numpy, pandas, matplotlib and scikit-learn

Core concepts of Machine Learning - Supervised and Unsupervised

Software requirements

Anaconda distribution 4.4.0 with Python 3.6.1 or above.

Here's a list of packages version used while building this course:

- **imageio:** 2.5.0
- **keras:** 2.2.4
- **keras-lr-finder:** 0.1
- **keras-vis:** 0.4.1
- **matplotlib:** 2.0.2
- **mlxtend:** 0.15.0.0
- **numpy:** 1.16.4
- **pandas:** 0.24.1
- **Pillow:** 6.0.0
- **scikit-learn:** 0.21.2
- **scipy:** 1.2.1
- **tensorboard:** 1.12.2
- **tensorflow:** 1.12.0
- **vis:** 0.0.5

Why deep learning?

The starting of 21st century has introduced us with data as the new fuel to drive today's society. Our dependency on electronic devices has increased at an exploding rate. With the availability of such humongous data generated every second along with the ample computational resources, many complex tasks have been solved. Following are a few real world examples:

1. **Facial Recognition Photo Tagging:** Are you on Facebook? If yes, have you ever uploaded a picture of yours along with your friends? Did you notice that just by hovering over your friends' faces, Facebook identifies their faces and asks you to tag them too? This is an example of deep learning image identification backed by the Deep Face (a deep learning architecture developed by Facebook, Inc.).

2. Natural Language Processing/Generation/Translation: Have you ever met a person from a region, whose language is difficult for you to decipher? What do you do if you have to converse with that person? Translation apps could be the answer! Google Translate is one of the targeted services to solve such frequent problems. The Google Translate product uses Google's Neural Machine Translation System (a deep learning architecture) to provide near human-expertise in translating languages.
3. Self-driving cars – Have you heard of the autonomous cars being developed by Tesla, Google, Uber and every other major automaker? Have you wondered how their self-governing navigation system works? They use sensors and on-board analytics to learn to recognize any obstacles in the path and react to them appropriately using Deep Learning.

Some other major platforms which use deep learning include: real-estate companies, e-commerce websites, IoT applications, education platforms, etc., to address various day to day challenges.

The fact that we are all consumers to at least one of these deep learning products out there is undeniable and that is exactly why it is important for each one of us to be knowledgeable about Deep Learning.

Although Deep learning isn't new in the market, however, it's presence has been rapidly rising in the past few years due to the available computational resources and data. The following Google trend of the term "Deep learning" confirms the hype.

Over these stipulated years, deep learning has left its fingerprints globally and it is being used by not only Fortune 500 companies but also mid to small size firms in technology, healthcare, marketing, finance and many other big sectors. And this trend without a doubt will continue to grow in leaps and bounds.

What is deep learning?

The definition of deep learning as quoted by Yann LeCun, Yoshua Bengio and Geoffrey Hinton in a paper titled "Deep Learning", published by them in Nature magazine, says:

"Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction".

Let us try to make sense of this definition with the help of a simple example.

Consider the image



below.

Let us assume that we are trying to teach a child how to recognize cats for the very first time in life. When the child is trying to learn to identify a cat, we never know exactly how the child is relating the word 'cat' to that specific mammal. Having said that, let us still try to reflect upon the various stages of learning of the child.

Stage 1: The child may wonder that all the mammals with fur are cats. So, initially the child tries to identify small features to arrive at the result. However, this could lead to the child incorrectly identifying a dog with fur as a cat. This tells us that the child still needs another stage of learning.

Similarly, a deep learning neural network starts with small features. With limited knowledge retaining capability (i.e. number of nodes and layers), it too tends to misidentify the target output initially.

Stage 2: To improve over the previous course of learning, the child starts combining few small features together like a mammal with long whiskers around nose. This way the child's capability starts increasing as the child learns with more experience, utilizing more memory to piece and remember the features together.

Similarly, a deep learning network joins small features and look for improving its performance and the higher performance probability is still dependent upon ample data and computing resources.

Stage 3: With enough experience, a child connects various small features together to finally distinguish a cat from other mammals.

Likewise, a deep learning network built from scratch does various mistakes due to less knowledge (small features) but it gradually improves with more data (taking less time, if provided with high-end computational resources) to connect those small features to build an intuition to arrive at an output.

Deep learning vs classical machine learning

The notion of deep learning is built on the core of classical machine learning approaches (like regression analysis, classification algorithms, the use of optimization algorithms, etc.) and proves worthy over them in the following situations:

when enormous amounts of data are available

when enormous computational power is available a non-interpretable model is not detrimental

But how exactly does deep learning improve upon classical machine learning? To answer this question let us consider the cat scenario again.

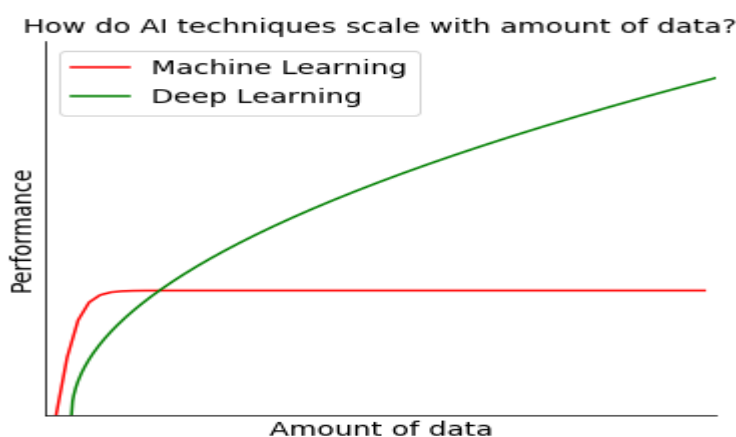
If a classical machine learning algorithm has to identify a cat's face, then it would rely on feature engineering to build its model. The quality of features decides the accuracy of identification. On the contrary, a deep learning algorithm doesn't need feature engineering from the user end! Instead, it asks for a sequence of input images and creates features

(whisker, eyes, nose, hair pattern, etc.) out of them, all by itself! In this sense, we can say that a deep learning algorithm learns just as a child does.

The higher the number of cats a child observes, the more accurate his/her ability to identify a cat becomes. Similarly, the more the number of training data samples a deep learning algorithm gets, the higher its accuracy becomes in identifying cats becomes.

Of course, similar to a child, a deep learning algorithm does perform mistakes while starting its learning journey but due to the availability of enormous data and computational resources in the present times it has proven to be far better than classical machine learning architectures in solving certain complicated problems (as seen before) and in some cases competitive to the human experts.

The following graph sums up the story. Higher the amount of data, higher is the deep learning architecture performance as compared to classical machine learning architectures.



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

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 $16 \times 16 \times 3$.

Implementation using Multi-Layer Perceptron

For a classical neural network, all joined connections in its first hidden layer will result in 768 weights ($16 \times 16 \times 3$). However, with an increased number of neurons and image size (say $300 \times 300 \times 3$ 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.

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:

Input image slice		Filter		Feature map
0 1 1 1 0 0 0				1 4 3 4 1
0 0 1 1 1 0 0				1 2 4 3 3
0 0 0 1 1 1 0				1 2 3 4 1
0 0 0 1 1 0 0	*	1 0 1	=	1 3 3 1 1
0 0 1 1 0 0 0		0 1 0		3 3 1 1 0
0 1 1 0 0 0 0		1 0 1		
1 1 0 0 0 0 0				

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.

Input volume
7x7x3 (+1 pad)

0	0	0	0	0	0	0
0	1	1	0	1	0	0
0	2	0	2	2	2	0
0	1	2	0	1	1	0
0	0	0	0	2	1	0
0	0	1	0	1	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	2	0	1	0
0	0	2	2	2	0	0
0	1	2	2	1	1	0
0	0	0	0	1	1	0
0	2	2	0	1	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	2	1	2	0
0	1	0	0	1	2	0
0	1	0	1	0	2	0
0	1	1	2	1	1	0
0	0	1	1	2	1	0
0	0	0	0	0	0	0

Filter 1
3x3x3

1	0	-1
0	-1	-1
-1	-1	0
-1	-1	-1
0	0	0
-1	1	0
0	-1	-1
0	1	0
1	0	0

Bias 1

1

Filter 2
3x3x3

-1	-1	0
1	0	0
1	1	-1
-1	-1	0
-1	1	0
0	-1	-1
-1	0	0
0	0	-1
-1	-1	1

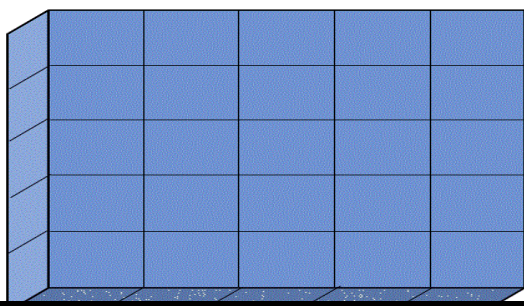
Bias 2

0

Output volume
3x3x2

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

Here's a GIF illustration to understand the process:



Pooling layer:-

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

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

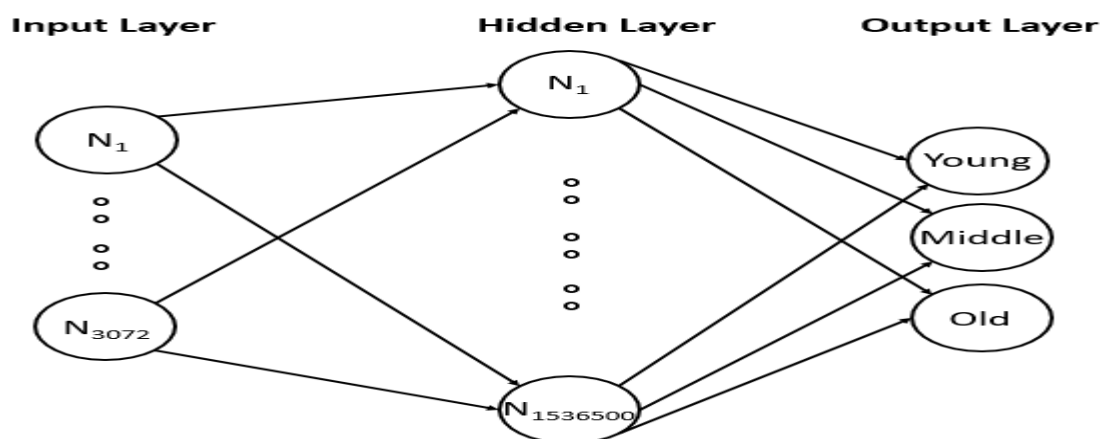
Let us start by importing basic modules:

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.

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

e 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:

```
# Training model along with validation data
```

```
model.fit(train_x, train_y, batch_size=batch_size, epochs=epochs, verbose=1,  
validation_split=0.2)
```

```
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.

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

```
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 is 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])
```

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')

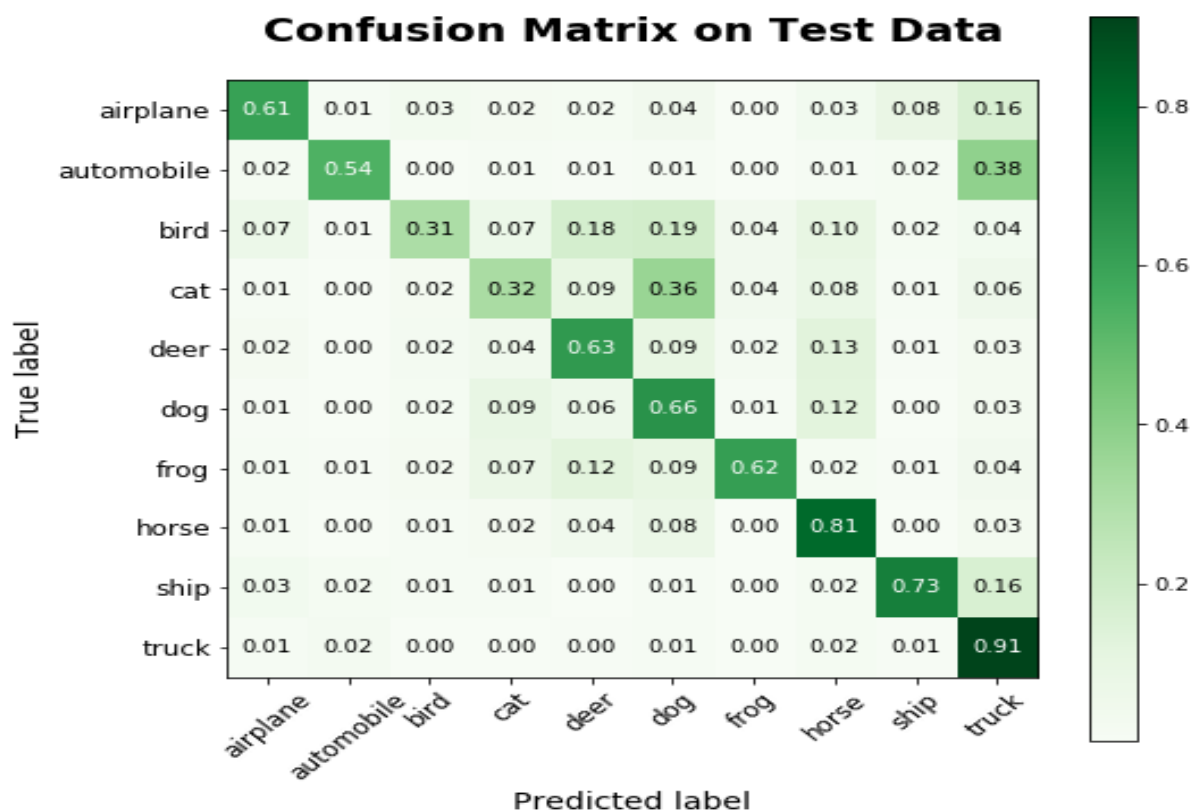
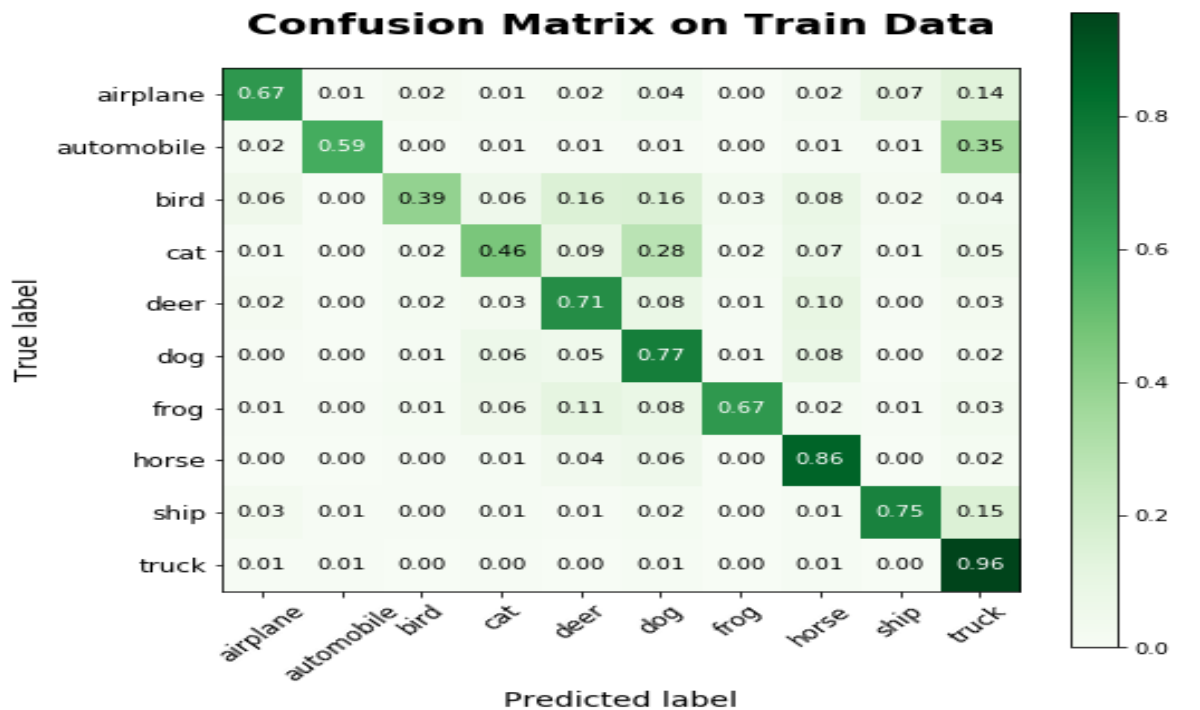
# Test Accuracy

```

```
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')
```



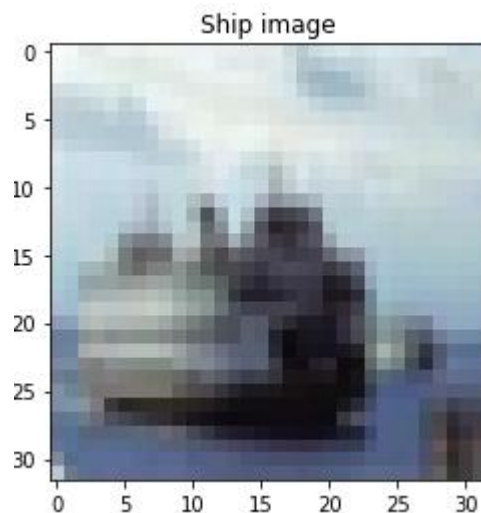
Now, let us learn to see what the dense layer of a model visualizes. For this, we will take our previous model and perform a prediction as shown:

```
from vis.visualization import visualize_saliency, visualize_cam, overlay
```

```

from vis.utils import utils
# Indexes of categories for our model
classes = encode_y.inverse_transform(np.arange(10))
classes
# array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
#        'horse', 'ship', 'truck'], dtype=object)
# Fetching the ship image
ship_img = utils.load_img('cifar10/'+str(test_idx[6])+'.png') # can use Image.open()
also.
plt.imshow(ship_img)
plt.title('Ship image')
plt.show()

```



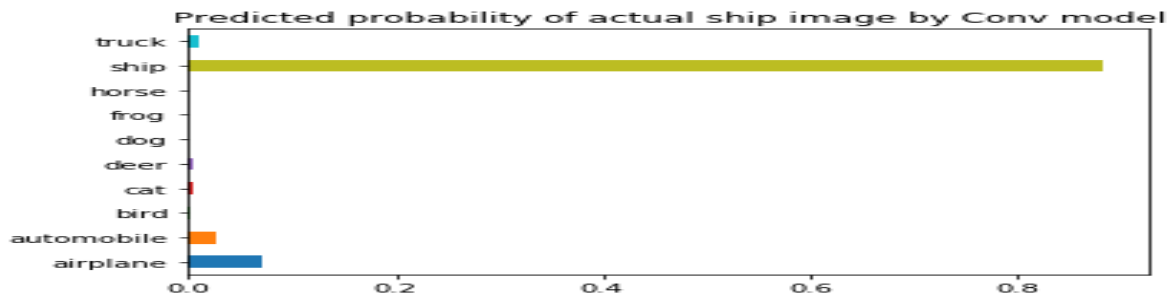
The current image is zoomed for illustration purposes.

Since we know that given target is a ship, now let us predict the probability of the classes based on our model.

```

# Predicting the probability for each of the class
ship_prob = model.predict(X_test[6:7]).ravel().copy()
pd.Series(ship_prob, index=classes).plot.barh()
plt.title('Predicted probability of actual ship image by Conv model')
plt.show()

```



So, the model has presented a high probability that the given image belongs to the ship class.

To visualize activation over final dense layer outputs, we need to switch the softmax activation to linear since the gradient of the output node will depend on all the other node activations.

Utility to search for layer index by name.

```
layer_idx = utils.find_layer_idx(model, 'pred_layer')
```

Swap softmax with linear

```
model.layers[layer_idx].activation = keras.activations.linear
```

```
model = utils.apply_modifications(model)
```

1. Saliency map

Saliency maps clarify which part does our model focuses on to get a prediction.

```
plt.figure(figsize=(12,6))
```

```
for i in range(len(classes)):
```

```
    plt.subplot(2, 5, i + 1)
```

```
    grads = visualize_saliency(model, layer_idx, filter_indices=i, seed_input=ship_img,
    backprop_modifier='guided')
```

```
    plt.xticks([])
```

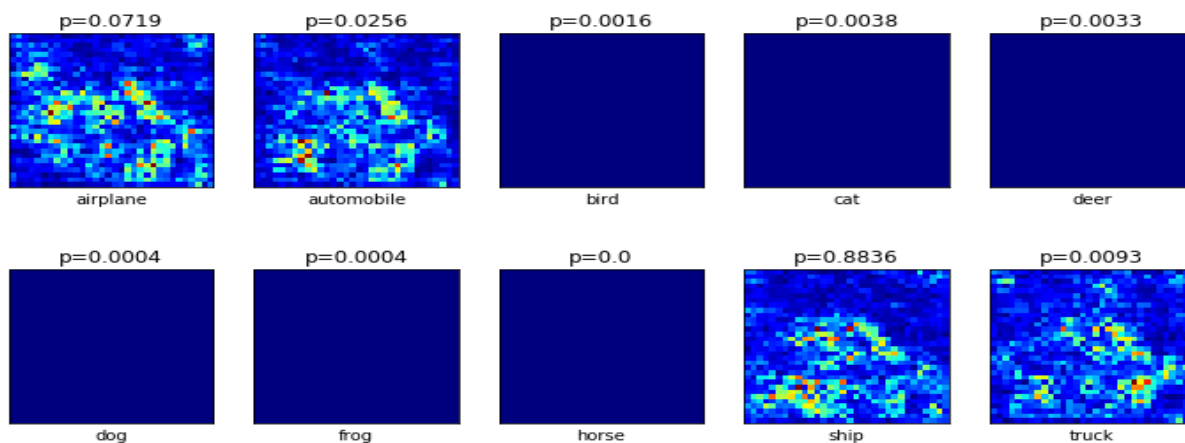
```
    plt.yticks([])
```

```
    plt.xlabel(classes[i])
```

```
    plt.title('p=' + str(np.round(ship_prob[i], 4)))
```

```
    plt.imshow(grads, cmap='jet')
```

```
plt.show()
```



s shown in the probability graph only the first two and last two nodes are active, leaving center six almost dead. The second last node meant for the ship has the highest probability and as seen in the above plot only emphasize on the region of the ship. Other three active nodes include noise and thus reduced the probability.

2. Class activation maps or Grad-cam

These maps contain more detail since they use Conv or Pooling features that contain more spatial detail which is lost in Dense layers. The only additional detail compared to saliency is the `penultimate_layer_idx`. This specifies the pre-layer whose gradients should be used.

```
plt.figure(figsize=(12,6))
```

```
for i in range(len(classes)):
```

```
    plt.subplot(2, 5, i + 1)
```

```
    cam_grads = visualize_cam(model, layer_idx, filter_indices=i, seed_input=ship_img,  
backprop_modifier='guided',
```

```
        penultimate_layer_idx=utils.find_layer_idx(model, 'BN_2'))#
```

```
batch_normalization_14
```

```
    plt.xticks([])
```

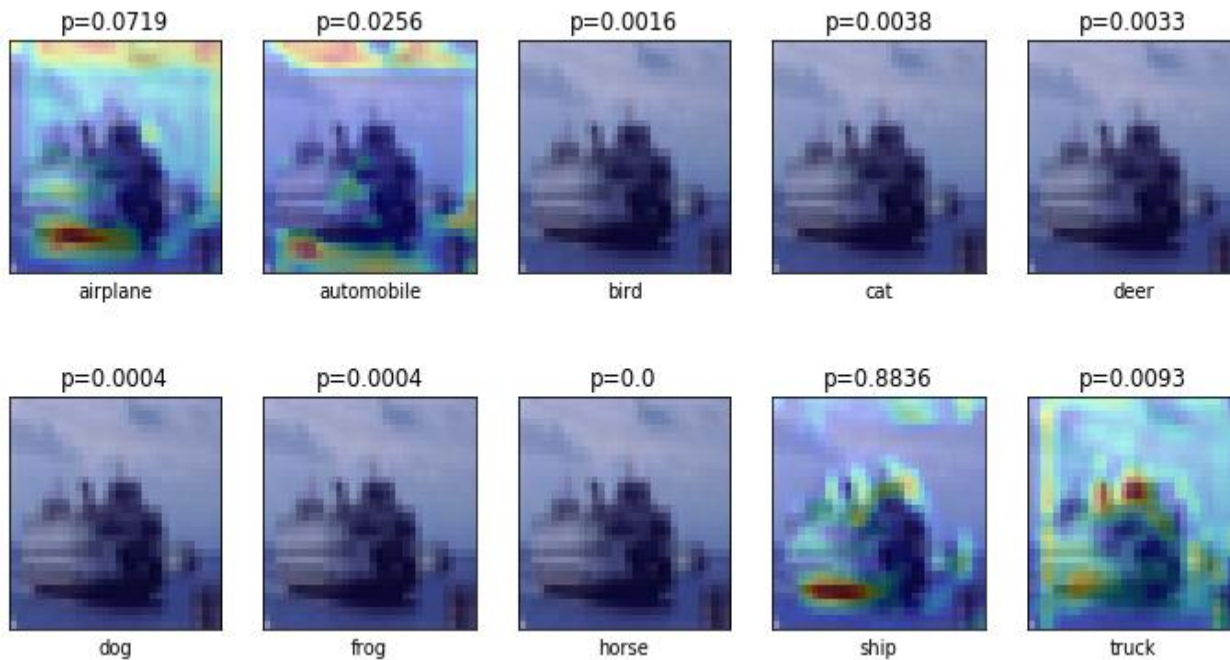
```
    plt.yticks([])
```

```
    plt.xlabel(classes[i])
```

```
    plt.title('p=' + str(np.round(ship_prob[i], 4)))
```

```
    plt.imshow(overlay(cam_grads, ship_img, alpha=0.3))
```

```
plt.show()
```



The ninth subplot constitutes of a contour plot overlayed over the original image which captures ship structure better than other three contour plots which are heavily affected by noise. Center six nodes are almost dead and thus no contour plot.

----- EXERCISE 4 :-

Module name : Predicting Sequential Data □ Implement a Recurrence Neural Network for Predicting Sequential Data.

To perform text generation using RNN, we need to show the model various examples to make a better prediction character by character. For this task, we will be using the Project Gutenberg's The Adventures of Sherlock Holmes, by Arthur Conan Doyle available as a .txt file. Download the file from [here](#) and read the file in Python.

Necessary libraries

from keras.models import Sequential

from keras.layers import Dense, Activation, LSTM, Dropout

from keras.optimizers import RMSprop

from keras.utils.data_utils import get_file

import keras

import random

import numpy as np

Reading the data

text = open('Sherlock Holmes.txt').read().lower()

**print('Given
characters')**

script has ' + str(len(text)) + '

Given script has 581862 characters

Since the dataset is too long, therefore, let us strip the dataset and perform basis preprocessing.

```
text = text[1302:]  
for ch in ['0','1','2','3','4','5','6','7','8','9','!','"', '$', '%', '&', '~', '^', '(', ')', '*',  
          '-', '/', ';', '@', '?', ':', '©', 'ç', 'ã', '\xa0', '\n', '\r', '.']:  
    if ch in text:  
        text=text.replace(ch, ' ')  
print(set(text))
```

```
{'k', 'w', 'è', 'â', 'a', ' ', 'v', 'x', 'n', 'e', 'i', ',', 'j', 'à', 'p', 'm', 'z', 'h', 'g', 'y', 'd', 't', 'f', 'q', 's',  
'o', 'r', 'u', '"', 'b', 'é', 'l', 'c'}
```

Now, we can create a sliding window function in which all the characters inside the window are treated as input and the following character is treated as output. We use the window size of 50 and step size as 3.

```
def window_transform(text, window_size, step_size):  
    inputs = []  
    outputs = []  
    n_batches = int((len(text)-window_size) / step_size)  
    for i in range(n_batches-1):  
        a = text[i * step_size:((i * step_size) + window_size)]  
        inputs.append(a)  
        b = text[(i * step_size) + window_size]  
        outputs.append(b)  
    return inputs, outputs  
# Calling the window function  
window_size = 50  
step_size = 3  
inputs, outputs = window_transform(text, window_size, step_size)
```

Let us verify the results from the above function:

```
inputs[502], outputs[502]  
('caine and ambition, the drowsiness of the drug, an', 'd')
```


As you can observe the length of the window (input size) is 50 which you can verify using `len(inputs[502])` and the corresponding output is the following character of the ongoing sentence which here is 'd'.

For a confirmation here is the sentence at `input[503]` sliding with a step size of 3 (taking three new characters).

```
'ne and ambition, the drowsiness of the drug, and t'
```

Now, let us try to formulate the problem in the context of machine learning. Above we saw the output of `set(text)` resulted in 33 unique characters, therefore, the given problem formulates in the multi-class classification problem.

To begin with, we first sort the output of `set(text)` and map them to a unique numerical value.

Sorting the unique elements

```
chars = sorted(list(set(text)))
```

Encoding

```
chars_to_indices = dict((c, i) for i, c in enumerate(chars))
```

Decoding

```
indices_to_chars = dict((i, c) for i, c in enumerate(chars))
```

For instance, `chars_to_indices['r']` results in **20** which defines that character **r** is mapped to value **20**.

Now, we have each character mapped to a numeric value, it is time to transform the input/output vector in the same numeric format:

```
def encode_io_pairs(text, window_size, step_size):
```

```
    num_chars = len(chars)
```

```
    # cut up text into character input/output pairs
```

```
    inputs, outputs = window_transform(text, window_size, step_size)
```

```
    # create empty vessels for one-hot encoded input/output
```

```
    X = np.zeros((len(inputs), window_size, num_chars), dtype=np.bool)
```

```
    y = np.zeros((len(inputs), num_chars), dtype=np.bool)
```

```
    # loop over inputs/outputs and tranform and store in X/y
```

```
    for i, sentence in enumerate(inputs):
```

```
        for t, char in enumerate(sentence):
```

```
            X[i, t, chars_to_indices[char]] = 1
```

```
            y[i, chars_to_indices[outputs[i]]] = 1
```

```
        return X,y
```

```
X, y = encode_io_pairs(text, window_size, step_size)
```

This completes the formatting of the data set. Now, we can build the LSTM network starting with the first layer having 120 nodes followed by a fully-connected linear layer and a softmax layer.

Designing the model

```
model = Sequential()
```

```
model.add(LSTM(120, input_shape=(window_size, len(chars))))
```

```
model.add(Dropout(0.22))
```

```
model.add(Dense(len(chars), activation='linear'))
```

```
model.add(Dense(y.shape[1], activation='softmax'))
```

Compiling the model

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Subsetting data for an example

```
Xsmall = X[:20000, :, :]
```

```
ysmall = y[:20000, :]
```

Model training

```
model.fit(Xsmall, ysmall, batch_size=500, epochs=10)
```

```
Epoch 1/10
20000/20000 [=====] - 44s 2ms/step - loss: 3.0559
Epoch 2/10
20000/20000 [=====] - 42s 2ms/step - loss: 2.8379
Epoch 3/10
20000/20000 [=====] - 50s 2ms/step - loss: 2.8064
Epoch 4/10
20000/20000 [=====] - 50s 2ms/step - loss: 2.7525
Epoch 5/10
20000/20000 [=====] - 46s 2ms/step - loss: 2.6520
Epoch 6/10
20000/20000 [=====] - 43s 2ms/step - loss: 2.5394
Epoch 7/10
20000/20000 [=====] - 43s 2ms/step - loss: 2.4665
Epoch 8/10
20000/20000 [=====] - 48s 2ms/step - loss: 2.4178
Epoch 9/10
20000/20000 [=====] - 44s 2ms/step - loss: 2.3781
Epoch 10/10
20000/20000 [=====] - 42s 2ms/step - loss: 2.3488
```

To proceed with the prediction, we need to follow a simple rule of thumb. We know that at a time, our script accepts a window size of 50 and takes the output as the 51st character.

Following this rule, we need to predict a character, later remove the first character from our previous window and add the newly predicted character at the end making it still a window of size 50 then predict the second character and keep following the process.

This method along with the number of characters to be predicted is coded below:

```
def predict_next_chars(model, input_chars, num_to_predict):
```

```

pred_chars = ''
for i in range(num_to_predict):
    # Converting this round's predicted characters to numerical input
    x_test = np.zeros((1, window_size, len(chars)))
    for t, char in enumerate(input_chars):
        x_test[0, t, chars_to_indices[char]] = 1.
    # make this round's prediction
    test_predict = model.predict(x_test, verbose = 0)[0]
    # translate numerical prediction back to characters
    r = np.argmax(test_predict)
    d = indices_to_chars[r]
    # update predicted_chars and input
    pred_chars += d
    input_chars += d
    input_chars = input_chars[1:]
return pred_chars

```

Now, all you're left with is the prediction of the new characters which can be performed as shown:

Prediction

```

start = 89
num_to_predict = 10
input_chars = text[start: start + window_size]
print('Complete sequence:', text[start: start + window_size + num_to_predict])
print('Input sequence:', input_chars)
print('Output sequence:', predict_next_chars(model, input_chars, num_to_predict =
num_to_predict))
Complete sequence: otion akin to love for irene adler  all emotions, and that o
Input sequence: otion akin to love for irene adler  all emotions,
Output sequence: an ou the

```

-----Exercise 5:

Module Name: Removing noise from the images □ **Implement Multi-Layer Perceptron algorithm for Image denoising hyperparameter tuning.**

In this module, we will start with the CIFAR-10 data set but this time we will introduce some random noise in each of the images. To initiate, let us read the images in the environment:

```
# Importing basic libraries
```

```
import pandas as pd
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from PIL import Image
```

```
import os
```

```
# Reading all the images in a python list
```

```
img_arr = []
```

```
for i in range(1, 151):
```

```
    img_path = os.path.join('cifar10/' + str(i) + '.png')
```

```
    img = np.array(Image.open(img_path))/255. # Scaling
```

```
    img_arr.append(img)
```

```
# Converting back to numpy array
```

```
img_arr = np.array(img_arr)
```

```
img_arr.shape
```

```
(150, 32, 32, 3)
```

So, as you can observe in the above code, we have used only 150 CIFAR-10 dataset images and stored all of these 32x32x3 dimensional images to a numpy array. Now, we can add noise to each of these images:

```
# Original image
```

```
plt.imshow(img_arr[4])
```

```
plt.show()
```



```
# Adding random noise to the images
```

```

noise_factor = 0.05
noisy_imgs = img_arr + noise_factor * np.random.normal(size=img.shape)
# Image with noise
plt.imshow(noisy_imgs[4])
plt.show()

```



We will continue the codes which are used to add the noise to the dataset. Let us first add the Keras modules:

```

from keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model

```

Now, let us design the auto-encoder having two downsampling layers in encoding with filters 32 and 64, followed by a final layer with 128 filters. In the decoding module, we have 128 and 64 filters in the first two upsampling layers followed by a final softmax layer with 3 nodes (representing each color channel).

```

def auto_encoder(img):
    # Encoder module
    f_size = 3 # filter size
    p_size = 1 # pool size
    conv_1 = Conv2D(32, (f_size, f_size), activation='relu', padding='same')(img)
    pool_1 = MaxPooling2D(pool_size=(p_size, p_size))(conv_1)
    conv_2 = Conv2D(64, (f_size, f_size), activation='relu', padding='same')(pool_1)
    pool_2 = MaxPooling2D(pool_size=(p_size, p_size))(conv_2)
    conv_3 = Conv2D(128, (f_size, f_size), activation='relu', padding='same')(pool_2)
    # Decoder module
    conv_4 = Conv2D(128, (f_size, f_size), activation='relu', padding='same')(conv_3)
    up_1 = UpSampling2D((p_size, p_size))(conv_4)
    conv_5 = Conv2D(64, (f_size, f_size), activation='relu', padding='same')(up_1)

```

```
up_2 = UpSampling2D((p_size, p_size))(conv_5)
```

```
decoded = Conv2D(3, (f_size, f_size), activation='sigmoid', padding='same')(up_2)
```

```
return decoded
```

The above function holds the complete architecture of the auto-encoder to be used in this scenario. Now, let us compile and train the model on 120 images out of which 20% are kept for validation for 10 epochs and remaining 30 images are left for test purposes.

Calling the network design function and compiling the model

```
img = Input(shape=(32, 32, 3))
```

```
model = Model(img, auto_encoder(img))
```

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Training the model

```
model.fit(noisy_imgs[:120], img_arr[:120], epochs=10, validation_split=0.2)
```

```
Train on 96 samples, validate on 24 samples
Epoch 1/10
96/96 [=====] - 15s 158ms/step - loss: 0.0021 - val_loss: 0.0021
Epoch 2/10
96/96 [=====] - 14s 148ms/step - loss: 0.0021 - val_loss: 0.0027
Epoch 3/10
96/96 [=====] - 14s 149ms/step - loss: 0.0021 - val_loss: 0.0022
Epoch 4/10
96/96 [=====] - 17s 174ms/step - loss: 0.0019 - val_loss: 0.0020
Epoch 5/10
96/96 [=====] - 16s 166ms/step - loss: 0.0018 - val_loss: 0.0020
Epoch 6/10
96/96 [=====] - 16s 171ms/step - loss: 0.0018 - val_loss: 0.0018
Epoch 7/10
96/96 [=====] - 16s 163ms/step - loss: 0.0018 - val_loss: 0.0018
Epoch 8/10
96/96 [=====] - 15s 158ms/step - loss: 0.0017 - val_loss: 0.0018
Epoch 9/10
96/96 [=====] - 15s 159ms/step - loss: 0.0016 - val_loss: 0.0017
Epoch 10/10
96/96 [=====] - 17s 174ms/step - loss: 0.0016 - val_loss: 0.0018
```

Let us perform the prediction on training and test data:

```
pred = model.predict(img_arr)
```

Training data prediction

```
plt.figure(figsize=(10, 5))
```

```
ax1 = plt.subplot2grid((1, 3), (0,0))
```

```
ax1.set_title('Original image', fontsize='large')
```

```
ax1.imshow(img_arr[4])
```

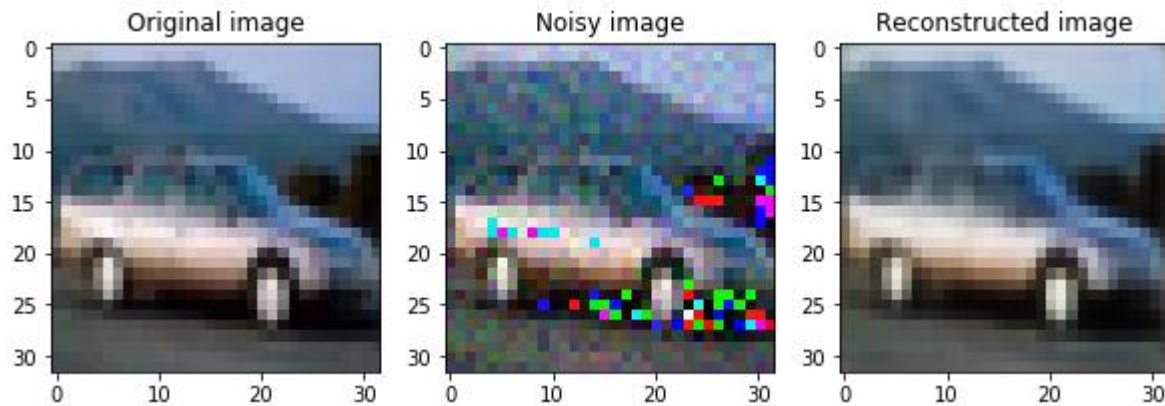
```
ax2 = plt.subplot2grid((1, 3), (0,1))
```

```
ax2.set_title('Noisy image', fontsize='large')
```

```
ax2.imshow(noisy_imgs[4])
```

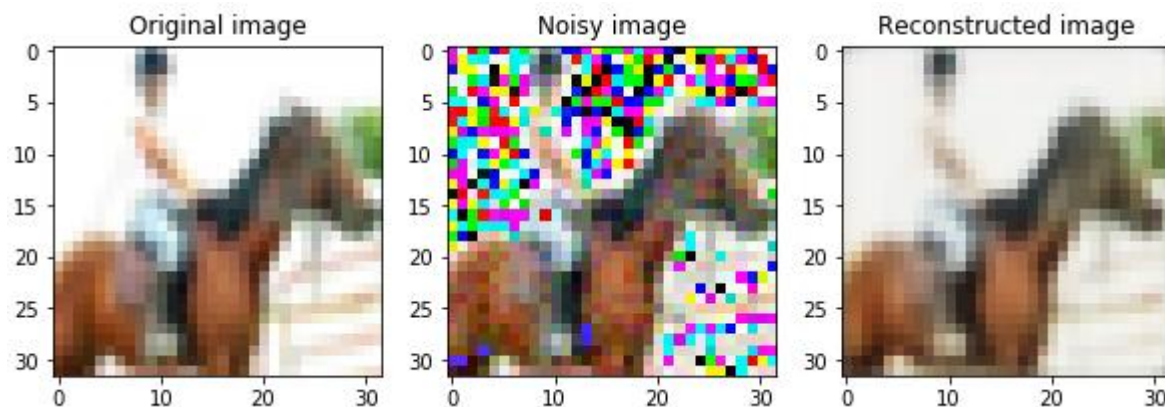


```
ax3 = plt.subplot2grid((1, 3), (0,2))
ax3.set_title('Reconstructed image', fontsize='large')
ax3.imshow(pred[4])
plt.show()
```



Test data prediction

```
plt.figure(figsize=(10, 5))
ax1 = plt.subplot2grid((1, 3), (0,0))
ax1.set_title('Original image', fontsize='large')
ax1.imshow(img_arr[131])
ax2 = plt.subplot2grid((1, 3), (0,1))
ax2.set_title('Noisy image', fontsize='large')
ax2.imshow(noisy_imgs[131])
ax3 = plt.subplot2grid((1, 3), (0,2))
ax3.set_title('Reconstructed image', fontsize='large')
ax3.imshow(pred[131])
plt.show()
```



**Exercise
6:
Module
Name:
Advance
d Deep
Learning**

Architectures

Implement Object Detection Using YOLO.

What is advanced deep learning Architecture?

Advanced deep learning architecture consists of set of rules and methods that describe the functionality, organization, and implementation of training the deep learning model to fit the data accurately. Advanced architecture has a proven track record of being a successful model. Pre-trained models appearing on the market, more industries will be able to discover the benefits of cost-effective object recognition for tasks that not so long before were impossible to automate.

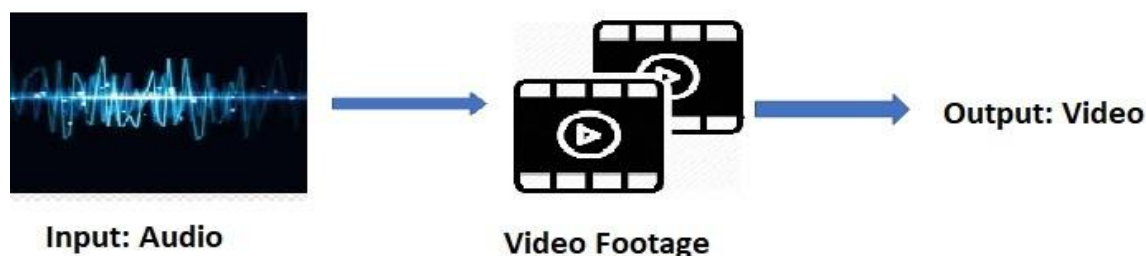
Applications of advanced deep learning architecture:

1. Synthesizing audio to video conversion:

There is an increase in capturing the video footage due to increase in video conferencing with tools like Skype, FaceTime, WebEx. There is significant video footage available online for many public figures in the form of interviews, speeches, newscasts, etc. Aside from being interesting purely from a scientific standpoint audio to video, has a range of important practical applications.

The ability to generate high quality video from audio could significantly reduce the amount of bandwidth needed in video transmission. For hearing-impaired people, video synthesis could enable lip-reading from over-the-phone audio. Digital humans are central to entertainment applications like film special effects and games.

The video footage can be used to train the deep learning model which can be further used to synthesis the audio to generate the video. However, analyzing the video footage is quite challenging, as the faces are often shown in a near-profile view, the face region is small, and the lighting, dress, hair, and make-up varies significantly from one video footage to the next.



2. Coloring the black and white photos or videos: Do you have your favorite old photos, movies? Are they in black and white? Do you like to see them colored? It is possible to restore the black and white movies or your favorite photos. The Deep Learning network can be trained to learn patterns that naturally occur in photos like the sky is usually blue, clouds are often white/gray and grass is typically green in order to restore these colors without human intervention.

3. Voice Generation

Siri and Alexa are two successful voice generating system. But these are not completely an autonomous system since, they were trained manually to convert text to voice. WaveNet, a deep generative model-Google and Deep Speech- Baidu are Deep Learning networks that generated voice automatically. systems created today learn to mimic human voices by themselves and improve with time. When letting an audience try to differentiate them from a real human speaking, it is much harder to do so.

4. Deep Learning networks for creating deep learning networks

Neural complete is a deep learning code that can generate new deep learning networks. It is not only written in Python, but also is trained on generating Python code.

5. Deep Dreaming

Have you noticed YouTube is packed nowadays with hallucinated images based on existing photos? Deep learning has let the computers to deep dream on to create hallucinated images by enhancing the existing photos.



6. Generate samples for image datasets

The performance of the deep learning model is mainly dependent on the dataset used for training. The quantity and the quality of the dataset is crucial in training the model. In many practical scenarios, the dataset may 't be sufficient to train the model. Advanced deep learning architectures are now helping the deep learning developers to generate examples to create the data set. Few samples of images are generated for MNIST data set using GAN architecture.



7. Automatic generation of text

Advanced deep learning is used to generate the articles. Computer have become more intelligent to automatically generate Wikipedia articles, computer programs using advanced LSTM architecture.

List of advanced deep learning architecture

- VGGnet
- YOLO
- Need For Advanced Deep Learning

Many Industries have shifted to deep learning techniques. Fig 1 shows some of the applications in use based on Deep Learning-DL. The reasons for the prevalent popularity and adoption of basic deep learning architecture are:

- The availability of heavy graphics cards like GPU, TPU which has affluence training and tuning of DL model. A huge number of layers can be used to train the DL model with ease.
- Accessibility of large collection of historical data set.
- Python ecosystem that consists of open source deep learning framework.
- What are the challenging problems that demands advanced deep learning?

1. Object classification, Detection and localization:

Classification and object detection are the main parts of computer vision. Recent advances in Deep Neural Networks (DNNs) have led to the development of DNN-driven autonomous cars that, using sensors like camera, LiDAR, etc., can drive without any human intervention. Most major manufacturers including Tesla, GM, Ford, BMW, and Waymo/Google are working on building and testing different types of autonomous vehicles which requires no driver input. Self-driving cars rely on artificial intelligence to work. Rapid and reliable video object recognition, localization is the foundation of such autonomous driving. Such autonomous system requires road signs, recognition of pedestrians, obstacles on the road, buildings in roadsides with 100% accuracy to cause the right and safe reactions from the vehicle. Application of object recognition has the potential to save lives by helping doctors diagnose diseases from high-resolution photographs, MRIs, and CT scans.

- Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image. Fig 2 indicates some of the object detection problems.

2. Audio, video, handwritten processing:

The challenges in NLP is not yet completely addressed. For example, every time Siri tries to answer a question that has not been programmed to respond, it fails miserably. Speech recognition, question answering, reputation monitoring, market intelligence and real time handwritten character recognition are some of the NLP related challenging tasks.

3. Human activity detection in a video:

Classification is finding what is in an image and object detection and localization is finding where is that object in that image. Detection is a more complex problem to solve as we need to find the coordinates of the object in an image. Real-time Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image.

Following are some of the challenging problems:

- Object recognition gets most of the attention. Designing AI solution to understand the scene requires to recognize where are the objects; and how far away are they. The brain seems to solve scene understanding with a statistical model that decomposes the scene into affordances and structured relationships that have soft and ambiguous boundaries. A similar approach is required to be adopted.
- Beyond the identification of object, machines are required to “be creative” and use randomness to make sense of the world.
- Identification of the human action in video

Object Detection:

In computer vision applications, deep learning is used to perform three most commonly occurring task in real time as depicted in fig.5

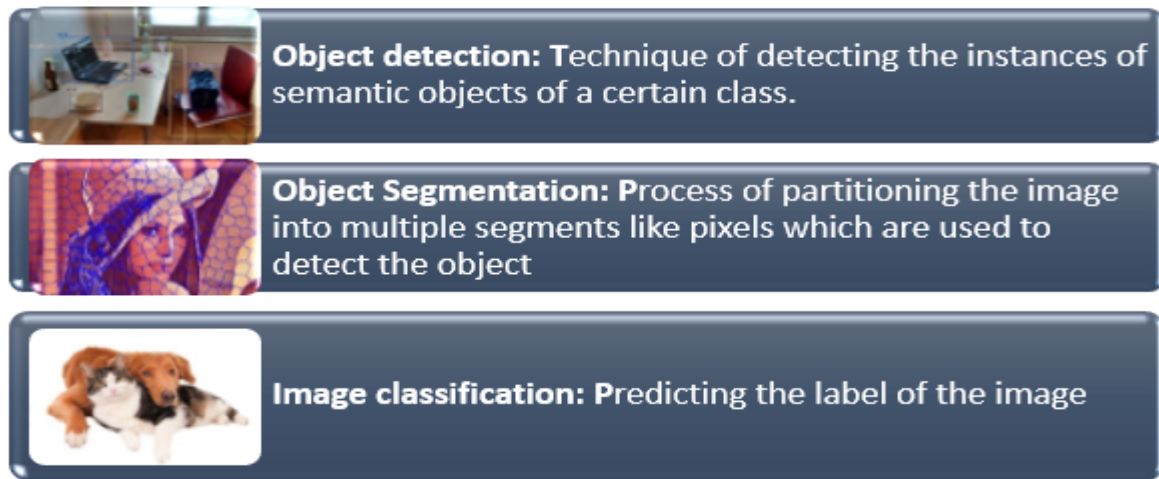


Fig 5. Difference between object classification, detection, and segmentation

Deep neural networks will be trained to extract the features from the input images to perform the given task. Fig 6 shows different levels of features associated with images. For object detection, deep neural networks use features. The hierarchical representation of image features is as shown in fig 6.

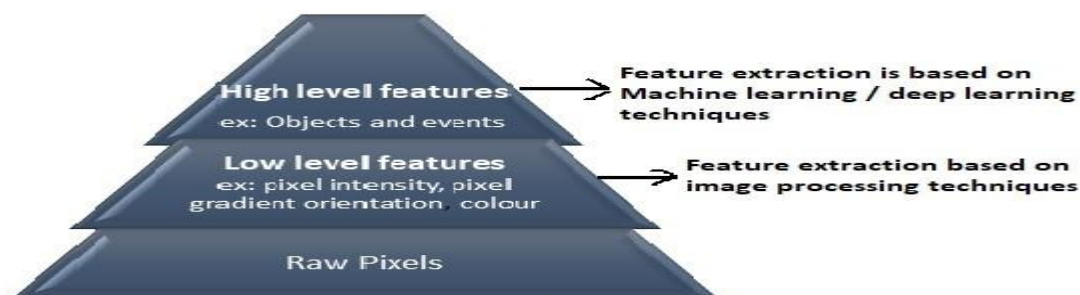


Fig 6: Hierarchical representation of image feature

It is possible to use:

- Binary classifiers. Ex: MNIST handwritten digit recognition using K-NN
- Other ML image classification techniques, ex: MNIST handwritten digit recognition using SVM
- Artificial Neural Network
- Convolutional neural networks for image classification, ex: MNIST handwritten digit recognition

Among the above listed ML/DL techniques, CNN is proved to be the best DNN architecture for Image classification. But for a real-time object detection, the advanced DL architecture namely YOLO is supreme. Let's look at how it works.

- 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

Step2: Divide the image into $M \times M$ grid of cells

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 $P_{ClassObject}$. This probability is conditioned on the grid cell containing one object.

The vector Y for first grid looks like this

The output of this step results in $3 \times 3 \times 8$ 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 13×13 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.

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:

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.

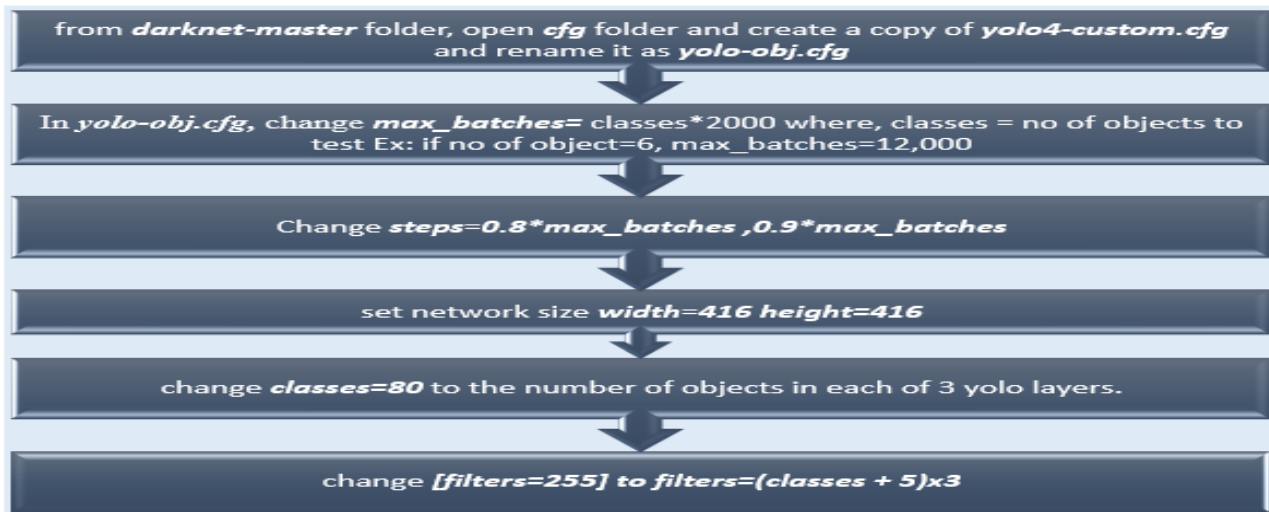
Setting up Pre-Trained models: How to Train YOLO to detect your conventional objects

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:

1. 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.



2. Download the pre trained weights from the link [yolov4.conv.137](https://github.com/yolov4/weights) 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

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 Makefile 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 details here :
https://colab.research.google.com/drive/1ITGZsfMaGUpBG4inDIQwIJVW476ibXk_

- 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.

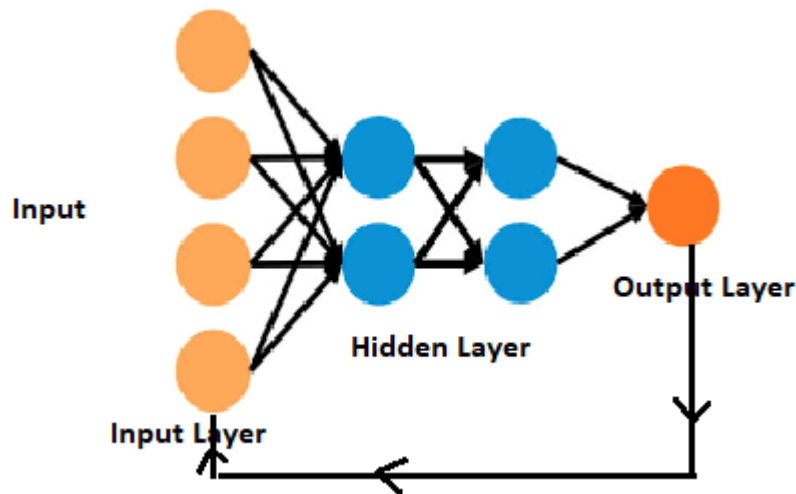
Limitations and drawbacks of the YOLO object detector

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 $M \times M$ 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: - Module Name: Optimization of Training in Deep Learning Design a Deep learning Network for Robust Bi-Tempered Logistic Loss.

Deep Learning is an iterative way of training the machine. Like, any iterative or a cyclic process, DL involves three main components namely, **formulate- training the NN, test the model and evaluate the model**. In other words, iteration in DL indicates the number of times the hyperparameter are updated. Hyperparameters are the core entities in any DL models. In the previous modules, t hyperparameter was discussed. The best combination of various permutations of the hyperparameters must be found to ensure the accurate results. Finding the best permutations of the hyperparameters turns out to be an optimization problem in context of training the neural network.



Training the deep learning models take long time. Training few complex DL models take hours to days. Advanced deep learning architecture like YOLO take 12-hours on COCO data set. In order to achieve the best training efficiency, the performance of the optimization algorithm becomes an important factor. Deep learning algorithms requires optimization in different circumstances but training the neural network is said to be most difficult task for the following reasons:

Time consuming: In real time, training a single neural network instance on several machines will take days to months in real time scenarios.

Expensive: Training the NN is expensive

For the above said reasons specialized optimization techniques are required.

Most of the deep learning models are stochastic in nature. Hence, their performance optimization techniques are different from conventional optimization techniques. In conventional optimization techniques, optimization of the performance is direct and is the objective. Whereas in neural network, performance optimization is indirect. i.e., optimization of the objective function will minimize the error or cost function and improving the overall prediction accuracy of the model.

The problem of training the neural network or the learning is an optimization problem. The fine-tuning of randomly chosen weights while training the NN will impact on the learning. Maximizing or minimizing the weights become the objective function during training. The objective function or loss function needs to be adjusted to minimize the

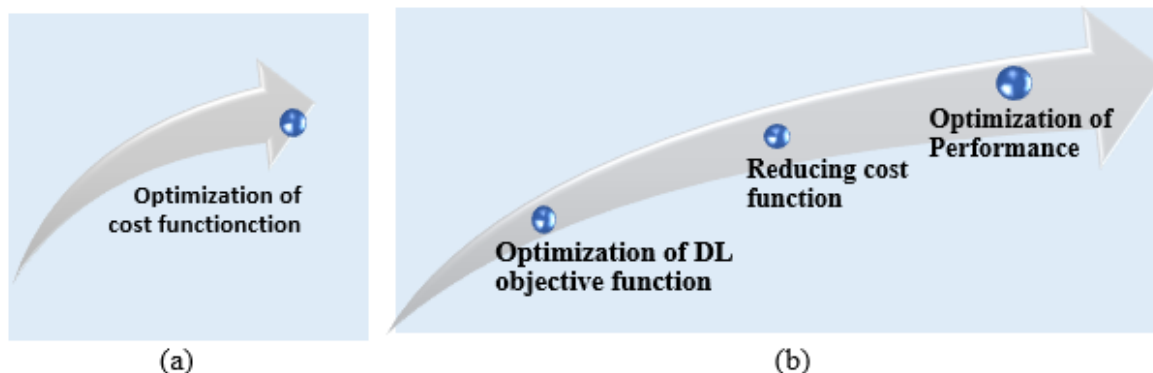


Fig 2. (a) conventional optimization (b) Deep learning Optimization

prediction error. Diminishing the loss function becomes the main target of the optimization on

procedure.

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

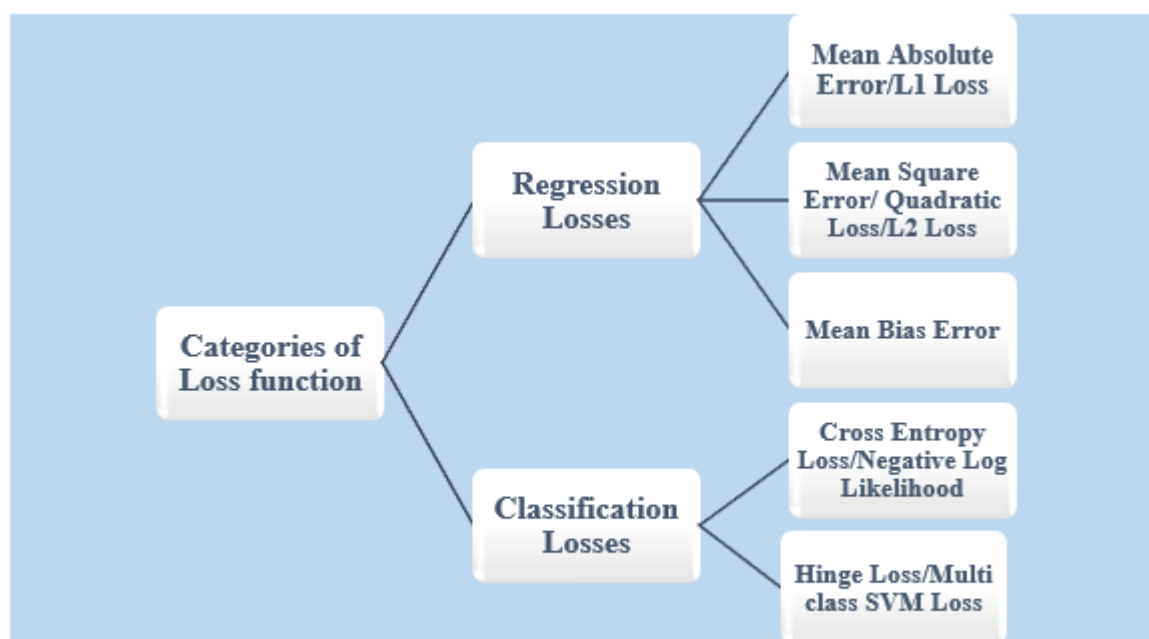


Fig 3. Loss functions

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 <i>confident but wrong</i> .
$MSE = \frac{\sum_{i=1}^n (actual - predicted)^2}{n}$	$cross\ entropy = -(actual_i \log(predicted) + (1 - actual_i) \log(1 - actual_i))$
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

```

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_crossentropy,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 metrics
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_crossentropy,
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))
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")
#Adadelata
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="Adadelata")
#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, Adadelata, Adam and adabound.

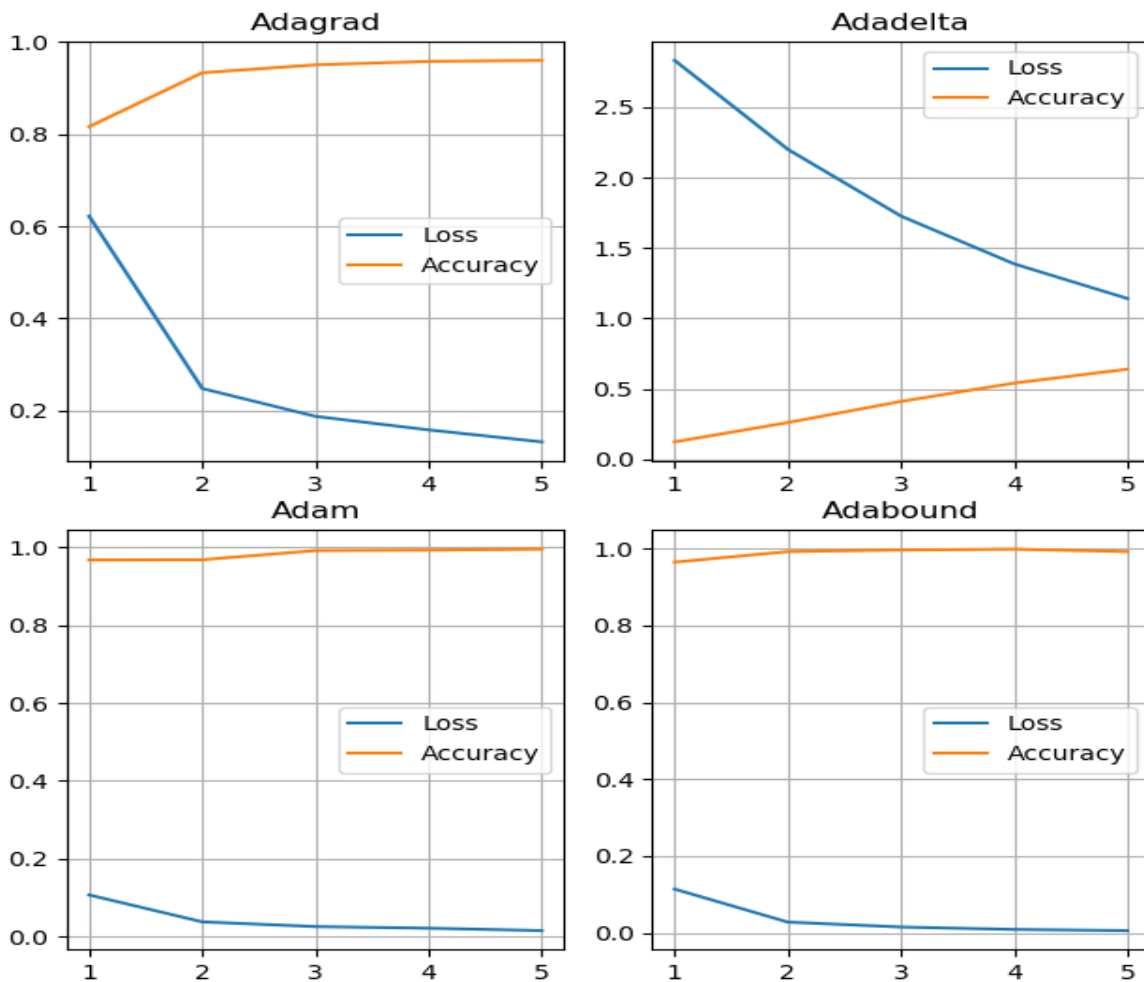


Fig 4. Comparison of loss function and accuracy for the optimizers Adagrad, Adadelta, Adam and adabound

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 loss function in deep learning models are used for specific purposes.

#	Advanc
1	Robust E
2	Minimar
3	Loss fun

4	Intersection over Union (IoU)-balanced Loss Functions for Single-stage Object Detection	<ul style="list-style-type: none"> High IoU can increase the correlation between classification and the task of localization. The loss aims at decreasing the gradient of the examples with low IoU and increasing the gradient of examples with high IoU. This increases the localization accuracy of models.
5	Loss function (Boundary) for the segmentation of highly unbalanced classes	<ul style="list-style-type: none"> Distance metric is used on the outline spaces or shapes instead of regions. Given an unbalanced image segmentation problem, suppose, this loss function objective is to alleviate the complications of regional losses. The integrals of the outline between the regions is used instead of taking the integrals on unbalanced regions. It offers evidence of data that is corresponding to regional losses.
6	Perceptual Loss Function	<ul style="list-style-type: none"> This loss function is a Feed-forward convolutional neural networks which are trained when an input image is transformed into an output image. high-quality images can be generated by defining and optimizing loss functions based on high-level features extracted from pre-trained networks

**Exercise 8:-
Module name:**

Advanced CNN

Exercise: Build AlexNet using Advanced CNN.

```

import keras
import numpy as np
from keras.datasets import mnist
import matplotlib.pyplot as plt
(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
```

```
model = Sequential()
model.add(Conv2D(filters = 96, input_shape = (60000,784, 3),kernel_size = (11, 11), strides
= (4, 4), padding = 'valid'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2, 2),strides = (2, 2), padding = 'valid'))
model.add(BatchNormalization())
```

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

```
model.add(Conv2D(filters = 384, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
model.add(BatchNormalization())
```

```
model.add(Conv2D(filters = 384, kernel_size = (3, 3),strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
model.add(BatchNormalization())
```

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

```
model.add(Flatten())
```

```
model.add(Dense(4096, input_shape = (224*224*3, )))
model.add(Activation('relu'))
model.add(Dropout(0.4))
```



```

model.add(BatchNormalization())

model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))
model.add(BatchNormalization())

model.add(Dense(10))
model.add(Activation('softmax'))

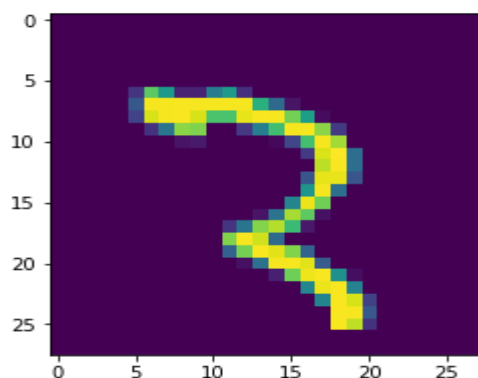
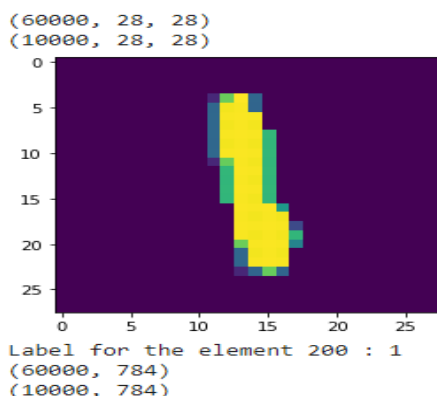
model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=['accuracy'])
y=to_categorical(y_train)

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

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

predictions = model.predict(x_test[0:100])
predictions[0]
np.argmax(predictions[0])
plt.imshow(x_test[0].reshape(28,28))

```



Exercise 9:Module name:

Autoencoders Advanced

Exercise: Demonstration of Application of Autoencoders.

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 are 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.

1. Reconstruction of sequence using Autoencoders

Setp1:Building an simple autoencoders to create simple 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
# 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 form, 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 : Module name: Advanced GANs

Exercise: 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
```

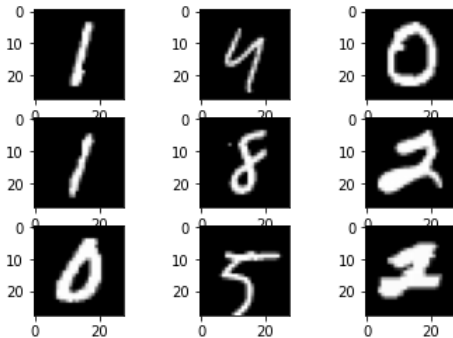
```

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 learn image effectively.

In this demonstration, ZCA is used to show GANs application in generating new image after eliminating the redundant pixels.

ZCA whitening

```

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')

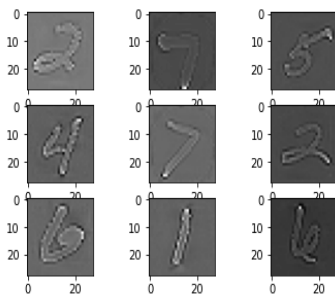
```

```

# define data preparation
datagen = ImageDataGenerator(zca_whitening=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

```

/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/image_data_generator.py:337: UserWarning: This ImageDataGenerator specifies `zca_whitening`, which may not be the best choice for your data. Consider using `zca_normalization` instead. See https://keras.io/preprocessing/image/ for more details.



c. Random Flips

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

```

from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
(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(horizontal_flip=True, vertical_flip=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)

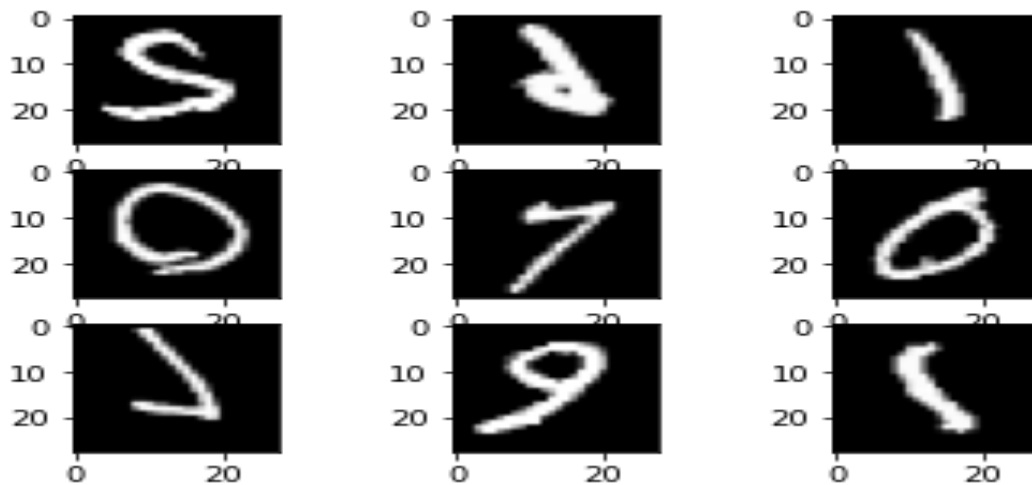
```

```

pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
# show the plot
pyplot.show()
break

```

Output:



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

```

Step 3: Defining parameters to be used in later processes

```

image_shape = (32, 32, 3)
latent_dimensions = 100

```

Step 4: Defining a utility function to build the Generator

```

def build_generator():
    model = Sequential()

```



```

    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"))
    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():
    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))
        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)
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

```

discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
                    optimizer=Adam(0.0002, 0.5),
                    metrics=['accuracy'])
discriminator.trainable = False
generator = build_generator()
z = Input(shape=(latent_dimensions,))
image = generator(z)
valid = discriminator(image)
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=[]
X = (X / 127.5) - 1.
valid = np.ones((batch_size, 1))
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):
    index = np.random.randint(0, X.shape[0], batch_size)
    images = X[index]
    noise = np.random.normal(0, 1, (batch_size, latent_dimensions))
    generated_images = generator.predict(noise)
    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)

    genr_loss = combined_network.train_on_batch(noise, valid)

```

```
if epoch % display_interval == 0:  
    display_images()
```

Output:

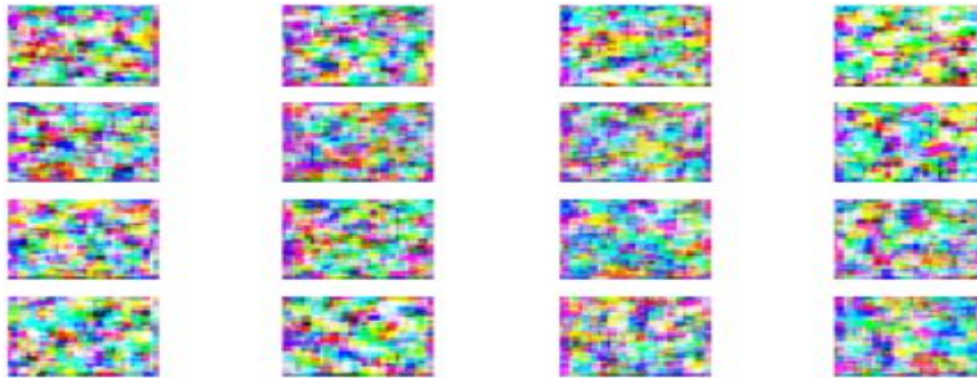


Fig1a.image generated at epoch size 20,



Fig1b. image generated at epoch size 2000,

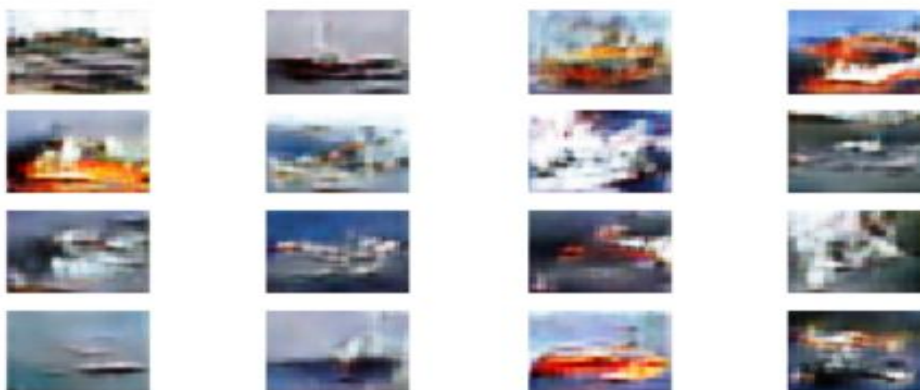


Fig1c.image generated at epoch size 5000

Exercise 11:

Module name : Capstone project

Exercise : Complete the requirements given in capstone project

Description: In this capstone, learners will apply their deep learning knowledge and expertise to a real world challenge.

Object Classification for automated CCTV

Problem Description:

Nowadays, Surveillance has become an essential part of any industry for safety and watch. Recent developments in technology like computer vision, machine learning has brought significant advancements in various automatic surveillance systems. Generally, CCTV will be running all the time and hence, consumes more memory.

One of the industries decides to adopt artificial intelligence for automating CCTV recording. The idea is to customize the CCTV operation based on the **object detection**. The industry has come up with the plan to automate the CCTV in a way that if some objects are recognized and categorized as belonging to specific class only then the recording should start. By using this method, the need for recording the images continuously gets avoided there by reducing the memory requirements.

So the, problem is to categorize the object type as human, vehicles, animals etc... Suppose you are asked to analyze this industry requirement and come up with a feasible solution that can help the company to customize the CCTV based image classification.

Instructions for problem solving:

As a deep learning developer, design a best model by training the neural network with 60,000 training samples.

- Use all the test image samples to test whether the product is labelled appropriately.
- You can use *tensorflow* / *Keras* for downloading the data set and to build the model.
- Fine tune the hyperparameters and perform the model evaluation.
- Substantiate your solution based on your insights for better visualization and provide a report on model performance.

Data set description:

Initially to test the model you can use the benchmark data set namely. *Fashion-MNIST* data set before deploying it. This dataset is a standard dataset that can be loaded directly. For more details click [here](#). The data set description is as follows:

- Size of training set = 60,000 images
- Number of samples/class = 600,000 images.
- Image size= Each example is a 28x28 grayscale image. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel. This pixel-value is an integer that ranges between 0 and 255.
- Number of class = 10 classes.

The training and test data sets have 785 columns. The details of the data set organization are as given below:

- Each row is a separate image
- Column 1 is the class label
- Remaining columns are pixel numbers (784 total)
- Each value is the darkness of the pixel (1 to 255)

Each training and test example are assigned with one of the following labels:

- Cars
- Birds
- Cats

- Deer
- Dog
- Frog
- Horses
- Ships
- Trucks
- Airplanes